Go进阶训练营
第12课
消息队列 - Kafka 答疑

大明

目录

Kafka 使用场景

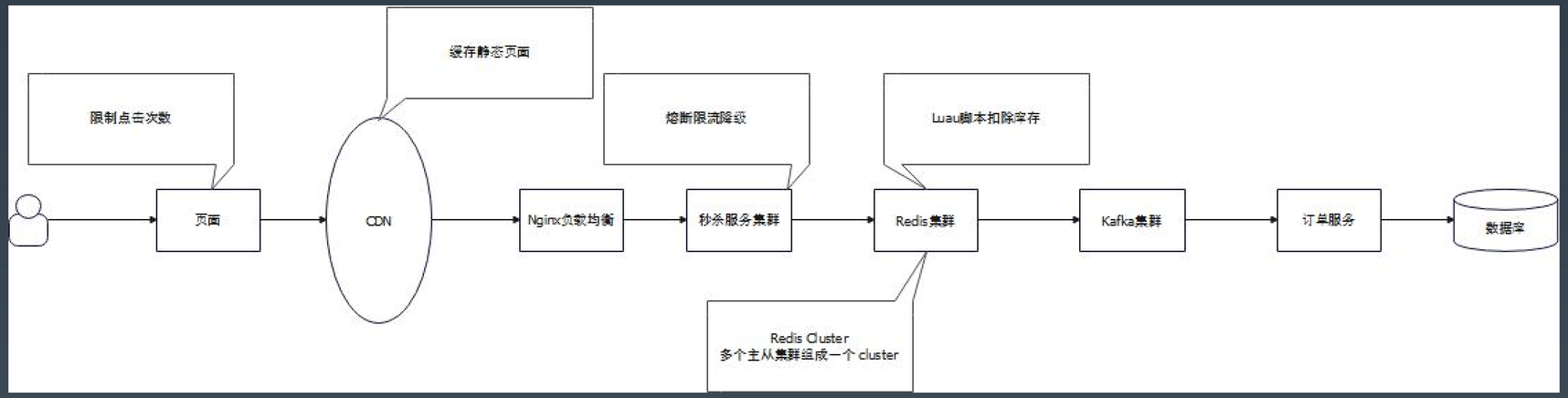
一般来说，使用消息队列是为了异步、解耦、削峰、通讯

具体到实际场景中，主要有：

- 秒杀
- 日志收集
- 通知：如邮件、短信
- 事件驱动
- 最终一致性的分布式事务方案

Kafka 使用场景——秒杀

秒杀是典型的削峰场景。常见的秒杀流程：

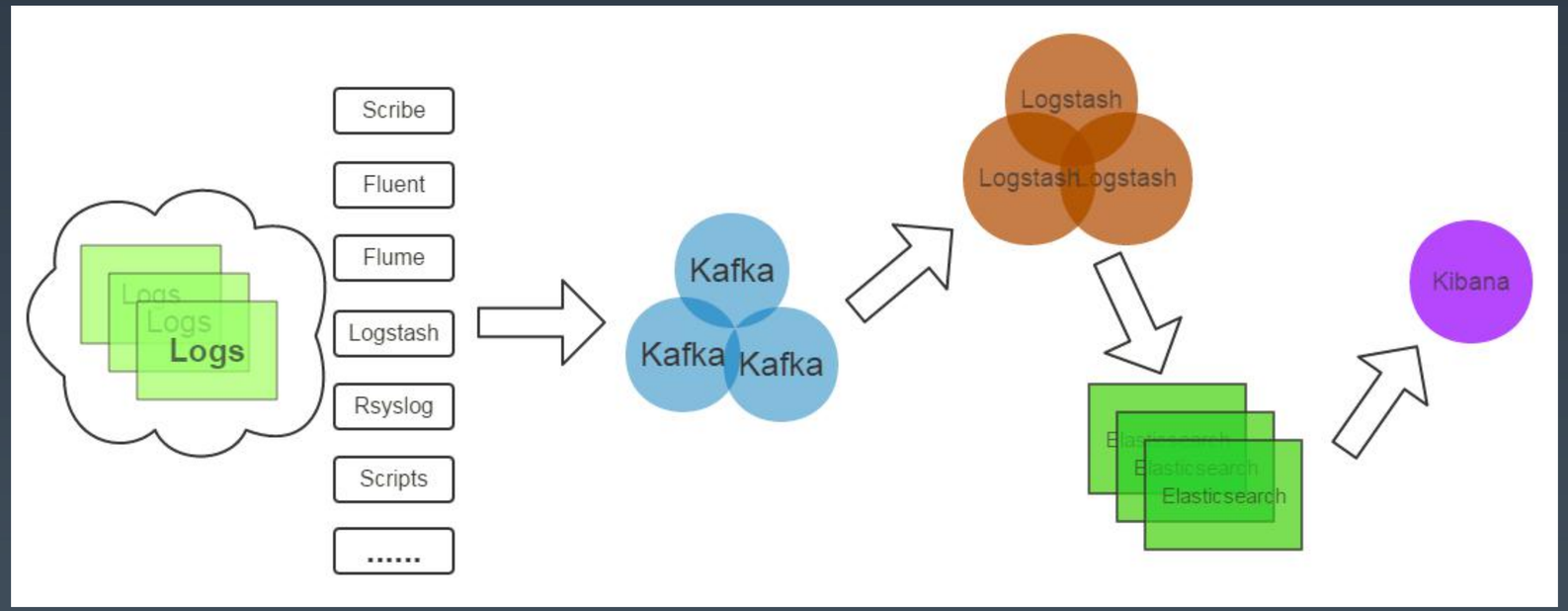


在扣减库存之后才丢到 Kafka 里面，而后消费消息，真实创建订单，并且异步通知用户支付。

类似于把 kafka 看做是蓄水池，一大波水过来都被装在池子里，然后慢慢放出去

Kafka 使用场景—— 日志收集

日志收集应该看做是处理大数据的典型应用



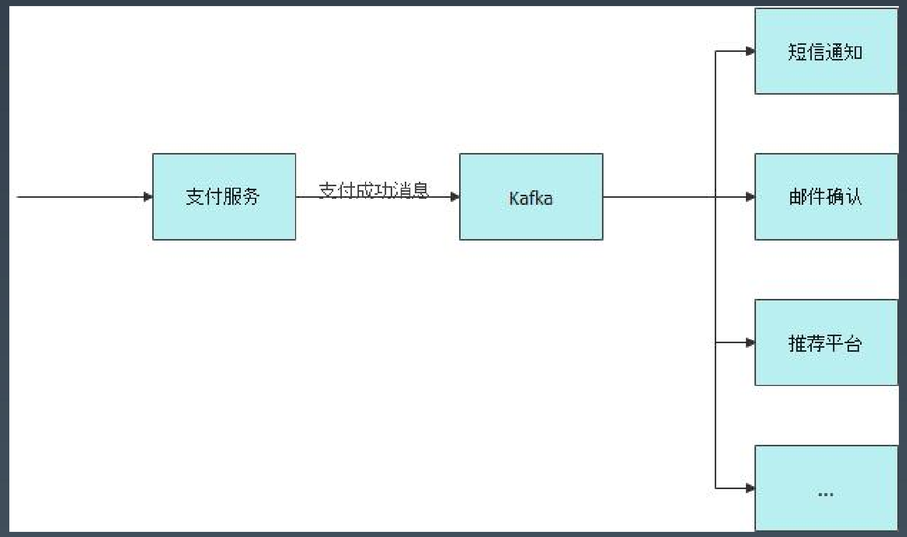
从各个地方收集的数据，都集中丢到 Kafka 里面，然后下游从 Kafka 里面消费数据。其实它和秒杀也是差不多，都是看做蓄水池。

不一样的地方是，它的数据来源更加多样，大家都按照一样的格式丢给它数据，所以可以看做是一个抽象屏障。因此它也是一个典型对接多个上游的例子。

图来自 https://developer.aliyun.com/article/59482

Kafka 使用场景——通知

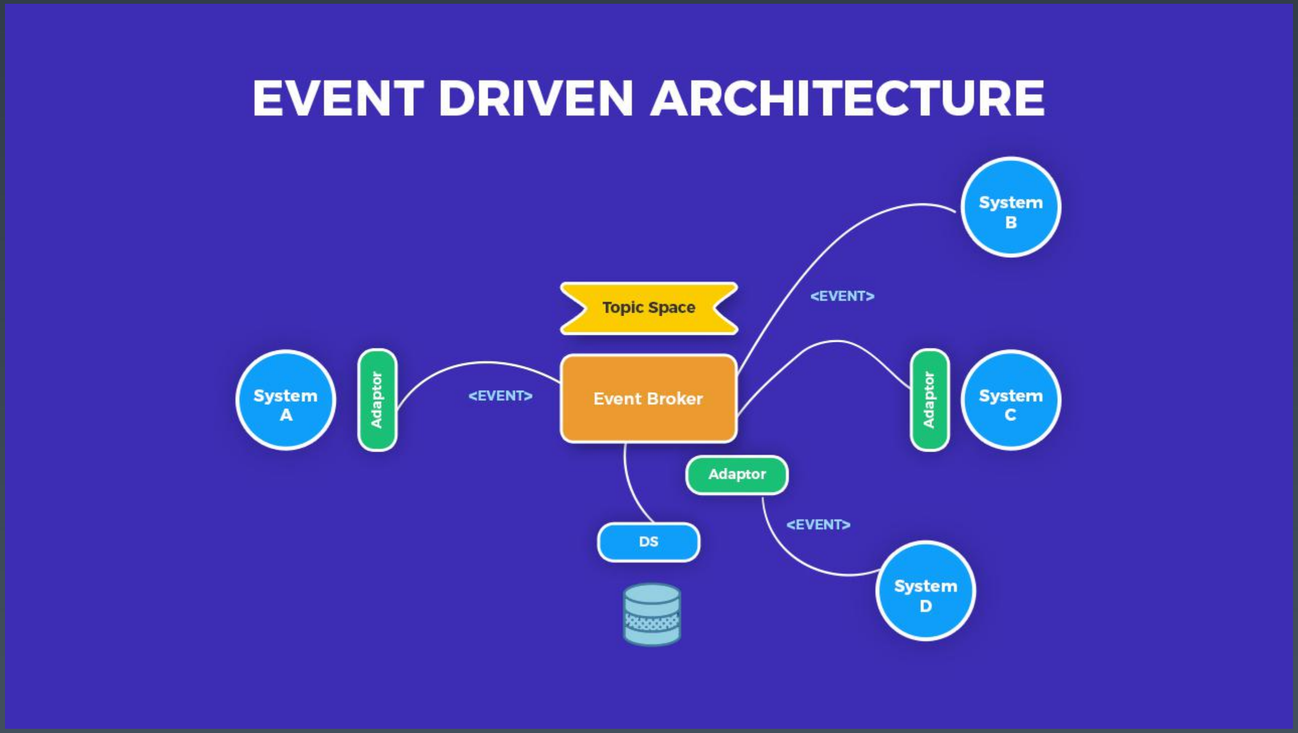
这是一个典型的解耦的场景。解耦大多数时候用于这样一个场景：我需要告诉下游我做了某件事（或者我达到了某个状态），但是我完全不知道也不关心哪些下游会关心我做了某件事（或者达到了某个状态）



如果用 RPC 来实现类似的功能，意味着挨个下游调用一遍（还得保证调用成功），来了新的下游又得重新修改代码。而 Kafka 就没这种烦恼；

Kafka 使用场景——事件驱动

事件驱动应该看做是和微服务架构平级的一种架构。事件驱动的事件可以是进程内事件，也可以是跨进程事件。跨进程事件就可以使用 Kafka



事件驱动一般是和状态结合在一起，每一次状态变更就会发布一个事件，而后驱动下游执行下一步。

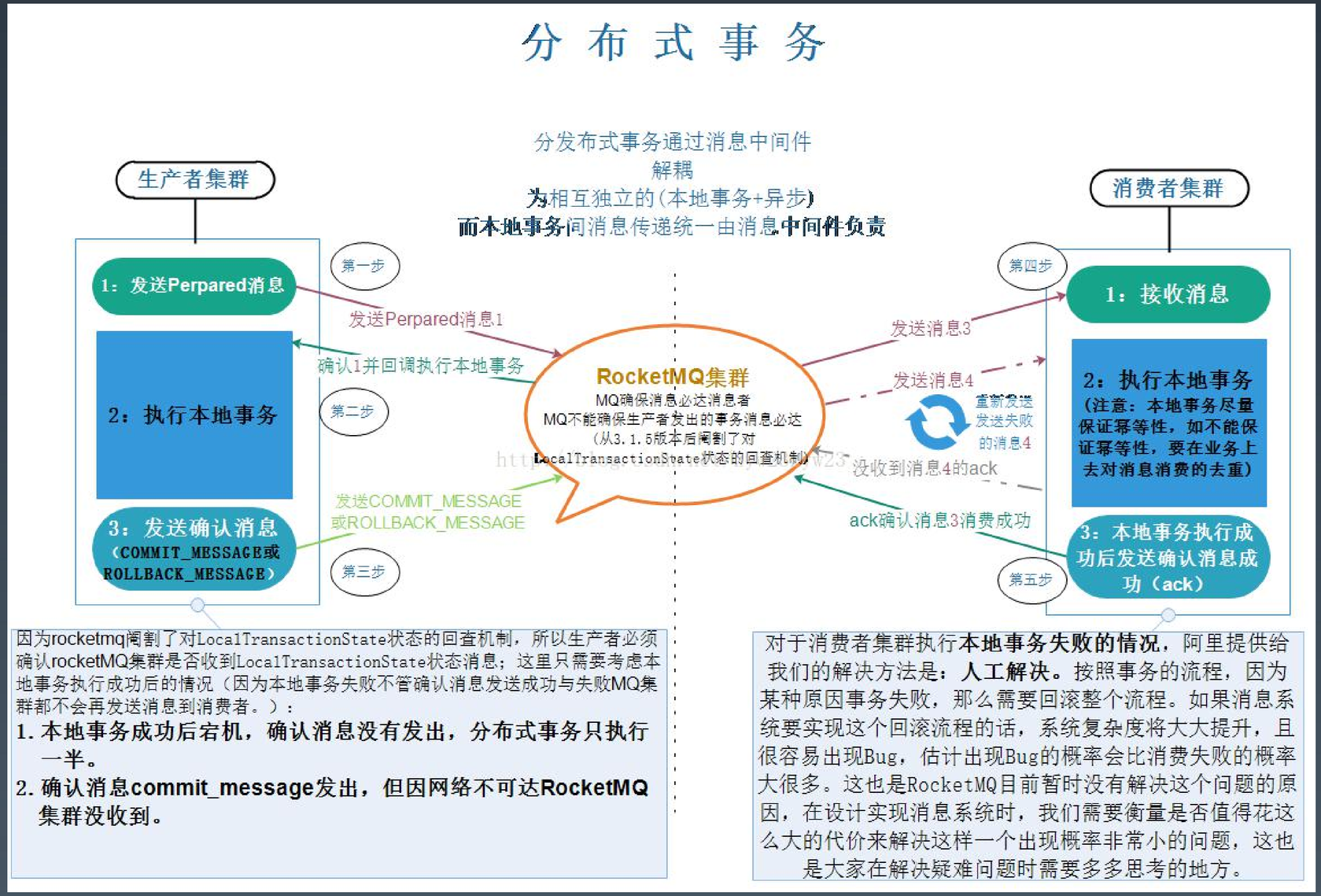
Kafka 使用场景——最终一致性的分布式事务方案

也就是我们之前在分布式事务里面聊到的：

分布式事务 => 本地事务 + 补偿机制

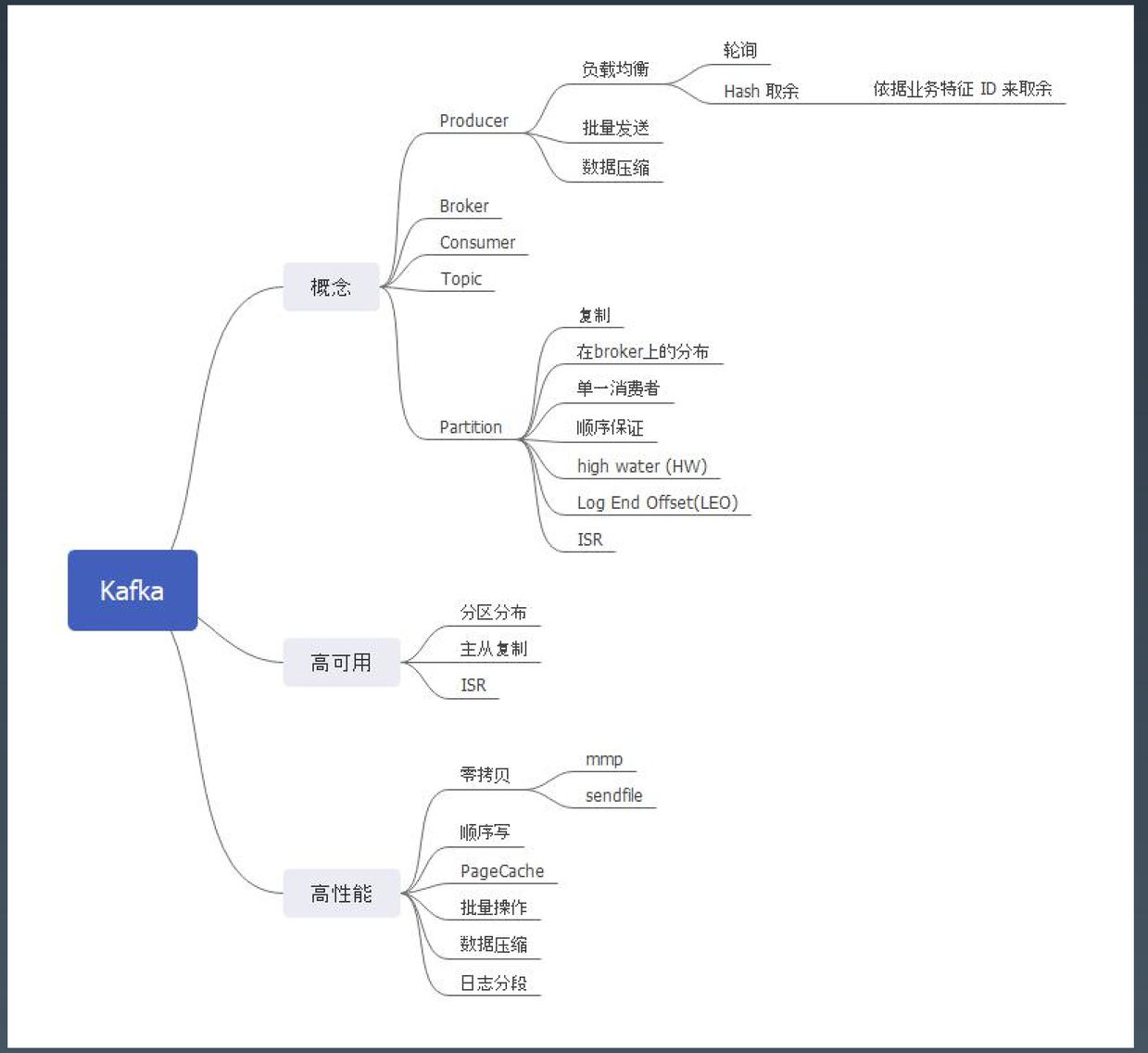
其中用 Kafka 来驱动多个本地事务

右图可以用 Kafka 来取代 rocket mq。（rocket mq 的回查机制在Kafka里面可以用本地轮询来补偿）



目录

Kafka 面试要点


Kafka
概念
Producer
负载均衡
轮询
Hash 取余
依据业务特征 ID 来取余
批量发送
数据压缩
Broker
Consumer
Topic
Partition
复制
在broker上的分布
单一消费者
顺序保证
high water (HW)
Log End Offset(LEO)
ISR
高可用
分区分布
主从复制
ISR
高性能
零拷贝
mmp
sendfile
顺序写
PageCache
批量操作
数据压缩
日志分段

Kafka 面试要点

- 你是否了解 Kafka？回答 kafka 的基本结构，也就是从 partition, broker 等几个基本概念开始回答；
- 你用消息队列做过什么？
- Kafka 的高性能是如何保证的？核心是零拷贝，Page Cache，顺序写，批量操作，数据压缩，日志分段存储；
- Kafka 的 ISR 是如何工作的？核心是理解Kafka 如何维护 ISR，什么情况下会导致一个 partition 进去（或者出来）ISR
- Kafka 的负载均衡策略有哪些？列举策略，要注意分析优缺点。更进一步可以讨论更加宽泛的负载均衡的做法，和 RPC 之类的负载均衡结合做对比
- 为什么 Kafka 的从 Partition 不能读取？违背直觉的问题，关键是要协调偏移量的代价太大；
- 为什么 Kafka 在消费者端采用了拉（PULL）模型？注意和 PUSH 模型做对比。最好是能够举一个适用 PUSH 的例子
- 分区过多会引起什么问题？又是一个违背直觉的问题，核心在于顺序写
- 如何确定合适的分区数量？
- 如何解决 Topic 的分区数量过多的问题？
- 如何保证消息有序性？方案有什么缺点？抓住核心，相关的消息要确保发送到同一个分区，例如 ID 为1的永远发到分区1
- Kafka能不能重复消费？当然可以。但是要强调，一般的消费者都要考虑幂等的问题
- 如何保证消息消费的幂等性？就是去重，简单就是数据库唯一索引，高级就是布隆过滤器 + 唯一索引
- 如何保证只发送（或者只消费）一次？属实没必要，做好消费幂等简单多了
- Rebalance 发生时机，rebalance 过程，rebalance 有啥影响？如何避免 rebalance？核心把 rebalance 的过程背下来
- 消息积压怎么办？没啥好办法，也就是加快消费，合并消息

https://github.com/flycash/interview-baguwen/tree/main/mq

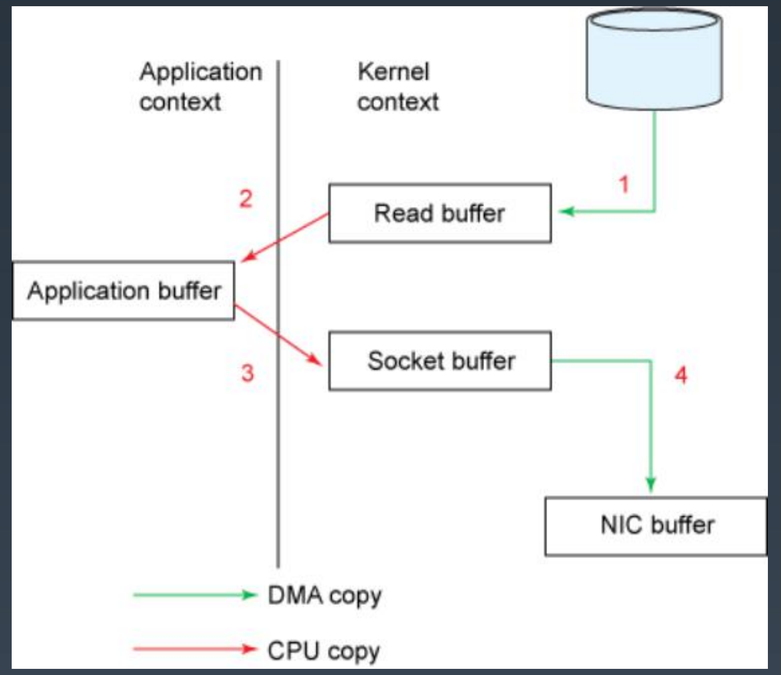
目录

中间件设计的通用技术

- 零拷贝
- AOF 和 WAL
- 写文件与刷盘
- 集群模式
- 主从模式

中间件设计通用技术——零拷贝

零复制（英语：Zero-copy；也译零拷贝）技术是指计算机执行操作时，CPU不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省CPU周期和内存带宽。

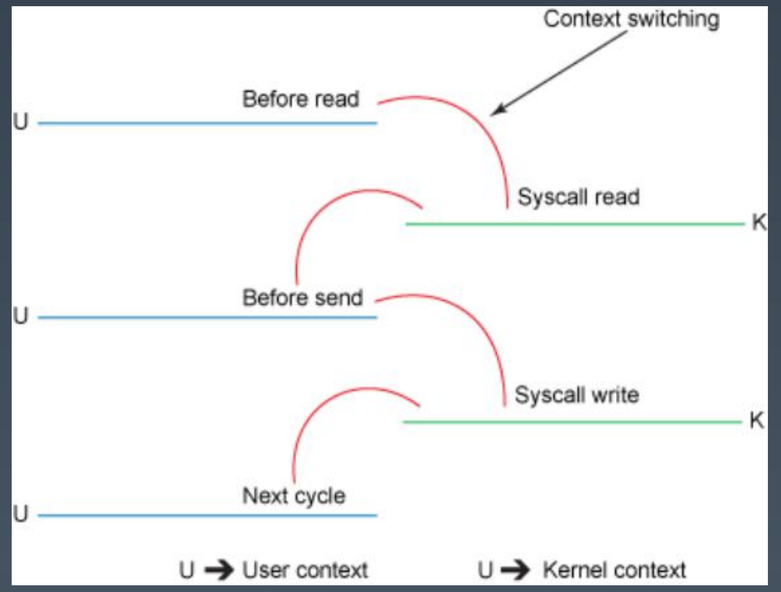


右边是一个典型的读取数据之后输出到某个地方。

比如说从文件读取，输出到网络；

或者从网络读取，输出到文件；

例子：Kafka, RabbitMQ



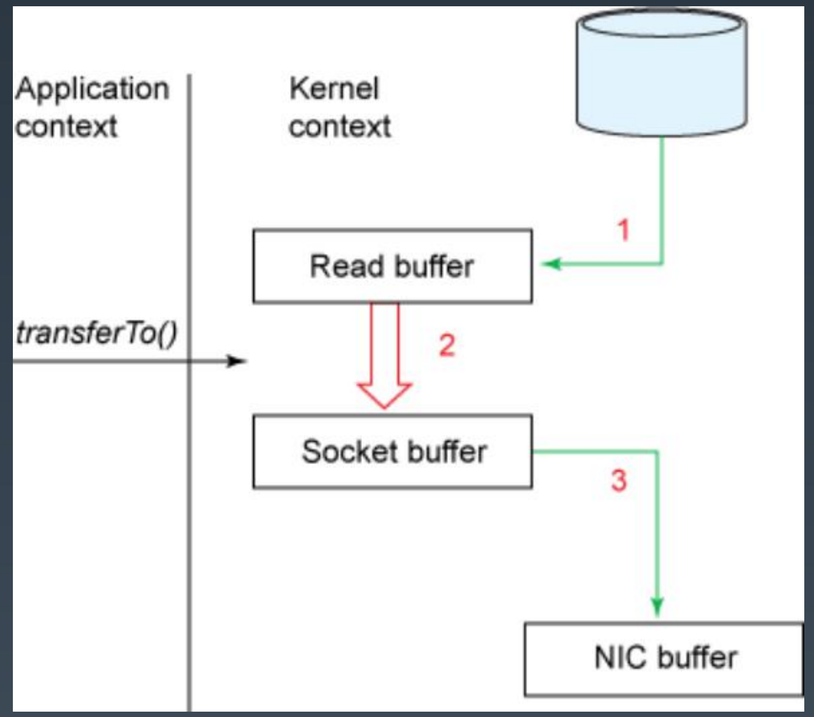
第一步想到的就是我们去除用户空间的切换，也就是 2和3 能不能去掉

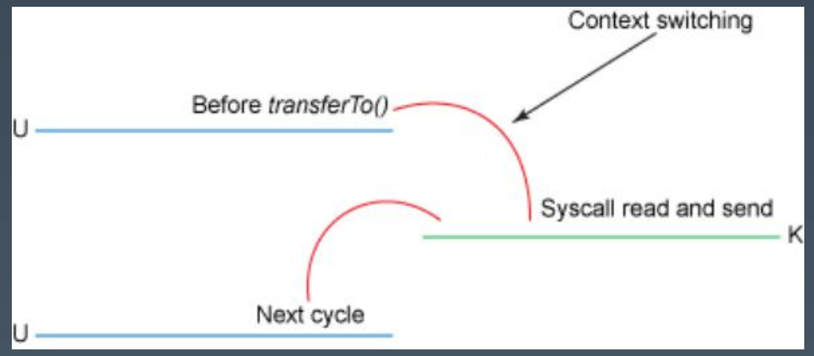
中间件设计通用技术——零拷贝

该模式省略了用户态之间的拷贝，但是依旧无法避免内核模式的内存拷贝

这就是 send_file 系统调用。

我们会考虑说，能不能 连 2 都去掉，毕竟读出来读到内核，那么写就直接从刚刚放数据那里写下去

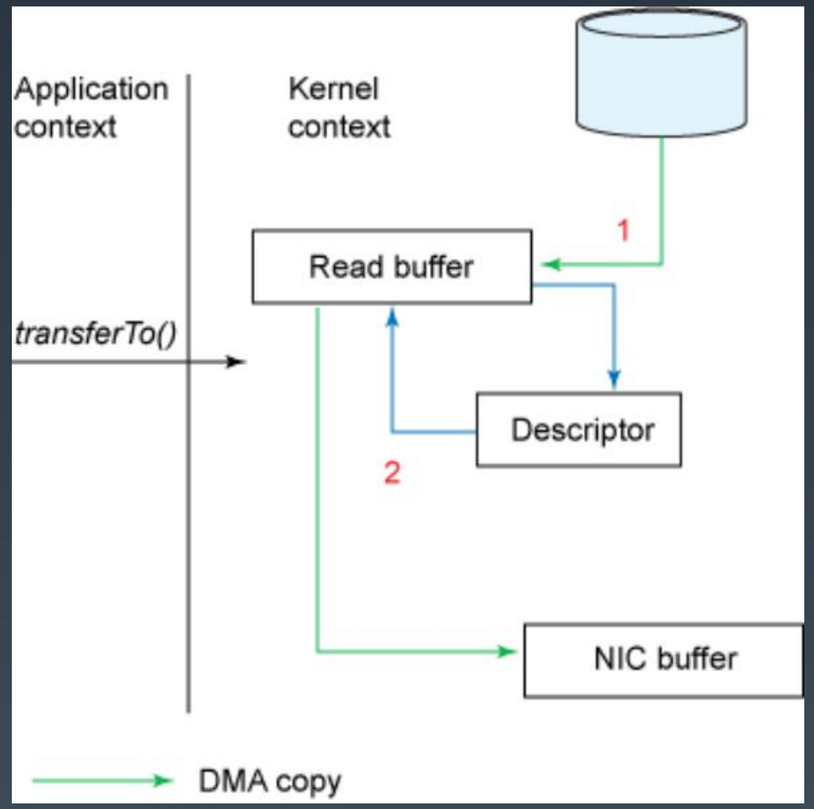




中间件设计通用技术——零拷贝

这就是目前真正的零拷贝，读出来的数据，
如果要写，它们是共用一个内存 buffer 的。

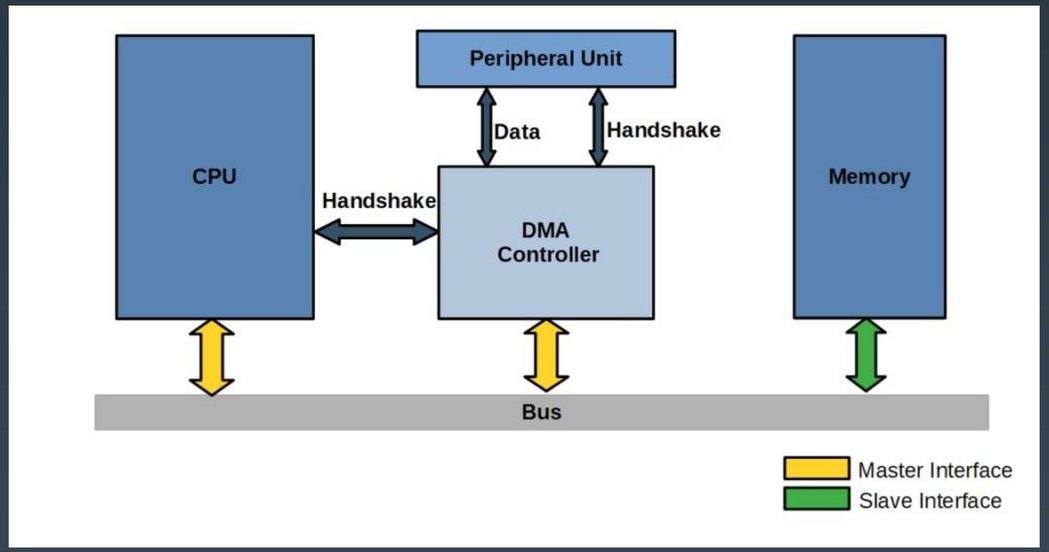
这里注意到我们用了一个 DMA 的东西。



中间件设计通用技术—— DMA

DMA：Direct Memory Access

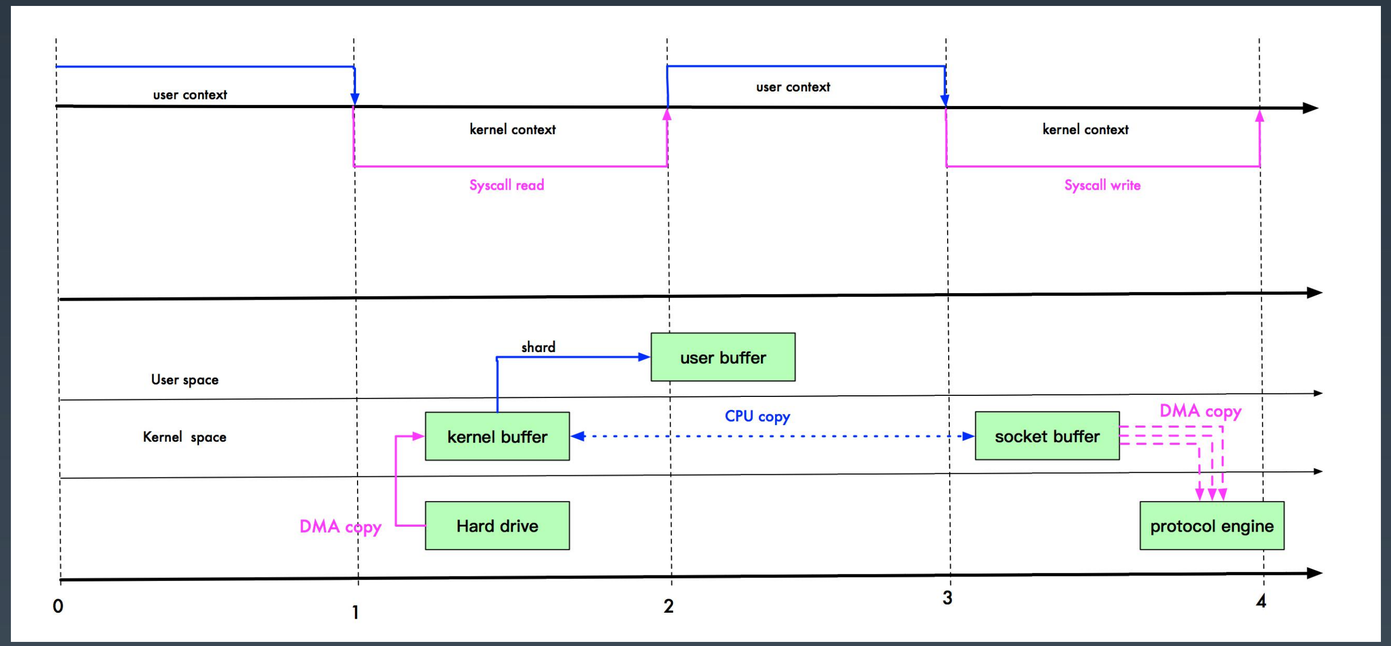
允许某些电脑内部的硬件子系统（电脑外设），可以独立地直接读写系统内存，而不需中央处理器（CPU）介入处理 。DMA是一种快速的数据传送方式

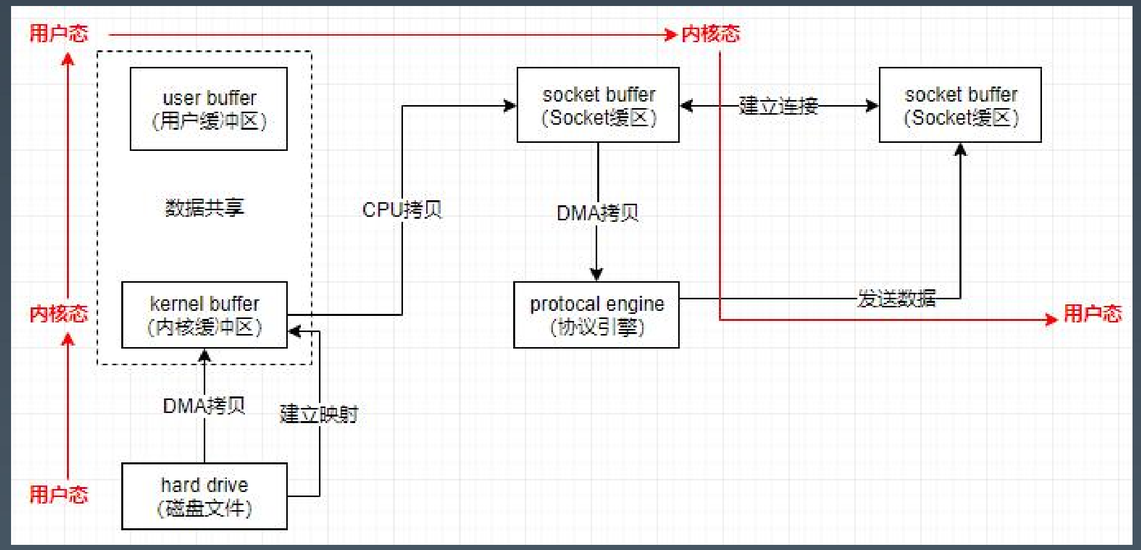


中间件设计通用技术—— mmap

mmap 实际上和原始的比起来只是少了一次拷贝。

右图中，user buffer 和 kernel buffer 共享同一块映射的数据





中间件设计的通用技术

- 零拷贝
- AOF 和 WAL
- 写文件与刷盘
- 集群模式
- 主从模式

中间件设计通用技术—— AOF 和 WAL

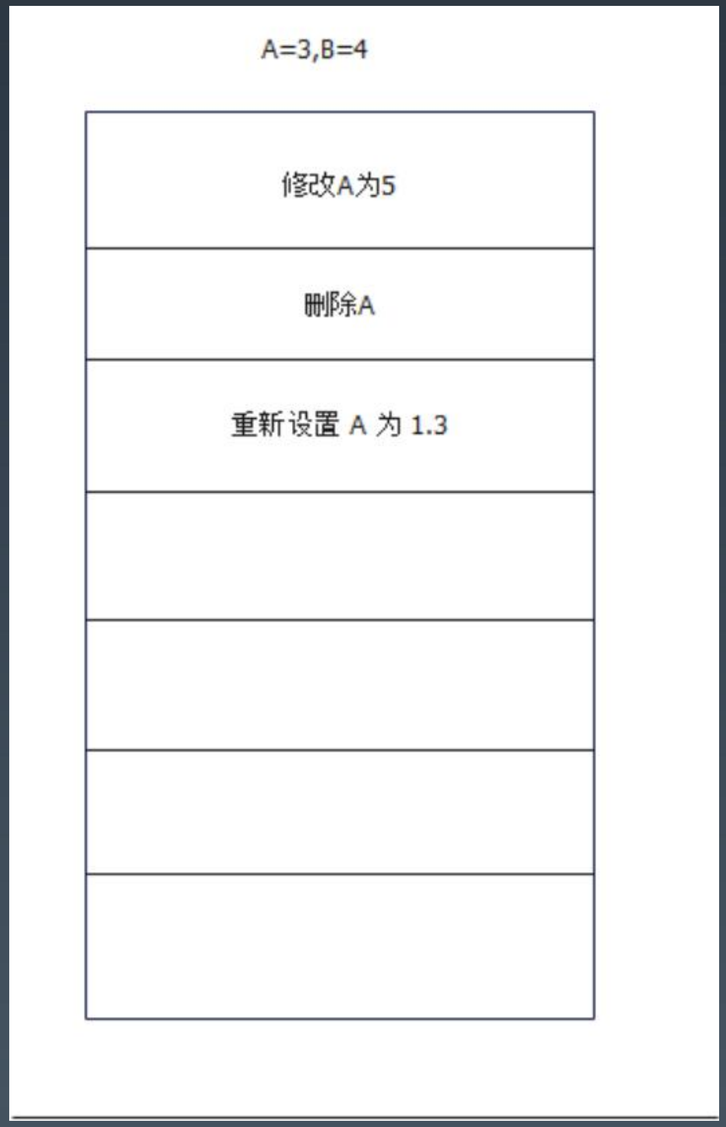
AOF：Append-Only File

WAL：Write-Ahead Log

总体上，如果 AOF 记录的是日志，那就是 WAL

特点：

- 只追加，不允许修改；
- 将随机写转成顺序写；



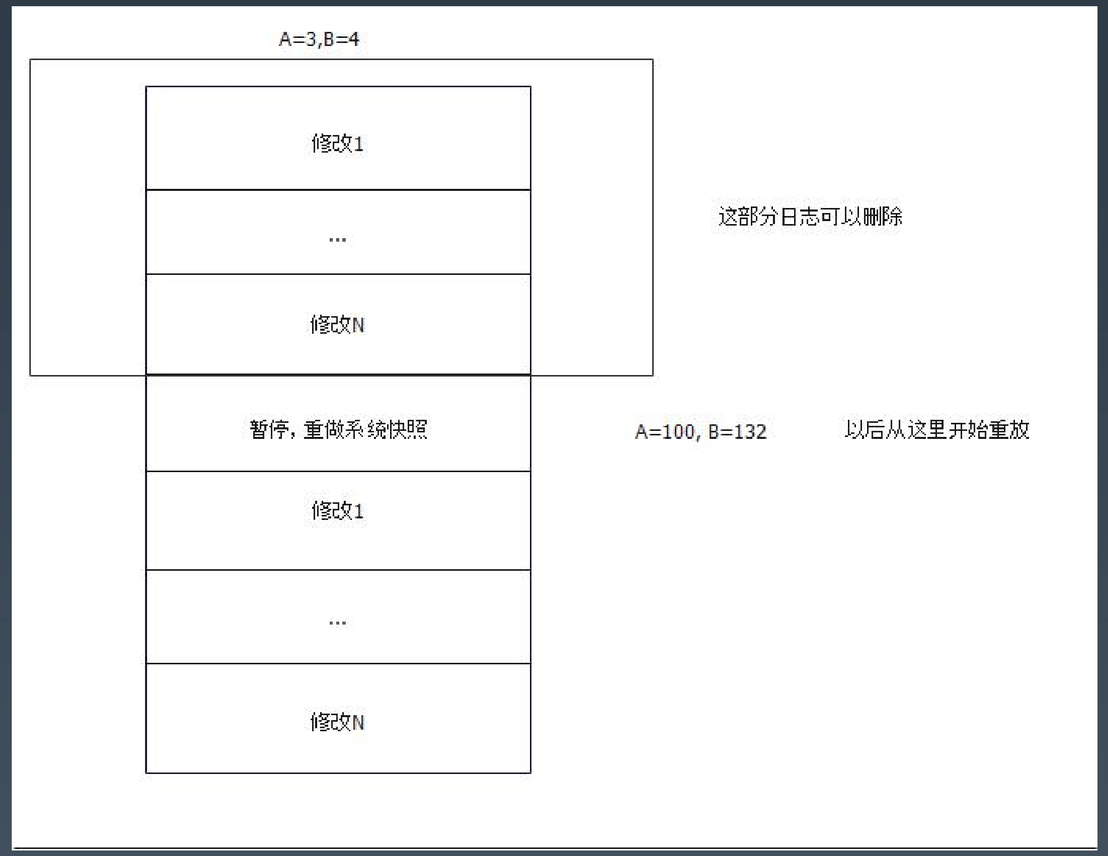
中间件设计通用技术—— AOF 和 WAL

一般的 AOF 用法：

1. 追求状态变更（但不一定是直接记录变更好的状态，可能连变更本身也会记录）

2. 通过重放 AOF 文件，能够将系统状态恢复到一直状态；

3. AOF 文件重写：一般是指直接将系统整个状态记录下来，以此作为一个基点，后续系统状态就相当于从这个点开始变化。（典型的就是 Redis）。严格意义上来说，并不是重写 AOF，而是重新记录系统状态快照

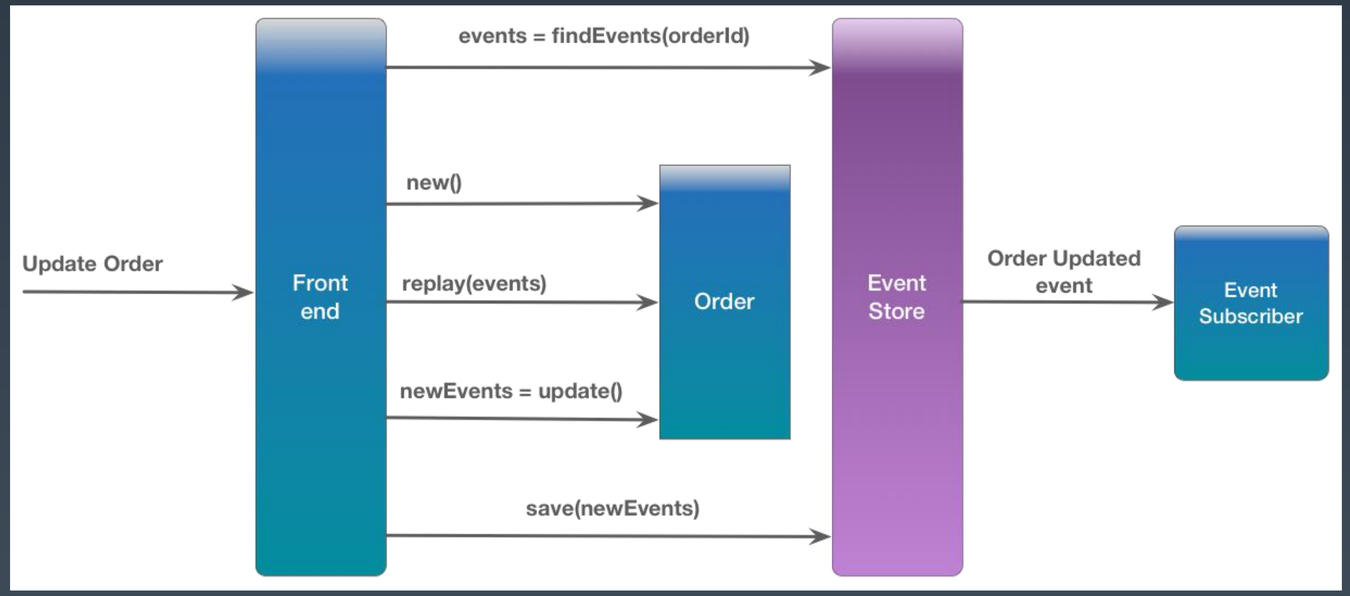


中间件设计通用技术—— AOF 和 WAL

这种模式非常类似于之前讨论过的 event sourcing.

也就是记录事件本身，但是不是记录系统当下状态。（记录引起状态变更的事件，而不是记录系统变更后的状态。因为系统变更后的状态，在你完成变更和记录下来之间有时间差，所以并发问题更难解决）

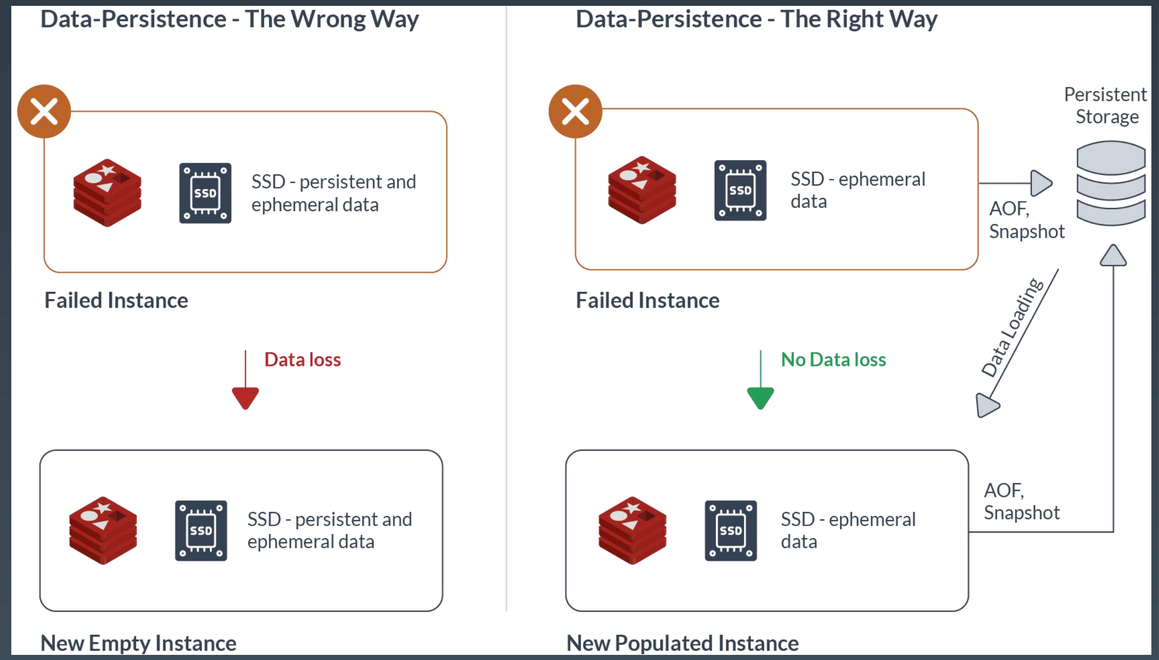
如果变更本身依赖于脱离系统的环境，那么就不能记录变更事件，而只能记录变更后的状态



中间件设计通用技术—— AOF 和 WAL

例子：

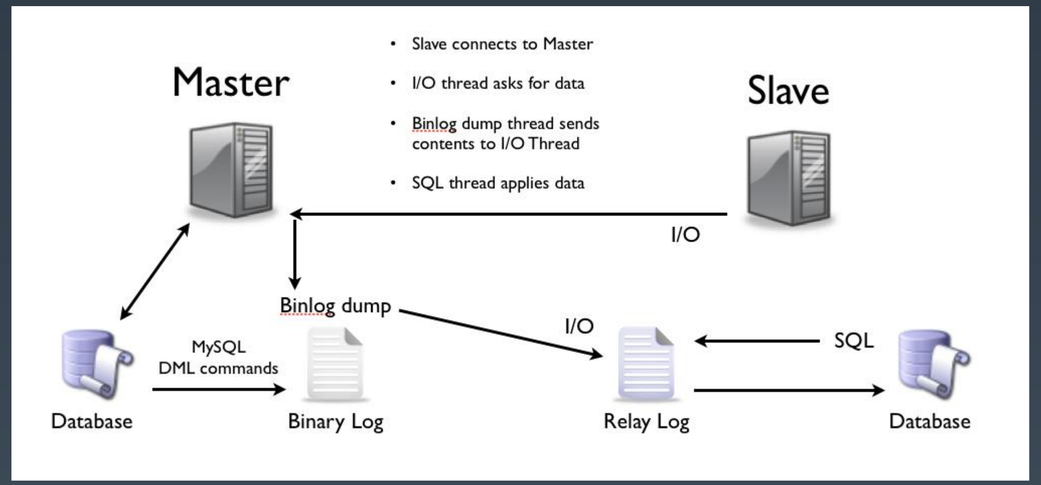
1. Kafka 顺序写：在一个分区内，新消息就是不断 Append 过去的
2. Redis AOF 持久化机制



中间件设计通用技术—— AOF 和 WAL

例子：

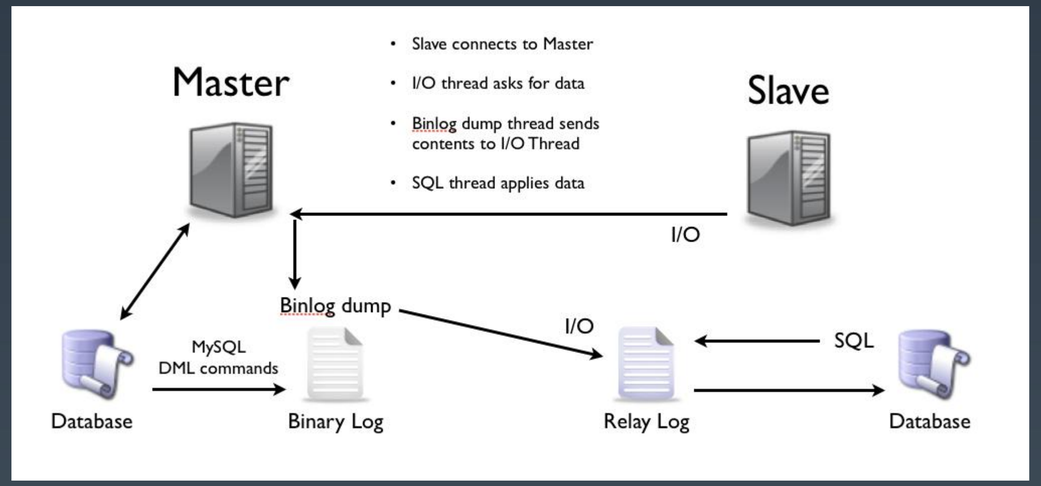
1. Kafka 顺序写：在一个分区内，新消息就是不断 Append 过去的
2. Redis AOF 持久化机制
3. MySQL binlog, redo log, undo log。（所谓 WAL 数据库）



中间件设计通用技术—— AOF 和 WAL

例子：

1. Kafka 顺序写：在一个分区内，新消息就是不断 Append 过去的
2. Redis AOF 持久化机制
3. MySQL binlog, redo log, undo log。（所谓 WAL 数据库）



MySQL binlog 有三种模式：

- 基于 SQL：也就是记录变更
- 基于行：记录变更后的状态
- 混合模式：可以认为是前两种模式的折中

中间件设计的通用技术

- 零拷贝
- AOF 和 WAL
- 写文件与刷盘
- 集群模式
- 主从模式

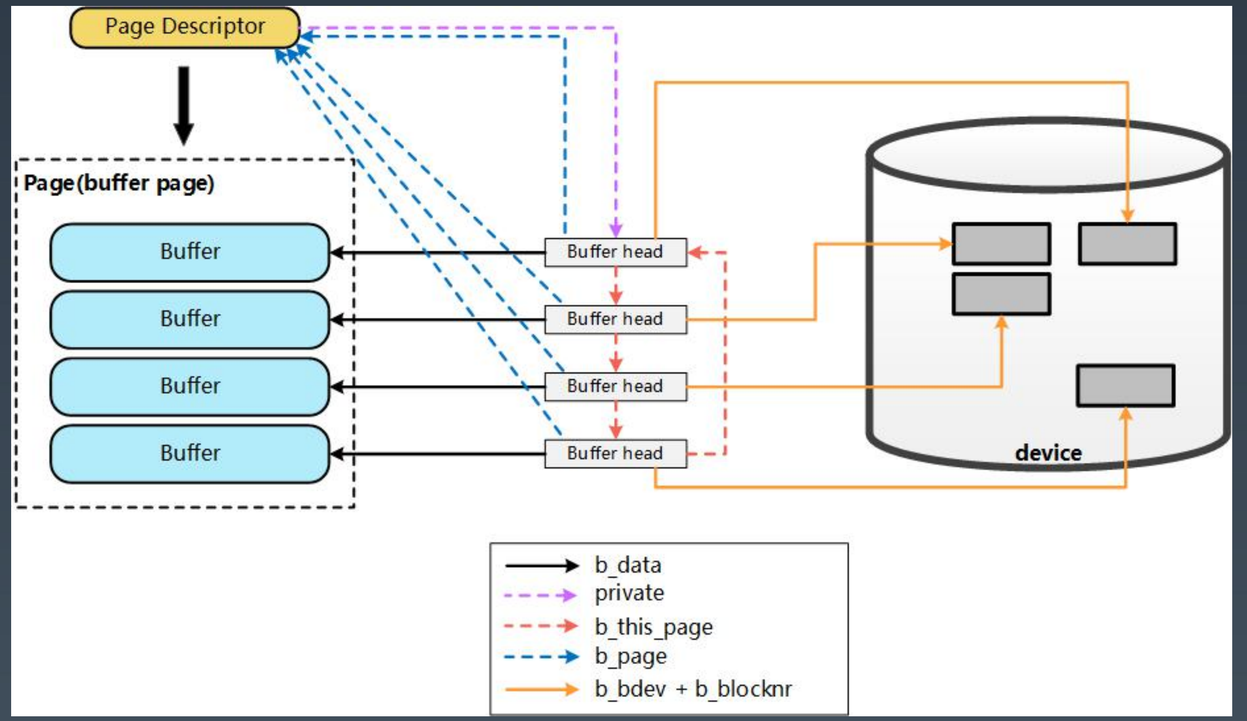
中间件设计通用技术—— 写文件

所有的中间件，如果工作在 Linux 上并且要写文件，都要考虑一个问题，什么时候真的把数据刷新到磁盘上。

Linux 文件系统有 page cache 和 buffer cache，本质上是一个东西的两种表现形式。

所以当我们讨论写文件的时候，就要仔细分辨：究竟写到了哪里？

从 Linux 的角度来看，要么就是写到了 page cache，要么就是写到了设备里面。



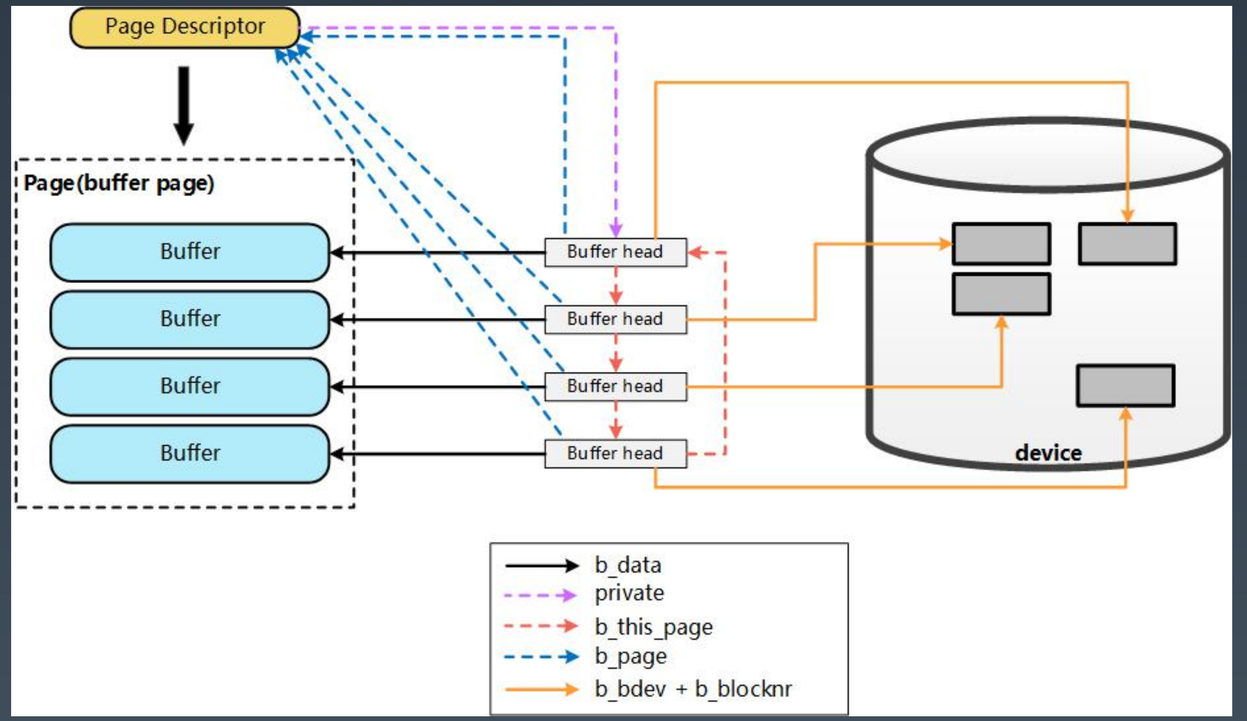
中间件设计通用技术—— 写文件

所有的中间件，如果工作在 Linux 上并且要写文件，都要考虑一个问题，什么时候真的把数据刷新到磁盘上。

Linux 文件系统有 page cache 和 buffer cache，本质上是一个东西的两种表现形式。

所以当我们讨论写文件的时候，就要仔细分辨：究竟写到了哪里？

从 Linux 的角度来看，要么就是写到了 page cache，要么就是写到了设备里面。



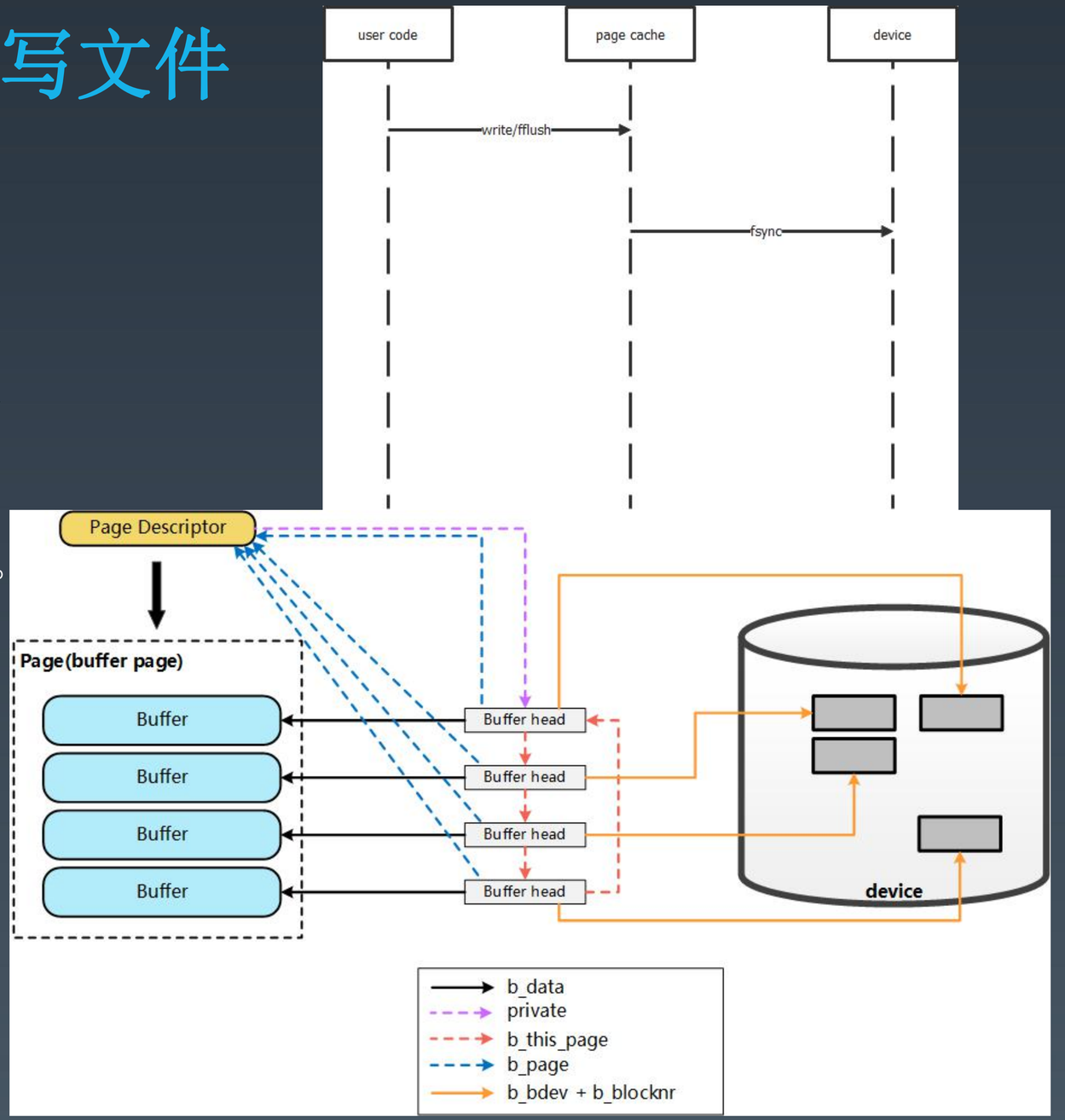
中间件设计通用技术—— 写文件

本质上，page cache 可以看成是一个 write-through 的缓存。

一般情况下，用户直接使用 write/fflush 只是写到了 page cache 里面。

这个时候如果宕机，那么依赖会丢失掉数据。

可以依赖于 Linux 自动刷新，也可以自己强制发起系统调用来刷新



中间件设计通用技术—— 写文件

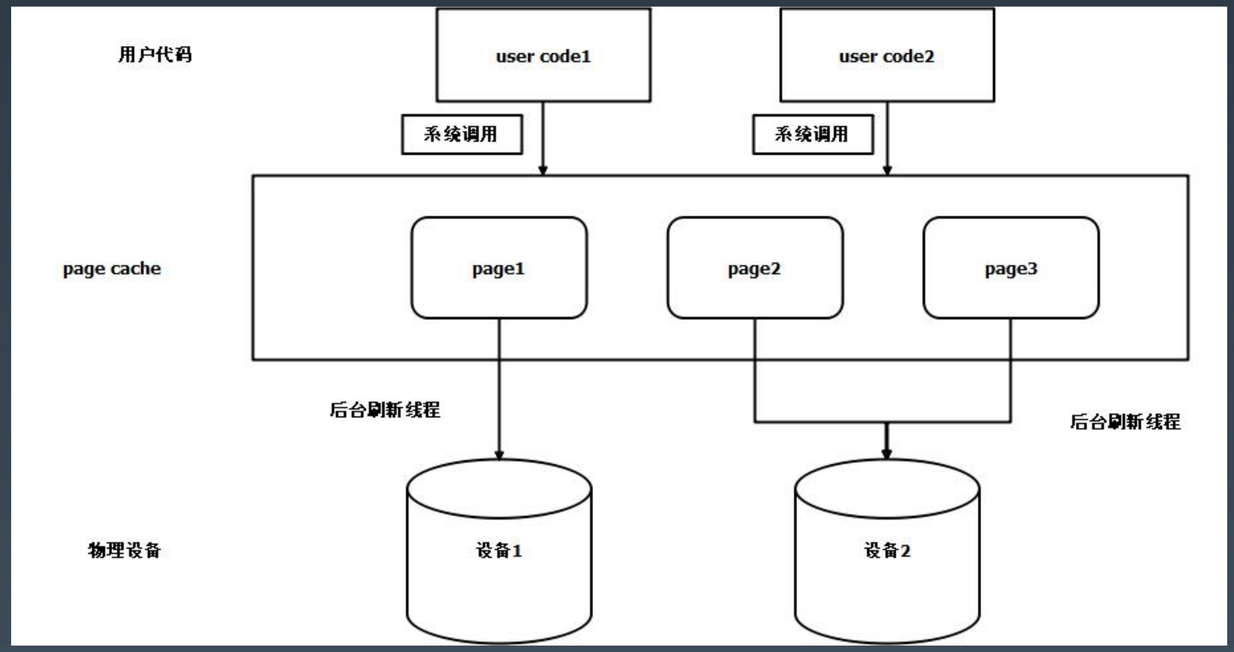
Linux 自动刷新 page cache：

linux 对每一个设备维持了一个后台线程，用于回写 page cache。

会在两种情况下触发：

1. 脏页很多
2. 周期性刷新

其实对于用户代码来说，整个过程是不可控的。因此依赖于自动刷新机制，就要容忍数据丢失

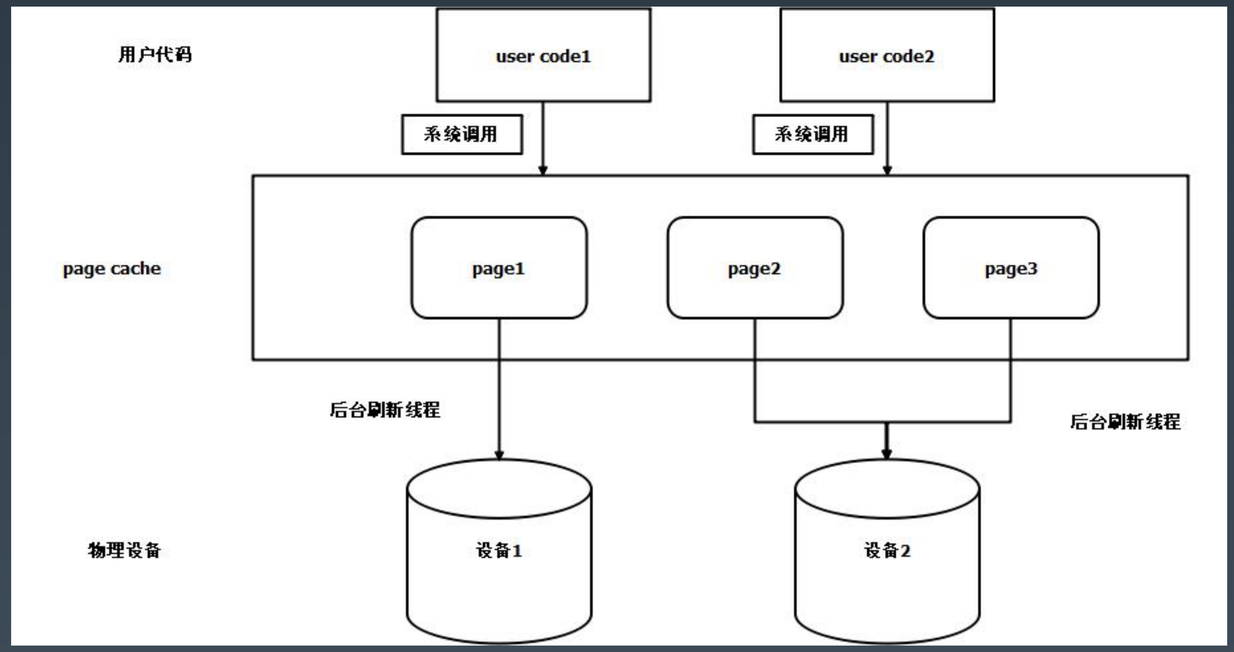


中间件设计通用技术—— 写文件

强制刷新：用户主动调用 fsync, msync, sync 几个系统调用。

那么问题来了，什么时候调用比较合适？

- 为了保险起见，每一次我都刷新到磁盘
- 为了性能考虑，我就完全依赖于操作系统，它爱什么时候刷，什么时候刷
- 前面两个都不合适，我就自己间隔一段时间刷，或者我自己数数更新了多少数据，到达阈值我就刷新



中间件设计通用技术—— 写文件

MySQL binlog：刷盘可以通过 sync_binlog参数来控制

- 0-系统自由判断
- 1-commit刷盘
- N-每N个事务刷盘

MySQL redo log：通过参数 innodb_flush_log_at_trx_commit 控制

- 0-写入 log buffer，每秒刷新到盘；
- 1-每次提交刷盘；
- 2-写入到 OS cache，每秒刷盘；

Redis AOF 刷盘时机：

- always: 每次都刷盘
- everysec: 每秒刷盘
- no: 由操作系统决定

Kafka 刷盘时机：

- log.flush.interval.ms：间隔多少毫秒刷新
- log.flush.interval.messages：每多少条消息刷新一次

但凡聊到可靠性和持久化的时候，都要注意刷盘时间

中间件设计通用技术—— 写文件

写入：

1. 写到中间件自己的 log buffer 就返回，需要考虑何时刷到系统缓存或者磁盘；
2. 写到系统缓存就返回（这种可能是因为自身没有 log buffer），要考虑刷新到系统磁盘的时机
3. 直接刷新到磁盘

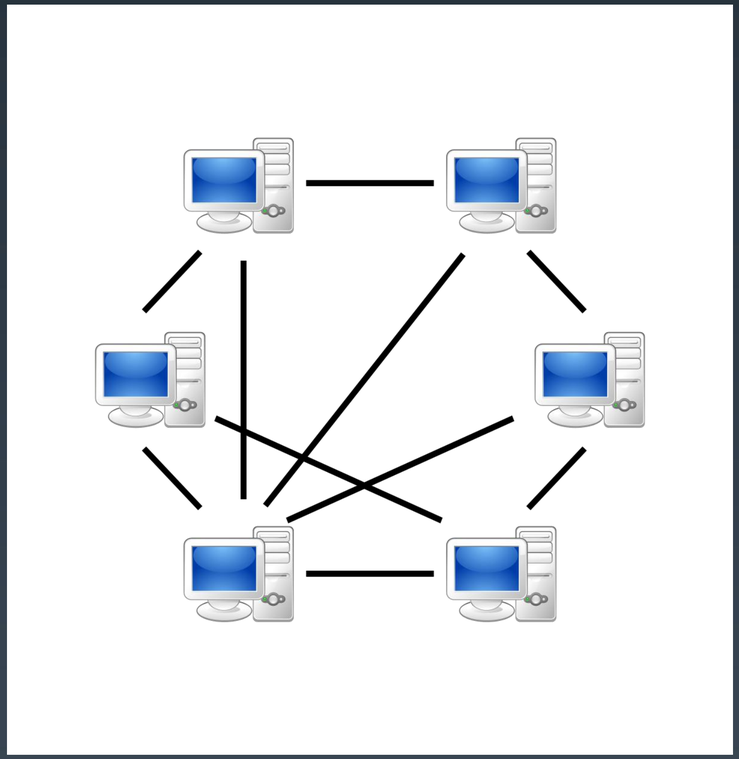
刷盘策略：

1. 每次都刷
2. 按次数
3. 按数据量
4. 按时间间隔

中间件设计的通用技术

- 零拷贝
- AOF 和 WAL
- 写文件与刷盘
- 集群模式
- 主从模式

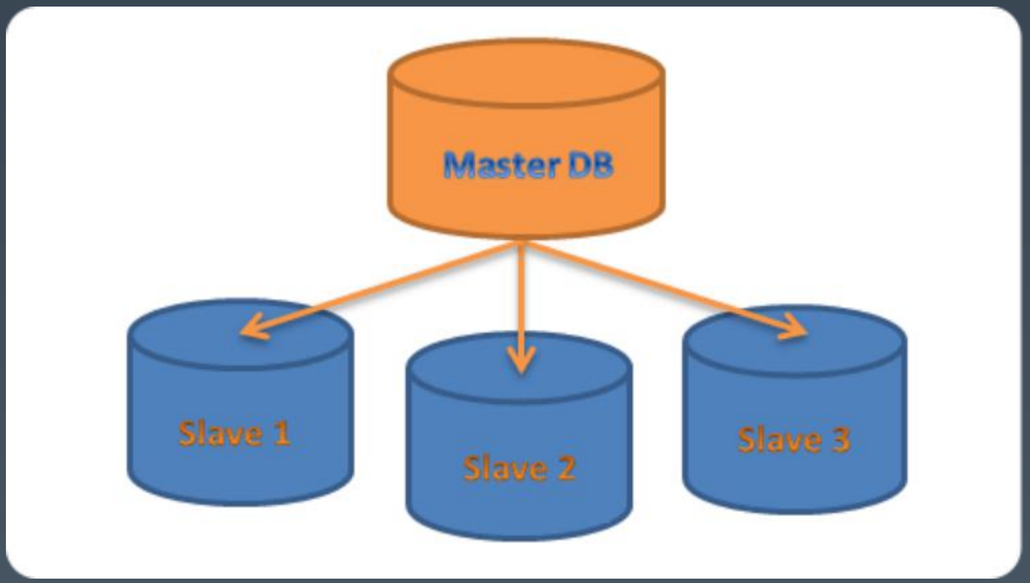
中间件设计通用技术—— 集群模式



核心的就是两种：

主从模式：是指一个集群里面，有一个主节点，其余节点是从节点。一般从节点会保持和主节点的数据同步，从节点提供只读服务，典型的是 MySQL 主从模式，Kafka 也可以看做是主从集群；

对等模式：每一个节点地位都是平等的，节点可以直接对外服务，也可以将请求路由到另外一个节点；每一个节点可以是包含全量数据，也可以只包含部分数据，Redis Cluster 就是对等模式；



中间件设计通用技术—— 集群模式

两种模式对比：

主从模式下，主节点容易成为写瓶颈；

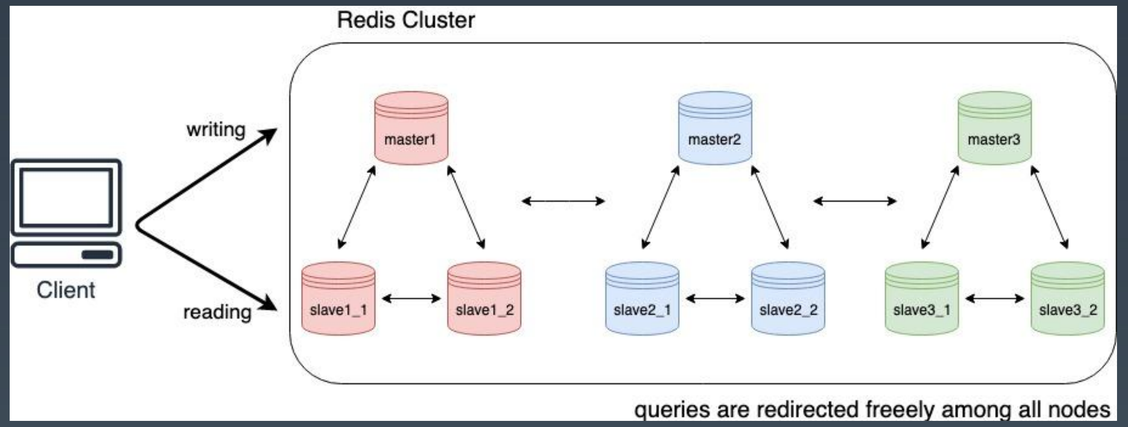
主从模式会出现脑裂的问题；

对等模式的复杂度和通信负载都要大；

对等模式所有节点之间达成一致要比主从模式难；

对等模式下写负载均匀分布；

对等模式如果节点只有部分数据，需要引入额外的机制（转发 or 中央控制）；



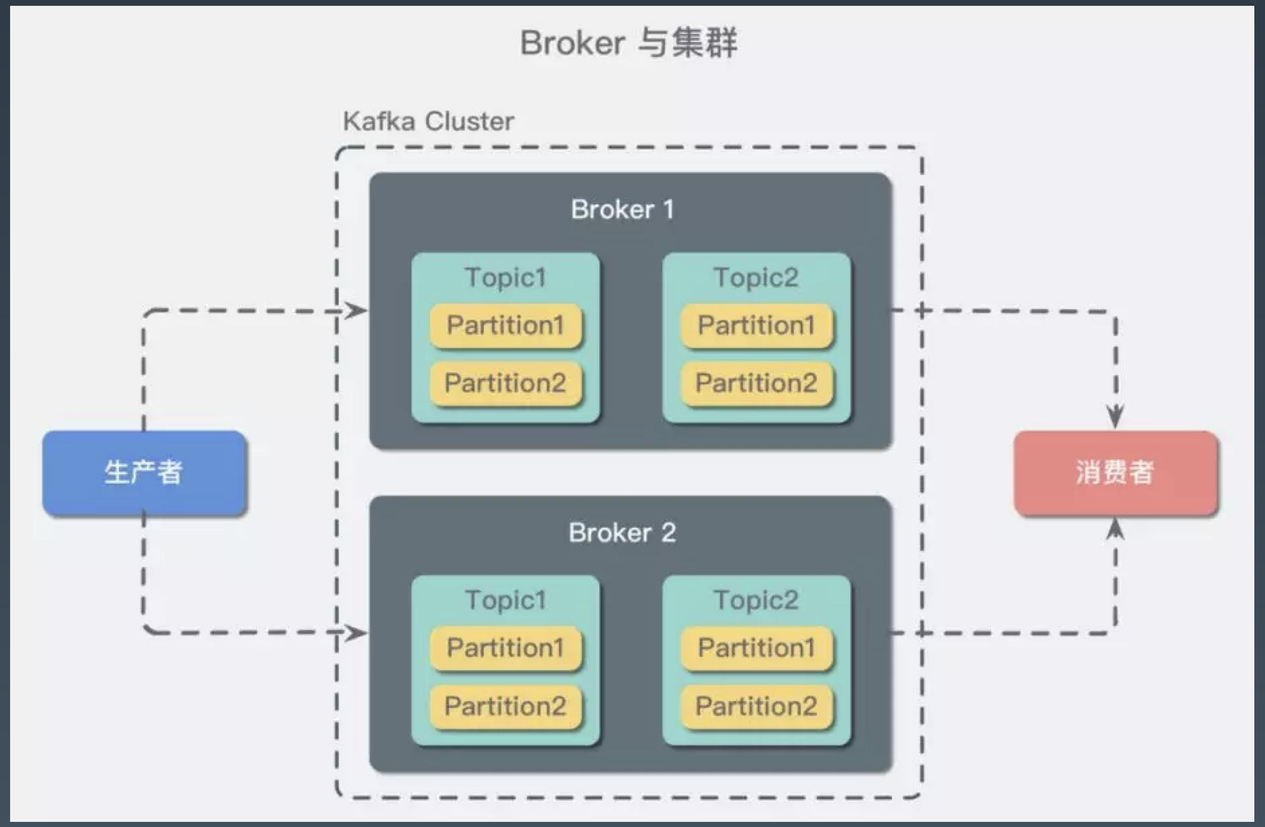
中间件设计通用技术—— 集群模式

例子：

Kfaka 集群：对于分区来说，有主从之分，所以是主从模式。但是从 broker 的角度看过去，一般每个 broker 都会有某个分区的主分区，broker 之间可以看成是对等的。

根据前面的讨论，因为分区只有部分数据，所以要么分区之间进行转发，要么要有一个协调者告诉客户端发到哪个分区。

Kafka 选择了协调者。也就是客户端要知道目标主分区在哪个 broker 上。而后从分区会和主分区保持同步



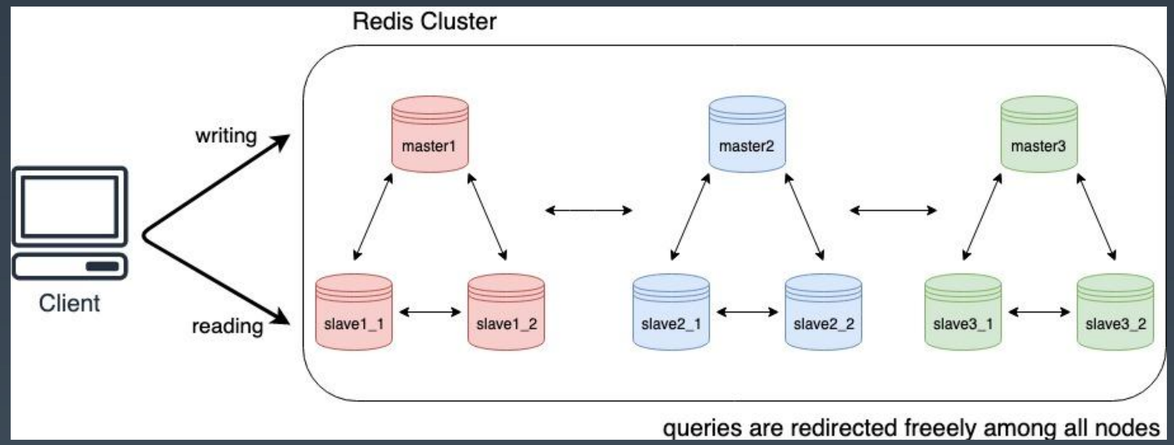
中间件设计通用技术—— 集群模式

例子：

Redis Cluster 整体是一个对等集群，但是每一个节点又同时是一个主从集群。即多个主从集群组合成了一个大的对等集群；

同样的，Redis Cluster 也需要客户端知道请求发到哪个节点，不过它采用的是转发模式。

所以衍生出来了智能客户端的说法。

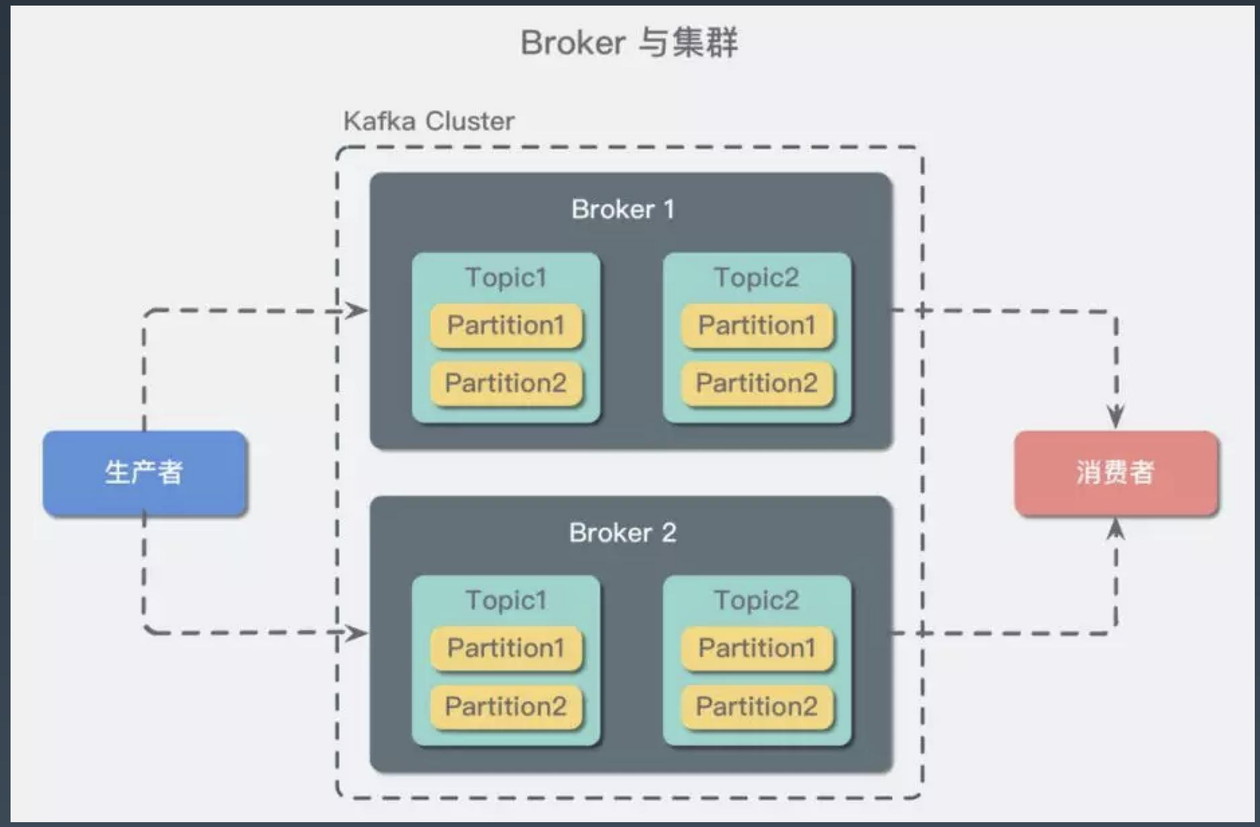


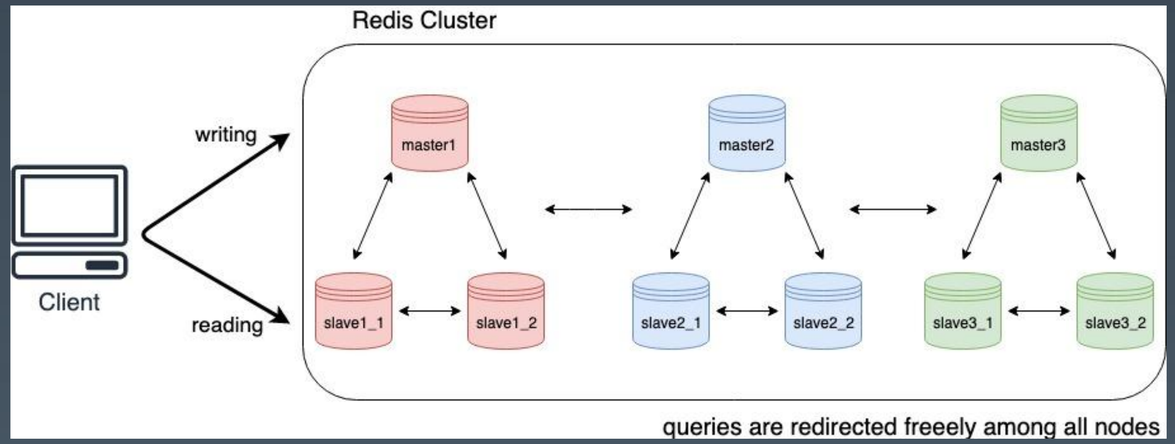
中间件设计通用技术——

一般主从和对等结构混合都是为了：

1. 用主从模式来解决可靠性问题，即数据通过同步保证不丢失；

2. 用对等结构来解决主节点性能瓶颈和脑裂的问题；





中间件设计的通用技术

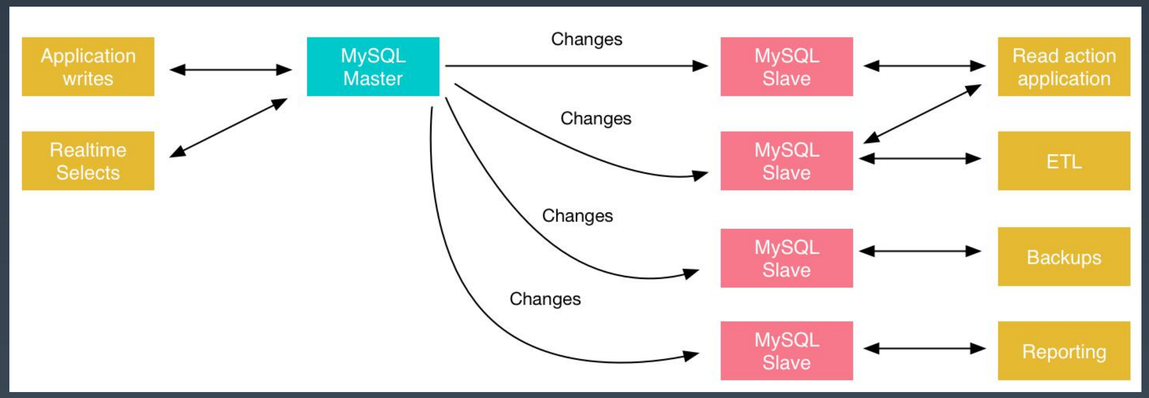
- 零拷贝
- AOF 和 WAL
- 写文件与刷盘
- 集群模式
- 主从模式

中间件设计通用技术—— 主从同步

从思路上来说，可以看做是跨网络的 AOF。

一般来说是主节点将（数据）变更事件通知从节点：

1. MySQL 的 binlog 的 SQL 模式
2. Redis 同步写命令
3. Kafka 同步消息



还有特殊的同步数据：MySQL binlog 行模式

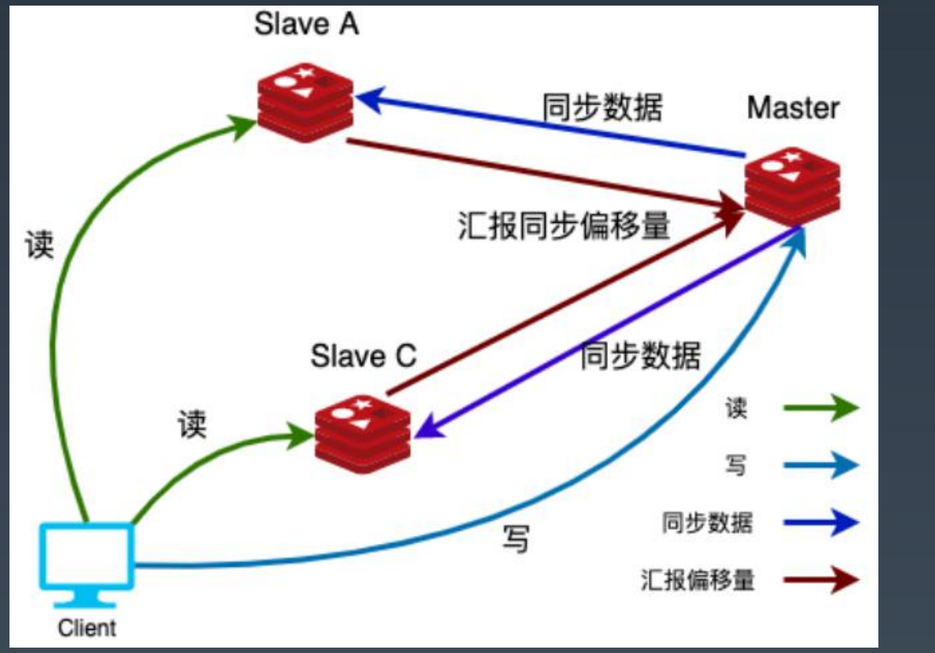
那么如果从节点和主节点失去连接之后会怎样？

中间件设计通用技术—— 主从同步

从节点和主节点失去联系之后：

1. 失去联系太久，重新连上去之后，只能全量同步。全量同步一般会选择同步数据，而不是同步变更；

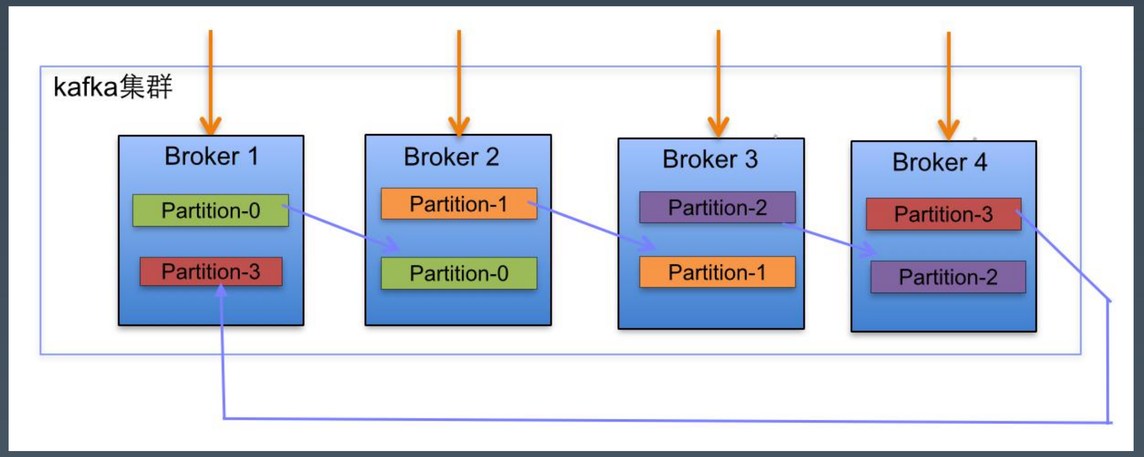
2. 失去联系不久，还能搞增量同步；



Redis 有增量同步和全量同步；

Kafka 有 ISR 和 OSR。在新副本加入的时候，可以看做是全量同步，直到赶上；后续时刻会保持增量同步。副本可能在 ISR 和 OSR 之间反复横跳；

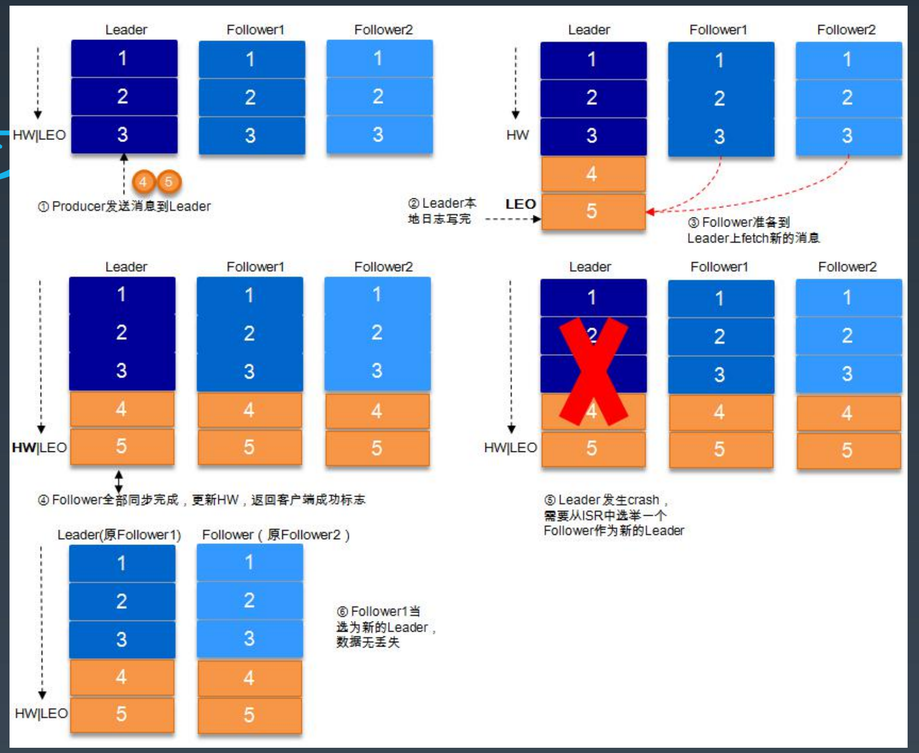
MySQL 如果加入新节点，我们会建议直接同步数据，而不是同步 binlog（此时 binlog 可能非常大），而后保持 binlog 同步

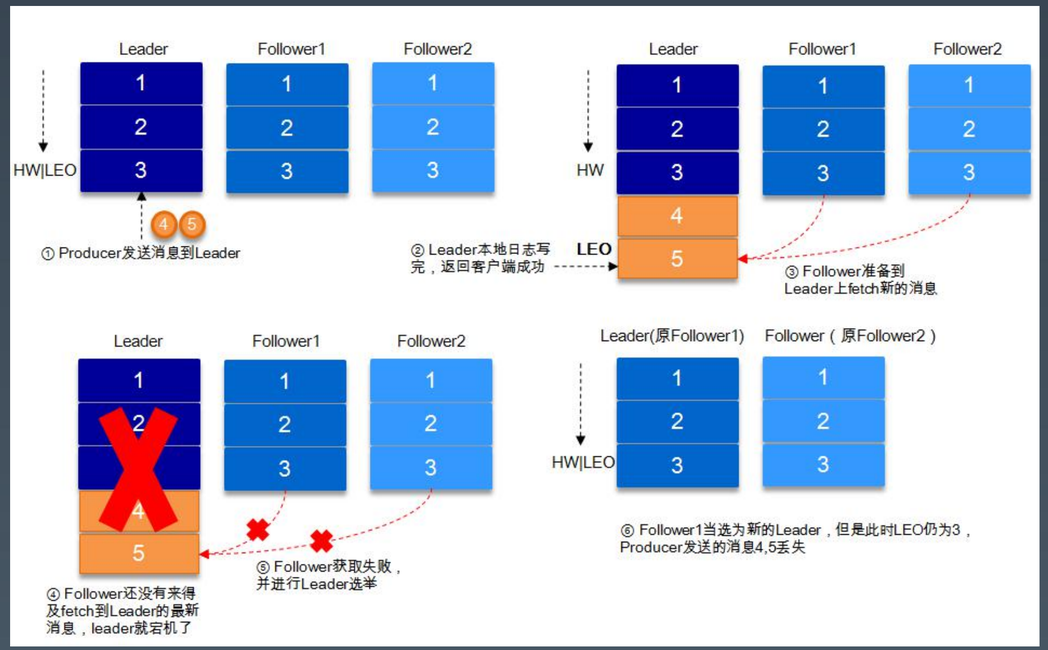


中间件设计通用技术—— 主从同步

主节点写入数据的时候，要不要一起写从节点？如果写，写多少从节点？

- 主节点写了就返回，Kafka request.required.acks=1，Redis 主从同步，MySQL 异步复制；
- 主节点写了之后，同步至少一个从节点就返回， MySQL 半同步；
- 主节点写了之后，同步部分从节点就返回，取决于配置；
- 主节点写了之后，同步多数从节点之后返回；
- 主节点写了之后，同步全部从节点之后才返回，MySQL 全同步；

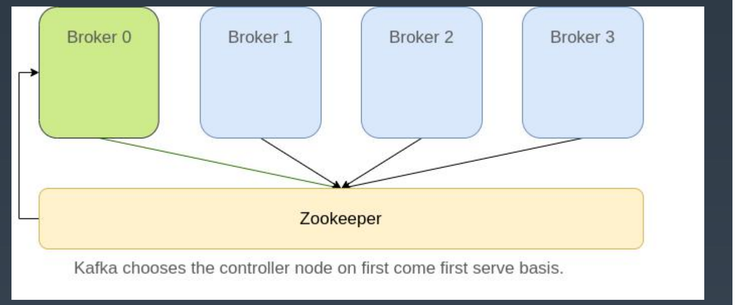


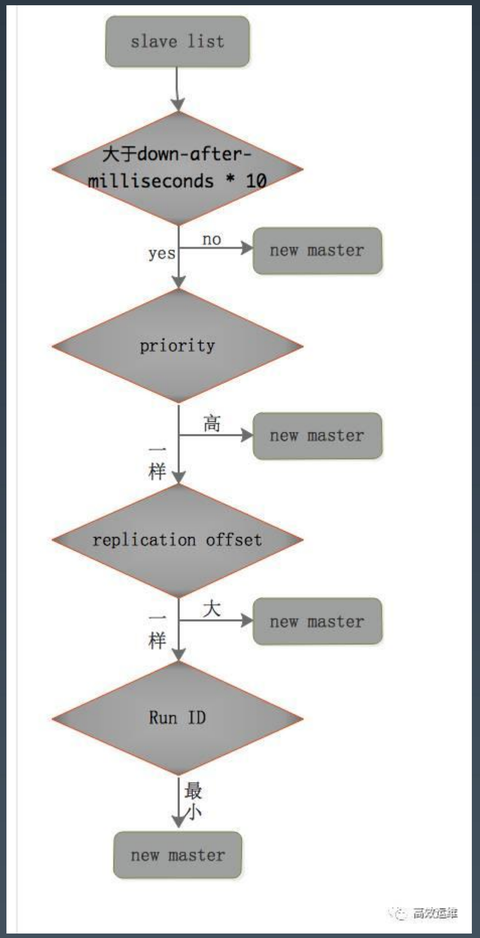


中间件设计通用技术—— 主从同步

主节点崩了怎么办？怎么挑从节点？挑哪个从节点？

- 从节点自己选举：Kafka controller 选举是通过竞争来实现的（在 zk 上注册一个节点），Redis Sentinel 选举
- 第三方挑一个从节点：Kafka controller 挑分区；Redis Sentinel 挑主节点





选谁？

- 数据最新：大多数选择
- 按照权重（优先级）：Redis 是按照优先级-偏移量来选择