微服务统一配置中心介绍

极客时间

扫码试看/订阅

《分布式缓存高手课》视频课程

常用开源统一配置中心介绍

- Apollo
  - Apollo（阿波罗）是携程框架部门研发的分布式配置中心，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性，适用于微服务配置管理场景。

- Qconf
  - QConf 是一个分布式配置管理工具。 用来替代传统的配置文件，使得配置信息和程序代码分离，同时配置变化能够实时同步到客户端，而且保证用户高效读取配置，这使的工程师从琐碎的配置修改、代码提交、配置上线流程中解放出来，极大地简化了配置管理工作。

常用开源统一配置中心介绍

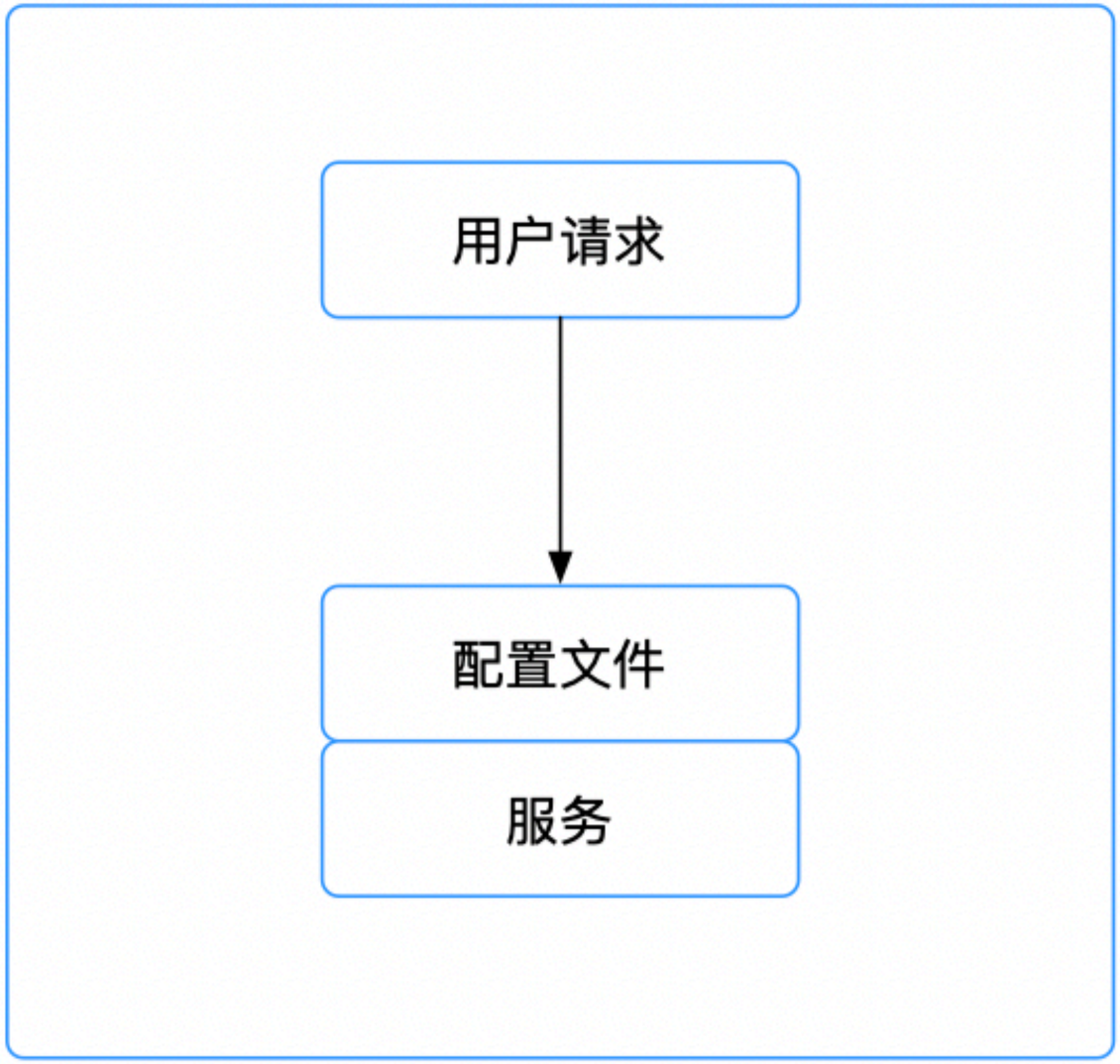
- Disconf
  - 专注于各种「分布式系统配置管理」的「通用组件」和「通用平台」，提供统一的「配置管理服务」

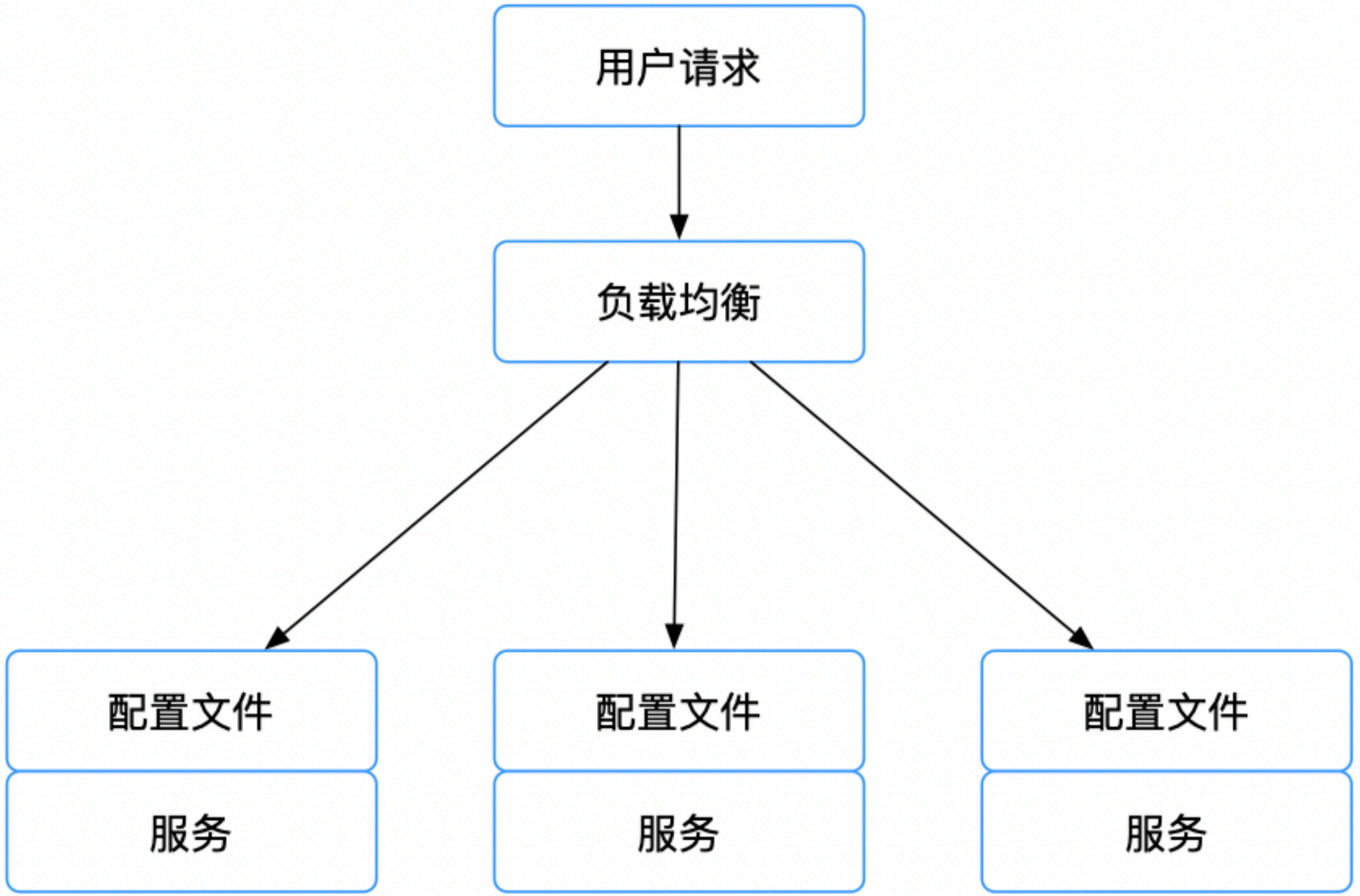
- Spring Cloud Config
  - Spring Cloud Config为分布式系统中的外部配置提供服务器和客户端支持。

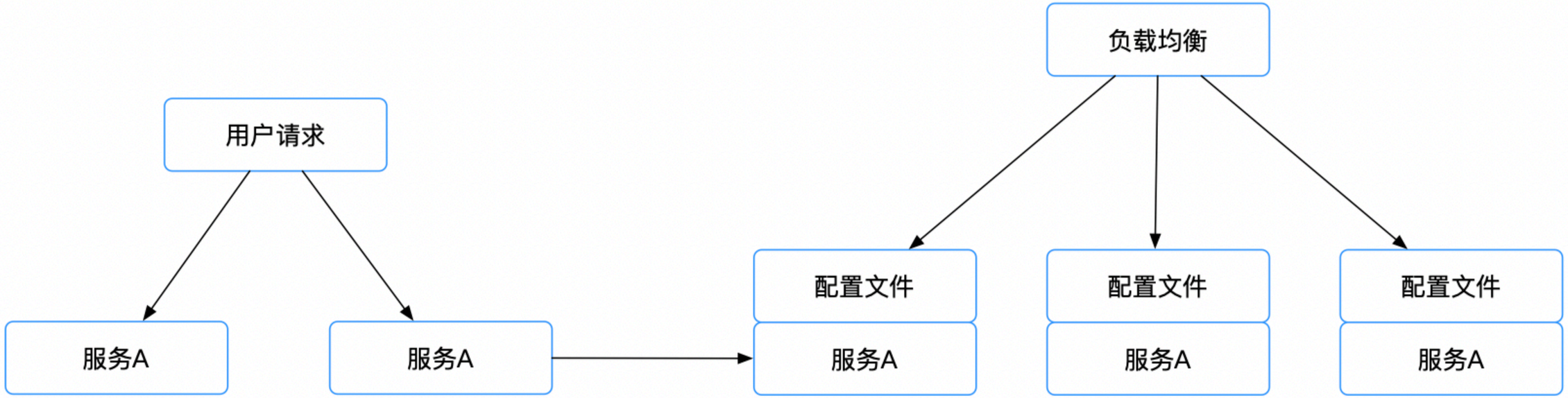
对比项目	diamond	disconf
配置存储	存储在MySQL	存储在MySQL
推拉模型	拉模型，默认每15秒拉一次	基于ZK的推模型
配置读写	支持实例对配置读写	只支持实例对配置读
容灾	多级容灾，配置数据会 Dump 在本地，避免中心服务器挂掉无法使用	多级容灾，优先读取本地配置
配置数据模型	只支持 KV 结构的数据	支持传统的配置文件方式，也支持 KV 结构
数据同步	基于数据库和本地文件 server 写数据时，先将数据写入数据库，再写入本地文件 client 订阅数据时，访问的是本地文件，不查询数据库，即使数据库出问题也不影响	基于 ZK 实时对数据进行推送，系统异常时主备切换

对比项目	Apollo	Spring Cloud Config
静态配置管理	支持	基于文件
动态配置管理	支持	支持
多环境	支持	无
配置生效时间	实时	重启生效，手动刷新
配置更新推送	支持	手动触发
配置版本管理	支持	无
灰度发布	支持	不支持
告警通知	支持	不支持
用户权限	支持	无

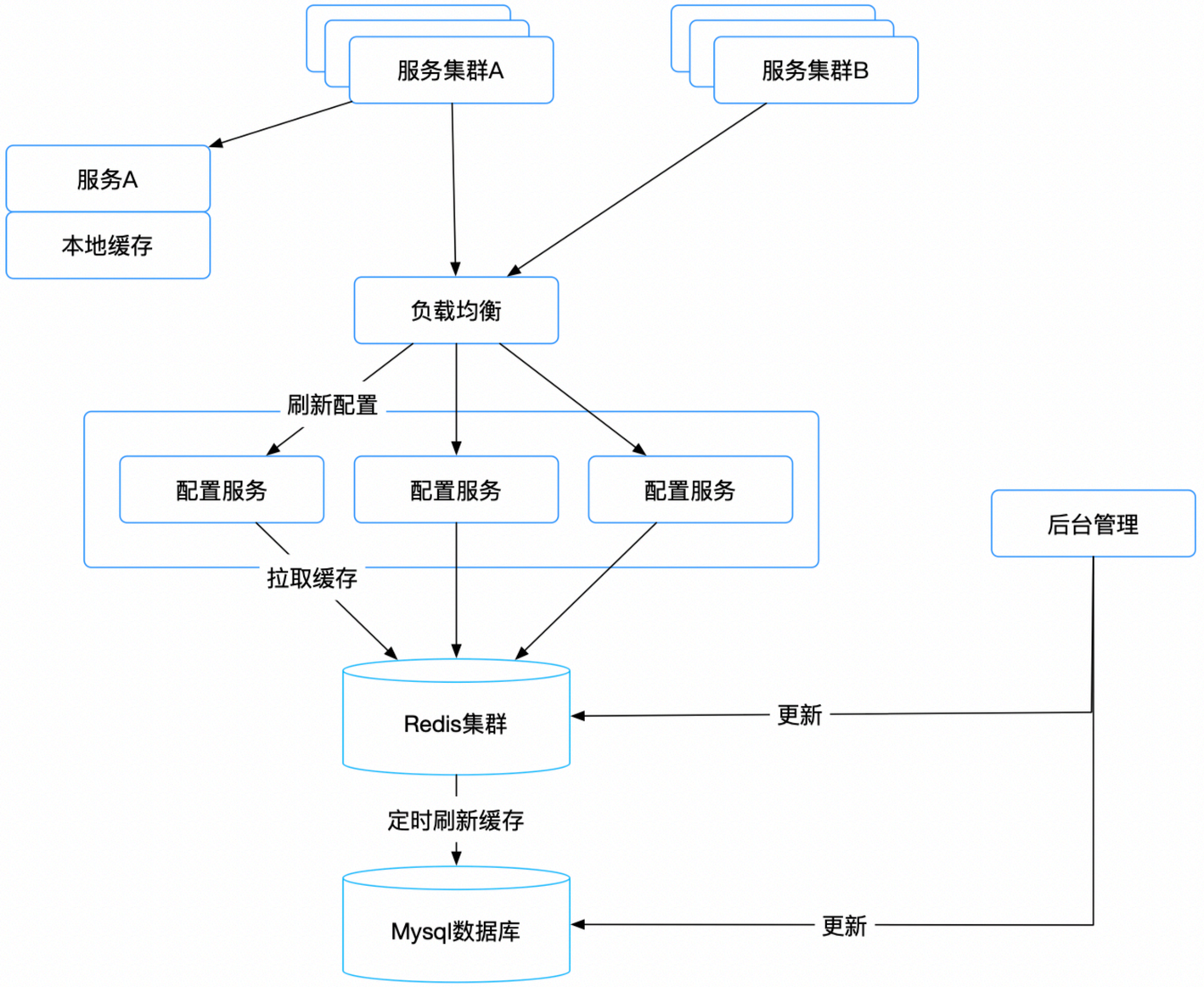
如何利用缓存来保存配置数据







- 理想中的配置中心具有哪些特点：
  - 配置的增删改查
  - 不同环境配置隔离（开发、测试、预发布、灰度/线上）
  - 高性能、高可用性
  - 请求量多、高并发
  - 读多写少


服务集群A
服务集群B
服务A
本地缓存
负载均衡
刷新配置
配置服务
配置服务
配置服务
拉取缓存
后台管理
Redis集群
更新
定时刷新缓存
Mysql数据库
更新

电商秒杀业务的架构介绍

- 秒杀与其他业务最大的区别在于：

- 秒杀与其他业务最大的区别在于：

- 秒杀的瞬间
  - 系统的并发量会非常的大
  - 并发量大的同时，网络的流量也会瞬间变大。

核心思路：

- 核心问题就在于如何在大并发的情况下能保证 DB 能扛得住压力，因为大并发的瓶颈在于DB。如果说请求直接从前端透传到 DB，显然，DB 是无法承受几十万上百万甚至上千万的并发量的。所以，我们能做的只能是减少对 DB 的访问，前端发出了100万个请求，通过我们的处理，最终只有10个会访问 DB，这样就可以了！

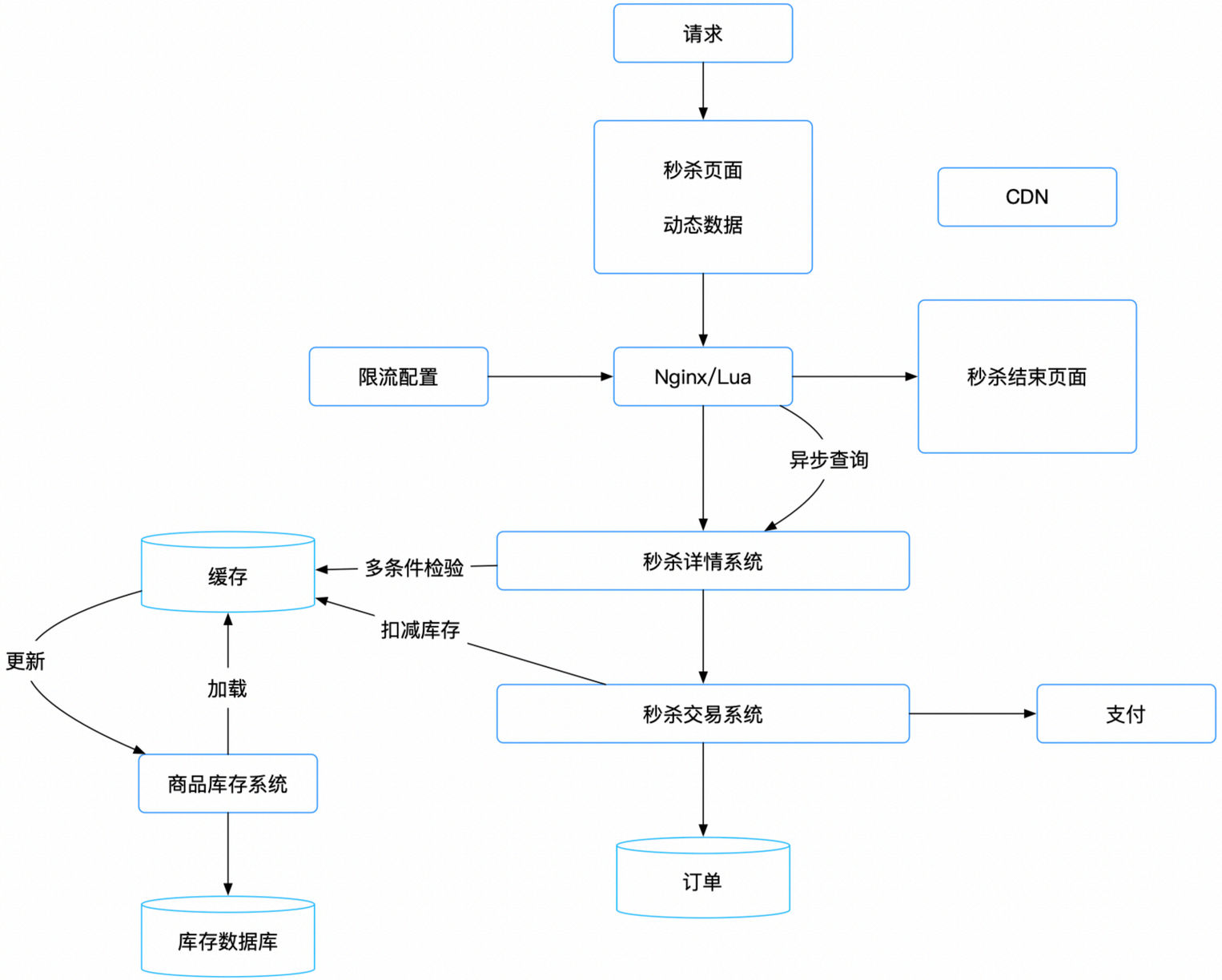
- 针对秒杀这种场景，因为秒杀商品的数量是有限的，这种做法刚好适用。

整体架构要求：

- 概括为："稳、准、快"，即对应"高可用、一致性、高性能"，其介绍分别如下：
- 高可用：保证系统的高可用和正确性，设计PlanB进行兜底。
- 一致性：保证秒杀减库存中的数据一致性。
- 高性能：涉及大量并发读写，所以需要支持高并发，从动静分离、热点发现与隔离、请求削峰与分层过滤、服务端极致优化来介绍。

常用的技术手段：

- 数据预热（预加载）
  - 将秒杀商品 提前加入到缓存系统如 ES、Redis 等，防止商品超卖和缓存穿透甚至雪崩。
- 限流
  - 通过网络代理层（如 Nginx）、SLB 负载均衡层、程序限流组件与算法（如 Guava 限流）、前端逻辑过滤等多种手段，防止大流量而造成服务拒绝或阻塞。
- 削峰
  - 通过异步通信的设计与解决方案如 RPC、MQ等具体的实现。
- 隔离
  - 通过部署隔离方案，在网络拓扑将秒杀系统独立部署，即使异常或宕机，至少不会导致主营业务系统受损或瘫痪，库表分区。


请求
秒杀页面
动态数据
CDN
限流配置
Nginx/Lua
秒杀结束页面
异步查询
缓存
多条件检验
秒杀详情系统
扣减库存
更新
加载
秒杀交易系统
支付
商品库存系统
订单
库存数据库

秒杀如何利用缓存来实现库存扣减

扣减库存有几种方式：

- 下单减库存
- 付款减库存
- 预扣库存

高并发的情况下如何扣减库存

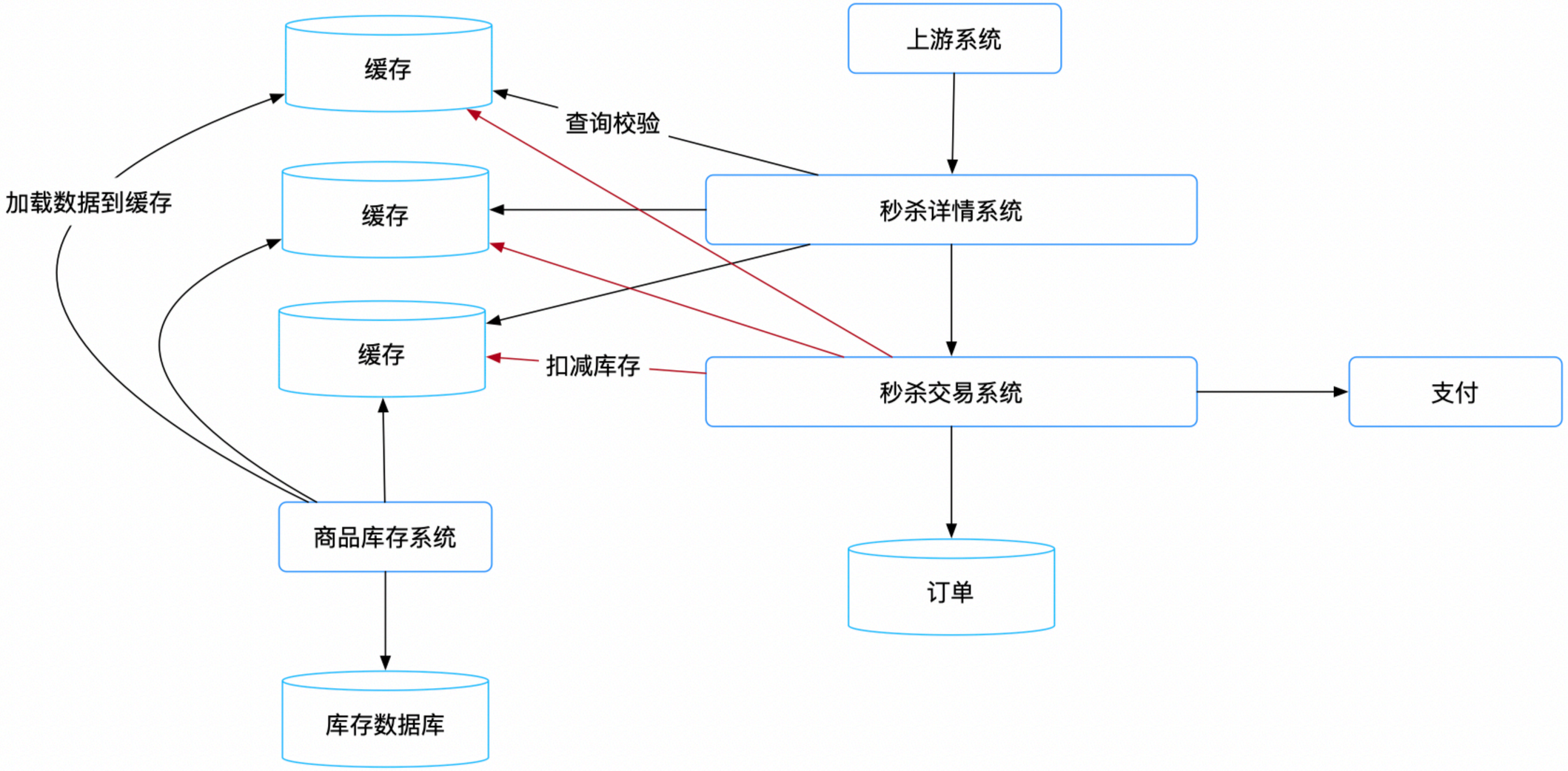
- Redis 本来就是单线程的，所以可以考虑把“检查库存-扣减库存-返回结果”的动作写成 lua 脚本去调用，一个调用就能原子化地操作库存了。

高并发的情况下如何扣减库存

- 利用SQL事务

```
beginTranse(开启事务)

try {

    //quantity为请求减掉的库存数量

    'update s_store set amount = amount – quantity where amount>=quantity and postID = 12345‘

} catch(Exception e){

    rollBack(回滚)

}

commit(提交事务)
```

库存数据放到缓存中


缓存
缓存
缓存
上游系统
查询校验
加载数据到缓存
秒杀详情系统
扣减库存
秒杀交易系统
支付
商品库存系统
订单
库存数据库

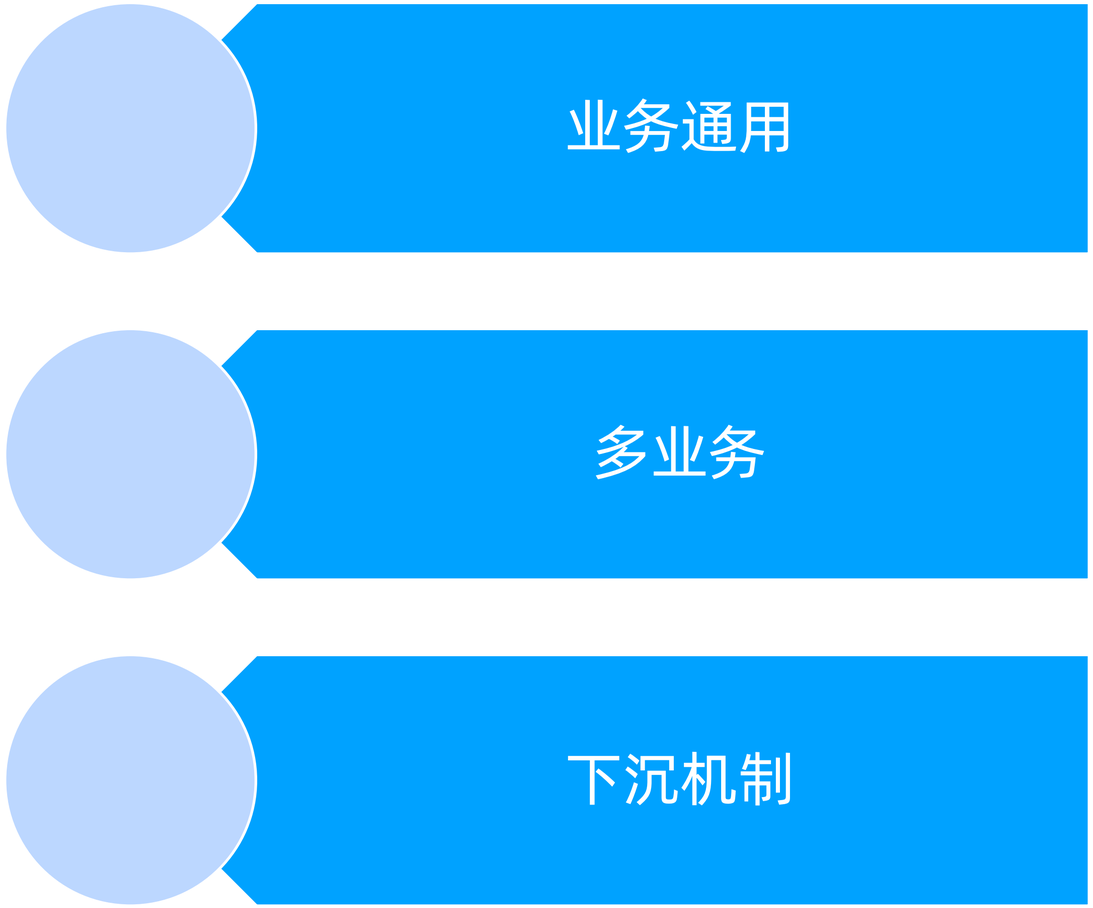
支付中台架构介绍

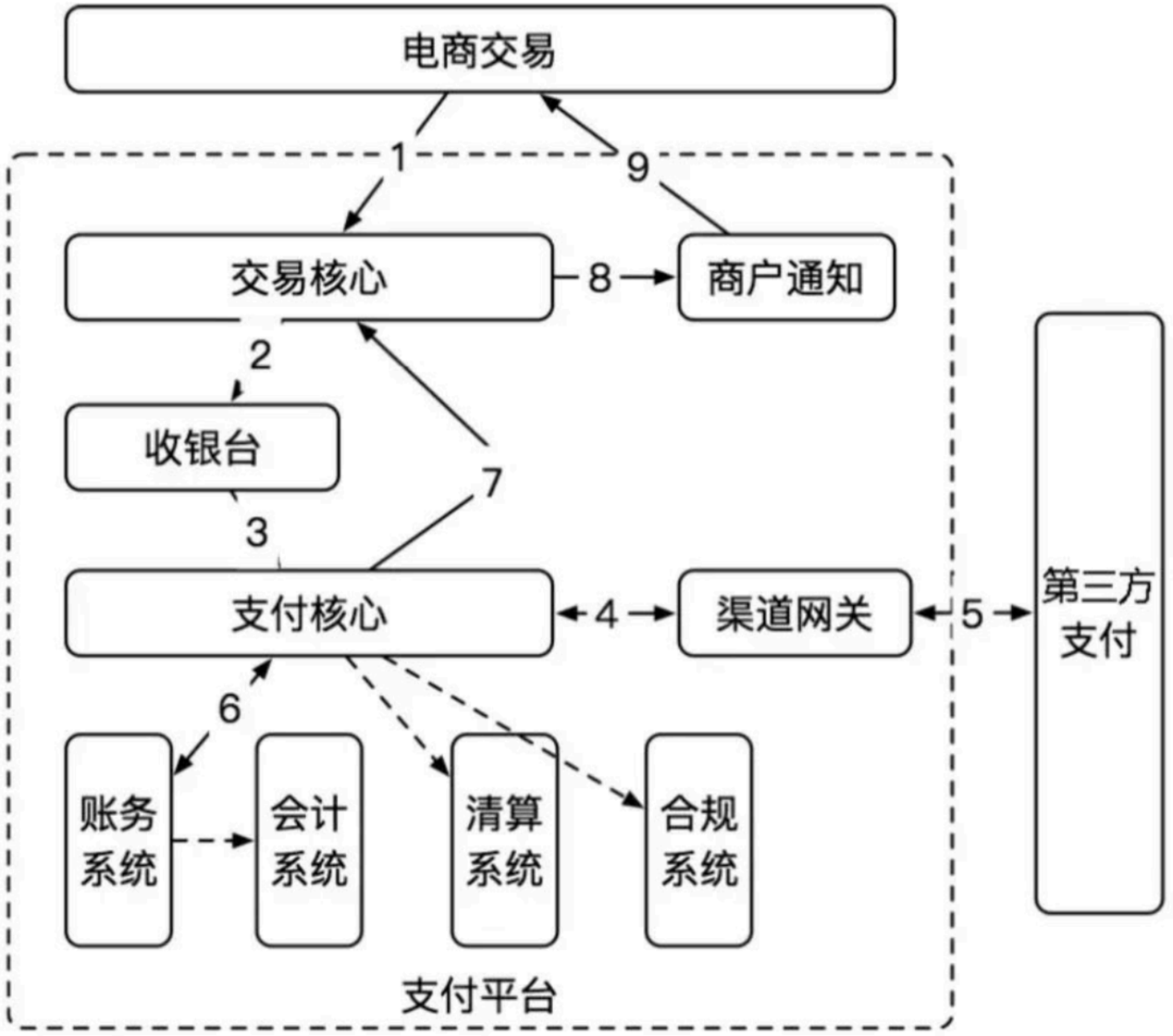
什么是中台？

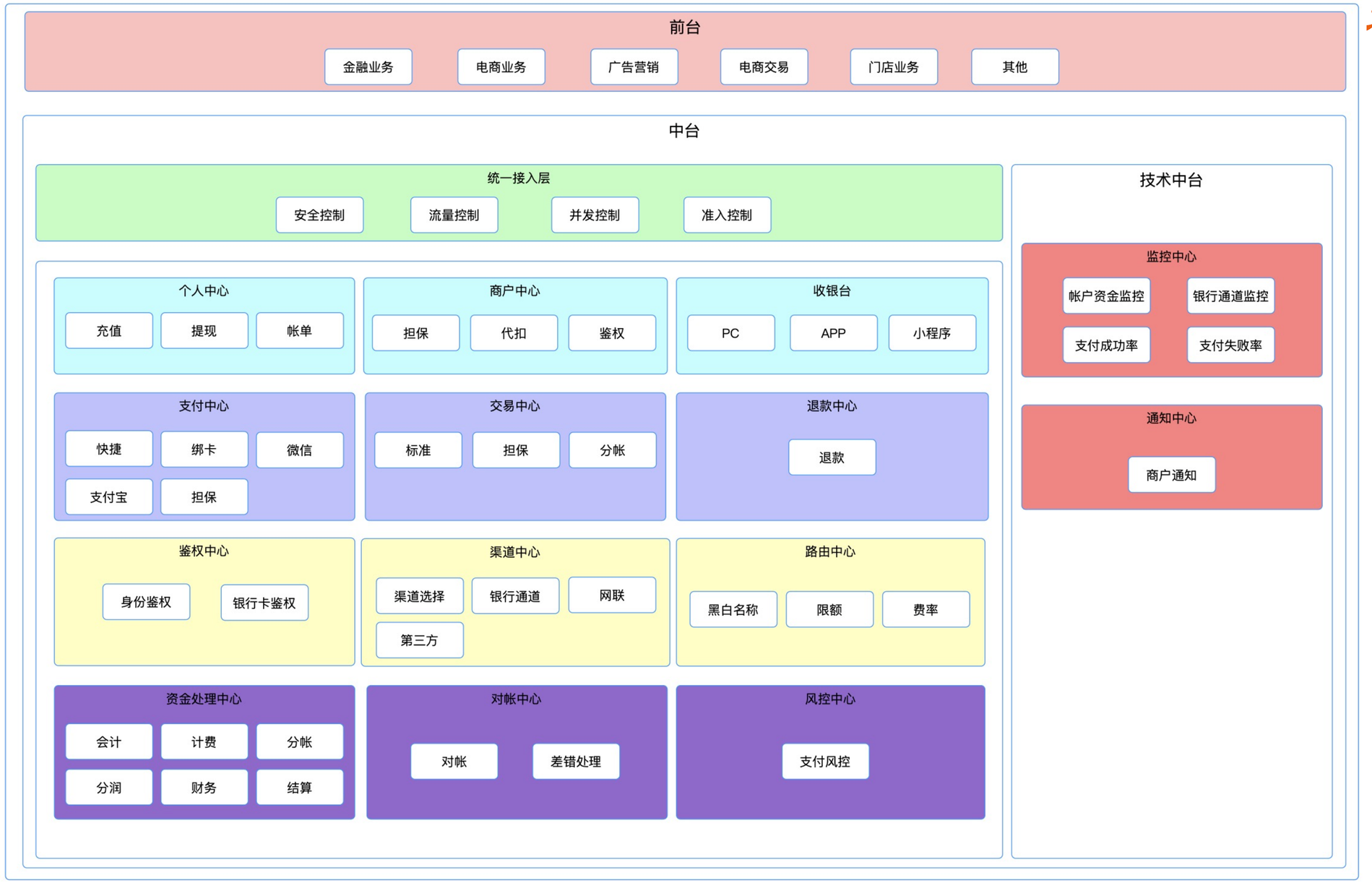
- 2019年初到现在，我们经常听到的一个词就是“中台”，那么什么是中台呢？在我看来，中台是提取各业务方的共性需求，形成共享复用能力，并提供灵活多变的定制化能力，为前台业务方提供了快速创新能力的平台。

- 中台是一种战略，涉及到组织、业务、技术、运营、数据等多方面，而其中又有两个非常重要的关键特性，那就是能力复用和灵活多变，灵活多变之中又包括服务编排和插件扩展点，而在今天的分享中，我将重点和你探讨下服务编排，进一步帮助你灵活地复用服务来定制自己的流程。

中台边界确定




电商交易
1
9
交易核心
8
商户通知
2
收银台
7
3
支付核心
4
渠道网关
5
第三方
支付
6
账务
系统
会计
系统
清算
系统
合规
系统
支付平台


前台
金融业务
电商业务
广告营销
电商交易
门店业务
其他
中台
统一接入层
安全控制
流量控制
并发控制
准入控制
个人中心
充值
提现
帐单
商户中心
担保
代扣
鉴权
收银台
PC
APP
小程序
支付中心
快捷
绑卡
微信
支付宝
担保
交易中心
标准
担保
分帐
退款中心
退款
鉴权中心
身份鉴权
银行卡鉴权
渠道中心
渠道选择
银行通道
网联
第三方
路由中心
黑白名称
限额
费率
资金处理中心
会计
计费
分帐
分润
财务
结算
对帐中心
对帐
差错处理
风控中心
支付风控
技术中台
监控中心
帐户资金监控
银行通道监控
支付成功率
支付失败率
通知中心
商户通知

基于 Redis 实现延时队列

- 在业务发展过程中，会出现一些需要延时处理的场景，比如：
  - 订单下单之后超过30分钟用户未支付，需要取消订单。
  - 订单一些评论，如果48小时用户未对商家评论，系统会自动产生一条默认评论。
  - 点我达订单下单后，超过一定时间订单未派出，需要超时取消订单等。。。

- 在业务发展过程中，会出现一些需要延时处理的场景，比如：
  - 订单下单之后超过30分钟用户未支付，需要取消订单。
  - 订单一些评论，如果48小时用户未对商家评论，系统会自动产生一条默认评论。
  - 点我达订单下单后，超过一定时间订单未派出，需要超时取消订单等。。。

- 处理这类需求，比较直接简单的方式就是定时任务轮训扫表。这种处理方式在数据量不大的场景下是完全没问题，但是当数据量大的时候高频的轮训数据库就会比较的耗资源，导致数据库的慢查或者查询超时。所以在处理这类需求时候，采用了延时队列来完成。

延时队列就是一种带有延迟功能的消息队列。

- 下面会介绍几种目前已有的延时队列：

延时队列就是一种带有延迟功能的消息队列。

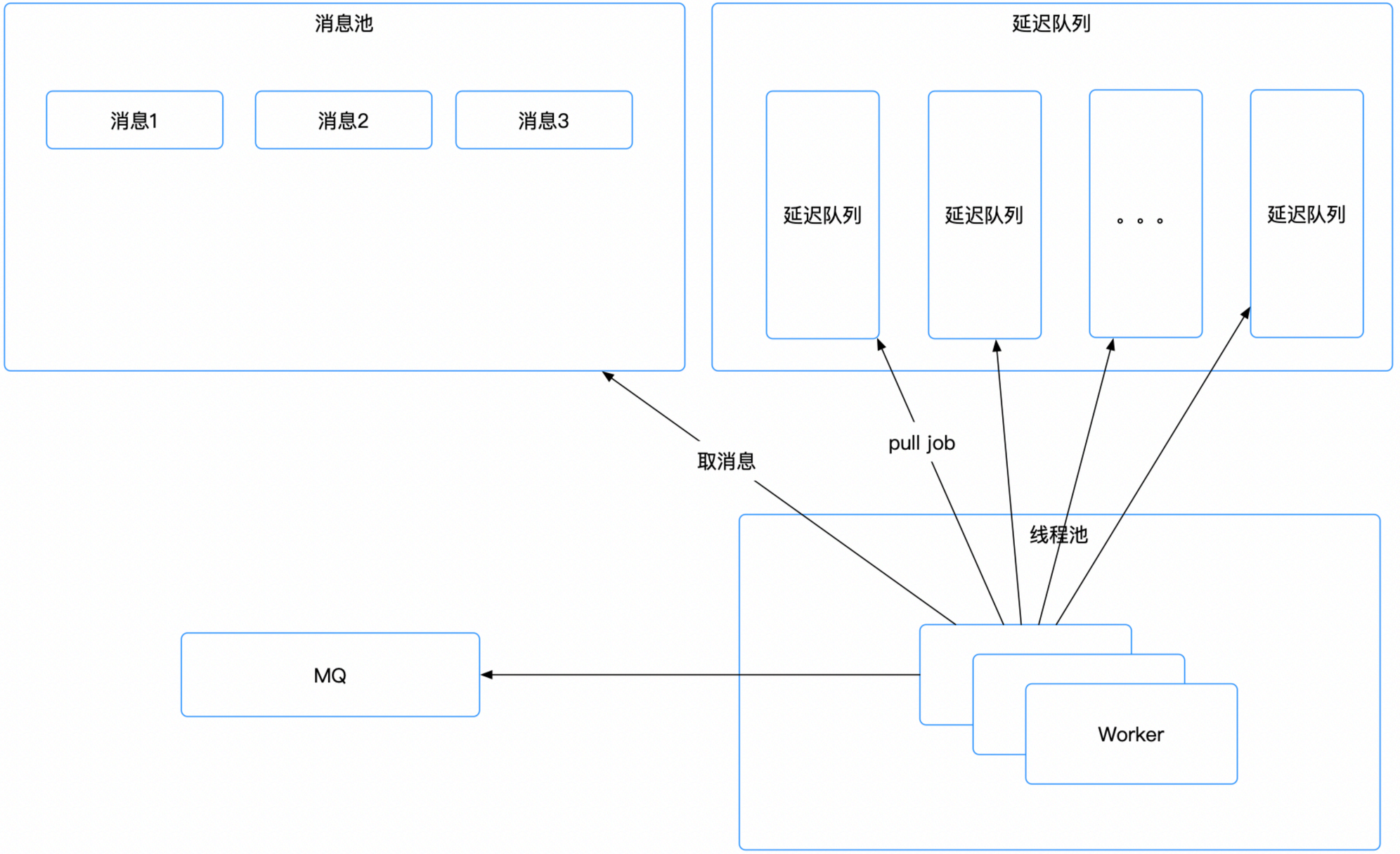
- 下面会介绍几种目前已有的延时队列：
  - Java 中 java.util.concurrent.DelayQueue
    优点：JDK 自身实现，使用方便，量小适用
    缺点：队列消息处于jvm内存，不支持分布式运行和消息持久化
  - Rocketmq 延时队列
    优点：消息持久化，分布式
    缺点：不支持任意时间精度，只支持特定 level 的延时消息
  - Rabbitmq 延时队列（TTL+DLX 实现）
    优点：消息持久化，分布式
    缺点：延时相同的消息必须扔在同一个队列

如果我们要自己实现一个延时队列需要考虑哪些点：

- 消息存储
- 过期延时消息实时获取
- 高可用性

实现思路：

- 将整个 Redis 当做消息池，以 k 形 式存储消息，key 为 id，value 为具体的消息 body；
- 使用 ZSET 做优先队列，按照 score 维持优先级（用当前时间+需要延时的时间作为 score）；
- 轮询 ZSET，拿出 score 比当前时间戳大的数据（已过期的）；
- 根据 id 拿到消息池的具体消息进行消费；
- 消费成功，删除改队列和消息；
- 消费失败，让该消息重新回到队列。


消息池
消息1
消息2
消息3
延迟队列
延迟队列
延迟队列
。。。
延迟队列
取消息
pull job
线程池
MQ
Worker

支付中台通知中心架构介绍

什么是通知中心

什么是通知中心

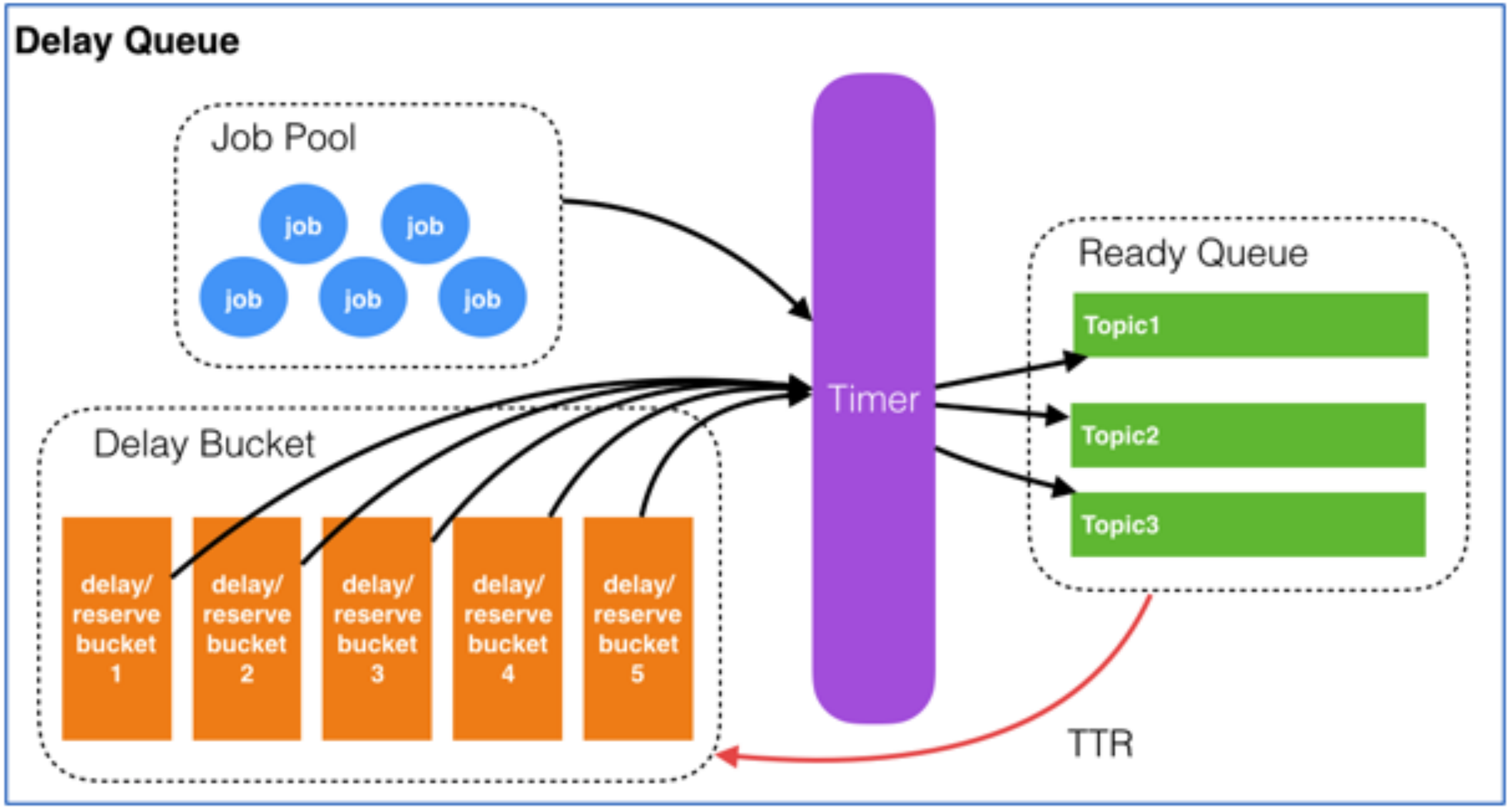
- 支付系统中，所有对外发起回调通知（包括 HTTP 回调、短信通知、邮件通知...），都经过通知中心发起。通知数据加密工作在网关层完成 。

设计思路：

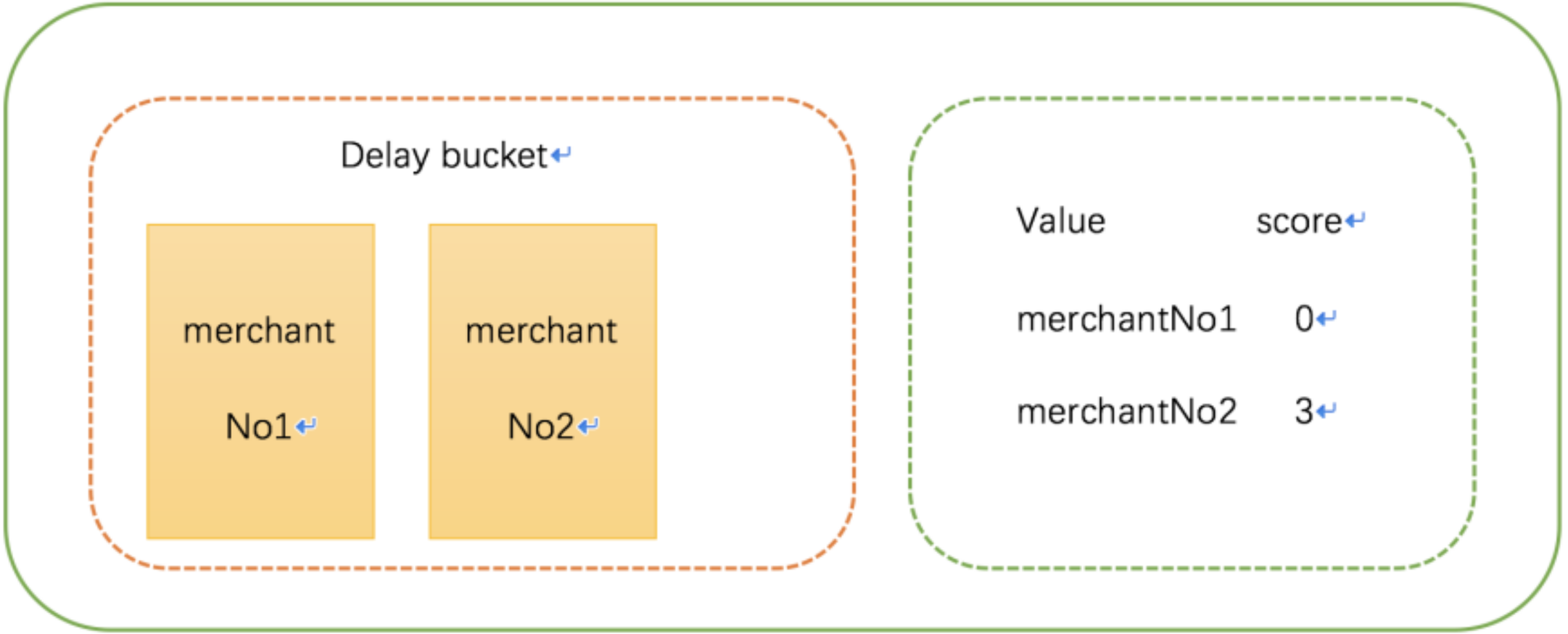
- 三个核心线程：

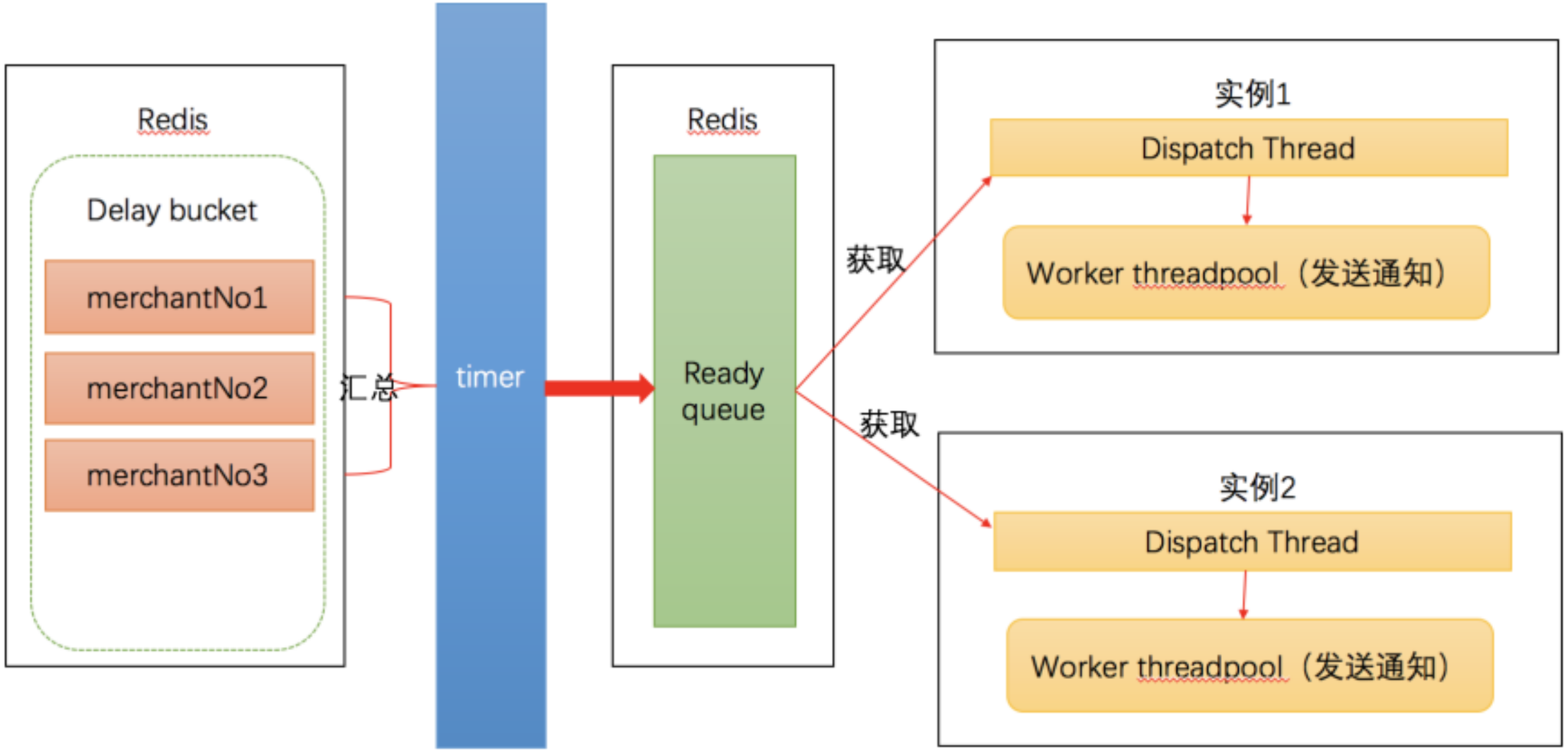
设计思路：

- 三个核心线程：
  - TimerThread： 主要实现定时器，每秒触发一次，将延时队列中可以发送的通知放入准备发送队列，需要加事物。也考虑使用外部定时任务或者 mq 消息监听实现。
  - CoreThread： 核心线程，主要从准备发送队列中取出数据然后分发到具体的工作线程进行发送通知（可以一次拿固定数量通知，从数据库查出通知信息，然后放入工作线程）。
  - WorkerThread： 工作线程，主要完成发送通知工作 。


Delay Queue
Job Pool
job
job
job
job
job
Timer
Ready Queue
Topic1
Topic2
Topic3
Delay Bucket
delay/ reserve bucket 1
delay/ reserve bucket 2
delay/ reserve bucket 3
delay/ reserve bucket 4
delay/ reserve bucket 5
TTR

- 熔断机制


Delay bucket
merchant No1
merchant No2
Value
score
merchantNo1
0
merchantNo2
3


Redis
Delay bucket
merchantNo1
merchantNo2
merchantNo3
汇总
timer
Redis
Ready queue
获取
获取
实例1
Dispatch Thread
Worker threadpool（发送通知）
实例2
Dispatch Thread
Worker threadpool（发送通知）

如何在高并发的场景下实现帐户扣减

- 一般情况下是如何扣减余额的？

```
select available amount from db
 if (available amount < amount) {
        return invest failed.
 }
 update available amount = available amount - amount;
```

线程加锁方案

- 采用 synchronize 或者 concurrent util lock 对上面这段代码加线程锁，多线程下串行执行，看着挺好，其实有几个问题：

线程加锁方案

- 采用 synchronize 或者 concurrent util lock 对上面这段代码加线程锁，多线程下串行执行，看着挺好，其实有几个问题：
  - 只对单进程下有效，多进程多 jvm 下无效！
  - 需要注意事务和加锁的顺序，数据库事务隔离级别，不然高并发下会有事务问题，这个问题不展开讨论了，单独另一篇文章里讲。
  - 一把大锁加住所有逻辑，性能可想而知的低。

数据库锁 select for update

- 利用数据库锁做分布式锁，select for update 会锁住某行数据，直到事务提交。这种方法最简单，在大部分情况下都能满足要求，所面临的问题点：

数据库锁 select for update

- 利用数据库锁做分布式锁，select for update 会锁住某行数据，直到事务提交。这种方法最简单，在大部分情况下都能满足要求，所面临的问题点：
  - 超时问题处理
  - 避免表锁
  - 避免死锁

乐观锁CAS方案

- 库存增加版本字段 version，在更新库存时比较版本号，例如：
- update amount_table set version = old_version + 1,stock = stock - 1 where version = query_version and id = xxx，
- 只有版本号没有变化，才能更新库存成功，如果版本号发生变化，则更新库存失败并进行重试。

帐户拆分

- 把一个主账户拆分成数个子账户，然后把请求分配到各个子账户上，这样单个账户的压力就小了。然后再用其他手段把子账户的数据合并成主账户数据，返回给用户。

汇总明细记帐

- 实时的交易全部是 insert 账务明细（insert 的开销很小，能够支持高并发。如果基于分布式部署，insert 的并发容量理论上可以无限大），然后定时（比如每半个小时）将之前半个小时内的账务明细sum 出一个结算总金额，一笔入账结算到指定账户。

- 这个方案的缺点就是：交易不能实时入账，其实如果控制好定时汇总入账的频度，比如分钟级，用户也是可以接受的。这种方式对收单类业务（账户加钱）非常实用，但是对支出类业务（账户减钱）类来说，有账户透支地风险。

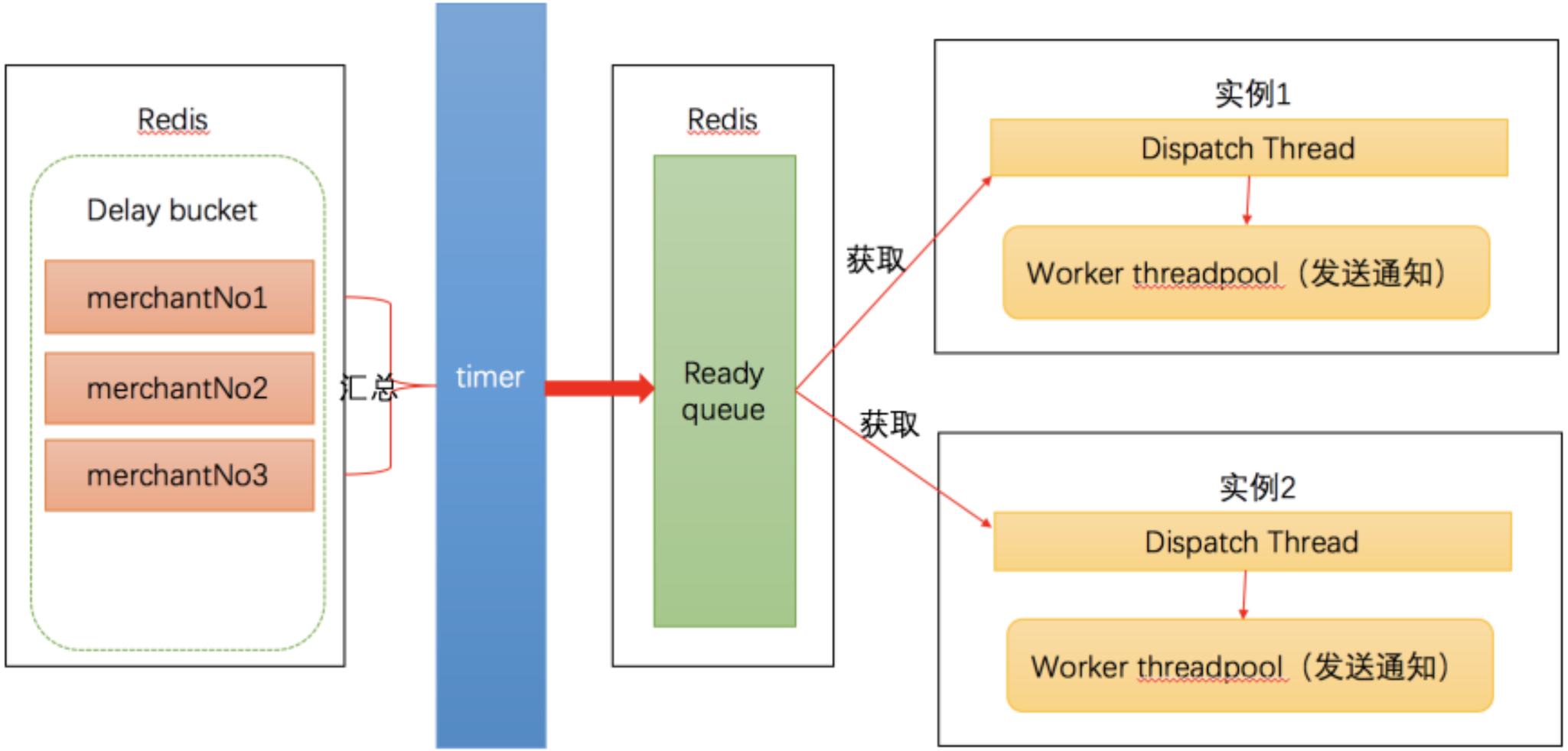
缓冲记帐

- 将实时同步的记账请求进行异步化，以达到记账实时性和系统稳定性之间平衡的记账手段，这就是”削峰填谷“。

- 方案不足：

- 热点账户在某几个高峰时间点需要缓冲记账来削峰填谷，并且能在日间填完。一旦账户的日间交易量暴增，导致日间队列根本来不及消化，整个队列越来越长，那就不存在谷可以填，这时候肯定会带来用户大量的投诉。另外这种方案对支出类业务（账户减钱）来讲，也会有账户透支地风险。

通知中心是如何利用 Redis 来发送和保存消息的


Redis
Delay bucket
merchantNo1
merchantNo2
merchantNo3
汇总
timer
Redis
Ready queue
获取
获取
实例1
Dispatch Thread
Worker threadpool（发送通知）
实例2
Dispatch Thread
Worker threadpool（发送通知）

极客时间

扫码试看/订阅

《分布式缓存高手课》视频课程