格家网络

# 交易链路的典型难题及优化方案

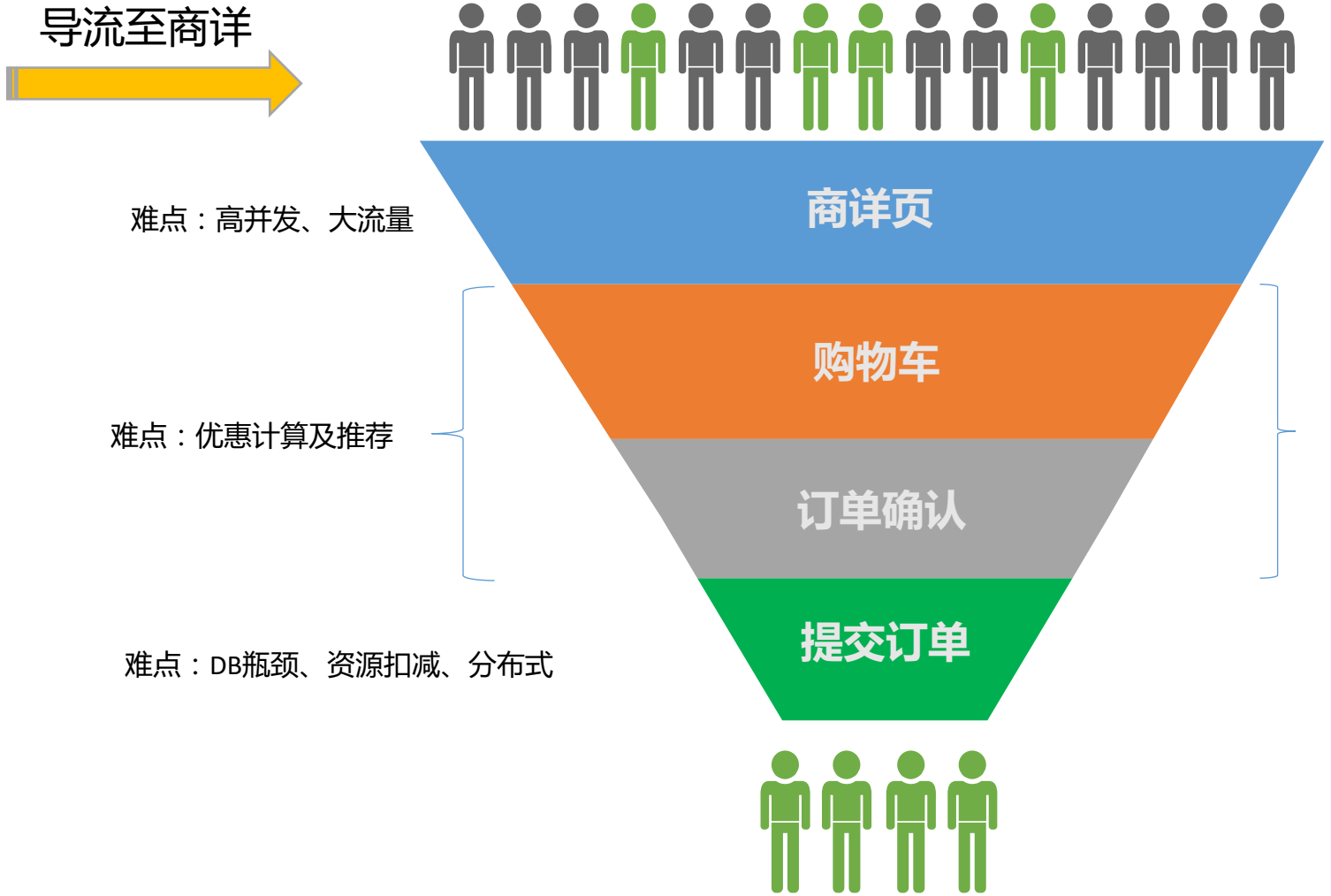
技术部-鲈鱼

# 目录

# 交易链路概览



导流至商详




难点：高并发、大流量

高效缓存

难点：优惠计算及推荐

健壮的SPI模型

难点：DB瓶颈、资源扣减、分布式

# 一、SPI设计思考

SPI是什么

全称为：Service Provider Interface

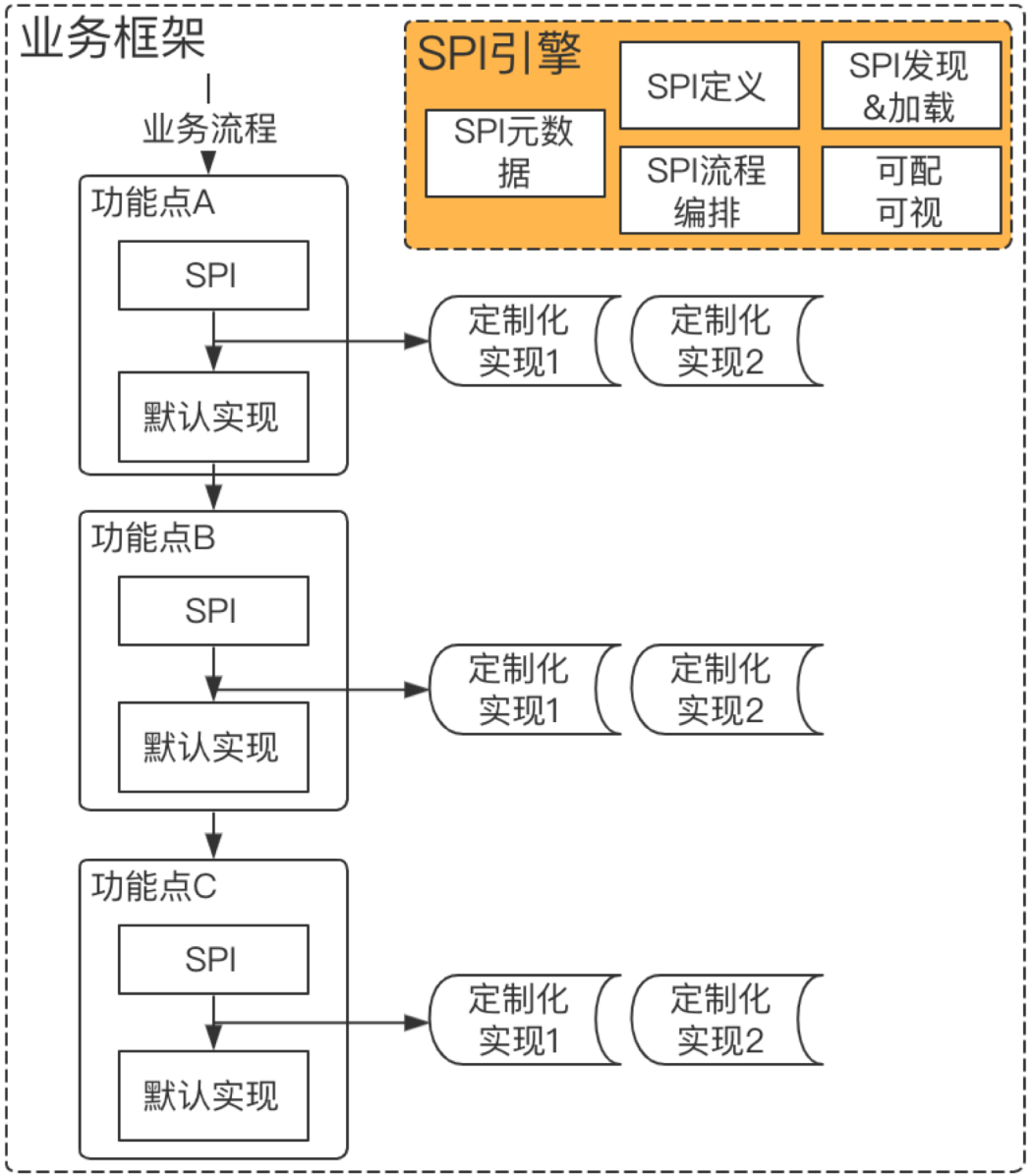
其实就是**面向开发者的功能扩展点。**

它通过模块化、可插拔的设计来实现系统的分而治之及灵活扩展。

与API的区别？

说来陌生，其实很常见

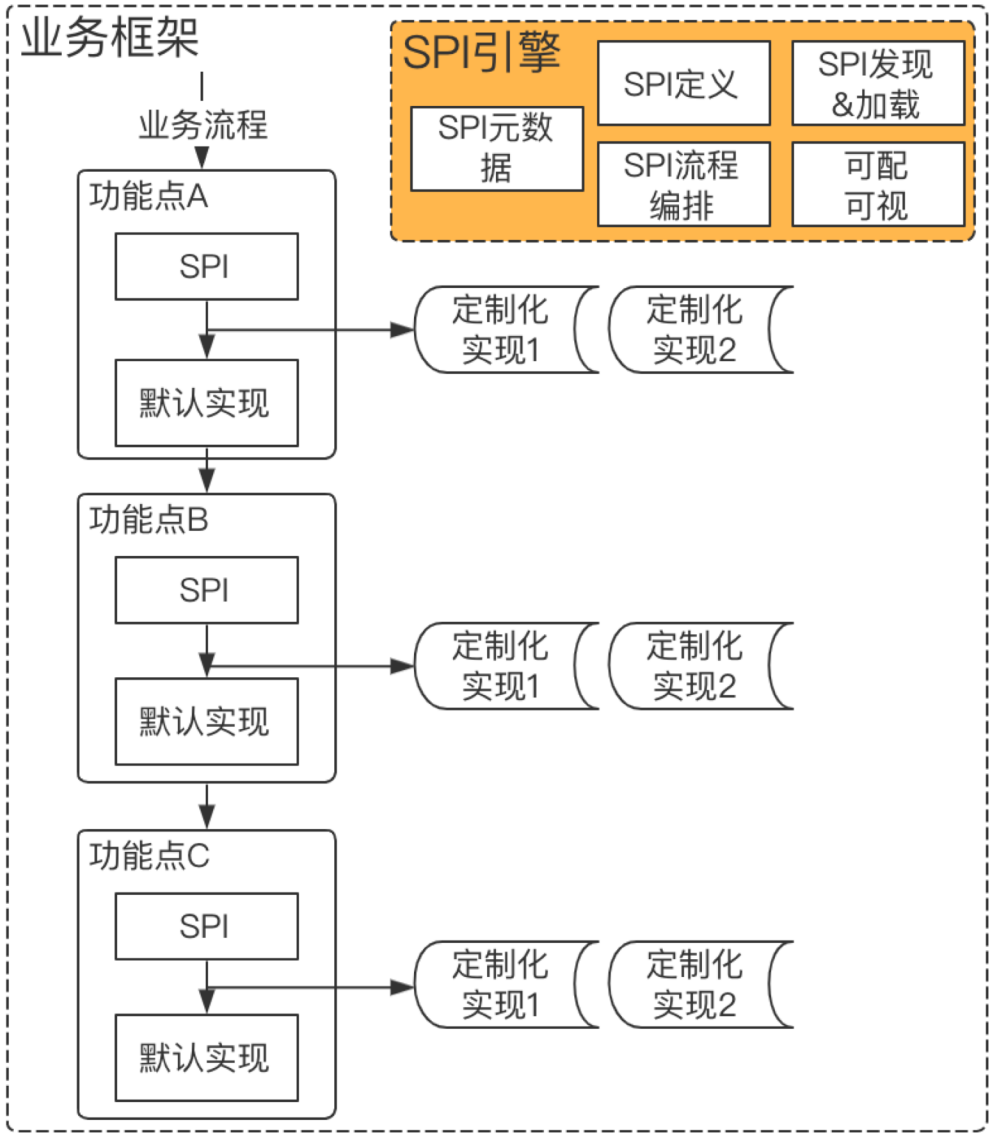
- 技术框架的定制化开发
- JDBC, Dubbo




将**一套SPI**组合进业务流程里，通过**SPI引擎**驱动，可以构建出既稳健又灵活的系统

# 一、SPI设计思考

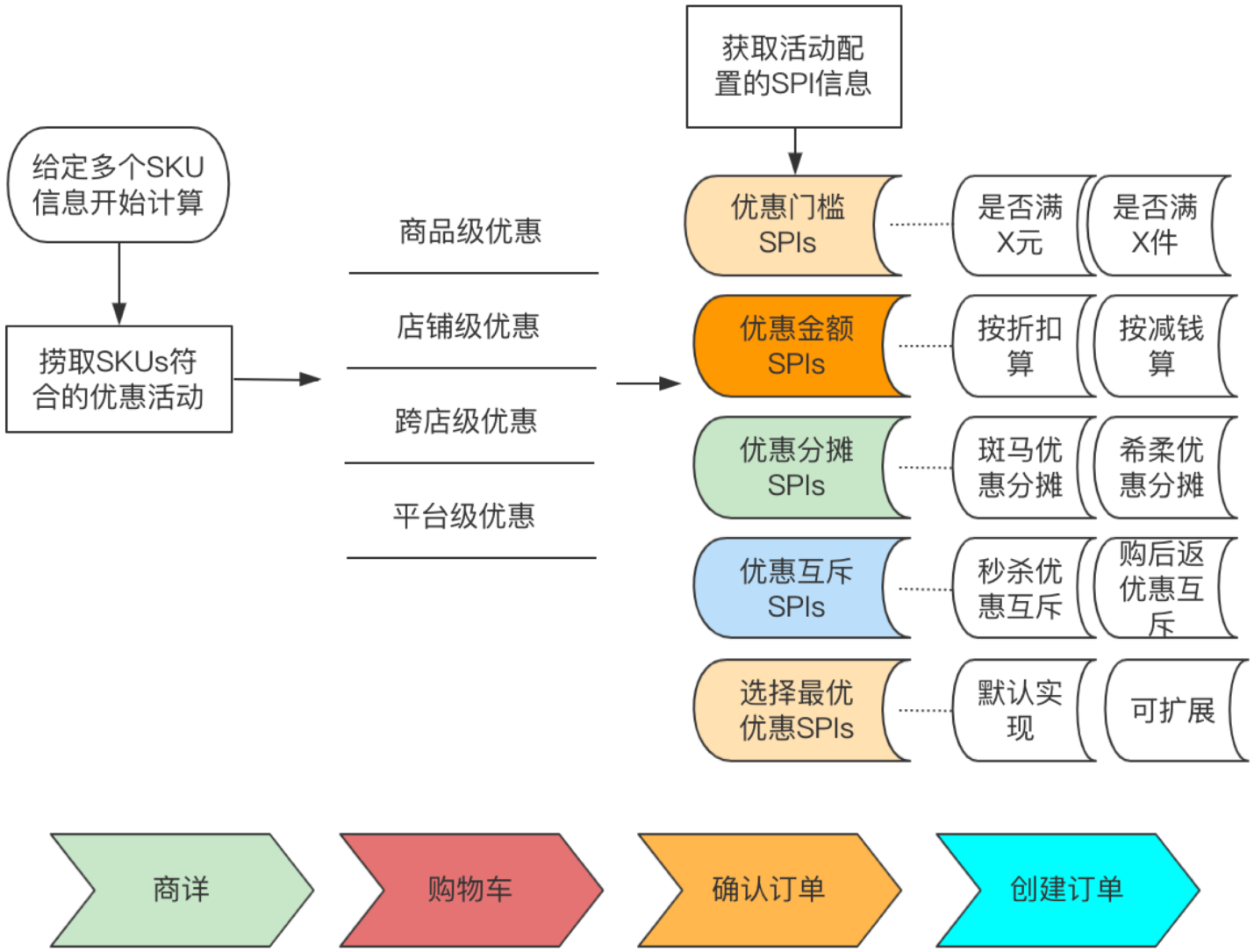
SPI的优势




- 设计模式的6大原则几乎都符合
- 业务流程抽象且稳定，易于维度，又可灵活扩展SPI，且扩展特定SPI对系统整体影响可控，易于测试
- 系统开发者和业务开发者在SPI上平等化。默认实现都可以被替换为定制化实现

# 一、SPI设计思考

SPI模式在格家系统的应用：优惠计算流程




- 商详、购物车、确认订单、提单一套SPI组件，高度复用
- 功能点固定，扩展简单

某优惠券活动配置SPI示例←

```
{
    "itemTypes": "9",
    "isNewUserCondition": false,
    "itemTags": "35",
    "isItemOverCondition": false,
    "isZhFeeReduceAction": false,
    "isShopTagCondition": false,
    "terminals": 0,
    "isUpperLimit": false,
    "isAreaCondition": false,
    "isItemTypeCondition": true,
    "isPostageReduceAction": false,
    "isDiscountAction": false,
    "isTopNDiscountAction": false,
    "useOrigPriceCondition": true,
    "isItemTagCondition": true,
    "isPriceOverCondition": true,
    "amountAt": 39900,
    "isPriceReduceAction": true,
    "reducedPrice": 3000,
    "isTerminalCondition": true,
    "originalTag": 1,
    "isAppCondition": false
}
```

# 一、SPI设计思考

SPI模式在格家系统的应用：下单流程

格家网络

**数据:**

1、12+下单流程

2、20+业务场景

3、100+业务规则

| 下单流程/功能点 | 地址查询 | 风控校验 | 价格计算 | 运费险 | 订单超时 | 是否物流单 |
|---|---|---|---|---|---|---|
| 商品交易（类下单流程） | 需要收货地址 | 支持风控校验 | 促销计价 | 支持运费险 | 默认30分钟超时 | 写物流单 |
| 虚拟充值（类下单流程） | 不需要收货地址 | 支持风控校验 | 原始计价 | 不支持运费险 | 默认30分钟超时 | 不需要写物流单 |
| 整单换货（类下单流程） | 需要外部地址 | 不支持风控校验 | 原始计价 | 不支持运费险 | 没有超时 | 写物流单 |
| 权益下单（类下单流程） | 不需要收货地址<br>(特殊：机票需要物流地址) | 支持风控校验 | 1、火车票自定义促销计价<br>2、打车接送模式原始计价 | 不支持运费险 | 1、机票10分钟。<br>2、保险2小时。<br>3、打车永不超时。 | 不需要写物流单<br>(特殊：机票需要物流地址) |
| 虚拟商品（类下单流程） | 不需要收货地址 | 支持风控校验 | 原始计价 | 不支持运费险 | 默认30分钟超时 | 不需要写物流单 |
| 企业支付（类下单流程） | 不需要收货地址 | 支持风控校验 | 原始计价 | 不支持运费险 | 没有超时 | 不需要写物流单 |
| 电子卡券（类下单流程） | 需要收货地址 | 支持风控校验 | 促销计价 | 不支持运费险 | 默认30分钟超时 | 写物流单 |
| | | | | | | |
| | 1、不同下单流程：要不要某个功能点（如地址、运费险、风控、运费等）<br>2、不同下单流程：需要某个功能点，但是有独有的规则逻辑（如超时时间，价格计算、地址等） | | | | | |

## 1、定义SPI

```
@SpiFunctionPoint(spiBeanName = CalcPrice.
public interface CalcPrice extends IBaseSp
    PayOrderResSDTO> {

    String BEAN = "calcPrice";

}
```

## 2、定制SPI实现

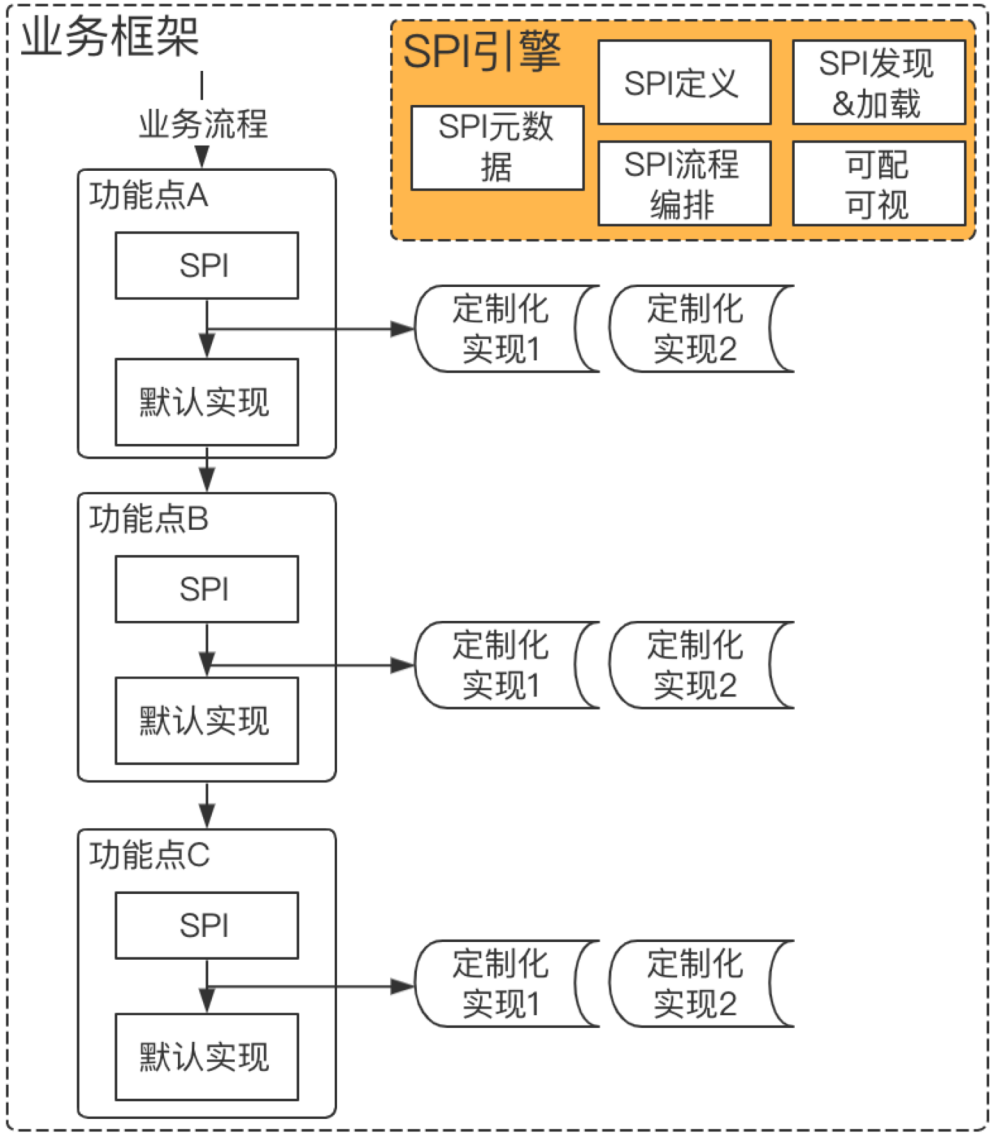

## 3、SPI引擎驱动执行

```
public interface CreateOrderSpiModule extends Module {

    /**
     * SPI方式创建订单
     *
     * @param orderSDTO the {@link OrderSDTO}
     */
    void createOrderBySpi(OrderSDTO orderSDTO);
}
```

# 一、SPI设计思考

如何设计一套SPI



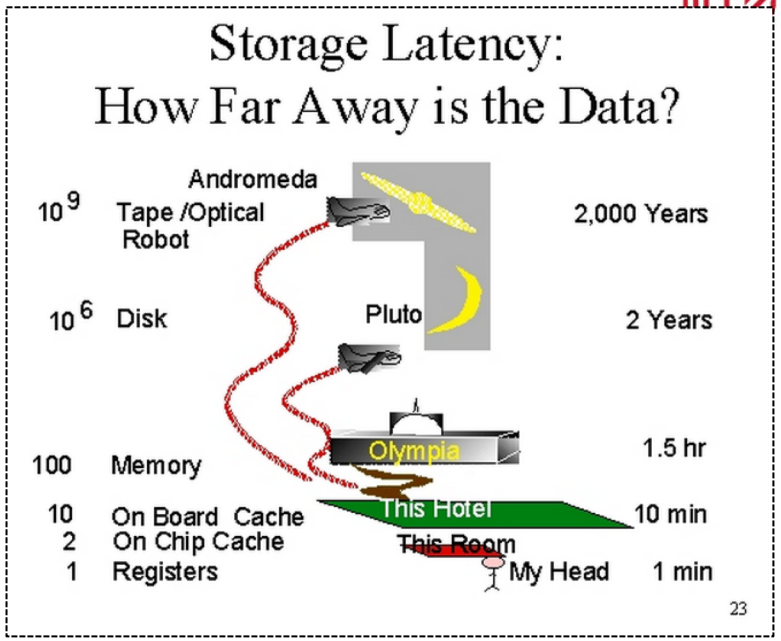

- 提炼抽象业务流程和功能点，抽象后的流程和功能点是通用、稳定的，不能随意变更
- 先大再小，或者先小再大，以扩展点的方式开发、开放新功能点
- 每个扩展点提供默认实现，但允许开发者自主更换
- 多个扩展点需要一个执行引擎统一调度管理，引擎

  必有功能：SPI的发现、加载和执行功能，

  可选功能：非代码编排、可视化

# 二、高效缓存 为什么要使用缓存？

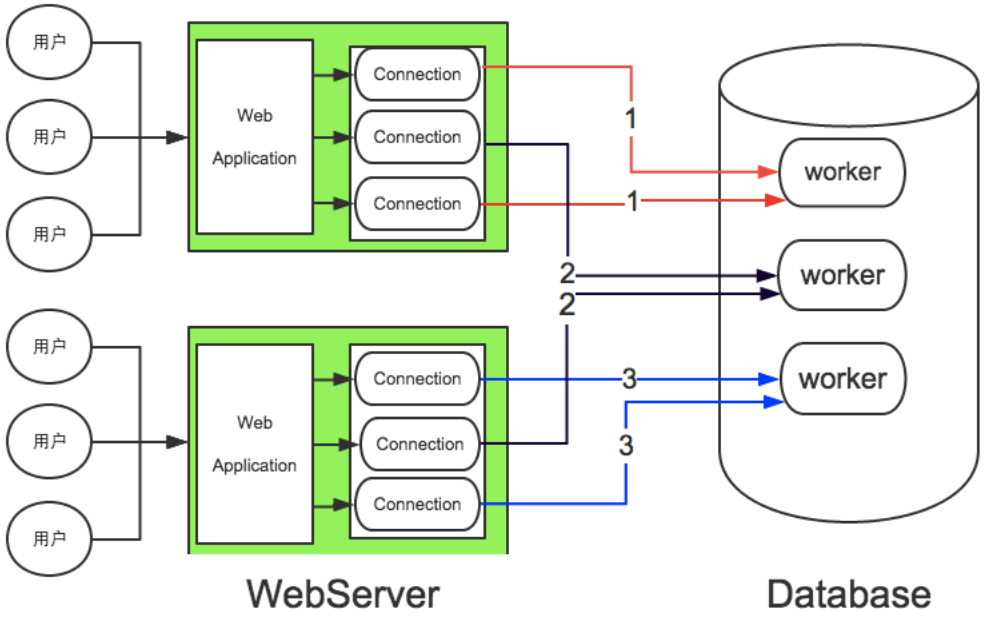
作用1：保护数据库　　作用2：降低延时、提高吞吐




数据库是最宝贵的资源，
十个故障，五个+是慢查询

为什么一个慢查询会导致故障？

因为：
TCP连接、
服务器
线程/进程数量
都是有限的




缓存使用策略

1. 手动、定时预热/预加载
2. 热点数据准实时探测及预加载
3. 做好系统限流

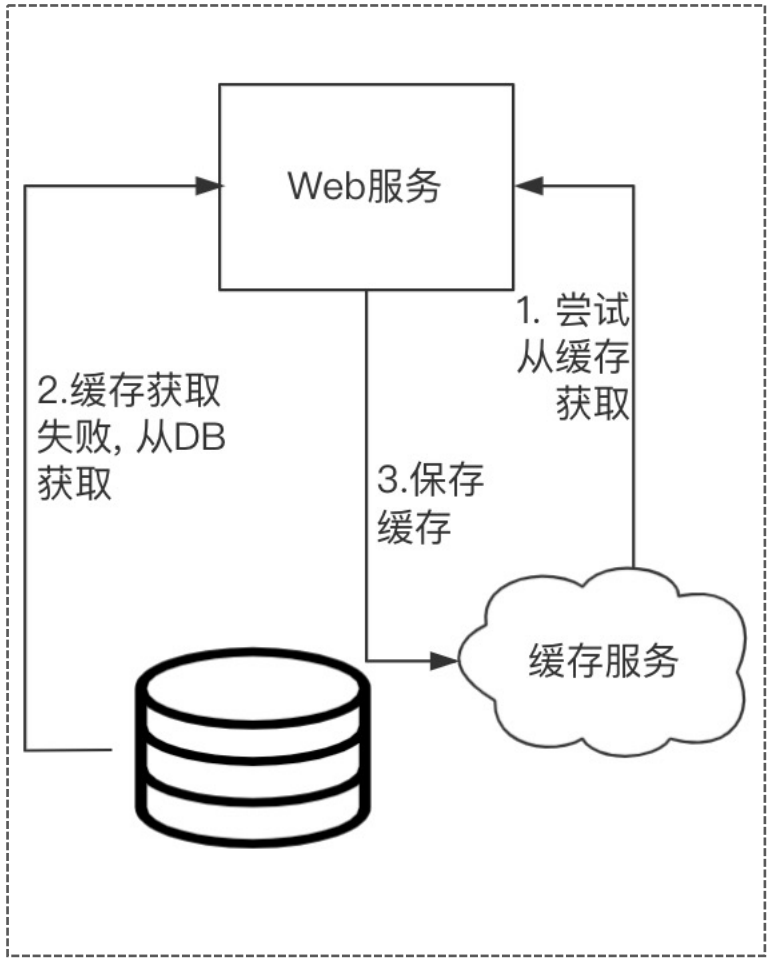
# 二、高效缓存 服务端常用缓存策略

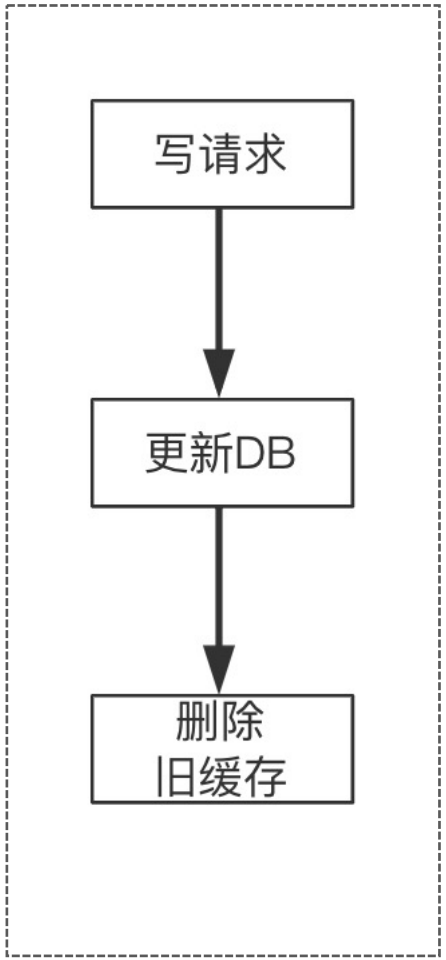


读请求

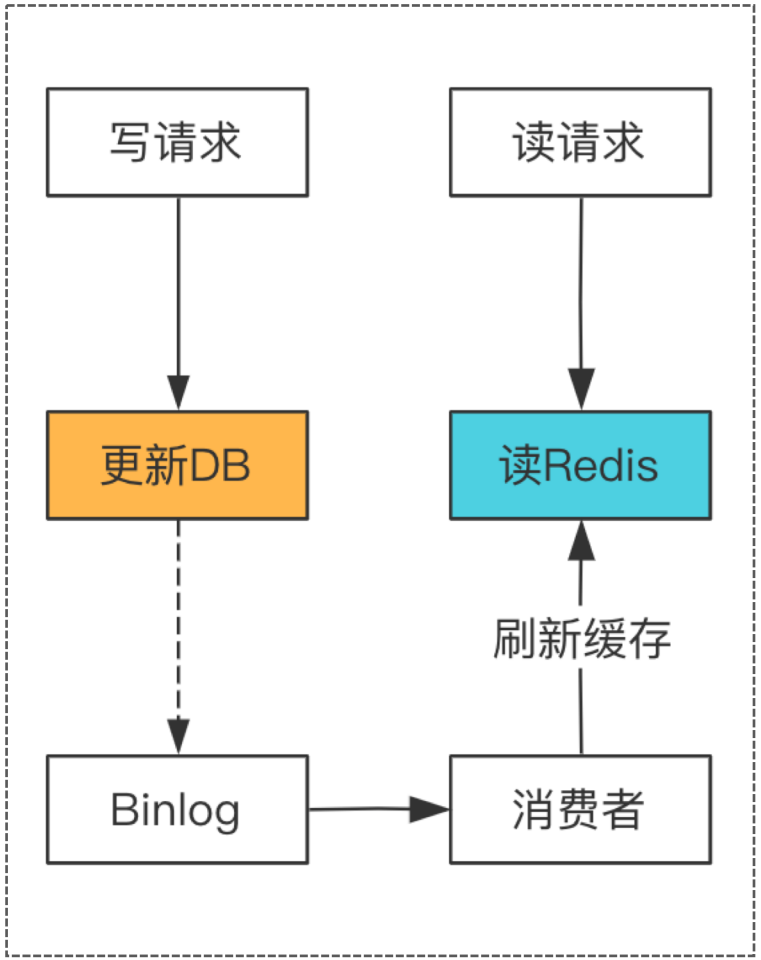
Cache-Aside Pattern

写请求

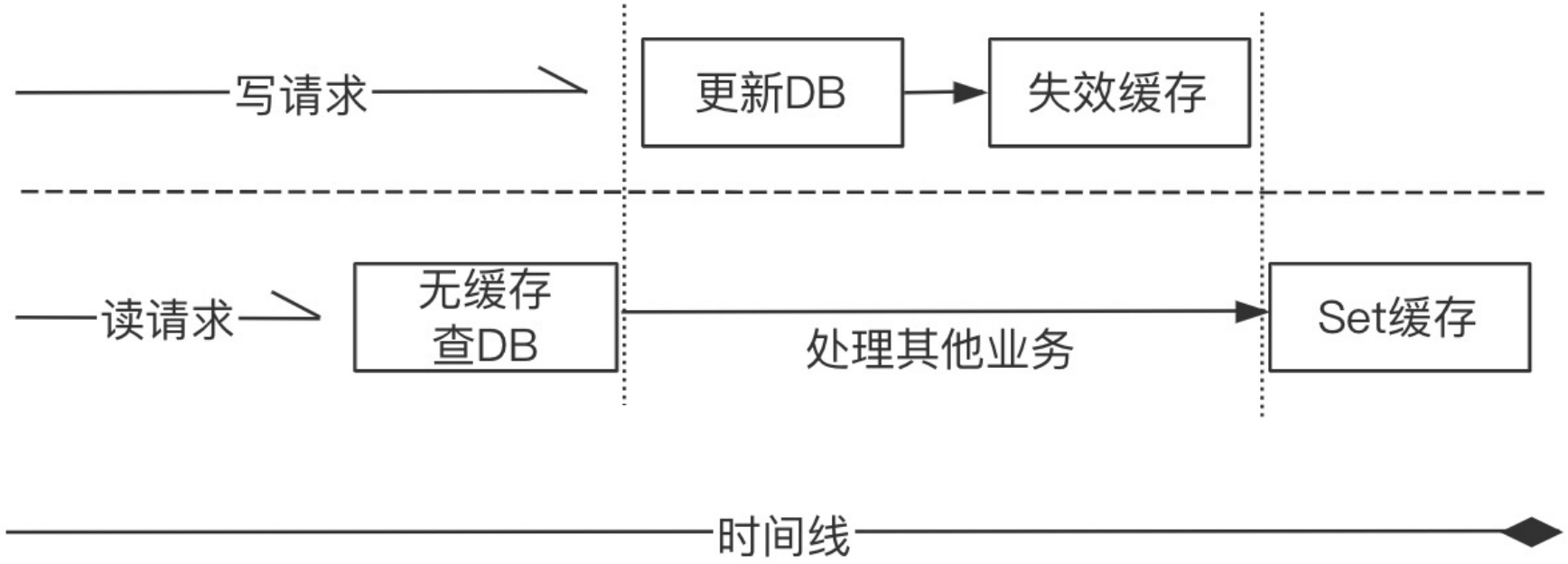



Binlog异步刷新



# 二、高效缓存 Cache-Aside Pattern脏缓存如何解决




## 方案1：缓存延时删除 + 失败重试

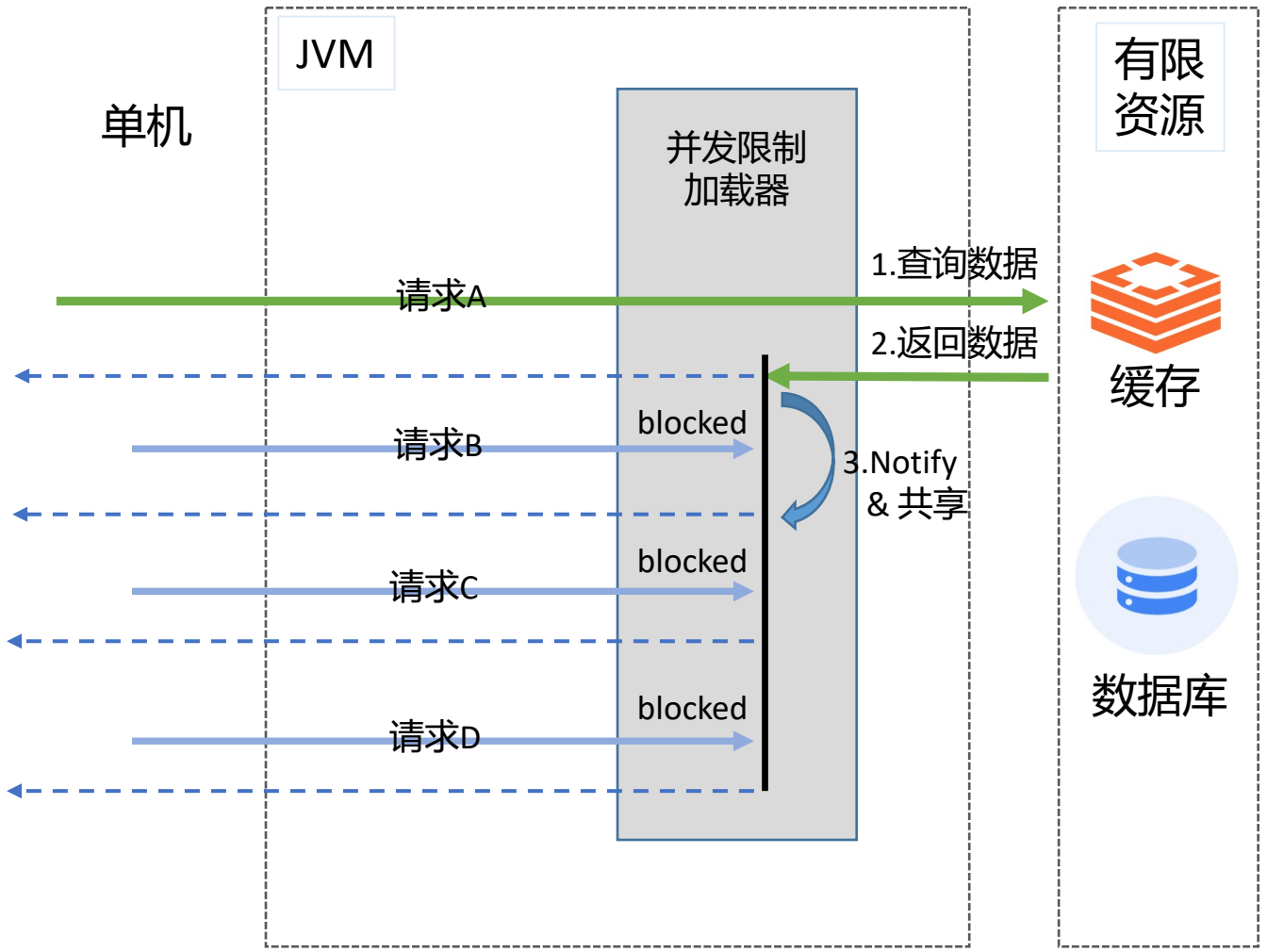
优缺点：实现简单，但延时到期前可能有不一致，缓存命中率降低，穿透请求增多

## 方案2：消费Binlog异步刷新

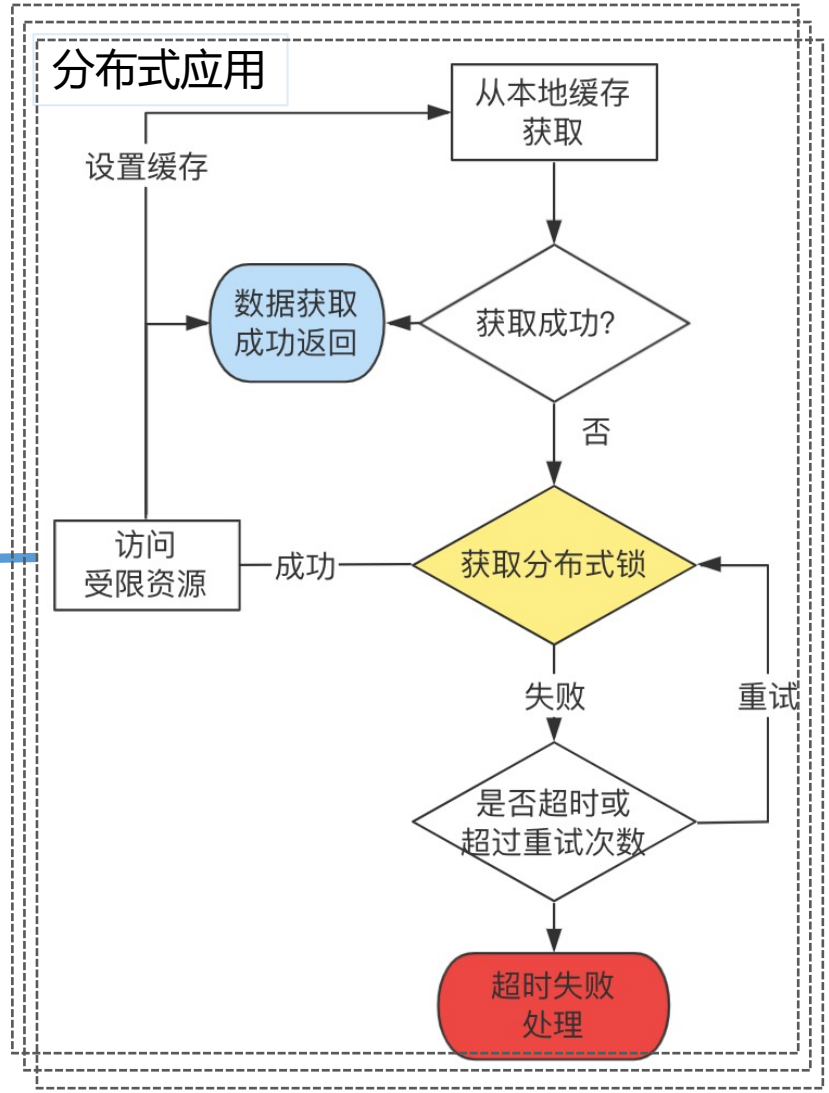
优缺点：对代码无侵入，但依赖和实现复杂

# 二、高效缓存

优化：缓存/DB防击穿



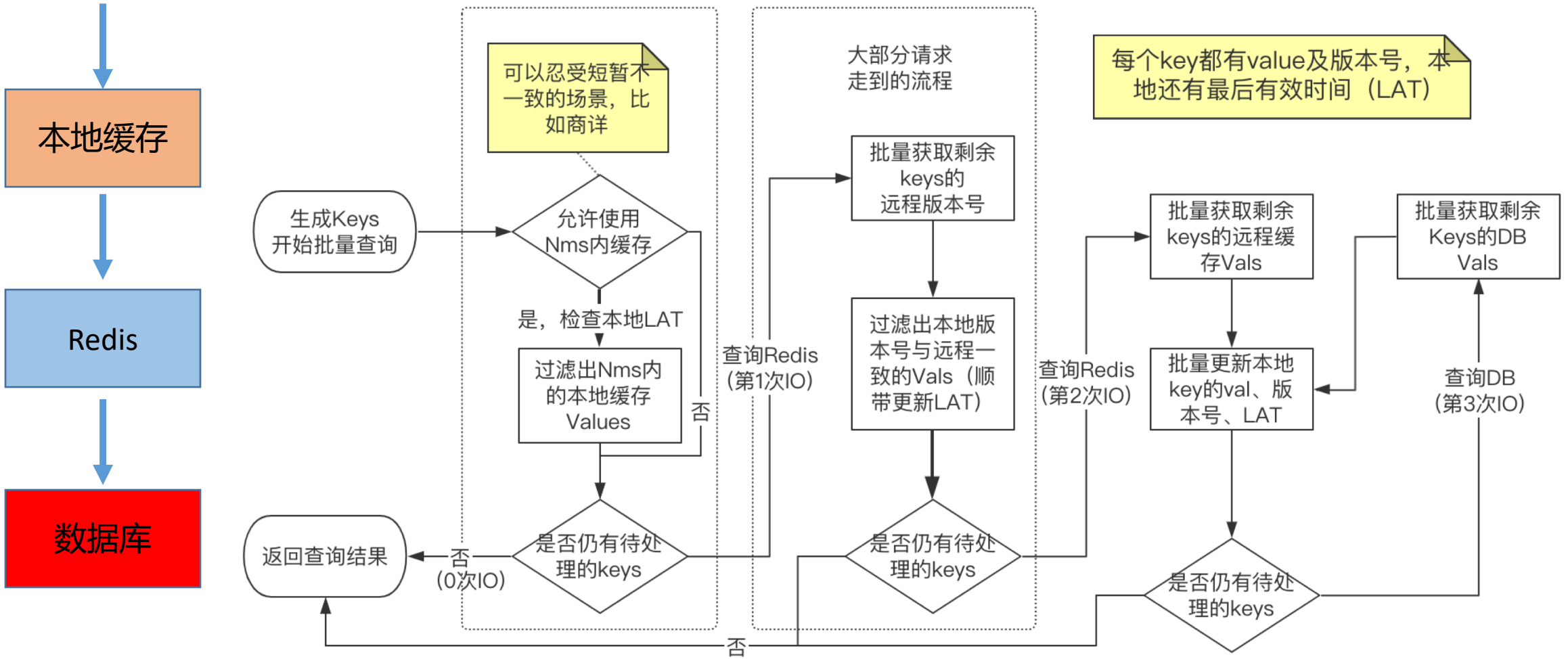

仅依赖Java自带包，访问结果可共享，不可限制多机访问




依赖分布式锁，访问结果不能共享，可限制多机访问

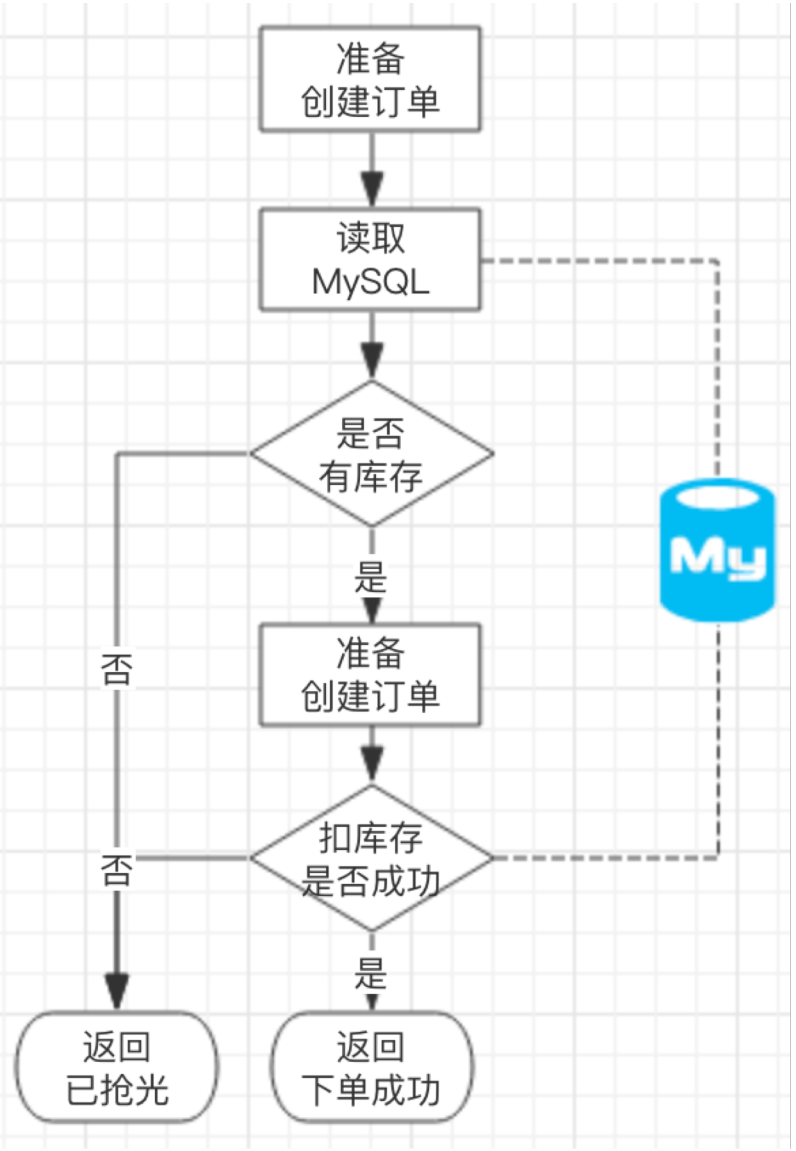
# 二、高效缓存

本地缓存与Redis的结合使用



# 三、资源扣减

## 库存扣减面对的问题




假如有10万个请求到达，

但商品库存只有1千个。

**试问下面哪一个分支的流量是最高的？**

上面的案例中，99%的流量都会

走向“已抢光”

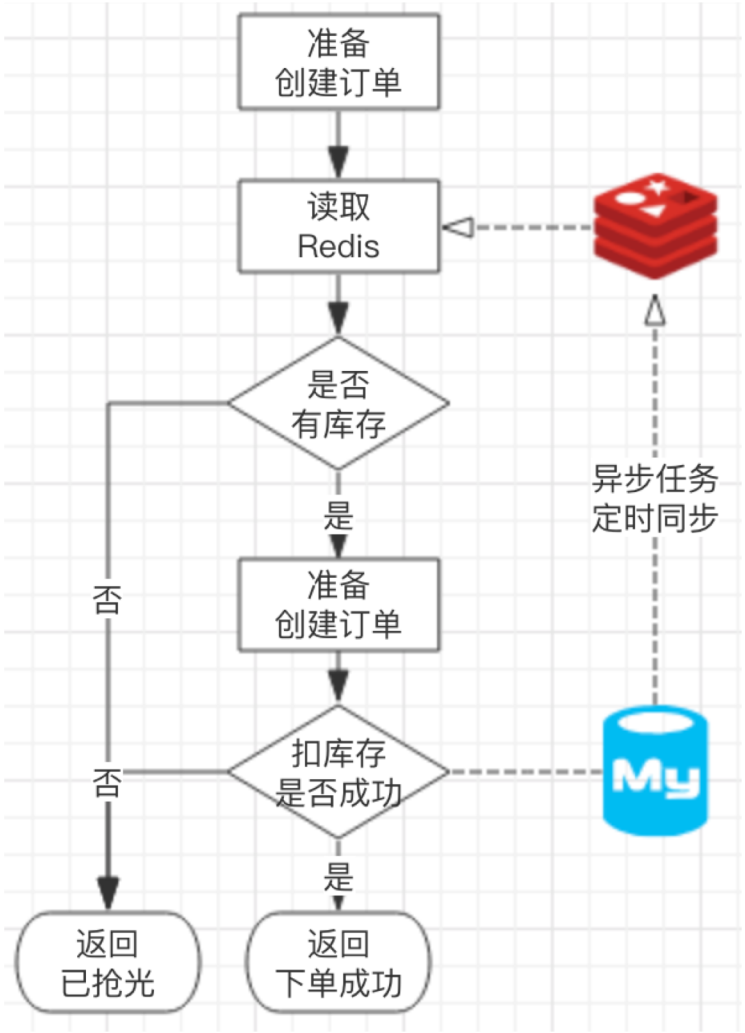
也就是说，99%的查询都是MySQL读操作；

但MySQL除了要应对高并发的读操作，还要

保证1千个正常流量商品可以扣库存成功

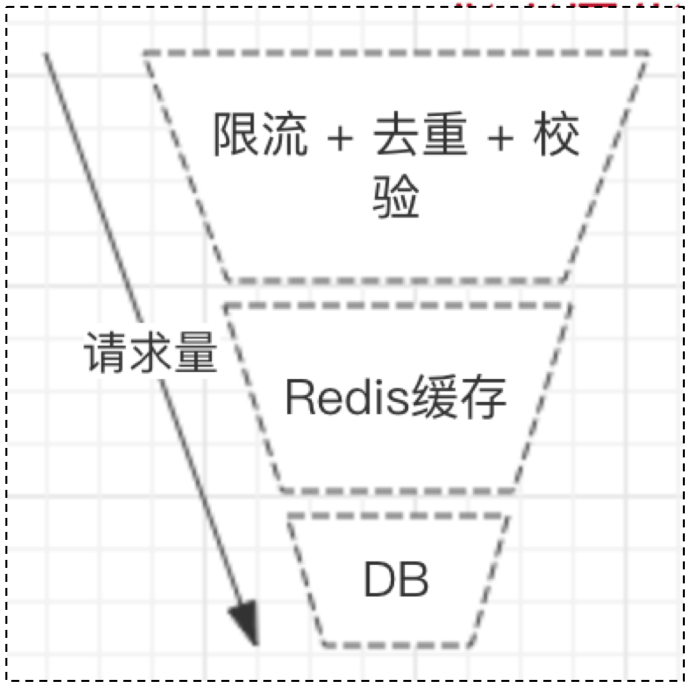
# 三、资源扣减

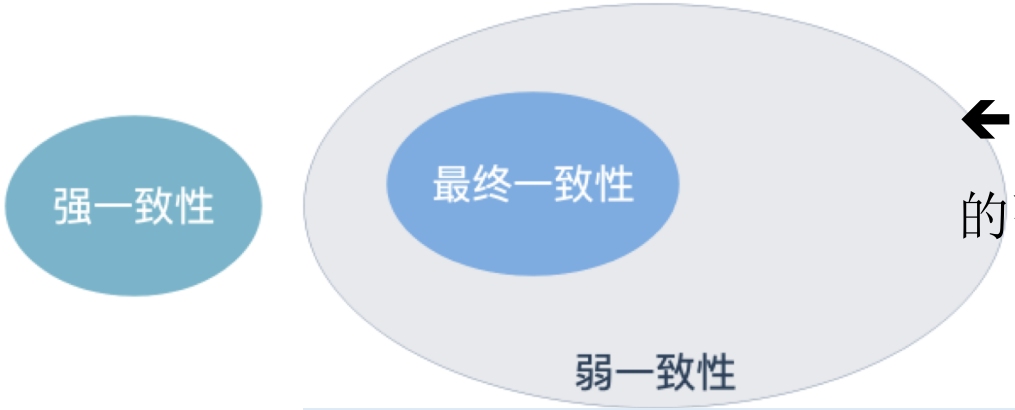
## 库存扣减面对的问题




← 假如有10万个请求到达，但商品库存只有1千个。在一定时间内读库存的压力，都转移给Redis

基于漏斗模型，将落到DB的请求量降至可控范围内的最低点➔



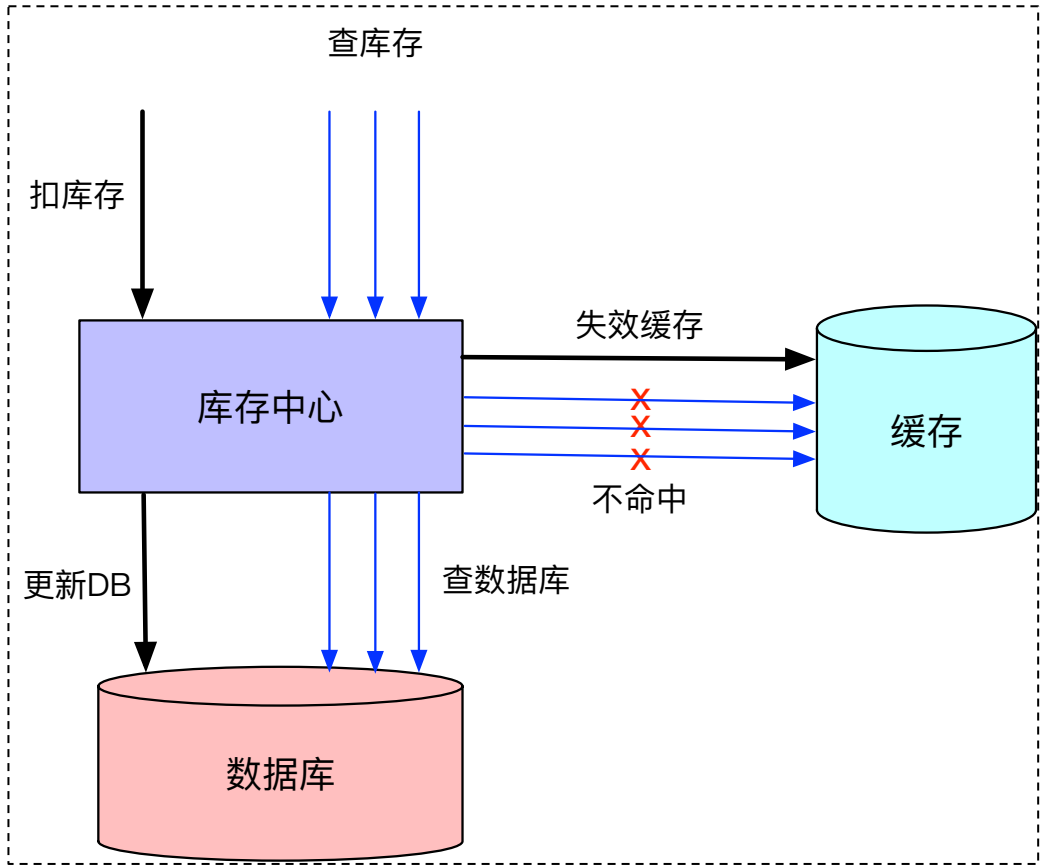



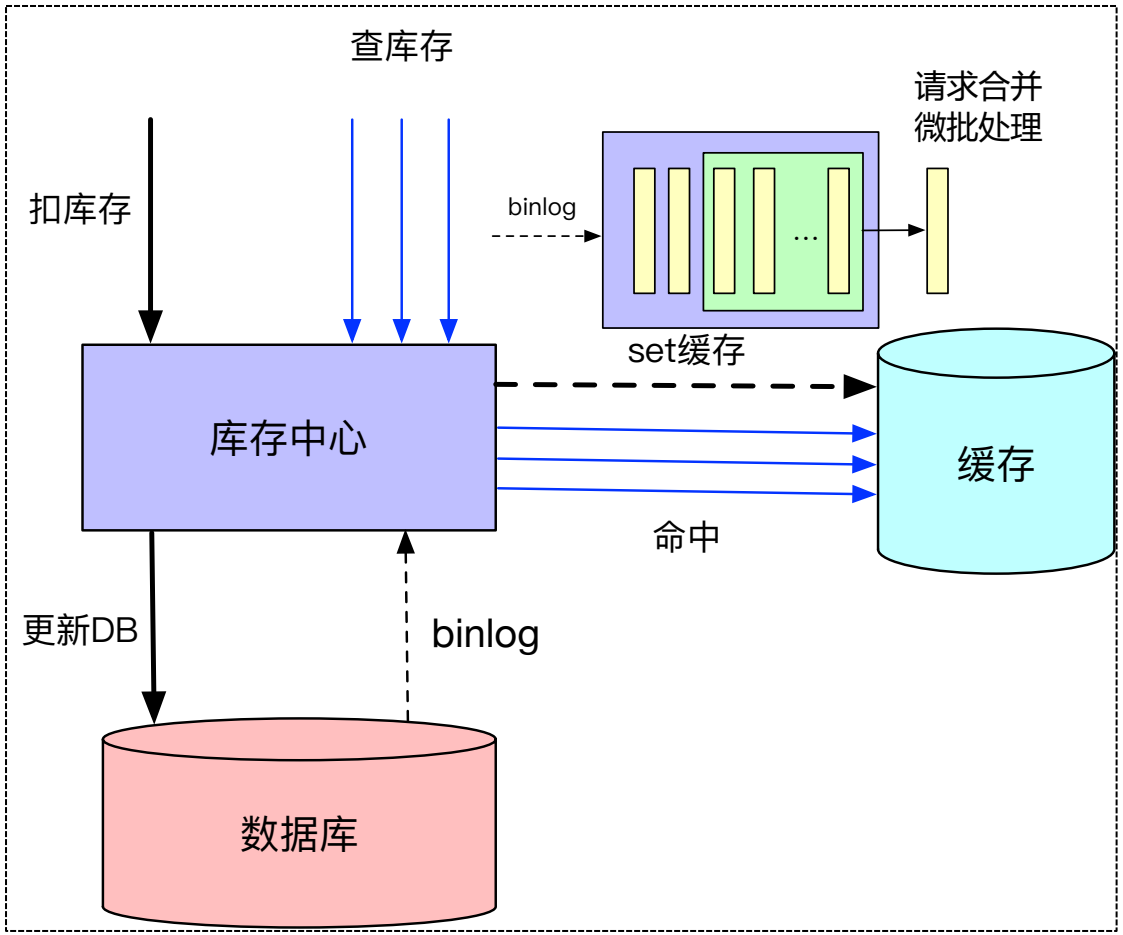

←一致性缺失，由DB的强一致性来保证

```
UPDATE item_stock SET stock = stock – xxx
WHERE stock >= xxx
```

# 三、资源扣减

写操作频繁，缓存命中率降低，DB频繁穿透



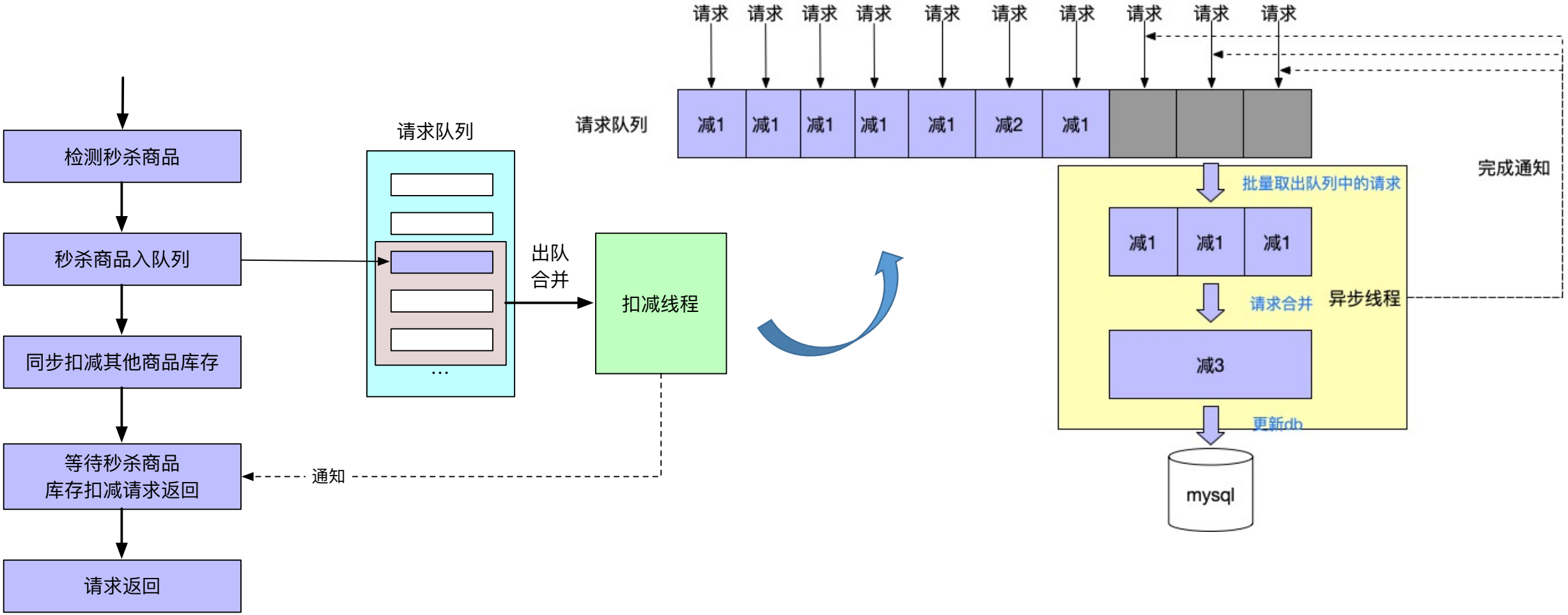




1、binlog消费不及时，降级为同步设置缓存

2、binlog异步设置缓存要防止过期数据覆盖的问题

# 三、资源扣减

秒杀场景下库存扣减的优化

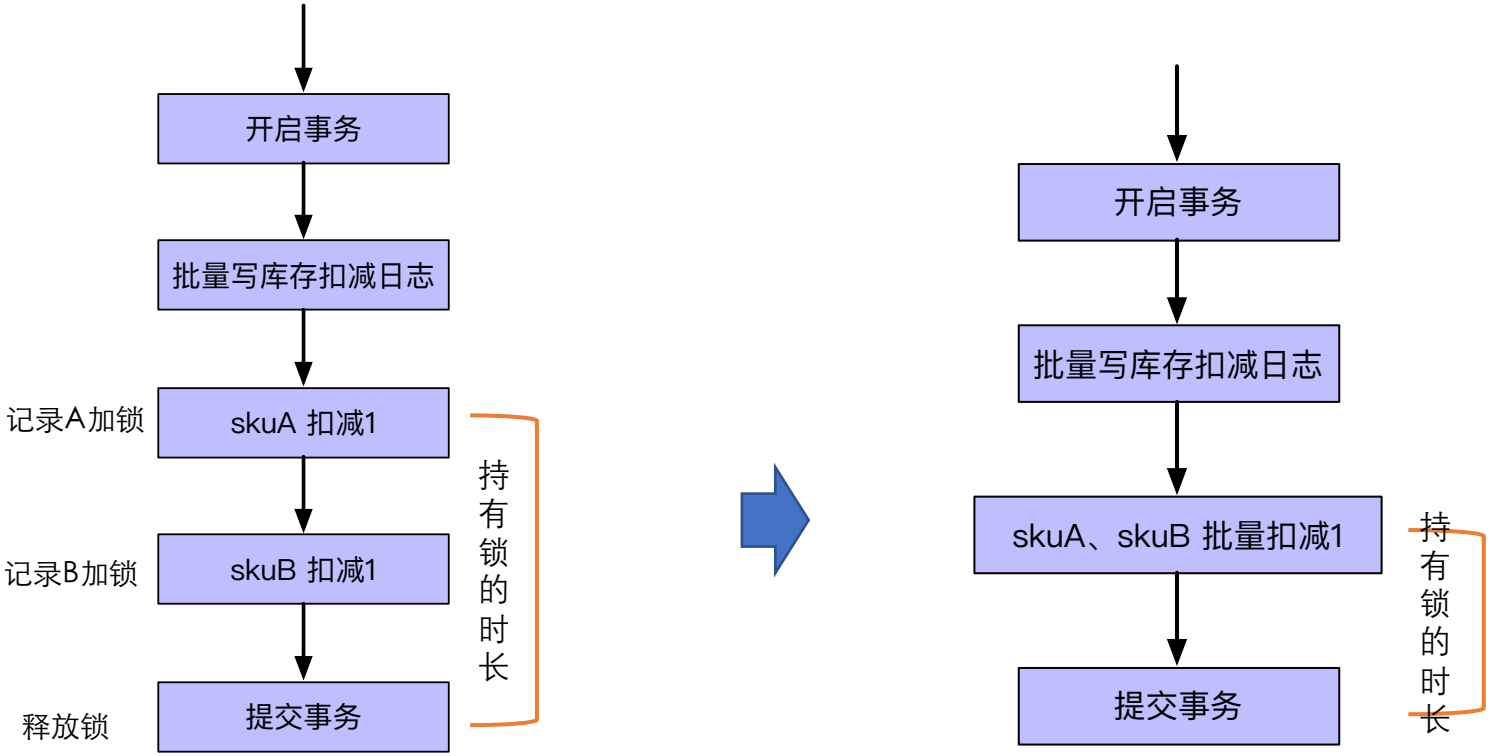


# 三、资源扣减

普通场景下库存扣减的优化

一单购买多个商品，如果购买数量相同，优化成一条sql批量扣减

```
UPDATE stock SET avail_count = avail_count - 1 WHERE sku_id IN (skuId1, skuId2, skuId3 ...)
```



# 三、资源扣减

普通场景下库存扣减的优化

格家网络

一单购买多个商品，如果购买数量不同，也可以用一句SQL搞定

update.sql 192 Bytes

```sql
UPDATE sku_stock
SET available_count =
        CASE sku_id
                WHEN AAA THEN 5
                WHEN BBB THEN 6
                WHEN CCC THEN 7
        END
WHERE sku_id in (AAA,BBB,CCC)
  AND available_count >= (CASE sku_id...)
```

```xml
<update id="batchLockNotPayCount">
    UPDATE sku_stock
    <set>
        avail_count = (
        CASE sku_id
        <foreach collection="updateOperateBOList" item="updateOperateBO">
            WHEN #{updateOperateBO.skuId}
            THEN avail_count - #{updateOperateBO.count,jdbcType=INTEGER}
        </foreach>
        END ),
        not_pay_lock = (
        CASE sku_id
        <foreach collection="updateOperateBOList" item="updateOperateBO">
            WHEN #{updateOperateBO.skuId}
            THEN not_pay_lock + #{updateOperateBO.count,jdbcType=INTEGER}
        </foreach>
        END ),
        order_id = (
        CASE sku_id
        <foreach collection="updateOperateBOList" item="updateOperateBO">
            WHEN #{updateOperateBO.skuId}
            THEN #{updateOperateBO.orderId}
        </foreach>
        END ),
        event_type =  (
        CASE sku_id
        <foreach collection="updateOperateBOList" item="updateOperateBO">
            WHEN #{updateOperateBO.skuId}
            THEN #{updateOperateBO.eventType}
        </foreach>
        END ),
        version = version + 1
    </set>
    <where>
        avail_count >= (
        CASE sku_id
        <foreach collection="updateOperateBOList" item="updateOperateBO">
            WHEN #{updateOperateBO.skuId,jdbcType=BIGINT}
            THEN #{updateOperateBO.count,jdbcType=INTEGER}
        </foreach>
        END )
        AND sku_id IN
        <foreach close=")" collection="updateOperateBOList" item="updateOperateBO" open="(" separator=",">
            #{updateOperateBO.skuId,jdbcType=BIGINT}
        </foreach>
        AND is_deleted = 0
    </where>
</update>
```

# 四、分库分表

什么是分库、分表

其实就是字面意思，很好理解：

分库：从单个数据库拆分成多个数据库的过程，将数据散落在多个数据库中

分表：从单张表拆分成多张表的过程，将数据散落在多张表内

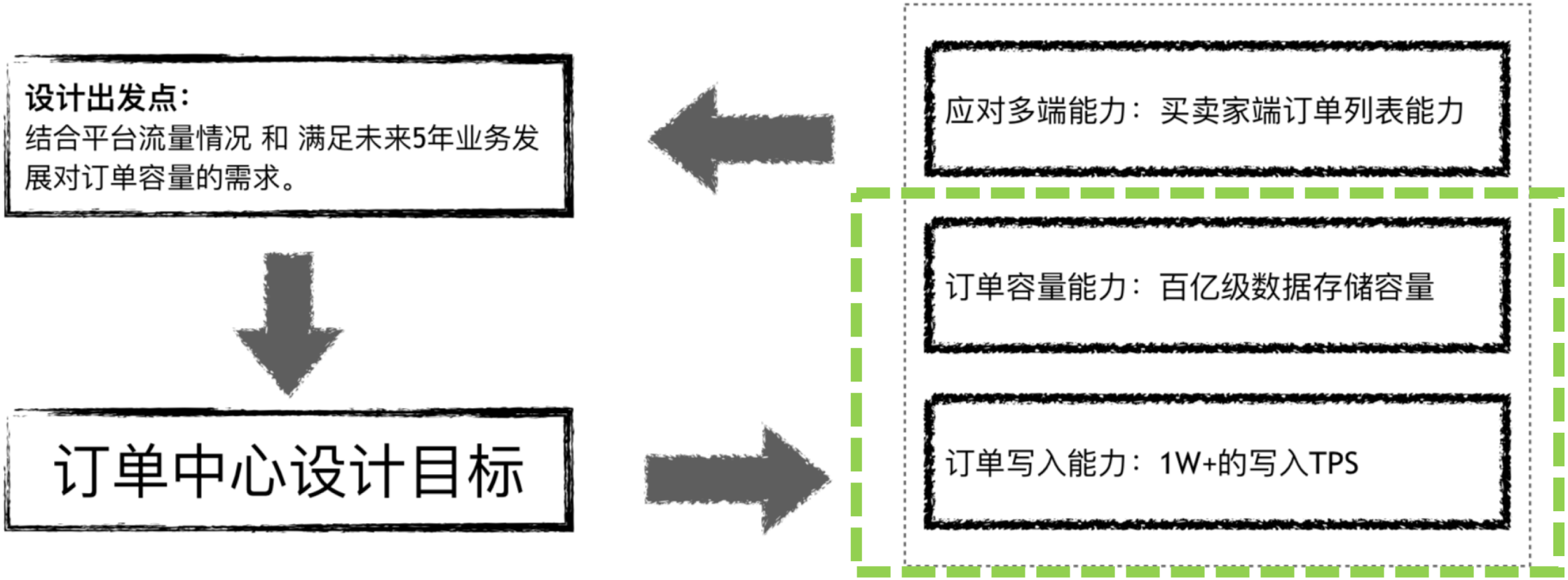
性能

可用性

| 切分方案 | 解决的问题 |
| --- | --- |
| 只分库不分表 | 数据库读/写QPS过高，数据库连接数不足 |
| 只分表不分库 | 单表数据量过大，存储性能遇到瓶颈 |
| 既分库又分表 | 连接数不足 + 数据量过大引起的存储性能瓶颈 |

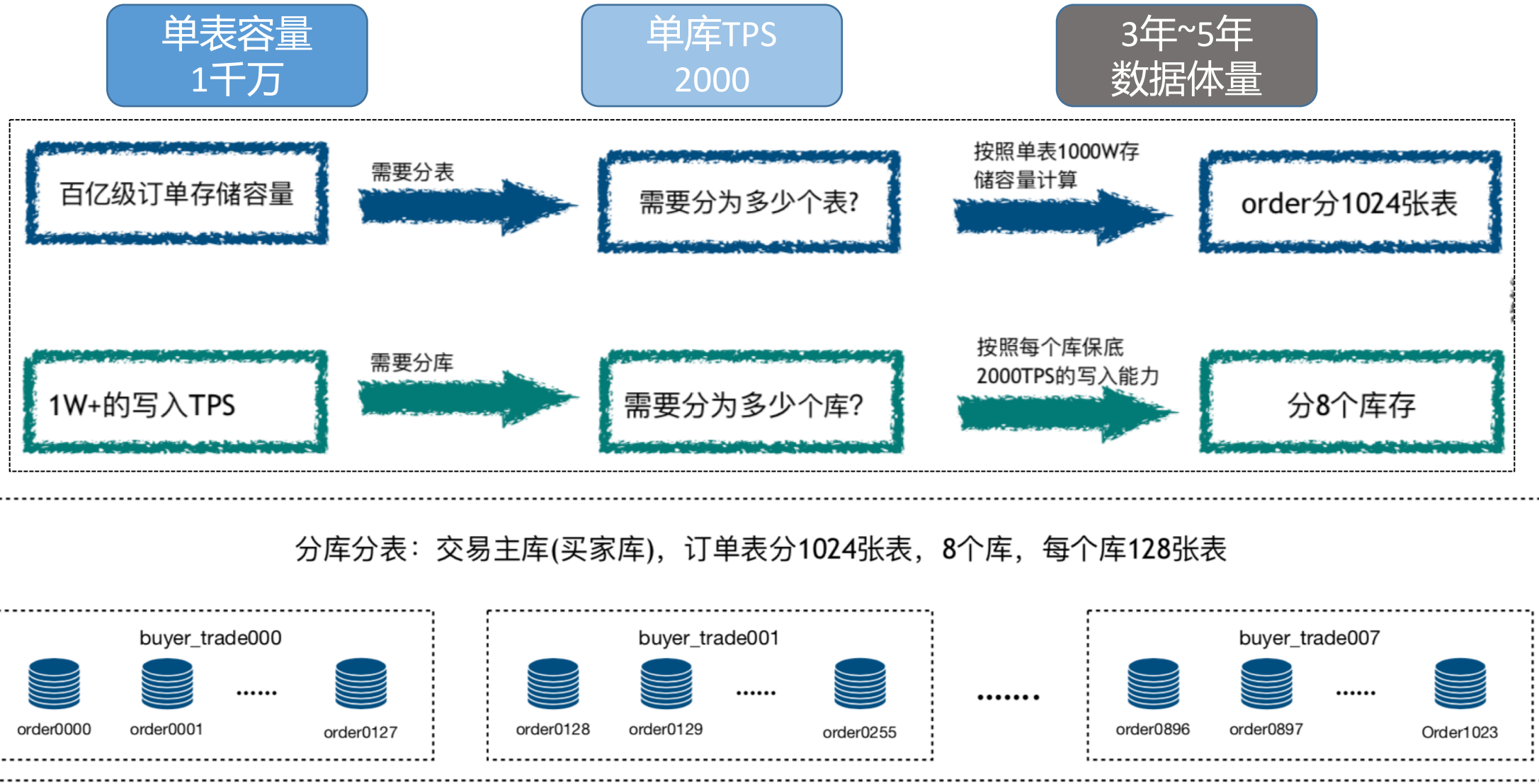
# 四、分库分表

为什么要分库分表



# 四、分库分表

如何计算分表、分库的数量

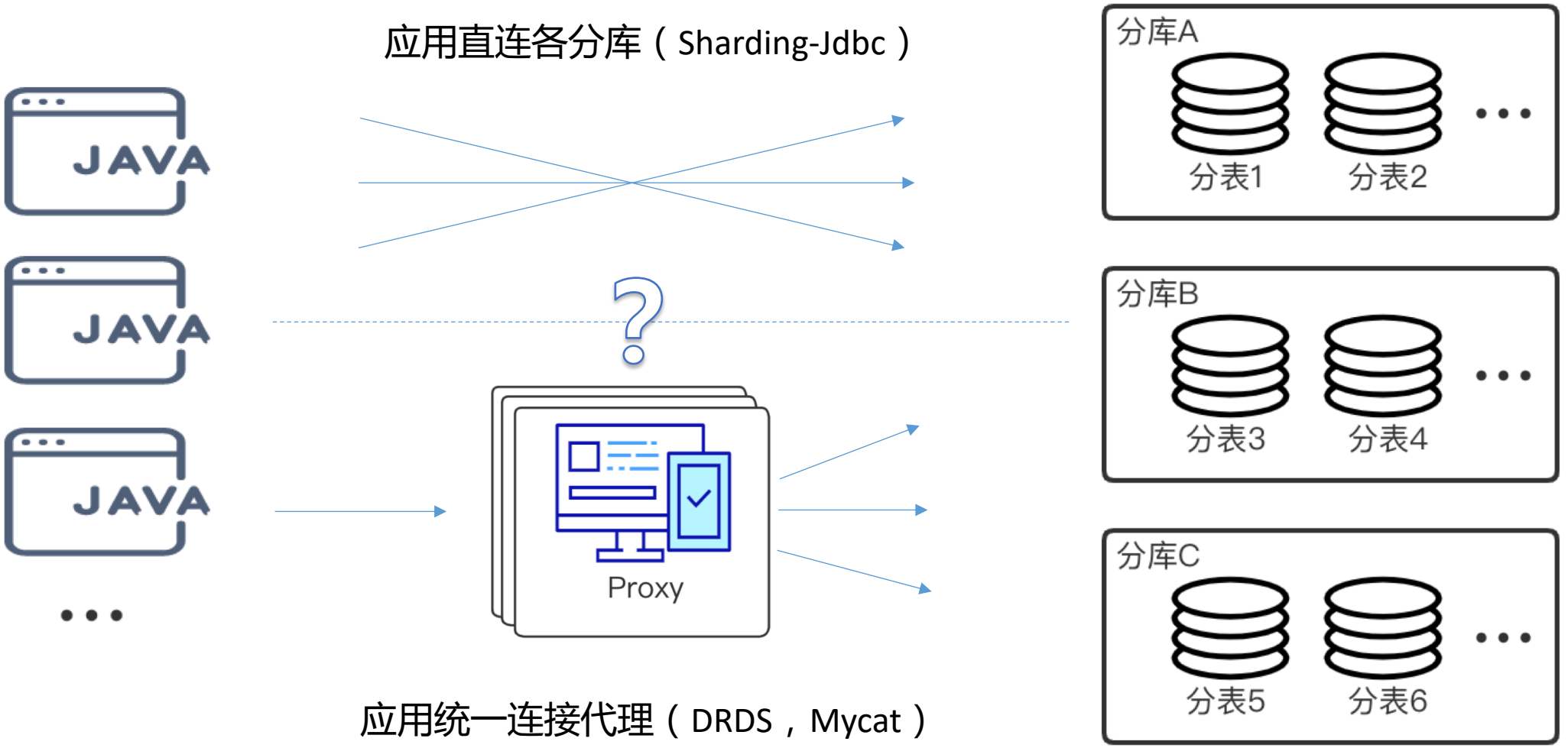


# 四、分库分表

多个库和表，如何分区及定位

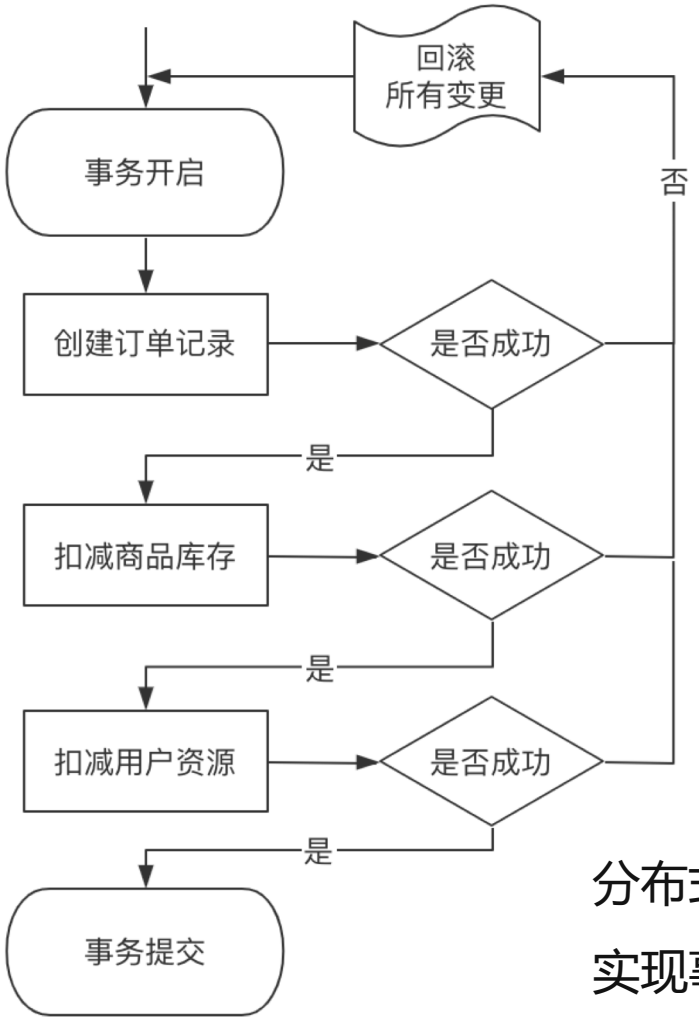

# 四、分库分表

## 应用连接方案



# 五、分布式事务 是什么

## 传统事务（单数据库内事务）




传统事务的四要素

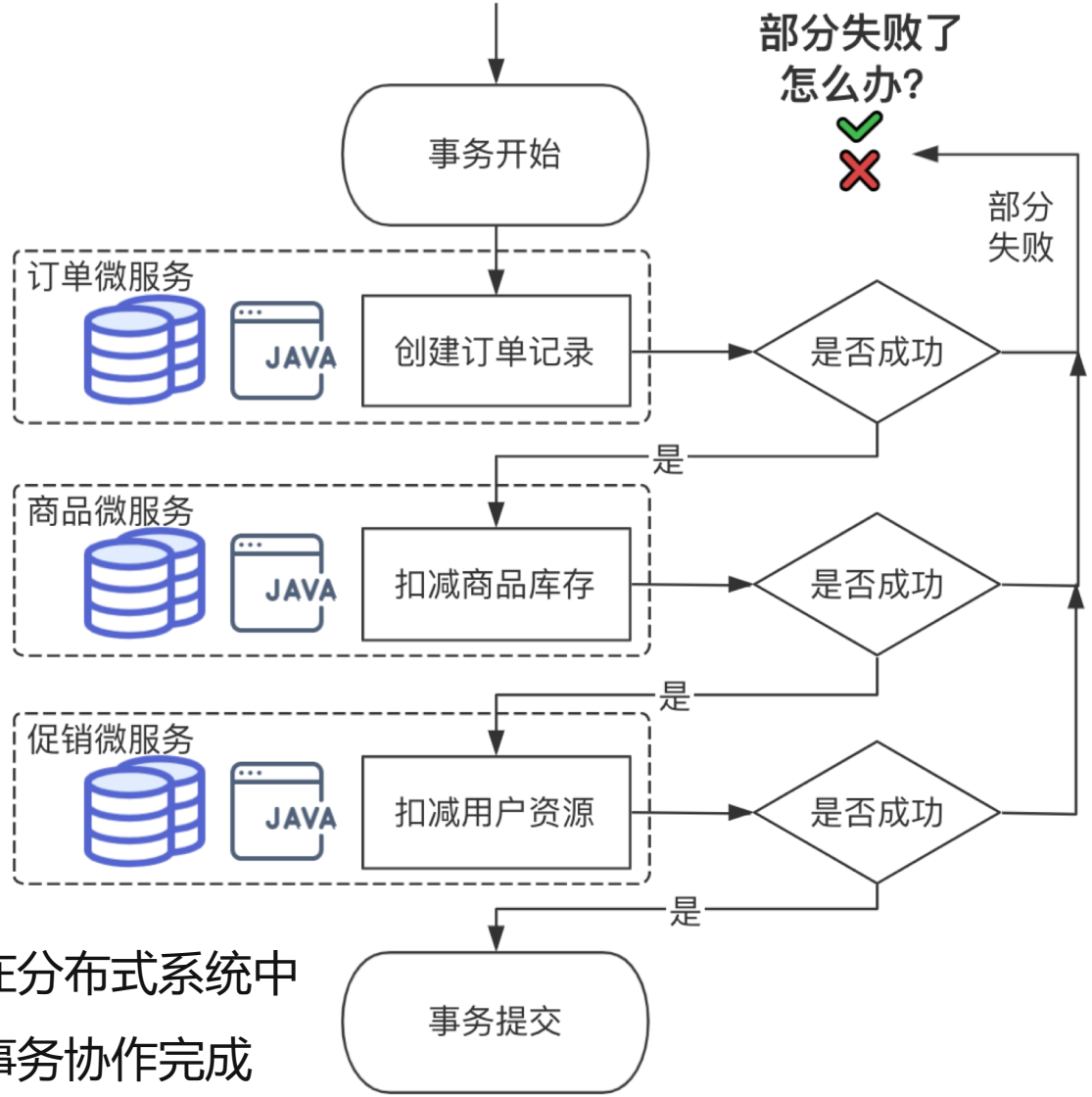
ACID：

原子性(Atomicity)

一致性(Consistency)

隔离性(Isolation)

持久性(Durability)




分布式事务，顾名思义就是要在分布式系统中实现事务，它依赖了多个本地事务协作完成

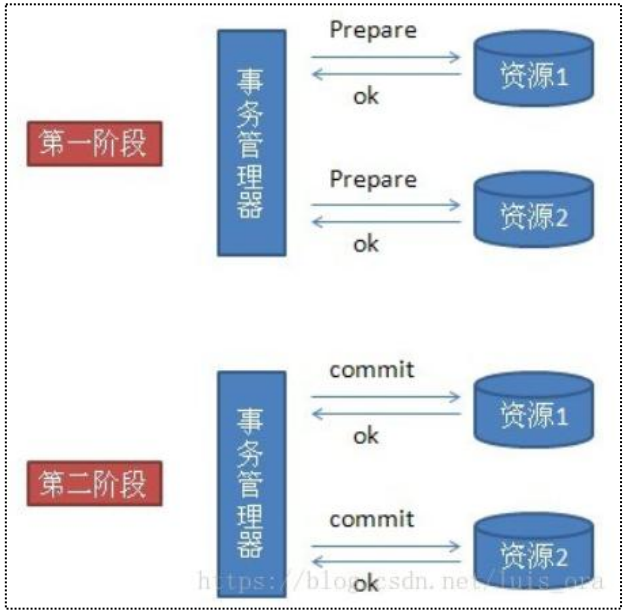
# 五、分布式事务 常见解决方案



分布式系统的理论”魔咒”: CAP只能3选2

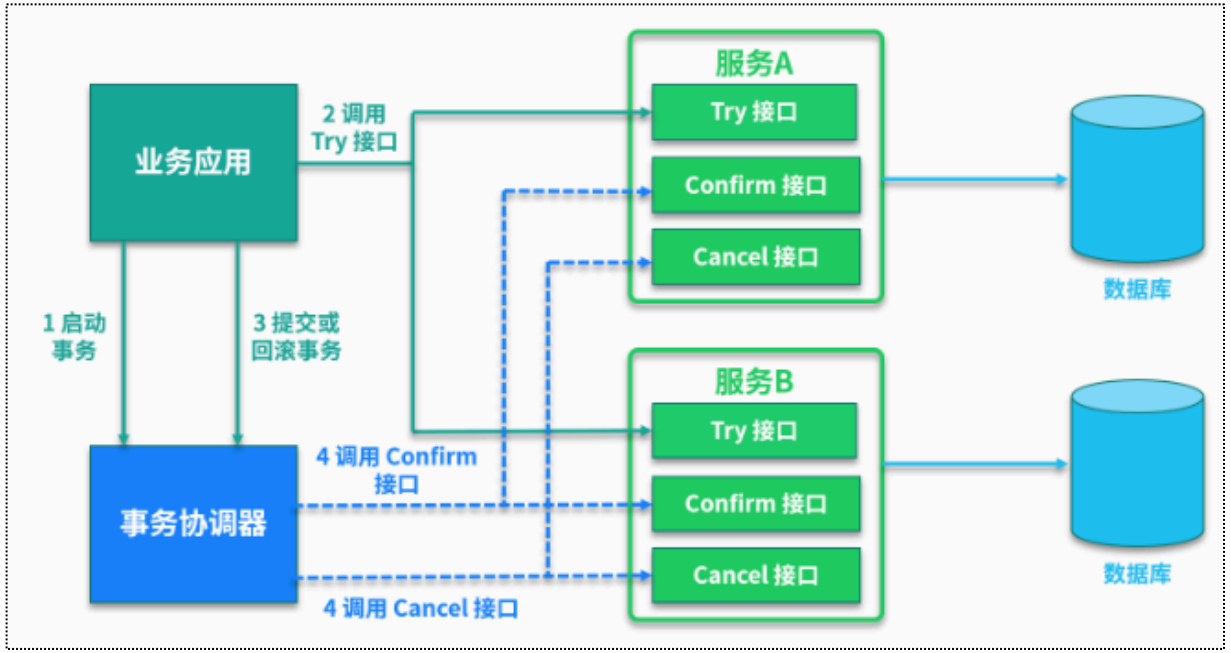
XA规范
（ 2PC ）

CP




TCC

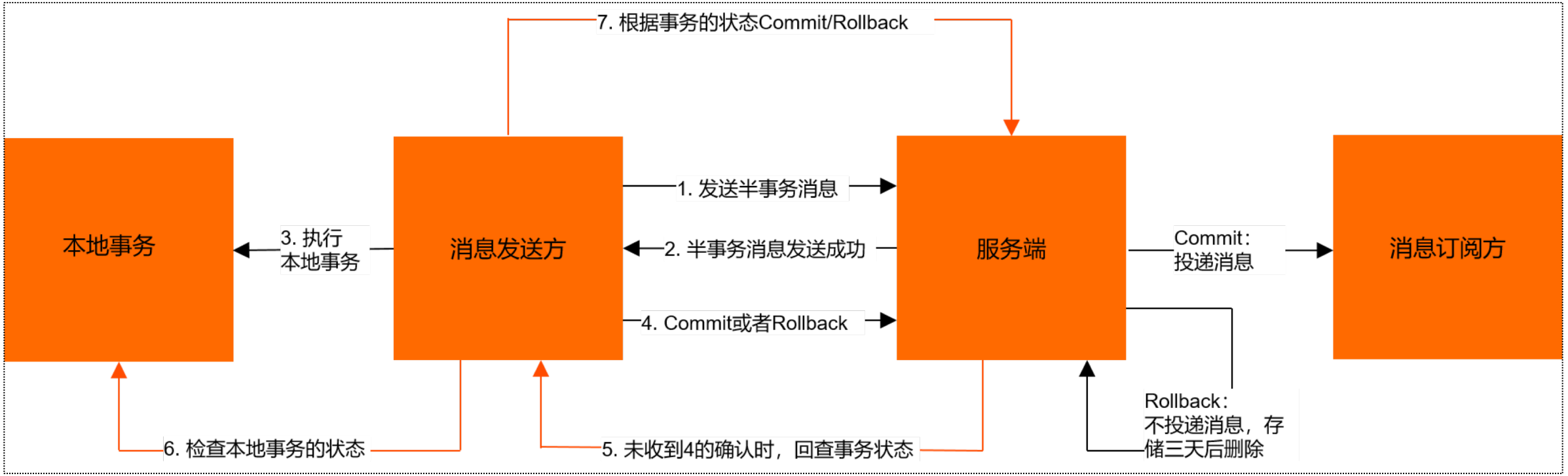
AP




高可用（保AP）场景的选择策略：CAP => BASE

# 五、分布式事务 常见解决方案

## 基于可靠消息方案的分布式事务



# 五、分布式事务

格家的下单业务事务方案

| | TCC | XA(2PC) | 交易下单 |
|---|---|---|---|
| | | | createOrder<br>创建不可见订单 |
| 阶段一 | try<br>请求多个业务方预留业务资源 | prepare<br>询问每个RM是否可以提交事务 | deductResources<br>直接减扣多个资源，直接消耗资源 |
| 阶段二 | confirm<br>确认执行每个业务方的资源操作 | commit<br>提交每个RM上的事务分支 | enableOrderVisible<br>设置订单可见，表示事务成功 |
| | cancel<br>取消执行每个业务方的资源操作 | rollback<br>回滚每个RM上的事务分支 | rollbackOrderDisable<br>回滚订单废单，表示事务失败 → 废单消息<br>下游业务方监听废单消息回滚资源 |

**方案思路：**

1、模拟两阶段提交
2、结合TCC事务方案
3、基于消息回滚资源

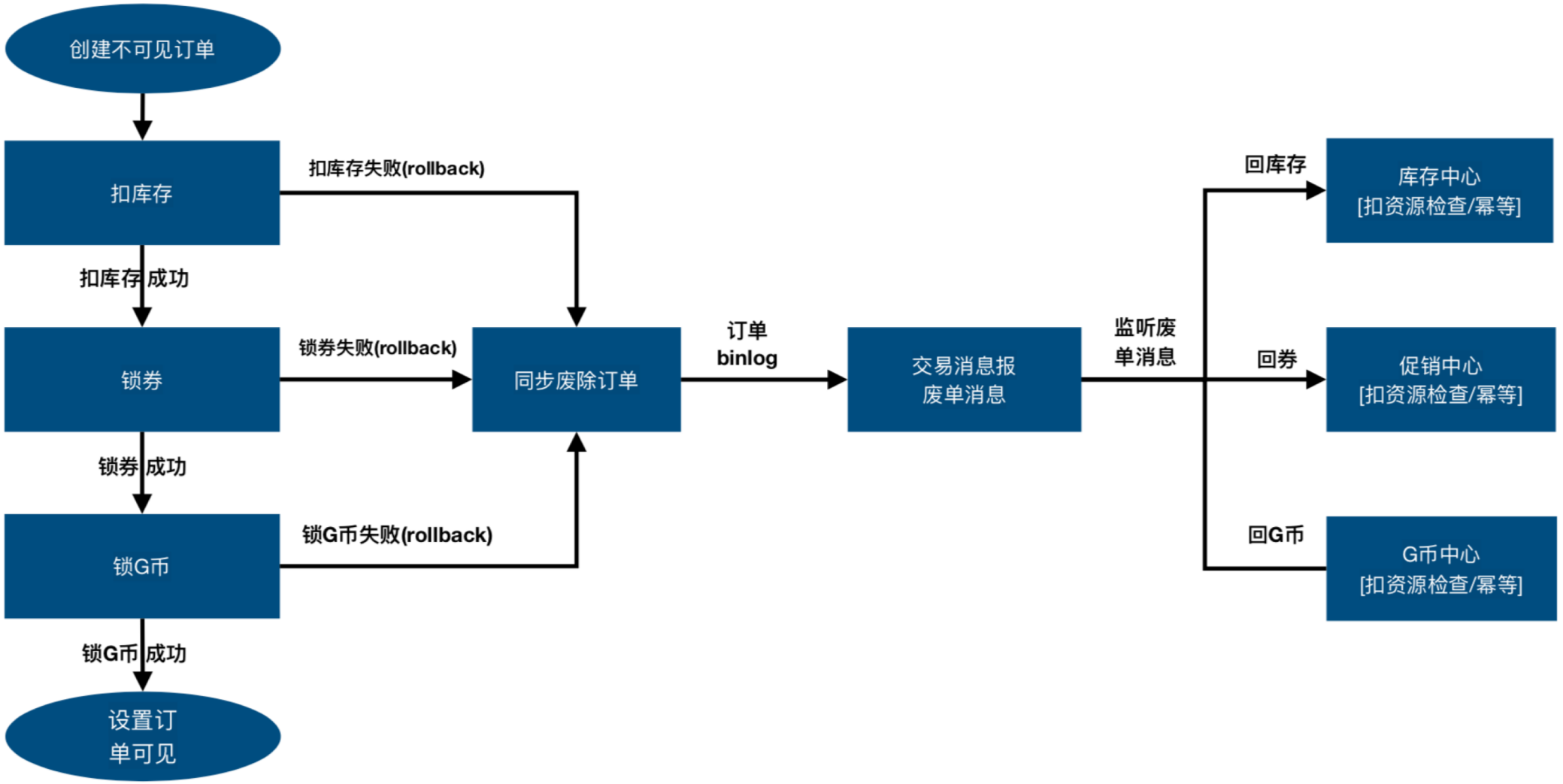
1、预留资源，业务资源方需要有中间状态
2、事务数据和业务数据分开
3、提要提供特定的事务框架
4、cancel操作又是一个分布式事务操作
5、事务成功、失败需要同时协调多方系统

1、预留资源，业务资源方需要有中间状态
2、事务数据和业务数据分开
3、需要引入独立事务协调者
4、需要实现特定的事务协议
5、rollback操作又是一个分布式事务操作

1、业务资源方不需要设计中间状态
2、订单数据同时作为事务记录
3、下单应用作为事务协调者
4、事务成功、失败只需要更新订单即可
5、无需特定协议，无需事务框架，完全基于业务系统
6、业务资源方基于废单消息进行回滚，避免二次分布式事务

# 五、分布式事务

格家的下单业务事务方案



# 问 & 答

# 谢谢

# 参考


1. 并发加载保护器代码: item-center, ItemQueryManager#queryByItemId
2. RDS MySQL CPU使用率较高排查手册: https://help.aliyun.com/document_detail/51587.htm
3. RANGE_HASH函数: https://help.aliyun.com/document_detail/71284.html
4. 利用SQL的CASE WHEN语句优化库存扣减持有锁时间：
   https://gejia.yuque.com/gzg0mk/ahny7t/35855724
5. 什么是分库分表，为什么要分库分表？ https://www.zhihu.com/question/448775613
6. 文曾-《交易平台化技术选型及落地》
7. 凌云-《商品和库存的设计和优化实践》