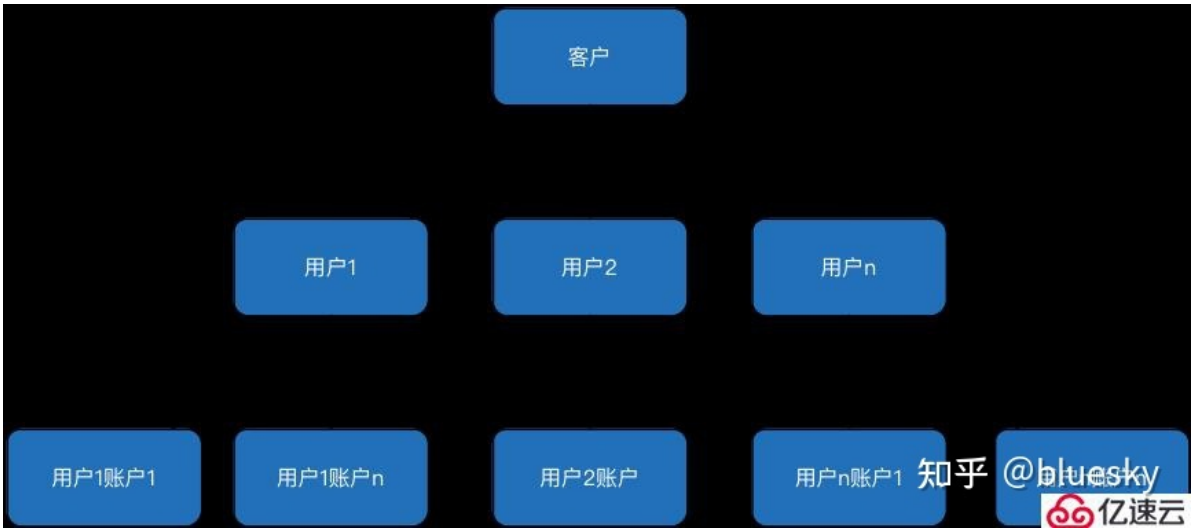
金融热点账户问题

账户结构

宜信支付结算账户体系是客户、用户、账户三层结构，证件号和证件类型唯一确定一个客户，客户号和机构号确定一个用户，一个用户下可开多个不同类型的账户。如图：



账户属性

账户系统的基础是账户，所有的操作都围绕着账户进行，账户包含以下一些属性：

- 会计科目：每个账户金额的变动要体现一些会计的属性，以便会计核算。
- 账户类别：分为个人账户、企业账户、平台类账户。
- 账户明细：账户的明细是反映账户余额变动的每笔详情，采用复式记账法，包含本对方账号、账户等信息、摘要、借方的发生额及余额等信息。
- 账户余额：记录账户的实时余额。

一、什么是子账户

子账户是为了方便用户进行资产管理，通过现有账户(以下称为主账户)进行创建和管理的投资账户，一个主账户可以创建多个子账户进行投资和资产管理。

子账户上限数量说明：

VIP1-VIP4，可创建2个子账号；
VIP5-VIP9，可创建20个子账号；
VIP10-VIP11，可创建100个子账号；
VIP12-VIP14，可创建200个子账号；
VIP15-VIP16，可创建300个子账号。

一个主账户可以创建多个子账户，目前对VIP1及以上级别开放子账号功能

何为热点账户?

热点账户即系统中，被高频繁地进行资金的进出操作，频繁出现加锁解锁操作的账户。比如，B2C系统中的企业虚拟户，用户购买商品时，资金从用户虚拟户转到企业虚拟户；用户退款时，资金从企业虚拟户转到用户虚拟户；这时，对企业虚拟户的资金进出操作就会成为整个业务的瓶颈。

基本解决思路

\1. 控制并发数：所有进出都先进到线程池中，以线程池的并发数控制实际对热点账户操作的并发数

\2. 缩小事务控制范围：对热点账户的操作放在尽量小的事务范围，减少时间分片，提高成功率

\3. 乐观锁：对热点账户的更新操作使用乐观锁，减少时间分片，提高成功率

改进型解决思路

资金进出账户分开设计，

入账账户设计：

思路：先记流水，定时增加账户余额，减少锁竞争

不足：账户余额更新不及时

出账账户设计：

思路：出账账户拆分成多个子账户，具体使用的账户通过前置进行hash分配（具体的hash函数会有更多方案）

不足：在子账户扣款到最后时，可能总余额是够的，但是子账户余额不足了

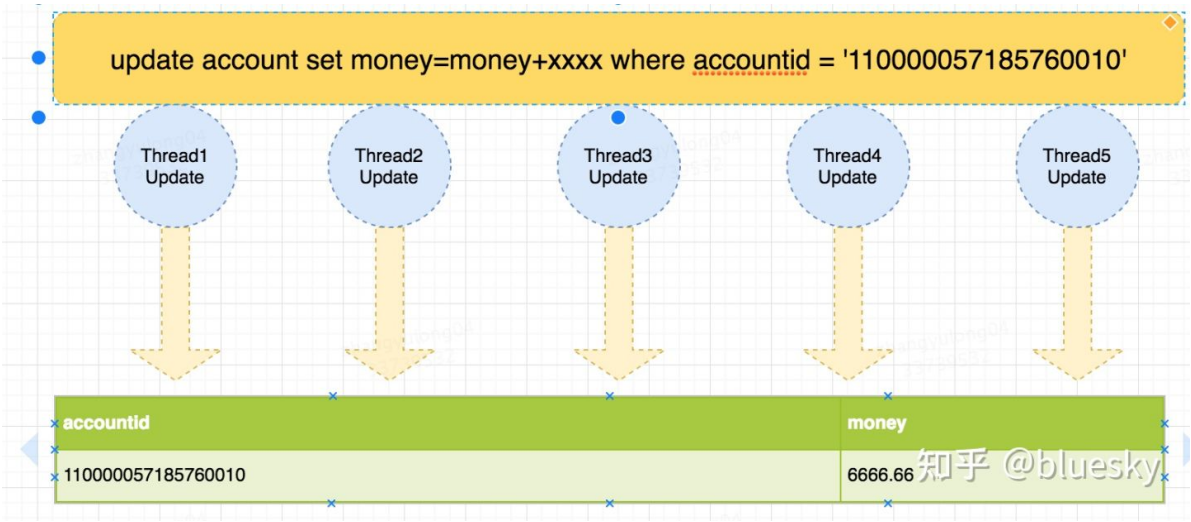
更复杂的方案

入账账户不变，

出账账户动态平衡，优化hash方案

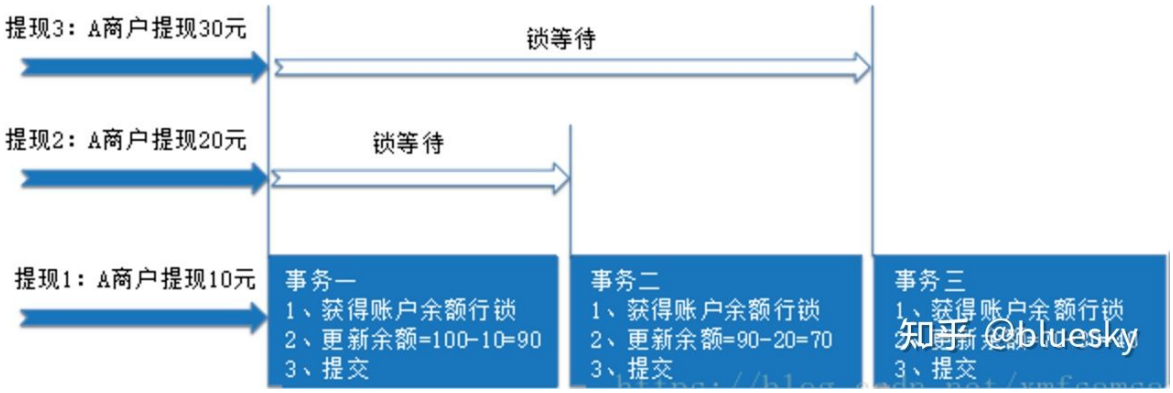
前言：方案设计前提
一般账务系统对账户的冲扣需要满足以下两点
1：更新账户表中的账户余额。
2：记录账户明细表中的账户更新前余额，账户更新后余额，操作金额。
其中对账户表中的余额更新一般是直接update，对账户明细表中的操作前金额，操作后金额和操作金额就是对账户表update的记录
1:为什么做热点账户设计
热点账户交易是性能瓶颈,在银行或者第三方支付系统的账务数据库的处理中，数据从一个账户转出，或者有数据转入一个账户，账户都会收到记账请求，并都有一个记账处理的过程。记账处理过程主要包括两部分，一是记录记账凭证，二是更新账户的余额。为了保证账户不被其他请求影响数据的准确性，在

进行记账处理时，会先对账户的资源加锁，记账处理完毕后会自动释放锁。随着账务处理业务量的增大，账务数据库中的账户常常会在瞬间产生多个并发操作，但所有对应的并发线程中只有一个线程能够持有当前账户的资源锁，其他线程必须等待该锁被释放后再逐一进行记账处理，这样该账户将会被频繁加锁释锁，使该账户成为账务数据库热点，产生性能瓶颈点，严重影响账务数据库的性能。



对于同一账户ID来说，由于实际业务需要更新账户可用余额和账户流水日志，所以单笔冲扣功能是在一个事物中进行操作,任何更新操作都会对数据上行锁，图例如下

A账户余额100元，并发三笔提现交易：分别提现10元、20元、30元



2:业界关于热点账户冲扣设计方案

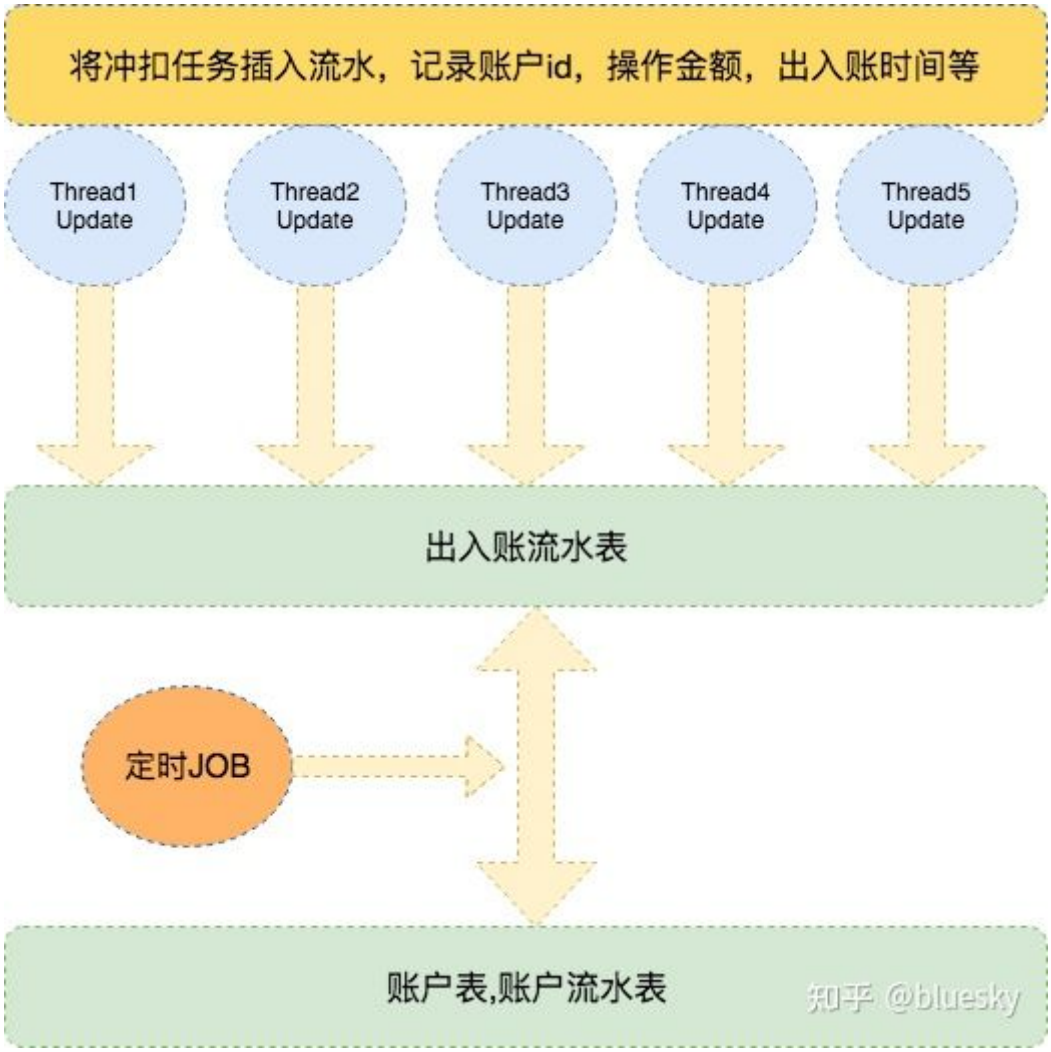
1.并发度控制

同一时刻，对同一账户修改的请求数越多，这个账户的锁等待问题就越严重，所谓并发度控制就是要控制同一时刻对热点账户请求的数量，可以通过控制上游支付系统并发请求数据或者账务系统处理的并发请求数来实现。这一方案的缺点是对业务是有损的，当热点账户出现的时候，支付或者账务处理失败率会增加，用户的体验会变差，较大的银行或者第三方支付公司用地比较少。

accountid	money
110000057185760010	6666.66

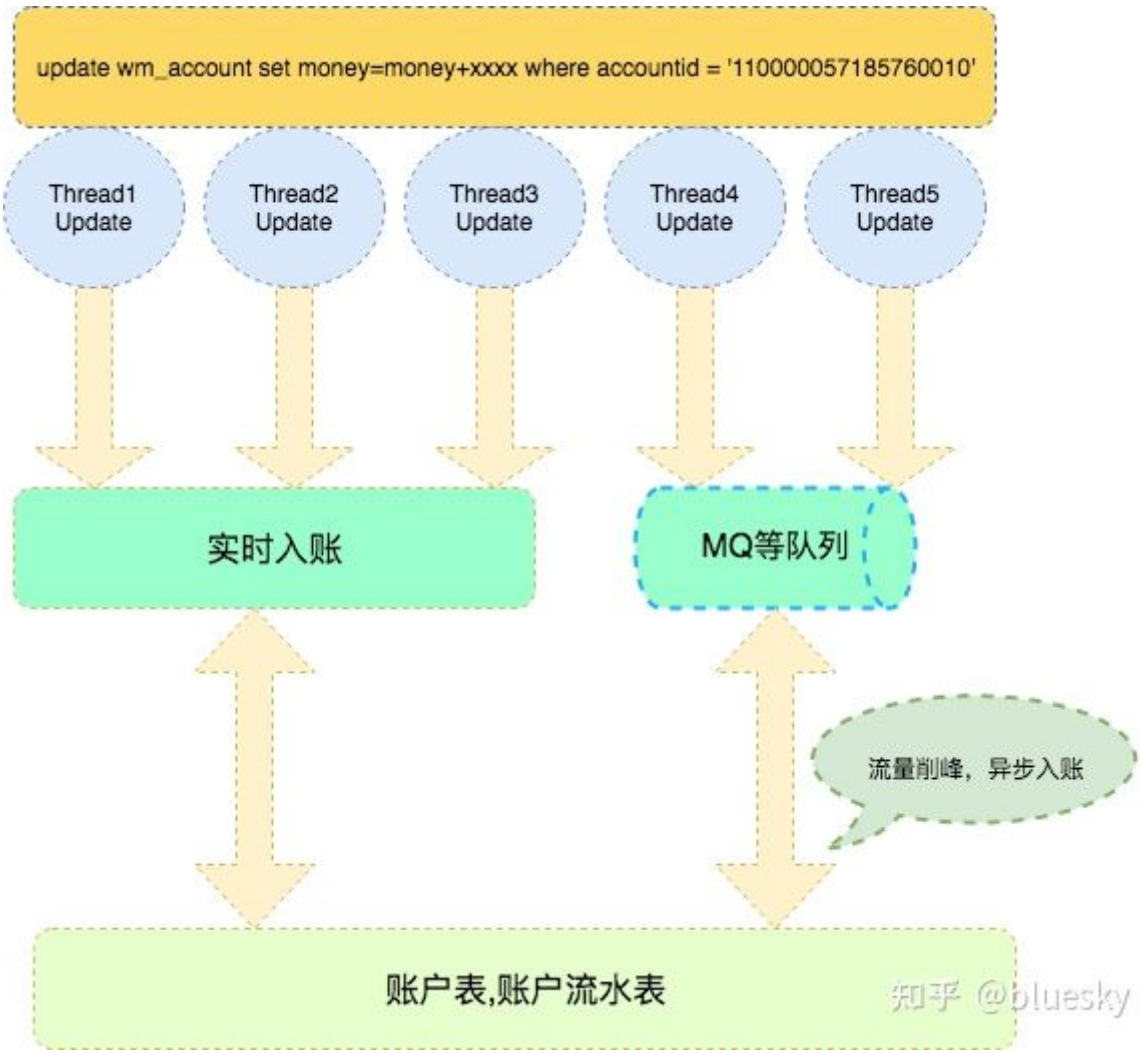
2.汇总明细记账

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。



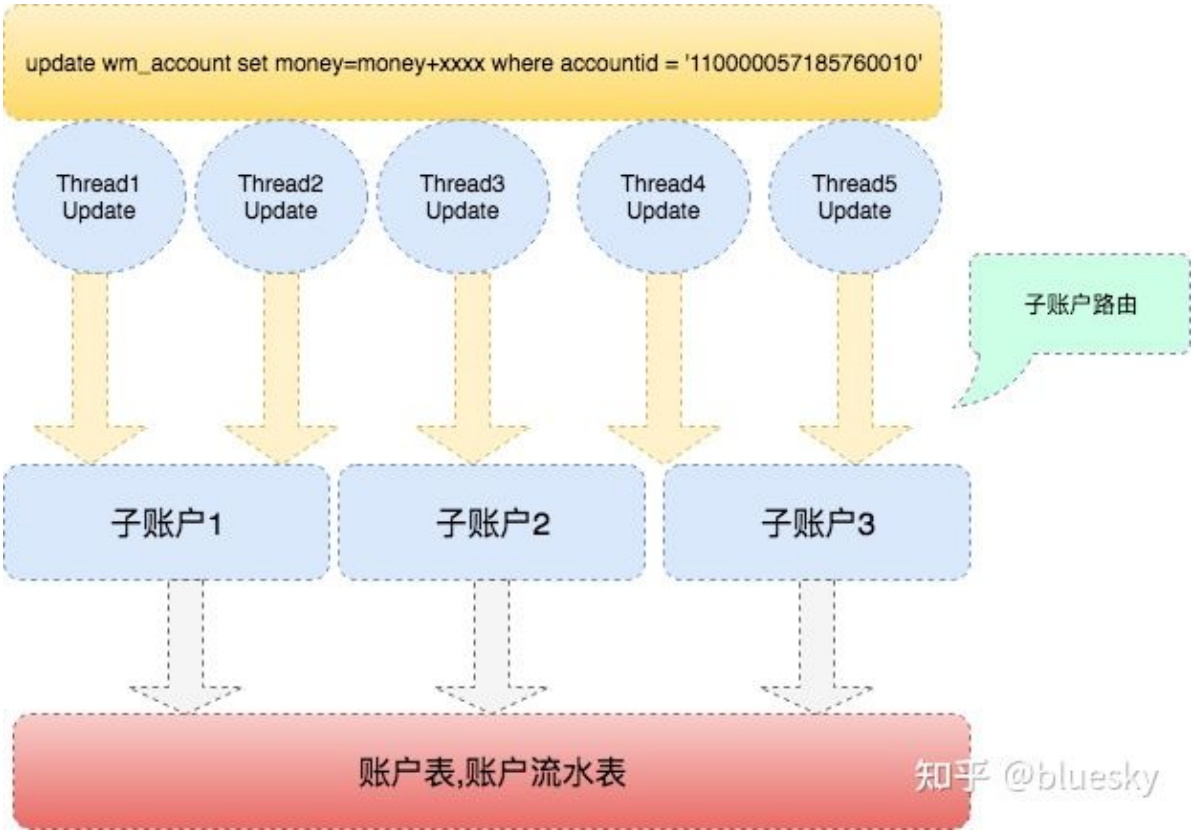
3.缓冲入账

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"。详细地讲，假如账务系统对同一个账户的处理阈值为100笔/s，24小时不间断服务(一天能处理86400000笔)。当业务高峰期来临的时候，热点账务的请求数会达到200笔/s。当账户的交易低于100笔/秒的时候，账务系统几乎还是实时地处理了记账请求，而当交易大于100笔/秒的时候，账务系统先返回结果，把账务处理丢到可靠的处理队列中，等并发量不大的时候慢慢消化，对用户来说感受到的体验还是很快就记账成功了。　　这个方案是有个前提是：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



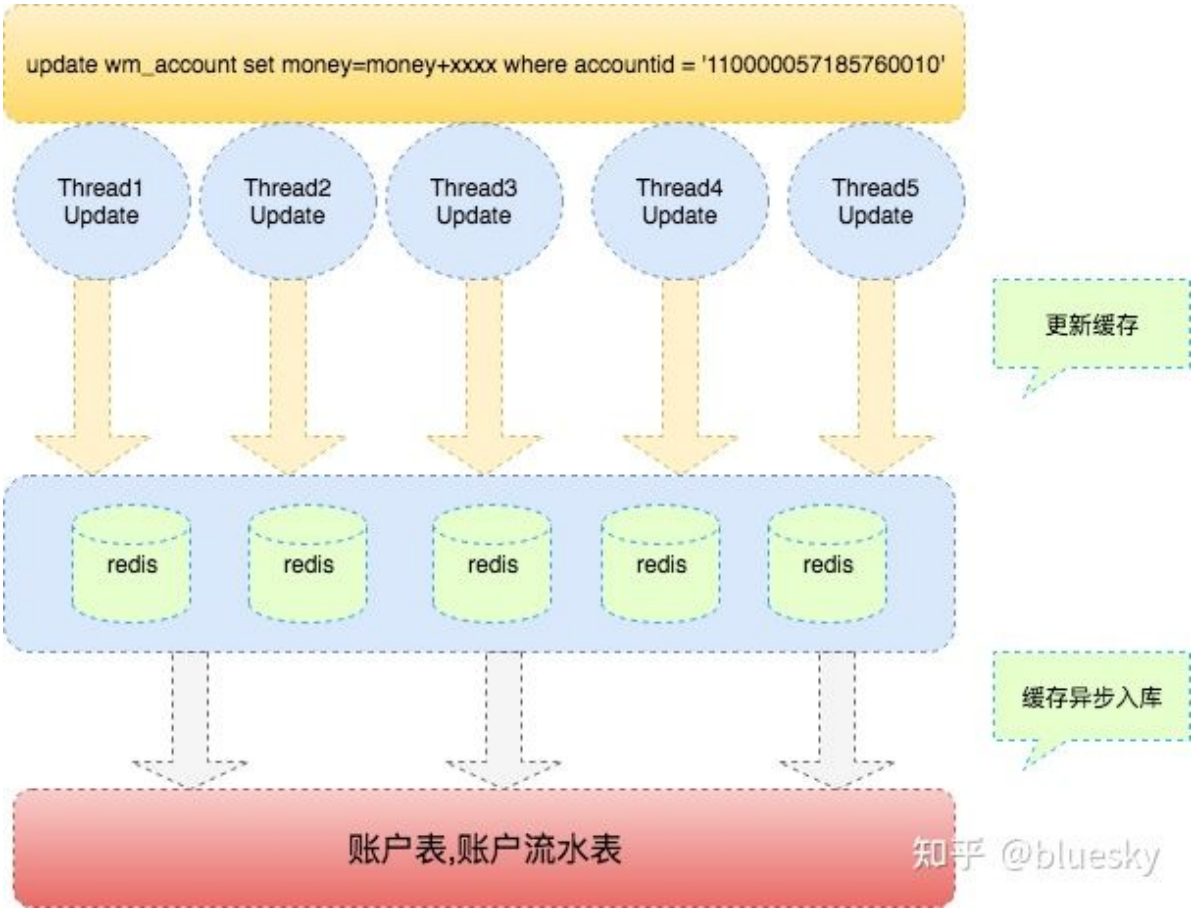
4.子账户拆分

具体来讲就是创建与热点账户对应的多个影子账户，所述影子账户与所述账户的数据结构相同，将所述影子账户设置为隐藏，并将所述账户的余额分散至各个影子账户。当账务系统接收到账务请求的时候，通过前置进行hash分配（具体的hash函数会有更多方案）选择影子账户进行记账，这样就将原来对一个账户的请求分散到多个影子账户中，分散了账务热点。　　这个方案也有缺点：通过算法选择的影子账户扣款，影子账户的余额可能是不足的，但账户的总余额是够的，这样可能影响账务处理的成功率。

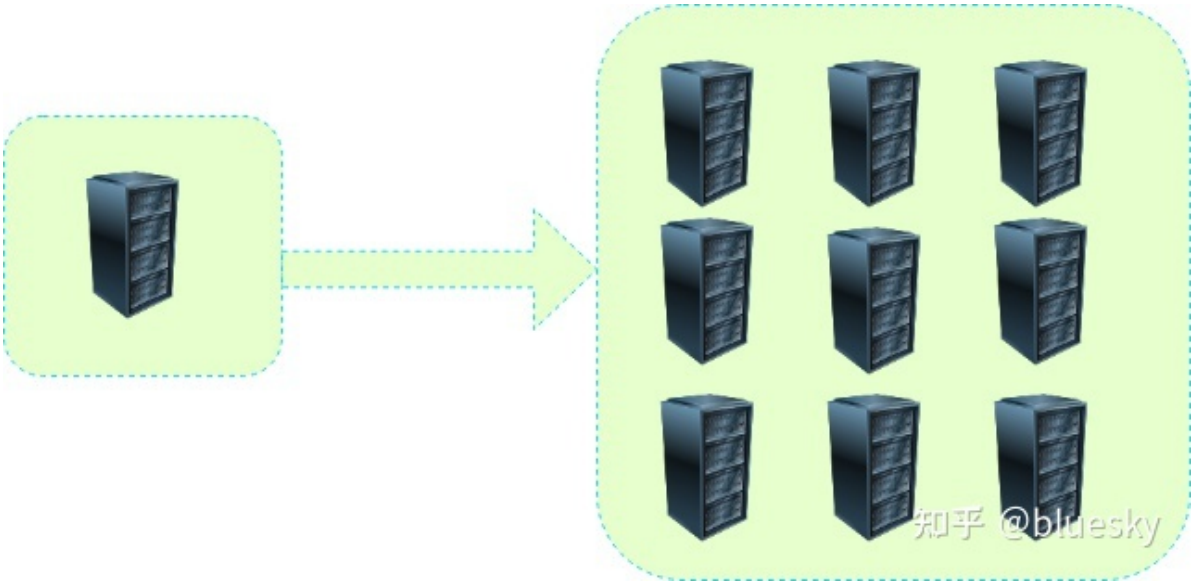


5.内存数据库+缓存入账

提高单台数据库服务器处理能力（I/O,CPU,memory）或者选取内存数据库实时地处理记账请求，然后异步地存储到可靠数据库上。



6.升级服务硬件,对CPU内存等进行升级



3：几种方案的对比

1：并发度控制

对单个账户并发操作进行限流降级控制，使得系统健康的完成入账出账操作，但是在并发很高的情况下还是会杀死很多正常的冲扣功能，会极大的提高冲扣的失败率，所以对我们账务系统来说不是允许的。

2：汇总明细入账

对账户的冲扣操作已流水的形式记录下来，通过定时job来将出入账流水更新到业务表中。这种做法对于频繁的入账来说性能提高明显，但是因为没有对总金额进行校验，对支出类业务（账户减钱）类来说，有账户透支地风险。并且对于金额的校验需要通过流水数据和当前可用余额来判定，有并发问题，计算很难准确。并且我们实际线上业务是【频繁出账，低频入账】，所以此办法不可取。

3：缓冲入账

需要动态判断流量低峰高峰，维护请求队列，有账户透支地风险，并且异步请求中结果不可控。

4：子账户拆分

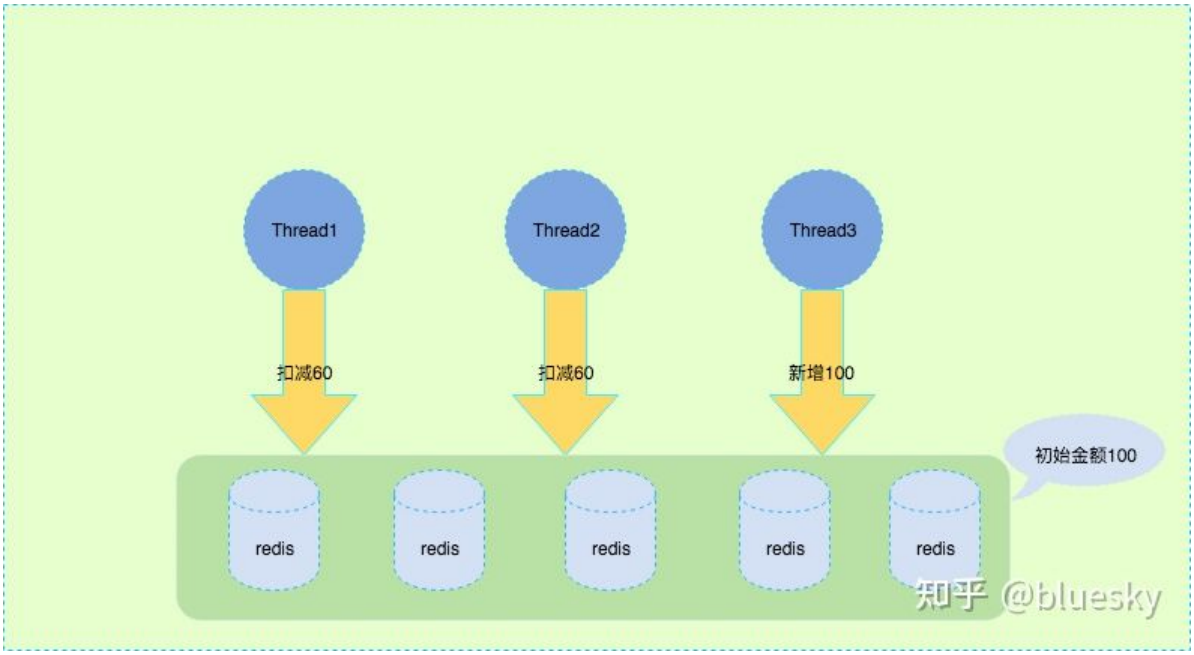
子账户拆分方案中对于子账户的扣款进行负载，可以满足对同一账户的高频访问负载到其子账户上，极大满足了并发的需求，子账户的余额可能是不足的，但账户的总余额是够的，这样可能影响账务处理的成功率，并且处理对子账户的扣款和入账来说需要做到金额相对平均比较复杂，对记录账户期初余额期末余额处理涉及到并发，相对复杂。

5：增加硬件处理能力CPU.内存等

备选方案,无法从根本上解决单点账户的并发压力。

6：内存数据库实时地处理记账,异步入库

使用redis做数据前置处理，将数据库中的热点账户金额初始同步到redis中，然后将操作记录流水，通过job定时任务刷新流水到业务表。这样将db和缓存分开极大的加大了并发性能，但是却衍生出来一个问题如下



假设redis初始金额为100，

（1） 当线程1对redis账户金额进行原子减操作时，剩余金额40,并记录流水表等待异步入账

（2） 当线程2对redis账户金额进行原子减操作时，剩余金额-20,此时金额已经为负，按照业务要求金额不能为负所以必须要做反向操作

（3） 当线程2还没有对redis余额进行反向操作维护的时候又出现线程3进行充值操作，此时金额又变成-20+100=80，已经出现金额混乱，对业务要求的期初余额期末余额无法准确的满足，所以对redis的金额进行同时冲扣会带来余额的并发问题。

但是对缓存进行操作和延迟批量流水入账可以极大的满足我们对性能的需求，所以在**【2.汇总明细记账】**和**【5.内存数据库+缓存入账】**的基础上进行改良来满足对我们的业务需求

4：详细方案设计

方案设计前提:

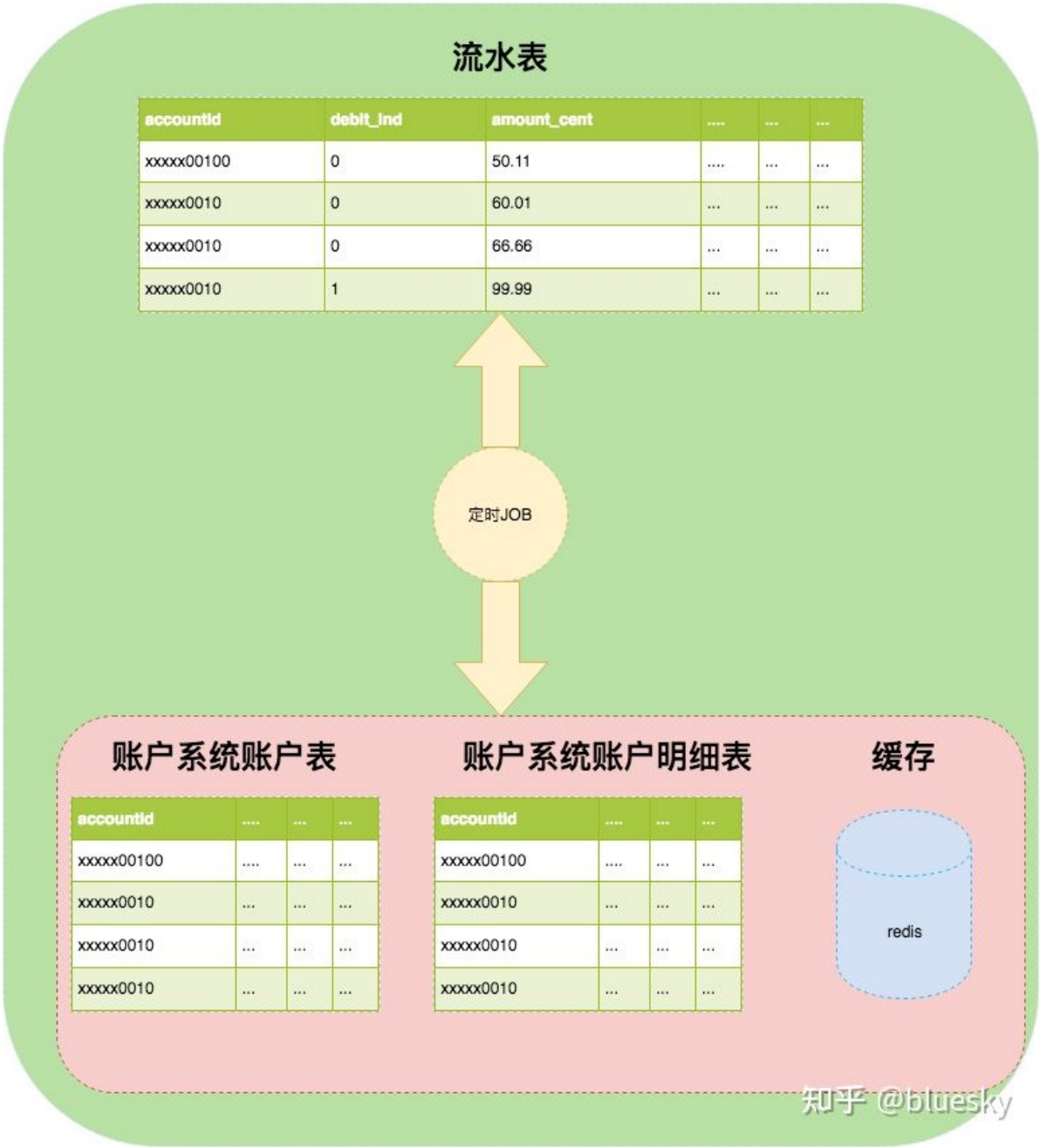
（1）：【对账户的余额的更新】：准确的更新账户余额，不允许出现多扣，少扣等情况。

（2）：【对账户操作记录的更新】：准确的记录账户流水表中期初余额，期末余额，操作金额等情况，不允许出现任何的金额错误发生。

前期准备:

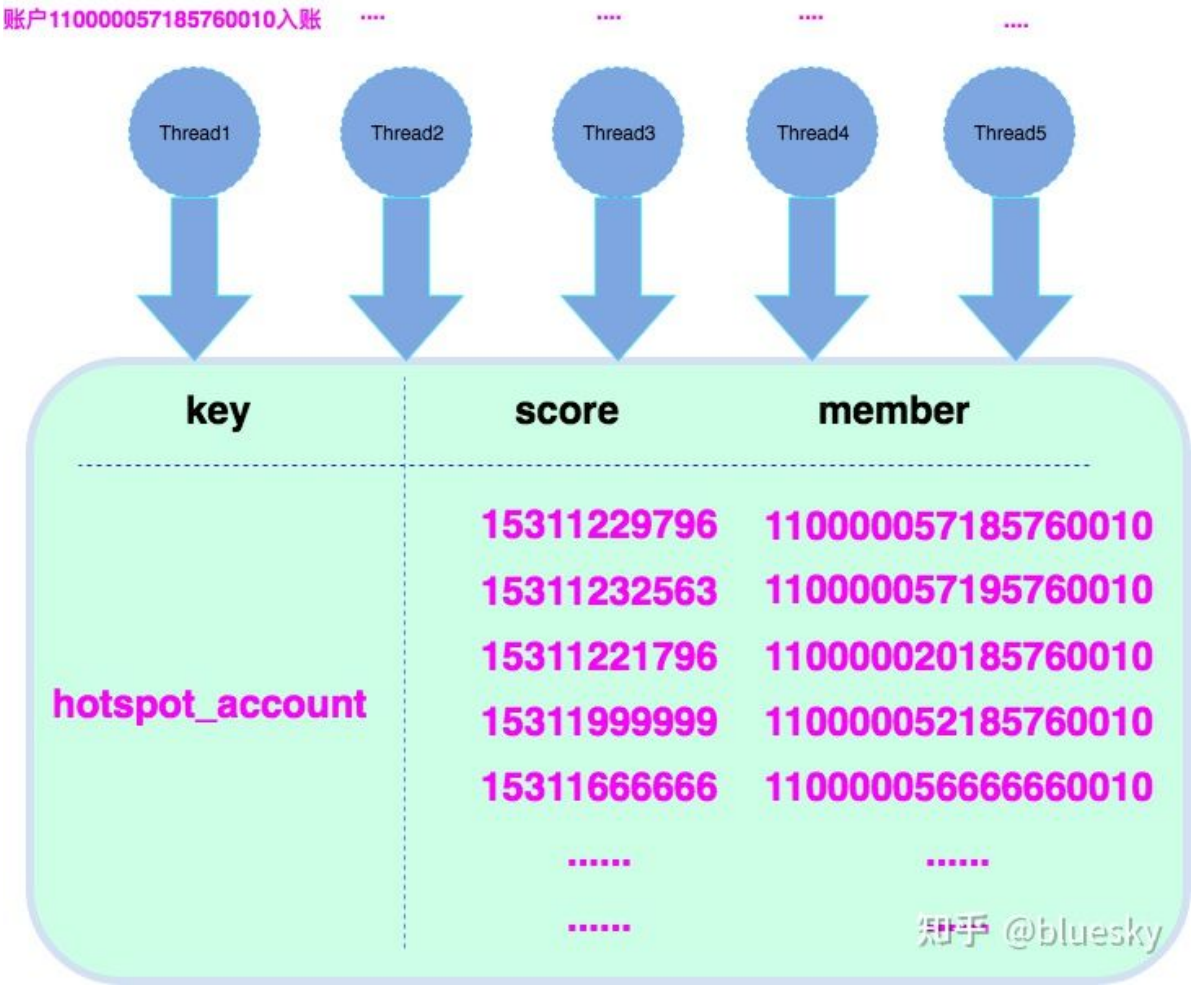
(1)：新增延迟入账【流水表】，新增入账，出账数据先入【流水表】,通过定时任务将【流水表】入账和出账数据同步到业务数据表中，并且负责新增入账数据的缓存同步工作。

下面的方案会对此表统一称为【流水表】



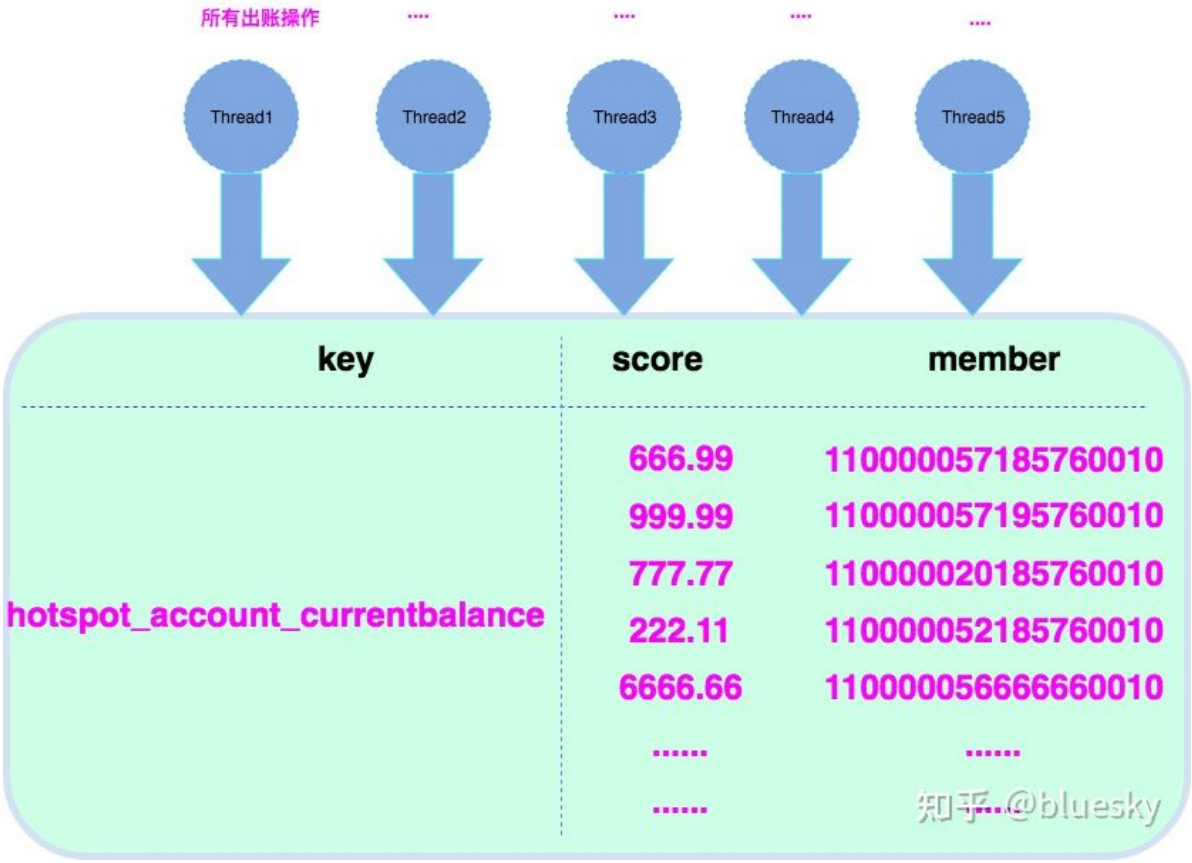
(2)： 新增【redis】数据结构【SortedSet（有序集合）】 key为【hotspot_account】

下面会对这个数据集合称为【缓存操作记录】



其中score为当前账户操作时间【新覆盖旧】,member为出入账的账户ID。key【hotspot_account】,所有账户的入账出账操作需要记录到hotspot_account中，主要是提供给**【图1中定时任务】**获取所有账户流水ID。

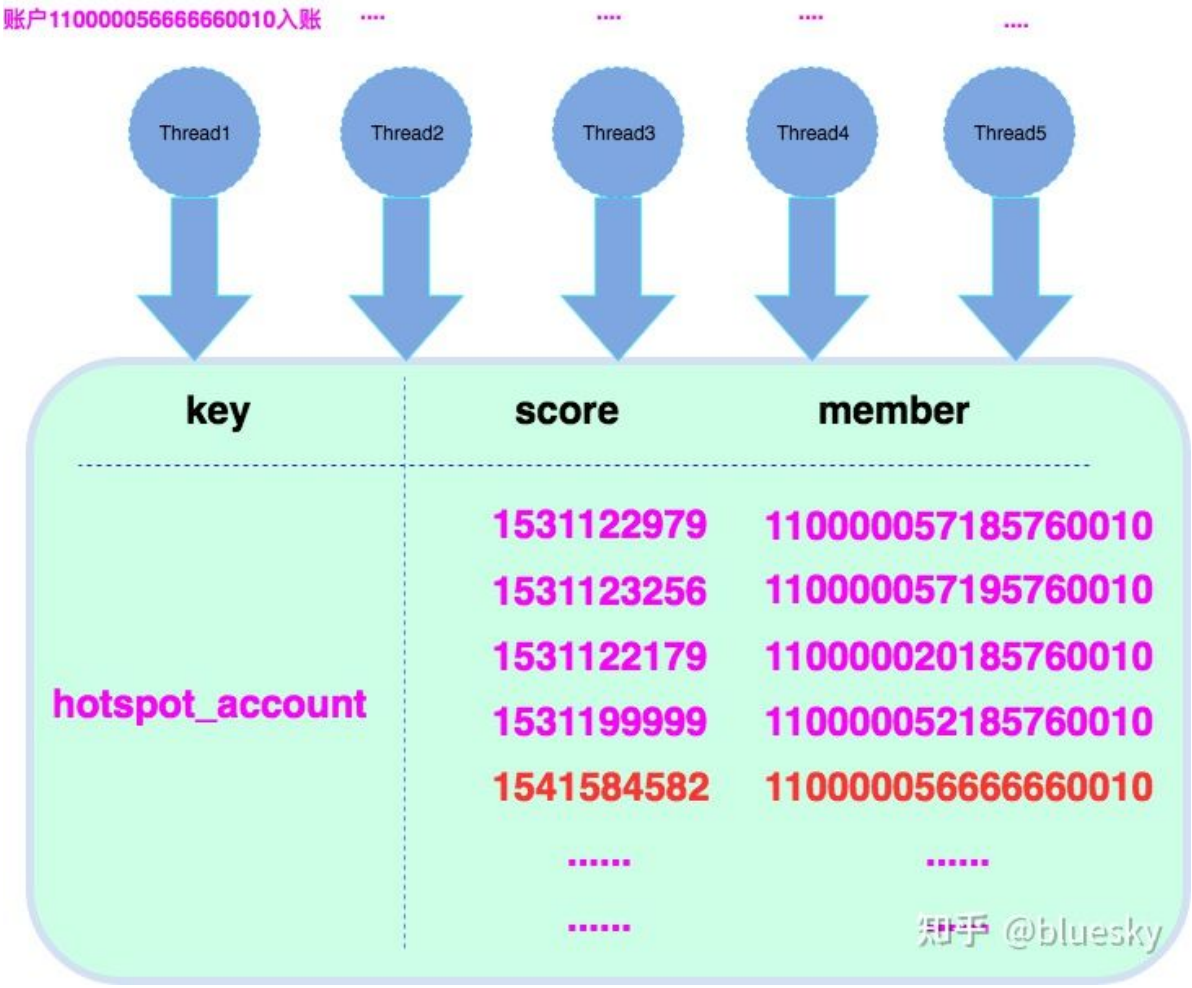
（3）新增【redis】数据结构【SortedSet（有序集合）】 key为【hotspot_account_currentbalance】下面会对这个数据集合称为【缓存余额】



其中：
score为当前账户可用余额，【热点账户新操作流程之前需要将数据库中热点账户的数据同步到hotspot_account_currentbalance中】
member为账户ID

到此，前期准备工作已经全部结束。

当账户金额充值新增时:
1:记录redis操作记录【hotspot_account】



如图所示红色数据部分，当账户**110000056666660010**入账时,插入或更新数据，member=**110000056666660010,**score为当前时间戳（秒）。

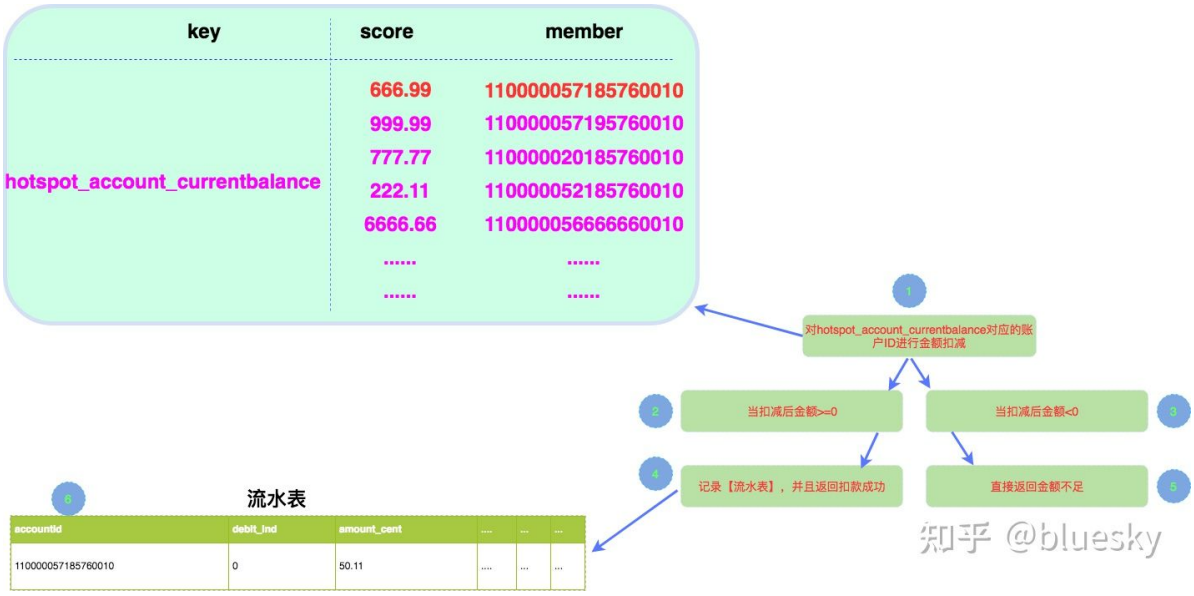
ps:操作指令【**ZINCRBY key increment member**】，当 key 不存在，或 member 不是 key 的成员时， ZINCRBY key increment member 等同于 ZADD key increment member 。

2：新增【流水表】，设置入账状态为未入账

当账户金额扣减时:

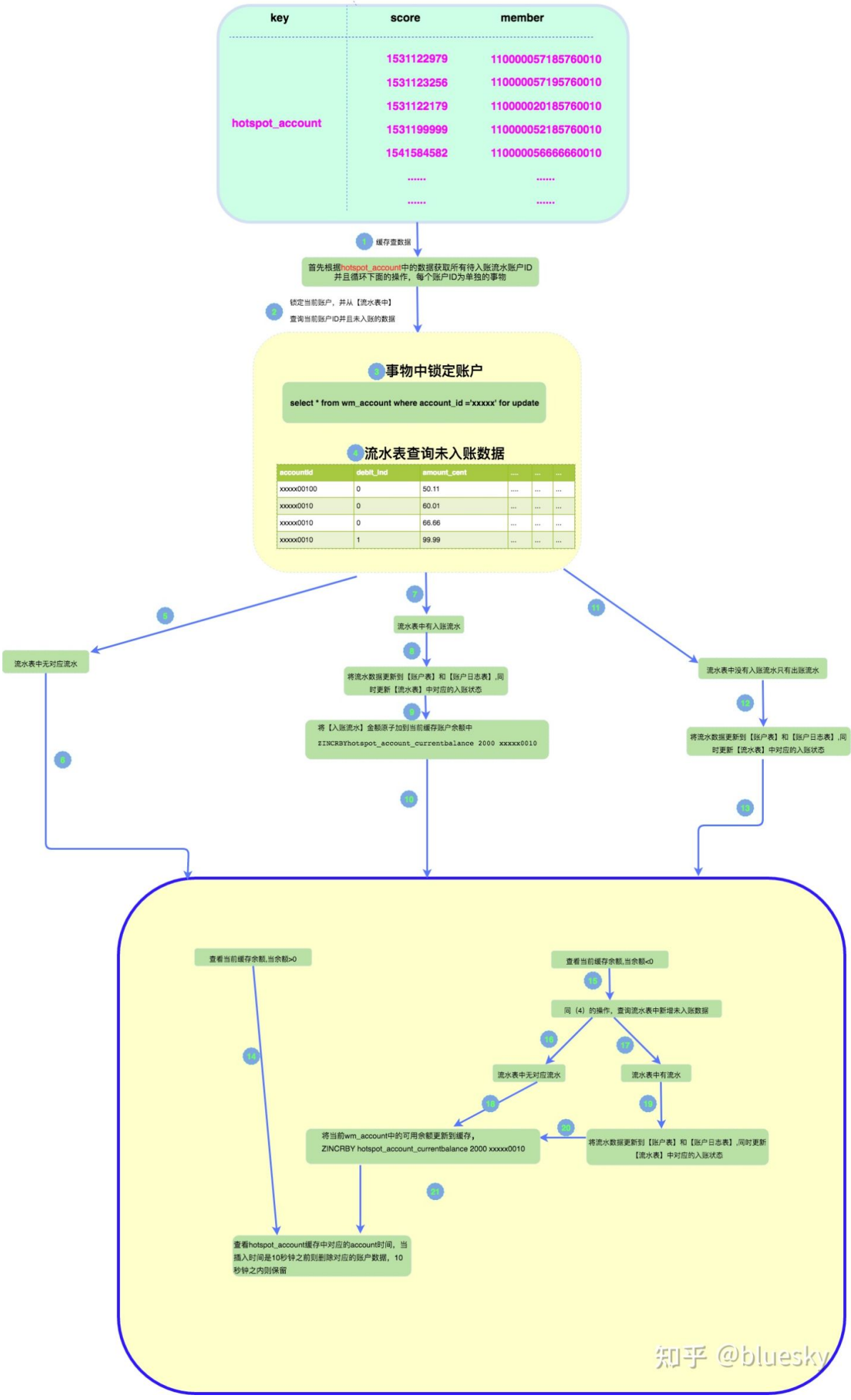
1:同金额充值相同首先记录redis操作记录【hotspot_account】。

2:直接对缓存**hotspot_account_currentbalance**对应的金额进行扣减。



3:定时任务

定时任务的作用是将流水表的数据更新到【账户表】，和【流水明细表】，并且设置【流水表中】数据已入账，同时要将新入账数据流水到更新【hotspot_account_currentbalance】中的可用账户余额，让扣减操作得以继续进行。以下操作流程：


key
score
member
hotspot_account
1531122979
110000057185760010
1531123256
110000057195760010
1531122179
110000020185760010
1531199999
110000052185760010
1541584582
110000056666660010
1 缓存查数据
首先根据hotspot_account中的数据获取所有待入账流水账户ID
并且循环下面的操作，每个账户ID为单独的事物
2 锁定当前账户，并从【流水表中】
查询当前账户ID并且未入账的数据
3 事物中锁定账户
select * from wm_account where account_id ='xxxxx' for update
4 流水表查询未入账数据
accountId
debit_ind
amount_cent
xxxxx00100
0
50.11
xxxxx0010
0
60.01
xxxxx0010
0
66.66
xxxxx0010
1
99.99
5
流水表中无对应流水
6
7
流水表中有入账流水
8
将流水数据更新到【账户表】和【账户日志表】,同
时更新【流水表】中对应的入账状态
9
将【入账流水】金额原子加到当前缓存账户余额中
ZINCRBYhotspot_account_currentbalance 2000 xxxxx0010
10
11
流水表中没有入账流水只有出账流水
12
将流水数据更新到【账户表】和【账户日志表】,同
时更新【流水表】中对应的入账状态
13
查看当前缓存余额,当余额>0
14
查看当前缓存余额,当余额<0
15
同（4）的操作，查询流水表中新增未入账数据
16
17
流水表中无对应流水
流水表中有流水
18
19
20
将当前wm_account中的可用余额更新到缓存，
ZINCRBY hotspot_account_currentbalance 2000 xxxxx0010
将流水数据更新到【账户表】和【账户日志表】,同时更新
【流水表】中对应的入账状态
21
查看hotspot_account缓存中对应的account时间，当
插入时间是10秒钟之前则删除对应的账户数据，10
秒钟之内则保留
知乎 @bluesky

修改于2019年10月24日:
在第15步的时候目前版本在极端情况下会出现一个问题，当定时任务发现了当前缓存余额<0时，会再从流水表中把数据重新查询一遍入账，这里有一个小小的问题，在查询过程中可能会有部分线程没有入账到数据库，这样会出现超扣的问题，也就是说线程1在
扣减redis的时候成功了，但是没有insert到流水表，这个时候又有一个线程2扣减redis的线程扣负了，这个时候定时任务发现余额为负，不应该直接同步余额正确的做法是加锁等待扣减操作流程执行完毕，由于加锁的复杂性，所以这里采用一个简单的办法，由于扣减redis和mysql操作基本都是瞬时的，所以直接sleep 5 s即可，这个时间足够发生full gc等一些其他未知因素的耗时了，当然感兴趣的小伙伴也可以加锁去自己拓展实现，会有一定性能上的影响，最后感谢群里小伙伴@Token指出的问题。

方案设计要点说明Q&A

1. 延迟入账,金额扣减和充值都以流水的形式记录，由定时任务来延迟入账，可能会出现已入账数据，但是定时job还没有入账而引发金额不足的情况，但是会在下次定时job启动之后入账，业务可接受

2. 缓存账户余额来判定当前余额是否充足，只有缓存余额充足才会记录【流水表】，可以确保金额不会为扣成负,反过来也就是说只要是【流水表】中存在扣减流水，那么此时金额一定充足,redis是单进程单线程的，所以redis的zincrby扣减操作是线程安全的。

3. 精确入账，所有的入账出账操作都要先经过缓存【hotspot_account_currentbalance】，所以当定时任务启动的时候只会入账缓存中记录过的数据,避免循环所有的热点数据，增加定时JOB执行效率

4. 【为什么定时JOB入账后要判断当前缓存中余额是否大于0】,当定时任务同步【流水表】和业务数据表之后需要把入账的数据同步到缓存中,并且当【流水表】中有充值流水的话需要把充值金额原子加入到缓存余额，来同步db和缓存这余额
（1）当缓存金额大于0的时候，说明目前缓存中金额充足无需同步，此时缓存中余额可能比数据库中的小，不过对业务无影响
（2）当缓存金额小于0的时候，说明目前金额不足了需要将数据中的余额同步到缓存余额中，但是这个入账过程中可能会有其他成功插入到【流水表】的流水，但是到目前为止，肯定不会有扣减的数据再插入到【流水表】中了，因为金额已经为0，所以这个时候大胆放心的再冲【流水表】中看看是否有并发的数据，如果存在再更新一下业务表数据，然后把当前account中的可用余额直接set到缓存中的可用余额即可

5. 设计方案中依靠zincrby的原子扣减操作可以保证缓存中可用余额小于等于DB中实际可用余额

6. 【定时JOB从流水表中入账业务表数据成功，并且同步缓存中余额后为什么最后一步要删除10秒钟之前的缓存操作日志数据】，这是因为如果10秒钟之内都不存在当前账户操作的记录那么就可以认为在定时任务对当前账户进行操作的过程中没有并发对这个账户的充值和扣款操作，所以可以删除掉缓存中对这个账户ID的操作记录，【可以确定的是我们的入账出账不会超过10秒】，反过来，如果误删除掉缓存中对这个账户的操作记录，也不会对之后的定时JOB有任何影响，因为所有的入账数据都保留在【流水表中】，这个删除10秒钟的操作意在减清定时JOB的工作量而已。

谈谈热点账户(一)

经常做银行系统或者支付结算系统的同学应该对热点账户问题都很熟悉，今天我们就谈谈热点账户。

在我关于微信红包的那个回答中，我提出过"对于较大型的支付公司，如果银行结算款实时入账，对于那个银行账户而言会有热点账户风险"。

那么什么是热点账户呢？我尝试简单地把我的理解和大家分享一下，希望对大家有帮助。

我们在银行、支付公司的账户，说到底就是存在数据库中的数据。

他们可能长这样（实际上不是长这样的，只是为了简单说明）：

银行卡号	户名	余额
62260902******	天顺	35.98

假设说，今天天顺的银行账号向匿名专家汇款0.74元，那么完成这笔转账后，天顺银行账户余额变成了35.24元，这个时候数据库里的数据变成了：

银行卡号	户名	余额
62260902******	天顺	35.24

在这个过程中，数据库做了哪些操作呢？

首先，数据库需要对这条数据中的余额字段，做一次更新，减少0.74元。

在更新前，他需要对这条数据进行锁定（lock），锁定完成后，对其余额进行更新，然后释放锁定（unlock）

有同学会问，为啥要搞那么复杂呢？直接更新余额不就好了。

其实这么做是为了防止其他业务对这个余额也同时进行操作，避免不必要的数据错误，保证同一时间只有一个业务主体在对余额进行操作。

那么问题来了，假设我们数据库对数据进行一次更新需要使用10毫秒的时间，业务量小的话还行，对于一分钟要处理几千甚至上万次的话，数据库就会来不及响应，最后崩溃。

对于较大型的支付公司和机构，如果他的银行账户在每次支付成功时都实时更新余额无异于给银行带来了十分大的系统压力和数据库瓶颈，而这个账户也就是我们常说的热点账户。

即便是到IT技术如此发达的今天，热点账户依然是困扰很多银行业务系统的顽疾。

那么，我们是否对于热点账户完全束手无策了呢？

当然不是，聪明的程序员和架构师们基于很多业务场景的特殊性做了很多变通和优化，让业务可以在尽量不受热点账户影响的情况下顺利完成。

第一种方法叫做汇总入账：

银行在T日业务完成日切后，将T日发生的所有成功交易进行统计，计算出一个总账，一笔汇总入账到指定的结算账户。

这也是目前最为广泛采用的一种结算模式，基本上主流的收单系统都是沿用这套方式。

这套方案的优势很明显：数据库压力很小，日间所有交易全部是insert进表（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），日终跑批只要sum出一个结算总金额，和内部户的会计流水比对无误，就可以明确日间交易产生的债权债务总账，一笔入账结算到指定账户。

这套方案的劣势也很明显：结算账户的资金并未实时结算更新，T日的交易款要到T+1日才能结算到账，但是，既然是收单业务，这点资金时效性相对业务可用性相比还是可以接受的，何况已经成为业内标准了，所以基于收单的业务场景目前仍然沿用此模式较多。

现在很多银行的核心账务处理能力实际上已经能够支持较大并发，但收单结算的模式还是保留了下来。

请大家注意一点：汇总结算模式的存在，热点账户是其中一个原因，但不是全部原因。写这篇文章不是要告诉大家，银行收单业务是T+1结算，完全是因为热点账户造成的。汇总结算只是正好在某些条件下，缓解了热点账户问题而已。

留给大家一个思考：大家不知是否发现，微信发红包，经常是红包已经看到了提示你公司拿了X元红包，

请到零钱查看，但跑到零钱后发现居然没有。
过了好大一会，零钱里才增加一条入账流水，显示红包金额，在这个过程可能发生了什么？

谈谈热点账户（二）
上次我们谈了热点账户，介绍了使用汇总记账的结算方案来规避日间单账户热点。
今天来看看缓冲记账。
缓冲记账，顾名思义是一种将实时同步的记账行为进行异步化，以达到记账实时性和系统稳定性之间平衡的记账手段，说到底就是削峰填谷。
具体他是怎么做的呢，我们可以举个例子看看。
假设有这么个情况：

- 支付宝和工行合作推出“实时转账到卡”产品，用户从支付宝端发起转账后可以立即看到账户资金入账。
- 支付宝在工商银行的银行收付账户里有一定的人民币备付金用来用户提现和转账使用。注：此账户用于流出代发时就是标准的热点账户。
- 工商银行的单账户处理阈值假设为30笔/秒，24小时不间断服务。（日总处理量2592000笔）
- 支付宝为了能够让用户尽快提现到账，要求工商银行开独立的单笔实时银企接口用于资金代发，不使用批量模式。
- 每当用户报送一笔转账或者提现交易，支付宝体内不hold，直接同步调用单笔实时代发接口要求工行处理。（言下之意，这套服务的处理阈值支付宝不理会，有交易就报送）
- 工行在这个代发过程中，不进行日间垫资。
- 支付宝一天在工行的总转账加提现笔数为100万笔，日间峰值可以达到60笔/秒

从上面的条件来看，两家公司合作推出这个产品，就是要让用户感受到秒级的转账体验，支付宝方面不会由于接口并发限制而挂起用户的交易，这个交易都是同步调用同步返回。换言之，用户看到的产品体验是：手机客户端输入转账信息，点击确定，支付宝转几秒菊花，返回转账成功/失败（没有中间业务状态）。
我们可以看到，支付宝全天在工行的代发账户总笔数其实是足够工行业务系统串行消化的，但支付宝的日间峰值却会在某几个瞬间超过工行这套接口的承载能力。
聪明的工行工程师们和产品经理会怎么办呢？
大家想到一个办法，对于日间报送的转账请求，工行只校验账户合法有效性，只要账户一切正常就直接返回转账成功，反之则转账失败。然后在后台做一个队列，把这些成功的交易排个队，以一秒30笔的速度进行处理。
支付宝报送的交易低于30笔/秒的时候，工行系统几乎还是实时地处理了账务变更，而当交易大于30笔/秒的时候，工行就先返回转账结果，把账务处理丢到队列中，等并发量不大的时候慢慢消化，对用户来说感受到的体验还是几秒钟就转账成功了，看上去皆大欢喜。
那么问题来了，这个缓冲记账的方案有些什么短板？
第一个短板，业务量明显过大时缓冲记账可能会带来业务问题
不知道大家是否记得我第一篇文章提到的微信红包？
在我当时抢微信红包的时候发生过一批红包中，部分几个实时入了微信钱包，部分几个在一个多小时候才入账。
假设微信使用的是缓冲记账的方式，那么这种情况是可以解释的。（当然实际上他们也可以使用了部分峰值时段交易事后批量入账的方式；或者我真的是运气不好，他们账务没问题只是内部调单了，一个多小时才恢复……）
我们可以想象，假如账务本身处理能力与实际业务量不在一个量级上，那缓冲记账的延迟会给业务造成一定困扰，比如用户会觉得很奇怪，明明你告诉我转账/红包领取成功了，但为啥我的账户里看不到呢？
上面工行和支付宝的例子里，支付宝只是在某几个高峰时间点需要缓冲记账来削峰填谷，一旦支付宝的日间交易量暴增，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，用户肯定会马上发起一连串投诉……
第二个短板，异步化带来的各类异常情况

最明显的就是工行的这个收付户，业务上是不允许日间垫资的，假设支付宝在工行的备付金完全够用的话缓冲记账自然还好说。一旦支付宝报送代发交易的时候并发量很大，工行都返回处理成功了，真扣钱时发现余额不足傻眼了……咋办？
另外还有极端情况：用户的卡在发起转账时还是正常状态，1秒后被注销了，由于缓冲记账了所以当时告知用户转账成功，实际执行记账时失败了。
以上两个问题决定了单纯的缓冲记账无法成为解决热点账户问题的银弹。但缓冲记账在某些特定场景还是带来了很大的价值，不能全盘否定。牺牲记账的时效性换取峰值时的稳定性，在很多场景依然有用。
tips：很多同学会说，既然缓冲记账额外还带来那么多麻烦，那带来价值太低了，何必要使用呢？
实际上，你现在所使用的PC，手机都在使用着“缓冲记账"类似的思想在解决问题。虽然你可能不信，但一颗cpu同时只能做一件事。你在终端向它输出的各种要求，其实他在某一个足够小的时刻维度里，都只在执行一件事——但它和它的小伙伴在能力范围内全部受理了你的指令，不停地中断手头的事情去满足你的要求。这种中断+继续的过程人类根本感知不到。当你的要求太多以至于他根本无法及时处理时，你就会感觉到”玩游戏有点卡“，当卡到一定程度后这种体验就不再被你接受，而你要做的就是换一个更强大的电脑来处理你的需求——而他的运行机制本身和那台前任是一摸一样的。
回到话题本身来，希望大家明白一个道理：
我们不光要关心缓冲记账是如何做的，更应该关心的是他可以运用在哪些场景，哪些领域。从而扬长避短，满足需求。

热点账户高并发解决方案

背景:2018年初，直播答题风靡全国。来的快，去的也快，抖音突然崛起，具有了挑战微信的实力。

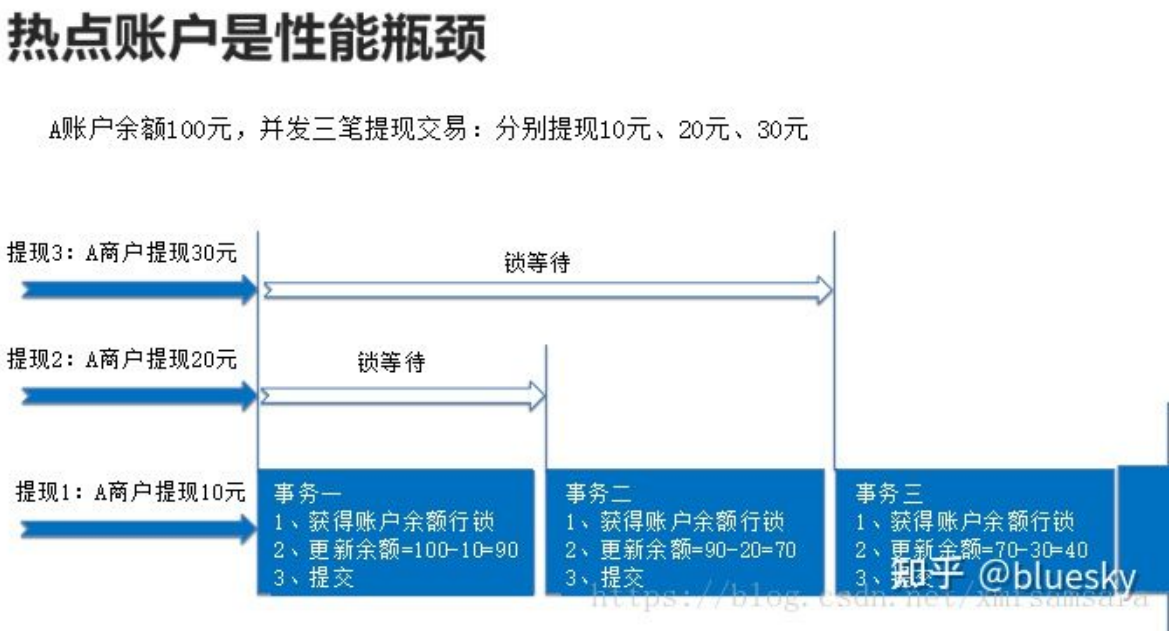
我司与头条合作，负责头条的红包雨业务。头条要求，200tps，最后最高达到140tps。

自此之后，公司开始了账户优化。

此前已经对第三方支付的账户进行了详细的描述。不再赘述。

账户就是余额加上流水（账户明细）。支付行业对数据的错误是零容忍的。

下面直接开始。

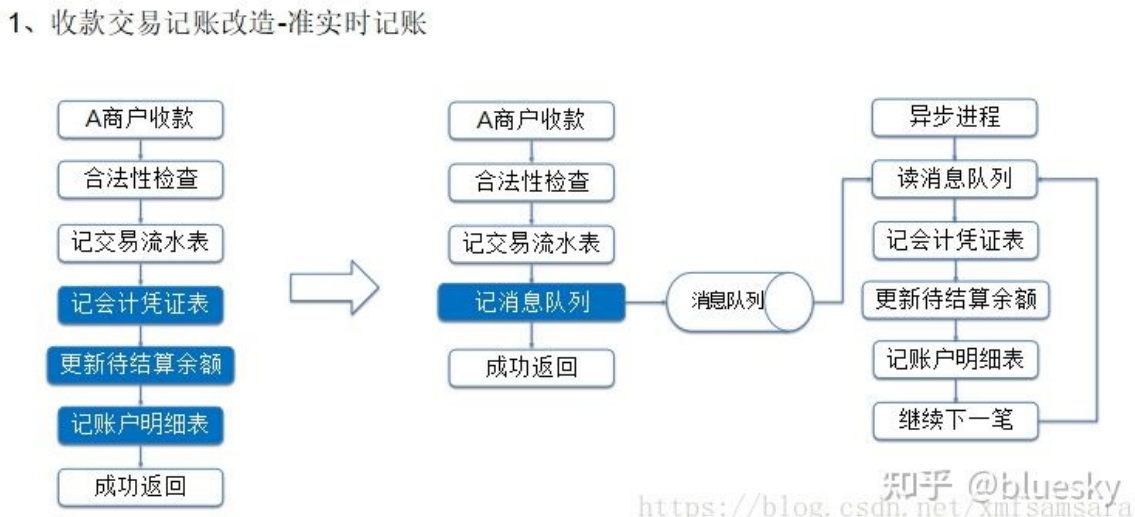


数据库行锁这里是你避免不了的。一次事务的时间是9ms的话，你的tps天花板最高就是110

所以第一招就是 减少事务时间，具体方法就是，减少事务的数据库操作，讲一些不太重要的操作进行分解。

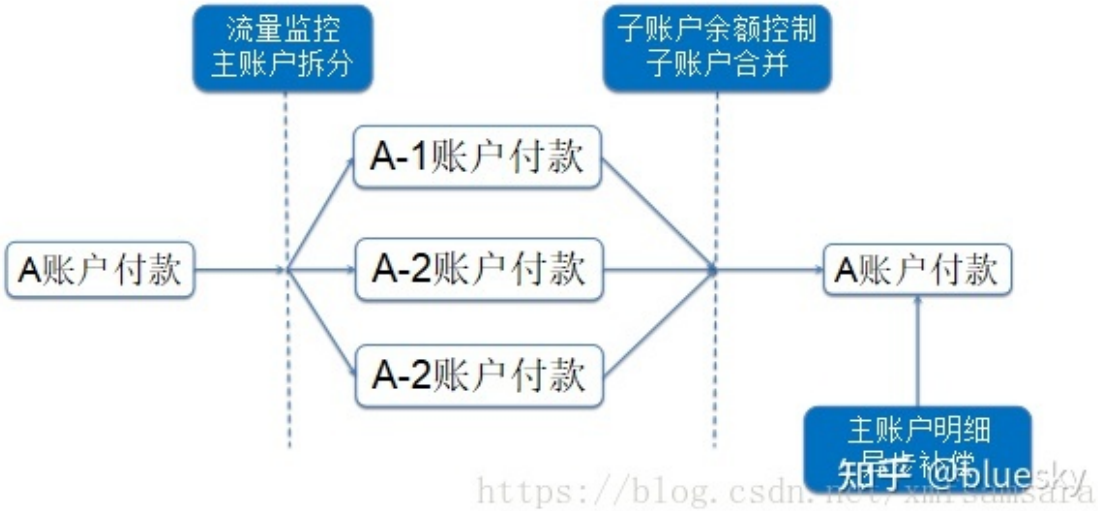
第二招 优化网络,,其实就是减少了应用到数据库的交换机，给高并发应用配置资源，没啥意思，掠过。

第三招 异步

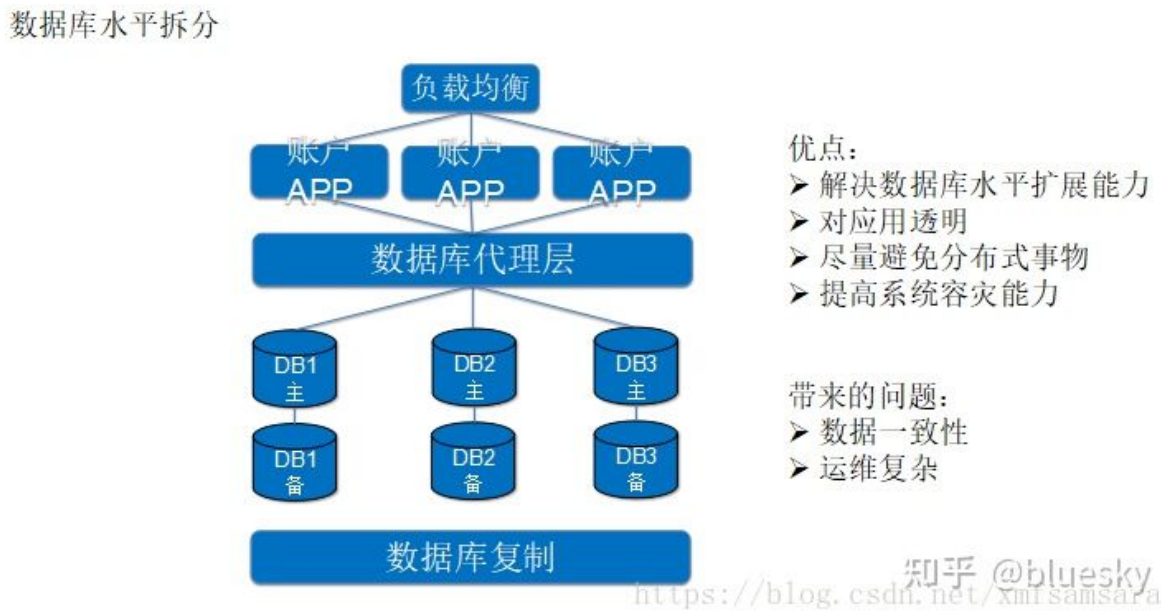


将这三点异步了，反正流水记录了，丢不了，有对账心里有底。下图同理

最后的大招，我们知道虽然进行了这些操作，但是对性能没有达到指数级别的提高。 事务持锁时间永远是你的瓶颈。这个就需要变个花样来打破。 将一个热点账户拆分为多个热点账户。 头条发红包的账户只有一个,我们将其拆分为三个虚的账户，哪一个没钱了，就销毁它，这种成本账户会监控其余额，一般不会没钱的，没钱就找头条充值。



当然还有分布式数据库，这个不是我擅长的掠过。



交易系统热点账户问题（一）

一、热点账户

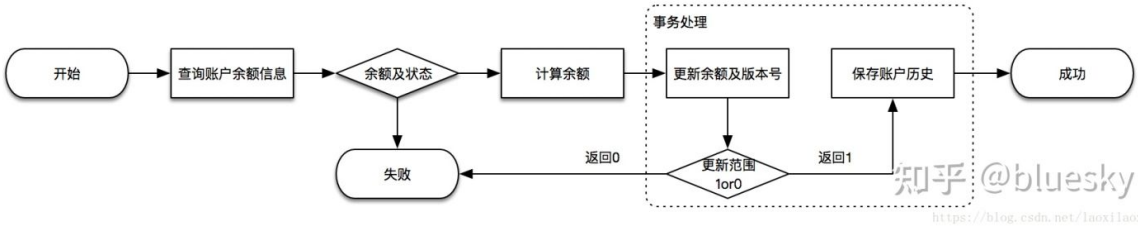
热点账户就是高频进行扣款、入账的账户，也就是热点账户该条数据为热点数据，会被频繁更新。一般热点账户分为两种，一种是频繁扣款的热点账户，另外一种是频繁入账的热点账户。

二、热点账户常见问题

1、性能瓶颈问题
2、数据库压力问题
3、成功率问题

三、纯修改余额方式及其特点

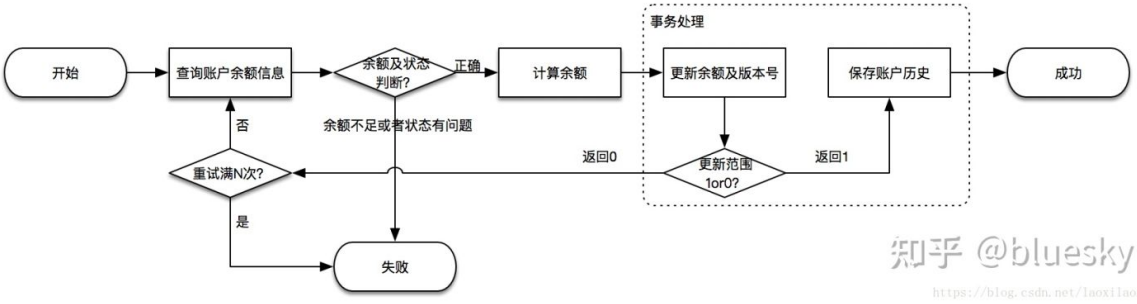
\1. 乐观锁



操作方式：
查询账户数据：
SELECT BALANCE, STATUS, VERSION, ... FROM ACCOUNT WHERE ID = ?
计算余额:
POST_BALANCE = BALANCE + AMOUNT
或者
POST_BALANCE = BALANCE - AMOUNT
更新账户余额：
UPDATE .... BALANCE = POST_BALANCE,VERSION = VERSION +1 WHERE ID = ? AND VERSION = ?
更新返回1，更新成功，返回0，更新失败，需抛出异常，回滚事务
插入账户历史:
INSERT ...

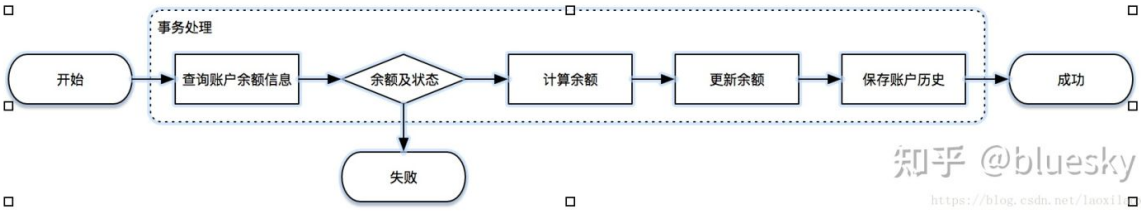
优点：
不会存在阻塞，响应时间快；
数据库没什么压力；
在内存里可以完成很多复杂操作；
缺点：
成功率不高，真的存在并发时，失败的请求比较多；
有效的性能依然不高；

一般应对方案：
采用重试的方式，立即重试三次，以提高成功率



个人看法：
这种重试在真正的有量的时候基本没啥作用，相反会徒增数据库的请求量，鄙人觉得这种重试只能解决请求量较小的时候的并发，比如突然同时进来两笔同一个账户的请求，处理失败的话进行重试是可以解决问题的；但是一瞬间进来200笔，甚至更多的话，这种重试没啥作用了。

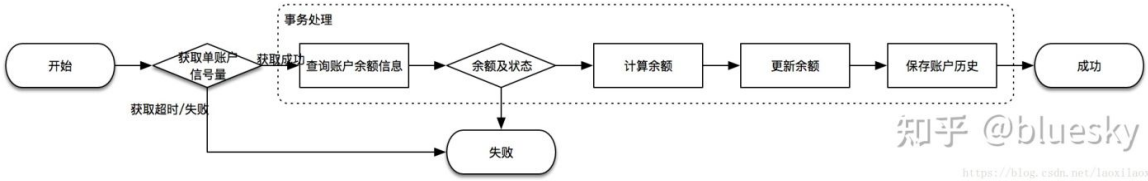
\2. 悲观锁



操作方式：
查询账户数据：
SELECT BALANCE ... FROM ACCOUNT FOR UPDATE WITH RS
计算余额:
POST_BALANCE = BALANCE + AMOUNT
或者
POST_BALANCE = BALANCE - AMOUNT
更新账户余额：
UPDATE .... BALANCE = POST_BALANCE,VERSION = VERSION +1 WHERE ID = ?
插入账户历史:
INSERT ...

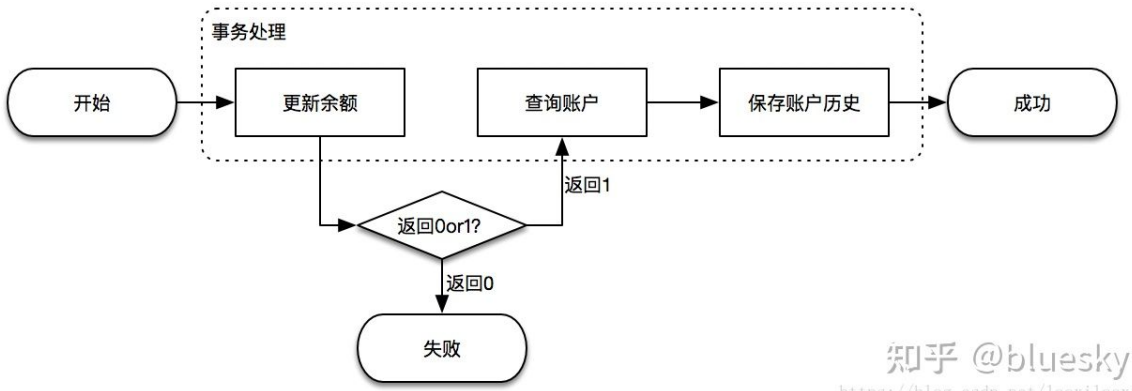
优点：
成功率高；
性能好；
在内存里可以完成很多复杂操作（余额签名）；
缺点：
会存在阻塞，响应时间长；
数据库压力大；

一般应对方案：
采用信号量做热点账户资源使用限制，可以控制数据库压力，为数据库分压，且保持在一个客观的性能水平。



个人看法：
这种方式能解决大部分的热点账户问题，也是本人之前采取的方式，不过偶尔会存在的超时问题也仅仅是一两笔，其余的都会被信号量拒绝了。

3.数据库行级锁1



操作方式：
更新余额:
入账:
UPDATE BALANCE = BALANCE +AMOUNT WHERE ID = ?
扣款：
UPDATE BALANCE = BALANCE - AMOUNT WHERE ID = ? AND BALANCE > AMOUNT
读取账户数据:（读取数据是为了在账户历史插入的时候保留发生后余额）

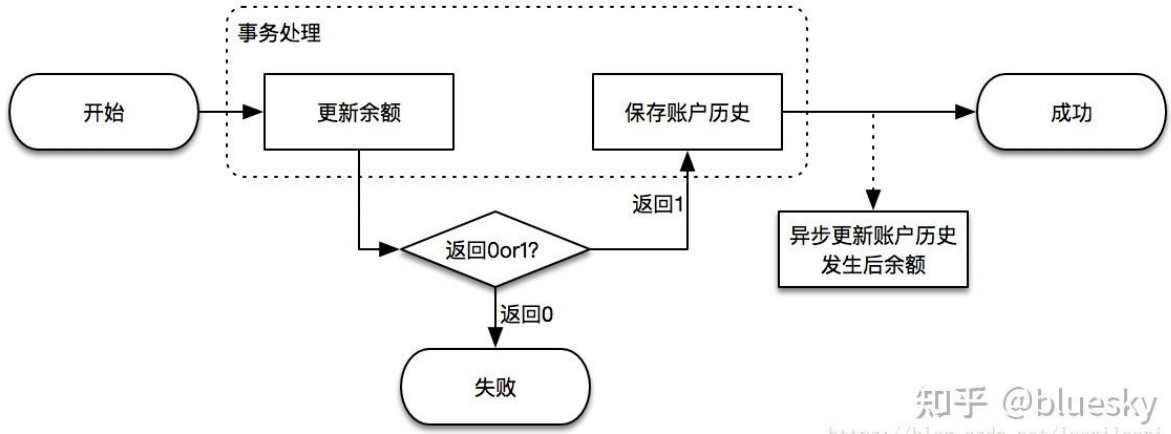
SELECT * FROM ACCOUNT WHERE ID = ? WITH CS
插入数据
INSERT ...

优点：
成功率高；
性能好（相对于2）；
数据库压力也会小（相对于2）；
相应时间也小（相对于悲观锁）；
缺点：
一些复杂的操作无法在内存完成了（余额签名）

一般应对方案：
复杂的操作异步化，延迟也就是毫秒级别，或者舍弃签名

个人看法：
这种方式与悲观锁相比好了很多，数据库压力小，性能高了，许增加上单账户限流或者信号量，防止单账户暴涨的量把数据库压爆。

4.数据库行级锁2



操作方式：
更新余额:
入账:
UPDATE BALANCE = BALANCE +AMOUNT WHERE ID = ?
扣款：
UPDATE BALANCE = BALANCE - AMOUNT WHERE ID = ? AND BALANCE > AMOUNT
插入数据
INSERT ...
异步更新发生后余额：（或者根据业务情况不需要该步骤）
UPDATE ACCOUNT_HISTORY SET POST_BALANCE = ? WHERE ACCOUNT_ID =?
优点：
成功率高；
性能好（相对于3）；
数据库压力也会小（相对3）；
相应时间也小（相对3）；
缺点：
代码复杂度高，需要异步化一些处理；

个人看法：
一些非核心部分的修改及操作，不需要就去掉，需要的话那就异步处理下。

交易系统热点账户问题（二）

一、概述

本篇着重介绍一下从业务层面的热点账户的经验与实践，修改余额的几种方式见上篇介绍[交易系统热点账户问题（一）]

二、业务场景分析

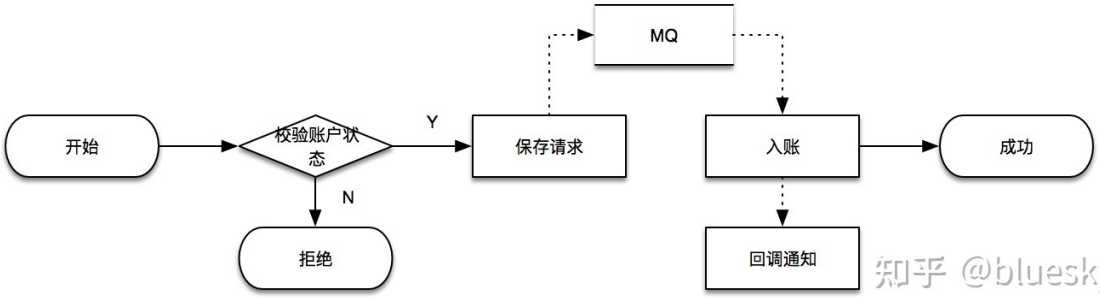
可将业务场景划分为如下：

高频入：B端收单账户（结算账户），业务中间账户（业务处理时记录在途资金）

高频扣：B端代扣代发账户（B端支付账户），B端手续费账户（用于做收支分离）

三、业务角度热点账户实践及其特点

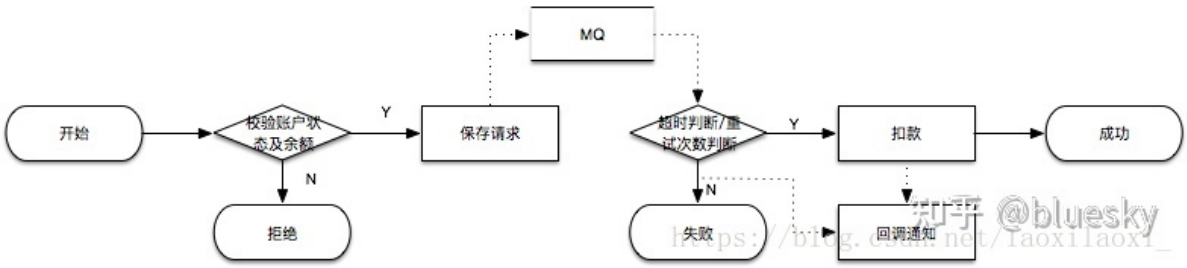
1.异步削峰入账



适用场景：

1.适用B端收单账户与业务中间账户，一般B端收单账户资金是在清结算场景时会扣除，不会影响正常的业务，且MQ处理一般也会在毫秒级别完成，高并发时延迟也是在秒级，所以异步入账对商户几乎无感知。

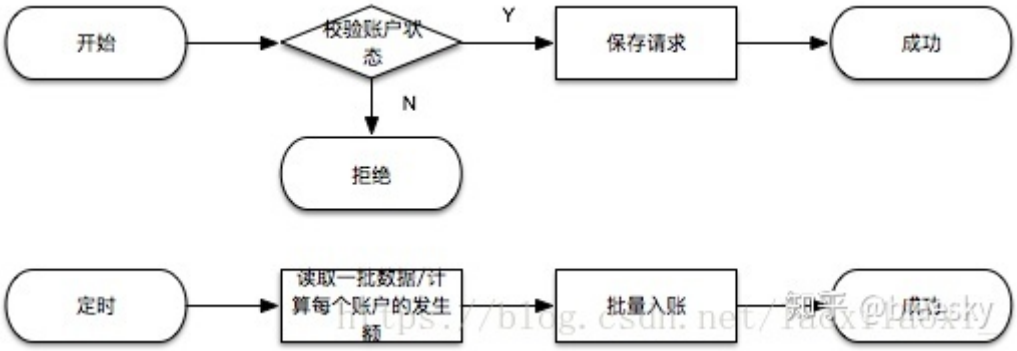
2.异步削峰扣款



适用场景：

1.适用B端代付代发账户以及手续费账户，一般B端高频扣款对相应时间都有一定的要求，且存在余额不足这种不可抗拒的失败因素，所以采用优先判断余额（脏读余额），拒绝掉大部分账户余额不足的请求，其次是异步增加超时以及重试次数的判断，确保请求的有效处理时间保持在一定范围里（超时时间及重试次数可由业务传入）。

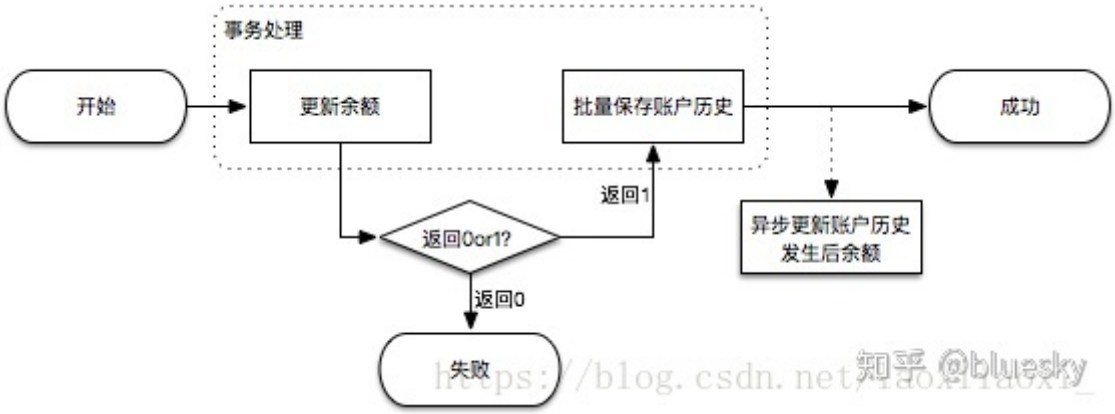
3.批量入账（缓冲入账）



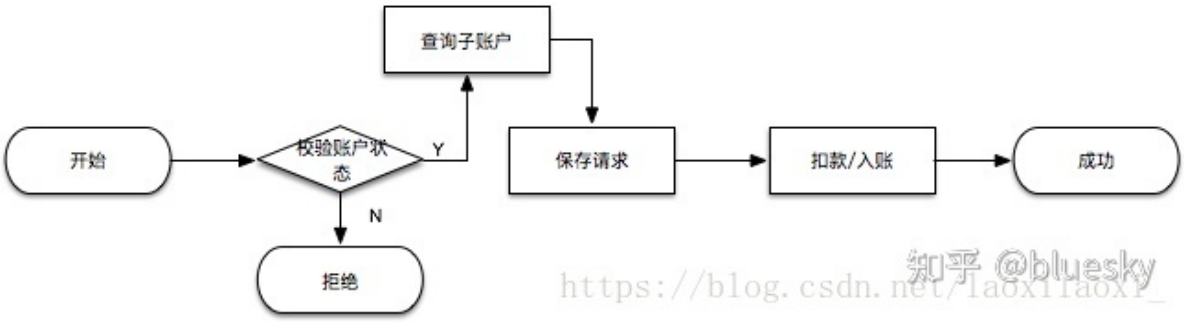
适用场景：

1.适用B端收单账户与业务中间账户，一般异步削峰入账可解决大部分高并发问题，但有一些超级B端商户的交易量超级大，可采用该种方式应对。同样，也不会影响该类商户的清结算。

其中，批量入账需新增一种修改余额的方式，一次更新余额，批量插入账户历史



4.子账户（拆分多账户）



适用场景：

1.适用与所有账户的扣款入账。

但是其中存在一些问题解决起来需要多写点代码:

①扣款入账实际操作的是子账户（就是一个单独的账户，只是存了与原本账户之间的关系），主账户不存放余额，余额都是存放在子账户，所以查询余额时，需要做特殊处理（求和子账户）。

②因为分散到多个子账户去处理扣款入账，那么账户的发生后余额就会是个问题（如果不需要发生后余额的话那就很赞了），扣入账产生的账户历史都存放的是子账户的发生后余额（或者不存），要是必须要的话，那就异步计算生成。

③多子账户扣款时会存在子账户余额不足的问题，但实际所有子账户的余额总和可能是充足的，所以需要在余额不足时做子账户资金归集。且若为代付代发的场景，那么对其进行充值时，需要做充值资金拆分，去加到每个子账户中。

四、其他角度方案

单热点账户常见问题及解决方案在上文已经说的差不多了，但是实际场景中并非为单个热点账户。例如，账户基数为1000W，其中高并发账户有1W-5W，在正常扣入账时，这些热点账户在数据库上产生的锁等以及占用的数据库连接都是共享资源，会相互影响，以至于性能很难提升。

分库分表，是个很常见也很好的解决方案，下节单独详说一下

参考：

架构设计-热点账户 - xuwc - 博客园

https://www.cnblogs.com/zyl2016/p/9928567.html

https://zhuanlan.zhihu.com/p/19960934

https://zhuanlan.zhihu.com/p/19998189

https://blog.csdn.net/laoxilaoxi_/article/details/80963905

https://blog.csdn.net/laoxilaoxi_/article/details/81120785?utm_source=blogxgwz2

https://blog.csdn.net/xmfsamsar