从0到1：饿了么风控计数服务是如何炼成的

作者/分享人：饿了么技术社区

查看本场Chat

向作者提问

引言

2017年4月份从饿了么正式进入多活领域开始，也预示着饿了么业务开始迈入下半场，此时风控团队面临着严峻的挑战，风控需要在事前、事中、事后进行全方位的防御.

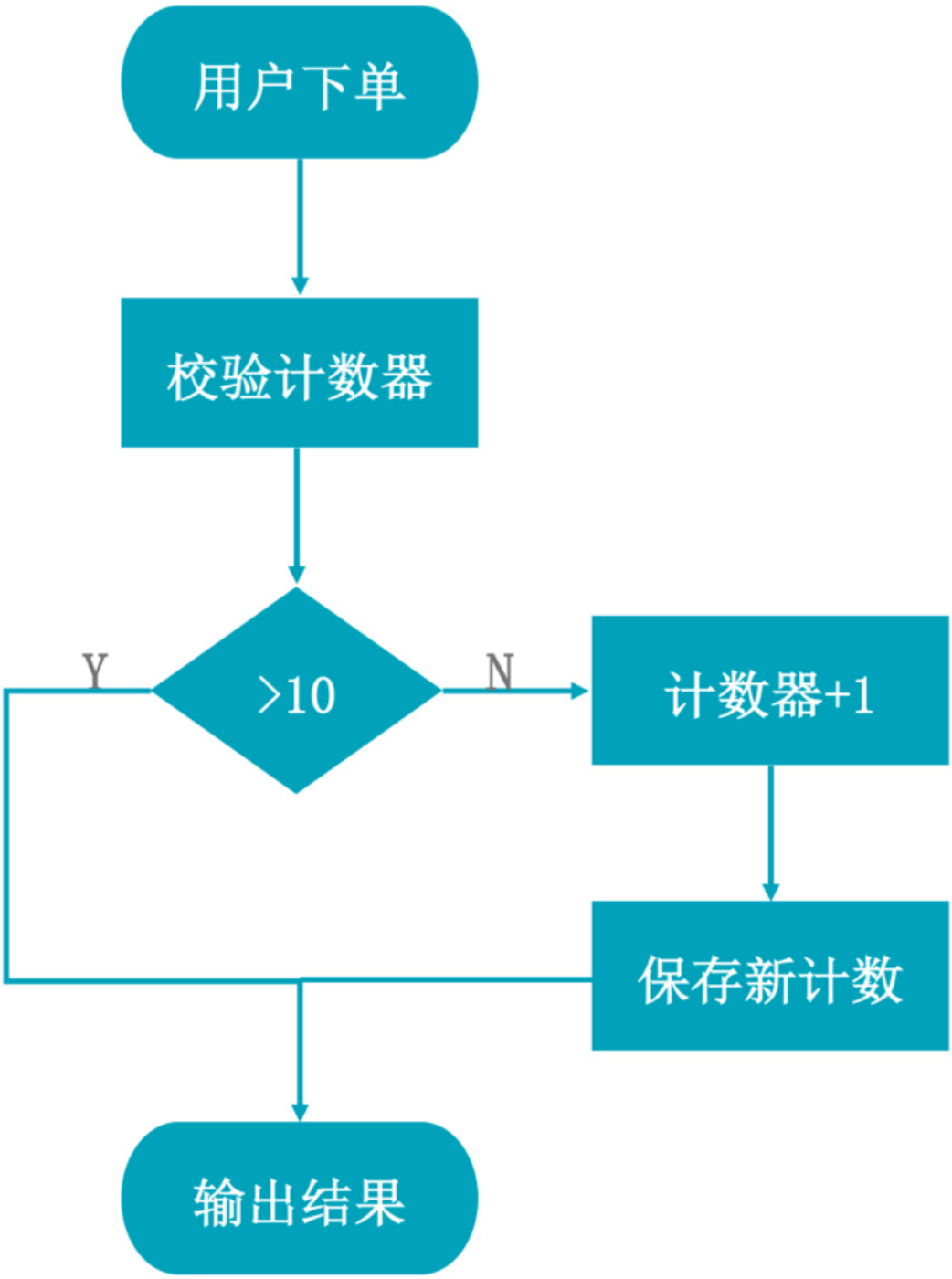
而计数器的业务几乎贯穿了整个风控的需求，规则根据计数器拦截用户风险动作，运营系统需要根据计数器分析出商家、用户的刷单行为。首先，各个系统充斥着大量重复相同的计数器代码，其次开发团队对于这样重复劳动除了感觉疲惫，还有点缺乏技术含量，最后这样的开发成本与模式，并不能快速满足风控业务的需求，此时一个通用的计数器服务迫在眉睫。

案例

首先回顾一下风控使用计数器的一个场景，让大家了解计数器在风控的作用。

限制用户下单数

假设每天用户在饿了么最多下10单，那么用户在下第11单的时候将会被风控拒掉，此场景的校验流程如下：



在这个场景中，涉及到计数器的部分包括如下几部分：

1. 获取计数器的逻辑
2. 设置计数器的逻辑

3. 计数器入库

方案：硬编码

由于历史原因，风控老计数器采用了硬编码的方式，伪代码如下：

```
if (getUserOrderCount(userId)>10) {
    return "Reject";
}

setUserOrderCount(userId);
saveUserOrderCount(userId);

return “Accept”;
```

优点：

当计数器种类较少，改动不频繁的时候，开发效率高。

缺点：

- 计数器改动成本高：例如改动计数器的存活周期，都需要走一遍发布流程 。
- 当计数器种类较多时，维护性差，大量重复劳动。

思考计数器新设计

在思考新设计之前，我们先来总结一下老计数器的几大缺点:

第一：重复劳动

之前计数器的相关逻辑，各个系统都进行了相应的开发，这段逻辑大部分是相同的，是属于重复劳动的部分。

第二：key的生成规则需要暴露给其他系统

如果A系统创建了计数器counter1，此时B系统和C系统需要使用计数器counter1，必须得知道A系统创建counter1时候 key的生成规则。

第三：计数器不可配置

之前计数器是硬编码在系统中，这就意味着每次变更，例如更改计数器的生命周期和统计方式，都需要重新上线，而每次上线都需要经过alpha到生产一系列过程，耗时比较长，灵活性不够高。

计数器模型

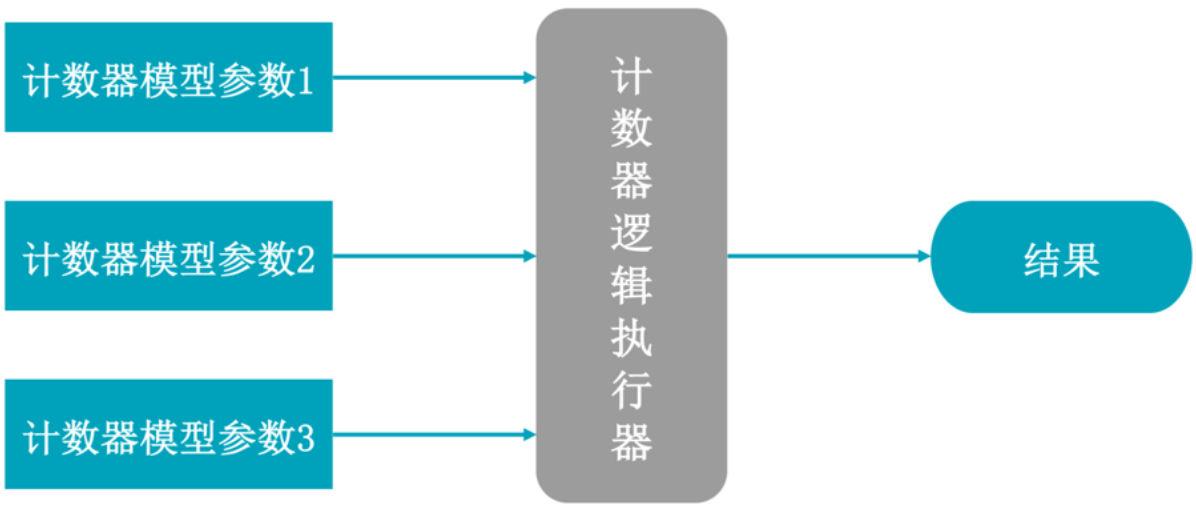
计数器的本质是对某一对象进行分组，对某一字段进行函数计算的过程。

select sum(字段) from table group by 对象。

如果将对象抽象成主体，字段抽象成客体，sum抽象成函数，那么计数器模型组成如下：

计数器模型 = 主体+客体+函数

那么计数器模型的设计如图所示：



主要由三个部分组成：

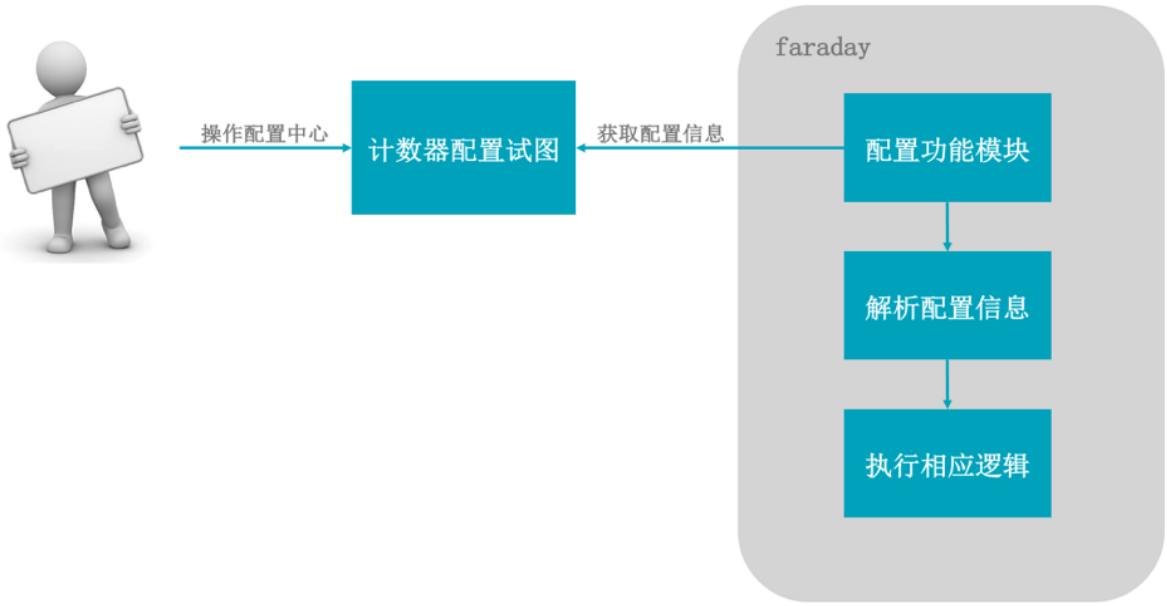
计数器模型参数：计数器构成的三个要素分别是主体，客体，函数。

计数器逻辑执行器：主要用来执行计数器模型中函数部分，例如count、sum、max等。

结果：计数器逻辑执行的结果。

faraday系统设计

鉴于上面所说的计数器模型，我们开发了faraday服务，新计数器主要分为计数器视图中心和faraday soa 服务，具体如下图所示：



视图配置中心：

主要提供给调用方人员配置计数器模型参数。

例如：计数器类型，是否持久化，存活周期等等。

配置完参数，调用方无须关注计数器的具体实现细节，内部的操作。

流程对于调用方来说是个黑匣子。

faraday 服务：

主要有三大模块组成，分别是：

1. 计数器模型配置器

   主要解析调用方的模型参数，并从配置视图中心获取计数器模型配置信息。

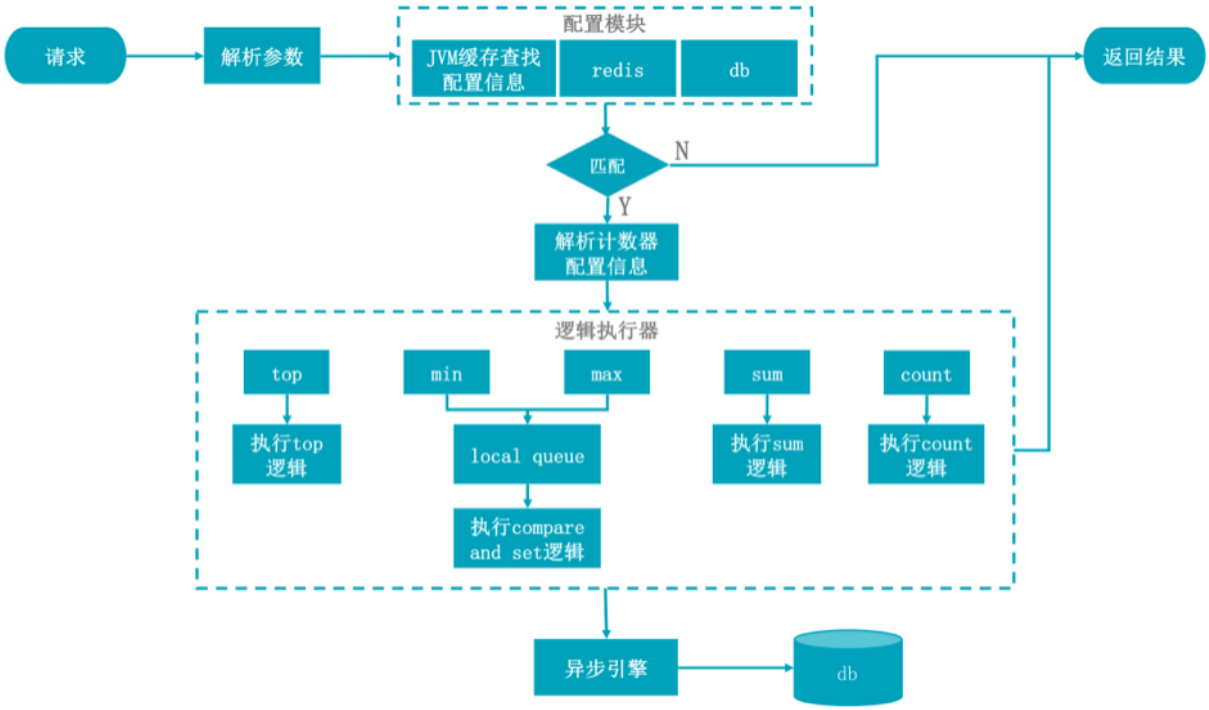
2. 计数器逻辑执行器

   负责各种计数器的逻辑操作，例如：count、sum、max、min等。

3. 异步引擎

作用有2个方面，第一是异步化入库，防止操作数据库，导致接口性能降低，第二是化并行为串行，降低高并发带来的数据不一致问题。

faraday 整体流程图如下：

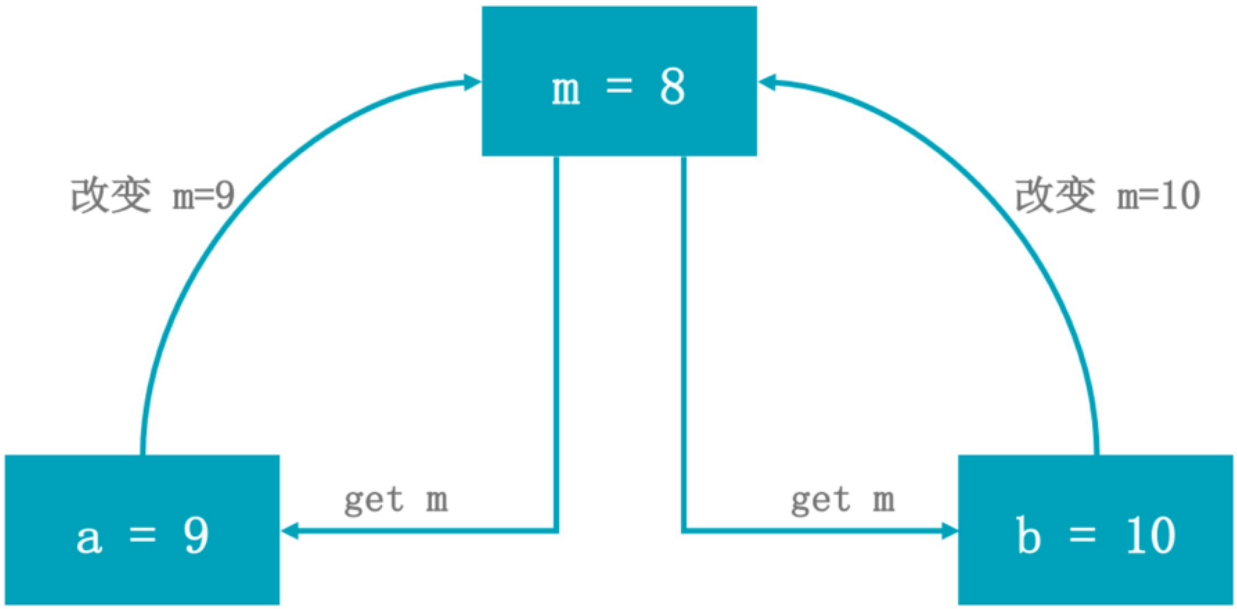


计数器类型

风控主要使用的计数器类型是count、sum、max、min、top，为了应对每天近800万的订单量，风控使用redis进行计数服务，主要是看中了redis不错的单机性能。

count和sum可以使用redis的incr 就可以办到。对于top类型的计数器，可以使用redis的sorted set，利用sorted set的score进行排序。

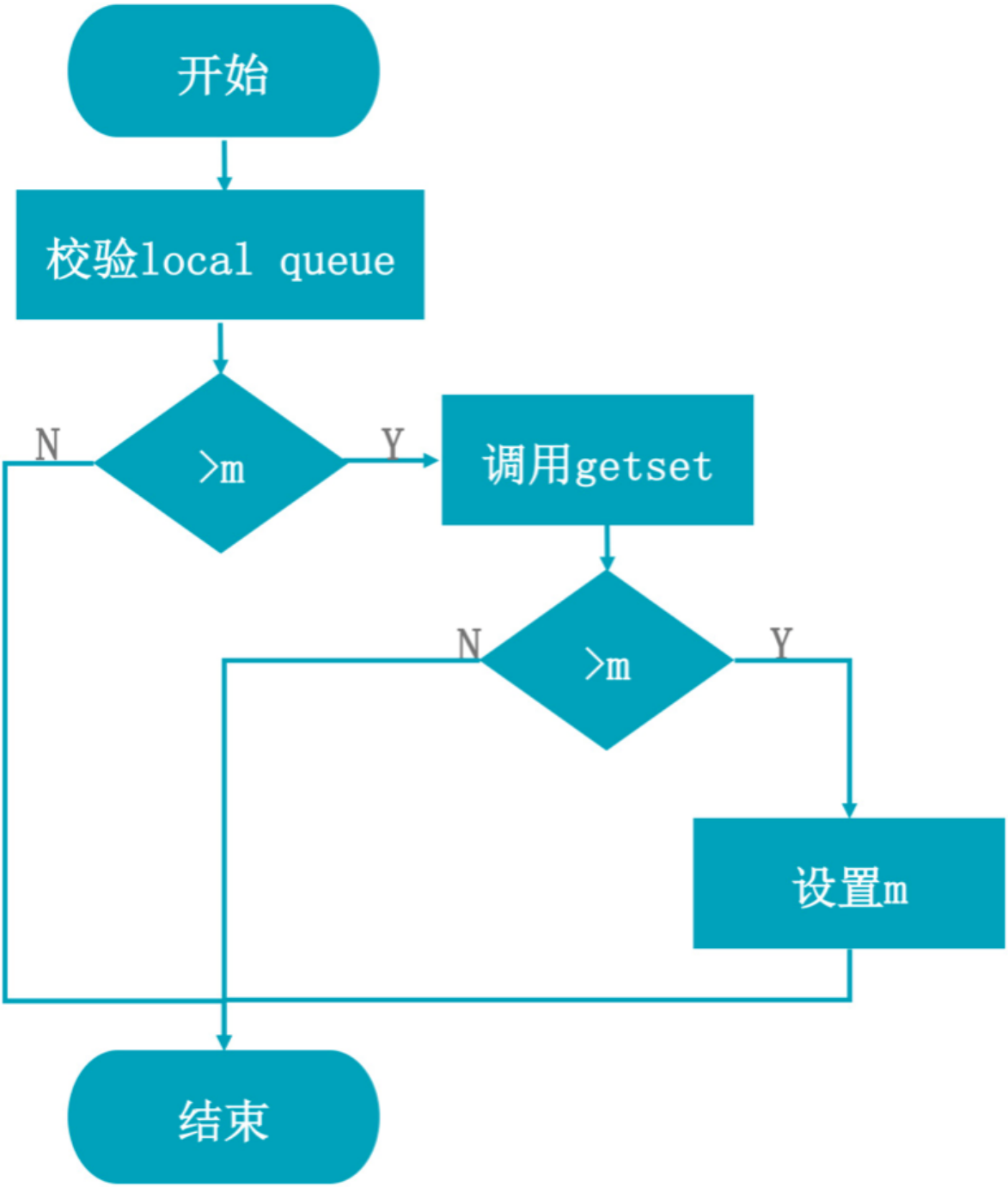
计数器中的max和min，计算的是最大值和最小值，最大值和最小值在高并发存储的时候会有一个问题，就以max为例讲解，如图：



在并发的时候，当a和b同时读到m的值是8，此时a=9，比m大，满足修改m的条件，去修改m=9；另外b=10，也满足修改m的条件，此时b也去修改m=10；因为修改的顺序不同，有可能最终m=9，与我们的预期值10不一致。

那么有没有什么办法彻底解决这种场景呢，答案肯定是有的，就是利用mq，将消息发给mq，然后部署一台机器，去单点消费这个mq，再去比较m的值，就不会存在同时修改m值的问题。在实际生产环境中，部署单个机器消费mq，肯定是行不通的，因为这样不满足可用性，而且单点消费还会出现服务挂掉，不消费mq，从而导致mq消息堆积等一系列问题。

在faraday中其实采用了一个折中方案，就是将消息发送给local queue，在local queue中先获取m的值作比较，然后利用redis中的getset方法，再去比较一次，这样可以大大降低高并发带来的赋值不一致问题，具体流程如图所示：

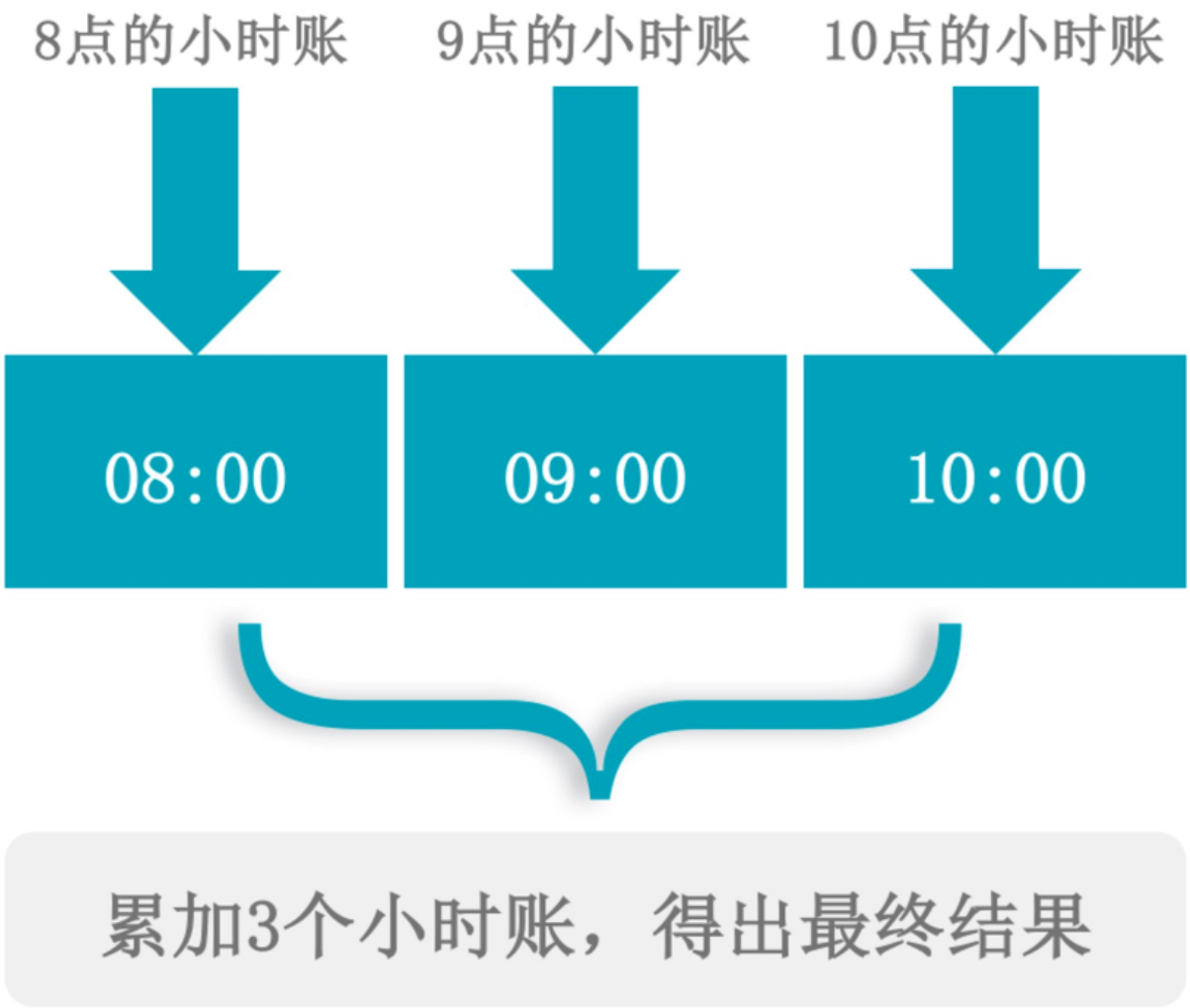


时间窗口设计

计数器设计中有一大难点是时间窗口的设计。当初想到的方案有2种：

第一种是每隔xx时间，例如：每隔1天，每隔3小时，每隔5分钟。如果计数器选择的类型是每隔1小时，就将一天划分成24个1小时；如果是每隔2小时，就将一天划分成12个2小时；这种时间划分有一个缺点，如果是每隔5小时这种的，是没有办法整除的。所以这种时间窗口方案被我们抛弃了。

第二种是最近xx时间，例如：最近1天，最近3小时，最近5分钟。我们还是以最近xx小时举例，不管我们设置的是最近3小时，还是最近5小时，我们在redis中都是以小时账的形式进行存储，获取的时候，只要将前几小时的小时账进行合并就可以了，那么同理如果是天，就是日账，月就是月账。如图所示：



对于这种时间窗口有一个弊端，如果滑动时间窗口粒度很长，计算的复杂度就会越高，例如 统计最近10小时的计数器，就需要累加10个小时账；为了应对这种粒度很长的时间窗口，我们提出了中间值的概念，将历史的时间窗口计数器计算成中间值，那么无论滑动窗口多长，只需要计算一次，即： 计数器 = 当前时间窗口值 + 中间值

最终风控采用的时间窗口是第二种方案。

计数器key的设计

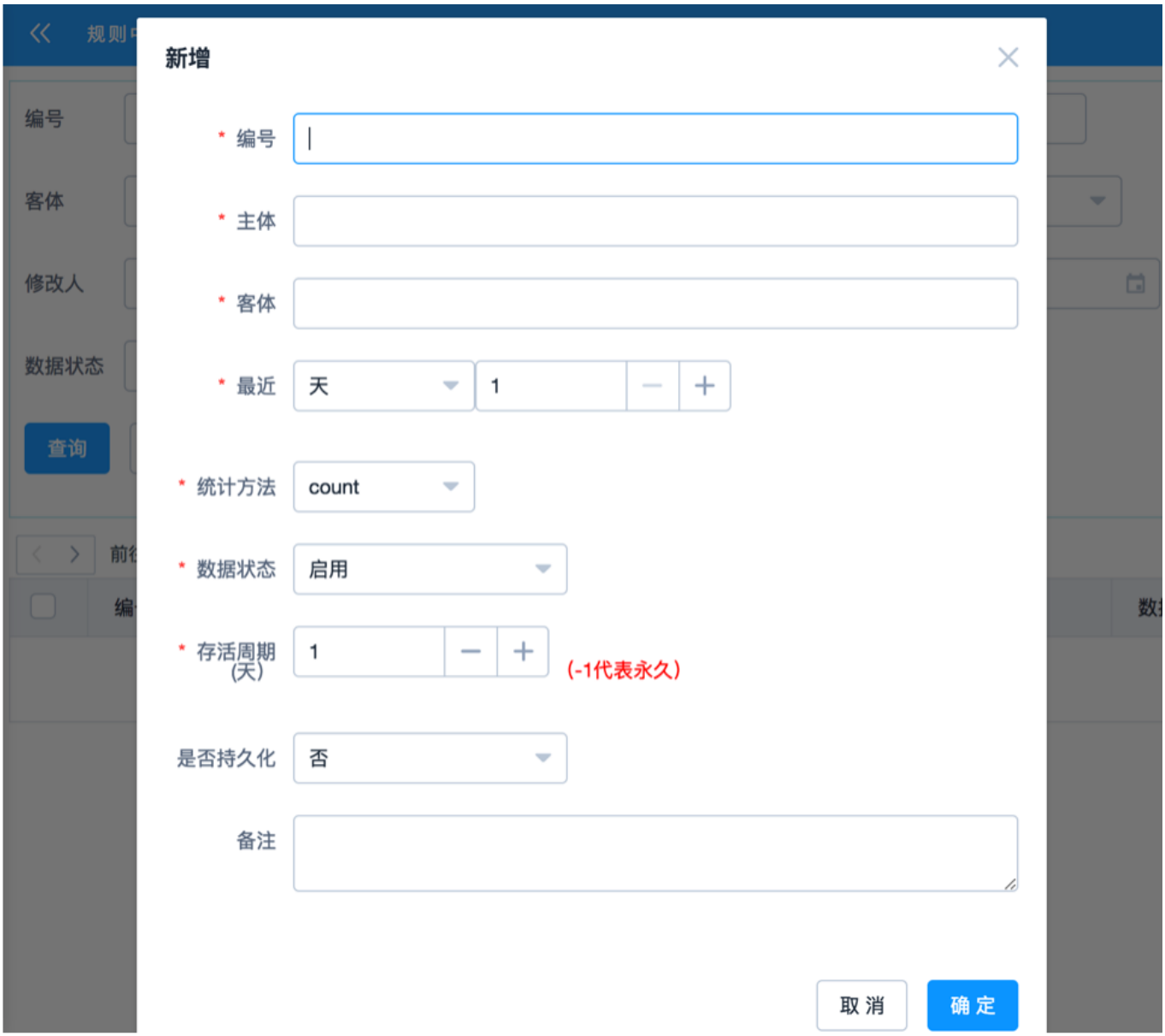
faraday 中key主要由 编号+主体+客体+统计方式+时间戳 组成，由于Redis是纯内存的，所以成本也不算低，为了降低成本，我们需要缩减key的长度，首先去掉了一些不必要的前缀，这些前缀加起来至33个byte，如果以1亿个计数器计算，去掉这些前缀，可以节省近31G的内存空间。另

外对于时间戳，我们是采用yyyyMMdd 这种字符串形式存储，第一比较容易阅读，第二相比time-stamp存储的字节更少。

faraday的接入步骤

视图配置

计数器的配置主要是由计数器的使用者自助完成。计数器在后台配置如图所示：



调用faraday服务

调用方只需要配置几行代码就可以完成计数器服务调用，伪代码如下：

```
Counter counter = Client.getClient(Counter.class);
   Map<String,String> counterParam = new HashMap<>();
   counterParam.put("mainBody","shop");
   counterParam.put("subBody","amount");
```

```
counterParam.put(“type”,”sum”);
counter.counter(counterParam);
```

新计数器的优势

最后新计数器的优势也很明显，主要体现在以下几点。

第一：避免重复劳动

将分离在各个系统的计数器逻辑，抽象成一个服务，避免了重复劳动，调用方系统不需要关心计数器逻辑的实现细节，而key的生成规则对调用方系统是透明的，调用方只需要传几个参数就可以获取和设置计数器的值。

第二：快速满足风控业务需求

新计数器可以在后台系统动态配置，计数器的特性可以在线动态修改和生效，避免了每次更改都要走上线流程的繁琐步骤。

第三：对计数器有更强的把控能力

新技术器是一个服务，一个系统，让专业的系统去做专业的事情，更有利于职责分离，当计数器出现问题的时候，更有利于排查问题，而不是像之前那样去check各个系统。

实录：《伍正云：饿了么风控计数服务实战解析》

写评论

哈比

09月11日 20:30 会有一次线上交流，大家可以在本篇文章下方「写评论」提问。这篇文章有什么地方不理解的，或者有什么主题相关的问题，都可以提问哦~ 线上交流时，伍正云老师将会按照提问的先后顺序进行回答：）

7天前 2 0

Quoraliao

中间值是历史的时间窗口计数器计算而来的结果。这句话的意思是记录每一次人为统计某时间段后保存下来的值，还是系统自动统计各种累加值之后的历史值存档?

20小时前 1 0

ecore

“对于这种时间窗口有一个弊端，如果滑动时间窗口粒度很长，计算的复杂度就会越高，例如 统计最近10小时的计数器，就需要累加10个小时账；为了应对这种粒度很长的时间窗口，我们提出了中间值的概念，将历史的时间窗口计数器计算成中间值，那么无论滑动窗口多长，只需要计算一次，即： 计数器 = 当前时间窗口值 + 中间值”uesrid:888 的用户在14:00–14:10下单5次，14:10–15:00下单8次，在15:00–16:00下单3次，在16:00–17:10分下单2次， 在17:15的时候如果获取该用户最近两小时的下单数，值是多少？计算逻辑是什么?

7天前 0

白桦林

饿了么除了这种计数器服务反作弊，有实时规则反作弊系统？实时规则和离线规则如何配合？

7天前 0 0

志刚

localqueue+getset这种方式 如果是集群部署，需要保证对应同一个key的请求落在同一台服务器上吗？ 如果不是，不同server之间对同一个key在各自的localqueue里面值都可能不一样 高并发场景下，这个判断会大量失效吧。 同样，在此流程中getandset能保证的是至少将本机的最新值写入了，但是这个值在集群范围内是否是最新的，就不一定了

1天前 0 0

Quoraliao

为了降低成本，faraday 的 key 去掉了一些不必要的 33 个 byte 的前缀，可以请您举例说明哪些是属于不必要的吗？这个取舍需要和其他部门的人一起商讨吗，主要需要请教哪些人员的意见？

20小时前 0 0

Quoraliao

时间戳用 yyyymmdd 字符串代替 timestamp 存储的优势是便于阅读，其次是节省内存，那这种方案有没有什么缺点？毕竟 timestamp 也有挺多优势的。

20小时前 0 0

Quoraliao

正如文章中说到的，Redis 的单机性能很不错，但是它是纯内存的，所以成本较高。除了 Redis，饿了么还考虑过别的方案吗，它们相比 Redis 主要有什么优劣势？

19小时前 0 0

Geek

麻烦老师能详细说下中间值这个吗？是等同于历史值，多长时间计算或者通过什么规则计算？

17小时前 0 0

Geek

使用localqueue 的方式集群环境下如何考虑？

17小时前 0

Geek

incr 有两种方式，一种是long一种是double ， 如果是统计金额，long 肯定不实用， double 会出现精度问题，这个是怎么解决的?

17小时前 0 0