【万字长文】论如何构建一个资金账户系统

导语

资金账户是互联网和金融业务中的非常常见的系统，尤其是在电商、支付等业务中必不可少。资金账户系统本身其核心模块的整体架构往往并不复杂，但其对于资金安全和可用性的要求往往非常高，导致需要建设好一个资金账户系统并不容易。本文以笔者在前司实际项目过程中实现的资金账户系统为例，探讨了在资金账户系统设计和实现中会遇到的问题以及相应的解决方案。需要强调的是，笔者也是资金相关系统的入门者，本文目的是抛砖引玉，有误之处，还请大家多多指正、多多探讨、不吝指教，笔者不胜感激。

一、什么是账户

我们先看看标准定义：账户是根据会计科目设置的，具有一定格式和结构，用于反映会计要素的增减变动情况及其结果的载体。

从业务视角来看账户其实就是用于记录某个主体的某类型资金的余额以及余额变动明细的数据载体。

如果从第三方支付这块业务来看的话，账户是支付机构内部为其服务对象（用户、商户、银行等）创建的物理记录，这些记录包含了对象的关键信息，如机构为对象分配的唯一ID、对象的余额、交易的流水、账户状态等等。 可以说账户是支付机构识别服务对象的根本，所有服务对象都必须要有账户。

综上所述，账户有3个关键的点：

账户余额：账户中的资金数目，这个账户有多少钱

账户流水：这个账户资金进进出出的明细记录，任何余额的变动都需要记录流水

交易凭证：用来记录交易过程的信息

三者主要内容包括：

1.1 余额

账户余额记录用户的资金数目，当发生交易时，会对余额进行更新操作。一般余额信息包括以下字段：

* 账户ID，这个ID有的平台会有编码规则，比如某些位用来区分个人或者商户，区分币种以及校验位等。
* 币种
* 余额，一般使用币种最小单位。
* 账户状态，比如是正常态、支付或者冻结等。
* 余额版本号，这个字段非常重要，体现了余额的变化过程，是与流水进行关联的关键。

1.2 流水

账户体系的目的是记账，仿佛只记录余额，对余额进行增删改查就可以了。但是这样做有两个弊端：

1. 没有办法体现出余额变化的过程。
2. 账户余额被篡改了没有办法追查。所以在记录余额的同时，还需要记录每一笔变化的过程，也就是账户的流水。

当余额发生变更时，需要同步记录流水，以此来跟踪余额的变化，流水信息一般包括如下内容：

* 流水ID
* 凭证ID，一般是业务单号（如订单ID、退款单ID等），也就是下面凭证中的ID。

* 发生金额
* 发生币种
* 起始余额
* 终止余额
* 余额版本号，与余额信息中的版本号字段对应。
* 资金方向，标识是入账还是出账。
* 源账户ID
* 目的账户ID
* 交易时间戳

总体来看，流水和余额互为冗余数据，流水不仅可以有效地减少由于内部错误导致的账户余额错误的问题，也便于账户系统与其他外部系统进行对账，所以账户系统记录流水是非常必要的。

在设计账户流水时，有几个重要的原则必须遵守：

1. 流水记录只能新增，一旦记录成功不允许修改和删除。即使是由于正当原因需要取消一笔已经完成的交易，也不应该去删除交易流水。正确的做法是再记录一笔“取消交易”的流水。
2. 流水中的余额版本号必须是连续递增的，我们需要用余额版本号来确定交易的先后顺序（注意：不是通过交易时间戳）。

1.3 凭证

凭证用来记录交易过程中的信息，是用户交易的依据。凭证对应到支付平台内部的各种单类，比如充值单，提现单，交易单等等。一般包括：凭证ID、交易参与方、交易金额、交易类型、交易状态（比如支付中，支付成功，转退款等）、交易渠道等信息。

凭证单理论上可以放到业务层，不放在账户核心层，这样账户层就只有余额和流水。 抓住了余额、流水这两个点我们基本就抓住了资金账户系统设计的核心了。

二、什么是资金账户系统

资金账户系统，简单的类比就是一个存折账本，即用户的每一笔资金变动全部在账本上反映出来。对于银行给我们开立的账户，称为银行账户；支付公司给我们开立的账户称为支付账户；电商平台给用户开立的账户称为电商虚拟账户；由于它不像银行一样真实的记录资金变动情况，只是将账记好而已，所以它

是虚拟的而不是实体账户。

常见场景：的零钱账户、支付公司的商家账户、电商平台的推广佣金账户、余额账户等均属于虚拟账户。

第三方支付作为中立的第三方，截断了用户和商户的资金流，资金先从用户账户转移到第三方支付平台账户，得到双方确认后再从支付平台账户转移到商户账户。

支付平台为客户提供了资金流转以及结算等服务，必须建立自己独立的资金账户系统，以此来保证每个客户资金的准确性以及资金变动的可追溯性，这套资金账户系统在支付平台中成为“核心”，整体类似于银行的账户核心，但会比银行账户核心简单一些。

2.1 资金账户系统的目的

账户的核心目的，就是将账记清楚，不能出现错计、漏记。同时在实际业务中，账户体系也可以解决特殊情况的业务，因为并不是所有的业务变动都会在业务中记录，但是却需要在账户系统中体现。

2.2 资金账户系统的构成

一个账户体系一般分为**账户结构**和**账务结构**两部分；账户结构用于记录一个账户基本信息、类型、当前余额等；而账务结构则是用于记录每个业务对应的余额变动情况。这里好理解，说白了就是第一节中说的余额和流水在系统中各自对应的实体。

三、实现一个支付资金账户系统

笔者前司所在的是第三方支付跨境业务，在业务发展过程中涉及到在腾讯云的云原生生产环境搭建一套资金账户系统的需求。所以就自己工作过程中学习和了解到的内容谈谈我对于构建资金账户系统的理解。由于笔者也是资金相关系统的入门者和初学者，所以若有错漏之处，还请大家多多指正、多多探讨、不吝指教。

资金账户体系是支付交易的基础，就像电池对于手机，油罐对于加油站，心脏对于人体。那么这么核心的系统是不是很难设计呢？其实其核心思路恰恰不难，这也印证了那样一句话“大道至简”。

但资金账户系统也有其特殊要求：

1. 资金账户作为交易的基础，对于可用性的要求在整个支付系统中几乎是最高的级别。
2. 资金账户因为涉及资金安全，故而对于系统的安全性要求非常高。
3. 资金账户作为一种资金池形态，要严格做好其合规性。

当然，资金系统如何去落实这些要求，也需要根据实际情况和业务的要求进行相应的调整。我们后面的所有内容，都会紧紧围绕着这些要求，追求既满足最佳实践要求，又不过度设计的原则。

3.1 重要原则

由于资金账户系统本身属于会计范畴，故一定是需要遵守一系列的会计准则的。比如：内部账户记账的一条原则：**有出必有入，出入必相等。** 所有的记账操作都必须遵循这一条原则，这是为了确保交易安全，避免凭空加钱和减钱行为的出现。同时，该原则也是各种核对（核心事务核对、备付金核对等）系统的依据之一。

账户记账的理论基础基本都是基于借贷记账法的，什么是借贷记账法呢?

3.1.1 复式记账法

复式记账法的英文为Double Entry Bookkeeping,是从单式记账法发展起来的一种比较完善的记账方法。也叫复式记账凭证。与单式记账法相比较，其主要特点是：对每项经济业务都以相等的金额在两个或两个以上的相互联系的账户中进行记录（即作双重记录，这也是这一记账法被称为“复式”的由来）；各账户之间客观上存在对应关系，对账户记录的结果可以进行试算平衡。复式记账法较好地体现了资金运动的内在规律，能够全面地、系统地反映资金增减变动的来龙去脉及经营成果，并有助于检查账户处理和保证账簿记录结果的正确性。在我国，复式记账曾有借贷记账法、增减记账法、收付记账法三种，但规定使用的只有借贷记账法一种。

3.1.2 借贷记账法

借贷记账法的记账规则可以概括为：**有借必有贷，借贷必相等。**

第一，在运用借贷记账法记账时，对每项经济业务，既要记录一个（或几个）账户的借方，又必然要记录另一个（或几个）账户的贷方，即“有借必有贷”；账户借方记录的金额必然等于账户贷方的金额，即“借贷必相等”。

第二，所记录的账户可以是同类账户，也可以是不同类账户，但必须是两个记账方向，既不能都记入借方，也不能都记入贷方；

第三，记入借方的金额必须等于记入贷方的金额。

会计恒等式：资产=负债+所有者权益

“借”对应资产的增加、负债及所有者权益的减少。

“贷”对应资产的减少、负债及所有者权益的增加。

这些概念由于涉及到太多的会计知识，比较抽象，要理解还需要费点功夫。我说一下我的简单理解。

1. 对于第三方支付业务来说，由于银行账户对应着备付金，故是属于资产。而商户和用户的账户属于支付平台内部账户，属于负债。
2. 每一笔资金流操作应该都是“成对”的，至少会在两个相关账户中记录金额相等、借贷相反的流水。理论上不应该有一笔单独存在的流水。两个账户的类型可以是相同的（比如都是负债类账户），也可以是不同的。比如转账，A的账户有出账的流水，则必然B的账户有一笔与之对应的入账流水，这里两个账户都是负债类账户。那有人可能会问，那么充值、退款、提现的流水呢，他们也会有两笔流水吗？答案是肯定的。它会对应着一笔负债类账户的流水和一笔资产类账户（银行账户）的流水，且两者借贷方向一致（同时加钱或者同时减钱）。

3.1.3 账户的基本操作

一般的账户系统都会提供开户、查询余额、充值、扣款、转账等基本接口。

3.2 我们的设计与实现

讲了这么多理论和原理，该进入正题了。我们在项目中是如何实现一个资金账户系统的呢?

这里首先了解一下我们的项目需求。我们的需求是要建立一个跨境收单系统，用于充当独立的主体对接境外的商户接入。由于独立部署在境外，因而最终选择的云原生的研发环境。**注意：由于项目的特殊背景，我们的资金账户系统仅需要记录商户相关的账户，而不需要记录用户相关的账户。并且承担的流量不大。**

3.2.1 资金流设计

首先是资金流设计，其中也包含了账户类型的设计。这里的原则是实事求是，一切从实际出发；如无必要，勿增实体。比如是否需要保留待结算B账户的问题，我们团队也是经过一番思考。最终考虑到产品规则设置我们的结算规则并非异步结算，而是单笔实时结算，故而我们取消了待结算B账户。最终整体账户类型和资金流设计如下。


关于手续费账户，可以分开收入和支出两个账户，更为安全，可用性也会更高。因为这样永远只有一个方向的资金流动。当然如果没那么高的性能要求，也可以合为一个统一账户。我们这里采用了后者的方式。

关于银行账户，它是一个虚拟账户，其可实时/动态反应每笔钱款变动，特别是实时记录在途的银行资金。为什么要引入银行账户，第一是因为要符合3.1的原则，避免单边账导致的借贷不相等的问题；第二是因为要确保系统虚拟资金流和银行物理资金流对齐，避免资金长短款。

本系统中结算、退款、提现、手续费充值等涉及外系统资金流向的情况，会记录银行账户流水。记录流水的方式有同步和异步两种，通常对于银行流水的记账采用异步的方式即可，我们采用的是一个定时任务异步导入银行账户流水。

3.2.2 存储选型

如果是整个系统是一座大厦的话，那存储就是地基。地基不牢，地动山摇。资金账户系统对于存储的要求是比一般的系统更高的。具体有哪些要求呢?

* **高性能。** 这样才能承受高并发的请求流量。
* **数据强一致。** 保证切换不同副本的情况下数据0丢失0出错。
* **支持透明分库分表。** 在分布式形态下，用户看到的逻辑表的实际物理存储可能是被打散分布到不同的物理节点上。希望做到对业务完全透明。这样一来作为开发者不需要关心分库分表细节。
* **支持不停机的容量弹性扩展。** 如果性能或容量不足以支撑业务发展时会自动扩展，升级过程中，开发者无需关心分布式系统内的数据迁移，均衡和路由切换。
* **支持跨区容灾、同城双活、跨城容灾等，故障自动切换其他副本，自动恢复，确保99.999%的可用性。**
* 最好有配套的**分布式事务解决方案。**

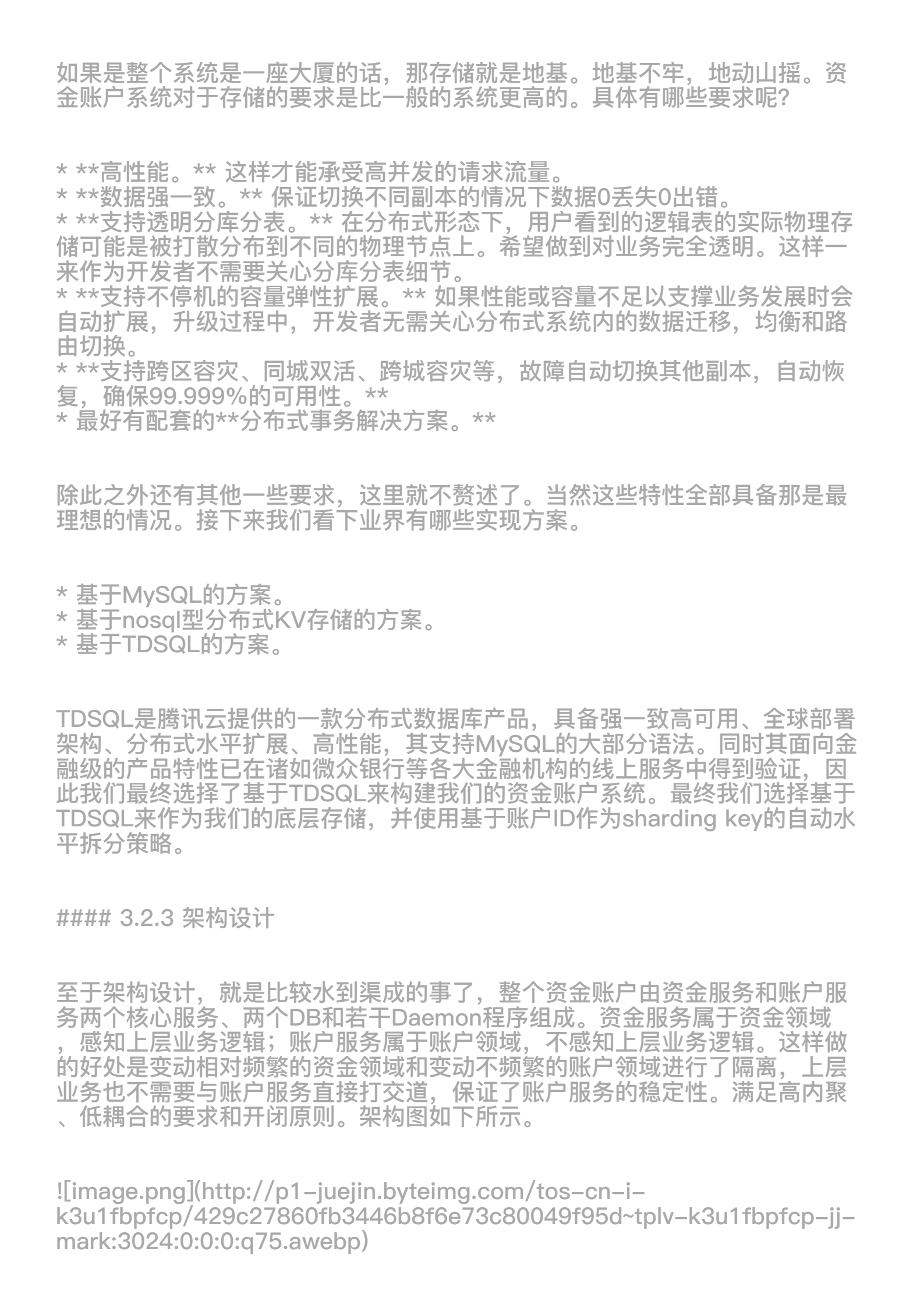
除此之外还有其他一些要求，这里就不赘述了。当然这些特性全部具备那是最理想的情况。接下来我们看下业界有哪些实现方案。

* 基于MySQL的方案。
* 基于nosql型分布式KV存储的方案。
* 基于TDSQL的方案。

TDSQL是腾讯云提供的一款分布式数据库产品，具备强一致高可用、全球部署架构、分布式水平扩展、高性能，其支持MySQL的大部分语法。同时其面向金融级的产品特性已在诸如微众银行等各大金融机构的线上服务中得到验证，因此我们最终选择了基于TDSQL来构建我们的资金账户系统。最终我们选择基于TDSQL来作为我们的底层存储，并使用基于账户ID作为sharding key的自动水平拆分策略。

3.2.3 架构设计

至于架构设计，就是比较水到渠成的事了，整个资金账户由资金服务和账户服务两个核心服务、两个DB和若干Daemon程序组成。资金服务属于资金领域，感知上层业务逻辑；账户服务属于账户领域，不感知上层业务逻辑。这样做的好处是变动相对频繁的资金领域和变动不频繁的账户领域进行了隔离，上层业务也不需要与账户服务直接打交道，保证了账户服务的稳定性。满足高内聚、低耦合的要求和开闭原则。架构图如下所示。



3.3 重难点

说到整个系统的重难点，其实就两个。一个是**资金安全**，一个是**高可用性**。而我们的目标，是在10w笔/天的预估量级下，保证资金的绝对安全和尽可能高的可用性（例如前司要求是5个9）。

3.4 资金安全建设

看了以上的内容，大家会知道说白了资金账户就是数据库里的一行记录，账户余额就是一个字段。因此不当操作极易引起安全事故。资金安全，“国之大事，死生之地，存亡之道，不可不察也”。 资金账户系统的最重要的要求就是资金安全。而资金安全的核心就是一句话：帐记清楚，不出错。

资金安全从源头防范上可以又分为两个方面：**1.防范外部攻击威胁风险。2.防止系统内部错误。**

最终目标是保证整个系统符合**合法性、完整性、正确性**三大要求。

3.4.1 STRIDE模型

首先我们看外部攻击威胁风险。光靠凭空想，是容易产生遗漏的。那么这里是否有理论支撑呢？答案是肯定的。

功能安全风险分析是对系统的系统性失效和随机性失效进行风险评估，对于网络安全风险，需要通过威胁分析识别系统的威胁场景，用于形成有对应威胁的控制措施和有效的分层防御，威胁分析是信息安全风险分析的重要组成部分。

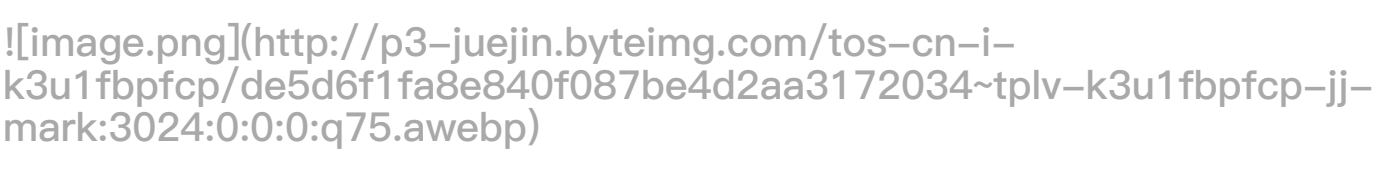
威胁识别有不同的办法，如STRIDE模型、攻击树、Kill Chain等。STRIDE在软件安全分析领域应用较多，是微软开发的用于威胁识别的工具，它把威胁分成如下6个维度来考察：

1. Spoofing（仿冒）
2. Tampering（篡改）
3. Repudiation（抵赖）
4. Information disclosure（信息泄露）
5. Denial of Service（拒绝服务）

6. Elevation of Privilege（特权提升）

微软将STRIDE应用于软件的威胁识别，这6个维度也可以扩展到系统层面。

我们也可以使用STRIDE来对我们的系统安全性进行分析。



3.4.2 权限控制

权限校验保证只有合法的资金操作请求会得到处理，它是资金安全的**第一道防线**。权限校验包括数据库、机器的权限校验、业务层的权限校验。主要目的是解决STRIDE模型中的仿冒、特权提升、抵赖等安全威胁。

数据库权限校验要求每个DB库都只允许本服务访问，不允许跨服务调用，且数据库密码要托管至开发人员无法获取的地方。

机器权限校验的主要措施是采用腾讯云TKE的内网隔离策略，同时限制开发人员登录现网机器实现的。

业务层的权限校验主要包含两个主要方法：票据校验和模块认证。

1）票据校验：

首先看理论基础。访问控制是指防止对任何资源进行未授权的访问，从而使计算机系统在合法的范围内使用，本质上是一种**授权机制**。访问控制有很多种模型，最常见的三种模型分别是自主式访问控制(DAC)、强制访问控制(MAC)和基于角色的访问控制(RBAC)：

* 自主式访问控制：主体对其拥有的数据具有绝对控制权，也可以授权别人访问和操作。适用于简单的业务场景。
* 强制访问控制：系统将对数据的操作分为不同的级别，强制要求对不同级别数据的访问和操作具有不同级别的鉴权。适用于对数据安全有极高要求的系统。
* 基于角色的访问控制：抽象出不同角色，赋予不同角色下用户以不同权限。适用于权限划分复杂的业务场景。

要想做支付业务，需要达到国家等保三级认证，认证要求系统必须对所有的主体和客体采取**强制访问控制。**

票据校验即是一种强制访问控制的方式。“票据”代表本次请求的**身份**。生成票据的前提是证明了身份。

举两个例子：

* 商户向商户API发起请求时需要带API密钥签名或证书签名，我们的API服务会验证这个签名，验证通过才会执行后面的业务逻辑。
* 商户在商户平台登录后，每次请求前会自动通过PHP的hook验证登录态，验证成功才会执行具体的业务逻辑。

在这些例子中，验证签名、登录态或其他能够证明身份的信息就是鉴权的过程，验证成功就代表我们相信本次请求真的是由商户发起的，即证明了商户的身份。

验证成功后，我们就用商户身份等关键信息，加上一个我们自己生成的签名（用于防伪造），生成一个字符串，称为**票据**。这个过程称为**颁票**。

票据的生成有鉴权的保证，因此票据本身可以用作鉴权。票据生成后随着请求向后传递，每个收到请求和票据的服务都可以通过验证票据签名合法性和身份一致性来完成鉴权。这个过程称为**验票**。

通过颁票和验票，我们可以将整个微服务系统边界复杂而多样的鉴权手段收归为一种统一的鉴权方式：

接入层服务鉴权颁票，其他服务验票。

我们的系统也基于类似的思想实现了一套自己的票据系统。

2）模块认证：

我们的服务部署在腾讯云TKE上，大部分以RPC接口的形式提供。只要知道提供Server的IP和Port，那大部分的接口都可以直接调用，如果有坏人进入了

TKE内网那这基本就是不设防，短板非常明显。而对于受保护的接口，一般是基于票据/IP限制等手段保护，而这些保护手段也面临着各种问题。如商户票据并不能识别调用源，IP限制在混合部署情况下可能误限制，运维成本高且安全性一般。所以这个时候一般都需要做一个上下游服务认证。上下游服务认证是指每个服务要限制调用自己的来源服务，不管这个服务部署在什么机器上。并且任何服务或者工具也无法伪装成别的服务来访问下游服务。

我们的系统也基于类似的思想实现了一套自己的模块认证。

3.4.3 防篡改

防篡改是指防止通过直接修改数据库字段等方式直接修改账户和流水等数据。主要目的是解决STRIDE模型中的篡改的安全威胁。这一点一般是通过计算重要字段的MAC (全称 Message Authentication Code)值，也就是摘要值，并保存到数据库中的方式来实现的。

具体实现一般是通过hmac算法计算关键字段的hash值来实现的。hmac是Hash-based Message Authentication Code的简写，就是指哈希消息认证码，包含有很多种哈希加密算法，sha256是其中一种。

对于账户系统而言，最重要的当属于账户表的数据，需要在插入和更新时对其重要字段合并起来使用hmac-sha256进行MAC值计算，并将计算结果作为一个字段（如data_mac,为保证对新老数据的兼容，通常还有一个data_mac_version字段表示其版本号）存储在账户表中。而在查询时增加校验MAC值的校验，如果校验失败可以采取告警等手段进行主动发现。

3.4.4 密钥防泄漏

可以看到，我们之前的诸多方案都会涉及一些敏感数据，如签名用的私钥、数据库的账号密码等。如果这些信息泄露了，那么之前所描述的方案也就不安全了。所本节的目的主要是防止发生STRIDE模型中提到的信息泄露问题。

业界通常实践是使用[密钥管理系统](http://cxyroad.com/”https://cloud.tencent.com/product/kms?from_column=20065&from=20065”)（KMS）来对密钥进行统一管理。密钥管理系统是一款安全管理类服务，可以让使用者轻松创建和管理密钥，保护密钥的保密性、完整性和可用性，满足用户多应用多业务的密钥管理需求，符合安全的要求。

我们先来看看业界最佳实践。以Google Drive为例，所使用文件加解密方案如

下：

其中图中的密钥管理模块就是KMS。

除了密钥的安全保存外，KMS还有一个优点，可以实现密钥的自动轮换。由于使用软件产生的伪随机数的安全度要低于专用硬件加密机生成的真随机数，KMS使用了基于硬件加密的真随机数，这大大提高了密钥的强度和随机性。

KMS是云平台和产品合规的基础安全组件，海外用户需要满足FIPS-140-2标准，国内用户需要支持国密。一般的密钥管理系统提供以下特性：

* 基于硬件加密机的真随机数
* 密钥的权限细粒度管控
* 密钥的自动轮换
* 密钥生命周期的管理（创建、开启、禁用、计划删除、销毁等）
* 自有密钥的导入
* 多级密钥管理

腾讯云提供了满足国内国际合规需求的KMS，我们直接使用。

3.4.5 限流限频

限流限频相信大家都很熟悉，目的主要是防止发生STRIDE模型中提到的拒绝服务问题。一般都是通过引入单机限流或者分布式限流组件来实现的。这里不再赘述。

3.4.6 另外一个角度

通过以上分析，STRIDE模型中提到的6大威胁都得到了相应的解决。但这仅仅是从防范外部安全威胁的角度来分析的，真实系统中资金安全问题，往往是来自于系统内部错误，所谓“堡垒是从内部攻破的”。那如何应对可能存在的系统内部错误呢？从方法论上往往都是通过对整个过程进行事前事中事后的分析来进行全方位的review。这里由于篇幅所限，不侧重review的过程，直接给出结论。

3.4.7 幂等性设计

幂等性：就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。

对于账户和交易类系统，做好接口的幂等性至关重要，既是保证资金安全的利器，也是一致性补偿的必要条件。 简单来说，幂等性就是要保证同一个单号有且仅能有一次操作成功。比如使用单号deal_no1做一笔5块钱的充值，假设充值成功后重复使用该单号做充值，程序能够判断该单号已充值过，可以返回调用方已充值成功的错误码，或返回成功处理（这里采用哪种方式，取决于上层是否需要感知已充值成功这一状态）。实际无论调用充值接口多少次，总共只有1次充值5块钱成功。

保证幂等性有若干种实现方式。通用可靠的做法一般是通过DB对单号做唯一索引，依赖DB来防重保证仅能有一笔入账成功。 我的系统中也是采用这一方式，保证资金账户系统所有提供给到上层的接口都是幂等的。

3.4.8 余额流水的一致性

由于资金余额和账户流水是最为重要的两个数据，必须保证同时写入/更新成功，如果出现不一致会导致严重的资金安全后果，因此需要保证强一致。此时几乎只能利用数据库的单机事务才能保证。实际上事务这个特性最初就是被设计用来解决交易问题的，在英文中，事务和交易就是同一个单词：Transaction。

确定使用数据库的单机事务实现，又有两种解决方案：

悲观锁方案： 创建资金流的时候，每次查询账户时，对该账户加排他锁。也就是在获取 accountid=1 的账户信息时对该行记录加锁，期间其他请求阻塞等待访问该记录。悲观锁适合写入频繁（写多读少）的场景。

乐观锁方案： 创建资金流的时候，每次查询账户时，不对该账户加锁，而是获取到账户当前的版本号version。在更新数据的时候需要比较程序中的version与数据库中的version是否相等，如果相等则进行更新，反之程序创建资金流，再次进行比较，直到两个version的数值相等才进行数据更新。乐观锁适合读取频繁（读多写少）的场景。

在我们的系统中，由于更新账户的频率比查询账户的频率高，基于以上我选择了第一种方案。

3.4.9 分布式事务与最终一致性

类似于转账这样的场景，涉及到两个账户进行操作，之前提到过我们是基于账户ID作为sharding key的，那么两个账户可能落在不同的机器上，这里就会涉及分布式事务问题。

分布式事务的解决方案有很多，对于转账这样的场景，业界通常的做法都是基于可靠消息队列、有限状态机与多重任务兜底的业务层最终一致性保障机制来实现的。

也就是把转账分为出账（扣钱）和入账（加钱）。出账为主事务，必须实时做成功；而入账作为从事务，是可以接受在异步流程做成功的。那么就使用消息队列驱动入账成功，这中间如果有极少失败，即使用兜底daemon保证。

我们最终采用的是TDSQL原生支持的分布式事务方案。TDSQL支持普通分布式事务协议和 XA 分布式事务协议。TDSQL（内核5.7或以上版本）默认支持分布式事务，且对客户端透明，像使用单机事务一样方便。TDSQL分布式事务采用两阶段提交算法（2PC）保证事务的原子性（Atomicity）和一致性（Consistency）。具体实现方式可以参见《TDSQL MySQL版 > 开发指南 > 分布式事务》的官方文档。

至于为什么这么选？首要的考虑还是你的系统诉求是怎么样的。根据CAP理论，在一个分布式系统中，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance）这三个要素最多只能同时实现两点，不可能三者兼顾。因此你的系统是要选择CP呢还是AP呢？我们根据我们的业务场景（量级小、异步驱动等特点）选择了CP，也就是2PC的实现方式，当时的一些对比选型的思考如下所示。

3.4.10 全方位的对账和审计

每个账户系统都不是孤立存在的，至少要和财务、订单、交易这些系统有着密切的关联。理想情况下，账户系统内的数据应该是自洽的。所有用户的账户余额加起来，应该等于这个电商公司在银行专用账户的总余额。账户系统的数据也应该和其他系统的数据能对的上。比如说，每个用户的余额应该能和交易系统中充值记录，以及订单系统中的订单对的上。

安全是支付公司的生命线。完整性和正确性靠对账和审计来发现。它们是资金安全的**最后一道防线。**

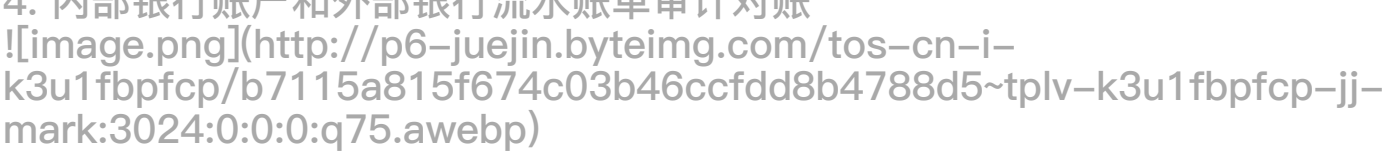
怎么证明账户系统的高一致呢？那就必须通过对账系统来验证，对账时间粒度越小，就能越早发现问题。

在对账和审计方面，从一致性、正确性、总分核对和及时性四个方面全面梳理并实现系统内和系统间的数据核对和审计，确保有异常发生时能及时发现问题。

* **一致性**：系统间关键数据一致，包含商户、账户、金额、单号、属性、状态等。
* **正确性**：业务规则的正确性，如收支分离实收手续费。
* **总分核对**：系统内总分数据一致，包含冻结总分（退、结、分）和手续费总分。
* **及时性**：业务单据超期未到终态，如解冻单、分账明细单、退款单。

资金账户的审计对账一般包括：

1. 逐层审计对账（由上层驱动）
2. 流水连续性审计对账
3. 余额正确性审计对账
4. 内部银行账户和外部银行流水账单审计对账



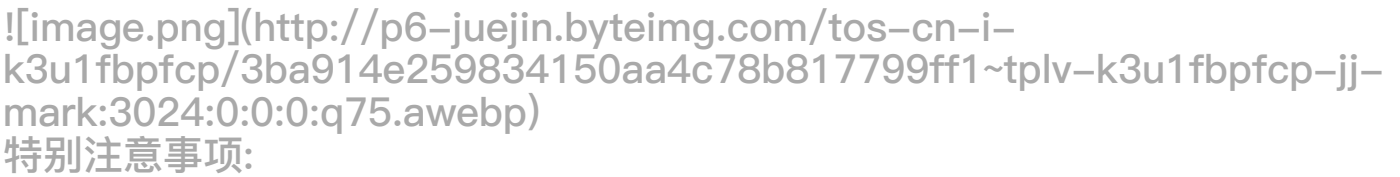
账户流水连续性审计的计算模型：

审计的公式：

1、单条流水期初余额 + 交易发生金额 = 期末余额

2、本条流水期初余额 = 上条流水的期末余额

3、本条流水的余额版本号 =上条流水的余额版本号 + 1


特别注意事项:

1）账户流水表需要具有【余额版本号】字段；

2）本账期最后一笔流水留待下个账期再审计。

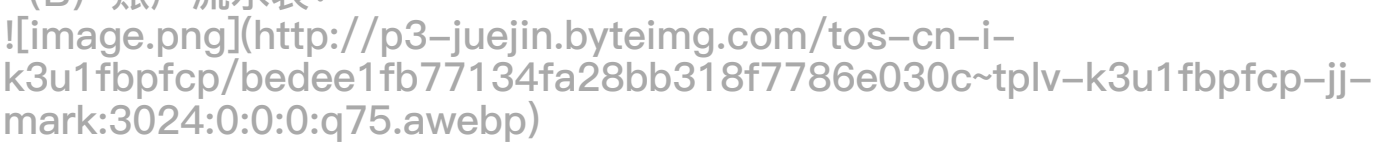
账户余额正确性审计的计算模型：

审计的公式：

1、当前账户余额=当前余额版本号对应的账户流水.期末余额

2、业务示例：

（A）账户表： 当账户余额版本号=3时，账户余额=180

（B）账户流水表：


特别注意事项:

1）账户流水表需要具有【余额版本号】字段；

2）先完成账户流水表的连续性审计，再进行账户余额正确性审计。

如果数据不一致，要第一时间生成业务异常记录并告警。

3.5.11 会计核算体系

所有上面措施做完，审计对账也都对齐了，就足够安全了吗？非也！有没有发现上述所有措施都是从研发的角度出发来解决资金安全的问题，只能说从工程系统的角度来看是完备的。但是从业务的角度来说，资金账户系统涉及公司重要财务数据，是需要财务协助审核才行的。从另外一个角度来看，有了财管的背书，我们系统的安全性也更加得到保障。因此我们需要建立会计核算体系。

建立会计核算体系的目的是保证资金变动可追溯，确保账实相符，及时核算监控资金，避免长短款风险。

需求简述：根据业务流程和特性，设置账户体系和会计记账科目，实时/T1动态记录各业务场景的会计账，并输出相关**营业总帐报表/会计调节表**等报表，供财务管理收单中心账务。

银行存款余额调节表（广义）

广义上的银行存款余额调节表，是指在银行对账单余额与企业账面余额的基础上，各自加上对方已收、本单位未收账项数额，减去对方已付、本单位未付账项数额，以调整双方余额使其一致的一种调节方法。该表主要目的是在于核对企业账目与银行账目的差异，也用于检查企业与银行账目的差错。

系统对账调节表（狭义)

下文简称：调节表，是针对境外收单中心系统各项业务的资金流转，结合境外收单中心系统信息流与外部银行资金明细进行对账调节的日常报表。其主要作用是监控境外收单中心系统资金流向，对业务操作做事后监督。

这里又涉及到另外一套会计账体系的搭建，限于篇幅就不再展开了。

3.5.12 研发流程规范

这也是一个不可忽视的重要影响因素。包括以下几点：

* 严格的现网机器和数据库权限管控
* 双人代码CR
* 完整的单元测试和接口测试覆盖
* 只能通过发布系统执行发布

3.5 可用性建设

可用性的定义：可用性就是一个软件系统处于可工作的时间比例。

计算机系统的可靠性用**平均无故障时间（MTTF）** 来度量，也就是说计算机平均能够正常运行多长时间，才发生一次故障。系统的可靠性越高，平均无故障时间越长。可维护性可以用**平均维修时间（MTTR）** 来度量，即系统发生故障后维修和重新恢复正常运行平均花费的时间。系统的可维护性越好，平均修复时间越短。

计算机系统的可用性定义可以定义为：MTTF / (MTTF + MTTR) * 100%

可用性可以归结为以下三个问题：**程序（节点）的可用性、服务的可用性、存储的可用性。**

3.5.1 程序（节点）可用性

其实它和服务可用性严格上说都可以归属于服务的可用性范畴。这里之所以分开列出来是想突出程序（节点）的可用性更多地是通过资源调度平台的特性来实现的。

腾讯云容器服务（Tencent Kubernetes Engine，TKE）是基于原生 kubernetes 提供以容器为核心的、高度可扩展的高性能容器管理服务。腾讯云容器服务完全兼容原生 kubernetes API，扩展了腾讯云的云硬盘、负载均衡等 kubernetes 插件，为容器化的应用提供高效部署、资源调度。我们主要通过TKE提供的以下特性来保证程序（节点）的可用性。

* 在私有网络中运行，可以使用自己的安全组和网络ACL，不与其他业务混布，提供了高隔离水平。
* 负载均衡，自动分配流量，自动添加和删除容器。
* 故障自动恢复出问题的节点。
* 自带完备的监控。

另外关于离线周期脚本的调度，则自己搭建和使用了AirFlow这一组件，它支持自动重试和错误通知，保证了离线周期脚本的高可用性。

3.5.2 服务可用性

1）限流限频

限流限频是可用性保证的兜底措施。保证DB不会被突然涌入的大量请求压垮，实现的方式就是拒绝超过自身处理能力的请求，对于一个写多读少的系统，这样做是无可厚非的，当然除此以外，我们还要想其他办法提高服务的可用性。

2）热key问题

对于电商秒杀活动，营销发券，企业红包等场景，均有大量用户需要在指定时间点对**单一账户**做资金转入操作。由于单个账户已无法再做分库分表处理，单组节点的性能上限就决定了单个账户的交易量上限。存储层的单账户瓶颈导致在该类场景下业务需要降级或者在业务层实现复杂逻辑，降低了业务的灵活性和商户体验。

我们来分析一下，热点账户账户的核心问题是每笔交易都会对余额字段进行update操作，更新时需要对账户进行加锁操作，频繁加锁释锁会对DB造成极大的性能压力，可能会超过DB的能承载的极限。

解决热点账户问题的常规思路一般有以下几种：

* 只记录流水不记余额
* 合并入账
* 多账户

要用更短的时间装满一个游泳池无非两种途径，一是让水流速度更快，二是多铺设几条入水管道。只记录流水和合并入账就是让水流得更快，多账户就是增加多个入水口。

特定应用场景下，可以不记余额只记录流水，这其实违背了记账原则，但在某些场景下，可以通过其他手段来补偿和保障，典型的应用比如C2B2C交易，中间的B账户可以不记余额，它的存在只是为了作为中间交易的对手方。通常对于不需要被直接查询余额的账户，可以考虑是否不记录余额，比如银行账户这种。

合并入账，就是先将余额更新操作hold住，暂存入待入账队列，累积到一定数量或者到一定时间时，将发生金额进行汇总合并，更新一次余额，从而大幅提升入账效率。合并入账借贷先后原则和前面类似，为了防范风险，只对入账

（负债类账户的“贷”）进行合并，出账仍然需要实时完成。

合并入账只能解决入账时的热点问题，而出账都是需要实时更新（为避免透支风险）。为了解决出账热点，需要引入多账户体系，即通过新建多个账户，把业务均摊到多个账户上，从而解决热点问题。

多账户不仅可以解决出账热点问题，对入账热点问题也有效，本质上多账户就是把一条DB记录的更新均分到了多个DB记录上。

按照账户的承载功能不同，多账户方案可以分成两种：

* 功能分离型多账户，出入帐功能分离，有的子账户承载入帐，有的子账户承载出账，有的承载两者。
* 功能完整性多账户，所有的分账户都同时承载出账和入帐。

多账户需要保存用户和多个账户的逻辑对应关系，这个对应关系可以放在业务层，也可以放在核心账户层。比如让商户多申请几个商户号，也是一种选择。

3）分布式事务带来的性能问题

如前文所提到的，资金账户系统不可避免地会引入分布式事务。分布式事务会导致可靠性以及性能都会受到影响（因为使用2PC等方式）。

这里的思路，一般都是通过降低ACID中的C（一致性）和I（隔离性）而提高A（可用性）。具体做法一般是在将一个分布式事务拆分成两个本地事务，即“先借后贷”，一个是“借”的实时事务，加一个“贷”的异步事务。当完成实时事务后就可以对外返回成功，异步事务后续驱动进行。本质是上通过MQ的事务消息追求达到最终一致性。但这样做也会引发一些副作用，在实际使用时需要根据业务场景进行选择。

3.5.3 存储可用性

关于存储的选型已经在5.3.2中讲过了。这里再强调一次，存储的可用性是整个资金账户系统可用性的基础。

而TDSQL作为一个高可用的分布式数据库，对于可用性的优势也体现在以下这些方面：

* **高性能。** 这样才能承受高并发的请求流量。
* **数据强一致。** 保证切换不同副本的情况下数据0丢失0出错。
* **支持透明分库分表。** 在分布式形态下，用户看到的逻辑表的实际物理存储可能是被打散分布到不同的物理节点上。希望做到对业务完全透明。这样一来作为开发者不需要关心分库分表细节。
* **支持不停机的容量弹性扩展。** 如果性能或容量不足以支撑业务发展时，升级过程中，开发者无需关心分布式系统内的数据迁移，均衡和路由切换。
* **支持跨区容灾、同城双活、跨城容灾等，故障自动切换其他副本，自动恢复，确保99.999%的可用性。**
* 最好有配套的**分布式事务解决方案。**

除了上面提到的这些点之外，定期的数据冷备也是必不可少的兜底措施。同样的，当选用其他存储时，如果以上这些点都能被全面考虑到，那么基本也就差不多了。

3.5.4 压测

这里注意：**要把压测当成对设计成果的验收，而不是作为发现问题的兜底手段。** 即将上线之前才发现的系统性问题，可能为时已晚。

但笔者还是建议在系统正式上线以前，在测试环境进行至少一次压测，目的有二。第一是验证可用性指标是否符合预期；第二是通过对于压测数据的监控观察和对账发现资金安全的漏洞，防范于未然。经过了压测环节，你的信心也将大大增强。

四、总结

本文以笔者在某一个跨境收单系统建设过程中实现的资金账户系统为例，探讨了在资金账户系统设计和实现中会遇到的问题以及相应的解决方案。可以肯定的是，**在资金账户系统建设过程中不会一帆风顺，还会遇到其他各种各样的问题，尤其是一些由于高并发场景引发的问题，在本场景中并不过多涉及**。这块笔者也还在持续探索和思考中。后续大家也可以多多和笔者探讨，本文也会持续更新。