[运维篇](#page510) [500](#page510)

[AIOps 在携程的践行](#page511) [501](#page511)

[记一个真实的排障案例：携程 Redis 偶发连接失败案例分析](#page520) [510](#page520)

[携程一次 Redis 迁移容器后 Slowlog“异常”分析](#page533) [523](#page533)

[数据库篇](#page540) [530](#page540)

[MySQL 锁之源码探索](#page541) [531](#page541)

[携程酒店订单 Elastic Search 实战](#page548) [538](#page548)

[携程数据库高可用和容灾架构演进](#page554) [544](#page554)

[风险控制篇](#page561) [551](#page561)

[基于红黑树的高效 IP 归属地查询方案](#page562) [552](#page562)

[携程基于大数据分析的实时风控体系](#page568) [558](#page568)

**数据库篇**

530

数据库篇



**MySQL 锁之源码探索**

**[作者简介]**姜宇祥，2012年加入携程，10年数据库核心代码开发经验，相关开发涉及达梦，MySQL 数据库。现致力于携程 MySQL 的底层研发，为特殊问题定位和处理提供技术支持。

锁是计算机程序运行时协调并发访问同一数据资源的机制。对于数据库系统来说，数据是一种供许多用户共享的资源，那么如何保证数据并发访问的一致性、有效性是必须解决的一个问题。所以，锁对于数据库来说，是非常重要的一个功能。通过各种锁，实现了数据库事务中的隔离性。本篇文章将从源码层面介绍 MySQL 的元数据锁和 InnoDB 的实现。

**一、MySQL 的架构与锁**

MySQL 在架构上分为两层，服务和存储引擎层。服务层集中了网络通讯、语法分析和计划生成等通用功能；存储引擎层主要负责数据的存储。元数据的并发管理集中在服务层，数据的并发管理在存储引擎层。因此对于元数据的锁在服务层进行实现，数据的隔离特性在存储引擎层实现。本篇将介绍服务层的元数据锁的实现，以及现下使用率最高的具有 ACID 特性的 InooDB 的数据锁。

**二、元数据锁**

**2.1 元数据锁类型**

****

**2.2 元数据锁申请与释放**

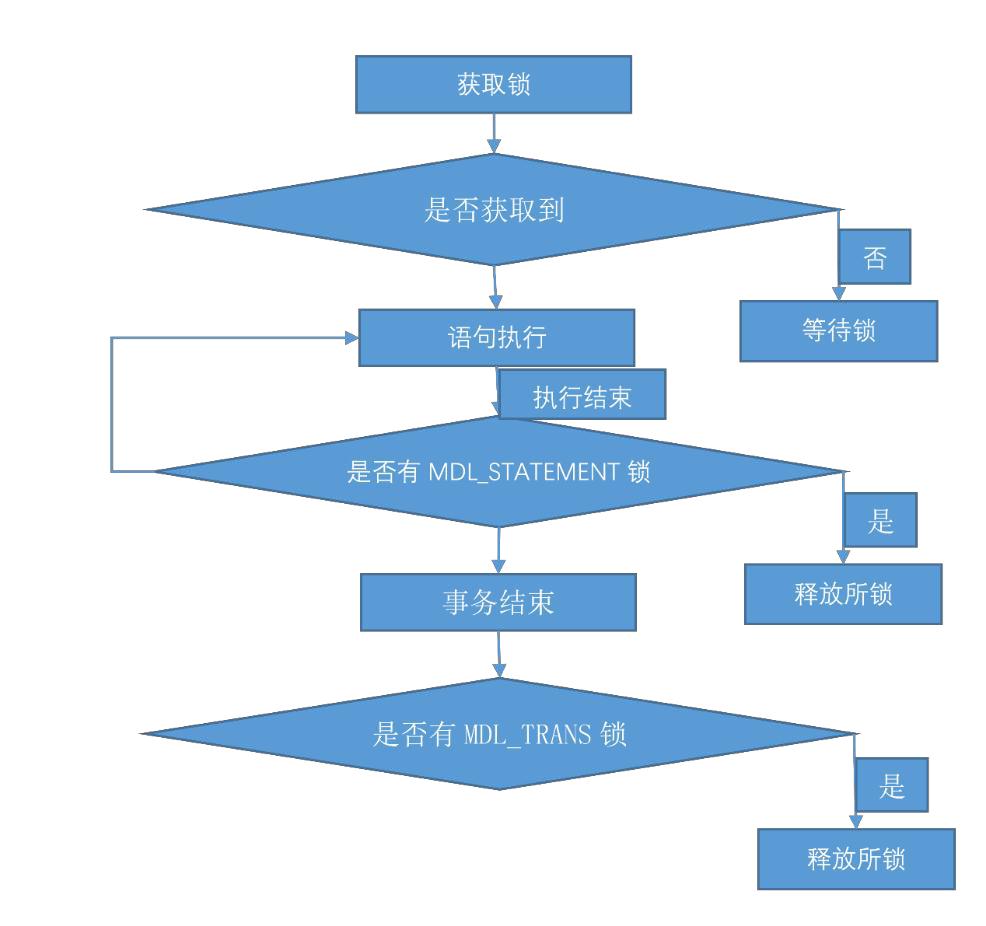
在申请元数据锁的同时，会指定锁释放的时间。在程序执行到指定位置时，如语句执行结束或者事务执行结束，会检查元数据锁的上锁情况，并释放那些需要在该位置释放的元数据锁。

531

数据库篇



抽象元数据锁的上锁和释放的过程，整理为如下流程图

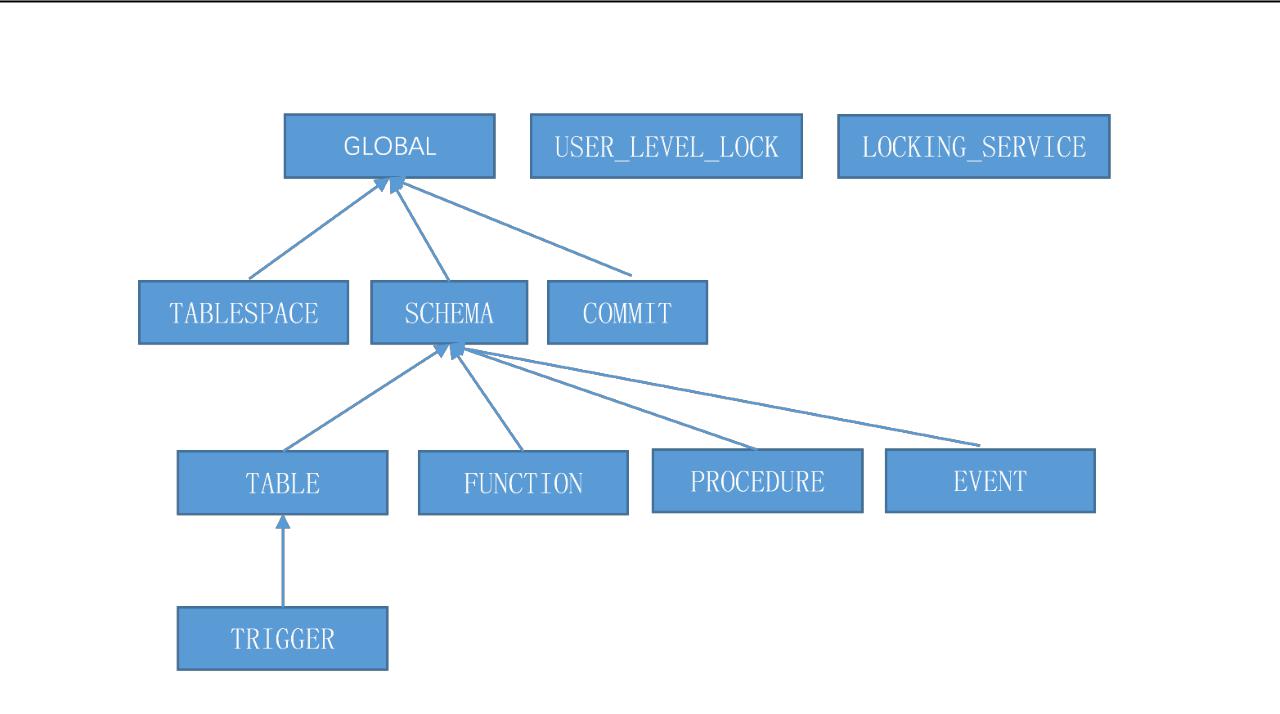


**2.3 元数据锁关系**

MySQL 的元数据也是有从属关系的。有些元数据进行上锁的同时，需要配合其他元数据锁，这里称这种关系为从属关系。这种从属关系如下图所示，其箭头所指方向为元数据锁所依赖关系。比如在为 SCHEMA 元数据加锁时，需要 GLOBAL 元数据锁。

532

数据库篇



**2.4 元数据锁级别**

由于对元数据的访问存在不同的需求，因此设置不同级别锁级别，用于对元数据及数据的访问控制。



**2.5 元数据锁源码**

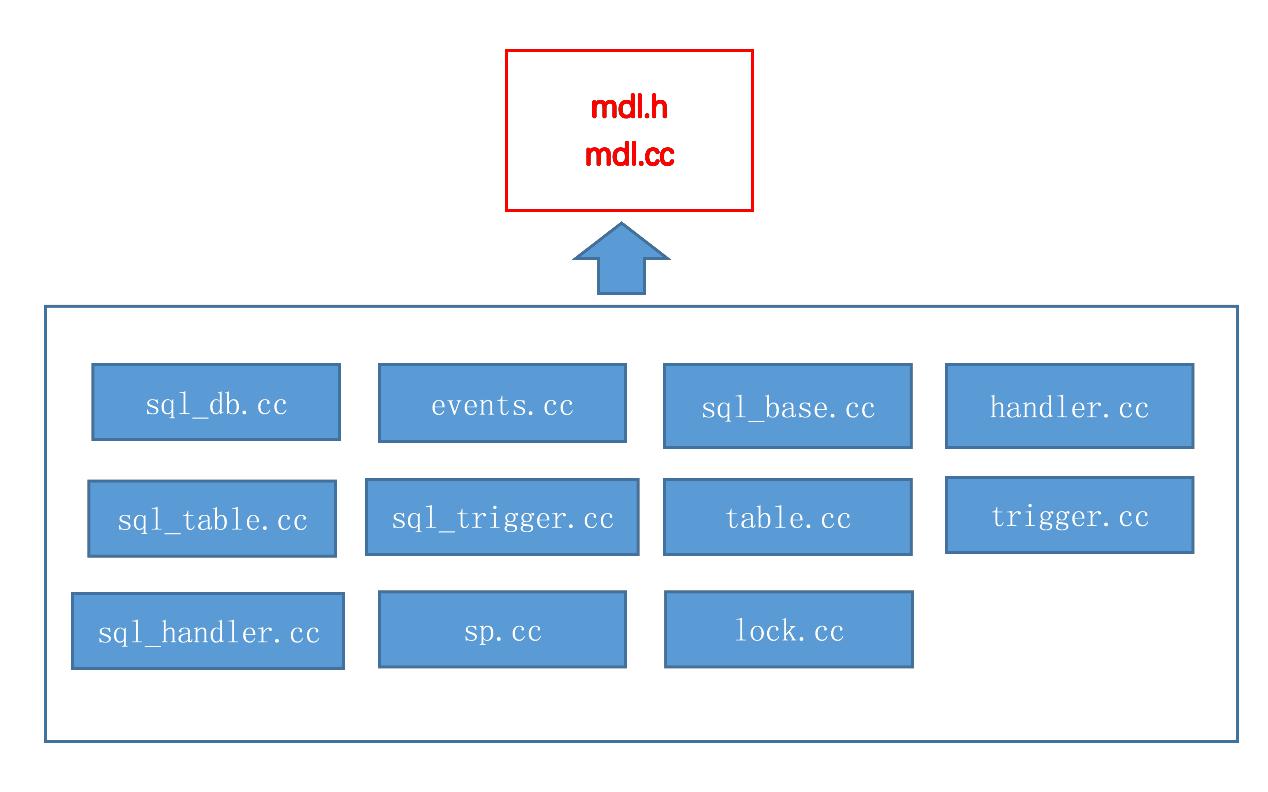
该部分介绍 MySQL 源码的主要源文件和主要函数。其中源文件以 mdl.h/mdl.cc 为核心，定义了元数据锁的主要数据结构和函数，lock.cc/sql\_db.cc 等源文件使用元数据锁所定义的数据结构和函数。

533

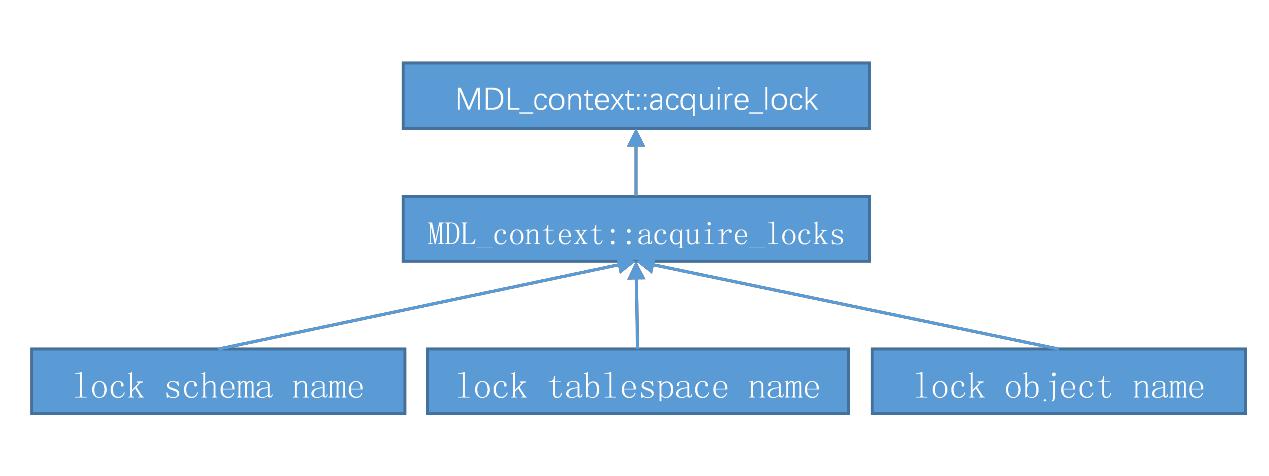
数据库篇



主要源文件及其关系如下，蓝框内所列源文件依赖于红框内的源文件所定义的内容。



主要函数如下图关系所列



**三、InnoDB 锁**

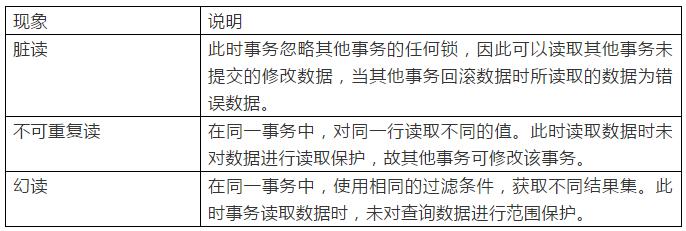
**3.1 事务隔离级简介**

存储引擎 InnoDB 实现事务的四个隔离级，也就是读未提交和读提交等四个事务隔离级。所谓事务隔离级，是并发访问中控制数据读写的方式。在这里先简单介绍这四个事务隔离级的来龙去脉，以便于理解 MySQL 的锁机制。

InnoDB 事务采用不同的锁机制，会产生不同的现象。

534

数据库篇

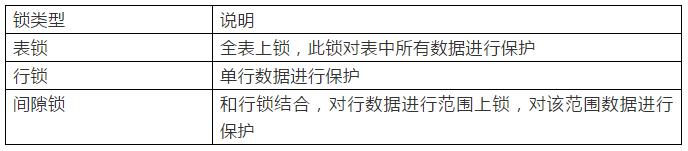


事务隔离级和各种现象的关系，“X”表示在该事务隔离级下现象可发生，“--”表示在该事务隔离级下现象不会发生。



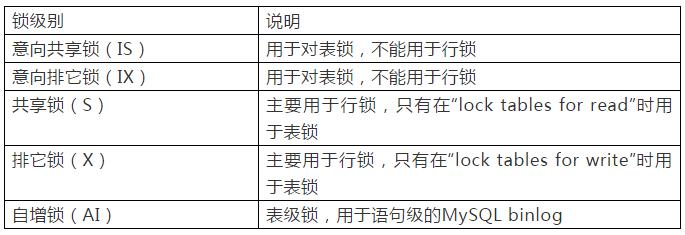
**3.2 InnoDB 的锁类型**

InnoDB 为保护并发访问下的数据，根据不同的粒度对数据进行。



**3.3 InnoDB 的锁级别**

InnoDB 的锁



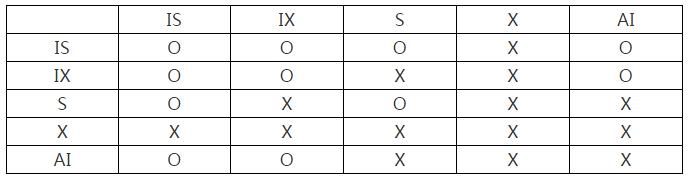
各个级别锁之间存在兼容性问题，如下表格列出各个级别锁之间的兼容性。“X”表示不兼容，

535

数据库篇



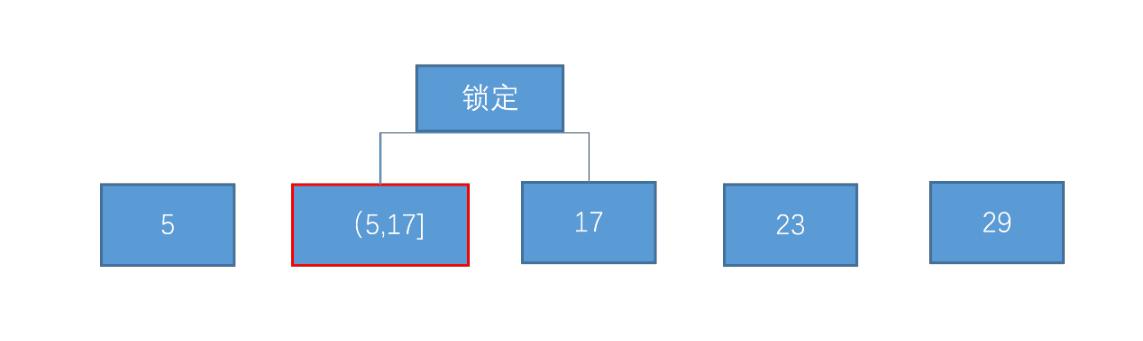
“O”表示兼容。



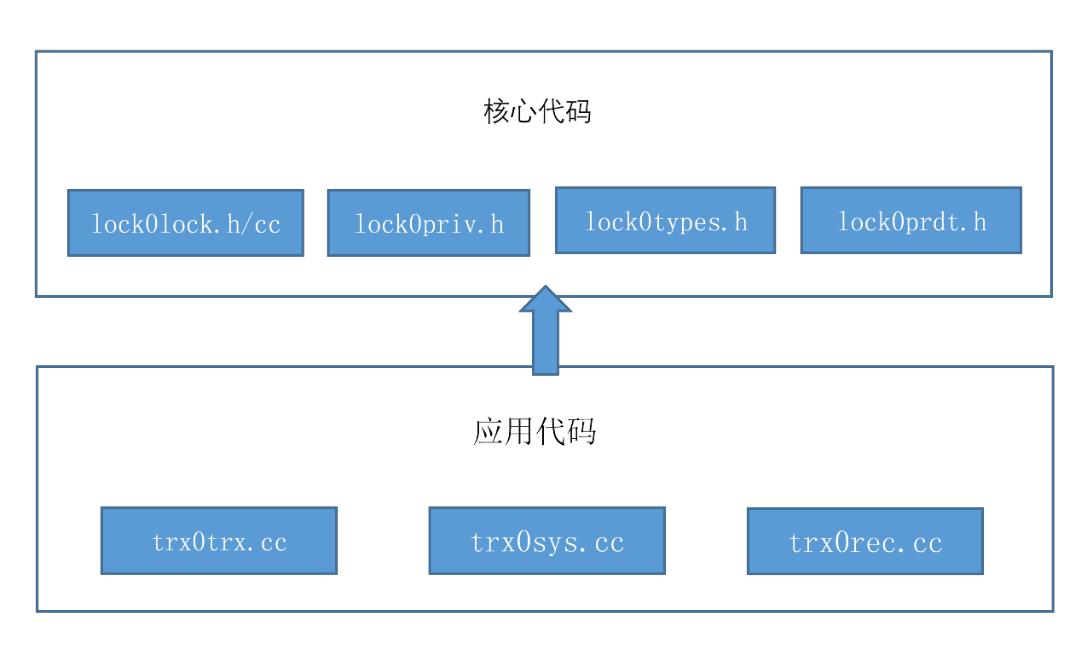
**3.4 间隙锁**

InnoDB 间隙锁（next-key lock）的用处是在 repeatable read 的隔离级下防止幻读现象的出现，所以一定要记住，在其他隔离级下是不会出现间隙锁的。间隙锁的原理是在通过对行锁进行特殊标识（此时的行锁就被称为间隙锁），指出在该范围内行记录的左开右闭区间被封锁，不可进行更新操作。正是采用这种针对行记录之间间隙上锁方式，所以称为间隙锁或者 next-key 锁。

如下图所示，假设索引中的 key 值为 5、17、23 和 29。当执行如下 SQL：begin; select \* from t4 where id2=16 forupdate；会对 key：17 创建一个行锁，并标识该行锁为间隙锁，其锁定区间为红框内 6 到 17，也就是这个区间的值域发生了变化，如果再发生变化可能会影响该区域的数据行集合，所以需要锁定该区域为不可更新。



**3.5 源码结构**

****

536

数据库篇



核心代码包含了有关锁的宏定义和函数定义等锁的类型定义和操作定义，应用代码为使用这些宏和函数的模块。

537

数据库篇



**携程酒店订单 Elastic Search 实战**

**[作者简介]**刘诚，携程酒店研发部技术专家。2014年加入携程，先后负责了订单处理多个项目的开发工作，擅长解决各种生产性能问题。

**一、业务场景**

随着订单量的日益增长，单个数据库的读写能力开始捉襟见肘。这种情况下，对数据库进行分片变得顺理成章。分片之后的写，只要根据分片的维度进行取模即可。可是多维度的查询应该如何处理呢？

一片一片的查询，然后在内存里面聚合是一种方式。可是缺点显而易见，对于那些无数据返回分片的查询，不仅对应用服务器是一种额外的性能消耗，对宝贵的数据库资源也是一种不必要的负担。

至于查询性能，虽然可以通过开线程并发查询进行改善，但是多线程编程以及对数据库返回结果的聚合，增加了编程的复杂性和易错性。可以试想一下分片后的分页查询如何实现，便可有所体会。

所以我们选择对分片后的数据库建立实时索引，把查询收口到一个独立的 web service，在保证性能的前提下，提升业务应用查询时的便捷性。那问题就来了，如何建立高效的分片索引呢？

**二、索引技术的选型**

实时索引的数据会包含常见查询中所用到的列，例如用户 ID，用户电话，用户地址等，实时复制分发一份到一个独立的存储介质上。查询时，会先查索引，如果索引中已经包含所需要的列，直接返回数据即可。如果需要额外的数据，可以根据分片维度进行二次查询。因为已经能确定具体的分片，所以查询也会高效。

**三、为什么没有使用数据库索引**

数据库索引是一张表的所选列的数据备份。

由于得益于包含了低级别的磁盘块地址或者直接链接到原始数据的行，查询效率非常高效。优点是数据库自带的索引机制是比较稳定可靠且高效的。缺陷是随着查询场景的增多，索引的量会随之上升。

订单自身的属性随着业务的发展已经达到上千，高频率查询的维度可多达几十种，组合之后的变形形态可达上百种。而索引本身并不是没有代价的，每次增删改都会有额外的写操作，同时占用额外的物理存储空间。索引越多，数据库索引维护的成本越大。所以还有其他选择么？

538

数据库篇



**四、开源搜索引擎的选择**

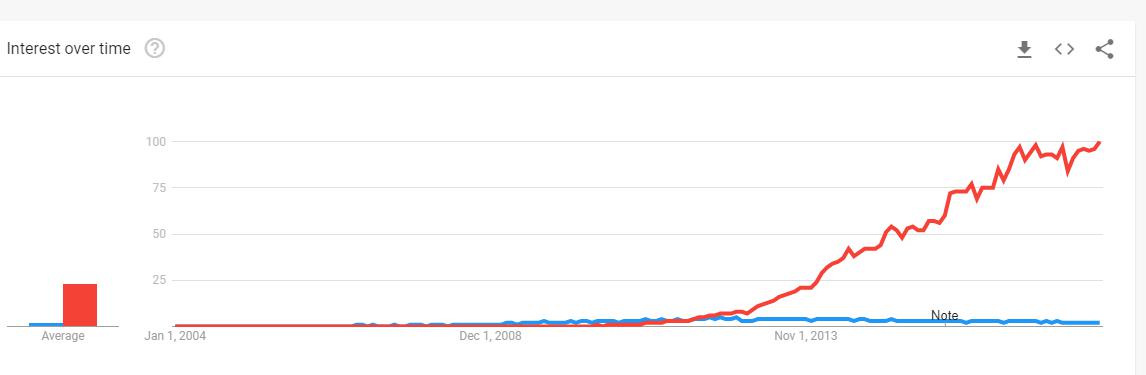
当时闪现在我们脑中的是开源搜索引擎 Apache Solr 和 Elastic Search。

Solr 是一个建立在 JAVA 类库 Lucene 之上的开源搜索平台。以一种更友好的方式提供 Lucene 的搜索能力。已经存在十年之久，是一款非常成熟的产品。提供分布式索引、复制分发、负载均衡查询，自动故障转移和恢复功能。

Elastic Search 也是一个建立在 Lucene 之上的分布式 RESTful 搜索引擎。通过 RESTful 接口和 Schema Fee JSON 文档，提供分布式全文搜索引擎。每个索引可以被分成多个分片，每个分片可以有多个备份。

两者对比各有优劣。在安装和配置方面，得益于产品较新，Elastic Search 更轻量级以及易于安装使用。在搜索方面，撇开大家都有的全文搜索功能，Elastic Search 在分析性查询中有更好的性能。在分布式方面，Elastic Search 支持在一个服务器上存在多个分片，并且随着服务器的增加，自动平衡分片到所有的机器。社区与文档方面，Solr 得益于其资历，有更多的积累。

根据 Google Trends 的统计，Elastic Search 比 Solr 有更广泛的关注度。



最终我们选择了 Elastic Search，看中的是它的轻量级、易用和对分布式更好的支持，整个安装包也只有几十兆。

**五、复制分发的实现**

为了避免重复造轮子，我们尝试寻找现存组件。由于数据库是 SQL Server 的，所以没有找到合适的开源组件。SQL Server 本身有实时监控增删改的功能，把更新后的数据写到单独的一张表。但是它并不能自动把数据写到 Elastic Search，也没有提供相关的 API 与指定的应用进行通讯，所以我们开始尝试从应用层面去实现复制分发。

**六、为什么没有使用数据访问层复制分发**

首先进入我们视线是数据访问层，它可能是一个突破口。每当应用对数据库进行增删改时，

539

数据库篇



实时写一条数据到 Elastic Search。但是考虑到以下情况后，我们决定另辟蹊径：

有几十个应用在访问数据库，有几十个开发都在改动数据访问层的代码。如果要实现数据层的复制分发，必须对现有十几年的代码进行肉眼扫描，然后进行修改。开发的成本和易错性都很高；

每次增删改时都写 Elastic Search，意味着业务处理逻辑与复制分发强耦合。Elastic Search 或相关其他因素的不稳定，会直接导致业务处理的不稳定。异步开线程写 Elastic Search？那如何处理应用发布重启的场景？加入大量异常处理和重试的逻辑？然后以 JAR 的形式引用到几十个应用？一个小 bug 引起所有相关应用的不稳定？

**七、实时扫描数据库**

初看这是一种很低效的方案，但是在结合以下实际场景后，它却是一种简单、稳定、高效的方案：

* 零耦合。相关应用无需做任何改动，不会影响业务处理效率和稳定性。
* 批量写 Elastic Search。由于扫描出来的都是成批的数据，可以批量写入 Elastic Search，避免 Elastic Search 由于过多单个请求，频繁刷新缓存。
* 存在大量毫秒级并发的写。扫描数据库时无返回数据意味着额外的数据库性能消耗，我们的场景写的并发和量都非常大，所以这种额外消耗可以接受。
* 不删除数据。扫描数据库无法扫描出删除的记录，但是订单相关的记录都需要保留，所以不存在删除数据的场景。

**八、提高 Elastic Search 写的吞吐量**

由于是对数据库的实时复制分发，效率和并发量要求都会较高。以下是我们对 Elastic Search 的写所采用的一些优化方案：

* 使用 upsert 替代 select + insert/update。类似于 MySQL 的 replace into，避免多次请求，成倍节省多次请求带来的性能消耗。
* 使用 bulkrequest，把多个请求合并在一个请求里面。Elastic Search 的工作机制对批量请求有较好的性能，例如 translog 的持久化默认是 request 级别的，这样写硬盘的次数就会大大降低提高写的性能。至于具体一个批次多少个请求，这个与服务器配置、索引结构、数据量都有关系。可以使用动态配置的方式，在生产上面调试。
* 对于实时性要求不高的索引，把 index.refresh\_interval 设置为 30 秒（默认是 1 秒）。这样可以让 Elastic Search 每 30 秒创建一个新的 segment，减轻后面的 flush 和 merge 压

力。

* 提前设置索引的 schema，去除不需要的功能。例如默认对 string 类型的映射会同时建立 keyword 和 text 索引，前者适合于完全匹配的短信息，例如邮寄地址、服务器名称，标签等，而后者适合于一片文章中的某个部分的查询，例如邮件内容、产品说明等。根

540

数据库篇



据具体查询的场景，选择其中一个即可。



对于不关心查询结果评分的字段，可以设置为 norms:false。



对于不会使用 phrase query 的字段，设置 index\_options: freqs。



541

数据库篇



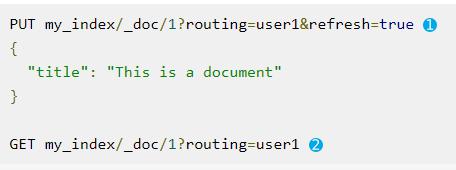
* 对于能接受数据丢失的索引或者有灾备服务器的场景，把 index.translog.durability 设置成 async（默认是 request）。把对 lucene 的写持久化到硬盘是一个相对昂贵的操作，所以会有 translog 先持久化到硬盘，然后批量写入 lucene。异步写 translog 意味着无需每个请求都去写硬盘，能提高写的性能。在数据初始化的时候效果比较明显，后期实时写入使用 bulkrequest 能满足大部分的场景。

**九、提高 Elastic Search 读的性能**

为了提高查询的性能，我们做了以下优化：

* 写的时候指定查询场景最高的字段为\_routing 的值。由于 Elastic Search 的分布式分区原则默认是对文档 id 进行哈希和取模决定分片，所以如果把查询场景最高的字段设为 \_routing 的值就能保证在对该字段查询时，只要查一个分片即可返回结果。

写：



查：



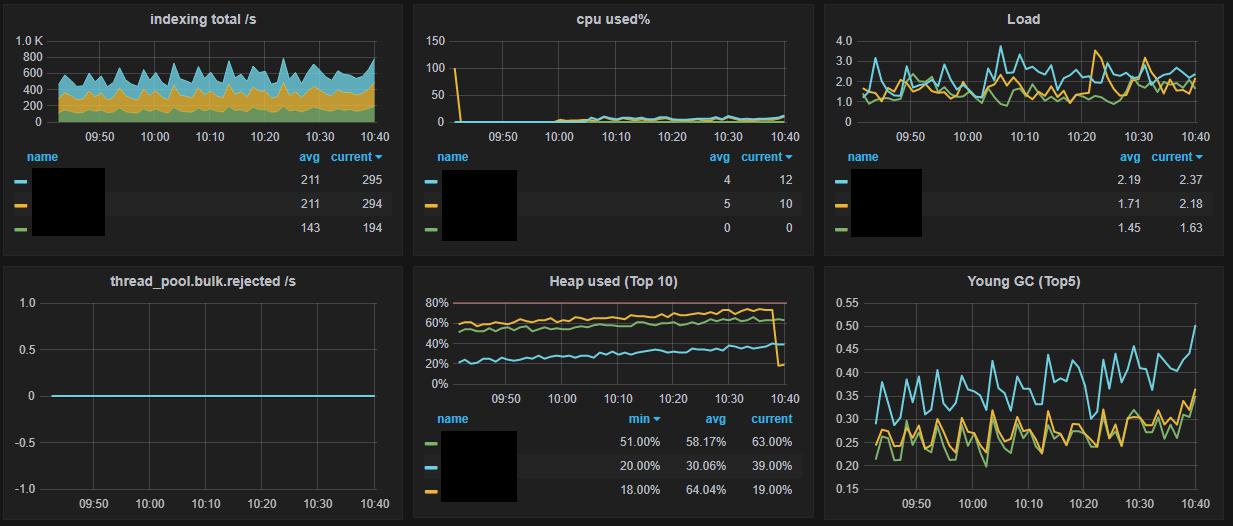
* 对于日期类型，在业务能够接受的范围内，尽可能降低精确度。能只包含年月日，就不要包含时分秒。当数据量较大时，这个优化的效果会特别的明显。因为精度越低意味着缓存的命中率越是高，查询的速度就会越快，同时内存的重复利用也会提升 Elastic Search 服务器的性能，降低 CPU 的使用率，减少 GC 的次数。

**十、系统监控的实现**

技术中心专门为业务部门开发了一套监控系统。它会周期性的调用所有服务器的 Elastic Search CAT API，把性能数据保存在单独的 Elastic Search 服务器中，同时提供一个网页给应用负责人进行数据的监控。

542

数据库篇



**十一、灾备的实现**

Elastic Search 本身是分布式的。在创建索引时，我们根据未来几年的数据总量进行了分片，确保单片数据总量在一个健康的范围内。为了在写入速度和灾备之间找到一个平衡点，把备份节点设置为 2。所以数据分布在不同的服务器上，如果集群中的一个服务器宕机，另外一个备份服务器会直接进行服务。

同时为了防止一个机房发生断网或者断电等突发情况，而导致整个集群不能正常工作，我们专门在不同地区的另一个机房部署了一套完全一样的 Elastic Search 集群。日常数据在复制分发的时候，会同时写一份到灾备机房以防不时之需。

**十二、总结**

整个项目的开发是一个逐步演进的过程，实现过程中也遇到了大量问题。项目上线后，应用服务器的 CPU 与内存都有大幅下降，同时查询速度与没有分片之前基本持平。在此分享遇到的问题和解决问题的思路，供大家参考。

**参考**

1）Elastic Search 官方文档；

2）<https://en.wikipedia.org/wiki/Database_index>

3）<https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B4%A2%E5%BC%95>

4）<https://logz.io/blog/solr-vs-elasticsearch/>

543

数据库篇



**携程数据库高可用和容灾架构演进**

**[作者简介]**郜德光，携程技术保障中心高级数据库经理，负责数据库相关的运维工作，参与了 SQL Server 和 MySQL 的高可用以及数据库容灾建设。喜欢钻研技术，对数据相关的技术一直保持着浓厚的兴趣。

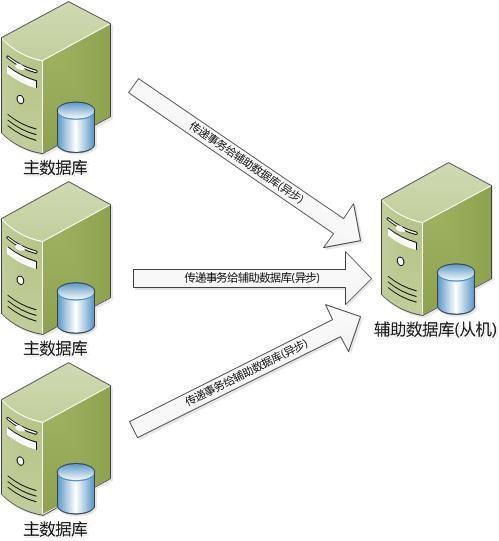
网站稳定性对任何一家公司来说都至关重要。业务长时间中断，不仅仅意味着收入的损失，更可能会失去客户。数据库的稳定性和 HA/DR 建设是网站高可用建设非常重要的一个环节。

本文分享了携程数据库（SQL Server，MySQL 和 Redis）的高可用和容灾的重构历程，以及重构的原因。也会简单分享一下 DR 切换工具，该工具可以一键将主站数据库切换到 DR 站点，用于在主站 IDC 故障时，快速恢复数据库服务。

**一、1.0 时代【1999~2008】**

自携程 1999 年成立到 2008 年左右，公司数据库产品主要是 SQL Server。

这个阶段是公司初创和快速发展时期，以优先发展业务为主。数据库的架构设计比较简单。高可用通过数据库的镜像技术来实现。镜像主要的架构如下图所示。甚至为了节省服务器资源，我们采用多个主数据库共享一个辅助数据库服务器的方式。



这种镜像方式搭建和运维都比较简单。主机如果出现故障，先尝试重启能否解决，如果不能恢复，则通过镜像切换的方式，切换数据库服务到从机。

544

数据库篇



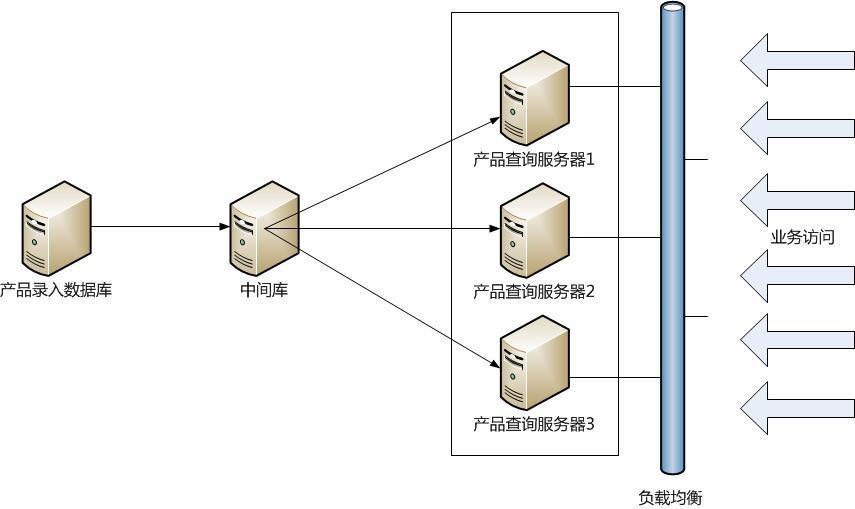
这种 HA 架构比较简单、粗糙，优点是不需要群集和共享存储等资源，成本低。缺点是主机故障时没法自动转移，业务恢复速度较慢，严格来讲还不具备 HA 的能力。

前期对于支撑当时业务发展来说已经够了，随着后期业务快速发展，已经无法满足业务稳定性的需要。

**二、2.0 时代【2008~2012】**

这一阶段，业务进入快速发展阶段，对数据库的依赖也越来越重。开始大量采用 SAN 共享存储来存放数据。单台数据库已无法支撑业务的压力，因此，也对数据库架构进行调整，引入了复制分发，用于读写分离。

架构如下图所示：产品的价格首先录入到录入数据库。通过 SQL Server 自有的表级别的复制分发技术，把数据传递到多个产品数据库，以供业务访问。如果业务访问压力大，则通过添加产品查询服务器的方式来进行扩容。



对于非读写分离场景的数据库，高可用是通过 Failover Cluster 群集方式。DR 依旧采用数据库镜像的方式。

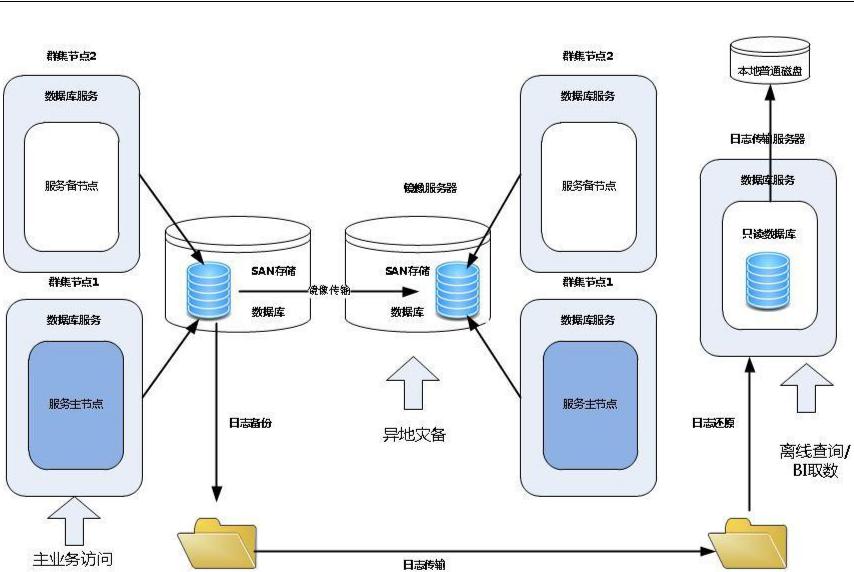
如下图所示：一旦服务器主节点硬件故障，则会通过自动故障转移，转移业务到服务备节点，切换时间大概在 2 分钟左右。主备服务器都连接后台共享存储。

同时，还对数据库服务搭建了镜像，一旦存储发生故障，主备服务节点都不可用的情况下，则通过切换镜像到镜像服务器上，镜像服务本身也是一个 Failover Cluster 群集，也做了高可用。

由于 SQL Server 的镜像服务器平时是不可读的，我们还通过一个日志传输服务，搭建了只读数据库，以供 BI 取数或查询。这个只读数据库也用于验证数据库备份的有效性。

545

数据库篇



由于业务的需求越来越复杂，数据库的架构开始逐渐复杂起来，主要体现在：

1）复制分发架构过于复杂，如城市表，基本上所有部门都需要使用，通过拉复制分发链路，把城市列表信息传递给其他部门的数据库，能快速解决问题，但随之而来，使得数据库的关系变成蜘蛛网状。维护起来相当困难。

2）过于依赖底层的 SAN 存储。SAN 存储内含相当复杂的存储技术和网络技术，复杂一点的问题就需要依赖供应商来解决。

2012 年，微软推出了 AlwaysOn 高可用性组，可以不依赖于共享存储。AlwaysOn 高可用性组同时也具备读写分离的功能，而且也能做容灾。所以，我们逐渐朝 AlwaysOn 这个架构演进，并用 SSD 来代替 SAN 存储。

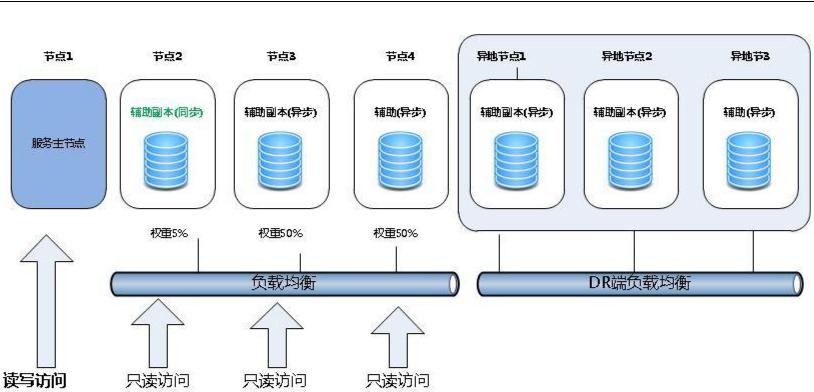
**三、3.0 时代【2012~2014】**

在 2014 年左右 AlwaysON 技术已经非常成熟，对于多 IDC 环境下支持也已经非常好，是 SQL Server 主流的 HA/DR 方案解决方法。

因此在 2014 年后，我们开始逐步把 SQL Server 改造为 Always ON 架构。架构如下图所示：写还是一个节点，但可提供多个节点的读。并且其中的一个节点是同步模式，用于做写节点的高可用。

546

数据库篇



纯 AG 架构

上面新的架构非常灵活。由于 2014 版本已经支持 8 个只读副本，AlwaysOn 延迟又低，因此完全解决了群集+镜像架构中遇到的复制分发读写分离架构瓶颈的问题，同时也不在需要为离线查询和 ETL 取数部署单独的只读数据库。读副本的备份功能也大大降低了备份时主机的压力。

在推进 Always ON 新架构过程中，我们也逐步用 SSD 来取代原有的 SAN。除了 SSD 的价格越来越便宜容量越来越大，相比 SSD，SAN 存储对 DBA 基本是黑盒子，需要专业的运维团队支持，运维成本更高。

**四、4.0 时代【2014~2018】**

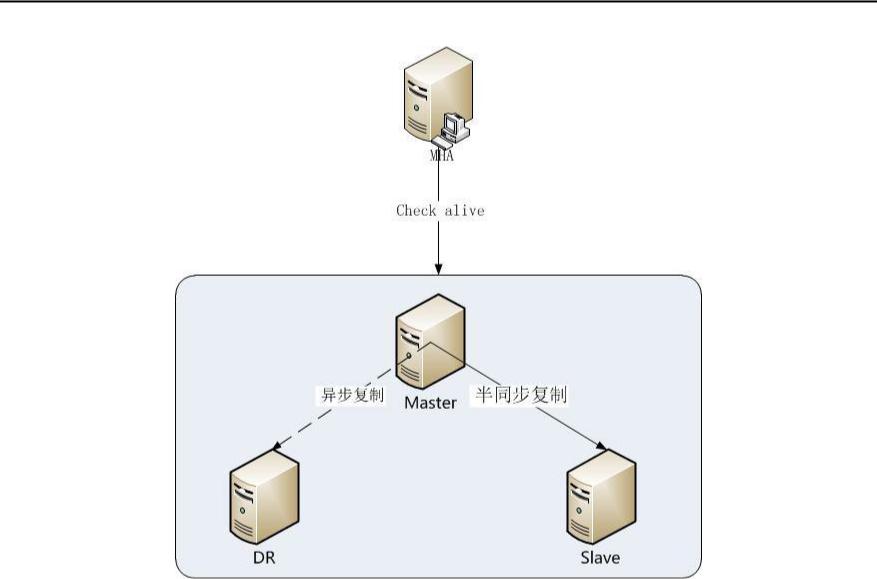
SQLServer AlwaysON 技术比较成熟，但其存在一个关键的问题，在于 AlwaysON 技术是闭源的。DBA 难以深入了解该技术的底层细节。虽然脱离了 SAN 存储的依赖，但还是依赖于 Windows 的 Failover Cluster 集群。

因此，2012 年开始在携程内部逐步推广开源的 MySQL 和 Redis 。在推广 MySQL 的时候，我们意识到 MySQL 的性能比不上 SQL Server, 所以同时推广数据库分库分表方案和前端 Redis 缓存。

MySQL 的高可用和 DR 是通过 MHA（Master High Availability）来实现的。具体架构如下：

547

数据库篇



通过 MHA 管理节点，来监听主节点的可用性情况，一旦发现主机不可访问，则切换到 slave 节点。MHA 的高可用模式，既兼顾到高可用，也兼顾到 DR。对于读写分离的需求，主要是通过分库分表方案来进行的。部分有采用读写分离。

对于 MHA 的方式，是通过域名/虚拟 IP 的方式，提供给业务使用。其最大的风险的是脑裂。主机没完全挂死，同时主从切换后，新的主机开始提供业务访问，就产生了脑裂。为了解决这个问题，我们在数据库访问层，引入了动态数据源技术，不通过域名/虚拟 IP 的方式，而是通过实 IP 的方式。提供业务访问，有效的降低了脑裂的风险，同时不通过域名解析，能加快主从/DR 切换时间。

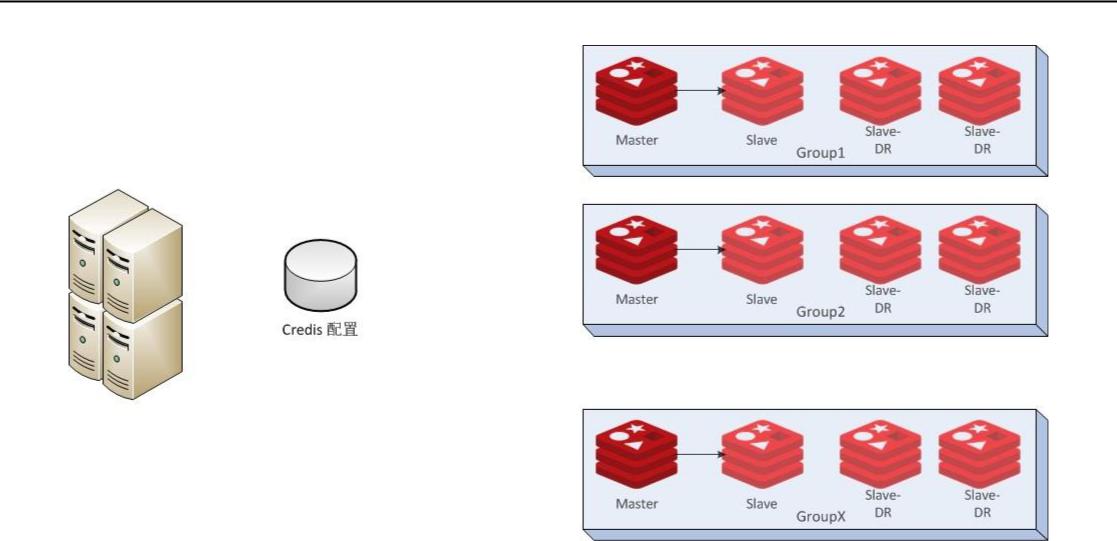
随着 MySQL 的引入，Redis 也逐渐广泛使用起来。由于 Redis 的 Key/Value 访问速度非常快，目前携程对 Redis 的依赖比对数据库还重要。Redis 本身需要有高可用和 DR 的方案。

对于 Redis，借助框架部门开发的中间件 CRedis，用于 Redis 的访问和高可用控制。

简要架构如下图所示：

548

数据库篇



说明如下：

1）业务访问的时候，通过引用框架的客户端组件读取配置信息。Redis 切换由哨兵发起，并且 Credis 配置能及时捕获，并通知客户端 IP 和端口的变化。

2）对 Redis 拆分多个 Group，以避免访问热点。每个 Group 包含多个 Redis 实例。

3）Redis 实例 Master/Slave 在一个机房， 另外两个实例 Slave-DR 在另个机房。

4）访问请求通过 CRedis 配置路由到指定的分片。

5）Slave-DR 实例和主实例的数据同步通过框架同步工具 XPipe 来实现。XPipe 已经开源，更多信息请访问：https://github.com/ctripcorp/x-pipe。

为应对日常 DR 演练以及硬件故障时快速恢复业务的场景，DBA 设计开发了集中、一键式 DR 自动化切换工具，支持所有数据库产品。用来帮助 DBA 快速、安全的完成数据库切换。

DR 切换工具支持不同的切换维度，覆盖了所有的场景：

1）单个或多个数据库群集，应对单机故障或日常维护等场景；

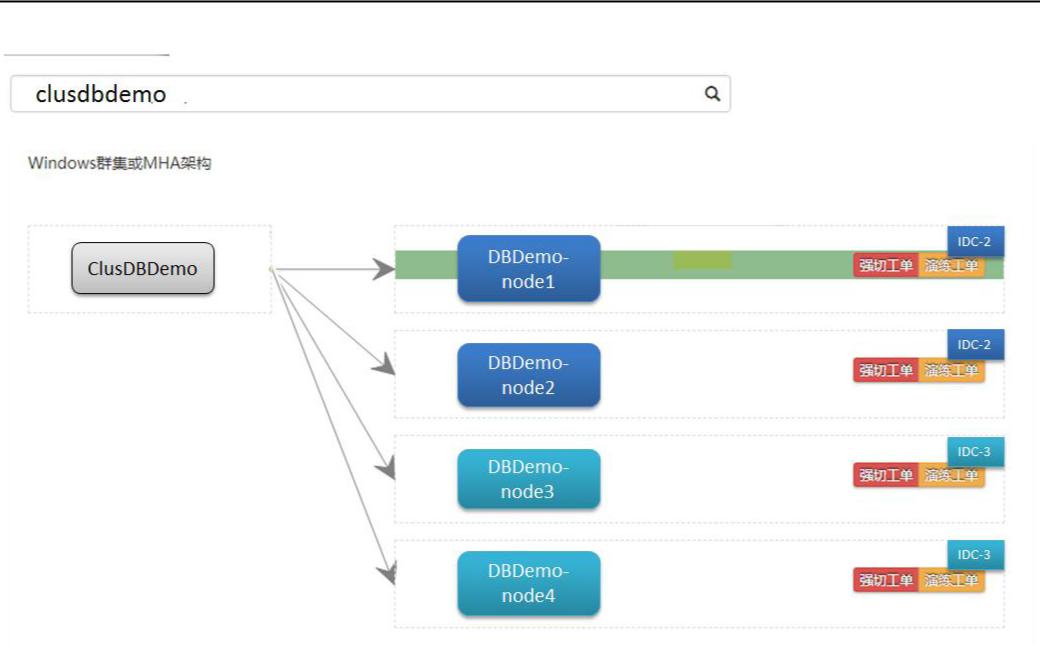
2）单个业务线下所有数据库群集，应对 DR 切换演练场景；

3）IDC 下所有数据库群集，应对主 IDC 故障场景；

目前为止已经利用 DR 工具成功完成了 200+次的切换演练。下面是用 DR 工具准备单群集切换的一个例子：

549

数据库篇



如上图所示，工具中搜索到对应的群集后，工具会根据元数据信息展示群集的整体架构：节点（名字、IP 地址等）、IDC、复制关系。

每个节点提供 2 种切换方式：

1）强切工单，强制切换。主机或主站 down 时，可能有数据丢失；

2）演练工单，正常切换。主机可用时，演练或计划内主机维护时使用；

生成的工单后续可以自动执行。

DBA 把 DR 复杂的切换恢复流程全部集成工具里面，工具同时支持 API 根据不同维度快速批量生成切换工单，支持并发切换，可以快速完成业务的切换恢复。

为保证 DR 工具的高可用性，在设计 DR 工具时也消除了工具对单个 IDC 的依赖。只要保证一个 IDC 可用，就可以通过工具发起切换。

从以上架构的演变过程，我们可以看到是从简单—复杂—简单的一个演变趋势，但是稳定性和可用性有了质的飞越。数据层本身变得越来越简单，而扩展了大量的辅助架构，来提升自动化运维能力。

希望以上的分享对大家有所帮助。

550

风险控制篇



**风险控制篇**

551

风险控制篇



**基于红黑树的高效 IP 归属地查询方案**

**【作者简介】**邢钦华，携程风控团队高级研发经理。2016年加入携程，是风控大数据平台Chloro 的设计和开发的主要参与者。专注于大数据流式处理和用户行为分析在互联网风控领域中的应用。

在实时风控系统中会涉及到非常多的数据衍生和解析，比如 IP 归属地解析、手机号归属地解析、银行卡卡 BIN 解析等等。以 IP 归属地为例，传统的实现 IP 归属地查询的方法是把 IP 地址信息存储到关系型数据库中，对于并发量比较少，实时性要求不高的情况下是可行的，但是一旦并发量增大时，会对关系型数据库产生很大的压力，并且访问速度会明显减慢，因此对于高并发、实时性要求高的场合这种查询方法就显得力不从心。

本文我们将以 IP 归属地为例，介绍一下携程风控是如何实现相对静态数据的高效衍生的。我们会把 IP 归属地信息保存到内存中，经过一系列的转变，最终形成红黑树，利用红黑树高效的查找性能，实现了高效的 IP 归属地解析方案，该方案可承担较大的并发访问压力，并拥有极低的响应延时。

实现方案：

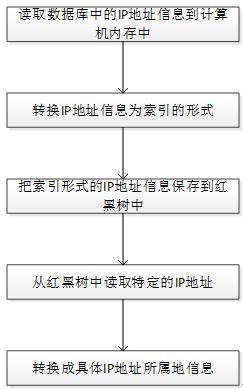


图 1

552

风险控制篇



如图 1 所示，首先把 IP 地址信息录入到数据库中，系统把已经录入好的 IP 地址信息从数据库中读取到计算机内存，经过一系列的索引形式的转换，把最终的索引以及把 IP 地址转成 long 形式的整数后存放到计算机内存中的红黑树中，当有访问请求获取 IP 的归属地信息时，首先把具体的 IP 地址转成 long 形式的整数，根据此证书到红黑树中查询到其对应的结点，获取该结点的索引数据，再根据该索引数据获取到 IP 归属地信息，并且返回给用户。

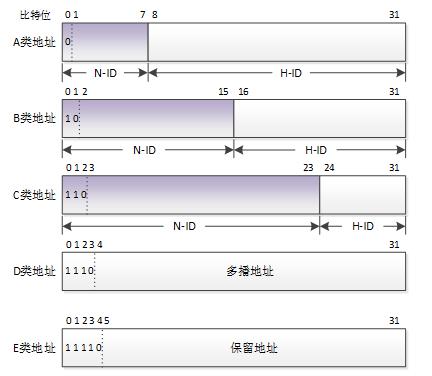


图 2

如图 2 所示为 IP 地址分类图，在 TCP/IP 协议中，IP 地址以二进制数字的形式出现，总共 4 个字节，即 32 个 bit，由网络编号（N-ID）和主机编号（H-ID）组成。根据网络地址的不同，IP 地址可以分为五类： A 类地址、B 类地址、C 类地址、D 类地址以及 E 类地址。

对于同一个物理网络上的计算机主机，其网络编号是相同的，不同的是主机编号。在为各个城市分配 IP 地址时，通常是把多个连续的 IP 地址段分配给某一城市，例如：1.12.0.0 到1.12.255.255，1.15.168.0 到 1.15.191.255 等连续的 IP 地址段都属于北京的 IP 地址。通常每个城市包含了多个连续的 IP 地址段，在这些 IP 地址段中的 IP 地址都属于该城市的 IP 地址，由于 IP 是有 4 个字节组成的，并且没有负数，可以把 IP 地址段转成两个 8 字节的 long 类型整数，在这两个整数之间的数字都属于该城市的 IP 地址。

IP 地 址 是 由 4 段 组 成 的 ， 如 1.15.168.0 ， 其 对 应 的 4 字 节 二 进 制 形 式 为00000001.00001111.10101000.00000000，根据计算机的计算特性，可以把第一段左移 24 位、第二段左移 16 位、第三段左移 8 位、第四段不移动，得到整数相加就是 IP 地址转换后的整数值，即 16777216 + 983040 + 43008 + 0 =17803264。

553

风险控制篇

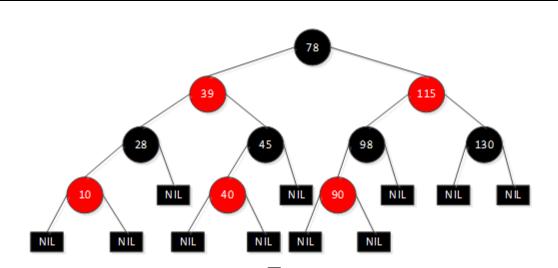


图 3

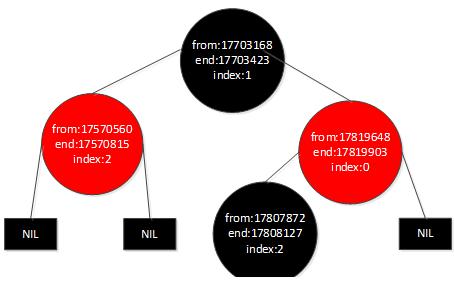


图 4

红黑树是一种非常成熟的数据结构，是每个结点都带有红色或者黑色的二叉查找树，是比较高效的，可以在 O(log n)时间内做查找，插入和删除，这里的 n 是树中元素的数目。红黑树在满足二叉查找树的要求外，还必须满足一下要求：

1、节点是红色或黑色。

2、根是黑色。

3、所有叶子都是黑色（叶子是 NIL 节点）。

4、每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）

5、从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

JSON 是一中数据格式，主要用于数据交换，易于人阅读和编写，同时也易于计算机解析和生成。

本系统中，IP 地址的归属地信息包含了国家（country）、地区（region）、市（city）等属性。

554

风险控制篇



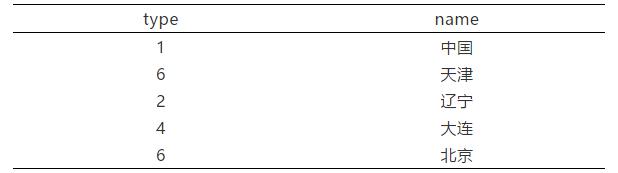
首先把 IP 地址信息从数据库中以 JSON 格式读取到计算机内存中，例如：

{"region":"天津","start":"1.15.232.0","end":"1.15.232.255","city":"天津","country":"中国"}，{"region":"辽宁","start":"1.14.33.0","end":"1.14.33.255","city":"大连","country":"中国"}，{"region":"北京","start":"1.15.186.0","end":"1.15.186.255","city":"北京","country":"中国"}，{"region":"北京","start":"1.12.27.0","end":"1.12.27.255","city":"北京","country":"中国"}，

其中 start 为数据该城市连续 IP 段的起始 IP，end 为结束 IP。

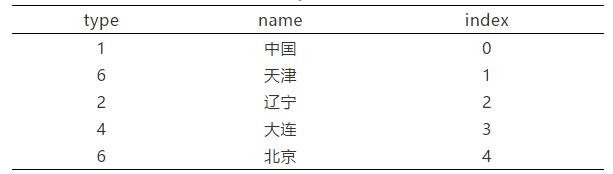
然后把这些 IP 归属地信息封装成 Area 类的集合。Area 类由 type 和 name 字段组成，其中 name 表示一个国家或者地区或者城市的名称，比如上面的 IP 地址信息中的中国、天津、北京、辽宁和大连。type 是由 country、region、city 单独或者任意相加组成，其中 country、region、city 分别用 1、2、4 表示，这样当 type 为 1 时表示一个国家，为 2 时表示一个省，为 4 时表示一个城市，为 3 时表示国家名和地区名相同，为 5 时表示国家名和城市名相同，为 7 时表示国家、地区、城市的名称相同。转换后的数据如下所示：

表 1



然后再把上面的 Area 类的集合数据转成 raw-meta 数据，其中 index 位 Area 类的集合的索引。

表 2



然后再根据 IP 地址信息和表 2 中的 raw-meta 数据转换成 fomatted-raw-ip 数据，其中国家索引为 IP 地址信息中 country 字段对应的表 2 中 index 列的相应值，地区索引为 region 字段对应的表 2 中 index 列的相应值，城市索引为 city 字段对应的表 2 中 index 列的相应值。

表 3

555

风险控制篇



进一步，表 3 中第 3、4 行的国家索引、地区索引，城市索引是相同，都是国家为中国，地区为北京，城市为北京，为了消除重复数据，再把 fomatted-raw-ip 数据转换为 segment-regions-ip 数据。

表 4



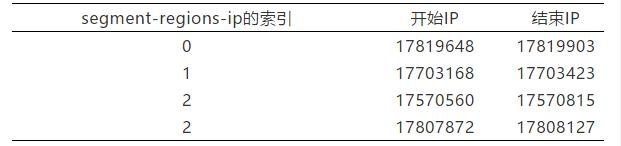
然后再把 IP 归属地信息和 segment-regions-ip 数据转为 compact-ex-ip-segment 数据，其中第一行为 segment-regions-ip 的索引。

表 5



表 5 的 IP 地址转换成 long 整数。

表 6



最后把 compact-ex-ip-segment 转成红黑树保存在计算机内存中，每个红黑树结点由 parent，left，right，color，from，end，index 组成。parent 为父节点，left 为左结点 right 为右结点，color 表示该结点为红色或者黑色，from 为起始 IP 转成的 long 整数，end 为结束 IP 转成的 long 整数，index 为 compact-ex-ip-segment 数据保存的索引，即表 6 的第一次列。

556

风险控制篇



由于红黑树中存放的是 IP 段的起始 IP 转换后的整数和结束 IP 转换后的整数，而需要查询的是具体 IP 地址转换后的整数，因此查询的规则是：先把 IP 转换为整数，从红黑树的 root 结点开始查起，当该整数小于结点中的 from 整数时，继续沿着红黑树该结点的左边查找，当该整数大于结点中的 end 整数时，继续沿着红黑树中该结点的右边查找，否则，该查找到的结点即为要查找的 IP 信息对应的结点。

若查找过程中，结点为 NIL 节点，说明该 IP 地址不是有效的 IP。例如 IP 地址为 1.15.186.10，首先把 IP 转成 long 型的整数，即 17807882。然后去红黑树中查找该整数对应的红黑树中的结点为 2，17807872，17808127，进而取到索引 2，即 segment-regions-ip 的索引，根据表 4 取到数据 0,4,4,2，其中 2 为 fomatted-raw-ip 集合的索引，0,4,4 为 raw-meta 数据的最后一列，即为 Area 类集合的索引，从而找到 country 为 0，即中国，region 为 4，即北京， city 为 4 即北京。因此该 IP 对应的国家为中国、地区为北京、城市为北京。

当红黑树形成以后，在具体 IP 查询过程中，从数据库中读取的 IP 地址信息的 JSON 格式数据已经不再需要，可以从内存中删除。最终留在内存中的数据为 Area 类的集合，segment-regions-ip 数据，红黑树，这样就可以减少计算机内存的使用。经统计，本系统中 IP 地址信息的总条数为 719296，经过对系统启动前后，计算机内存的比较，系统启动后共占用了大约 20 兆（MB）的内存，在现在计算机技术快速发展的时代，一般家用的计算机的内存也有 2 千兆（GB）或者更多，公司专门的服务器的内存甚至高达几十 GB，因此这些数据的内存占用量可以说是微不足道的。

当 IP 地址信息的条数增加时，只需要以目前的格式添加到数据库中，然后重启应用程序即可，当需要更详细的 IP 地址信息时，比如经纬度、运营供应商，除了在数据库中添加相应信息外，只需要增加 Area 类的相应信息的字段，重启应用程序即可，因此，此系统具有良好的扩展性。

该方案不仅适用于 IP 归属地查询，也适用于其他相对静态的数据的快速解析。

557

风险控制篇



**携程基于大数据分析的实时风控体系**

**【作者简介】**刘江，携程金融管理部风险管理总监，负责携程集团的全面风险管理工作。拥有近 15 年风险管理经验，先后在广发银行、OperaSolutions、阿里巴巴和腾讯等公司任重要管理岗位，一直从事风控政策、风控模型、大数据征信等相关工作。

携程反欺诈体系经过超过 10 年的发展和积累，在大数据实时并行计算和实时多维关联分析方面已经非常成熟，是整个体系稳定高效运行的基础。

近两年来，我们在大数据和人工智能方向投入研发资源，产出了设备指纹、CDNA、实时复杂变量计算引擎等一系列创新项目，取得到很好的应用效果。2017 年整体卡 BP 降低 50%以上，远低于同行平均水平，为携程业务的发展以及全球化化进程提供了有利条件。

**一、性能和复杂度可以兼得**

携程的风控系统，和大部分第三方支付平台一样，也是以实时风控系统为主：

1、支付环节一般留给风控校验的时间不会超过 1s，业务风控点上更是希望风控能在 100ms 内就能通过；对性能的追求，也是对极致用户体验的追求。

2、携程近两年每年的订单增幅在 50%以上，营销活动、恶意占资源等业务风控的干预量更是每年 10x 以上的幅度增长。

3、规则数量两年翻了五倍，同时规则使用更多的数据不再仅限于产品信息、支付信息、账号信息，行为数据等弱关联数据开始大量的应用于规则分析。

4、在实时风控场景里大量部署复杂模型，使模型也能和规则一样能直接拒绝交易；平均来看、执行一个模型以及相关的变量计算所需的资源可能与 200 条普通规则相当，对系统的架构和性能都是很高的挑战。

5、欺诈份子的技术也在不断进步，更隐蔽，我们需要更多的数据来识别，比如对模拟器的识别、对代理服务器的识别，都投入了不小的研发资源。

给大家看一些数据：

一笔支付请求背后，携程风控的规则计算复杂度：

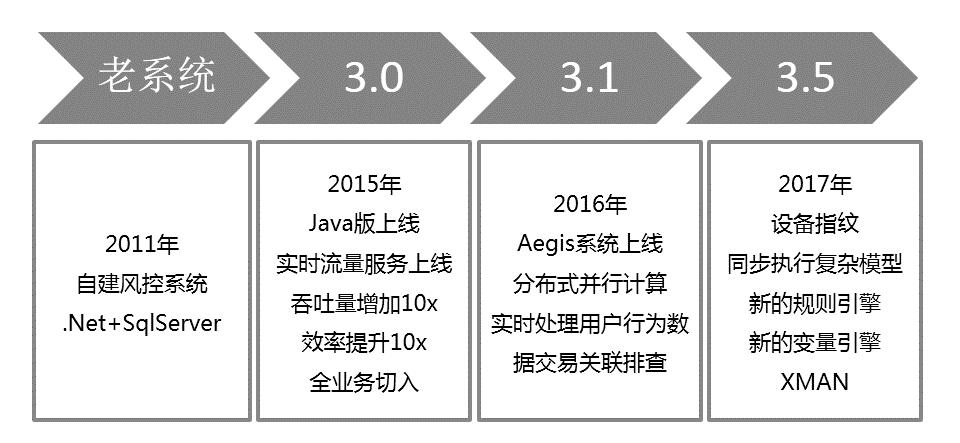
558

风险控制篇



期间计算生成的变量个数接近 2000 个，90%以上的变量是 Velocity 和 Ratio 类型的变量，甚至较大一部分是精确到当笔交易的；执行完整个规则校验，风控返回给支付系统通过或拒绝的指令，平均耗时不到 150ms，99.9%线也只有 500ms 左右。

**二、携程风控架构变迁简史**

****

携程自建风控系统开始于 2011 年左右，直到 2015 年正好赶上公司技术栈从.Net 往 Java 平台转变，风控系统也迎来了一次完全的重写。

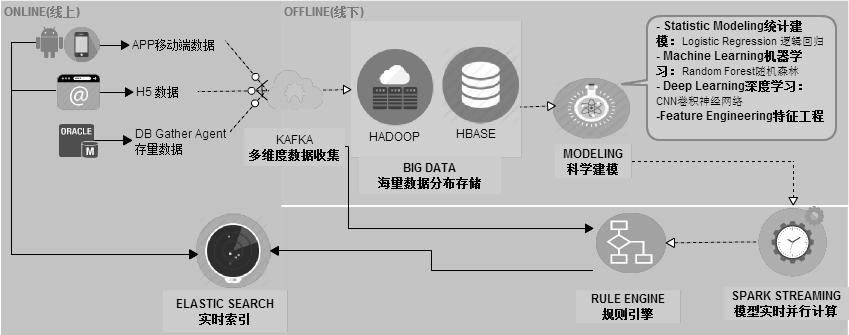
新系统的架构、设计复杂度、预计的处理能力也充分考虑了公司的业务发展预期，第一次让技术走在了业务到来之前。经过每年一个大版本的迭代，到目前为止，携程风控的技术水平已经处于行业第一梯队。

**三、架构概述及核心服务**

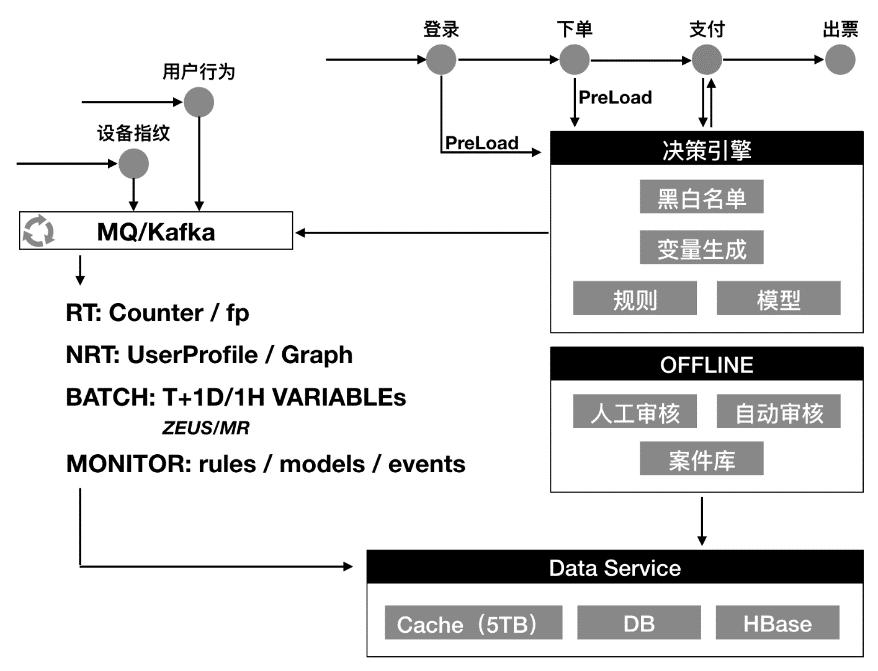
下面我们看看携程风控的架构实现：

559

风险控制篇



上图可能有点抽象，我们看一个具体的例子：



概念：登录／注册、下单、支付、支付结果通知、出票等等这些我们称之为风控接入点。

有些接入点是做实时校验用的、有些是收集数据用的，在携程整个大系统内一共有超过 400 个风控接入点，审核或监控携程交易的每一个环节，保障着每一笔交易的安全和用户的利益。

每天风控收集上来的数据超过 50 亿条，其中超过 1 亿左右的请求需要风控实时校验风险并返回给业务系统当前操作是否可以继续。

用户从登录开始风控就已经开始在介入，在用户浏览、下单的过程中，对这个用户的风险评

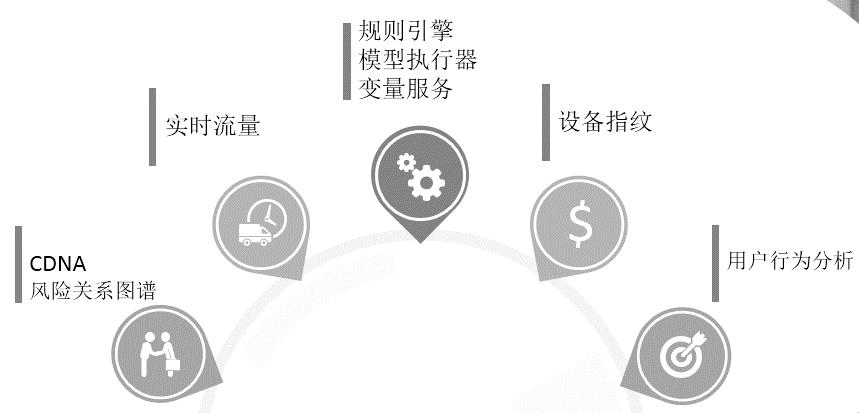
560

风险控制篇

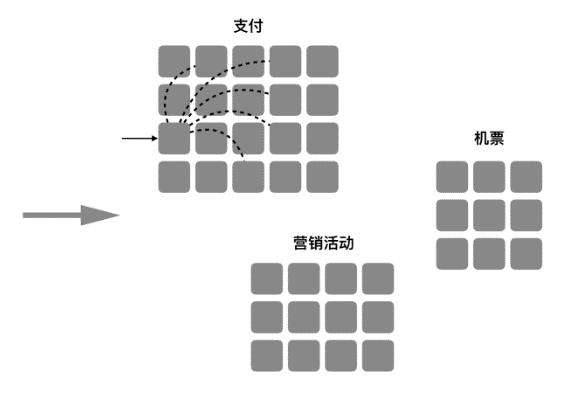


估和计算一直在持续，等到用户发起支付请求时，风控的热数据里已经有了完整的关于这个用户画像数据，风控引擎可以在这些数据的基础上实时计算和衍生出规则和模型需要的变量。

支撑风控系统的高可用、高性能，离不开强大的基础设施，下面我向大家展示一下携程风控的几个核心服务和组件：



**3.1 风控引擎**

****

我们给他起了一个名字叫 Matrix，意思是像魔方一样灵活多变。 数以千计的规则是分布式并行执行的、以保证规则数量和执行耗时没有明显的正相关性；并且风控引擎可以按业务动态分组，既保证了业务之间良好计算资源的隔离性、也提供了足够的灵活性。

561

风险控制篇



**3.2 规则引擎**

初始版本基于 drools 实现，不过经过两个版本的迭代优化后，已经完全替换成自主研发的引擎，新引擎兼容 drools 的脚本，迁移到新引擎几乎零成本。迁移后规则执行性能提升一个数量级以上且具有更好的稳定性。

**3.3 模型执行引擎**

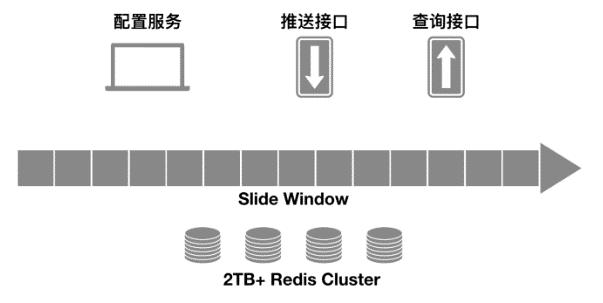
风控引擎支持把 SAS 或 SPARK 等工具训练出来的模型直接在风控系统中部署，支持 DOT 和 PMML 等多种格式。

我们自主实现了 DOT 模型文件的解释器，执行效率相对于 Python 执行提升 20 倍以上。



**3.4 实时流量服务**

内部称为 Counter Server，负责衍生计算所有 Velocity 变量和 Ratio 变量，重要性不言而喻， Counter 的性能直接影响到整笔交易的耗时和准确性。



我们基于 Redis 集群构建了一个 Slide window，实现上其实很轻量，但确是很好用，把时间窗口的刻度映射到了 redis 的 key 上，目前支持秒、分钟、小时、日、月等的精度。可以根据变量的要求灵活、动态的配置各类实时统计项。目前集群容量在 2-5TB 之间。

562

风险控制篇



Counter 服务每天支撑了超过 100 亿次查询，单次流量查询的平均耗时仅 1ms 左右，保证了变量衍生的可靠性。

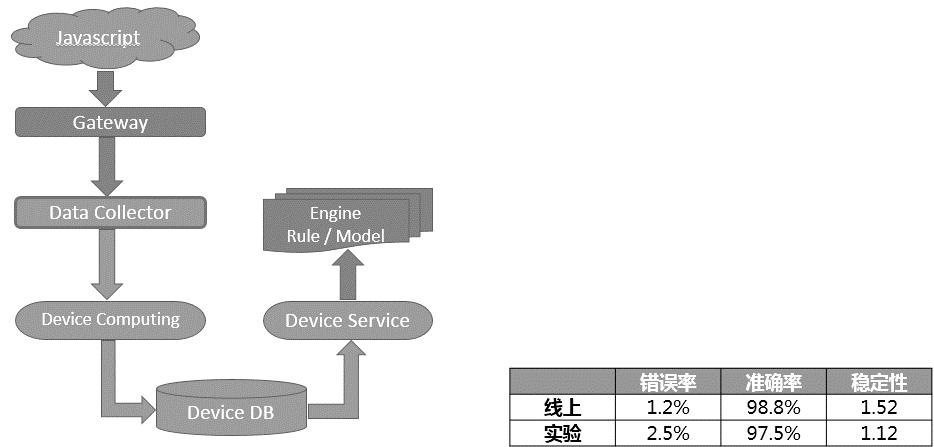
**3.5 设备指纹**

传统上都用 IP 来标识一个设备，但是随着移动网络的普及，IP 基本已经失去这个功能了，你取到的很多都是基站 IP、出口 IP，封掉一个 IP 可能会误杀一片。

在 APP 里可以使用 IMEI 或 IDFA 硬件 ID 来识别设备，但在 PC 和 H5 需要一个比 IP 更准确的设备识别标识。已经有一些公司走在了前面，比如业内知名的 ThreatMetrix、国内也有几家专业做设备指纹的服务商。

设备指纹是风控识别欺诈交易的关键技术，此类核心技术要掌握在自己手里，携程风控研发的设备指纹服务，已经在携程全站部署以及携程集团旗下的多个站点部署，应用后规则抓取准确性提升非常明显。

设备指纹的架构及关键指标：



**3.6 CDNA**

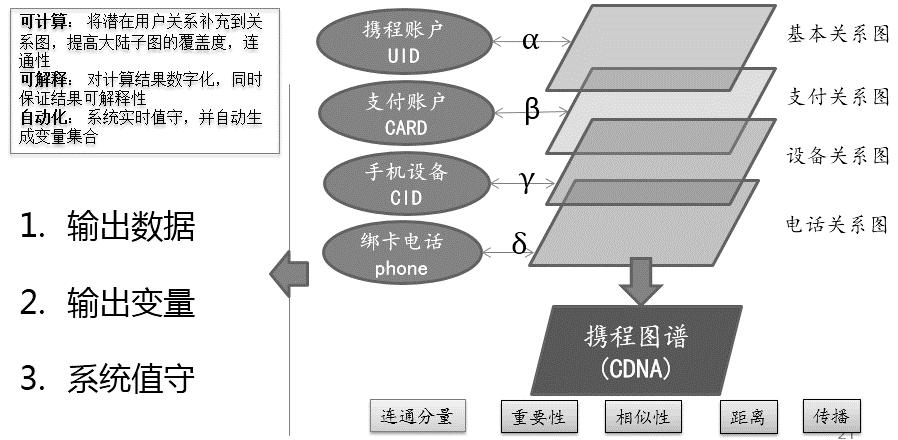
我们需要完整且深入的了解对于同一个人或同一类欺诈团伙在携程“一生”的行为以及“足迹”。

基于此目标，研发了 CDNA 服务，通过对所有流经风控的数据进行多维度的无限极收敛关联，把同一个人的数据聚合在了一起；CDNA 服务每天处理超过 100TB 的数据。

通过 CDNA 对于发现新的欺诈特征很有帮助，让规则抓取更准确。

563

风险控制篇



**3.7 代理和模拟器识别**

欺诈分子的技术也在不断的演进，作案的隐匿性更强，代理服务器和模拟器是非常好的隐匿手段，在交易刷单、信用卡欺诈等很多场景都会见到。

我们研究了 TCP Signature、Time Gap、用户行为、针对各类模拟器的实验数据等，有了一套自己的方法论和识别方案。

**四、人工规则 vs 模型**

模型对于规则的补充意义是非常显著的，可以弥补人工规则的盲区，模型可以很好的覆盖历史欺诈特征、可以大大减少规则数量。

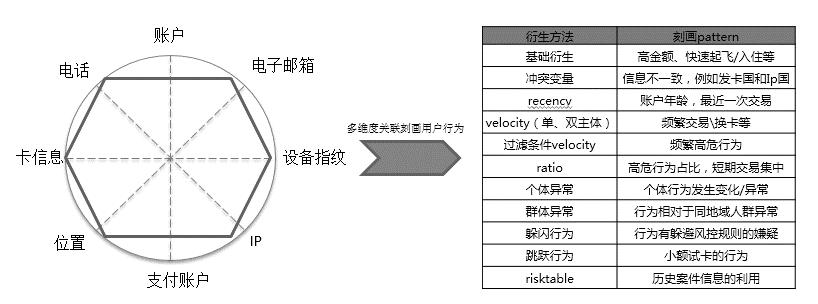
不管是规则还是模型，都需要建立在对业务上下文充分理解的基础之上。脱离业务上下文、仅针对数据本身的分析而提取出的特征往往是有偏颇的、不全面的，实际上线效果必然也不会很理想。

简单介绍我们的特征变量提取方法：

变量衍生方法：

564

风险控制篇



**五、结束语**

“Make the Travel More Freely and Securely”，是携程风控的内部文化和使命。随着携程全球化步伐的不断推进，交易量日益增长的情况下，国内外的黑产技术也日趋成熟，欺诈形势越来越严峻。

携程是 OTA 行业的领导者，携程反欺诈技术团队也将引领反欺诈领域的技术进步，提前研究并掌握大数据和人工智能等先进工具的应用，以应对未来更大的挑战，给用户提供更好的服务。

565