美团点评

美团点评
2018 技术年货

CODE A BETTER LIFE

序

春节已近，年味渐浓。

又到了我们献上技术年货的时候。

不久前，我们已经给大家分享了技术沙龙大套餐，汇集了过去一年我们线上线下技术沙龙 99位讲师，85个演讲，70+小时 分享。

今天出场的，同样重磅——技术博客全年大合集。

2018年，是美团技术团队官方博客第5个年头， 博客网站 全年独立访问用户累计超过300万，微信公众号（meituantech）的关注数也超过了15万。

由衷地感谢大家一直以来对我们的鼓励和陪伴！

在2019年春节到来之际，我们再次精选了114篇技术干货，制作成一本厚达1200多页的电子书呈送给大家。

这本电子书主要包括前端、后台、系统、算法、测试、运维、工程师成长等7个板块。疑义相与析，大家在阅读中如果发现Bug、问题，欢迎扫描文末二维码，通过微信公众号与我们交流。

也欢迎大家转给有相同兴趣的同事、朋友，一起切磋，共同成长。

最后祝大家，新春快乐，阖家幸福。

目录 – 运维篇

互联网企业数据安全体系建设

作者: 赵彦

一、背景

Facebook数据泄露事件一度成为互联网行业的焦点，几百亿美元市值瞬间蒸发，这个代价足以在地球上养活一支绝对庞大的安全团队，甚至可以直接收购几家规模比较大的安全公司了。

虽然媒体上发表了很多谴责的言论，但实事求是地讲，Facebook面临是一个业界难题，任何一家千亿美元的互联网公司面对这种问题，可能都没有太大的抵抗力，仅仅是因为全球区域的法律和国情不同，暂时不被顶上舆论的浪尖罢了。但是全球的趋势是越来越重视隐私，在安全领域中，数据安全这个子领域也重新被提到了一个新的高度，所以笔者就借机来说一下数据安全建设。（按照惯例，本文涉及敏感信息的部分会进行省略处理或者一笔带过。）

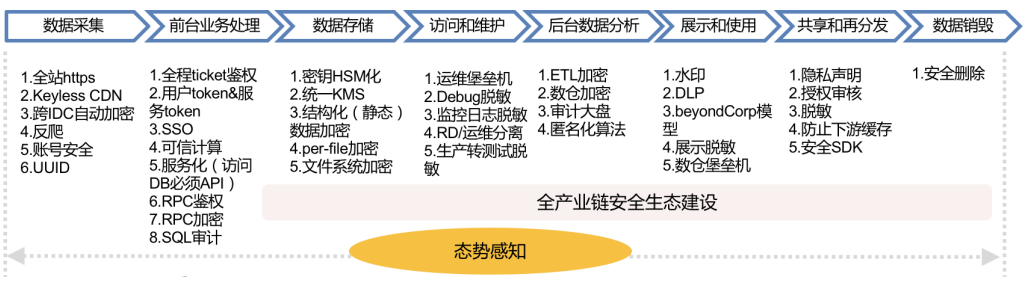
二、概念

这里特别强调一下，“隐私保护”和“数据安全”是两个完全不同的概念，隐私保护对于安全专业人员来说是一个更加偏向合规的事情，主要是指数据过度收集和数据滥用方面对法律法规的遵从性，对很多把自身的盈利模式建立在数据之上的互联网公司而言，这个问题特别有挑战。有些公司甚至把自己定义为数据公司，如果不用数据来做点什么，要么用户体验大打折扣，要么商业价值减半。GDPR即将实施，有些公司或将离场欧洲，就足见这件事的难度不容小觑。当然市场上也有一些特别推崇隐私保护的公司，他们很大程度上并不能真正代表用户意愿，而只是因为自家没有数据或缺少数据，随口说说而已。

数据安全是实现隐私保护的最重要手段之一。对安全有一定了解的读者可能也会察觉到，数据安全并不是一个独立的要素，而是需要连同网络安全、系统安全、业务安全等多种因素，只有全部都做好了，才能最终达到数据安全的效果。所以本文尽可能的以数据安全为核心，但没有把跟数据安全弱相关的传统安全体系防护全部列出来，对于数据安全这个命题而言尽可能的系统化，又避免啰嗦。另外笔者也打算在夏季和秋季把其他子领域的话题单独成文，譬如海量IDC下的入侵防御体系等，敬请期待。

三、全生命周期建设

尽管业内也有同学表示数据是没有边界的，如果按照泄露途径去做可能起不到“根治”的效果，但事实上以目前的技术是做不到无边界数据安全的。下图汇总了一个全生命周期内的数据安全措施：



四、数据采集

数据泄露有一部分原因是用户会话流量被复制，尽管有点技术门槛，但也是发生频率比较高的安全事件之一，只是是很多企业没有感知到而已。下面从几个维度来说明数据采集阶段的数据保护。

流量保护

全站HTTPS是目前互联网的主流趋势，它解决的是用户到服务器之间链路被嗅探、流量镜像、数据被第三方掠走的问题。这些问题其实是比较严重的，比如电信运营商内部偶有舞弊现象，各种导流劫持插广告（当然也可以存数据，插木马），甚至连AWS也被劫持DNS请求，对于掌握链路资源的人来说无异于可以发动一次“核战争”。即使目标对象IDC入侵防御做的好，攻击者也可以不通过正面渗透，而是直接复制流量，甚至定向APT，最终只是看操纵流量后达到目的的收益是否具有性价比。

HTTPS是一个表面现象，它暗示着任何互联网上未加密的流量都是没有隐私和数据安全的，同时，也不是说有了HTTPS就一定安全。HTTPS本身也有各种安全问题，比如使用不安全的协议TLS1.0、SSL3，采用已经过时的弱加密算法套件，实现框架安全漏洞如心脏滴血，还有很多的数字证书本身导致的安全问题。

全站HTTPS会带来的附带问题是CDN和高防IP。历史上有家很大的互联网公司被NSA嗅探获取了用户数据，原因是CDN回源时没有使用加密，即用户浏览器到CDN是加密的，但CDN到IDC源站是明文的。如果CDN到源站加密就需要把网站的证书私钥给到CDN厂商，这对于没有完全自建CDN的公司而言也是一个很大的安全隐患，所以后来衍生出了Keyless CDN技术，无需给出自己的证书就可以实现CDN回源加密。

广域网流量未加密的问题也要避免出现在“自家后院”——IDC间的流量复制和备份同步，对应的解决方案是跨IDC流量自动加密、TLS隧道化。

业务安全属性

在用户到服务器之间还涉及两个业务安全方向的问题。第一个问题是账号安全，只要账号泄露（撞库&爆破）到达一定数量级，把这些账号的数据汇总一下，就必定可以产生批量数据泄露的效果。

第二个问题是反爬，爬虫的问题存在于一切可通过页面、接口获取数据的场合，大概1小时爬个几百万条数据是一点问题都没有的，对于没有彻底脱敏的数据，爬虫的效果有时候等价于“黑掉”服务器。账号主动地或被动地泄露+爬虫技术，培育了不少黑产和数据获取的灰色地带。

UUID

UUID最大的作用是建立中间映射层，屏蔽与真实用户信息的关系链。譬如在开放平台第三方应用数据按需自主授权只能读取UUID，但不能直接获取个人的微信号。更潜在的意义是屏蔽个体识别数据，因为实名制，手机号越来越能代表个人标识，且一般绑定了各种账号，更改成本很高，找到手机号就能对上这个人，因此理论上但凡带有个体识别数据的信息都需要“转接桥梁”、匿名化和脱敏。譬如当商家ID能唯一标识一个品牌和店名的时候，这个原本用于程序检索的数据结构也一下子变成了个体识别数据，也都需要纳入保护范畴。

五、前台业务处理

鉴权模型

在很多企业的应用架构中，只有在业务逻辑最开始处理的部分设置登录态校验，后面的事务处理不再会出现用户鉴权，进而引发了一系列的越权漏洞。事实上越权漏洞并不是这种模型的全部危害，还包括各种K/V、RDS（关系型数据库）、消息队列等等，RPC没有鉴权导致可任意读取的安全问题。

在数据层只知道请求来自一个数据访问层中间件，来自一个RPC调用，但完全不知道来自哪个用户，还是哪个诸如客服系统或其他上游应用，无法判断究竟对当前的数据（对象）是否拥有完整的访问权限。绝大多数互联网公司都用开源软件或修改后的开源软件，这类开源软件的特点是基本不带安全特性，或者只具备很弱的安全特性，以至于完全不适用于海量IDC规模下的4A模型（认证、授权、管理、审计）。外面防御做的很好，而在内网可以随意读写，这可能是互联网行业的普遍现状了。主要矛盾还是鉴权颗粒度和弹性计算的问题，关于这个问题的解决方案可以参考笔者的另外一篇文章《初探下一代网络隔离与访问控制》，其中提到Google的方法是内网RPC鉴权，由于Google的内网只有RPC一种协议，所以就规避了上述大多数安全问题。

对于业务流的鉴权模型，本质上是需要做到Data和App分离，建立Data默认不信任App的模型，而应用中的全程Ticket和逐级鉴权是这种思想下的具体实现方法。

服务化

服务化并不能认为是一个安全机制，但安全却是服务化的受益者。我们再来温习一下当年Bezos在Amazon推行服务化的一纸号令：

1）所有团队今后将通过服务接口公开他们的数据和功能。 2）团队必须通过这些接口相互通信。 3）不允许使用其他形式的进程间通信：不允许直接链接，不允许直接读取其他团队的数据存储，不支持共享内存模式，无后门。唯一允许的通信是通过网络上的服务接口调用。 4）他们使用什么技术并不重要。HTTP，Corba，Pubsub，自定义协议 – 无关紧要。贝索斯不在乎。 5）所有服务接口无一例外都必须从头开始设计为可外部化。也就是说，团队必须规划和设计能够将接口展示给外部开发人员。没有例外。6）任何不这样做的人都会被解雇。

服务化的结果在安全上的意义是必须通过接口访问数据，屏蔽了各种直接访问数据的途径，有了API控制和审计就会方便很多。

内网加密

一些业界Top的公司甚至在IDC内网里也做到了加密，也就是在后台的组件之间的数据传输都是加密的，譬如Goolge的RPC加密和Amazon的TLS。由于IDC内网的流量比公网大得多，所以这里是比较考验工程能力的地方。对于大多数主营业务迭代仍然感觉有压力的公司而言，这个需求可能有点苛刻了，所以笔者认为用这些指标来衡量一家公司的安全能力属于哪一个档位是合理的。私有协议算不算？如果私有协议里不含有标准TLS（SHA256）以上强度的加密，或者只是信息不对称的哈希，笔者认为都不算。

数据库审计

数据库审计/数据库防火墙是一个入侵检测/防御组件，是一个强对抗领域的产品，但是在数据安全方面它的意义也是明显的：防止SQL注入批量拉取数据，检测API鉴权类漏洞和爬虫的成功访问。

除此之外，对数据库的审计还有一层含义，是指内部人员对数据库的操作，要避免某个RD或DBA为了泄愤，把数据库拖走或者删除这种危险动作。通常大型互联网公司都会有数据库访问层组件，通过这个组件，可以审计、控制危险操作。

六、数据存储

数据存储之于数据安全最大的部分是数据加密。Amazon CTO Werner Vogels曾经总结：“AWS所有的新服务，在原型设计阶段就会考虑到对数据加密的支持。”国外的互联网公司中普遍比较重视数据加密。

HSM/KMS

业界的普遍问题是不加密，或者加密了但没有使用正确的方法：使用自定义UDF，算法选用不正确或加密强度不合适，或随机数问题，或者密钥没有Rotation机制，密钥没有存储在KMS中。数据加密的正确方法本身就是可信计算的思路，信任根存储在HSM中，加密采用分层密钥结构，以方便动态转换和过期失效。当Intel CPU普遍开始支持SGX安全特性时，密钥、指纹、凭证等数据的处理也将以更加平民化的方式使用类似Trustzone的芯片级隔离技术。

结构化数据

这里主要是指结构化数据静态加密，以对称加密算法对诸如手机、身份证、银行卡等需要保密的字段加密持久化，另外除了数据库外，数仓里的加密也是类似的。比如，在 Amazon Redshift 服务中，每一个数据块都通过一个随机的密钥进行加密，而这些随机密钥则由一个主密钥进行加密存储。用户可以自定义这个主密钥，这样也就保证了只有用户本人才能访问这些机密数据或敏感信息。鉴于这部分属于比较常用的技术，不再展开。

文件加密

对单个文件独立加密，一般情况下采用分块加密，典型的场景譬如在《互联网企业安全高级指南》一书中提到的iCloud将手机备份分块加密后存储于AWS的S3，每一个文件切块用随机密钥加密后放在文件的meta data中，meta data再用file key包裹，file key再用特定类型的data key（涉及数据类型和访问权限）加密，然后data key被master key包裹。

文件系统加密

文件系统加密由于对应用来说是透明的，所以只要应用具备访问权限，那么文件系统加密对用户来说也是“无感知”的。它解决的主要是冷数据持久化后存储介质可访问的问题，即使去机房拔一块硬盘，或者从一块报废的硬盘上尝试恢复数据，都是没有用的。但是对于API鉴权漏洞或者SQL注入而言，显然文件系统的加密是透明的，只要App有权限，漏洞利用也有权限。

七、访问和运维

在这个环节，主要阐述防止内部人员越权的一些措施。

角色分离

研发和运维要分离，密钥持有者和数据运维者要分离，运维角色和审计角色要分离。特权账号须回收，满足最小权限，多权分立的审计原则。

运维审计

堡垒机（跳板机）是一种针对人肉运维的常规审计手段，随着大型IDC中运维自动化的加深，运维操作都被API化，所以针对这些API的调用也需要被列入审计范畴，数量级比较大的情况下需要使用数据挖掘的方法。

工具链脱敏

典型的工具脱敏包括监控系统和Debug工具/日志。在监控系统类目中，通常由于运维和安全的监控系统包含了全站用户流量，对用户Token和敏感数据需要脱敏，同时这些系统也可能通过简单的计算得出一些运营数据，譬如模糊的交易数目，这些都是需要脱敏的地方。在Debug方面也出过Debug Log带有CVV码等比较严重的安全事件，因此都是需要注意的数据泄漏点。

生产转测试

生产环境和测试环境必须有严格定义和分离，如特殊情况生产数据需要转测试，必须经过脱敏、匿名化。

八、后台数据处理

数仓安全

目前大数据处理基本是每个互联网公司的必需品，通常承载了公司所有的用户数据，甚至有的公司用于数据处理的算力超过用于前台事务处理的算力。以Hadoop为代表的开源平台本身不太具备很强的安全能力，因此在成为公有云服务前需要做很多改造。在公司比较小的时候可以选择内部信任模式，不去过于纠结开源平台本身的安全，但在公司规模比较大，数据RD和BI分析师成千上万的时候，内部信任模式就需要被抛弃了，这时候需要的是一站式的授权&审计平台，需要看到数据的血缘继承关系，需要高敏数据仍然被加密。在这种规模下，工具链的成熟度会决定数据本地化的需求，工具链越成熟数据就越不需要落到开发者本地，这样就能大幅提升安全能力。同时鼓励一切计算机器化&程序化&自动化，尽可能避免人工操作。

对于数据的分类标识、分布和加工，以及访问状况需要有一个全局的大盘视图，结合数据使用者的行为建立“态势感知”的能力。

因为数仓是最大的数据集散地，因此每家公司对于数据归属的价值观也会影响数据安全方案的落地形态：放逐+检测型 or 隔离+管控型。

匿名化算法

匿名化算法更大的意义其实在于隐私保护而不在于数据安全（关于隐私保护部分笔者打算另外单独写一篇），如果说对数据安全有意义，匿名化可能在于减少数据被滥用的可能性，以及减弱数据泄漏后的影响面。

九、展示和使用

这个环节泛指大量的应用系统后台、运营报表以及所有可以展示和看到数据的地方，都可能是数据泄露的重灾区。

展示脱敏

对页面上需要展示的敏感信息进行脱敏。一种是完全脱敏，部分字段打码后不再展示完整的信息和字段，另一种是不完全脱敏，默认展示脱敏后的信息，但仍然保留查看明细的按钮（API），这样所有的查看明细都会有一条Log，对应审计需求。具体用哪种脱敏需要考虑工作场景和效率综合评估。

水印

水印主要用在截图的场景，分为明水印和暗水印，明水印是肉眼可见的，暗水印是肉眼不可见暗藏在图片里的识别信息。水印的形式也有很多种，有抵抗截屏的，也有抵抗拍照的。这里面也涉及很多对抗元素不一一展开。

安全边界

这里的边界其实是办公网和生产网组成的公司数据边界，由于办公移动化程度的加深，这种边界被进一步模糊化，所以这种边界实际上是逻辑的，而非物理上的，它等价于公司办公网络，生产网络和支持MDM的认证移动设备。对这个边界内的数据，使用DLP来做检测，DLP这个名词很早就有，但实际上它的产品形态和技术已经发生了变化，用于应对大规模环境下重检测，轻阻断的数据保护模式。

除了DLP之外，整个办公网络会采用BeyondCorp的“零信任”架构，对整个的OA类应用实现动态访问控制，全面去除匿名化访问，全部HTTPS，根据角色最小权限化，也就是每个账号即使泄露能访问到的也有限。同时提高账号泄露的成本（多因素认证）和检测手段，一旦检测到泄露提供远程擦除的能力。

堡垒机

堡垒机作为一种备选的方式主要用来解决局部场景下避免操作和开发人员将敏感数据下载到本地的方法，这种方法跟VDI类似，比较厚重，使用门槛不高，不适合大面积普遍推广。

十、共享和再分发

对于业务盘子比较大的公司而言，其数据都不会是只在自己的系统内流转，通常都有开放平台，有贯穿整个产业链的上下游数据应用。Facebook事件曝光其实就属于这类问题，不开放是不可能的，因为这影响了公司的内核—–赖以生存的商业价值。

所以这个问题的解决方案等价于：1）内核有限妥协（为保障用户隐私牺牲一部分商业利益）；2）一站式数据安全服务。

防止下游数据沉淀

首先，所有被第三方调用的数据，如非必要一律脱敏和加密。如果部分场景有必要查询明细数据，设置单独的API，并对账号行为及API查询做风控。

其次如果自身有云基础设施，公有云平台，可以推动第三方上云，从而进行（1）安全赋能，避免一些因自身能力不足引起的安全问题；（2）数据集中化，在云上集中之后利于实施一站式整体安全解决方案（数据加密，风控，反爬和数据泄露检测类服务），大幅度降低外部风险并在一定程度上降低作恶和监守自盗的问题。

反爬

反爬在这里主要是针对公开页面，或通过接口爬取的信息，因为脱敏这件事不可能在所有的环节做的很彻底，所以即便通过大量的“公开”信息也可以进行汇聚和数据挖掘，最终形成一些诸如用户关系链，经营数据或辅助决策类数据，造成过度信息披露的影响。

授权审核

设置专门的团队对开放平台的第三方进行机器审核及人工审核，禁止“无照经营”和虚假三方，提高恶意第三方接入的门槛，同时给开发者/合作方公司信誉评级提供基础。

法律条款

所有的第三方接入必须有严格的用户协议，明确数据使用权利，数据披露限制和隐私保护的要求。像GDPR一样，明确数据处理者角色和惩罚条约。

十一、数据销毁

数据销毁主要是指安全删除，这里特别强调是，往往数据的主实例容易在视野范围内，而把备份类的数据忽略掉。 如果希望做到快速的安全删除，最好使用加密数据的方法，因为完整覆写不太可能在短时间内完成，但是加密数据的安全删除只要删除密钥即可。

十二、数据的边界

数据治理常常涉及到“边界”问题，不管你承不承认，边界其实总是存在的，只不过表达方式不一样，如果真的没有边界，也就不存在数据安全一说。

企业内部

在不超越网络安全法和隐私保护规定的情况下，法律上企业对内部的数据都拥有绝对控制权，这使得企业内部的数据安全建设实际上最后会转化为一项运营类的工作，挑战难度也无非是各个业务方推动落地的成本。但对规模比较大的公司而言，光企业内部自治可能是不够的，所以数据安全会衍生出产业链上闭环的需求。

生态建设

为了能让数据安全建设在企业内部价值链之外的部分更加平坦化，大型企业可能需要通过投资收购等手段获得上下游企业的数据控制权及标准制定权，从而在大生态里将自己的数据安全标准推行到底。如果不能掌控数据，数据安全也无从谈起。在话语权不足的情况下，现实选择是提供更多的工具给合作方，也是一种数据控制能力的延伸。

十三、ROI和建设次第

对于很多规模不大的公司而言，上述数据安全建设手段可能真的有点多，对于小一点公司即便什么事不干可能也消化不了那么多需求，因为开源软件和大多数的开发框架都不具备这些能力，需要DIY的成分很高，所以我们梳理一下前置条件，优先级和ROI，让数据安全这件事对任何人都是可以接受的，当然这种情况其实也对应了一些创业空间。

基础

账号、权限、日志、脱敏和加密这些都是数据安全的基础。同时还有一些不完全是基础，但能体现为优势的部分：基础架构统一，应用架构统一，如果这两者高度统一，数据安全建设能事半功倍。

日志收集

日志是做数据风控的基础，但这里面也有两个比较重要的因素：

1. 办公网络是否BeyondCorp化，这给数据风控提供了极大地便利。
2. 服务化，所有的数据调用皆以API的形式，给日志记录提供了统一的形式。

数据风控

在数据安全中，“放之四海皆准”的工作就是数据风控，适用于各类企业，结合设备信息、账号行为、查询/爬(读)取行为做风控模型。对于面向2C用户类，2B第三方合作类，OA员工账号类都是适用的。具体的策略思想笔者打算在后续文章《入侵防御体系建设》中详细描述。

作者简介

赵彦，现任美团集团安全部高级总监，负责集团旗下全线业务的信息安全与隐私保护。加盟美团前，曾任华为云安全首席架构师，奇虎360企业安全技术总监、久游网安全总监、绿盟科技安全专家等职务。白帽子时代是Ph4nt0m Security Team的核心成员，互联网安全领域第一代资深从业者。

关于美团安全

美团安全部的大多数核心人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级IDC规模攻防对抗的经验。安全部也不乏CVE“挖掘圣手”，有受邀在Black Hat等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。

目前，美团安全部涉及的技术包括渗透测试、Web防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级IDC规模、数十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，

包括网络层、虚拟化/容器层、Server 软件层（内核态/用户态）、语言虚拟机层（JVM/JS V8）、Web 应用层、数据访问层等，并能够基于“大数据+机器学习”技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领域不断探索的机会。

招聘信息

美团安全部正在招募Web&二进制攻防、后台&系统开发、机器学习&算法等各路小伙伴。如果你想加入我们，欢迎简历请发至邮箱 zhaoyan17@meituan.com

具体职位信息可参考这里： https://mp.weixin.qq.com/s/ynEq5LqQ2uBcEaHCu7Tsiw

美团安全应急响应中心MTSRC主页： security.meituan.com

SQL解析在美团的应用

作者: 广友

数据库作为核心的基础组件，是需要重点保护的对象。任何一个线上的不慎操作，都有可能给数据库带来严重的故障，从而给业务造成巨大的损失。为了避免这种损失，一般会在管理上下功夫。比如为研发人员制定数据库开发规范；新上线的SQL，需要DBA进行审核；维护操作需要经过领导审批等等。而且如果希望能够有效地管理这些措施，需要有效的数据库培训，还需要DBA细心的进行SQL审核。很多中小型创业公司，可以通过设定规范、进行培训、完善审核流程来管理数据库。

随着美团的业务不断发展和壮大，上述措施的实施成本越来越高。如何更多的依赖技术手段，来提高效率，越来越受到重视。业界已有不少基于MySQL源码开发的SQL审核、优化建议等工具，极大的减轻了DBA的SQL审核负担。那么我们能否继续扩展MySQL的源码，来辅助DBA和研发人员来进一步提高效率呢？比如，更全面的SQL优化功能；多维度的慢查询分析；辅助故障分析等。要实现上述功能，其中最核心的技术之一就是SQL解析。

现状与场景

SQL解析是一项复杂的技术，一般都是由数据库厂商来掌握，当然也有公司专门提供 SQL解析的API 。由于这几年MySQL数据库中间件的兴起，需要支持读写分离、分库分表等功能，就必须从SQL中抽出表名、库名以及相关字段的值。因此像Java语言编写的Druid，C语言编写的MaxScale，Go语言编写的Kingshard等，都会对SQL进行部分解析。而真正把SQL解析技术用于数据库维护的产品较少，主要有如下几个:

- 美团开源的SQLAdvisor。它基于MySQL原生态词法解析，结合分析SQL中的where条件、聚合条件、多表Join关系给出索引优化建议。
- 去哪儿开源的Inception。侧重于根据内置的规则，对SQL进行审核。
- 阿里的Cloud DBA。根据官方文档介绍，其也是提供SQL优化建议和改写。

上述产品都有非常合适的应用场景，在业界也被广泛使用。但是SQL解析的应用场景远远没有被充分发掘，比如：

- 基于表粒度的慢查询报表。比如，一个Schema中包含了属于不同业务线的数据表，那么从业务线的角度来说，其希望提供表粒度的慢查询报表。
- 生成SQL特征。将SQL语句中的值替换成问号，方便SQL归类。虽然可以使用正则表达式实现相同的功能，但是其Bug较多，可以参考pt-query-digest。比如pt-query-digest中，会把遇到的数字都替换成“?”，导致无法区别不同数字后缀的表。
- 高危操作确认与规避。比如，DBA不小心Drop数据表，而此类操作，目前还无有效的工具进行回滚，尤其是大表，其后果将是灾难性的。
- SQL合法性判断。为了安全、审计、控制等方面的原因，美团不会让研发人员直接操作数据库，而是提供RDS服务。尤其是对于数据变更，需要研发人员的上级主管进行业务上的审批。如果研发人员，写了一条语法错误的SQL，而RDS无法判断该SQL是否合法，就会造成不必要的沟通成本。

因此为了让所有有需要的业务都能方便的使用SQL解析功能，我们认为应该具有如下特性。

- 直接暴露SQL解析接口，使用尽量简单。比如，输入SQL，则输出表名、特征和优化建议。
- 接口的使用不依赖于特定的语言，否则维护和使用的代价太高。比如，以HTTP等方式提供服务。

千里之行，始于足下。下面我先介绍下SQL的解析原理。

原理

SQL解析与优化是属于编译器范畴，和C等其他语言的解析没有本质的区别。其中分为，词法分析、语法和语义分析、优化、执行代码生成。对应到MySQL的部分，如下图

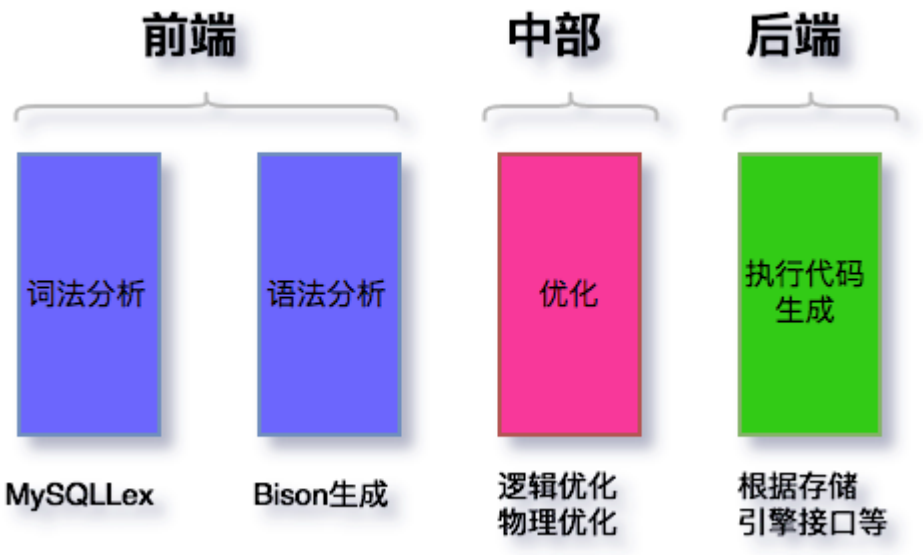


图1 SQL解析原理

词法分析

SQL解析由词法分析和语法/语义分析两个部分组成。词法分析主要是把输入转化成一个个Token。其中Token中包含Keyword（也称symbol）和非Keyword。例如，SQL语句 select username from userinfo，在分析之后，会得到4个Token，其中有2个Keyword，分别为select和from：

关键字	非关键字	关键字	非关键字
select	username	from	userinfo

通常情况下，词法分析可以使用 Flex 来生成，但是MySQL并未使用该工具，而是手写了词法分析部分（据说是为了效率和灵活性，参考 此文 ）。具体代码在sql/lex.h和sql/sql_lex.cc文件中。

MySQL中的Keyword定义在sql/lex.h中，如下为部分Keyword：

```
{ "&&",              SYM(AND_AND_SYM)},
{ "<",               SYM(LT)},
{ "<=",              SYM(LE)},
{ "<>",              SYM(NE)},
{ "!=",              SYM(NE)},
{ "=",               SYM(EQ)},
{ ">",               SYM(GT_SYM)},
{ ">=",              SYM(GE)},
```

```
{ "<<",              SYM(SHIFT_LEFT)},
{ ">>",              SYM(SHIFT_RIGHT)},
{ "<=>",             SYM(EQUAL_SYM)},
{ "ACCESSIBLE",      SYM(ACCESSIBLE_SYM)},
{ "ACTION",          SYM(ACTION)},
{ "ADD",             SYM(ADD)},
{ "AFTER",           SYM(AFTER_SYM)},
{ "AGAINST",         SYM(AGAINST)},
{ "AGGREGATE",       SYM(AGGREGATE_SYM)},
{ "ALL",             SYM(ALL)},
```

词法分析的核心代码在sql/sql_lex.c文件中的，MySQLLex→lex_one_Token，有兴趣的同学可以下载源码研究。

语法分析

语法分析就是生成语法树的过程。这是整个解析过程中最精华，最复杂的部分，不过这部分MySQL使用了Bison来完成。即使如此，如何设计合适的数据结构以及相关算法，去存储和遍历所有的信息，也是值得在这里研究的。

语法分析树

SQL语句：

```
select username, ismale from userinfo where age > 20 and level > 5 and 1 = 1
```

会生成如下语法树。

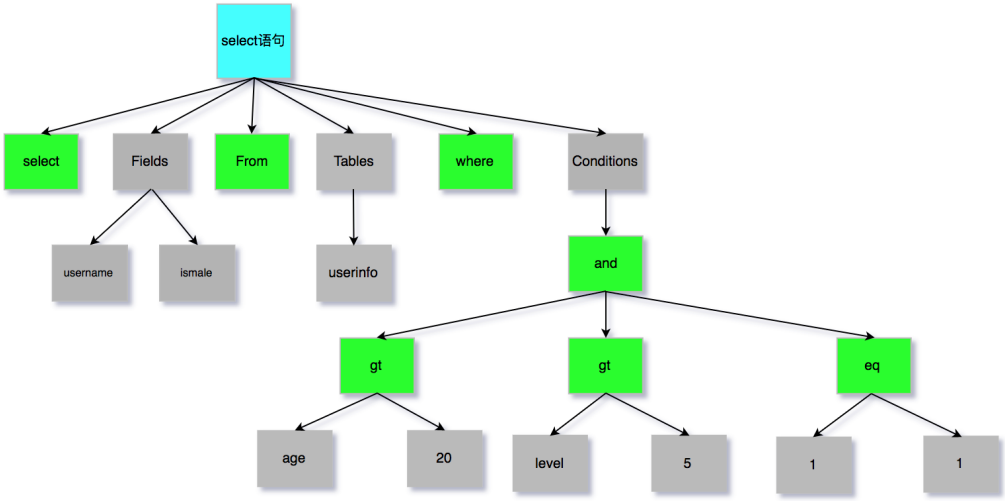


图2 语法树

对于未接触过编译器实现的同学，肯定会好奇如何才能生成这样的语法树。其背后的原理都是编译器的范畴，可以参考维基百科的一篇 文章，以及该链接中的参考书籍。本人也是在学习MySQL源码过程中，阅读了部分内容。由于编译器涉及的内容过多，本人精力和时间有限，不做过多探究。从工程的角度来说，学会如何使用Bison去构建语法树，来解决实际问题，对我们的工作也许有更大帮助。下面我就以Bison为基础，探讨该过程。

MySQL语法分析树生成过程

全部的源码在sql/sql_yacc.yy中，在MySQL5.6中有17K行左右代码。这里列出涉及到SQL：

```
select username, ismale from userinfo where age > 20 and level > 5 and 1 = 1
```

解析过程的部分代码摘录出来。其实有了Bison之后，SQL解析的难度也没有想象的那么大。特别是这里给出了解析的脉络之后。

```
select /*select语句入口*/:

          select_init

          {

            LEX *lex= Lex;

            lex->sql_command= SQLCOM_SELECT;

          }

        ;

select_init:
          SELECT_SYM /*select 关键字*/ select_init2

        | '(' select_paren ')' union_opt

        ;

select_init2:
          select_part2
          {
            LEX *lex= Lex;
            SELECT_LEX * sel= lex->current_select;
            if (lex->current_select->set_braces(0))
            {
              my_parse_error(ER(ER_SYNTAX_ERROR));
              MYSQL_YYABORT;
            }
            if (sel->linkage == UNION_TYPE &&
                sel->master_unit()->first_select()->braces)
            {
              my_parse_error(ER(ER_SYNTAX_ERROR));
              MYSQL_YYABORT;
            }
          }
          union_clause
        ;
select_part2:
          {
            LEX *lex= Lex;
            SELECT_LEX *sel= lex->current_select;
            if (sel->linkage != UNION_TYPE)
              mysql_init_select(lex);
            lex->current_select->parsing_place= SELECT_LIST;
          }

          select_options select_item_list /*解析列名*/
          {
            Select->parsing_place= NO_MATTER;
          }
          select_into select_lock_type
        ;

select_into:
          opt_order_clause opt_limit_clause {}
        | into
        | select_from /*from 字句*/
        | into select_from
        | select_from into
        ;
select_from:
          FROM join_table_list /*解析表名*/ where_clause /*where字句*/ group_clause having_clause
          opt_order_clause opt_limit_clause procedure_analyse_clause
          {
            Select->context.table_list=
              Select->context.first_name_resolution_table=
                Select->table_list.first;
          }
        | FROM DUAL_SYM where_clause opt_limit_clause
          /* oracle compatibility: oracle always requires FROM clause,
             and DUAL is system table without fields.
             Is "SELECT 1 FROM DUAL" any better than "SELECT 1" ?
```

```
          Hmmm :) */
        ;

where_clause:
          /* empty */  { Select->where= 0; }
        | WHERE
          {
            Select->parsing_place= IN_WHERE;
          }
          expr /*各种表达式*/
          {
            SELECT_LEX *select= Select;
            select->where= $3;
            select->parsing_place= NO_MATTER;
            if ($3)
              $3->top_level_item();
          }
        ;

/* all possible expressions */
expr:
           | expr and expr %prec AND_SYM
          {
            /* See comments in rule expr: expr or expr */
            Item_cond_and *item1;
            Item_cond_and *item3;
            if (is_cond_and($1))
            {
              item1= (Item_cond_and*) $1;
              if (is_cond_and($3))
              {
                item3= (Item_cond_and*) $3;
                /*
                  (X1 AND X2) AND (Y1 AND Y2) ==> AND (X1, X2, Y1, Y2)
                */
                item3->add_at_head(item1->argument_list());
                $$ = $3;
              }
              else
              {
                /*
                  (X1 AND X2) AND Y ==> AND (X1, X2, Y)
                */
                item1->add($3);
                $$ = $1;
              }
            }
            else if (is_cond_and($3))
            {
              item3= (Item_cond_and*) $3;
              /*
                X AND (Y1 AND Y2) ==> AND (X, Y1, Y2)
              */
              item3->add_at_head($1);
              $$ = $3;
            }
            else
            {
              /* X AND Y */
              $$ = new (YYTHD->mem_root) Item_cond_and($1, $3);
              if ($$ == NULL)
                MYSQL_YYABORT;
            }
          }
```

在大家浏览上述代码的过程，会发现Bison中嵌入了C++的代码。通过C++代码，把解析到的信息存储到相关对象中。例如表信息会存储到TABLE_LIST中，order_list存储order by子句里的信息，where字句存储在Item中。有了这些信息，再辅助以相应的算法就可以对SQL进行更进一步的处理了。

核心数据结构及其关系

在SQL解析中，最核心的结构是SELECT_LEX，其定义在sql/sql_lex.h中。下面仅列出与上述例子相关的部分。

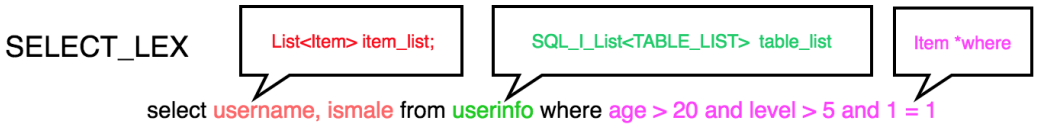


图3 SQL解析树结构

上面图示中，列名username、ismale存储在item_list中，表名存储在table_list中，条件存储在where中。其中以where条件中的Item层次结构最深，表达也较为复杂，如下图所示。

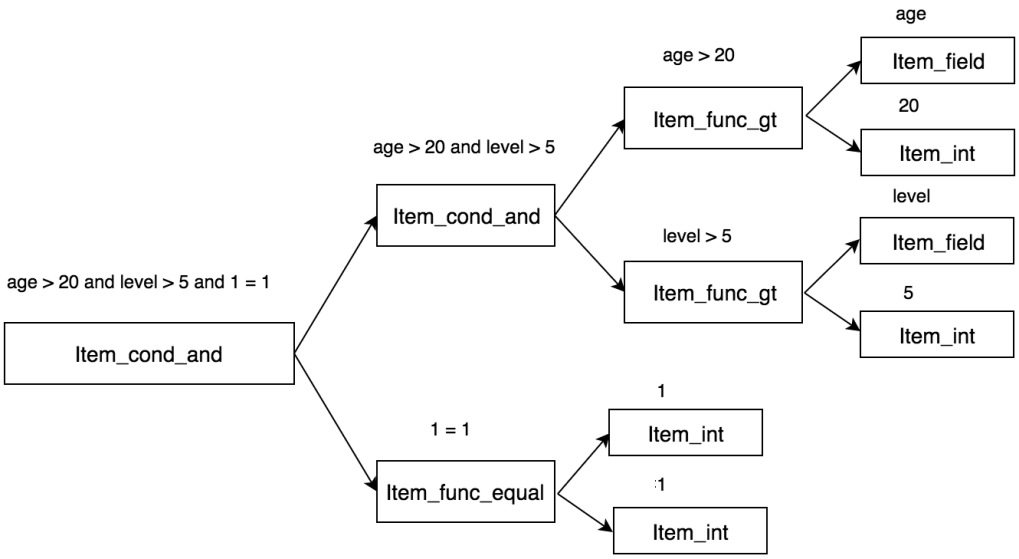


图4 where条件

SQL解析的应用

为了更深入的了解SQL解析器，这里给出2个应用SQL解析的例子。

无用条件去除

无用条件去除属于优化器的逻辑优化范畴，可以仅仅根据SQL本身以及表结构即可完成，其优化的情况也是较多的，代码在sql/sql_optimizer.cc文件中的remove_eq_conds函数。为了避免过于繁琐的描述，以及大段代码的粘贴，这里通过图来分析以下四种情况。

- a）1=1 and (m > 3 and n > 4)
- b）1=2 and (m > 3 and n > 4)
- c）1=1 or (m > 3 and n > 4)
- d）1=2 or (m > 3 and n > 4)

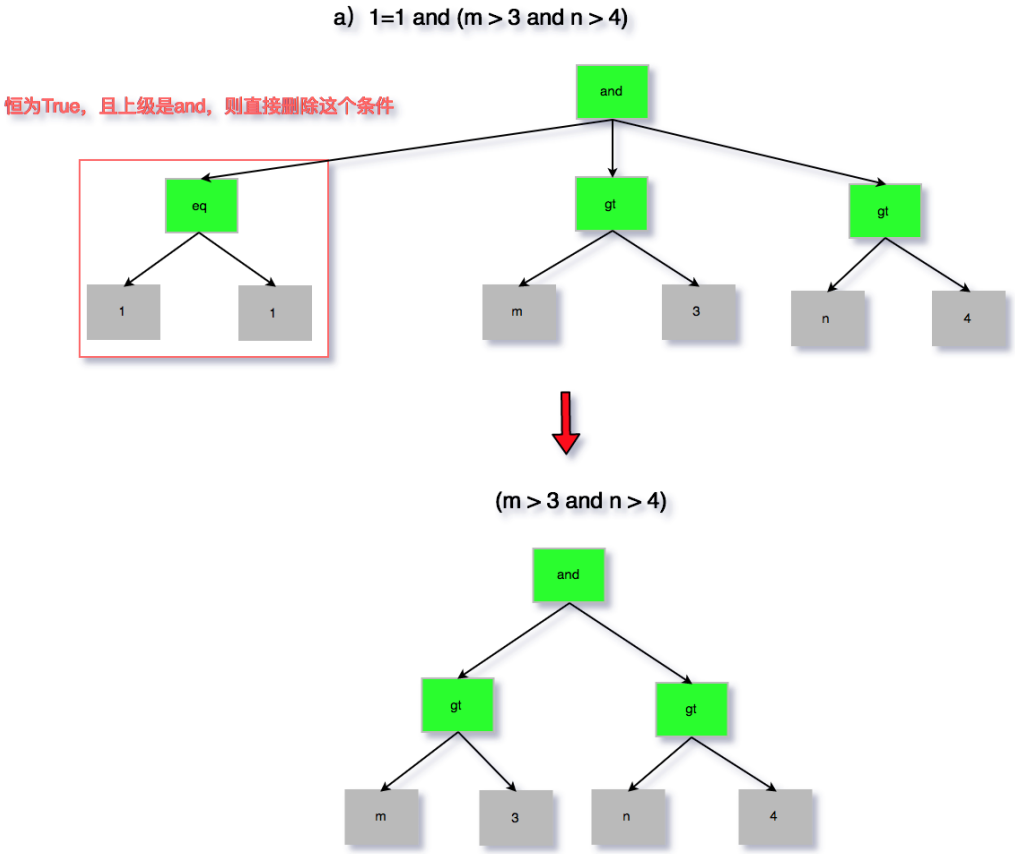


图5 无用条件去除a

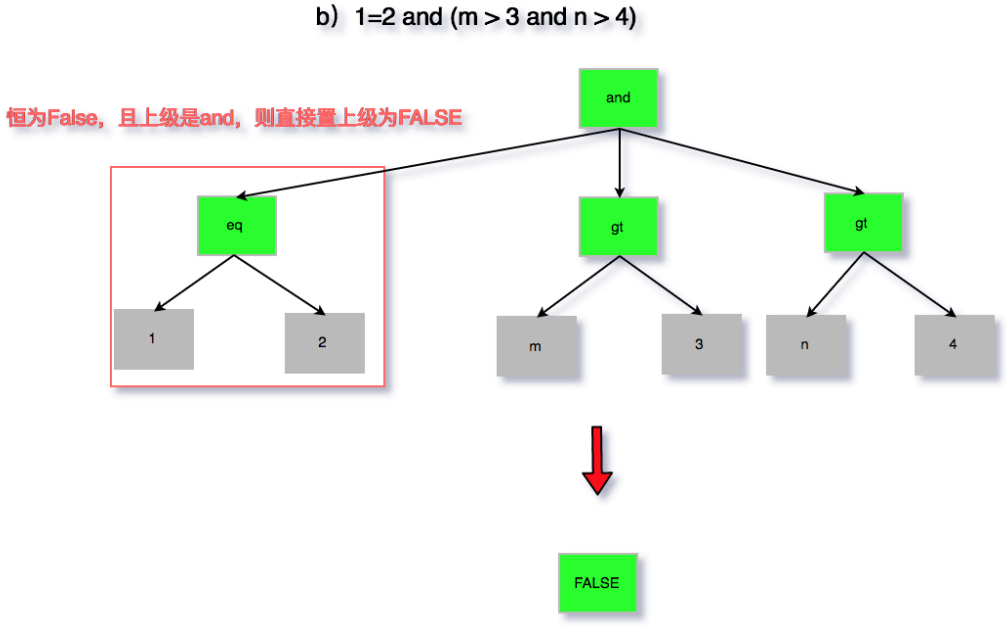


图6 无用条件去除b

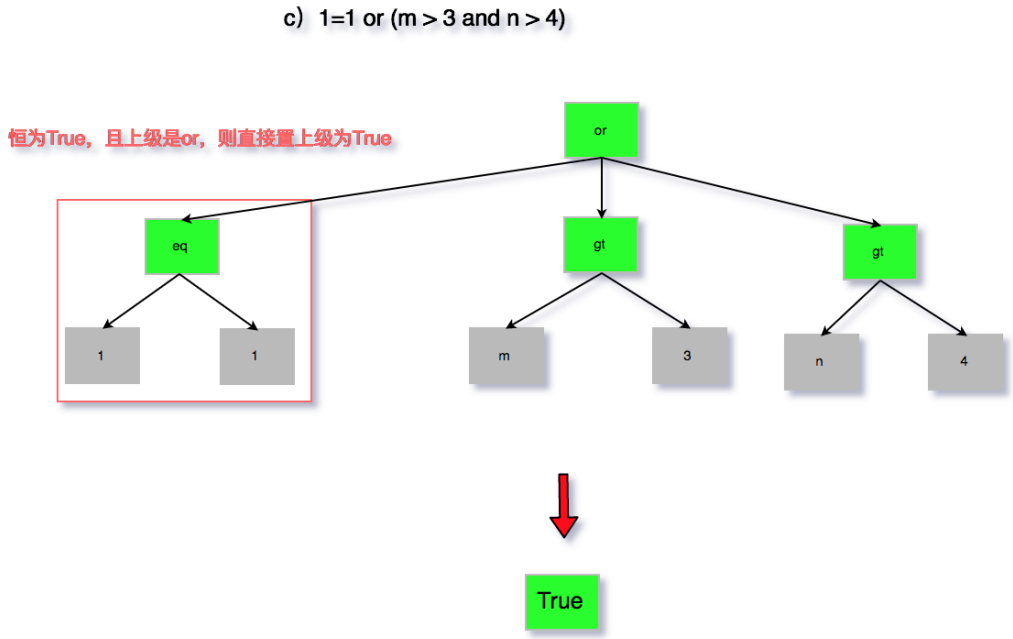


图7 无用条件去除c

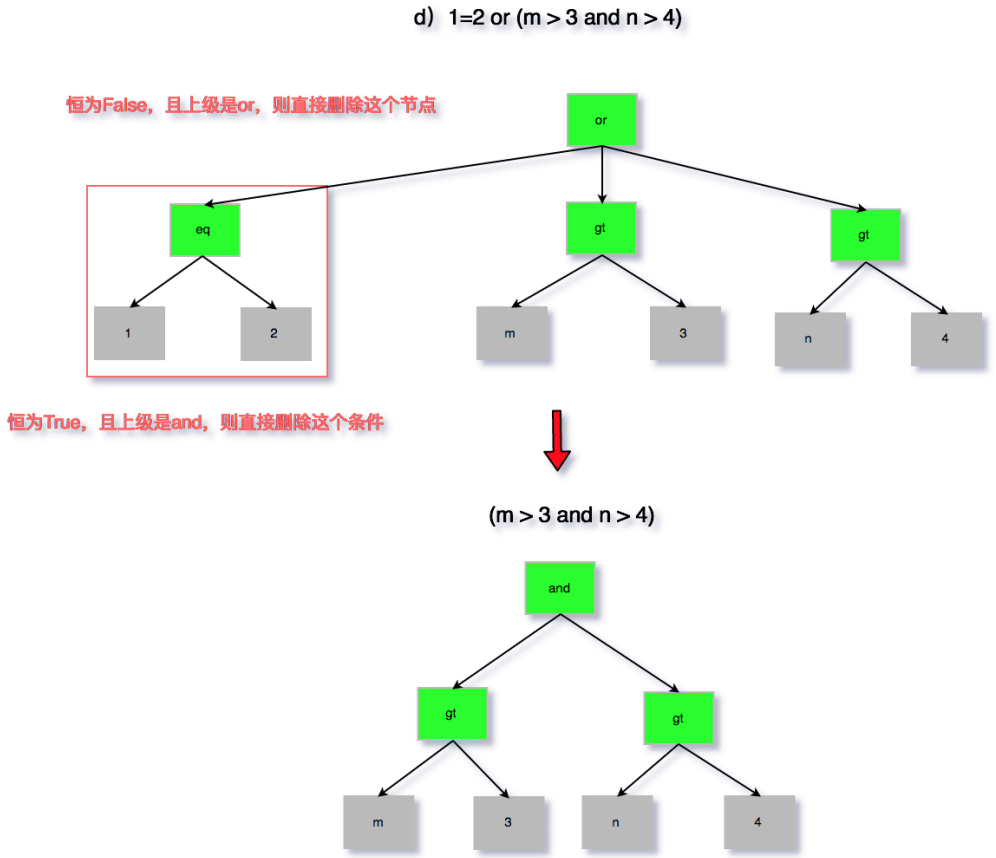


图8 无用条件去除d

如果对其代码实现有兴趣的同学，需要对MySQL中的一个重要数据结构Item类有所了解。因为其比较复杂，所以MySQL官方文档，专门介绍了 Item类 。阿里的MySQL小组，也有类似的 文章 。如需更详细的了解，就需要去查看源码中sql/item_*等文件。

SQL特征生成

为了确保数据库，这一系统基础组件稳定、高效运行，业界有很多辅助系统。比如慢查询系统、中间件系统。这些系统采集、收到SQL之后，需要对SQL进行归类，以便统计信息或者应用相关策略。归类时，通常需要获取SQL特征。比如SQL:

```
select username, ismale from userinfo where age > 20 and level > 5; ```
SQL特征为:
```sql
select username, ismale from userinfo where age > ? and level > ?
```

业界著名的慢查询分析工具pt-query-digest，通过正则表达式实现这个功能但是这类处理办法Bug较多。接下来就介绍如何使用SQL解析，完成SQL特征的生成。

SQL特征生成分两部分组成。

- a) 生成Token数组
- b) 根据Token数组，生成SQL特征

首先回顾在词法解析章节，我们介绍了SQL中的关键字，并且每个关键字都有一个16位的整数对应，而非关键字统一用ident表示，其也对应了一个16位整数。如下表:

标识	select	from	where	>	?	and	ident
整数	728	448	878	463	893	272	476

将一个SQL转换成特征的过程:

原SQL	select	username	from	userinfo	where	age	>	20
SQL特征	select	ident:length:value	from	ident:length:value	where	ident:length:value	>	?

在SQL解析过程中，可以很方便的完成Token数组的生成。而一旦完成Token数组的生成，就可以很简单的完成SQL特征的生成。SQL特征被广泛用于各个系统中，比如pt-query-digest需要根据特征对SQL归类，然而其基于正则表达式的实现有诸多bug。下面列举几个已知Bug:

原始SQL	pt-query-digest生成的特征	SQL解析器生成的特征
select * from email_template2 where id = 1	select * from mail_template? where id = ?	select * from email_template2 where id = ?
REPLACE INTO a VALUES(‘INSERT INTO foo VALUES (1),(2)’)	replace into a values(\‘insert into foo values(?+)	replace into a values (?)

因此可以看出SQL解析的优势是很明显的。
```

学习建议

最近，在对SQL解析器和优化器探索的过程中，从一开始的茫然无措到有章可循，也总结了一些心得体会，在这里跟大家分享一下。

- 首先，阅读相关图书书籍。图书能给我们系统认识解析器和优化器的角度。但是针对MySQL的此类图书市面上很少，目前中文作品可以看一看《数据库查询优化器的艺术：原理解析与SQL性能优化》。
- 其次，要阅读源码，但是最好以某个版本为基础，比如MySQL5.6.23，因为SQL解析、优化部分的代码在不断变化。尤其是在跨越大的版本时，改动力度大。
- 再次，多使用GDB调试，验证自己的猜测，检验阅读质量。

最后，需要写相关代码验证，只有写出来了才能算真正的掌握。

作者简介

- 广友，美团到店综合事业群MySQL DBA专家，2012年毕业于中国科学技术大学，2017年加入美团，长期致力于MySQL及周边工具的研究。
- 金龙，2014年加入美团，主要从事相关的数据库运维、高可用和相关的运维平台建设。对运维高可用与架构相关感兴趣的同学可以关注个人微信公众号“自己的设计师”，定期推送运维相关原创内容。
- 邢帆，美团到店综合事业群MySQL DBA，2017年研究生毕业后加入美团，目前已经对MySQL运维有一定经验，并编写了一些自动化脚本。

招聘信息

美团DBA团队招聘各类DBA人才，base北京上海均可。我们致力于为公司提供稳定、可靠、高效的在线存储服务，打造业界领先的数据库团队。这里有基于Redis Cluster构建的大规模分布式缓存系统Squirrel，也有基于Tair进行大刀阔斧改进的分布式KV存储系统Cellar，还有数千各类架构的MySQL实例，每天提供万亿级的OLTP访问请求。真正的海量、分布式、高并发环境。欢迎各位朋友推荐或自荐至jinlong.cai#dianping.com。

浅谈大型互联网企业入侵检测及防护策略

作者: 弼政

前言

如何知道自己所在的企业是否被入侵了？是没人来“黑”，还是因自身感知能力不足，暂时还无法发现？其实，入侵检测是每一个大型互联网企业都要面对的严峻挑战。价值越高的公司，面临入侵的威胁也越大，即便是Yahoo这样的互联网鼻祖，在落幕（被收购）时仍遭遇全量数据失窃的事情。安全无小事，一旦互联网公司被成功“入侵”，其后果将不堪想象。

基于“攻防对抗”的考量，本文不会提及具体的入侵检测模型、算法和策略，那些希望直接照搬“入侵策略”的同学可能会感到失望。但是我们会将一部分运营思路分享出来，请各位同行指点，如能对后来者起到帮助的作用，那就更好了，也欢迎大家跟我们交流探讨。

入侵的定义

典型的入侵场景：

> 黑客在很远的地方，通过网络远程控制目标的笔记本电脑/手机/服务器/网络设备，进而随意地读取目标的隐私数据，又或者使用目标系统上的功能，包括但不限于使用手机的麦克风监听目标，使用摄像头偷窥监控目标，使用目标设备的计算能力挖矿，使用目标设备的网络能力发动DDoS攻击等等。亦或是破解了一个服务的密码，进去查看敏感资料、控制门禁/红绿灯。以上这些都属于经典的入侵场景。

我们可以给入侵下一个定义：就是黑客在未经授权的情况下，控制、使用我方资源（包括但不限于读写数据、执行命令、控制资源等）达到各种目的。从广义上讲，黑客利用SQL注入漏洞窃取数据，或者拿到了目标域名在ISP中的帐号密码，以篡改DNS指向一个黑页，又或者找到了目标的社交帐号，在微博/QQ/邮箱上，对虚拟资产进行非授权的控制，都属于入侵的范畴。

针对企业的入侵检测

企业入侵检测的范围，多数情况下比较狭义：一般特指黑客对PC、系统、服务器、网络（包括办公网、生产网）控制的行为。

黑客对PC、服务器等主机资产的控制，最常见的方法是通过Shell去执行指令，获得Shell的这个动作叫做GetShell。

比如通过Web服务的上传漏洞，拿到WebShell，或者利用RCE漏洞直接执行命令/代码（RCE环境变相的提供了一个Shell）。另外，通过某种方式先植入“木马后门”，后续直接利用木马集成的SHELL功能对目标远程控制，这个也比较典型。

因此，入侵检测可以重点关注GetShell这个动作，以及GetShell成功之后的恶意行为（为了扩大战果，黑客多半会利用Shell进行探测、翻找窃取、横向移动攻击其它内部目标，这些区别于好人的特性也可以作为重要的特征）。

有一些同行（包括商业产品），喜欢报告GetShell之前的一些“外部扫描、攻击探测和尝试行为”，并美其名曰“态势感知”，告诉企业有人正在“试图攻击”。在笔者看来，实战价值并不大。包括美团在内的很多企业，基本上无时无刻都在遭受“不明身份”的攻击，知道了有人在“尝试”攻击，如果并不能有效地去行动，无法有效地对行动进行告警，除了耗费心力之外，并没有太大的实际价值。

当我们习惯“攻击”是常态之后，就会在这样的常态下去解决问题，可以使用什么加固策略，哪些可以实现常态化的运营，如果有什么策略无法常态化运营，比如需要很多人加班临时突击守着，那这个策略多半在不久之后就会逐渐消逝掉。跟我们做不做这个策略，并没有本质上的区别。

类似于SQL注入、XSS等一些不直接GetShell的Web攻击，暂时不在狭义的“入侵检测”考虑范围，建议可以划入“漏洞”、“威胁感知”等领域，另行再做探讨。当然，利用SQL注入、XSS等入口，进行了GetShell操作的，我们仍抓GetShell这个关键点，不必在乎漏洞入口在何处。

“入侵”和“内鬼”

与入侵接近的一种场景是“内鬼”。入侵本身是手段，GetShell只是起点，黑客GetShell的目标是为了之后对资源的控制和数据的窃取。而“内鬼”天然拥有合法的权限，可以合法接触敏感资产，但是基于工作以外的目的，他们对这些资源进行非法的处置，包括拷贝副本、转移外泄、篡改数据牟利等。

内鬼的行为不在“入侵检测”的范畴，一般从内部风险控制的视角进行管理和审计，比如职责分离、双人审计等。也有数据防泄密产品（DLP）对其进行辅助，这里不展开细说。

有时候，黑客知道员工A有权限接触目标资产，便定向攻击A，再利用A的权限把数据窃取走，也定性为“入侵”。毕竟A不是主观恶意的“内鬼”。如果不能在黑客攻击A的那一刻捕获，或者无法区分黑客控制的A窃取数据和正常员工A的访问数据，那这个入侵检测也是失败的。

入侵检测的本质

前文已经讲过，入侵就是黑客可以不经过我们的同意，来操作我们的资产，在手段上并没有任何的限制。那么如何找出入侵行为和合法正常行为的区别，将其跟合法行为进行分开，就是“入侵发现”。在算法模型上，这算是一个标记问题（入侵、非入侵）。

可惜的是，入侵这种动作的“黑”样本特别稀少，想通过大量的带标签的数据，有监督的训练入侵检测模型，找出入侵的规律比较难。因此，入侵检测策略开发人员，往往需要投入大量的时间，去提炼更精准的表达模型，或者花更多的精力去构造“类似入侵”的模拟数据。

一个经典的例子是，为了检测出WebShell，安全从业人员可以去GitHub上搜索一些公开的WebShell样本，数量大约不到1000个。而对于机器学习动辄百万级的训练需求，这些数据远远不够。况且GitHub上的这些样本集，从技术手法上来看，有单一技术手法生成的大量类似样本，也有一些对抗的手法样本缺失。因此，这样的训练，试图让AI去通过“大量的样本”掌握WebShell的特征并区分出它们，原则上不太可能完美地去实现。

此时，针对已知样本做技术分类，提炼更精准的表达模型，被称为传统的特征工程。而传统的特征工程往往被视为效率低下的重复劳动，但效果往往比较稳定，毕竟加一个技术特征就可以稳定发现一类WebShell。而构造大量的恶意样本，虽然有机器学习、AI等光环加持，但在实际环境中往往难以获得成功：自动生成的样本很难描述WebShell本来的含义，多半描述的是自动生成的算法特征。

另一个方面，入侵的区别是看行为本身是否“授权”，而授权与否本身是没有任何显著的区分特征的。因此，做入侵对抗的时候，如果能够通过某种加固，将合法的访问收敛到有限的通道，并且给该通道做出强有力的区分，也就能大大的降低入侵检测的成本。例如，对访问来源进行严格的认证，无论是自然人，还是程序API，都要求持有合法票据，而派发票据时，针对不同情况做多纬度的认证和授权，再用IAM针对这些票据记录和监控它们可以访问的范围，还能产生更底层的Log做异常访问模型感知。

这个全生命周期的风控模型，也是Google的BeyondCorp无边界网络得以实施的前提和基础。

因此，入侵检测的主要思路也就有2种：

- 根据黑特征进行模式匹配（例如WebShell关键字匹配）。
- 根据业务历史行为（生成基线模型），对入侵行为做异常对比（非白既黑），如果业务的历史行为不够收敛，就用加固的手段对其进行收敛，再挑出不合规的小众异常行为。

入侵检测与攻击向量

根据目标不同，可能暴露给黑客的攻击面会不同，黑客可能采用的入侵手法也就完全不同。比如，入侵我们的PC/笔记本电脑，还有入侵部署在机房/云上的服务器，攻击和防御的方法都有挺大的区别。

针对一个明确的“目标”，它被访问的渠道可能是有限集，被攻击的必经路径也有限。“攻击方法”+“目标的攻击面”的组合，被称为“攻击向量”。

因此，谈入侵检测模型效果时，需要先明确攻击向量，针对不同的攻击路径，采集对应的日志（数据），才可能做对应的检测模型。比如，基于SSH登录后的Shell命令数据集，是不能用于检测WebShell的行为。而基于网络流量采集的数据，也不可能感知黑客是否在SSH后的Shell环境中执行了什么命令。

基于此，如果有企业不提具体的场景，就说做好了APT感知模型，显然就是在“吹嘘”了。

所以，入侵检测得先把各类攻击向量罗列出来，每一个细分场景分别采集数据（HIDS+NIDS+WAF+RASP+应用层日志+系统日志+PC……），再结合公司的实际数据特性，作出适应公司实际情况的对应检测模型。不同公司的技术栈、数据规模、暴露的攻击面，都会对模型产生重大的影响。比如很多安全工作者特别擅长PHP下的WebShell检测，但是到了一个Java系的公司……

常见的入侵手法与应对

如果对黑客的常见入侵手法理解不足，就很难有的放矢，有时候甚至会陷入“政治正确”的陷阱里。比如渗透测试团队说，我们做了A动作，你们竟然没有发现，所以你们不行。而实际情况是，该场景可能不是一个完备的入侵链条，就算不发现该动作，对入侵检测效果可能也没有什么影响。每一个攻击向量对公司造成的危害，发生的概率如何进行排序，解决它耗费的成本和带来的收益如何，都需要有专业经验来做支撑与决策。

现在简单介绍一下，黑客入侵教程里的经典流程（完整过程可以参考杀伤链模型）：

入侵一个目标之前，黑客对该目标可能还不够了解，所以第一件事往往是“踩点”，也就是搜集信息，加深了解。比如，黑客需要知道，目标有哪些资产（域名、IP、服务），它们各自的状态如何，是否存在已知的漏洞，管理他们的人有谁（以及如何合法的管理的），存在哪些已知的泄漏信息（比如社工库里的密码等）……

一旦踩点完成，熟练的黑客就会针对各种资产的特性，酝酿和逐个验证“攻击向量”的可行性，下文列举了常见的攻击方式和防御建议。

高危服务入侵

所有的公共服务都是“高危服务”，因为该协议或者实现该协议的开源组件，可能存在已知的攻击方法（高级的攻击者甚至拥有对应的0day），只要你的价值足够高，黑客有足够的动力和资源去挖掘，那么当你把高危服务开启到互联网，面向所有人都打开的那一刻，就相当于为黑客打开了“大门”。

比如SSH、RDP这些运维管理相关的服务，是设计给管理员用的，只要知道密码/秘钥，任何人都能登录到服务器端，进而完成入侵。而黑客可能通过猜解密码（结合社工库的信息泄露、网盘检索或者暴力破解），获得凭据。事实上这类攻击由于过于常见，黑客早就做成了全自动化的全互联网扫描的蠕虫类工具，云上购买的一个主机如果设置了一个弱口令，往往在几分钟内就会感染蠕虫病毒，就是因为这类自动化的攻击者实在是太多了。

或许，你的密码设置得非常强壮，但是这并不是你可以把该服务继续暴露在互联网的理由，我们应该把这些端口限制好，只允许自己的IP（或者内部的堡垒主机）访问，彻底断掉黑客通过它入侵我们的可能。

与此类似的，MySQL、Redis、FTP、SMTP、MSSQL、Rsync等等，凡是自己用来管理服务器或者数据库、文件的服务，都不应该针对互联网无限制的开放。否则，蠕虫化的攻击工具会在短短几分钟内攻破我

们的服务，甚至直接加密我们的数据，甚至要求我们支付比特币，进行敲诈勒索。

还有一些高危服务存在RCE漏洞（远程命令执行），只要端口开放，黑客就能利用现成的exploit，直接GetShell，完成入侵。

防御建议： 针对每一个高危服务做入侵检测的成本较高，因为高危服务的具体所指非常的多，不一定存在通用的特征。所以，通过加固方式，收敛攻击入口性价比更高。禁止所有高危端口对互联网开放可能，这样能够减少90%以上的入侵概率。

Web入侵

随着高危端口的加固，黑客知识库里的攻击手法很多都会失效了。但是Web服务是现代互联网公司的主要服务形式，不可能都关掉。于是，基于PHP、Java、ASP、ASP.NET、Node、C写的CGI等等动态的Web服务漏洞，就变成了黑客入侵的最主要入口。

比如，利用上传功能直接上传一个WebShell，利用文件包含功能，直接引用执行一个远程的WebShell（或者代码），然后利用代码执行的功能，直接当作Shell的入口执行任意命令，解析一些图片、视频的服务，上传一个恶意的样本，触发解析库的漏洞……

Web服务下的应用安全是一个专门的领域（道哥还专门写了本《白帽子讲Web安全》），具体的攻防场景和对抗已经发展得非常成熟了。当然，由于它们都是由Web服务做为入口，所以入侵行为也会存在某种意义上的共性。相对而言，我们比较容易能够找到黑客GetShell和正常业务行为的一些区别。

针对Web服务的入侵痕迹检测，可以考虑采集WAF日志、Access Log、Auditd记录的系统调用，或者Shell指令，以及网络层面Response相关的数据，提炼出被攻击成功的特征，建议我们将主要的精力放在这些方面。

0day入侵

通过泄漏的工具包来看，早些年NSA是拥有直接攻击Apache、Nginx这些服务的0day武器的。这意味着对手很可能完全不用在乎我们的代码和服务写成什么样，拿0day一打，神不知鬼不觉就GetShell了。

但是对于入侵检测而言，这并不可怕：因为无论对手利用什么漏洞当入口，它所使用的Shellcode和之后的行为本身依然有共性。Apache存在0day漏洞被攻击，还是一个PHP页面存在低级的代码漏洞被利用，从入侵的行为上来看，说不定是完全一样的，入侵检测模型还可以通用。

所以，把精力聚焦在有黑客GetShell入口和之后的行为上，可能比关注漏洞入口更有价值。当然，具体的漏洞利用还是要实际跟进，然后验证其行为是否符合预期。

办公终端入侵

绝大多数APT报告里，黑客是先对人（办公终端）下手，比如发个邮件，哄骗我们打开后，控制我们的PC，再进行长期的观察/翻阅，拿到我们的合法凭据后，再到内网漫游。所以这些报告，多数集中在描述黑客用的木马行为以及家族代码相似度上。而反APT的产品、解决方案，多数也是在办公终端的系统调用层面，用类似的方法，检验“免杀木马”的行为。

因此，EDR类的产品+邮件安全网关+办公网出口的行为审计+APT产品的沙箱等，联合起来，可以采集到对应的数据，并作出相似的入侵检测感知模型。而最重要的一点，是黑客喜欢关注内部的重要基础设施，包括但不限于AD域控、邮件服务器、密码管理系统、权限管理系统等，一旦拿下，就相当于成为了内网的“上帝”，可以为所欲为。所以对公司来说，重要基础设施要有针对性的攻防加固讨论，微软针对AD的攻防甚至还发过专门的加固白皮书。

入侵检测基本原则

不能把每一条告警都彻底跟进的模型，等同于无效模型。入侵发生后，再辩解之前其实有告警，只是太多了没跟过来/没查彻底，这是“马后炮”，等同于不具备发现能力，所以对于日均告警成千上万的产品，安全运营人员往往表示很无奈。

我们必须屏蔽一些重复发生的相似告警，以集中精力把每一个告警都闭环掉。这会产生白名单，也就是漏报，因此模型的漏报是不可避免的。

由于任何模型都会存在漏报，所以我们必须在多个纬度上做多个模型，形成关联和纵深。假设WebShell静态文本分析被黑客变形绕过了，在RASP（运行时环境）的恶意调用还可以进行监控，这样可以选择接受单个模型的漏报，但在整体上仍然具备发现能力。

既然每一个单一场景的模型都有误报漏报，我们做什么场景，不做什么场景，就需要考虑“性价比”。比如某些变形的WebShell可以写成跟业务代码非常相似，人的肉眼几乎无法识别，再追求一定要在文本分析上进行对抗，就是性价比很差的决策。如果通过RASP的检测方案，其性价比更高一些，也更具可行性一些。

我们不太容易知道黑客所有的攻击手法，也不太可能针对每一种手法都建设策略（考虑到资源总是稀缺的）。所以针对重点业务，需要可以通过加固的方式（还需要常态化监控加固的有效性），让黑客能攻击的路径极度收敛，仅在关键环节进行对抗。起码能针对核心业务具备兜底的保护能力。

基于上述几个原则，我们可以知道一个事实，或许我们永远不可能在单点上做到100%发现入侵，但是我们可以通过一些组合方式，让攻击者很难绕过所有的点。

当老板或者蓝军挑战，某个单点的检测能力有缺失时，如果为了“政治正确”，在这个单点上进行无止境的投入，试图把单点做到100%能发现的能力，很多时候可能只是在试图制造一个“永动机”，纯粹浪费人力、资源，而不产生实际的收益。将节省下来的资源，高性价比的布置更多的纵深防御链条，效果显然会更好。

入侵检测产品的主流形态

入侵检测终究是要基于数据去建模，比如针对WebShell的检测，首先要识别Web目录，再对Web目录下的文件进行文本分析，这需要做一个采集器。基于Shell命令的入侵检测模型，需要获取所有Shell命令，这可能要Hook系统调用或者劫持Shell。基于网络IP信誉、流量payload进行检测，或者基于邮件网关对内容的检查，可能要植入网络边界中，对流量进行旁路采集。

也有一些集大成者，基于多个Sensor，将各方日志进行采集后，汇总在一个SOC或者SIEM，再交由大数据平台进行综合分析。因此，业界的入侵检测相关的产品大致上就分成了以下的形态：

- 主机Agent类：黑客攻击了主机后，在主机上进行的动作，可能会产生日志、进程、命令、网络等痕迹，那么在主机上部署一个采集器（也内含一部分检测规则），就叫做基于主机的入侵检测系统，简称HIDS。
    - 典型的产品：OSSEC、青藤云、安骑士、安全狗，Google最近也发布了一个Alpha版本的类似产品 Cloud Security Command Center。当然，一些APT厂商，往往也有在主机上的Sensor/Agent，比如FireEye等。
- 网络检测类：由于多数攻击向量是会通过网络对目标投放一些payload，或者控制目标的协议本身具备强特征，因此在网络层面具备识别的优势。
    - 典型的产品：Snort到商业的各种NIDS/NIPS，对应到APT级别，则还有类似于FireEye的NX之类的产品。
- 日志集中存储分析类：这一类产品允许主机、网络设备、应用都输出各自的日志，集中到一个统一的后台，在这个后台，对各类日志进行综合的分析，判断是否可以关联的把一个入侵行为的多个路径刻画出来。例如A主机的的Web访问日志里显示遭到了扫描和攻击尝试，继而主机层面多了一个陌生的进程和网络连接，最后A主机对内网其它主机进行了横向渗透尝试。
    - 典型的产品：LogRhythm、Splunk等SIEM类产品。
- APT沙箱：沙箱类产品更接近于一个云端版的高级杀毒软件，通过模拟执行观测行为，以对抗未知样本弱特征的特点。只不过它需要一个模拟运行的过程，性能开销较大，早期被认为是“性价比不高”的解决方案，但由于恶意文件在行为上的隐藏要难于特征上的对抗，因此现在也成为了APT产品的核心组件。通过网络流量、终端采集、服务器可疑样本提取、邮件附件提炼等拿到的未知样本，都可以提交到沙箱里跑一下行为，判断是否恶意。
    - 典型产品：FireEye、Palo Alto、Symantec、微步。
- 终端入侵检测产品：移动端目前还没有实际的产品，也不太有必要。PC端首先必备的是杀毒软件，如果能够检测到恶意程序，一定程度上能够避免入侵。但是如果碰到免杀的高级0day和木马，杀毒软件可能会被绕过。借鉴服务器上HIDS的思路，也诞生了EDR的概念，主机除了有本地逻辑之外，更重要的是会采集更多的数据到后端，在后端进行综合分析和联动。也有人说下一代杀毒软件里都会带上EDR的能力，只不过目前销售还是分开在卖。
    - 典型产品：杀毒软件有Bit9、SEP、赛门铁克、卡巴斯基、McAfee ；EDR产品不枚举了，腾讯的iOA、阿里的阿里郎，一定程度上都是可以充当类似的角色；

入侵检测效果评价指标

首先，主动发现的入侵案例/所有入侵 = 主动发现率。这个指标一定是最直观的。比较麻烦的是分母，很多真实发生的入侵，如果外部不反馈，我们又没检测到，它就不会出现在分母里，所以有效发现率总是虚高的，谁能保证当前所有的入侵都发现了呢？（但是实际上，只要入侵次数足够多，不管是SRC收到的情报，还是“暗网”上报出来的一个大新闻，把客观上已经知悉的入侵列入分母，总还是能计算出一个主动发现率的。）

另外，真实的入侵其实是一个低频行为，大型的互联网企业如果一年到头成百上千的被入侵，肯定也不正常。因此，如果很久没出现真实入侵案例，这个指标长期不变化，也无法刻画入侵检测能力是否在提升。

所以，我们一般还会引入两个指标来观测：

- 蓝军对抗主动发现率
- 已知场景覆盖率

蓝军主动高频对抗和演习，可以弥补真实入侵事件低频的不足，但是由于蓝军掌握的攻击手法往往也是有限的，他们多次演习后，手法和场景可能会被罗列完毕。假设某一个场景建设方尚未补齐能力，蓝军同样的姿势演习100遍，增加100个未发现的演习案例，对建设方而言并没有更多的帮助。所以，把已知攻击手法的建成覆盖率拿出来，也是一个比较好的评价指标。

入侵检测团队把精力聚焦在已知攻击手法的优先级评估和快速覆盖上，对建设到什么程度是满足需要的，要有自己的专业判断（参考入侵检测原则里的“性价比”原则）。

而宣布建成了一个场景的入侵发现能力，是要有基本的验收原则的：

- 该场景日均工单 < X单，峰值 < Y单；当前所有场景日平均<XX，峰值 <YY，超出该指标的策略不予接收，因为过多的告警会导致有效信息被淹没，反而导致此前具备的能力被干扰，不如视为该场景尚未具备对抗能力。
- 同一个事件只告警首次，多次出现自动聚合。
- 具备误报自学习能力。
- 告警具备可读性（有清晰的风险阐述、关键信息、处理指引、辅助信息或者索引，便于定性），不鼓励Key-Value模式的告警，建议使用自然语言描述核心逻辑和响应流程。
- 有清晰的说明文档，自测报告（就像交付了一个研发产品，产品文档和自测过程是质量的保障）。
- 有蓝军针对该场景实战验收报告。
- 不建议调用微信、短信等接口发告警（告警和事件的区别是，事件可以闭环，告警只是提醒），统一的告警事件框架可以有效的管理事件确保闭环，还能提供长期的基础运营数据，比如止损效率、误报量/率。

策略人员的文档应当说明当前模型对哪些情况具备感知能力，哪些前提下会无法告警（考验一个人对该场景和自己模型的理解能力）。通过前述判断，可以对策略的成熟度形成自评分，0-100自由大致估算。单个场景往往很难达到100分，但那并没有关系，因为从80分提升到100分的边际成本可能变的很高。不建议追求极致，而是全盘审视，是否快速投入到下一个场景中去。

如果某个不到满分的场景经常出现真实对抗，又没有交叉的其它策略进行弥补，那自评结论可能需要重审并提高验收的标准。至少解决工作中实际遇到的Case要优先考虑。

影响入侵检测的关键要素

讨论影响入侵检测的要素时，我们可以简单看看，曾经发生过哪些错误导致防守方不能主动发现入侵：

- 依赖的数据丢失，比如HIDS在当事机器上，没部署安装/Agent挂了/数据上报过程丢失了/Bug了，或者后台传输链条中丢失数据。
- 策略脚本Bug，没启动（事实上我们已经失去了这个策略感知能力了）。
- 还没建设对应的策略（很多时候入侵发生了才发现这个场景我们还没来得及建设对应的策略）。
- 策略的灵敏度/成熟度不够（比如扫描的阈值没达到，WebShell用了变形的对抗手法）。
- 模型依赖的部分基础数据错误，做出了错误的判断。

- 成功告警了，但是负责应急同学错误的判断/没有跟进/辅助信息不足以定性，没有行动起来。

所以实际上，要让一个入侵事件被捕获，我们需要入侵检测系统长时间、高质量、高可用的运行。这是一件非常专业的工作，超出了绝大多数安全工程师能力和意愿的范畴。所以建议指派专门的运营人员对以下目标负责：

- 数据采集的完整性（全链路的对账）。
- 每一个策略时刻工作正常（自动化拨测监控）。
- 基础数据的准确性。
- 工单运营支撑平台及追溯辅助工具的便捷性。

可能有些同学会想，影响入侵检测的关键要素，难道不是模型的有效性么？怎么全是这些乱七八糟的东西？

实际上，大型互联网企业的入侵检测系统日均数据量可能到达数百T，甚至更多。涉及到数十个业务模块，成百上千台机器。从数字规模上来说，不亚于一些中小型企业的整个数据中心。这样复杂的一个系统，要长期维持在高可用标准，本身就需要有SRE、QA等辅助角色的专业化支持。如果仅依靠个别安全工程师，很难让其研究安全攻防的时候，又兼顾到基础数据质量、服务的可用性和稳定性、发布时候的变更规范性、各类运营指标和运维故障的及时响应。最终的结果就是能力范围内可以发现的入侵，总是有各种意外“恰好”发现不了。

所以，笔者认为，以多数安全团队运营质量之差，其实根本轮不到拼策略（技术）。当然，一旦有资源投入去跟进这些辅助工作之后，入侵检测就真的需要拼策略了。

此时，攻击手法有那么多，凭什么先选择这个场景建设？凭什么认为建设到某程度就足够满足当下的需要了？凭什么选择发现某些样本，而放弃另一些样本的对抗？

这些看似主观性的东西，非常考验专业判断力。而且在领导面前很容易背上“责任心不足”的帽子，比如为困难找借口而不是为目标找方法，这个手法黑客攻击了好多次，凭什么不解决，那个手法凭什么说在视野范围内，但是要明年再解决？

如何发现APT？

所谓APT，就是高级持续威胁。既然是高级的，就意味着木马很大可能是免杀的（不能靠杀毒软件或者普通的特征发现），利用的漏洞也是高级的（加固到牙齿可能也挡不住敌人进来的步伐），攻击手法同样很高级（攻击场景可能我们都没有见过）。

所以，实际上APT的意思，就约等于同于不能被发现的入侵。然而，业界总还有APT检测产品，解决方案的厂商在混饭吃，他们是怎么做的呢？

- 木马免杀的，他们用沙箱+人工分析，哪怕效率低一些，还是试图做出定性，并快速的把IOC（威胁情报）同步给其它客户，发现1例，全球客户都具备同样的感知能力。
- 流量加密变形对抗的，他们用异常检测的模型，把一些不认识的可疑的IP关系、payload给识别出来。当然，识别出来之后，也要运营人员跟进得仔细，才能定性。

- 攻击手法高级的，他们还是会假定黑客就用鱼叉、水坑之类的已知手法去执行，然后在邮箱附件、PC终端等环节采集日志，对用户行为进行分析，UEBA试图寻找出用户异于平常的动作。

那么，我们呢？笔者也没有什么好的办法，可以发现传说中的“免杀”的木马，但是我们可以针对已知的黑客攻击框架（比如Metasploit、Cobalt Strike）生成的样本、行为进行一些特征的提取。我们可以假设已经有黑客控制了某一台机器，但是它试图进行横向扩散的时候，我们有一些模型可以识别这个主机的横向移动行为。

笔者认为，世界上不存在100%能发现APT的方法。但是我们可以等待实施APT的团队犯错，只要我们的纵深足够的多，信息足够不对称，想要完全不触碰我们所有的铃铛，绝对存在一定的困难。

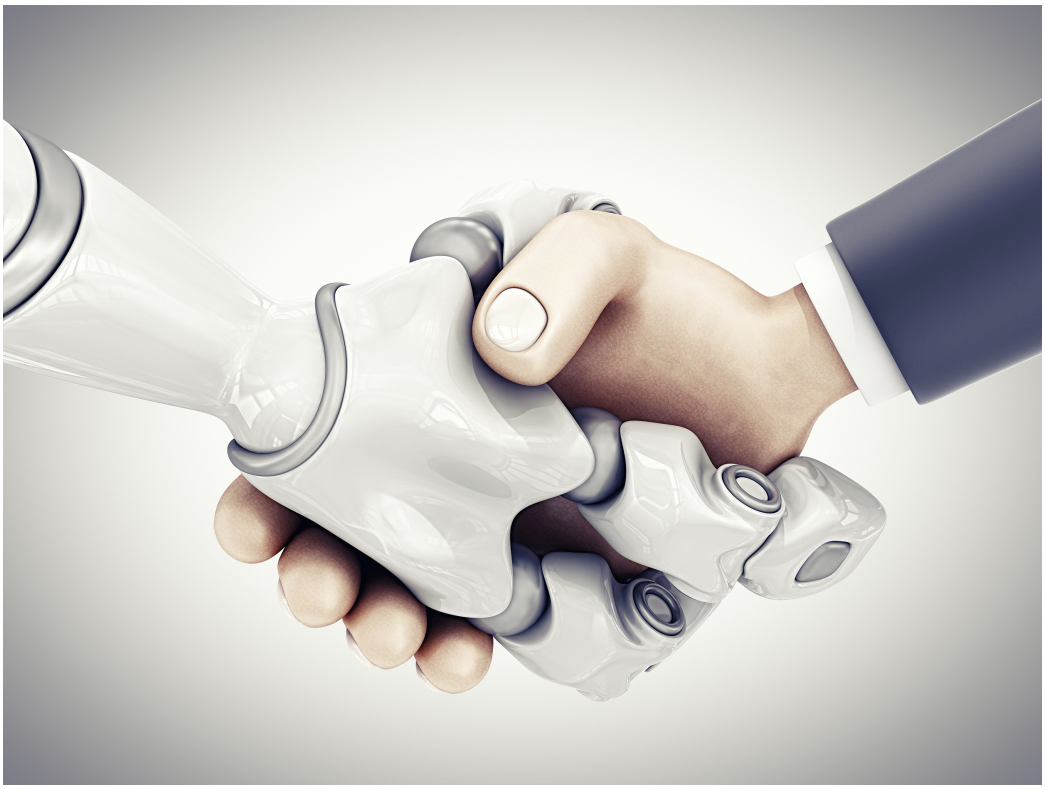
甚至，攻击者如果需要小心翼翼的避开所有的检测逻辑，可能也会给对手一种心理上的震慑，这种震慑可能会延缓对手接近目标的速度，拉长时间。而在这个时间里，只要他犯错，就轮到我们出场了。

前面所有的高标准，包括高覆盖、低误报，强制每一个告警跟进到底，“掘地三尺”的态度，都是在等待这一刻。抓到一个值得敬佩的对手，那种成就感，还是很值得回味的。

所以，希望所有从事入侵检测的安全同行们都能坚持住，即使听过无数次“狼来了”，下一次看到告警，依然可以用最高的敬畏心去迎接对手（告警虐我千百遍，我待告警如初恋）。

AI在入侵检测领域的正确姿势

最近这两年，如果不谈AI的话，貌似故事就不会完整。只不过，随着AI概念的火爆，很多人已经把传统的数据挖掘、统计分析等思想，比如分类、预测、聚类、关联之类的算法，都一律套在AI的帽子里。



其实AI是一种现代的方法，在很多地方有非常实际的产出了。以WebShell的文本分析为例，我们可能需要花很长很长的时间，才能把上千个样本里隐含的几十种样本技术类型拆分开，又花更长的时间去一一建

设模型（是的，在这样的场景下，特征工程真的是一个需要更长时间的工作）。

而使用AI，做好数据打标的工作，训练、调参，很快就能拿到一个实验室环境不那么过拟合的模型出来，迅速投产到生产环境上。熟练一点可能1–2个月就能做完了。

在这种场景下，AI这种现代的方法，的确能极大地提高效率。但问题是，前文也提到过了，黑客的攻击黑样本、WebShell的样本，往往极其稀缺，它不可能是完备的能够描述黑客入侵的完整特征的。因此，AI产出的结果，无论是误报率还是漏报率，都会受训练方法和输入样本的影响较大，我们可以借助AI，但绝对不能完全交给AI。

安全领域一个比较常见的现象是，将场景转变成标记问题，要难过于通过数学模型把标记的解给求出来。此时往往需要安全专家先行，算法专家再跟上，而不能直接让算法专家“孤军奋战”。

针对一个具体的攻击场景，怎么样采集对应的入侵数据，思考这个入侵动作和正常行为的区别，这个特征的提取过程，往往决定了模型最终的效果。特征决定了效果的上限，而算法模型只能决定了有多接近这个上限。

此前，笔者曾见过一个案例，AI团队产出了一个实验室环境效果极佳，误报率达到1/1000000的WebShell模型，但是投放到生产环境里初期日均告警6000单，完全无法运营，同时还存在不少漏报的情况。这些情况随着安全团队和AI工程师共同的努力，后来逐渐地解决。但是并未能成功的取代原有的特征工程模型。

目前业界有许多产品、文章在实践AI，但遗憾的是，这些文章和产品大多“浅尝辄止”，没有在真实的环境中实践运营效果。一旦我们用前面的标准去要求它，就会发现，AI虽然是个好东西，但是绝对只是个“半成品”。真正的运营，往往需要传统的特征工程和AI并行，也需要持续地进行迭代。

未来必然是AI的天下，但是有多少智能，前面可能就要铺垫多少人工。愿与同行们一起在这个路上继续探索下去，多多交流分享。

关于美团安全

美团安全部的大多数核心开发人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级IDC规模攻防对抗的经验。安全部也不乏CVE“挖掘圣手”，有受邀在Black Hat等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。

目前，美团安全部涉及的技术包括渗透测试、Web防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级IDC规模、数十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，包括网络层、虚拟化/容器层、Server 软件层（内核态/用户态）、语言虚拟机层(JVM/JS V8)、Web应用层、数据访问层等，并能够基于大数据+机器学习技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领

域不断探索的机会。

安利个小广告

美团安全部正在招募Web&二进制攻防、后台&系统开发、机器学习&算法等各路小伙伴。如果你想加入我们，欢迎简历请发至邮箱zhaoyan17@meituan.com

具体职位信息可参考 这里

美团安全应急响应中心MTSRC主页： security.meituan.com

美团针对Redis Rehash机制的探索和实践

作者: 春林 赵磊

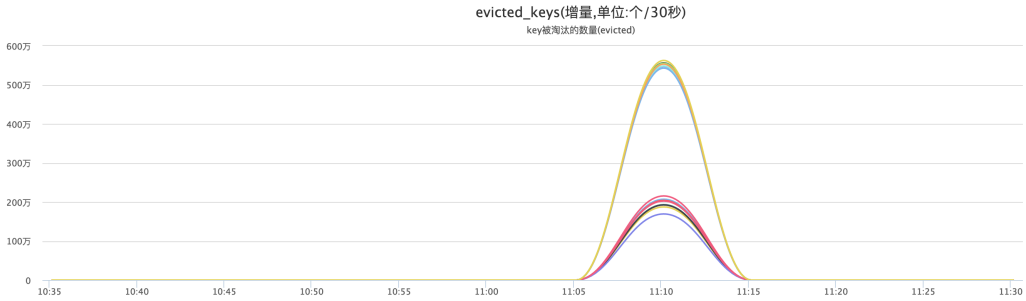
背景

Squirrel（松鼠）是美团技术团队基于Redis Cluster打造的缓存系统。经过不断的迭代研发，目前已形成一整套自动化运维体系：涵盖一键运维集群、细粒度的监控、支持自动扩缩容以及热点Key监控等完整的解决方案。同时服务端通过Docker进行部署，最大程度的提高运维的灵活性。分布式缓存Squirrel产品自2015年上线至今，已在美团内部广泛使用，存储容量超过60T，日均调用量也超过万亿次，逐步成为美团目前最主要的缓存系统之一。

随着使用的量和场景不断深入，Squirrel团队也不断发现Redis的若干”坑”和不足，因此也在持续的改进Redis以支撑美团内部快速发展的业务需求。本文尝试分享在运维过程中踩过的Redis Rehash机制的一些坑以及我们的解决方案，其中在高负载情况下物理机发生丢包的现象和解决方案已经写成博客。感兴趣的同学可以参考： Redis 高负载下的中断优化 。

案例

Redis 满容状态下由于Rehash导致大量Key驱逐



我们先来看一张监控图（上图，我们线上真实案例），Redis在满容有驱逐策略的情况下，Master/Slave均有大量的Key驱逐淘汰，导致Master/Slave 主从不一致。

Root Cause 定位

由于Slave内存区域比Master少一个repl-backlog buffer（线上一般配置为128M），正常情况下Master到达满容后根据驱逐策略淘汰Key并同步给Slave。所以Slave这种情况下不会因满容触发驱逐。

按照以往经验，排查思路主要聚焦在造成Slave内存陡增的问题上，包括客户端连接、输入/输出缓冲区、业务数据存取访问、网路抖动等导致Redis内存陡增的所有外部因素，通过Redis监控和业务链路监控均没有定位成功。

于是，通过梳理Redis源码，我们尝试将目光投向了Redis会占用内存开销的一个重要机制——Redis Rehash。

Redis Rehash 内部实现

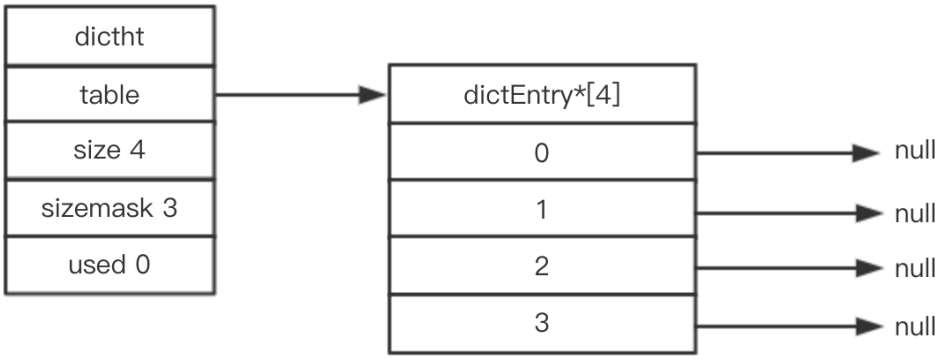
在Redis中，键值对（Key-Value Pair）存储方式是由字典（Dict）保存的，而字典底层是通过哈希表来实现的。通过哈希表中的节点保存字典中的键值对。类似Java中的HashMap，将Key通过哈希函数映射到哈希表节点位置。

接下来我们一步步来分析Redis Dict Reash的机制和过程。

(1) Redis 哈希表结构体：

```
/* hash表结构定义 */
typedef struct dictht {
    dictEntry **table;   // 哈希表数组
    unsigned long size;  // 哈希表的大小
    unsigned long sizemask; // 哈希表大小掩码
    unsigned long used;  // 哈希表现有节点的数量
} dictht;
```

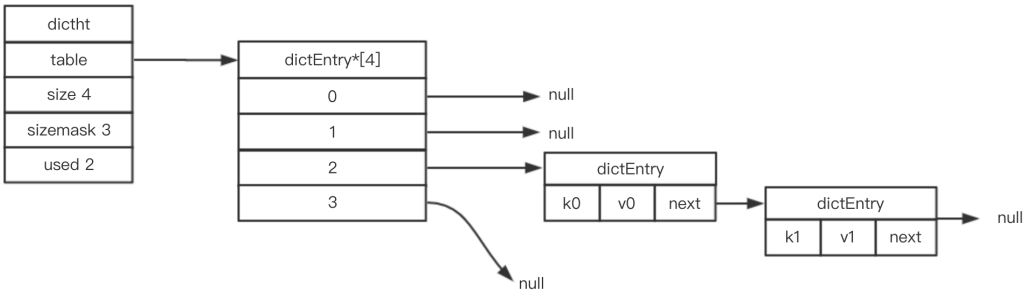
实体化一下，如下图所指一个大小为4的空哈希表（ Redis默认初始化值为4）：



(2) Redis 哈希桶

Redis 哈希表中的table数组存放着哈希桶结构（dictEntry），里面就是Redis的键值对；类似Java实现的HashMap，Redis的dictEntry也是通过链表（next指针）方式来解决hash冲突：

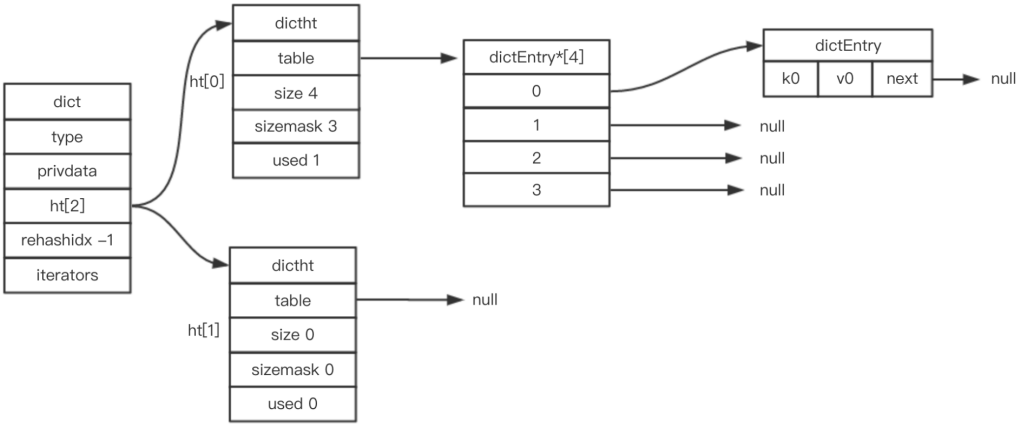
```
/* 哈希桶 */
typedef struct dictEntry {
    void *key;      // 键定义
    // 值定义
    union {
        void *val;     // 自定义类型
        uint64_t u64; // 无符号整形
        int64_t s64;  // 有符号整形
        double d;     // 浮点型
    } v;
    struct dictEntry *next;  //指向下一个哈希表节点
} dictEntry;
```



(3) 字典

Redis Dict 中定义了两张哈希表，是为了后续字典的扩展作Rehash之用：

```
/* 字典结构定义 */
typedef struct dict {
    dictType *type;  // 字典类型
    void *privdata;  // 私有数据
    dictht ht[2];    // 哈希表[两个]
    long rehashidx;   // 记录rehash 进度的标志，值为-1表示rehash未进行
    int iterators;   //  当前正在迭代的迭代器数
} dict;
```



总结一下：

- 在Cluster模式下，一个Redis实例对应一个RedisDB(db0);
- 一个RedisDB对应一个Dict;
- 一个Dict对应2个Dictht，正常情况只用到ht[0]；ht[1] 在Rehash时使用。

如上，我们回顾了一下Redis KV存储的实现。Redis内部还有其他结构体，由于跟Rehash不涉及，不再赘述。

我们知道当HashMap中由于Hash冲突（负载因子）超过某个阈值时，出于链表性能的考虑，会进行Resize的操作。Redis也一样【Redis中通过dictExpand()实现】。我们看一下Redis中的实现方式：

```
/* 根据相关触发条件扩展字典 */
static int _dictExpandIfNeeded(dict *d)
{
    if (dictIsRehashing(d)) return DICT_OK;  // 如果正在进行Rehash，则直接返回
    if (d->ht[0].size == 0) return dictExpand(d, DICT_HT_INITIAL_SIZE);  // 如果ht[0]字典为空，则创建并初始化ht[0]
    /* (ht[0].used/ht[0].size)>=1前提下，
       当满足dict_can_resize=1或ht[0].used/t[0].size>5时，便对字典进行扩展 */
    if (d->ht[0].used >= d->ht[0].size &&
        (dict_can_resize ||
         d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
    {
        return dictExpand(d, d->ht[0].used*2);   // 扩展字典为原来的2倍
    }
```

```
    return DICT_OK;
}


...

/* 计算存储Key的bucket的位置 */
static int _dictKeyIndex(dict *d, const void *key)
{
    unsigned int h, idx, table;
    dictEntry *he;

    /* 检查是否需要扩展哈希表，不足则扩展 */
    if (_dictExpandIfNeeded(d) == DICT_ERR)
        return -1;
    /* 计算Key的哈希值 */
    h = dictHashKey(d, key);
    for (table = 0; table <= 1; table++) {
        idx = h & d->ht[table].sizemask;  //计算Key的bucket位置
        /* 检查节点上是否存在新增的Key */
        he = d->ht[table].table[idx];
        /* 在节点链表检查 */
        while(he) {
            if (key==he->key || dictCompareKeys(d, key, he->key))
                return -1;
            he = he->next;
        }
        if (!dictIsRehashing(d)) break;  // 扫完ht[0]后，如果哈希表不在rehashing，则无需再扫ht[1]
    }
    return idx;
}

...

/* 将Key插入哈希表 */
dictEntry *dictAddRaw(dict *d, void *key)
{
    int index;
    dictEntry *entry;
    dictht *ht;

    if (dictIsRehashing(d)) _dictRehashStep(d);  // 如果哈希表在rehashing，则执行单步rehash

    /* 调用_dictKeyIndex() 检查键是否存在，如果存在则返回NULL */
    if ((index = _dictKeyIndex(d, key)) == -1)
        return NULL;


    ht = dictIsRehashing(d) ? &d->ht[1] : &d->ht[0];
    entry = zmalloc(sizeof(*entry));   // 为新增的节点分配内存
    entry->next = ht->table[index];  //  将节点插入链表表头
    ht->table[index] = entry;   // 更新节点和桶信息
    ht->used++;    //  更新ht

    /* 设置新节点的键 */
    dictSetKey(d, entry, key);
    return entry;
}

...
/* 添加新键值对 */
int dictAdd(dict *d, void *key, void *val)
{
    dictEntry *entry = dictAddRaw(d,key);  // 添加新键

    if (!entry) return DICT_ERR;  // 如果键存在，则返回失败
    dictSetVal(d, entry, val);   // 键不存在，则设置节点值
    return DICT_OK;
}
```

继续dictExpand的源码实现：

```
int dictExpand(dict *d, unsigned long size)
{
    dictht n; // 新哈希表
    unsigned long realsize = _dictNextPower(size);  // 计算扩展或缩放新哈希表的大小(调用下面函数_dictNextPower())

    /* 如果正在rehash或者新哈希表的大小小于现已使用，则返回error */
    if (dictIsRehashing(d) || d->ht[0].used > size)
        return DICT_ERR;
```

```
    /* 如果计算出哈希表size与现哈希表大小一样，也返回error */
    if (realsize == d->ht[0].size) return DICT_ERR;

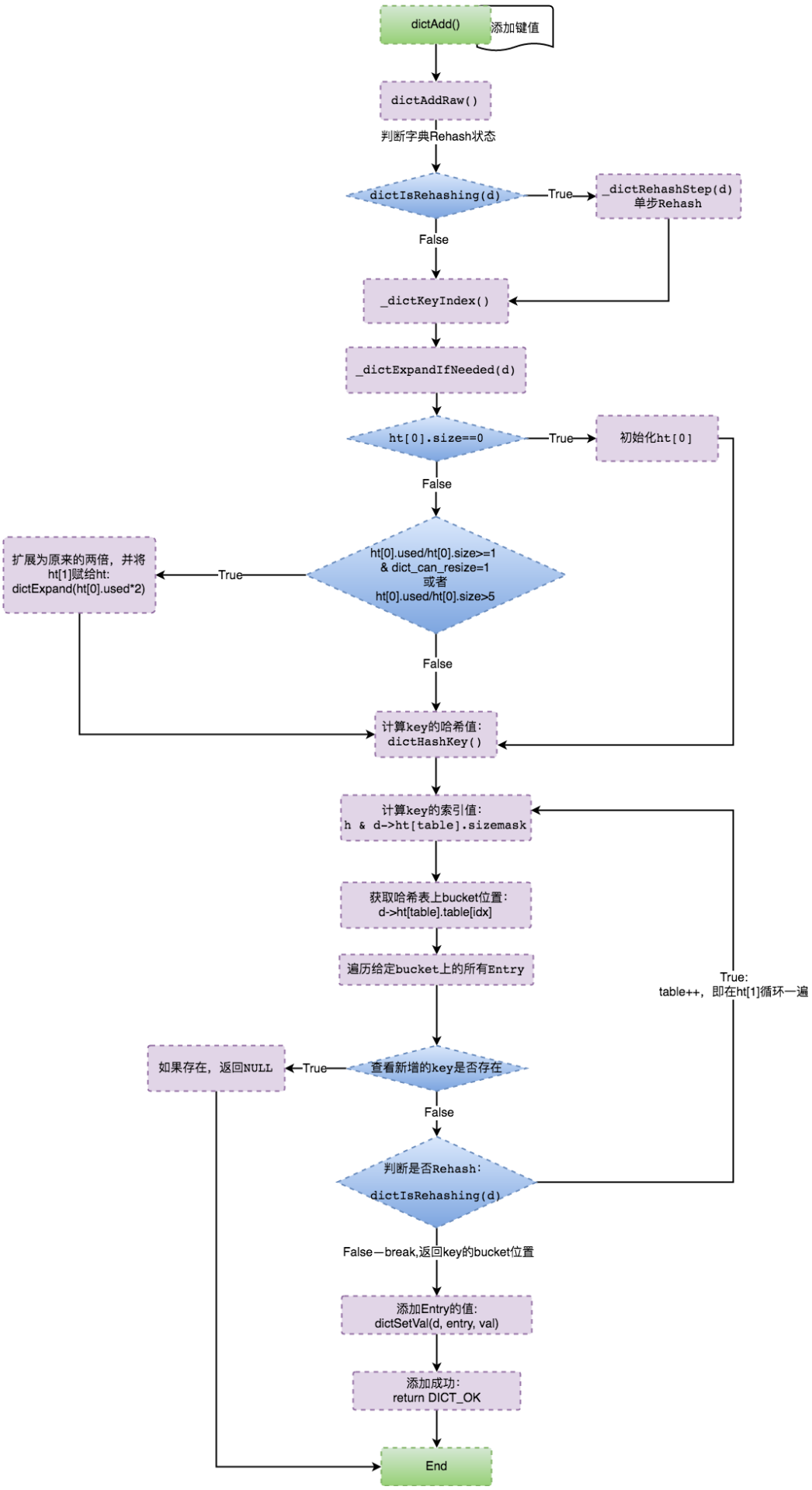
    /* 初始化新哈希表 */
    n.size = realsize;
    n.sizemask = realsize-1;
    n.table = zcalloc(realsize*sizeof(dictEntry*));  // 为table指向dictEntry 分配内存
    n.used = 0;

    /* 如果ht[0] 为空，则初始化ht[0]为当前键值对的哈希表 */
    if (d->ht[0].table == NULL) {
        d->ht[0] = n;
        return DICT_OK;
    }

    /* 如果ht[0]不为空，则初始化ht[1]为当前键值对的哈希表，并开启渐进式rehash模式 */
    d->ht[1] = n;
    d->rehashidx = 0;
    return DICT_OK;
}
...
static unsigned long _dictNextPower(unsigned long size) {
    unsigned long i = DICT_HT_INITIAL_SIZE;  // 哈希表的初始值：4


    if (size >= LONG_MAX) return LONG_MAX;
    /* 计算新哈希表的大小：第一个大于等于size的2的N 次方的数值 */
    while(1) {
        if (i >= size)
            return i;
        i *= 2;
    }
}
```

总结一下具体逻辑实现：



可以确认当Redis Hash冲突到达某个条件时就会触发dictExpand()函数来扩展HashTable。

DICT_HT_INITIAL_SIZE初始化值为4，通过上述表达式，取当4*2^n >= ht[0].used*2的值作为字典扩展的size大小。即为：ht[1].size 的值等于第一个大于等于ht[0].used*2的2^n的数值。

Redis通过dictCreate()创建词典，在初始化中，table指针为Null，所以两个哈希表ht[0].table和ht[1].table都未真正分配内存空间。只有在dictExpand()字典扩展时才给table分配指向dictEntry的内存。

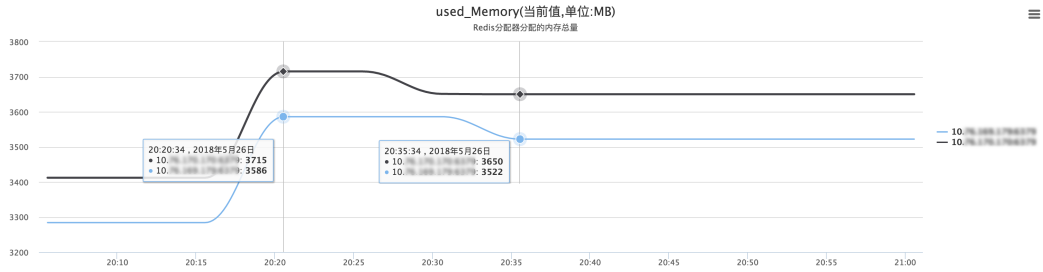
由上可知，当Redis触发Resize后，就会动态分配一块内存，最终由ht[1].table指向，动态分配的内存大小为：realsize*sizeof(dictEntry*)，table指向dictEntry*的一个指针，大小为8bytes（64位OS），即ht[1].table需分配的内存大小为：8*2*2^n （n大于等于2）。

梳理一下哈希表大小和内存申请大小的对应关系：

ht[0].size	触发Resize时，ht[1]需分配的内存
4	64bytes
8	128bytes
16	256bytes
…	…
65536	1024K
…	…
8388608	128M
16777216	256M
33554432	512M
67108864	1024M
…	…

复现验证

我们通过测试环境数据来验证一下，当Redis Rehash过程中，内存真正的占用情况。



上述两幅图中，Redis Key个数突破Redis Resize的临界点，当Key总数稳定且Rehash完成后，Redis内存（Slave）从3586M降至为3522M：3586-3522=64M。即验证上述Redis在Resize至完成的中间状态，会维持一段时间内存消耗，且占用内存的值为上文列表相应的内存空间。

进一步观察一下Redis内部统计信息：

```
/* Redis节点800万左右Key时候的Dict状态信息:只有ht[0]信息。*/
"[Dictionary HT]
Hash table 0 stats (main hash table):
 table size: 8388608
 number of elements: 8003582
 different slots: 5156314
 max chain length: 9
 avg chain length (counted): 1.55
 avg chain length (computed): 1.55
 Chain length distribution:
   0: 3232294 (38.53%)
   1: 3080243 (36.72%)
   2: 1471920 (17.55%)
   3: 466676 (5.56%)
   4: 112320 (1.34%)
   5: 21301 (0.25%)
   6: 3361 (0.04%)
   7: 427 (0.01%)
   8: 63 (0.00%)
   9: 3 (0.00%)
"

/* Redis节点840万左右Key时候的Dict状态信息正在Rehasing中，包含了ht[0]和ht[1]信息。*/
"[Dictionary HT]
[Dictionary HT]
Hash table 0 stats (main hash table):
 table size: 8388608
 number of elements: 8019739
 different slots: 5067892
 max chain length: 9
 avg chain length (counted): 1.58
 avg chain length (computed): 1.58
 Chain length distribution:
   0: 3320716 (39.59%)
   1: 2948053 (35.14%)
   2: 1475756 (17.59%)
   3: 491069 (5.85%)
   4: 123594 (1.47%)
   5: 24650 (0.29%)
   6: 4135 (0.05%)
   7: 553 (0.01%)
   8: 78 (0.00%)
   9: 4 (0.00%)
Hash table 1 stats (rehashing target):
 table size: 16777216
 number of elements: 384321
 different slots: 305472
 max chain length: 6
 avg chain length (counted): 1.26
 avg chain length (computed): 1.26
 Chain length distribution:
   0: 16471744 (98.18%)
   1: 238752 (1.42%)
   2: 56041 (0.33%)
   3: 9378 (0.06%)
   4: 1167 (0.01%)
   5: 119 (0.00%)
   6: 15 (0.00%)
"
```

```
/* Redis节点840万左右Key时候的Dict状态信息(Rehash完成后);ht[0].size从8388608扩展到了16777216。*/
"[Dictionary HT]
Hash table 0 stats (main hash table):
 table size: 16777216
 number of elements: 8404060
 different slots: 6609691
 max chain length: 7
 avg chain length (counted): 1.27
 avg chain length (computed): 1.27
 Chain length distribution:
   0: 10167525 (60.60%)
   1: 5091002 (30.34%)
   2: 1275938 (7.61%)
   3: 213024 (1.27%)
   4: 26812 (0.16%)
   5: 2653 (0.02%)
   6: 237 (0.00%)
   7: 25 (0.00%)
"
```

经过Redis Rehash内部机制的深入、Redis状态监控和Redis内部统计信息，我们可以得出结论：

当Redis 节点中的Key总量到达临界点后，Redis就会触发Dict的扩展，进行Rehash。申请扩展后相应的内存空间大小。

如上，Redis在满容驱逐状态下，Redis Rehash是导致Redis Master和Slave大量触发驱逐淘汰的根本原因。

除了导致满容驱逐淘汰，Redis Rehash还会引起其他一些问题：

- 在tablesize级别与现有Keys数量不在同一个区间内，主从切换后，由于Redis全量同步，从库tablesize降为与现有Key匹配值，导致内存倾斜；
- Redis Cluster下的某个分片由于Key数量相对较多提前Resize，导致集群分片内存不均。 等等...

Redis Rehash机制优化

那么针对在Redis满容驱逐状态下，如何避免因Rehash而导致Redis抖动的这种问题。

- 我们在Redis Rehash源码实现的逻辑上，加上了一个判断条件，如果现有的剩余内存不够触发Rehash操作所需申请的内存大小，即不进行Resize操作；
- 通过提前运营进行规避，比如容量预估时将Rehash占用的内存考虑在内，或者通过监控定时扩容。

Redis Rehash机制除了会影响上述内存管理和使用外，也会影响Redis其他内部与之相关联的功能模块。下面我们分享一下由于Rehash机制而踩到的第二个坑。

Redis使用Scan清理Key由于Rehash导致清理数据不彻底

Squirrel平台提供给业务清理Key的API后台逻辑，是通过Scan来实现的。实际线上运行效果并不是每次都能完全清理干净。即通过Scan扫描清理相匹配的Key，较低频率会有遗漏、Key未被全部清理掉的现象。有了前几次的相关经验后，我们直接从原理入手。

Scan原理

为了高效地匹配出数据库中所有符合给定模式的Key，Redis提供了Scan命令。该命令会在每次调用的时候返回符合规则的部分Key以及一个游标值Cursor（初始值使用0），使用每次返回Cursor不断迭代，直

到Cursor的返回值为0代表遍历结束。

Redis官方定义Scan特点如下：

1. 整个遍历从开始到结束期间， 一直存在于Redis数据集内的且符合匹配模式的所有Key都会被返回；
2. 如果发生了rehash，同一个元素可能会被返回多次，遍历过程中新增或者删除的Key可能会被返回，也可能不会。

具体实现

上述提及Redis的Keys是以Dict方式来存储的，正常只要一次遍历Dict中所有Hash桶就可以完整扫描出所有Key。但是在实际使用中，Redis Dict是有状态的，会随着Key的增删不断变化。

接下来根据Dict四种状态来分析一下Scan的不同实现。Dict的四种状态场景：

1. 字典tablesize保持不变，没有扩缩容；
2. 字典Resize，Dict扩大了（完成状态）；
3. 字典Resize，Dict缩小了（完成状态）；
4. 字典正在Rehashing（扩展或收缩）。

(1) 字典tablesize保持不变，在Redis Dict稳定的状态下，直接顺序遍历即可；(2) 字典Resize，Dict扩大了，如果还是按照顺序遍历，就会导致扫描大量重复Key。比如字典tablesize从8变成了16，假设之前访问的是3号桶，那么表扩展后则是继续访问4~15号桶；但是，原先的0~3号桶中的数据在Dict长度变大后被迁移到8~11号桶中，因此，遍历8~11号桶的时候会有大量的重复Key被返回；(3) 字典Resize，Dict缩小了，如果还是按照顺序遍历，就会导致大量的Key被遗漏。比如字典tablesize从8变成了4，假设当前访问的是3号桶，那么下一次则会直接返回遍历结束了；但是之前4~7号桶中的数据在缩容后迁移带可0~3号桶中，因此这部分Key就无法扫描到；(4) 字典正在Rehashing，这种情况如(2)和(3)情况一下，要么大量重复扫描、要么遗漏很多Key。

那么在Dict非稳定状态，即发生Rehash的情况下，Scan要如何保证原有的Key都能遍历出来，又尽少可能重复扫描呢？Redis Scan通过Hash桶掩码的高位顺序访问来解决。

序号	二进制(低位进一)
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100
5	0000 0101
6	0000 0110
7	0000 0111

序号	二进制(高位进一)
0	0000 0000
4	0000 0100
2	0000 0010
6	0000 0110
1	0000 0001
5	0000 0101
3	0000 0011
7	0000 0111

（Dict长度为 8，二进制掩码为 0000 0111）

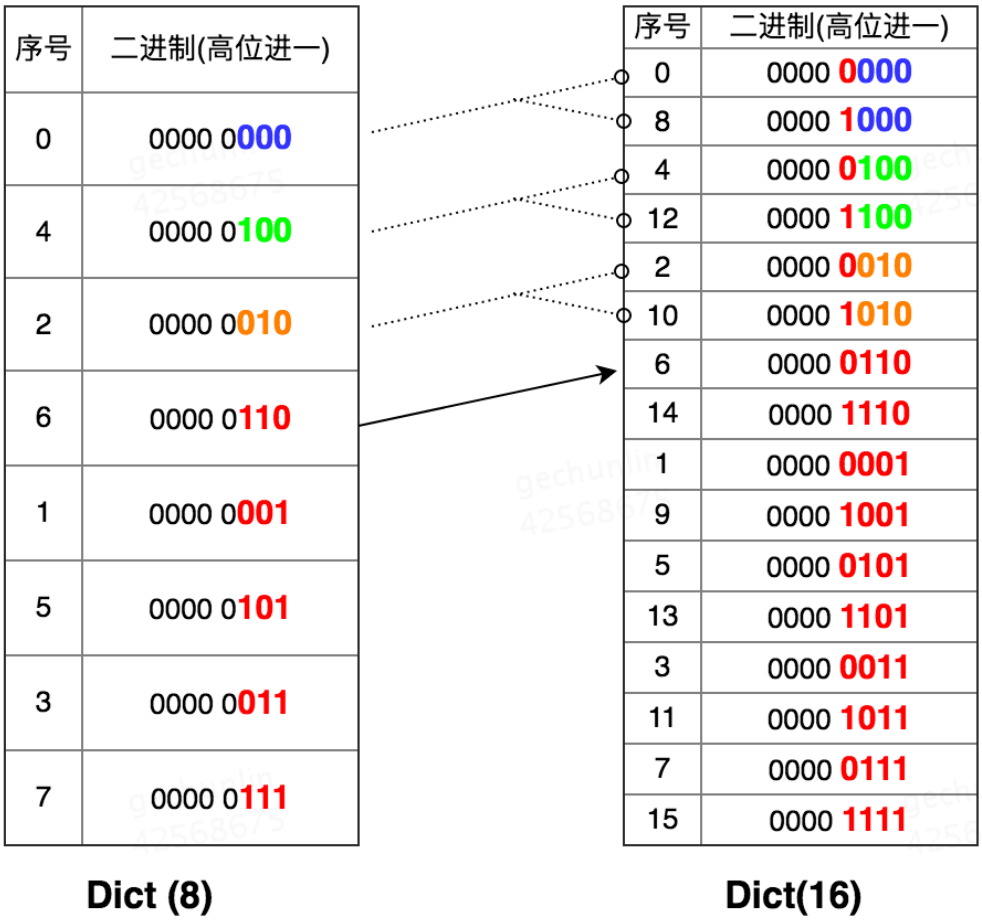
高位顺序访问即按照Dict sizemask（掩码），在有效位（上图中Dict sizemask为3）上从高位开始加一枚举；低位则按照有效位的低位逐步加一访问。

- 低位序：0→1→2→3→4→5→6→7
- 高位序：0→4→2→6→1→5→3→7

Scan采用高位序访问的原因，就是为了实现Redis Dict在Rehash时尽可能少重复扫描返回Key。

举个例子，如果Dict的tablesize从8扩展到了16，梳理一下Scan扫描方式:

1. Dict(8) 从Cursor 0开始扫描；
2. 准备扫描Cursor 6时发生Resize，扩展为之前的2倍，并完成Rehash；
3. 客户端这时开始从Dict(16)的Cursor 6继续迭代；
4. 这时按照 6→14→1→9→5→13→3→11→7→15 Scan完成。



可以看出，高位序Scan在Dict Rehash时即可以避免重复遍历，又能完整返回原始的所有Key。同理，字典缩容时也一样，字典缩容可以看出是反向扩容。

上述是Scan的理论基础，我们看一下Redis源码如何实现。

(1) 非Rehashing 状态下的实现：

```
if (!dictIsRehashing(d)) {      // 判断是否正在rehashing，如果不在则只有ht[0]
        t0 = &(d->ht[0]);  // ht[0]
        m0 = t0->sizemask;  // 掩码

        /* Emit entries at cursor */
        de = t0->table[v & m0];  // 目标桶
        while (de) {
            fn(privdata, de);
            de = de->next;        // 遍历桶中所有节点，并通过回调函数fn()返回
        }
    ...
     /* 反向二进制迭代算法具体实现逻辑——游标实现的精髓 */
    /* Set unmasked bits so incrementing the reversed cursor
     * operates on the masked bits of the smaller table */
    v |= ~m0;

    /* Increment the reverse cursor */
    v = rev(v);
    v++;
    v = rev(v);

    return v;
}
```

源码中Redis将Cursor的计算通过Reverse Binary Iteration（反向二进制迭代算法）来实现上述的高位序扫描方式。

(2) Rehashing 状态下的实现：

```
...
  else {     // 否则说明正在rehashing，就存在两个哈希表ht[0]、ht[1]
        t0 = &d->ht[0];
        t1 = &d->ht[1];  // 指向两个哈希表

        /* Make sure t0 is the smaller and t1 is the bigger table */
        if (t0->size > t1->size) {  确保t0小于t1
            t0 = &d->ht[1];
            t1 = &d->ht[0];
        }

        m0 = t0->sizemask;
        m1 = t1->sizemask;  // 相对应的掩码

        /* Emit entries at cursor */
        /* 迭代(小表)t0桶中的所有节点 */
        de = t0->table[v & m0];
        while (de) {
            fn(privdata, de);
            de = de->next;
        }

        /* Iterate over indices in larger table that are the expansion
         * of the index pointed to by the cursor in the smaller table */
        /* */

        do {
            /* Emit entries at cursor */
            /* 迭代(大表)t1 中所有节点，循环迭代，会把小表没有覆盖的slot全部扫描一遍 */
            de = t1->table[v & m1];
            while (de) {
                fn(privdata, de);
                de = de->next;
            }

            /* Increment bits not covered by the smaller mask */
            v = (((v | m0) + 1) & ~m0) | (v & m0);

            /* Continue while bits covered by mask difference is non-zero */
        } while (v & (m0 ^ m1));
    }

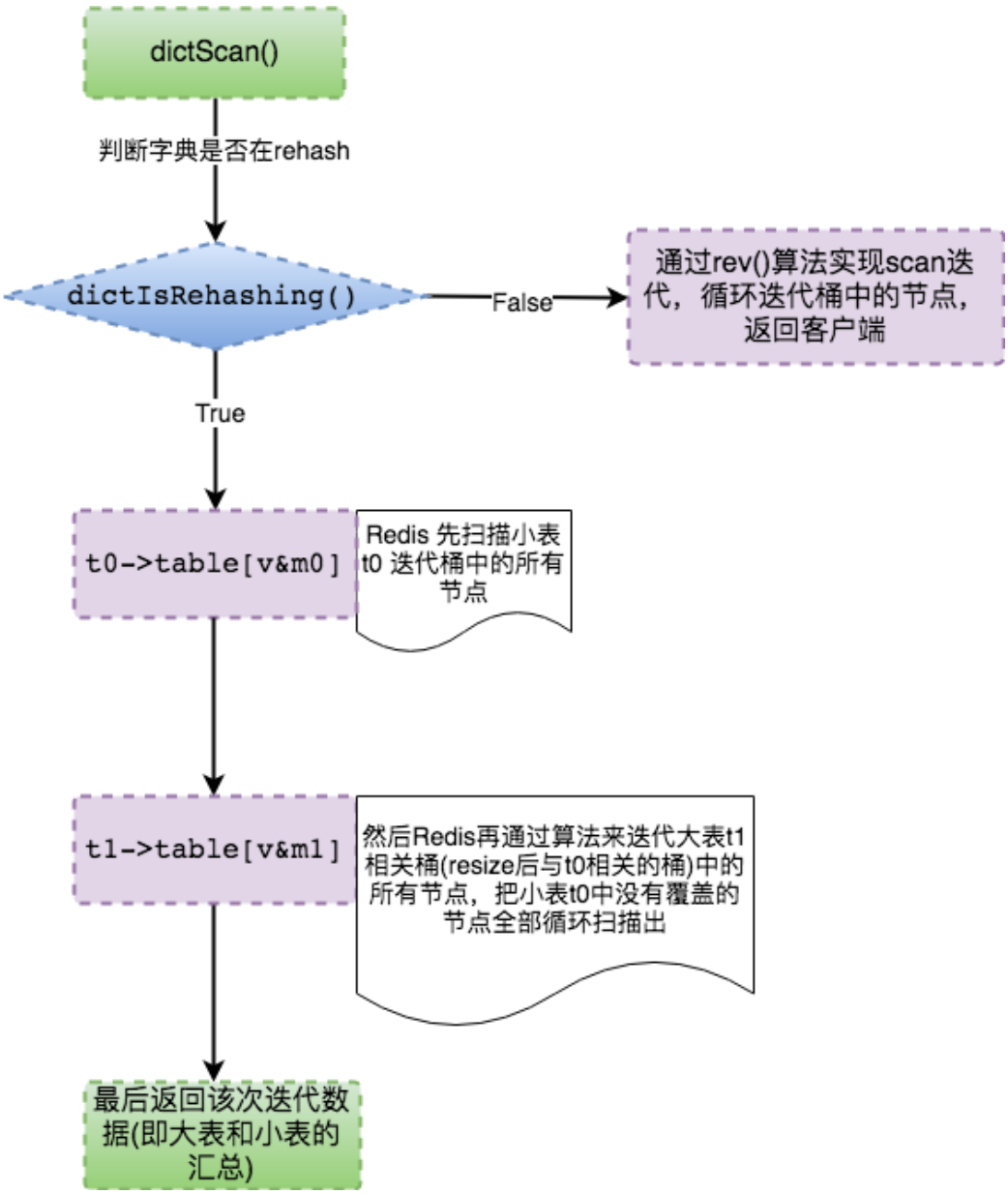
    /* Set unmasked bits so incrementing the reversed cursor
     * operates on the masked bits of the smaller table */
    v |= ~m0;

    /* Increment the reverse cursor */
    v = rev(v);
    v++;
    v = rev(v);

    return v;
```

如上Rehashing时，Redis 通过else分支实现该过程中对两张Hash表进行扫描访问。

梳理一下逻辑流程：



Redis在处理dictScan()时，上面细分的四个场景的实现分成了两个逻辑：

1. 此时不在Rehashing的状态：

这种状态，即Dict是静止的。针对这种状态下的上述三种场景，Redis采用上述的Reverse Binary Iteration（反向二进制迭代算法）： Ⅰ. 首先对游标（Cursor）二进制位翻转； Ⅱ. 再对翻转后的值加1； Ⅲ. 最后再次对 II 的结果进行翻转。

通过穷举高位，依次向低位推进的方式（即高位序访问的实现）来确保所有元素都会被遍历到。

这种算法已经尽可能减少重复元素的返回，但是实际实现和逻辑中还是会有可能存在重复返回，比如在Dict缩容时，高位合并到低位桶中，低位桶中的元素就会被重复取出。

2. 正在Rehashing的状态：

Redis在Rehashing状态的时候，dictScan()实现通过一次性扫描现有的两种字典表，避免中间状态无法维护。

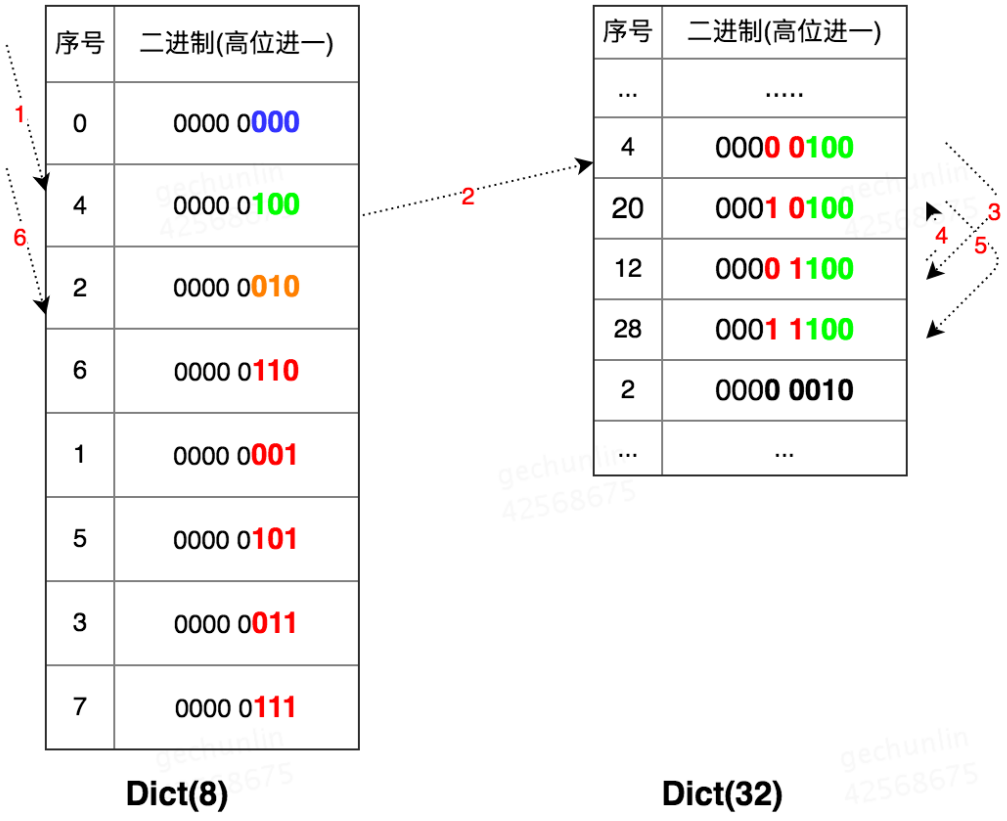
具体实现就是在遍历完小表Cursor位置后，将小表Cursor位置可能Rehash到的大表所有位置全部遍历一遍，然后再返回遍历元素和下一个小表遍历位置。

Root Cause 定位

Rehashing状态时，游标迭代主要逻辑代码实现：

```
/* Increment bits not covered by the smaller mask */
v = (((v | m0) + 1) & ~m0) | (v & m0);   //BUG
```

Ⅰ. v低位加1向高位进位； Ⅱ. 去掉v最前面和最后面的部分，只保留v相较于m0的高位部分； Ⅲ. 保留v的低位，高位不断加1。即低位不变，高位不断加1，实现了小表到大表桶的关联。



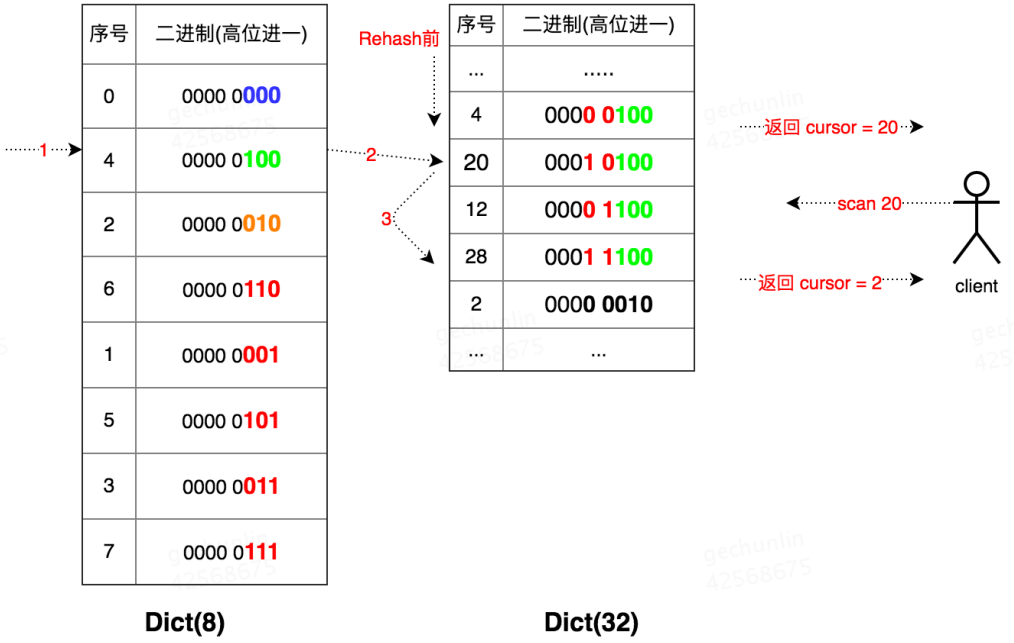
举个例子，如果Dict的tablesize从8扩展到了32，梳理一下Scan扫描方式：

1. Dict(8) 从Cursor 0开始扫描；
2. 准备扫描Cursor 4时发生Resize，扩展为之前的4倍，Rehashing；
3. 客户端先访问Dict(8)中的4号桶；
4. 然后再到Dict(32)上访问:4→12→20→28。

这里可以看到大表的相关桶的顺序并非是按照之前所述的二进制高位序，实际上是按照低位序来遍历大表中高出小表的有效位。

大表t1高位都是向低位加1计算得出的，扫描的顺序却是从低位加1，向高位进位。Redis针对Rehashing时这种逻辑实现在扩容时是可以运行正常的，但是在缩容时高位序和低位序的遍历在大小表上的混用在一定

条件下会出现问题。



再次示例，Dict的tablesize从32缩容到8：

1. Dict(32) 从Cursor 0开始扫描；
2. 准备扫描Cursor 20时发生Resize，缩容至原来的四分之一即tablesize为8，Rehashing；
3. 客户端发起Cursor 20,首先访问Dict(8)中的4号桶；
4. 再到Dict(32)上访问:20→28;
5. 最后返回Cursor = 2。

可以看出大表中的12号桶没有被访问到，即遍历大表时，按照低位序访问会遗漏对某些桶的访问。

上述这种情况发生需要具备一定的条件：

1. 在Dict缩容Rehash时Scan;
2. Dict缩容至至少原Dict tablesize的四分之一，只有在这种情况下，大表相对小表的有效位才会高出二位以上，从而触发跳过某个桶的情况；
3. 如果在Rehash开始前返回的Cursor是在小表能表示的范围内（即不超过7），那么在进行高位有效位的加一操作时，必然都是从0开始计算，每次加一也必然能够访问的全所有的相关桶；如果在Rehash开始前返回的cursor不在小表能表示的范围内（比如20），那么在进行高位有效位加一操作的时候，就有可能跳过，或者重复访问某些桶的情况。

可见，只有满足上述三种情况才会发生Scan遍历过程中漏掉了一些Key的情况。在执行清理Key的时候，如果清理的Key数量很大，导致了Redis内部的Hash表缩容至少原Dict tablesize的四分之一，就可能存在一些Key被漏掉的风险。

Scan源码优化

修复逻辑就是全部都从高位开始增加进行遍历，即大小表都使用高位序访问，修复源码如下：

```
unsigned long dictScan(dict *d,
                       unsigned long v,
                       dictScanFunction *fn,
                       dictScanBucketFunction* bucketfn,
                       void *privdata)
{
    dictht *t0, *t1;
    const dictEntry *de, *next;
    unsigned long m0, m1;

    if (dictSize(d) == 0) return 0;

    if (!dictIsRehashing(d)) {
        t0 = &(d->ht[0]);
        m0 = t0->sizemask;

        /* Emit entries at cursor */
        if (bucketfn) bucketfn(privdata, &t0->table[v & m0]);
        de = t0->table[v & m0];
        while (de) {
            next = de->next;
            fn(privdata, de);
            de = next;
        }

        /* Set unmasked bits so incrementing the reversed cursor
         * operates on the masked bits */
        v |= ~m0;

        /* Increment the reverse cursor */
        v = rev(v);
        v++;
        v = rev(v);

    } else {
        t0 = &d->ht[0];
        t1 = &d->ht[1];

        /* Make sure t0 is the smaller and t1 is the bigger table */
        if (t0->size > t1->size) {
            t0 = &d->ht[1];
            t1 = &d->ht[0];
        }

        m0 = t0->sizemask;
        m1 = t1->sizemask;

        /* Emit entries at cursor */
        if (bucketfn) bucketfn(privdata, &t0->table[v & m0]);
        de = t0->table[v & m0];
        while (de) {
            next = de->next;
            fn(privdata, de);
            de = next;
        }

        /* Iterate over indices in larger table that are the expansion
         * of the index pointed to by the cursor in the smaller table */
        do {
            /* Emit entries at cursor */
            if (bucketfn) bucketfn(privdata, &t1->table[v & m1]);
            de = t1->table[v & m1];
            while (de) {
                next = de->next;
                fn(privdata, de);
                de = next;
            }

            /* Increment the reverse cursor not covered by the smaller mask.*/
            v |= ~m1;
            v = rev(v);
            v++;
            v = rev(v);

            /* Continue while bits covered by mask difference is non-zero */
        } while (v & (m0 ^ m1));
    }

    return v;
}
```

我们团队已经将此PR Push到Redis官方： Fix dictScan(): It can't scan all buckets when dict is shrinking，并已经被官方Merge。

至此，基于Redis Rehash以及Scan实现中涉及Rehash的两个机制已经基本了解和优化完成。

总结

本文主要阐述了因Redis的Rehash机制踩到的两个坑，从现象到原理进行了详细的介绍。这里简单总结一下，第一个案例会造成线上集群进行大量淘汰，而且产生主从不一致的情况，在业务层面也会发生大量超时，影响业务可用性，问题严重，非常值得大家关注；第二个案例会造成数据清理无法完全清理，但是可以再利用Scan清理一遍也能够清理完毕。

注：本文中源码基于Redis 3.2.8。

作者简介

- 春林，2017年加入美团，毕业后一直深耕在运维线，从网络工程师到Oracle DBA再到MySQL DBA多种岗位转变，现在美团主要负责Redis运维开发和优化工作。
- 赵磊，2017年加入美团，毕业后一直从事Redis内核方面的研究和改进，已提交若干优化到社区并被社区采纳。

招聘信息

美团Squirrel技术团队，负责整个美团大规模分布式缓存Squirrel的研发和运维工作，支撑了美团业务快速稳定的发展。同时，Squirrel团队也将持续不断的将内部优化和发现的问题提交到开源社区，回馈社区，希望跟业界一起推动Redis健硕与繁荣。如果有对Redis感兴趣的同学，欢迎参与进来：hao.zhu#dianping.com。

Redis 高负载下的中断优化

作者: 骁雄 春林

背景

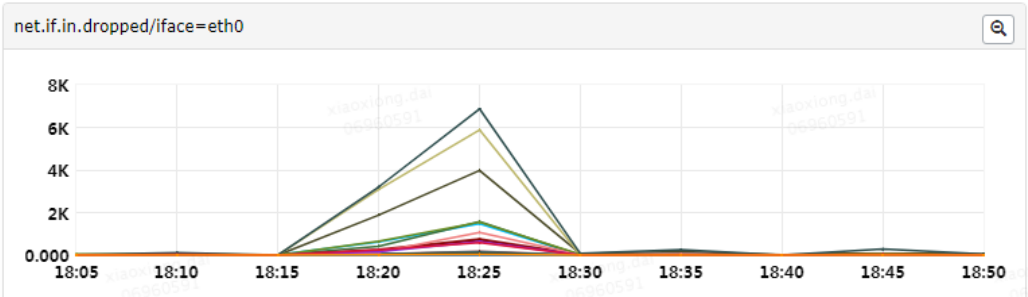
2017年年初以来，随着Redis产品的用户量越来越大，接入服务越来越多，再加上美团点评Memcache和Redis两套缓存融合，Redis服务端的总体请求量从年初最开始日访问量百亿次级别上涨到高峰时段的万亿次级别，给运维和架构团队都带来了极大的挑战。

原本稳定的环境也因为请求量的上涨带来了很多不稳定的因素，其中一直困扰我们的就是网卡丢包问题。起初线上存在部分Redis节点还在使用千兆网卡的老旧服务器，而缓存服务往往需要承载极高的查询量，并要求毫秒级的响应速度，如此一来千兆网卡很快就出现了瓶颈。经过整治，我们将千兆网卡服务器替换为了万兆网卡服务器，本以为可以高枕无忧，但是没想到，在业务高峰时段，机器也竟然出现了丢包问题，而此时网卡带宽使用还远远没有达到瓶颈。

定位网络丢包的原因

从异常指标入手

首先，我们在系统监控的 `net.if.in.dropped` 指标中，看到有大量数据丢包异常，那么第一步就是要了解这个指标代表什么。



这个指标的数据源，是读取 `/proc/net/dev` 中的数据，监控Agent做简单的处理之后上报。以下为 `/proc/net/dev` 的一个示例，可以看到第一行Receive代表in，Transmit代表out，第二行即各个表头字段，再往后每一行代表一个网卡设备具体的值。

```
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
    lo:492120557291 175609331    0    0    0     0          0         0 492120557291 175609331    0    0    0     0       0          0
  eth0:34078756962782 116484580295    0    0  376     0          0    754417 226332888804626 253629046090    0    0    0     0       0          0
  eth1:27368795870156 101510657198    0    0  169     0          0    753948     93451700    753644    0    0    0     0       0          0
 bond0:61447552832938 217995237493    0    0  545     0          0   1508365 226332982256326 253629799734    0    0    0     0       0          0
```

其中各个字段意义如下：

字段	解释
bytes	The total number of bytes of data transmitted or received by the interface.

packets	The total number of packets of data transmitted or received by the interface.
errs	The total number of transmit or receive errors detected by the device driver.
drop	The total number of packets dropped by the device driver.
fifo	The number of FIFO buffer errors.
frame	The number of packet framing errors.
colls	The number of collisions detected on the interface.
compressed	The number of compressed packets transmitted or received by the device driver. (This appears to be unused in the 2.2.15 kernel.)
carrier	The number of carrier losses detected by the device driver.
multicast	The number of multicast frames transmitted or received by the device driver.

通过上述字段解释，我们可以了解丢包发生在网卡设备驱动层面；但是想要了解真正的原因，需要继续深入源码。

`/proc/net/dev` 的数据来源，根据源码文件 `net/core/net-procfs.c` ，可以知道上述指标是通过其中的 `dev_seq_show()` 函数和 `dev_seq_printf_stats()` 函数输出的：

```
static int dev_seq_show(struct seq_file *seq, void *v)
{
    if (v == SEQ_START_TOKEN)
        /* 输出/proc/net/dev表头部分    */
        seq_puts(seq, "Inter-|   Receive                            "
                  "                     |  Transmit\n"
                  " face |bytes    packets errs drop fifo frame "
                  "compressed multicast|bytes    packets errs "
                  "drop fifo colls carrier compressed\n");
    else
        /* 输出/proc/net/dev数据部分    */
        dev_seq_printf_stats(seq, v);
    return 0;
}

static void dev_seq_printf_stats(struct seq_file *seq, struct net_device *dev)
{
    struct rtnl_link_stats64 temp;

    /* 数据源从下面的函数中取得    */
    const struct rtnl_link_stats64 *stats = dev_get_stats(dev, &temp);

    /* /proc/net/dev 各个字段的数据算法    */
    seq_printf(seq, "%6s: %7llu %7llu %4llu %4llu %4llu %5llu %10llu %9llu "
           "%8llu %7llu %4llu %4llu %4llu %5llu %7llu %10llu\n",
           dev->name, stats->rx_bytes, stats->rx_packets,
           stats->rx_errors,
           stats->rx_dropped + stats->rx_missed_errors,
           stats->rx_fifo_errors,
           stats->rx_length_errors + stats->rx_over_errors +
            stats->rx_crc_errors + stats->rx_frame_errors,
           stats->rx_compressed, stats->multicast,
           stats->tx_bytes, stats->tx_packets,
           stats->tx_errors, stats->tx_dropped,
           stats->tx_fifo_errors, stats->collisions,
           stats->tx_carrier_errors +
            stats->tx_aborted_errors +
            stats->tx_window_errors +
            stats->tx_heartbeat_errors,
           stats->tx_compressed);
}
```

`dev_seq_printf_stats()` 函数里，对应drop输出的部分，能看到由两块组成： `stats->rx_dropped+stats ->rx_missed_errors` 。

继续查找 `dev_get_stats` 函数可知， `rx_dropped` 和 `rx_missed_errors` 都是从设备获取的，并且需要设备驱动实现。

```
/**
 *  dev_get_stats   - get network device statistics
 *  @dev: device to get statistics from
 *  @storage: place to store stats
 *
 *  Get network statistics from device. Return @storage.
 *  The device driver may provide its own method by setting
 *  dev->netdev_ops->get_stats64 or dev->netdev_ops->get_stats;
 *  otherwise the internal statistics structure is used.
 */
struct rtnl_link_stats64 *dev_get_stats(struct net_device *dev,
                    struct rtnl_link_stats64 *storage)
{
    const struct net_device_ops *ops = dev->netdev_ops;
    if (ops->ndo_get_stats64) {
        memset(storage, 0, sizeof(*storage));
        ops->ndo_get_stats64(dev, storage);
    } else if (ops->ndo_get_stats) {
        netdev_stats_to_stats64(storage, ops->ndo_get_stats(dev));
    } else {
        netdev_stats_to_stats64(storage, &dev->stats);
    }
    storage->rx_dropped += (unsigned long)atomic_long_read(&dev->rx_dropped);
    storage->tx_dropped += (unsigned long)atomic_long_read(&dev->tx_dropped);
    storage->rx_nohandler += (unsigned long)atomic_long_read(&dev->rx_nohandler);
    return storage;
}
```

结构体 `rtnl_link_stats64` 的定义在 `/usr/include/linux/if_link.h` 中：

```
/* The main device statistics structure */
struct rtnl_link_stats64 {
    __u64   rx_packets;     /* total packets received   */
    __u64   tx_packets;     /* total packets transmitted    */
    __u64   rx_bytes;       /* total bytes received     */
    __u64   tx_bytes;       /* total bytes transmitted  */
    __u64   rx_errors;      /* bad packets received     */
    __u64   tx_errors;      /* packet transmit problems */
    __u64   rx_dropped;     /* no space in linux buffers    */
    __u64   tx_dropped;     /* no space available in linux  */
    __u64   multicast;      /* multicast packets received   */
    __u64   collisions;

    /* detailed rx_errors: */
    __u64   rx_length_errors;
    __u64   rx_over_errors;     /* receiver ring buff overflow  */
    __u64   rx_crc_errors;      /* recved pkt with crc error    */
    __u64   rx_frame_errors;    /* recv'd frame alignment error */
    __u64   rx_fifo_errors;     /* recv'r fifo overrun      */
    __u64   rx_missed_errors;   /* receiver missed packet   */

    /* detailed tx_errors */
    __u64   tx_aborted_errors;
    __u64   tx_carrier_errors;
    __u64   tx_fifo_errors;
    __u64   tx_heartbeat_errors;
    __u64   tx_window_errors;

    /* for cslip etc */
    __u64   rx_compressed;
    __u64   tx_compressed;
};
```

至此，我们知道 `rx_dropped` 是Linux中的缓冲区空间不足导致的丢包，而 `rx_missed_errors` 则在注释中写的比较笼统。有资料指出， `rx_missed_errors` 是fifo队列（即 `rx ring buffer` ）满而丢弃的数量，但这样的话也就和 `rx_fifo_errors` 等同了。后来公司内网络内核研发大牛王伟给了我们点拨：不

同网卡自己实现不一样，比如Intel的igb网卡 `rx_fifo_errors` 在 `missed` 的基础上，还加上了 `RQDPC` 计数，而 `ixgbe` 就没这个统计。RQDPC计数是描述符不够的计数， `missed` 是 `fifo` 满的计数。所以对于 `ixgbe` 来说， `rx_fifo_errors` 和 `rx_missed_errors` 确实是等同的。

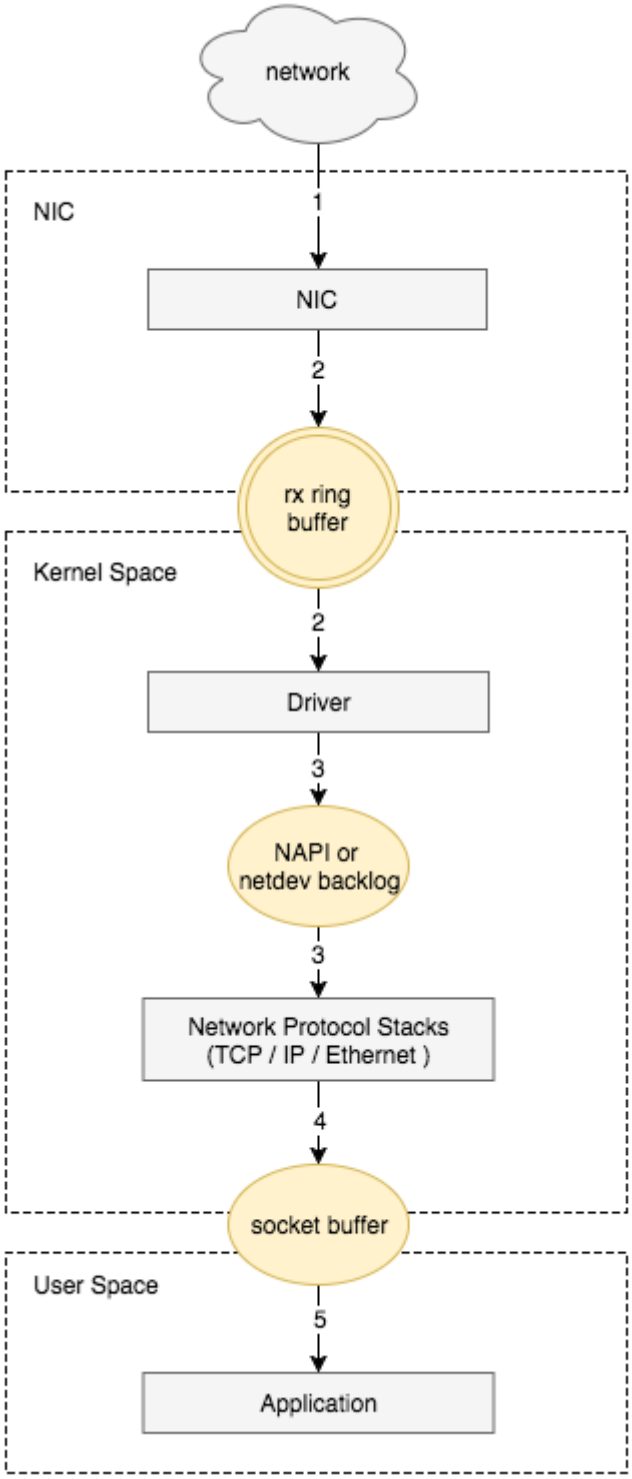
通过命令 `ethtool -S eth0` 可以查看网卡一些统计信息，其中就包含了上文提到的几个重要指标 `rx_dropped` 、 `rx_missed_errors` 、 `rx_fifo_errors` 等。但实际测试后，我发现不同网卡型号给出的指标略有不同，比如 `Intel ixgbe` 就能取到，而 `Broadcom bnx2/tg3` 则只能取到 `rx_discards` （对应 `rx_fifo_errors` ）、 `rx_fw_discards` （对应 `rx_dropped` ）。这表明，各家网卡厂商设备内部对这些丢包的计数器、指标的定义略有不同，但通过驱动向内核提供的统计数据都封装成了 `struct rtnl_link_stats64` 定义的格式。

在对丢包服务器进行检查后，发现 `rx_missed_errors` 为0，丢包全部来自 `rx_dropped` 。说明丢包发生在Linux内核的缓冲区中。接下来，我们要继续探索到底是什么缓冲区引起了丢包问题，这就需要完整地了解服务器接收数据包的过程。

了解接收数据包的流程

接收数据包是一个复杂的过程，涉及很多底层的技术细节，但大致需要以下几个步骤：

1. 网卡收到数据包。
2. 将数据包从网卡硬件缓存转移到服务器内存中。
3. 通知内核处理。
4. 经过TCP/IP协议逐层处理。
5. 应用程序通过 `read()` 从 `socket buffer` 读取数据。

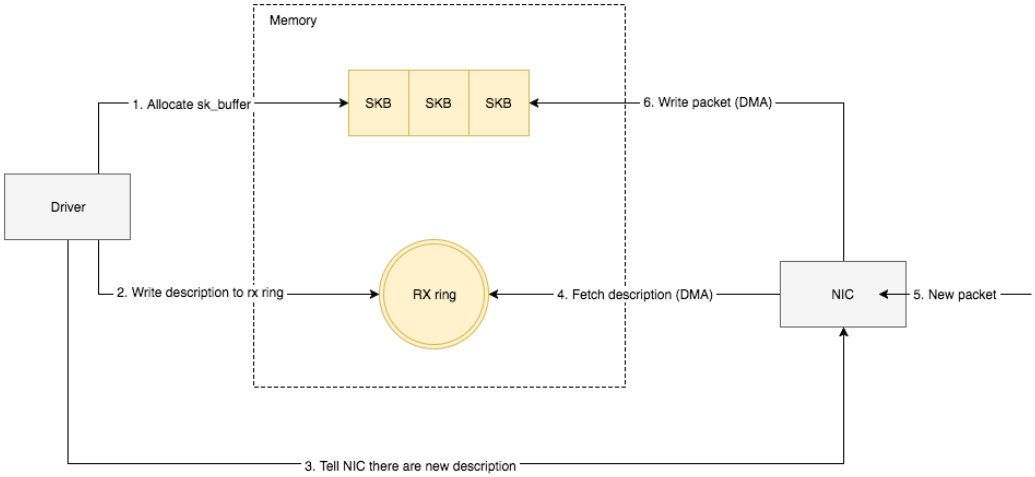


将网卡收到的数据包转移到主机内存（NIC与驱动交互）

NIC在接收到数据包之后，首先需要将数据同步到内核中，这中间的桥梁是 `rx ring buffer` 。它是由NIC和驱动程序共享的一片区域，事实上， `rx ring buffer` 存储的并不是实际的packet数据，而是一个描述符，这个描述符指向了它真正的存储地址，具体流程如下：

1. 驱动在内存中分配一片缓冲区用来接收数据包，叫做 `sk_buffer` ；
2. 将上述缓冲区的地址和大小（即接收描述符），加入到 `rx ring buffer` 。描述符中的缓冲区地址是DMA使用的物理地址；
3. 驱动通知网卡有一个新的描述符；

4. 网卡从 `rx ring buffer` 中取出描述符，从而获知缓冲区的地址和大小；
5. 网卡收到新的数据包；
6. 网卡将新数据包通过DMA直接写到 `sk_buffer` 中。



当驱动处理速度跟不上网卡收包速度时，驱动来不及分配缓冲区，NIC接收到的数据包无法及时写到 `sk_buffer` ，就会产生堆积，当NIC内部缓冲区写满后，就会丢弃部分数据，引起丢包。这部分丢包为 `rx_fifo_errors` ，在 `/proc/net/dev` 中体现为fifo字段增长，在ifconfig中体现为overruns指标增长。

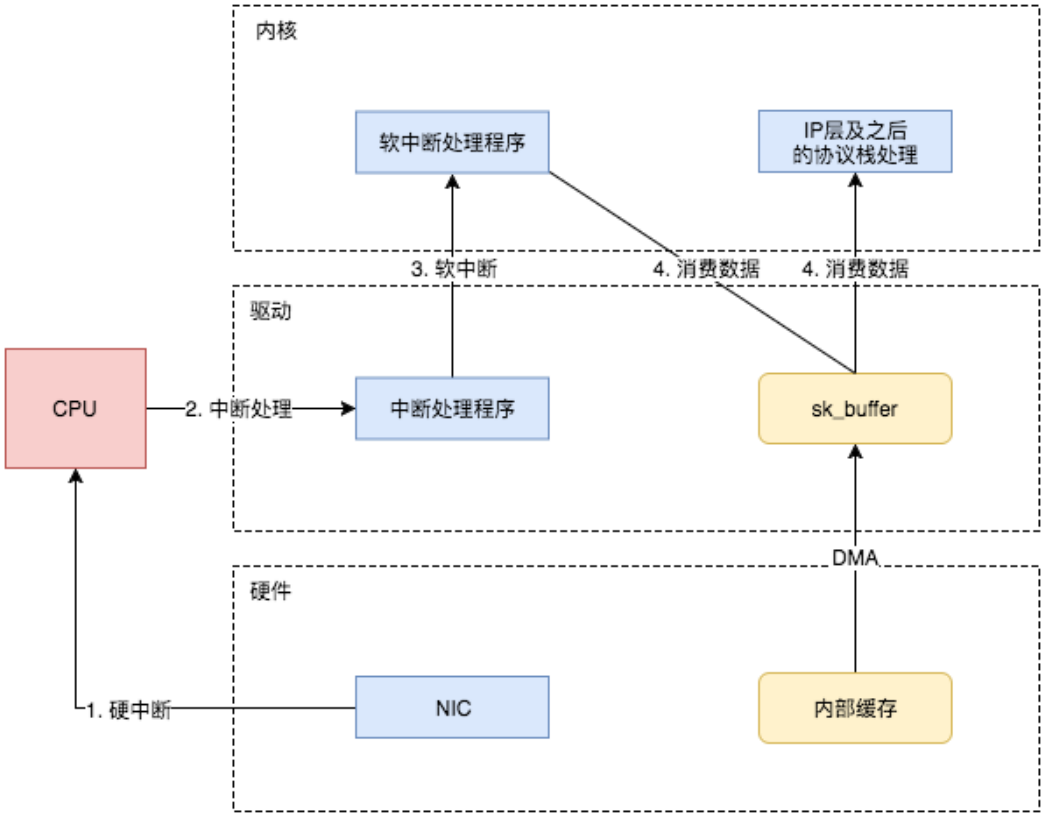
通知系统内核处理（驱动与Linux内核交互）

这个时候，数据包已经被转移到了 `sk_buffer` 中。前文提到，这是驱动程序在内存中分配的一片缓冲区，并且是通过DMA写入的，这种方式不依赖CPU直接将数据写到了内存中，意味着对内核来说，其实并不知道已经有新数据到了内存中。那么如何让内核知道有新数据进来了呢？答案就是中断，通过中断告诉内核有新数据进来了，并需要进行后续处理。

提到中断，就涉及到硬中断和软中断，首先需要简单了解一下它们的区别：

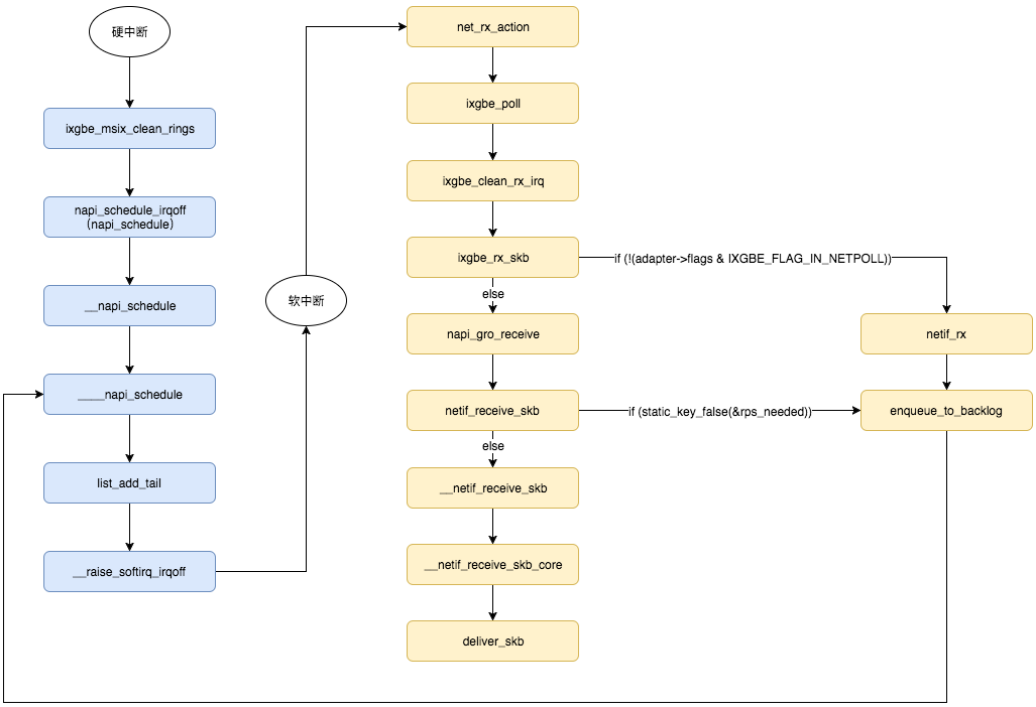
- 硬中断： 由硬件自己生成，具有随机性，硬中断被CPU接收后，触发执行中断处理程序。中断处理程序只会处理关键性的、短时间内可以处理完的工作，剩余耗时较长工作，会放到中断之后，由软中断来完成。硬中断也被称为上半部分。
- 软中断： 由硬中断对应的中断处理程序生成，往往是预先在代码里实现好的，不具有随机性。（除此之外，也有应用程序触发的软中断，与本文讨论的网卡收包无关。）也被称为下半部分。

当NIC把数据包通过DMA复制到内核缓冲区 `sk_buffer` 后，NIC立即发起一个硬件中断。CPU接收后，首先进入上半部分，网卡中断对应的中断处理程序是网卡驱动程序的一部分，之后由它发起软中断，进入下半部分，开始消费 `sk_buffer` 中的数据，交给内核协议栈处理。



通过中断，能够快速及时地响应网卡数据请求，但如果数据量大，那么会产生大量中断请求，CPU大部分时间都忙于处理中断，效率很低。为了解决这个问题，现在的内核及驱动都采用一种叫NAPI（new API）的方式进行数据处理，其原理可以简单理解为 中断+轮询，在数据量大时，一次中断后通过轮询接收一定数量包再返回，避免产生多次中断。

整个中断过程的源码部分比较复杂，并且不同驱动的厂商及版本也会存在一定的区别。 以下调用关系基于Linux-3.10.108及内核自带驱动 `drivers/net/ethernet/intel/ixgbe` ：



注意到， enqueue_to_backlog 函数中，会对CPU的 softnet_data 实例中的接收队列（ input_pkt_queue ）进行判断，如果队列中的数据长度超过 netdev_max_backlog ，那么数据包将直接丢弃，这就产生了丢包。 netdev_max_backlog 是由系统参数 net.core.netdev_max_backlog 指定的，默认大小是 1000。

```
 /*
 * enqueue_to_backlog is called to queue an skb to a per CPU backlog
 * queue (may be a remote CPU queue).
 */
static int enqueue_to_backlog(struct sk_buff *skb, int cpu,
                  unsigned int *qtail)
{
    struct softnet_data *sd;
    unsigned long flags;

    sd = &per_cpu(softnet_data, cpu);

    local_irq_save(flags);

    rps_lock(sd);

    /* 判断接收队列是否满，队列长度为 netdev_max_backlog  */
    if (skb_queue_len(&sd->input_pkt_queue) <= netdev_max_backlog) {


        if (skb_queue_len(&sd->input_pkt_queue)) {
enqueue:
            /*  队列如果不会空，将数据包添加到队列尾  */
            __skb_queue_tail(&sd->input_pkt_queue, skb);
            input_queue_tail_incr_save(sd, qtail);
            rps_unlock(sd);
            local_irq_restore(flags);
            return NET_RX_SUCCESS;
        }

        /* Schedule NAPI for backlog device
         * We can use non atomic operation since we own the queue lock
         */
        /*  队列如果为空，回到 ____napi_schedule加入poll_list轮询部分，并重新发起软中断  */
        if (!__test_and_set_bit(NAPI_STATE_SCHED, &sd->backlog.state)) {
            if (!rps_ipi_queued(sd))
                ____napi_schedule(sd, &sd->backlog);
        }
        goto enqueue;
    }

    /* 队列满则直接丢弃，对应计数器 +1 */
    sd->dropped++;
    rps_unlock(sd);

    local_irq_restore(flags);

    atomic_long_inc(&skb->dev->rx_dropped);
    kfree_skb(skb);
    return NET_RX_DROP;
}
```

内核会为每个 CPU Core 都实例化一个 softnet_data 对象，这个对象中的 input_pkt_queue 用于管理接收的数据包。假如所有的中断都由一个 CPU Core 来处理的话，那么所有数据包只能经由这个CPU的 input_pkt_queue ，如果接收的数据包数量非常大，超过中断处理速度，那么 input_pkt_queue 中的数据包就会堆积，直至超过 netdev_max_backlog ，引起丢包。这部分丢包可以在 cat /proc/net/softnet_stat 的输出结果中进行确认：

```
84b7a331 014feb13 007182fb 00000000 00000000 00000000 00000000 00000000 00000000
6c747c96 0000004c 0000943f 00000000 00000000 00000000 00000000 00000000 00000000
abaf19a3 00000022 000589d6 00000000 00000000 00000000 00000000 00000000 00000000
732a3049 00000004 0000bb09 00000000 00000000 00000000 00000000 00000000 00000000
a287ccfc 00000010 000056ae 00000000 00000000 00000000 00000000 00000000 00000000
6f412392 00000002 00008ecf 00000000 00000000 00000000 00000000 00000000 00000000
a1e6fabc 00000008 0005481f 00000000 00000000 00000000 00000000 00000000 00000000
6b63ed28 00000001 0000a3d1 00000000 00000000 00000000 00000000 00000000 00000000
605184a5 0000000d 00005c13 00000000 00000000 00000000 00000000 00000000 00000000
1d7da0d0 0000000d 00000006 00000000 00000000 00000000 00000000 00000000 00000000
63fc3494 00000010 000457d8 00000000 00000000 00000000 00000000 00000000 00000000
1fc0d80e 0000000f 00000008 00000000 00000000 00000000 00000000 00000000 00000000
5cda7e08 0000000a 00001484 00000000 00000000 00000000 00000000 00000000 00000000
1ccbc9c6 00000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000
60dba7c2 00000016 00041330 00000000 00000000 00000000 00000000 00000000 00000000
1cfc7b3e 0000000a 00000001 00000000 00000000 00000000 00000000 00000000 00000000
137a9fbe 0000000d 0000000b 00000000 00000000 00000000 00000000 00000000 00000000
15ea42f3 00000016 00000004 00000000 00000000 00000000 00000000 00000000 00000000
125841df 0000000e 00000014 00000000 00000000 00000000 00000000 00000000 00000000
17ff5e71 00000017 00000006 00000000 00000000 00000000 00000000 00000000 00000000
119ec12c 00000014 00000006 00000000 00000000 00000000 00000000 00000000 00000000
165b83bd 00000002 00000001 00000000 00000000 00000000 00000000 00000000 00000000
10d598cd 00000025 00000002 00000000 00000000 00000000 00000000 00000000 00000000
154cf4a7 00000025 00000002 00000000 00000000 00000000 00000000 00000000 00000000
```

其中每行代表一个CPU，第一列是中断处理程序接收的帧数，第二列是由于超过 `netdev_max_backlog` 而丢弃的帧数。 第三列则是在 `net_rx_action` 函数中处理数据包超过 `netdev_budge` 指定数量或运行时间超过2个时间片的次数。在检查线上服务器之后，发现第一行CPU。硬中断的中断号及统计数据可以在 `/proc/interrupts` 中看到，对于多队列网卡，当系统启动并加载NIC设备驱动程序模块时，每个RXTX队列会被初始化分配一个唯一的中断向量号，它通知中断处理程序该中断来自哪个NIC队列。在默认情况下，所有队列的硬中断都由CPU 0处理，因此对应的软中断逻辑也会在CPU 0上处理，在服务器TOP 的输出中，也可以观察到 %si 软中断部分，CPU 0的占比比其他core高出一截。

到这里其实有存在一个疑惑，我们线上服务器的内核版本及网卡都支持NAPI，而NAPI的处理逻辑是不会走到 `enqueue_to_backlog` 中的， `enqueue_to_backlog` 主要是非NAPI的处理流程中使用的。对此，我们觉得可能和当前使用的Docker架构有关，事实上，我们通过 `net.if.dropped` 指标获取到的丢包，都发生在Docker虚拟网卡上，而非宿主机物理网卡上，因此很可能是Docker虚拟网桥转发数据包之后，虚拟网卡层面产生的丢包，这里由于涉及虚拟化部分，就不进一步分析了。

驱动及内核处理过程中的几个重要函数：

(1)注册中断号及中断处理程序，根据网卡是否支持 `MSI/MSIX` ，结果为： `MSIX` → `ixgbe_msix_clean_rings` ， `MSI` → `ixgbe_intr` ，都不支持 → `ixgbe_intr` 。

```
/**
 * 文件: ixgbe_main.c
 * ixgbe_request_irq - initialize interrupts
 * @adapter: board private structure
 *
 * Attempts to configure interrupts using the best available
 * capabilities of the hardware and kernel.
 **/
static int ixgbe_request_irq(struct ixgbe_adapter *adapter)
{
    struct net_device *netdev = adapter->netdev;
    int err;

    /* 支持MSIX, 调用 ixgbe_request_msix_irqs 设置中断处理程序*/
    if (adapter->flags & IXGBE_FLAG_MSIX_ENABLED)
        err = ixgbe_request_msix_irqs(adapter);
    /* 支持MSI, 直接设置 ixgbe_intr 为中断处理程序 */
    else if (adapter->flags & IXGBE_FLAG_MSI_ENABLED)
        err = request_irq(adapter->pdev->irq, &ixgbe_intr, 0,
                  netdev->name, adapter);
```

```
    /* 都不支持的情况，直接设置 ixgbe_intr 为中断处理程序 */
    else
        err = request_irq(adapter->pdev->irq, &ixgbe_intr, IRQF_SHARED,
                  netdev->name, adapter);

    if (err)
        e_err(probe, "request_irq failed, Error %d\n", err);

    return err;
}

/**
 * 文件：ixgbe_main.c
 * ixgbe_request_msix_irqs - Initialize MSI-X interrupts
 * @adapter: board private structure
 *
 * ixgbe_request_msix_irqs allocates MSI-X vectors and requests
 * interrupts from the kernel.
 **/
static int (struct ixgbe_adapter *adapter)
{
    …
    for (vector = 0; vector < adapter->num_q_vectors; vector++) {
        struct ixgbe_q_vector *q_vector = adapter->q_vector[vector];
        struct msix_entry *entry = &adapter->msix_entries[vector];

        /* 设置中断处理入口函数为 ixgbe_msix_clean_rings */
        err = request_irq(entry->vector, &ixgbe_msix_clean_rings, 0,
                  q_vector->name, q_vector);
        if (err) {
            e_err(probe, "request_irq failed for MSIX interrupt '%s' "
                  "Error: %d\n", q_vector->name, err);
            goto free_queue_irqs;
        }
    …
    }
}
```

(2)线上的多队列网卡均支持MSIX，中断处理程序入口为 `ixgbe_msix_clean_rings` ，里面调用了函数 `napi_schedule(&q_vector->napi)` 。

```
/**
 * 文件：ixgbe_main.c
 **/
static irqreturn_t ixgbe_msix_clean_rings(int irq, void *data)
{
    struct ixgbe_q_vector *q_vector = data;

    /* EIAM disabled interrupts (on this vector) for us */

    if (q_vector->rx.ring || q_vector->tx.ring)
        napi_schedule(&q_vector->napi);

    return IRQ_HANDLED;
}
```

(3)之后经过一些列调用，直到发起名为 `NET_RX_SOFTIRQ` 的软中断。到这里完成了硬中断部分，进入软中断部分，同时也上升到了内核层面。

```
/**
 * 文件：include/linux/netdevice.h
 *  napi_schedule - schedule NAPI poll
 *  @n: NAPI context
 *
 * Schedule NAPI poll routine to be called if it is not already
 * running.
 */
static inline void napi_schedule(struct napi_struct *n)
{
    if (napi_schedule_prep(n))
    /*  注意下面调用的这个函数名字前是两个下划线 */
        __napi_schedule(n);
}

/**
 * 文件：net/core/dev.c
 * __napi_schedule - schedule for receive
```

```
 * @n: entry to schedule
 *
 * The entry's receive function will be scheduled to run.
 * Consider using __napi_schedule_irqoff() if hard irqs are masked.
 */
void __napi_schedule(struct napi_struct *n)
{
    unsigned long flags;

    /*  local_irq_save用来保存中断状态，并禁止中断 */
    local_irq_save(flags);
    /*  注意下面调用的这个函数名字前是四个下划线，传入的 softnet_data 是当前CPU */
    ____napi_schedule(this_cpu_ptr(&softnet_data), n);
    local_irq_restore(flags);
}

/* Called with irq disabled */
static inline void ____napi_schedule(struct softnet_data *sd,
                      struct napi_struct *napi)
{
    /* 将 napi_struct 加入 softnet_data 的 poll_list */
    list_add_tail(&napi->poll_list, &sd->poll_list);

    /* 发起软中断 NET_RX_SOFTIRQ */
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
}
```

(4) NET_RX_SOFTIRQ 对应的软中断处理程序接口是 net_rx_action() 。

```
/*
 *  文件：net/core/dev.c
 *  Initialize the DEV module. At boot time this walks the device list and
 *  unhooks any devices that fail to initialise (normally hardware not
 *  present) and leaves us with a valid list of present and active devices.
 *
 */

/*
 *       This is called single threaded during boot, so no need
 *       to take the rtnl semaphore.
 */
static int __init net_dev_init(void)
{
    …
    /*  分别注册TX和RX软中断的处理程序 */
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
    …
}
```

(5)net_rx_action功能就是轮询调用poll方法，这里就是ixgbe_poll。一次轮询的数据包数量不能超过内核参数net.core.netdev_budget指定的数量（默认值300），并且轮询时间不能超过2个时间片。这个机制保证了单次软中断处理不会耗时太久影响被中断的程序。

```
/* 文件：net/core/dev.c  */
static void net_rx_action(struct softirq_action *h)
{
    struct softnet_data *sd = &__get_cpu_var(softnet_data);
    unsigned long time_limit = jiffies + 2;
    int budget = netdev_budget;
    void *have;

    local_irq_disable();

    while (!list_empty(&sd->poll_list)) {
        struct napi_struct *n;
        int work, weight;

        /* If softirq window is exhuasted then punt.
         * Allow this to run for 2 jiffies since which will allow
         * an average latency of 1.5/HZ.
         */

        /* 判断处理包数是否超过 netdev_budget 及时间是否超过2个时间片 */
        if (unlikely(budget <= 0 || time_after_eq(jiffies, time_limit)))
            goto softnet_break;
```

```
        local_irq_enable();

        /* Even though interrupts have been re-enabled, this
         * access is safe because interrupts can only add new
         * entries to the tail of this list, and only ->poll()
         * calls can remove this head entry from the list.
         */
        n = list_first_entry(&sd->poll_list, struct napi_struct, poll_list);

        have = netpoll_poll_lock(n);

        weight = n->weight;

        /* This NAPI_STATE_SCHED test is for avoiding a race
         * with netpoll's poll_napi().  Only the entity which
         * obtains the lock and sees NAPI_STATE_SCHED set will
         * actually make the ->poll() call.  Therefore we avoid
         * accidentally calling ->poll() when NAPI is not scheduled.
         */
        work = 0;
        if (test_bit(NAPI_STATE_SCHED, &n->state)) {
            work = n->poll(n, weight);
            trace_napi_poll(n);
        }

        ……
    }

}
```

(6) ixgbe_poll 之后的一系列调用就不一一详述了，有兴趣的同学可以自行研究，软中断部分有几个地方会有类似 if (static_key_false(&rps_needed)) 这样的判断，会进入前文所述有丢包风险的 enqueue_to_backlog 函数。 这里的逻辑为判断是否启用了RPS机制，RPS是早期单队列网卡上将软中断负载均衡到多个 CPU Core 的技术，它对数据流进行hash并分配到对应的 CPU Core 上，发挥多核的性能。不过现在基本都是多队列网卡，不会开启这个机制，因此走不到这里， static_key_false 是针对默认为 false 的 static key 的优化判断方式。这段调用的最后， deliver_skb 会将接收的数据传入一个IP层的数据结构中，至此完成二层的全部处理。

```
/**
 *  netif_receive_skb - process receive buffer from network
 *  @skb: buffer to process
 *
 *  netif_receive_skb() is the main receive data processing function.
 *  It always succeeds. The buffer may be dropped during processing
 *  for congestion control or by the protocol layers.
 *
 *  This function may only be called from softirq context and interrupts
 *  should be enabled.
 *
 *  Return values (usually ignored):
 *  NET_RX_SUCCESS: no congestion
 *  NET_RX_DROP: packet was dropped
 */
int netif_receive_skb(struct sk_buff *skb)
{
    int ret;

    net_timestamp_check(netdev_tstamp_prequeue, skb);

    if (skb_defer_rx_timestamp(skb))
        return NET_RX_SUCCESS;

    rcu_read_lock();

#ifdef CONFIG_RPS
    /* 判断是否启用RPS机制 */
    if (static_key_false(&rps_needed)) {
        struct rps_dev_flow voidflow, *rflow = &voidflow;
        /* 获取对应的CPU Core */
        int cpu = get_rps_cpu(skb->dev, skb, &rflow);

        if (cpu >= 0) {
            ret = enqueue_to_backlog(skb, cpu, &rflow->last_qtail);
            rcu_read_unlock();
```

```
            return ret;
        }
    }
#endif
    ret = __netif_receive_skb(skb);
    rcu_read_unlock();
    return ret;
}
```

TCP/IP协议栈逐层处理，最终交给用户空间读取

数据包进到IP层之后，经过IP层、TCP层处理（校验、解析上层协议，发送给上层协议），放入 `socket buffer` ，在应用程序执行read() 系统调用时，就能从socket buffer中将新数据从内核区拷贝到用户区，完成读取。

这里的 `socket buffer` 大小即TCP接收窗口，TCP由于具备流量控制功能，能动态调整接收窗口大小，因此数据传输阶段不会出现由于 `socket buffer` 接收队列空间不足而丢包的情况（但UDP及TCP握手阶段仍会有）。涉及TCP/IP协议的部分不是此次丢包问题的研究重点，因此这里不再赘述。

网卡队列

查看网卡型号

```
  # lspci -vvv | grep Eth
01:00.0 Ethernet controller: Intel Corporation Ethernet Controller 10-Gigabit X540-AT2 (rev 03)
        Subsystem: Dell Ethernet 10G 4P X540/I350 rNDC
01:00.1 Ethernet controller: Intel Corporation Ethernet Controller 10-Gigabit X540-AT2 (rev 03)
        Subsystem: Dell Ethernet 10G 4P X540/I350 rNDC


# lspci -vvv
07:00.0 Ethernet controller: Intel Corporation I350 Gigabit Network Connection (rev 01)
        Subsystem: Dell Gigabit 4P X540/I350 rNDC
        Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0, Cache Line Size: 128 bytes
        Interrupt: pin D routed to IRQ 19
        Region 0: Memory at 92380000 (32-bit, non-prefetchable) [size=512K]
        Region 3: Memory at 92404000 (32-bit, non-prefetchable) [size=16K]
        Expansion ROM at 92a00000 [disabled] [size=512K]
        Capabilities: [40] Power Management version 3
                Flags: PMEClk- DSI+ D1- D2- AuxCurrent=0mA PME(D0+,D1-,D2-,D3hot+,D3cold+)
                Status: D0 NoSoftRst+ PME-Enable- DSel=0 DScale=1 PME-
        Capabilities: [50] MSI: Enable- Count=1/1 Maskable+ 64bit+
                Address: 0000000000000000  Data: 0000
                Masking: 00000000  Pending: 00000000
        Capabilities: [70] MSI-X: Enable+ Count=10 Masked-
                Vector table: BAR=3 offset=00000000
                PBA: BAR=3 offset=00002000
```

可以看出，网卡的中断机制是MSI-X，即网卡的每个队列都可以分配中断（MSI-X支持2048个中断）。

网卡队列

```
 ...
 #define IXGBE_MAX_MSIX_VECTORS_82599    0x40
...


 u16 ixgbe_get_pcie_msix_count_generic(struct ixgbe_hw *hw)
 {
     u16 msix_count;
     u16 max_msix_count;
     u16 pcie_offset;
```

```
    switch (hw->mac.type) {
    case ixgbe_mac_82598EB:
        pcie_offset = IXGBE_PCIE_MSIX_82598_CAPS;
        max_msix_count = IXGBE_MAX_MSIX_VECTORS_82598;
        break;
    case ixgbe_mac_82599EB:
    case ixgbe_mac_X540:
    case ixgbe_mac_X550:
    case ixgbe_mac_X550EM_x:
    case ixgbe_mac_x550em_a:
        pcie_offset = IXGBE_PCIE_MSIX_82599_CAPS;
        max_msix_count = IXGBE_MAX_MSIX_VECTORS_82599;
        break;
    default:
        return 1;
    }
...
```

根据网卡型号确定驱动中定义的网卡队列，可以看到X540网卡驱动中定义最大支持的IRQ Vector为0x40(数值:64)。

```
static int ixgbe_acquire_msix_vectors(struct ixgbe_adapter *adapter)
{
    struct ixgbe_hw *hw = &adapter->hw;
    int i, vectors, vector_threshold;

    /* We start by asking for one vector per queue pair with XDP queues
     * being stacked with TX queues.
     */
    vectors = max(adapter->num_rx_queues, adapter->num_tx_queues);
    vectors = max(vectors, adapter->num_xdp_queues);

    /* It is easy to be greedy for MSI-X vectors. However, it really
     * doesn't do much good if we have a lot more vectors than CPUs. We'll
     * be somewhat conservative and only ask for (roughly) the same number
     * of vectors as there are CPUs.
     */
    vectors = min_t(int, vectors, num_online_cpus());
```

通过加载网卡驱动，获取网卡型号和网卡硬件的队列数；但是在初始化misx vector的时候，还会结合系统在线CPU的数量，通过Sum = Min(网卡队列，CPU Core) 来激活相应的网卡队列数量，并申请Sum个中断号。

如果CPU数量小于64，会生成CPU数量的队列，也就是每个CPU会产生一个external IRQ。

我们线上的CPU一般是48个逻辑core，就会生成48个中断号，由于我们是两块网卡做了bond，也就会生成96个中断号。

验证与复现网络丢包

通过霸爷的 一篇文章，我们在测试环境做了测试，发现测试环境的中断确实有集中在 `CPU 0` 的情况，下面使用 `systemtap` 诊断测试环境软中断分布的方法:

```
global hard, soft, wq

probe irq_handler.entry {
hard[irq, dev_name]++;
}

probe timer.s(1) {
println("==irq number:dev_name")
foreach( [irq, dev_name] in hard- limit 5) {
printf("%d,%s->%d\n", irq, kernel_string(dev_name), hard[irq, dev_name]);
}

println("==softirq cpu:h:vec:action")
foreach( [c,h,vec,action] in soft- limit 5) {
printf("%d:%x:%x:%s->%d\n", c, h, vec, symdata(action), soft[c,h,vec,action]);
}
```

```
println("==workqueue wq_thread:work_func")
foreach( [wq_thread,work_func] in wq- limit 5) {
printf("%x:%x->%d\n", wq_thread, work_func, wq[wq_thread, work_func]);
}

println("\n")
delete hard
delete soft
delete wq
}

probe softirq.entry {
soft[cpu(), h,vec,action]++;
}

probe workqueue.execute {
wq[wq_thread, work_func]++
}


probe begin {
println("~")
}
```

下面执行 `i.stap` 的结果:

```
==irq number:dev_name
87,eth0-0->1693
90,eth0-3->1263
95,eth1-3->746
92,eth1-0->703
89,eth0-2->654
==softirq cpu:h:vec:action
0:ffffffff81a83098:ffffffff81a83080:0xffffffff81461a00->8928
0:ffffffff81a83088:ffffffff81a83080:0xffffffff81084940->626
0:ffffffff81a830c8:ffffffff81a83080:0xffffffff810ecd70->614
16:ffffffff81a83088:ffffffff81a83080:0xffffffff81084940->225
16:ffffffff81a830c8:ffffffff81a83080:0xffffffff810ecd70->224
==workqueue wq_thread:work_func
ffff88083062aae0:ffffffffa01c53d0->10
ffff88083062aae0:ffffffffa01ca8f0->10
ffff88083420a080:ffffffff81142160->2
ffff8808343fe040:ffffffff8127c9d0->2
ffff880834282ae0:ffffffff8133bd20->1
```

下面是 `action` 对应的符号信息:

```
addr2line -e /usr/lib/debug/lib/modules/2.6.32-431.20.3.el6.mt20161028.x86_64/vmlinux ffffffff81461a00
/usr/src/debug/kernel-2.6.32-431.20.3.el6/linux-2.6.32-431.20.3.el6.mt20161028.x86_64/net/core/dev.c:4013
```

打开这个文件，我们发现它是在执行 `static void net_rx_action(struct softirq_action *h)` 这个函数，而这个函数正是前文提到的， `NET_RX_SOFTIRQ` 对应的软中断处理程序。因此可以确认网卡的软中断在机器上分布非常不均，而且主要集中在 `CPU 0` 上。通过 `/proc/interrupts` 能确认硬中断集中在 `CPU 0` 上，因此软中断也都由 `CPU 0` 处理，如何优化网卡的中断成为了我们关注的重点。

优化策略

CPU亲缘性

前文提到，丢包是因为队列中的数据包超过了 `netdev_max_backlog` 造成了丢弃，因此首先想到是临时调大 `netdev_max_backlog` 能否解决燃眉之急，事实证明，对于轻微丢包调大参数可以缓解丢包，但对于大量丢包则几乎不怎么管用，内核处理速度跟不上收包速度的问题还是客观存在，本质还是因为单核处理中断有瓶颈，即使不丢包，服务响应速度也会变慢。因此如果能同时使用多个 `CPU Core` 来处理中断，就能显著提高中断处理的效率，并且每个CPU都会实例化一个 `softnet_data` 对象，队列数也增加了。

中断亲缘性设置

通过设置中断亲缘性，可以让指定的中断向量号更倾向于发送给指定的 CPU Core 来处理，俗称“绑核”。命令 grep eth /proc/interrupts 的第一列可以获取网卡的中断号，如果是多队列网卡，那么就会有多行输出：

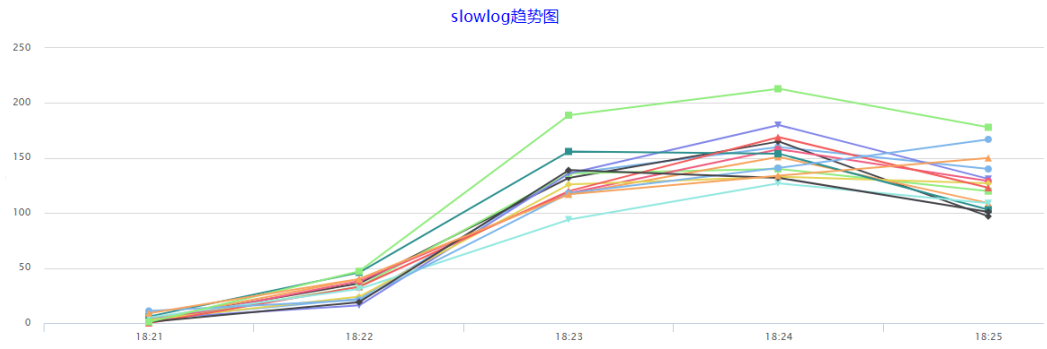
```
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
135:          1          0          0          0          0          0          0          0   IR-PCI-MSI-edge      eth0
136:  527938126          0  315864811          0  331494359          0  166362734          0   IR-PCI-MSI-edge      eth0-TxRx-0
137: 3920620869  636228352  624047238          0  306026265          0  160392484          0   IR-PCI-MSI-edge      eth0-TxRx-1
138: 1308525190          0  578643538          0  580156872          0 1618476592          0   IR-PCI-MSI-edge      eth0-TxRx-2
139: 3158683235          0  356503336  578376147  319069355          0  506989733          0   IR-PCI-MSI-edge      eth0-TxRx-3
140: 4133597158          0  327907259          0  717801407          0  179516622          0   IR-PCI-MSI-edge      eth0-TxRx-4
141: 4221862708          0  325846602          0  338278055  409075354  177630767          0   IR-PCI-MSI-edge      eth0-TxRx-5
142:  327432506          0     138961          0          0          0 2748897340          0   IR-PCI-MSI-edge      eth0-TxRx-6
143: 1657413323          0  328991618          0  338213329          0  204525401  492307994   IR-PCI-MSI-edge      eth0-TxRx-7
144:          1          0          0          0          0          0          0          0   IR-PCI-MSI-edge      eth1
145:  718868368          0  306939492          0  337766750          0  183415711          0   IR-PCI-MSI-edge      eth1-TxRx-0
146: 3570866221          0  681075539          0  339390926          0  185842906          0 3 IR-PCI-MSI-edge      eth1-TxRx-1
147: 2258777018          0  341129658          0  718866281          0 2252739909          0   IR-PCI-MSI-edge      eth1-TxRx-2
148: 3026116890          0  341811416          0 1379935197          0  532457615          0   IR-PCI-MSI-edge      eth1-TxRx-3
149:  428831714          0  337604357          0  312685854          0  226823171          0   IR-PCI-MSI-edge      eth1-TxRx-4
150: 1754126180          0  325384817          0  330443034          0  192058999          0   IR-PCI-MSI-edge      eth1-TxRx-5
151:  249233769          0          0          0      81195          0          0          0   IR-PCI-MSI-edge      eth1-TxRx-6
152:  638608879          0  121078355          0  356335776          0  192316458          0   IR-PCI-MSI-edge      eth1-TxRx-7
```

中断的亲缘性设置可以在 cat /proc/irq/${中断号}/smp_affinity 或 cat /proc/irq/${中断号}/smp_affinity_list 中确认，前者是16进制掩码形式，后者是以 CPU Core 序号形式。例如下图中，将16进制的400转换成2进制后，为 10000000000，“1”在第10位上，表示亲缘性是第10个 CPU Core 。

```
# cat /proc/irq/141/smp_affinity
0000,00000400
# cat /proc/irq/141/smp_affinity_list
10
```

那为什么中断号只设置一个 CPU Core 呢？而不是为每一个中断号设置多个 CPU Core 平行处理。我们经过测试，发现当给中断设置了多个 CPU Core 后，它也仅能由设置的第一个 CPU Core 来处理，其他的 CPU Core 并不会参与中断处理，原因猜想是当CPU可以平行收包时，不同的核收取了同一个queue的数据包，但处理速度不一致，导致提交到IP层后的顺序也不一致，这就会产生乱序的问题，由同一个核来处理可以避免了乱序问题。

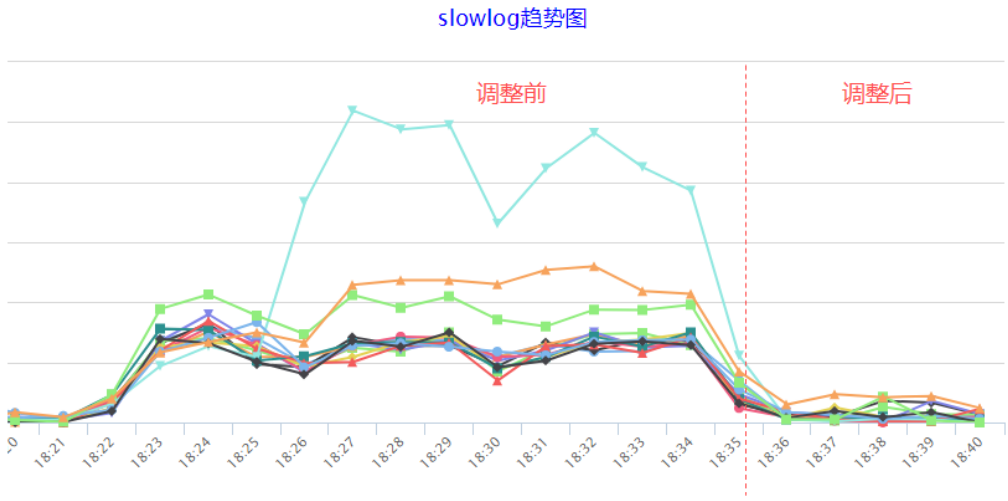
但是，当我们配置了多个Core处理中断后，发现Redis的慢查询数量有明显上升，甚至部分业务也受到了影响，慢查询增多直接导致可用性降低，因此方案仍需进一步优化。



Redis进程亲缘性设置

如果某个 `CPU Core` 正在处理Redis的调用，执行到一半时产生了中断，那么CPU不得不停止当前的工作转而处理中断请求，中断期间Redis也无法转交给其他core继续运行，必须等处理完中断后才能继续运行。Redis本身定位就是高速缓存，线上的平均端到端响应时间小于1ms，如果频繁被中断，那么响应时间必然受到极大影响。容易想到，由最初的 `CPU 0` 单核处理中断，改进到多核处理中断，Redis进程被中断影响的几率增大了，因此我们需要对Redis进程也设置CPU亲缘性，使其与处理中断的Core互相错开，避免受到影响。

使用命令 `taskset` 可以为进程设置CPU亲缘性，操作十分简单，一句 `taskset -cp cpu-list pid` 即可完成绑定。经过一番压测，我们发现使用8个core处理中断时，流量直至打满双万兆网卡也不会出现丢包，因此决定将中断的亲缘性设置为物理机上前8个core，Redis进程的亲缘性设置为剩下的所有core。调整后，确实有明显的效果，慢查询数量大幅优化，但对比初始情况，仍然还是高了一些些，还有没有优化空间呢?



通过观察，我们发现一个有趣的现象，当只有CPU 0处理中断时，Redis进程更倾向于运行在CPU 0，以及CPU 0同一物理CPU下的其他核上。于是有了以下推测：我们设置的中断亲缘性，是直接选取了前8个核心，但这8个core却可能是来自两块物理CPU的，在 `/proc/cpuinfo` 中，通过字段 `processor` 和 `physical id` 能确认这一点，那么响应慢是否和物理CPU有关呢？物理CPU又和NUMA架构关联，每个物理CPU对应一个 `NUMA node` ，那么接下来就要从NUMA角度进行分析。

```
PID    COMMAND                     PSR
4972   /usr/local/bin/redis-server 0
8897   /usr/local/bin/redis-server 14
46790  /usr/local/bin/redis-server 0
61575  /usr/local/bin/redis-server 0
86927  /usr/local/bin/redis-server 10
90249  /usr/local/bin/redis-server 8
94854  /usr/local/bin/redis-server 18
98208  /usr/local/bin/redis-server 10
102625 /usr/local/bin/redis-server 18
116323 /usr/local/bin/redis-server 12
124845 /usr/local/bin/redis-server 16
135070 /usr/local/bin/redis-server 0
143051 /usr/local/bin/redis-server 10
155179 /usr/local/bin/redis-server 12
168789 /usr/local/bin/redis-server 6
182278 /usr/local/bin/redis-server 4
193082 /usr/local/bin/redis-server 18
194616 /usr/local/bin/redis-server 10
```

NUMA

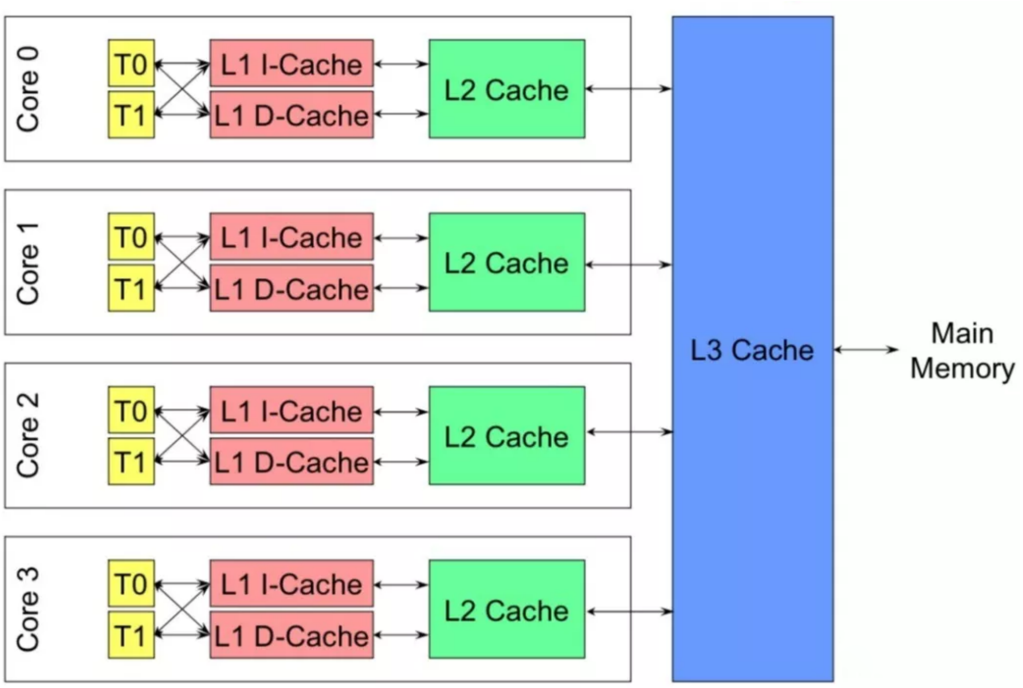
SMP 架构

随着单核CPU的频率在制造工艺上的瓶颈，CPU制造商的发展方向也由纵向变为横向：从CPU频率转为每瓦性能。CPU也就从单核频率时代过渡到多核性能协调。

SMP(对称多处理结构)：即CPU共享所有资源，例如总线、内存、IO等。

SMP 结构：一个物理CPU可以有多个物理Core，每个Core又可以有多个硬件线程。即：每个HT有一个独立的L1 cache，同一个Core下的HT共享L2 cache，同一个物理CPU下的多个core共享L3 cache。

下图(摘自 内核月谈)中，一个x86 CPU有4个物理Core，每个Core有两个HT(Hyper Thread)。



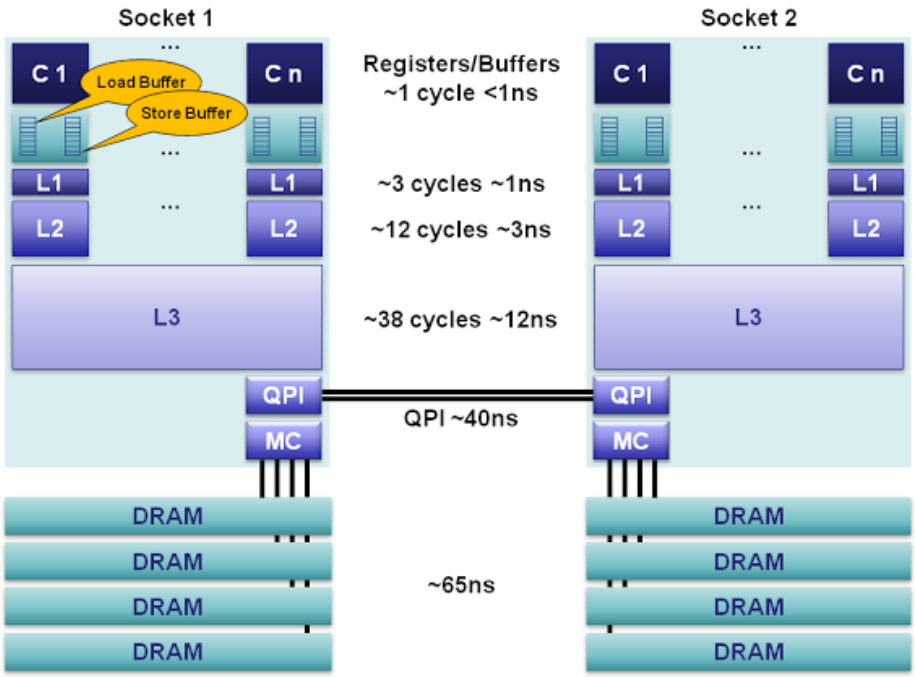
NUMA 架构

在前面的FSB(前端系统总线)结构中，当CPU不断增长的情况下，共享的系统总线就会因为资源竞争(多核争抢总线资源以访问北桥上的内存)而出现扩展和性能问题。

在这样的背景下，基于SMP架构上的优化，设计出了NUMA(Non-Uniform Memory Access)非均匀内存访问。

内存控制器芯片被集成到处理器内部，多个处理器通过QPI链路相连，DRAM也就有了远近之分。(如下图所示：摘自 CPU Cache)

CPU 多层Cache的性能差异是很巨大的，比如：L1的访问时长1ns，L2的时长3ns...跨node的访问会有几十甚至上百倍的性能损耗。

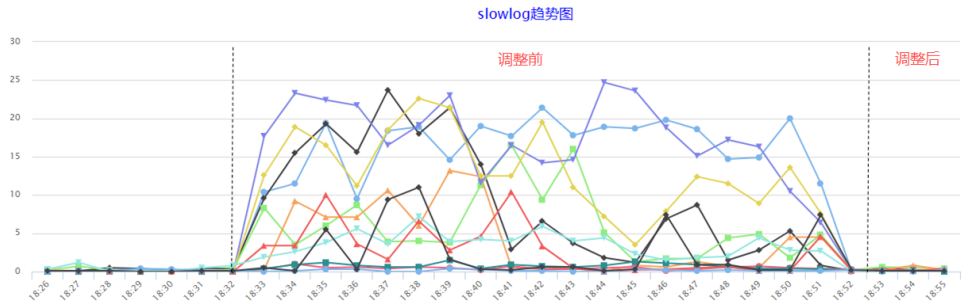


NUMA 架构下的中断优化

这时我们再回归到中断的问题上，当两个NUMA节点处理中断时，CPU实例化的 `softnet_data` 以及驱动分配的 `sk_buffer` 都可能是跨Node的，数据接收后对上层应用Redis来说，跨Node访问的几率也大大提高，并且无法充分利用L2、L3 cache，增加了延时。

同时，由于 `Linux wake affinity` 特性，如果两个进程频繁互动，调度系统会觉得它们很有可能共享同样的数据，把它们放到同一CPU核心或 `NUMA Node` 有助于提高缓存和内存的访问性能，所以当一个进程唤醒另一个的时候，被唤醒的进程可能会被放到相同的 `CPU core` 或者相同的NUMA节点上。此特性对中断唤醒进程时也起作用，在上一节所述的现象中，所有的网络中断都分配给 `CPU 0` 去处理，当中断处理完成时，由于 `wakeup affinity` 特性的作用，所唤醒的用户进程也被安排给 `CPU 0` 或其所在的numa节点上其他core。而当两个 `NUMA node` 处理中断时，这种调度特性有可能导致Redis进程在 `CPU core` 之间频繁迁移，造成性能损失。

综合上述，将中断都分配在同一 `NUMA Node` 中，中断处理函数和应用程序充分利用同NUMA下的L2、L3缓存、以及同Node下的内存，结合调度系统的 `wake affinity` 特性，能够更进一步降低延迟。



参考文档

1. Intel 官方文档🔗
2. Redhat 官方文档🔗

作者简介

- 骁雄，14年加入美团点评，主要从事MySQL、Redis数据库运维，高可用和相关运维平台建设。
- 春林，17年加入美团点评，毕业后一直深耕在运维线，从网络工程师到Oracle DBA再到MySQL DBA 多种岗位转变，现在美大主要职责Redis运维开发和优化工作。

招聘信息

美团点评DBA团队招聘各类DBA人才，Base北京上海均可。我们致力于为公司提供稳定、可靠、高效的在线存储服务，打造业界领先的数据库团队。这里有基于Redis Cluster构建的大规模分布式缓存系统Squirrel，也有基于Tair进行大刀阔斧改进的分布式KV存储系统Cellar，还有数千各类架构的MySQL实例，每天提供万亿级的OLTP访问请求。真正的海量、分布式、高并发环境。欢迎各位朋友推荐或自荐至jinlong.cai#dianping.com。

初探下一代网络隔离与访问控制

作者: 赵彦

概述

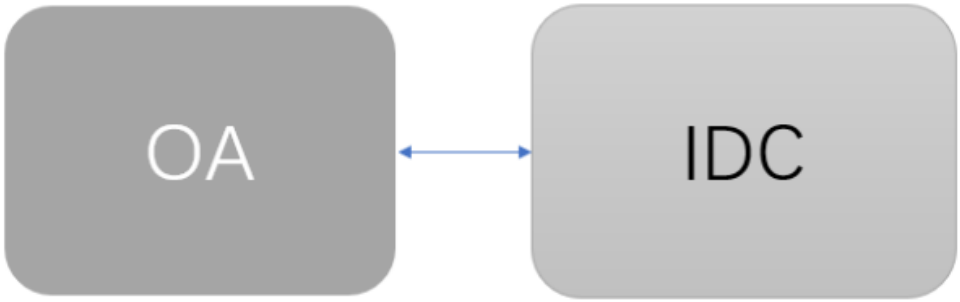
安全域隔离是企业安全里最常见而且最基础的话题之一，目前主要的实现方式是网络隔离（特别重要的也会在物理上实现隔离）。对于很小的公司而言，云上开个VPC就实现了办公网和生产网的基础隔离，但对于有自建的IDC、网络基础设施甚至自己构建云基础设施的大型公司而言，网络隔离是一项基础而复杂的安全建设。基础在这里的意思并非没有技术含量，而是强调其在安全体系里处于一个根基的位置，根基做不好，上层建设都不牢靠。

隔离的目标是为了抑制风险传播的最大范围，使受害范围限定在某个安全域内，类似于船坞的隔离模式，一个仓进水不会沉船。从攻击的角度看，网络隔离可以阻拦攻击者单点得手后的横向扩散，防御者视角更具体的思路可以参考 《互联网企业安全高级指南》 一书有关纵深防御的章节。

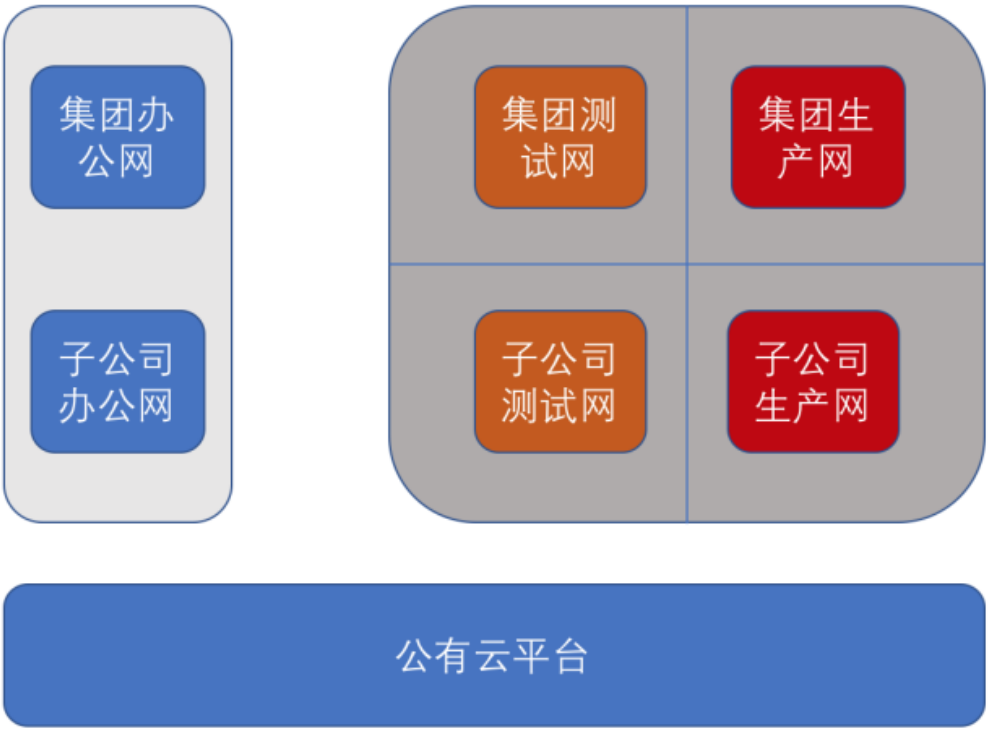
网络隔离通常分为（1）IDC隔离以及（2）办公网隔离，除此之外，还有（3）办公网跟IDC之间的访问控制。受限于安全策略的敏感性，本文不会披露过于详细的策略或实现方案。

IDC隔离

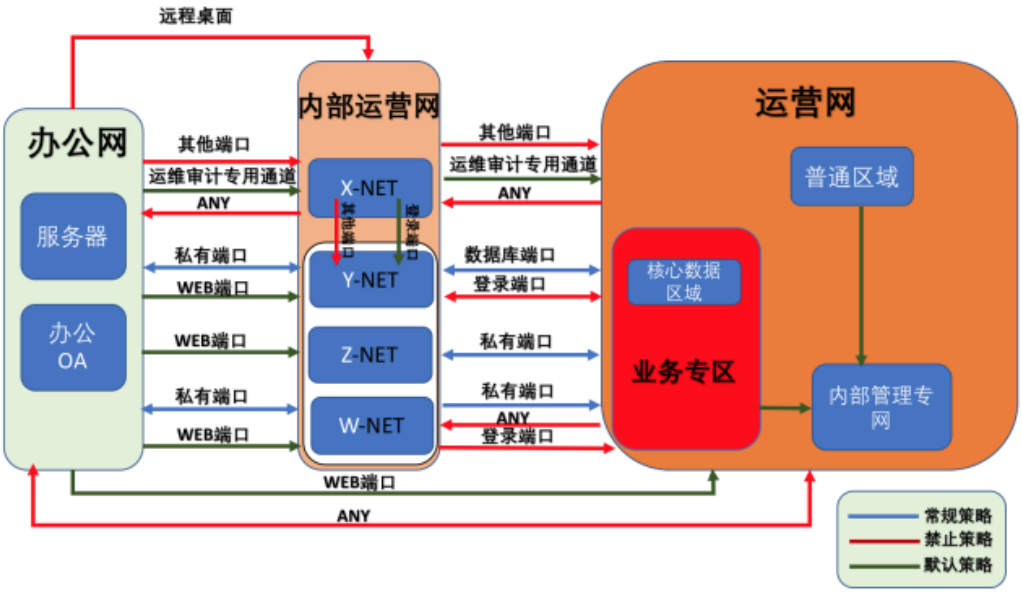
以下先从IDC隔离说起，安全做的没有特点的公司通常是这么分的：办公网内部没有任何隔离，生产网内部也没有任何隔离。



甚至有些市值好几百亿美元的公司也这么分。这样分的结果是对运维比较便利，但从安全来说相当于什么也没做。比较普遍的做法如下图所示（某电商公司的例子）：

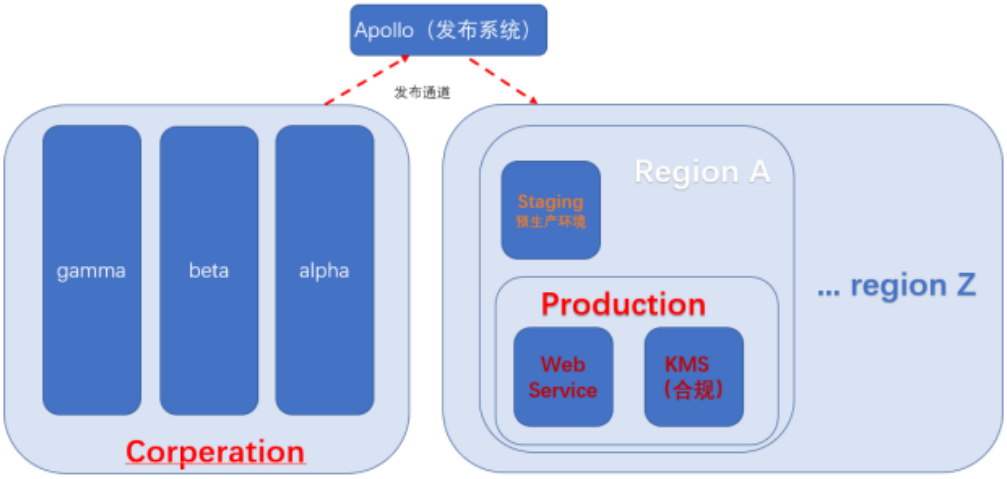


首先会在IDC整体上区分公有云和私有云，其次在私有云内部区分生产和测试环境，然后是办公网跟IDC的隔离。当然这个粒度是比较粗的，实际在每个安全域内部还有更细的划分，安全域之间设置运维和数据通道，数据在安全域之间流转时需要脱敏和审计。再来看另一个大型互联网公司安全域的例子：



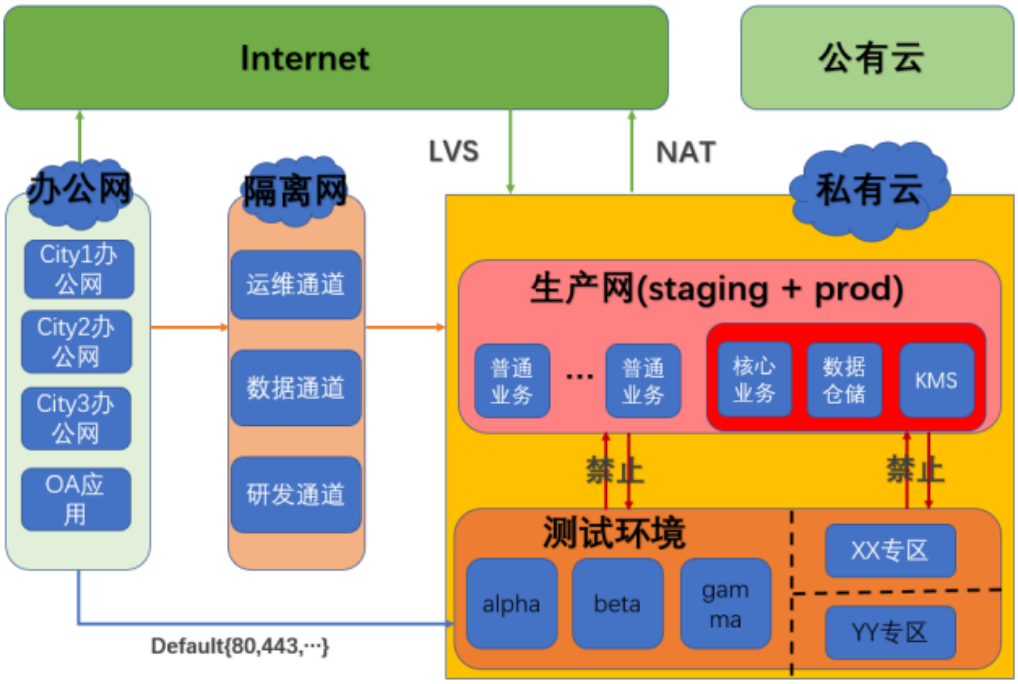
里面更细粒度的定义了OA–IDC之间的通道，即图中标示为内部运营网的部分，例如通常运维的堡垒机就属于内部运营网，思路还是把运维通道，数据通道，测试环境都放在了内部运营网，可以理解为整体上就是OA–测试（内部运营通道）–生产，3大安全域。

除了国内主要的互联网公司，再看一个国际电商和云计算巨头的例子：



办公网包含测试环境，通过发布系统与生产环境隔离，生产环境中除了密钥管理等强制合规性需求外，基本没有做太多隔离，推测是为了网络资源的弹性考虑，staging可以理解为是CI末端的一个预发布环境，在全球范围内会跟云计算一样区分Region（北美、亚太……Region之间几乎不互通），AZ（Availability Zone，可用区，可以理解为是同一个IDC内有用相同的灾备等级）的隔离概念。

基本上上述例子可以代表全球范围内绝大部分互联网和云计算公司的安全域隔离方法论了。于是，我们吸收各家的优点，整理出一个相对通用的安全域划分示例，如下图所示：

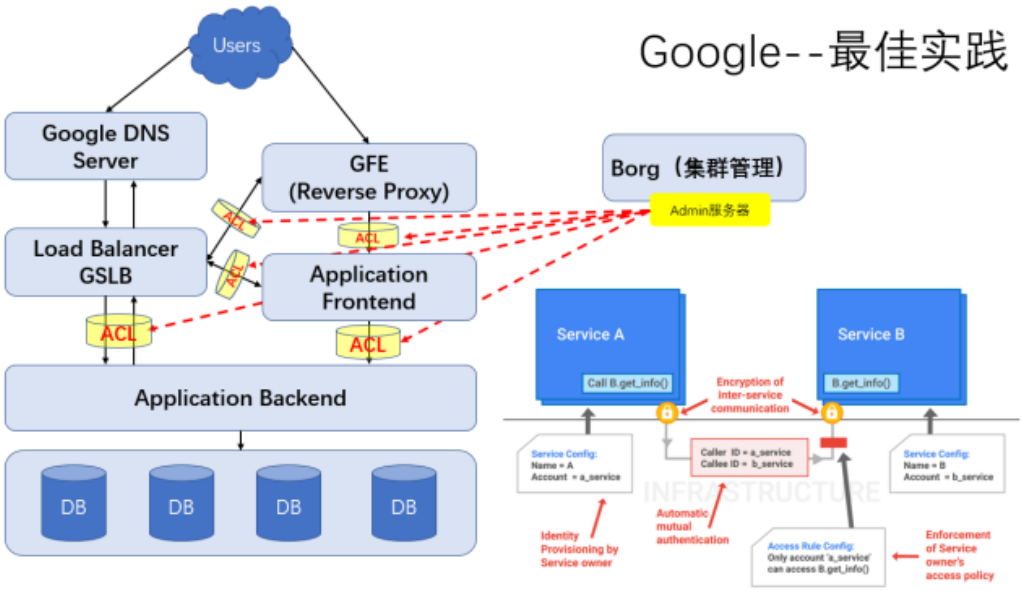


在大多数公司，如果安全工作做得认真到位，没有太激进或者技术引领之类的需求基本上也算OK了。只不过网络隔离这件事天生跟弹性计算是对立的，隔离的越细致，对于快速扩容、服务编排、资源回收都是不利的。在海量IDC运维环境下，会给追求全自动化资源管理的理念引入障碍。

另一方面，虽然明确定义了很多诸如生产测试分离之类的规则，但在实际的日常使用中往往会有很多问题。互联网公司的研发不是传统的瀑布模型，也不一定都有专职的且供应量足够的测试人员，尤其对于扩展中的业务很多流程和规则往往处于模糊地带，测试环境也未必能满足所有的测试需求，例如全链路压测

等，这些问题带来的挑战是安全隔离对业务需求产生阻碍，但是业务又不能中断，所以会有很多变相操作，例如直接去生产环境测试，从而跳过了安全预设的场景和规则，最后使得隔离看起来有点虚幻。

笔者细数了一下网络隔离诞生于上个世纪，是自网络安全开始就有的概念，产生于一个互联网并不发达也没有海量IDC的时代，所以这种模式可能有点局部（并非全部和绝对）过时了。直到发现Google的模式，阐述了一种全新的思路：不使用基于网络的隔离，而是用应用级别的隔离来实现访问控制，如下图所示：



我们从几个层面详细分析这一方案。

首先Google这个规模的IDC管理的理念是必须通过自动化管理实现，人肉是不可能的，当然人肉审批ACL策略也是不可能的。集群全部通过自动化管理的前提是高度的弹性计算能力：所有机器的安装初始化，上线，应用实例部署，到自动擦除数据下线，回收资源都是自动化的，所以过度的隔离也会抑制生产能力。以前这些自动化工作都通过sshd服务来，且需要root权限，Google认为这是一个巨大的安全隐患，所以重新设计了一个admin服务运行于所有的VM、容器实例，这个admin服务本质上是一个RPC服务，支持认证、ACL驱动和审计，并且只需要完成工作的最小权限。相当于原来通过SSH管道执行的命令变成了通过admin服务的一堆RPC调用指令，每个参数都可审计。这是Google这套机制的背景之一。

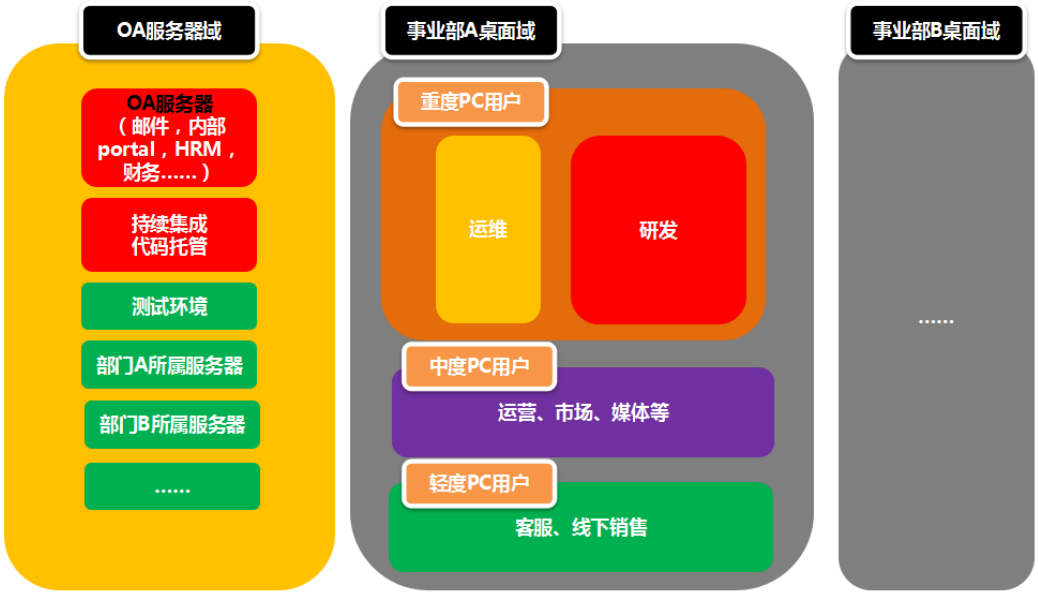
第二点，这套机制能work的前提是：Google的IDC内网只有RPC协议，没有像其他公司一样的mysql，ssh，rpc，http等各种协议，所以只对RPC服务做访问控制就相当于给所有的攻击面做访问控制，但是对于有着各种复杂协议的普适性IDC内网场景来说，这一点前提是不ready，不能说没有用，但显然其他协议仍然有攻击面，仍然可用于内网渗透和横向拓展。这也是Google自研技术栈比较深导致的跟大多数公司基础设施不太一样的地方，所以看官可能也察觉到这不是一个放之四海皆准的方案。

第三点，这套机制的工作原理，可以参考图中右半部分。服务调用通过RPC鉴权：譬如某类型的前台服务A只能访问某个类型的后台服务B，也可以衍生出业务X的前台服务A只能访问业务X的后台服务B，而不能访问业务Y的后台服务B。测试跟生产分离，也只需要将测试和生产定义成不同的业务大类。从攻击者的立场来看，假如你搞定了内网的一台机器，你拿出扫描器去扫其他机器，虽然路由可达，但是因为不具有对应的RPC权限，所以没有对其他应用的访问token，相当于被隔离。

不过可惜的是，前置条件IDC内网收敛为一种RPC协议这一条绝大多数公司都不符合，所以这种形式的可落地改进方案还有待讨论，但是笔者认为这代表了未来的方向，对于超大规模IDC自适应的安全隔离无法通过简单的划分安全域和手工ACL审批来实现。

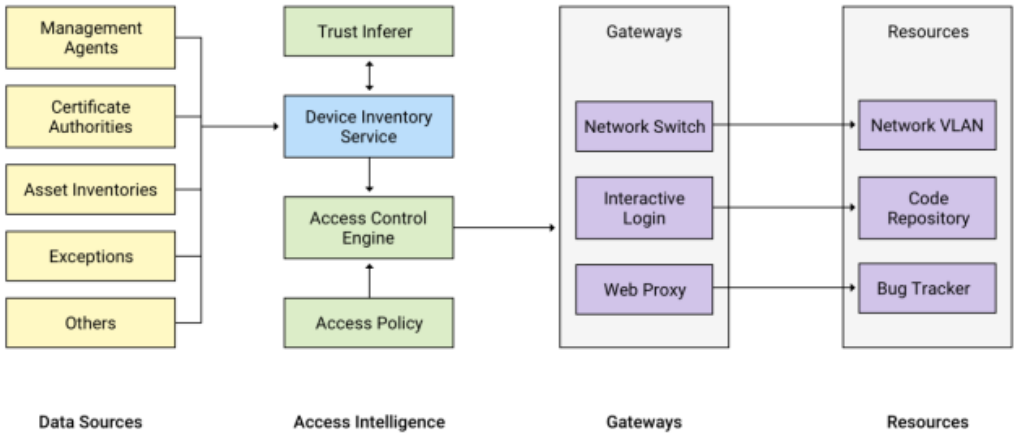
办公网隔离

笔者在《互联网企业安全高级指南》中描述过一种OA网络的划分方法，如下图所示：



首先把IT应用（假如在内网的话）跟桌面用户划开，桌面用户根据职能部门划动态VLAN，这种一种传统的安全域方式。策略收敛比较到位的话也能起到不错的效果。

Google的BeyondCorp（ https://cloud.google.com/beyondcorp/ ）不走寻常路，本质上是办公应用云化和移动化后的产物。



图示的是BeyondCorp这套机制，通过识别当前设备状态，使用动态的访问控制策略决定当前设备能访问哪些OA应用。这套机制跟传统的OA安全域最本质的区别是：

- Bird传统的访问控制策略都是基于IP/MAC的。
- BeyondCorp模型是基于设备/账号的。
- 传统的模型访问控制是静态的，后者则是动态的。

- 传统的模型是ACL，BeyondCorp则有点风控的意味。

对于企业来说，如果移动办公程度非常高，那么应用云端化和SSO这些都是现成的，只需要梳理资产，改造gateway支持风控引擎就可以实现BeyondCorp。而对很多非高度移动化的公司而言，如果传统的安全域划分、网络监控、终端安全管理都做的非常到位的情况下，强制改造成BeyondCorp的成本是非常高的，笔者倾向于认为ROI可能不足以支撑安全团队的绩效。

OA和IDC之间

有几个必要的通道：

1. SSH远程访问通道，通过跳板机，全程审计，权限回收。
2. 数据安全环境：数据开发，BI报表等，一切需要接触数据仓库的开发运营人员的工作集散地，通常这里会有一些类似虚拟化桌面的审计。
3. 数据传输通道：因为各种原因debug、测试需要上传/回传数据，只能在指定的传输通道进行，必须符合脱敏跟审计的要求。
4. 代码发布渠道：通常大型互联网公司都有自己的一套发布系统，甚至还有跟R&D同学本机磁盘映射的方案，所以这个通道的安全都在发布系统上做。
5. 基础设施管理通道：运维通道的特殊版本，可能会集成一些运维管理工具或自动化平台，需要能够触达全部的IDC资源。

总结

对于大部分企业来说传统的安全域划分方法仍然够用，至少几年内没有太大的问题，但对于IDC规模超大的企业来说，弹性计算和安全会成为对立面，如果需要走Google的模式需要高度的集群管理自动化，较高的服务治理水平，高度统一的技术栈和极度收敛的内网协议。

如果OA网络需要BeyondCorp模式，需要办公云化和移动化程度比较高。退一步回到一家公司整体的安全建设上，如果其他地方还有很多短板，在网络隔离上追求极致不是一个正确的策略，相较而言应该将资源（对于较大的安全团队而言也包括自研的资源）都投入到优先级更高的地方，譬如基础设施的攻击缓解能力。

站在一个高P的视角，追求技术的领先是一件本分的事情，而对于安全负责人来说，风险管理和ROI永远是安全工作的核心，也因此在2017年底这个时间点看对于绝大多数公司来说Google模式是没有复制的必要的，不幸的是笔者所在的公司恰好具备了这方面的基础条件，所以安全团队俨然一副在路上的架势。）

作者简介

赵彦，现任美团集团安全部高级总监，负责集团旗下全线业务的信息安全与隐私保护。加盟美团前，曾任华为云安全首席架构师，奇虎360企业安全技术总监、久游网安全总监、绿盟科技安全专家等职务。白帽子时代是Ph4nt0m Security Team的核心成员，互联网安全领域第一代资深从业者。

关于美团安全

美团安全部的大多数核心人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级IDC规模攻防对抗的经验。安全部也不乏CVE“挖掘圣手”，有受邀在Black Hat等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。

目前，美团安全部涉及的技术包括渗透测试、Web防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级IDC规模、数十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，包括网络层、虚拟化/容器层、Server 软件层（内核态/用户态）、语言虚拟机层（JVM/JS V8）、Web应用层、数据访问层等，并能够基于“大数据+机器学习”技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领域不断探索的机会。

招聘信息

美团安全部正在招募Web&二进制攻防、后台&系统开发、机器学习&算法等各路小伙伴。如果你想加入我们，欢迎简历请发至邮箱 zhaoyan17@meituan.com

具体职位信息可参考这里： https://mp.weixin.qq.com/s/ynEq5LqQ2uBcEaHCu7Tsiw

美团安全应急响应中心MTSRC主页： security.meituan.com

互联网公司数据安全保护新探索

作者: 鹏飞

近年来，数据安全形势越发严峻，各种数据安全事件层出不穷。在当前形势下，互联网公司也基本达成了一个共识：虽然无法完全阻止攻击，但底线是敏感数据不能泄漏。也即是说，服务器可以被挂马，但敏感数据不能被拖走。服务器对于互联网公司来说，是可以接受的损失，但敏感数据泄漏，则会对公司产生重大声誉、经济影响。

在互联网公司的数据安全领域，无论是传统理论提出的数据安全生命周期，还是安全厂商提供的解决方案，都面临着落地困难的问题。其核心点在于对海量数据、复杂应用环境下的可操作性不佳。

例如数据安全生命周期提出，首先要对数据进行分类分级，然后才是保护。但互联网公司基本上都是野蛮生长，发展壮大以后才发现数据安全的问题。但存量数据已经形成，日以万计的数据表在增长，这种情况下如何实现数据分类分级？人工梳理显然不现实，梳理的速度赶不上数据增长速度。

再例如安全厂商提供的数据审计解决方案，也都是基于传统关系型数据库的硬件盒子。Hadoop环境下的数据审计方案是什么？面对海量数据，很多厂商也买不起这么多硬件盒子啊。

因此，互联网公司迫切需要一些符合自身特点的手段，来进行数据安全保障。为此，美团信息安全中心进行了一些具体层面的探索。这些探索映射到IT的层面，主要包括应用系统和数据仓库，接下来我们分别阐述。

一、应用系统

应用系统分为两块，一是对抗外部攻击，是多数公司都有的安全意识，但意识不等于能力，这是一个负责任企业的基本功。传统问题包括越权、遍历、SQL注入、安全配置、低版本漏洞等，这一类在OWASP的Top10风险都有提到，在实践中主要考虑SDL、安全运维、红蓝对抗等手段，且以产品化的形式来解决主要问题。这里不做重点介绍。

1.1 扫号及爬虫

新的形势下，还面临扫号、爬虫问题。扫号是指撞库或弱口令：撞库是用已经泄漏的账号密码来试探，成功后轻则窃取用户数据，重则盗取用户资金；弱口令则是简单密码问题。对于这类问题，业界不断的探索新方法，包括设备指纹技术、复杂验证码、人机识别、IP信誉度，试图多管齐下来缓解，但黑产也在不断升级对抗技术，包括一键新机、模拟器、IP代理、人类行为模仿，因此这是个不断的对抗过程。

举个例子，有公司在用户登录时，判断加速等传感器的变化，因为用户在手机屏幕点击时，必然会带来角度、重力的变化。如果用户点击过程中这些传感器没有任何变化，则有使用脚本的嫌疑。再加上一个维度去判断用户近期电量变化，就可以确认这是一台人类在用的手机，还是黑产工作室的手机。黑产在对抗中发现公司用了这一类的策略，则很轻易的进行了化解，一切数据都可以伪造出来，在某宝上可以看到大量的此类技术工具在出售。

爬虫对抗则是另一个新问题，之前有文章说，某些公司的数据访问流量75%以上都是爬虫。爬虫不带来任何业务价值，而且还要为此付出大量资源，同时还面临数据泄漏的问题。

在互联网金融兴起后，爬虫又产生了新的变化，从原来的未授权爬取数据，变成了用户授权爬取数据。举例来说，小张缺钱，在互联网金融公司网站申请小额贷款，而互联网金融公司并不知道小张能不能贷，还款能力如何，因此要求小张提供在购物网站、邮箱或其他应用的账号密码，爬取小张的日常消费数据，作为信用评分参考。小张为了获取贷款，提供了账号密码，则构成了授权爬取。这和以往的未授权爬取产生了很大的变化，互联网金融公司可以进来获取更多敏感信息，不但加重了资源负担，还存在用户密码泄漏的可能。

对爬虫的对抗，也是一个综合课题，不存在一个技术解决所有问题的方案。解决思路上除了之前的设备指纹、IP信誉等手段之外，还包括了各种机器学习的算法模型，以区分出正常行为和异常行为，也可以从关联模型等方向入手。但这也是个对抗过程，黑产也在逐渐摸索试探，从而模拟出人类行为。未来会形成机器与机器的对抗，而决定输赢的，则是成本。

1.2 水印

近年来业界也出现了一些将内部敏感文件，截图外发的事件。有些事件引起了媒体的炒作，对公司造成了舆论影响，这就需要能够对这种外发行为进行溯源。而水印在技术上要解决的抗鲁棒性问题，针对图片的水印技术包括空间滤波、傅立叶变换、几何变形等，简单的说是将信息经过变换，在恶劣条件下还原的技术。

1.3 数据蜜罐

是指制作一个假的数据集合，来捕获访问者，从而发现攻击行为。国外已经有公司做出了对应的产品，其实现可以粗暴地理解为，在一个数据文件上加入了一个“木马”，所有的访问者再打开后，会把对应记录发回服务器。通过这个“木马”，可以追踪到攻击者细节信息。我们也曾做过类似的事情，遗憾的是，这个数据文件放在那里很久，都无人访问。无人访问和我我们对蜜罐的定位有关，现阶段我们更愿意把它作为一个实验性的小玩意，而不是大规模采用，因为“木马”本身，可能带有一定的风险。

1.4 大数据行为审计

大数据的出现，为关联审计提供了更多的可能性，可以通过各种数据关联起来分析异常行为。这方面，传统安全审计厂商做了一些尝试，但从客观的角度来看，还比较基础，无法应对大型互联网公司复杂情况下

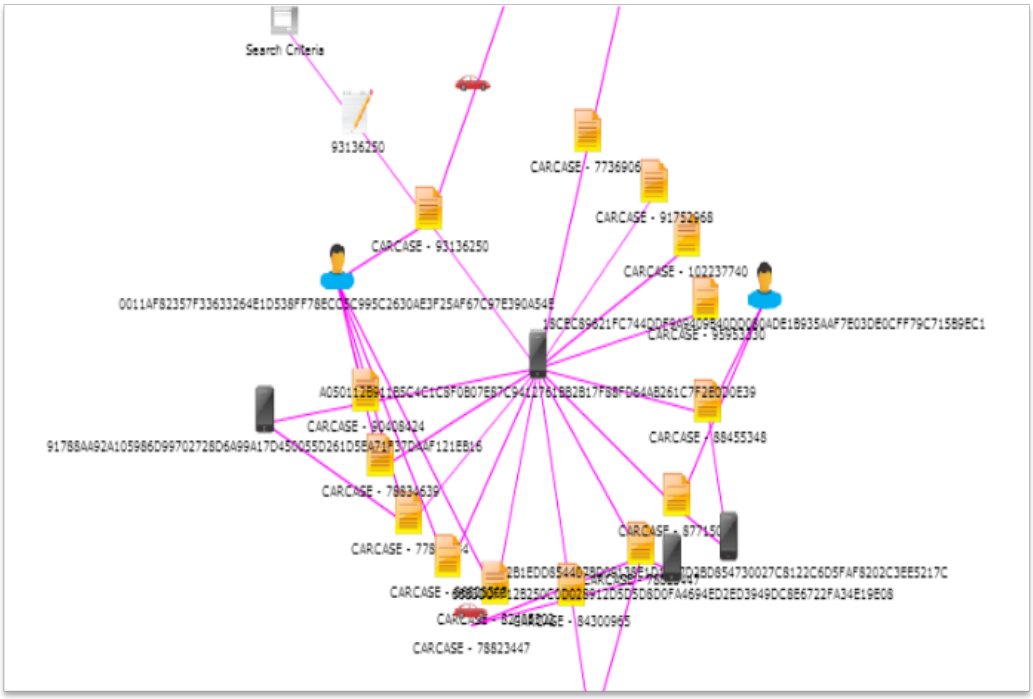
的行为审计，当然这不能苛求传统安全审计厂商，这与生意有关，生意是要追求利润的。这种情况下，互联网公司就要自己做更多的事情。

例如防范内鬼，可以通过多种数据关联分析，通过“与坏人共用过一个设备”规则，来发现内鬼。举一反三，则可以通过信息流、物流、资金流等几个大的方向衍生出更多符合自身数据特点的抓内鬼规则。

除此之外，还可以通过UEBA（用户与实体行为分析）来发现异常，这需要在各个环节去埋点采集数据，后端则需要对应的规则引擎系统、数据平台、算法平台来支撑。

例如常见的聚类算法：某些人与大多数人行为不一致，则这些人可能有异常。具体场景可以是：正常用户行为首先是打开页面，选择产品，然后才是登录、下单。而异常行为可以是：先登录，然后修改密码，最后下单选了一个新开的店，使用了一个大额优惠券。这里每一个数据字段，都可以衍生出各种变量，通过这些变量，最后可以有一个异常判断。

再例如关联模型，一个坏人团伙，通常是有联系的。这些维度可以包括IP、设备、WiFi MAC地址、GPS位置、物流地址、资金流等若干维度，再结合自己的其他数据，可以关联出一个团伙。而团伙中如果有一个人标记为黑，则关系圈则会根据关系强弱进行信誉打分降级。



UEBA的基础是有足够的数据支撑，数据可以是外部的数据供应商。例如腾讯、阿里都提供一些对外数据服务，包括对IP信誉的判断等，使用这些数据，可以起到联防联控的效果。也可以是内部的，互联网公司总会有若干条业务线服务一个客户，这就要看安全人员的数据敏感度了，哪些数据能为自己所用。

1.5 数据脱敏

在应用系统中，总会有很多用户敏感数据。应用系统分为对内和对外，对外的系统脱敏，主要是防止撞号和爬虫。对内的系统脱敏，主要是防止内部人员泄漏信息。

对外系统的脱敏保护，可以分层来对待。默认情况下，对于银行卡号、身份证、手机号、地址等关键信息，强制脱敏，以****替换关键位置，这样即使被撞库或者爬虫，也获取不到相关信息，从而保护用户数

据安全。但总有客户需要看到自己或修改自己的完整信息，这时就需要分层保护，主要是根据常用设备来判断，如果是常用设备，则可以无障碍的点击后显示。如果非常用设备，则推送一个强验证。

在日常业务中，美团还有一个特点。外卖骑手与买家的联系，骑手可能找不到具体位置，需要和买家进行沟通，这时至少包括了地址、手机号两条信息暴露。而对于买家信息的保护，我们也进行了摸索试探。手机号码信息，我们通过一个“小号”的机制来解决，骑手得到的是一个临时中转号码，用这个号码与买家联系，而真实号码则是不可见的。地址信息，我们在系统中使用了图片显示，在订单完成之后，地址信息则不可见。

对内系统的脱敏保护，实践中可以分为几个步骤走。首先是检测内部系统中的敏感信息，这里可以选择从Log中获取，或者从JS前端获取，两个方案各有优劣。从Log中获取，要看公司整体上对日志的规范，不然每个系统一种日志，对接周期长工作量大。从前端JS获取，方案比较轻量化，但要考虑性能对业务的影响。

检测的目的是持续发现敏感信息变化，因为在内部复杂环境中，系统会不断的改造升级，如果缺少持续监控的手段，会变成运动式工程，无法保证持续性。

检测之后要做的事情，则是进行脱敏处理。脱敏过程需要与业务方沟通明确好，哪些字段必须强制完全脱敏，哪些是半脱敏。应用系统权限建设比较规范的情况下，可以考虑基于角色进行脱敏，例如风控案件人员，是一定需要用户的银行卡完整信息的，这时候可以根据角色赋予免疫权限。但客服人员则不需要查看完整信息，则进行强制脱敏。在免疫和脱敏之间，还有一层叫做半脱敏，是指在需要的时候，可以点击查看完整号码，点击动作则会被记录。

就脱敏整体而言，应该有一个全局视图。每天有多少用户敏感信息被访问到，有多少信息脱敏，未脱敏的原因是什么。这样可以整体追踪变化，目标是不断降低敏感信息访问率，当视图出现异常波动，则代表业务产生了变化，需要追踪事件原因。

二、数据仓库

数据仓库是公司数据的核心，这里出了问题则面临巨大风险。而数据仓库的治理，是一个长期渐进的建设过程，其中安全环节只是其中一小部分，更多的则是数据治理层面。本文主要谈及安全环节中的一些工具性建设，包括数据脱敏、隐私保护、大数据行为审计、资产地图、数据扫描器。

2.1 数据脱敏

数据仓库的脱敏是指对敏感数据进行变形，从而起到保护敏感数据的目的，主要用于数据分析人员和开发人员对未知数据进行探索。脱敏在实践过程中有若干种形式，包括对数据的混淆、替换，在不改变数据本身表述的情况下进行数据使用。但数据混淆也好，替换也好，实际上都是有成本的，在大型互联网公司的海量数据情况下，这种数据混淆替换代价非常高昂， 实践中常用的方式，则是较为简单的部分遮盖，例如对手机号的遮盖，139****0011来展示，这种方法规则简单，能起到一定程度上的保护效果。

但有些场景下，简单的遮盖是不能满足业务要求的，这时就需要考虑其他手段，例如针对信用卡号码的的Tokenization，针对范围数据的分段，针对病例的多样性，甚至针对图片的base64遮盖。因此需要根据不同场景提供不同服务，是成本、效率和使用的考量结果，

数据遮盖要考虑原始表和脱敏后的表。原始数据一定要有一份，在这个基础上是另外复制出一张脱敏表还是在原始数据上做视觉脱敏，是两种不同成本的方案。另外复制一张表脱敏，是比较彻底的方式，但等于每张敏感数据表都要复制出来一份，对存储是个成本问题。而视觉脱敏，则是通过规则，动态的对数据展现进行脱敏，可以较低成本的实现脱敏效果，但存在被绕过的可能性。

2.2 隐私保护

隐私保护上学术界也提出了一些方法，包括K匿名、边匿名、差分隐私等方法，其目的是解决数据聚合情况下的隐私保护。例如有的公司，拿出来一部分去除敏感信息后的数据公开，进行算法比赛。这个时候就要考虑不同的数据聚合后，可以关联出某个人的个人标志。目前看到业界在生产上应用的是Google的DLP API，但其使用也较为复杂，针对场景比较单一。隐私保护的方法，关键是要能够进行大规模工程化，在大数据时代的背景下，这些还都是新课题，目前并不存在一个完整的方法来解决隐私保护所有对抗问题。

2.3 大数据资产地图

是指对大数据平台的数据资产进行分析、数据可视化展现的平台。最常见的诉求是，A部门申请B部门的数据，B作为数据的Owner，当然想知道数据给到A以后，他是怎么用的，有没有再传给其他人使用。这时候则需要有一个资产地图，能够跟踪数据资产的流向、使用情况。换个角度，对于安全部门来说，需要知道当前数据平台上有哪些高敏感数据资产，资产的使用情况，以及平台上哪些人拥有什么权限。因此，通过元数据、血缘关系、操作日志，形成了一个可视化的资产地图。形成地图并不够，延伸下来，还需要能够及时预警、回收权限等干预措施。

2.4 数据库扫描器

是指对大数据平台的数据扫描，其意义在于发现大数据平台上的敏感数据，从而进行对应的保护机制。一个大型互联网公司的数据表，每天可能直接产生多达几万张，通过这些表衍生出来更多的表。按照传统数据安全的定义，数据安全第一步是要分类分级，但这一步就很难进行下去。在海量存量表的情况下，该怎样进行分类分级？人工梳理显然是不现实的，梳理的速度还赶不上新增的速度。这时候就需要一些自动化的工具来对数据进行打标定级。因此，数据库扫描器可以通过正则表达式，发现一些基础的高敏感数据，例如手机号、银行卡等这些规整字段。对于非规整字段，则需要通过机器学习+人工标签的方法来确认。

综上，数据安全在业务发展到一定程度后，其重要性越发突出。微观层面的工具建设是一个支撑，在尽量减少对业务的打扰同时提高效率。宏观层面，除了自身体系内的数据安全，合作方、投资后的公司、物流、骑手、商家、外包等各类组织的数据安全情况，也会影响到自身安全，可谓“唇亡齿寒”。而在当前各类组织安全水平参差不齐的情况下，就要求已经发展起来的互联网公司承担更多的责任，帮助合作方提高安全水平，联防共建。

作者简介

- 鹏飞，美团集团安全部数据安全负责人，负责集团旗下全线业务的数据安全与隐私保护。

关于美团安全

美团安全部的大多数核心人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级IDC规模攻防对抗的经验。安全部也不乏CVE“挖掘圣手”，有受邀在Black Hat等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。

目前，美团安全部涉及的技术包括渗透测试、Web防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级IDC规模、数十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，包括网络层、虚拟化/容器层、Server 软件层（内核态/用户态）、语言虚拟机层（JVM/JS V8）、Web应用层、数据访问层等，并能够基于“大数据+机器学习”技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领域不断探索的机会。

招聘信息

美团安全部正在招募Web&二进制攻防、后台&系统开发、机器学习&算法等各路小伙伴。如果你想加入我们，欢迎简历请发至邮箱 zhaoyan17@meituan.com

具体职位信息可参考这里： https://mp.weixin.qq.com/s/ynEq5LqQ2uBcEaHCu7Tsiw

美团安全应急响应中心MTSRC主页： security.meituan.com

数据库智能运维探索与实践

作者: 应钢

> **从自动化到智能化运维过渡时，美团DBA团队进行了哪些思考、探索与实践？本文根据赵应钢在“第九届中国数据库技术大会”上的演讲内容整理而成，部分内容有更新。**

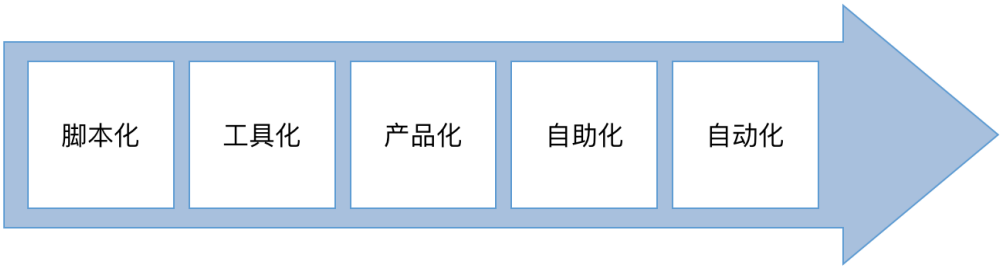
背景

近些年，传统的数据库运维方式已经越来越难于满足业务方对数据库的稳定性、可用性、灵活性的要求。随着数据库规模急速扩大，各种NewSQL系统上线使用，运维逐渐跟不上业务发展，各种矛盾暴露的更加明显。在业务的驱动下，美团点评DBA团队经历了从“人肉”运维到工具化、产品化、自助化、自动化的转型之旅，也开始了智能运维在数据库领域的思考和实践。

本文将介绍美团点评整个数据库平台的演进历史，以及我们当前的情况和面临的一些挑战，最后分享一下我们从自动化到智能化运维过渡时，所进行的思考、探索与实践。

数据库平台的演变

我们数据库平台的演进大概经历了五个大的阶段：



第一个是脚本化阶段，这个阶段，我们人少，集群少，服务流量也比较小，脚本化的模式足以支撑整个服务。

第二个是工具化阶段，我们把一些脚本包装成工具，围绕CMDB管理资产和服务，并完善了监控系统。这时，我们的工具箱也逐渐丰富起来，包括DDL变更工具、SQL Review工具、慢查询采集分析工具和备份闪回工具等等。

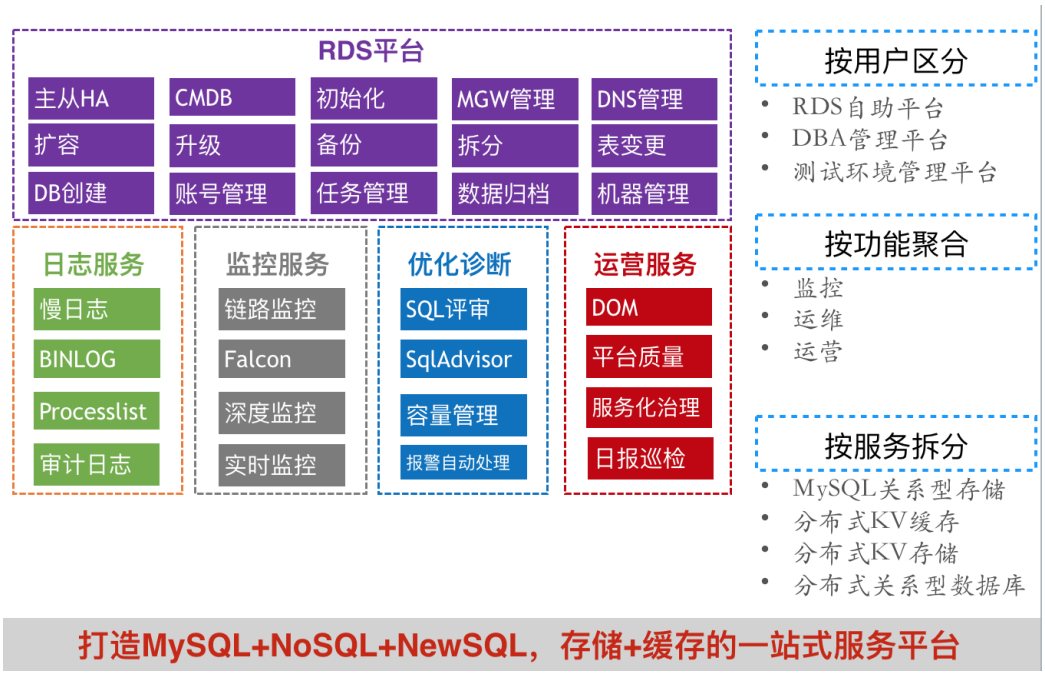
第三个是产品化阶段，工具化阶段可能还是单个的工具，但是在完成一些复杂操作时，就需要把这些工具组装起来形成一个产品。当然，并不是说这个产品一定要做成Web系统的形式，而是工具组装起来形成一套流程之后，就可以保证所有DBA的操作行为，对流程的理解以及对线上的影响都是一致的。我们会在易用性和安全性层面不断进行打磨。而工具产品化的主要受益者是DBA，其定位是提升运维服务的效率，减少事故的发生，并方便进行快速统一的迭代。

第四个是打造私有云平台阶段，随着美团点评业务的高速发展，仅靠十几、二十个DBA越来越难以满足业务发展的需要。所以我们就把某些日常操作开放授权，让开发人员自助去做，将DBA从繁琐的操作中解放出来。当时整个平台每天执行300多次改表操作；自助查询超过1万次；自助申请账号、授权并调整监控；自助定义敏感数据并授权给业务方管理员自助审批和管理；自定义业务的高峰和低峰时间段等等；自助下载、查询日志等等。

第五个是自动化阶段，对这个阶段的理解，其实是“仁者见仁，智者见智”。大多数人理解的自动化，只是通过Web平台来执行某些操作，但我们认为这只是半自动化，所谓的自动化应该是完全不需要人参与。目前，我们很多操作都还处于半自动化阶段，下一个阶段我们需要从半自动过渡到全自动。以MySQL系统为例，从运维角度看包括主从的高可用、服务过载的自我保护、容量自动诊断与评估以及集群的自动扩缩容等等。

现状和面临的挑战

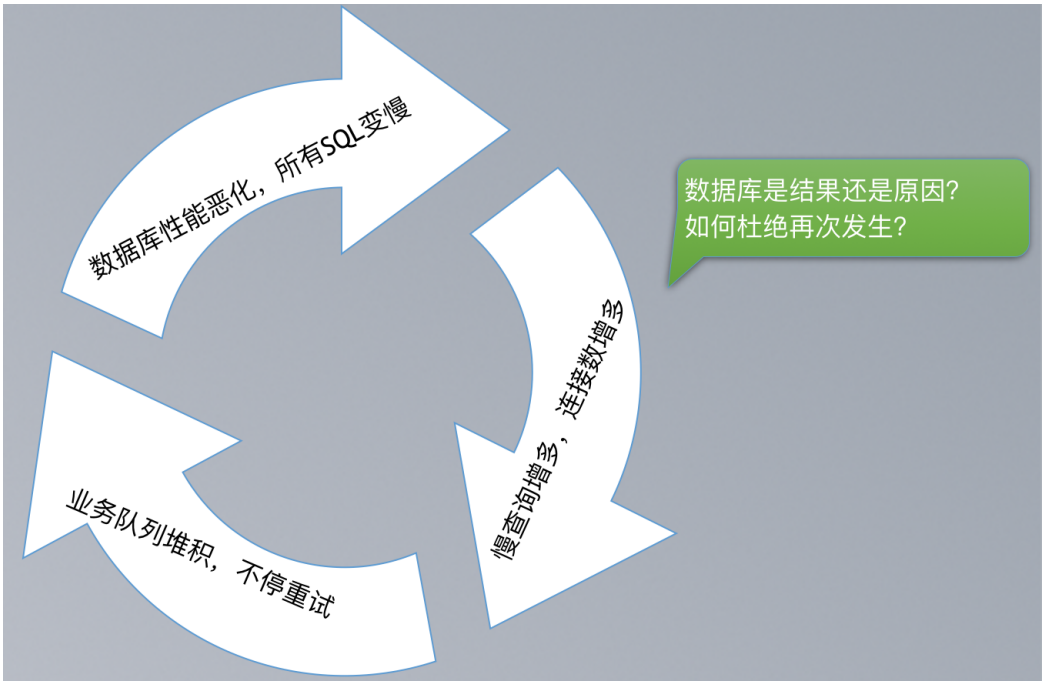
下图是我们平台的现状，以关系数据库RDS平台为例，其中集成了很多管理的功能，例如主从的高可用、MGW的管理、DNS的变更、备份系统、升级流程、流量分配和切换系统、账号管理、数据归档、服务与资产的流转系统等等。



而且我们按照逻辑对平台设计进行了划分，例如以用户维度划分的RDS自助平台，DBA管理平台和测试环境管理平台；以功能维度划分的运维、运营和监控；以存储类型为维度划分的关系型数据库MySQL、分布式KV缓存、分布式KV存储，以及正在建设中的NewSQL数据库平台等等。未来，我们希望打造成“MySQL+NoSQL+NewSQL，存储+缓存的一站式服务平台”。

挑战一：RootCause定位难

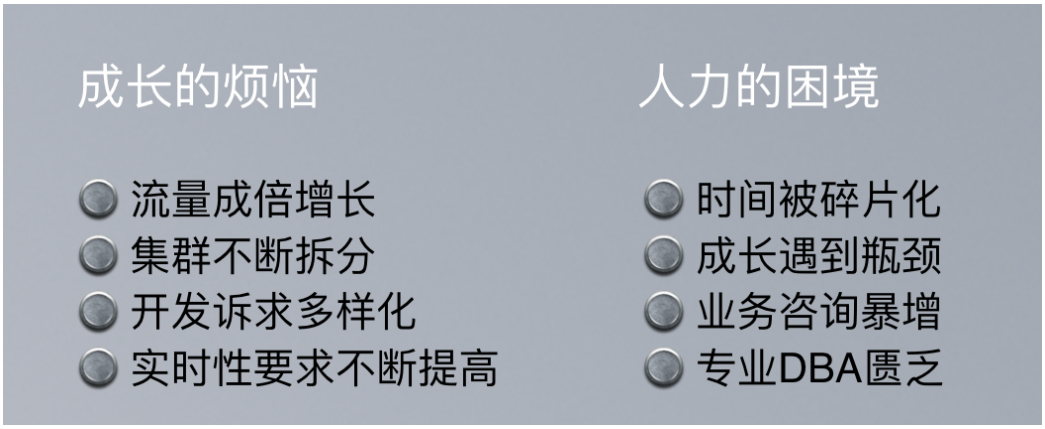
即便我们打造了一个很强大的平台，但还是发现有很多问题难以搞定。第一个就是故障定位，如果是简单的故障，我们有类似天网、雷达这样的系统去发现和定位。但是如果故障发生在数据库内部，那就需要专业的数据库知识，去定位和查明到底是什么原因导致了故障。



通常来讲，故障的轨迹是一个链，但也可能是一个“多米诺骨牌”的连环。可能因为一些原因导致SQL执行变慢，引起连接数的增长，进而导致业务超时，而业务超时又会引发业务不断重试，结果会产生更多的问题。当我们收到一个报警时，可能已经过了30秒甚至更长时间，DBA再去查看时，已经错过了最佳的事故处理时机。所以，我们要在故障发生之后，制定一些应对策略，例如快速切换主库、自动屏蔽下线问题从库等等。除此之外，还有一个比较难的问题，就是如何避免相似的故障再次出现。

挑战二：人力和发展困境

第二个挑战是人力和发展的困境，当服务流量成倍增长时，其成本并不是以相同的速度对应增长的。当业务逻辑越来越复杂时，每增加一块钱的营收，其后面对应的数据库QPS可能是2倍甚至5倍，业务逻辑越复杂，服务支撑的难度越大。另外，传统的关系型数据库在容量、延时、响应时间以及数据量等方面很容易达到瓶颈，这就需要我们不断拆分集群，同时开发诉求也多种多样，当我们尝试使用平台化的思想去解决问题时，还要充分思考如何满足研发人员多样化的需求。



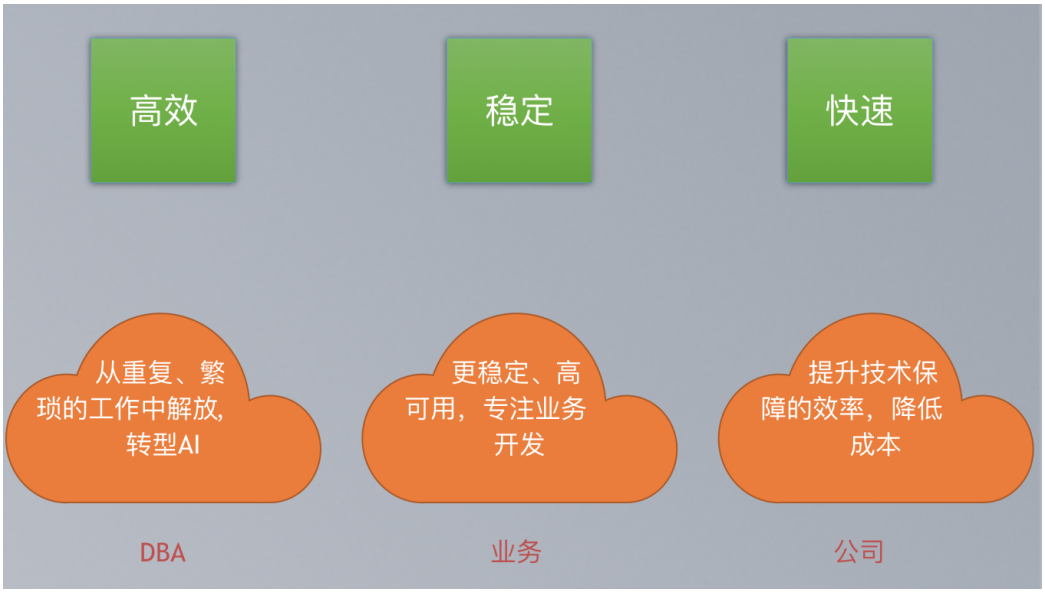
人力困境这一问题，从DBA的角度来说，时间被严重的碎片化，自身的成长就会遇到瓶颈，比如经常会做一些枯燥的重复操作；另外，业务咨询量暴增，尽管我们已经在尝试平台化的方法，但是还是跟不上业务发展的速度。还有一个就是专业的DBA越来越匮乏，越来越贵，关键是根本招聘不到人手。

在这种背景下，我们必须去思考：如何突破困局？如何朝着智能化转型？传统运维苦在哪里？智能化运维又能解决哪些问题？

首先从故障产生的原因来说，传统运维是故障触发，而智能运维是隐患驱动。换句话来说，智能运维不用报警，通过看报表就能知道可能要出事了，能够把故障消灭在“萌芽”阶段；第二，传统运维是被动接受，而智能运维是主动出击。但主动出击不一定是通过DBA去做，可能是系统或者机器人操作；第三，传统运维是由DBA发起和解决的，而智能运维是系统发起、RD自助；第四，传统运维属于“人肉救火”，而智能运维属于“智能决策执行”；最后一点，传统运维需要DBA亲临事故现场，而智能运维DBA只需要“隐身幕后”。

从自动化到智能化

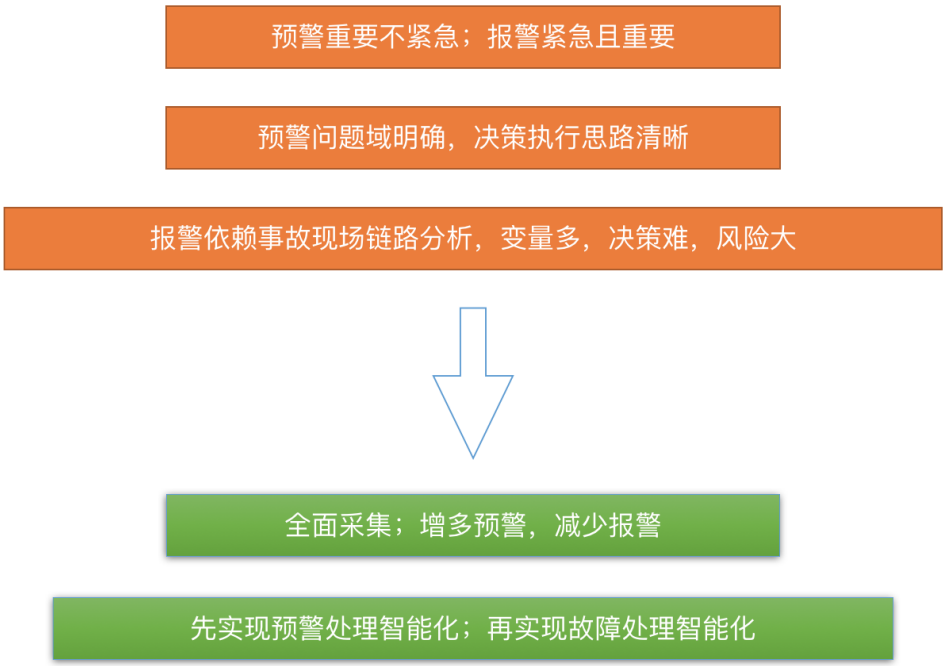
那么，如何从半自动化过渡到自动化，进而发展到智能化运维呢？在这个过程中，我们会面临哪些痛点呢？



我们的目标是为整个公司的业务系统提供高效、稳定、快速的存储服务，这也是DBA存在的价值。业务并不关心后面是MySQL还是NoSQL，只关心数据是否没丢，服务是否可用，出了问题之后多长时间能够恢复等等。所以我们尽可能做到把这些东西对开发人员透明化，提供稳定高效快速的服务。而站在公司的角度，就是在有限的资源下，提升效率，降低成本，尽可能长远地解决问题。

上图是传统运维和智能运维的特点分析，左边属于传统运维，右边属于智能运维。传统运维在采集这一块做的不够，所以它没有太多的数据可供参考，其分析和预警能力是比较弱的。而智能运维刚好是反过来，重采集，很多功夫都在平时做了，包括分析、预警和执行，智能分析并推送关键报表。

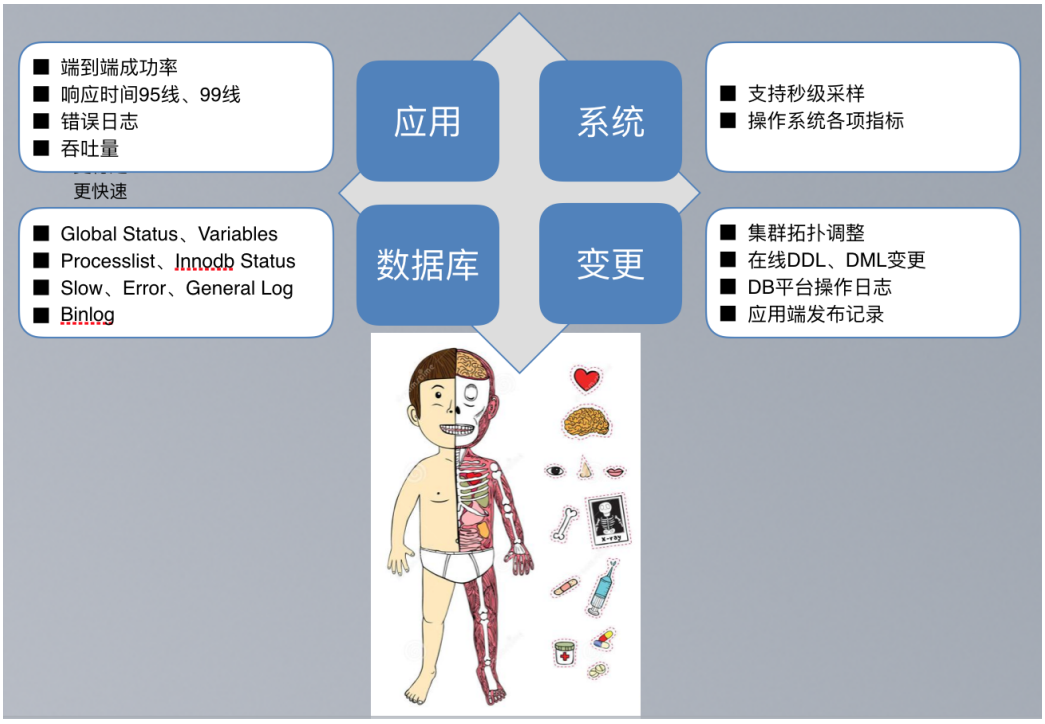
而我们的目标，是让智能运维中的“报警+分析+执行”的比重占据的越来越少。



决策执行如何去做呢？我们都知道，预警重要但不紧急，但报警是紧急且重要的，如果你不能够及时去处理的话，事态可能会扩大，甚至会给公司带来直接的经济损失。

预警通常代表我们已经定位了一个问题，它的决策思路是非常清晰的，可以使用基于规则或AI的方式去解决，相对难度更小一些。而报警依赖于现场的链路分析，变量多、路径长，所以决策难，间接导致任何决

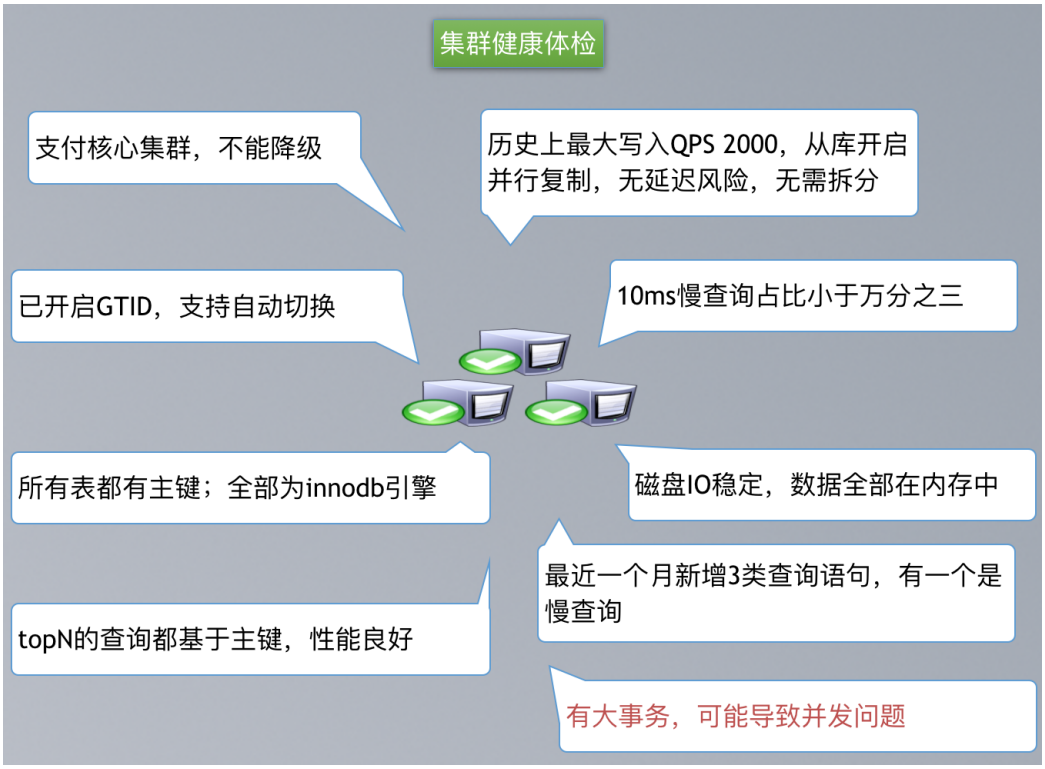
策的风险可能都变大。所以说我们的策略就是全面的采集数据，然后增多预警，率先实现预警发现和处理的智能化。就像我们既有步枪，也有手枪和刺刀，能远距离解决敌人的，就尽量不要短兵相接、肉搏上阵。



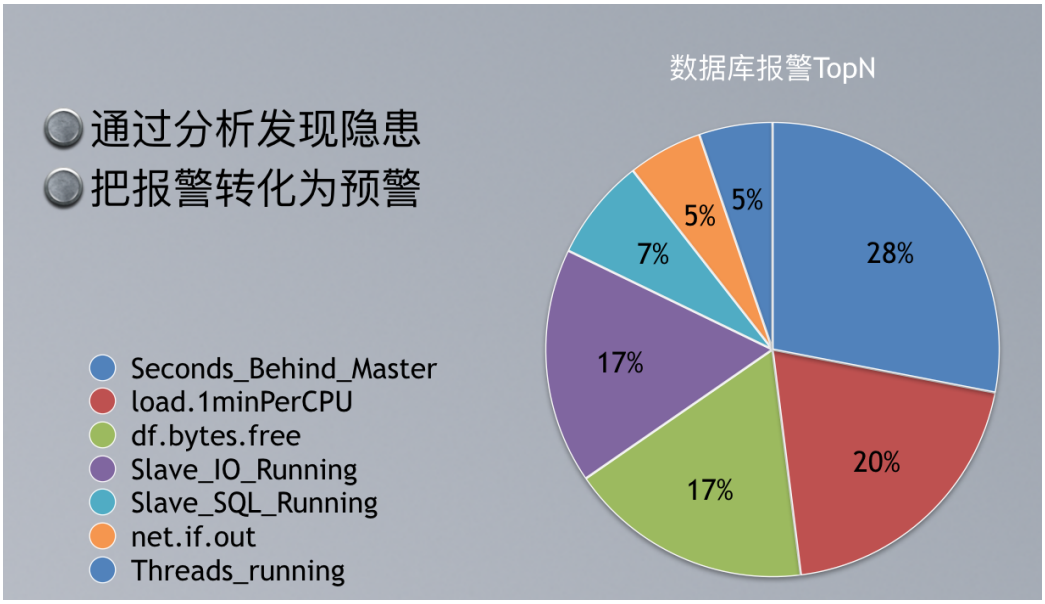
数据采集，从数据库角度来说，我们产生的数据分成四块，Global Status、Variable，Processlist、InnoDB Status，Slow、Error、General Log和Binlog；从应用侧来说，包含端到端成功率、响应时间95线、99线、错误日志和吞吐量；从系统层面，支持秒级采样、操作系统各项指标；从变更侧来看，包含集群拓扑调整、在线DDL、DML变更、DB平台操作日志和应用端发布记录等等。

数据分析，首先是围绕集群分析，接着是实例、库，最后是表，其中每个对象都可以在多项指标上同比和环比，具体对比项可参考上图。

通过上面的步骤，我们基本可以获得数据库的画像，并且帮助我们从整体上做资源规划和服务治理。例如，有些集群实例数特别多且有继续增加的趋势，那么服务器需要scale up；读增加迅猛，读写比变大，那么应考虑存储KV化；利用率和分布情况会影响到服务器采购和预算制定；哪几类报警最多，就专项治理，各个击破。



从局部来说，我们根据分析到的一些数据，可以做一个集群的健康体检，例如数据库的某些指标是否超标、如何做调整等等。

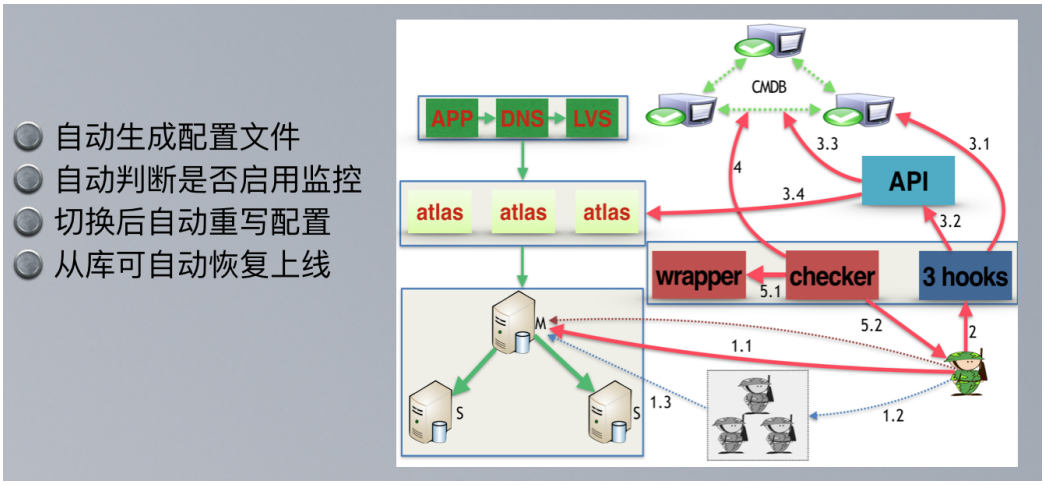


数据库预警，通过分析去发现隐患，把报警转化为预警。上图是我们实际情况下的报警统计分析结果，其中主从延迟占比最大。假设load.1minPerCPU比较高，我们怎么去解决？那么，可能需要采购CPU单核性能更高的机器，而不是采用更多的核心。再比如说磁盘空间，当我们发现3T的磁盘空间普遍不够时，我们下次可以采购6T或更大空间的磁盘。

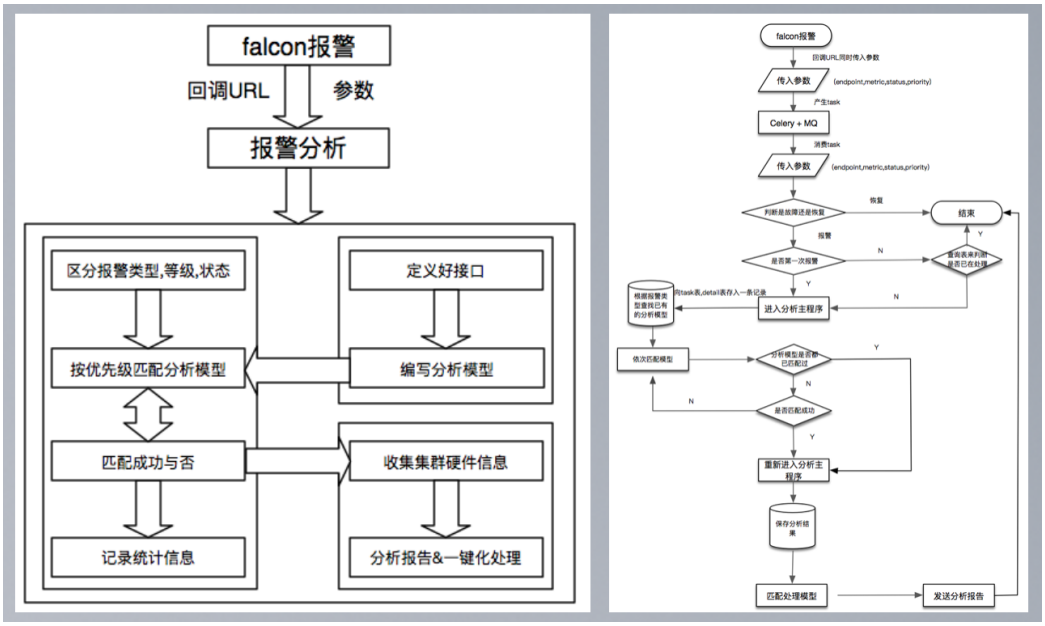
机器名	业务组	服务组	磁盘大小	磁盘空间剩余比例	预测剩余日期	binlog文件个数	binlog文件大小
[redacted]	商家销售	[redacted]	493G	25.61	4	99	98G
[redacted]	商家销售	[redacted]	493G	26.54	4	100	99G
[redacted]	数据应用	[redacted]	296G	13.51	5	41	40G
[redacted]	云计算	[redacted]	296G	13.37	5	52	51G
[redacted]	外卖研发	[redacted]	296G	10.44	6	21	20G
[redacted]	外卖研发	[redacted]	493G	9.43	6	43	42G
[redacted]	外卖研发	[redacted]	296G	11.73	7	22	20G
[redacted]	外卖研发	[redacted]	296G	11.34	7	21	20G
[redacted]	外卖研发	[redacted]	493G	9.11	7	36	35G
[redacted]	酒店后台研发	[redacted]	5.9T	25.56	7	301	303G

针对空间预警问题，什么时候需要拆分集群？MySQL数据库里，拆分或迁移数据库，花费的时间可能会很久。所以需要评估当前集群，按目前的增长速度还能支撑多长时间，进而反推何时要开始拆分、扩容等操作。

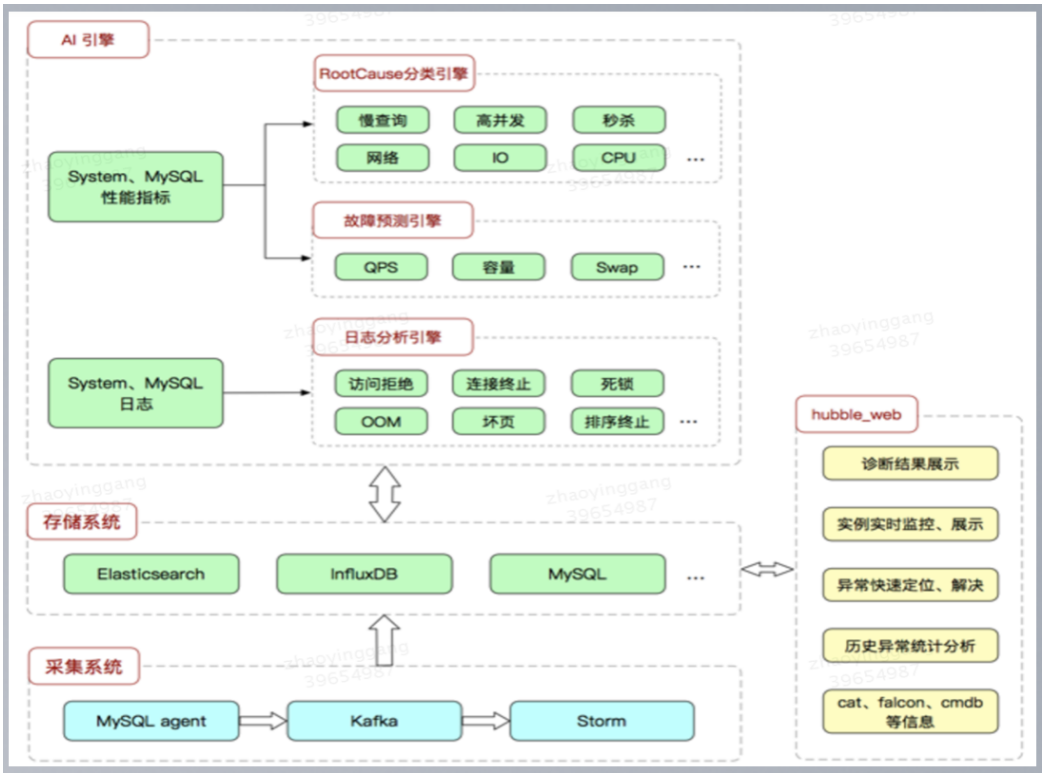
针对慢查询的预警问题，我们会统计红黑榜，上图是统计数据，也有利用率和出轨率的数据。假设这是一个金融事业群的数据库，假设有业务需要访问且是直连，那么这时就会产生几个问题：第一个，有没有数据所有者的授权；第二个，如果不通过服务化方式或者接口，发生故障时，它可能会导致整个金融的数据库挂，如何进行降级？所以，我们会去统计出轨率跟慢查询，如果某数据库正被以一种非法的方式访问，那么我们就会扫描出来，再去进行服务治理。



从运维的层面来说，我们做了故障快速转移，包括自动生成配置文件，自动判断是否启用监控，切换后自动重写配置，以及从库可自动恢复上线等等。



报警自动处理，目前来说大部分的处理工作还是基于规则，在大背景下拟定规则，触发之后，按照满足的前提条件触发动作，随着库的规则定义的逐渐完善和丰富，可以逐步解决很多简单的问题，这部分就不再需要人的参与。



展望

未来我们还会做一个故障诊断平台，类似于“扁鹊”，实现日志的采集、入库和分析，同时提供接口，供全链路的故障定位和分析、服务化治理。

展望智能运维，应该是在自动化和智能化上交叠演进，在ABC（AI、Big Data、Cloud Computing）三个方向上深入融合。在数据库领域，NoSQL和SQL界限正变得模糊，软硬结合、存储计算分离架构也被越来越多的应用，智能运维正当其时，我们也面临更多新的挑战。我们的目标是，希望通过DB平台的不断建设加固，平台能自己发现问题，自动定位问题，并智能的解决问题。

作者简介

- 应钢，美团点评研究员，数据库专家。曾就职于百度、新浪、去哪儿网等，10年数据库自动化运维开发、数据库性能优化、大规模数据库集群技术保障和架构优化经验。精通主流的SQL与NoSQL系统，现专注于公司业务在NewSQL领域的创新和落地。

新一代数据库TiDB在美团的实践

作者: 应钢 李坤 昌俊

1. 背景和现状

近几年，基于MySQL构建的传统关系型数据库服务，已经很难支撑美团业务的爆发式增长，这就促使我们去探索更合理的数据存储方案和实践新的运维方式。而随着分布式数据库大放异彩，美团DBA团队联合基础架构存储团队，于 2018 年初启动了分布式数据库项目。

图 1 美团点评产品展示图

在立项之初，我们进行了大量解决方案的对比，深入了解了业界的 scale-out（横向扩展）、scale-up（纵向扩展）等解决方案。但考虑到技术架构的前瞻性、发展潜力、社区活跃度以及服务本身与 MySQL 的兼容性，我们最终敲定了基于 TiDB 数据库进行二次开发的整体方案，并与 PingCAP 官方和开源社区进行深入合作的开发模式。

美团业务线众多，我们根据业务特点及重要程度逐步推进上线，到截稿为止，已经上线了 10 个集群，近 200 个物理节点，大部分是 OLTP 类型的应用，除了上线初期遇到了一些小问题，目前均已稳定运行。初期上线的集群，已经分别服务于配送、出行、闪付、酒旅等业务。虽然 TiDB 的架构分层相对比较清晰，服务也是比较平稳和流畅，但在美团当前的数据量规模和已有稳定的存储体系的基础上，推广新的存储服务体系，需要对周边工具和系统进行一系列改造和适配，从初期探索到整合落地，仍然还需要走很远的路。下面将从以下几个方面分别进行介绍：

- 从 0 到 1 的突破，重点考虑做哪些事情。
- 如何规划实施不同业务场景的接入和已有业务的迁移。
- 上线后遇到的一些典型问题介绍。

- 后续规划和对未来的展望。

2. 前期调研测试

2.1 对 TiDB 的定位

我们对于 TiDB 的定位，前期在于重点解决 MySQL 的单机性能和容量无法线性和灵活扩展的问题，与 MySQL 形成互补。业界分布式方案很多，我们为何选择了 TiDB 呢？考虑到公司业务规模的快速增长，以及公司内关系数据库以 MySQL 为主的现状，因此我们在调研阶段，对以下技术特性进行了重点考虑：

- 协议兼容 MySQL：这个是必要项。
- 可在线扩展：数据通常要有分片，分片要支持分裂和自动迁移，并且迁移过程要尽量对业务无感知。
- 强一致的分布式事务：事务可以跨分片、跨节点执行，并且强一致。
- 支持二级索引：为兼容 MySQL 的业务，这个是必须的。
- 性能：MySQL 的业务特性，高并发的 OLTP 性能必须满足。
- 跨机房服务：需要保证任何一个机房宕机，服务能自动切换。
- 跨机房双写：支持跨机房双写是数据库领域一大难题，是我们对分布式数据库的一个重要期待，也是美团下一阶段重要的需求。

业界的一些传统方案虽然支持分片，但无法自动分裂、迁移，不支持分布式事务，还有一些在传统 MySQL 上开发一致性协议的方案，但它无法实现线性扩展，最终我们选择了与我们的需求最为接近的 TiDB。与 MySQL 语法和特性高度兼容，具有灵活的在线扩容缩容特性，支持 ACID 的强一致性事务，可以跨机房部署实现跨机房容灾，支持多节点写入，对业务又能像单机 MySQL 一样使用。

2.2 测试

针对官方声称的以上优点，我们进行了大量的研究、测试和验证。

首先，我们需要知道扩容、Region 分裂转移的细节、Schema 到 KV 的映射、分布式事务的实现原理。而 TiDB 的方案，参考了较多的 Google 论文，我们进行了阅读，这有助于我们理解 TiDB 的存储结构、事务算法、安全性等，包括：

- Spanner: Google’s Globally-Distributed Database
- Large-scale Incremental Processing Using Distributed Transactions and Notifications
- In Search of an Understandable Consensus Algorithm
- Online, Asynchronous Schema Change in F1

我们也进行了常规的性能和功能测试，用来与 MySQL 的指标进行对比，其中一个比较特别的测试，是证明 3 副本跨机房部署，确实能保证每个机房分布一个副本，从而保证任何一个机房宕机不会导致丢失超过半数副本。我们从以下几个点进行了测试：

- Raft 扩容时是否支持 Learner 节点，从而保证单机房宕机不会丢失 $^{2}/_{3}$ 的副本。
- TiKV 上的标签优先级是否可靠，保证当机房的机器不平均时，能否保证每个机房的副本数依然是绝对平均的。
- 实际测试，单机房宕机，TiDB 在高并发下，QPS、响应时间、报错数量，以及最终数据是否有丢失。
- 手动 Balance 一个 Region 到其他机房，是否会自动回来。

从测试结果来看，一切都符合我们的预期。

3. 存储生态建设

美团的产品线丰富，业务体量也比较大，业务对在线存储的服务质量要求也非常高。因此，从早期做好服务体系的规划非常重要。下面从业务接入层、监控报警、服务部署等维度，来分别介绍一下我们所做的工作。

3.1 业务接入层

当前 MySQL 的业务接入方式主要有两种，DNS 接入和 Zebra 客户端接入。在前期调研阶段，我们选择了 DNS + 负载均衡组件的接入方式，TiDB-Server 节点宕机，15s 可以被负载均衡识别到，简单且有效。业务架构如下图所示：

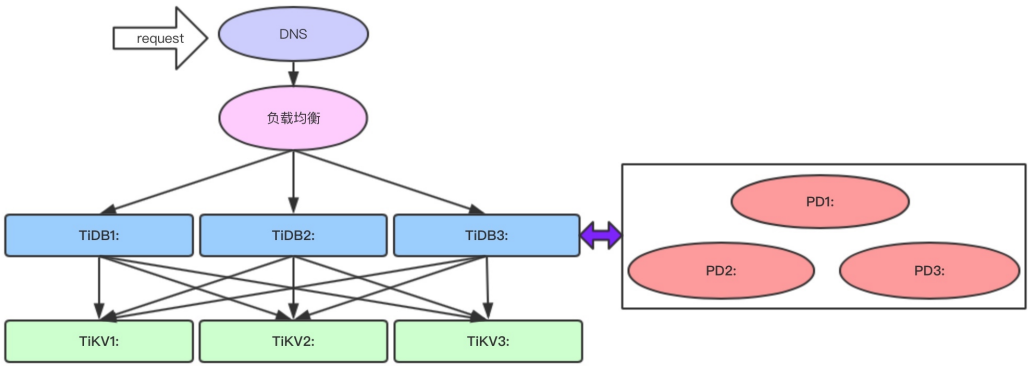


图 2 业务架构图

后面，我们会逐渐过渡到当前大量使用的 Zebra 接入方式来访问 TiDB，从而保持与访问 MySQL 的方式一致，一方面减少业务改造的成本，另一方面尽量实现从 MySQL 到 TiDB 的透明迁移。

3.2 监控报警

美团目前使用 Mt-Falcon 平台负责监控报警，通过在 Mt-Falcon 上配置不同的插件，可以实现对多种组件的自定义监控。另外也会结合 Puppet 识别不同用户的权限、文件的下发。只要我们编写好插件脚本、需要的文件，装机和权限控制就可以完成了。监控架构如下图所示：

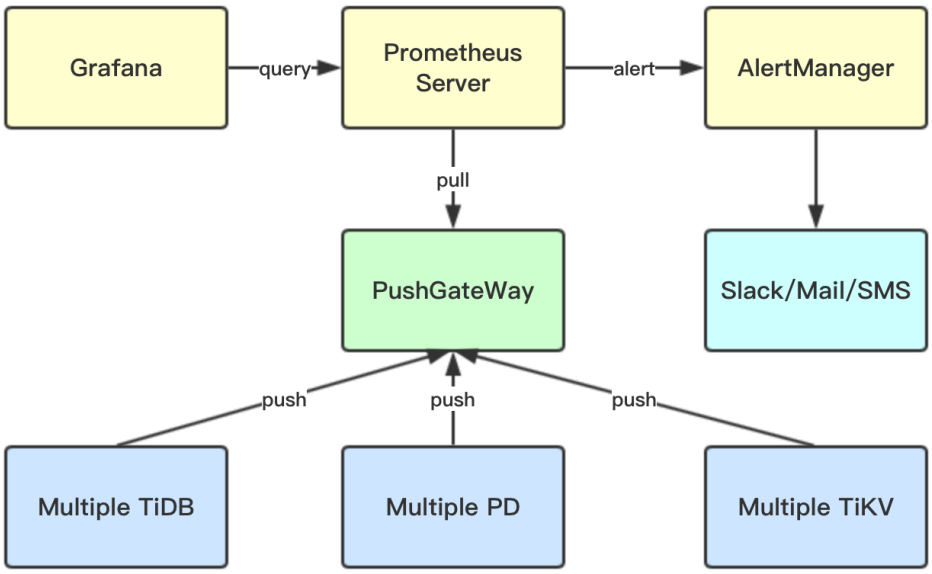


图 3 监控架构图

而 TiDB 有丰富的监控指标，使用流行的 Prometheus + Grafana，一套集群有 700+ 的 Metric。从官方的架构图可以看出，每个组件会推送自己的 Metric 给 PushGateWay，Prometheus 会直接到 PushGateWay 去抓数据。

由于我们需要组件收敛，原生的 TiDB 每个集群一套 Prometheus 的方式不利于监控的汇总、分析、配置，而报警已经在 Mt-Falcon 上实现的比较好了，在 AlertManager 上再造一个也没有必要。因此我们需要想办法把监控和报警汇总到 Mt-Falcon 上面，包括如下几种方式：

- 方案一：修改源代码，将 Metric 直接推送到 Falcon，由于 Metric 散落在代码的不同位置，而且 TiDB 代码迭代太快，把精力消耗在不停调整监控埋点上不太合适。
- 方案二：在 PushGateWay 是汇总后的，可以直接抓取，但 PushGateWay 是个单点，不好维护。
- 方案三：通过各个组件（TiDB、PD、TiKV）的本地 API 直接抓取，优点是组件宕机不会影响其他组件，实现也比较简单。

我们最终选择了方案三。该方案的难点是需要把 Prometheus 的数据格式转化为 Mt-Falcon 可识别的格式，因为 Prometheus 支持 Counter、Gauge、Histogram、Summary 四种数据类型，而 Mt-Falcon 只支持基本的 Counter 和 Gauge，同时 Mt-Falcon 的计算表达式比较少，因此需要在监控脚本中进行转换和计算。

3.3 批量部署

TiDB 使用 Ansible 实现自动化部署。迭代快，是 TiDB 的一个特点，有问题能快速进行解决，但也造成 Ansible 工程、TiDB 版本更新过快，我们对 Ansible 的改动，也只会增加新的代码，不会改动已有的代码。因此线上可能同时需要部署、维护多个版本的集群。如果每个集群一个 Ansible 目录，造成空间的浪费。

我们采用的维护方式是，在中控机中，每个版本一个 Ansible 目录，每个版本中通过不同 inventory 文件来维护。这里需要跟 PingCAP 提出的是，Ansible 只考虑了单集群部署，大量部署会有些麻烦，像一些

依赖的配置文件，都不能根据集群单独配置（咨询官方得知，PingCAP 目前正在基于 Cloud TiDB 打造一站式 HTAP 平台，会提供批量部署、多租户等功能，后续会比较好地解决这个问题）。

3.4 自动化运维平台

随着线上集群数量的增加，打造运维平台提上了日程，而美团对 TiDB 和 MySQL 的使用方式基本相同，因此 MySQL 平台上具有的大部分组件，TiDB 平台也需要建设。典型的底层组件和方案：SQL 审核模块、DTS、数据备份方案等。自动化运维平台展示如下图所示：

图 4 自动化运维平台展示图

3.5 上下游异构数据同步

TiDB 是在线存储体系中的一环，它同时也需要融入到公司现有的数据流中，因此需要一些工具来做衔接。PingCAP 官方标配了相关的组件。

公司目前 MySQL 和 Hive 结合的比较重，而 TiDB 要代替 MySQL 的部分功能，需要解决 2 个问题：

- MySQL to TiDB
  - MySQL 到 TiDB 的迁移，需要解决数据迁移以及增量的实时同步，也就是 DTS，Mydumper + Loader 解决存量数据的同步，官方提供了 DM 工具可以很好的解决增量同步问题。
  - MySQL 大量使用了自增 ID 作为主键。分库分表 MySQL 合并到 TiDB 时，需要解决自增 ID 冲突的问题。这个通过在 TiDB 端去掉自增 ID 建立自己的唯一主键来解决。新版 DM 也提供分表合并过程主键自动处理的功能。
- Hive to TiDB & TiDB to Hive

- Hive to TiDB 比较好解决，这体现了 TiDB 和 MySQL 高度兼容的好处，insert 语句可以不用调整，基于 Hive to MySQL 简单改造即可。
- TiDB to Hive 则需要基于官方 Pump + Drainer 组件，Drainer 可以消费到 Kafka、MySQL、TiDB，我们初步考虑用图 5 中的方案通过使用 Drainer 的 Kafka 输出模式同步到 Hive。

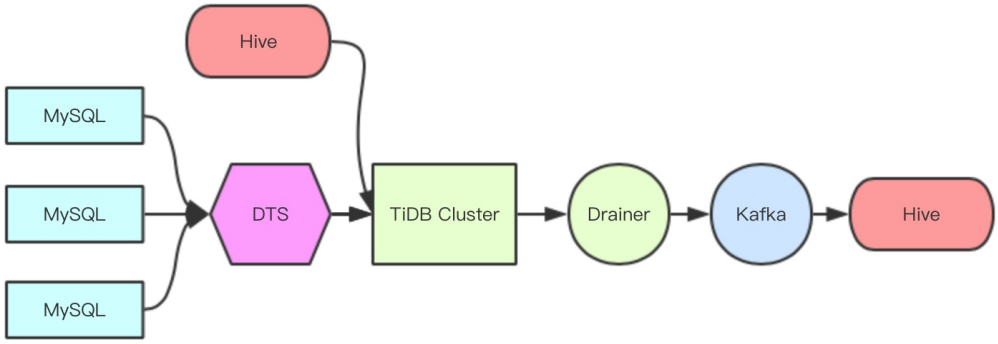


图 5 TiDB to Hive 方案图

4. 线上使用磨合

对于初期上线的业务，我们比较谨慎，基本的原则是：离线业务 -> 非核心业务 -> 核心业务。TiDB 已经发布两年多，且前期经历了大量的测试，我们也深入了解了其它公司的测试和使用情况，可以预期的是 TiDB 上线会比较稳定，但依然遇到了一些小问题。总体来看，在安全性、数据一致性等关键点上没有出现问题。其他一些性能抖动问题，参数调优的问题，也都得到了快速妥善的解决。这里给 PingCAP 的同学点个大大的赞，问题响应速度非常快，与我们美团内部研发的合作也非常融洽。

4.1 写入量大、读 QPS 高的离线业务

我们上线的最大的一个业务，每天有数百 G 的写入量，在前期，我们也遇到了较多的问题。

业务场景：

- 稳定的写入，每个事务操作 100~200 行不等，每秒 6W 的数据写入。
- 每天的写入量超过 500G，以后会逐步提量到每天 3T。
- 每 15 分钟的定时读 Job，5000 QPS（高频量小）。
- 不定时的查询（低频量大）。

之前使用 MySQL 作为存储，但 MySQL 到达了容量和性能瓶颈，而业务的容量未来会 10 倍的增长。初期调研测试了 ClickHouse，满足了容量的需求，测试发现运行低频 SQL 没有问题，但高频 SQL 的大并发查询无法满足需求，只在 ClickHouse 跑全量的低频 SQL 又会 overkill，最终选择使用 TiDB。

测试期间模拟写入了一天的真实数据，非常稳定，高频低频两种查询也都满足需求，定向优化后 OLAP 的 SQL 比 MySQL 性能提高四倍。但上线后，陆续发现了一些问题，典型的如下：

4.1.1 TiKV 发生 Write Stall

TiKV 底层有 2 个 RocksDB 作为存储。新写的数据写入 L0 层，当 RocksDB 的 L0 层数量达到一定数量，就会发生减速，更高则发生 Stall，用来自我保护。TiKV 的默认配置：

- level0-slowdown-writes-trigger = 20
- level0-stop-writes-trigger = 36

遇到过的，发生 L0 文件过多可能的原因有 2 个：

- 写入量大，Compact 完不成。
- Snapshot 一直创建不完，导致堆积的副本一下释放，RocksDB-Raft 创建大量的 L0 文件，监控展示如下图所示：

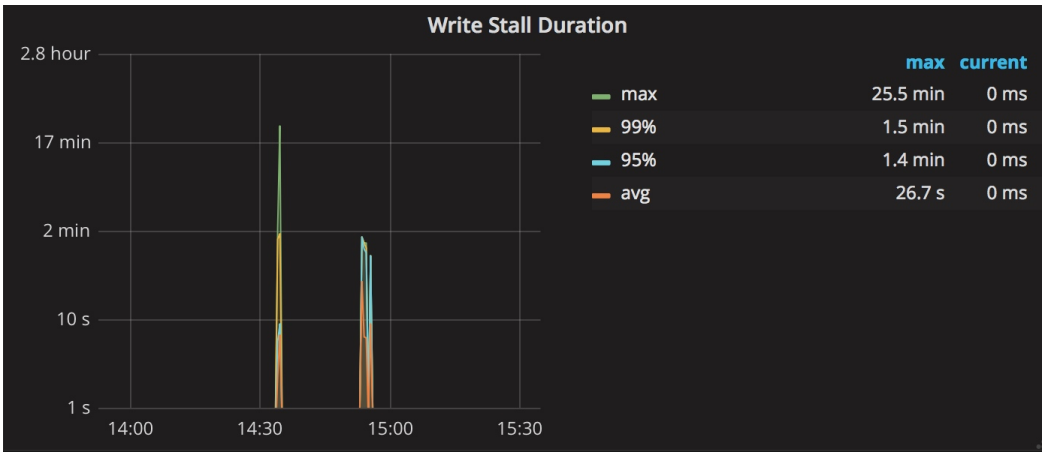


图 6 TiKV 发生 Write Stall 监控展示图

我们通过以下措施，解决了 Write Stall 的问题：

- 减缓 Raft Log Compact 频率（增大 raft-log-gc-size-limit、raft-log-gc-count-limit）
- 加快 Snapshot 速度（整体性能、包括硬件性能）
- max-sub-compactions 调整为 3
- max-background-jobs 调整为 12
- level 0 的 3 个 Trigger 调整为 16、32、64

4.1.2 Delete 大量数据，GC 跟不上

现在 TiDB 的 GC 对于每个 kv-instance 是单线程的，当业务删除数据的量非常大时，会导致 GC 速度较慢，很可能 GC 的速度跟不上写入。

目前可以通过增多 TiKV 个数来解决，长期需要靠 GC 改为多线程执行，官方对此已经实现，即将发布。

4.1.3 Insert 响应时间越来越慢

业务上线初期，insert 的响应时间 80 线（Duration 80 By Instance）在 20ms 左右，随着运行时间增加，发现响应时间逐步增加到 200ms+。期间排查了多种可能原因，定位在由于 Region 数量快速上涨，Raftstore 里面要做的事情变多了，而它又是单线程工作，每个 Region 定期都要 heartbeat，带来了性能消耗。tikv-raft propose wait duration 指标持续增长。

解决问题的办法：

- 临时解决。
- 增加 Heartbeat 的周期，从 1s 改为 2s，效果比较明显，监控展示如下图所示：

图 7 insert 响应时间优化前后对比图

- 彻底解决。
- 需要减少 Region 个数，Merge 掉空 Region，官方在 2.1 版本中已经实现了 Region Merge 功能，我们在升级到 2.1 后，得到了彻底解决。
- 另外，等待 Raftstore 改为多线程，能进一步优化。（官方回复相关开发已基本接近尾声，将于 2.1 的下一个版本发布。）

4.1.4 Truncate Table 空间无法完全回收

DBA Truncate 一张大表后，发现 2 个现象，一是空间回收较慢，二是最终也没有完全回收。

- 由于底层 RocksDB 的机制，很多数据落在 Level 6 上，有可能清不掉。这个需要打开 cdynamic-level-bytes 会优化 Compaction 的策略，提高 Compact 回收空间的速度。
- 由于 Truncate 使用 delete_files_in_range 接口，发给 TiKV 去删 SST 文件，这里只删除不相交的部分，而之前判断是否相交的粒度是 Region，因此导致了大量 SST 无法及时删除掉。
- 考虑 Region 独立 SST 可以解决交叉问题，但是随之带来的是磁盘占用问题和 Split 延时问题。
- 考虑使用 RocksDB 的 DeleteRange 接口，但需要等该接口稳定。
- 目前最新的 2.1 版本优化为直接使用 DeleteFilesInRange 接口删除整个表占用的空间，然后清理少量残留数据，目前已经解决。

4.1.5 开启 Region Merge 功能

为了解决 region 过多的问题，我们在升级 2.1 版本后，开启了 region merge 功能，但是 TiDB 的响应时间 80 线（Duration 80 By Instance）依然没有恢复到当初，保持在 50ms 左右，排查发现 KV 层返回的响应时间还很快，和最初接近，那么就定位了问题出现在 TiDB 层。研发人员和 PingCAP 定位在产生执行计划时行为和 2.0 版本不一致了，目前已经优化。

4.2 在线 OLTP，对响应时间敏感的业务

除了分析查询量大的离线业务场景，美团还有很多分库分表的场景，虽然业界有很多分库分表的方案，解决了单机性能、存储瓶颈，但是对于业务还是有些不友好的地方：

- 业务无法友好的执行分布式事务。
- 跨库的查询，需要在中间层上组合，是比较重的方案。

- 单库如果容量不足，需要再次拆分，无论怎样做，都很痛苦。
- 业务需要关注数据分布的规则，即使用了中间层，业务心里还是没底。

因此很多分库分表的业务，以及即将无法在单机承载而正在设计分库分表方案的业务，主动找到了我们，这和我们对于 TiDB 的定位是相符的。这些业务的特点是 SQL 语句小而频繁，对一致性要求高，通常部分数据有时间属性。在测试及上线后也遇到了一些问题，不过目前基本都有了解决办法。

4.2.1 SQL 执行超时后，JDBC 报错

业务偶尔报出 privilege check fail。

是由于业务在 JDBC 设置了 QueryTimeout，SQL 运行超过这个时间，会发行一个 `kill query` 命令，而 TiDB 执行这个命令需要 Super 权限，业务是没有权限的。其实 kill 自己的查询，并不需要额外的权限，目前已经解决了这个问题：

https://github.com/pingcap/tidb/pull/7003，不再需要 Super 权限，已在 2.0.5 上线。

4.2.2 执行计划偶尔不准

TiDB 的物理优化阶段需要依靠统计信息。在 2.0 版本统计信息的收集从手动执行，优化为在达到一定条件时可以自动触发：

- 数据修改比例达到 tidb_auto_analyze_ratio。
- 表一分钟没有变更（目前版本已经去掉这个条件）。

但是在没有达到这些条件之前统计信息是不准的，这样就会导致物理优化出现偏差，在测试阶段（2.0 版本）就出现了这样一个案例：业务数据是有时间属性的，业务的查询有 2 个条件，比如：时间+商家 ID，但每天上午统计信息可能不准，当天的数据已经有了，但统计信息认为没有。这时优化器就会建议使用时间列的索引，但实际上商家 ID 列的索引更优化。这个问题可以通过增加 Hint 解决。

在 2.1 版本对统计信息和执行计划的计算做了大量的优化，也稳定了基于 Query Feedback 更新统计信息，也用于更新直方图和 Count-Min Sketch，非常期待 2.1 的 GA。

5. 总结展望

经过前期的测试、各方的沟通协调，以及近半年对 TiDB 的使用，我们看好 TiDB 的发展，也对未来基于 TiDB 的合作充满信心。

接下来，我们会加速推进 TiDB 在更多业务系统中的使用，同时也将 TiDB 纳入了美团新一代数据库的战略选型中。当前，我们已经全职投入了 3 位 DBA 同学和多位存储计算专家，从底层的存储，中间层的计算，业务层的接入，再到存储方案的选型和布道，进行全方位和更深入的合作。

长期来看，结合美团不断增长的业务规模，我们将与 PingCAP 官方合作打造更强大的生态体系：

- Titan：Titan 是 TiDB 下一步比较大的动作，也是我们非常期待的下一代存储引擎，它对大 Value 支持会更友好，将解决我们单行大小受限，单机 TiKV 最大支持存储容量的问题，大大提升大规模部署的性价比。

- Cloud TiDB （Based on Docker & K8s）：云计算大势所趋，PingCAP 在这块也布局比较早，今年 8 月份开源了 TiDB Operator，Cloud TiDB 不仅实现了数据库的高度自动化运维，而且基于 Docker 硬件隔离，实现了数据库比较完美的多租户架构。我们和官方同学沟通，目前他们的私有云方案在国内也有重要体量的 POC，这也是美团看重的一个方向。
- TiDB HTAP Platform：PingCAP 在原有 TiDB Server 计算引擎的基础上，还构建 TiSpark 计算引擎，和他们官方沟通，他们在研发了一个基于列的存储引擎，这样就形成了下层行、列两个存储引擎、上层两个计算引擎的完整混合数据库（HTAP），这个架构不仅大大的节省了核心业务数据在整个公司业务周期里的副本数量，还通过收敛技术栈，节省了大量的人力成本、技术成本、机器成本，同时还解决了困扰多年的 OLAP 的实效性。后面我们也会考虑将一些有实时、准实时的分析查询系统接入 TiDB。

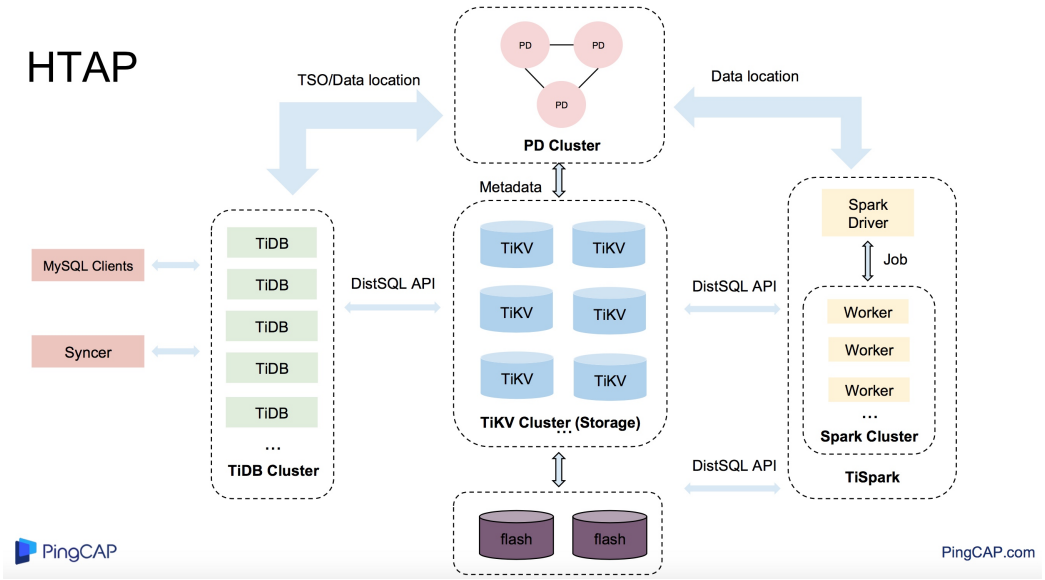


图 8 TiDB HTAP Platform 整体架构图

- 后续的物理备份方案，跨机房多写等也是我们接下来逐步推进的场景，总之，我们坚信未来 TiDB 在美团的使用场景会越来越多，发展也会越来越好。

目前，TiDB 在业务层面、技术合作层面都已经在美团扬帆起航，美团点评将携手 PingCAP 开启新一代数据库深度实践、探索之旅。后续，还有美团点评架构存储团队针对 TiDB 源码研究和改进的系列文章，敬请期待。

作者简介

- 应钢，美团点评研究员，数据库专家。曾就职于百度、新浪、去哪儿网等，10年数据库自动化运维开发、数据库性能优化、大规模数据库集群技术保障和架构优化经验。精通主流的SQL与NoSQL系统，现专注于公司业务在NewSQL领域的创新和落地。
- 李坤，2018年初加入美团，美团点评数据库专家，多年基于MySQL、Hbase、Oracle的架构设计和维护、自动化开发经验，目前主要负责分布式数据库Blade的推动和落地，以及平台和周边组件的建设
- 昌俊，美团点评数据库专家，曾就职于BOCO、去哪儿网，6年MySQL DBA从业经历，积累了丰富的数据库架构设计和性能优化、自动化开发经验。目前专注于TiDB在美团点评业务场景的改造和落地。

Android Hook技术防范漫谈

作者: 礼赞 毅然

背景

当下，数据就像水、电、空气一样无处不在，说它是“21世纪的生产资料”一点都不夸张，由此带来的是，各行业对于数据的争夺热火朝天。随着互联网和数据的思维深入人心，一些灰色产业悄然兴起，数据贩子、爬虫、外挂软件等等也接踵而来，互联网行业中各公司竞争对手之间不仅业务竞争十分激烈，黑科技的比拼也越发重要。随着移动互联网的兴起，爬虫和外挂也从单一的网页转向了App，其中利用Android平台下Dalvik模式中的 `Xposed Installer` 和 `Cydia Substrate` 框架对App的函数进行Hook这一招，堪称老牌经典。

接下来，本文将分别介绍针对这两种框架的防护技术。

Xposed Installer

原理

Zygote

在Android系统中App进程都是由Zygote进程“孵化”出来的。Zygote进程在启动时会创建一个虚拟机实例，每当它“孵化”一个新的应用程序进程时，都会将这个Dalvik虚拟机实例复制到新的App进程里面去，从而使每个App进程都有一个独立的Dalvik虚拟机实例。

Zygote进程在启动的过程中，除了会创建一个虚拟机实例之外还会将 `Java Rumtime` 加载到进程中并注册一些Android核心类的JNI（Java Native Interface，Java本地接口）方法。一个App进程被Zygote进程孵化出来的时候，不仅会获得Zygote进程中的虚拟机实例拷贝，还会与Zygote进程一起共享 `Java Rumtime` ，也就是可以将 `XposedBridge.jar` 这个Jar包加载到每一个Android App进程中去。安装 `Xposed Installer` 之后，系统 `app_process` 将被替换，然后利用Java的 `Reflection` 机制覆写内置方法，实现功能劫持。下面我们来看一下细节。

Hook和Replace

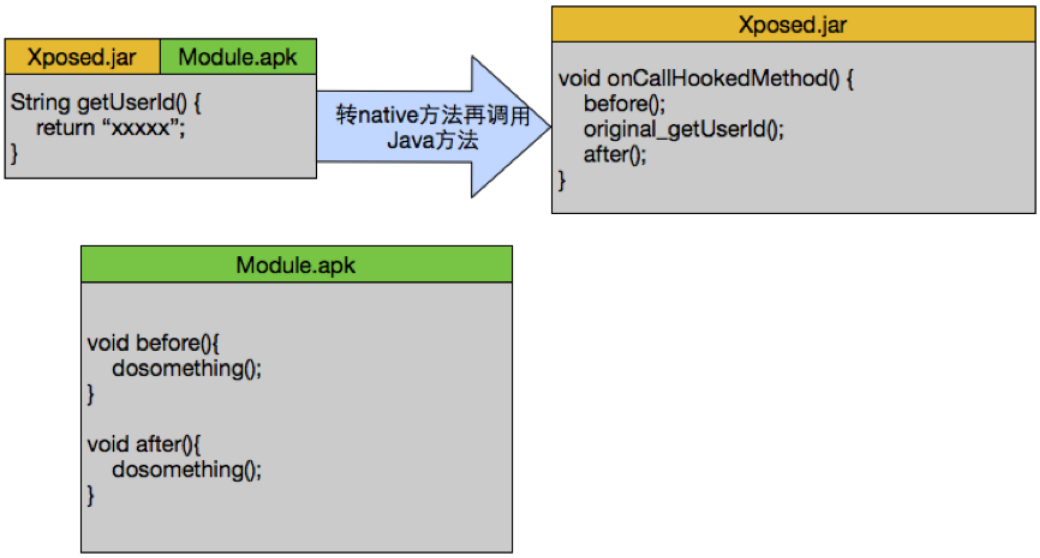
`Xposed Installer` 框架中真正起作用的是对方法的Hook和Replace。在Android系统启动的时候，Zygote进程加载 `XposedBridge.jar` ，将所有需要替换的Method通过 `JNI` 方法 `hookMethodNative` 指向Native方法 `xposedCallHandler` ，这个方法再通过调用 `handleHookedMethod` 这个Java方法来调用被劫持的方法转入Hook逻辑。

上面提到的 `hookMethodNative` 是 `XposedBridge.jar` 中的私有的本地方法，它将一个方法对象作为传入参数并修改Dalvik虚拟机中对于该方法的定义，把该方法的类型改变为Native并将其实现指向另外一个

B方法。

换言之，当调用那个被Hook的A方法时，其实调用的是B方法，调用者是不知道的。在hookMethodNative的实现中，会调用 `XposedBridge.jar` 中的 `handleHookedMethod` 这个方法来传递参数。 `handleHookedMethod` 这个方法类似于一个统一调度的Dispatch例程，其对应的底层的C++函数是 `xposedCallHandler` 。而 `handleHookedMethod` 实现里面会根据一个全局结构 `hookedMethodCallbacks` 来选择相应的Hook函数并调用他们的 `before` 和 `after` 函数，当多模块同时Hook一个方法的时候 `Xposed` 会自动根据 `Module` 的优先级来排序。

调用顺序如下：A.before –> B.before –> original method –> B.after –> A.after。



检测

在做Android App的安全防御中检测点众多， `Xposed Installer` 检测是必不可少的一环。对于Xposed框架的防御总体上分为两层：Java层和Native层。

Java层检测

需要说明的是，Java层的检测基本只能检测出基础的 `Xposed Installer` 框架，而不能防护其对App内方法的Hook，如果框架中带有反检测则Java层检测大多不起作用。

下面列出Java层的检测点，仅供参考。

① 通过PackageManager查看安装列表

最简单的检测，我们调用Android提供的 `PackageManager` 的API来遍历系统中App的安装情况来辨别是否有安装 `Xposed Installer` 相关的软件包。

```
PackageManager packageManager = context.getPackageManager();
List applicationInfoList = packageManager.getInstalledApplications(PackageManager.GET_META_DATA);
for (ApplicationInfo applicationInfo: applicationInfoList) {
    if (applicationInfo.packageName.equals("de.robv.android.xposed.installer")) {
        // is Xposed TODO... }
    }
```

通常情况下使用 Xposed Installer 框架都会屏蔽对其的检测，即Hook掉 PackageManager的 getInstalledApplications 方法的返回值，以便过滤掉 de.robv.android.xposed.installer 来躲避这种检测。

② 自造异常读取栈

Xposed Installer 框架对每个由Zygote孵化的App进程都会介入，因此在程序方法异常栈中就会出现 Xposed 相关的“身影”，我们可以通过自造异常 Catch 来读取异常堆栈的形式，用以检查其中是否存在 Xposed 的调用方法。

```
try {
    throw new Exception("blah");
} catch(Exception e) {
    for (StackTraceElement stackTraceElement: e.getStackTrace()) {
        // stackTraceElement.getClassName() stackTraceElement.getMethodName() 是否存 在Xposed
    }
}
```

```
E/GEnvironment: no such table: preference (code 1): while compiling: SELECT keyguard_show_livewallpaper FROM preference
...
at com.meituan.test.extpackage.ExtPackageManager.checkUpdate(ExtPackageManager.java:127)
at com.meituan.test.MiFGService$1.run(MiFGService.java:41)
at android.os.Looper.loop(Looper.java:136)
at android.app.ActivityThread.main(ActivityThread.java:5072)
at java.lang.reflect.Method.invokeNative(Native Method)
at java.lang.reflect.Method.invoke(Method.java:515)
...
at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:793)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:609)
at de.robv.android.xposed.XposedBridge.main(XposedBridge.java:132) //发现Xposed模块
at dalvik.system.NativeStart.main(Native Method)
```

③ 检查关键Java方法被变为Native JNI方法

当一个Android App中的 Java 方法被莫名其妙地变成了 Native JNI 方法，则非常有可能被 Xposed Hook 了。由此可得，检查关键方法是不是变成 Native JNI 方法，也可以检测是否被Hook。

通过反射调用 Modifier.isNative(method.getModifiers()) 方法可以校验方法是不是 Native JNI 方法，Xposed同样可以篡改 isNative 这个方法的返回值。

④ 反射读取XposedHelper类字段

通过反射遍历 XposedHelper 类中的 fieldCache 、 methodCache 、 constructorCache 变量，读取HashMap缓存字段，如字段项的key中包含App中唯一或敏感方法等，即可认为有 Xposed 注入。

```
public final class XposedHelpers {
        private XposedHelpers() {}

        private static final HashMap<String, Field> fieldCache = new HashMap<>();
        private static final HashMap<String, Method> methodCache = new HashMap<>();
        private static final HashMap<String, Constructor<?>> constructorCache = new HashMap<>();
        private static final WeakHashMap<Object, HashMap<String, Object>> additionalFields = new WeakHashMap<>();
        private static final HashMap<String, ThreadLocal<AtomicInteger>> sMethodDepth = new HashMap<>();
```

```
boolean methodCache = CheckHook(clsXposedHelper, "methodCache", keyWord);

private static boolean CheckHook(Object cls, String filedName, String str) {
    boolean result = false;
    String interName;
    Set keySet;
    try {
        Field filed = cls.getClass().getDeclaredField(filedName);
        filed.setAccessible(true);
```

```
        keySet = filed.get(cls)).keySet();
        if (!keySet.isEmpty()) {
            for (Object aKeySet: keySet) {
                interName = aKeySet.toString().toLowerCase();
                if (interName.contains("meituan") || interName.contains("dianping") ) {
                    result = true;
                    break;
                        }
                }
            }
                ...
    return result;
}
```

Native层检测

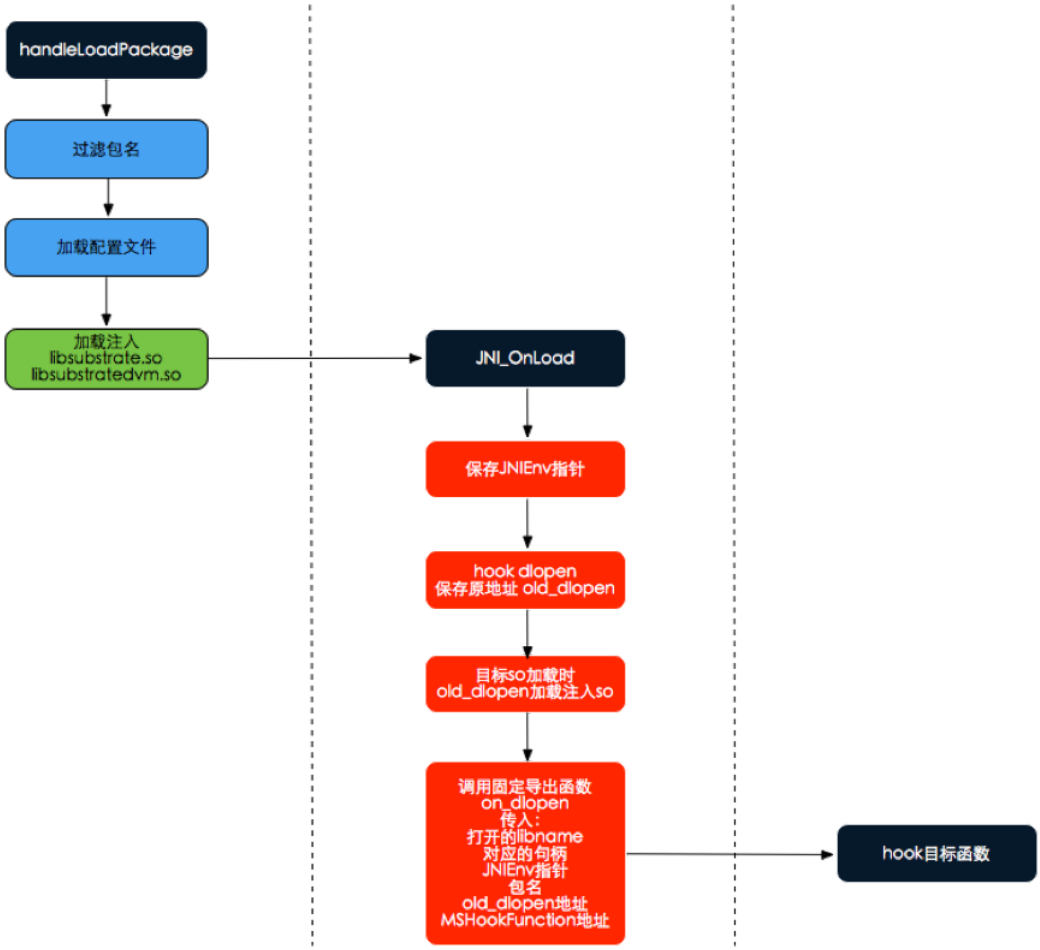
由上文可知，无论在Java层做何种检测，Xposed都可以通过Hook相关的API并返回指定的结果来绕过检测，只要有方法就可以被Hook。如果仅在Java层检测就显得很徒劳，为了有效提搞检测准确率，就须做到Java和Native层同时检测。每个App在系统中都有对应的加载库列表，这些加载库列表在 `/proc/` 下对应的 `pid/maps` 文件中描述，在Native层读取 `/proc/self/maps` 文件不失为检测Xposed Installer的有效办法之一。由于 `Xposed Installer` 通常只能Hook Java层，因此在Native层使用C来解析 `/proc/self/maps` 文件，搜检App自身加载的库中是否存在 `XposedBridge.jar` 、相关的Dex、Jar和So库等文件。

```
bool is_xposed()
{
   bool rel = false;
   FILE *fp = NULL;
   char* filepath = "/proc/self/maps";
   ...
   string xp_name = "XposedBridge.jar";
   fp = fopen(filepath,"r"))
   while (!feof(fp))
   {
       fgets(strLine,BUFFER_SIZE,fp);
       origin_str = strLine;
       str = trim(origin_str);
       if (contain(str,xp_name))
       {
           rel = true; //检测到Xposed模块
           break;
       }
   }
        ...
}
```

Cydia Substrate

原理

`Cydia Substrate` 注入Hook的一个典型流程如下图所示，在Java层配置注入的关键So库 `libsubstrate.so` 和 `libsubstratedvm.so` 。考虑到Java层检测强度太低，Substrate的检测主要在Native层来实现。



检测

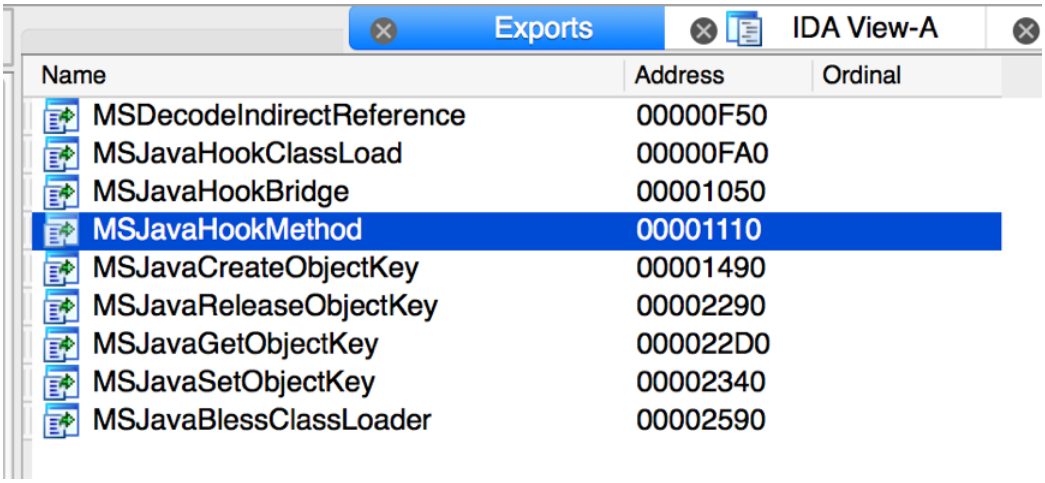
动态加载式检测

读取 `/proc/self/maps` ，列出了App中所有加载的文件。

```
root@cancro:/ # cat /proc/32167/maps | grep substrate
4011c000-40122000 r-xp 00000000 b3:1c 458914     /data/app-lib/com.saurik.substrate-2/libAndroidBootstrap0.so
40122000-40124000 rw-p 00005000 b3:1c 458914     /data/app-lib/com.saurik.substrate-2/libAndroidBootstrap0.so
41762000-41765000 r-xp 00000000 b3:1c 458915     /data/app-lib/com.saurik.substrate-2/libAndroidLoader.so
41765000-41767000 rw-p 00002000 b3:1c 458915     /data/app-lib/com.saurik.substrate-2/libAndroidLoader.so
41767000-4176b000 r-xp 00000000 b3:1c 458912     /data/app-lib/com.saurik.substrate-2/libsubstrate.so
4176b000-4176d000 rw-p 00003000 b3:1c 458912     /data/app-lib/com.saurik.substrate-2/libsubstrate.so
4176d000-41770000 r-xp 00000000 b3:1c 458909     /data/app-lib/com.saurik.substrate-2/libAndroidCydia.cy.so (deleted)
41770000-41771000 r--p 00002000 b3:1c 458909     /data/app-lib/com.saurik.substrate-2/libAndroidCydia.cy.so (deleted)
41771000-41772000 rw-p 00003000 b3:1c 458909     /data/app-lib/com.saurik.substrate-2/libAndroidCydia.cy.so (deleted)
41772000-41777000 r-xp 00000000 b3:1c 458910     /data/app-lib/com.saurik.substrate-2/libsubstrate-dvm.so
41777000-41778000 r--p 00004000 b3:1c 458910     /data/app-lib/com.saurik.substrate-2/libsubstrate-dvm.so
41778000-41779000 rw-p 00005000 b3:1c 458910     /data/app-lib/com.saurik.substrate-2/libsubstrate-dvm.so
4177e000-41784000 r-xp 00000000 b3:1c 458911     /data/app-lib/com.saurik.substrate-2/libDalvikLoader.cy.so
41784000-41785000 r--p 00005000 b3:1c 458911     /data/app-lib/com.saurik.substrate-2/libDalvikLoader.cy.so
41785000-41786000 rw-p 00006000 b3:1c 458911     /data/app-lib/com.saurik.substrate-2/libDalvikLoader.cy.so
```

上图为 `Cydia Substrate` 在Android 4.4上注入后的进程maps表，其中 `libsubstrate.so` 和 `libsubstrate-dvm.so` 两个文件为Substrate必载入文件。通过 `IDA Pro` 分析对其分析。

先来看 `libsubstrate-dvm.so` 的导出表，共有9个函数导出。



当进程maps表中出现 `libsubstrate-dvm.so` ，可以尝试去load该so文件并调用 `MSJavaHookMethod` 方法，它会返回该方法的地址即判定为恶意模块（第三方程序）。

```
void * dlopen = lookup_symbol("/data/app-lib/com.saurik.substrate-2/libsubstrate-dvm.so","MSJavaHookMethod");
```

```
void* lookup_symbol(char* libraryname,char* symbolname)
{
    void *imagehandle = dlopen(libraryname, RTLD_GLOBAL | RTLD_NOW);
    if (imagehandle != NULL){
        void * sym = dlsym(imagehandle, symbolname);
        if (sym != NULL){
            return sym; //发现Cydia Substrate相关模块
            }
      ...
}
```

该方式基于载入库文件的文件名或文件路径和导出函数来判断是否为恶意模块，如果完全依赖此方式来判断可能会误判，但也不失为检测方式的一个点。

基于方法特征码检测

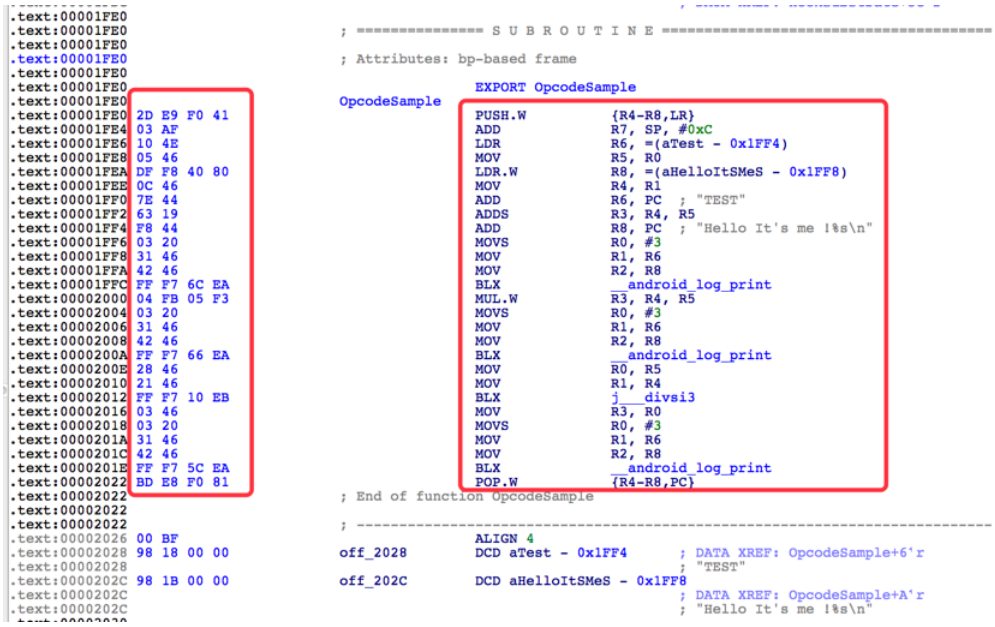
特征码即用来判断某段数据属于哪个计算机字段。在非Root环境下一般一个正常App在启动时候，系统会调度相关大小的内存、空间给App使用，此时App的运行环境内产生的数据、内存、存储等是独立于其它App的（即独立运行在沙箱中）。因为处于运行沙箱环境中的进程对沙箱的内存有最高读写权限，当我们的App进程被恶意模块附加或注入时，就可以通过对当前进程的PID所对应的maps中加载的模块进行合法校验。这里的模块校验我们可以采取对单个模块内容取样来判断是否为恶意模块，这种方式被定义为“基于方法的特征码检测”。

下面对一段程序段中 `OpcodeSample` 方法来提取特征码。

方法原型：

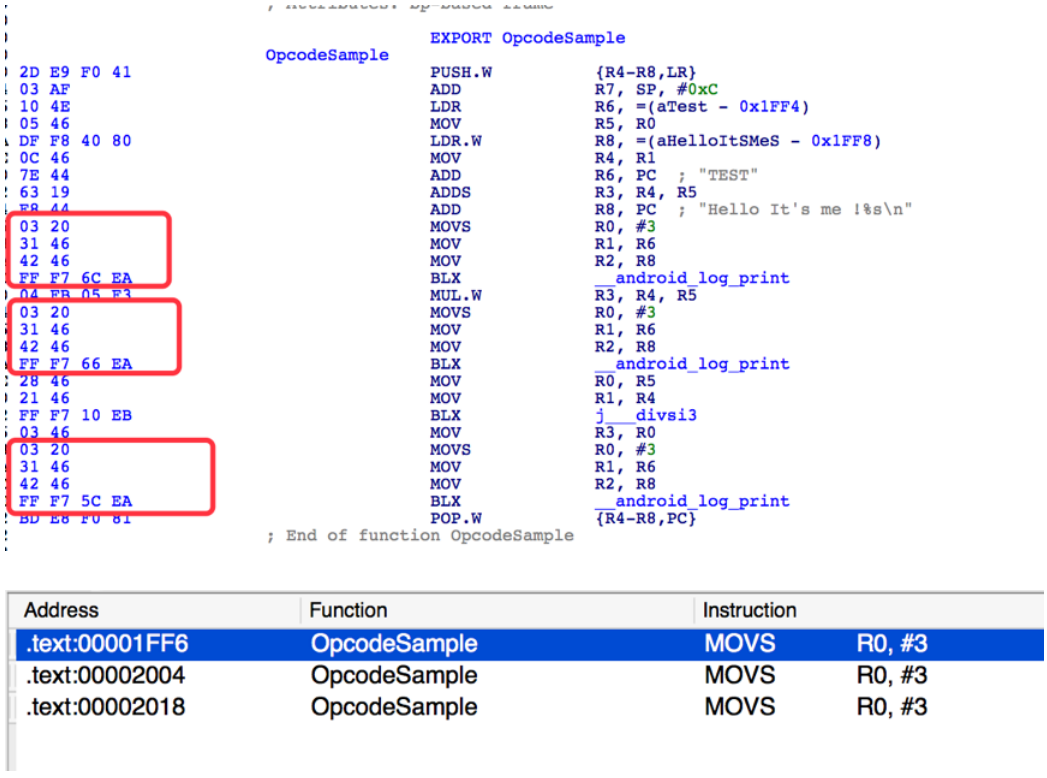
```
#define  LOGD(fmt, args...)  __android_log_print(ANDROID_LOG_DEBUG,LOG_TAG, fmt, ##args)
void OpcodeSample(int a ,int b){
    int c,d,e;
    c = a + b;
    d = a * b;
    e = a / b;
    LOGD("Hello It's c !%s\n", c);
    LOGD("Hello It's d !%s\n", d);
    LOGD("Hello It's e !%s\n", e);
    return;
}
```

通过 IDA Pro 对其分析。



左侧红色方框代表为 OpcodeSample 方法的操作码，右边为操作码对应ARM平台的指令集。我们要在左侧的操作码中取出一段作为 OpcodeSample 的定位特征码，选用 __android_log_print 方法调用指令集上下文，来确定特征码。

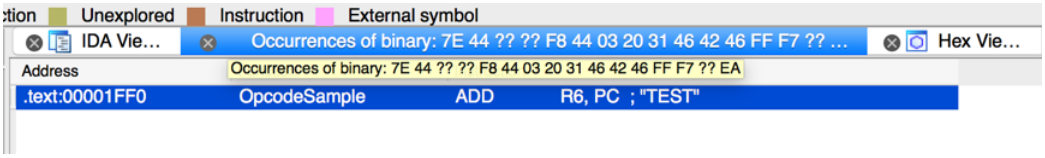
```
第一次取样: "03 20 31 46 42 46 FF F7 ?? EA"
```



Address	Function	Instruction	
.text:00001FF6	OpcodeSample	MOVS	R0, #3
.text:00002004	OpcodeSample	MOVS	R0, #3
.text:00002018	OpcodeSample	MOVS	R0, #3

通过第一次取样，查找结果有三处相似，再进一步分析。这次我们加入一个常量取样：

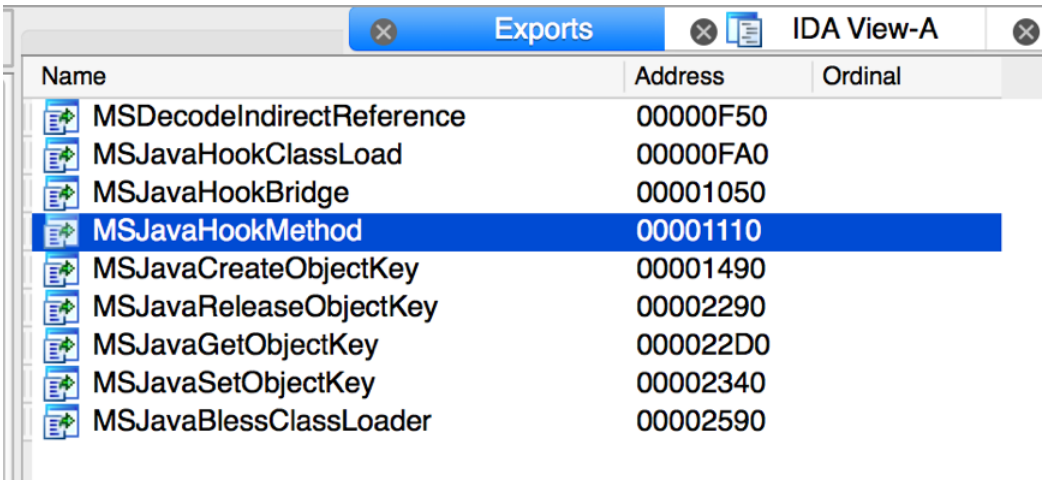
```
第二次取样: "7E 44 ?? ?? F8 44 03 20 31 46 42 46 FF F7 ?? EA"
```

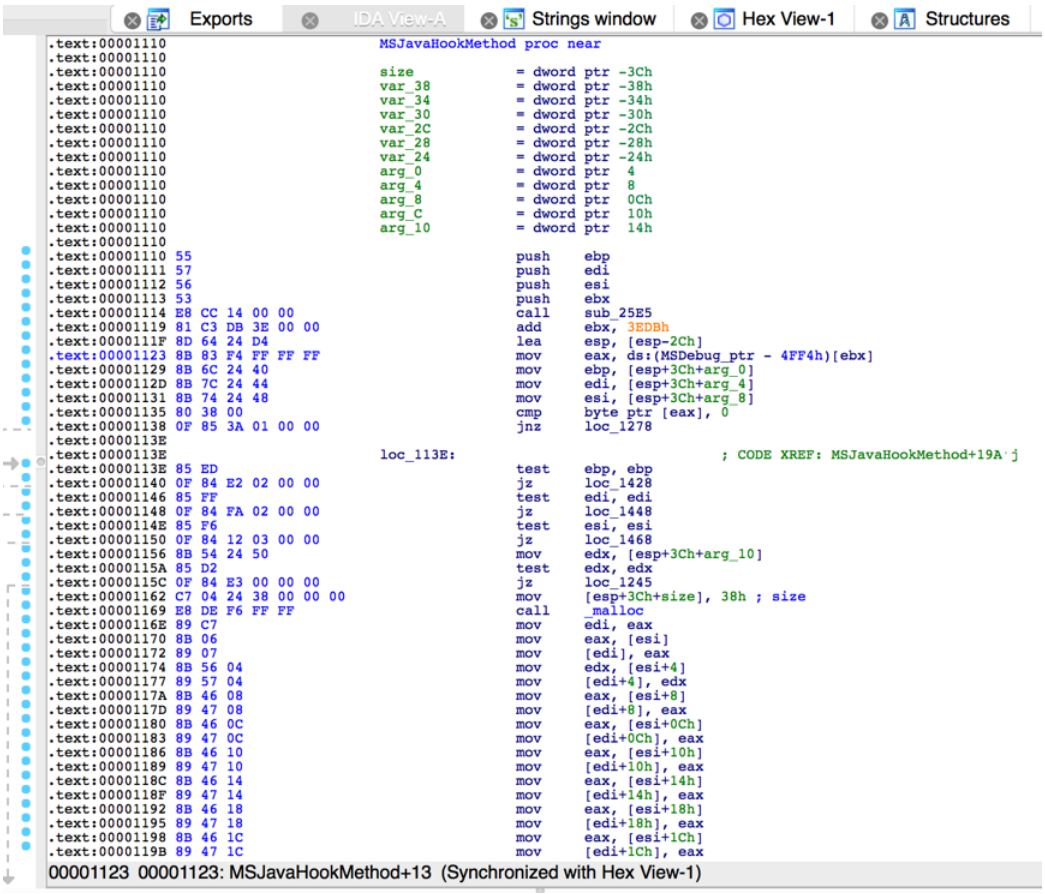


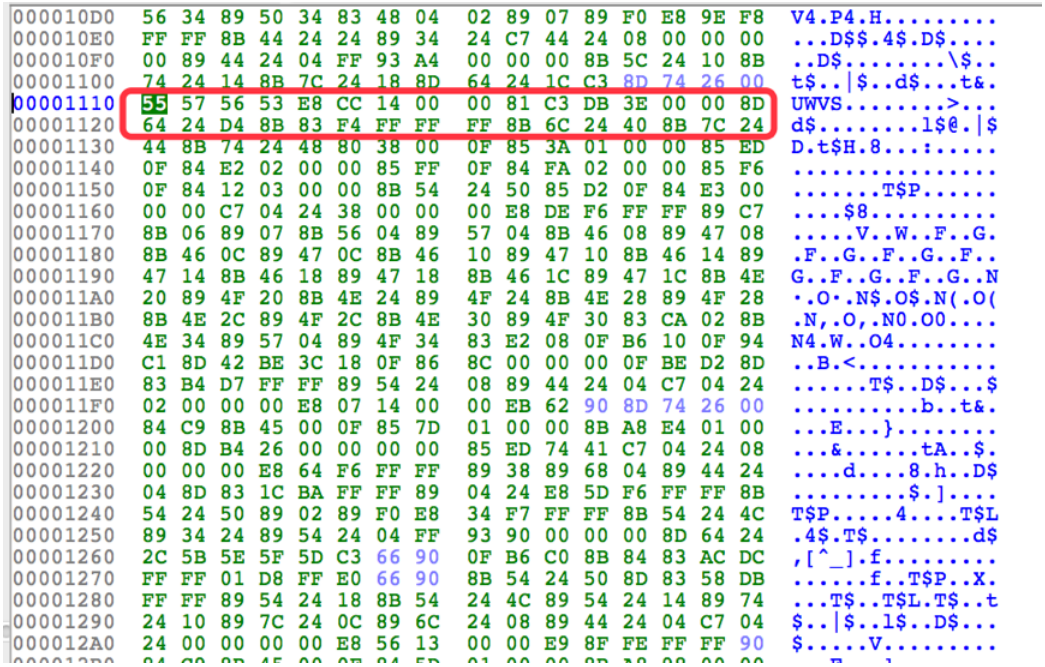
继而得出唯一特征码，到此，我们对特征码方法取样有了初步的了解。下面来把它转为实用的技能——动态加载式检测+特征码结合。

我们对 `libsubstrate-dvm.so` 中导出函数 `MSJavaHookMethod` 来精准定位。

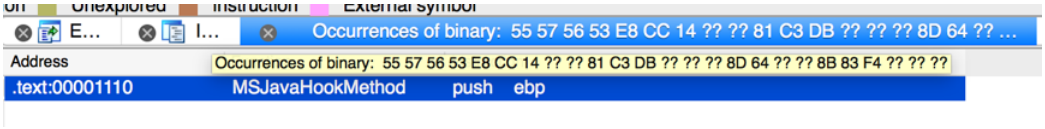
`IDA PRO` 导出函数表如图：







第三次取样："55 57 56 53 E8 CC 14 ?? ?? 81 C3 DB ?? ?? ?? 8D 64 ?? ?? 8B 83 F4 ?? ?? ??"



以上即为对 `Cydia Substrate` 的注入检测识别，通过检测 `/proc/self/maps` 下的加载 `so` 库列表得到各个库文件绝度路径，通过 `fopen` 函数将 `so` 库的内容以16进制读进来放在内存里面进行规则比对，采用字符串模糊查找来检测是否命中黑名单中的方法特征码。

总结

在安全对抗领域，相比攻击方，防守方历来处于弱势的一方。上文所提到的 `Xposed Installer` 和 `Cydia Substrate` 的检测也仅仅是保障App安全的手段之一。App安全的防御不应仅仅依赖于此，应该构建起整体的安全防御闭环，尽可能在所有已知的可能攻击点都追加检测，再配合代码加固，将防御代码隐藏。遗憾的是App防御代码隐藏再深也终究会被破解，仅仅依赖于客户端的防御显然是不足的。移动互联网领域的整体安全防御应该是走端云结合协作之道，共同防御，方能在攻防对抗中占据优势地位。

作者简介

- 礼赞，美团安全工程师，2016年11月加入美团。专注于二进制、移动端攻防相关工作，现负责美团Android移动安全组件的建设工作。
- 毅然，美团技术专家，2016年初加入美团。致力于美团配送App组的Android App crash解决工作、Android App性能优化、Android App反外挂、反爬虫。目前主导负责美团配送Android App移动安全相关建设。

招聘信息

美团集团安全部正在招募Web&二进制攻防、后台&系统开发、机器学习&算法等各路小伙伴。

我们想做的事情：

构建一套基于海量IDC环境下，横跨网络层、虚拟化层、Server 软件层（内核态/用户态）、语言执行虚拟机层(JVM/Zend/JavaScript V8)、Web应用层、数据访问层(DAL)的，基于大数据+机器学习的全自动人机识别与安全事件感知系统。规模上对应美团全线业务的服务器，技术栈覆盖了几乎大多数云环境下的互联网应用，数据规模也将是很大的挑战。

此外我们还关注全球互联网领域在企业安全建设方面的最佳实践，努力构建类似于Google的内置式安全架构和纵深防御体系，对在安全和工程技术领域有所追求的同学来说应该是一个很好的机会。

如果你想加入我们，欢迎将简历发至邮箱zhaoyan17#meituan.com，具体职位信息点击“ 岗位招聘 ”查看。

另外友情打个招聘：美团配送App团队，负责美团骑手、美团众包、美团跑腿等配送相关App的研发，涉及技术领域包括但不限于App的稳定性建设、App性能监控和优化、App动态化。对上述领域感兴趣的请联系 yulei10#meituan.com 。

美团数据平台Kerberos优化实战

作者: 鹏飞

背景

Kerberos 是一种网络认证协议，其设计目标是通过密钥系统为客户端、服务器端的应用程序提供强大的认证服务。

作为一种可信任的第三方认证服务，Kerberos是通过传统的密码技术（如：共享密钥）执行认证服务的，被Client和Server同时信任。KDC是对该协议中第三方认证服务的一种具体实现，一直以来都是美团数据平台的核心服务之一，在Hive、HDFS、YARN等开源组件的权限认证方面有着广泛的应用。该服务将认证的密钥事先部署在集群的节点上，集群或者新节点启动时，相应节点使用密钥得到认证。只有被认证过节点才能被集群所接纳。企图冒充的节点由于没有相关密钥信息，无法与集群内部的节点通信，从而有效的确保了数据的安全性、节点的可信赖性。

但随着平台业务的快速增长，当前线上KDC的处理能力不足和不能可靠监控的问题被凸显的日益严重：线上单台KDC服务器最大承受QPS是多少？哪台KDC的服务即将出现压力过大的问题？为什么机器的资源非常空闲，KDC的压力却会过大？如何优化？优化后瓶颈在哪儿？如何保证监控指标的全面性、可靠性和准确性？这都是本文需要回答的问题。从本次优化工作达成的最终结果上来看，单台服务器每秒的处理性能提升16倍左右，另外通过共享内存的方式设计了一个获取KDC各项核心指标的接口，使得服务的可用性进一步提升。

为方便大家，表1总结并解释了本文中后续会涉及到一些专业名词的简称：

名词缩写	名词解释
BDB	Berkeley DB，是一种简单、小巧、可靠、高性能的数据库
LDAP	一种轻量目录访问协议，英文全称是Lightweight Directory Access Protocol，一般都简称为LDAP
AS	全称Authentication Service，KDC中负责认证服务
KDC	Key Distribution Center，密钥分配中心，KDC在kerberos中通常提供AS和TGS两种服务
TGT	Client在从KDC那边获得Ticket之前，需要先获得这个Ticket的认购权证， 这个认购权证在Kerberos中被称为TGT：Ticket Granting Ticket
TGS	全称Ticket-Granting Service，KDC中负责授予票据服务
PREAUTH	KDC防止暴力、多次尝试破解密码的一种机制
IDC	互联网数据中心

表1 专业名词解释

图1为美团数据平台KDC当前服务架构，目前整个KDC服务部署在同一个IDC。

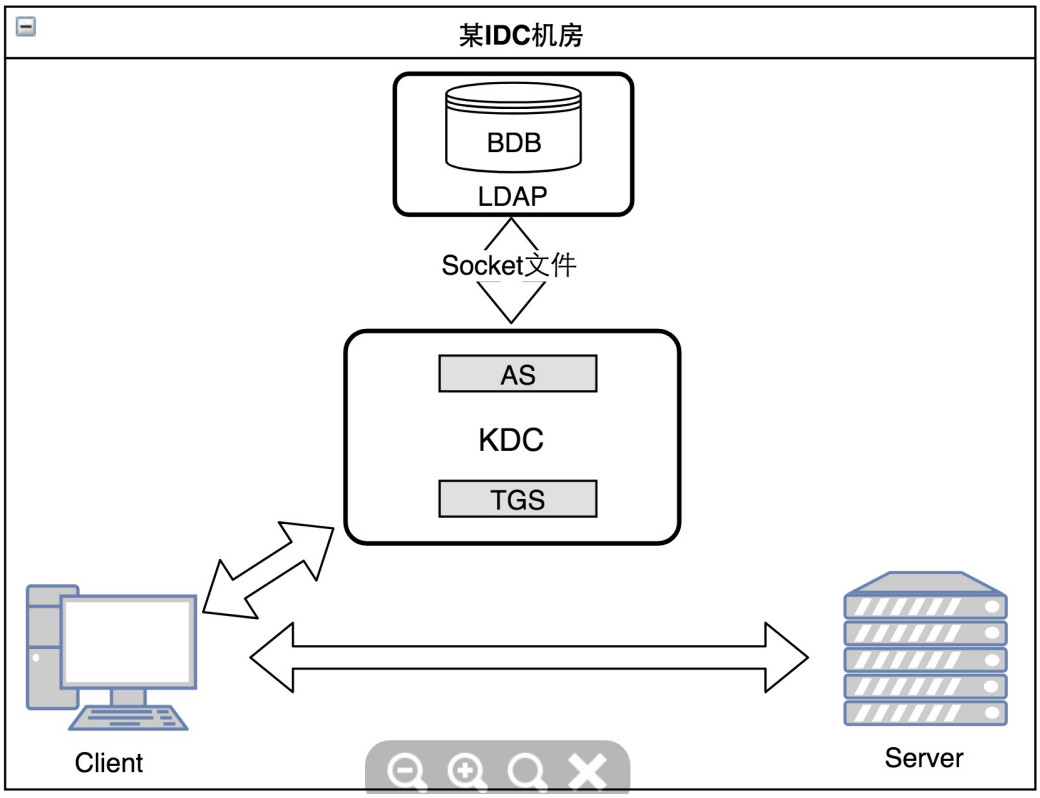


图1 KDC主体流程图

KDC原理介绍

Client、 KDC和Server在认证阶段主要有Client和KDC的AS、Client和KDC的TGS以及Client和Server的交互三个过程，下文将详细介绍三个过程的交互，其中图2中的步骤1和2、3和4、5和6分别对应下文的A、B、C三部分：

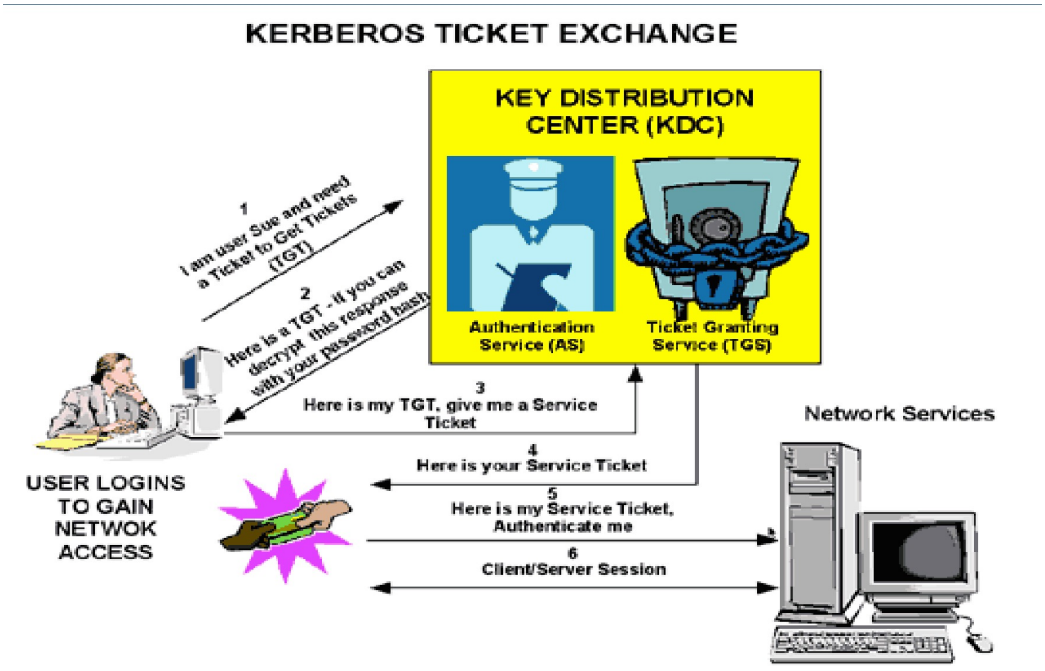


图2 KDC原理图

A. Client和AS的交互

1. 用户以明文的形式发送自己的信息、以及想要申请的TGT Principal（默认为KDC的TGT：krbtgt/REALM@REALM）等信息给AS服务；
2. AS服务验证该用户信息存在数据库中后，给客户端返回两大块信息：
   - 使用用户的密钥加密其申请的TGT和一个Session Key返回用户,用户得到加密信息后，使用自己密钥解密得到其申请的TGT和Session Key（后续都简称 SK_{CandK}）。
   - 以KDC自身密钥加密用户申请的TGT、SK_{CandK}、用户自己信息等；简称{TGT}K_{tgs}，该部分信息被Client保存在本地。

B. Client和TGS的交互

1. Client访问TGS获取访问网路中某一Server的ticket请求。Client端会把第一部分中保存在本地的{TGT}K_{tgs}和用SK_{CandK}加密的客户端信息发送KDC的TGS模块；
2. TGS收到请求后，检查请求Server存在数据库中后，用自己的密钥解密得到TGT中的SK_{CandK}，然后便可以解密得到用户的信息并验证其合法性；通过后，TGS会生成一个新的Session Key，简称SK_{CandS}；同时返回两部分信息：
   - 用SK_{CandK}加密的SK_{CandS}、能够访问Service的ticket等信息。
   - 用Service密钥加密的Client info、SK_{CandS}等信息，简称{ $T_{Service}$ }$K_{Service}$。

C. Client和Server的交互

1. Client拿到访问Service的ticket后，向Service发起请求，同时将两部分信息发送给Server：1）通过SK_{CandS}加密的Client info等信息。2）第二部分TGS返回客户端的{ $T_{Service}$ }KService；
2. Server端收到Client的请求信息后，用自己的密钥解密获取到$T_{Service}$信息，就能够解密SK_{CandS}加密的客户端信息，和$T_{Service}$中的客户端信息进行对比，通过后，整个KDC认证过程结束，Client和Service进行正常的服务通信。

主要优化工作

通过对KDC原理的分析，很容易判断只有前两部分才可能直接给KDC服务带来压力，因此本文涉及到的工作都将围绕上一部分的前两个环节展开分析。本次优化工作采用Grinder这一开源压测工具，分别对AS、TGS两个请求过程，采用相同机型（保证硬件的一致性）在不同场景下进行了压力测试。

优化之前，线上KDC服务启动的单进程；为最低风险的完成美团和点评数据的融合，KDC中keytab都开启了PREAUTH属性；承载KDC服务的部分服务器没有做RAID。KDC服务出现故障时，机器整体资源空闲，怀疑是单进程的处理能力达到上限；PREAUTH属性进一步保证提升了KDC服务的安全性，但可能带来一定的性能开销；如果线上服务器只加载了少量的keytab信息，那么没有被加载到内存的数据必然读取磁盘，从而带来一定的IO损耗。

因此本文中，对以下三个条件进行变动，分别进行了测试：

1. 对承载KDC服务的物理机型是否做RAID10；
2. 请求的keytab在库中是否带有PRAUTH属性；
3. KDC是否启动多进程（多进程设置数目和物理机核数一致）。（实际测试工作中进行了多次测试）

A. Client和AS交互过程的压测

表2为AS压测的一组平均水平的测试数据，使用的物理机有40核，因此多进程测试启动40个进程。

AS请求条件	测试结果
单进程，有PREAUTH，不做RAID	49
单进程，有PREAUTH，做RAID10	53
单进程，无PREAUTH，不做RAID	100
单进程，无PREAUTH，做RAID10	104
40进程，有PREAUTH，不做RAID	115
40进程，有PREAUTH，做RAID10	990
40进程，无PREAUTH，不做RAID	2000
40进程，无PREAUTH，做RAID10	1985

表2 AS压测

分析表2中的数据，很容易提出如下问题从而需要进一步探索：

1. 比较表2中第一行和第二行、第三行和第四行，主机做不做RAID为什么对结果几乎无影响?

该四组（测试结果为49、53、100和104所在表2中的行）数据均在达到处理能力上限一段时间后产生认证失败，分析机器的性能数据，内存、网卡、磁盘资源均没有成为系统的瓶颈，CPU资源除了某个CPU偶尔被打满，其他均很空闲。分析客户端和服务端的认证日志，服务端未见明显异常，但是客户端发现大量的Socket Timeout错误（测试设置的Socket超时时间为30s）。由于测试过程中，客户端输出的压力始终大于KDC的最大处理能力，导致KDC端的AS始终处于满负荷状态，暂时处理不了的请求必然导致排队；当排队的请求等待时间超过设置的30s后便会开始超时从而认证出错，且伴随机器某一CPU被打满（如图3）。 显然KDC单进程服务的处理能力已经达到瓶颈且瓶颈存在单核CPU的处理能力，从而决定向多进程方向进行优化测试。

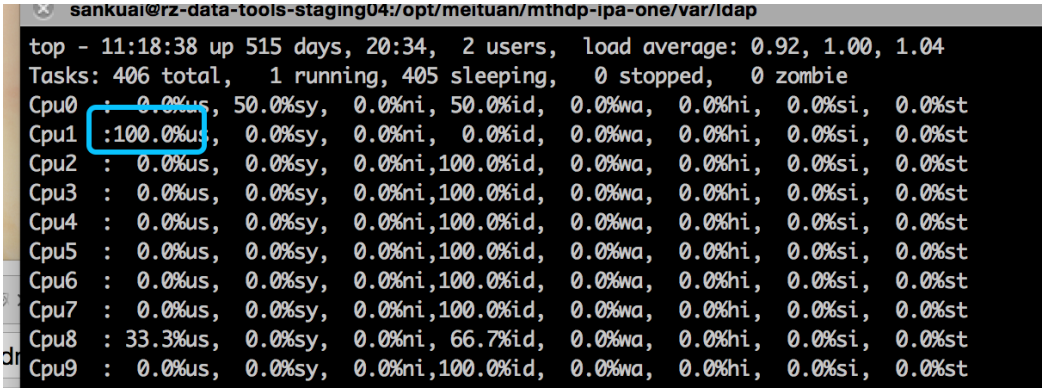


图3 单进程KDC打满某一CPU

图4为本次压力测试的一个通用模型，假设KDC单位时间内的最大处理能力是A，来自客户端的请求速率稳定为B且 B>A ； 图中黄色区域为排队的请求数，当某一请求排队超过30s，便会导致Socket Timedout错误。

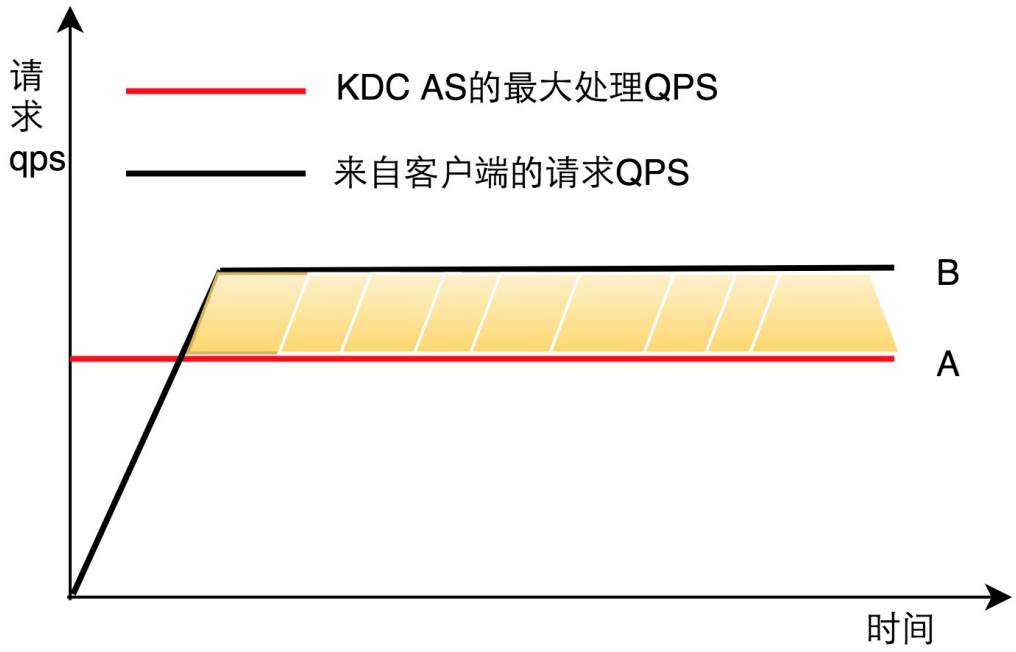


图4 AS处理能力和Client压力模型

2. 比较表2中第1和3行、第2和4行、第7和8行相比，为什么有PREAUTH属性的认证QPS大致是无该属性处理能力的一半?

如果Client的keytab在KDC的库中不带有PREAUTH这一属性，Client发送请求，KDC的AS模块验证其合法性之后返回正确的结果；整个过程只需要两次建立链接进行交互便可完成。如果带有PREAUTH属性，意味着该keytab的认证启动了Kerberos 5协议中的 pre-authentication概念：当AS模块收到Client的请求信息后；故意给Client返回一个错误的请求包，Client会“领悟到”这是KDC的AS端需要进行提前认证；从而Client获取自己服务器的时间戳并用自己的密钥加密发送KDC，KDC解密后和自身所在服务器的时间进行对比，如果误差在能容忍的范围内；返回给Client正确的TGT响应包；过程如图5所示。

图5 库中keytab有无preauth属性的区别

3. 根据对问题2的分析，表2中第5和7行的值的比例应该近似为1:2，为什么第5行的值只有115，结果和理论差距如此之大?

KDC的库中对客户端的keytab开启PREAUTH属性，客户端每认证一次，KDC需要将该次认证的时间戳等信息写到本次磁盘的BDB数据库的Log中；而关闭PREAUTH属性后，每次认证只需要从库中读取数据，只要给BDB数据库分配的内存足够大，就可以最大程度的减少和本次磁盘的交互。KDC40进程且开启PRAUTH，其AS处理能力的QPS只有115，分析机器性能的相关指标，发现瓶颈果然是单盘的IO，如图6所示。使用BDB提供的工具，查看美团数据平台KDC服务的BDB缓存命中率为99%，如图7所示：

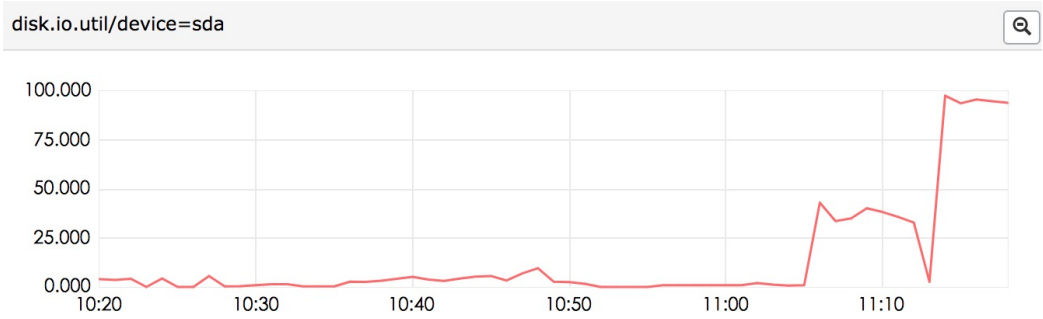


图6 无RAID多KDC进程服务器磁盘IO

```
[sankuai@rz-data-tools-staging04 ldap]$ db_stat -m
320MB 2KB 24B   Total cache size
1       Number of caches
1       Maximum number of caches
320MB 8KB       Pool individual cache size
0       Maximum memory-mapped file size
0       Maximum open file descriptors
0       Maximum sequential buffer writes
0       Sleep after writing maximum sequential buffers
0       Requested pages mapped into the process' address space
94M     Requested pages found in the cache (99%)
540     Requested pages not found in the cache
```

图7 美团KDC缓存命中率

4. KDC AS处理能力在多进程做RAID条件下，有无preauth属性，KDC服务是否有瓶颈？如果有在哪里？

经多次实验，KDC的AS处理能力受目前物理机CPU处理能力的限制，图8为有PREAUTH属性的CPU使用情况截图，无PREAUTH结果一致。

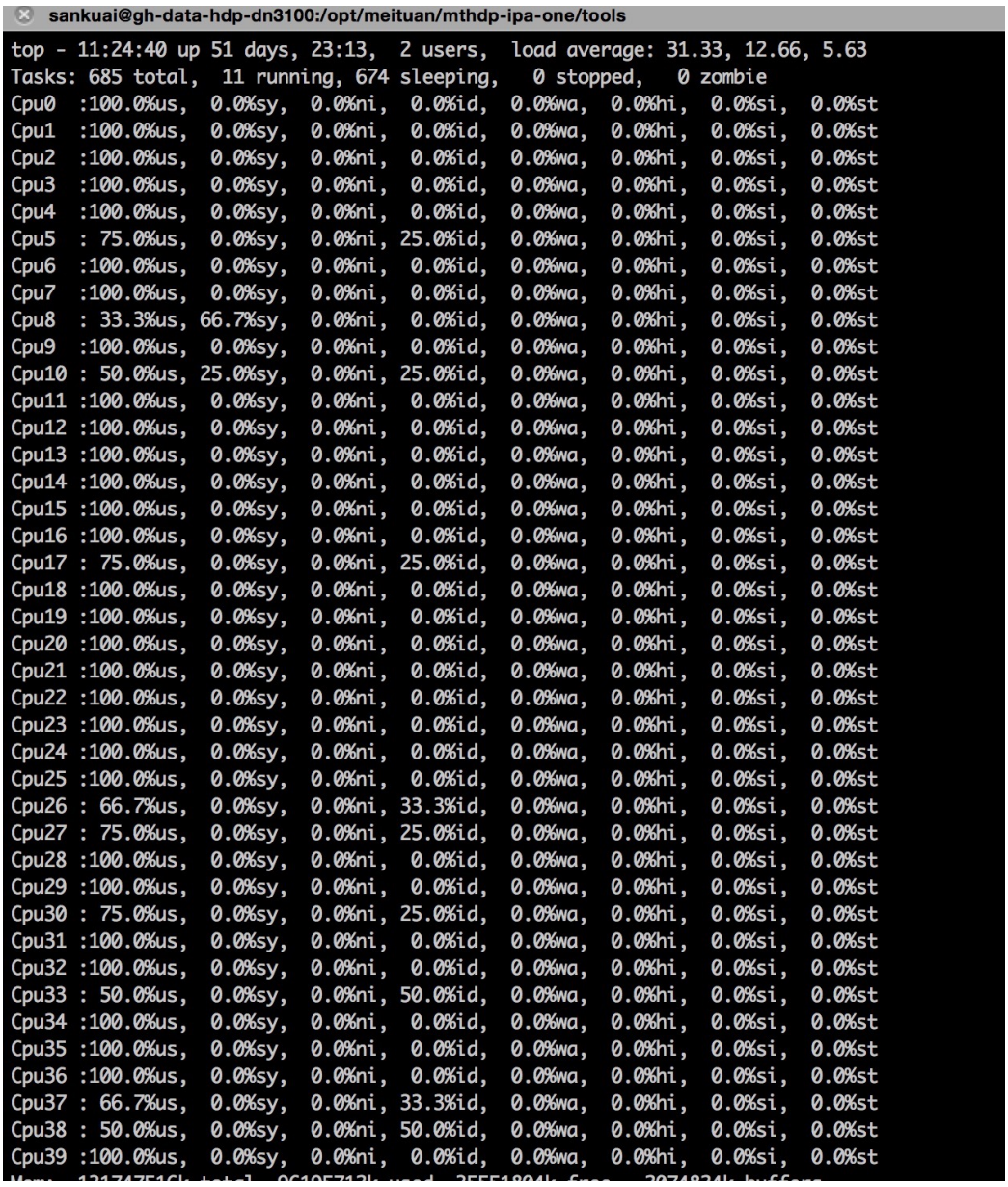


图8 40进程有PREAUTH，AS对CPU资源的使用情况

B. Client和TGS交互过程的压测

表3为TGS压测的一组平均水平的测试数据：

TGS请求条件	测试结果
单进程，有PREAUTH，不做RAID	63
单进程，有PREAUTH，做RAID10	58
单进程，无PREAUTH，不做RAID	66
单进程，无PREAUTH，做RAID10	61
40进程，有PREAUTH，不做RAID	1303
40进程，有PREAUTH，做RAID10	1342
40进程，无PREAUTH，不做RAID	1347
40进程，无PREAUTH，做RAID10	1339

表3 TGS压测

分析表3中的数据，可以发现KDC对TGS请求的处理能力和主机是否做RAID无关,结合KDC中TGS的请求原理，就较容易理解在BDB缓存命中率足够高的条件下，TGS的请求不需要和本次磁盘交互；进一步做实验，也充分验证了这一点，机器的磁盘IO在整个测试过程中，没有大的变化，如图9所示，操作系统本身偶尔产生的IO完全构不成KDC的服务瓶颈。KDC单进程多进程的对比，其处理瓶颈和AS一致，均受到CPU处理能力的限制（单进程打满某一CPU，多进程几乎占用整台机器的CPU资源）。从Kerberos的设计原理分析，很容易理解，无论KDC库中的keytab是否带有PREAUTH属性，对TGS的处理逻辑几乎没有影响，压测的数据结果从实际角度验证了这一点。

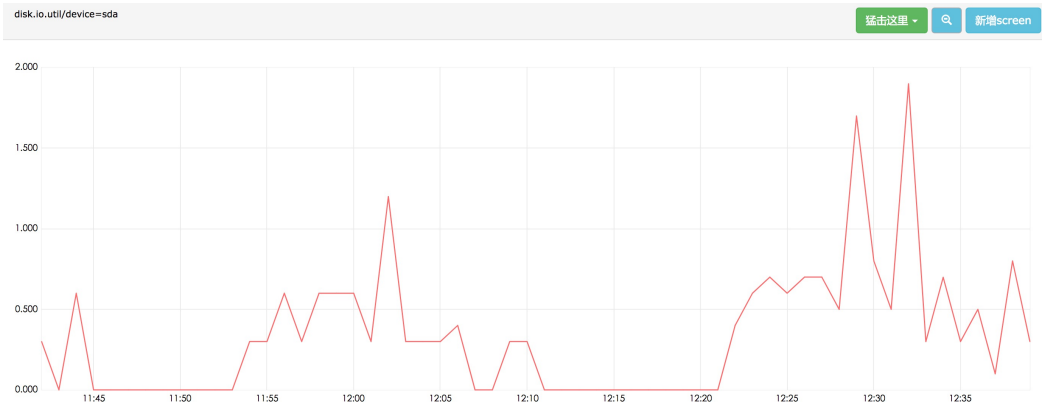


图9 TGS压测，IO资源的使用情况

C. 其他问题

Client和KDC的交互，支持TCP和UDP两种协议。在网络环境良好的情况下，两种协议的KDC的测试结果理论上和实际中几乎一致。但是在原生代码中，使用TCP协议，在客户端给KDC造成一定压力持续6s左右，客户端开始认证出错，在远未达到超时时限的情况下，Client出现了 `socket reset` 类的错误。KDC查看内核日志，发现大量 `possible SYN flooding on port 8089(KDC的服务端口). Sending cookies` ，且通过 `netstat -s` 发现机器的 `xxxx times the listen queue of a socket overflowed` 异常增高，种种现象表明可能是服务端的半连接队列、全连接队列中的一个或者全部被打满。主要原理如图10所示：

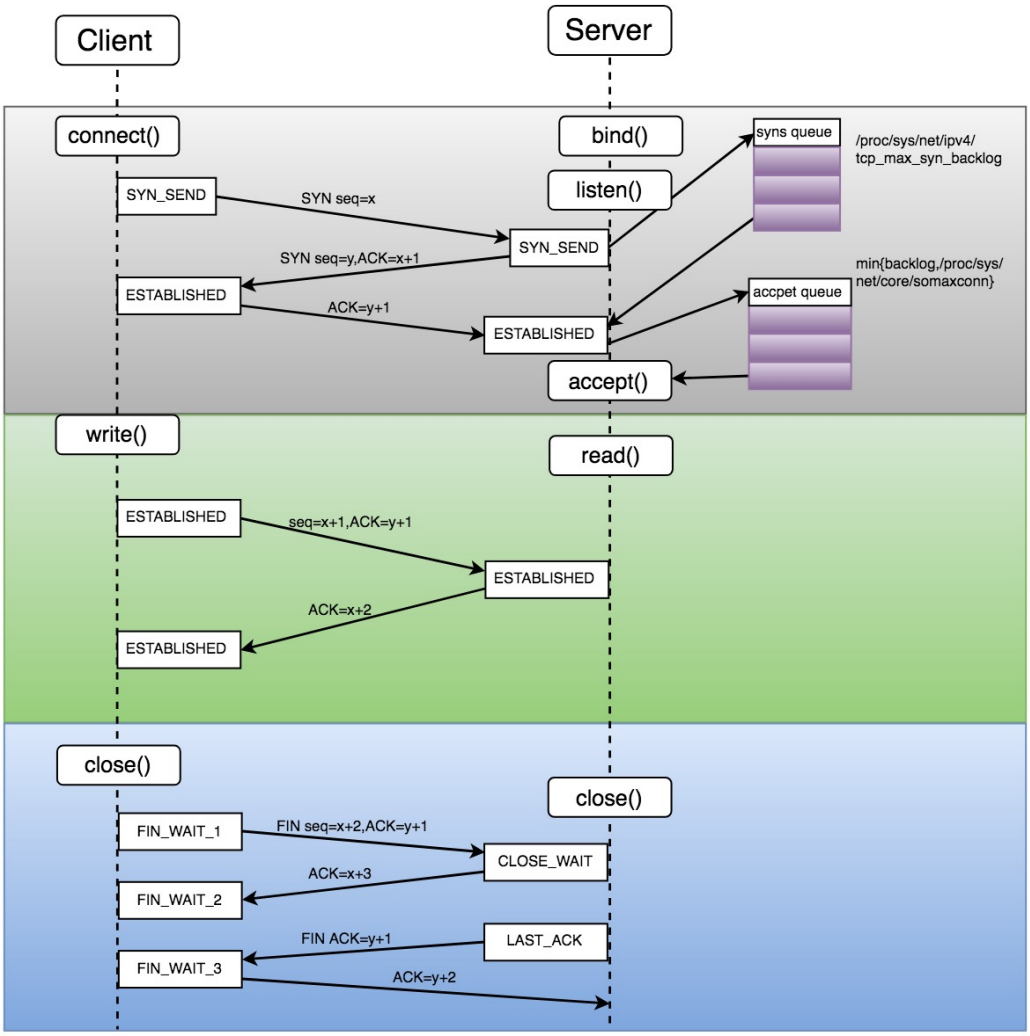


图10 半连接、全连接原理图

发现KDC服务所在服务器：半队列 `/proc/sys/net/ipv4/tcp_max_syn_backlog`为`2048` 。

全队列：1）系统参数 `/proc/sys/net/core/somaxconn＝65535` ，查看代码 `listen()` 函数的传入值为5。

故而判断TCP的瓶颈在于全队列，因此目标为将 `listen` 函数的第二个 `backlog` 参数变成可控可传入。

KDC可监控的设计和实现

开源社区对Kerberos实现的KDC完全没有对外暴露可监控的接口，最初线上的场景主要通过检索Log进行相关指标的监控，在统计服务QPS、各种错误的监控等方面，存在准确准确监控难的尴尬局面。为了实现对KDC准确、较全面的监控，对KDC进行了二次开发，设计一个获取监控指标的接口。对监控的设计，主要从以下三个方面进行了考虑和设计。

A. 设计上的权衡

1. 监控的设计无论在什么场景下，都应该尽可能的不去或者最小程度的影响线上的服务，本文最终采用建立一块共享内存的方式，记录各个KDC进程的打点信息，实现的架构如图11所示。每个KDC进程对应共享内存中的一块区域，通过n个数组来存储KDC n个进程的服务指标：当某个KDC进程处理一个请求后，该请求对监控指标的影响会直接打点更

新到其对应的Slot 数组中。更新的过程不受锁等待更新的影响，KDC对监控打点的调用仅仅是内存块中的更新，对服务的影响几乎可以忽略不计。相比其他方式，在实现上也更加简单、易理解。

2. 纪录每个KDC进程的服务情况，便于准确查看每个进程的对请求的处理情况，有助于定位问题多种情况下出现的异常，缩短故障的定位时间。例如：能够准确的反应出每个进程的请求分布是否均匀、请求处理出现异常能够定位到具体是某个进程出现异常还是整体均有异常。

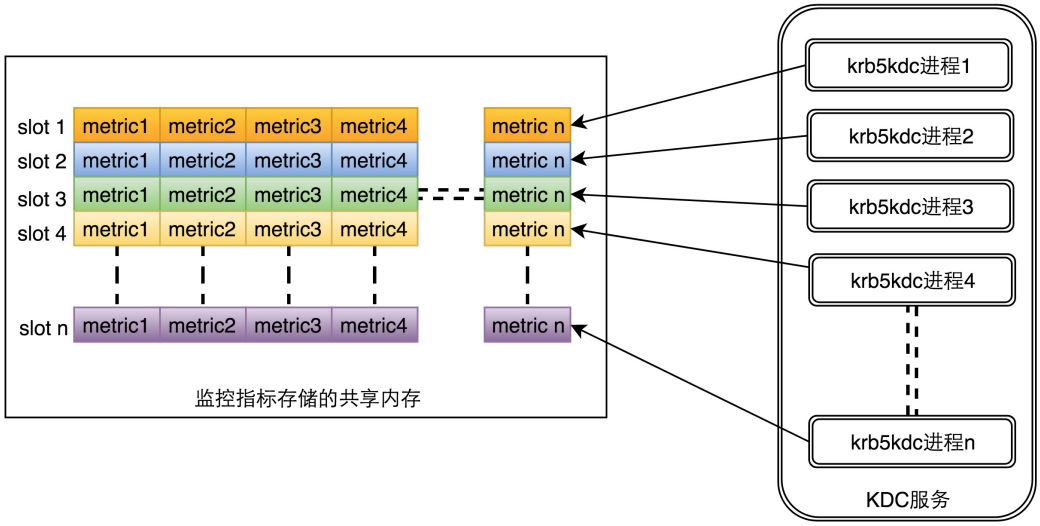


图11 KDC监控设计的整体架构

B. 程序的可扩展性

任何指标的采集都是随着需求进行变更的，如果程序设计上不具有良好的扩展性，会后续的指标扩展带来很大的困扰。第一版KDC监控指标的采集只区分请求的成功与失败两种类型，美团数据平台KDC库中所有的keytab都具有PREAUTH属性。根据上文可知，去掉PREAUTH属性后，AS请求的QPS能够提升一倍。后续随着服务规模的进一步增长，如果AS请求的处理能力逐步成为瓶颈，会考虑去掉PREAUTH属性。为了准确监控去掉PREAUTH属性这一过程是否有、有多少请求出现错误，需要扩展一个监控指标，因此有了KDC监控的第二版。整个过程只需要修改三个地方，完成两个功能的实现：

1. 添加指标 ；
2. 打点逻辑的添加。

整个修改过程简单明了，因此，该KDC监控程序的设计具有非常好的扩展性。图12为监控指标的罗列和注释：

```
enum kdc_stat_type
{
    /* for tcp */
    KDC_STAT_TCP_NEW_CONN = 0,    /* 累计新建连接数 */
    KDC_STAT_TCP_READ_ERR,        /* 累计read socket错误数：没有读到合法的请求包（长度错误等）*/
    KDC_STAT_TCP_WRITE_ERR,       /* 累计write socket错误次数*/
    KDC_STAT_TCP_PKT,             /* 累计收到的请求数 */
    KDC_STAT_TCP_PKT_ERR,         /* 累计包非法个数tcp_pkt - tcp_pkt_err = tcp_as_req + tcp_tgs_req */
    KDC_STAT_TCP_AS_REQ,          /* 累计AS请求个数*/
    KDC_STAT_TCP_AS_RES_PREAUTH,  /* 累计返回PRE-AUTH的AS请求数 */
    KDC_STAT_TCP_AS_RES_ERR,      /* 累计返回错误的AS请求数（不包含KRB5KDC_ERR_PREAUTH_REQUIRED） */
    KDC_STAT_TCP_TGS_REQ,         /* 累计TGS请求个数*/
    KDC_STAT_TCP_TGS_RES_ERR,     /* 累计返回错误的TGS请求数，不包含缺PREAUTH属性导致的错误 */
    KDC_STAT_TCP_TGS_RES_NO_PREAUTH_ERR,   /* 累计TCP协议缺PREAUTH属性导致的错误 */

    /* for udp */
    KDC_STAT_UDP_READ_ERR,        /* 累计read错误数：没有读到合法的请求包（长度错误等）*/
    KDC_STAT_UDP_WRITE_ERR,       /* 累计write socket错误次数*/
    KDC_STAT_UDP_PKT,             /* 累计收到的请求数 */
    KDC_STAT_UDP_PKT_ERR,         /* 累计包非法个数udp_pkt - udp_pkt_err = udp_as_req + udp_tgs_req */
    KDC_STAT_UDP_AS_REQ,          /* 累计AS请求个数*/
    KDC_STAT_UDP_AS_RES_PREAUTH,  /* 累计返回PRE-AUTH的AS请求数 */
    KDC_STAT_UDP_AS_RES_ERR,      /* 累计返回错误的AS请求数（不包含KRB5KDC_ERR_PREAUTH_REQUIRED） */
    KDC_STAT_UDP_TGS_REQ,         /* 累计TGS请求个数*/
    KDC_STAT_UDP_TGS_RES_ERR,     /* 累计返回错误的TGS请求数,不包含缺PREAUTH属性导致的错误 */
    KDC_STAT_UDP_TGS_RES_NO_PREAUTH_ERR,  /* 累计UDP协议缺PREAUTH属性导致的错误 */

    KDC_STAT_TOTAL_CNT,
};
```

图12 KDC监控指标及含义

C. 接口工具kstat的设计

获取KDC监控指标的接口工具主要分为两种：

1. 获取当前每个KDC进程对各个指标的累积值，该功能是为了和新美大的监控平台Falcon结合，方便实现指标的上报实现累加值和分钟级别速率值的处理；
2. 获取制定次数在制定时间间隔内每个进程监控指标的瞬时速率，最小统计间隔可达秒级，方便运维人员登陆机器无延迟的查看当前KDC的服务情况，使其在公司监控系统不可用的情况下分析服务的当前问题。具体使用见图13 。

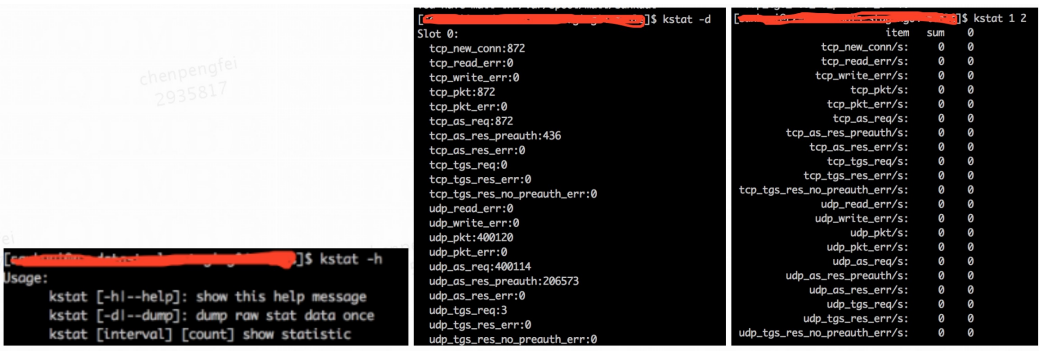


图13 kstat的使用帮助和两种功能使用样例

总结

通过本次对KDC服务的压测实验和分析，总结出KDC最优性能的调整方案为：

1. KDC服务本身需要开启多进程和以充分利用多核机器的CPU资源，同时确保BDB的内存资源足够，保证其缓存命中率达到一定比例（越高越好，否则查询库会带来大量的磁盘读IO）；
2. 选择的物理机要做RAID，否则在库中keytab带有PREAUTH属性的条件下，会带来大量的写，容易导致磁盘成为KDC的性能瓶颈。通过建立一块共享内存无锁的实现了KDC多进程指标的收集，加上其良好的扩展性和数据的精确性，极大的提高了KDC服务的可靠性。

相比原来线上单进程的处理能力，目前单台服务器的处理性能提升10+倍以上。本次工作没有详细的论述TCP协议中半队列、全队列的相关参数应该如何设定才能达到最优，和服务本身结合到一起，每个参数的变更带来的影响具体是啥？考虑到TCP本身的复杂性，我们将在未来的文章中详细讨论这个问题。

参考文档

- http://blog.csdn.net/m1213642578/article/details/52370705
- http://grinder.sourceforge.net/
- http://www.cnblogs.com/Orgliny/p/5780796.html
- http://www.zeroshell.org/kerberos/Kerberos-operation/
- http://blog.csdn.net/wulantian/article/details/42418231

作者简介

- 鹏飞，美团基础数据部数据平台大数据SRE组，离线计算组SRE负责人，2015年11月加入美团。

招聘信息

如果你对如何保证海量数据服务的稳定性、海量服务器大规模运维感兴趣，想亲历互联网大数据的爆发式增长，请和我们一起。欢迎加入美团数据平台大数据SRE组。有兴趣的同学可以发送简历到：chenpengfei#meituan.com。

镣铐之舞：美团安全工程师Black Hat USA演讲

作者: Ju Zhu

背景

2018年8月9日，全球顶级安全会议——Black Hat USA在美国拉斯维加斯的曼德勒海湾会议中心落下了帷幕，这场盛会在全球黑客心中几乎等同于“世界杯”和“奥斯卡”一样的存在。这场一年一度的盛会已经有着21年的悠久历史，也被公认为世界信息安全行业的最高盛会之一。

作为在国内安全领域拥有多年的实战经验和技术积累的安全团队，美团安全研究院再次受邀参加了本次盛会。在议题通过率不足20%的严苛筛选条件下，美团安全工程师Ju Zhu与小伙伴一起带着本次演讲的议题“Art of Dancing with Shackles: Best Practice of App Store Malware Automatic Hunting System（App Store恶意代码自动捕获系统最佳实践）”，荣登Black Hat USA 2018的国际舞台。

议题解读

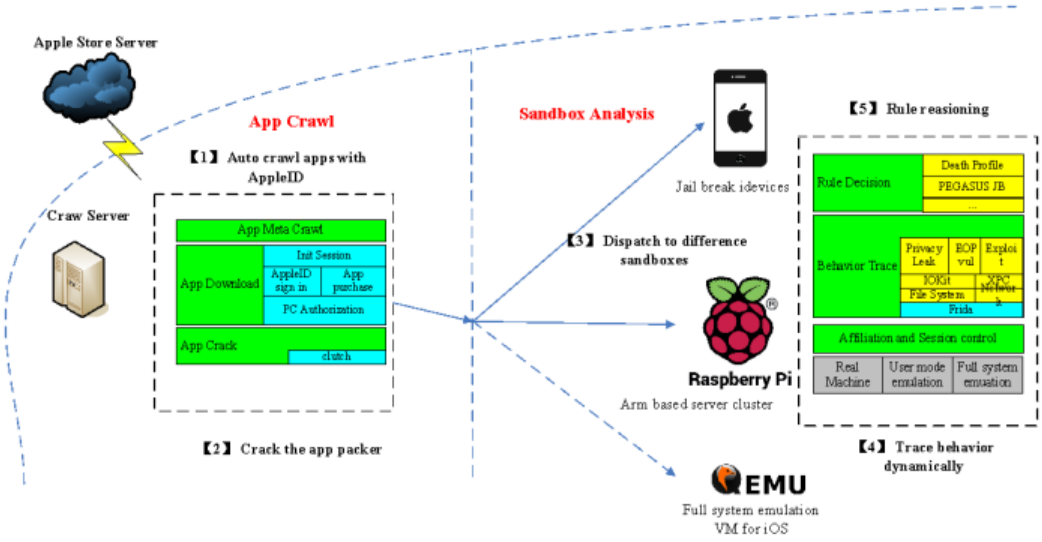
Apple的iOS系统是所有流行操作系统中最安全的系统之一，正因为如此，也是黑客重要的攻击目标和研究对象。虽然攻击难度大，但一旦成功，所获得的商业价值非常之高，所以iOS系统备受黑客“青睐”。

因为苹果的商业模式比较特别，而且iOS系统并非开源系统，同时Apple高度重视安全，所以对iOS系统进行了周密的安全设计，这使iOS系统的安全性远超其他操作系统。安全界对于大规模自动化捕获该平台的高级威胁，一直面临困难和挑战，同时当终端用户遭受真正的APT攻击（如PEGASUS）时，防御检测解决方案也无法获得足够的权限来检测深层攻击，甚至都无法获得足够的信息。

本议题正是在这个方向经过深入研究做了突破性的进展，成功设计出一套可大规模自动获取应用样本的Crawl系统，通过使用基于Raspberry Pi构建集群实现了低成本、可扩展的安全沙箱自动分析系统，最终实现了自动化收集样本并对样本进行自动化安全分析的APT攻击捕获及分析系统。

首先，我们先来看一下这个系统的整体架构。

系统整体架构



实际上，整个iOS恶意软件Hunt系统基本上分为两个不同的部分：

- 第一部分，是App Crawl系统，主要用于从App Store收集新发布或现有的应用程序。当然作为感染链源之一，还会收集来自第三方App Store甚至公共恶意软件存储库（例如Virus Total）的应用程序以增强我们的恶意软件数据库。除应用程序外，其他潜在的恶意文件类型（如Profile）也是我们的收集目标，（可以参考我在BlackHat Asia 2018中关于”野外iOS Profile攻击“的名为“Death Profile”的文章）。
- 另一部分，是沙盒分析系统，主要是动态跟踪应用程序行为，并根据规则决策引擎关联行为日志以给出最终结果。实际上，沙盒系统包含不同的类型，包括基于Frida的iOS真实设备、ARM服务器的用户模式仿真（例如Raspberry Pi系统）以及完整的系统仿真VM。

系统构成

具体来说，整个系统主要由五个模块构成。

1. 自动Crawl系统：自动化爬行及抓取各App应用市场的应用程序，包括App Store以及其他第三方市场，本系统中通过逆向分析成功的实现了自动化的用户登录、购买及下载应用程序。
2. App Crack系统：解密从App Store下载的应用程序，方便沙盒进行动态行为分析。
3. 沙盒分析系统：突破传统基于真机（iOS设备）沙盒的系统设计，创新的使用了基于Raspberry Pi模式和QEMU模式，低成本、可扩展的集群方式来实现动态监控应用程序的运行行为，例如File、Network、XPC、IOKit和Profiled等。
4. 动态跟踪行为系统：主要用来收集沙盒系统中所运行样本的各种监控行为日志。
5. 决策引擎系统：基于开源的Nools系统，实时或非实时地根据监控日志，来判断样本行为。

那么，它们是怎么有效的运转起来的呢？

系统运行流程

- 首先，通过自动化爬虫系统构建相应的登录、购买、下载操作，从iTunes服务器抓取应用程序，并发送给Crack系统。之后Crack系统将解密Apple的DRM，并生成可在越狱设备和模拟器上运行的IPA文件。

- 然后，构建IPA运行环境暨沙盒分析系统，我们引入了两个解决方案，第一个是传统的在真正越狱设备上分析这些应用程序；第二个是创新的使用了基于Raspberry Pi的模拟器集群来运行并分析应用程序。
- 最后，使用基于开源的Frida框架，经过定制化的开发，动态跟踪每个IPA应用程序的行为，再通过决策引擎检查IPA是否为恶意应用，是否可能存在APT攻击。

下面，我们将基于各个模块的分解来详谈它们的运作模式。

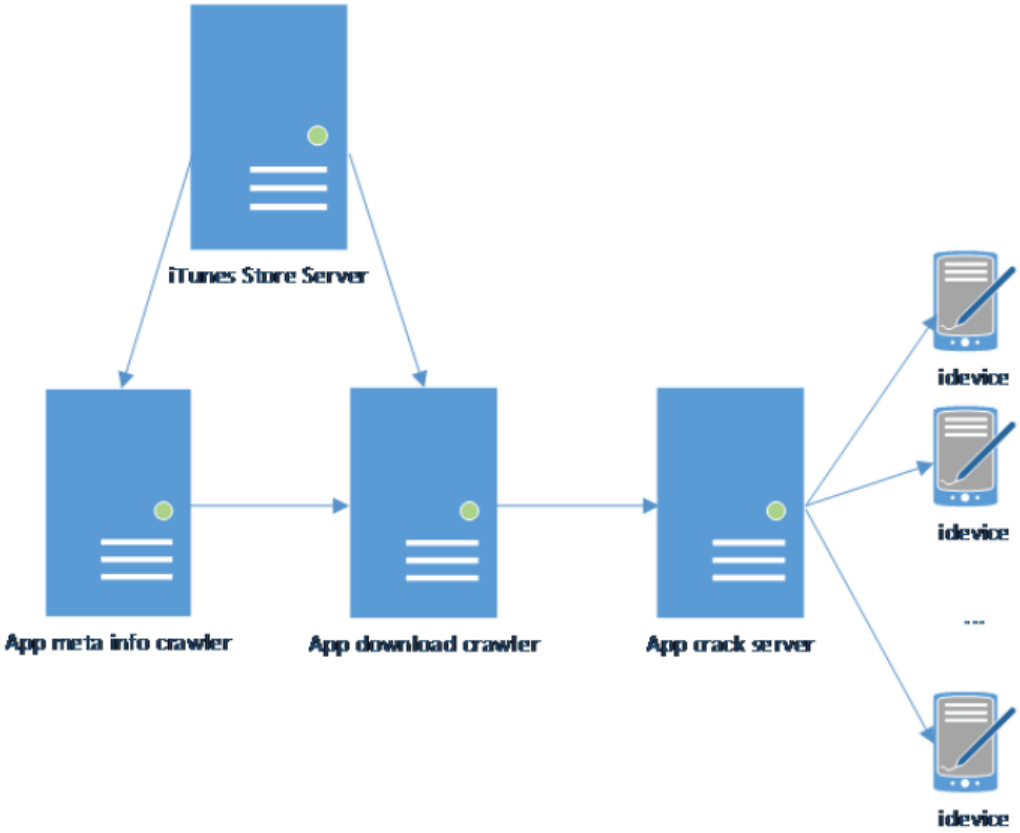
首先，App Store应用程序的Crawl基本可以理解为iTunes协议的灰盒逆向工程。

经过研究，我们发现在PC主机上通过iTunes实现App Store Crawl的基本步骤包括以下几个方面：

- 第一步，是抓取目标应用的Meta数据，例如名称、类别、大小等等。
- 第二步，是使用Apple ID登录，购买产品，使用iTunes授权PC端，以满足应用下载的要求，并将应用保存到本地磁盘。这里，我们必须使用很多技巧来征服App Store的反爬机制。
- 最后一步，是破解下载的应用程序。由于App Store上的所有应用程序都由Apple打包，这显然会阻止厂商基于安全方面的动静态分析，因此需要将目标应用程序的运行时内存转储为普通代码。

因此，基于上面的流程，我们可以设计成以下架构。

自动Crawl & Crack系统架构



从架构图来看，该系统实现的功能，包括应用程序Meta信息Crawl、Apple ID登录、PC授权、iOS设备授权、IPA签名和安装后的Crack。它实际上是一个基于iTunes Store应用程序的自动系统。

App Meta信息Crawler负责获取应用程序详细信息，包括下载URL和价格信息。应用下载抓取工具可以通过这些网址自动下载应用。然后，这些应用程序将发送到每个越狱设备进行解密，这将用于以后的静态和动态分析。

自动Crawl系统

对于Crawl系统，我们可以分为三个部分来运作。

1. App Meta信息Crawler：App Store是有区域限制的，即区域A的Apple ID无法下载区域B的应用程序。因此，针对不同区域，我们设计了不同的Spider。而获得的Meta信息，包含了App ID、下载地址、图标和其他一些基本信息。
2. App下载Crawler：通过逆向分析多个二进制文件及通讯协议，构造Apple ID的登录及购买请求，可以自动化下载与“从iTunes客户端下载的IPA文件”相同的IPA文件。
3. 导入DRM数据：上面下载的IPA文件，实际上是不能直接安装运行的，因为还缺少一个Sinf文件，它是一个包含授权等信息的DRM数据文件。对于Apple来说，它们只为每个应用程序保留了一份Copy。当用户购买App时，服务器将动态生成DRM信息，并将其放入应用程序购买的响应数据中发送回来。之后iTunes或者App Store将负责把DRM数据重新打包到IPA文件中。因此，我们仅需简单地将之前获得“Sinf数据”保存并下载到IPA文件中即可。

我们都知道，从App Store下载的App是加密的。这样不利于我们使用越狱设备和模拟器来分析行为，所以还需要对IPA进行解密。下面我们说一下，Crack系统的技术要点。

App Crack系统

如果用户的帐户从来没有在iOS设备上登录过，则它购买的App是无法在该设备上运行的，即DRM保护。如果在一台设备上，用户登录了自己的帐户，Apple会认为用户授权此设备，而使用该帐户购买的一切应用程序则可安装、也可运行。但是我们需要让这一切变得自动化。

通过逆向“设置”程序，我们发现“StoreServices.framework”是用来管理账户信息的，最后我们做了一个Tweak，并配合Undocument API实现了Apple ID登录过程。

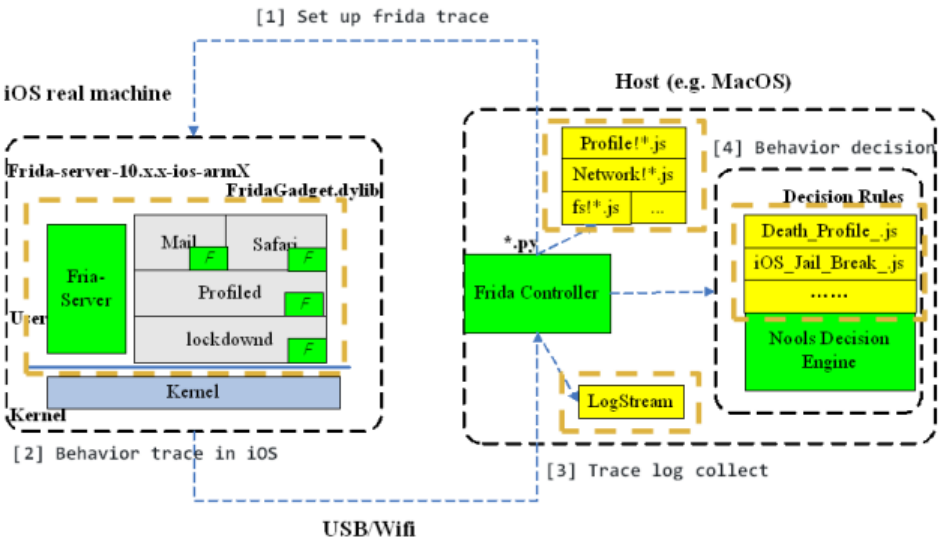
我们有了大量待分析的样本，下面的工作就是静态和动态分析。业界对于静态分析已经非常成熟，比如MachOView等等，这里我们就不多介绍了。而动态分析，目前主要以基于Frida的系统居多。

Frida是一个功能强大且便携的Hook系统，支持移动（例如iOS和Android）和PC系统（例如MacOS）。更重要的是，它允许在没有配置和编译的情况下根据脚本（例如JavaScript）控制Hook点。所以说，它是目前最流行的动态分析框架系统。当然它们都必须依赖真机设备。

接下来，我们来介绍一下传统的基于真机（iOS设备）的沙盒系统。

沙盒分析系统

传统的基于真机（iOS设备）沙盒系统



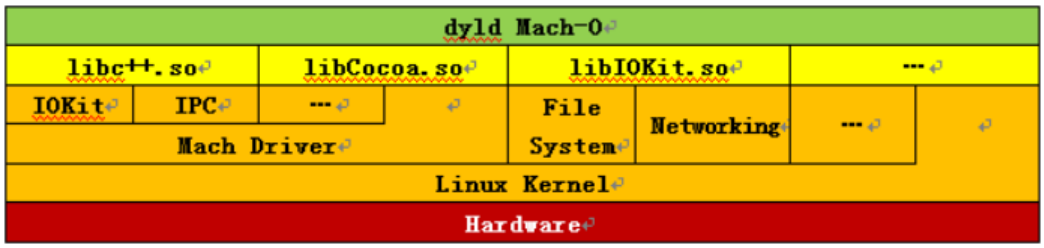
从上图来看，这种基于Frida实现真机（iOS设备）沙盒系统的工作流程主要包括以下几个方面：

- 首先，给iOS设备配置Frida，目的是为了进行行为跟踪；
- 然后，Frida控制器模块将在iOS设备上触发样本运行，或者其他任何操作（例如：安装配置文件，使用浏览器访问网站等），并跟踪感兴趣的系统行为；
- 最后，将行为日志收集到主机端，该日志将成为决策引擎系统的输入，它会根据需要，实时或非实时地判断样本行为。

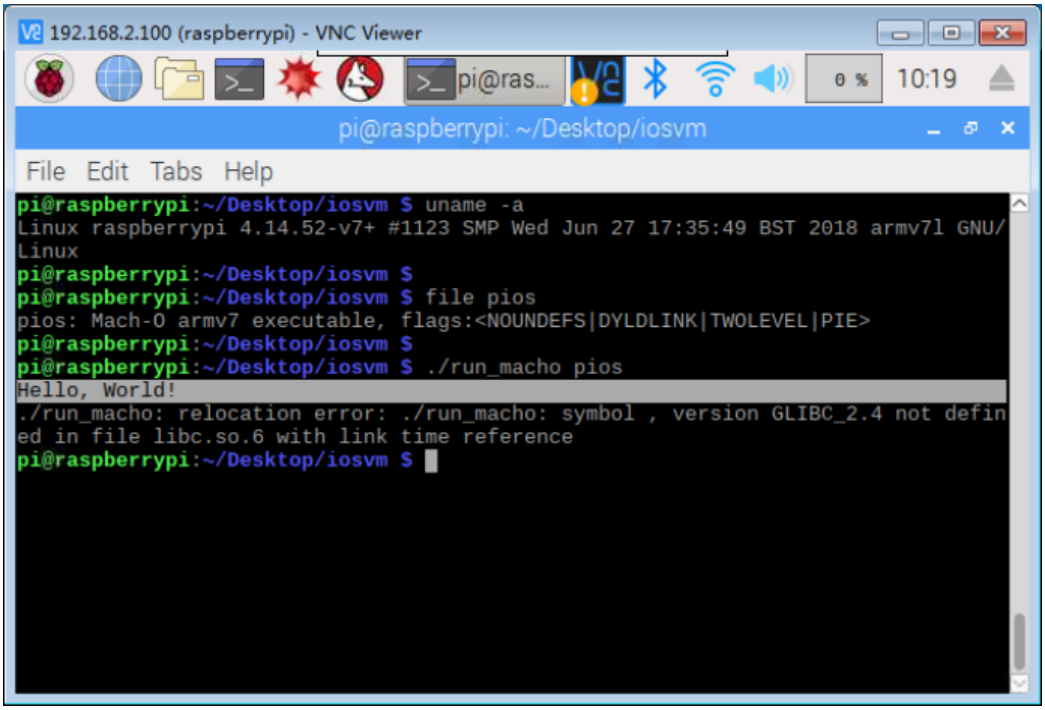
虽然Frida一直是App动态检测的主流，但是如果我们需要检测大量样本或者大量Case时，则会出现严重瓶颈，因为面临大量的真机（iOS设备）投入，而且成本、扩展性都是致命问题，所以我们创新的使用了低成本Raspberry Pi来替代它，并成功的实现了虚拟化、集群化。

基于Raspberry Pi的iOS虚拟机

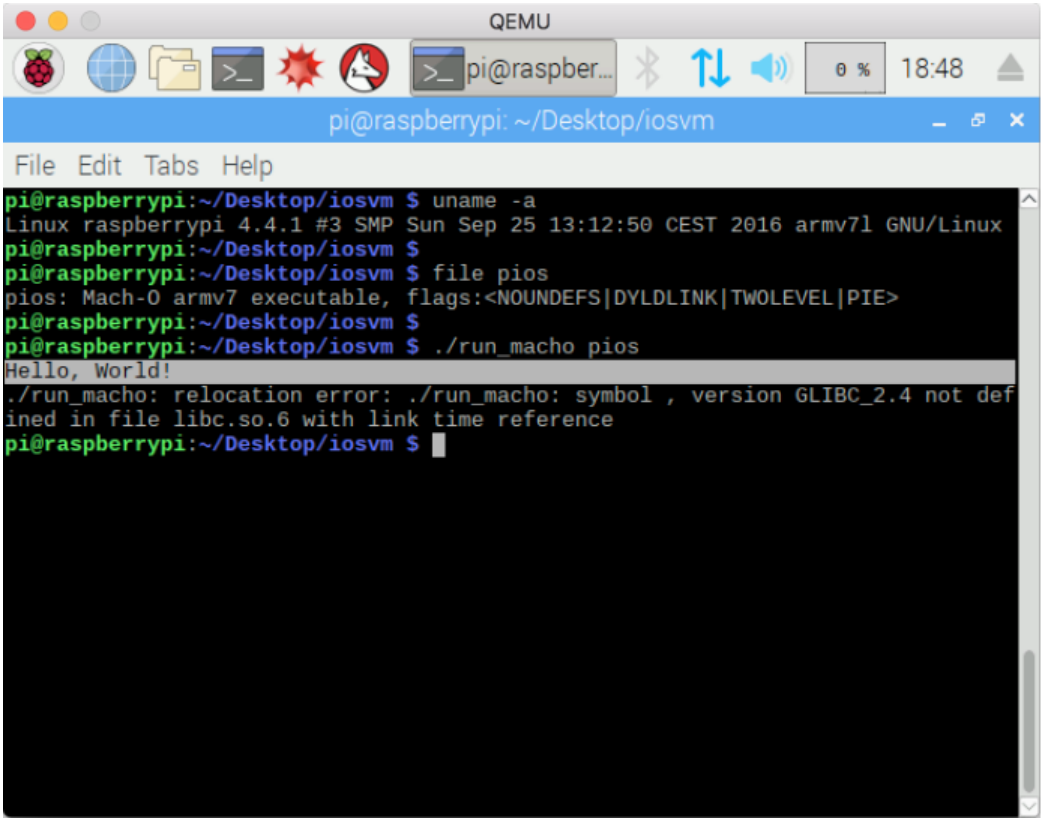
在虚拟化方面，我们实现了一个动态加载器，它可以加载iOS可执行文件，并重新实现了System Library和Framework，以保证iOS可执行文件能够正常运行等等。



这样我们就可以轻松动态监控Mach-O的行为，并将这些日志提交给决策引擎，以确定该应用程序是否为恶意软件，甚至是否是APT攻击的一个Chain等等。



如果希望利用现有的服务器来运维，我们也可以将其移植到QEMU中运行。



通过这种“低成本硬件仿真器”的设计，每天都可以自动化扫描大量的样本，从而节省了成本，并提升了可扩展性，提高了样本的检测效率。

这种高效的沙盒分析系统，必然会产生大量的分析日志，所以，我们需要一种高性能、高实时性的规则决策引擎系统来做最后的判断处理。

决策引擎系统

Nools是一个基于Rete并使用JavaScript实现的规则引擎推理系统。它可以支持连续日志输入时的实时判断模式，而且用其编写的决策规则，具有强灵活性和可移植性，使得我们对于样本的检测获得了高可用性。

总结

一直以来，业界对于“iOS大量样本检测实现自动化Hunt高级威胁”都没有很好的实践，而我们已经证明了基于自动抓取、安全沙箱自动分析系统以及iOS虚拟化的高级威胁Hunt系统的可行性。而这样大量的样本检测Case和日志，也为以后我们引入AI系统提供了必要条件。

关于美团安全

美团安全部的大多数核心人员，拥有多年互联网以及安全领域实践经验，很多同学参与过大型互联网公司的安全体系建设，其中也不乏全球化安全运营人才，具备百万级IDC规模攻防对抗的经验。安全部也不乏CVE“挖掘圣手”，有受邀在Black Hat等国际顶级会议发言的讲者，当然还有很多漂亮的运营妹子。

目前，美团安全部涉及的技术包括渗透测试、Web防护、二进制安全、内核安全、分布式开发、大数据分析、安全算法等等，同时还有全球合规与隐私保护等策略制定。我们正在建设一套百万级IDC规模、数

十万终端接入的移动办公网络自适应安全体系，这套体系构建于零信任架构之上，横跨多种云基础设施，包括网络层、虚拟化/容器层、Server 软件层（内核态/用户态）、语言虚拟机层（JVM/JS V8）、Web 应用层、数据访问层等，并能够基于“大数据+机器学习”技术构建全自动的安全事件感知系统，努力打造成业界最前沿的内置式安全架构和纵深防御体系。

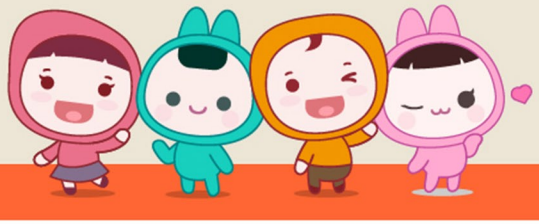
随着美团的高速发展，业务复杂度不断提升，安全部门面临更多的机遇和挑战。我们希望将更多代表业界最佳实践的安全项目落地，同时为更多的安全从业者提供一个广阔的发展平台，并提供更多在安全新兴领域不断探索的机会。

招聘信息

美团安全部正在招募Web&二进制攻防、后台&系统开发、机器学习&算法等各路小伙伴。如果你想加入我们，欢迎简历请发至邮箱 zhaoyan17@meituan.com

具体职位信息可参考这里： https://mp.weixin.qq.com/s/ynEq5LqQ2uBcEaHCu7Tsiw

美团安全应急响应中心MTSRC主页： security.meituan.com



扫码关注技术团队
微信公众号

tech.meituan.com
美团技术博客

CODE A BETTER LIFE
一 行 代 码　亿 万 生 活