人工智能 架构 无线大前端
大数据 质量 运维等

携程技术中心 出品

携程技术

2018 年度合辑

Ctrip 携程

ctrip tech

序

2019 年已悄然来到我们身边，回首过去的 2018 年，携程 GMV 达到 6900 亿元，首次成为全球 OTA 行业第一；线下门店超 7000 家，国内城市覆盖率达到 80%；海外平台月度活跃用户量已达到 9000 万，携程全品牌月活跃用户数超过 2 亿。

在背后支撑和驱动业务快速发展的，正是数千名日夜奋战的携程技术人。携程技术工程团队逐步成长为成熟和有业界影响力的互联网技术团队，技术已成为推动携程业务发展的持续源动力。

2018 年是携程技术人繁忙的一年：

- 后端技术方面，已初步实现从.Net 到 Java 技术栈的转型，分布式服务框架、消息队列、缓存、服务监控、DR 支持等框架和中间件日臻成熟；
- 大数据作为日常业务决策和产品优化的必备工具和基础设施，从底层数据平台到上层应用均发展迅速，『用数据说话』已成为技术和业务同事的共识；
- 人工智能尤其深度学习作为当今业界技术潮流，携程当然也没有落下，持续的探索和落地 AI 技术，在个性化推荐、智能客服、图像识别和智能运维等业务场景发挥越来越大的作用；
- 无线大前端领域，无论 React Native 经过两年多的打磨成长为替代传统 Native 和 Hybrid 的无线开发技术，还是 NodeJS 技术栈的成熟和应用数的高速增长，都表明携

程在业界快速发展的大前端技术潮流中准确把握了方向，并且脚踏实地的实现了落地；

- 得益于运维基础设施和私有云技术的日益成熟，DevOps 在携程早已不是概念，而是携程技术人每天的日常操作；
- 风控作为金融服务中的核心技术，今年我们实现了携程和去哪儿风控系统的统一，并且成功输出到携程收购的英国机票搜索引擎 Skyscanner。
- 2018 年也是携程尝试技术出海的一年，海外呼叫中心和海外公有云应用部署都标志着携程技术人会成为业务出海的坚定后盾。

技术从来不是闭门造车，这一年我们也在积极和业界通过各种渠道进行交流，无论是微信公众号、专题技术沙龙，还是举办携程技术峰会，以及参加各类业界技术会议，我们会一直以开放和充满热情的心态和小伙伴们探讨技术。我们相信不同的技术都有其特定的应用场景，在交流中碰撞出新思路的火花，是我们进行技术交流的目标。

携程目前使用了各个技术领域的大量开源技术，例如 Redis、Dubbo、CAT、Kubernetes、React Native 等，我们也曾开源 Apollo 等『GitHub 高星』作品。作为技术交流的终极形式，我们期待未来能更多以开放源码的形式回馈技术社区，据悉 3 月左右 CRN 框架核心部分即将开源。Talk is cheap, show me the code.

这本书作为 2018 年携程技术的结晶，集合来自携程技术中心微信公众号全年度的重要技术文章，涵盖了从无线大前端到后端架构，从大数据到 AI 应用，从研发质量到 DevOps 几乎所有主流的互联网技术方向。总共 69 篇，细分为 8 个领域：人工智能、架构、无线大前端、

大数据、质量、运维、数据库、风险控制。

欢迎大家针对书中的技术问题深入探讨，如果发现问题或者有更好的想法，期待及时向我们提供反馈。

新的一年我们会继续在技术路上狂奔，欢迎业界小伙伴们通过关注携程技术中心微信公众号，参加携程主办的专题技术沙龙和会议，大家都能有更多的成长和收获。

携手致远，百战功程！

携程技术副总裁 李小林

2019.1.19 于上海

“携程技术”微信公众号（ID:ctriptech）

分享来自携程技术人的一手干货文章，发起技术活动，发布热招职位，

和技术圈小伙伴一起学习成长~

目录

人工智能篇

机器学习在酒店呼叫中心自动化中的应用

[作者简介]周振伟，携程数据智能部数据科学工程师，同济大学硕士，主要承担酒店服务领域的数据分析和挖掘工作。

无论是出门旅游还是商务出行，在外能有一个舒适的住处，往往都是首先要解决的问题。OTA提供的酒店预订功能无疑为此提供了巨大的便利。

打开携程 APP，看中一家不错的酒店下单后，会有一个等待酒店确认的过程。携程将用户预订的消息发送给酒店，酒店进行查房，确认是否有空余房间，然后回复携程，再由携程通知用户确认结果。这个过程通常在半小时到一小时内完成，很多时候只需十分钟，在这背后，携程的呼叫中心起到了重要的作用。

面对每天全国海量的出行和预订需求，避免用户长时间的等待，加上与酒店相关的各种复杂事宜，呼叫中心的工作一直在高度紧张和繁忙的氛围中进行。然而随着业务量的不断增长，完全靠人力完成这些工作效率是非常低的。在大数据和人工智能时代，我们自然会想到用数据的力量代替一部分人力工作，即实现和提升流程的自动化，提高工作效率。

本文以订单确认智能外呼为例，分享携程在提升呼叫中心自动化方面所做的工作，介绍携程是如何利用机器学习技术，改造呼叫中心外呼流程的。

一、呼叫中心职能

携程呼叫中心的职能主要包含以下四部分：

（1）订单确认前处理：从用户下单到确认订单为止的过程，称为前处理。确认订单是指用户下单后酒店确认是否可以正常入住，然后携程告知用户确认结果。除一部分房型可以直接确认用户不用等酒店回复以外，大部分房型都需要由酒店亲自确认。这部分房型只要有用户下单就必然经历这个过程，因此前处理是携程呼叫中心工作量最大的事务之一。

（2）订单确认后处理：从确认订单到用户入住和离店为止的过程，称为后处理。对用户来说，订单确认后只需到那一天到达酒店入住即可。对携程来说还有很多事宜需要和酒店沟通，例如入住人或时间变更、离店后的审核结算等。此外，确认订单时如果酒店告知房间已满无法入住，携程会通知用户并推荐用户改订其他酒店，该过程也属于后处理。

（3）用户订单操作：即通常所说的客服，接听用户来电，处理用户取消订单、修改入住人或时间、增加特殊要求、开发票等方面的需求。

（4）投诉处理：负责处理用户投诉

除处理投诉必须人工协调以外，其他三项职能的工作都有既定的流程，因此都包含大量重复性的工作。经过长期的工作积累发现，这些流程整体保持高效，但部分环节效率略低，可以用自动化的方式代替或改进。

二、订单确认智能外呼

上文提到，如果酒店长时间没有确认订单，呼叫中心会人工外呼给酒店催单。问题是，应该采取怎样的外呼策略，可以使订单确认外呼这项工作更加高效。

2.1 传统的订单外呼流程

传统的订单确认流程很简单，等待酒店回复——到指定时间仍未回复——人工外呼酒店。这是一个很自然的流程，然而实际效率并不高。主要有两点：

（1）向酒店发出预订消息后，只能一直等待，直到快要超时了再去催单吗？事实上，有些酒店确认订单的速度一向很慢，或者某个特殊的时段、某些特定的房型酒店总是很晚才给回复。可能我们已经预料到酒店会很晚才给回复，但也会等到指定时间外呼催单，这样用户就白白等待了很长时间。或者，我们可以提前外呼这些酒店催单？

（2）实际操作中发现，很多时候，酒店只是在电话里表示了解情况，通话结束后仍然过很久才给回复，那这次外呼就变得没有意义，可能酒店已经习惯或者确实需要那么多时间来做确认，有没有给酒店外呼结果是一样的。毕竟人力有限，不可能所有订单都做外呼，应该优先去做外呼有意义的订单，而可能无效的外呼就应暂缓。那么，在外呼之前，能否预判这次外呼是否有效？

顺着以上两个问题的思路，就把一个依次排队的傻瓜式外呼，转变为智能化的预测式外呼。这里的关键，就是要提前找出哪些订单需要提前外呼，以及预判外呼的有效性。这是一个需要自动执行的预测功能，机器学习在这里派上了用场。

2.2 改造后的智能外呼流程

改造后的预测式外呼流程加入了两个机器学习模型：回复时长预测模型和外呼有效性模型。

（1）通过回复时长预测模型，预测订单的回复时长是否会超过一定容忍范围。在容忍范围内回复的，暂时不用做什么。如果等到容忍范围后仍未回复，再进入队列准备外呼。预测回复时长超过容忍范围的，提前外呼，提前对可能需要很长时间回复的酒店进行催单。

（2）准备外呼前，每个订单由外呼有效性模型做一次判定，即本次外呼是否有效。判定为有效的，优先安排人工外呼；判定为无效的，说明此时电话打出去也没有意义，人工外呼延后。但为了应对模型误判，防止确认超时，我们会用 IVR 自动语音外呼做一次催单。如果 IVR 没有起作用，在剩下的时间里，就对这些仍未确认的订单安排人工外呼。

总结一下这个预测式外呼流程，我们把可能会很晚回复，并且立即外呼也是有效的订单，尽可能优先安排人工外呼；而其他需要较长时间回复，或者打电话不起作用的外呼延后。比起原本订单一个个排队外呼的方式，新流程对资源的分配更为合理。

2.3 模型的构建

下面介绍一下上面所用的两个模型。这是两个有监督模型，我们结合业务背景提取相关特征，构建数据集，离线进行模型训练。

以外呼有效性模型为例，决定外呼是否有效的因素有很多，包括下单所处的时间点，几天后入住，是否是特殊或热卖的房型，携程与酒店的合作关系，酒店历史的外呼数量，无效外呼数量，以及酒店的操作习惯等。训练过程中我们尝试了多种模型，最终选择 XGBoost 模型上线。

模型以及新流程上线后，对比上线前后一段时间的数据，总订单量增长了 25%，而实际进行人工外呼额订单占比减少了 1/3，同时确认用户的平均时长没有显著变化，说明在没有影响客户体验的前提下，通过以模型预测进行资源重新分配的方式，减少了大量无效外呼，流程得到了优化，呼叫中心工作的运行效率得到大幅提升。

三、总结和展望

以上我们介绍了大数据和机器学习在提升携程呼叫中心自动化方面的一些应用，重点介绍了预测式智能外呼的细节。这些项目的成功上线，验证大数据和机器学习在提升自动化，优化资源分配，改进流程提高效率方面可以发挥重要作用。我们分析现状，寻找流程中的不足和改进点，用机器学习加以改进，在其他项目中也值得借鉴。

未来携程将致力于打造一个全新的、更加数据化智能化的呼叫中心，从而更好地服务用户，为行业树立一个新的标杆。

携程度假智能云客服平台

[作者简介]李健，携程度假大数据开发总监。2013 年底加入携程，在攻略社区及度假负责自然语言处理、图像、推荐等领域的开发管理工作。

写在前面

在人工智能时代， AI 技术会以提供更精准更高效的方式在流程改进、沟通费力度下降、沟通效率提高、成本降低及收益提升等众多方面全面改变目前的商业模式、推动业务发展。携程度假的智能云客服平台在这方面做了很多有益的尝试，大大提升了携程度假客服的效率和用户体验。

一、智能云客服平台概述

在智能云客服平台上线前，在包括 IM/微信、在线客服和电话客服在内的多个服务渠道的各个行为阶段都存在着不少痛点问题。由于度假的产品涵盖酒店、机票和门票等多个方面，服务的多样性更加明显，给痛点问题的解决带来了更多的困难。

为了解决上述痛点问题，我们建立的智能云客服平台目标主要围绕以下六个方面进行改善：

1）服务流转效率；
2）服务响应时间；
3）标准化服务；
4）自动化服务；
5）违规管理；
6）流程优化；

当前的智能云客服平台已经在智能问答、服务渠道管理、服务流程优化和供应商管理四个方

面上线了近 80 个模型，取得了很好的效果。

二、系统架构

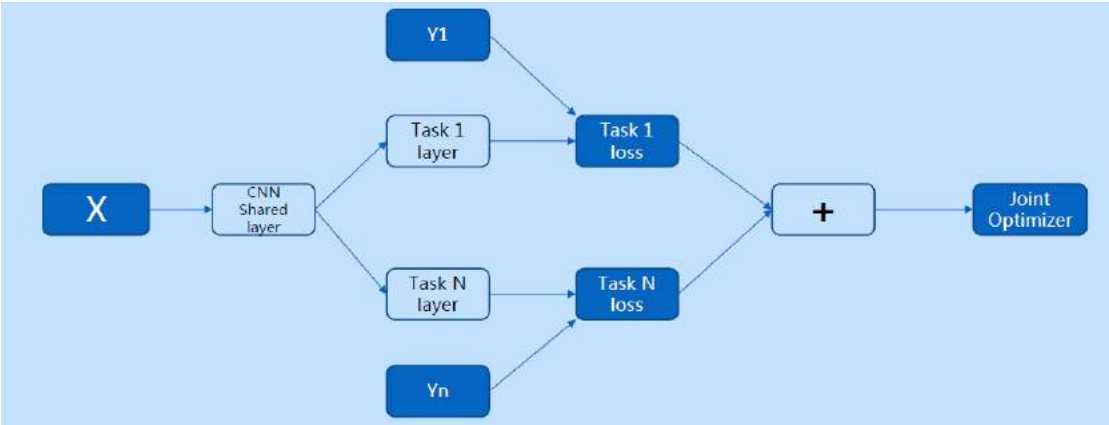
下图展示了我们智能云客服平台的系统架构，其中在算法部分，我们设计并上线的自动纠错模型，使用户意图理解的准确率从 60%提升到了 90%以上；在工程部分，我们设计并构建了 EasyAI 平台，与很多 AI 平台往往提供给算法工程师使用不同，我们的 EasyAI 面向的是不熟悉技术的业务群体，通过这个平台，提升了业务 50%的工作效率。

三、典型算法模型介绍

3.1 用户意图模型

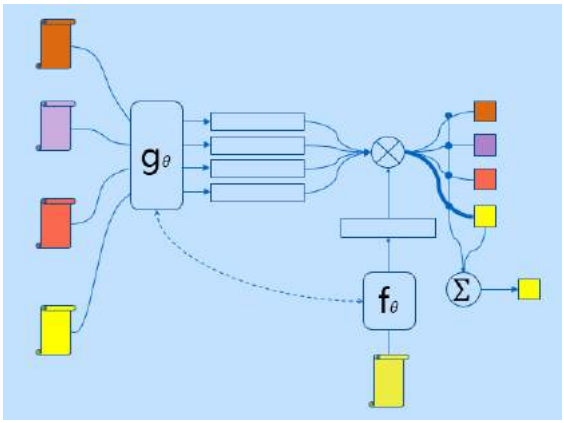
用户在和客服聊天的时候，往往会有各种各样的意图，我们需要实时判断用户是否存在购买某一个具体产品的意图，或者是否能够形成一个明确的意图方便后续的产品推荐。我们采用了基于深度学习的多任务多标签的模型来实现，如果用户被判为存在上述意图，则会基于该

意图，为用户提供更个性化的服务。



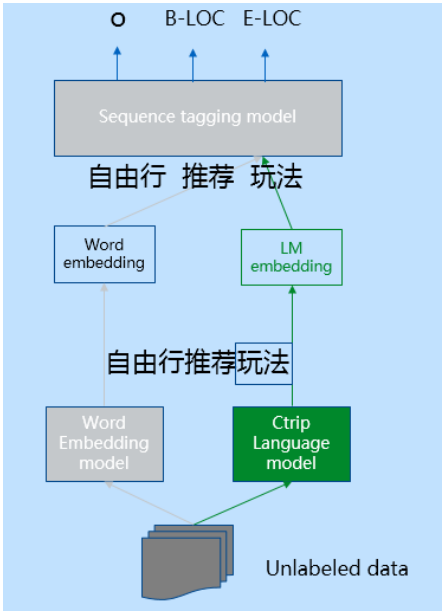
3.2 群监督模型

因为用户和客服的沟通都是基于群的方式来实现的，所以我们需要使用群监督模型来更好得对用户意图进行分析和判断。在真实情况下，用户的大部分聊天内容比较集中，样本量也非常大，而在某些意图上的样本量缺极度缺乏，为此我们构建了一个有效的群监督模型来解决样本极度不平衡的问题，该模型上线后，准确率从 10%+提升到了 80%+。



3.3 内容抽取模型

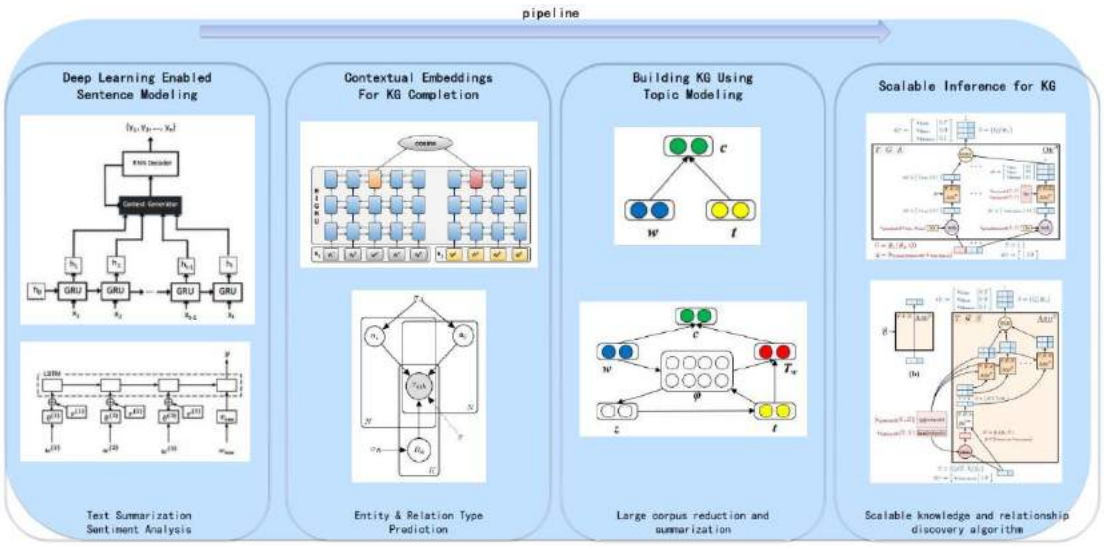
在智能云客服平台的诸多应用场景中都会使用到内容抽取功能，为此，我们构建了一个基于半监督学习的内容抽取通用模型。具体地，我们在常用的内容抽取模型的基础上加入自己建立的语言模型来实现半监督学习，最终准确率提升了 7%。



3.4 知识图谱构建

在知识图谱的构建过程中，我们首先划分 Domain，然后在每个 Domain 下去产生一个 Schema。产生 Schema 的方式分为人工和自动两种，自动产生 Schema 的效果往往不是很理想，所以我们采用人工定义 Schema 的方式。

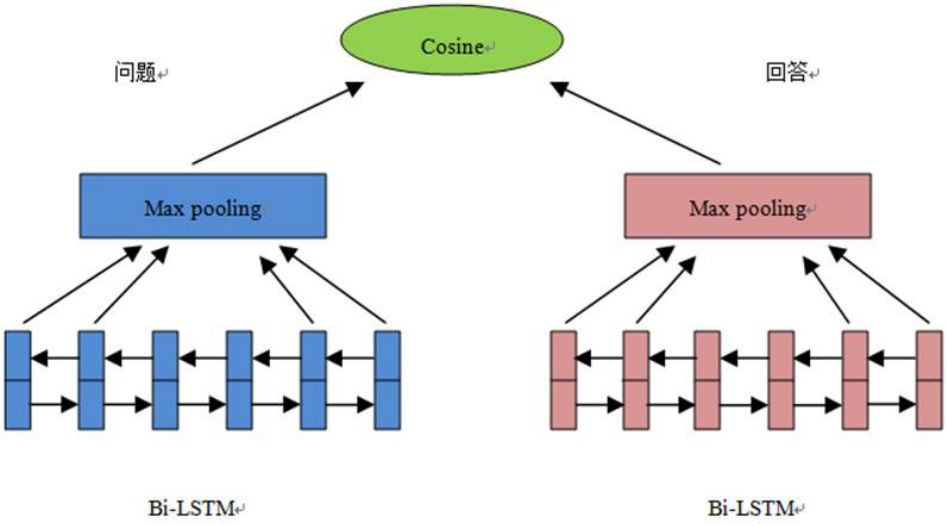
接下来我们在 Schema 的基础上实现信息的抽取，并进一步完成知识图谱的补全和融合。下图展示了我们从知识图谱的构建、抽取、补全、融合到推理过程中使用的所有模型。



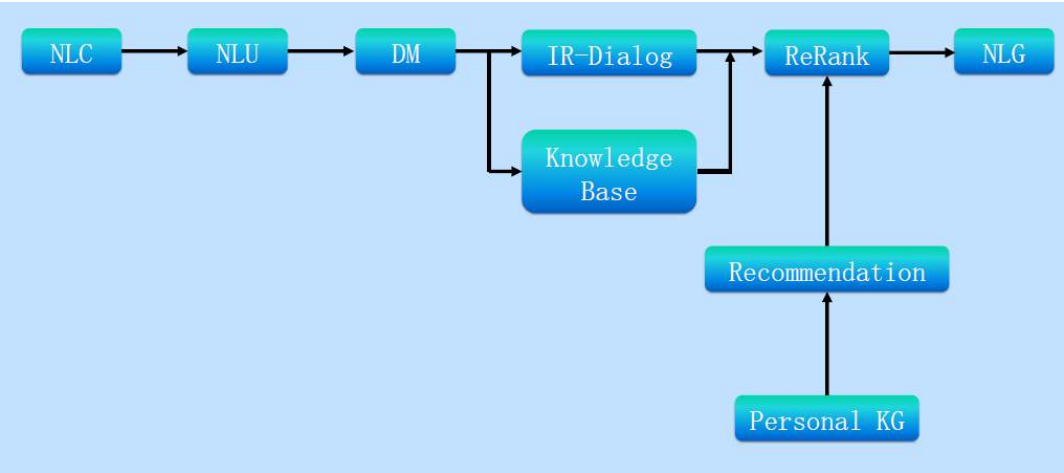
3.5 自助问答模型

智能云客服平台的自助问答系统经历了从 1.0 到 2.0 的发展过程，在自助问答 1.0 系统中，我们建立粗粒度检索式模型来实现和用户的单轮对话。具体地，我们采用 Bi-LSTM+Attention+CNN 的方式来实现，分别对问题和回答建立模型，最后计算两个模型输出

向量之间的相似度。

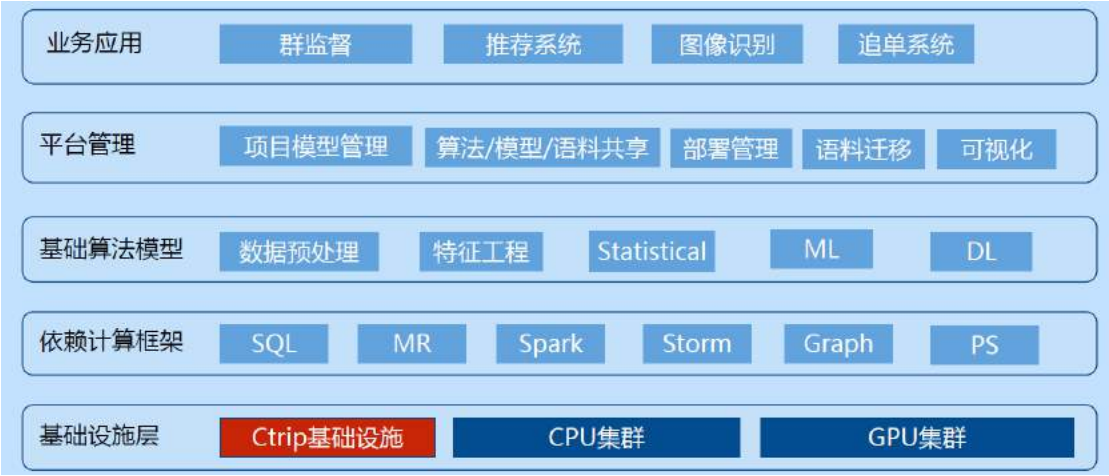


在自助问答 2.0 系统中，我们实现了机器和用户的多轮对话。对于用户的聊天内容，首先进行内容自动纠错和自然语言理解，然后进入对话管理，为了保证对话管理的性能，我们开发了核心组件。在对话管理之后，根据对用户意图和对话控制的理解，会选择检索式粗粒度回答和知识库细粒度回答两种方式。在获取所有的回答之后，我们会对其进行重新排序，同时，我们还会根据对用户意图的理解和当前用户所处的环节进行相应的推荐，最后生成最佳回答。下图展示了自助问答 2.0 系统的处理流程。



四、EasyAI 平台介绍

在第二部分中，我们提到了 EasyAI 平台，这里我们做进一步的介绍。我们建立 EasyAI 平台的初衷是为了直接给业务进行操作。利用 EasyAI 平台，我们可以提高语料标注和模型训练的效率，实现标注语料的共享，同时也可以缩短建设周期，减少模型的重复建设。下图展示了我们 EasyAI 平台的系统架构。



写在最后

以上对携程度假智能云客服平台进行了介绍，但 AI 对于携程度假的价值远远不限于此，后续我们会进一步将知识图谱应用于智能推荐和搜索，同时也会实践旅游领域的机器阅读，减少知识图谱构建的费力度，最终实现服务全流程的自动化和智能化。

机器学习算法在饿了么供需平衡系统中的应用

[作者简介]陈宁，饿了么人工智能与策略部高级算法专家，负责供需平衡系统的算法与研发工作。获新加坡南洋理工大学计算机博士学位，研究方向包括：数据挖掘，机器学习，自然语言处理，软件工程等。本文来自陈宁在第二届携程云海机器学习沙龙上的分享。

即时配送物流系统是外卖领域核心价值所在。区别于其他物流，这个行业的“物流”是希望 30 分钟内能够送到，这就对我们的算法模型具有很大的挑战。智能调度系统更是即时配送物流系统中的核心。

在本文中，我将介绍饿了么的智能调度系统，着重介绍其中的压力平衡子系统，并通过两个实例，分享机器学习算法在构建压力平衡系统中起到的关键作用以及取得的成果。

希望通过本文，帮助一线的机器学习算法工程师和爱好者们了解饿了么即时配送系统中压力平衡系统的构建，以及如何利用常见机器学习算法有效地解决 O2O 场景下的实际问题。

一、饿了么智能调度系统

饿了么智能调度系统是外卖即时配送领域中最核心的环节，该系统替代了调度员大部分的工作，减少了人力介入的程度，实现了自动化、智能化的派单。

智能调度系统主要可以划分为四个子系统，如图 1 所示：

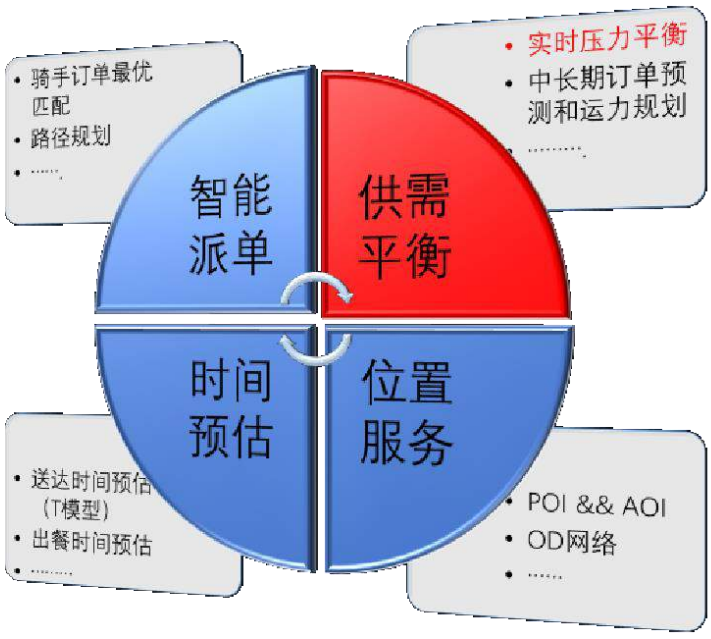


图 1 饿了么智能调度系统

- 智能派单：包括骑手与订单的匹配策略，骑手的路径规划等；
- 时间预估：包括骑手行程时间预估，骑手楼内停留时间预估，出餐时间预估与下单送达时间预估等；
- 供需平衡：包括实时的压力平衡，中长期订单预测、运力规划和短期的骑手排班等；
- 位置服务：包括配送范围划定，商圈/楼宇/配送点的建设，位置校准等。

以上各个子系统相辅相成，组成了整个饿了么智能调度系统。本文中我将重点介绍机器学习算法在供需平衡中的实时压力平衡系统中起到的作用。

二、压力平衡系统

2.1 系统目标

压力平衡系统要解决的问题是：当配送供给(骑手)与用户需求（订单）出现日内的异常不匹配时，为了保证用户体验不受到过大损伤，及时有效地使用调控手段来平衡需求与供给。

图 2 压力平衡系统目标

造成用户需求和配送供给日内异常不匹配的原因有很多：比如遇到恶劣的天气、商家搞一些临时性的大活动，骑手运力本身不充足等。

当出现异常不匹配时(压力高于某个阈值)，压力平衡系统会采取一定的措施。比如上调配送费、缩小配送范围、下满减活动、关店等。

2.2 算法框架

图 3 展示了压力平衡算法的总体架构：

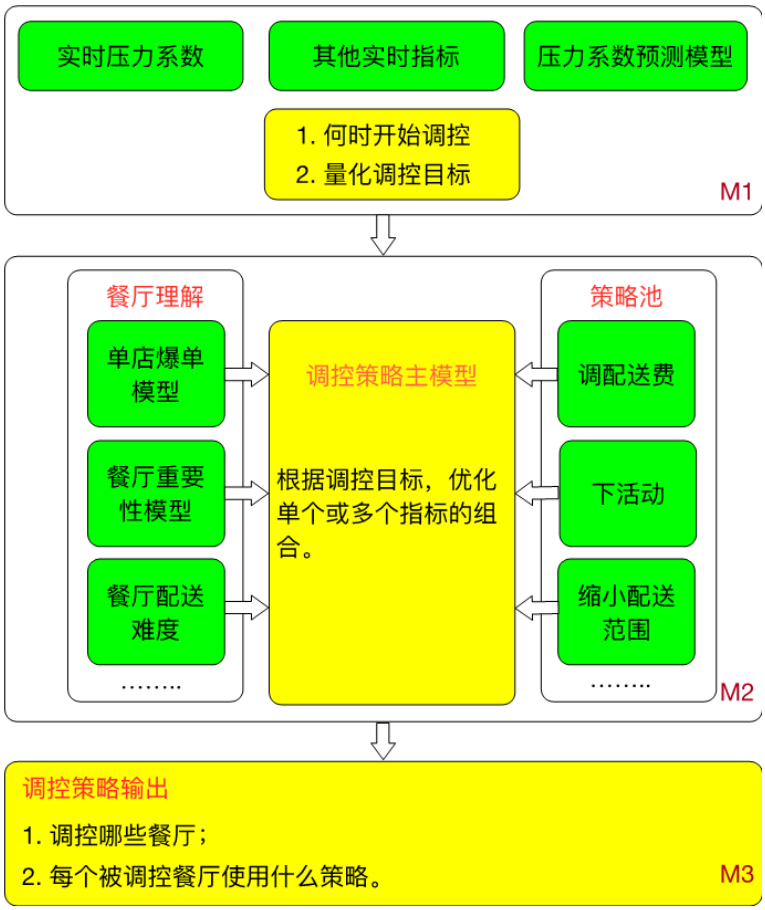


图 3 压力平衡算法框架图

首先，图 3 的最上方（M1）展示的是团队(饿了么组织运力的最小单元，一个团队由若干骑手组成）压力系数及其辅助指标的实时监控和预测模块。这个模块主要的作用是估算自动调控开始的最佳时机和调控的量化控目标：即当压力系数大于一定的阈值时，开始自动采取一系列的措施，使得压力系数降低到可接受的范围。

其次，图 3 的中部（M2）展示的主要是两个模块：1）餐厅理解；2）调控策略池。其中，餐厅理解：利用餐厅重要性排序模型，单店爆单模型，餐厅配送难度等信息来细化调控的基本单元，它使我们知道哪些餐厅是应该首先被调控的。调控策略池包含了调控可用的调控手段，例如：增加配送费，下大额满减活动，缩小餐厅的配送范围等。

最后，图 3 的底部（M3）展示的是自动调控策略生成主模型：我们会根据量化的调控目标，来优化单个或多个指标的组合。通过数据、算法来决定调控哪些餐厅，每个餐厅使用什么样的调控策略。

图中每个模块都通过数据驱动，应用了机器学习算法来实现相应的目标。图中绿色标记是我们已经完成的，标记黄色部分是我们正在或者规划中的工作。在第四章中，我将通过两个例子来介绍我们取得的一些成果。

2.3 数据监控

我们还构建了一套策略实时监控系统，方便相关人员查看。例如，图 4 显示的是某个团队的实时压力系数。团队压力系数，我们做到了归一化(所有团队值域一致，含义一致)，并且能根据天气、温度等情况自适应地变化。当前，这个指标我们做到了每隔 5 分钟更新一次。在第四章中，我将会更加详细地介绍团队压力系数的计算方式。

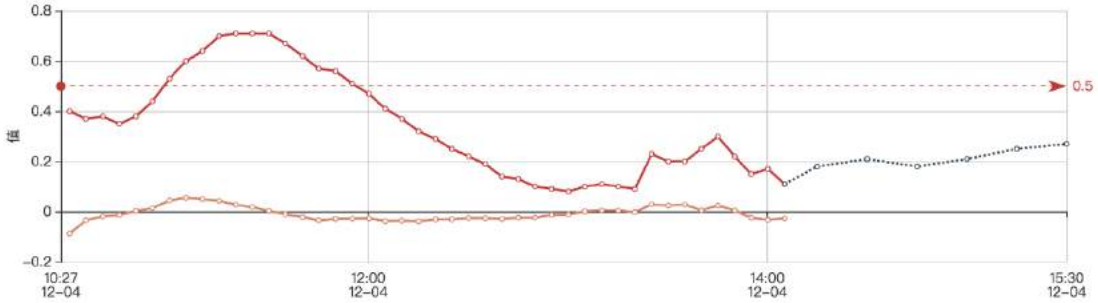


图 4 压力系数监控看板

三、机器学习算法在压力平衡中的应用

在这个章节中，我将通过两个案例来介绍机器学习算法在构建饿了么压力平衡系统中起到的关键作用。

3.1 骑手最大背单能力模型

骑手的最大背单能力反应了一个骑手的水平，是骑手画像的非常重要的组成部分。第一版，由于我们只是为了得到团队的最大背单能力，所以我们只使用了简单的规则，计算了团队的平均最大背单能力，用这个平均最大背单能力作为这个团队骑手的最大背单能力。显然，这是非常不合理的。

在第二版中，我们通过一些负责的规则，将骑手分成了若干等级，每个等级的骑手具有一个相同的最大背单能力。这个版本的骑手最大背单能力具有了一些个性化，但是任然无法细致、有效地区分不同骑手的水平。

在第三版中，我们采用了机器学习的方法，把这个问题抽象成一个二分类的问题，从而得到了每个骑手在不同背单量情况下超时的概率。图 5 显示了以上介绍的迭代内容，下面介绍 V3.0 版本我们的思路。

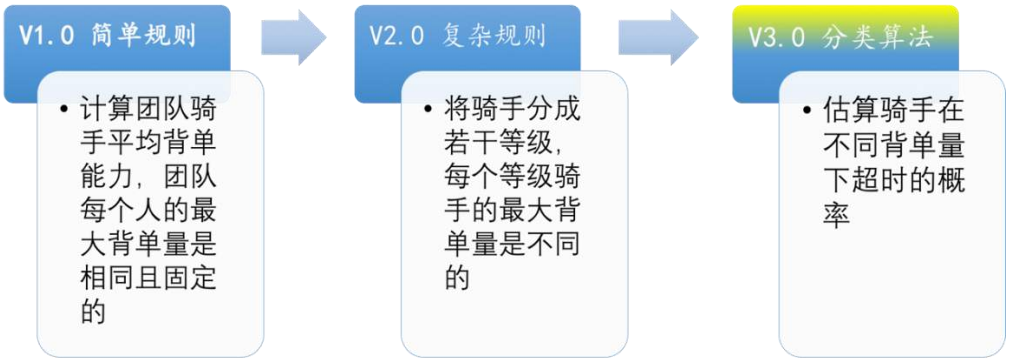


图 5 骑手最大背单能力模型迭代

解决这个问题的关键思路在于，我们将骑手最大背单能力的估算转化为一个二分类问题。具体步骤如下：

第一步：我们队每张骑手的运单构建一条训练数据；
第二步：统计这张运单运送过程中骑手的最大背单量作为其中一个特征；
第三步：统计其他特征，包括骑手画像相关特征，天气特征等；
第四步：构建二分类的 Label，一张运单超时记为 1，不超时记为 0；
第五步：使用大量的训练数据，训练得到一个线性的二分类模型；
第六步：给定一个骑手，固定其他特征，并滑动最大背单量特征，得到该骑手在不同最大背单量下超时的概率。

下面，我详细介绍一下特征工程部分，主要包括以下几块内容：

1）骑手历史最大非超时背单量。包含前 28 天、21 天、14 天、7 天、5 天、3 天的骑手平均最大非超时背单量；
2）天气和温度信息；
3）骑手的个性化信息：包括骑手的等级，骑手的工作天数，骑手所在团队的信息等；
4）时间信息：主要包括该日是一周的哪一天，是否是工作日两个特征；
5）运单信息：运单运送过程中骑手的身上最大背单量。

接下来我介绍一下模型的选择和训练集和预测集的构造。根据简单有效原理(即：奥卡姆剃刀定律)，我们采用了常用的线性模型，即 Logistic Regression(LR)模型，我们的考虑如下：

- 线性模型，模型非常简单，有效，计算速度快；
- 模型可解释性很强，便于业务人员理解；
- 问题需要输出[0,1]之间的概率值，LR 使用 sigmoid 函数能够将预测值转化为概率值。

训练集的构造我们采用了最近 14 天的运单数据，即数据日期范围是[T-13,T]， 时间范围限定在了午高峰(10:30~12:30)。由于超时单相比非超时单占比非常小，所以我们对正负样本的比例进行了调整，对非超时样本按照 10%左右进行了降采样。

在构造测试集时，我们对运单运送过程中骑手的最大背单量特征进行滑动，从 0 ~ 30 按照 0.5 的间隔进行构造。然后通过训练的模型进行预测，从而得到骑手在不同背单量下的超时概率。如图 6 所示，是 taker_id =342853 的骑手在不同 max_order_num_on_taker 情况下超时的 probability。

taker_id	team_id	max_order_num_on_taker	prediction	probability
342853	6203118	0.5	0	0.006921988
342853	6203118	1	0	0.008718609
342853	6203118	1.5	0	0.010976393
342853	6203118	2	0	0.013810711
342853	6203118	2.5	0	0.017364056
342853	6203118	3	0	0.021811417
342853	6203118	3.5	0	0.027366133
342853	6203118	4	0	0.034285886
342853	6203118	4.5	0	0.042878221
342853	6203118	5	0	0.053504566
342853	6203118	5.5	0	0.066581201
342853	6203118	6	0	0.08257497
342853	6203118	6.5	0	0.101990881
342853	6203118	7	0	0.125348314
342853	6203118	7.5	0	0.153142716
342853	6203118	8	0	0.185791036
342853	6203118	8.5	0	0.223562173
342853	6203118	9	0	0.266498785
342853	6203118	9.5	0	0.314343137
342853	6203118	10	0	0.366485328
342853	6203118	10.5	0	0.421953981
342853	6203118	11	0	0.479464115
342853	6203118	11.5	1	0.537523733
342853	6203118	12	1	0.594583233
342853	6203118	12.5	1	0.649197266
342853	6203118	13	1	0.700164227
342853	6203118	13.5	1	0.746616491
342853	6203118	14	1	0.78805063
342853	6203118	14.5	1	0.824303739
342853	6203118	15	1	0.855492981
342853	6203118	15.5	1	0.881938563
342853	6203118	16	1	0.904087056
342853	6203118	16.5	1	0.922445842
342853	6203118	17	1	0.937533355

图 6 骑手在不同背单量情况下超时的概率

下面我来介绍一个骑手最大背单模型的重要应用场景，即计算团队的压力系数：

我们定义团队压力系数 = load / (q1 + q2 + ... qn)
1）load=团队负载
2）qn=该团队第 n 位骑手的个性化最大背单量(使用骑手最大背单模型计算的数值)

团队压力系数的定义简洁有效，直接支持了饿了么即时配送相关 10 多个相关业务系统：包括智能派单，压力平衡，运单分流，T 模型，客服系统等。

3.2 团队压力系数预测模型

正如章节 3.1 所介绍，团队压力系数是即时配送中一项非常重要的基础指标，因此，如果我们能提前预测这一指标将会对多个业务系统产生巨大的价值。基于此，我们构建了一个实时的团队压力系数预测模型，它能够每隔 5 分钟对团队未来 1 个半小时(每 15 分钟一个时间

片，共 6 个时间片)的压力系数进行预测。图 4 中的虚线即展示了其中一次预测的值。我们将这个模型应用在以下 2 个场景中：

1）定量描述压力平衡自动调控的目标，辅助确定自动调控的时间点，进行提前调控；
2）应用在智能派单策略上，通过压力系数的预知，规划不同的派单策略。

如下图所示：截止目前，我们完成了团队压力系数预测模型的三轮迭代。通过持续迭代，我们大幅提高了模型预测的准确率和预测的频率。

四、总结展望

通过本文，我们希望读者能够对饿了么即时配送体系中的压力平衡系统以及如何利用常见的机器学习算法有效解决 O2O 场景下的实际问题有所了解。展望未来，压力平衡系统中仍然有很多问题可以抽象成机器学习问题（例如：排序学习，时间序列等），希望在不久的未来，有机会和大家分享我们更多的成果。

携程个性化推荐算法实践

[作者简介]携程基础业务研发部-数据产品和服务组，专注于个性化推荐、自然语言处理、图像识别等人工智能领域的先进技术在旅游行业的应用研究并落地产生价值。目前，团队已经为携程提供了通用化的个性化推荐系统、智能客服系统、AI 平台等一系列成熟的产品与服务。

携程作为国内领先的 OTA，每天向上千万用户提供全方位的旅行服务，如何为如此众多的用户发现适合自己的旅游产品与服务，挖掘潜在的兴趣，缓解信息过载，个性化推荐系统与算法在其中发挥着不可或缺的作用。而 OTA 的个性化推荐一直也是个难点，没有太多成功经验可以借鉴，本文分享了携程在个性化推荐实践中的一些尝试与摸索。

推荐流程大体上可以分为 3 个部分，召回、排序、推荐结果生成，整体的架构如下图所示。

召回阶段，主要是利用数据工程和算法的方式，从千万级的产品中锁定特定的候选集合，完成对产品的初步筛选，其在一定程度上决定了排序阶段的效率和推荐结果的优劣。

业内比较传统的算法，主要是 CF[1][2]、基于统计的 Contextual 推荐和 LBS，但近期来深度学习被广泛引入，算法性取得较大的提升，如：2015 年 Netflix 和 Gravity R&D Inc 提出的利用 RNN 的 Session-based 推荐[5], 2016 年 Recsys 上提出的结合 CNN 和 PMF 应用于 Context-aware 推荐[10]，2016 年 Google 提出的将 DNN 作为 MF 的推广，可以很容易地将任意连续和分类特征添加到模型中[9]，2017 年 IJCAI 会议中提出的利用 LSTM 进行序列推荐[6]。2017 年携程个性化团队在 AAAI 会议上提出的深度模型 aSDAE，通过将附加的 side information 集成到输入中，可以改善数据稀疏和冷启动问题[4]。

对于召回阶段得到的候选集，会对其进行更加复杂和精确的打分与重排序，进而得到一个更

小的用户可能感兴趣的产品列表。携程的推荐排序并不单纯追求点击率或者转化率，还需要考虑距离控制，产品质量控制等因素。相比适用于搜索排序，文本相关性检索等领域的 pairwise 和 listwise 方法，pointwise 方法可以通过叠加其他控制项进行干预，适用于多目标优化问题。

工业界的推荐方法经历从线性模型 + 大量人工特征工程[11] -> 复杂非线性模型-> 深度学习的发展。Microsoft 首先于 2007 年提出采用 Logistic Regression 来预估搜索广告的点击率[12]，并于同年提出 OWLQN 优化算法用于求解带 L1 正则的 LR 问题[13]，之后于 2010 年提出基于 L2 正则的在线学习版本 Ad Predictor[14]。

Google 在 2013 年提出基于 L1 正则化的 LR 优化算法 FTRL-Proximal[15]。2010 年提出的 Factorization Machine 算法[17]和进一步 2014 年提出的 Filed-aware Factorization Machine[18]旨在解决稀疏数据下的特征组合问题，从而避免采用 LR 时需要的大量人工特征组合工作。

阿里于 2011 年提出 Mixture of Logistic Regression 直接在原始空间学习特征之间的非线性关系[19]。Facebook 于 2014 年提出采用 GBDT 做自动特征组合，同时融合 Logistic Regression[20]。

近年来，深度学习也被成功应用于推荐排序领域。Google 在 2016 年提出 wide and deep learning 方法[21]，综合模型的记忆和泛化能力。进一步华为提出 DeepFM[15]模型用于替换 wdl 中的人工特征组合部分。阿里在 2017 年将 attention 机制引入，提出 Deep Interest Network[23]。

携程在实践相应的模型中积累了一定的经验，无论是最常用的逻辑回归模型（Logistic Regression），树模型（GBDT，Random Forest）[16]，因子分解机（FactorizationMachine），以及近期提出的 wdl 模型。同时，我们认为即使在深度学习大行其道的今下，精细化的特征工程仍然是不可或缺的。

基于排序后的列表，在综合考虑多样性、新颖性、Exploit & Explore 等因素后，生成最终的推荐结果。本文之后将着重介绍召回与排序相关的工作与实践。

一、数据

机器学习 = 数据 + 特征 + 模型

在介绍召回和排序之前，先简单的了解一下所用到的数据。携程作为大型 OTA 企业，每天都有海量用户来访问，积累了大量的产品数据以及用户行为相关的数据。实际在召回和排序的过程中大致使用到了以下这些数据：

- 产品属性：产品的一些固有属性，如酒店的位置，星级，房型等。
- 产品统计：比如产品一段时间内的订单量，浏览量，搜索量，点击率等。
- 用户画像：用户基础属性，比如年纪，性别，偏好等等。
- 用户行为：用户的评论，评分，浏览，搜索，下单等行为。

值得注意的是，针对统计类信息，可能需要进行一些平滑。例如针对历史 CTR 反馈，利用贝叶斯平滑来预处理。

二、召回

召回阶段是推荐流程基础的一步，从成千上万的 Item 中生成数量有限的候选集，在一定程度上决定了排序阶段的效率和推荐结果的优劣。而由 OTA 的属性决定，用户的访问行为大多是低频的。这就使得 user-item 的交互数据是极其稀疏的，这对召回提出了很大的挑战。在业务实践中，我们结合现有的通用推荐方法和业务场景，筛选和摸索出了几种行之有效的方法：

Real-timeIntention

我们的实时意图系统可以根据用户最近浏览下单等行为，基于马尔科夫预测模型推荐或者交叉推荐出的产品。这些候选产品可以比较精准的反应出用户最近最新的意愿。

BusinessRules

业务规则是认为设定的规则，用来限定推荐的内容范围等。例如机票推酒店的场景，需要通过业务规则来限定推荐的产品只能是酒店，而不会推荐其他旅游产品。

Context-Based

基于 Context 的推荐场景和 Context 本身密切相关，例如与季候相关的旅游产品（冬季滑雪、元旦跨年等）。

LBS

基于用户的当前位置信息，筛选出的周边酒店，景点，美食等等，比较适用于行中场景的推荐。地理位置距离通过 GeoHash 算法计算，将区域递归划分为规则矩形，并对每个矩形进行编码，筛选 GeoHash 编码相似的 POI，然后进行实际距离计算。

CollaborativeFiltering

协同过滤算法是推荐系统广泛使用的一种解决实际问题的方法。携程个性化团队在深度学习与推荐系统结合的领域进行了相关的研究与应用，通过改进现有的深度模型，提出了一种深度模型 aSDAE。该混合协同过滤模型是 SDAE 的一种变体，通过将附加的 side information 集成到输入中，可以改善数据稀疏和冷启动问题，详情可以参见文献[4]。

SequentialModel

现有的矩阵分解(Matrix Factorization)方法基于历史的 user-item 交互学习用户的长期兴趣偏好，Markov chain 通过学习 item 间的 transition graph 对用户的序列行为建模[3]。事实上，在旅游场景下，加入用户行为的先后顺序，从而能更好的反映用户的决策过程。我们结合 Matrix Factorization 和 Markov chain 为每个用户构建个性化转移矩阵，从而基于用户的历史行为来预测用户的下一行为。在旅游场景中，可以用来预测用户下一个目的地或者 POI。

除此之外，也可以使用 RNN 来进行序列推荐，比如基于 Session 的推荐[5]，使用考虑时间间隔信息的 LSTM 来做下一个 item 的推荐等[6]。

此外，一些常见的深度模型(DNN, AE,CNN 等)[7][8][9][10]都可以应用于推荐系统中，但是针对不同领域的推荐，需要更多的高效的模型。随着深度学习技术的发展，相信深度学习将会成为推荐系统领域中一项非常重要的技术手段。以上几种类型的召回方法各有优势，在实践中，针对不同场景，结合使用多种方法，提供给用户最佳的推荐，以此提升用户体验，增加用户粘性。

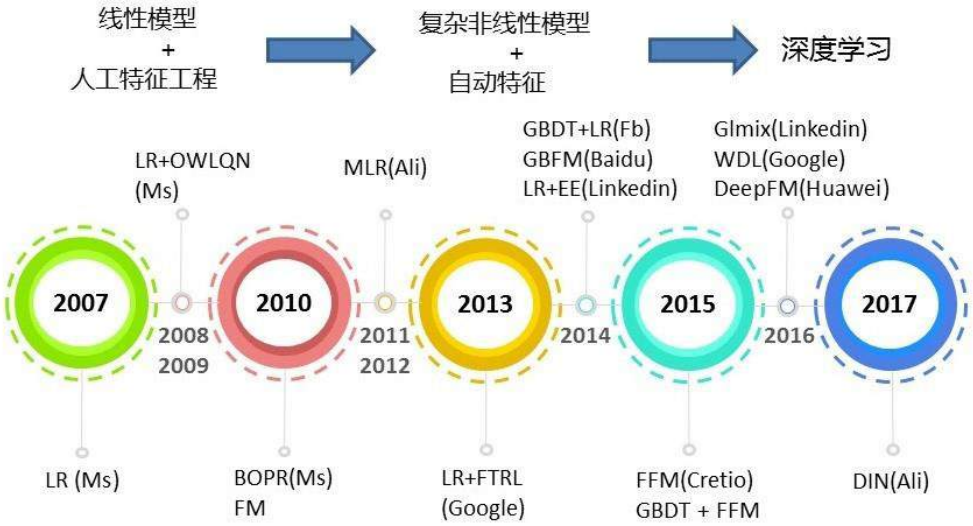
三、排序

以工业界在广告、搜索、推荐等领域的实践经验，在数据给定的条件下，经历了从简单线性模型 + 大量人工特征工程到复杂非线性模型 + 自动特征学习的演变。在构建携程个性化推荐系统的实践过程中，对于推荐排序这个特定问题有一些自己的思考和总结，并将从特征和模型这两方面展开。

Model

个性化排序模型旨在利用每个用户的历史行为数据集建立其各自的排序模型，本质上可以看作多任务学习(multi-task learning)。事实上，通过加入 conjunctionfeatures，也就是加入 user 和 product 的交叉特征，可以将特定的 multi-task 任务简化为单任务模型。

梳理工业界应用的排序模型，大致经历三个阶段，如下图所示：



本文并不准备详细介绍上图中的算法细节，感兴趣的读者可以查看相关论文，以下几点是我们的一些实践经验和体会。

1) 在实践中选用以 LR 为主的模型，通过对数据离散化、分布转换等非线性处理后使用 LR。一般的，采用 L1 正则保证模型权重的稀疏性。在优化算法的选择上，使用 OWL-QN 做 batch learning，FTRL 做 online learning。

2）实践中利用因子分解机（FactorizationMachine）得到的特征交叉系数来选择喂入 LR 模型的交叉特征组合，从而避免了繁杂的特征选择工作。一般的受限于模型复杂度只进行二阶展开。对于三阶以上的特征组合可以利用基于 mutual information 等方法处理。已有针对高阶因子分解机（HighOrder FM）的研究，参见文献[24]。

3）对于 Wide and Deep Learning，将 wide 部分替换 gbdt 组合特征，在实验中取得了较好的效果，并将在近期上线。后续的工作将针对如何进行 wide 部分和 deep 部分的 alternatingtraining 展开。

Feature Engineering

事实上，虽然深度学习等方法一定程度上减少了繁杂的特征工程工作，但我们认为精心设计的特征工程仍旧是不可或缺的，其中如何进行特征组合是我们在实践中着重考虑的问题。一般的，可以分为显式特征组合和半显式特征组合。

显式特征组合

对特征进行离散化后然后进行叉乘，采用笛卡尔积(cartesian product)、内积(inner product)等方式。

在构造交叉特征的过程中，需要进行特征离散化；针对不同的特征类型，有不同的处理方式。

1）numerical feature

无监督离散化：根据简单统计量进行等频、等宽、分位点等划分区间
有监督离散化：1R 方法，Entropy-BasedDiscretization (e.g. D2，MDLP)

2）ordinal feature（有序特征）

编码表示值之间的顺序关系。比如对于卫生条件这一特征，分别有差，中，好三档，那么可以分别编码为(1,0,0),(1,1,0),(1,1,1)。

3）categorical feature (无序特征)

离散化方法	具体做法
OHE(one hot encoding)	用h个变量代表h个level
Dummy Encoding	将一个有h个level的变量变成h-1个变量
Hash Trick	转化为固定长度的hash variable

- 离散化为哑变量，将一维信息嵌入模型的 bias 中，起到简化逻辑回归模型的作用，降低了模型过拟合的风险。
- 离散特征经过 OHE 后，每个分类型变量的各个值在模型中都可以看作独立变量，增强

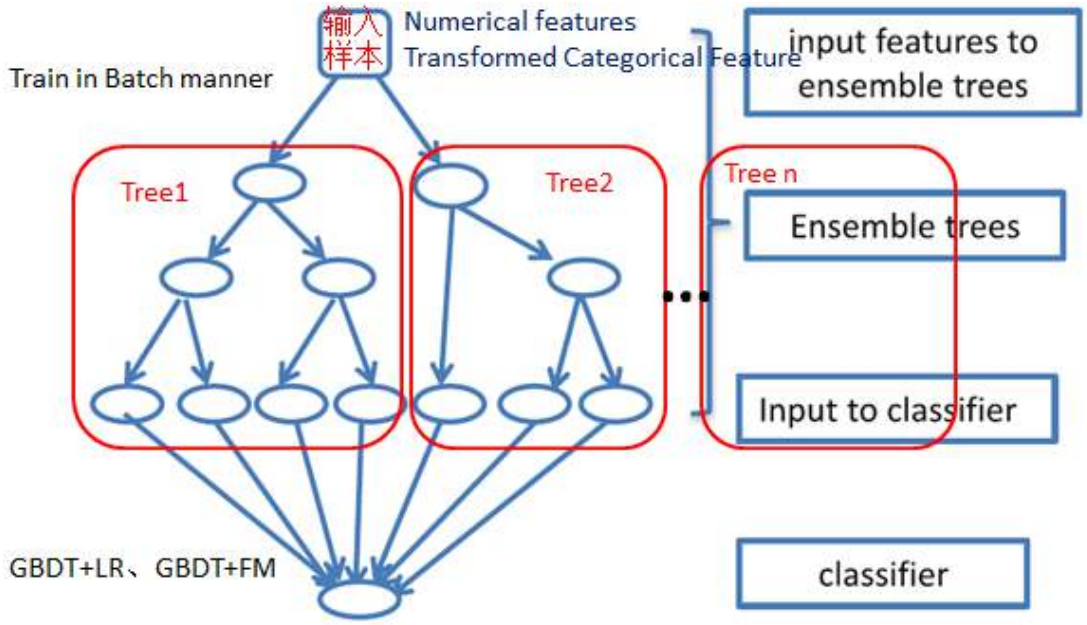
拟合能力。一般的，当模型加正则化的情况下约束模型自由度，我们认为 OHE 更好。

- 利用 feature hash 技术将高维稀疏特征映射到固定维度空间

半显式特征组合

区别于显式特征组合具有明确的组合解释信息，半显式特征组合通常的做法是基于树方法形成特征划分并给出相应组合路径。

一般做法是将样本的连续值特征输入 ensemble tree，分别在每颗决策树沿着特定分支路径最终落入某个叶子结点得到其编号，本质上是这些特征在特定取值区间内的组合。ensemble tree 可以采用 Gbdt 或者 random forest 实现。每一轮迭代，产生一棵新树，最终通过 one-hotencoding 转化为 binary vector，如下图所示。



以下几点是我们在实践中的一些总结和思考。

1）在实验中发现如果将连续值特征进行离散化后喂入 gbdt，gbdt 的效果不佳，AUC 比较低。这是因为 gbdt 本身能很好的处理非线性特征，使用离散化后的特征反而没什么效果。xgboost 等树模型无法有效处理高维稀疏特征比如 user id 类特征，可以采用的替代方式是：将这类 id 利用一种方式转换为一个或多个新的连续型特征，然后用于模型训练。

2）需要注意的是当采用叶子结点的 index 作为特征输出需要考虑每棵树的叶子结点并不完全同处于相同深度。

3）实践中采用了 Monte Carlo Search 对 xgboost 的众多参数进行超参数选择。

4）在离线训练阶段采用基于 Spark 集群的 xgboost 分布式训练，而在线预测时则对模型文件直接进行解析，能够满足线上实时响应的需求。

此外，在实践发现单纯采用 Xgboost 自动学到的高阶组合特征后续输入 LR 模型并不能完全替代人工特征工程的作用；可以将原始特征以及一些人工组合的高阶交叉特征同 xgboost 学习到的特征组合一起放入后续的模型，获得更好的效果。

四、总结

完整的推荐系统是一个庞大的系统，涉及多个方面，除了召回、排序、列表生产等步骤外，还有数据准备与处理，工程架构与实现，前端展现等等。在实际中，通过把这些模块集成在一起，构成了一个集团通用推荐系统，对外提供推服务，应用在 10 多个栏位，60 多个场景，取得了很好的效果。本文侧重介绍了召回与排序算法相关的目前已有的一些工作与实践，下一步，计划引入更多地深度模型来处理召回与排序问题，并结合在线学习、强化学习、迁移学习等方面的进展，优化推荐的整体质量。

References

[1] Koren, Yehuda,Robert Bell, and Chris Volinsky. "Matrix factorization techniques forrecommender systems." Computer 42.8 (2009).
[2] Sedhain, Suvash,et al. "Autorec: Autoencoders meet collaborative filtering." Proceedingsof the 24th International Conference on World Wide Web. ACM, 2015.
[3] Rendle, Steffen,Christoph Freudenthaler, and Lars Schmidt-Thieme. "Factorizing personalized markov chains for next-basket recommendation." Proceedings ofthe 19th international conference on World wide web. ACM, 2010.
[4] Dong, Xin, etal. "A Hybrid Collaborative Filtering Model with Deep Structure for Recommender Systems." AAAI. 2017.
[5] Hidasi, Balázs,et al. "Session-based recommendations with recurrent neuralnetworks." arXiv preprint arXiv:1511.06939 (2015).
[6] Zhu, Yu, et al."What to Do Next: Modeling User Behaviors by Time-LSTM." Proceedingsof the Twenty-Sixth International Joint Conference on Artificial Intelligence,IJCAI-17. 2017.
[7] Barkan, Oren,and Noam Koenigstein. "Item2vec: neural item embedding for collaborative filtering." Machine Learning for Signal Processing (MLSP), 2016 IEEE 26thInternational Workshop on. IEEE, 2016.
[8] Wang, Hao,Naiyan Wang, and Dit-Yan Yeung. "Collaborative deep learning forrecommender systems." Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2015.
[9] Covington, Paul,Jay Adams, and Emre Sargin. "Deep neural networks for youtube recommendations." Proceedings of the 10th ACM Conference on RecommenderSystems. ACM, 2016.
[10] Kim, Donghyun, et al. "Convolutional matrix factorization fordocument context-aware recommendation." Proceedings of the 10th ACM Conference on Recommender Systems. ACM, 2016.[11] https://mli.github.io/2013/03/24/the-end-of-feature-engineering-and-linear-model/[12] Richardson, Matthew, Ewa Dominowska, and Robert Ragno."Predicting clicks: estimating the click-through rate for new ads."Proceedings of the 16th international conference on World Wide Web. ACM, 2007[13] Andrew, Galen, and Jianfeng Gao. "Scalable

training of L1-regularized log-linear models." Proceedings of the 24th international conference on Machine learning. ACM, 2007.[14] Graepel, Thore, et al. "Web-scale bayesian click-through ratepredictionfor sponsored search advertising in microsoft's bing searchengine." Omnipress, 2010.
[15] McMahan, H. Brendan, et al. "Ad click prediction: a view fromthe trenches." Proceedings of the 19th ACM SIGKDD international conferenceon Knowledge discovery and data mining. ACM, 2013.
[16] Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable treeboosting system." Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. ACM, 2016.
[17] Rendle, Steffen. "Factorization machines." Data Mining(ICDM), 2010 IEEE 10th International Conference on. IEEE, 2010.
[18] Juan, Yuchin, et al. "Field-aware factorization machines for CTR prediction." Proceedings of the 10th ACM Conference on RecommenderSystems. ACM, 2016.
[19] Gai, Kun, et al. "Learning Piece-wise Linear Models from Large Scale Data for Ad Click Prediction." arXiv preprint arXiv:1704.05194(2017).
[20] He, Xinran, et al. "Practical lessons from predicting clickson ads at facebook." Proceedings of the Eighth International Workshop onData Mining for Online Advertising. ACM, 2014.
[21] Cheng, Heng-Tze, et al. "Wide & deep learning forrecommender systems." Proceedings of the 1st Workshop on Deep Learning forRecommender Systems. ACM, 2016.
[22] Guo, Huifeng, et al. "DeepFM: A Factorization-Machine basedNeural Network for CTR Prediction." arXiv preprint arXiv:1703.04247(2017).
[23] Zhou, Guorui, et al. "Deep Interest Network for Click-Through Rate Prediction." arXiv preprint arXiv:1706.06978 (2017).
[24] Blondel, Mathieu, et al. "Higher-order factorization machines." Advances in Neural Information Processing Systems. 2016.
[25] http://breezedeus.github.io/2014/11/20/breezedeus-feature-hashing.html
[26] https://en.wikipedia.org/wiki/Categorical_variable
[27] https://www.zhihu.com/question/48674426
[28] 多高的 AUC 才算高？https://zhuanlan.zhihu.com/p/24217322

平安银行算法实践

[作者简介]潘鹏举，平安银行大数据平台 AI 算法和分析团队负责人。2017 年加入平安，组建 AI 和算法团队，推动 AI 在银行业务的应用。此前曾在携程任职，撸代码、写文档、出规范、带团队，参与设计算法工程化架构，带领算法团队助力酒店服务提升。本文来自潘鹏举在第二届携程云海机器学习沙龙上的分享。

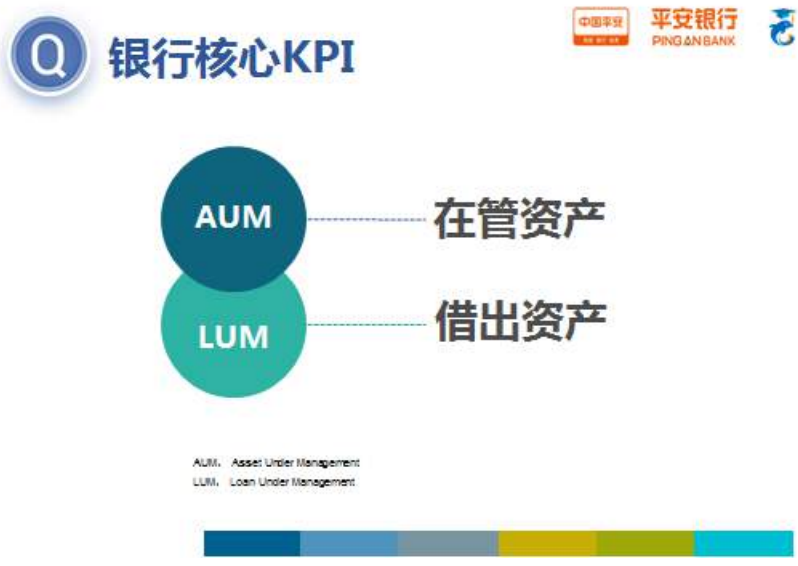
背景

银行是偏传统的行业，目前正在遭受互联网和 P2P 等公司的竞争压力，所以我们正在进行零售转型，拥抱互联网和金融科技。在最近不到一年的时间里，我们在算法方面做了一些尝试和探索，并对未来的一些算法应用有一些思考。整体来说，今天我想分享的内容主要分三大块：

1、业务背景介绍
2、算法实践分享
3、一些思考

一、业务背景

首先来看银行的核心 KPI。从下图中可以知道，银行核心的 KPI 有两个：AUM 和 LUM。



AUM 表示的是在管资产，包括了一些存款、大额存单、同业拆借、央妈给银行代管的钱，表示的是银行从其他渠道拿到的钱；LUM 表示的是借出资金，信用卡也是其中一种类别，表示的是银行借给其他人、政府或者机构的金额。只要借出资产的利率超过借入资产的利率，那么这笔资金的转换就是有收益的。

在这两个资产汇总，在管资产是重中之重，因为国家有规定，LUM/AUM 应该小于某个比例，所以银行资产规模的大小其实是由 AUM 决定的。LUM 相对来说是比较容易达成，需要钱的个人、机构是很多的，但因为风控方面的考虑同时很多人没有人行征信报告，所以正常情况下从银行申请贷款是件非常痛苦的事情，也正因为如此，现如今 P2P 行业、消费金融行业层出不穷，因为供给和需求的不对等催生了如今的现象。

从上面的业务背景描述，我们可以看到 AUM 的核心是客群管理，因为我们在 LUM 方面比较谨慎，换位一下，银行要从其他渠道获取资金也是非常艰难的。LUM 的核心是风控，坏账率是需要关注的一个指标，当然坏账率高低和利差高低有直接关系，这也就是为什么现在的消金公司的逾期率和坏账率比银行高很多的原因。以上两点之外，我们也可以了解到银行受到的一些挑战：1. 受监管 2. 合规性 3.获客难 4. 数据稀疏。

受监管是因为平安银行属于全国性商业银行，受到国家政策的限制和监管，所做的一切都是合规性先行，我们不能做非政府允许或者非个人授权的事情。数据稀疏是针对老客户和新客户来说，老客户因为我们的产品的频次较低，所以能够获取的用户数据较少。而新客户因为我们可以获取的数据量较少，所以我们经常会跟外部的供应商进行合作。在这点上，其实是个机会，所以针对银行、保险和证券这部分的金融领域公司来说，外部的数据供应商是有一定的机会的。

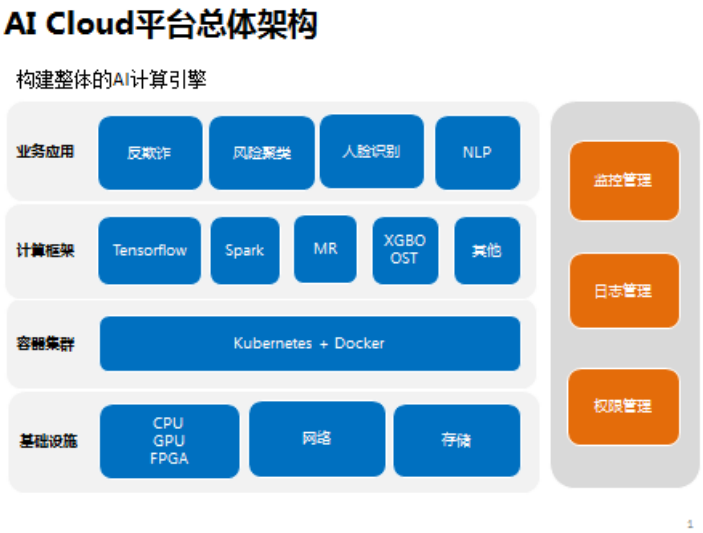
面对这些挑战，我们也尝试在用一些算法实践来帮助提升银行的 AUM 或者降低风控风险。

二、算法实践

在说算法实践之前，我们整体的数据应用的架构如下。分别是基础设施、数据层、应用层三部分。

数据服务层主要提供的一些通用的产品和服务，而最底层的基础设施层，我们使用的现在常用的大数据技术组件，比如 hadoop，hive，spark，ES，Redis 等组件。

这里说一下 AI 基础能力这块，说的是专门针对的算法的一个平台级别的设计，其中核心思想是基于 Docker 来进行算法方面的封装和部署应用，达到一键部署和训练的目的。目前我们正在自研这个平台，如果未来设计完成了，可以针对这部分内容进行专题分享。基于新型的架构设计，我们大大的加快算法的迭代速度，加快产出。



之前提到了一些架构面的事情，现在重点说一下我们在算法应用的探索，我们这边整体的应用概览如下：

从此图中，我们的 AI 能力建设大致区分成了深度学习和算法应用这两块。应用层面中机器学习主要说的算法应用，结合业务的场景进行的算法的应用，帮助提升智能化和自助化。知识图谱是针对现有知识的梳理和串联，这部分数据在风控方面用处很多，我们现在在尝试串联一切，通过图谱的方式来去发现数据不一致的地方。

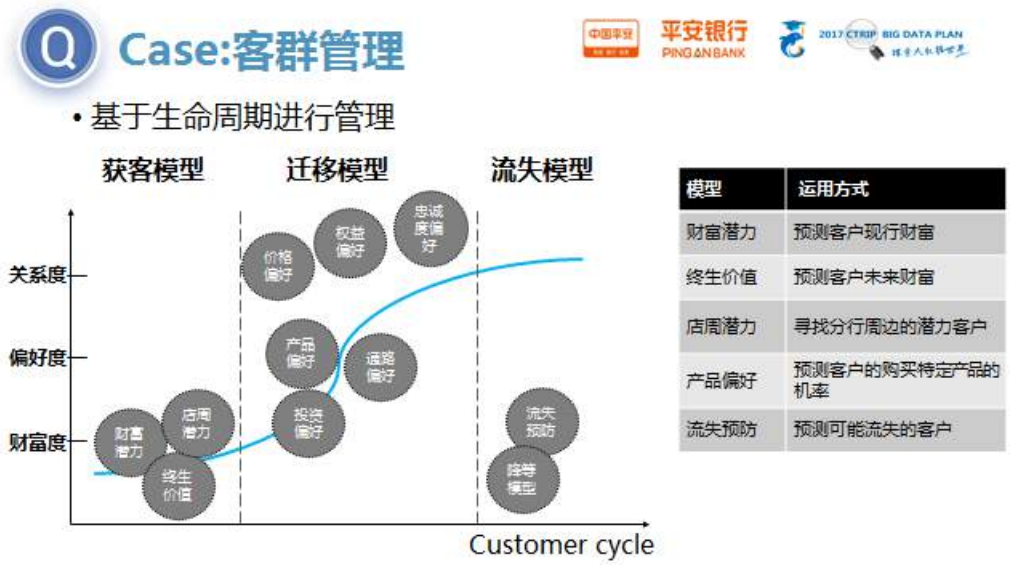
图像、语音和文本是根据现在业务存在的大量人工的事情，可以借助现在新型的技术和方法来提高自动化。文本这块，我们服务于 app 搜索和智能客服这两块。这里提一下文本这块，因为是场景化的需求居多，所以我们会重点发展自有的技术。

关于算法应用这块，我们的原则是：如果利用一些开源的技术短时间内也和市场主流的产品有一定的差距，那么这部分技术可以采用采购的方法来短时间内达到市场主流竞争中去。举个例子，语音的 ASR 方面，科大讯飞的效果最高那么我们就会直接采用讯飞的技术进行 ASR 转换，但是在语音前处理和后处理方面，我们就会自研，因为在语音方面的去噪和增强方面，很多供应商是要做定制化开发的，对于定制化开发这部分，我们会自研，否则我们就只能一直依赖外部供应商，这个从长远来说，对我们是不利的。

说完了面上的应用。我们来看一些具体的例子，有助于理解我们在某些应用是怎么思考的。

先看第一个例子，关于客群管理的。

案例 1：客群管理



从上图可以看到，我们的客群是根据客户的生命周期来进行分析和管理的。横轴是客户的生命周期，从获客、迁移到流失，纵轴是业务的细分维度，这样交叉可以区分出不同的客群出来。根据不同的客群我们会给出不同的 offer 和权益，不同的权益享受不同的待遇。

在客群管理中，我们会建不同的子模型用来对客户进行精准化运营。这些不同的子模型，我们的落地方案基本是走客户画像这套体系，落地在标签系统里面，这样其他人想调用的时候都可以直接拉出相应的结果出来。关于客群经营这块，其实很多公司都有涉及，但是都没有深入挖掘并细化，没有形成一整套的解决方案，都是各个部门各自为战。

根据我们的测试上来说，通过精细化运营的方式比传统的粗放式运营方式，成本和效益最少会提高 5 倍。我们银行也在积极的进行探索，希望能够用更加智能化的方式来提高运营，提升业务产出。

案例 2：客户画像

接下来我们看一下客户画像如何赋能到一线人员。下图是我们在给一线人员开发的客群分析的界面。

可以从上图看到，我们提供了不同的数据职能模块，前面是一些事实性的标签数据，用来表示当前客户的一些基本信息，后面是一些算法给出的推荐和预测结果。

其中需要留意的是，在预测类标签中，我们额外提供了很多的辅助决策类的数据，比如流失预测概率中，我们会提供计算流失概率的重要因素，把这些重要因素展示给一线人员，告诉他们什么因素导致此客户的流失概率较高。同时，我们也会提供模型预估的流失准确率和实际准确率的比较结果，用来发现当前模型是否有比较显著的下降，以方便我们及时的进行模型的更新。

从这个案例中，告诉我们赋能一线的时候，不仅仅需要提供精准预测的结果，还需要提供更多的决策依据，否则无法指导执行的人进行有针对性的改善。当然，提供辅助决策的前提是预测结果是不是直接可以用来决策。接下来我们来看一下直接利用模型结果决策的例子。

案例 3：业务预测

我们接着看比较常规的业务预测

这个是某业务量预测需求，项目背景是去预测未来 3 天的业务量，根据这个业务量我们会进行排班的设计，所以这个需求是直接用预测结果进行决策的例子。

针对这个项目背景和需求，我们先拉取了数据来进行分析，发现历史数据不全，能拉取到的数据就不到半年，所以周期性的规律我们都没有办法捕捉。所以我们尝试了右边的模型融合的方法，尝试了不同的预测方法进行预测，然后再结合规则进行了最终模型结果的输出。

其中历史同天平均值表示的是最近三个月的同一天的业务量预测平均值。

其中规则的方法针对的是月初的情况，我们发现月初的结果和其他的走势很不一样，所以我们针对月初使用了一些固定的规则来进行预测。

通过以上方法，我们可以把预测的绝对偏差控制在 9%以下，在数据量如此小的情况下能达到如此精度，我们觉得还是做的不错的。也在此把方法分享给大家，看一下对大家有没有一些启发。

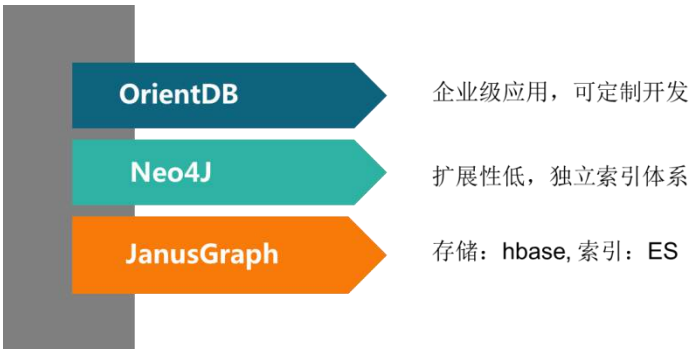
说完了一些预测类的事情，我们接下来说一下我们在图谱方面的尝试，这是个体力活。

案例 4：图谱

上图是我们在图谱方面做的两个尝试，左边的这张图是卡号、进件号码和激活号码之间的聚类结果，我们根据这个聚类结果发现了一些疑似薅羊毛的团伙，并针对这些团伙进行定点的分析，比如地域，发现的结果还是蛮有意思的。

右边这个图是我们根据左图的测试案例进行外衍生出来的知识图谱结果，这个是我们的一个数据产品。我们可以利用这个产品查询到不同电话之间有关系的人、不同卡号之间有关系的人出来，这个产品在对某些风险案件的反查和一些新的规则的发现是有帮助的。

在对图谱的存储上，我们尝试部署了一些图数据库，比如 neo4j、OrientDB 和 JanusGraph。对比结果如下：



OrientDB 和 JanusGraph，我们都在一定的尝试，目前在使用 OrientDB 做了一些地址方面的 POI 存储，用来进行多个三元组的存储。

关于图数据库上，它是一个存储介质。在图挖掘上，我们认为重要的是根据场景来进行分析和挖掘，所以多做一些数据分析和探索是重中之重，存储只是解决了快速部署应用的问题。

我们希望在未来有更多的图谱方面的挖掘和应用，万物串联是我们在做各种应用的基础能力。

三、一些思考

在我们做一些应用实践的时候，我们也有一些感悟，这些感悟对我们加快数据赋能公司也有一些帮助。

第一个感悟是:

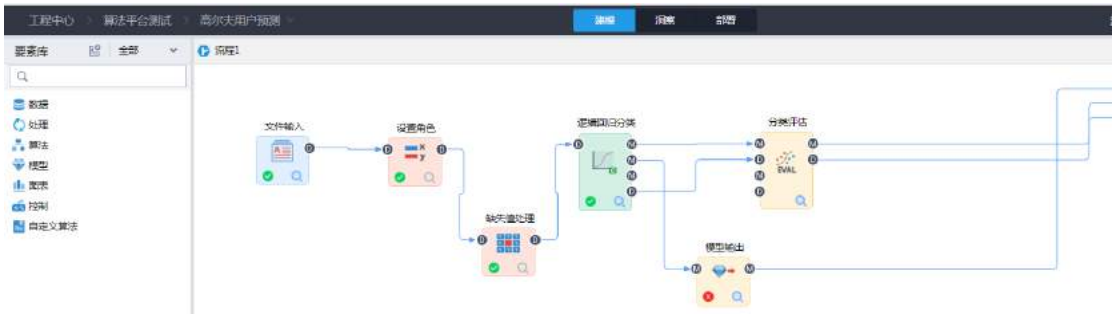
在很多算法落地的场景中，我们都比较需要数据产品经理的角色，比如客户画像、客群管理、业务预测等，数据产品经理会梳理好业务和数据对接的场景，让算法工程师的工作职能变得相对专一一点，他只要了解业务，数据梳理并把模型开发上线就可以，而在最终的页面展示和业务系统对接和沟通、协调上面会由数据产品经理去完成，他们同时会兼任一些项目管理的工作。

通过这个分工，可以让整个算法项目进度完成的更加顺畅。我以前带团队的时候，这个角色一般是由我来承担的，即分组经理承担，但是分组经理因为顾及的面较多，涉及到多个项目的跟进和资源协调，所以在单个项目完成度上打了一定的折扣。

现在，通过新增的数据产品经理角色，可以加快数据落地和闭环过程。算法应用人员、数据产品经理、分组经理。外部系统开发各司其职，很好的完成了数据赋能的任务。

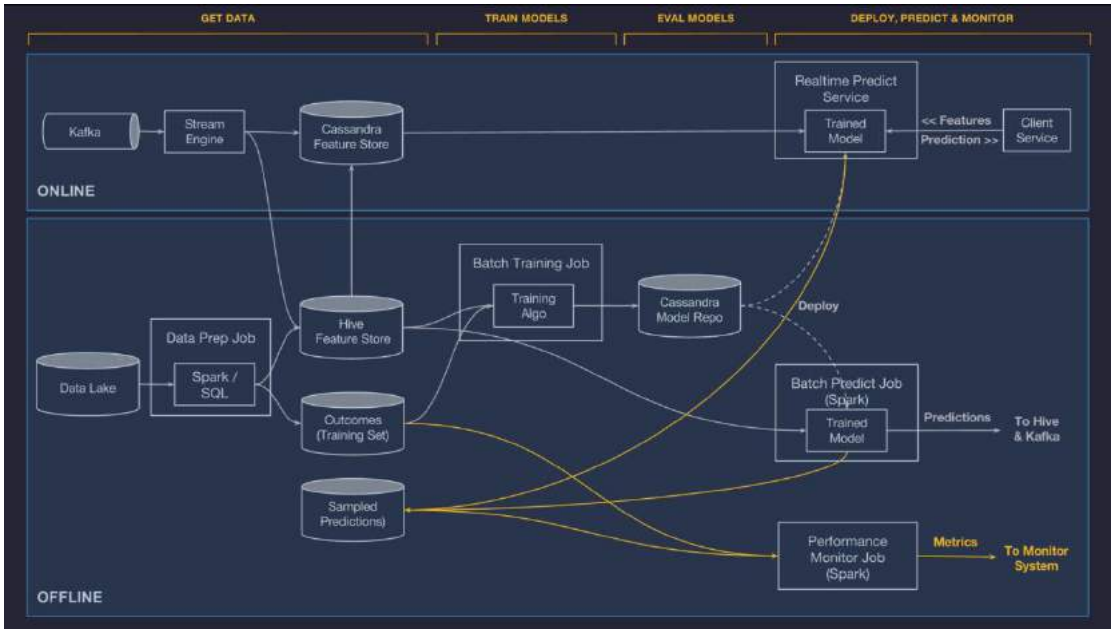
第二个感悟是:

我们需要一个统一的 AI 算法平台，集模型训练、部署、可视化于一体。通过这个平台可以减少很多不同算法工程师的重复工作。



上图是我们目前使用的一个算法平台，挺好看的，但是功能上需要完善很多，所以我们也在设计一个新的算法平台，用来减轻一些重复工作量。现在外界也有很多公司也在开发相应的产品，其实目的就是想通过减少使用算法的门槛来提高算法在公司的使用情况。从目前来看，很多算法的壁垒和门槛已经慢慢的变低了，所以未来对算法工程师的要求就更加严格了。

最后，说一下理想中的算法平台可以参考一下 UBER 设计的平台模式，从数据源头到部署运维一体化，覆盖了 online 和 offline 渠道。


GET DATA
TRAIN MODELS
EVAL MODELS
DEPLOY, PREDICT & MONITOR
Realtime Predict Service
Kafka
Stream Engine
Cassandra Feature Store
Trained Model
<< Features
Prediction >>
Client Service
ONLINE
Batch Training Job
Training Algo
Cassandra Model Repo
Deploy
Data Prep Job
Hive Feature Store
Data Lake
Spark / SQL
Outcomes (Training Set)
Batch Predict Job (Spark)
Trained Model
Predictions
To Hive & Kafka
Sampled Predictions)
Performance Monitor Job (Spark)
Metrics
To Monitor System
OFFLINE

携程“小诗机”背后的机器学习和自然语言处理技术

[作者简介]孙玉霞，携程技术中心基础业务部算法工程师，南京师范大学硕士，主要研究方向为自然语言处理，参与小诗机，智能客服，产品强化等相关项目。

2017 年年初，携程推出了“诗情画意”小诗机，让机器能够“理解”，“欣赏”用户拍摄的照片，并基于小诗机自有的庞大知识库体系，写出符合图片的意境和内容的古诗。

目前，小诗机和上海地区诗人盲测实验结果显示，小诗机已经达到人类诗人的水平，专业评委和大众评委无法区分出哪些是小诗机的“大作”，整体评分排名靠前。

同时，小诗机还具有基于图片检索古诗，作藏头诗，宝塔诗等功能。小诗机是人工智能在人类创造力和理解力上的挑战，让人们在旅游的同时拥有诗和远方。

运用了知识图谱和图片识别、自然语言深度学习方面的前沿技术，是大数据和人工智能的结合产物。我们使用机器学习进行景点实体关系和特征的提取，构建大规模的旅游景点知识图谱，包括全球数万景点、地区、美食、天气等数据；使用自然语言技术基于 30 多万首古今诗篇进行语义层级的主题诗歌自动生成。强大的知识支撑和先进技术的融合造就了“诗情画意”的小诗机。部分样例如下：

基于景物识别结果成诗。如图 1，图片包括楼，林木，梅花等元素，成诗时结合了当时的季节天气等信息，如日色，冬晴光。

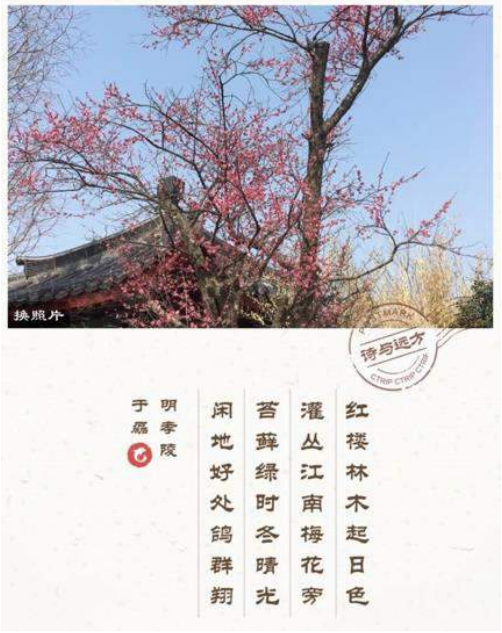


图 1 景物识别信息成诗

基于人物性别，年龄，表情等成诗。如图 2，图片中是一个表情严肃的年轻男人。

图 2 人物识别信息成诗

基于景点信息成诗。如图 3，成诗结果融合了西湖的特色和周边景点，比如曲院，孤山，寒湖，莲叶，茶园，寺庙等。

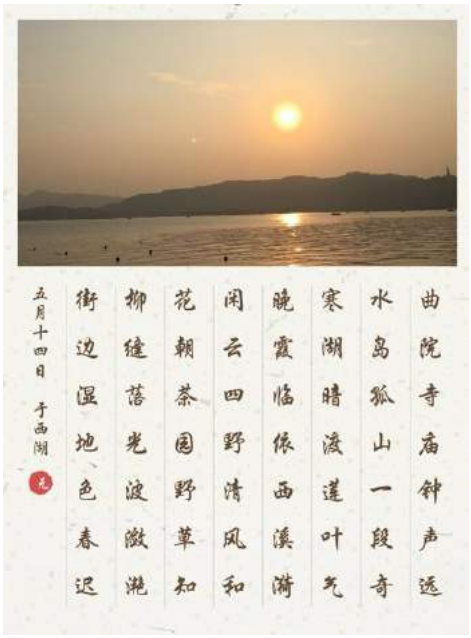


图 3 景点信息成诗

为了评估小诗机的效果，我们邀请 5 位文学爱好者（高校文学院学生或者老师，具有较好的文学功底）跟小诗机进行 PK：基于 11 组主题赋诗，文学爱好者有一天的时间进行诗歌创

作，而小诗机即兴创作，同时邀请两位专业诗人和四位大众诗人作为评委。11 个主题的评比结果如表 1，小诗机在 2 个主题下获得第一名，3 个主题下获得第二名，2 个主题下获得第三名，综合排名第三，通过 了诗歌创作的“图灵测试”。

表 1　基于 11 场次的评比结果

	冠军	亚军	季军
赵**	5	0	1
邹**	2	6	2
小诗机	2	3	2
卿**	2	2	2
沈**	0	0	2
张**	0	0	0

下面我们通过整体流程介绍、知识图谱构建、图片识别、成诗引擎这四个方面来揭开“小诗机”的面纱。

一、整体流程

小诗机的基本流程如图 4，主要包括知识图谱、图片识别和写诗引擎。

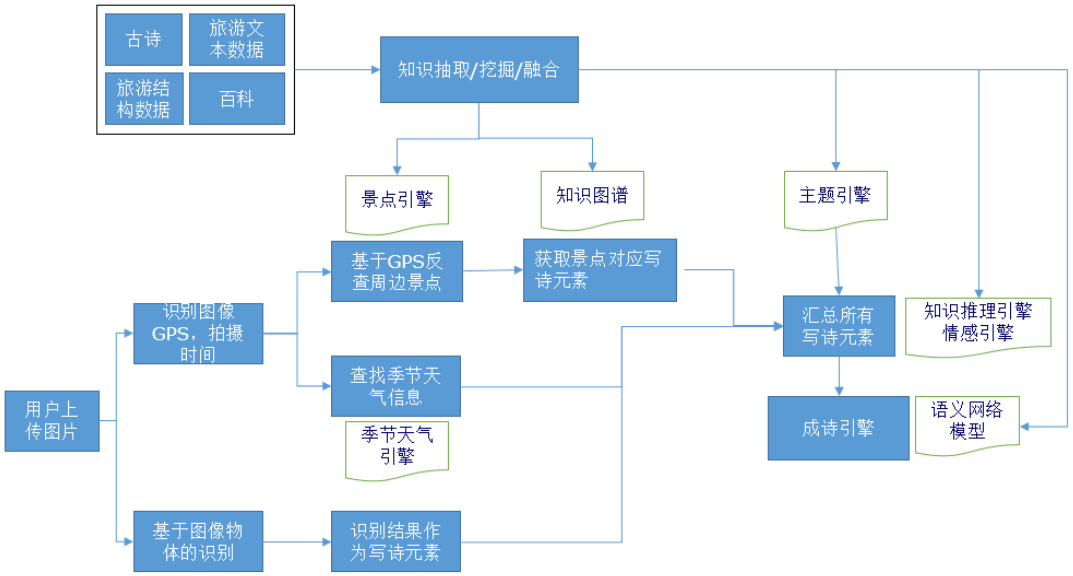


图 4　小诗机的整体流程

二、知识图谱构建

2.1 知识来源

为了构建全面的旅游行业知识图谱，我们融合了多个维度的知识源，一方面是携程特有的大

量的旅游行业的数据和景点数据，另一方面是互联网上关于城市、历史、传说、民俗方面的数据，具体分为如下三个部分：

非结构化数据： 携程的景点简介，用户游记，用户评论等文本信息。

半结构化数据：以维基百科，百度百科为代表的大规模知识库，包含了大量结构化和半结构化数据，可以高效地应用到知识图谱中。

结构化数据：包括自由行，团队游，酒店，景点，用户意图等较为全面的结构化旅游数据。

针对结构化数据，除了使用一些相关的维度信息，如景点所属的城市，同时也会进行一些相关的统计分析，实现深入的信息提取和筛选，如对每个城市下重要的景点进行排名。

2.2 知识抽取

文本数据的知识抽取会涉及到许多自然语言处理的基本技术，包括分词，词性标注，依存句法分析，语义角色标注，命名实体识别等，实现对文本数据的清洗和结构化。主要的提取目标包括如下几个部分：

实体抽取：利用 CRF++和字典结合的方式，从文本中自动识别人名，地名（国家，省市州，城市，景点），机构，节气，时间等实体[16]。

关系抽取：主要利用一些启发式的规则和半监督的方法来从新学到的实例中学习新的 pattern 并扩充 pattern 集合挖掘各个实体之间的关系。

主题抽取：为了构建更全面的旅游知识体系，同时使用改进的 tfidf，卡方 + TextRank，LDA 等方法[17]综合进行文章主题，关键词，摘要的抽取，从而实现景点实体和特征实体的关系挖掘。

2.3 知识融合

我们的数据来自于不同的数据源，需要对各个数据源提取出来的知识进行融合。实体的融合和匹配是多数据源融合的核心，一方面通过语义向量维度/字面维度的相似度来进行实例、属性、概念等的匹配[19][20]，另一方面结合实体对应的相关属性以及结构信息进行多维度多层次的匹配[18]，最终利用自定义权重进行融合。

2.4 知识推理

在构建的实体关系库上，使用基于符号逻辑推理方法，推理出新的实体关系以及约束关系。

三、图片识别

对于计算机视觉任务而言，大多模型都是基于卷积神经网络（Convolutional Neural Network, CNN）。LeNet5[12]是最早的卷积神经网络，但是由于硬件的限制并没有被广泛的采用。在视

觉领域竞赛 ILSVRC 2012 上，AlexNet[14]的出现使得图片分类的正确率得到了大幅度的提升，从传统的 74.2%提升到 83.6%，它是 LeNet5 的一种更深更宽的版本，首次利用 GPU 进行加速运算，并成功应用了 ReLU、Dropout 和 LRN 等技巧。在此基础上，后续出现了 VGGNet[13]、GoogleNet[11]、ResNet[15]等，网络的深度和宽度不断提升，准确率也达到并超过了人类肉眼辨识的水平。

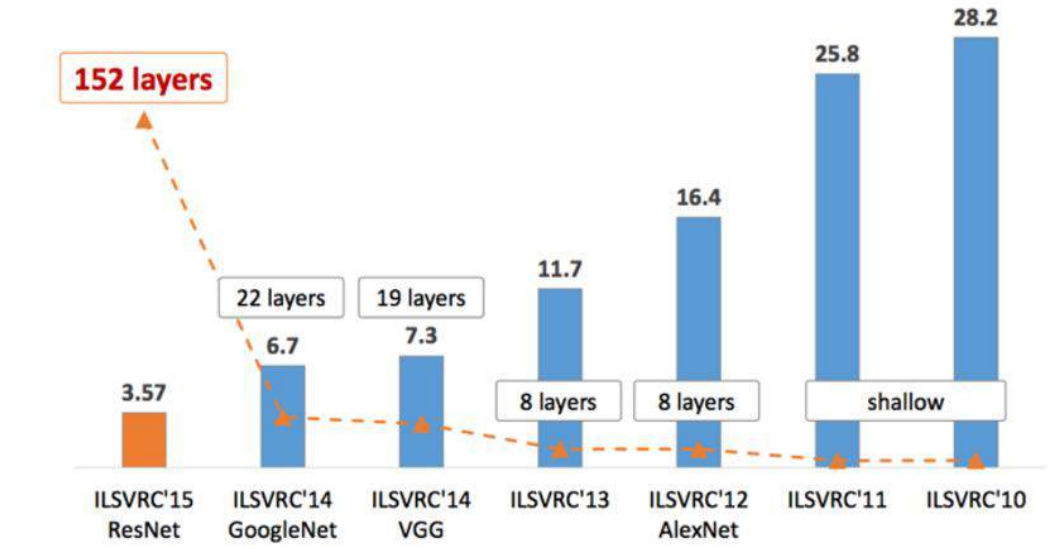


图 5 ILSVRC 历年的 Top-5 错误率

计算机视觉识别领域具有大量的公开数据集，这些数据集大部分具有高质量的标注。迁移学习能很好地运用这些数据，让我们在拥有少量标注数据的情况下取得较好的图片识别结果[1]。

使用 inception-v3[2]模型+迁移学习的方式，进行快速类别扩充。GoogLeNet 在扩大网络提升效果的同时拥有更好的计算效率，它是一个庞大的网络结构，如图 6。

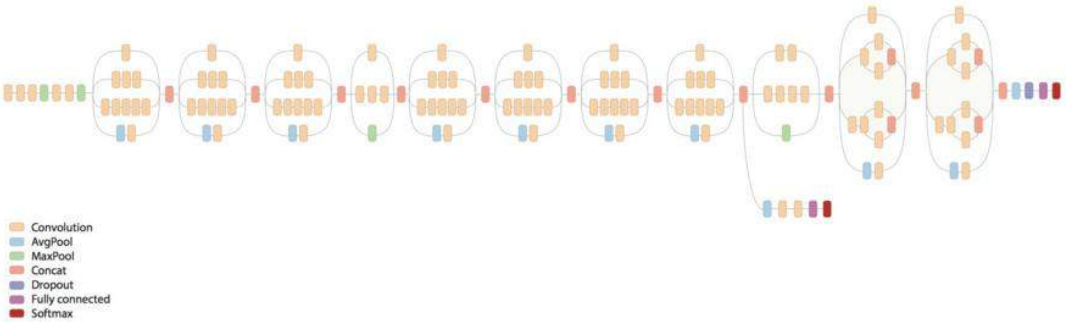


图 6 GoogLeNet 神经网络结构

其中 inception 模块是其核心，具体 inception 结构如图 7。

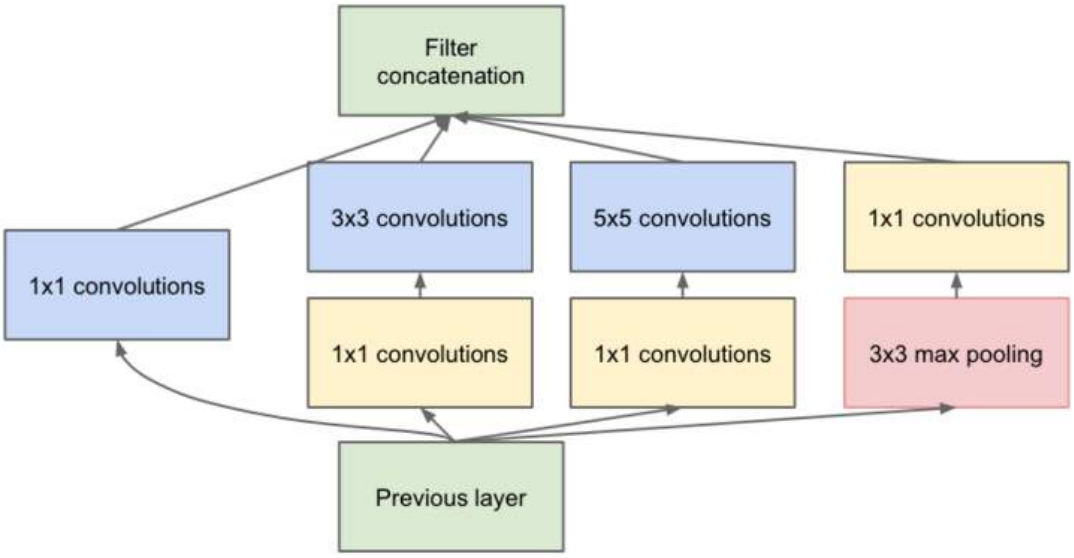


图 7 Inception 神经网络结构

使用基于 ImageNet 的 Inception-v3 模型，由于我们多分类数据量比较小，同时数据内容 ImageNet 内容有一定的差异性，因此并没有直接利用网络的高层进行特征的抽取，而是打开高层网络和浅层分类网络同时进行训练，结果表明，该方法比直接利用网络高层进行特征提取效果提升了 8 个百分点。这是由于网络的高层是对特征的进一步抽象和提取，跟原任务数据具有较高的耦合性，而底层网络抽取的特征则为一些基础特征，在此基础上融合分类网络的训练，对新任务具有更好的拟合性。最终在旅游数据集上，我们在内部的类别标签体系上取得了 92.5%的 mAP。

在此同时，相似类别语料的添加也显得尤为重要，譬如对于“温泉”类别特别容易识别成“小河”或者“游泳池”，添加相近的类别语料供算法进行学习，能够有效提升识别结果的精确度。如图 8，列出了图片的识别结果。

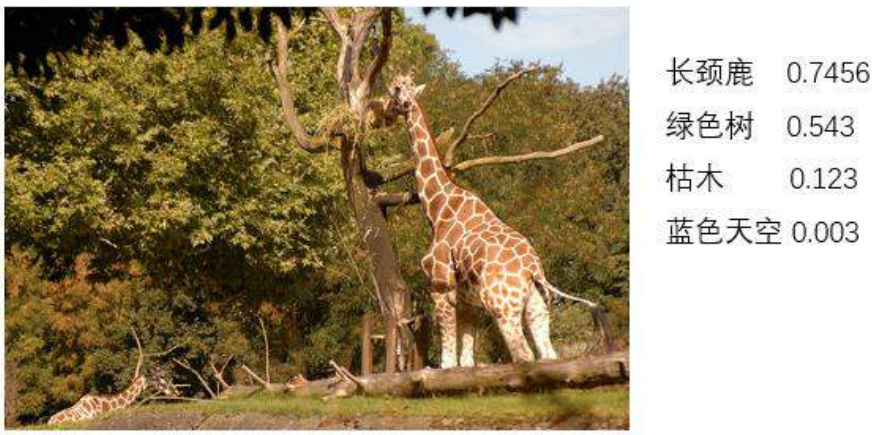


图 8 图片识别结果

四、成诗引擎

传统诗歌生成主要借助统计模型，同时结合人工规则。在统计机器翻译的方法中[4]，使用统计机器翻译模型结合韵律规则进行下一句的生成。在遗传算法中[3]，则根据开发者对诗歌的理解，人工定义韵律，流畅性，主题相关性等各个评价因子，构建评估函数，同时使用统计语言模型进行诗歌的自动生成。

随着深度学习技术的日益发展，使用神经网络进行诗歌自动生成是现在的流行趋势。传统的诗歌自动生成模型，由于古诗语言表达的简洁性和语料数量的限制，基于统计的语言模型具有比较严重的稀疏性。在深度学习中，利用 RNN 训练和获取诗歌的语言模型，在一定程度上缓解了该问题[5]。

在此同时，encoder-decoder 框架做为成诗的基本模型，利用 encoder 模块进行历史内容[6][8]、主题内容[7][8]的编码，decoder 模块进行诗句的生成，对主题关联性和语义连贯性，使用不同方法进行尝试，从而构建相关模型。模型[8][9]根据给定的关键词首先对每句话进行主题的规划，从而避免在逐句成诗过程中，主题关联性的弱化。模型[8][6]利用 attention 机制，在成诗过程中进行主题、历史生成内容的融合，而模型[7]则是使用 hierarchical 的 RNN 框架，进一步保证整体主题和语义的统一性。

对于小诗机而言，需要结合多重元素空间和多维主题，而且诗歌语料和主题体系相关性存在缺失，比较难进行端到端的深度学习模型的应用，因此结合了传统模型和深度学习模型，具体从如下几个维度进行了综合优化：

主题规划和相关性：

- 对景点/图片与主题元素的相关性进行打分，多个维度进行综合排序，使整个主题元素空间在全局上权重化，同时结合一定的写诗惯例，基于概率模型动态预先确定每句话的主题元素。
- 基于关键词语义空间的相关主题词获取。

语义流畅性：

- 利用 RNNLM[21]在一定程度上缓解传统统计语言模型的稀疏性，提升诗歌的流畅性，如图 9。

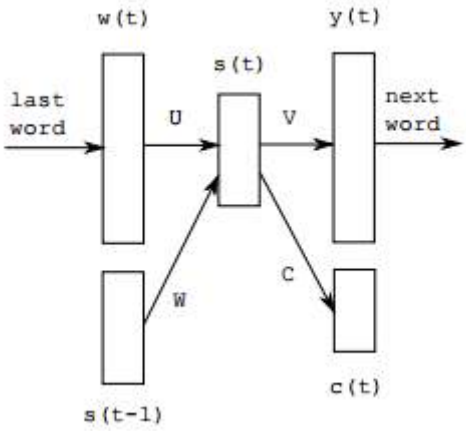


图 9　RNN based language model with classes

- 基于语义层级体系获取语言模型，从多个层次上来把握诗歌的语义搭配粒度， 使得表达更加的流畅。

选优框架

- 基于贪婪算法和局部最优的二排算法进行诗歌的自动生成，从而在保证性能和诗歌基本韵律的基础上，一定程度上确保诗歌主题的相关性和表达的多样性。

- 遗传算法也是对解空间寻求最优化，把古诗的生成看作是一个状态空间搜索问题。将主题相关性，诗歌流畅度、韵律要求等综合融入到评估函数中，适应度函数的设计则较为复杂，同时遗传算法速度较慢，更适合对于速度要求不高的场景。

五、总结

小诗机在总体上，是从技术和产品维度上的创新和探索，一方面构建了大规模全面的旅游知识图谱，另外一方面结合深度学习的方法进行基于图片的成诗，使得诗具有较好的主题性和流畅度。

在此基础上，后续会进行进一步的标签细化，包括知识库和图片识别两个方面，使得小诗机拥有更加宽阔的旅游视野和更加细腻的观察维度。在此同时，进行成诗引擎的深入优化，适应更加多元化的元素，真正引领智慧旅游。

【相关文献】

[1] http://cs231n.github.io/transfer-learning/
[2] Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the Inception Architecture for Computer Vision[J]. 2015:2818-2826.
[3] 周昌乐，游维，丁晓君．一种宋词自动生成的遗传算法及其机器实现[J]．软件学报，2010, 21(3):427-437.
[4] He J, Zhou M, Jiang L. Generating chinese classical poems with statistical machine

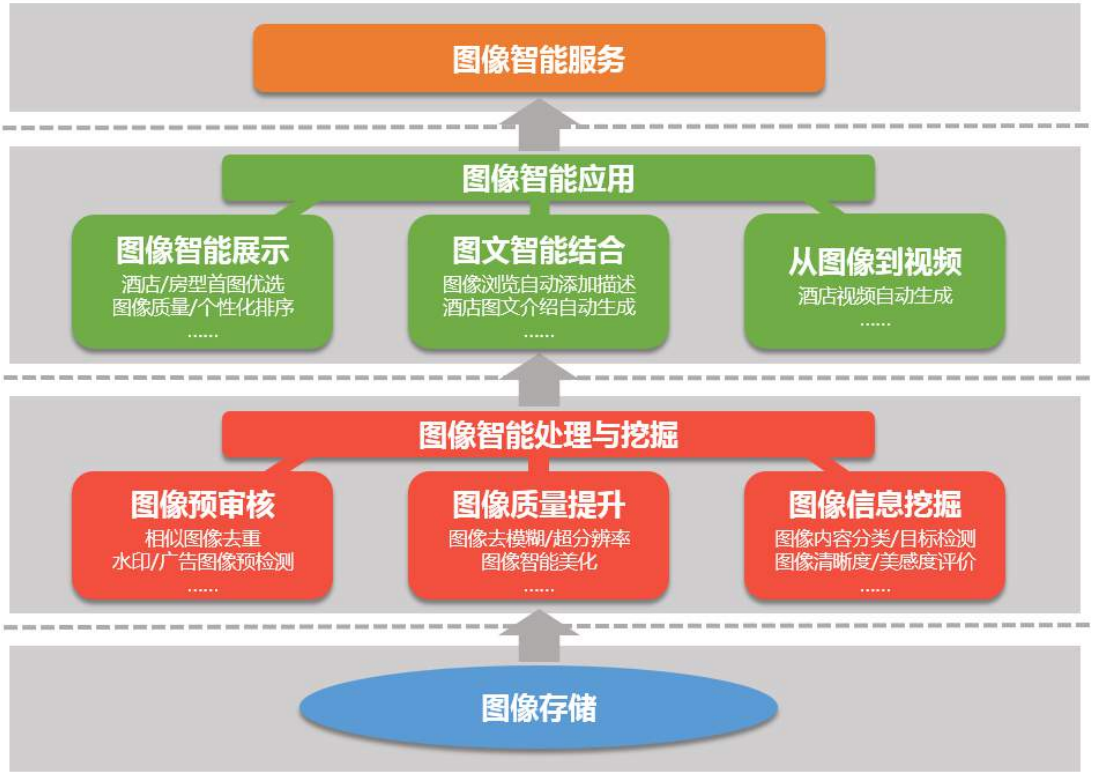
translation models[C]. Twenty-Sixth AAAI Conference on Artificial Intelligence. AAAI Press, 2012:1650-1656.
[5] Mikolov T, Karafiát M, Burget L, et al. Recurrent neural network based language model[C]. INTERSPEECH 2010, Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September. DBLP, 2010:1045-1048.
[6] Wang Q, Luo T, Wang D, et al. Chinese song iambics generation with neural attention-based model[C]. International Joint Conference on Artificial Intelligence. AAAI Press, 2016:2943-2949.
[7] Rui Y. i, Poet: Automatic Poetry Composition through Recurrent Neural Networks with Iterative Polishing Schema. Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence. IJCAI-16,2238-2244
[8] Wang Z, He W, Wu H, et al. Chinese Poetry Generation with Planning based Neural Network[J]. 2016.
[9] Ghazvininejad M, Shi X, Choi Y, et al. Generating Topical Poetry[C]. Conference on Empirical Methods in Natural Language Processing. 2016:1183-1191.
[10] https://zhuanlan.zhihu.com/p/25084737?columnSlug=easyml
[11] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions[C]. IEEE Conference on Computer Vision and Pattern Recognition. IEEE Computer Society, 2015:1-9.
[12] Yann LeCun, Leon B, Youshua B, Patrick H. Gradient based Learning Applied to Document Recognition. PROC OF THE IEEE, NOVEMBER 1988
[13] Simonyan K, Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition[J]. Computer Science, 2014.
[14] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks[C]// International Conference on Neural Information Processing Systems. Curran Associates Inc. 2012:1097-1105.
[15] He K, Zhang X, Ren S, et al. Deep Residual Learning for Image Recognition[J]. 2015:770-778.
[16]何炎祥, 罗楚威, 胡彬尧. 基于 CRF 和规则相结合的地理命名实体识别方法[J]. 计算机应用与软件, 2015, 32(1):179-185.
[17]Wu W, Zhang B, Ostendorf M. Automatic Generation of Personalized Annotation Tags for Twitter Users.Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 2-4, 2010, Los Angeles, California, USA. DBLP, 2010:689-692.
[18] Otero-Cerdeira L, Rodríguez-Martínez F J,Gómez-Rodríguez A. Ontology Matching: A Literature Review [J]. Expert Systems with Applications, 2015, 42(2):949–971.
[19] Lambrix P, Tan H. SAMBO—A System for Aligning and Merging Biomedical Ontologies [J]. Web Semantics Science Services and Agents on the World Wide Web, 2006, 4(3):196-206.
[20] Li J, Wang Z, Zhang X, et al. Large Scale Instance Matching via Multiple Indexes and Candidate Selection[J]. Knowledge-Based Systems, 2013, 50(3):112-120.
[21] T Mikolov , S Kombrink , A Deoras ,L Burget , Černocký. RNNLM --- Recurrent Neural Network Language Modeling Toolkit.

携程图像智能化建设之路

[作者简介]李翔，携程数据智能部信息科学组图像技术负责人，专注于计算机视觉和机器学习的研究和应用，现阶段致力于酒店图像智能化，在包括 ICCV 和 CVPR 在内的学术会议和国际期刊上发表 10 余篇论文。

携程作为 OTA 行业的领跑者，拥有全球百万家酒店数以亿计的酒店图像，酒店图像数量还在以每天数十万的速度增长。面对海量酒店图像，如何完成智能处理与挖掘，大幅减少图像的人工干预，又如何实现智能应用，改善用户获取酒店信息的速度、准确性和完整性，提高用户满意度，这些都成为急需解决的问题。

相比学术界追求的模型创新性，我们更加关注技术实践在落地场景的效果，力求以简单而有效的方法来解决实际业务问题。为解决上述问题，我们从 0 到 1，围绕酒店图像智能化进行了一系列研究和探索。当前酒店图像智能化的整体架构如下图所示。



其中，通过图像智能处理与挖掘，能够节省大量人工处理图像的成本，挖掘丰富的图像信息，在此基础上，通过图像智能应用可以进一步有效地为用户和酒店创造巨大价值。下面我们将围绕这两个部分展开，从多个具体实践出发，分享携程酒店图像智能化从 0 到 1 的建设之路。

一、图像智能处理与挖掘

图像智能处理与挖掘是酒店图像智能化的基础，包括图像预审核、图像质量提升和图像信息挖掘三个环节。

1.1 图像预审核

图像预审核是整个酒店图像智能化的第一步，旨在通过一系列图像技术，辅助人工高效地完成海量图像的审核工作，减少人力成本的投入，当前包括相似图像自动去重、水印/广告等视觉可见的不合规图像的预检测等等。下面介绍其中一些实践。

相似图像去重

酒店图像之间的相同/相似主要表现为 1）尺寸形变；2）裁剪残缺；3）色彩变化；4）旋转变化；5）拍摄视角移动等多种情况，如下图所示。

相似图像去重一般分为 1）图像特征表达的提取和 2）图像之间相似度计算两个主要步骤。对于图像特征表达的提取，常见的手工设计特征有颜色、纹理、HOG、SIFT 和 SURF 等；此外基于深度学习的深层特征表达也经常被使用。对于图像之间相似度计算，常见的无监督距离度量方法有欧式距离、曼哈顿距离和余弦距离等；常见的有监督距离度量方法有 LMNN、KISSME、LFDA 和 MFA 等。然而这些方法基于浮点特征计算相似度，计算速度普遍较慢，因此通过哈希学习方法将图像特征转换为二元编码，再利用汉明距离进行相似度的快速计算更加符合工业界对图像数据处理速度的要求。

对于相同/相似的酒店图像，大部分全局特征（比如颜色、纹理和 HOG）不能很好地解决图像裁剪残缺和旋转变化等问题；一些局部特征（比如 SIFT 和 SURF）与基于深度学习的特征虽然表达效果较好，但是由于特征提取复杂，计算速度过于缓慢。

针对以上特征提取方法存在的缺陷，我们采用一种快速特征点提取和描述算法 ORB 作为图

像的特征表达，并使用汉明距离完成相似度计算。ORB 特征具有以下优点：1）特征提取速度快；2）在大多数情况下，去重效果能够与 SIFT/SURF 持平；3）提取的特征直接是二元编码形式，无需使用哈希学习方法就可以直接利用汉明距离快速计算相似度。

在真实应用中，我们还对其进一步优化，从而弥补 ORB 特征不具有尺度不变性的不足，也降低了形变和模糊等因素对 ORB 特征的影响，在保证性能的同时，提升了图像去重的准确率。

图像水印检测

水印在图像中的视觉显著性很低，具有面积小，颜色浅，透明度高等特点，带水印图像与未带水印图像之间的差异往往很小，区分度较低。

我们将图像水印检测问题转化为一种特殊的单目标检测任务。在深度学习兴起前，可变形部件模型（DPM）一直是流行的目标检测方法。深度学习出现后，以 R-CNN、SPPNet、FastR-CNN、FasterR-CNN、SSD、YOLO/YOLO2 等为代表的一系列基于深度卷积神经网络的目标检测方法成为主流。

一个鲁棒的目标检测网络，需要大量的标签数据来支撑，然而收集和标记一个大规模水印目标检测数据集是一件十分耗时耗力的工作。为了解决该问题，我们设计了一套训练数据自动生成和自动标记的数据集制作系统，以较小的人力投入建立了一个多元化的大规模水印目标检测数据集。

在该数据集上，我们进一步对比 FasterR-CNN、SSD 和 YOLO2 三种主流的目标检测方法。通过对性能和效果的综合评估，我们选择在 YOLO2 的基础上进行改进，使其更加适应水印单目标检测任务，从而实现最终的水印图像检测器。基于该检测器，我们实现对数据集中常见上百种水印的完美检测，对数据集中未出现的水印也展现出很好的检测效果。

1.2 图像质量提升

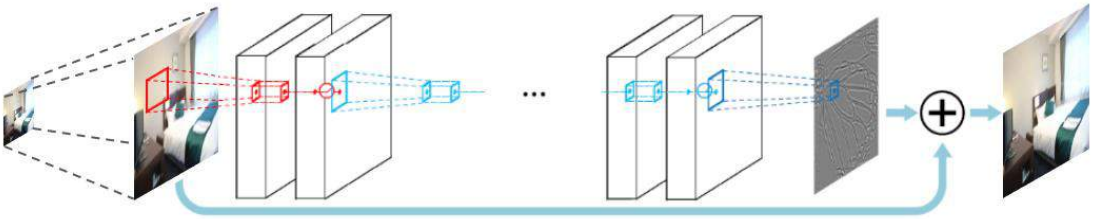
图像质量提升旨在通过一系列图像技术，改善酒店图像的质量，进一步减少人工对图像的处理成本，当前包括图像去模糊、小图放大和图像智能美化等等。下面主要分享我们在酒店小图放大中的一些实践。

酒店小图放大

酒店如果存在低分辨率图像通常会被转为高分辨率图像展示给用户，从而使用户获取更多图像细节，更好地增加对浏览酒店的了解。然而，低分辨率图像直接放大，图像会显得模糊，图像细节信息很难恢复。

为了令低分辨率图像放大后变得更加清晰，我们引入图像超分辨率技术。最简单的图像超分辨率方法是图像插值，但插值后的图像很容易形成锯齿边缘，细节恢复效果较差。传统的图像超分辨率方法一般采用稀疏表示和字典学习的方式来实现，利用大量高-低分辨率样本对作为先验信息进行图像细节恢复，常见的方法有 SR、ANR、SF 和 A+等。随着深度学习的发

展，通过全卷积神经网络学习低分辨率图像到高分辨率图像的端到端映射函数成为主流方法，其中以 SRCNN、DRCN、VDSR、SRResNet、SRGAN 和 SRDenseNet 等方法为典型代表。



在真实的小图放大场景中，我们选择 VDSR 来构建超分辨率网络，结构如上图所示。具体地，我们组合多种倍数的图像样本进行混合训练，令一个模型能够同时适应多种不同倍数的超分辨率。

然而，在实际应用中我们还遇到一些问题：1）超分辨率网络使用的损失函数一般是最小均方误差（MSE），该函数使重建结果有较高的信噪比，但是缺少高频信息，会使图像出现过度平滑的纹理。2）真实的低分辨率酒店图像往往存在有损压缩，图像本身具有块效应，直接使用超分辨率网络恢复细节，会使图像的块效应更加严重。

为了进一步提升图像超分辨率的效果，我们在 VDSR 基础上，针对上述问题做了一系列改进，在保证网络输出的图像更加自然的同时，也大幅降低了块效应的影响。利用上述模型，低分辨率图像的高分辨率重建能够被高效地实现，酒店图像质量得到明显提升，人工处理图像的成本大幅减少。

1.3 图像信息挖掘

图像信息挖掘旨在通过一系列图像技术，自动快速地挖掘出图像中蕴含的丰富内容，为每张酒店图像都建立一个信息档案，为下一步图像智能应用打下坚实的基础，当前包括图像内容分类、图像多目标检测和图像质量评价等等。下面对其中的一些实践做简单介绍。

图像内容分类

酒店图像是对酒店各方面信息的直观展示，因为我们需要尽可能地帮助用户方便快捷地发现他们想要浏览的图像内容，所以酒店图像的内容分类显得尤为重要。

随着深度学习的出现，尤其是卷积神经网络的兴起，利用大量已经标注类别的酒店图像样本直接训练一个深度卷积神经网络，比如常用的 AlexNet、VGGNet、ResNet、DenseNet 和基于 Inception 的一系列网络等，就可以实现对酒店图像的分类。然而，如果通过人工标注大量训练样本代价极大，如果训练样本不足则会导致网络过拟合。为了能够实现在标注少量酒店图像的情况下达到良好的分类效果，我们利用深度网络有效的迁移学习能力，对在大规模数据集上已经预训练的网络权重进行微调。

在实际应用中，我们没有借助应用最为广泛的 ImageNet 数据集，因为该数据集图像的内容和酒店图像差异过大，影响了网络迁移学习的效果。为了尽可能的提升网络的迁移能力，我

们借助与酒店图像内容最为接近的自然场景图像数据集上预训练的 VGGNet 作为初始设置。同时，我们进一步利用水平翻转、随机裁剪和色彩抖动等方式对标注的小规模酒店图像数据集进行数据增强。

基于以上的方法，我们实现了酒店图像内容十余个类别的准确区分，为之后图像智能应用做好了准备。

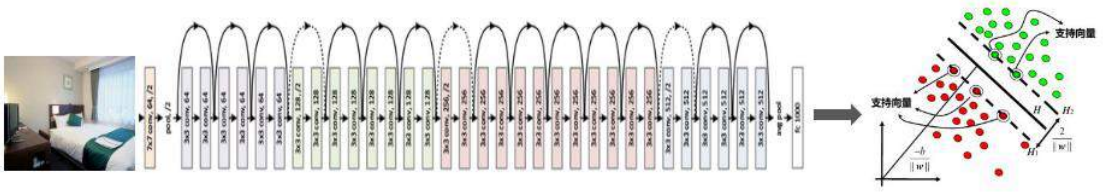
图像质量评价

在上一节中，我们介绍了通过酒店图像分类模型来挖掘图像的类别信息。接下来我们需要更进一步对所有酒店图像进行质量评价，为每张酒店图像计算质量分数来表征其质量的高低。

起初我们选择清晰度这一客观指标作为图像质量评价的标准，然而我们发现仅以清晰度作为图像质量评价的标准存在不足，清晰度高但内容不好看的图像为数不少。因此我们更希望能够从美学角度来对图像质量进行评价。

图像美感度是一个非常主观的概念，很难有一个统一的标准去量化，为了能够尽可能准确地量化图像的美感度分数，我们选择深度学习的方法来实现美感度评价。具体地，我们把计算分数的回归问题转化为判断图像好看与否的分类问题，为了把模型的判断结果转化为美感度分数，我们在模型的最后一层输出图像被判断为好看的概率，作为图像的美感度分数。通过这种方法可以很直观地把一个二元分类器的输出结果转化成打分的结果。

在实际应用中，我们又再次遇到了同样的问题，缺少大量标注了好看/不好看标签的训练图像。由于在酒店图像分类中，我们利用卷积神经网络强大的迁移学习能力取得了成功，所以我们决定继续沿用这种方法。由于酒店图像的美观度受到内容、色彩和构图等多方面的影响，所以我们不再像酒店图像分类那样只借助内容单一的场景图像数据集，而是将包罗万象的 ImageNet 数据集和场景图像数据集混合进行 ResNet 模型的预训练，力求让尽可能多的图像参与层数更多的 ResNet 模型的学习，令更深的网络能够记住更多图像的内容，对图像内不同区域之间关系的理解更加深入，从而进一步提高网络在美感度评价上的迁移能力。



在数据集标注的过程中，由于图像好看与否主观性依赖很强，所以我们综合了多人评判结果作为每张酒店图像好看与否的标注。由于图像经过翻转、裁剪或色彩抖动都会令图像的美感度发生变化，所以我们未对数据集进行数据增强。为了防止直接微调网络出现过拟合，我们转而利用特征迁移，在 ResNet 的深层特征表达基础上，训练支持向量机实现酒店图像好看/不好看的二元分类模型，模型结构如上图所示。通过图像质量评价，我们获取了酒店图像的质量分数，为后续图像智能应用提供了重要依据。

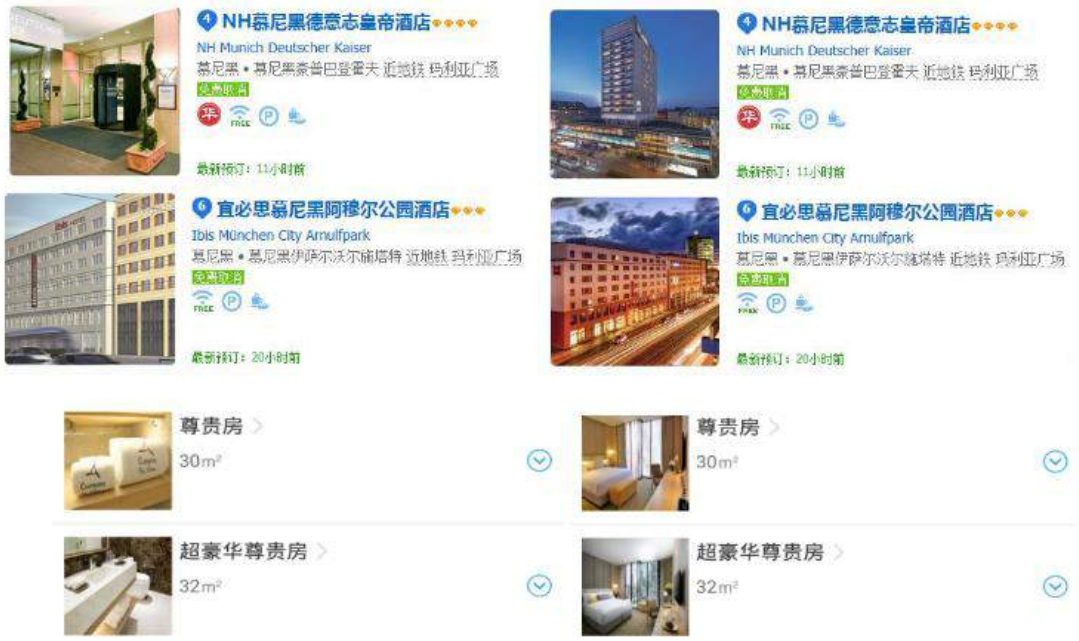
二、图像智能应用

图像智能应用能够为用户和酒店创造巨大价值，是图像智能化的重要组成部分，当前包括图像智能展示、图文智能结合和酒店视频等应用场景。

2.1 图像智能展示

酒店和房型的首图如何挑选才能提升用户的满意度，酒店图像如何排序才能使用户快速获取想要的酒店信息。我们针对这些问题，对图像进行一系列智能展示，在包括酒店/房型的首图优选、酒店图像分类展示和质量/个性化排序等方面做了诸多尝试，努力提升用户体验。

酒店/房型的首图是用户对酒店/房型的初始印象，除了商家或运营人工指定首图外，绝大部分酒店和房型的首图都是由机器进行挑选。为此，我们基于图像智能挖掘获取的图像分辨率、内容类型、清晰度/美感度分数，设计了完整的首图优选模型，力求把图像多种不同信息自适应融合起来，为酒店挑选最优的酒店图像展示给用户。



通过我们的首图优选模型，酒店和房型的首图质量大大提高，对比效果如上图所示（左为原版，右为新版）。更优质的酒店和房型首图使得用户预订转化率显著上升，在提高用户满意度的同时，也给酒店带来了价值。

2.2 图文智能结合

在图像单一展示的基础上，我们还进行了一些图像和文本智能结合的探索，包括图像自动添加描述和酒店图文介绍自动生成等工作。

我们尝试对酒店图像自动添加描述，使用户在浏览图像的过程中能够浏览文字，加深其对图像内容的了解。然而，利用基于深度学习的 ImageCaption 模型从图像得到的描述性文字显得生硬而平淡，直接展示给用户很不自然。我们在此基础上进行改进，进一步结合海量用户

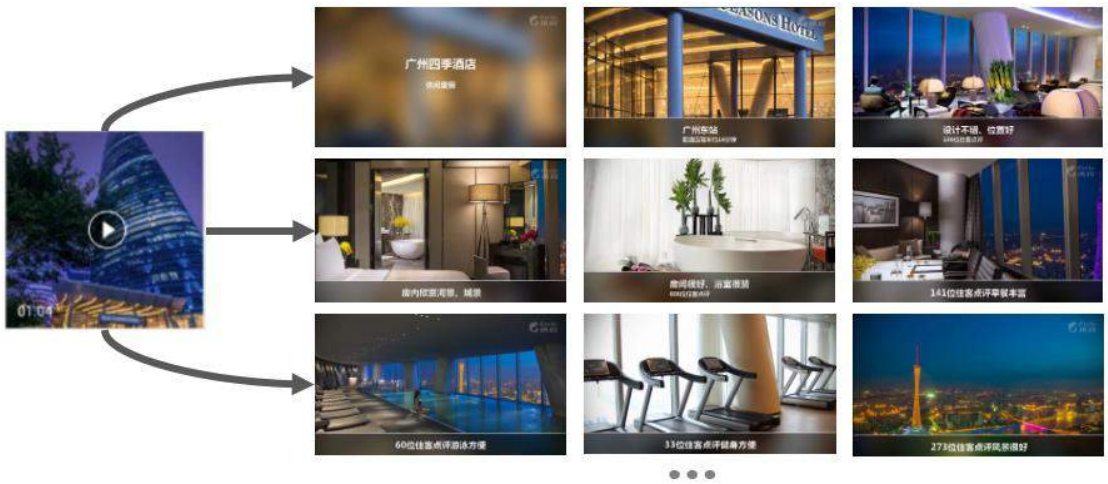
点评数据，利用真实用户的评论内容来描述酒店图像。

点评不仅流畅而且自带情感，用户在浏览图像的同时可以看到其他用户对该图像内容的真实评价，方便快捷地提升了用户获取酒店信息的丰富度和可靠性，用户浏览费力度也得到显著下降。下图展示了图像浏览中结合点评描述的应用场景。

2.3 从图像到视频

随着移动通信的发展和 WIFI 的普及，用户已经不再满足于只浏览静态图像，视频的观看需求日益增长。我们在这方面也做了一些实践，通过视频播放形式对酒店信息进行连续展示，使用户观看酒店视频就可以对酒店信息进行更全面的了解，减少用户的浏览费力度，提升用户体验。

为每家酒店拍摄视频需要投入大量人力物力，为了快速获取酒店高质量精美视频，我们利用酒店图像来自动生成酒店视频。具体地，我们借助图像挖掘中获取的丰富的图像信息，设计了一个智能选图系统，实现视频图像的自动挑选。同时，我们通过酒店的文本信息自动匹配相应的字幕。下图展示了一个酒店视频包含的部分图像和字幕。



酒店视频上线后，取得了很好的反响，用户通过观看酒店视频加深了对酒店的了解。现在酒店视频日均观看数万次，用户预订转化率和间夜量显著上升，用户和酒店实现了双赢。我们还在继续对酒店视频不断优化和迭代，力求为用户和酒店创造出更大价值。

三、总结与展望

我们通过介绍携程多个真实的图像智能化案例，分享了从 0 到 1 的图像智能化建设之路，但计算机视觉和机器学习对于携程图像智能化的价值远远不限于此。接下来我们将继续在多个图像应用场景进行深入挖掘，力求为携程图像智能化贡献更多的力量。

面向前端工程师的机器学习引导课

[作者简介]古映杰，携程度假研发部高级研发经理。负责前端框架和基础设施的设计、研发与维护。开源库 react-lite 作者。

本文面向前端工程师，使用熟悉的前端技术 JavaScript，讲解机器学习里的基础概念和算法。

演示可在浏览器里运行的机器学习 DEMO，包括拟合线段中心点，拟合矩形中心点，线性回归，AI 玩 Flappy-Bird 和 2048 游戏，识别手写数字等效果。

一、什么是人工智能（AI）？

智能行为和现象，有不同的来源。有的来自生物，有的来自机器。

我们可以把那些来生物的智能，称之为自然智能。它们是通过自然选择逐步演化而来的智能。

而另外一些由人类设计的机器所表现的智能，就是人工智能，简称 AI。

二、人工智能的两大分类

按照解决问题的能力，我们可以把人工智能，分成两类：

强人工智能：拥有自我意识，具备解决通用问题的能力
弱人工智能：没有自我意识，具备解决特定问题的能力

目前，我们能看到的人工智能，几乎都是弱人工智能，在解决特定问题的能力上，超越了人类。

2.1、强智能之拉普拉斯妖

我们可以通过物理学里的一个思想实验，来侧面理解强智能，是怎样的。

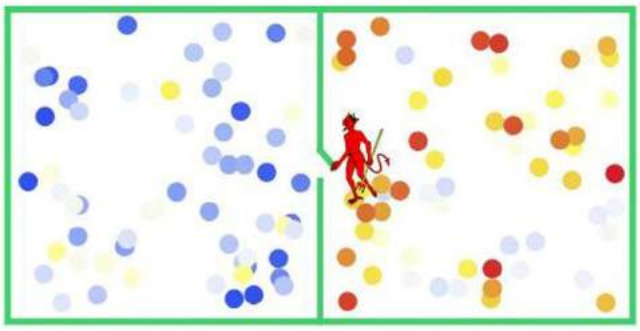
拉普拉斯是著名的数学家，他提出了一个看法：如果一个智能体，知道宇宙中每个原子确切的位置和动量，那么就可以通过牛顿定律推演出宇宙的过去以及未来；宇宙中的一切问题，都可以得到精确的解答。

然拉普拉斯妖，后来被证明是不可能的。但它确实反映了解决通用问题的一种做法。

2.2、弱智能之麦克斯韦妖

另一个物理学的思想实验，来自麦克斯韦。这个思想实验的目的，是为了挑战热力学第二定律。该定律指出，封闭系统最终会达到热平衡。

于是，麦克斯韦假设存在一种智能体：麦克斯韦妖，看守暗门，观察分子运动速度。使较快的向某侧流动，较慢向另一侧流动。经过充分长的时间，两侧温差会越来越大 。温度高低是分子运动剧烈程度的宏观表现，通过分隔不同运动速率的分子，就让系统的两个部分的分子有了不同的剧烈程度。



三、什么是机器学习?

机器学习，是英文 Machine Learning 的直译。它是实现人工智能的其中一种方式。前面说过的拉普拉斯妖就是不是机器学习的方式。

人类的学习，可以归纳为这种现象：随着经验的增加，解决问题的能力得到提升。

对机器而言，经验其实就是数据。机器学习就是：用数据训练程序以优化其表现的算法。

机器学习是一个相对宽泛的概念，只要满足它的定义，就属于机器学习。并不是一定要 GPU/TPU 训练，一定要多少行代码，一定要解决多宏大的难题，才叫机器学习。用数据训练一个模型，拟合一个点，也是机器学习的体现。

四、机器学习的分类

机器学习有两种分类：监督学习和非监督学习。

区分这两种学习方式的依据很简单，就是训练数据是否包含了答案。包含答案，就是监督学习；不包含答案，就是非监督学习。当然，这里也存在中间状态，包含一个渐进式答案，或者其它形式的间接答案等，就叫半监督学习。

五、什么是深度学习?

深度学习，是英文 Deep Learning 的直译。它是实现机器学习的其中一种方式。机器学习还包含其它实现方案。

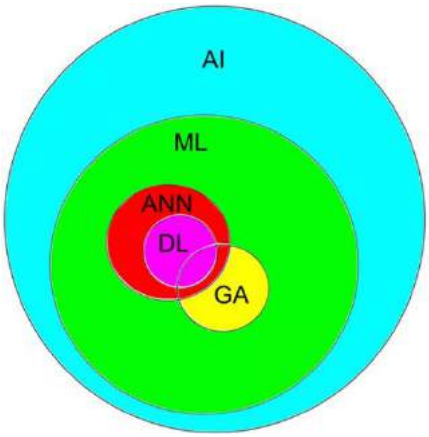
深度学习里，用到了人工神经网络，这是一个用计算机模拟大脑神经元运作模式的算法。同时，这个人工神经网络的隐藏层数量还必须足够多，才能构成深度神经网络。然后喂之以大量的训练数据，就是深度学习了。

换一个角度，如果隐藏层数量不多，而是每个隐藏层里包含的神经元数量很多，在形态上，它就是一个往宽度发展的神经网络结构。这时，可能就叫广度学习了。

目前，深度学习还是主流，它的训练效率，优于广度学习。

六、AI & ML & DL & GA 的关系

下图展示了人工智能，机器学习，人工神经网络，深度学习和遗传算法之间的关系。我们可以看到，除了遗传算法是交集关系意外，其余的是分离出一个个子集的关系。



七、前端工程师与人工智能

前端工程师跟人工智能，有什么关系呢?

这个问题应该反过来问，首先，按照目前的发展，将来人工智能会跟所有人产生紧密关联。前端工程师也是人类，作为人类，应该在某种程度上了解人工智能，而后能更好地使用人工智能的产品。

其次，前端工程师，也是程序员。我们可以在程序员的层面，比普通人更好地理解人工智能背后的机制。

再次，我们才是前端工程师。在前端工程师的角度，去审视人工智能未来是否会取代我们的工作。它取代我们之后，会不会有新的工作岗位出来，比如 AI 前端工程师。这时，那些对人工智能更加了解的前端工程师，就更容易得到相关的岗位了。

更何况，了解人工智能，说不定有机会转型成 AI 程序员。

编程语言、开源社区、IDE 等持续发展，会不断降低人工智能的开发门槛。过去的经验告诉我们，五年前高级程序员才能做到的事情，现在一个的普通程序员也有望做到。

人工智能这个概念，是在上世纪 50~60 年代提出的。当时，关心人工智能的那帮人，都是计算机里的鼻祖人物和数学家们。但是现在，作为前端工程师的我，居然也能写上一些代码，在 web 页面上跑起机器学习的 DEMO，这正是反映了门槛的降低趋势。

八、UI 开发的三种模式

手写标签和样式代码，生成页面
可视化拖拽 UI 组建，生成页面
直接输入设计稿，输出可用页面(https://github.com/tonybeltramelli/pix2code)

要想在前端自动化上做到极致，也无法跳开人工智能环节。pix2code 这个项目就是一个案例。虽然现在它的能力还很弱，使用场景很有限。但我们不要忘记，人工智能的发展速度，可能是指数级的。谷歌的阿尔法狗，一开始只是打败了人类业余围棋玩家，后面战胜了人类顶尖玩家李世石，再后面击败了当时的世界第一柯洁。然后，再无对手，又开始抛弃人类经验，通过自我对弈的方式，打败之前的版本。再到后面，竟已不局限于围棋，拓展到了国际象棋，日本将棋等其它棋类游戏里。这里的发展速度，是非常快的。

所以，不能因为 pix2code 目前的能力，而盲目乐观。

九、数学知识回顾：什么是导数?

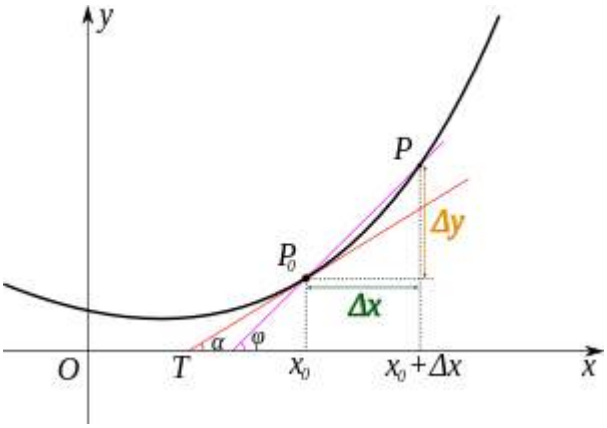
霍金在写《时间简史》时说过，书里每出现一个公式，书的销量就减半。这反映了公式在科普和推广阶段的负面作用。

但是，不用公式和代码，又要讲明白机器学习，几乎是不可能的。听着玩儿的话，我们提供的 DEMO 和前面的概念介绍，也足够满足要求了。

接下来，是直面公式和代码的时刻，不能逃避，在真正理解后，你会发觉，原来公式才是最

简单、最容易理解的那个。那些比喻、类比和段子，最后都不会在头脑里保留多久，只是在短时间内，营造了学到东西的虚假体验。

$$f'(x_0)=\lim_{\Delta x\to 0}\frac{\Delta y}{\Delta x}=\lim_{\Delta x\to 0}\frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}$$



上面是导数的代数形式，下面是导数的几何形式。我们可以看到，导数，其实是围绕一个点来谈论的。当我们说一条线的导数是多少多少时，只是一个特例，恰好那条线上的每个点的导数都是同一个值。

当我们选取了一个点 x0 之后，在 X 轴上追加一个无穷小的增量，然后用 X 加上无穷小增量得到 x1，求得 x1 对应的 Y 轴的值 y1 之后，通过 x1 - x0 和 y1 - y0，我们得到了一个小三角形。这个小三角形的 y 轴长度，除以 x 轴长度，就时这个点 x0 的导数值。

关键点有两个。一个是找到小三角形，它是直角三角形。第二个除法，是用两个直角边相除，就得到了导数。导数反映了这个点跟下一个点的变化幅度和趋势。

十、数学知识回顾：什么是复合函数?

$$y=\frac{1}{2}\cdot(1-x)^2$$
$$y=\frac{1}{2}\cdot x^2$$
$$y=1-x$$

不是所有函数都那么简单，很多函数很复杂，甚至不能写在一条公式内，甚至不能用公式表达出来。

不过，我们还是可以从函数组合的角度，对许多复杂函数进行解构。

比如上面的第一条函数，虽然它本身已经够简单了。但其实还可以拆分成两个子函数的组合。

先计算第三条公式的结果 Y，在把 Y 作为 X 值，带入第二条公式内，求得另一个 Y 值。

这个过程就相当于对第一条公式进行 Y 值计算。

十一、数学知识回顾：什么是链式法则?

简单函数，有一些导数计算公式可以套用。那复杂函数里，又应该如何求导呢?

前面我们了解了复合函数的概念，它可以拆分成简单函数之间的组合关系。通过这个组合关系，我们可以用简单函数的导数，计算出复杂函数的导数。

$$\frac{\Delta f}{\Delta x} = \frac{\Delta f}{\Delta g} \cdot \frac{\Delta g}{\Delta x}$$

$$y = \frac{1}{2} \cdot (1 - x)^2 \Rightarrow \frac{\Delta y}{\Delta x} = (1 - x) * -1$$

在介绍导数时，我们特别强调了“除法”，链式法则就是利用，除法和函数组合时产生的分子和分母的颠倒关系，不断地链式相乘，消掉中间变量，最后得到我们想要的目标导数值。

十二、数学知识回顾：什么是梯度下降?

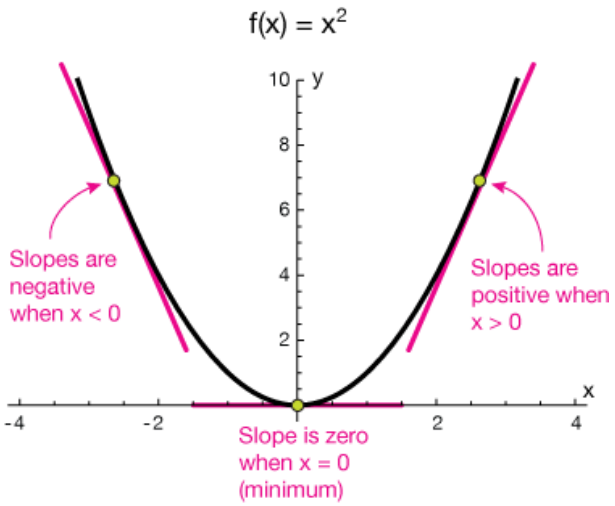
前面的数学知识部分，已经是本次分享里最难的环节了。后面是相对轻松的部分。

梯度下降，是机器学习里的重要概念。不管你去看谁的书或者视频，都绕不开这个概念。可能其中有些老师会打个比方说：假设你在一座山的山顶或者山腰上，你周围弥漫着浑厚的迷雾，无法看清下山的路，这是你要下山，要用什么方式？就是用脚探索更低点，然后逐步走下来。这就是梯度下降。

听了这个比喻，你理解了梯度下降吗?

反正我没有，下山我会，梯度下降嘛，还是不懂。

我们可以看到，比喻终究是间接的，含混的。还是要直面更直接和纯净的知识。



梯度下降，缘于这么个观测事实：

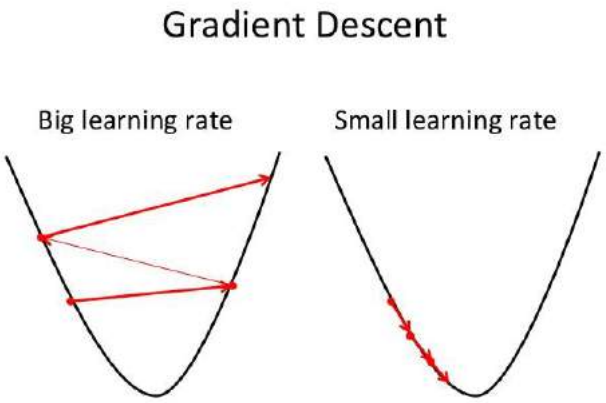
1）当我的导数是一个正数时，更低点在 -X 轴的方向； 2）当我的导数是一个负数时，更低点在 +X 轴的方向； 3）那么，不管我的导数是正数还是负数，它的反方向，就是更低点的方向。 4）沿着导数的反方向走，必将走到一个相对最低点。

这就是梯度下降。如果你理解了它，那你可以很容易地理解，什么是梯度上升。就是更高点，在沿着导数的方向嘛。

上图是一个 U 形图，只有一个最低点。如果是一个波浪图，那就有很多个局部最低点，如何找到全局最低点，目前也没有完美解决方案，还是一个难题。

十三、数学知识回顾：什么是学习率？

梯度下降，只是为我们指明了更低点的方向，但却没有告诉我们距离。所以我们必须选择一个步伐大小，这个步伐大小，被称之为学习率。



如果设置的学习率过大，步伐太大，一下子跨到对面去了，下次又走导数的反方向跨回来。就这样来来回回地振荡，没有走到一个令人满意的低点。

如果设置的学习率过小，走了很久，也仿佛在原地踏步。这也不行。

所以，选择一个合适的学习率，很重要。如何自动选择最优学习率，目前也是一个难题。

十四、前端人工智障之拟合线段中心点

在线 DEMO
https://lucifier129.github.io/simple-machine-learning-demo/01/index.html
源代码
https://github.com/Lucifier129/simple-machine-learning-demo/blob/master/01/index.js

$$x = a \rightarrow e = \frac{1}{2} \cdot (t - a)^2 \rightarrow \frac{\Delta e}{\Delta a} = (t - a) \cdot -1$$

$$a = a + \eta \cdot -\frac{\Delta e}{\Delta a}$$

要拟合一个点，先要设置一个预测模型，在这里用 x = a 就可以了。最初 a 的值是随机的，或者设置为 0，也可以，权当瞎猜。

然后我们用生成的训练数据来喂给学习算法。数据里都包含了答案，我们就可以得到一个上图误差公式。它叫均方误差。用正确答案减去瞎猜的答案，得到一个值，再对其求平方，保证是正数。

有了均方误差公式，我们就可以用之前学到链式法则求导技巧，把 Y 轴设置成误差值 e，把 X 轴设置成参数 a，然后用梯度下降寻找误差最小的点，下一个 a 参数就是： a + 学习率 * -1 * 导数值。其中，（学习率 * -1 * 导数值）得到的就是往导数反方向走的那一小步的步长。

当误差最小点为 y = 0 点时，可以说，我们找到了理想的参数 a，它完美的，零误差地拟合了目标点。

值得一提的是，你可以看到误差 e 对模型输出的 y 的导数，由链式法则计算出来，恰好包含了一个 -1，而找最低点，是用导数的反方向，也包含一个 -1，这相乘就是负负得正，可以消掉。在有些机器学习的介绍公式和代码实现里，它们就省略这两个语义不同的 -1。这对于初学者来说，是非常要不得的。它使得梯度下降的形式，看起来像梯度上升，容易混淆概念，徒增学习难度。

十五、前端人工智障之拟合矩形中心点

矩形中心点，无非就是上面的 X 轴坐标点的基础上，增加一个 Y 轴的维度嘛。

在线 DEMO
https://lucifier129.github.io/simple-machine-learning-demo/02/index.html
源代码
https://github.com/Lucifier129/simple-machine-learning-demo/blob/master/02/index.js

$$y = b \rightarrow e = \frac{1}{2} \cdot (t - b)^2 \rightarrow \frac{\Delta e}{\Delta b} = (t - b) \cdot -1$$

$$b = b + \eta \cdot -\frac{\Delta e}{\Delta b}$$

每个训练数据，都包含了 (x, y) 两个数据，我们增加一个对 y 轴的参数 y = b。然后用相同的套路，分别同步训练即可。

```
a = a + learningRate * - gradientA
b = b + learningRate * - gradientB
```

十六、前端人工智障之线性回归

在线 DEMO
https://lucifier129.github.io/simple-machine-learning-demo/03/index.html
源代码
https://github.com/Lucifier129/simple-machine-learning-demo/blob/master/03/index.js

$$y = a \cdot x + b \rightarrow e = \frac{1}{2} \cdot (t - y)^2 \rightarrow \frac{\Delta e}{\Delta y} = (t - y) \cdot -1$$
$$\frac{\Delta e}{\Delta a} = \frac{\Delta e}{\Delta y} \cdot \frac{\Delta y}{\Delta a} \rightarrow (t - y) \cdot -1 \cdot x$$
$$\frac{\Delta e}{\Delta b} = \frac{\Delta e}{\Delta y} \cdot \frac{\Delta y}{\Delta b} \rightarrow (t - y) \cdot -1 \cdot 1$$

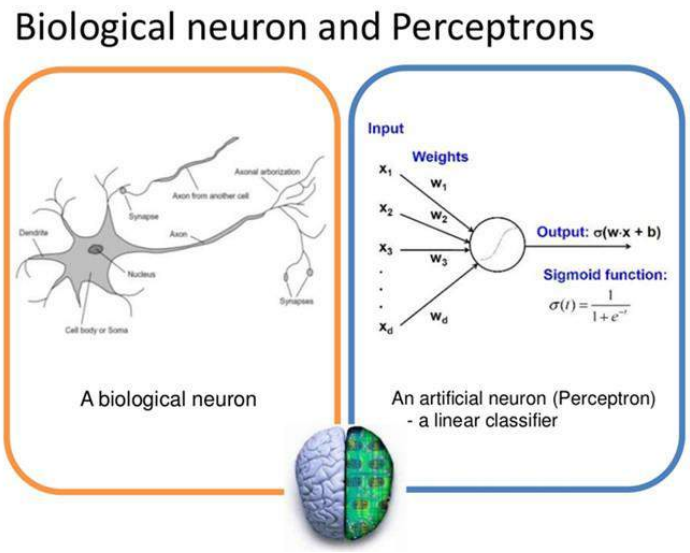
线性回归也是，除了预测模型的公式变成了 y = a * x + b，其它套路都一样。

在预测时，a 是常量系数，x 是自变量，y 是因变量。当调整参数 a 时，预测时输入的 x 成了 a 的常量系数了。所以对 y 求导 a，得到的值是 x 。

这里存在一个正向计算预测结果，反向修复参数误差的过程。

十七、什么是感知机(Perceptrons)?

在线 DEMO
https://lucifier129.github.io/simple-machine-learning-demo/04/index.html
源代码
https://github.com/Lucifier129/simple-machine-learning-demo/blob/master/04/index.js

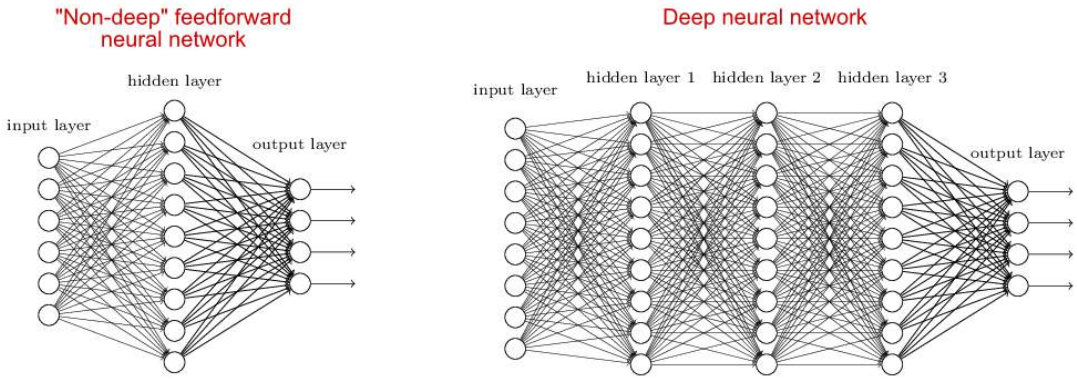


线性回归的公式是 y = a * x + b，从数学角度，很明显可以推广一下，成为 y = w1 * x1 + w2 * x2 + w3 * x3 + ... + wn * xn + b 的多因素形式。

在上面的基础上，再增加一个激活函数，就构造了一个被称之为感知机的模式。它最初是模拟大脑神经元在接受到刺激之后，根据是否达到阈值，来决定是否放电，刺激其它神经元。

感知机可以对多因素的事物，进行线性分类。比如对肿瘤进行良性肿瘤和恶性肿瘤的分类，对邮件进行垃圾邮件和非垃圾邮件的分类。不过，感知机的分类实现线性的，如果一个分类无法用线性分割出来，感知机就无法解决它。著名的"与或问题"，就曾经打击到了人工神经网络的研究热度。

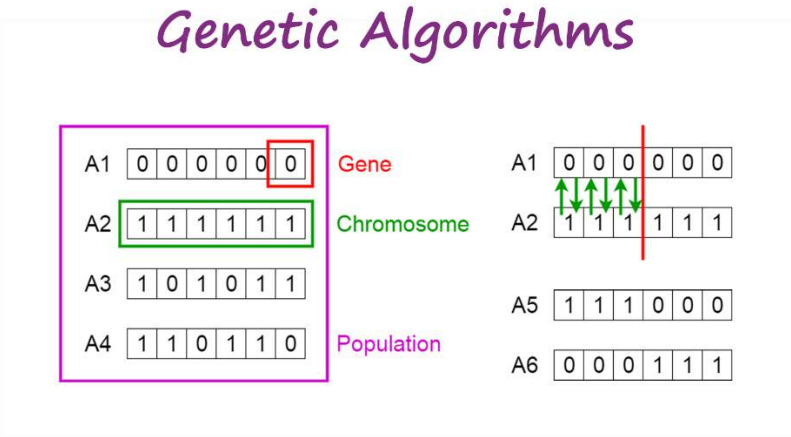
十八、什么是人工神经网络(ANN)?



感知机是线性回归里的公式推广后的形态，而人工神经网络，则可以看成是感知机的推广形态。不只是一个人工神经元，而是多个神经元以某种网络结构链接在一起，配合非线性的激活函数，它可以实现更强大的分类和拟合能力。

反向传播算法，就是一种对多层人工神经网络进行调参的算法。它的原理，正如我们前面介绍的求导、链式法则和梯度下降。感兴趣的同学，可以尝试自行推导和实现一下反向传播算法。

十九、什么是遗传算法(GA)?



遗传算法是一个伟大的算法。对人类而言，可能是最伟大的算法。

因为，广义上的算法，不局限于编程领域。任何确定性的、在有限步骤下完成的特定步骤序列，都可以称之为算法。它可以是代码形式的，也可以是物理形式的。

人类本身，就可以看成遗传算法跑在大自然这个生态系统平台上的产物。

现在我们人类，又把遗传算法应用到了计算机平台上。所有能编码到一个序列的问题，都可以用遗传算法解决。区别在于，遗传算法未必是最好最经济的解决方案罢了。

像太阳系，或者宇宙，拥有庞大的资源，可以花费数十亿年的时间，跑遗传算法（自然选择），最后演化出人类智能。我们人类的算力，却着实有限，所以我们通常需要更高性能的算法。

遗传算法模拟了自然选择里的几个概念，基因、染色体以及种群，通过让两个染色体按照某个突变率交换基因位的编码，来不断得到新的，可能更好的解决方案。

遗传算法和人工神经网络结合起来，产生了一个被成为神经进化的算法。后面我们可以看到这个算法应用在玩 Flappy-bird 游戏的效果。

二十、前端人工智障 DEMO

神经网络+遗传算法，玩 Flappy-Bird
https://lucifier129.github.io/factor-network/examples/build/#Flappy-Bird-Of-Neuroevolution-Without-Labeled-Data

神经网络+反向传播，玩 Flappy-Bird
https://lucifier129.github.io/factor-network/examples/build/#Flappy-Bird-Of-Back-Propagation
神经网络+遗传算法+变种，玩 Flappy-Bird
https://lucifier129.github.io/factor-network/examples/build/#Flappy-Bird-Of-Back-Propagation
十大高手，玩 Flappy-Bird
https://lucifier129.github.io/factor-network/examples/build/#Flappy-Bird-Of-Ten-Masters
反向传播，识别手写数字
https://lucifier129.github.io/factor-network/examples/build/#MNIST-Handwritten-Digit-Of-Back-Propagation

Flappy-bird 等游戏，在每一帧里，都没有包含明显的正确决策，所以监督学习在这里比较难以派上用场。我们要用非监督的方式来学习。

第一个 DEMO，遗传算法+神经网络，生成一批神经网络模型，指导每只小鸟是否 flap 飞起来的决策。当鸟儿碰壁死绝，用每只鸟儿得到的 score 分数进行排名，然后让排名靠前的神经网络，繁殖更多，而排名靠后的繁殖更少，或者淘汰。经过 N 代的演化，最后得到优秀的神经网络参数模型。

第二个 DEMO，用第一个 DEMO 训练出来的优秀模型，作为正确答案的生成器，我们就可以得到一个监督学习的用场了。把优秀 Bird 的决策当作正确答案，喂数据给反向传播算法，当两者的误差趋于一个很小的值时，另一个优秀的模型就训练出来了。

第三个 DEMO，还是遗传算法+神经网络，只是这次，不是用分数高低来进行排名。分数不是答案，所以是非监督学习。这次，我们用第一个 DEMO 训练出来的优秀模型，作为正确答案的生成器。然后计算每只演化中的 Bird 模型的误差，误差大，排名靠后，误差小，排名靠前。由于训练包含了正确答案，所以是监督学习。最后，我们演化出了一个跟第一个 DEMO 训练出的优秀模型误差很小的新模型。

如此可见，遗传算法+神经网络拥有更强的解决问题的能力，既可以监督学习，也可以非监督学习。但反向传播算法，只需要调整一个神经网络参数模型，而遗传算法的版本，却动辄成百上千个模型，性能自然不那么好。

第四个 DEMO，是用训练的 10 只优秀 Bird 模型角逐。目前为止，跑到 1 亿分，我也没见它们任何一个倒下。

第五个 DEMO，是机器学习里经典的 MNIST 手写数字训练集，采用的是反向传播算法的监督学习。可以识别你写在画板上的数字。

二十一、结语

尽管前端并非机器学习的主场，但作为学习，它可能挺适用的。起码我们可以更容易地在网页上看到效果。

当然，如果想更深入理解机器学习，需要去看更专业的书籍或教程。本次分享，主要是作为一个引导，激发大家对机器学习的兴趣。

证件全文本 OCR 技术，了解一下

[作者简介]周源，携程技术平台研发中心高级研发经理，从事软件开发 10 余年。2012 年加入携程，先后参与支付、营销、客服、用户中心的设计和研发。

本文从计算机视觉的前世今生，到证件全文本 OCR 的实践，带你了解人工智能、计算机视觉、深度学习、卷积神经网络等技术。无论是计算机视觉的入门者还是从业者，希望都可以有所收获。

一、什么是 OCR

光学字符识别（英语：Optical Character Recognition, OCR），是指对文本资料的图像文件进行分析识别处理，获取文字及版面信息的过程。

一般的识别过程包括：

图像输入：对于不同的图像格式，有着不同的存储格式，不同的压缩方式，目前有 OpenCV、CxImage 等开源项目。

预处理：主要包括二值化，噪声去除，倾斜校正等。

二值化：摄像头拍摄的图片，大多数是彩色图像，彩色图像所含信息量巨大，对于图片的内容，我们可以简单的分为前景与背景，为了让计算机更快的、更好地识别文字，我们需要先对彩色图进行处理，使图片只剩下前景信息与背景信息，可以简单的定义前景信息为黑色，背景信息为白色，这就是二值化图。

噪声去除：对于不同的文档，我们对噪声的定义可以不同，根据噪声的特征进行去噪，就叫做噪声去除。

倾斜校正：由于一般用户，在拍照文档时，都比较随意，因此拍照出来的图片不可避免的产生倾斜，这就需要文字识别软件进行校正。

版面分析：将文档图片分段落，分行的过程就叫做版面分析。由于实际文档的多样性和复杂性，目前还没有一个固定的，最优的切割模型。

字符切割：由于拍照条件的限制，经常造成字符粘连，断笔，因此极大限制了识别系统的性能。

字符识别：这一研究已经是很早的事情了，比较早有模板匹配，后来以特征提取为主，由于文字的位移，笔画的粗细，断笔，粘连，旋转等因素的影响，极大影响特征的提取的难度。

版面还原：人们希望识别后的文字，仍然像原文档图片那样排列着，段落不变，位置不变，

顺序不变地输出到 Word 文档、PDF 文档等，这一过程就叫做版面还原。

后处理、校对：根据特定的语言上下文的关系，对识别结果进行校正，就是后处理。

目前的主流实现是 CNN+RNN+CTC 和 CNN+RNN 基于 Attention 的方法。

二、携程证件 OCR 项目

2.1 项目目标

根据携程的实际使用场景，使用 OCR 技术识别身份证、护照、火车票、签证等证件的中文英文及数字文本信息。

2.2 主要进展及展望

2016 年，实现客户端身份证、护照英文数字的识别。
2017 年，实现 Offline 场景身份证、护照、火车票等中文识别。
2018 年，实现 APP 实时识别身份证、护照、火车票、驾驶证、行驶证、签证等中英文文本。

2.3 大概的精度情况

2.3.1.数字英文

误识率<0.5%【线上数据统计】
拒识率~5%
干扰因素包括：曝光、倾斜、远照

2.3.2.中文

1:N 有引导（指引导用户将证件放于相机框中）
FAR= 1%【线上数据统计】
拒识率 ~ 20%

1:N+1　无引导
FAR= 3%【线上数据统计】
拒识率 ~ 30%

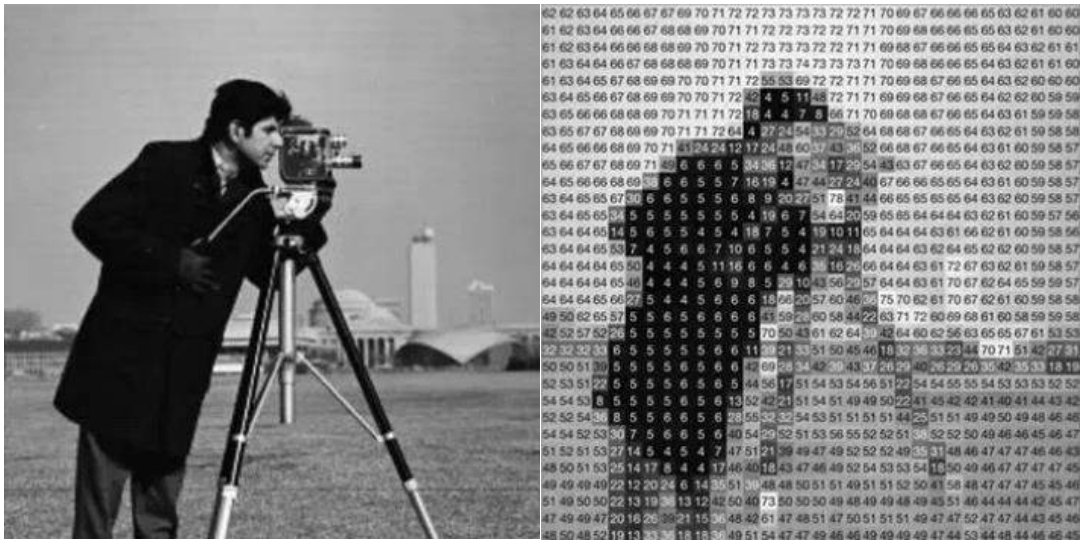
-曝光、图像质量低计入 FAR
-遮挡计入拒识
-考虑外籍证件
-考虑少数民族

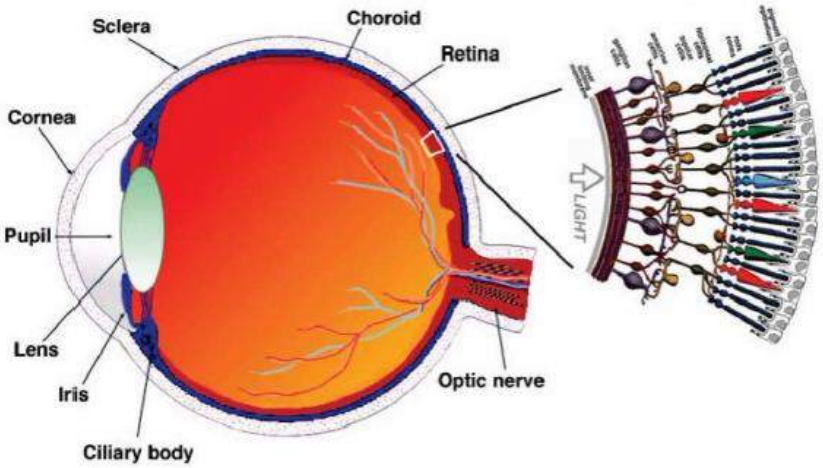
三、关键知识

3.1 计算机视觉

3.1.1 什么是计算机视觉

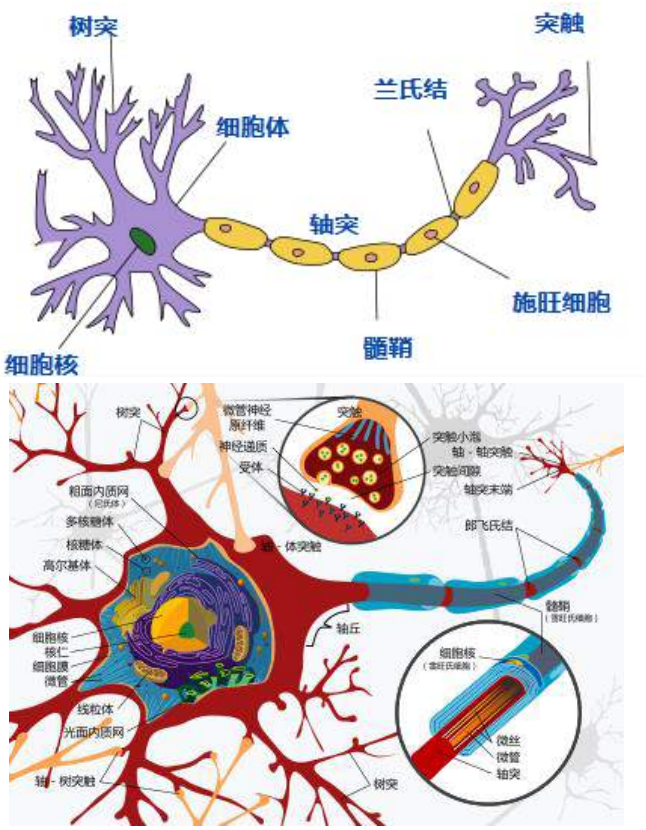
解读 w*h*3 个 0 ~ 255 之间的数字中蕴含的、人类可理解的内容。



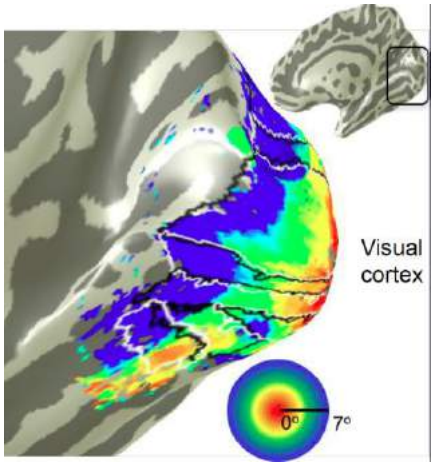


3.1.2 人类视觉的启示

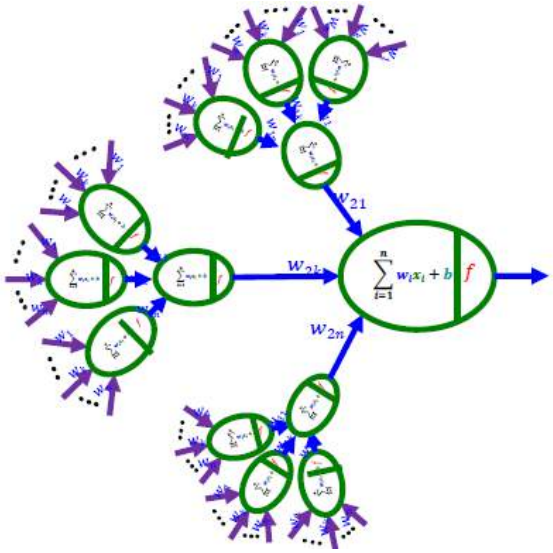
感受野：这个位置里适当的刺激能够引起该神经元反应的区域。



层级感受野：一个神经细胞看的更远（视野更大）、能处理更负责的任务。



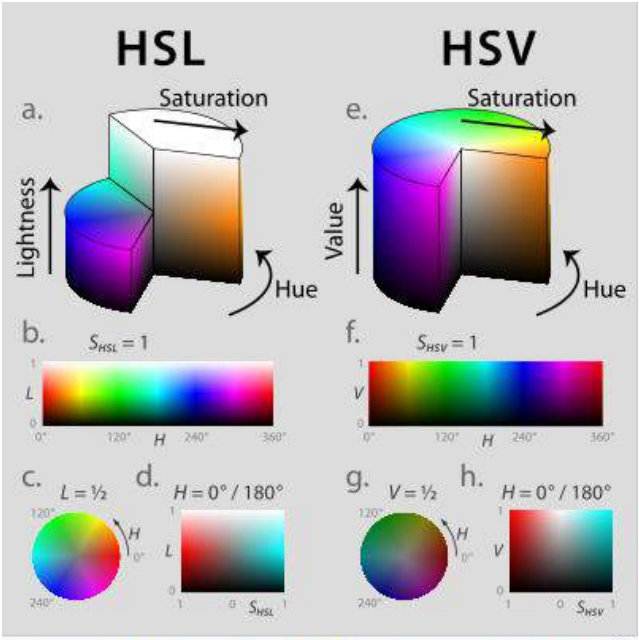
神经网络从输入到输出，中间有多个隐藏的层。



3.1.3 HSV 和灰度图

HSV是一种将RGB色彩模型中的点在圆柱坐标系中的表示法。这两种表示法试图做到比RGB基于笛卡尔坐标系的几何结构更加直观。

- 色相（H）是色彩的基本属性，就是平常所说的颜色名称，如红色、黄色等。
- 饱和度（S）是指色彩的纯度，越高色彩越纯，低则逐渐变灰，取0-100%的数值。
- 明度（V），亮度（L），取0-100%。



HSL（a~d）和HSV（e~h）。上半部分（a、e）：两者的3D模型截面。下半部分：将模型中三个参数的其中之一固定为常量，其它两个参数的图像。

HSV 模型通常用于计算机图形应用中。在用户必须选择一个颜色应用于特定图形元素各种

应用环境中，经常使用 HSV 色轮。在其中，色相表示为圆环；可以使用一个独立的三角形来表示饱和度和明度。典型的，这个三角形的垂直轴指示饱和度，而水平轴表示明度。在这种方式下，选择颜色可以首先在圆环中选择色相，在从三角形中选择想要的饱和度和明度。

$$h = \begin{cases} 0^\circ & \text{if } max = min \\ 60^\circ \times \frac{g-b}{max-min} + 0^\circ, & \text{if } max = r \text{ and } g \geq b \\ 60^\circ \times \frac{g-b}{max-min} + 360^\circ, & \text{if } max = r \text{ and } g < b \\ 60^\circ \times \frac{b-r}{max-min} + 120^\circ, & \text{if } max = g \\ 60^\circ \times \frac{r-g}{max-min} + 240^\circ, & \text{if } max = b \end{cases}$$

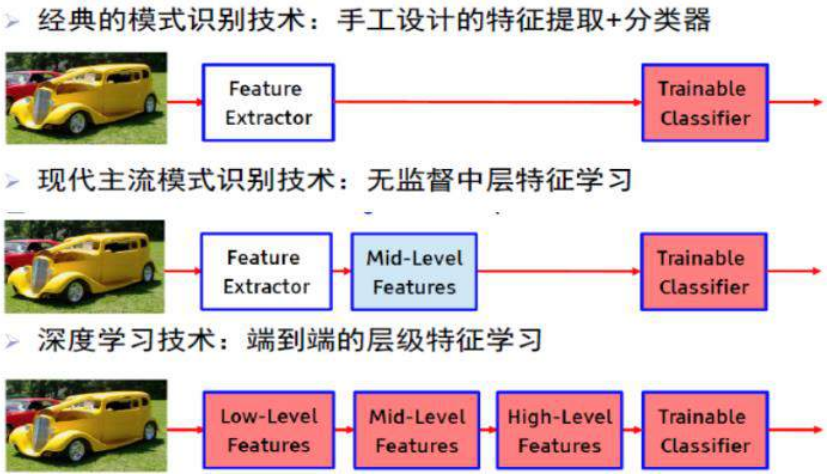
$$s = \begin{cases} 0 & \text{if } l = 0 \text{ or } max = min \\ \frac{max-min}{max+min} = \frac{max-min}{2l}, & \text{if } 0 < l \leq \frac{1}{2} \\ \frac{max-min}{2-(max+min)} = \frac{max-min}{2-2l}, & \text{if } l > \frac{1}{2} \end{cases}$$

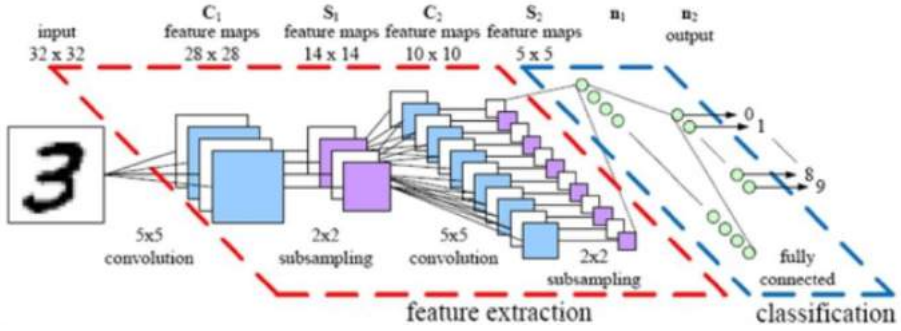
$$l = \frac{1}{2}(max + min)$$

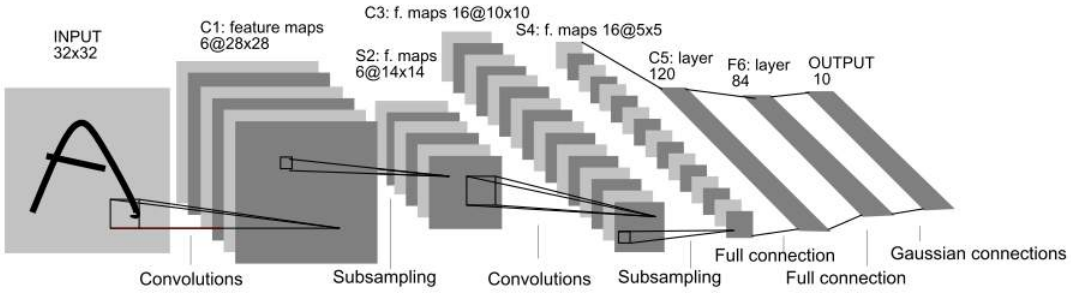
图像的灰度化：在计算机领域中，灰度（Grayscale）数字图像是每个像素只有一个采样颜色的图像。这类图像通常显示为从最暗黑色到最亮的白色的灰度，尽管理论上这个采样可以是任何颜色的不同深浅，甚至可以是不同亮度上的不同颜色。

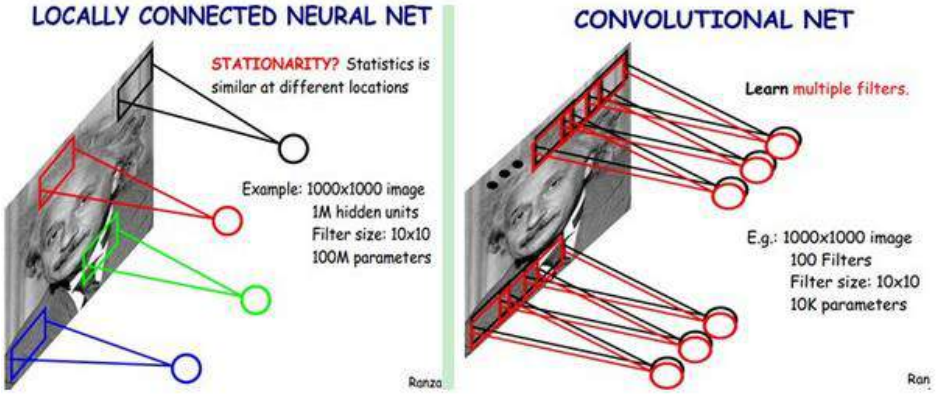
灰度图像与黑白图像不同，在计算机图像领域中黑白图像只有黑白两种颜色，灰度图像在黑色与白色之间还有许多级的颜色深度。但是，在数字图像领域之外，“黑白图像”也表示“灰度图像”，例如灰度的照片通常叫做“黑白照片”。在一些关于数字图像的文章中单色图像等同于灰度图像，在另外一些文章中又等同于黑白图像。

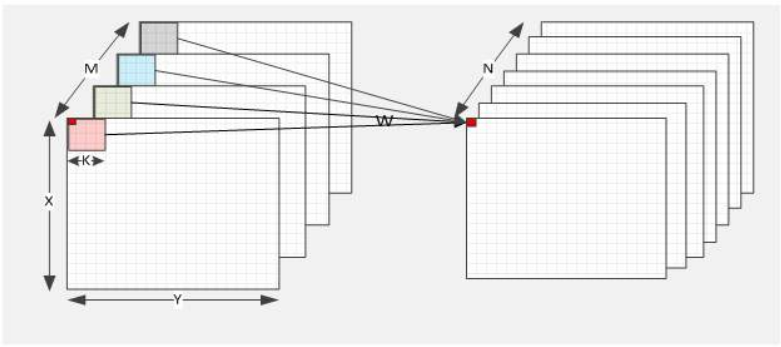
3.2 基于卷积神经网络的深度学习模型











一个$k \times k$的卷积核，一次卷积计算可以生成feature map的一个点，一次卷积计算需要：$k \times k \times$M次乘法

$$y_j^{l-1} = \sum_{i=0}^{M-1} x_i^{l-1} \otimes W_{ij}^l + b_j^l$$

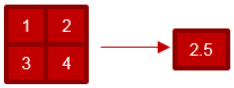
y_j^{l-1}表示第l-1层的第j通道的值，x_i^{l-1}表示第l-1层的第i通道的值.j$\in \{0, \ldots, N-1\}$

3.3 二值化和池化

二值化（英语：Thresholding）是图像分割的一种最简单的方法。二值化可以把灰度图像转换成二值图像。把大于某个临界灰度值的像素灰度设为灰度极大值，把小于这个值的像素灰度设为灰度极小值，从而实现二值化。

根据阈值选取的不同，二值化的算法分为固定阈值和自适应阈值。比较常用的二值化方法则有：双峰法、P 参数法、迭代法和 OTSU 法等。

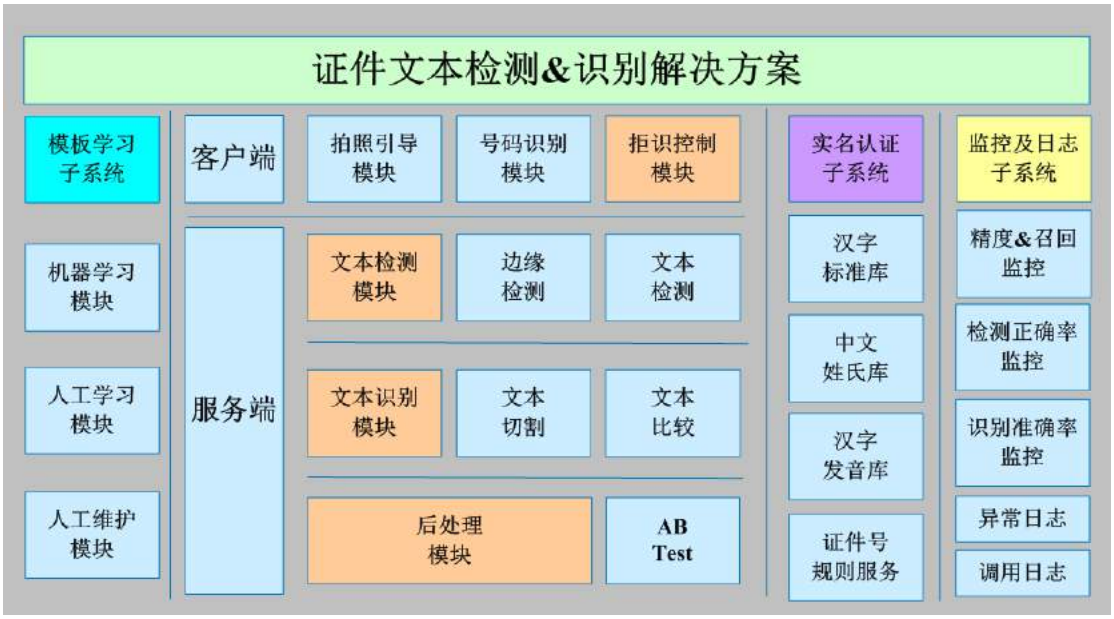
最大值池化示意图



平均池化示意图

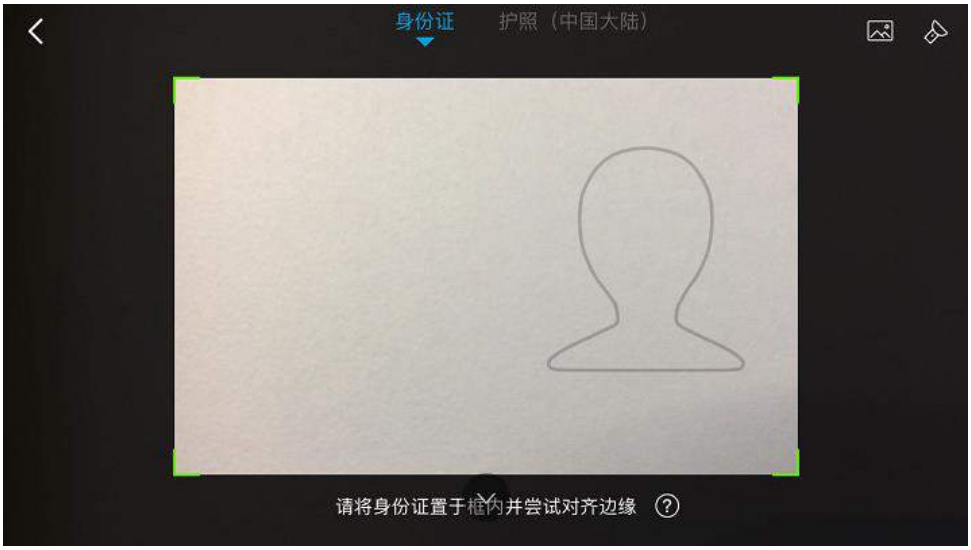
四、证件 OCR 的架构及实现

4.1 架构图



4.2 应用场景

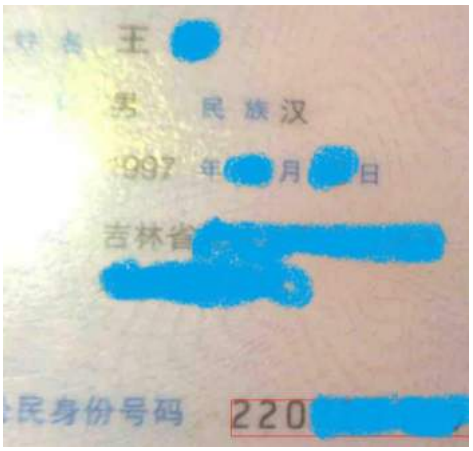
	英文数字	中文	检测	识别	集成于APP	整图
身份证 -我携接入	☑	☑	☑	☑	☑	☒
护照 -我携接入	☑	☒	☒	☑	☑	☒
护照 -邮轮接入	☑	☑	☒	☑	☒	☒
身份证 -机票接入	☑	☒	☒	☑	☑	☒
护照 -度假接入	☑	☒	☒	☑	☑	☒
火车票 -金融接入	☑	☑	☑	☑	☒	☑
身份证 -金融接入	☑	☑	☑	☑	☑	☒



4.3 拒识检测

以下图为例，当用户将我们待识别区域（即姓名位置）遮挡时，我们会执行拒识处理。

同理如下图，待识别区域发生明显曝光时，我们也会加入拒识处理。



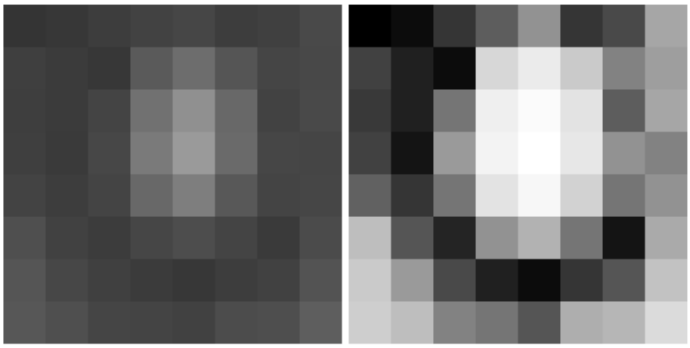
拒识处理使用直方图均衡等技术。

直方图均衡化是图像处理领域中利用图像直方图对对比度进行调整的方法。

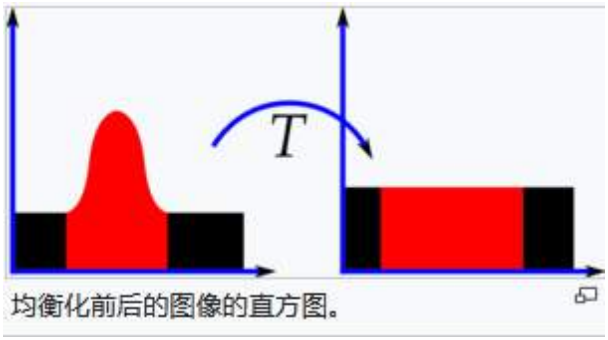
这种方法通常用来增加许多图像的全局对比度，尤其是当图像的有用数据的对比度相当接近的时候。通过这种方法，亮度可以更好地在直方图上分布。这样就可以用于增强局部的对比度而不影响整体的对比度，直方图均衡化通过有效地扩展常用的亮度来实现这种功能。

这种方法对于背景和前景都太亮或者太暗的图像非常有用，尤其是可以带来 X 光图像中更好的骨骼结构显示以及曝光过度或者曝光不足照片中更好的细节。

一个主要优势是它是个相当直观的技术并且是可逆操作，如果已知均衡化函数，那么就可以恢复原始的直方图，并且计算量也不大。一个缺点是它对处理的数据不加选择，它可能会增加背景噪声的对比度并且降低有用信号的对比度。



原始图像　　均衡化后图像



拒识处理在客户端或页面前端完成，没有前端的场景则在后端完成。拒识处理还使用二分搜索算法通过大量样本判断目标图片是否拒识。

```
defbinary_search(arr,start,end,hkey):
ifstart > end:
    return-1
mid= start + (end - start) / 2
ifarr[mid] > hkey:
    returnbinary_search(arr, start, mid - 1, hkey)
ifarr[mid] < hkey:
    returnbinary_search(arr, mid + 1, end, hkey)
returnmid
```

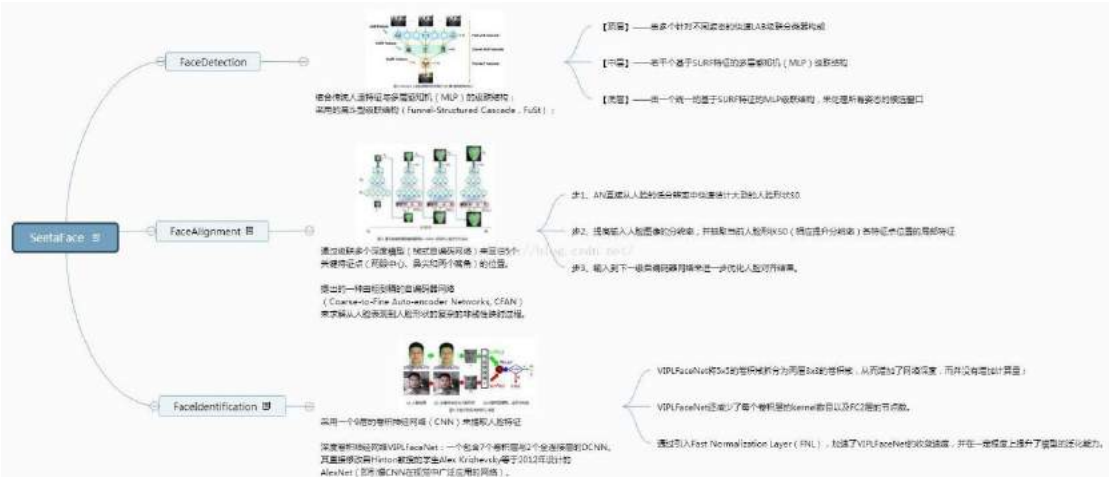
线上版本拒识的精度在 98%-99%之间，拒错率在 20%以下。

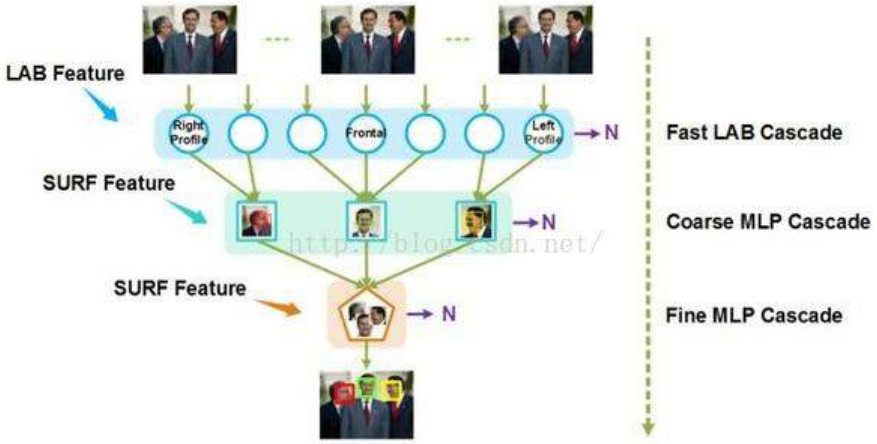
4.4 文本检测

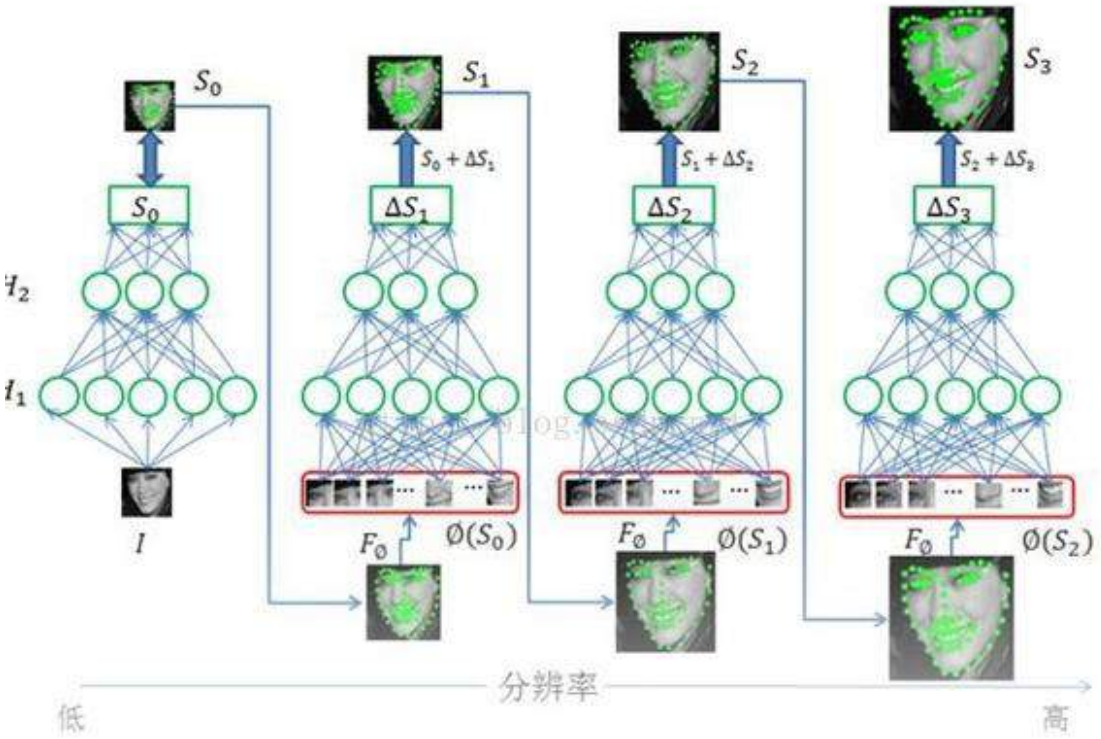
文本检测分为有引导和无引导两类，有引导的文本检测，我们使用先验知识（比如人脸，证件边缘等）和大量样本深度学习目标函数定位待识别区域。

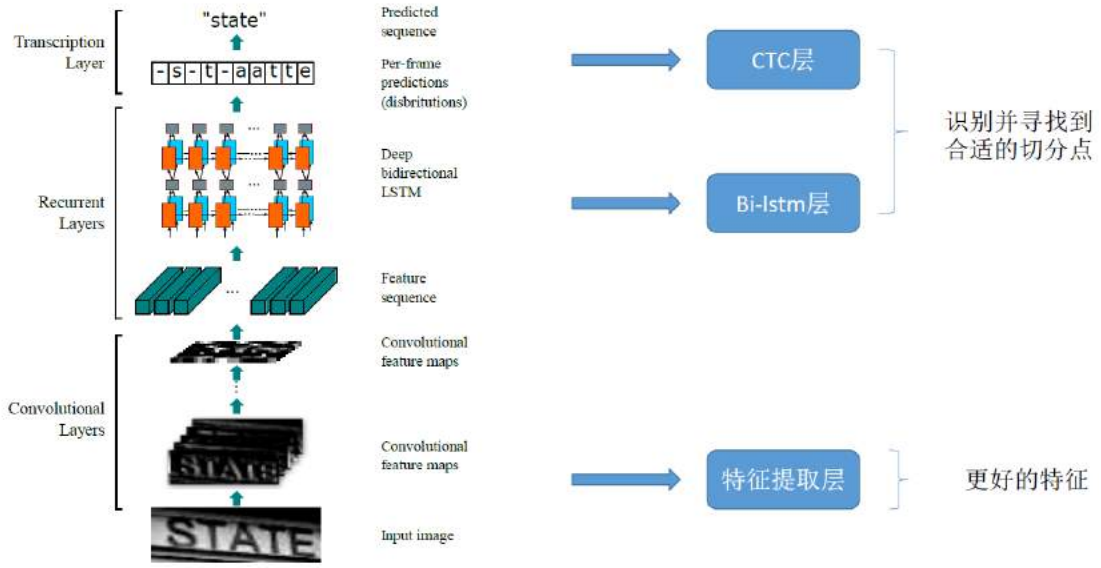
而无引导的情况，则完全使用基于 Attention 的整行识别技术做范文本处理。

人脸识别部分，借鉴了山世光老师的开源项目 seetaface/SeetaFaceEngine，并针对我们证件场景做了一些定制开发。


SeetaFace
FaceDetection
FaceAlignment
FaceIdentification


LAB Feature
Right Profile
Frontal
Left Profile
N
Fast LAB Cascade
SURF Feature
N
Coarse MLP Cascade
SURF Feature
N
Fine MLP Cascade


S_0
S_1
S_2
S_3
$S_0 + \Delta S_1$
$S_1 + \Delta S_2$
$S_2 + \Delta S_3$
ΔS_1
ΔS_2
ΔS_3
H_2
H_1
I
F_0
$\emptyset(S_0)$
F_0
$\emptyset(S_1)$
F_0
$\emptyset(S_2)$
分辨率
低
高



4.5 文本识别

文本识别部分我们使用灰度投影在切割无关信息（比如少数民族的拼音、外籍护照的发音注释等）、二值化归一化/下采样池化等技术做比较识别。并且引入了 HOG、LBP、Haar 等特征的权重机制。

```
int* v = NULL;//垂直投影
    int* h = NULL;//水平投影
    CvScalar s,t;//投影时矩阵的元素
    IplImage* pBinaryImg = NULL;//二值化后图像
    IplImage* pVerticImg = NULL;//垂直投影图像
    IplImage* pHorizImg = NULL;//水平投影图像
    int x,y;//图像像素坐标
    v=new int[pBinaryImg->width];
        h=new int[pBinaryImg->height];
        for(i=0;i<pBinaryImg->width;i++)
            v[i]=0;
        for(i=0;i<pBinaryImg->height;i++)
            h[i]=0;

        for( x=0;x<pBinaryImg->width;x++)
        {
            for(y=0;y<pBinaryImg->height;y++)
            {
               s=cvGet2D(pBinaryImg,y,x);       //t=cvGet2D(paint,y,x);
                if(s.val[0]==0)
                    v[x]++;       //cvSet2D(paint,y,x,t);
            }
```

```
}

for( y=0;y<pBinaryImg->height;y++)
{
    for( x=0;x<pBinaryImg->width;x++)
    {
       s=cvGet2D(pBinaryImg,y,x);       //t=cvGet2D(paint,y,x);
        if(s.val[0]==0)
            h[y]++;
    }
}
pVerticImg = cvCreateImage( cvGetSize(pBinaryImg),8,1 );
pHorizImg = cvCreateImage( cvGetSize(pBinaryImg),8, 1);
cvZero(pVerticImg);
cvZero(pHorizImg);
for(x=0;x<pBinaryImg->width;x++)
{
    for(y=0;y<v[x];y++)
    {
       t=cvGet2D(pVerticImg,y,x);     //s=cvGet2D(paint,y,x);     //t=cvGet2D(paint,y,x);
        t.val[0]=255;
       cvSet2D(pVerticImg,y,x,t);
    }
}

for(y=0;y<pBinaryImg->height;y++)
{
    for(x=0;x<h[y];x++)
    {
       t=cvGet2D(pHorizImg,y,x);     //s=cvGet2D(paint,y,x);     //t=cvGet2D(paint,y,x);
        t.val[0]=255;
       cvSet2D(pHorizImg,y,x,t);
    }

}
```

4.5.1 HOG 特征

方向梯度直方图（Histogram of Oriented Gradient, HOG）特征是一种在计算机视觉和图像处理中用来进行物体检测的特征描述子。它通过计算和统计图像局部区域的梯度方向直方图来构成特征。Hog 特征结合 SVM 分类器已经被广泛应用于图像识别中，尤其在行人检测中获得了极大的成功。需要提醒的是，HOG+SVM 进行行人检测的方法是法国研究人员 Dalal 在 2005 的 CVPR 上提出的，而如今虽然有很多行人检测算法不断提出，但基本都是以 HOG+SVM 的思路为主。

4.5.2 LBP 特征

LBP（Local Binary Pattern，局部二值模式）是一种用来描述图像局部纹理特征的算子；具有旋转不变性和灰度不变性等显著的优点。它是首先由 T. Ojala, M.Pietikäinen，和 D. Harwood 在 1994 年提出，用于纹理特征提取。而且，提取的特征是图像的局部的纹理特征。

4.5.3 Haar 特征

Haar-like 特征最早是由 Papageorgiou 等应用于人脸表示，Viola 和 Jones 在此基础上，使用 3 种类型 4 种形式的特征。

Haar 特征分为三类：边缘特征、线性特征、中心特征和对角线特征，组合成特征模板。特征模板内有白色和黑色两种矩形，并定义该模板的特征值为白色矩形像素和减去黑色矩形像素和。Haar 特征值反映了图像的灰度变化情况。例如：脸部的一些特征能由矩形特征简单的描述，如：眼睛要比脸颊颜色要深，鼻梁两侧比鼻梁颜色要深，嘴巴比周围颜色要深等。但矩形特征只对一些简单的图形结构，如边缘、线段较敏感，所以只能描述特定走向（水平、垂直、对角）的结构。

4.6 一些后处理的思路

在识别主体逻辑外，我们根据实际场景加入了一些后处理逻辑。

比如身份证号验证、护照号验证、汉字权重验证、中文姓氏库验证、中文发音验证等。

五、参考资料

1）https://zh.wikipedia.org/wiki/(2018)
2）Learning2See_VAI_Progrss&Trends - 山世光
3）Gradient-Based Learning Applied toDocument Recognition in Proceedings of the IEEE, 1998
4）ImageNetClassification with Deep Convolutional Neural Networks
5）Noteson Convolutional Neural Networks
6）Practical Recommendations for Gradient-BasedTraining of Deep Architectures
7）Stochastic Pooling for Regularization of DeepConvolutional Neural Networks
8）Maxout Networks

【注】文中所列代码并非项目实际代码，都为说明算法需要的开源代码。

携程 AI 模型引擎设计与实践

[作者简介]李媚，酒店数据智能组应用开发工程师。2016 年加入携程，先后负责了酒店交叉推荐，优选频道个性化酒店排序等服务开发工作。

前言

近年来人工智能的发展成果在互联网行业得到了广泛的推崇和应用，各大巨头纷纷借助 AI 打造个性化、精细化服务，加速扩张生态领域。从推荐系统到实时风控，从广告系统到图像处理，模型服务在携程各个业务领域发挥着日益重要的作用。然而回顾现有的模型上线模式，不难发现仍存在一定的缺陷：

1、训练数据准备工作需要手工完成。数据清洗和特征挖掘是模型训练的前期工作，既包括从原始数据清洗出特征数据，也包括对清洗出的特征进行处理。由于缺乏统一的特征管理平台，目前训练需要的原始数据仍需算法工程师自行收集、整理、清洗。

2、不少模型处于离线预测阶段。相对于离线预测，实时预测能够依据用户的实时行为和最新的数据信息作出下一步预测，有效提高预测的准确性。但实时数据存在复杂、多变等特性，以及实时预测对性能上的要求更加严苛，工程技术门槛高，不少团队选择了相对容易实现的离线预测方式。

3、实时模型服务的开发周期长。实时模型服务离不开实时特征准备、业务逻辑开发、模型调用开发等步骤。实时特征一般由各项目的开发工程师自行维护，不可避免地存在特征重复开发的现象，带来开发资源和存储资源的浪费。此外，一个预测场景一般由一个模型服务提供支持，新的模型服务需求需要完全从头开始开发，开发周期较长。

结合上述现状以及在酒店个性化推荐、信息与图片等领域积累的丰富的模型上线经验，携程数据服务组推出了模型引擎平台——旨在通过搭建一个综合性的模型服务平台减少模型上线过程中的重复工作，实现模型的快速上线迭代，健全线上模型的监控评估机制。

一、平台构建

1.1 设计目标

作为一个综合型的平台，模型引擎致力于提供从数据处理、模型训练到模型上线的全闭环服务，为此我们制定了如下目标：

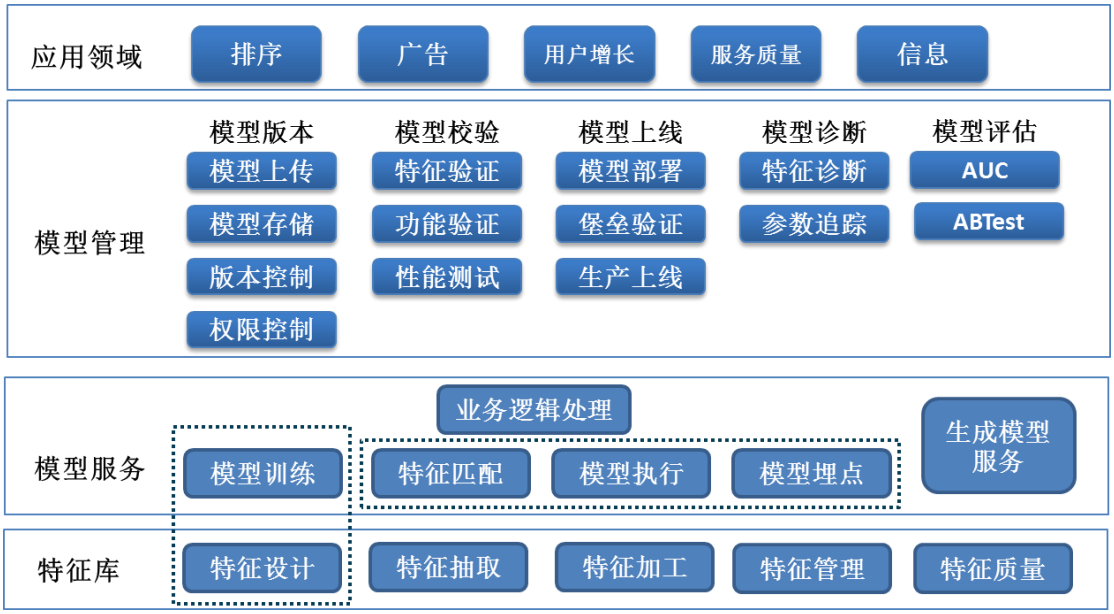
1）服务于产品经理、数据科学家、开发工程师、测试工程师团队，通过服务全景图的形式串联各环节；

2）作为一个实时预测平台，模型引擎服务接受秒级延迟的实时数据，毫秒级地返回预测结果；

3）广泛地适用于携程各类业务场景，支持包括 ABTesting、模型灰度上线、多模型融合等功能。

1.2 总体架构

针对上述设计目标，模型引擎设置了特征管理、工程管理、模型管理、产品管理四大模块，总体逻辑架构图如下：



1.2.1 特征管理

特征管理模块主要解决特征准备问题，对应了逻辑架构图中的特征库一层。

根据使用场景，特征分为离线特征和在线特征。离线特征主要在模型训练阶段使用，数据科学家对存储在 Hdfs 中的静态数据、落地的消息数据等尝试不同的样本选择、变换处理和组合等，再配合算法及参数进行模型训练。训练完成后，数据科学家将最终的特征方案交给开发工程师，再由开发工程师将离线特征搬移到在线环境，如 Redis，ES，Aerospike 等。

在线特征开发工作既包括将离线存储中的特征导入到实时存储，也包括对实时消息（Kafka、Qmq），例如用户点击、下单、酒店起价变化等进行处理，不断更新在线数据。

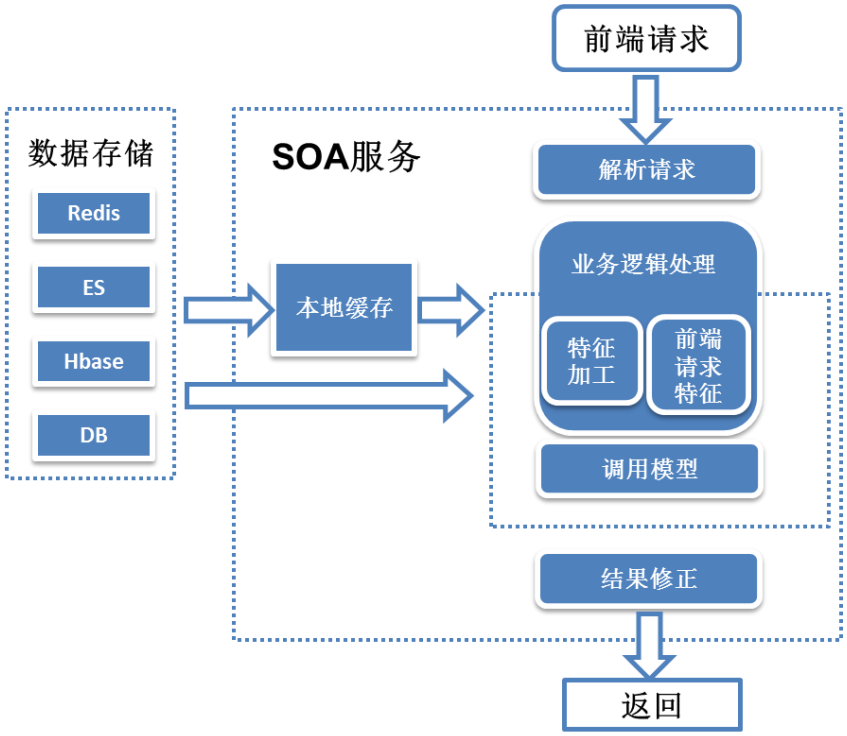
模型引擎维护了一个公共的特征库，支持算法工程师通过界面操作的方式添加离线特征，复用已有特征；同时也实现了通过配置自助导出离线特征到实时环境的功能和对流信息的实时处理功能，接入方借助提供的 API 就可以进行数据访问。

同时，对重要特征进行监控与有效性分析是保障模型服务质量的首要环节。相对于离线特征，实时特征有时效性和一致性等要求；而且实时特征一般以 KV 形式存储，不能简单地通过 SQL 语句进行统计。如果每个工程师单独为自己维护的特征开发监控 Job，那么开发工作量就会

大大加重。模型引擎由于维护了公共的特征库，可以对特征池中的特征建立了统一的长效监控机制，常见的监控指标，如缺失率、新鲜度、活跃度等，都可以通过模型引擎平台查看和增添。

1.2.2 工程管理

工程管理模块主要解决模型的工程实现问题，对应了逻辑架构图中的模型服务一层。在实时场景中，模型服务一般以 SOA 服务形式供使用方调用，包含了建立本地缓存、实现业务逻辑、调用模型、返回预测结果等步骤，如下图所示。



针对这些典型的开发步骤，模型引擎提供了通用的组件辅助开发工程师，降低工程实现难度。

在实时批量预测场景中一次调用需要用到大量的原始数据，从外部存储获取数据难以保证服务性能，因此工程师会选择建立本地缓存减少数据拉取时间。然而维护本地缓存会带来了不少额外负担，例如不同数据源对新鲜度的要求不同，本地缓存的更新机制要区分设计；本地缓存数据量大时，需要考虑 JVM 参数的配置，以避免频繁的 Full GC 和长时间的 GC。为此，模型引擎开发了 Feature Bus 组件，通过统一的 XML 配置，支持自动构建本地缓存，同时也屏蔽了底层存储的读取细节，免去繁琐的本地缓存开发工作。

在调用模型方面，模型引擎提供了 Code Gen 服务，支持模型服务的自助生成。各模型采用的算法虽然林林总总，但在工程上调用的方式却不外乎几种，例如 PMML、Python 包、Xgboost 等等。模型引擎对常用的调用方式进行了封装，供开发工程师在项目中直接使用。一方面开发工程师不再需要对没有接触过的模型调用方式进行摸索，另一方面对业务逻辑和模型调用进行了解耦，后续算法升级只需要简单地升级模型服务的版本。而且在生成的模型服务中嵌入了统一的监控，对线上模型服务的质量可以进行集中的管理。

此外，最终进入模型的待预测数据也是算法工程师所关心的。不少原始数据经过了开发工程师手动编码加工，在遇到模型预测效果与线下不一致时首先需要确定这部分数据的正确性。在模型服务中会收集这部分数据并上传，配合平台提供的可视化界面供算法工程师和测试工程师判断数据的健康情况。进一步地，也可以利用这些数据实时更新模型，实现自动化模型训练。

最后，模型拆分、多模型融合功能也在这一层支持。总之，开发工程师只需要将主要精力放在业务逻辑的实现上，以及借助组件串通完整流程。

1.2.3 模型管理

模型管理模块主要实现了模型文件的上传、关联特征、模型发布、迭代回退等功能，对应了逻辑架构图中的模型管理一层。

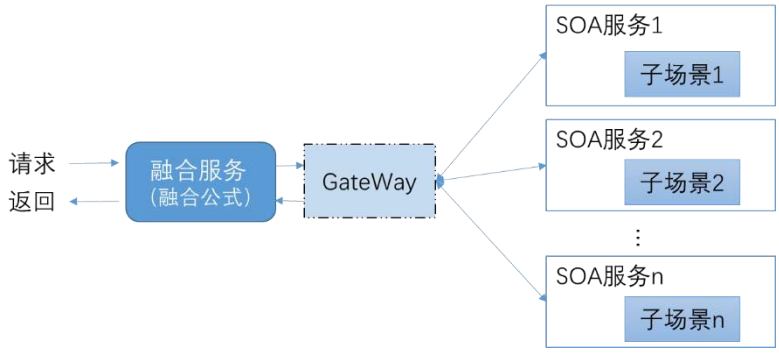
结合携程的文件服务系统，算法工程师可以在模型管理模块便捷地上传模型。在迭代模型时，为了减少再次测试、发布的环节，模型引擎平台支持算法工程师更替、回退模型文件，实现模型文件的实时生效。在操作的安全性方面，在上传新模型之后，平台会根据样本输入输出数据进行自动校验，确保模型文件的正确性；也会对模型的调用进行压测，符合预设结果的模型才可以进入部署阶段。此外，因为模型文件一旦发布就会即刻生效，为了避免误操作，由测试工程师进行最后的审核上线操作。

现有的机器学习中很多模型的使用都类似黑盒，尽管经过了离线数据的训练以及测试环境的验证，对于模型在实时环境中的预测效果依然难以预料。为此，模型管理模块提供了模型灰度上线功能，支持在正式对外发布前对模型进行内部验证。算法工程师只需要简单配置白名单和上传堡垒模型文件就可以在生产环境直观体验新模型的效果，既不需要开发工程师的介入，也不会影响外网用户。

1.2.4 产品管理

产品管理模块主要实现了场景的创建和管理。在模型引擎中一个具体的模型预测业务称为一个场景。项目一般由产品经理发起，所以由产品经理负责创建场景，填写项目信息，并关联相应的开发人员。创建完毕后，算法工程师和开发工程师登录平台就可以在模型管理和工程管理模块下进行操作，为场景实现添砖加瓦。

在模型引擎最初设计中，一个场景仅支持一个模型，根据实验版本不同上传不同的模型文件。然而随着预测业务精细化，在一些场景中，为了达到最优效果，会对模型进行进一步的细分，拆分出多个模型，这些模型的大部分业务逻辑和模型特征都相同，或者模型间存在着依赖关系，因此不适合将这些模型独立为各个场景。因为模型引擎中采用了子场景的概念来支持一个场景下多个模型的情况。其中子场景间的关联关系又分为并行、串行、融合、switch 等多种类型，由此可以支持串行模型，多模型融合等。其中多模型融合的调用方式如下：

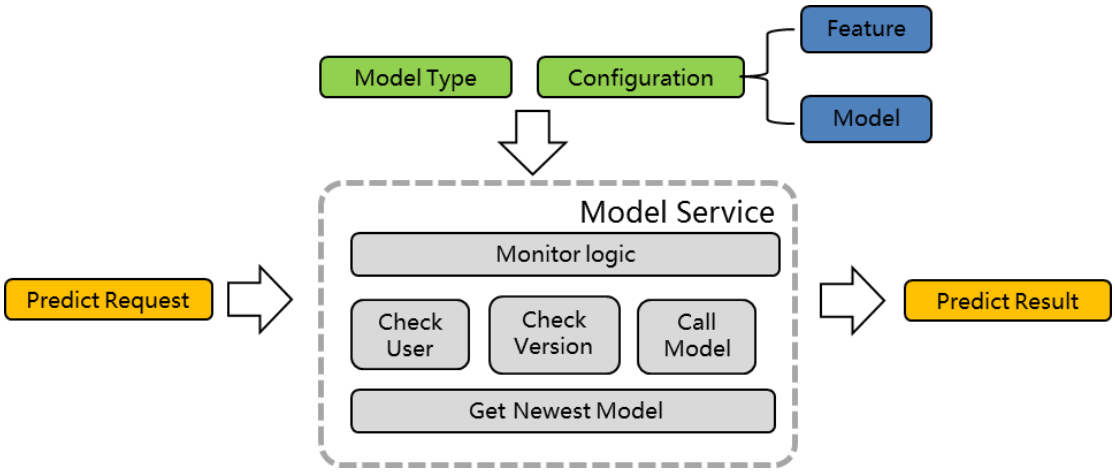


此外，产品管理模块还可以展示模型服务场景的全景图。为了帮助用户清晰地了解各个环节，产品管理关联了一个场景下的特征模块，模型模块，工程实现模块和监控模块，完整串联起了模型上线的整个生命周期，通过点击就可以跳转查看各个模块的明细信息和运行状态；尤其是多子场景的结构展示，可以方便地让相关人员了解整体结构和模型调用链路。

二、主要组件

2.1 Model Service

Model Service 即 CodeGen 自助生成的模型服务，它支持主流机器学习算法的调用，如 Pmml，Python 包，XGB 等，结构图如下。



当模型确定时，在工程中实现的方式也就确定了，此时开发工程师只需要根据模型的类型，生成对应的模型服务。出于性能上的考虑，目前的实现方式是将生成的服务包引用到工程中，在本地调用。在调用时，将处理好的待预测数据输入，就可以得到预测结果，支持单条或批量调用。当然，模型算法在不断演进中，模型引擎开发组也会负责升级模型服务，此时使用方只需要简单地升级版本即可。

同时，模型服务也封装了最新模型版本的获取功能，自动感知和下载最新的模型文件，实现模型文件更替的分钟级别生效，屏蔽特征不变场景下版本迭代的细节。此外，不同实验版本调用不同模型文件，内测人员和外网用户调用不同模型的支持也由模型服务控制，调用方只

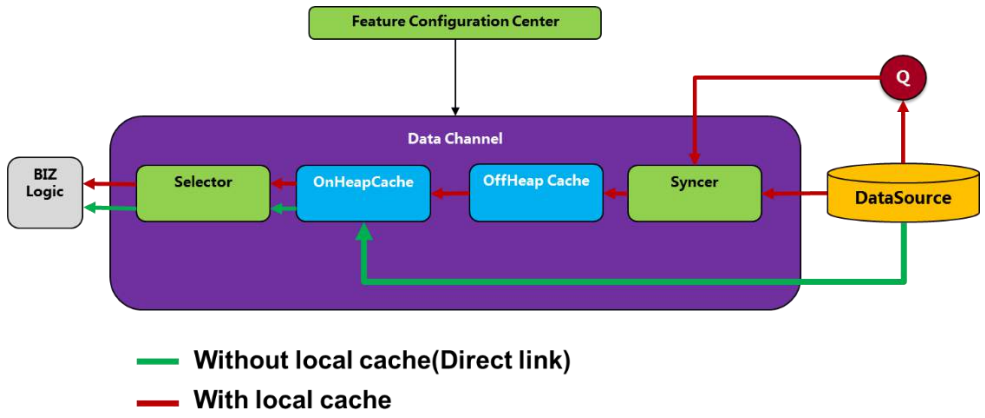
需将实验版本和用户 ID 传入即可。

对待预测数据、预测结果的记录上传和模型服务的监控也由模型服务统一负责。通过规则设定，对上传的数据进行实时判断，及时发现异常。当然开发可以根据各场景的调用量、调用耗时要求选择抽样记录部分数据，避免对存储造成压力。

除了传统意义上的模型，规则和公式也可以认为是广义上的模型。相对于狭义上的模型，公式和规则训练复杂度低，实现简单，在现阶段上线模型代价比较高昂的情况下，是不错的选择。为了支持公式、规则变更等能够实时生效，模型服务也会在后续的版本中增加对其的支持。

2.2 Feature Bus

Feature Bus 组件可以协助开发人员自动构建本地缓存，托管数据加载问题，其结构图如下所示。



通过配置 XML 文件指定数据源的存储方式、存储格式、本地缓存格式，结合 Feature Bus 的 jar 包，开发工程师可以轻松地在服务中构建本地缓存。大批量数据缓存在本地容易产生 FullGC，进而引起服务响应的暂停，因此 Feature Bus 采取了堆内缓存+堆外缓存的设计方式。堆外内存把内存对象分配在 Java 虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机），能够在一定程度上减少垃圾回收对应用程序造成的影响。对于不需要缓存的轻量级数据，也可以通过 Feature Bus 直接读取。通过统一的 API 方式，开发工程师不再需要为不同的存储方式编写不同的读取代码。

另外，在实时服务中，缓存数据需要保证数据的实时性和一致性。缓存数据的更新又分为批量更新和增量更新。批量更新一般适用于相对静态的数据，例如对酒店静态信息，需要一天更新一次。Feature Bus 支持在 XML 中指定更新的频率，根据实际需求进行批量更新。增量更新数据一般以消息的形式存在，例如用户点击行为，酒店起价变动数据等。在模型引擎的设计中，对这部分数据处理的延迟要求是秒级。Feature Bus 支持在 XML 中配置消息来源、Topic、格式，进而对指定的实时消息进行监听，实时更新本地缓存，有效地将延迟控制在 1 秒以内。不管是批量更新还是增量更新对使用方都是透明的，使用方只需要评估使用场景，进行合理的配置即可。

三、总结展望

模型引擎是携程数据服务组对日常开发工作经验的总结和升华，从最贴近实际的场景出发，为模型上线的各环节提供便利。同时，作为一个综合性平台，模型引擎也从特征质量监控、模型调用监控等方面完善了对模型服务质量的把控。

项目自 2017 年年底启动，已经打通了模型上传、审核上线、更新迭代流程，支持 AB 实验，模型灰度验证上线、多模型融合、模型特征在线诊断等功能，接入了酒店排序类、图像识别类、广告类项目等十个项目。

目前模引擎平台已经进入了二期迭代，将在丰富特征库、支持更多类型的模型、完善模型质量在线评估等方面继续发力，并积极推广到其他 BU。

“猜你所想，答你所问”，携程智能客服算法实践

[作者简介]元凌峰，携程平台中心 AI 研发部资深算法工程师，负责携程智能客服算法研发，对 Chatbot 相关的 NLP 算法和推荐排序等算法感兴趣。2015 年硕士毕业于上海交通大学图像模式研究所，后加入携程负责实时用户意图和小诗机等项目。

概述

作为国内 OTA 的领头羊，携程每天都在服务着成千上万的旅行者。为了保障旅行者的出行，庞大的携程客服在其中扮演着十分重要的角色。但在客服的日常工作中，有很大一部分的行为是重复劳动，这对于客服来说是一种资源浪费。如何从客服工作中解放生产力，提高生产效率成为技术需要解决的一大难题。

随着近几年深度学习算法突破和硬件的升级，人工智能技术在多个领域遍地开花，其中一大应用场景便是智能客服。2017 年初，携程开始大力推进客服智能化的技术研发，目前在酒店售后客服场景，智能客服已经能够解决 70%的问题，不仅提升了客服效率，在服务响应方面也有很大提升。

那么，机器学习或者深度学习在客服领域到底能做什么？怎么做？本文将围绕这两方面介绍携程在智能客服领域中的一些实践经验。

我们先回答这些算法在客服领域到底能做什么。

回答这个问题要回到客服聊天工具这个产品本身，用户在使用客服聊天工具时，最希望的是能够第一时间在客服界面上看到自己想咨询的问题，然后直接找到答案。如果第一眼没有看到想要的问题，那就希望在和“客服”交互过程中以最少的交互次数获取到需要的答案。

在这个过程中，算法不外乎要做的就是两件事：猜你所想，答你所问。

我们先说猜这件事，类似推荐，在用户还没有做出任何输入时，我们会根据用户的信息、当前上下文信息以及咨询的产品信息来猜测用户进入咨询界面时想问什么问题，从而得到一堆问题的排序展示给用户。如果第一步没有猜到用户想要的问题，用户就会通过输入框来简单描述自己的情况和想要咨询的问题，在用户输入的过程中，我们也会结合用户输入的内容通过算法来实时猜测用户可能咨询的问题，并以 input suggestion 的方式给到用户。若上述都无法让用户找到自己想要的答案，那就是答这件事要解决的。

我们会采用 QA 模型对用户发送的话术内容进行分析和匹配，得到用户可能想问的问题，并反馈给用户。这样就完成了一个对话回合，但其实除了上述提到的几个点以外，还有很多地方需要算法参与，比如问题挖掘、关联问题推荐以及上下文对话等等。

下面我们就重点介绍几种常用的算法如何发挥作用。

一、Question-AnswerMatch

标准 Q 匹配模型是机器人进行交互的基础模型，对匹配率的要求较高。传统的做法直接根据关键词检索或 BM25 等算法计算相关性排序，但这些方法缺点是需要维护大量的同义词典库和匹配规则。后来发展出潜在语义分析技术（Latent Semantic Analysis，LSA[1,2]），将词句映射到低维连续空间，可在潜在的语义空间上计算相似度。

接着又有了 PLSA（Probabilistic Latent Semantic Analysis）[3]、LDA（LatentDirichlet Allocation）[4]等概率模型，形成非常火热的浅层主题模型技术方向。这些算法对文本的语义表示形式简洁，较好地弥补了传统词汇匹配方法的不足。不过从效果上来看，这些技术都无法完全替代基于字词的匹配技术。

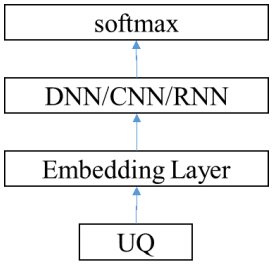
随着深度学习技术兴起后，基于神经网络训练的 Word2vec[5]来进行文本匹配计算引起了广泛的兴趣，而且所得的词语向量表示的语义可计算性进一步加强。但是无监督的 Word2vec 在句子匹配度计算的实用效果上还是存在不足，而且本身没有解决短语、句子的语义表示问题。

因此，研究者们开始研究句子级别上的神经网络语言模型，例如 Microsoft Research 提出的 DSSM 模型（Deep Structured Semantic Model）[6]，华为实验室也提出了一些新的神经网络匹配模型变体[7,8,9]，如基于二维交互匹配的卷积匹配模型。中科院等研究机构也提出了诸如多视角循环神经网络匹配模型（MV-LSTM）[10]、基于矩阵匹配的的层次化匹配模型 MatchPyramid[11]等更加精致的神经网络文本匹配模型。虽然模型的结构非常多种，但底层结构单元基本以全链接层、LSTM、卷积层、池化层为主（参考论文[12,13,14,15]的做法）。

1.1 分类和排序

在语义模型的训练框架里，大致可以分为两类：分类和排序。

采用分类的方法，一般最后一层接的是多类别的 softmax，即输入是用户 Q，分类结果是所属的标准 Q 类别。



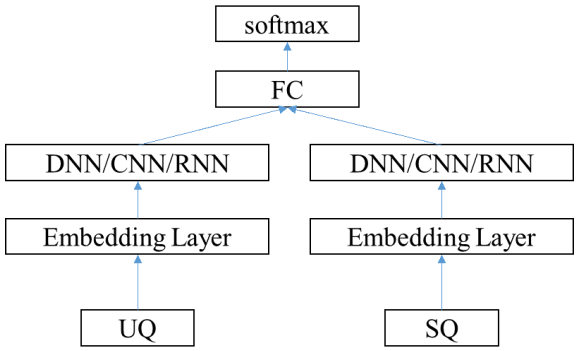
基于分类的模型结构

排序学习有三种类型：point-wise, pair-wise 和 list-wise。在 QA 中我们常用的是 point-wise

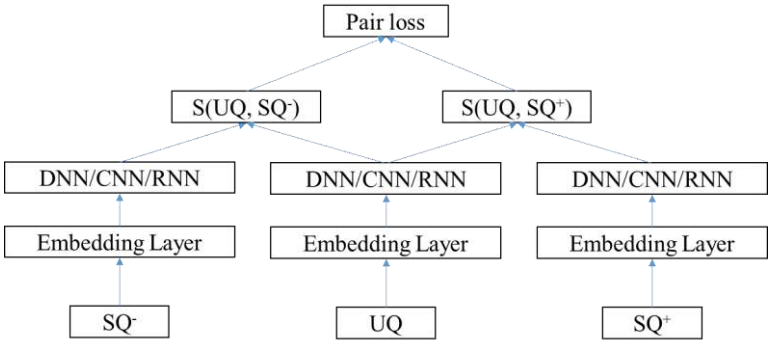
和 pair-wise。其中 point-wise 的方法直接把问题转换成二分类，判断当前用户问题是否属于带匹配的问题，最后根据隶属概率值可以得到问题的排序。而 pair-wise 学习的是和两两之间的排序关系，训练目标是最大化正样本对和负样本对的距离：

$$\max L = ||f(uq, q^{+}) - f(uq, q^{-})||_{d}$$

其中表示某种距离度量。



基于 point-wise 的模型结构

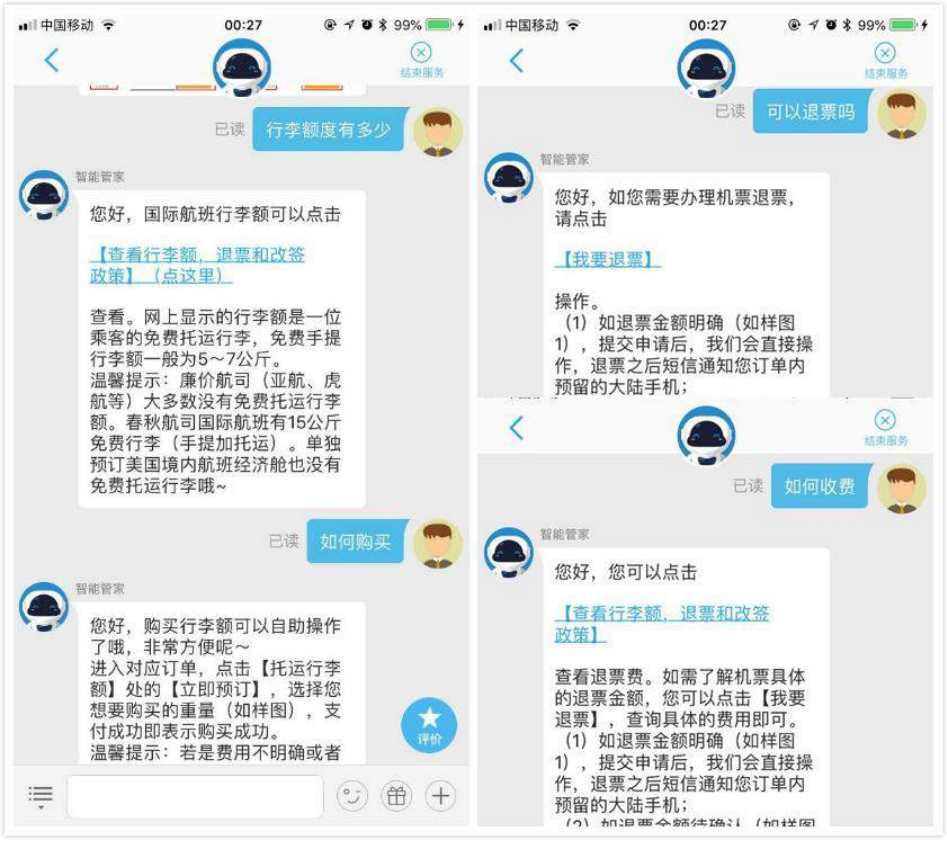


基于 pair-wise 的模型结构

1.2 单轮和上下文

一个流畅的会话是需要机器人拥有上下文多轮对话的能力，然而目前大部分 QA 机器人都是基于单轮匹配，如果用户继续上一个问题补充提问，单轮 QA 模型则会断章取义，无法准确识别当前句的准确意图。因此需要把上下文信息进行表示计算，实现多轮匹配模型。

提到多轮对话场景，大家都会想到 Task 任务式对话。但二者在上下文表示方面还是存在一些差异。Task（goal-driven system）是根据预定义的槽位和状态来表示上下文，并且依照某个业务逻辑的对话管理策略（也可以通过 POMDP[16]的方法来构建策略）来引导用户到想要搜索的内容。而 QA（non-goal-driven system）不是面向槽管理的，而是根据用户会话意图来调整对话过程。在 QA 的上下文会话管理方法中，大致可分为两个方向，一个是 Rule-Based[17]的上下文模型；另一个是 Model-Based[18,19]的上下文模型。



Rule-Based

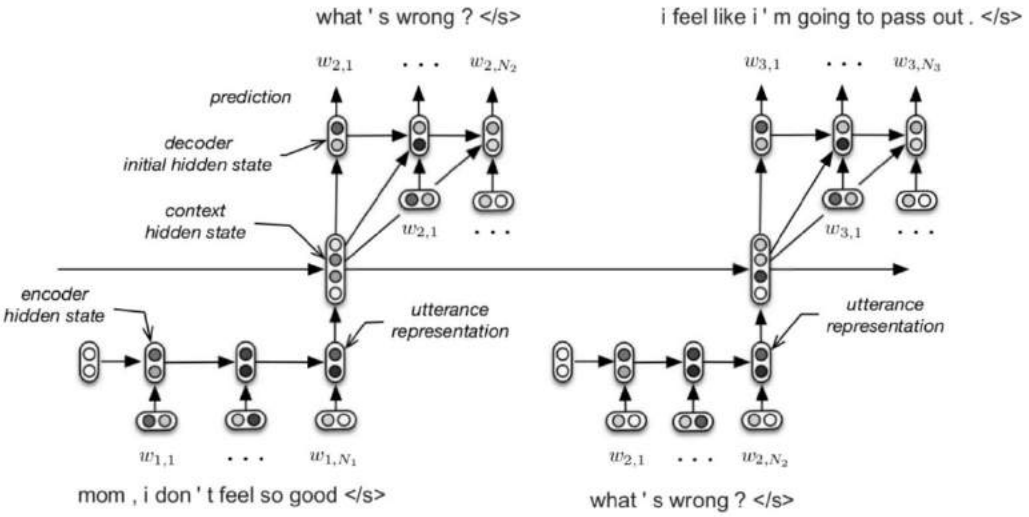
通过预定义一些先验知识来表示上下文，在会话中不断修改上下文的先验知识并根据上下文记录信息来重排序。

Model-Based

Model-Based 相对于 Rule-Based 的好处就是能够提升泛化能力。有研究者[18,19]就利用模型这种特性，把上下文信息表征在向量里，并通过层次化模型来学习和推断。该模型主要有三个结构：句子级 encoder 模型、context 级别 encoder 模型以及 response decoder 模型。其中 context encoder 模型是解决上下文信息的关键，整个模型用数学模型表示如下：

$$
\begin{aligned}
P_\theta(U_1,\ldots,U_M) &= \prod_{m=1}^{M} P_\theta(U_m|U_{<m}), \\
&= \prod_{m=1}^{M}\prod_{n=1}^{N_m} P_\theta(w_{m,n}|w_{m,<n},U_{<m}), \\
U_{<m} &= \{U_1,\ldots,U_{m-1}\}, w_{m,<n} = \{w_{m,1},\ldots,w_{m,n-1}\},
\end{aligned}
\tag{1}
$$

基于我们的场景对该模型稍作修改并训练后得到效果如下图。



user >> 为什么前台要押金，我不是给过担保金吗
robot >> 请您稍等5分钟，Demi查询下订单哦。
user >> 怎么样，是要交吗？什么时候返回给我
robot >> 您的订单是担保订单哦。
user >> 什么时候退呢
robot >> 您好，您的订单是担保订单，担保金额会在订单入住且审核成交后3个工作日内操作退款，预计会在3-7个工作日

user >> 您好
robot >> 您好,请问有什么可以帮您?
user >> 我可以换房间吗
robot >> 您好，您的订单是预付订单，房可以取消修改，您可以点击订单详情页点击【修改订单】按钮下方查看取消政策。
user >> 入住人信息写错了，可以修改么
robot >> 请您提供一下订单号的哦。
user >> 332234
robot >> 请您告知修改姓名，我为您协调一下哦
user >> 王二小
robot >> 我联系酒店申请一下，请您稍作等待哦
user >> 好的，谢谢
robot >> 不客气，为您服务是我们的荣幸~。

二、Input suggestion

输入联想在搜索引擎或者商品搜索中有着十分重要的应用场景，但客服场景的联想提示又区别于搜索，用户的输入是各式各样的，联想结果不一定和用户输入的在词语上完全一致，只需语义上一致即可，而且是能够猜中用户意图。下面聊聊我们在该场景迭代的三种算法。

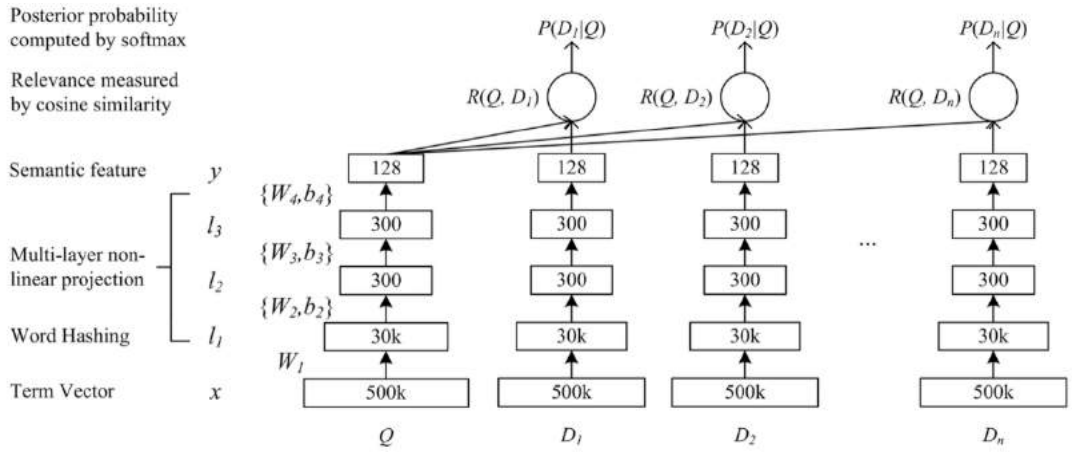
第一版本采用最常用的 Trie 树结构。字典树的结构简单实用。在搜索中，可以把所有的候选词条建立一个字典树，然后根据用户输入的前缀到 Trie 树中检索候选集，展示给用户。该结构优点是简单有效，能够快速上线。但缺点就是召回率较低，这是因为字典树要求用户输入的词语必须和候选集合里的短语句子要有一致的前缀。对此我们也做了泛化优化，例如去除掉停用词或者无意义词语等，尽可能提高召回。但提升有限。

第二阶段，我们直接采用 point-wise 的排序模型。因为第一阶段的数据积累和人工标注，我们已经有了一定的历史曝光点击数据。考虑到线上联想的请求性能要求较高，我们先尝试了逻辑回归模型。在特征方面，我们主要抽取了三类特征：一类是基于 word2vec 得到的句子特征，另一类是传统的 TF-IDF 特征，最后一类是重要词汇特征（这类特征是通过数据挖掘得到的对应场景的重要词）。该模型上线后整体的使用率比字典树有了明显提升，尤其召回率大幅度提高。

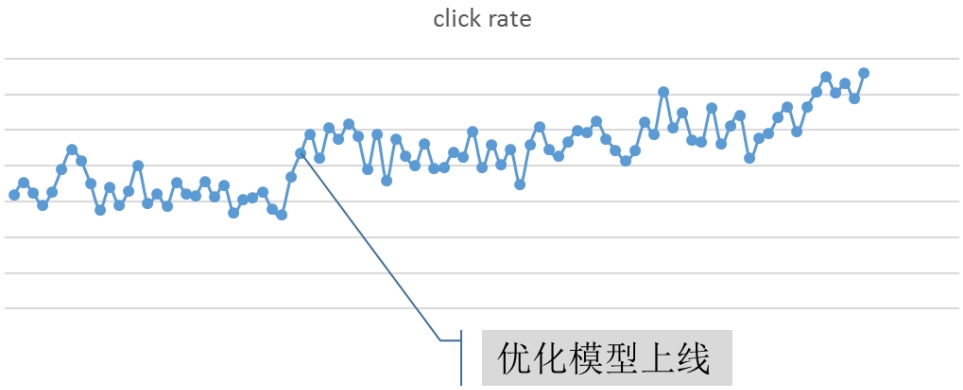
但是，第二阶段的模型仍然存在很多缺陷，我们把线上的曝光未点击数据分析了下，发现如下问题：

线上存在很多拼音汉字混搭的 case，模型没有解决能力；
用户输入的话术存在很多错别字；
联想请求场景以超短文本为主，大都集中在 2-6 个字

微软的 DSSM 模型在解决短文本语义匹配上有很好的效果，该模型主要亮点引入了 word hashing 操作，该操作能够很好的解决了 OOV（out of vocabulary）问题。其次就是深度神经网络增强了特征的表达能力，该模型计算图如下：



该模型上线后带来显著提升，整个输入联想场景的迭代效果如图。



三、总结

限于篇幅，除了上述提到的几种场景，机器学习和深度学习算法还在其他多个场景中辅助携程客服的工作，帮助提升客服工作效率和用户体验。

人工智能并不是新兴的黑科技，只不过近几年深度学习的快速发展让这个词重新活跃在大众视野中。尼尔逊教授对人工智能下了这样一个定义：“人工智能是关于知识的学科——怎样表示知识以及怎样获得知识并使用知识的科学。”

我们更愿意把人工智能看成是人工+智能算法+数据的一个综合体。算法工程师的定义不是简简单单懂算法就可以，而是要懂得如何用算法去优化人工提高效率，如何用算法去挖掘有效信息，紧密地让数据、算法、人工形成一个闭环。

Reference

[1] DennisS, Landauer T, Kintsch W, et al. Introduction to latent semanticanalysis[C]//Slides from the tutorial given at the 25th Annual Meeting of theCognitive Science Society, Boston. 2003.
[2] DeerwesterS, Dumais S T, Furnas G W, et al. Indexing by latent semantic analysis[J].Journal

of the American society for information science, 1990, 41(6): 391-407.
[3] HofmannT. Unsupervised learning by probabilistic latent semantic analysis[J]. Machinelearning, 2001, 42(1-2): 177-196.
[4] BleiD M, Ng A Y, Jordan M I. Latent dirichlet allocation[J]. Journal of machineLearning research, 2003, 3(Jan): 993-1022.
[5] MikolovT, Chen K, Corrado G, et al. Efficient estimation of word representations invector space[J]. arXiv preprint arXiv:1301.3781, 2013.
[6] HuangP S, He X, Gao J, et al. Learning deep structured semantic models for websearch using clickthrough data[C]//Proceedings of the 22nd ACM internationalconference on Conference on information & knowledge management. ACM, 2013:2333-2338.
[7] LuZ, Li H. A deep architecture for matching short texts[C]//Advances in NeuralInformation Processing Systems. 2013: 1367-1375.
[8] JiZ, Lu Z, Li H. An information retrieval approach to short text conversation[J].arXiv preprint arXiv:1408.6988, 2014.
[9] HuB, Lu Z, Li H, et al. Convolutional neural network architectures for matchingnatural language sentences[C]//Advances in neural information processingsystems. 2014: 2042-2050.
[10]Wan,Shengxian, Yanyan Lan, Jiafeng Guo, Jun Xu, Liang Pang, and XueqiCheng."A Deep Architecture for Semantic Matching with MultiplePositionalSentence Representations." In AAAI, pp. 2835-2841. 2016.
[11]Pang,Liang, Yanyan Lan, Jiafeng Guo, Jun Xu, Shengxian Wan, and XueqiCheng."Text Matching as Image Recognition." In AAAI, pp. 2793-2799.2016.s
[12] FengM, Xiang B, Glass M R, et al. Applying deep learning to answer selection: Astudy and an open task[J]. arXiv preprint arXiv:1508.01585, 2015.
[13] LaiS, Xu L, Liu K, et al. Recurrent Convolutional Neural Networks for TextClassification[C]//AAAI. 2015, 333: 2267-2273.
[14] SantosC, Tan M, Xiang B, et al. Attentive pooling networks[J]. arXiv preprintarXiv:1602.03609, 2016.
[15] KimY. Convolutional neural networks for sentence classification[J]. arXiv preprintarXiv:1408.5882, 2014.
[16] YoungS, Gašić M,Thomson B, et al. Pomdp-based statistical spoken dialog systems: A review[J].Proceedings of the IEEE, 2013, 101(5): 1160-1179.
[17] LangleyP, Meadows B, Gabaldon A, et al. Abductive understanding of dialogues aboutjoint activities[J]. Interaction Studies, 2014, 15(3): 426-454.
[18] SerbanI V, Sordoni A, Bengio Y, et al. Building End-To-End Dialogue Systems UsingGenerative Hierarchical Neural Network Models[C]//AAAI. 2016, 16: 3776-3784.
[19] SordoniA, Galley M, Auli M, et al. A neural network approach to context-sensitivegeneration of conversational responses[J]. arXiv preprint arXiv:1506.06714,2015.

知识图谱在旅游领域有哪些应用？携程度假团队这样回答

[作者简介]鞠建勋，携程度假 AI 研发团队资深算法工程师，主要负责携程度假自然语言处理相关的 AI 项目。硕士毕业于南京大学，有五年的自然语言处理经验，专注于自然语言处理和知识图谱方面的应用和算法研发。

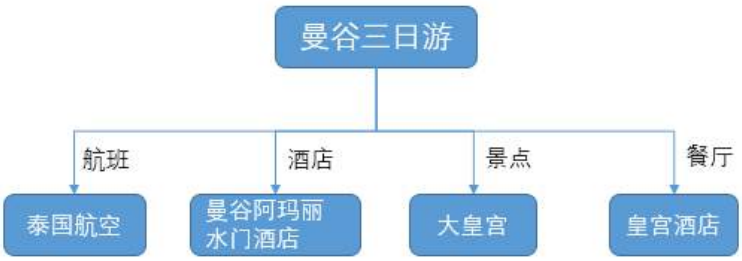
随着互联网和大数据的发展，数据呈现爆炸式增长的态势。知识图谱以其强大的语义处理能力和开放组织能力，为大数据时代的知识化组织和智能应用奠定了基础。

旅游行业作为综合性行业，包含交通、游览、住宿、餐饮、购物、文娱等多个环节，每个环节都有着海量的数据，并且有着相当庞大的应用场景。

同时，旅游领域的数据有着领域范围大，涉及知识面广，知识层级多的特点，比如一个简单的团队游产品就会和飞机、酒店、景点、火车、汽车、餐厅等等多个实体产生关联。对于旅游行业，尤其是互联网旅游行业，如何构建和应用一个旅游领域的知识图谱成为一个非常有价值的问题。

本文将从旅游领域知识图谱的特点，知识图谱的构建，知识图谱的应用三个方面介绍知识图谱在互联网旅游行业的应用。

一、旅游领域知识图谱的特点



如上图所示，曼谷三日游是一个旅游产品，它包含泰国航空、曼谷阿玛丽水门酒店、大皇宫、皇宫酒店等子产品，每个子产品都有自己不同的数据属性和架构。

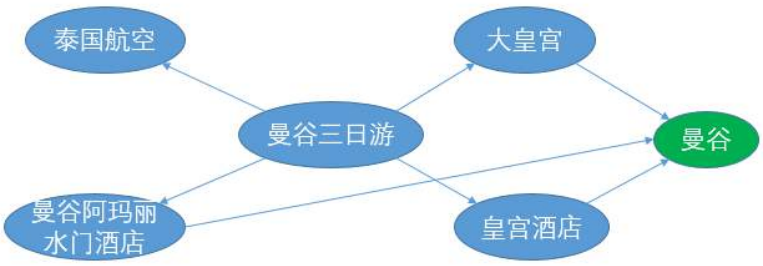
泰国航空，包含航班号、出发时间、出发地点、到达时间、到达地点等数据；曼谷阿玛丽水门酒店，包含酒店房型、酒店价格、酒店地址、酒店设施等信息；景点包含门票、地址、景点类型等信息；而餐厅有着菜品、价格、地址等重要信息。

如果这个旅游产品的信息采用传统数据库进行存储，需要多张数据表进行存储，不便于对于复杂数据的整合，也不便于进行一些涉及多领域的推理，而且一旦涉及到复杂的查询，就需

要多张表的关联，不利于知识的快速获取。

为了解决上述问题，我们构建一个图数据库去存储这些产品的数据。当数据以图的形式存在，就可以发现很多路径存在于两个事物之间，通过路径的跟踪以及一些图的算法就很容易得到一些事物之间的关系。这个图数据库就构成了一个旅游领域的知识图谱。

我们把图数据库中的泰国航空、曼谷阿玛丽水门酒店、大皇宫、皇宫酒店、曼谷、泰国等等设为节点，并称之为实体；把他们之间的关系设为节点和节点之间的边；而把航班号、酒店房型、地址等信息称之为实体的属性。



例如，游客想知道，大皇宫属于哪个国家，通过“大皇宫”->[所属城市]->“曼谷”->[所属国家]->“泰国”的“实体->关系->实体”递推关系就可以很容易找到答案；而传统数据库需要专门为查询国家写一套解决逻辑。又例如，我们想查询一下大皇宫附近的酒店价格，可以通过“大皇宫”->[附近酒店]->“xx 酒店”->[价格]->“xx 元”的关系找到答案；而传统数据库则需要多张表的联合查询才有可能完成这个查询。

此外，知识图谱还有强大的语义概括能力和语义抽象能力。我们可以把实体的类型抽象成 Class(类别)，也叫做本体，通过本体与本体之间的关系来完成一些推理或语义分析。

比如我们把大皇宫归于“古代建筑”这个类别，同时定义“古代建筑”属于“建筑”的子类，“建筑”则有“高度”的属性，那么大皇宫也有“高度”的属性。我们不需要给每个子类定义一些通用属性，而只要通过继承父类的属性就可以完成定义。

二、旅游领域知识图谱的构建

旅游领域知识图谱的构建，来自于企业数据和外部数据的融合。

首先，构建一个旅游领域的知识图谱，需要企业内部数据作为基础，因为企业内部数据是和企业产品息息相关的，而产品是一切应用的基础。

旅游产品中的信息大部分作为基础数据存在，比如酒店房型，酒店地址等等，我们需要这些信息来构建酒店的知识图谱。而对于一些变化非常频繁的数据，则不会导入到知识图谱中，比如售卖数量等等。我们把这类知识图谱统称为行业知识图谱，它具有高深度和专业性的特点。

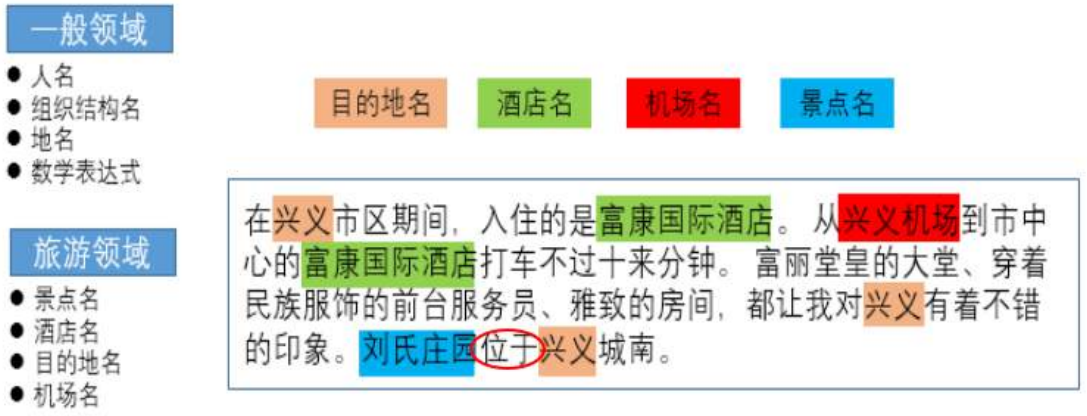
此外，我们还需要用到外部数据作为补充，比如我们经常会用到 wiki 上的常识信息来补充知识图谱，我们把这部分知识图谱称为通用知识图谱。

通常常识信息不会存储在企业内部数据库，且具有非常大的广度，比如大皇宫的面积有多大，高度有多高等等。这些常识信息对于产品售卖可能没有太大作用，但对于回答客户的问题或提供搜索依据是有很大帮助的。

通用知识图谱的广度和行业知识图谱的深度相结合，可以互相补充，形成更加完善的知识图谱。

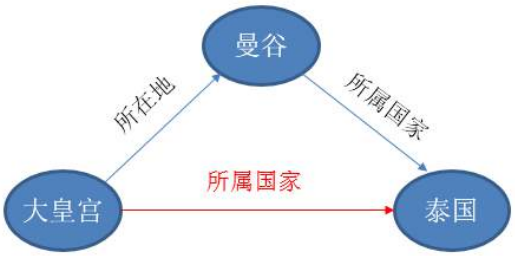
外部数据经常会以大段文本的形式存在，因此我们需要从文本中抽取关系。通常采用自然语言处理的 NER 方法去提取文本中的实体，结合句法分析、远程监督等传统方法来提取实体与实体之间的关系。

如下图所示，我们可以抽取出刘氏庄园这个景点名，同时可以抽取出兴义这个目的地名，也可以挖掘出它们之间的“位于”关系。那么这段文本中的“刘氏庄园”->[位于]->“兴义”这条知识图谱数据就成功被抽取出来了。



知识图谱的存储可以选择 RDF 数据库，或者图数据库。RDF 数据库是 W3C 标准的以三元组的形式存储的知识图谱，有标准的推理引擎，易于进行推理和关系发现；而图数据库则以节点和边的方式存储知识图谱，节点和边都可以带属性，没有标准的推理引擎，但图数据库的遍历效率高，性能更好，更适合企业级的海量数据。

当构建好旅游领域的知识图谱之后，可以对知识图谱进行补全。比如大皇宫的所属城市是曼谷，而曼谷是泰国的一个城市，假如缺失了“大皇宫”->[所属国家]->“泰国”这条关系，我们可以通过关系补全来把这条关系加入知识图谱。关系补全的方法有很多，如 TransE、TransH 和 TransR 等，本文就不一一阐述了。



三、旅游领域知识图谱的应用

知识图谱的应用很广，结合知识图谱和旅游领域的特点，旅游知识图谱可以应用在以下几个方面：

3.1 旅游问答机器人

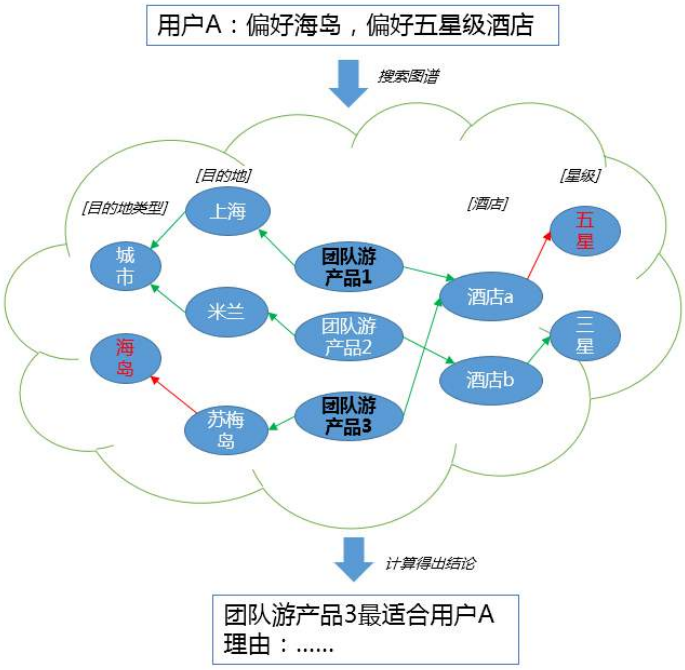
通常问答机器人不擅长去“寻找”答案，而是需要人来提前“设定”好答案。但有了知识图谱之后，这种情况会得到一些改变。

基于知识库的问答系统我们称之为 KBQA（Knowledge based Question Answering），而知识图谱是知识库的一种。KBQA 更擅长回答“what”“when”“where”问题，只要捕捉到问句中的实体和关系，就可以在知识图谱中“寻找”到相应的实体以及它的属性，而通过实体的属性可以解答大部分的“what”“when”“where”问题。

目前有很多问答模型利用知识图谱来解答问题。通常把实体和关系作为一种特征，参与问句意图的计算。我们首先解析问题的语义特征，提取问题中的实体和关系，然后将实体与关系映射到知识图谱中，把知识图谱中对应的实体属性返回作为答案，或者将知识图谱中的实体、关系、属性作为特征，参与下一步的模型计算。

3.2 旅游内容推荐

旅游产品非常多样化，所以很依赖推荐。我们往往会抓住旅游产品和用户的浅层特征去推荐，而忽略了一些深层次的特征。当用户和某个知识图谱实体产生关联，我们可以用知识图谱去补充用户特征。



如上图所示，当我们通过数据挖掘发现用户 A 的兴趣点是海岛和五星级酒店后，我们可以通过知识图谱关联出相关的团队游产品。知识图谱提供的信息可以作为推荐系统的一个重要维度，参与下一步的计算，为精准推荐增加一块砝码。

3.3 旅游内容搜索

旅游产品的数量很多，因此对于搜索的要求很高。对于一些复杂的搜索语句，知识图谱可以利用自身的推理能力搜索出深层的答案。

例如，用户在搜索框输入“大皇宫附近的酒店”，知识图谱可以通过“大皇宫”->[附近酒店]->“xx 酒店”搜索出相关的酒店。如果用户在搜索框输入“大皇宫附近酒店的价格”，知识图谱可以通过“大皇宫”->[附近酒店]->“xx 酒店“->[价格]->”xx 元“搜索出酒店的价格属性。知识图谱利用了图数据库的优势，非常方便地进行图遍历，加强了搜索引擎解决复杂搜索的能力。

四、总结

知识图谱在旅游场景还有很多其他应用，本文就不一一列举了。

知识图谱这项技术还处于发展阶段，还有很大的提升空间，结合日益增长的旅游市场，我们非常看好知识图谱应用于旅游领域的前景。我们相信，未来的知识图谱还会有更多的应用和更多的技术，为互联网旅游行业的发展添砖加瓦。

如何选出最“美”图片展示给你？携程做了基于深度学习的图像美感评分系统

[作者简介]路婵，携程度假 AI 研发团队算法工程师，专注于计算机视觉和机器学习的研究与应用。现阶段致力于度假图像智能化，多次参加国内外数据竞赛并获奖。

概述

作为 OTA 行业的领跑者，携程每天服务成千上万的客户。景点的图片介绍成为用户了解景区、玩乐产品的重要参考。

为了给用户带来更好的使用体验，携程门票列表页通常由人工筛选更具代表性的优质图片，指定为每个产品的首图。这种人工指定的方式主观性强，费力度大，无法做到准确而及时的更新。面对大量景点图像，如何智能选择更优质的图像，提高用户满意度，改善用户体验，大幅减少图像的人工干预，成为急需解决的问题。

近几年深度学习在图像领域取得了突破性的进展，基于深度学习的应用也层出不穷。深度神经网络对图像在特定目标域拥有极强的感知与决策能力，将其应用到图像美感评价上，可以综合图像的美学与语义信息，自动选择更优质的首图，为用户带来更好的体验。优化前后对比如下：

一、技术简介

构建智能美感评分系统，旨在自动筛选更优质的图像进行展示。本文还通过 CAM 可视化方法，对网络的决策进行了解释。在业务处理中，通过识别图像类别，筛选更适合的图片作为首图进行展示。

1.1 图像美感评分

图像美感的量化是图像处理和计算机视觉中的一个问题，其主要目的是预测与人类感知相关的质量分数。与图像质量评价（Image Quality Assessment,IQA）处理的像素级的退化（degradation）问题不同，美感评价提取图像中与情感和美感相关的语义层次特征。

图像美感评价主要分为两个部分：特征提取与决策。

传统特征提取方法通过人工设计的低层特征（颜色、纹理、清晰度等）和高层特征（景深、区域对比度等），作为图像的美感特征，通过训练一个分类器或回归模型，得到图像的美学质量评分。

相较于传统方法，深度卷积神经网络拥有强大的自动特征学习能力，在图像美感评价方面展现出良好的性能，成为解决该问题的主流方法。

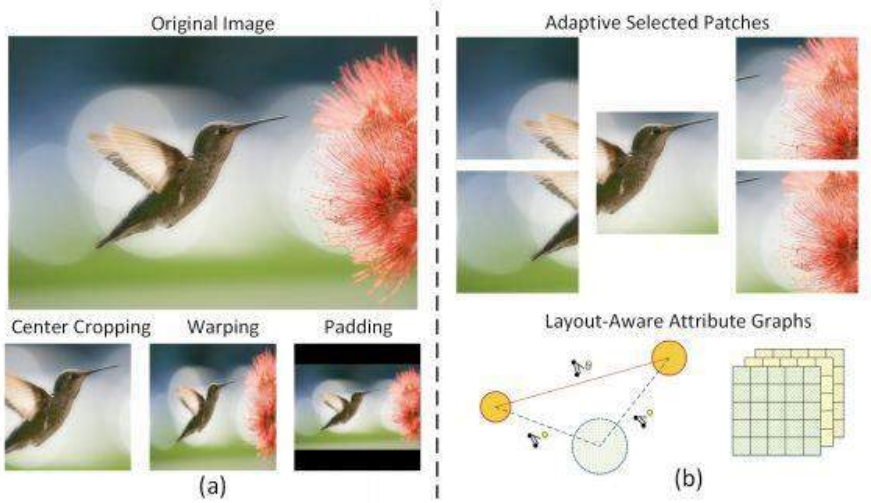
1.1.1 研究现状

美感的评估难点主要集中在全局语义与局部信息的特征抽取上。

为了兼顾全局语义特征与局部信息，研究者往往采用整张图像与多次裁剪得到的图像片段（multipatch）同时作为输入。

Lu 等的 RAPID 模型[1]基于 AlexNet，采用多路网络，分别将整张图像与多次随机裁剪得到的部分图像作为输入，提取全局美学特征、局部美学特征，将质量评估作为二元分类问题，对图像美感进行预测。

此外，作者还通过使用图像风格监督（style-column）或者图像语义监督（semantic-column）CNN 合并图像风格信息来进一步提高网络的表现。A-Lamp 架构[2]采用特定的多次裁剪得到而不是随机剪裁，并结合图像整体,更有效地评估图像美学质量。多次裁剪输入示意图如下：



在美感度模型的训练框架里，多使用分类的方法进行最后评分的决策。

采用分类的方法，一般最后接一层 softmax，根据监督信息的不同，可以是判断图像好看与否的二元分类，或者是对多人评价分布的拟合。前者将输出好看的概率作为美感度分数，后者将十个等级的评分求加全和得到最终结果。

Google[3]提出 NIMA, 通过学习每张图像的评分直方图, 对任意给定的图像预测评级分布。该方法能够更准确的预测出人类的偏好。

不同于以往的二分类处理，Kong 等[4]提出对图像美学进行排序，构造了新的数据集，对每张图片的 8 个美学指标进行打分。通过对每个采样图像对的排序学习建立基础美学特征提取器，并添加属性预测等内容分支，使用多任务学习方式进行微调。

1.1.2 应用

在实际应用中，我们用基于 ImageNet 预训练的 Resnet 作为特征提取基础模型。在网络输入上，我们有选择地采用多次裁剪得到的多个图像块，以及整张图像填充（padding）后缩放到固定尺寸作为输入，以兼顾全局信息与局部特征。

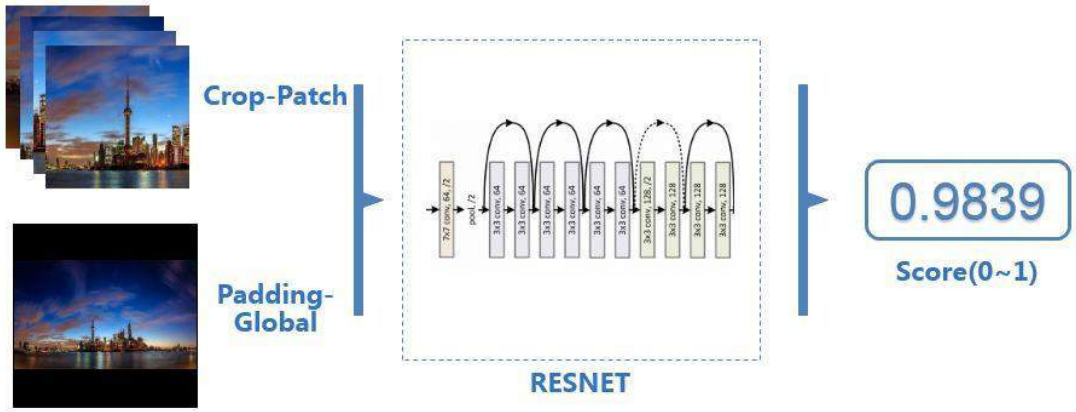
数据增广的处理上，常用的数据增广方法难以应用在该问题中，剪裁会对图像构图产生负面影响，缩放产生的形变影响了实际的比例和分辨率，因此我们只使用了水平方向的翻转。

值得注意的是, 分数分布在中间档的数据通常处在模棱两可的状态, 其数据量占比也比较大，采用二分类的方法，若强行设定一个阈值来区分低质量和高质量，会带来许多错误的标注。因此, 在正负样例的划分上需要特别注意不要引入这些模糊数据。由于美感数据集获取不易，半监督、hard-mining 是优化模型的重要方法。

决策部分我们分别尝试二分类与 1-10 评分分布直方图学习两种决策方式。

由于评分分布需要大量人工标注，难以迁移到实际场景的数据中，我们只用该方法对美学数据集进行了训练，以提升特征提取的效果。

基于上述基础模型，我们又加入了自己标注的实际场景数据，替换最后决策层为二分类 softmax，进行微调。最终将好看的概率作为美感度分数。美感评分网络示意图如下：



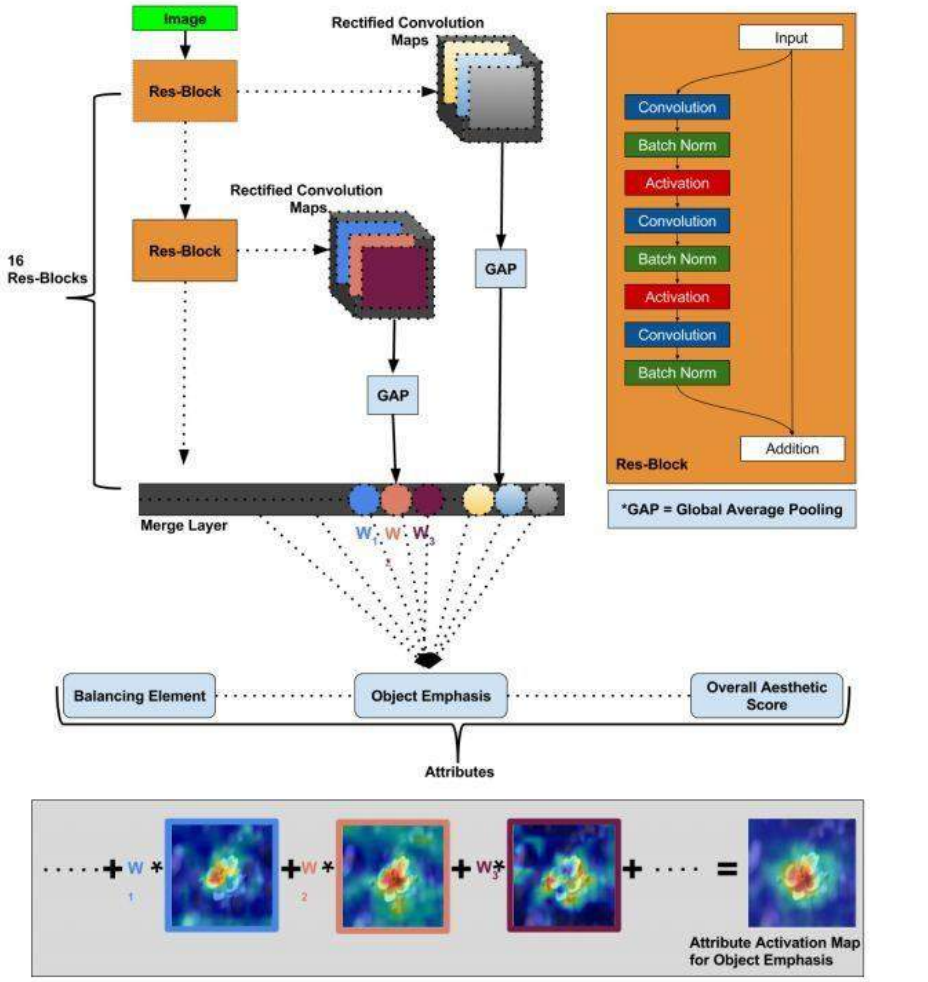
1.2 可视化

通过对图像的美感度打分，我们可以量化图像美感这一抽象概念。但是这些方法并不能直观的说明为什么一张图被判断为好看、不好看，以及是哪些方面影响了这张图像的美感。

为了解释网络对图像美感的决策，我们尝试了 CAM 等可视化方法。

CAM 或梯度加权类激活映射(Grad-CAM)这类方法，都是通过得到每对特征图对应的权重，最后求一个加权和，通过对高层特征图的权重加权计算，可以达到很高的类别判别能力。这类方法使用流入 CNN 最后一层卷积层的分类权重或梯度信息来理解每个神经元对于目标决定的重要性，解释网络可能做出的决策。[5][6][7]

可视化方法与效果示意图如下：



1.3 图像内容匹配

实际业务场景中，数据种类众多，在候选图集中还有许多图像不适合作为首图进行展示，比如表演时刻表、景区美食等不符合景区主题的图像。

为了选出更优质的图像，我们在美感评分基础上，进行了图像内容筛选的后处理。由于后处

理的类别并非互斥关系，我们建立了一个多标签的分类训练模型，为每张图片打上相应的类别标签，只选取符合要求的类别图片，在其中使用高分图片最为最终首图。筛选最优质图片示意图如下：

二、小结

深度学习在图像识别领域取得了很好的成果，利用 DCNN 能够有效提取图像的特征，描述图片的美感。图像的美感问题是个高度抽象的问题，数据量相对较少的情况下，需要更深且权值级别更低的网络。为了得到较为全面的图像美学特征，图片全局和局部都有体现美感的特征，需要结合两者，才能更有效的对图像质量进行评估。

相关文献

[1] Lu, Xin, et al. "Rapid: Ratingpictorial aesthetics using deep learning." Proceedings of the 22ndACM international conference on Multimedia. ACM, 2014.
[2] Ma, Shuang, Jing Liu, and Chang Wen Chen."A-lamp: Adaptive layout-aware multi-patch deep convolutional neuralnetwork for photo aesthetic assessment." Proc. IEEE Conf. Comput.Vis. Pattern Recognit.(CVPR). 2017.
[3] Talebi, Hossein, and Peyman Milanfar."Nima: Neural image assessment." IEEE Transactions on ImageProcessing 27.8 (2018): 3998-4011.
[4] Kong, Shu, et al. "Photo aestheticsranking network with attributes and content adaptation." EuropeanConference on Computer Vision. Springer, Cham, 2016.
[5] Zhou, Bolei, et al. "Learning deepfeatures for discriminative localization." Proceedings of the IEEEConference on Computer Vision and Pattern Recognition. 2016.
[6] Selvaraju,Ramprasaath R., et al. "Grad-CAM: Visual Explanations from Deep Networksvia Gradient-Based Localization." ICCV. 2017.
[7] Malu, Gautam, Raju S. Bapi, and BipinIndurkhya. "Learning Photography Aesthetics with Deep CNNs." arXivpreprint arXiv:1707.03981 (2017).

全球顶级算法赛事 Top5 选手，跟你聊聊推荐系统领域的“战斗机”

[作者简介]朱麟，携程酒店研发部排序算法组资深算法工程师，主要负责携程酒店排序相关的 AI 项目，多年行业相关经验。博士毕业于中国科技大学，专注于推荐系统算法的应用和研发。

摘要

随着人工智能和大数据技术的飞速发展，推荐系统近年来非常流行，应用于各行各业。推荐的对象包括：电影、音乐、新闻、书籍、学术论文、搜索查询、分众分类、以及其他产品。

推荐系统产生推荐列表的方式通常有两种：协同过滤和基于内容的推荐，近年来很多基于这些方法的创新层出不穷。

携程酒店研发部排序推荐算法团队也在该领域的实践中作了很多有意义的探索，并在业务中得到应用。带着这些积累，我们在今年参加了一些关于推荐系统的国际知名数据挖掘竞赛，2018 ACM WSDM 挑战赛和 2018 ACM RecSys 挑战赛，均取得 Top5 的好成绩。

以下将结合两次比赛的实际内容，谈谈我们所采用的算法策略和心得。

一、比赛简介

2018 年 ACM WSDM 竞赛[1]，作为网络数据挖掘顶级学术会议 WSDM 的一部分，由 ACM 和亚洲顶级的音乐流媒体服商 KKBOX 联合举办。 比赛目标是搭建一个推荐系统，通过预测在一定时间内用户再次点播历史收听过歌曲给用户进行推荐，竞赛使用的数据集由音乐流媒体平台 KKBOX 提供，包含以下信息：用户、歌曲的 metadata，听歌活动与 App 的信息等等。

2018 年 ACM RecSys 竞赛[2]，作为推荐系统顶级会议 RecSys 的一部分，由 ACM 和国际知名音乐服务商 Spotify 联合举办。比赛的目标是搭建一个推荐系统，自动延续用户歌单。竞赛所用的数据集由 Spotify 提供，包含以下信息：用户、歌曲的元数据,，一百万个用户自建的歌单以及这些歌单的元数据。

二、方法创新

2.1 对于特征工程的创新

坊间常说：“数据和特征决定了机器学习的上限，而模型和算法只是逼近这个上限而已”。可见特征工程在机器学习过程占有非常重要的地位。

在 ACM WSDM 竞赛中，我们在特征工程上做了充分的探索 ，除了构造常规的类别和统计特征，值得一提的是我们挖掘出了蕴藏在数据中的时序信息。考虑到给定的数据集是按出现的时间顺序排序的，所以尽管没有给明显的时间戳，我们依然可以从两方面探索考虑到输入数据的时间序列特性的特征：

2.1.1 物品的“年龄”相关特征

不像通常推荐比赛的设定，物品集和用户集都被假设是固定的，在 KKBOX 提供的为期两年的数据集（2016 和 2017）中，新发的歌曲被不断地加入到音乐库，新的用户也不断地使用 KKBOX，可以理解为新的物品和用户不断加入到已有推荐系统。

一方面，推荐近期新发歌曲是非常重要的，因为在实际中用户往往偏好新的内容；另一方面，考虑歌曲的“年龄”和用户的“年龄”对偏差纠正也是很重要的，这是因为大量的物品和用户在数据集中不是“均一”分布的。

基于这些观测，我们首先构造了一些特征来衡量对应的客户/歌曲/艺术家/作曲家/谱曲家在此次用户-歌曲听歌活动前的出现时间。相似地，我们也构造了一些特征，表示对应客户/歌曲/艺术家/作曲家/谱曲家在此次用户-歌曲听歌活动后出现频率。

2.1.2 “会话”特征

对于在线流系统的推荐任务而言，通常从基于会话的角度来考虑用户和系统/物品更为有益。会话是一组发生在给定时间间隔内的相互联系。

举个例子，一个用户可能在较短的时期内，会不时地听属于同一个艺术家或者同一个风格的音乐，识别这样的会话很有可能提高推荐准确率。

之前提到过，在此次竞赛中，听歌活动的时间戳并没有被直接给出，因此取而代之的是，我们贪心地将临近的属于同一个用户的听歌记录归为一个会话。

基于估计的会话，我们构造了三种特征，分别衡量用户在数据集中参与会话的数量，用户在一个会话中平均听歌数量，当前会话含有的歌曲数量。

在 ACM RecSys 中，我们考虑到出现在同一歌单的歌曲共现的性质，借用 word2vec 的思路，将歌单当作句子，将歌曲当作单词进行训练，得到每一首歌的 embedding，并基于这些 embedding 计算歌曲间相似度作为特征。

2.2 对于模型的创新

在实际的推荐系统应用中，各种机器学习方法百家齐放。能够基于已有方法，针对不同的实际问题作出自己的创新，集百家所长，往往能带来更优异的结果。

2.2.1 解决冷启动问题

冷启动问题是协同过滤推荐算法中被广泛关注的一个经典问题，常常为实际推荐系统带来严重的挑战，它的存在严重影响了推荐系统的推荐质量。对于电子商务推荐系统，每天都有大量的新用户访问系统，每天都有相当数量的新项目添加到系统中冷启动。

在 ACM WSDM 竞赛中，我们亦遇到了同样的挑战。

在模型里，客户和歌曲都被表示成类别特征，所以此环境下的冷启动问题可以被理解成带有高基数的类别特征的"维度诅咒"：因为训练集的有限，不太可能观察到每个这样类别特征的可能的值，所以，如果学习模型过于依赖这些不大可靠的类别特征提供的信息，特征可能不能很好地推广到未来的测试集上。

在本次竞赛中，我们尝试通过借助去噪自编码和 dropout 的思想来改善这种问题，并且在没有高基数类别特征，像用户 id，歌曲 id，艺术家名字，作曲家，谱曲家等等的情况下重新训练模型。在这样去掉不可靠特征的情况下训练出的模型，和原模型融合后会得到更好的实验表现。

2.2.2 创新协同过滤

作为推荐系统经典方法的协同过滤可以分为基于用户和基于物品两种。

基于物品的协同过滤简单来说是利用某兴趣相投的群体的喜好来推荐用户感兴趣的信息，应用很广。但是，基于物品的协同过滤也存在缺点，往往会更倾向推荐热门的物品，使得一些小众的物品得不到重视而较少被推荐。

在 2018 ACM RecSys Challenge 中，我们针对基于物品的协同过滤进行一些改进，得到了较为不错的模型效果。

该竞赛的挑战目标，是开发出一个自动延续歌单系统。比赛提供的数据集可以简要概括成两部分，第一部分是百万歌单数据集（MPD），包含一百万个由用户创建的歌单和相关的元数据，比如歌单的名字，描述，艺术家/专辑/歌曲的数量等多种统计信息。

第二部分是一万个未完成的歌单和相关元数据， 分别含有 1-250 首歌不等歌单。评测指标是对该一万个未完成的歌单，从比赛给定的一百万首歌曲预测出最可能在该歌单的 500 首歌曲。

我们推荐的核心思想是假设给歌单 u 自动延续（推荐），将和歌单每首歌曲平均相似度最高的歌曲选出，来自动延续歌单。如何衡量这些歌曲和歌单的相似度?

一个简单直接的方法是，计算歌曲和歌单每首歌曲平均相似度。在参考一些以前的工作后，我们将该计算分为了以下三步：

(a) 计算每首歌的特征向量

对于每首歌曲，我们都可以得到它的特征向量，已知数据集中有一百万的歌单，那么代表每首歌的特征向量将为一百万维，每一维的计算可见以下公式：

$$(X_i)_{[v]} = \begin{cases} \dfrac{\left|T(u) \mathrm{I}\ T(v)\right|}{\left|T(v)\right|^{\alpha}} & i \in v \\ 0 & else \end{cases}$$

T(u) - 歌单 u 的歌曲集合

T(v) – 对于任何包含歌曲 i 的歌单的歌曲集合

$\alpha \in (0,1)$ - 超参，控制着长歌单的影响

(b) 计算歌曲 i 和歌曲 j 的相似度

$$s_{i,j} = \frac{(x_i)^T x_j}{\left|P(i)\right|^{\beta} \left|p(j)\right|^{1-\beta}}$$

P(i) – 含有歌曲 i 的歌单集合

P(j) – 含有歌曲 j 的歌单集合

$\beta \in (0,1)$ - 超参，控制着热门歌曲的影响

(c) 歌曲 i 和歌单 u 的相似度

通过(a)和(b)计算得到歌曲与歌曲之间的相似度，对于一个未完成的歌单 u，我们计算歌曲 i 和歌单 u 中每一首歌曲的相似度，加权平均分即为该歌曲与歌单 u 的相似度。

$$r_{ui} = \frac{1}{\left|T(u)\right|} \sum_{j \in T(u)} s_{ij}$$

通过(a)和(b)计算得到歌曲与歌曲之间的相似度，对于一个未完成的歌单 u，我们计算歌曲 i 和歌单 u 中每一首歌曲的相似度，加权平均分即为该歌曲与歌单 u 的相似度。

至此，我们实现了基于物品的协同过滤推荐，通过超参α和超参β得以控制当歌单较长以及物品过于热门的影响，但是该方法步骤(c)中的加权平均计算歌曲与歌单相似度过于平等得看待所有的特征，使得相对的特征重要性被忽略了只能得到次优解。

针对以上问题，我们提出判别式重新加权方法（Discriminative Reweighting）进一步改进。

2.2.3 判别式重新加权

我们采用的判别式重新加权算法主要借鉴了 SLIM 算法。

SLIM(Sparse Linear Model) [3]，中文名是稀疏线性推荐算法，该方法基于物品相似度的推广形式， 以 M 表示评分矩阵，S 表示相似度矩阵，在计算评分时作为对应物品的权重，优化目标即为使得 M 和 MS 差值最小，同时通过添加对权重的正则使得权重更加稀疏以此达到更好的模型推荐效果。

相似 SLIM 模型，对于以上平均看待权重而忽略特征重要性的问题，通过求解以下 L2 正则 SVC 问题，我们可以有效地学习到更多具有判别意义的权重，从而学得更精准的歌曲和歌单的相似度：

$$\min_{w} \frac{1}{2} w^T w + C \sum_{i \in T} (\max(1 - y_i w^T s_{ui}, 0))^2$$

其中标签 y_i 表示 i 是否属于 $T(u)$。

2.2.4 判别式重排序与模型集成

如今推荐系统领域，乃至机器学习各领域，常常用多种不同类别的模型集成来完成推荐，比如 Facebook 利用 XGBoost 和 LogisticRegression 集成，得到的模型在点击率得以极大的提升。多模型集成不仅能充分挖掘数据特性，也能更好综合预测结果，提升模型表现。

在 ACM RecSys 挑战赛中，考虑到协同过滤方法仅仅依靠歌曲和歌曲，歌曲和歌单共同出现的频率来计算相似度，但还有其他数据，像歌曲名，歌单名等等的信息我们尚未使用，因此我们集成了原有的协同过滤模型和 GDBT，将协同过滤做粗排，筛选出初步的候选集，然后通过元数据构造特征，与协同过滤计算的概率分特征一起，通过 GDBT 进一步作精排，得到最终的推荐结果。

在 ACM WSDM Cup 中，通过嵌入式模型，如 Factorization Machine[4]或深度神经网络，将类别 id 嵌入低维度空间来表示潜在的偏好，这种方法能推广到先前未观测到的类别特征对，因此除了基于特征工程的分类模型-Gradient Descent Boosting Tree(GDBT)，我们还采用了 factorization machine 来学习每个用户歌曲对的潜在因素，并基于观测到的似然值而排序给定的歌曲。

三、总结

在推荐系统领域，通过对数据集不断挖掘更为有效的特征，针对已有方法在特定问题上的创新，以及更充分有效挖掘数据特性的模型集成往往能带来更加优质的推荐，带来更好效益的模型，更充分发挥人工智能的优势去服务广大的消费者。

在携程的日常服务中，个性化的推荐算法以及不断提升的推荐质量，也将为旅行者带去更良好的消费体验，找到让每一位旅行者都满意的旅行产品。

（携程酒店研发部陈毅鸿，何博文对本文亦有贡献）

参考文献

[1] Y.Chen, X. Xie, S.-D. Lin, and A. Chiu, "WSDM Cup 2018: Music Recommendationand Churn Prediction," in WSDM,Marina Del Rey, CA, USA, 2018, pp. 8-9.
[2] C.-W. Chen, P. Lamere, M. Schedl, and H.Zamani, "Recsys challenge 2018: automatic music playlistcontinuation," in RecSys, 2018,pp. 527-528.
[3] X. Ning and G. Karypis, "Slim: Sparselinear methods for top-n recommender systems," in ICDM, 2011, pp. 497-506.
[4] S. Rendle, "Factorization machines,"in ICDM, 2010, pp. 995-1000.

携程实时智能异常检测平台的算法及工程实现

[作者简介]陈剑明，携程网站运营中心数据分析高级经理，负责网站容量规划、ATP 基线预测及 RCA 损失计算、成本分摊、运维数据仓库建设，利用机器学习和深度学习相结合，进行运维方向的数据分析与预测。本文来自陈剑明在“2018 携程技术峰会”上的分享。

在运维领域，异常检测是很重要的基础，决定了告警的数量和质量，也影响了发现异常之后所需要采取的行动的可靠性。

然而实际生产环境下异常的判断往往带有主观性，准确率、召回率等指标缺乏客观依据，标注成本又高，导致误报和漏报始终都是一对矛盾的存在。

本文主要介绍我们在这些已知问题基础上，进一步优化“异常检测”算法的探索和成果，并介绍了如何解决告警实时性的工程技术方案。

一、引言

日常工作中我们经常会接收到频繁的异常告警，处理起来眼花缭乱，容易遗漏问题点。如何降低误报率，让有限的注意力集中在真正需要关注的异常上？

在机器学习领域异常检测是个很大的主题，传统的统计学方法就有很多，深度学习领域也有不同的算法模型来检测异常，很多论文都对此做过研究和探索，然而直接借鉴并不能让我们获得理想的效果，需要一种方法可以帮助我们：

- 降低报警总量到可以人工逐个处理的程度
- 不能以增加漏报真正的故障为代价
- 提升告警的实时性
- 算法即服务，有较强的可移植性

二、大而全的监控衍生出的问题

不管运维还是开发，大家都明白一个道理，系统跑得好不好，监控工具少不了，监控是我们的眼睛。

特别是规模比较大的系统，我们需要一个大而全的实时监控系统，如果这个系统还能判断业务异常，并能实时发送给相应的人员来处理，那就更棒了。

DevOPS 火了这么多年，相信很多同仁也在自己的公司实施部署了具备这样能力的监控系统，我们想和大家探讨的是，接下来可能会面临一些什么问题，以及我们一路是怎么走过来的。

当我们有了一个这么强大的实时监控告警系统，将几千上万数十万个监控指标接入进去的时候，问题就来了，这么多指标该如何去设置告警？

不那么重要的指标可以不设告警，等出问题的时候再来追踪查看；有些系统级的指标可以设置简单的告警规则，比如 CPU 使用率、磁盘利用率，超过一定百分比报出来就好；还有很多指标不那么好设置统一的规则，比如访问量，响应时间，连接数，错误量，下单量，怎么样才算是不正常？这在很多时候是个感性的判断。

为了让自己变得理性，我们自然而然地会去量化，常见的量化方式是和过去的某一个时间相比较，比如和一周前的同一时刻前后几分钟的均值相比，降幅如果超过了一定比例，并且连续出现，就判断为异常。这种方式面临两个局限：

1、成千上万个指标，需要人工去设置，费事费力，业务变化会导致数据形态上的变化，规则维护成本很高；

2、每个指标的业务背景不同，降幅超过多少才算是异常，这是个艺术，没有标准答案，每个人都有自己的判断，需要经验。

如果监控系统是我们的眼睛，那么报警系统就是神经。面对这么多监控数据，怎么样让神经既不那么敏感，也不那么大条，还能自动适应各种刺激，这是我们尝试去探索和改善的问题。很自然的，我们希望引入一套算法来解决这个问题。

三、统计模型的困扰

一旦引入算法，在整个建模过程中，绕不开两个问题：明确的算法评估标准和足够的样本数量。

首先，明确的评估标准，在算法迭代和检验阶段非常重要，否则算法调优就没了方向。虽然二分类问题在理论上有召回率和准确率这些评价指标，但这两个指标在我们这种检测场景下本身不可衡量。

原因是不管监控如何强大，总有我们发现不了的异常，而一个人认为的异常，换一个人来看，也许就不认为是异常。异常本身没有明确的定义，也没有一个全量的边界，如何检验一个算法的好坏？

其次，生产系统中能够确定的异常，相对于监控的数据总量来说，样本非常的少，需要大量标注，标注又是个体力活，况且在这个场景下，标注人员的标准都不一致，而且这些少量的样本还分布在这么多的指标里，我们如果要从这么少的样本里学习出一些特征，也将是非常困难的。

那怎么办？

面对这两个近乎无解的问题，我们没有什么特别好的方法，但希望能找到一个子集，在这个

子集阈内这两个问题是有确定边界的，然后寻求一个最优解，再慢慢拓展到整个数据集合。

首先是对算法模型的评估标准，我们需要知道哪些异常，或者说故障，是大家一致认可没有歧义的。

我们有个 NOC 团队负责 7*24 小时接受来自各个渠道的告警和报障，包括监控系统、用户报障、内部人员发现的问题，如果经确认是一个影响了订单的生产故障，则会记录下来，这就给第一个问题提供了一个相对最为接近可衡量状态的条件。

首先一个故障只要影响了订单，那么对于订单这个指标来说，这是无可争议的异常，其次，所有问题都汇集到一个处理中心，我们可以大致认为，这里所记录的问题，就代表了系统存在的所有可见异常。

所以，我们把目光锁定在所有订单类型的监控项上，一是因为只有这类监控的异常才是会被广泛认可，二是这类问题只要发现就不会被错过，一定会有人确认是不是一个真的异常。这就为我们的算法迭代和优化提供了一个明确的检验标准和可靠标注。

四、算法选择和设计目标

我们以订单这类指标为入手点来调试算法，接下来就是算法的选择。

不论采用哪种算法来检测，一个显而易见的好处就是可以减少规则的维护成本，我们不再需去给每一个指标设定合适的告警规则。

目前业界采用比较多的方式是引入统计分析的各种方法，框定一个滑动的样本集，对这个样本集进行一些数据处理和转化，经过归一化，去周期，去趋势，再将最新采集到的数据点经过同样的转换，和样本集的残差序列的统计量进行比较，比如距离、方差、移动平均、分位数等，超出一定的范围就判断为异常，或是综合各种离群点计算的方法来做个投票，多数算法认为异常则报异常。

起初我们也借鉴了这种做法，却发现虽然可以不用维护告警规则了，但报警的质量并没有提升。

和规则化告警面临的问题是类似的，当我们把置信度设成一个理论上的显著值，比如 95%时，会检测到很多的异常，当我们不胜其烦，把置信度调高时，报出的异常总量是会不断减少，但与此同时，会发现相比于原来的规则化告警，遗漏没有报出来的故障也逐渐增多，误报和漏报始终都是一对矛盾在出现在天平的两端。

通常的思路是，当两个因素互斥的时候，我们需要寻找一个折中点，让两者达到均衡以获取一个相对最稳妥的结果。

但生产环境不接受稳妥，宁可报警多一些，也不能接受漏报。但另一方面，报警接收人员要开始神经衰弱了，每个都仔细排查没那么多时间，万一漏看一个刚好和故障相关，又得担责任，他们想收到真正值得排查的告警。现实逼得我们不能搞平衡，必须鱼和熊掌要兼得，这

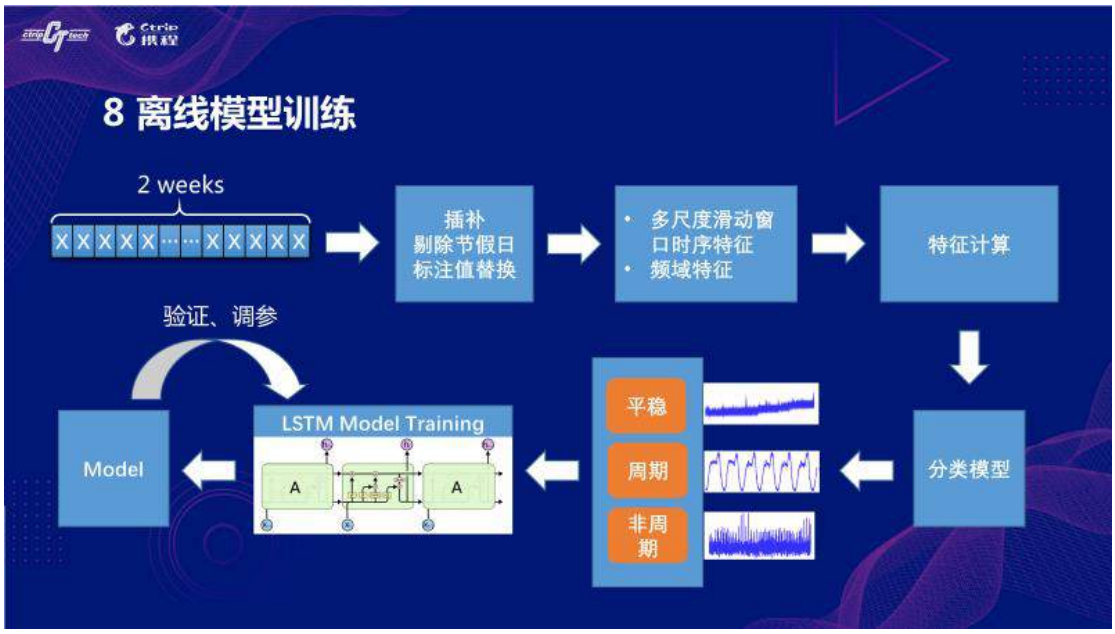
并不容易。

我们需要设计一套新的算法，降低报警总量到可以人工逐个处理的程度，同时不能以增加漏报真正的生产订单故障为代价，并且这套算法的设计还不能太复杂，影响到告警的实时性，最好还能做到算法即服务，有较强的可移植性，提供给其他的监控系统使用。

自然而然的，基于神经网络的深度学习算法成为我们进一步探索的工具。

RNN 模型比较适合处理序列变化的数据，符合我们时序特征的场景，而他的改进版 LSTM 模型，能够通过控制传输状态来选择性地记住较重要的长期数据，能在更长的序列上有良好的表现，业界也有很多成功的应用。这里重点介绍一下如何引入到我们的场景中。

五、算法的描述和检验



这是一个离线训练的过程示意图。

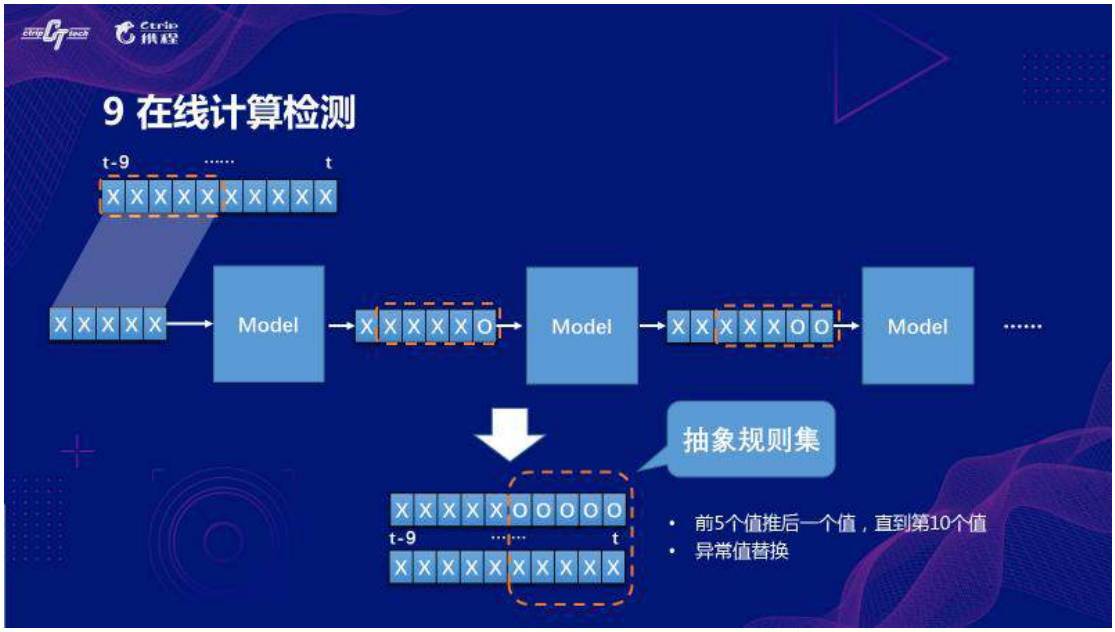
我们把历史数据拿过来，先做个清洗工作，对缺失值进行插补以及节假日数据的剔除。剔除节假日是因为训练当中如果包含了这部分数据，模型就会有偏差。当然缺失值如果超过 20%，那就干脆不训练。

然后对这个序列做特征提取，特征工程的目标是把时序序列分为三大类，周期型、平稳型、非周期。

我们使用了多尺度滑动窗口时序特征的方法，将一个滑动窗口内的数据和前 n 个周期做统计量上的对比，均值、方差、变化率等这些，这样基本上就可以把明显的周期性和平稳型数据给分离出来。

剩下的时序中，有些是波动很大的随机序列，有的则是带有趋势的周期性序列，通过时序分析法把周期性去掉，再用频域分析尝试分解成频谱。对于带有明显频谱的，则归类为周期型时序，而频谱杂乱的，则归类为非周期性。

为什么要分成三类后面会讲到。分出来之后我们定义 LSTM 需要的各个变量，然后是调用 TensorFlow 进行 LSTM 模型训练、验证和调参的过程。



在线计算检测阶段，滑动窗口取最近的 10 个数据点，用前 5 个点作为模型的输入来预测后 1 个点的值，循环输入模型直到预测出后 5 个点的值，并用这几个预测数据点和实际值进行比较。

除非遇到极端情况，否则只一个点是无法判断是否异常的，所以我们需要推测 5 个点，不是将来的 5 个点，而是过去的 5 个点，结合一些基本的规则，来对这 5 个点的实际数据做出异常判断。

这里还是需要结合最基本的规则，而什么样的数据类型采用哪些基本的规则，经过反复尝试发现是不同的，这就是为什么我们要先将数据指标分类。

我们将异常分为连续 1 个点、2 个点、3 个点这几种情况，每种情况下抽象出来一些最基本的规则，连续几个异常点，结合不同的变化幅度，临近周期内的统计量变化，以及多个点位的形态变化，设定了数套规则集，对应了高中低 3 个不同的检验敏感度，并调校验证不同敏感度下模型的表现。

我们以两个指标作为算法迭代和优化的依据:

1、和现有的规则化系统相比，同样的监控指标下，算法产生的告警量是否少于当前规则系统所产生的告警量;

2、以所有汇集到 NOC 的生产故障为全集，通过算法检出的数量是否高于当前规则系统检出的数量。

最终得到这样一个结果： 相比于规则化告警系统，算法的告警量平均压缩了 10 倍左右，而检出的故障数量反而还略高。

虽然故障检出率提升了，但还是存在漏报的情况，我们分析下这些漏报的故障，主要表现为以下几个方面:

- 指标看山去正常波动，肉眼及系统均无法检出（用户人工报障）
- 业务订单量较小导致漏报—— 绝对值越小，少量波动就会造成巨大的影响
- 历史数据波动比较剧烈，异常下跌幅度小—— 数据随机性太强，不容易准确分辨
- 趋势变化明显—— 缩短模型训练周期捕捉周期性

六、实时性工程

尽管如此，这已经是一个让各方都能相对接受的结果，而需要解决的最后一个问题就是实时性。

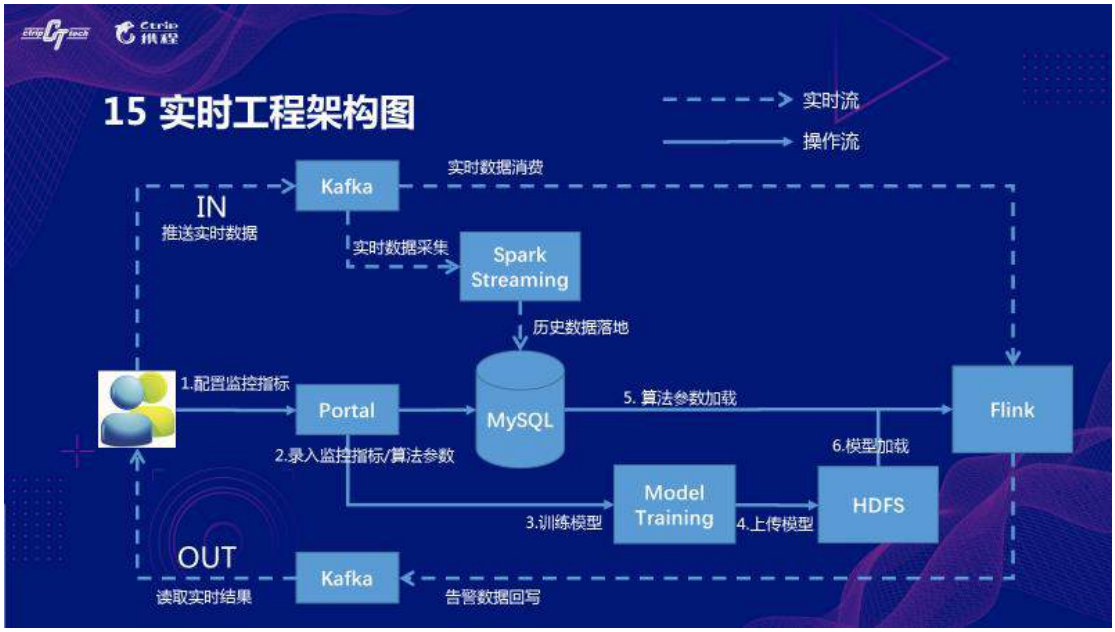
我们原本的算法都是 python 的实现，在实时处理上 python 没有什么优势，我们把采集到的数据落地，清洗插补，过模型计算，再触发报警，这些没有办法流式处理，只能通过 cron 的方式一步步调用。

而大家知道 cron 的最小时间间隔是 1 分钟，这就导致了从收到异常数据点到发出报警信息，中间需要经历 3~4 分钟的时间间隔，而一个规则告警系统，数据落地之后只需要对比规则，只需要 1~2 分钟时间就能将告警发出来，对于重要的业务指标来说，这两三分钟的报警延迟就意味着真金白银的损失，也是不可接受的。

实时化工程的方案选型，我们考虑了 storm/spark streaming/Flink 这几种，最终选择了 Flink，原因在于它能满足我们的要求，滑动窗口灵活，数据可以基于自身的时间戳来统计，不会因为数据延迟而落到下一个时间窗口来统计，这会给我们减少很多数据处理的麻烦事儿。加上 Flink 本身是为实时计算设计的，容错性比较好，我们不用考虑很多数据达到的异常问题。同时也预留了支持秒级数据采集粒度的能力。

但是有一个问题就是检测结果的数据校验需要实时计算，否则来回调用 python 会增加时间消耗，所以就将算法以 Java 代码重新实现。

我们的系统设计是这样的：



用户将每个指标的实时数据按照一定格式推送到 Kafka 队列中，并且通过 Portal 确定哪些指标是需要做异常检测的，如果指标有历史数据的话，提供 2 周的历史数据用于训练模型，或者可以不提供，等待两周的数据积累。

我们拿到这些数据之后，对所有满足训练条件的指标（有足够的历史数据）进行离线训练，生成模型之后放在 HDFS 中，Flink 加载新生成的模型，每个流过的指标如果有匹配的模型，则流入模型计算，否则丢掉，最后将计算结果回吐到指定的 Kafka 队列，供用户方消费。所有模型每两周重新训练一次，若发现用户上传新的指标，则触发训练，Flink 每 5 分钟检查一次最新的模型并加载替换老模型。

这样一来，实时数据不需要落地就能进入模型马上计算得出结果，相比于规则告警系统先落地再计算的方式平均提升了 40s 左右，更接近实时。而这种方式也将算法和工程实现抽象出来，对外以队列的方式提供输入输出，任何一套监控系统只要按照约定的格式传入时序数值，就能使用这套方案来进行实时检测。

七、算法的局限

这就是我们目前为止的尝试，大家可以发现，这种方式还是存在不少局限性的，比如每一个指标训练一个模型，每个模型十几兆，说实话不小，每次训练需要 10 分钟，所有模型全量训练一次，对服务器的压力不小。

10000 个指标如果 CPU 资源不够的话，光训练就要好几天，运行时也需要 100G+内存来加载，所需的计算资源随着需要检测的指标数量呈线性增长关系。

其次对绝对值较小的指标没有好的解决方案，因为这类指标稍有变化，震荡幅度就特别大，很容易误报。还有就是非周期性随机序列，这种指标无法学习出什么模式，也就无法判断了。

八、尾声

我们现在仍然在不断尝试，希望能提取出一个通用模型，以在面对更大监控项的时候能节省算力。也非常希望能和这方面感兴趣的朋友们多多交流学习，路漫漫其修远，我们一直都在探索的路上，心存敬畏。

行业智能客服构建探索

[作者简介]戴祥鹰，就职于携程数据智能部。此前先后供职于腾讯、百度，主要从事搜索、推荐、知识图谱、自动问答等相关工作。硕士毕业于哈尔滨工业大学。本文为作者加入携程前所做项目工作的经验总结。

引文

近年来科技产业蓬勃发展。一方面，随着互联网的普及和发展，用户在使用互联网产品过程中产生了海量的数据；另一方面，硬件设备和算法也取得重大突破。

在数据积累、算法、算力都取得巨大进步的前提下，人工智能爆发；伴随着 AlphaGo 战胜人类冠军，这个概念也开始进入了普罗大众的视野。

谈到人工智能，我们首先要了解一个重要概念：图灵测试。

下面引用其在百度百科中的解释：图灵测试是指测试者与被测试者（一个人和一台机器）隔开的情况下，通过一些装置（如键盘）向被测试者随意提问。进行多次测试后，如果有超过30%的测试者不能确定出被测试者是人还是机器，那么这台机器就通过了测试，并被认为具有人类智能。

图灵测试在上世纪 50 年代提出，从图灵测试的解释中可以看到，人机对话系统是衡量人工智能的重要场景，也是随后人工智能研究的重点方向。

在本轮的人工智能热潮中，人机对话系统依然是重点方向之一，并且以智能客服或智能助手的方式落地，多数用以解决企业在线服务中人工服务成本高，响应速度受限和服务时间受限等业务问题。

本文重点聊一聊在医疗行业智能助手探索中遇到的问题，以及为此尝试的方法。把客服类项目中需要的对数据构建、用户问题分析及理解的思考过程分享给各位读者，希望对同类项目的思考有所帮助。

一、问题背景

我们面对的是一个在线医疗服务场景：患者在线上通过网站或者 app 提出问题，医生在线做出回答，服务的过程会产生多轮的问答交互。

在这个场景中，业务上有两个突出的问题；第一，在线医生资源不足；第二，医生响应回复不及时。这两个问题影响用户的产品体验，平台信任度，进而影响用户留存，用户转化等业务指标。

在这个背景下，我们提出利用 AI 技术，构建一个可对话的医疗智能助手，用来缓解以上业

务问题。

最终，我们花费四个月的时间开发和迭代了一个智能助手，它可以在医生没有响应时给予用户及时的反馈，并通过与用户对话来收集用户的信息，还会自动计算用户的高概率疾病，用于医生参考。（注：系统是由具有医学背景的客服人员使用，不会出现机器人直接回答用户问题的情况）

由于行业的特殊性，系统对技术指标的要求是非常严格的，即，在高精确度的前提下，尽最大可能提升召回率。

下面我们就来看看智能助手的整体思路。

二、解决思路

整体方案分为两部分：
1）行业物料构建；
2）智能助手搭建；

2.1、行业物料构建

2.1.1 数据获取

理想的情况是拿到标准的电子病历；一方面，从<用户提问,确诊疾病>数据对中学习疾病分类和预测模型，用于对用户的病情自述做科室和疾病预测；另一方面，从问诊记录中统计疾病与症状(含体征)的关联关系，并计算转移概率，从而可以在问诊过程中动态计算下一步需要问询的状态。

实际情况是，电子病历是医院、医疗机构的机密数据，我们无从获取，因此需要寻找此类数据的替代品。

从目标需求出发，我们最终锁定了两类公开的替代数据源。一类是在线医疗网站上的多轮问答数据，从这些数据中可以标注出问题与最终疾病的 pair 对。另一类是医学书籍，从中我们可以抽取整理出疾病-症状的关联关系。

互联网上的医疗对话数据示例

2.1.2 数据处理

2.1.2.1 数据结构化

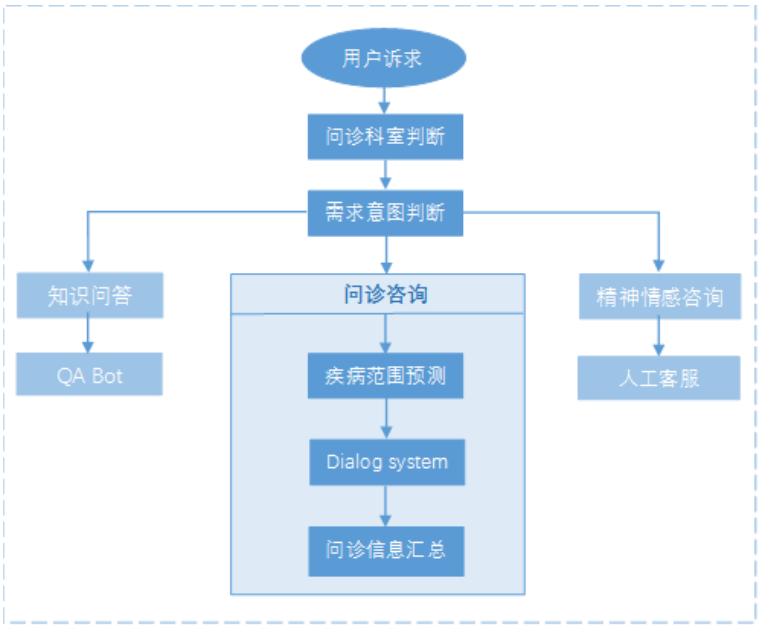
无论是对话数据还是书籍数据，都有多个来源和版本。我们首先将其转换为统一的数据格式，再从中抽取出关键字段信息，最后辅助专家标注审核，给数据附上标签信息。例如，对话数据中关于结论疾病的部分，需要医学专家审核确认，缺少的给予补充。

2.1.2.2 数据归一化

数据归一化重点针对核心医学概念，疾病及症状。由于语言描述的丰富性，一个概念通常会有多种表达方式，例如：一个疾病叫小儿腹泻病，又叫婴儿腹泻病，小儿消化功能紊乱等。我们参考医学系统命名法 SNOMED CT[1]，将其中一个选为标准名，其它作为别名，并建立映射关系。

2.2、智能助手搭建

2.2.1 整体框架



智能助手相关的服务逻辑如上，出于业务敏感性考虑，隐去了部分模块和细节，整体流程简要概述如下。

在经过科室判断和用户意图识别后，被甄别为真正有多轮问诊诉求的用户被引至问诊服务，知识满足类需求由自动问答服务来满足，而精神/感情咨询类直接由人工服务。

问诊服务核心的功能是：通过与用户进行多轮问答，询问和收集用户的信息，并预测用户最可能的疾病范围。在与用户对话过程中，问诊助手提出的问题要符合一定的条件：1）符合客观逻辑，如：不应该向男性患者询问妇科问题，不应该把仅适合儿童的问题提给成年人。2）使得对话过程尽快收敛，即每轮的提问应该在当前状态下最有利于疾病范围确定；或者最有利于确定 bot 无法满足。

下面，我们就问诊模块中几个关键任务点展开，阐述面对的问题和技术方案。

2.2.2 关键任务

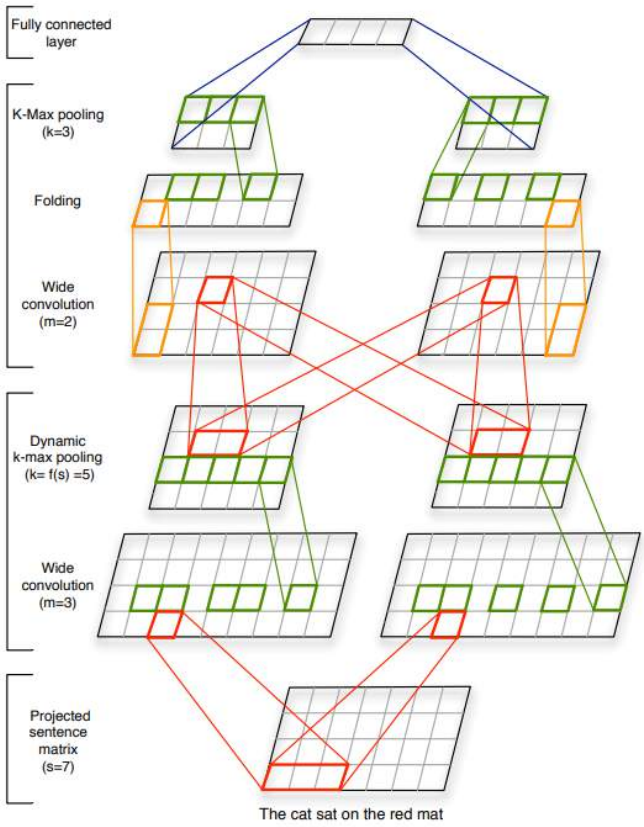
本节介绍框架下的几个关键任务。

2.2.2.1 疾病范围预测

疾病有几万个之多，如果让问诊模块直接判断所有的用户问题，每次对话的搜索空间太大，而对话轮次必然是有限的，现实中 Bot 不可能跟用户询问太多轮次；因此，必须减小会话过程的搜索空间。

我们采用逐层分解的办法；第一层；科室划分；第二层，需求意图判断；第三层，静态疾病范围预测(发生在用户提问的第一轮)；第四层，动态疾病范围预测(发生在除第一轮后的每一轮回话中)；通过逐层划分，保证问诊模块每次会话时的搜索空间是可控的。第三层和第四层区别在于处理的输入特征不同，采用的模型是一致的。

下面就介绍用于疾病范围预测的模型，我们将其定义为一个多分类问题，采用 Dynamic Convolutional Neural Network(DCNN)[2]模型来实现。



上图描述了 DCNN 算法的运行过程；其中，以长度为 7 的句子为例，embedding size 为 4；网络有 2 层卷积，卷积宽度分别为 3 和 2；卷积后的 k-max pooling 中 k 的取值分别为 5 和 3。

DCNN 与一般 CNN 的区别在于，max-pooling 的维度取值是动态计算的，有利于特征提取；另一个区别是多了一个 feature folding 层，用于特征叠加。正因为该模型的特点决定了其可以更好地提取特征，符合业务场景中存在多个特征片段的特点，我们才选取其作为预测模型；从项目效果表现上看，DCCN 也超出了同层数 CNN 模型、及 FastText 模型的表现。

利用 DCNN 计算用户输入在科室下目标疾病范围上的概率分布，可以容易的得到 Top-k 个目标疾病，并通过医疗 Knowledge Graph 中症状-疾病间的关联关系排除部分非目标疾病，进一步缩小搜索范围。

2.2.2.2 对话过程中的信息抽取

对话过程中，智能助手需要不停的从用户的反馈中获取关键信息，例如：患者性别，患者年龄，过往病史等基本信息，以及最重要的症状表现；表现包括出现和未出现，在症状上来说就是有出现该症状或者没有出现该症状。

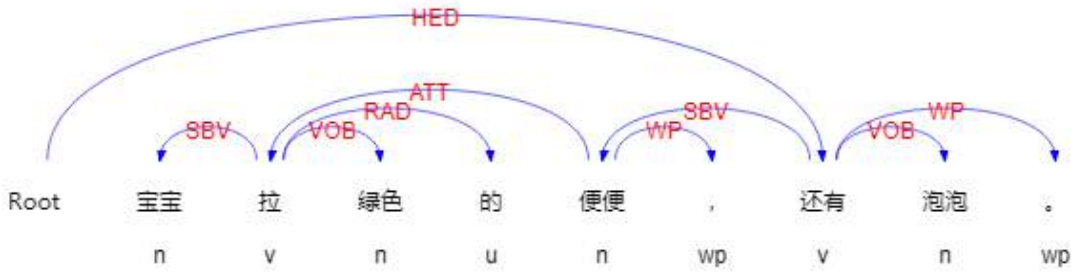
逐步积累的用户信息用于决策后续的搜索空间，因此，对话过程中的信息抽取成为另一个关键任务。信息抽取是一个非常有挑战的任务，鉴于业务对抽取结果的高要求，我们采取了 2 种方法组合的形式来解决片段抽取问题，采用分类模型来解决正负语义判定问题。

下面就介绍解决信息抽取的主要方法。

1）语义解析

通过对数据的分析我们发现，部分症状描述是由一定规律的，例如符合动宾关系，例如发烧，打喷嚏，拉肚子等。因此，我们通过句法分析获得句子结构，通过定义句法模板提取后续片段。

例如，对于用户的问题：“宝宝拉绿色的便便，还有泡泡。”，句法结构如下：



通过获取句法结构，可以抽取候选症状；例如，我们提取以 VOB 为核心的内容块，可以得到：拉绿色的便便、便便还有泡泡这样的症状描述；然后再对症状片段进行语义归一。

2）Bi-lstm+CRF

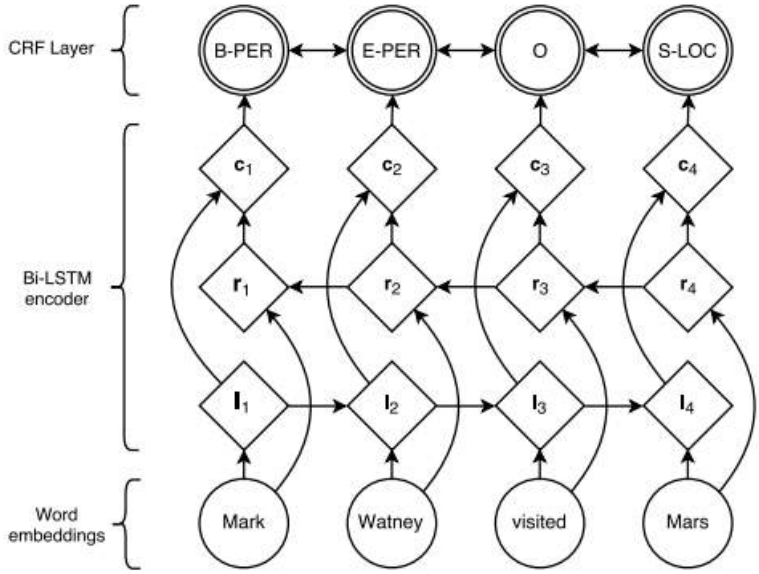
句法解析能解决一部分抽取问题，但会话中普遍存在口语化表达，导致基于句法解析的方式召回不全，因此我们必须寻找新的办法。

我们将症状抽取定义为一个标注任务，即从句子中识别什么位置是一个症状的开始，什么位置是症状的结束。下面是一个句子的标记，O 代表其他，B 代表症状开始，I 代表症状内部。

宝宝嗓子有痰，腹泻并伴有拉水的症状。……
O O B I I I O B I O O O B I O O OO……

在这个任务中，我们引入了 Bi-lstm+CRF 来解决序列标注任务，该模型将考虑双向的信息作为输入，使用 LSTM 提取序列特征，而 CRF 有效利用句子层面的标记信息；可以获取整体更好的标签序列。

下图是模型的示意图，引用于模型的经典论文[3]。



2.2.2.3 问题生成

问题生成指，在对话系统经过计算确定下一个要提问的问题点后，系统生成自然语言问题，用于向用户提问。问题生成涉及两个点，一个是选择问题生成点（这也是对话管理的一部分），另一个问题是话术构建及拼接。下面简单介绍一下提问点计算和模板选择时的一些考量。

1）提问点计算

a、深度：症状-> 症状子属性
b、广度：该疾病下并列的其它症状

因为部分症状存在子属性，例如症状的轻重缓急，症状的持续时间等维度。某一症状被选取为下一个提问点时，需要考虑两个维度上的选择，选择的依据是根据历史数据计算那个维度更利于对话收敛。

例如，疾病“咳嗽变异性哮喘”的主要症状包含咳嗽，而且是夜间或凌晨咳嗽特别厉害，其它时间几乎没有咳嗽发生，那么当会话获取到该疾病的几个关键症状且包含咳嗽时，只有往前判断一步判断咳嗽发生时间，即可以大概率判断用户是否感染该疾病。

2）生成内容

从对话数据中总结话术模板，基于模板生成问题。在模板选择时，同类问题尽量随机选择候选模板，避免用户认为是跟机器人在交互。

三、技术延伸

在该技术系统实现的基础上，以医学对话系统构建为目标，在基础系统上引入强化学习技术，并在第三方标注的独立数据集上进行了实验和验证，效果与传统的方法相比获得明显提升，

我们的成果在自然语言处理会议 ACL 2018 上发表，具体参见相关的论文[4]。

引用：

[1] https://www.nlm.nih.gov/healthit/snomedct/
[2] A Convolutional Neural Network for ModellingSentences
http://www.aclweb.org/anthology/P14-1062
[3] Neural Architectures for Named EntityRecognition；
https://www.aclweb.org/anthology/N16-1030
[4] Task-oriented Dialogue System forAutomatic Diagnosis
http://www.aclweb.org/anthology/P18-2033

架构篇

携程软件 SBC 实践

[作者简介]韩海龙，携程通信技术中心工程师，负责 VoIP，软交换相关领域技术研究与开发，及携程呼叫中心语音中继接入工作。

一、SBC 简介

随着互联网及 RTC 通信技术的不断发展，使得 VoIP 技术　近几年又火了起来。VoIP 就是 Voice Over Internet Protocol，简单来说就是只要你有质量不错的网络条件，就可以和外界进行语音通信了。只不过传统的语音通信都是通过模拟线路来进行信号传输的，而 VoIP 则是通过因特网借助 IP 包来传输数字语音信号。

在 VoIP 网络架构中，不同于传统的语音交换机、网关等语音设备，SBC 在 VoIP 通信中应用广泛，作用十分重要。SBC 的全称是 Session Border Controller。简单来说，SBC 是部署在网络边界，用来控制 SIP 会话的设备或软件。Session 为会话，Border 为通信网络边界，Controller 为控制器。

目前在市面上，商用的 SBC 厂家非常的多，大多是专用的硬件物理设备；由于市场的需求，也有一些厂家推出了软件的 SBC，但是一般语音编解码的板卡还是用 DSP 来实现的。

总的来说，SBC 没有太确切的定义，但就 RFC 的一些描述和个人的理解，SBC 应该就是基于 SIP 的 B2BUA（背靠背代理），能够解析 SIP 协议，并对 SIP 协议进行各种操作，比如添加 SIP Header，修改 SDP 等等。SBC 一般部署在语音网络边界，用于控制 SIP 信令，通常也包含了语音流的建立，控制与释放，因为部署在边界，就设计到两边 SIP 业务参数的不同，所以适配的功能也是必不可少的。

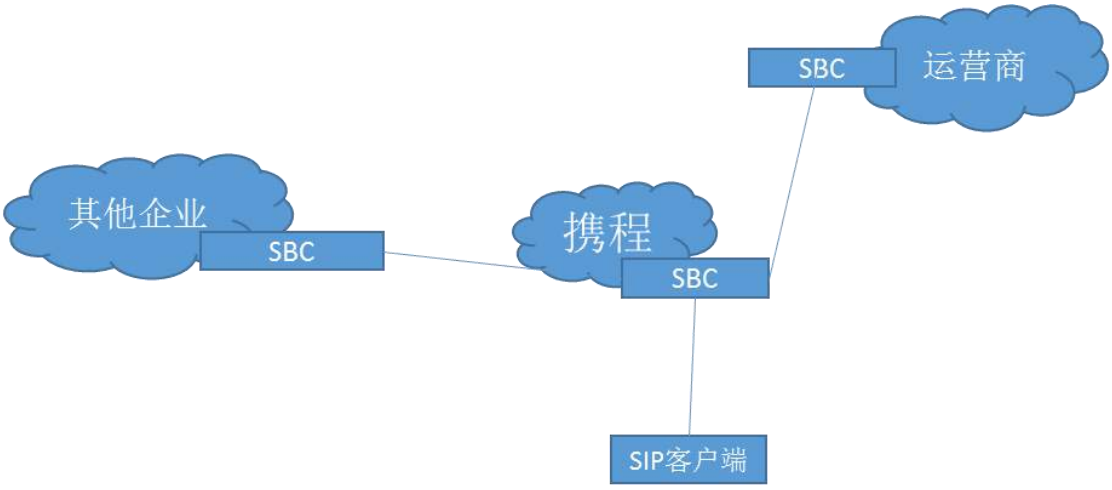
在 VoIP 网络安全方面，SBC 也起到语音会话层面的安全，QoS，准入控制等作用。更为简单的说，SBC 就像是 VoIP 的防火墙，提供了 IP 语音网络的接入服务。

其实简单来讲，SBC 的核心功能可以概括为：

1） 协议转换；
2） codec 编码转换；
3） 信令及媒体的 NAT；
4） 内部通信网络拓扑隐藏；
5） 权限及安全控制

二、SBC 应用场景

就使用场景来讲，个人认为大概分为 3 个场景：



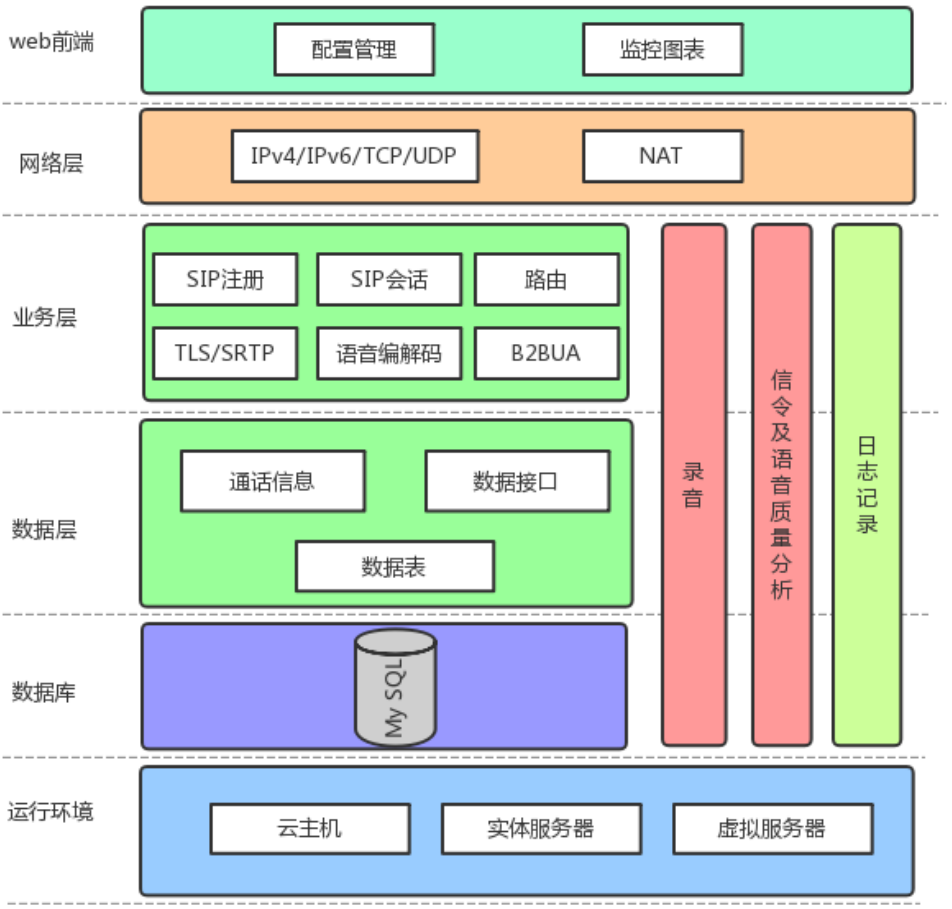
1） 企业之间的 SIP 组网，比如公司之间，或者总公司和分公司之间可以通过专线或者 Internet 进行 IP 语音系统对接；
2） SIP 客户端接入，比如软件的 SIP client 通过公网，由 SBC 充当代理接入到 IP 语音网络中；
3） 运营商 IMS 对接，可以与 SIP trunk 开放的运营商进行语音中继接入的实现。

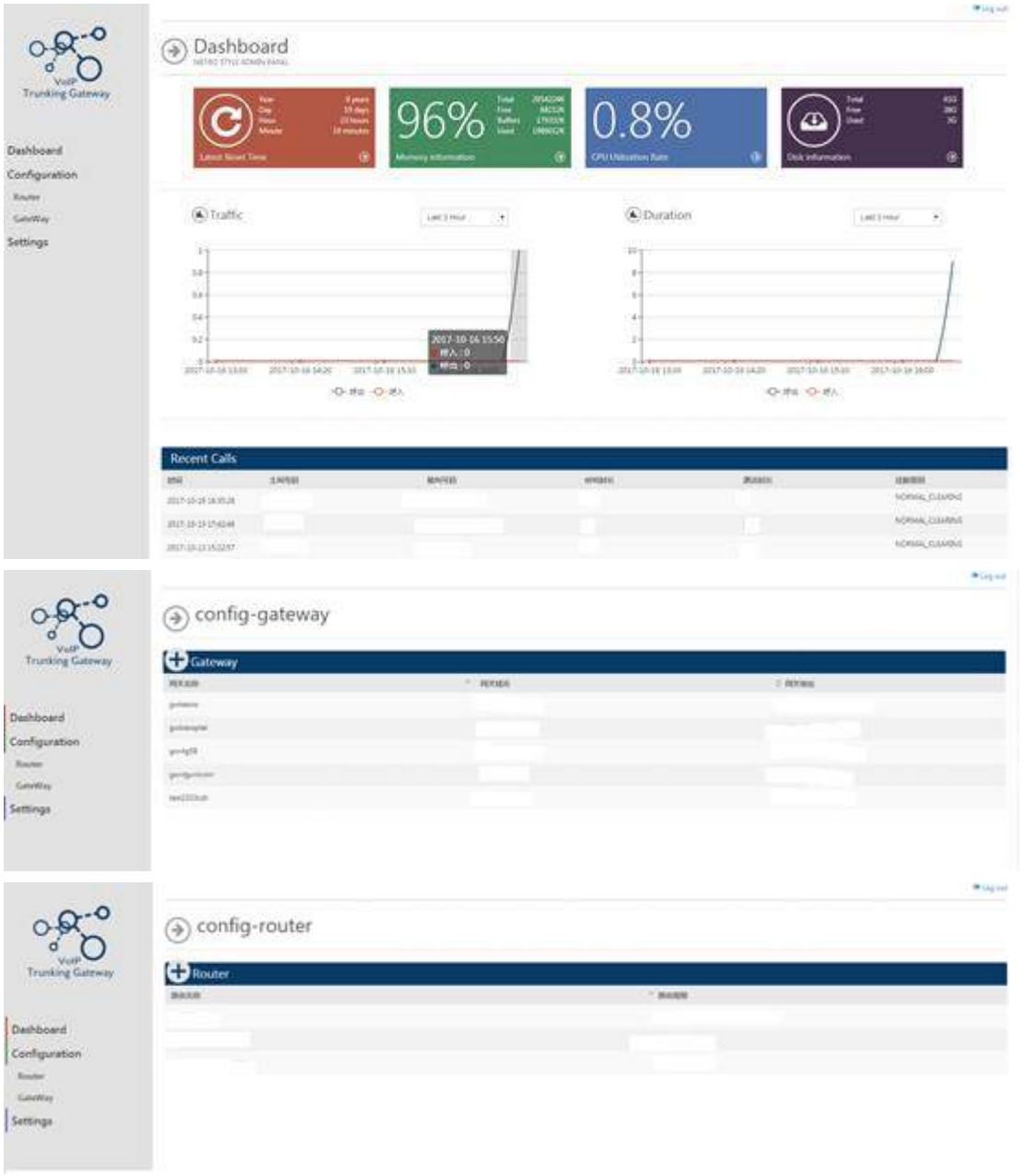
目前在 IP 通信电话系统中，无论是中继线路，移动办公，企业组网等都进行了大量的 VoIP 实践。在实践过程中，需要 SBC 设备的接入；由于是互联网公司，那通信应用也要朝着互联网发展方向，我们决定选择开源+软交换的方法来满足自身对 SBC 的需求，同时进行了向成熟产品方向的改造。

去朝着这个方向走，其实也是通过了解，认为 SBC 虚拟化，软件化是可行的。Linux OS 的架构以及 CPU 的不断强劲，虚拟机包括 docker 等技术的不断成熟，都使的软件的 SBC 可以有不错的性能。

三、软件 SBC 实践经验

首先介绍下我们软件 SBC 的整体架构，下图从类 ISO 的分层模型来展示我们 SBC 功能模块，以及管理界面：





下面根据不同场景，来介绍下我们的一些实践经验及踩过的坑。

3.1 移动软电话 VAG（VoIP accessing gateway）

携程有一个服务于全公司的办公 APP，有需求将 VoIP 软电话的功能也嵌入到 APP 里，方便公司同事可以在 wifi 或者 4G 网络环境下联系同事或者进行电话会议。在此场景下，就需要实现移动 APP 端 client 通过 SBC 接入到携程内部电话网络中，并打通语音网络，实现 APP 拨打内部办公电话和拨打 PSTN 电话的功能。

通过技术选型，我们采用了 OpenSIPS+RTPProxy 组合的方式来实现 APP 端软电话的接入，我们称之为 VAG。OpenSIPS 是一个已经非常成熟的开源 SIP 服务器，它不仅仅可以当作 SIP 代理，同时它包含了一些应用层的功能，比如我们上文提到的 SIP 背靠背代理功能。通过

OpenSIPS，我们可以轻松的实现 SBC 需要的 SIP 协议转换，NAT 功能，拓扑隐藏等等。

VAG 大致的架构如下：

实现过程：

1）通过 OpenSIPS 实现了 SIP client 注册消息的转发，将 client 的注册消息转发至后端办公电话系统上，实现 client 在服务端的注册与鉴权；

2）client 发起呼叫时，invite 消息将发向 VAG，VAG 中 OpenSIPS 将 invite 消息转发到后端办公电话系统，可以高效处理 transaction 以及 dialog；

3）Invite relay 的时候 VAG 实现 SIP 消息公网与私网的 NAT，NAT 不止是 IP 包地址的转换，还包括 SIP 应用层 NAT 穿越；

4）信令建立好后，根据 SDP 中协商的媒体地址，SIP 客户端通过 VAG 与办公电话系统建立 RTP 的传输，此处也包含了 RTP 流的 NAT 穿越；

5）会话结束后，VAG 通过 relay BYE 消息，结束双发的会话。

常见问题：

1）在会话过程中需要注意 SIP 信令的 NAT 穿越问题，否则会出现 32s 自动拆线，挂不断等问题。踩过的坑就是 client 发来的 200OK 地址要修改为其公网地址等；在 openSIPs 中需要将其公网地址及对应的端口配置在配置文件中：

```
advertised_address = "x.x.x.x"
advertised_port=5060
```

2）很多情况下电话拨打都可以振铃、接通，但是没有声音；这时候就出现了 RTP NAT 的问题，根因就是 client 或者服务端双方的 RTP 流都发到了错误的地址，基本都是发到了对端的一个内网地址上，那这样是肯定没声音的。所以要注意在会话建立阶段，双方 SDP 协商中提供的其可用的 media 地址，RTP 流地址传输对了，那自然就可以正常通话了；

3）大家可能注意到 VAG 实现了三家运营商网络的接入，也是为了不通运营商的手机用户可

以使用本运营商的网络接入，提高接入速度及质量。此处的实现可以通过交换机网络接入，VAG 多网卡或者虚拟网卡来实现，需要对应做好 SIP 及 RTP NAT 处理。

3.2 内部分公司组网 VIG（VoIP interconnect gateway）

公司内部的分公司或者子公司之间需要实现语音网络的打通，提高沟通效率或者通话费用的节省；如果通过 PSTN 方式的话，成本高，也很难实现内部的统一通信。

如果企业内部各物理节点或者独立语音系统，通过网络实现内部的 SIP 组网，IP 语音网络打通，那上述的需求就完美解决了。

在实践过程中，我们总公司和分公司之间就是通过 VIG 来实现双方语音网络互通的。这里我们使用了 FreeSWITCH 来作为 VIG 技术选型。VIG 大致架构如下：

实现过程：

1）双方在自身语音网络边界部署 VIG，VIG 则和各自内部通信交换核心组建 SIP trunk；

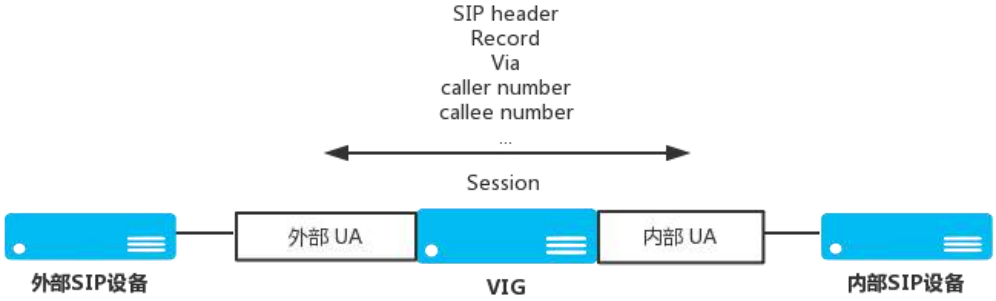
2）通信时，SIP 请求通过双方 VIG 组建的 SIP trunk 进行通信，VIG 作为中间人同时处理 SIP 消息中的随路数据及双方语音编解码的适配与转换；

3）双方在对接时，可以起到自身网络拓扑的隐藏；一方面隔离了双方原有的通信网络，安全性提高了，另一方面做到双方应用对彼此透明，一切通信都是通过 VIG 来进行。

常见问题：

1）如果双方通过网络专线打通内网网络，那其实 VIG 就不必考虑太多的 NAT 问题，因为一切通信都是通过内网地址来进行的。

2）双方通过 VIG 实现通信网络组网后，会遇到 SIP 协议适配，号段冲突等各层次的问题，那就需要 VIG 进行双方固有语音网络设备协议适配，比如一些商用硬件 PBX，IVR 系统，话机等。



3）双方本是独立的语音系统，打通后势必会碰到号段冲突的问题，此时在 VIG 上实现一些号段的映射转换等，我这边是通过添加插码来识别，然后通话删除插码来进行内部路由的。

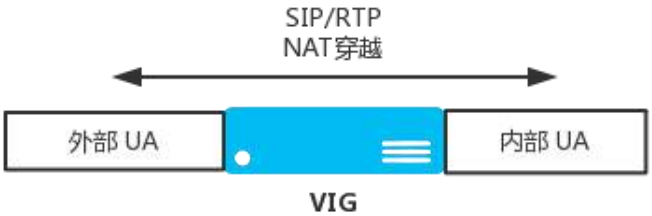
3.3 携程 SIP 语音中继接入（VoIP trunking gateway）

语音中继线路，之前都是通过传统中继线路+网关的方式来对内提供服务的。但随着运营商 SIP 中继技术的不断成熟及不断的开放；通过 SBC 实现 SIP 中继的接入是未来的发展方向。在 VTG 实践中，我们使用了 FreeSWITCH 作为 VTG 的技术基底。VTG 大致架构如下：

实现过程：

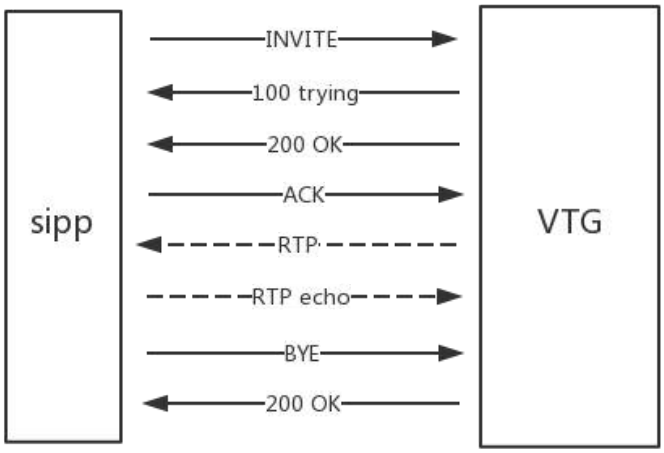
1）自身部署 VTG，运营商 SIP 中继通过专线的方式对接 VTG 服务器，此时 VTG 服务器需要两个外卡来实现对外与运营商 SBC 对接，对内与内部电话系统对接。

2）如果运营商提供的是公网 IP，那还需要通过 VTG 解决 SIP 及 RTP NAT 问题。解决的办法可以通过，建立两个 UA，一个对内，一个对外，然后在 VTG 内部将两个 UA 对接起来。



常见问题：

1）对接中继线路，VTG 需能承受大量话务并发，故需对其进行高并发的压力测试；我们使用的是 SIPp 来模拟定量的 caps 及并发呼叫，测试信令流程如下：



具体的测试数据，与自身服务器配置及网络环境有一定关系，这里也就不分享了，但是测试结果是满足我们的需求的；

2）在高可用方面，我们采用的是虚拟 IP 漂移应用主备的方式；在 keepalived 和 heartbeat 两款软件方面都有过使用经验，个人比较推荐 keepalived，使用及配置起来更为方便。加入脚本后，如果检测到主机应用宕机，可以在 1s 内将虚拟 IP 切换到备机上，备机继续提供服务。这里有个坑就是，在配置 keepalived 过程中，如果出现虚拟 IP 无法切换或者脑裂问题时，可以通过抓取日志消息对比，再看看服务器所处网络环境的通讯模式，大多就可以解决问题了。

3） 在对接测试的过程中，也出现过 DTMF 失效的情况，各种抓包分析排查下来，发现运营商的 SBC 用的是 inband 的模式，我们这边也是适配了 inband，但是还是不行，最后才发现 inband 模式只在 G711 编码的模式生效，其他有过压缩的编码方式确实会导致 DTMF 传输出现问题。

四、总结

上面就是 SBC 的几个典型的应用场景。当基本功能都具备后，就考虑向一个产品去优化。软件 SBC 不仅支持私有云，同时也支持公有云的部署；支持 SBC 系统性能与业务层的监控告警；支持数据实时落库，也提供标准的数据接口。目的就是让我们的软件 SBC 可以成为一个专业的软件 SBC 解决方案。

总结一下，以上向大家介绍了我们在开源软件SBC的实践经验，有坑，但是更多的是对VoIP、SBC 技术的深入了解，希望对大家有所帮助。

其实在未来的 5G 或者 IMS 网络中，SBC 会扮演这越来越重要的角色，希望大家可以相互学习，相互分享，一起提高。

携程图片服务架构

[作者简介] 胡健，携程框架高级研发经理，目前负责多媒体服务的构建和研发工作。

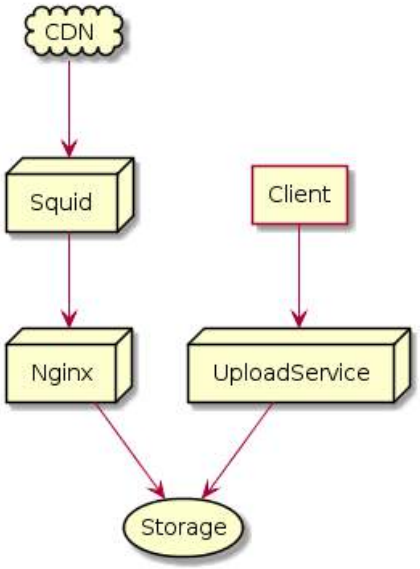
近些年携程业务突飞猛进，用户遍及世界各地。公司对用户体验也越来越重视，每一个小的功能改动、页面改版的背后，都有大量的 A/B 实验提供保障。与此同时，与用户体验息息相关的媒体文件的应用质量也被放到重要位置，如图片加载延时、成功率、清晰度等数据。

本文将分享携程图片服务架构，包括服务架构的演变过程，以及在生产上实际遇到的一些问题，避免大家重复踩坑。

一、服务架构

1.1 初始阶段

携程图片的服务架构主要经历了三次比较大的调整。早些年为了满足业务快速上线的需求，我们做了简单实现，架构如下：



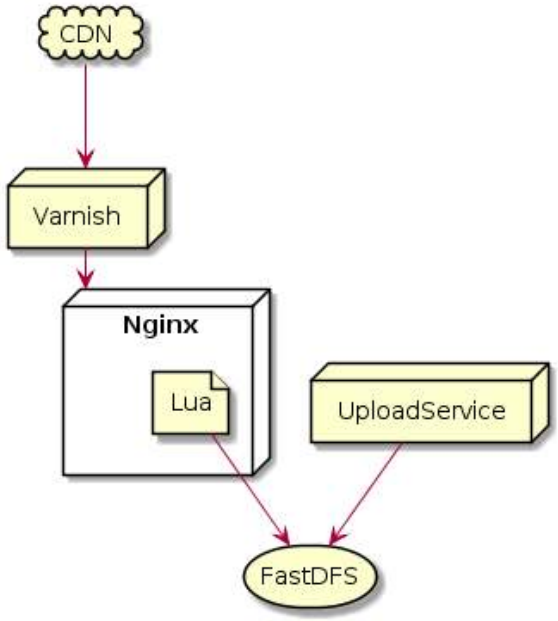
这个架构开发工作量不大，因为当时业务对图片尺寸的需求单一，也没有复杂的图片组合处理需求，因此有大量图片都被 Squid 缓存住，缓存命中率很高，取图速度非常快。

图片裁剪命令的执行，则由业务发布的时候上传处理。存储通过 NFS 让整个 Nginx 服务集群共享。直到移动端流量开始爆发的时候，这个架构有点力不从心。

首先，同一张原图需要裁剪出大量不同尺寸的小图片，占用了大量存储资源。其次，业务图片越来越多加上大量不同尺寸的小图片的出现，导致 Squid 缓存命中率变差，大量流量穿透到 NFS 上，I/O 迅速变为瓶颈。

从监控看，当时的 NFS Read I/O 一直处于高水位水平，告警更是 24 小时不断，回源流量的上升也导致 Squid 服务集群开始变得不稳定，经常需要重启。鉴于这些问题，我们做了下面架构上的调整。

1.2 发展阶段



用 Varnish 替换了 Squid，作为缓存和反向代理服务。

从实际监控情况看，同等压力下 Varnish 的表现比 Squid 更稳定，Varnish 虚拟内存 swap 机制比 Squid 自己管理的更好，因此性能上更优，并且 Varnish 配置方便，对运维友好。

当然 Squid 也有更适合的使用场景，选择 Varnish 是因为在当前场景下更符合我们的需求。

为了解决 Varnish 节点宕机会引发大量缓存数据失效，LB 上对 URL 做了一致性 Hash，这样能尽量减少缓存失效带来的其他节点数据的迁移，同时也解决了 Varnish 利用率的问题。

Nginx 内嵌 Lua 脚本用于在图片访问的时候直接对图片进行处理，而不是上传的时候处理，这样很多不同尺寸的小图不用在存储上保留，存储上少了大量 I/O，并且减少存储量的同时也会减轻运维的压力。

从访问效率看，因为图片需要实时处理，服务响应延时相比上一个版本有大幅上升，平均延时大概在 300 毫秒左右。但是这个影响实际对端的影响有限。

首先，国内 CDN 普遍质量较好，95%以上的图片资源访问都会被 CDN 挡掉，正常情况下回源流量不会太大。其次，我们 Varnish 集群命中率大概在 40~50%之间，所以整体图片实时处理压力占整体流量约 1%~2%之间，这些流量访问延时会上升 300 毫秒左右是完全能够接

受的。

存储用 FastDFS 替换了 NFS，当时 Ceph 还不像现在那么稳定，FastDFS 的特性又能够满足我们需求，并且架构简单，源码能完全掌控。事实证明，FastDFS 集群完全支撑了每天数亿次的原图读写操作，并多次在多机房 DR 演练中完成各项指标。

当时这个架构的核心是 Lua 的图片处理模块，Coroutine 的性能非常好，当有大量图片回源请求的时候，CPU 不会浪费在线程的 context switch 上，开发也很直白，在 I/O 操作的时候不需要用异步方式编码，并且 Lua 的执行在 Nginx 里足够高效。

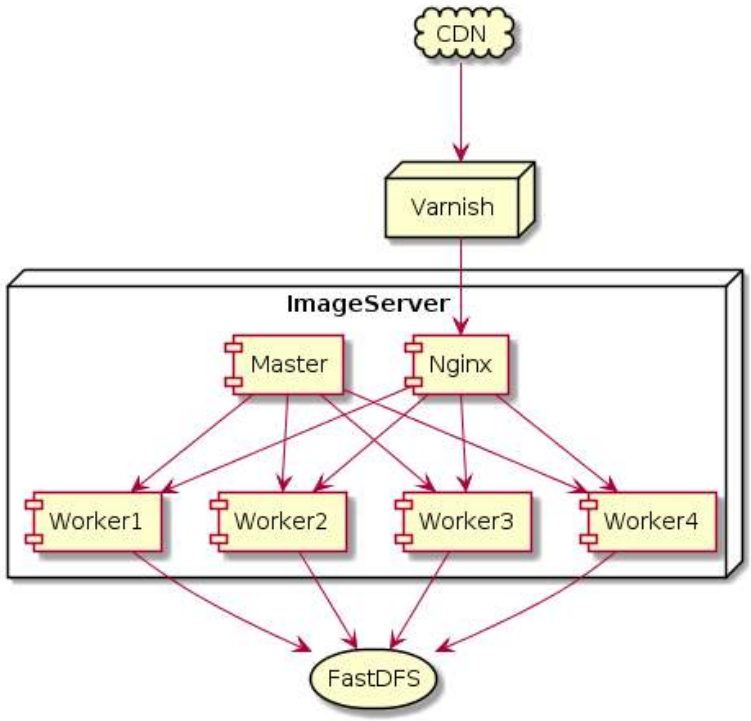
这里唯一的缺点是 Lua 扩展性相对较弱，很多模块需要自己写，比如对接我们自己的监控系统的时候就遇到难题。

随着业务的发展，用户对图片的处理要求越来越高，多重滤镜的应用，需要在 Lua 里实现很多功能，并且很多基础数据结构都要自己写或者依赖第三方，不仅开发工作量大，稳定性和正确性的验证也需要花费不少的精力。

是不是还有一种技术方案可替代，既能享受协程带来的简单，高效。又能兼顾扩展性和完善的功能包，不用重复造轮子。

1.3 现阶段

我们选择了 Golang 做为当前版本的开发语言，架构如下：



采用多进程单协程图片处理模型。图片库主要依赖的是 GraphicsMagick，和少部分

ImageMagick，通过封装 cgo 调用实现。

Golang 调用 cgo 会申明一个进入 syscall 的指令，意味着调度器会创建一个 M 去执行 goroutine。因此当有大量并发调用，并且图片处理足够慢，比如一张像素特别大的原图，就会引发大量线程同时存在，造成不必要 context switch，CPU load 看上去很高，实际效率很低。

因此我们通常会通过 Master 进程 fork 出和 CPU 相等数量的 Worker 进程做图片处理，每个进程只有一个协程来处理图片，每个进程会创建一个可配置的 buffer 用于保存原图的 blob，这样能最大化利用单协程的利用率。

采用这种架构当时主要还为了规避 GM 本身的一个问题，参考我们向作者提交的 issue: https://sourceforge.net/p/graphicsmagick/mailman/graphicsmagick-help/?viewmonth=201708.

问题描述是 setjmp 函数和 longjmp 函数在某些操作系统非线程安全，作者需要一个全局锁来保证线程安全。因此多线程调用本身是低效的。

这个问题在 java 或者.net 封装的 GM 也会存在。上一个版本的 Lua 不存在这个问题，因为 Nginx 本身会 fork 多个 Worker 进程进行图片处理，并且只可能存在一个正在运行的协程。事实上 Linux 执行这两个函数本身是线程安全的，作者可以通过 build 的时候来决定是不是需要加上线程安全的 flag。在发表本文的时候，作者已经在最新的 release 中修复了这个 bug。

这里的 Nginx 不仅仅用来做 LB，因为 Nginx 能提供很丰富的脚本，可以省去很多开发工作量，并且当有获取原图的需求，可以通过 Nginx sendfile 直接从存储取回，节省不必要的系统开销。

LB 算法并不是简单的 RR，我们会根据每个进程的 CPU 消耗，以及原图像素，buffer 消耗等维度动态算出各进程的负载量，如果 Nginx RR 到一个负载非常大的进程，可以通过返回重定向状态码让 Nginx 重新跳转，这里可能会出现几次网络跳转，但是因为是 Loopback，网络上的消耗相对图片处理的消耗可以忽略不计。

Master 进程用来管理 Worker 进程，当有 Worker 意外 Crash，则会重新拉起一个 Worker 进程，始终保持和 CPU 数量一致。Master 进程的健康安全会定期 Report 给监控系统做告警。

二、小结

当前的图片服务架构，支撑了携程每天上亿次原图处理，平均图片处理延时控制在 200 毫秒以内，图片处理失败率小于万分之一，从发布至今节点没有出现宕机现象，偶尔 Worker 进程有性能问题和 Crash 也通过日志和分析工具逐一解决。

如上所述，携程图片服务架构经历了三次改版，从一开始没有设计复杂的架构，只是为了解决碰到实际问题而重构，到后来根据遇到的问题，不断调整，也说明了没有完美的架构，只有适合的架构。

当然，要提供稳定图片服务，架构是一方面，也必须有其他技术上的支持，比如图片本身质量和尺寸的优化，盗链和版权问题，端到端的实时监控和预警机制，不良内容识别，产品图片管理和编辑功能，以及海外用户图片访问加速问题。这些问题每个都能写下不少篇幅的文章，有时间再和小伙伴分享。

目前，携程图片服务已在 github 上开源了小部分功能，开源地址：https://github.com/ctripcorp/nephele，后续会逐步完善，欢迎 PR。

揭秘携程三端通用框架中的CRNWEB

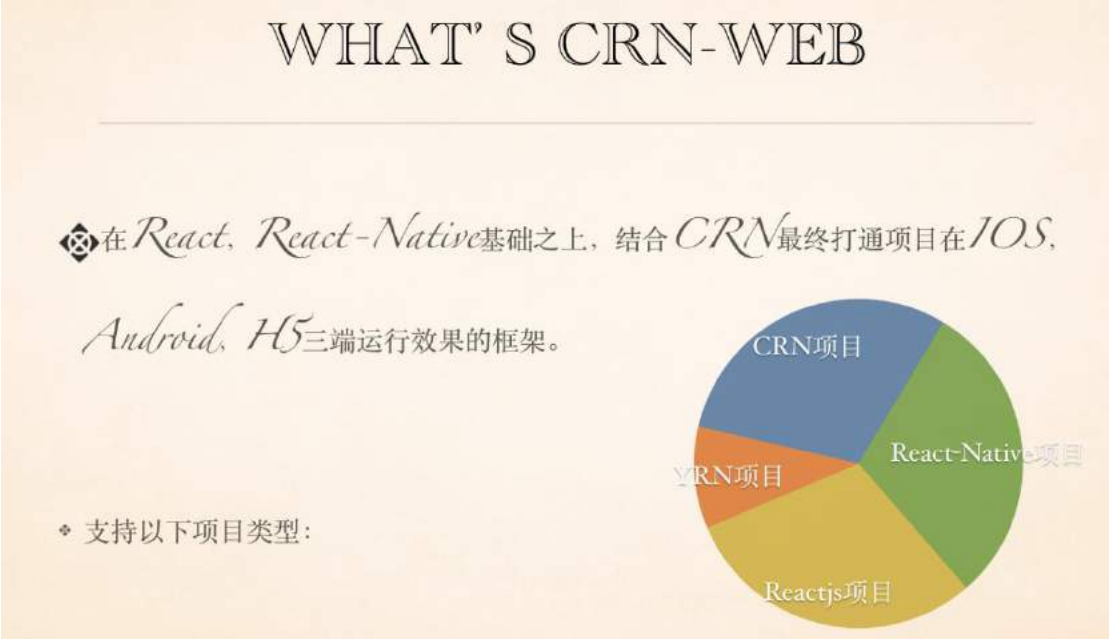
[作者简介] 郑勇，携程高级技术经理，目前主要负责 CRNWEB 框架的开发工作，以及在携程内部的使用推广和性能优化。

前言

React-Native 自从 2015 年推出，就一直火到了现在，一度在技术圈言必 RN，激发一波广泛的思潮。携程基础业务研发团队迅速跟进，在 React-Native 基础之上，开发出了 CRN 这一适合携程业务高速发展的、抹平了 iOS 和 Android 端组件开发差异的、做了大量性能提升的框架。然而无论是 CRN 还是 React-Native 本身都无法解决移动板块中的一大版图——WEB 平台。

而现实是：存在大量的业务需求需要三端的支持，单独再开发一套 H5 成本高昂，后期的维护成本也很高，需求同步难，用户体验不一致等问题都会非常明显，而携程基础业务前端框架团队一直都在致力于解决 iOS 和 Android 之后，将 BU 业务代码无缝接入 WEB 平台的技术方案，于是 CRN-WEB（简称 CW）应运而生。

一、CRNWEB 是什么?



CRN-WEB 的使命就是在 CRN 和 React-Native 的基础之上，构建一个三端打通的平台，能够实现 BU 的一套业务逻辑代码，能够根据平台情况运行在三端之上，并带来用户体验上的一致性（和 React-Native 保持一致）和优越性（使用 Virtual DOM，PWA 等技术提升性能）。

1.1 设计共性

对于 CRN-WEB 这样一个框架，我们在设计之初就可以提取一些软件设计方面共性的问题：

1）易用性，CW 框架必须简单易用，大幅度降低开发成本、运维成本和学习成本，将是这个框架的核心价值，如何做到呢？

2）一致性，和现有技术框架的集成问题，即如何将 CRN-WEB 与 CRN 和 React-Native 进行友好的集成，各自发挥各自的功能，如何保证各平台间的一致性？

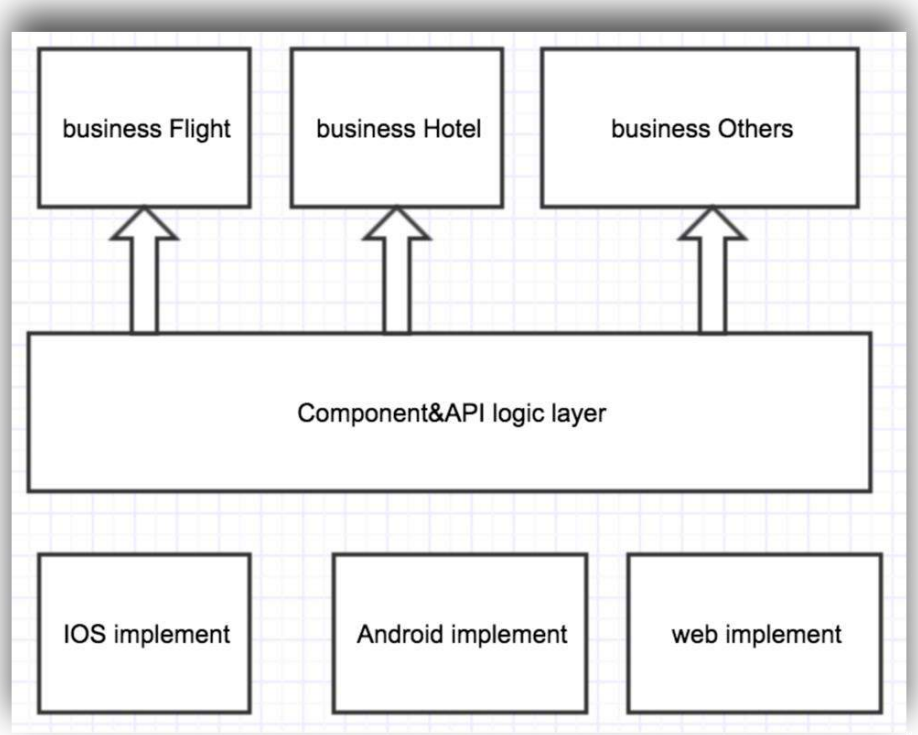
3）稳定性，React-Native 版本迭代迅速，版本间差异较大，既然三端打通，共用 BU 源码，那么 BU 的 React-Native 项目或者 CRN 项目在接入 CW 框架后，必须能够稳定运行在 WEB 平台上，如何保证项目稳定运行？

4）兼容性，WEB 平台是非多（浏览器厂商多，版本多，私有规范多，差异多...),兼容性问题一直是 WEB 项目开发头疼的事情，如何处理好兼容性问题？无疑是非常棘手的。

5）扩展性，包括 React-Native 本身都还在不断的变动，增加新功能，再加上公司级别的功能性需求，业务级别的功能需求，将令如何保持框架扩展性变得非常麻烦。

1.2 我们的设计思想

That's a big business，的确，这些问题很难处理，但是经过深入的思考，我们提出了这样的设计思想。



React-Native 为解决 iOS 和 Android 两端兼容提供了解决方案，它是如何做到的呢？当然

RN 团队经过了大量的工作和思考，最终他们提供了一套规范，即 React-Native，与其说它是一个框架不如说它是一套规范，对，我就是这么认为的。

如果 CRNWEB 的设计也基于 React-Native 的规范，把 React-Native 抽象成一个逻辑层，为不同的平台提供相同的 Component 和 API 输出和相同的 APP 主要运行流程，然后在规范之下各个平台各自实现，即 iOS Implement，Android Implement，WEB Implement，那么从设计上来看是比较完美的。

对于业务方而言如 Flight 项目，Hotel 项目等等，无需关心底层的技术实现，使用 React-Native 这一套开发技术体系基本上就足矣。

1.3 设计优势

这样设计同时还可以解决好几个问题:

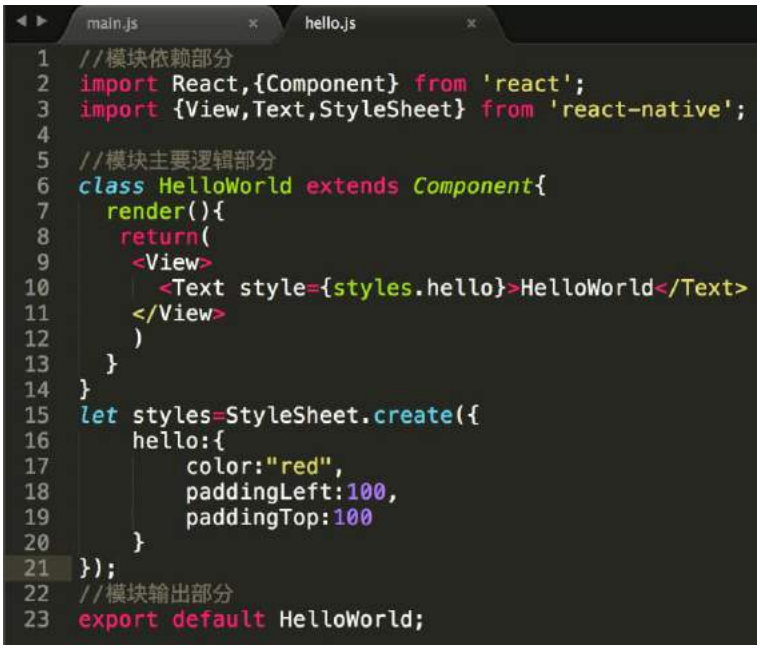
比如易用性，我们采用了 React-Native 的规范，那么我们就可以使用开发人员熟悉的技术，熟悉的规范，熟悉的知识，熟悉的流程，无需额外学习太多其它规范和技术栈。

否则 BU 的学习成本，接入成本太高，起不到降低开发成本的作用，当然为了解决易用性，还有很多其它方面的工作，比如提供一整套的开发流程，开发工具，发布工具，技术支持等等。

再比如一致性问题，和 React-Native，CRN 使用相同的规范，这样的设计保持了天然的一致性。

二、CRNWEB 是如何工作的呢？

我们依然从程序设计的传统，Hello wolrd 开始。



熟悉 React-Native 的同学可能一眼就能够看出来，这完全就是 React-Native 的原代码，你说的对，它不仅是一份 RN 的源代码，也是一份 CRN-WEB 的源代码。它虽然是一个最简单的 Hello World，但是它几乎包含了 React-Native 的 Component 和 API，以及主要的运行流程。

2.1 主题结构分成三个部分：

1）头部的依赖部分，使用 ES6 的语法 import，导入依赖的程序包 React 和 React-Native；
2）中间部分实现模块主要逻辑；
3）尾部使用 ES6 语法 export 导出模块输出；

在 CRN-WEB 中也是这样，毫无差异。

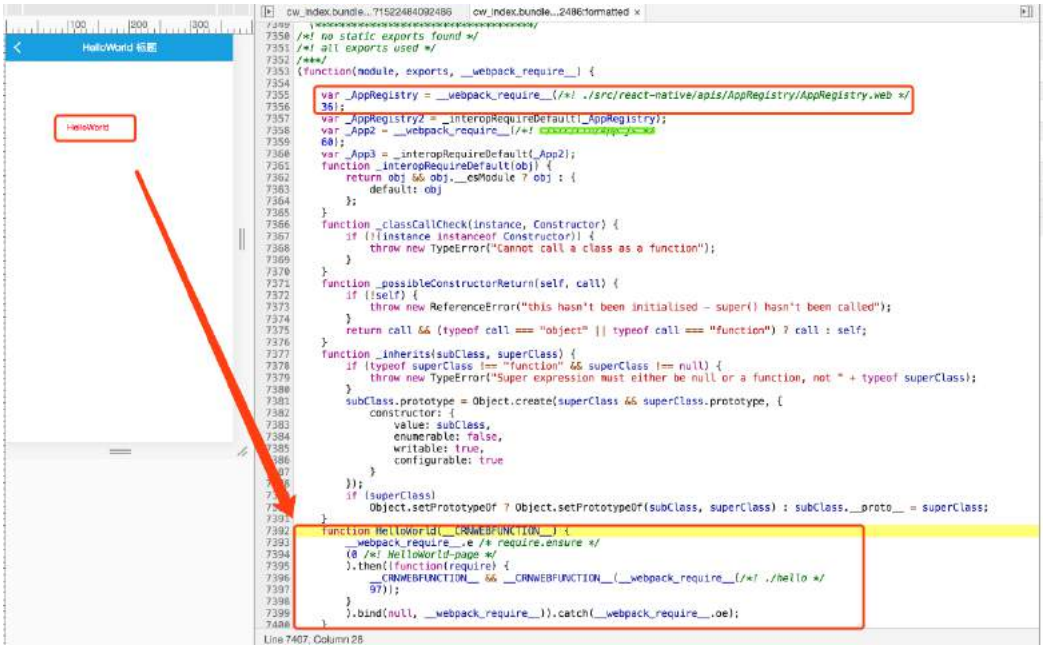
2.2 那么 CRNWEB 是如何让和 React-Native 相同的源代码运行在 Web 平台的呢?

要实现这种能力，那么它必须满足两个最重要的必要条件。第一点，我们要实现在 Web 平台上面，跟 React-Native 上面具有相同功能的 Component 和 API，比如这里的 View 和 Text，这个就是我们后面要讲到的组件系统。

第二点，我们要有一种机制使得我们的 React-Native 原代码能够在 Web 上面运行起来，调用我们 WEB 平台上的 Component 和 API，使得我们对代码拥有足够的控制能力。这个就是我们后面要讲到的打包系统。

三、运行分析

HelloWorld 代码编写完成，配置好环境，执行 CRNWEB 命令，查看编译后运行效果和运行结果。



3.1 入口组件

CRNWEB 仍然使用了 AppRegistry 作为程序入口注册组件，当然这里的 AppRegistry 已经被换成了 AppRegistry.web 这个 WEB 版本的 Implementation，AppRegistry 实现了 registerComponent 作为程序入口，承担 App 初始化工作，例如：

1）运行环境初始化，例如识别是 h5 还是 hybrid；
2）注入默认的全局性样式，例如抹平浏览器差异的样式；
3）全局性请求参数的解构和传递；
4）初始化全局性组件的容器等等；

3.2 同步组件的异步转换

HelloWorld 组件就是一个标准的 React-Native 组件，在 CRNWEB 中为了提高性能，将 HelloWorld 组件转化为异步组件 HelloWorld(__CRNWEBFUNCTION__)，从而实现页面级别的按需加载，仅在需要的页面运行时进行加载。

这在 WEB 环境下是非常重要的一项优化，这是专门针对 WEB 环境下脆弱的网络环境而作出的改进，特别是在页面众多，组件数量大，组件体量大的较大型 WEB 项目中，性能提升非常显著，这在 BU 的实践中得到了的认可。

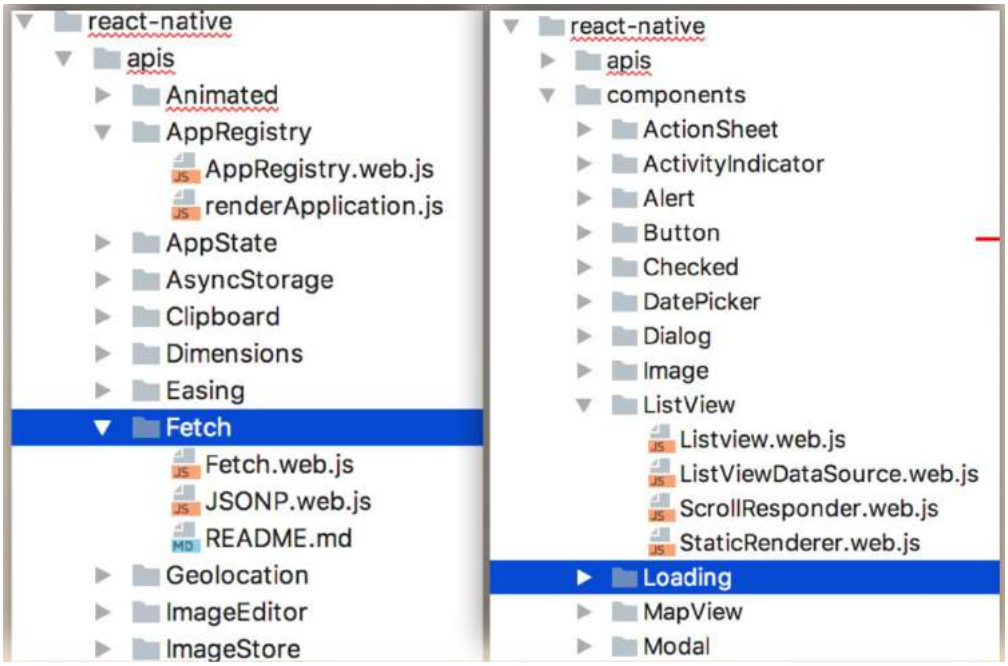
3.3 具体的业务逻辑页面的编译转化

而原来的 HelloWorld 业务逻辑被打包到了模块号为 97 的 package 中，并处理好了它的依赖，如下：

我们可以看到原来的 ES6 语法书写的代码被编译成了 ES5 语法的代码，因为 ES5 在 WEB 环境下有着更广泛和友好的支持，兼容性更好。而 HelloWold 中引入的 View，Text，StyleSheet 等等组件，也全部变成了 WEB 版本的具体实现，这里使用了一招瞒天过海。

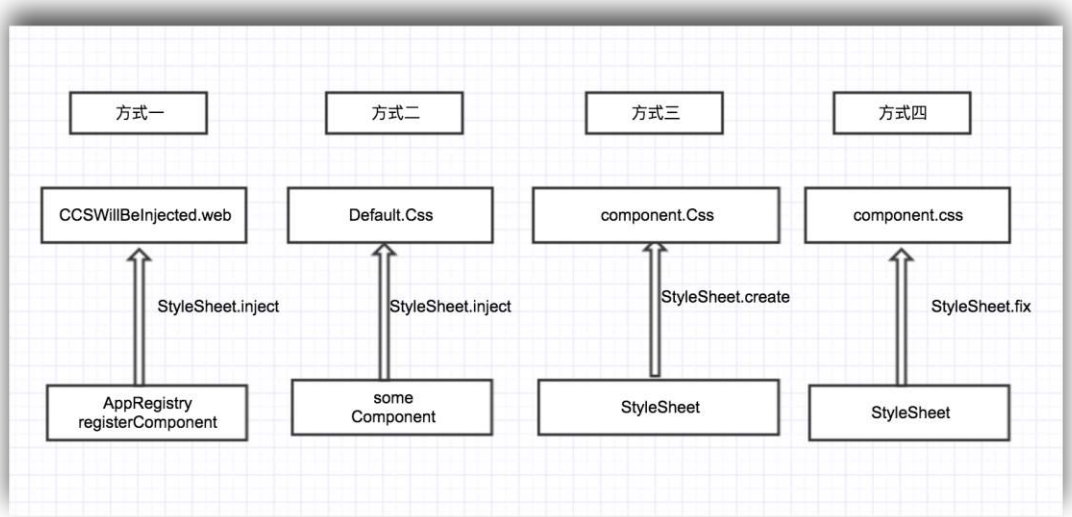
3.4 组件系统

而 View，Text 等等众多的 React-Native 原生组件对应的 WEB 版本的具体实现，就构成了 CRNWEB 的组件系统，篇幅有限不做展开。



3.5 样式处理

而 HelloWorld 里引入的 StyleSheet 就是样式处理系统中的入口文件。



CRNWEB 的样式处理系统我们主要提供四种方式：

首先是 APPRegistry，我们需要注入一些默认的全局样式，这个前面已经有所提到。后面三种其实都是对于组件样式的处理。

第二种是对组件的默认样式，可能有一些组件，历史的组件，我们也给它提供了这种能力。

第三种是一种预处理，组件样式的一个预处理，基本上都要用到 StyleSheet.Create，这个和 React-Native 保持一致。

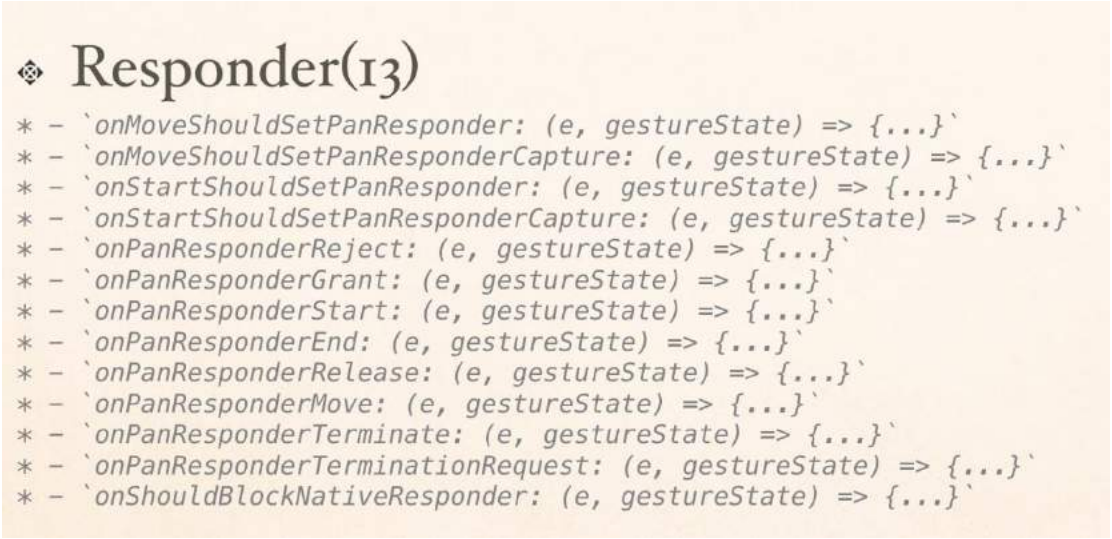
第四种我们对样式的一个实时处理系统。

样式处理系统的任务就是处理样式的问题，包括但不限于：

1）平台间样式的差异性，比如 Border，在 React-Native 下，它是分散的每一个属性值进行一个独立的编写，而在 Web 上面它的 Border 是一个混合制，所以这就是平台之间的差异，CRNWEB 框架就会开一个任务去对它进行修复。
2）浏览器间的差异，比如有的浏览器支持 FlexBox，有的不支持，而且即使是支持 FlexBox，支持的程度，版本也不一样，这些都是需要具体处理的修复任务。
3）一些共性上的问题，如单位处理，颜色处理等等。
4）一些差异性样式问题，如前缀处理，视口问题。
5）Web 不支持的样式，如 BoxShadow 的实现等。

3.6 事件处理

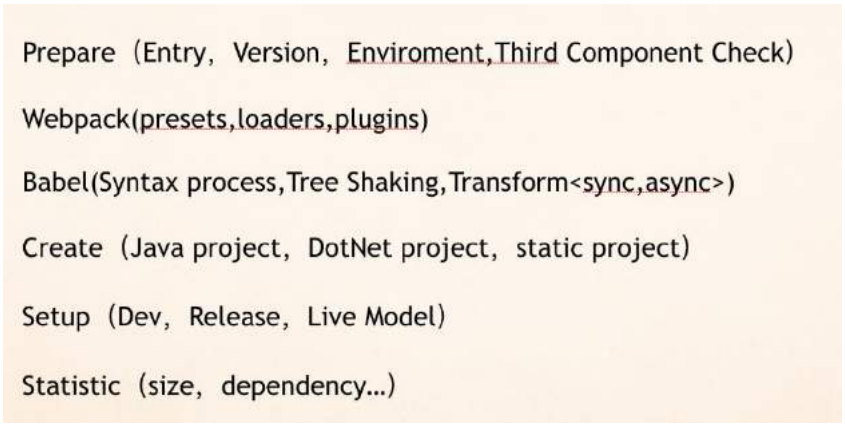
CRNWEB 中还有非常重要的一大块逻辑就是事件处理，我们专门提供了一个事件处理系统来进行处理。



我们使用了 PanResponder，它提供一个对触摸响应系统的 Responder 的可预测的包装，和 React-Native 保持一致的事件处理流程，所以在事件的处理流程和兼容性方面和 React-Native 保持了高度一致性。

四、打包系统概述

最后 CRNWEB 的一大重头戏就是打包系统。



4.1 打包任务

CRNWEB 打包系统的任务非常多，从流程上看，大概分为以下几个阶段：

1) Prepare 阶段，需要对它进行入口检查，版本检查，环境检查以及第三方的依赖检查等等，各种的预先准备条件都在在这个阶段进行处理。

2）进入到 Webpack 的打包构建流程，我们编写了很多 Webpack 的插件，对它打包进行各种处理和优化。

3）构建过程当中会去调到 Babel，利用 Babel 对原代码进行编译，对语法进行处理，对于代码的同步、异步转换这样一些比较核心的内容，都是通过编写的 Babel 插件对原代码进行处理实现的。

4）进入到 Create 阶段，因为有的 Bu 需要生成 JAVA 工程，有的需要.Net 的工程，还有的只需要一个 Static 静态工程，在这个阶段需要对它进行一个工程的一个创建。

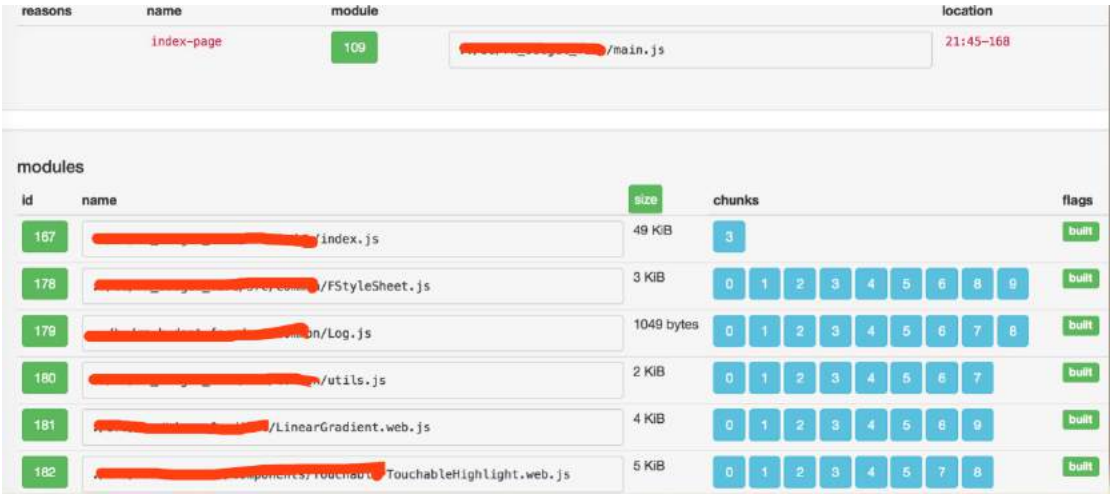
5）对这个工程进行启动，我们提供了开发版和生产版，他们的侧重有所不同。像开发版的话，它的主要诉求是打包编译的速度要快，这样才可以提高效率。而对于生产版它最核心的需求就是，要使你的 size 最小化，使你的运行效果最好。我们使用了很多优化手段对它进行了处理。

6）最后涉及到的一块，需要对它的 size，它的依赖进行各种各样的优化。我们实践下来发现，BU 代码量是非常非常多，业务也是非常非常复杂。怎么办呢？我们对业务工程进行 size 的分析，依赖分析等等各种更深入到代码层面的分析和处理，从而寻找到最佳实践的解决方案。

4.2 一些关键优化点

随着业务蓬勃发展，页面越来越多，组件越来越大，无论对于 Native 还是对于 Web 来说，这都是无法回避的挑战，精简打包 size 成为重要工作，对于 size 这一块我们做了很多优化处理，包括但不限于：

1）对于 React，进行了优化和增减，以及一些切入式的处理，只保留需要的部分。
2）使用了 tree shaking 技术，排除掉了很多死代码。
3）对组件进行高级别抽象，增加重用度。
4）减少组件层级，简单而有效的方案，既减少 size 又提升性能。
5）运用各种 cache 技术，提升用户体验。



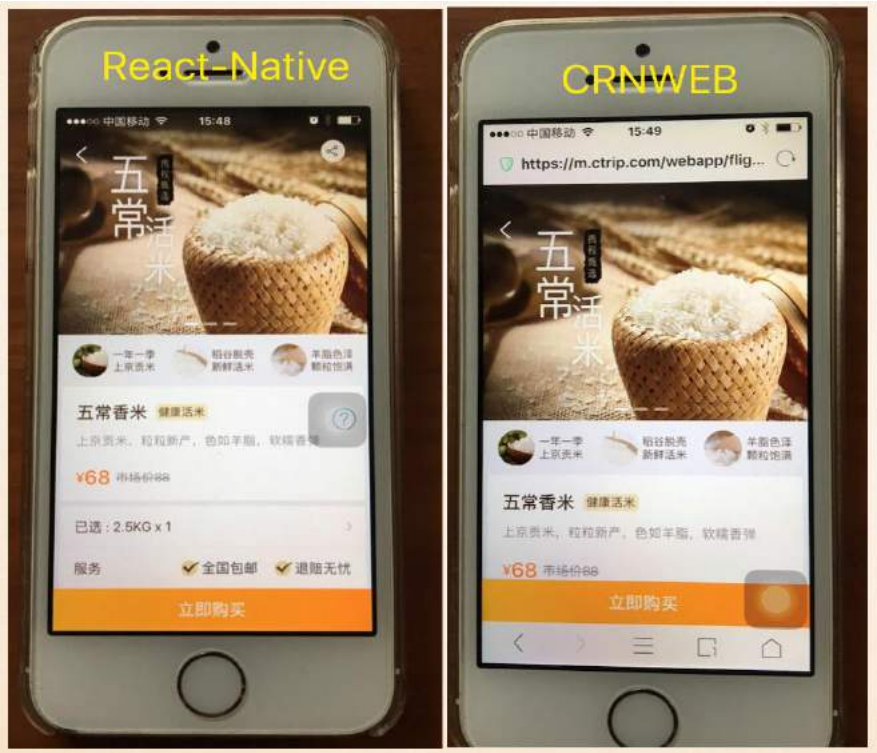
另外我们使用了一些工具，能很好的将项目中的模块依赖关系呈现出来，比如说 Log 这个模块被哪些页面引用，首页这个页面引用了哪些具体的模块（如：

FStyleSheet,Log,utils,LinearGradient.web 等等)，模块有多大多少个，都可以非常清晰的展示出来。这样就非常方便对代码进行优化和处理，并使数据可视化了。

我们现在项目有多大，它的主要代码组成结构是什么样的，它的每一个模块，每一个依赖，每一个组件 size 占比多少，都可以进行精确的数据分析。比如说最大的模块，你为什么最大，你包含了哪些业务逻辑，是不是必须的。这些数据都可以进行再一步的思考。

CRNWEB 目前已经支持到了 React-Native 的最新版本 0.54 版本，React 升级到 16.2 版本，已经有众多页面升级上线。

最后看看实际项目运行效果对比:



快速排障，VI 能帮你做什么

[作者简介] 宋通，携程框架研发资深工程师，参与过分布式消息系统等多个中间件及框架产品的设计与研发，对分布式系统设计及程序性能优化有持续的兴趣。

一、VI 是什么

一般情况下，在携程我们是不建议研发同学直接从办公网络访问生产环境服务器的。这样做，除了安全方面的原因外，更重要的就是要维护生产环境机器运行环境的统一性。但这样也给故障排除增加了一些复杂性，比如在排障过程中可能会遇到以下场景：

1）明明我的 pom 里写的依赖某中间件版本是 A，本地运行也没问题，为啥到生产环境跑起来就感觉像依赖了版本 B？

2）程序报了连接数不够的异常，生产环境想看看系统参数，还要联系网站运营同学帮忙，要么还要东奔西走申请各种服务器权限？

为了解决这个问题，同时为了能更快地进行故障排除，我们研发了一款中间件 —— VI。

VI 的全称是 Validate Internal，直译过来是“内部验证”。看到这个名称就会有同学问了，内部是哪里的内部？验证是要验证什么？

简单来讲，“内部”是指应用程序的内部，包括应用程序所处的环境、用到的框架等静态依赖，以及 CPU、内存使用情况等运行时状态；而“验证”则是指对程序的静态依赖、运行时状态进行实时监控，以辅助应用 Owner 来验证当前应用状态是否符合预期。

因此，对应用程序而言，VI 可以实时采集程序及容器状态，并通过友好的可视化界面进行展示。

这段描述会让不少同学想到比较常见的 Metrics 中间件。有很多 Metrics 中间件可以以 Json 等相对友好的形式，将用户自定义的一些 Metrics 数据暴露出来；除此之外还有 Java 自带的 JMX（Java Management Extensions），也可以通过注册自定义的 MBean 来获取各种程序内部信息。那么，VI 相对于这些 Metrics 中间件的优势在哪里呢？

1）强大的数据自定义展示功能： VI 提供了一套默认的数据展示模板，用户只需以简单的 api 调用将待展示的数据交给 VI，就可以在本机的 VI 界面上看到友好的展示和交互：

2）完备的容器信息采集：就我们常见的运行环境而言，业务代码的运行，自上而下会用到很多容器：Spring IOC，Tomcat，JVM，Docker 等。常见的 Metrics 中间件多数只提供了基本的 Metrics 框架，很少包含具体的数据采集模块。

但是，这些容器的状态往往会对业务逻辑的运行产生很大的影响。每个用户都增加完整的数据采集模块没有必要，但排障时再想添加又已经晚了，而且普通用户也并没有权限登录生产环境去查看这些数据。对于这类情况，VI 可以以更加友好的方式，提供非常完备的支持：

3）一站式自助排障体验：对同一个应用而言，不同团队的关注点是不同的。产品经理关注用户体验，业务开发关注业务逻辑，框架关注稳定性和性能，运维关注容器各项指标。所以，一旦有用户反馈程序有问题，那么我们的排障之路就会很漫长。

基于友好的交互界面和完备的基本信息采集，VI 在一定程度上有助于缓解这个问题。VI 可以帮助你从应用的角度而不是从“理论上”的角度来看应用程序运行时真实感受到的环境、框

架、容器的方方面面，而且不需要基础团队的支持、不需要申请各种权限，也不需要为了查看不同的数据在各种工具间转来转去，显著地节约了问题定位耗时，减少因排障速度而增加的业务损失。

4）发布系统支持：完善的发布流程生命周期控制，可以很大程度上减少因发布而引发的生产故障。在这一点上，VI 通过自定义点火/健康检测组件，为发布流程的生命周期控制提供了必要的支持。

除了可以自定义业务点火逻辑，各种常见的公共框架组件也集成了 VI 的点火组件。通过与发布系统集成，从流程上控制了点火/健康检测失败的应用不会在生产环境提供服务。

虽然 VI 可以暴露很多底层细节情况，但 VI 本身与我们常见的监控系统（例如 CAT） 还是有很大区别的，主要表现在以下方面：

1）历史追溯：VI 是无法追溯历史监控数据的，用户只能从 VI 获取到实时的监控数据。之所无法追溯历史数据，是因为 VI 的设计目标之一是帮助用户快速排除当前正在发生的故障，并不关心历史曾经出现的问题。而且考虑到有些监控数据的获取代价较高，且业务正常时并无太大参考价值，所以 VI 被设计为只有当用户存在访问行为时，才开始采集数据，当用户访问行为结束后即关闭数据采集。

2）告警：告警功能几乎是监控系统的标配，但 VI 并不具备这样的能力，原因是 VI 并未在后台持续采集数据，因而并没有衡量系统指标变化的能力，所以也无法针对指标变化提供告警功能。

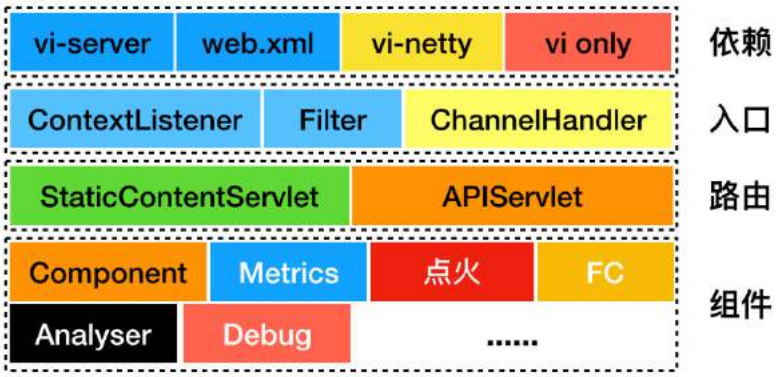
3）数据处理/分析能力：VI 并没有中央节点来处理/分析应用的全局数据，因而无法从宏观上对应用健康情况进行评价，而这一能力是完善的监控系统需要考虑到的。

因此，虽然 VI 可以实时采集很多监控数据，但 VI 的设计目标并不是成为一个监控系统，而是帮助用户快速定位/解决问题的工具。

接下来的内容会向大家简单介绍下 VI 两个基本功能（交互设计/在线调试）的设计细节，供大家参考。

二、交互设计

VI 的使用方式非常简单，常见的 Web 应用只需要添加 Maven 依赖，应用启动后即可通过 VI Portal 或直接 IP 访问特定 URL 来使用 VI。那么 VI 是怎么实现只增加一个 Jar 包依赖就可以提供页面交互呢？是否所有类型的应用都可以这么简单地使用 VI 的页面交互？



1) 依赖：VI 接入最简单的方式就是只添加 Maven 依赖，而无需进行配置和额外代码编写。这种接入方式条件就是需要使用 servlet 3.0 及以上版本，默认已被 Tomcat 7 及以上版本所支持。

对大多数业务应用场景来说，这个条件是非常容易达成的。但如果应用并没有使用 Web 容器，就需要根据实际情况，增加 vi-server 依赖或者 vi-netty 依赖。其中，vi-server 依赖内嵌了 jetty，本质也是通过启动一个内嵌的 web 容器来实现界面交互；vi-netty 则使用了 netty 自带的 http handler，为应用程序监听的端口增加了 http 协议处理能力，以启用 VI 界面交互。

2）入口：VI 的交互界面本身并未跳出 J2EE 的规范，因此其入口也仅根据运行环境不同略有变化，这部分对用户是基本透明的。入口的基本思路是，在应用程序生命周期尽可能早的地方，进行且只进行一次初始化；尽可能利用常见框架来带动完成 VI 的初始化动作。

3) 路由：VI 路由动作分为两块，静态资源路由和数据路由。其中，静态资源路由负责将 http request 请求的静态资源，从 jar 包的 resources 中加载并 response 给请求方；数据路由则有点像 dispatcher 的角色，根据不同的 api 路径，找到对应的 component，从中获取数据并反馈给请求方。

4）组件：VI 的内容生产者。根据需求，组件大致可以分为控制型组件和数据型组件两种类型。其中，控制型组件是会影响到程序状态的，比如点火组件如果点火失败，程序是不会对外提供服务的，这类组件可能并不是由外界请求触发，而是在 VI 整体初始化的时候就被触发了。另一类数据型组件则不会影响到程序状态，数据型组件只被动地收集数据，而且只有在真正使用 VI 时才会开始收集，以期尽可能减少对应用程序的影响。

从上述描述可以看出，VI 交互设计主要遵循了几个基本原则：

1）尽量避免侵入业务程序：这个也是目前各种中间件都尽量遵循的原则之一。这样做有几个好处，接口越少，用户的学习成本就越低；接口越简单，滥用和误用的可能性就会越少；业务代码无侵入，就意味着中间件代码与业务代码耦合性会很低，排障难度也会变小。

2) 模块化/插件化：VI 允许业务自定义点火逻辑，它自身的点火逻辑也是通过同样的机制来实现的；VI 允许业务自定义 metrics，它本身也通过 metrics 组件提供出很多常见的

metrics 出来。因此，对 VI 来说，很多功能，只是定义了一层 SPI，用户可以自己去实现这套 SPI，甚至 VI 本身很多功能也按照同样的逻辑来实现的。这么做的好处是灵活、层次结构分明，便于维护和扩展。

3）尽可能少的资源消耗：对应用程序来说，VI 算是辅助类型的组件，因此，VI 不应该对应用程序的稳定性产生不好的影响。考虑到这一点，VI 只有在有用户打开界面时才实时开启数据采集功能，且对整体内存消耗做了严格的限制，尽最大的努力以减少 VI 本身对应用程序资源的消耗。

三、在线调试

我们有时候会遇到这种问题，同一份代码，在测试环境运行得很好，一旦部署到生产环境就会偶尔出现意外的问题，排除掉代码和运行环境的问题后，一般会猜想是否上下游的某些数据交互存在问题。但不巧的是，可能对应的数据处理逻辑事先并未考虑到监控埋点。这种时候我们可能不得不修改代码，增加监控埋点，重新打包发布，再验证先前的猜想，而猜想验证失败后，很可能又要从头再来一遍。

VI 的在线调试功能可以极大地降低这种情况下的排障负担。

所谓的在线调试并不是真的给应用加个断点把应用阻塞住，而仅仅是给业务代码的某个类某一行加个标签。VI 会在标签生效后，动态修改对应类的字节码，并促使 JVM 重新加载该类。VI 修改的类字节码部分非常简单且可靠，没有任何变更操作，只采集了对应标签的上下文信息，通过交互界面展示给用户，告诉用户断点位置的上下文情况。

所以，这个功能的难点在于如何在程序运行时动态修改字节码。

JDK 的 java/lang/instrument 包提供了这种功能。JDK 对这个包的说明是这样的：Provides services that allowJava programming language agents to instrument programs running on the JVM. Themechanism for instrumentation is modification of the byte-codes of methods.

也就是说这个包，通过提供字节码修改的功能，为 Java Agent 提供了操纵运行时程序的能力。

如果有了解过 Java Agent，应该知道通常我们要为 JVM 增加 Agent，是通过增加 JVM 参数来实现的，VI 在线调试的第一版就是这样做的。这样做有个问题，需要每个希望使用在线调试功能的同学都修改自己的 JVM 参数；或者需要 OPS 同学帮忙统一添加 JVM 参数，无论哪种方式都不符合 VI 尽可能简化接入方式的目标。所以 VI 在随后的版本，改为使用反射的方式，通过调用 HotSpotVirtualMachine 的 loadAgent 方法，来实现动态加载 JavaAgent 的能力。

自此，VI 的在线调试功能就变得很好用了，无痛接入，界面友好，动态启用。用户不必通过重复猜想-埋点-打包上线-验证猜想的过程，就可以获取想要的信息，极大地缩短了此类问题的排障时间。

四、结语

做程序员久了，谁没写过几个 BUG，有 BUG 不可怕，找不到 BUG 才最可怕，因为定位问题太过耗时而影响到用户更可怕。这么可怕的事情，携程 VI 希望能尽最大努力帮到程序猿们。

Meteor 实时计算平台架构与实践

[作者简介] 何彬，携程市场营销研发部高级研发经理，10 多年互联网研发和架构经验。2014 年加入携程，负责携程广告、新媒体推广和市场大数据平台的构建、研发工作。

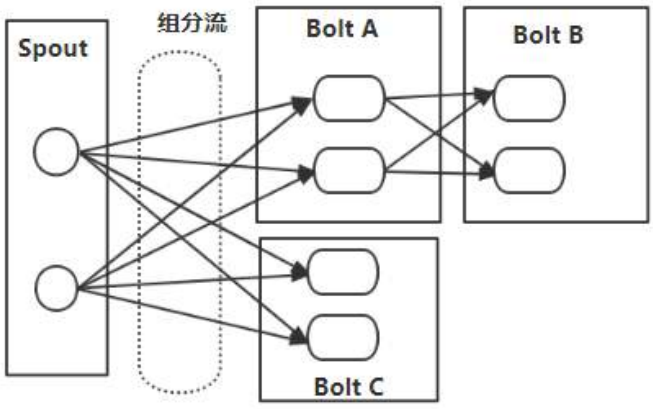
一、前言

营销场景计算需求多种多样，场景模型也纷繁复杂，计算要求的资源配置也大小不一，系统更新部署步骤繁琐，人工操作亦有极大的安全风险。随着公司个性化营销的推广，类似的资源需求也越来越多，大集群支持、资源共享、资源效率是重点关注的问题。

本文将介绍携程市场营销基于 storm 框架的 meteor 实时计算平台，解决日益增长的市场部业务需求。

二、什么是 Meteor

随时市场业务的不断发展，对实时计算的需求也逐渐增大。一直以来，我们根据市场的不同需求定制开发所要计算的 Storm 应用，Storm 实时运行的应用包逻辑上是一个 topology，一个 Storm 的 topology 相当于 MapReduce 的一个 job，不同是 MapReduce 的 job 有明确的起始和结束，而 Storm 的 topology 一旦被初始化就会一直运行下去，形成的 topology 是有 spout、bolt 通过数据流分组连接起来的图结构。如下：



按照以上的结构拓扑图，很多情况下出现一个问题，针对一个实时计算的场景 topology，当我们需要改变当前拓扑的某一个或者某几个 spout、bolt，又或者我们仅需要增加一个处理节点，我们该如何处理？

Storm 虽然可以通过 rebalance 进行动态调整 worker, executor 等并发数，但是不支持 spout、bolt 节点的动态调整，一旦 topology 被初始化，其 spout、bolt 节点的数量和配置参数也就相对固定。传统的架构方法是新增一个 Storm 应用，面对很多实时计算的需求，就需要新增这样的 Storm 应用，这些应用所要求的开发资源和集群规模也就越来越多、越来越大且难于

管理。

传统的解决方案已经成为业务发展的一个瓶颈。如何满足日益增长的计算需求、提高开发及系统资源的利用率成了我们急迫需要解决的问题。

因此，我们对 Storm 进行了二次封装，结合节点管理，图形计算、自动编译、动态打包、自动发布及部署等工具进行了一次系统的封装，封装后的平台在我们内部称之为 Meteor，意思是快速达成美好的愿景。

三、技术架构实现

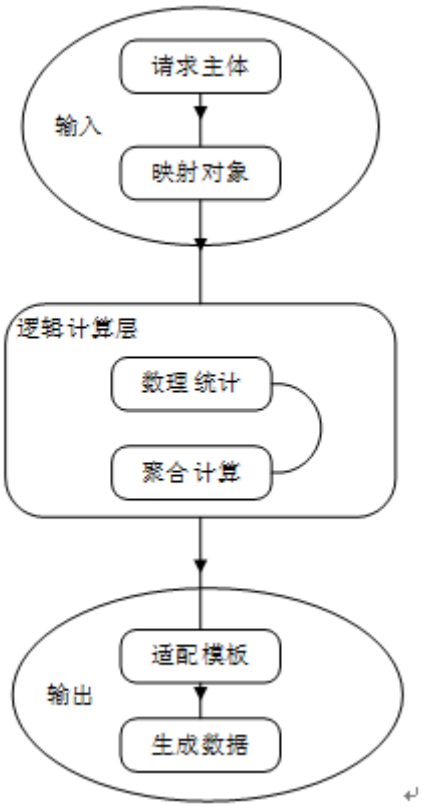
下图给出的是 Meteor 的系统架构，自底向上分为驱动层、数据操作层、图计算层、服务层、应用层，其中驱动层、数据操作层、图计算层是 Meteor 的核心层。

最下层是驱动层。驱动层包括 Meteor 分别在 Storm、Spark 等分布式计算系统上的实现，也就是对上层提供了一个统一的接口，使上层只需要处理场景计算等逻辑，而不需要关心在分布式计算系统上的实现过程。

其上是数据操作层，主要包括逻辑表达式算法等操作。再往上是图计算层，也是我们要了解的核心，包含 Graph DAG 数据流图的实现（图的创建、编译、打包、发布和执行）。再往上是服务层和应用层。

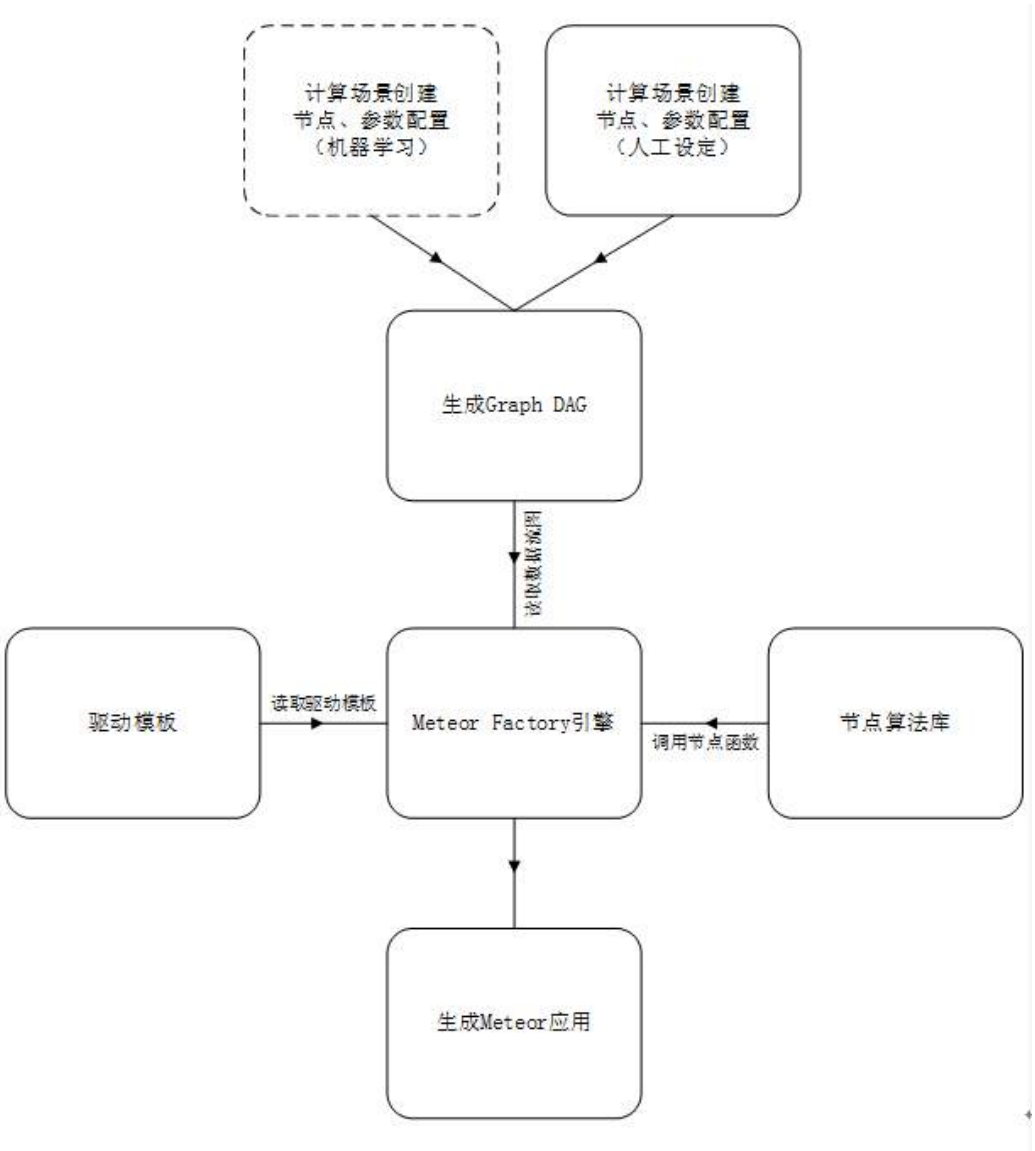
Meteor 是用数据流图做处理的，Meteor 的数据流图是由计算节点（node）组成的有向无环图（directed acycline graph，DAG）。我们先创建一个数据流图（也称为网络结构图），如图所示，看一下数据流图中的各个要素。图中讲述了 Meteor 的运行原理。图中包含输入（input）、逻辑计算（function）、输出（output）等部分。

它的计算过程是，首先从输入流开始，一层一层进行前向传播运算。逻辑计算可以定义一个或多个节点，每个节点代表一种算法，不同算法定义不同的传参，根据参数的配置可以调整计算的结果。然后进入输出层，输出层根据不同的客户端输出不同的数据格式，最后生成数据。如图所示，生成的每个数据流图从上往下依次进行计算。

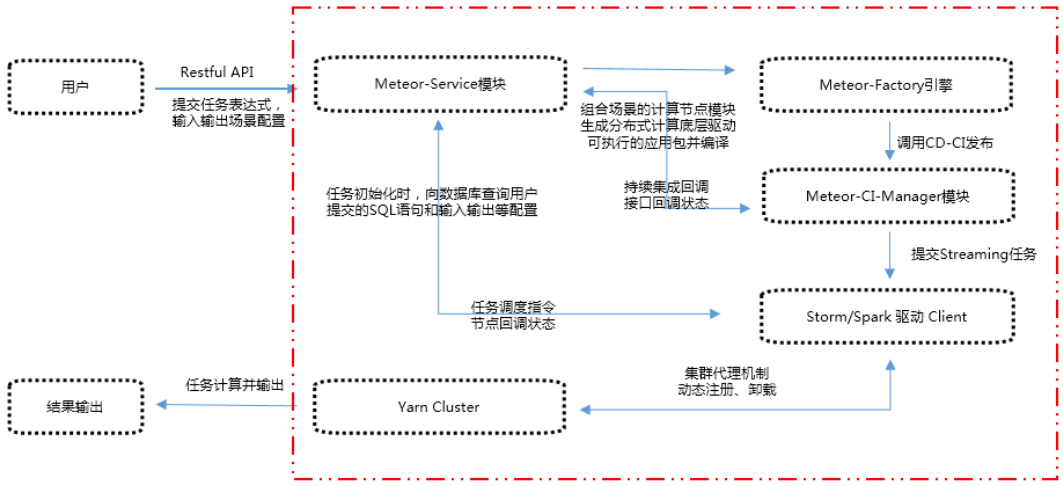


Meteor 数据流图由 Meteor 治理中心统一管理和运维，所有的数据层和计算节点统一在 Meteor Service 中进行注册，分配和调度。Meteor Service 是整个系统的核心模块，用户通过 RestAPI 调用 Service 接口服务，提交场景配置和节点算法参数，目前由人工的方式根据不同的业务需求创建计算场景和计算节点参数配置。(计划由机器学习取代，机器学习直接生成数据流图)

生成后的数据流图注入 Meteor Factory 进行加工，Meteor Factory 是 Meteor 的应用引擎模块，主要是将组合场景的计算节点模块进行代码集成并编译打包，根据数据流图中配置的计算算法和参数，从节点算法库中调取相应的代码，触发 Factory 代码生成器，代码生成器根据 Storm 驱动模板生成相应的代码，生成好的代码执行自动编译并打成 Storm 可执行的应用包。

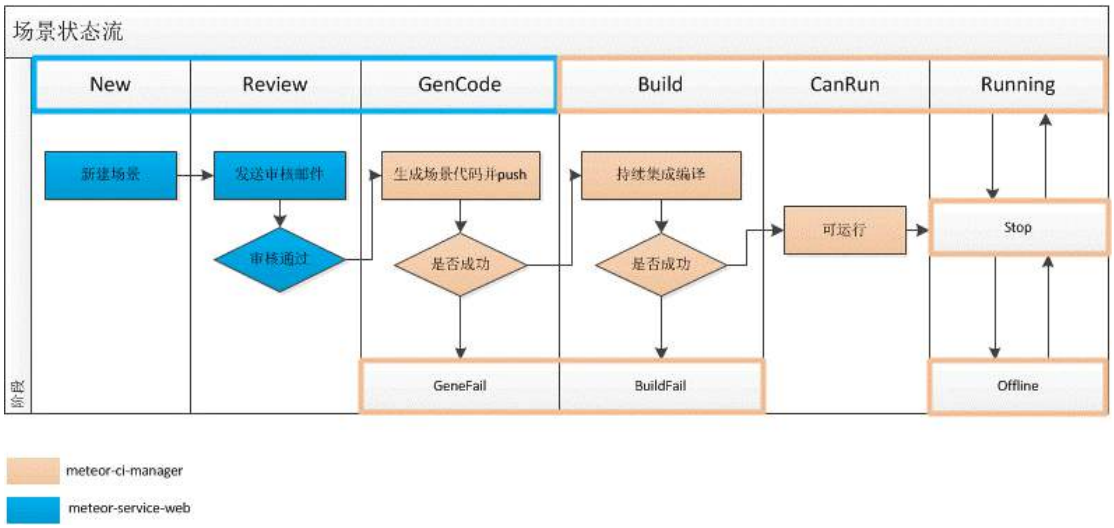


Meteor CI Service 模块将编译好的应用包和发布系统进行集成，由发布系统调用底层 Storm 客户端驱动，自动将应用包发布到 Storm。Meteor CI Service 和 Storm 客户端驱动的任务调度通过 Meteor Service 进行管理。



Meteor 任务调度由不同的状态控制和管理，以保证整个系统运行的有序性。

1）场景被创建，不同的场景由不同的节点模块组成，场景创建时选取相应的节点模块，此时场景的数据状态为 NEW;
2）新建完成的场景需要被审核，场景新建完成后提交给相应的审核，提交审核过程中的的场景数据状态为 REVIEW;
3）当审核通过后场景代码开始生成，代码生成过程中的场景状态为 GENCODE;
4）代码生成完成后进行编译动作，编译过程的场景状态为 BUILD;
5）编译结束该场景就可以被执行了，可以被执行的场景状态为 CANRUN;
6）当场景在运行过程中状态为 RUNNING。



四、Meteor 的特性

4.1 高可用

Topology HA

Meteor Service 会定期与 topology 进行心跳交互，若 Meteor Service 检测到 topology 心跳超时，则会重新调起一个新的 topology，新的 topology 会将自身信息写入 Zookeeper 中，其它 Container 与 Supervisor 将通过 Zookeeper 来识别到新的 topology，从而保障 topology 的 HA。

Container/Worker HA

Container 会定期与 Meteor Service 进行交互，若 Meteor Service 检测到 Container 心跳超时，则会重新从资源池里调起一个新的 Container 接管原来失效 Container 的任务，并把新的任务分配写入 Zookeeper 中，以便其它 Container 识别新的 Container 的位置，从而保障 Container 的 HA。

4.2 二级调度

封装后的 Storm 只需管理 topology 的调度，其它如 UI 访问、任务下发、拓扑、metrics、节点心跳等，均由 Meteor Service 的二级调度。

4.3 资源隔离

封装后每个 topology 实例下只有一个 supervisor，并且每个 supervisor 里只用一个 worker，通过每个 worker 来进行资源隔离。此外，不同节点可以任意组合一个新的 topology，同时我们引入权限管理，不用用户申请的计算资源（数据和节点算法）可以做到相互隔离，每个任务只能运行在授权的通道内，以此保证不同用户申请的资源不会被他人调用。

4.4 自动发布和部署

由于一个场景一个 topology，而一个 topology 实例可以理解为一个虚拟机，用户资源申请具有随机性、配置个性化等特点，因此对我们配置管理上必需具有自适应性。对此我们通过本地生成应用包，通过产品化把计算管理配置、Storm 与 CD-CI 发布系统打通，并把资源配置、应用包的发布和部署等功能产品化，以达到自动发布和部署的目的。

五、应用效果

如前言中介绍，由于携程市场营销要面对非常复杂的业务场景，携程的数据本身就会有很多的维度和指标细分，数据分析团队和 BI 人员也会针对数据的不同维度和各种维度组合进行各种各样的分析和计算。

A 场景举例：

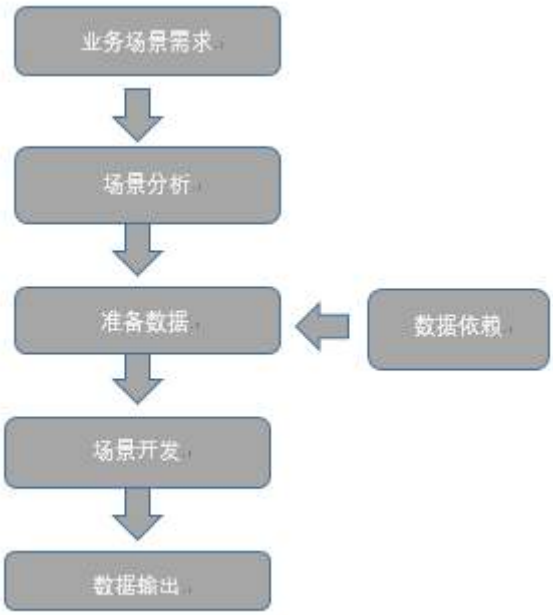
正在浏览上海某五星级酒店详情页
且三天内有访问过同样酒店的浏览历史
且三个月内没有下过任何订单

且是 APP 激活
且会员等级是普通
且如果是公众号关注用户，在 A 媒体投 X 广告，如果是渠道预装用户，在 B 媒体投放 Y 广告

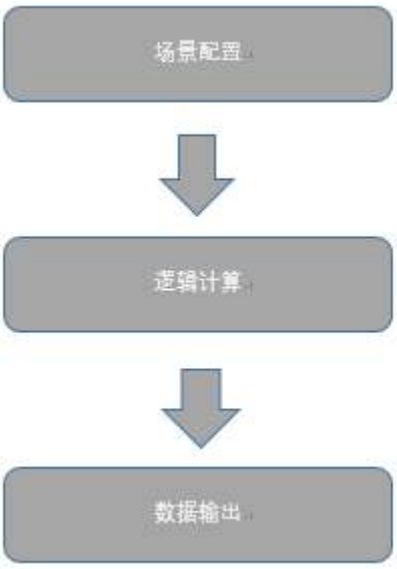
B 场景举例：

正在浏览北京到上海机票列表页
且三天内有访问过上海酒店的浏览历史
且一个月内没有下过任何订单
且是渠道预装用户
且会员等级是普通
如果用户权重指数为 5，在 B 媒体投放 Y 广告，如果用户权重指数是 8，在 C 媒体投放 Z 广告

以上场景传统解决方案，需要按照每一个业务需求进行分析，得出每一个条件判断的数据来源和依赖，对接数据源获取数据，在数据准备好后逐个进入 Storm 应用的开发、发布和部署，实时计算的数据结果可能还要对接不同的客户端。



Meteor 平台的解决方案只需要三个步骤即可完成数据结果的输出，按照业务需求选择合适的计算类型和参数配置，启动计算场景，就可以得出相应的计算结果，并且可以实时调整计算逻辑（判断条件）。



以上两个场景，可以发现传统的解决方案和 Meteor 的差异还是比较明显的，主要体现在以下几个方面：

- 业务场景多且复杂，单个场景开发效率低下；
- 随着场景数据量和数据分类的增加，单个场景数据的计算、存储和维护成本越来越大；
- 单个场景的开发应用不可复用，开发资源利用率不高；
- 数据查询重度依赖底层数据框架和数据结构，查询效率得不到满足；
- 单场景数据存储和处理对于实时性要求不能满足；
- 单个场景开发作业，受限开发资源和周期；

Meteor 平台通过统一的管理配置模式，实时进行计算节点的动态配置、调度和计算，业务人员可以很方便的进行业务场景的创建、运行、暂停、下线等操作。实现后台配置化，自动化交付上线。完成从被动的需求开发(业务驱动)到主动的满足需求(技术驱动)的重要转变。主要业务优势可以总结为以下几个方面：

- 计算节点一次开发多次复用；
- 场景条件支持任意搭配和多重组合；
- 支持业务场景从原来的几十个场景扩展到上万个场景；
- 提升数十倍的开发效率，交付效率从原来的平均 4 个工作日缩短到 2 个小时；
- 满足业务需求快速上线，提升营销投放效率；
- 平台场景状态、数据流量、节点计算、异常容错监控可控；
- 平台投放场景运营数据可视化；

Meteor 平台上线后，对市场营销业务提供实时数据计算和数据查询服务，目前已经实现每天几十亿级的数据查询，覆盖 90%的实时数据计算任务，系统指标和性能指标主要体现在以下几个方面：

- 计算节点的响应时间 99%在 50ms 以内；

- 计算节点插件化开发，大大降低系统复杂性和耦合度，提高系统稳定性可以达到 99.9%；
- 底层驱动多元化，可适配多种流处理计算框架；

六、结语

基于 storm 框架的 meteor 实时计算平台，是携程市场团队自行研发的自动化的实时计算平台。

它基于 storm 框架，通过重新设计整个底层架构及运行逻辑，并添加权限管理、限速、监控、日志等辅助功能，经过产品化并与公司发布平台打通，达到了用户申请即可用、配置个性化、大规模集群的要求，操作高效且自动化。

携程国际化进程中，是怎么做站点多语言处理的?

[作者简介] 祁劢，携程国际业务部内容研发团队 Leader，目前主要负责信息类项目产品设计、技术架构与团队管理。CG 爱好者，喜欢细致描绘世间百态的通俗小说，喜欢探索，乐于体验各地风土人情。

一、项目背景

携程国际业务部门（IBU）是携程多语言站点的研发及维护部门，主要面向国际客户提供携程优质的产品与服务。在国际业务部门研发的众多信息处理产品中，IBU 内容研发团队的携程内容翻译平台-CTran，主要负责对酒店、机票等业务的内容信息进行多语言处理，为携程国际化提供支持。

作为 CTran 系列的产品设计与技术架构，我与团队一起经历了 CTran 项目的演进与革新。

CTran 目前最新的 V3 版本于 2018 年年初正式上线，是经过重新设计和整体重构的技术产品，其设计既是对以往解决方案的反思，也希望通过技术建立新的流程，解决当前面临的业务问题。

在此与大家分享我们实现 CTran V3 收获的经验，希望我们推进国际化内容建设的阶段性成果，可以给本部门同类项目，以及行业同类需求提供借鉴，共同促进业务发展。

二、国际化面临的语言问题

携程中文系统拥有海量的中文旅游类商品资源，为国内用户提供丰富的旅游服务。IBU 主要面向海外用户，如何合理利用中文系统的资源宝库，为海外用户提供优质的无语言障碍的服务，是一个难题。

从多语言转换的难易程度来说，结构化的信息是易于进行多语言转换与维护的。

然而，携程中文数据主要面向国内用户，其业务数据形态设计上并没有考虑过翻译转换的成本。由于旅游商品的特殊性，存在大量的非结构化信息需要转换，虽然信息维护部门也意识到了非结构化信息对国际化的影响，但由于种种原因，信息的弱结构化甚至非结构化的情况仍然较为普遍。因此，IBU 需要针对多语言转换需要重新对数据进行规整与清理。

此外，信息是持续变化的，为了给用户提供确切的多语言产品信息，需要积极同步源信息的变更情况，及时进行翻译转换。

携程的旅游产品可以略分为机票和酒店两个方面。机票由于数据自由发挥的余地较小，常规

静态信息居多，多语言化较有优势，酒店数据情况则相对较复杂，对于一个酒店类商品，要达到能销售的程度，需要投入较多翻译资源对非结构化的数据进行处理。

CTranV3 多语言翻译平台承担了机票及酒店内容信息的多语言翻译任务。对于机票业务，目前覆盖处理的是政策类资源的翻译，主要通过实时翻译接口直接为用户提供服务；对于酒店资源，比较关注数据的变化，主要通过离线方式进行多语言翻译处理。

三、CTran V3 介绍

3.1 概要

CTran V3 整体结构如下：

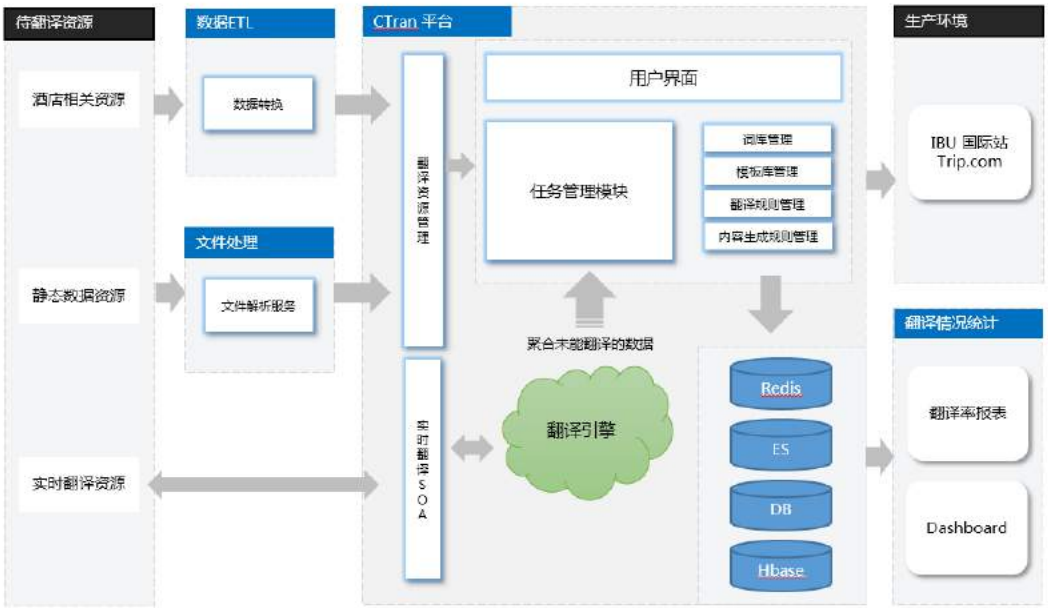


图 1 – CTran V3 整体结构

整体而言，数据流入 V3 后，经过离线或在线的翻译处理，再通过各种输出接口流出。

暂时未能自动化翻译处理的数据，将由翻译人员介入，通过 V3 提供的翻译辅助、内容分析等工具，译完后输出。翻译人工维护的结果也将将通过多级词库的形式存储到 V3 系统中，用于将来的批量处理等流程；对于线上的翻译错误，V3 也提供了高效的数据修正手段，方便大规模数据修正与多语言信息质量提升。

具体业务方面，酒店相关资源的翻译流程首先经过数据转换，通过离线方式翻译处理后，推送到 IBU 的酒店相关信息数据库，最后在线上呈现。机票、铁路等其他可以实时转换的信息，由调用方直接请求 V3 实时翻译服务，获得翻译结果。其他静态资源，则可以通过文件任务接口，利用 V3 的翻译辅助等进行处理，翻译流程结束后，再以文件形式获得翻译结果。

以上是 V3 与外界交互的几种主要形式。下面主要以酒店相关信息的翻译为例，介绍 V3 是

如何对数据进行翻译转换的。

3.2 酒店数据流入

酒店数据的多语言翻译是从数据源头开始关心的。CTranV3 通过数据同步，持续跟踪酒店业务数据的变化。数据信息经过适当转换后，通过与自身存储的中文酒店信息镜像进行比对，识别差异，对“内容变化引发需要翻译”这个事实作出判断，生成翻译请求。

数据的同步由 Ctran V3 的子项目——多语言数据服务（Langs）负责完成的。Langs 主要职责除了获取数据外，还用来接受 V3 翻译服务的酒店数据翻译更新推送，并负责确保将其推送到 IBU 的 Online 信息数据库，以完成业务信息上线。所以，Langs 是翻译服务与数据源、多语言业务方之间纽带。

Langs 同步信息处理的第一步是对业务数据按类型进行拆分，包装成统一的对象后，通过消息队列推送给数据比对模块。数据比对模块基于内容变更情况形成的需要翻译与否的判断，相关功能是通过表驱动(Table-Driven)的形式实现的。

所谓表驱动，即通过配置表定义运行逻辑。数据变化形成状态变化，各种状态组合形成状态集，通过状态集可以设定具体的操作。Langs 首先对状态进行判断，组合状态后查表找到对应的操作集合，按配置进行数据处理。

众多变化中较为核心的是内容的变化。在获取数据后，Langs 首先依照特定规则对数据进行清理，归整成特定的形式，在不影响语义的情况下再做一致性判断。

内容数据变更的判断并非是简单的文本相等比对，举个简单的例子，如果名称里出现一些符号，对于特定类型是不会影响语义的，也就不需要重复进行翻译，因此可以将一致性判断的条件放宽。类似这样的对特定类型的判断规则有很多，其目的也是尽量减少翻译所需要处理的数据量，降低翻译成本。

比对的结果会形成具体的状态，类似“有变化”，“无变化”等。某些情况，文本数据的“置空”可以视为一种逻辑删除，也被独立表示成一种状态。

状态变化与操作的配置表，在概念上可以如表格所示：

中文变更	英文变更	状态变更	行为 1	行为 2	行为 3
有变化	有变化	有变化	更新中文	更新英文	发送待翻译消息
无变化	无变化	无变化	无需操作	无需操作	无需操作
…	…	…	…	…	…

表 1 表驱动配置示例

其中，黑底色的列是状态组合，白底色的列是对应的操作集合。通过结合 Java 枚举类型及 Spring Bean 注册机制，在项目内创建特定的 Bean，利用 Spring 对于类型 Bean 的自动发现机制同时获取 Java Annotation 的元信息配置，自动将代码逻辑与数据库保存的配置表建立

关联。

表驱动的设计提升了逻辑处理的抽象程度，更重要的是，由于操作逻辑的外露，通过良好定义的行为说明文档，许多也许需要写在代码中的 If / Else 条件判断也可以由非技术人员，比如运营人员，通过 Excel 进行调配。由于规则本身具有较强的描述性，查询日志即可明白某条数据是如何被消费与处理的，便于错误排查。当希望获取周期内变化情况时，仅需要根据特定的规则组合查询日志即可，也很方便。

类似这样业务逻辑配置化的例子在 V3 中还有很多，对于有调整变更需要的模块，这样的设计一定程度上帮助提升了软件的可维护性。

3.3 翻译任务

CTran V3 的翻译是以任务为单位的， 作为一个维护着上亿数据的翻译平台，当数据规模超过了团队能够处理的极限，就需要对数据进行规划，有重点的投入翻译力量，有先后的进行翻译。

常规做法一般是预设优先级，并且将这样的优先级直接与数据绑定。旧版 CTran 在最初数据较少的情况下，也是这么做的，导致当数据大规模增加时对优先级进行调整的困难。

因此，V3 并未对酒店数据本身做优先级标记，而是通过另一种方式对待所维护的数据。优先级的最终目的是筛选数据，所以，如果能够支持不同的条件组合，对大规模数据进行快速搜索过滤，也就可以实现优先级的动态调整。

这样，优先级就变成了一种搜索/数据筛选的配置，修改优先级也就是修改搜索条件。之后，在过滤好的数据集合之上，译者可以自行决定任务规模，制定翻译计划。

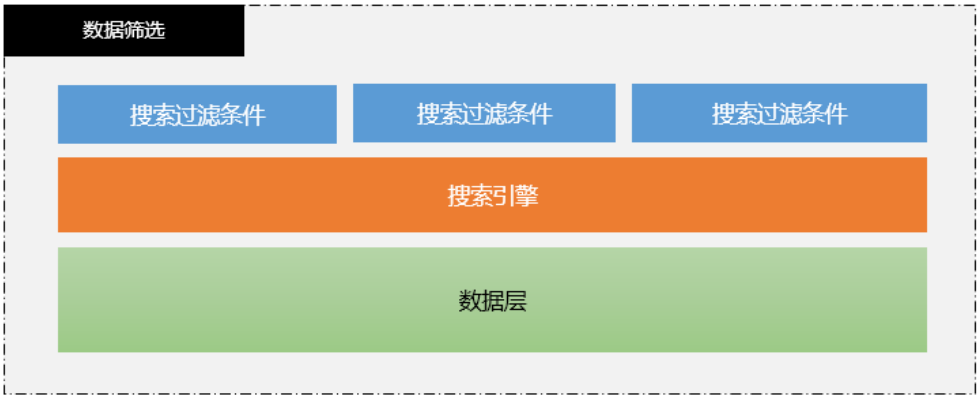


图 2 数据筛选

为了加速搜索，V3 通过一些常规的工程手段，比如简化数据存储，适当冗余数据，优化数据库索引，混合 Elastic Search 搜索引擎，将分页搜索改为上下翻页搜索等策略和技巧，对酒店合理业务需要的搜索条件组合进行针对性的优化，保证了数据定位的速度。

在可以快速进行搜索后，就需要定义优先级配置文件了。所谓优先级定义文件就是一些维度条件的组合，比如酒店所在城市、星级、类别、子母酒店规则甚至酒店基本 ID 本身。

而优先级定义文件的来源可以是用户上传或者 BI 统计的用户 PV 酒店数据、主动定量分析等方式形成。支持动态优先级以后，运维即可以通过交互的方式，查看在某个维度下数据分类的情况，安排翻译计划。

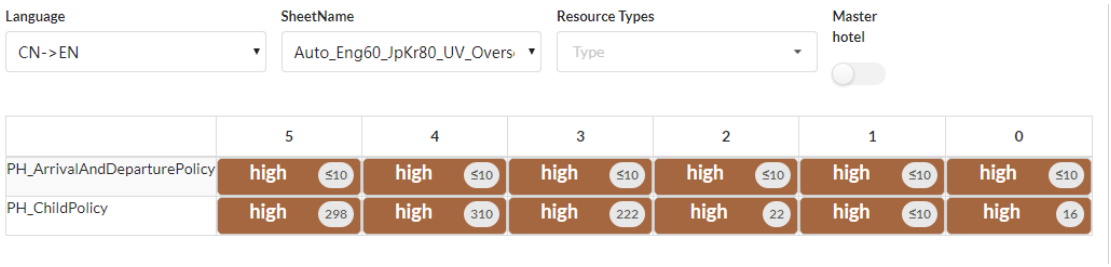


图 3 优先级选择示例

获得数据过滤的结果后，要进入翻译流程，还需要再确定一下翻译的范围。

V3 允许根据用户工作计划设置自动或者由用户手动的决定一个任务需要包含的待翻译数据范围。自选数据加入任务包，类似网上购物“加入购物车”的概念，并且流程上允许在形成的任务包上进行数量调整或者部分优先完成，保证了工作的灵活性。

翻译人员在平台的操作一般流程可以归纳为：选择数据，创建任务，进行翻译。简单而直接。

除了常规的翻译辅助、内容校验等支持，以任务为单位的 V3 也很注重译者的协作。借鉴类似 IM 软件建群的方式，V3 允许翻译人员自由组成多个翻译组，并且不限制个人加入组的数量。任务包将在译者所加入组的成员间流转，方便了翻译内容的校审与协作。每一步转移均会有相应的图形化呈现及日志记录。

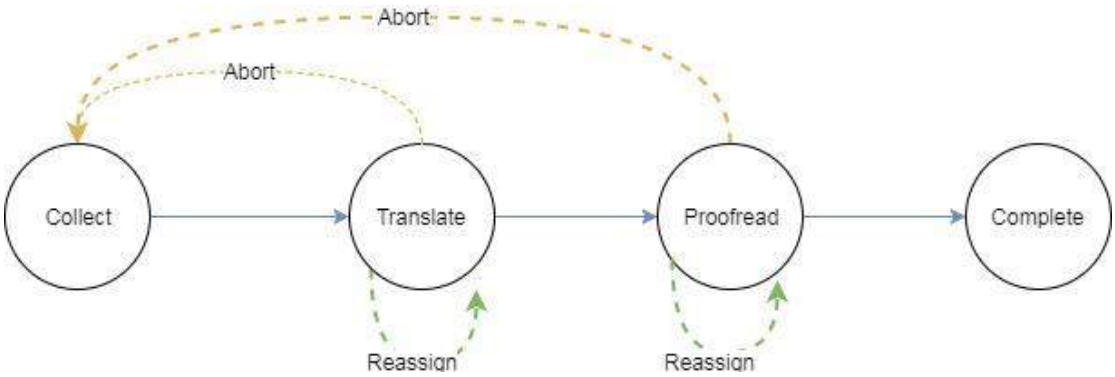


图 4 翻译流程

可以看到，常规流程可分为“收集->翻译->校审->完成”几个阶段，通过允许“重分配（Reassign）”这样的转换，可以方便的支持多译者间翻译任务流转。校审阶段依照设置也是允许退回操作的。这样的设计为翻译基本流程提供了足够的支持，足以覆盖绝大多数情况。并且，我们在业务上允许译者将翻译任务包进行拆分，提交已完成的部分，确保了任务完成

的灵活性，日志记录及图表呈现会忠实呈现各种状态迁移。

图 5 翻译流转图形表示

数据变更方面，V3 利用公司的大数据存储，通过有效设计事件日志，对所有影响线上的数据变更情况均记录有日志，通过合理的事件记录，任何变更都有迹可循。

3.4 数据分析与报表

要处理数据，首先要了解数据。

CTran V3 对接了携程的大数据处理平台，进行各种常规数据分析，帮助译者、运营及研发了解数据。常用的分析手段包括利用自然语言处理工具对句子按句法边界、词法边界进行拆分，统计频率，句式模板识别、关联数据翻译情况分析等，同时也会对翻译情况等生成统计报表，反映现实状况。

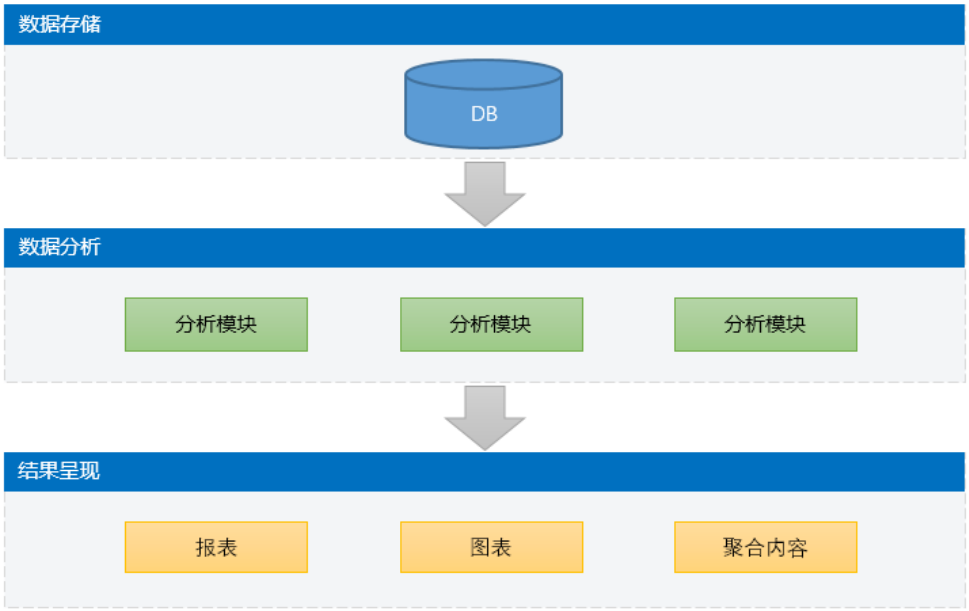


图 6 数据分析

V3 在处理全量数据分析的同时，结合上面提到的任务的动态优先级划分，也会自动对用户划定范围的数据进行分析，所有的动态优先级条件划分均可以作为既定范围进行分析操作，以便用户可以有重点的看到分析结果。分析结果在 V3 中提供展示，并通过各种辅助工具帮助翻译人员识别特征数据，重点翻译特征语句，调优 V3 自动翻译规则和算法。

翻译情况统计报表会按时间跨度自动预聚合，以方便在不同时间跨度查询下仍然保证查询速度，又能够进行足够精度的范围查询。目前支持从小时到按月等维度进行数据查询。统计报

表不仅会通过 Dashboard 在 V3 中展示，也支持通过邮件的形式发送给运营人员。我们在公司邮件服务的基础上，建立了支持自定义收件人组合及邮件模板维护的邮件发送平台 yeMail，用以解决相关问题。

虽然从数据规模上看，V3 所覆盖的数据不过千万级别，但我们设计各种数据分析通路其实也是为未来铺路。解决自然语言翻译这样的问题绝不能只着眼于 V3 目前维护的数据，一定要广泛利用各种数据来源，有针对性的尝试各种可能性，设计各种流水线就是为了方便这样的处理。

3.5 翻译引擎的工作原理

CTran V3 处理批量翻译的另一个手段是使用自研的翻译引擎。V3 的引擎不同于通用的翻译引擎，不是为处理未知形态的数据翻译而设计，通用的翻译引擎也并不适用于携程类型化的翻译。

V3 所有翻译处理均是在数据类型确定的前提下所编写的转换逻辑，务求确保翻译准确。由于自然语言处理的特殊性，并不存在唯一的在所有场景下均能在各方面都取得良好成效的解决方案，所以我们按照情况对翻译逻辑器件进行调配组合，通过责任链共同构成现在的翻译引擎。常规实现可分类如下：

1）基于句式模板的翻译；
2）基于模糊匹配的翻译；
3）基于词拆分的翻译；
4）依照类型的业务类型的一些多级词库优先取值规则；

3.6 基于句式模板的识别

基本原理是对内容的逆模板化。酒店数据等等已经设定好分类类型的数据往往包含一些可以枚举的成分，比如城市名称等信息。这些可枚举的成分即可作为可识别的对象，依据类型进行变量替换，并且编序。

变量替换之后的内容，即为句式模板，依照句式模板找到数据库中对应的翻译模板，然后再对可枚举的词进行词库查找，依照语法规则进行形式变化、对位替换后，即可产生翻译结果。

这种操作由于模板的特性，会受输入内容影响，导致语义相近的句子需要建立多个模板进行映射。由于相似度并不能确定性的定位一个无法满足全匹配的模板，我们目前使用的手段是通过一些预定义规则对模板化前的数据进行清理归约，等价替换，以降低其多样性。

3.7 基于模糊匹配的识别

这是通过已有的规则去尝试套用到给定句式上，找到匹配后进行翻译的手段。适用于可识别的变量需要枚举的量过大的情况。通过模糊匹配找到对应的模板后，再对有差异的成分进行提取，尝试转换后使用译文词典翻译找到映射后再合并处理。

3.8 基于词拆分的翻译

主要有两种方式较为常用。第一种很直接，主要通过基于词典的分词器对传入内容进行拆分后，根据预定义的多语言类型词典，依照人工翻译规则确定不同语言的排序，然后适当转换后形成翻译结果。

另一种是通过识别短语中心词，然后依照中心词拆分，分析成分后翻译转换输出。前一种主要在携程的基础房型翻译中使用，后一种则针对国内地址等这样中心词（字）特定的类型比较有效。

3.9 翻译引擎调优

V3 中多数翻译器对数据的处理过程均是可见的。诸如模板等翻译组件需要协同译者利用她们的专业知识帮助完成。V3 通过暴露翻译器的中间数据结果让译者理解翻译器的工作流程，针对不同引擎的特性输出不同的中间解析结果文件，方便译者依照情况进行翻译调整，进而提升引擎的能力。

3.10 内容构建支持

酒店描述随意性较强，对翻译造成一定程度的困难。通过识别句子边界找到翻译模板进行多语言化也许有一定效果，但并不是最有效率的手段。

考虑到大批量的数据缺乏可能造成的搜索引擎收录排名的影响，适当采用内容构建补充数据也许是一种可行的方式。CTran V3 的酒店描述生成即是从数据构建的角度试图解决多语言信息完整性的问题。工程上这样的方式在游戏产业尤其是角色扮演游戏中比较常见，其原理是通过一些数值信息，联合内容模板，依照某种条件的选取策略，最终输出整体内容。

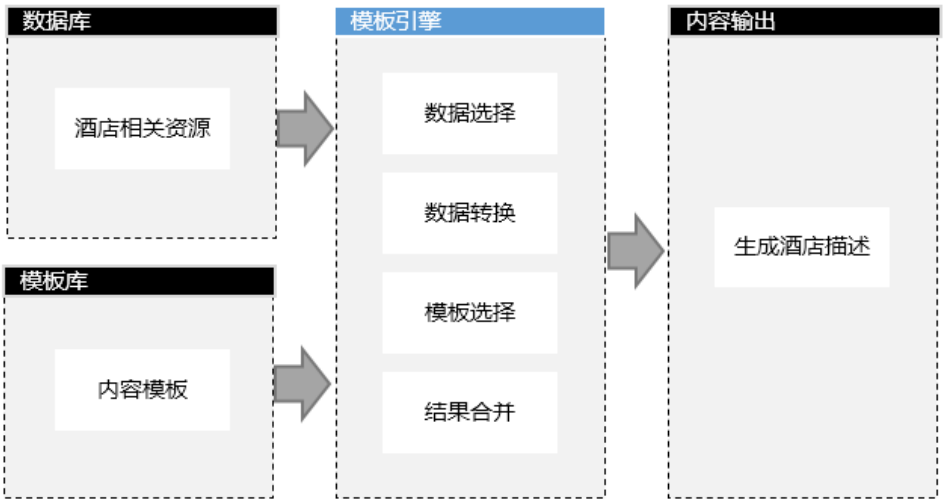


图 7 内容构建

V3 借鉴了这种方式，通过酒店关联信息以及内容模板，依照既定条件随机拼合，用来创建酒店描述。

酒店信息包括周边景点、商圈、车站等，利用各种地理信息与其他量化的信息做模板选择，只要通过不断维护关联内容模板就可以对酒店描述不断调整与更新，并且内容也是可控的。

这种手段的缺陷是生成结果比较生硬，句子间缺乏过渡，上下文关联不紧密，数据随着规模上升，重复度较高。由于其信息多少在酒店本身有一些涉及，整体可读性也不高。但是，相信随着自然语言处理手段的进一步使用，数据构成的进一步丰富，这类生成信息应该会提供更好的阅读感受。

Type	Level	Language	Content
Introduction	A	ja_JP	慌しい一日の終わりには、#{[hotelName]}でおくつろぎください。
Introduction	A	ja_JP	旅先の様々な魅力を発見したいお客様に、#{[hotelName]}は最適です。

图 8 模板配置

3.11 Job 管理

CTran V3 需要周期性的对数据进行大批量操作，需要有可靠易用的任务调度平台。我们跟随着携程的技术步伐，先后使用各种 Job 平台对数据进行批量维护，也经历了一个从自研 Job 管理软件到将信任托付给携程通用调度平台的过程。

V3 当前主要使用携程的 Job 调度平台 Qschedule 进行 Job 调度。Qschedule 下维护的 V3 相关 Job 已经超过 70 个，分别对数据的状态一致性、批量翻译、索引重建和缓存更新、词库定期快照等诸多方面提供支持。Job 的数量随着时间推移还会不断增加，为了合理对 Job 进行维护，IBU 内容团队定义了 Job 编写规范，可以让活跃 Job 自动登记到 V3 平台中，方便对 Job 进行分类维护。

Name	Type	App Id	Description	Switch Type	Sharding Strategy	Param
[redacted]	SYNC	[redacted]	房型拼接分词+特殊分词+酒店 全量同步Redis	NONE	NO_SHARDING	-
[redacted]	SYNC	[redacted]	扫描origin表新过期或新增的过期数据，发送消息取消Ctran任务	Q_CONFIG	NO_SHARDING	warni lastSc 时间 sendT
[redacted]	TRANSLATION	[redacted]	Flit fail translation 打包	NONE	NO_SHARDING	-

图 9 Job 维护

Job 在编写时使用了特定的 Java 的 Annotation 元数据标注，Job 在运行时会写入统一存储，然后 V3 中即可读取到相应的配置数据，用以分组排列展示。

翻译更新等会涉及到频繁的扫表。为了降低大批量数据扫描对数据库操作的压力，V3 对于

数据定位方式做了两方面的优化。其一是通过各种预建的搜索索引定位数据，其二是在 Qshedule 的 Job 之上建立 Job 管理抽象，通过携程消息平台中间件 Qmq 消息进行触发。

当人工通过页面设置操作逻辑时，其实是在构建能够触发 Job 的 Qmq 消息。通过定期聚合待发送的 Qmq 消息，可以将时间窗口内的多个逻辑合并成一个具体的 Job，统一运行，并且监控其状态。而且，相应的 Job 也是可以通过页面进行中断的。

3.12 实时翻译

CTran V3 的实时翻译服务是由 Flit 子项目提供的。Flit 是 V3 翻译引擎的对外服务接口。

Flit 实时翻译服务为提升 V3 内部翻译引擎实时性做了多项优化。我们在常规算法优化的基础上，针对实时请求的特点，为翻译引擎建立了多级缓存。上线以后，IBU Online 机票翻译 99%的请求的响应时间低于 5ms。

Flit 翻译服务也为自动聚合未能翻译数据提供了支持。结合公司提供的大数据聚合分析功能，未翻译数据会自动进行聚合统计，在 V3 中形成待翻译数据列表、统计图等，通过邮件等方式提醒翻译人员介入处理。

3.13 静态文件翻译支持

CTran V3 提供了以 Excel 文件为代表的文件导入翻译功能，可以通过导入 Excel 文档，交互式指定原文列的方式进行翻译任务创建。

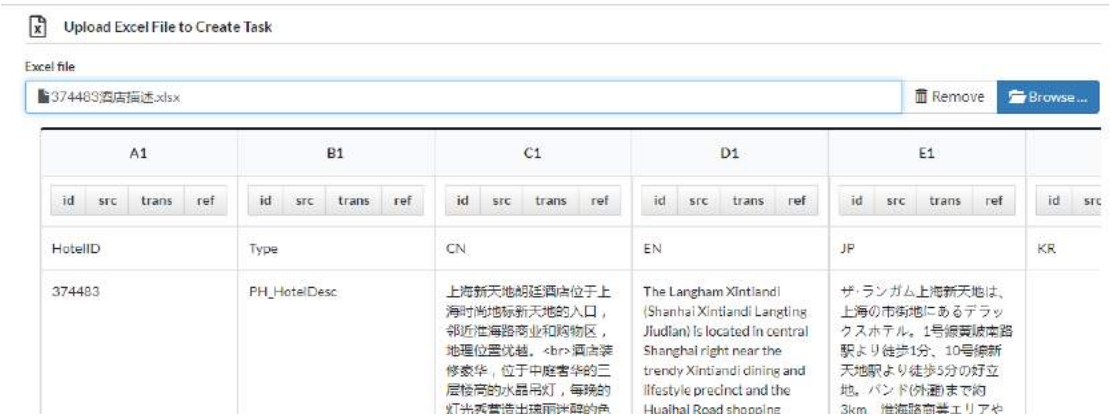


图 10 Excel 任务创建

Excel 在 V3 中不仅可以作为任务本身，也可以作为离线翻译的手段。对于酒店类型的常规翻译，通过 Excel 将任务导出/导入也可以看成是翻译流程的一部分。也就是说，任务翻译不仅可以通过在 V3 页面上进行，也能随时导出成 Excel 文件，通过操作后再次导入更新翻译的方式进行。工程实现上，为了支持 Excel 处理，V3 对 Apache POI（Excel 操作库）的流式读写、配置式内容读取等做了很多实用的工具类包装，方便了业务开发。

四、写在后面

CTran 是 IBU 国际化进程中诞生的数据处理方案。V3 是在深入理解了 IBU、携程数据的前提下，为改进业务流程，从存储层、核心实现到用户交互彻底重构的产品。

在紧缺的翻译及研发人力资源配置的情况下，V3 的功能设计是克制而务实的，力争合理用好每一分资源。架构上我们采用了很多的携程自研基础框架。有强大的公司技术作为后盾，研发工程师才能心无旁骛的投身于业务解决问题方案实现。

重新设计的 CTran V3 希望通过简洁直观的设计，展现工程师的细致与对效率的追求，让良好架构的产品与用户共同成长，让机器与人更好协作。

V3 于 2018 年 1 月 17 日上线至今，用户整体反馈良好，更令我们欣喜的是，翻译人员、运营团队、产品经理等用户与平台研发的互动越来越频繁，显然她们理解了 V3 的技术设计，依赖技术，能够对产品的不足提出自己的想法，帮助我们创造更有效率的工具。

国际业务内容研发团队所有产品的研发策略有别于其他业务类团队，从功能的设计取舍到技术架构的调整均由研发团队直接把控。我们尊重并积极收集各方反馈意见，自我驱动持续迭代，保证了项目的专业性，也鼓励研发人员对用户体验与业务意义的思考与主动发现。内容团队欢迎各方对我们的产品提出改进建议，支持我们的工作。

技术是驱动业务发展的核心力量，好的技术不仅以人为本，关心业务价值与生产率提升，还应该为用户的想象提供助力，为产品改进提供方向与信心。

高效开发与高性能并存的 UI 框架——携程 Flutter 实践

[作者简介] 段天章，携程支付中心 Android 端主力研发，目前主要负责中文版、国际版移动端 Android 支付模块研发工作。开源社区爱好者，热爱移动端新技术。

Flutter 已经开源了三年，但是最近两年才开始在开源社区活跃起来，尤其是最近还发布了 Preview 1 版本。作为可以实现一套代码同时在 iOS、Android 平台上运行的又一个新的 UI 框架，Flutter 提供给开发者的不仅仅是高速实现，还有高质量、流畅的 UI。免费开源的协议对于开发者来说也很友好。

本文将从 Flutter 架构理念与 UI 渲染逻辑，来解释为什么 Flutter 的渲染效率非常高，以及从 Flutter 开发实践的角度，介绍框架的特性及 Flutter 开发中所遇到的问题，希望给对 Flutter 感兴趣的小伙伴在选型时一些启发和思考，避免重复踩坑。

一、Flutter Layers

Flutter 的主要设计人之一 Ian Hickson，之前是 HTML 规范编写者，因此 Flutter 的设计理念也与 HTML 的实现方法有很多相似之处。

Flutter 最初的理念是实现跨平台的 Material Design 的跨平台框架。平台框架大致可以分为四层：

dart:ui：最底层的是 UI 层，由 Flutter 引擎所暴露的库，可以理解为一个布局层。
Rendering：这一层是抽象的布局层，它依赖于 UI 层，可以构建一个 UI 树，通过更新 UI 树来更新 UI。
Material 与 Widgets：最后就是 Material 层使用 Widget 层来构建 UI。

起初 Flutter 是没有 Rendering 层的，直接通过坐标计算每个像素点需要显示什么，这让框架的代码变得特别复杂，每当 UI 更新的时候需要重新计算这些坐标是否需要改变。后来增加 Randering 层来抽象 UI 显示的位置，通过抽象位置来判断像素点是否需要更新。

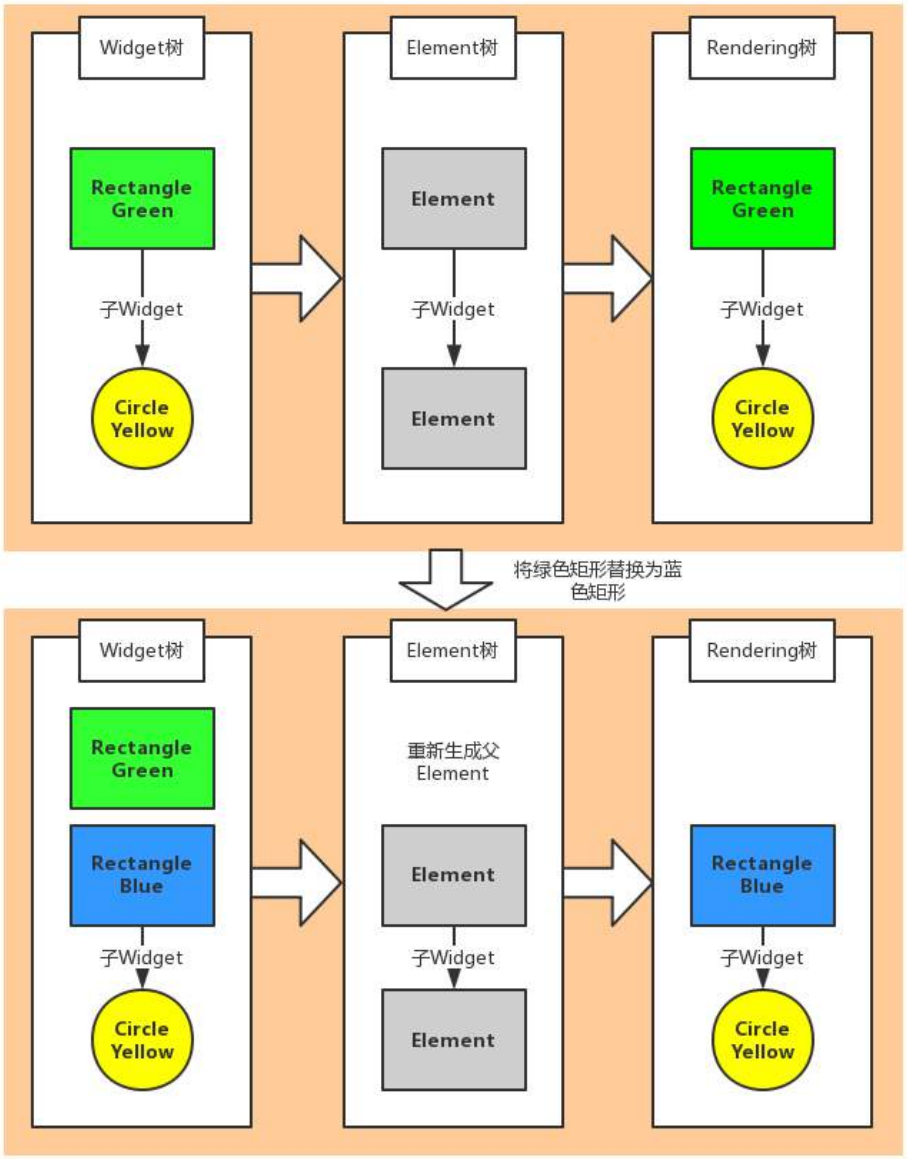
在 Flutter 项目的初期，Dart-lang 也不是特别成熟。Dart 虚拟机在垃圾回收的频率与回收机制表现当时并不是特别好，比如当时 Flutter 如果运行一个时间很长的动画，动画结束之后所占用的内存对于 Flutter 框架就是一个很大的垃圾。后来 Dart 团队在垃圾回收上进行了很多优化，使 Flutter 在 UI 显示更流畅。

如今，国内最大的使用厂商应该就是阿里闲鱼了，在 Flutter 发布 Preview 1 版本的时候，闲鱼 App 也一起协同展示了他们用 Flutter 编写的商品详情页面。我也在使用 Flutter 仿小米计算器开发后，体验到 release 版的流畅度确实堪比原生：

二、Flutter 的 UI 渲染

Flutter 渲染效率堪比原生，快于 RN。Flutter 更新 UI 的时候，并不是更新整个 UI，而是更新所需要更新的部分。比如从网络异步下载一个图片，设置到“Image”（ImageView）中，如果这个 Image Widget 大小并没有改变，只需要将图片对象传入 Widget 中，接着直接重新绘制这一个 Widget 就可以了。为了达到这样的 UI 渲染理念，Flutter 是如何设计的呢？

2.1 FlutterUI 渲染过程



Flutter 的 UI 渲染过程简单可以分为 3 个分支，Widget 树、Element 树、Rendering 树。

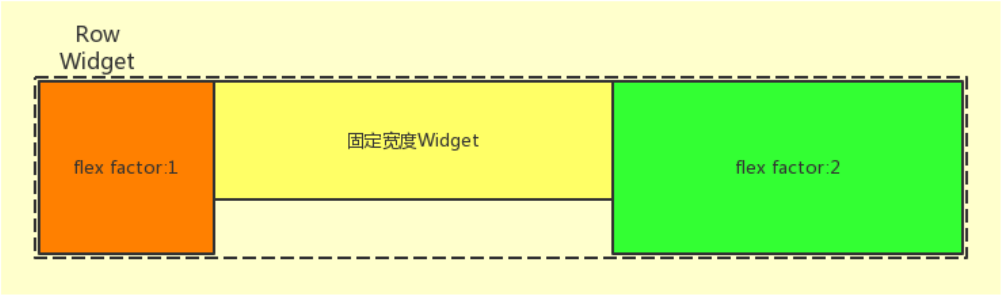
当 Widget 改变的时候，只有将它添加到 Element 树上时，才会改变 Rendering 树，展示到 UI 界面上。将它添加到 Element 树的方法就是 setState()方法，它会自动寻找改变了的 Widget，然后添加到 Element 树，等待后续的操作。

可以看到，矩形的子 Widget 并没有改变，所以在 Element 树上也没有改变，到了 Rendering 树也没有重新渲染，这种设计理念对于刷新 UI 操作可以大大提高效率。

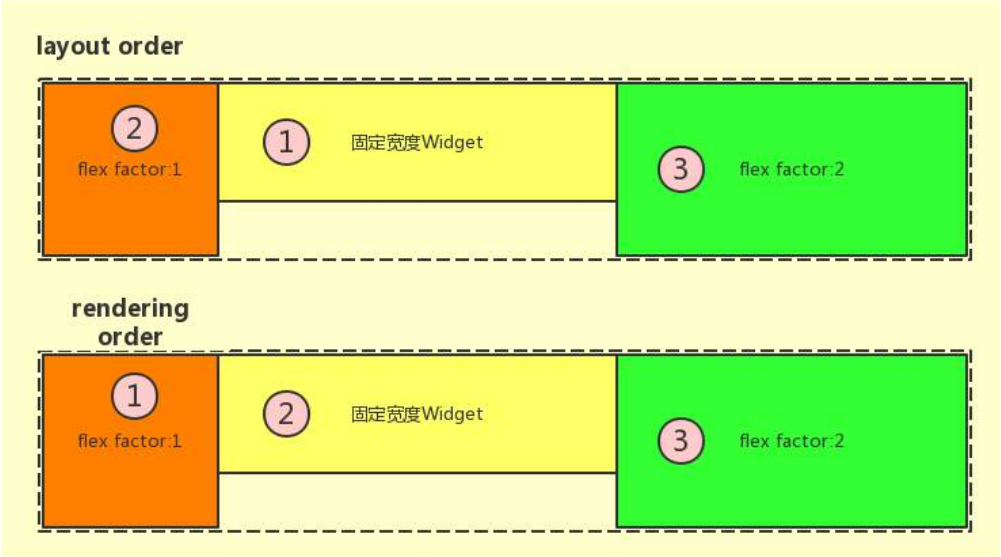
2.2 FlutterUI 渲染 —— onDraw 与 onLayout

与其他的 UI 框架渲染逻辑不同的是，Widget 的 Draw 与 Layout 的顺序不一定相同。比如在 Android 端 onDraw 与 onLayout 的顺序是相同的。关于 Flutter 框架的渲染顺序大家可以看

以下的例子：



在 Row Widget 中有三个子 Widget，其中中间的是固定宽度的 Widget，还有两个是根据剩下宽度比例占用位置的 Widget，其中绿色 Widget 是橙色的宽度的两倍。而他们的 layout order 与 rendering order 如下：



这么做是因为 Flutter 为了保证对于每个 Widget 的访问是单一线性的。所以在 layout order 中 Flutter 框架就会先 layout 固定宽度的 Widget，然后再 layout 比例宽度的 Widget。接着到了 Rendering 树再会根据 Element 树的顺序逐个对每个 Widget 进行渲染。

三、Flutter 框架 UI 特性

3.1 Dart 语言

Flutter 的开发语言是由 ChromeV8 引擎团队的领导者 Lars Bak 主持开发的 Dart。Dart 语言语法类似于 C。Dart 语言为了更好的适应 FlutterUI 框架，在内存分配和垃圾回收做了很多优化。

因为 Dart 在连续分配多个对象的时候，所需消耗的资源非常少。Dart 虚拟机可以快速分配

内存给短期生存的对象，这样可以使很复杂的 UI 在 60ms 内完成一帧的渲染（实际感觉每一帧渲染时间更短），这样就保证了 Flutter 可以平滑的展示 UI 滑动及动画等效果。Flutter 团队与 Dart 团队的密切合作让提升效率变得更加容易。

3.2 FlutterUI 开发样式

Flutter 在开发 UI 界面的时候，又比较像 HTML 的标签式语言，前文也提到，这是受 Flutter 创始人之一的 Ian Hickson 影响。其实很多 UI 布局都是类似标签的样式来编写的，比如 Android 的 XML 以及网页的 HTML，所以 Flutter 会采用这样一个成熟的布局开发样式。

```
new Scaffold(
        appBar: new AppBar(
          title: new Text(widget.title),
        ),
        body: new Center(
          child: new Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              new Text(
                'You have pushed the button this many times:',
              ),
              new Text(
                '$_counter',
                style: Theme.of(context).textTheme.display1,
              ),
              new FlatButton(
                color: Colors.blue,
                )
            ],
          ),
        ),
        floatingActionButton: new FloatingActionButton(
          onPressed: _incrementCounter,
          tooltip: 'Increment',
          child: new Icon(Icons.add),
        ),
      );
```

3.3 Flutter 插件、依赖与包管理器

Flutter 与 RN 一样，在原生开发中很依赖于插件来调用系统 API，毕竟它是一个 UI 框架。但是现阶段的 Flutter 插件并不是像 RN 那么全，可以看到维护 Flutter 的开发者只有 200 多人，而维护 react-native 的开发者已经近 1700 人了，一个数量级之差的维护者肯定在插件数量与开发体验上差别很大。

在包管理上，flutter 并不需要依赖第三方类似于 RN 的 npm 包管理器来添加依赖，flutter 本身就自带了包管理器，只需要在 pubspec.yaml 文件中添加相关依赖即可。但是，因为 Google 的库在国不能访问，需要添加环境变量指定库镜像才可以使用。

```
export PUB_HOSTED_URL=https://pub.flutter-io.cn
export FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.cn
```

3.4 Flutter 框架特性

在代码实现上，Flutter 并没有 Android 的 findViewById，页面布局是通过有状态 Widget（StatefulWidget）和无状态 Widget（StatelessWidget）实现的。顾名思义，无状态的 Widget 就是一些不可以改变的 UI，而需要改变的 UI 则是通过有状态的 Widget 来实现，并且通过 setStatus()来刷新 UI 的状态：

```
...
Text(
      '$_counter',
      style: Theme.of(context).textTheme.display1,
),
...
setState(() {
  _counter--;
});
```

这种方法很简单的实现了动态化的 UI 及 Android 长久以来希望达到的目标 —— data binding。

四、Flutter 待完善的方面及使用中遇到的问题

4.1 Flutter 至今没有反射

Dart 并不是没有反射，dart:mirrors 就具有 Mirror 概念的反射。在安全、分发、部署方面，Mirror-Base 具有很大优势。但是反射生成的代码冗长，会使 Flutter 编译过后的包很大。Flutter 通过将 Dart 编译成原生代码本身就会增加包大小，再加上反射的话包大小更会进一步扩大。所以 Flutter 团队在现阶段并没有开放 dart:mirrors 的使用。

没有反射也就意味着 Json String to Model 也没有办法完成，对于这一点，官方也比较无奈。至今 Flutter 中 Dart 只支持将 JsonString 转化为 Map，然后再由开发者手写代码将 key 值一一对应到相应的字段上。

```
/**
      "result": {
      "status": "ALREADY",
      "scur": "CNY",
```

```
    "tcur": "EUR",
    "ratenm": "人民币/欧元",
    "rate": "0.127839",
    "update": "2018-07-13 23:28:01"
    }
 */

///
class ExchangeResult {
  final String status;
  final String scur;
  final String tcur;
  final String ratem;
  final String rate;
  final String update;

  ExchangeResult(this.status, this.scur, this.tcur, this.ratem, this.rate,
      this.update,);

  ExchangeResult.fromJson(Map<String, dynamic> json)
      : status = json['status'],
        scur = json['scur'],
        tcur = json['tcur'],
        ratem = json['ratem'],
        rate = json['rate'],
        update = json['update'];

}
    Map exchangeMap = json.decode(Utf8Codec().decode(response.bodyBytes));
    var resultModel = new ExchangeResult.fromJson(exchangeMap);
```

4.2 Dart-langhttp 请求 response 解码问题

Http 请求返回的 response 中 Header 会包含编码格式 charset=utf-8，官方给出的 Demo 如下：

```
 Var dataURL = "http://api.k780.com?app=finance.rate&scur=CNY&tcur=GBP&appkey=35134&sign=fb020c3129435bb5ff21b7113e9cb1c1&format=json";
    var response = await http.get(dataURL);
    print(response.body);
```

看起来是非常简单的实现了异步请求服务，但是如果返回的 charset 后面多加了一个";"的话(charset=utf-8;)，http client 就不会自动根据 header 中的 charset 解析，会返回错误：

[ERROR:topaz/lib/tonic/logging/dart_error.cc(16)] Unhandled exception:
Error on line 1, column 33: Invalid media type: expected

所以，如果要解析返回的 json string，必须要指定 UTF8 字符解析 response 才可以：

print(Utf8Codec().decode(response.bodyBytes));

4.3 Flutter 并不能指定 Dart lang version

安装 Flutter 的同时也会安装 Dart lang SDK，集成在 Flutter 的 SDK 中的 $FLUTTER_SDK/bin/cache/dart-sdk。假如你发现一个 Dart lang bug，那就需要更改 DartSDK 的代码，但是这个修正并不能让你马上使用。因为 Flutter 与 Dart lang SDK 的 version 是一一绑定好的。

五、总结

Flutter 虽然在现阶段问题比较多，但是相对于 RN 也有自身的优势。

在性能方面，Flutter 的表现比 RN 更为优秀。Flutter 也可以与原生混编，不过 Flutter 项目在编译过后生成的安装包相对于原生开发的项目来说会有所增大，相信这是 Flutter 团队今后要解决的一大难题。

不过随着 google 与开源社区的不断支持，相信 Flutter 在跨平台移动应用开发中将成为一种新趋势。

携程度假起价引擎架构演变

[作者简介] 陈少伟，携程度假研发部资深开发工程师，主要负责度假起价引擎的研发工作，喜欢钻研技术，对新技术有浓厚的兴趣。

背景介绍

携程度假为客户提供了非常丰富的旅游线路，每个旅游线路涉及到不同的出发地，不同的出发地下有不同可出发班期，每个班期都有对应的这一天的价格。旅游产品的价格由多个资源组成的，任何一个资源价格发生变化，都会影响到产品的价格。

为尽快捕捉到价格变化，我们不断优化调整架构，使得价格越来越准确，计算越来越快，同时也对被调服务及硬件产生了极大的压力，也带来了新的瓶颈。为了解决瓶颈再进行新的架构调整，如此周而复始，架构不断演变，每个版本的优化都带来准确率、计算速度的跃迁。

本文将按照我们主要架构调整的版本（引擎 1.0、引擎 2.0、引擎 3.0）来介绍度假起价引擎的架构演变过程，以及过程中踩过的坑及优化思路，希望通过该分享，可以给公司同类项目以及行业同类需求提供一些启发和借鉴。

一、业务范围及名词解释

业务范围：如下图，引擎计算的价格包含“产品起价“和“班期起价”，这一部分都是离线计算的。

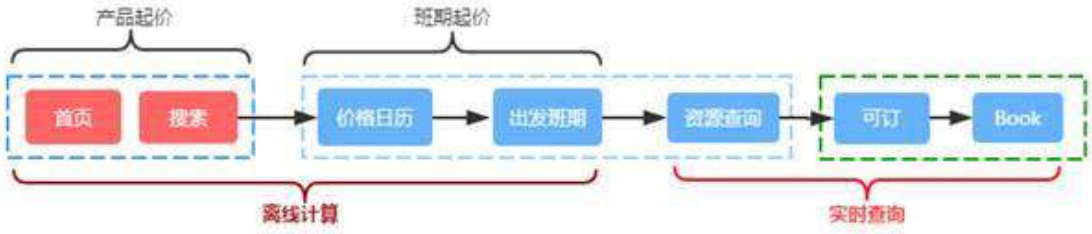


图 1 度假起价引擎业务范围

任务单元：分以为下两种，

(i) 资源任务单元(以下简称为资源任务)，指定产品、出发地、日期下的某一类资源价格的计算过程，存在于引擎 2.0、引擎 3.0 中，一个班期任务单元可分为多个资源任务单元；

(ii) 班期任务单元(以下简称为班期任务)：指定产品 ID，出发地，出发日期下多个资源任务单元的组合，存在于引擎 1.0、引擎 2.0、引擎 3.0 中；

班期价格：指的是一个产品、出发地、出发日期对应的单人推荐价格，一个单人价格的组成则是该产品、出发地、出发日期下不同资源单人价格的加和。涉及的资源可能有：机票(大系

统机票、度假采购机票)、酒店(系统酒店、手工酒店)、保险、邮轮、玩乐、门票、优惠等。

任务量：产品、出发地、出发日期 3 个维度相乘即为总任务量，引擎 2.0、引擎 3.0 又把一个班期任务单元拆分为多个资源,平均一个班期任务单元可拆分为 2 到 3 个资源任务单元。引擎 1.0、引擎 2.0、引擎 3.0 的任务量大概如下：

(i) 引擎 1.0：班期任务数接近 3000W 左右。
(ii) 引擎 2.0：班期任务数在 6000W 左右。
(iii)引擎 3.0：班期任务数在 54000W 左右；

限流：为保持系统的稳定运行而采取的自我限制手段，按限流方式分为两种：

(i) 外部限流（集群限流）：无限制调用可能会导致内部或者外部接口崩溃，接口的访问量由外部接口统计并提供，超过一定额度即不再调用，当前线程休眠，以分钟为单位；

(ii) 内部限流（单机限流）：经过引擎 2.0，3.0 的优化后，多个任务可合并为一个接口调用，当合并的量级越大，接口压力越小，DB（sqlserver,mysql,hbase 等）的更新压力越大，在某些情况下会对 DB 产生很大冲击，因此而做的内部自我限制，使得对 DB 的调用更为平稳，以秒为单位；

准确率:如果引擎计算出来的价格和用户实际访问的价格差异在一个限定区间内则认为价格是计算准确的，否则是不准确的；

引擎模块：

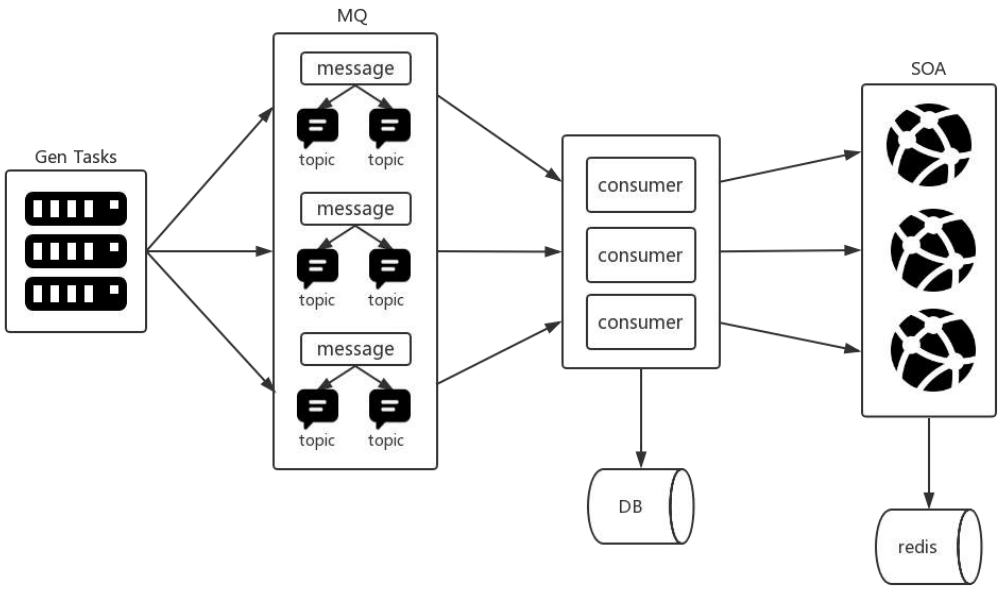


图 2 引擎模块

二、系统的核心和难点

核心：引擎的主要工作就是计算产品班期的价格，旅游产品的价格由多个资源组成的，任何一个资源价格发生变化，都会影响到产品的价格。尽快捕捉到资源价格变化，并准确体现到产品价格上是引擎的核心指标。

难点：随着业务发展，任务量不断上涨，在接口调用量被限的情况下(主要是外部门接口因业务原因需要限流)，更准确及时的更新产品价格。

三、引擎 1.0

总任务量 3000W 左右，只有班期任务，单个班期任务内各个资源串行计算，汇总为一个班期价格并更新到 MYSQL 数据库，同时把 MYSQL 价格相关数据通过 JOB 的方式同步到 SQL SERVER 数据库，生产的数据源为 SQL SERVER。任务队列的载体为 redis list,sortedset(.net 封装)；

主要存在问题：

(i) 任务队列：使用 redissortedset(.net 封装)做为优先队列来使用，当数据量比较大时，对 sortedset 的调用量达到 1.2W/min 时，会偶发出现网络异常，随着业务增长，数据量、调用频率也会同步上升，问题可能会不断放大。

(ii) 任务生成：任务生成部分涉及到了分组聚合排序，单机方式生成任务信息后发送到相关的队列中。在 3000W 任务的情况下暂无压力，随着产品数量、班期任务的增加，单机分组聚合排序是有一定风险的：

a、随着任务增长，会有内存瓶颈；
b、单机执行任务生成速度慢；
c、存在单点故障。

鉴于以上的问题，我们着手进行引擎 2.0 的改造。

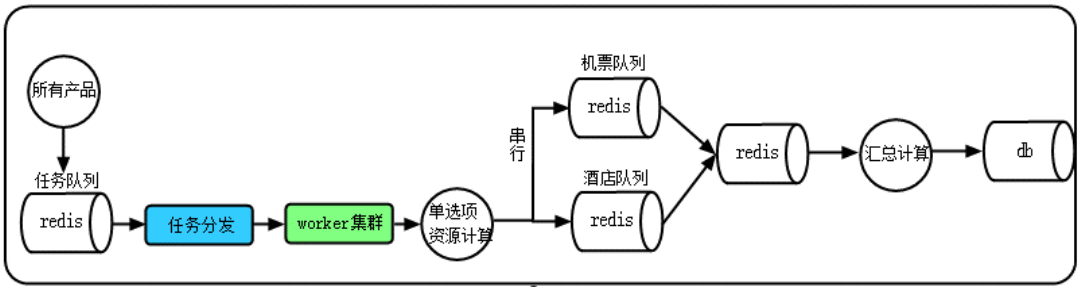


图 3 引擎 1.0

四、引擎 2.0

为解决引擎 1.0 中存在的问题，我们进行了以下的优化。

4.1 使用 Hermes 替换 Redis

消息队列的选型，公司内部有使用的消息队列类型有 Rabblitmq、CMessaging（内部框架）、Hermes(内部框架)等，结合业务场景、使用量以及运维成熟度等多方面考量，选择了 Hermes。

4.2 任务生成优化

班期量从 3000W 增加到 6000W，单机生成任务瓶颈明显，改进方案：

(i) 单机生成改成集群生成
(ii) 使用 spark 集群进行分组、排序、聚合并发送消息

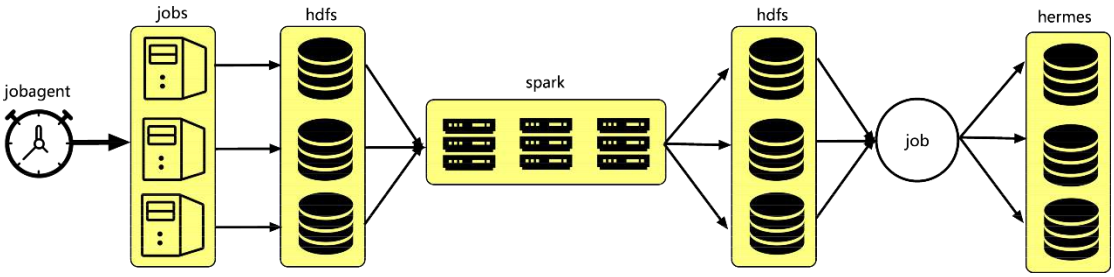


图 4 生成优化

4.3 任务计算优化

随着业务的发展，任务量从 3000W 至 13000W，资源价格汇总存在明显瓶颈，通过按资源计算的方式加以优化，如下图：

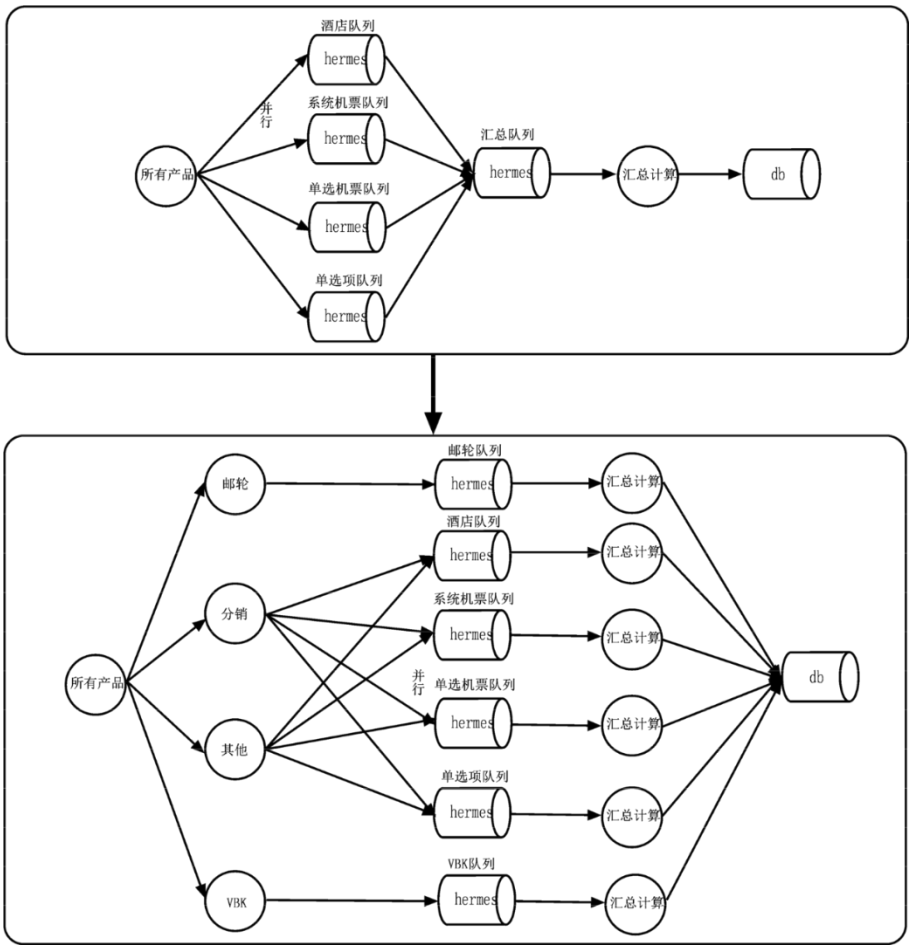


图 5 任务计算优化过程

由上图我们可以看到，原先所有资源计算完之后扔到同一个队列进行处理，任务量小的时候问题还不大，任务量大的时候就成了瓶颈。去掉资源汇总队列由各个队列直接处理，使得瓶颈不再存在，并提高了计算的效率。同时对不同资源进行解耦，各资源的计算频率可灵活控制。

同时，各个资源解耦的情况下方便对各自资源特点做定向优化。如下:

(i) 机票计算优化：机票由于外部接口业务原因进行限流，一轮基本需要 1 个星期才可以算完，价格严重不准，原因在于任务量大，而机票限流，所以我们并不能无限制的调用机票接口获取价格。

后来我们仔细的分析了国际机票的请求，发现机票请求的主要因素为出发日期、出发地、目的地，不同的产品对应的机票的规则虽然可能都不一样，但是对于同样出发日期、出发地、目的地来说对机票的请求却是一样的。

国际机票总任务 1800W，但经过这样一处理，不重复的任务数却只有 350W，这样相当于任务数压缩成不到原任务的 1/5，而这个思路同样可以复用到国内机票(1/10)。

所以我们针对航线数据做了一个反向索引，以航线为 key，不同的产品做为 value，这样不

同的产品但是同个出发到达机场及时间可以命中同一条航线的索引，通过索引可以减少大量重复的机票请求。

既节省了对机票接口的调用量，又使机票资源的计算速度大大提升，原先需要 3 天计算的任务，现在 1 天内可以完成，并且航线(出发日期、出发地、目的地)的组合是相对固定的，也就是说随着任务数的增加，航线的增加却相对要比任务数增加要缓慢得多，这样实际请求数是比较可控的。

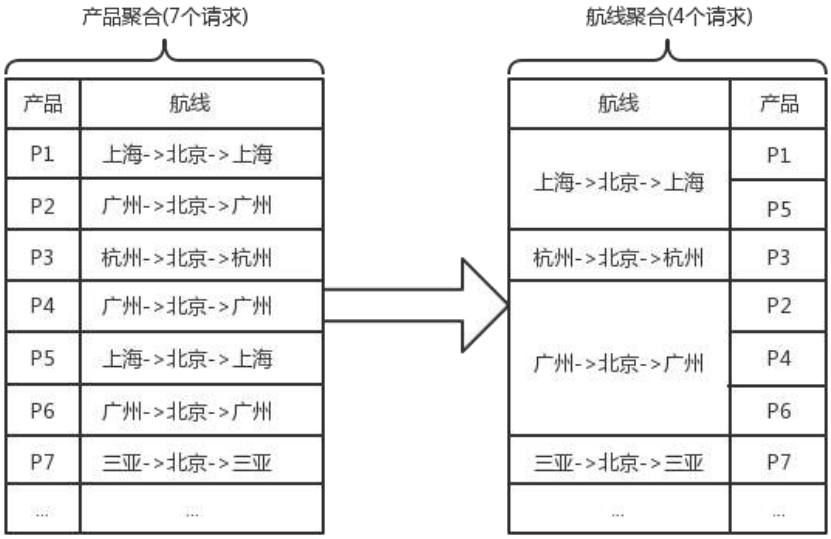


图 6 航线聚合

(ii) 酒店计算优化：通过了解发现，酒店资源的价格只跟目的地有关系，跟出发地是没有关系的，而原来引擎计算酒店的方式是按一个产品、出发地、出发日期来一个一个计算的，极大浪费了酒店资源的流量，平均一个产品有 10 个出发地，不同的产品也可能为同一个目的地，所以原来的计算方式可以根据目的地来进行聚合，聚合后对接口的调用量减少到了不足原来的十分之一，计算时间也由原来的 2 天到现在的 8.5 小时。

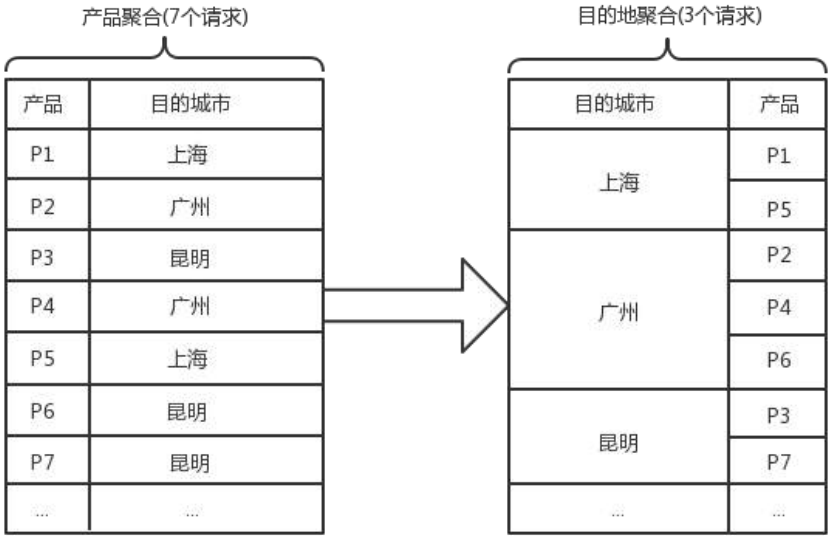


图 7 目的地聚合

(iii)VBK 计算优化：不同出发地配的资源基本都是一样的，但是按现有的引擎计算流程，是按单出发地进行计算的，那么有多少出发地，相同的资源价格就会被计算多少次，其实是一种重复计算。

把 VBK 产品拎出来单独处理，引擎的其他流程计算排除 VBK 产品，VBK 产品的引擎计算以班期为单位，计算一个班期的价格推广到 N 个出发地，减少重复请求，并且 VBK 产品的库存、价格变动通过消息通过引擎进行价格更新。

如果按以前的大流程计算，VBK 计算到一轮可能差不多得 3 天时间，而现在 VBK 产品计算可以计算 4 次，VBK 产品现有班期总数大概有 3000W 多，占总引擎班期数近一半，该类型产品的速度优化对引擎的价格新鲜度有重要意义。

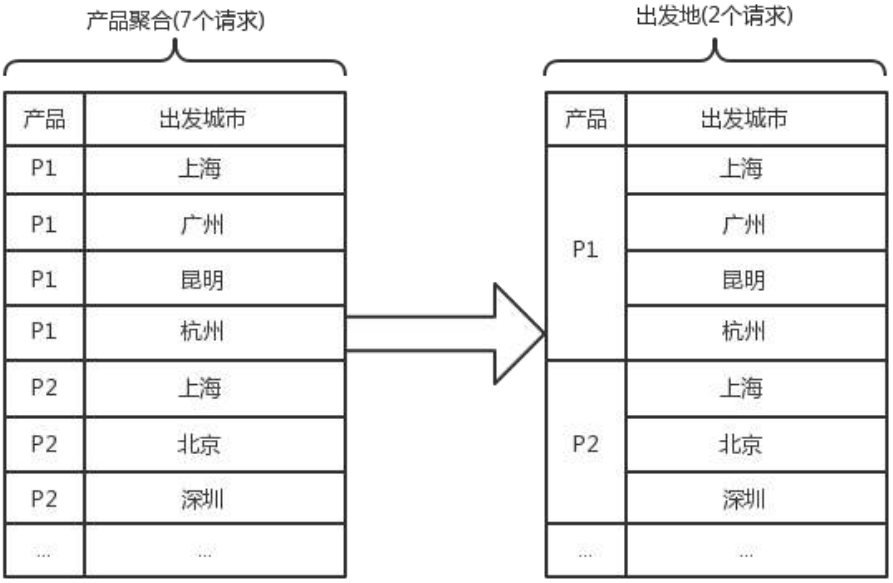


图 8 出发地聚合

4.4 总结

引擎 2.0 优化后的效果：
(i) 任务生成速度：5 小时至 1.25 小时；
(ii) 任务计算周期：2 周至 1.5 天

五、引擎 3.0

随着任务量增加，班期数从 6000W 增加到 54000W，系统面临如下新的瓶颈：

(i)MYSQL 数据库存在 IO 瓶颈
(ii) 任务生成后消息分发不及时

针对以上两个问题，我们启动 3.0 改造计划。

5.1 使用 HBase 替换 mysql

随着计算量的增加，MYSQL 数据写入存在 IO 瓶颈，更换 SSD、分库分表后依然未彻底解决问题，考虑到引擎使用 DB 如下场景：写入多、查询少，查询条件简单，再加上我们是离线计算方式计算价格，比较适合 NOSQL 数据库的适用场景，在综合评估 HBase 的各项指标、开发成本、运维成熟度之后，基本符合我们的预期。

价格是引擎的核心指标，如果某次发布导致价格计算错误将产生严重的后果，为了防止最严重的后果出现，我们利用 HBASE 的多版本机制，当某次发布导致计算价格不对，可指定版本立即回滚回正确的上一版数据。

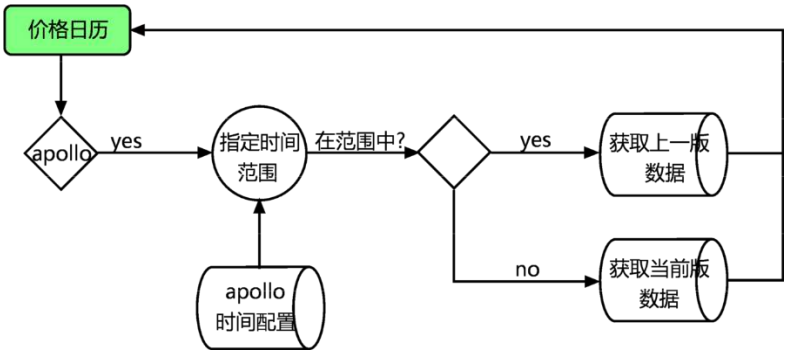


图 9 价格快速恢复

5.2 任务生成优化

之前任务信息是以 HDFS 文件的方式存储，通过 JOB 方式读取并发送消息，存在问题：

a、单机消息发送慢，需要 4 小时发送完；
b、任务发送过程无法发布。

解决方案：通过 spark 在生成完任务就通过各个节点把消息发到各个资源队列，速度快且不影响发布。

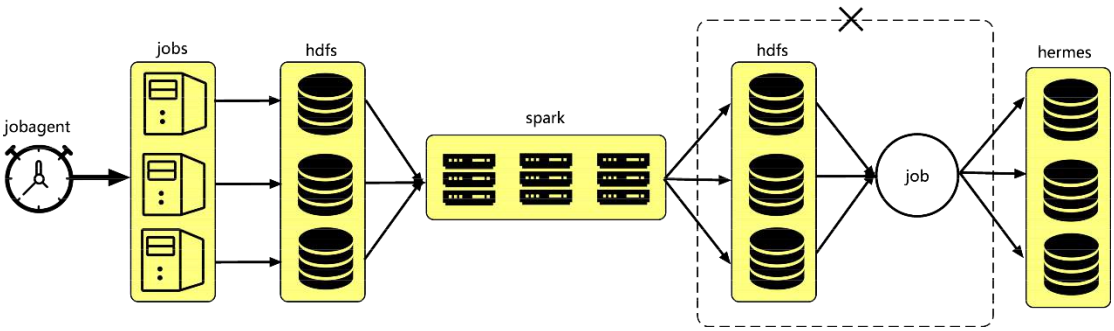


图 10 引擎 3.0

5.3 热门任务策略优化

热门任务的策略原先是按过去三天访问过班期次数>3，按产品、出发地聚合任务并提升计算频率，按 1 天计算 4 次，总任务量大概是 2000W 左右。但按这个策略实际效果也并没有达到预期，因为过去 3 天访问量高的班期并不意味着这些班期的价格是不准确的，由此我们做如下优化：

(i) 策略调整：把过去三天访问过的价格差异超过过大的班期按产品、出发地聚合任务提升频率计算，1 天计算 3 次，总任务量大概是 500W 左右，完成优化后，计算的准确率下限提升了 2%左右；

(ii) 分批加载优化：通过数据采集，并对价格差异过大的产品做分析发现，机票、酒店所占价格差异超过 10%的比例是最高的前 2 位，其中又以机票占比最高，进一步对机票的分析发

现，机票很多的价格差异是由于分批加载导致的，因此如果可以减少分批加载造成的价格差异的话，那么必然可以提升起价的准确率。

(iii)即将要做的：按(i)优化的方式对用户的访问覆盖面并不高，实际上确有一定的局限性，假如用户的目的地是厦门，那么按(i)的优化方式只会去跑用户访问过的产品，但不代表用户不会去访问其他到厦门的产品，根据我们对数据的分析，如果按出发地、班期、目的地为维度进行聚合的话，计算对用户访问的覆盖面还将增大一倍；

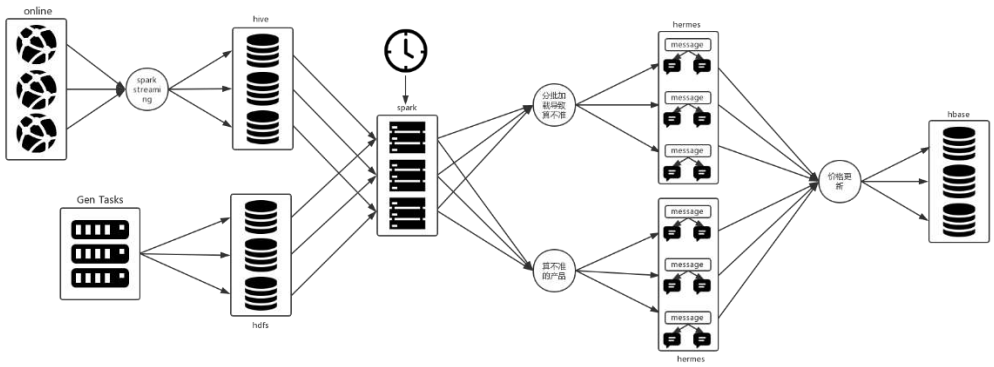


图 11 热门任务策略优化

六、优化结果

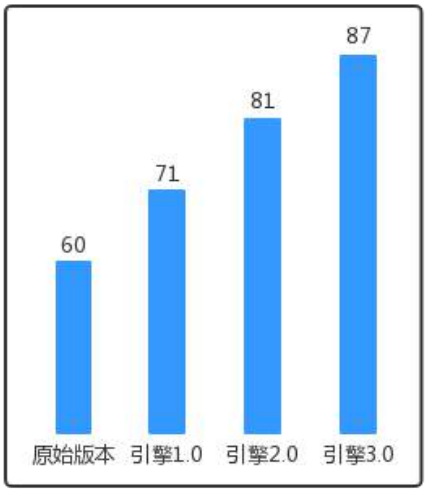


图 12 引擎各版本准确率

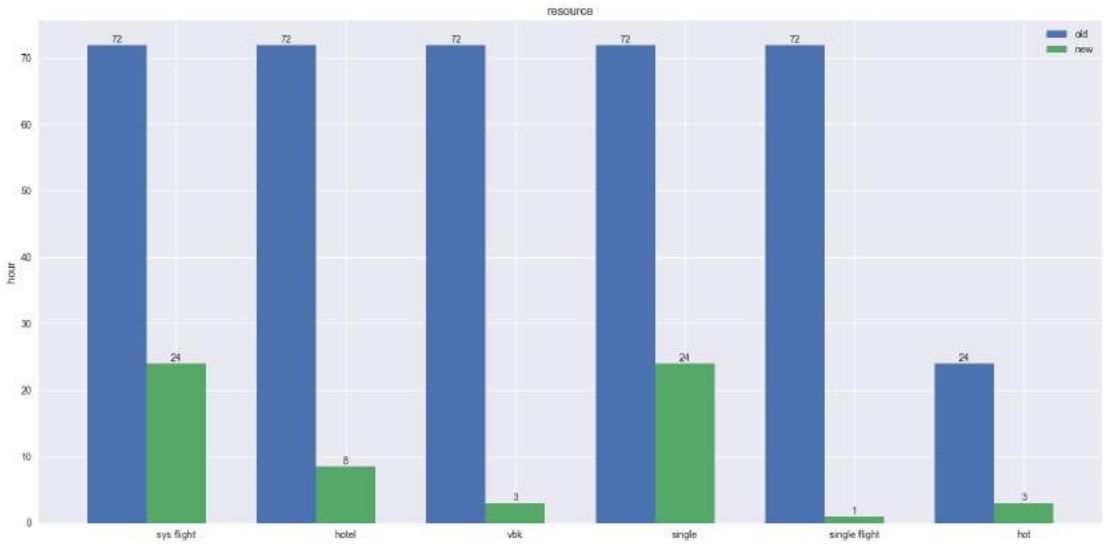


图 13 计算时间对比

七、总结

综上，以上的很多优化并不是孤立的，有很多优化都是并行着进行的。

一个优化做完之后，计算效率、准确率提升的同时，也带来了计算速度的增加，计算速度的增加同样会增加被调服务的调用次数，不管是对自身系统还是对被调服务各方面压力都是增加。

因此整个的架构确实是演化出来的，而并不是事先设计好的。架构是为业务服务的，业务上的需求决定了架构演化的方向。

携程国际 BU 的 SEO 重构实践

[作者简介] 熊聘，携程国际事业部公共研发团队 Leader，目前主要负责国际化相关的基础组件和市场相关项目的研发。开源社区爱好者，喜欢阅读优秀的开源项目源码，对新技术有着深厚的兴趣。

最近一段时间一直在对原有的 SEO 项目进行重构，目前已经进入重构的后期阶段，想和大家一起分享一下整个重构的细节，希望对大家有所帮助。

一、什么是 SEO 项目

SEO（Search EngineOptimization），搜索引擎优化，利用搜索引擎（目前主要指 Google）的规则提高网站在搜索引擎内的自然排名和网站的品牌影响力。

用户在搜索引擎上搜索相应的关键字，点击搜索结果直接跳转到 SEO 的着陆页（Landing Pages），然后通过 Landing Pages 将流量引到需要推广的网站，从而将这些流量转化成订单 。

SEO 项目主要是根据不同的推广维度设计相应的 Landing Pages，并为这些 Landing Pages 提供相应的数据，目前该项目主要涵盖携程酒店和机票两大产线，后续可能会接入更多的产线。

二、为什么要重构

原 SEO 项目主要存在以下几个方面的问题：

代码耦合：前端代码和服务端代码全部耦合在同个项目里面，在开发过程中相互依赖，页面信息模块化，可配置化，支持 AB 测试等一系列的功能都很难实现。

数据存储：SEO 项目的数据和之前的其它系统存储在同一个 DB 中，并且部分数据表是共用的，必然导致某些表中的字段从 SEO 项目的角度来看是无用的但又不能去掉。

数据更新：数据全部更新完一次约 2-3 天，整个过程需要人工干预，如果更新过程中出现了任何问题需要重新进行全量更新，并且还存在脏数据 ，主要分为两类：一类是数据表中某个字段的值部分是正确的，部分是不正确的；另一类是数据不完整，比如：如果某个城市没有任何酒店或者机场，则这条城市数据是没有意义的，因为在做城市维度推广的时候，这个城市下面是没有任何酒店或者机场的数据的。

需求无法满足：在 SEO 页面的底部需要根据一定的规则计算相关的链接信息，计算某一个站点的某一个产线在某一种特定语种下需要的时间约为 4 小时，现有 16 个站点，每个站点有 15 种语种和有 3 个产线，计算出所有站点下所有语种和产线的链接信息需要的时间为 16*15*3*4=2880 小时（120 天），显然目前的实现方案是无法满足业务需求的。

为什么会存在以上这些问题？主要原因如下：

需求迭代太快：IBU 一直处于高速发展的状态，很多需求都是需要在很短的时间，快速的完成，并且对未来需求的变化很难把握，在做需求的过程中难免会选择一些短期的，尝试性的，快速见效的方案。

开发人员少：在重构之前整个 SEO 项目仅 1-2 个人来完成所有的开发工作，当需求源源不断涌现时，开发人员难免会措手不及，应接不暇。

数据复杂：目前 SEO 几乎需要与机票和酒店相关的所有数据，而这些数据的收集过程又是极其分散、复杂和繁锁的，收集某条数据时可能需要采集多个数据源的数据才能将这条数据中的所有字段补全，并且数据量大导致更新时间长，数据之间的关联性高从而导致数据更新过程中加大了保证数据完整性的难度。

三、技术选型

项目的技术选型对整个项目来说至关重要，这里主要表述的是服务端的技术选型。

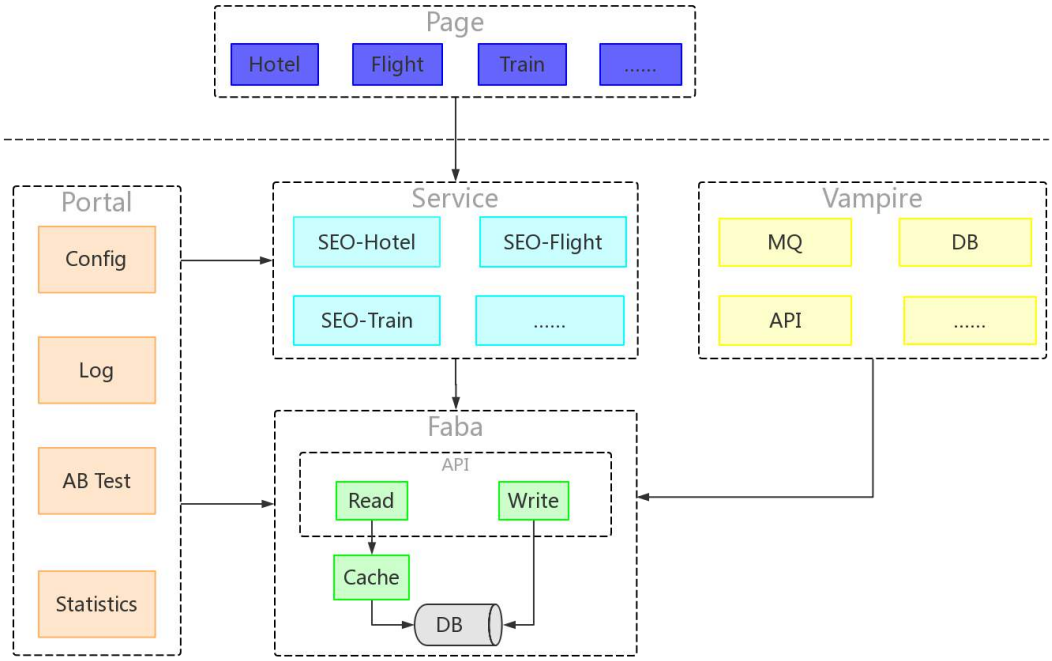
开发语言：去掉了之前的部分代码采用 Java 实现，部分代码采用 PHP 实现的方案，将开发语言统一定为 Java。主要原因是公司主推 Java 语言，团队中所有成员最熟练掌握的编程语言都是 Java。

数据存储：在数据存储方面主要采用 MySQL 数据库，去掉了之前一部分数据采用 ES 存储，另一部分数据采用 MySQL 存储的方案。SEO 项目中酒店的数据量最大，也仅千万级，对于 MySQL 来说是完全没有问题的。

RPC 框架：公司提供了两种对外暴露服务的方式，一种是通过 Baiji 契约实现的，另一种是 CDubbo。未选择后者的主要原因是当时刚推出来，在稳定性上可能会略差于前者，同时整个团队对前者的理解更深入，使用的也多一些，降低学习成本。

四、设计方案

SEO 项目的整体架构如下图所示：



4.1 Vampire

主要是用来采集数据并转换成格式化的数据。采集数据的方式主要有增量和全量两种，数据来源可以是 MQ、DB 和 API（后面还会接入更多的数据源）。其核心思想是通过并发的方式拉取来自不同数据源的数据并将这些数据进行转换成格式化的数据，然后调用 Faba 的 Write 接口将数据写入 DB 中。

由于全量数据的数据量较大，所以在整个过程中拉取全量数据最为复杂。

从目前来看更新全量数据绝大多数情况是采用调用 API 的方式，需要考虑被调用 API 的 QPS、响应时间、更新一次的时间间隔、API 的返回报文大小（有些情况需要考虑分页）、API 的超时时间、Gateway 超时时间、网络带宽、数据之间的依赖关系等，从而确定 Vampire 在调用 API 时的线程数 、调用频次、调用周期、调用时间（一般在非高峰期调用）、部署时的机器数量、虚拟机的 CPU 核数和内存大小等，针对调用不同的 API 需要对不同参数进行优化。

增量更新相对而言简单一些，主要采用 MQ 的对接方式，需要考虑先发送的消息后到达，后发送的消息先到达的情况、重复消息、消息丢失、MQ 中队列的大小等。在整个拉取数据的过程中还需要考虑数据提供方可能出现脏数据或者无法支撑 Vampire 带来的流量，因此还需要支持暂停、恢复、强制更新等功能。

无论是增量还是全量的方式拉取数据，最后都需要转换成格式化的数据并写入 DB，这个转换过程的处理速度至关重要，因为 Vampire 从整体上来看其实是一个生产者和消费者模型，生产者是接入的各种不同数据源，而消费者则是将拉取的数据进行转化然后调用 Faba 提供的写接口，快速完成数据的转换工作。

理想情况下应该是生产者的生产速度等于消费者的消费速度，当生产速度大于消费速度时，生产出来未被来得及消费的数据就会囤积在内存中，容易造成 OOM，所以在实际使用的时候一般是消费速度大于生产速度。

而对于 Vampire 而言，生产者的速度是接入各数据源的流量之和，随着数据源的增加而增加，但是消费者的消费能力是固定的，所以要想提高整个数据采集和转化的吞吐量，本质上是要提高消费者的速度，也就是提高 Faba 的 Write 接口的速度（后面会详细讲解 Faba 处理数据的机制）。

目前生产环境部署了 4 台 8 核 8G 的虚拟机，Vampire 的处理能力可以达到每秒 10K+，处理 1000W 条数据耗时约 30min。

4.2 Faba

该子项目主要是为整个 SEO 项目提供数据 Read 和 Write 操作。其中 Write 接口主要由 Vampire 调用，用来补充数据，Vampire 将采集到并转换好的数据通过调用 Faba 的 Write 接口将数据写入 DB，Read 接口主要是对外提供访问数据的方式，由 Service 来调用。

Write 接口主要是采用异步的方式实现的，Vampire 在调用时会将数据先暂存到一个消息队列中，然后再来消费这些数据，这样处理的好处在于：首先提高了 Write 接口的 QPS 和响应时间，其次可以将一些相同的操作进行合并成批量的操作，从而尽量减少 DB 连接数的消耗，最后可以在写入时尽可能的对一个批次的写入的数据进行去重，减少不必要的写入操作。

Write 接口的设计需要考虑三个方面的因素：

第一、支持幂等。因为写入的数据来源于消息队列，消息队列会有重试的机制，所以在写入的时候需要支持幂等。

其实消息队列也不能保证数据是有序到达的，数据是否有序到达仅对增量拉取数据有影响，对于全量拉取数据没有影响，因为在全量拉取数据时，每条数据当且仅当只会被拉取一次，所以对每条数据的更新操作是相互独立的无需考虑先后顺序。

对于增量拉取数据而言，假设一条城市数据在同一时刻先后将城市名称从 A 修改到 B，再从 B 修改到 C，这两条更新的操作会被有序的推送到 Vampire，然后再由 Vampire 转换成格式化数据后调用 Faba 的 Write 接口，从消息队列中消费这两条数据时可能会先收到城市名称从 B 修改到 C 的数据，后收到从 A 修改到 B 的数据，这时会以两条数据发生修改的时间做为时间戳，在 DB 中更新数据时只更新当前时间戳大于这条数据在 DB 中的更新时间，其余的全部过滤掉，也就是城市名称从 B 修改到 C 的数据会被更新到 DB，从 A 修改到 B 的数据会被过滤掉。

第二、消费速率。很容易看出在整个写入的过程中的瓶颈是 DB 的写操作，公司 DB 的连接池大小是 100，也就是说通过多线程来消费消息队列中的数据，线程池的大小不要超过 100，确定了消费者的消费能力，生产者的生产能力只需要通过简单的计算就可以确定了，理论上只需要将生产者单位时间内生产数据的总量等于消费者线程数 100*每个批次内数据平均条

数，这只是一个理想的情况。

实际情况可能还需要考虑三个因素：

1）消息队列的大小，也就是囤积数据的能力，这个与机器内存有关；

2）可接受的数据延时时间，也就是一条数据从进入消息队列到写入 DB 的时间；

3）IO 的处理能力，往 DB 中写数据会产生大量的 IO 操作，特别是在进行批量写入操作时，之前由于这个因素没有考虑到，导致和 SEO 的 DB 在同一台物理机器上的其它 DB 的以前正常的读写操作出现大量的超时告警。

第三、数据优先级，Vampire 会从不同的数据源来拉取数据，不同的数据源会提供某一条数据中的若干个字段，不同的数据源的数据质量也会有所不一样，也就是不同数据源对同一条数据中的若干个字段有不同的优先级，优先级高的数据质量高，这个优先级是在接入数据源时定义的，所以在更新数据时还需要根据数据的优先级来判断数据是否更新，目前同一条数据的同一个字段的数据源只有一个，所以可以先不考虑这方面的问题。

Write 接口的性能

Read 接口目前主要是从 DB 中读取数据，其性能主要取决于以下两个方面的因素：

第一、数据库表结构的设计，在设计时尽量减少数据的冗余，将原来的每张数据表垂直拆分成多张数据表，根据业务需求建立好索引，让每一条查询的 SQL 语句都走索引，对于复杂的 SQL 查询，拆分成多条简单的 SQL，然后让每条简单的 SQL 都命中索引，并且将这些简单的 SQL 尽可能的复用，如果某一条 SQL 查询出来的结果会比较大需要分页，这时会通过对 SQL 的执行进行解析，确定出合理的页大小，对于复杂查询和分页查询多数据情况下都是通过执行多条简单的 SQL 将返回的结果通过程序组装的方式完成的。

第二、接口的设计，对外的 Read 接口在设计时也是尽量的简单，这里的简单包括入参简单和返回值简单。

入参简单指的是调用接口传入的参数尽可能的少且传入的每个参数都是必要的，例如：某一个接口有 A、B 和 C 三个参数，假设通过 A 和 C 这两个参数可以间接推导出 B 这个参数，这时 B 这个参数就是没有必要的，应该去掉；

返回值简单指的是返回的报文不要太多，在设计时一般小于 4KB，同时返回的报文中的数据字段都是有用的。

在整套接口拆分的过程中还需要考虑两个重要的因素：

1）所有接口通过若干次的组合调用是否可以获取 DB 中的所有有用数据；
2）完成一个特定的功能需要调用多个简单接口的次数尽可能的少，尽量多调用响应快的接口，少调用响应略慢的接口 。

在单机 4 核 4G，Tomcat 连接数 200，DB 连接数 100 的环境下，数据量为 1KW+时，Read 接口直接访问数据库，不走缓存，对于简单的查询 QPS 最高可以达到 1400+，对于复杂的多条件分页查询 QPS 最低可达到 400+

Faba 中的缓存分为本地缓存和分布式缓存两种。

对于本地缓存主要存储一些数据体量小，访问频次高，数据不一致性要求低的数据；分布式缓存主要是通过 Redis 作为载体来实现的，存储一些数据体量相对较大，value 小，访问频次高的数据。

同时在缓存数据时对数据量小的数据尽量做到全量缓存，定期更新，对于数据量大的数据采用 LRU 淘汰策略来更新缓存，在缓存空间固定的情况下，提高缓存命中率。由于根据目前的需求来看仅通过直连 DB 的方式达到的 QPS 已经可以满足了，所以开发缓存的优先级较低，目前还在开发过程，接口性能方面的数据暂时还不能给出。

4.3 Service

根据对业务需求的分析发现，每个产线的 SEO 页面都是由若干套页面组成的，每套页面都是从不同的角度来推广，每个页面由若干个 Module 组成，一个 Module 对应一个接口。

以机票为例：机票的 SEO 页面会包含出发地和机场这两套页面，出发地这套页面由 A、B 和 C 这三个 Module 组成，机场这套页面由 B、C 和 D 这三个 Module 组成，这时仅需要开发 4 个接口分别实现 A、B、C 和 D 这 4 个 Module 对应的功能即可，可以很好的提高接口的复用性。同时，也可以通过配置让一个页面中的某个 Module 在不同的语种、币种、城市等维度中展示不同的数据。

4.4 Page

该项目主要由前端团队负责，这里不做详细描述。

4.5 Portal

主要由 4 个模块组成，其中 Config 模块可以根据不同的语种、币种等条件进行配置来控制 Service 中的各接口在不同参数情况下的返回结果、排序方式等；Log 模块主要用来记录 Vampire 数据更新的进度、更新时长和日志等；

AB Test 模块主要是配合 Config 模块实现不同配置之间的对比，从而帮助业务人员更好的做出抉择；Statistic 模块主要用来统计 Faba 中缓存的命中率等性能方面的数据。

四、总结

SEO 项目的核心在于数据，如何采集数据，更新数据，将质量较好的数据在每次的更新中逐渐沉淀下来是整个项目的关键；接口、数据表设计的尽量简单是提高整个项目性能的根本。

本文只是大致描述了一下 SEO 项目重构的整体方案，对于设计方案中的具体实现细节并未做过多的描述，同时有些非核心功能还在开发中，对此感兴趣的同学可以留言，也欢迎大家拍砖。

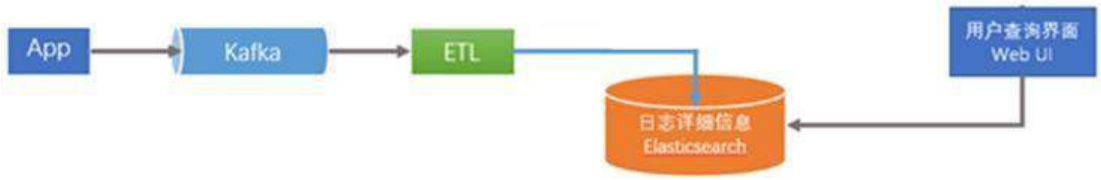
携程机票日志追踪系统架构演进

[作者简介] 许鹏，携程高级研发经理，负责机票大数据基础平台的构建和运维。

机票业务看起来简单，实际上整个流程的处理链条很长，调用关系也非常复杂，上下游涉及的各类日志种类约 60 个，每种日志都有独立格式和请求/响应报文，日生产的日志数据量约 50-100 亿，如果时间范围再扩大到 15 天，数据量轻松的达到千亿级以上。

如何在海量的数据中提取想要的数据，这不是一件容易的事情。在大多数情况下，我们需要一种稳定而快速的架构，帮助我们在资源和性能之间获得平衡，于是我们开始了探索之旅。

一、初始架构



1.1 ElasticSearch

首先需要解决存储和查询的问题，海量的数据需要存储起来，供查询使用。如何有效的存储和查询这些日志数据，是系统设计时要回答的首要问题。

日志数据存储的特点和要求：

- 支持海量写入，TPS 要能够支撑>50K/s
- 支持灵活的 schema
- 支持灵活的数据查询，响应时间要尽可能短，时延<5s
- 对于过期的数据，支持海量删除

按照以上指标，我们对市面上的产品进行摸底和预研，选定了三种存储方式来进行对比：Cassandra、HBase、ElasticSearch。

1.1.1 Cassandra

Cassandra 支持海量的数据写入，但是查询字段单一，同时对于数据删除不够友好，不支持行级别的 TTL。当有大量的 cell 过期后，很容易出现 TombStone 的问题，并且在数据定期清理的过程中，很容易出现数据写入超时等现象。

1.1.2 HBase

1）HBase 支持海量数据写入，在过期数据处理层面，不容易产生 Cassandra 才有的 TombStone 现象。但在查询接口层面，需要调用 api 才行，使用难度较高，尽管引入 apache phoenix 可以通过 SQL 来进行查询，但这增强了系统解决方案的复杂度。
2）HBase 对于 Row Key 的查询能够快速返回，如果变更查询条件，响应会下降非常明显。

1.1.3 Elasticsearch

在排除了 Cassandra 和 HBase 之后，开始尝试 Elasticsearch，通过研究发现，Elasticsearch 可以很好的满足我们的需求：

- 支持灵活的数据结构，支持 schemaless
- 可以通过水平扩展来支持海量的数据写入
- 查询方式灵活，响应时间短，平均查询响应低于<1s
- 结合别名和每天创建新索引，可以很好的移除过期数据，同时操作过程对用户透明

1.2 Kafka

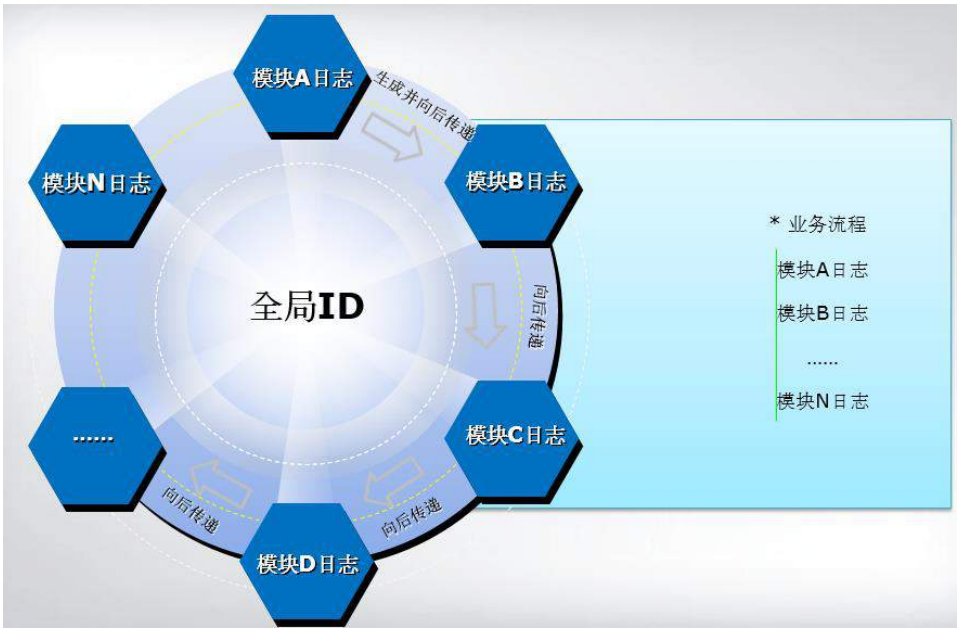
Kafka 作为消息队列，在存储日志数据的同时，隔离开数据产生的应用和数据处理流程。

1.3 ETL

为了把海量日志从 Kafka 近实时的导入到 Elasticsearch，我们采用 spark 来进行处理，当前数据导入延迟不超过 5s。

1.4 全局 ID

每一次用户会话请求会被赋予一个单独的全局 ID(TransactionID)，这个全局 ID 会在各个模块之间的消息传递中出现。通过这样一个全局 ID，开发人员可以追踪请求在整个链路中的处理情况。



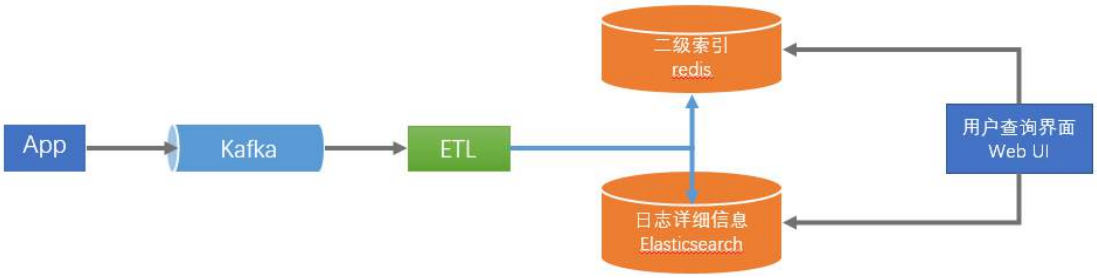
各开发模块将含有全局 ID 的日志信息存储到 Kafka 集群中。

二、架构演进

第一代架构采用 Elasticsearch 解决了日志存储的问题，单日志查询的表现令人满意。

在实际系统使用过程中发现，由于机票日志种类繁多， 同时对 50 个以上日志并行查询会导致 ElasticSearch 集群整体状态变黄甚至变红，集群变的不稳定，整体反应速度变得非常缓慢。

硬件扩容 Or 提升性能，在架构层次需要进行决策，扩容能够解决一些问题，但是对于携程机票而言，后续还会有更多的日志接入，架构层面必须均衡资源和性能的平衡，而不是单纯的硬件扩容，我们决定在架构层面进一步演进来提升性能。



2.1 增加二级索引

通过分析，发现由于 Elasticsearch 会保存最近 15 天的日志，如果针对每一个 TransactionID，都去查询 15 天的所有日志，那么查询响应时间会变得缓慢。

实际上每一个 TransactionID 不可能都存在于 60 多种日志中，做了很多多余的查询，如果能够精确的查询就好了。

为了增强查询的精确性，我们采用只对存有 TransactionID 的索引进行查询，我们建立了二级索引，通过二级索引，可以将 TransactionID 映射到一到多个具体的 Elasticseaerch 索引，然后对这些索引发起查询请求，获取详情信息。

也就是说，我们建立了索引，在查询前能准确的知道一个 TransactionID 在哪些日志、哪些日期中存在。

这样可以准确的查询这些日志，去掉不需要查询的日志。

通过二级索引的设置，查询速度获得很大的提升，由原来的 20-30 秒提升到 5 秒以内。

2.2 冷热数据分离

二级索引的建立解决了很大一部分问题，随着而来又产生了新的问题。

每天的二级索引数据量高达 5 亿条，随着时间推移二级索引数据量迅速增长，查询速度出现了抖动甚至大幅度下降，二级索引本身变成了瓶颈。

对二级索引我们再次做出了优化，对冷热数据进行切割，当天的二级索引会存储到 redis 中，因为系统使用中发现，用户一般对于当天的请求处理情况关注的比较多。Redis 可以在 5ms 以内返回二级索引结果。

对于历史的二级索引，会将信息从 Redis 导入到 Elasticsearch 中。

三、小结

目前，机票日志追踪系统仍然在不断的、持续的演进中，比如最新的二级索引中冷数据不再存储到 ElasticSearch，而是存储在 codis 集群中，ETL 我们采用更快更好的批量灌入方式等等。

随着大数据技术的不断发展和进步，相信我们的架构也会不断的升级换代，架构的升级必然带来效能的提升，这就是技术的魅力所在。

我们始终相信，架构没有最好，只有更好。

携程国际站点Trip.com的无线异步启动框架

[作者简介] 赵辉，专注 Android 平台和 Java 技术栈，目前主要负责 Trip.com App 的性能、网络、存储等基础框架，热爱阅读源码。

受携程全球化战略的影响，IBU（国际业务部）迎来了高速发展时期，Trip.com app 作为国际业务的载体，接入的业务线与日俱增，随之而来的一系列问题也日趋明显。

如何管理启动流程和优化启动时间便是其中之一，经过若干版本的迭代优化，Trip.com app 的启动时间有了明显改观，更重要的是，我们完成了对整个 app 的启动流程监控，使得在多个版本迭代过程中启动时间始终维持在较低水平。

本文并不是优化启动时间的“最佳实践”文章，不会去具体分析如何优化 Android/iOS 的启动时间，而是对 Trip.com 这样的平台型 app 在启动流程优化方面的一些思考及实践的经验。

如果想了解启动时间优化最佳实践，可以参考 Android Developer 上的 App startup time 和 iOS wwdc 上的 Optimizing App Startup Time。

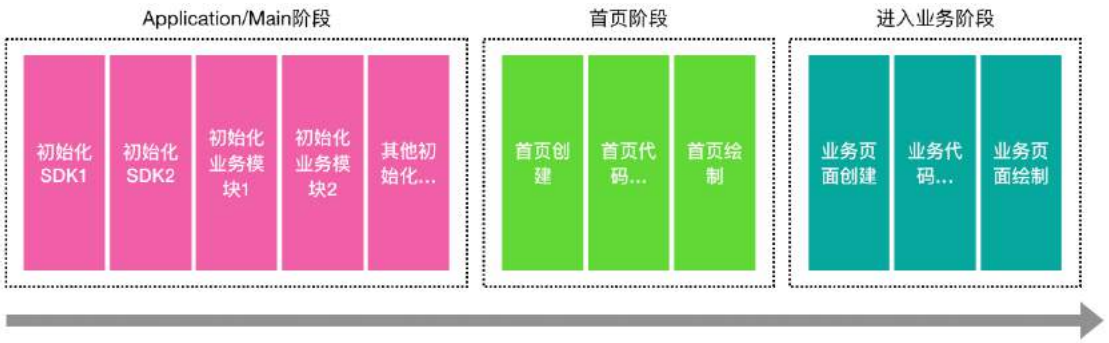
介绍我们的启动流程方案之前，我们先看下一般 app 的启动流程。

一、一般的启动流程

抽象地来说，Android 和 iOS 的启动流程大致分为三个阶段：

1）启动入口
2）进入首页
3）进入业务线页面

如图：



我们这里把用户进入二级页面也算了进来，注意这里所有的流程都是同步按序执行，流程非常简单，但是不可避免地会比较耗时且难以管理维护。

一个新的业务模块、框架层模块或者 sdk（下文简称模块）接入平台，如果需要初始化，就需要在启动入口或者 app 首页加入该模块的初始化代码，而所有业务团队的初始化代码耦合在一起最明显的问题就是：无法进行有效地监控，所以启动时间随着版本迭代越来越长，而且无法区分时间变长的原因是哪些代码导致的，甚至可能由于部分模块的问题导致启动过程发生 Crash，所有这些问题都是不可忍受的。

所以，我们需要从框架的角度重新思考：像 Trip.com 这样一个承载很多垂直业务的平台型 App 该如何优化启动流程？

二、从框架角度思考启动流程

对模块来说，它的初始化代码理论上只需要关心以下几点：

这里我们假设有个业务模块叫“酒店模块”，一个框架层的模块叫“Storage 模块”，有个第三方 sdk 叫“ImageLoader 模块”：

1）代码执行在哪些模块初始化之后，比如“酒店模块”执行初始化代码之前需要保证“Storage 模块”初始化完成；
2）进入模块的任何页面或者使用模块功能之前，其模块的初始化代码必须已经执行完成，比如“ImageLoader 模块”在使用之前必须保证已经执行完它的 init 方法；

在一般启动流程中，以上几点显然很容易支持：

1）“酒店模块”初始化代码写在“Storage 模块”初始化之后即可；
2）“ImageLoader”使用之前启动流程一定已经走完；

但是在这些基础能力之外，我们同时希望：

1）“酒店模块”的初始化代码可以写在酒店项目代码中；
2）如果“酒店模块”和“ImageLoader 模块”如果没有依赖关系的话，可以让它们同时进行初始化；
3）“酒店模块”的初始化不会影响用户进入首页，更不会影响用户进入机票模块；
4）可以很方便地查看 app 启动的流程、每个模块初始化消耗的时间和模块间的依赖关系；
5）“酒店模块”如果发生了 crash，只会影响用户使用酒店的功能；

这时候我们需要有这样一个启动框架来支持这些能力，这些能力抽象如下：

2.1 组件化

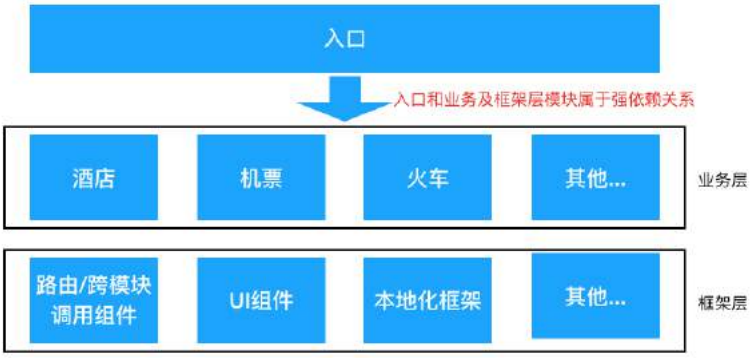
“酒店模块”的初始化代码可以写在酒店项目代码中

对有很多垂直业务的 app 来说，很容易想到的一个框架思想就是"组件化"，显然，组件化的思想也适用于启动流程。

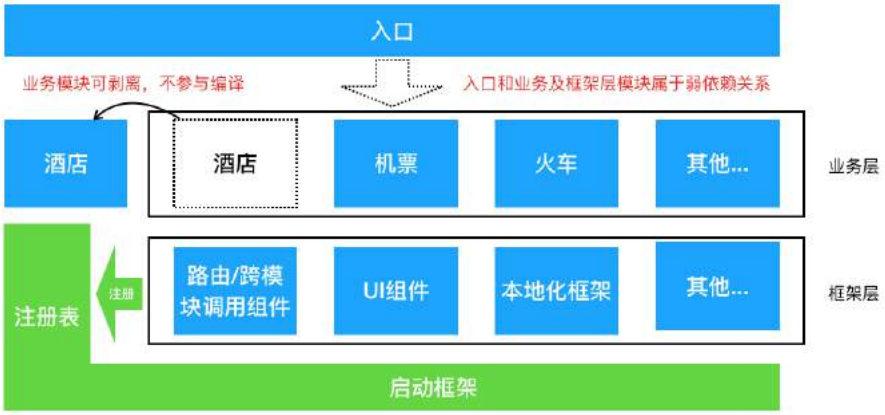
模块的初始化代码应该放在各自的项目模块中，由启动框架统一调度执行，反过来，启动框架也有助于整体项目的组件化拆分。

事实上，Trip.com app 的组件化很大程度上也依赖了启动框架：每个模块代码物理隔离，在启动框架中进行各自的初始化，这些初始化代码包含了组件化架构必要的路由框架和跨模块调用框架。

之前的项目架构：



组件化之后的项目架构：



可以看出来，使用了启动框架之后，原来入口和下层模块之间的依赖就解开了，意味着业务模块可以单独编译或者剥离项目，实现了真正的组件化。

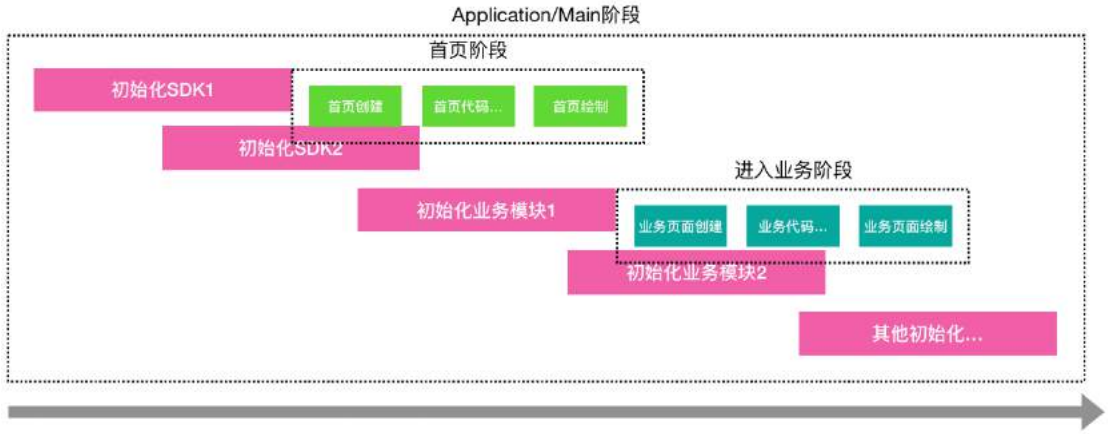
另外对于平台研发团队来说，由于代码都在各个团队的项目中，所以我们可以把启动过程中发现的问题分发给各个负责的团队解决，这也是“组件化”方式开发的好处之一。

2.2 并发执行

如果“酒店模块”和“ImageLoader 模块”如果没有依赖关系的话，可以让它们同时进行初始化。

框架应该可以让启动流程中的任务尽可能地并发以保证最大化利用 cpu，缩短启动时间。当然，影响启动时间的因素很多，比如启动任务的属性是 io 密集还是 cpu 密集、任务执行线程的优先级、是否有足够的 cpu 时间片分配给启动任务同时不会影响 ui 线程、任务间的依赖关系、并发执行的线程数设置多少，所有这些因素或许根本没有办法去精确度量，这也是启动框架无线端部分最重要且最复杂的部分。

并发执行模型如下图，可以看出来，相比于按序执行的普通启动流程，用户看到首页及进入业务模块的时间点都提前了很多。



2.3 业务模块互不影响

“酒店模块”的初始化不会影响用户进入首页，更不会影响用户进入机票模块

用户进入首页的时候不需要等待“酒店模块”初始化完成，而进入“酒店模块”的时候也不需要“机票模块”初始化完成，而只需要保证“酒店模块”及部分依赖的基础模块初始化完成即可。这一点听上去相当 Cool，但是换用户的角度来说，这难道不是基本诉求吗？

2.4 启动流程可视化

可以很方便地查看 app 启动的流程、每个模块初始化消耗的时间和模块间的依赖关系。

如果可以让开发测试，甚至产品经理可以很方便地看到我们的启动流程中都有哪些任务、每个任务执行了多久、任务之间的时序状况如何，那对了解 app 的启动会有非常大的帮助，而且，有了这些可视化数据，我们就可以比较版本迭代过程中的变化，从而发现问题，有目的地进行持续优化。

2.5 启动 Crash 风险规避

“酒店模块”如果发生了 Crash，只会影响用户使用酒店的功能

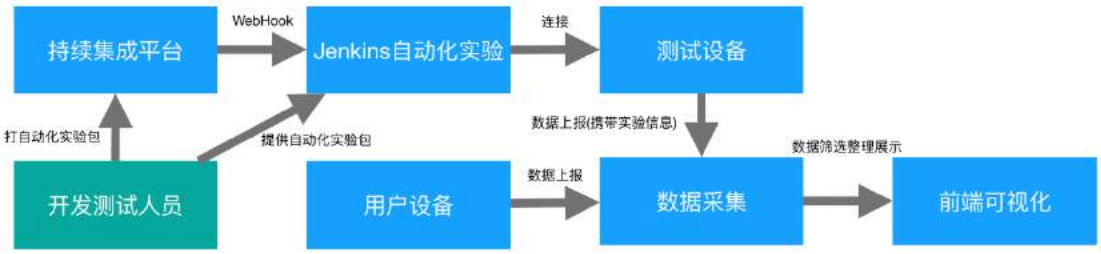
这一点和第三点概念类似，酒店模块一旦出现异常，一方面，我们希望暴露问题，让相关开发人员及时排查，另一方面，我们也不希望用户直接 Crash，而是可以顺利使用机票、火车等功能。

Trip.com 无线平台研发团队经过多个版本的思考和实践，实现了一整套解决方案：Rocket，业务线只需要使用 Rocket 提供的简单的 api 就可以完成接入。

三、Rocket 启动框架

简单来说，Rocket 做了三件事：
1）无线两端（Android、iOS）的启动框架
2）启动自动化实验
3）Debug 及 Release 阶段监控

下图描述了 Rocket 整体方案的流程：



下面就无线端启动框架和自动化实验简单介绍其实现。

3.1 无线端启动框架

无线端需要整理并划分出启动流程中有逻辑关联的代码块，这些代码块可以认为是一个个的启动任务。

由于启动任务的代码需要分散在各个项目模块中（组件化），所以启动框架需要有分发的能力：即整合各模块中的启动任务。实现方式的话，以 Android 为例，使用编译时注解或者配置文件的方式都是可以的。

接下来的问题是如何并发？采用哪种线程池？

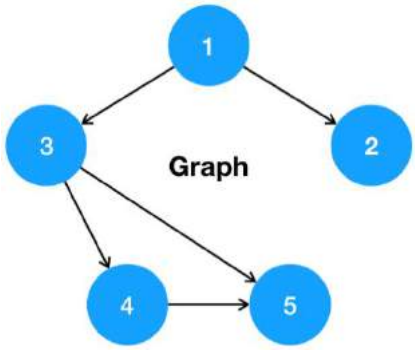
虽然启动任务执行快速，但是经过大量测试发现，采用固定线程数的线程池会比不定数量线程池效果好，那么到底需要多少线程？

上文提到，整体启动时间的影响因素非常复杂，很难量化，所以 Trip.com 采用自动化实验的方式来确定到底需要多少线程，这一点会在自动化实验设计一节展开。

而并发执行带来了两个问题：

1）任务依赖如何解决？

假设有 5 个启动任务，依赖关系如图：



可以很清晰地看到，任务之间的依赖问题其实就是有向无环图的排序，只要每个任务声明自己依赖哪些其他任务，框架就可以拿到任务执行的顺序，至于并发状态下保证执行顺序，Android 和 iOS 可以有不同的实现，以 Android 为例，目前使用了 CAS 类型的锁来实现依赖任务间的时序。

2）如果应用必须等待几个必要任务完成才可以进入首页，如何处理？

针对这个问题，我们还是可以通过 CAS 类型的锁来保证几个必要任务执行结束（锁释放）之后才允许执行首页流程的代码，同样的思路可以解决等待任务的所有场景，比如：锁住酒店页面的代码，直到酒店模块所依赖的模块全部初始化结束再释放锁。

3.2 自动化实验设计

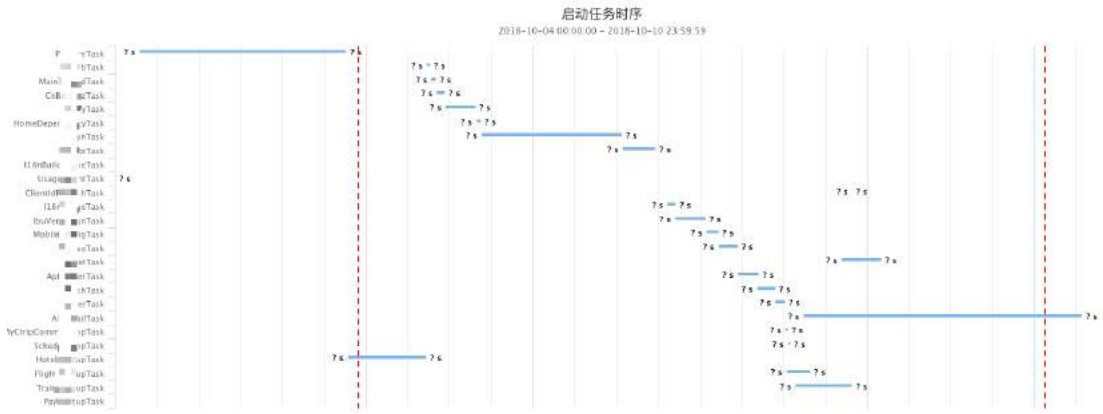
上文提到，影响启动时间的因素很多，且很多因素都难以度量，比如启动线程池的线程数量、不同类型任务间的依赖关系、每个任务执行线程的优先级、应用是否首次启动等等。

在不能够以理论来决定这些因素的时候，我们可以换个思路，用实际运行结果来决定：对几台用户主流设备进行若干次启动，并记录下启动的各项数据上报。利用实验，我们可以对这些影响因素进行手动调优，比如手动调整线程数，进行 30 次测试，最后决定线程数量。

另外，通过这样的方式，app 在上线之前可以先大致预判出上线之后的启动时间，实践证明，这样的方式可以有效测量并缩短启动时间。

实验的脚本考虑到兼容两端，所以我们使用 Appium，它基于 Android 的 UiAutomator 和 iOS 的 UIAutomation，无需修改项目代码，所以理论上只需要 Android 和 iOS 两端收集的实验数据契约一致即可。在上报实验数据的同时，我们会同时生成实验号进行上报，这样就可以在我们 Nemo（前端可视化框架）上筛选出本次实验的各种看板。

下图展示了我们前端的启动时序图：



因为我们保证了实验上报的数据和生产环境的用户数据一致，所以这些前端可视化看板在实验阶段和生产阶段是一样的。

四、未来

借助 Rocket 的一整套方案，Trip.com app 启动时间减少超过 40%，在不需要大量维护精力的情况下，启动时间连续若干版本维持在了较低水平。

当然目前也存在一些不足，比如无线端的任务调度的实现方式还有优化空间、可视化的前端看板也需要提供排查、分发启动问题的能力等等。

启动时间和启动流程的优化对 app 至关重要，希望本文可以给读者带来一些思考。

携程 Redis 海外机房数据同步实践

[作者简介] 孟文超，携程技术中心框架研发部高级经理。2016 年加入携程，目前负责框架数据（数据库，缓存）及相关项目。此前曾在大众点评工作，任基础架构部门通信团队负责人。

一、背景

随着携程国际化业务的发展，为了给海外用户提供更好的服务，公司开始在欧洲部署业务（使用 Amazon 云），欧洲的用户访问欧洲的本地服务。携程机票业务重依赖于 Redis，同时目前的数据产生大部分都在上海，这样就对 Redis 数据同步至欧洲产生了极大的需求，部署在欧洲的业务只需读取 Redis 数据即可。

数据传输如果走专线，将会产生高昂的专线费用（1MB 的带宽，每月约 1 万 RMB），如果能够通过公网，基本可以将数据传输费用降低到忽略不计。这样产生了第二个需求：数据传输走公网。

XPipe（https://github.com/ctripcorp/x-pipe）是携程内部开源的一套 Redis 多机房系统，现有的功能在以往的基础上继续扩展，可以参考公众号的另外一篇文章《携程 Redis 多数据中心解决方案-XPipe》。

二、系统分析设计

在这样的需求背景下，产生了下面几个问题：

1）公网数据网络传输性能不可靠，Redis 内存缓存的增量数据有限，是否会产生频繁的全量同步？
2）数据如何从上海内网传输至公网，再传输至 Amazon 内网？
3）公网传输的性能能满足业务需求吗，会不会导致延时过高，甚至无法追上上海的数据产生速度？

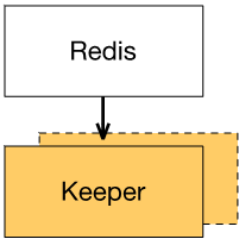
2.1 Redis 全量同步问题

Redis 数据复制本身的工作原理可以参考一下官方手册：
https://redis.io/topics/replication。

本质上是说在 Redis Master 内存里面会以 RingBuffer 的数据结构缓存一段增量数据；如果网络瞬断的话，slave 将会继续从自上一次中断的位置同步数据，如果续不上，就会进行一次全量同步。

因为数据在内存中缓存，而内存有限且昂贵，一个思路就是将数据缓存在磁盘里面。

在我们的具体方案中，设计了一个 Keeper 节点，作为 Redis Slave 向 Master 同步数据，同时将同步后的数据存入本地磁盘，海外数据同步通过 Keeper 进行数据传输，这样就产生了下面的结构：



关于 Keeper 的高可用及其相关的设计，可以参考前文所述的公众号文章。

2.2 数据从上海内网传输到公网问题

公网到内网调用可以使用反向代理软件，因为 Redis 协议为基于 TCP 自定义的文本协议，因此我们要使用支持 TCP 的反向代理工具。

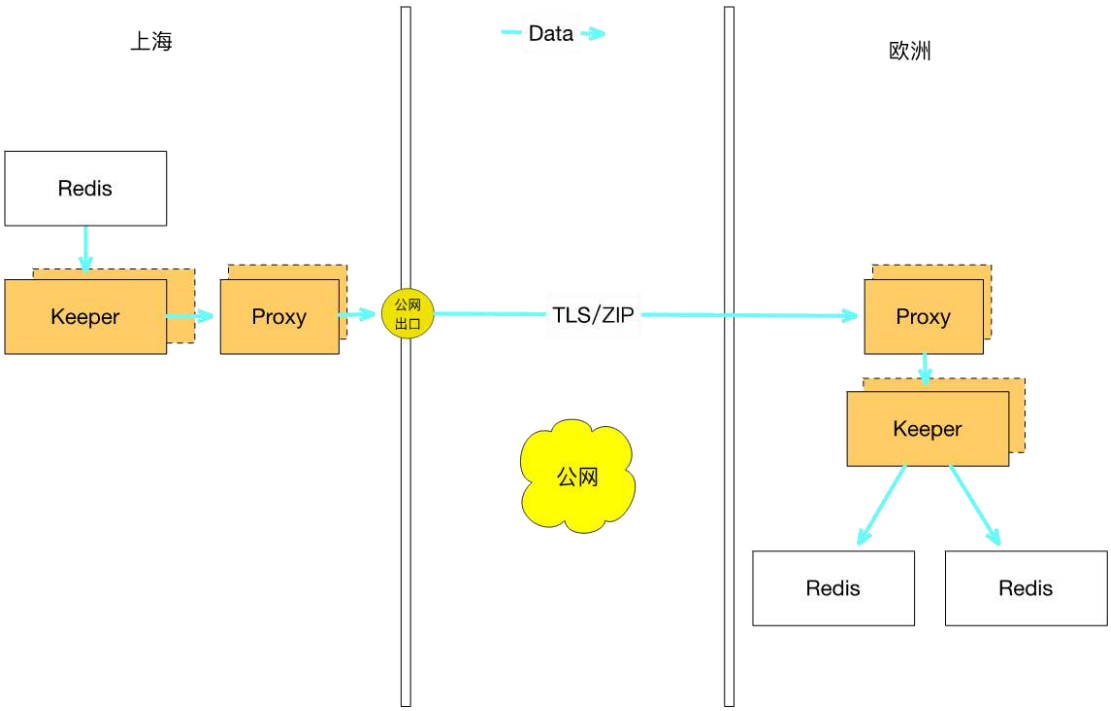
Http 的反向代理可以通过域名、URL 等信息进行路由，定位到目标服务器；TCP 协议的反向代理通过暴漏的端口来路由到不同的服务器集群。

公司内部的的 Redis 集群非常多，如果使用目前的反向代理软件，就意味着要在公网开多个端口，不同的端口路由到不同的 Redis 集群，单个 IP 支持的端口有限，过多端口也会带来更多的安全以及管理问题。

基于此，我们设计了支持动态路由的 TCP Proxy，假设有四个点：S(Source) P1(Proxy) P2(Proxy) D(Destination)，A 需要通过 Proxy 建立到 D 的连接，整个过程如下：

S 建立到 P1 的 TCP 连接
S 发送路由信息：proxy route P2 D
P1 收到信息后建立到 P2 的 tcp 连接
P1 发送路由信息：proxy route D
P2 收到路由信息后建立到 D 的 tcp 连接

整体架构图如下：



如上图所示，上述箭头方向为数据传输方向，TCP 连接建立方向正好相反。Proxy 本身无状态，所以天然可以做到高可用；Proxy 本身支持数据加密、压缩等功能。

用户可能会产生疑问，为什么欧洲还需要一组 Proxy 集群，而不是 Keeper 直接连接上海的 Proxy?

这个主要是基于功能隔离的考虑，Proxy 专注处理好加密、压缩等传输层需要考虑的问题，Keeper 只需要考虑业务相关的功能。加密、压缩算法的优化和变更不会影响 Keeper 。

2.3 公网传输性能问题

一般考虑网络问题时，需要考虑带宽、延时、丢包三组要素，公网传输是高带宽，高延时，高丢包。网络上有很多关于 Long Fat Network 以及网络性能调优的文章，具体不赘述，这里主要描述一下整个实践过程。

首先我们调整了 TCP 的发送接收窗口：

```
net.core.wmem_max=50485760
net.core.rmem_max=50485760
net.ipv4.tcp_rmem=4096 87380 50485760
net.ipv4.tcp_wmem=4096 87380 50485760
```

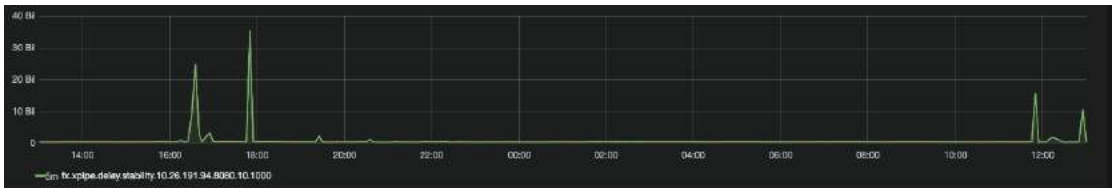
上海到欧洲的网络延时在 200ms 左右，调整完成之后发现在 24 小时的稳定性测试中，会有多个时间点带宽无法打上去，导致数据同步延时过高。仔细观察了一下当时 TCP 连接的状态（通过 ss 命令），发现发送数据时，TCP 发送窗口（cwnd）因为时不时的丢包，导致一直很小，问题主要出在数据发送方。

据此，我们在测试环境比较了多种发送端的拥塞控制（CongestionControl）算法：Cubic, Reno, Htcp. BBR， 下面是在 1%丢包率下不同算法带宽比较：

算法	平均带宽
Cubic	0.15 MBytes/sec
Htcp	0.11 MBytes/sec
Reno	0.09 MBytes/sec·
BBR	13.6 MBytes/sec·

在测试 Case 下，BBR 算法的带宽比其他算法提升了几乎 100 倍，其他丢包率的情况下，也有更好的表现。关于 BBR 算法的更多资料可以参考：
https://netdevconf.org/1.2/papers/bbr-netdev-1.2.new.new.pdf

因为我们的场景中数据发送方是在上海，所以在上海端的服务器部署了 BBR 算法，下面是在 10MB 数据传输速率下，公网 24 小时延时测试数据(单位为纳秒)，数据最大延迟为 88S。

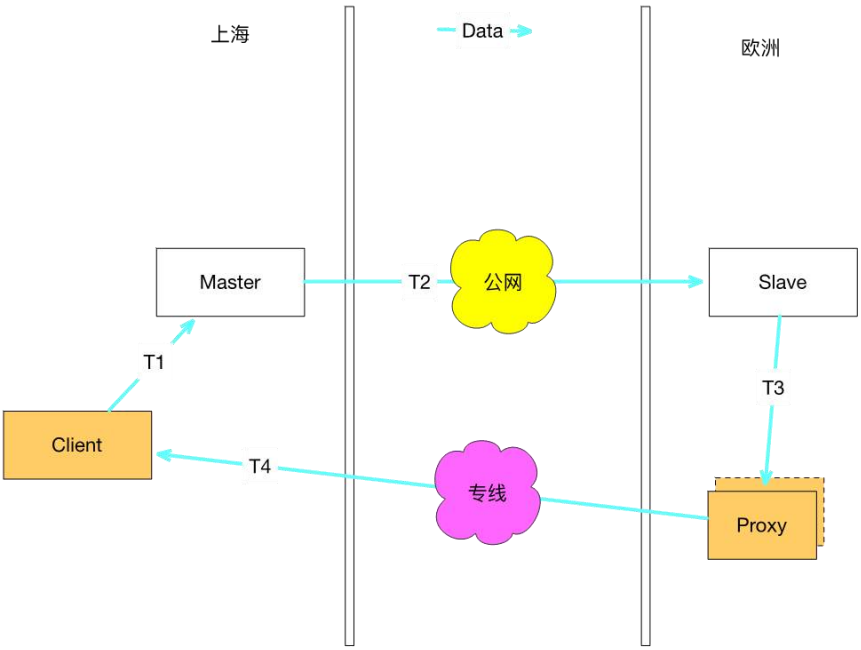


公网和专线比较：

	公网	专线
丢包	高(约 1%)	低(约 0.05%)
带宽	高	低
延时	中	中
价格	非常低	高， 1W RMB/MByte/月

2.4 监控、报警

整个系统中最重要的指标是数据从上海到欧洲的延时是多少。根据此数据可以判定系统是否健康。系统延时监控架构如下图所示：



测试 Client Publish 一个时间戳数据给 Master，然后将这个数据从 Slave 这边订阅收回，整个延时测试时间为：T1+T2+T3+T4。将数据收回计算延时时间主要是为了避免不同服务器时钟不一致产生的影响。

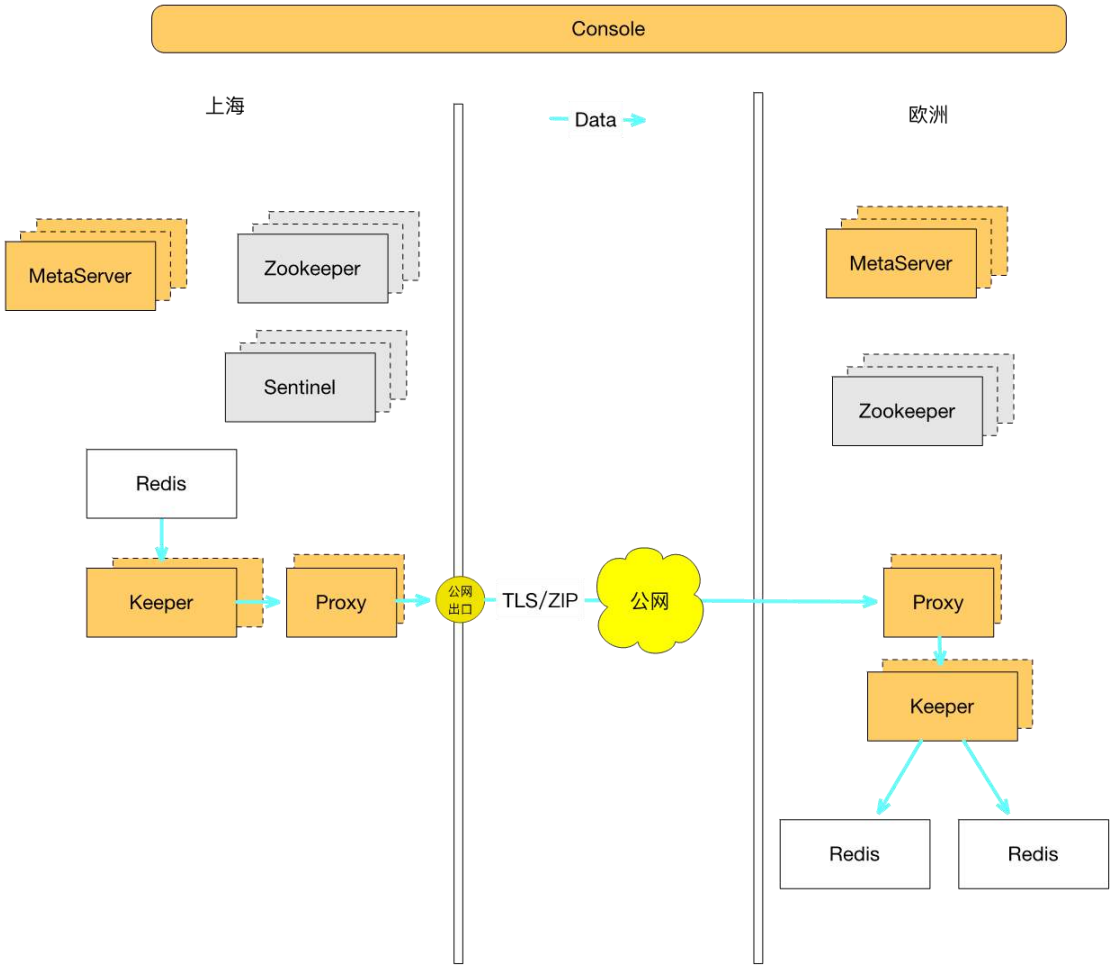
为什么数据订阅走专线？

1）监控系统两秒 Publish 一个时间戳，数据量足够小，不会占用过大专线带宽；
2）我们关注的延时其实主要是 T1+T2 的时间。专线比较稳定，如果数据传输到欧洲走公网，订阅回来也走公网，可能会导致延时测试数据不准确；

为什么数据订阅通过 Proxy 转发？

这个主要是公司安全策略问题，欧洲的服务器只暴漏出了有限的端口供上海访问，而 Redis 服务器可能部署在很多的端口上面，这样可以通过 Proxy 进行转发代理。

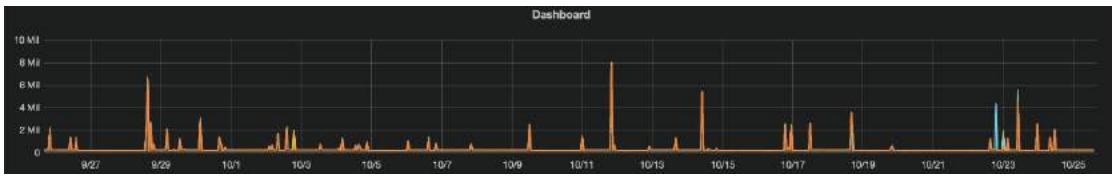
2.5 整体系统架构



整体系统架构如上图所示。Console 用来管理多机房的元信息数据，同时提供用户界面，供用户进行配置和 DR 切换等操作。Keeper 负责缓存 Redis 操作日志。Proxy 主要解决公网传输问题。Meta Server 管理单机房内的所有 Keeper 状态，并对异常状态进行纠正。Zookeer 用来供 Meta Server 和 Keeper 进行 Leader 选举。

三、实践结果

整个系统目前从 2018 年 7 月上线，稳定运行至今。下图是某一业务集群的一个分片 30 天延时监控数据（单位为微秒）。平均延时 180ms 左右，最大可能波动到 1-2min。



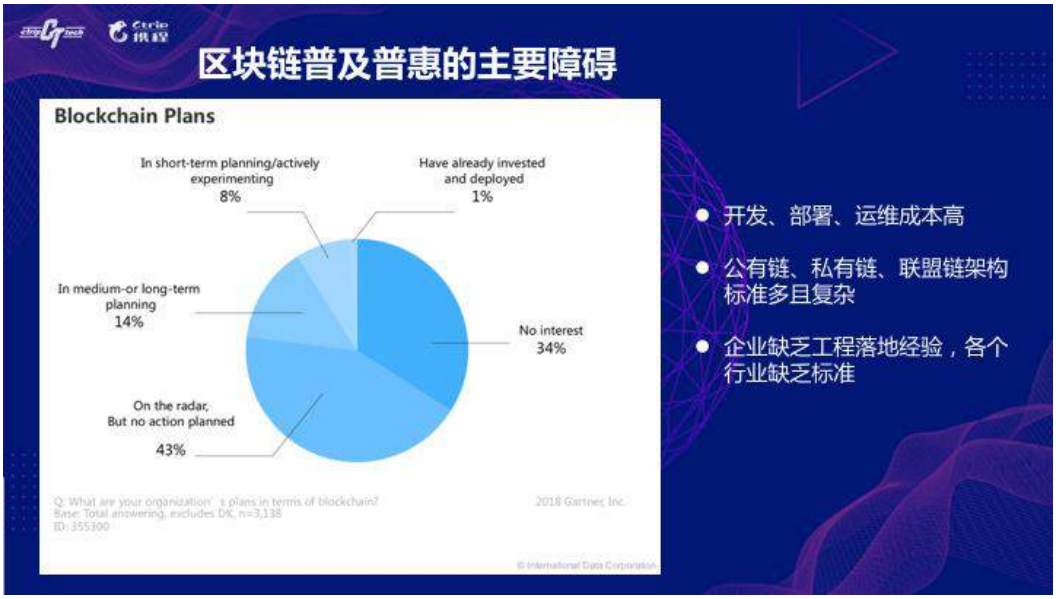
同时，整个系统已开源：XPipe：https://github.com/ctripcorp/x-pipe

HyperLedger Fabric 在携程区块链服务平台的应用实战

[作者简介]何鑫铭，携程技术中心创新研发部区块链技术专家，携程区块链技术平台技术负责人，精通当前主流区块链开源技术框架，热衷于研究区块链底层设计和区块链应用创新。本文来自何鑫铭在“2018 携程技术峰会”上的分享。

一、在业务应用区块链技术之前，我们需要做什么？

在主题介绍之前，一起来看一张图：



上图是 Gartner 提供的一份 2018 年关于企业对区块链技术规划的调研结果，结果表明在受访的企业中（包含高科技、IT、互联网企业）大概有 66%的企业表示对区块链技术感兴趣，但是真实投入研发并且在正式环境部署过的企业大概只有 1%。

区块链技术还未发展到或者人们还没有认识到其所带来的价值和意义，这是现状。其中一个原因是——这个技术的易用性现在是很低的。

易用性有以下三个方面：
1）开发、部署、运维成本高；
2）公有链、私有链、联盟链框架众多，且技术标准还未形成统一共识；
3）各个企业缺乏工程落地经验，各个行业更是缺乏应用落地范本。

现在区块链技术所处的阶段，个人理解像是 web 技术发展的早期，还未形成统一的 web 技

术核心标准（如 http 协议等），所以也就没有太多的 web 应用创造出来。

携程作为国内互联网旅游行业的领军企业，非常重视技术对于业务的创新推动，最近一年我们在非常积极的进行技术研究，以及技术应用的创新。

我们发现，即使是最成熟的如 Fabric、以太坊这样的开源技术框架，也远远没有达到生产环境对于稳定性、高可用性、高并发支持等这些基本要素的要求，而这些框架的学习成本、使用成本、运维成本也非常高，让现有的业务部门技术同事兼职来现学现用，是一件非常困难的事情。

因此，需要做一个支撑业务应用的区块链服务平台，去屏蔽掉最底层区块链系统的网络架构、框架搭建、应用集成、运维监控的复杂性，并且需要做各种源码、环境、使用方式的优化，以让上层业务应用能最高效的应用区块链技术。

平台的目标是各个业务部门的技术同事在充分了解区块链技术理念、基本概念以及如何解决业务痛点的前提下，能够不深入学习以太坊、Fabric 等底层区块链技术框架，就能快速将当前业务与区块链技术结合。包含快速开发、快速部署、快速上线、有效运维以及能够满足对于应用上线的基本要求。

二、携程区块链技术服务平台（CBaas）介绍

下图为 CBaas 平台的技术栈：

因为区块链的部署（尤其是 fabric）对于容器技术是重度依赖的，所以需要一个可应用于生产环境的 swarm/k8s 集群服务。

这样的区块链平台，与独立的业务系统是完全不同的，而更像是集开发、发布、测试、运行、运维于一体的完整的应用操作系统。对于我们现有的应用发布、运维体系有较大冲突，需要

paas 层服务团队提供更为灵活的支持。

上面一层是区块链的底层框架，首选支持的是目前最为成熟的联盟链框架-HyperLedger Fabric，Fabric 目前在国内外是落地最多的框架了。其次是以太坊，以太坊是区块链 2.0 即智能合约平台最重要的框架，其影响力和社会熟知度是比较高的，以太坊更适合做激励型社区类应用。最后是我们正在做的自己的底层区块链框架 CtripChain。

在应用 Fabric 的时候，我们改了一些或者说是扩展了一些框架的源代码。Fabric 是一个在技术、代码设计上非常灵活的框架，因此我们将改动抽象出了代码上的一个插件层，如国密算法、PBFT 共识等。

服务层是 CBaas 平台的主要逻辑所在，我们将 Fabric 等这些框架在更上层抽象出了网络、联盟、通道、节点等概念。

可以通过 CBaas console 页面，你灵活的根据自己的需求搭建区块链技术架构，进行动态节点扩容、动态对链进行治理等。

值得一提的是，我们改变了传统的 baas 平台集中式、上帝式的区块链治理模式。

我们认为，联盟链不应该是由一个组织、一个用户来进行治理，引入了多企业租户共同治理的模式。比如一个既有通道、既有联盟增加新的企业成员，应该由通道/联盟中的组织一起进行签名审批，并且将签名审批结果提交到链上，与链上策略模块提前在线上协商制定好的背书策略签名一致才可以通过。

整个过程中，所有企业在平台上都是一个独立的企业租户，甚至可以将企业租户对应的节点，部署到自己的内网中。只要保证与企业联合建立的联盟网络能够进行 rpc 通信就可以。

服务层其他方面，一个是公链的锚定，这与业务有一定关系。在做区块链创新的过程中，我们也跟业内有名的区块链公司合作，将联盟链上的一些东西，跟公链做 hash 锚定，来进一步增强联盟链的公信力。

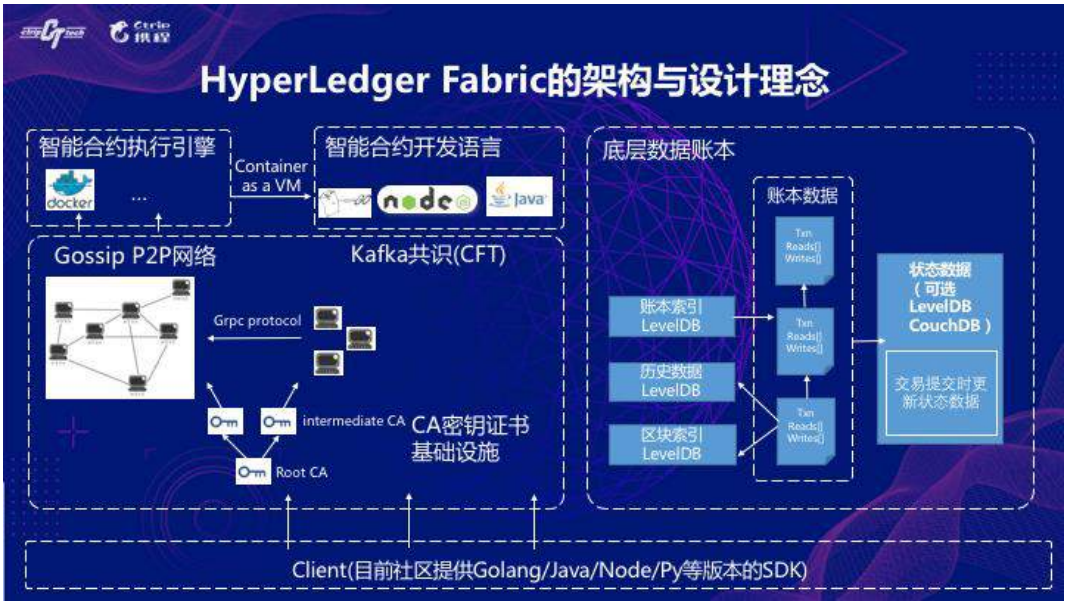
另一块是 api service，进一步对 fabric、以太坊等区块链的客户端 SDK 进行了封装，业务开发人员可以直接调用 CBaas 平台上的 restful 接口来进行合约的调用，甚至是联盟、通道等动态治理。

最后是我们定义的区块链的“前端”展现端，这块包括 portal 工作台、外部节点安装包、OpenAPI，区块链浏览器（可以用于汇报展示用），以及内部的一个智能合约集市，一些比较好的智能合约可以共享在集市上。

三、联盟链框架的选择——HyperLedger Fabric 的架构与设计理念

在做 CBaas 平台选择支持的底层框架时，我们对于 Hyperledger Fabric 的代码研究的一些经验，希望可以给大家在做联盟链底层技术选择时一些参考。

下面是 Hyperledger Fabric 的整体组成，也是当前主流区块链 2.0 技术框架的通用型架构，包含 client SDK、p2p 网络、共识引擎、智能合约执行引擎、底层数据账本，以及联盟链独有的权限体系。



我们首选 Fabric 框架，除了其社区活跃、落地应用案例较多外，另一个比较重要的因素是它的模块设计非常灵活，体现在技术设计、代码模块设计上。

3.1 fabric 模块化设计之合约执行引擎的解耦

首先看一下 fabric 在合约执行引擎层的解耦，我们做一个 fabric 与以太坊的对比。

以太坊的 evm 定义了一个适合在公链网络中可以在以太坊节点上运行的简单、确定、轻量、安全并且能够计算合约运行成本的智能合约虚拟机。是首个在区块链网络上支持智能合约运行并取得成就，且到目前为止，没有因为 evm 的 bug 导致重大事故发生（目前出现的事故都是合约代码的 bug）。

当然，evm 的设计并非没有缺陷，evm 存在的问题：

1）账本数据结构与 evm 代码绑定较深，修改会互相影响。

现在很多机构自研的公链/联盟链，选用的智能合约执行引擎大多都是 evm，如果不加以较为大量的源代码修改、测试，基本意味着底层账本的数据格式、结构都是以太坊的了。当然这个问题在未来也有可能不是问题，前提是 evm 能作为智能合约引擎以及底层账本的技术标准。

2）采用 256 位整数运算，致使 32 位/64 位 x86 处理器相对低效。

这其实是为了适应更大的内存寻址和复杂的密码学运算以实现安全的 gas 模型而设计的，

但是当 evm 被用于联盟链时，这种低效的处理是完全没有必要的，因为联盟链无需消耗 gas。

3）evm 是一个基于栈的虚拟机，大多数操作都使用栈。

栈都是有栈深度的，意味着当使用超过栈深的时候就会报错栈溢出。

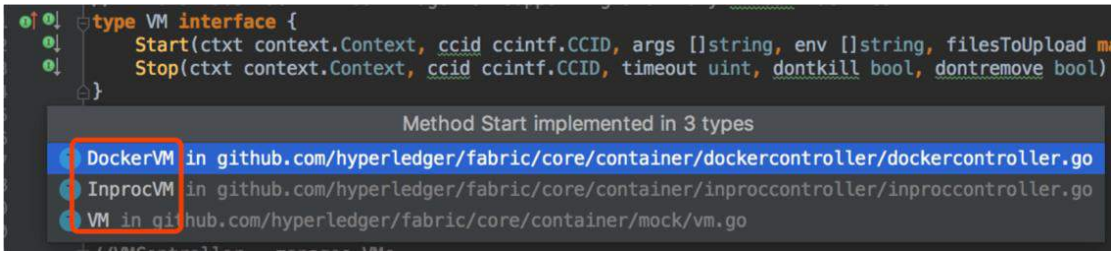
4）evm 的标准库太少。

这个一方面跟 solidity 的语言生态有关，另一方面是因为如果引入大量的第三方库，可能会意味着引入不必要的代码和 gas 消耗。

关于 evm 的问题，这里不深入探讨，网上很多技术大牛对 evm 缺陷有更深入的分析。再来看一下 fabric，fabric 有两块设计来做智能合约执行引擎解耦：

1）在代码层面，定义了 container 接口层，该接口目前有 3 个实现：DockerVM（执行用户合约）、InproVM（执行系统合约）、MockVM（UnitTest 的 mock 环境）。

目前 fabric 的智能合约引擎可以理解为是基于 docker 容器的，当节点主应用部署一个智能合约时，会 socket 连接节点宿主机的 docker，动态生成一个可以执行智能合约语言的 docker 容器。这是 fabric 自带的一个实现，因为 fabric 在这块是代码解耦的，意味着可以实现该接口来实现自己的智能合约执行引擎，如 jvm 等。



2）DockerVM 的实现中，docker 容器与区块链节点的通信方式为 grpc，意味着通过协议的模式进行了代码层的解耦。这与以太坊中 evm 代码与节点代码难以解耦不同。

我们总结了 fabric 以下优缺点：

优点：

1）代码层面上实现了对 VM 和节点进行脱耦，并且易于扩展新的 VM 方式。
2）dockerVM 的原理，理论可支持众多开发语言开发智能合约。

缺点：

依赖 docker 运行环境，严重限制 fabric 节点的部署可能性；docker 作为沙箱环境相对复杂，安全性、稳定性都面临较大的挑战，难以适用于公链环境，但是可以应用在一个确定运行环境的联盟链上。

3.2 fabric 模块化设计之链上代码逻辑的解耦

这一点我觉得是 fabric 明显优于现在的区块链 2.0 众多联盟链框架的地方，也是很多区块链 3.0，如 EOS 等正在做的东西，那就是——将更多主链上的逻辑（非用户开发的智能合约）作为链上的事务，或者是作为链上的智能合约来设计。

这就意味着，首先链上的逻辑可以更灵活的被修改甚至可以在不需要在有可能引起分叉的代码升级的前提下进行运行时修改；再就是链上逻辑的修改可以像智能合约一样，被共识。

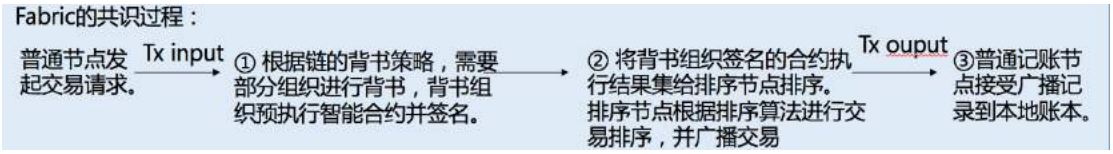
Fabric 将节点代码中的部分逻辑，如背书过程、交易验证过程、智能合约生命周期管理、配置管理（对应 escc、vscc、cscc、lscc 系统链码）都作为链上合约来设计，称之为系统合约。

这些过程是可以被链的共识机制所覆盖的，所以才有了 fabric 可以通过定义各种策略，来实现非中心化地干预这些内置处理流程，如可以定义背书策略、智能合约初始化策略等。

不过现在 fabric1.3 的版本并没有做到链上的逻辑可以被灵活修改甚至是运行时修改，到现在只是开放了开发者可以通过代码替换来自定义修改 escc、vscc。现在的开发者可以通过修改这两个系统合约，实现很多 fabric 目前实现不了的功能，比如：基于数据状态的背书策略、匿名交易场景（公钥匿名）等。

3.3 fabric 模块化设计之共识引擎的解耦

我们先来回顾一下 fabric 的共识过程：



其实 fabric 的共识过程是比较有自己的特点的，跟公链的共识过程也有比较大的不同：公链的共识者，同时承担合约预执行、交易排序的职责；fabric 中排序节点只做排序，合约预执行由背书节点做。（fabric 中背书节点与排序节点的组合=公链如以太坊中的共识节点）。

不过，fabric 在共识这一块的解耦是跟智能合约执行引擎比较类似的做法：

1）排序节点代码侧定义了 consenter 接口，可以通过实现 consenter 接口拓展共识排序算法.
2）刚讲了 fabric 的共识要算上背书节点背书和交易验证模块，对应 escc/vscc 两个系统合约，恰好这两个系统合约是可以修改的。

以下附录这一点的完整总结。

3.4 fabric 模块化设计之权限控制的解耦

权限控制其实作为联盟链重要的特征，在 fabric 中体现的淋漓尽致，我们来看一下 fabric 是如何做整个链上的权限控制的呢？

在设计常规的多租户企业级软件时，我们往往都会先定义软件的使用企业、企业用户、系统角色，再定义每个页面、菜单，然后再将企业、用户、角色与页面、菜单结合起来，这样就可以设置哪个企业、什么样的用户、什么样的角色有权限访问某个页面、菜单……

其实 fabric 的设计与这种企业软件的设计类似，首先 fabric 中权限的最高级别是 msp，msp 可以是一个组织，如 org1，用来做整个区块链的企业租户切分，msp 之下，fabric 又定义了

用户、节点，组成权限体系的角色 role 层级。如：Org1.admin、Org1.member、Org1.peer。

而组织，包括组织下的用户、节点等都有一个唯一的 ID，这个唯一的 ID 在区块链中成为 identity（以太坊的 identity 比较简单，它是一个公链所以 identity 只代表用户），每个 identity 基于非对称密码学对应一对公私钥。

区块链在运行时，全靠这个 identity 来标识身份。fabric 有一个子项目叫 fabric-ca，提供这个 identity 的管理机制，即一套 PKI 公钥基础设施。

fabric 将很多链上的过程，都定义成了上面所讲的企业软件中的“功能”，而功能与角色或者 ID 的对应访问关系，叫做“ACL”。

现在比较好理解了吧，其实 ACL 就是企业级软件中的哪个企业、什么样的用户、什么样的角色有权限访问某个功能。以下截图是部分 fabric 中现有的 ACL，我们可以通过修改这个 ACL，达到修改 fabric 中某个过程中的权限控制。

```
# ACL policy for invoking chaincodes on peer
peer/Propose: /Channel/Application/Writers

# ACL policy for chaincode to chaincode invocation
peer/ChaincodeToChaincode: /Channel/Application/Readers
```

以下附录这一点的完整总结。

3.5 Fabric 对于同构链中多链以及多链通信的设计

这一点是有别于前面三点的，前面三点都是说明 fabric 是如何做技术设计的解耦。而这一点我们聊聊 fabric 对于同构链中多链以及多链通信的设计。

首先解释一下什么是多链问题，我们知道，其实我们所熟知的以太坊、比特币主链其实都是一条比较大型的公有链。而其实除了主链外，基于比特币、以太坊源码有很多机构自己重新搭建的一条新的链。

比如同样的两条以太坊搭建的链，就说明两条链是同构链，而两条链之间如果需要通信，就是同构链的通信，也就是多链通信。刚刚讲的只是多链的一部分形态，还有侧链、子链、平行链等更多多链形态，为大家容易理解，不做赘述。

对于 fabric，首先它定义了通道的概念，即一个 fabric 联盟链网络，可以有多个通道，每个通道对应本地一套单独的账本，这个通道可以理解为一个类似于子链的概念。

我们可以把每个通道，看成是一个较为独立的子链，这个子链的账本是物理隔离的，不过每个子链需要共享父链的排序节点。

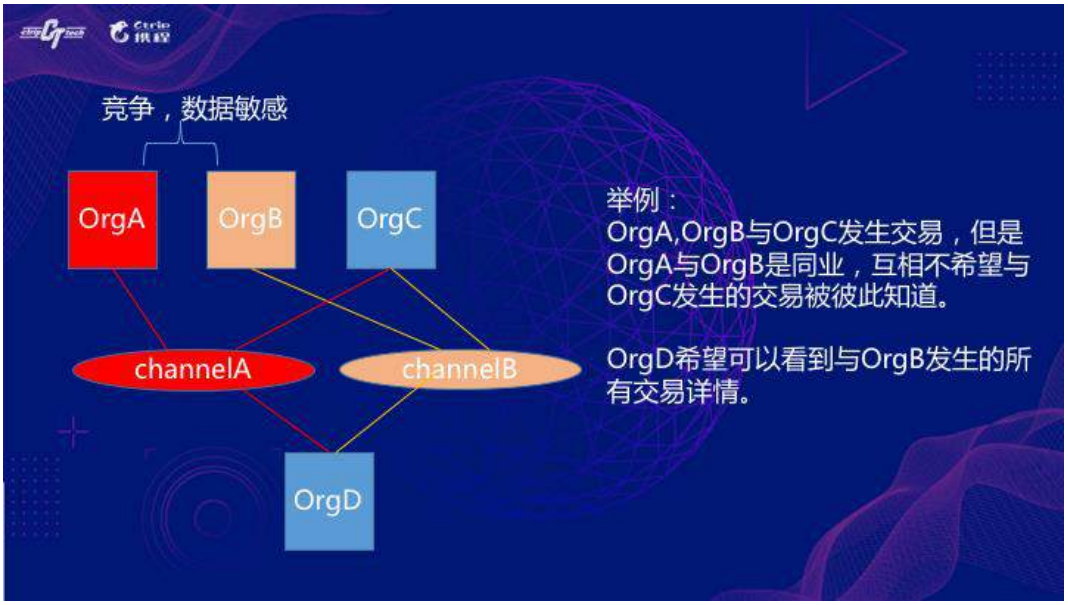
目前 fabric 中跨通道的通信，是通过智能合约间的调用实现的，如同时在 channel1/channel2 上的节点安装的合约 1/合约 2 可以互相调用，即两个通道只有在存在交集节点的情况下，才可以通信，还未实现完全独立的通道之间的数据互通。

fabric 中通道的设计其实可以做很多远远超过你预期的事情，如隐私数据保护、缓解节点数据无法分片问题、实现并行计算支持高并发。

四、fabric 在链上保存原始数据（非哈希）并可以按需分享的一种解决方案

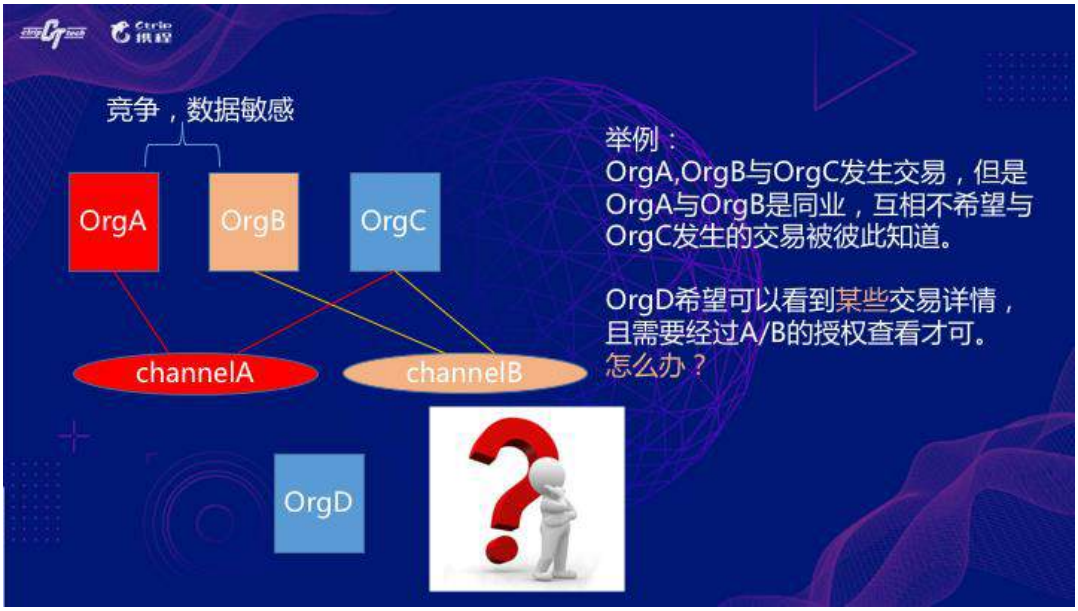
下面分享我们在 fabric 应用过程，这个分享标题完整版为：在保护数据隐私的前提下，如何用 fabric 在链上保存原始数据（非哈希）并可以按需分享的一种解决方案。

首先来看一个区块链应用的场景：



如 A 与 C，B 与 C 分别在发生交易，但是 A 和 B 是同业，互相不希望与 C 发生的交易被彼此知道。所以放到 fabric 中，大家肯定要分别设计两个 channel，来屏蔽两个交易方，通过 channel 可以做一些交易对手间的信息共享。

而这个时候假设存在 D，在实际业务中有权利查看 AC 和 BC 的全部业务数据，则 D 可以分别加入到 channelA,B 中。这个模型是没问题的，那我们现在把问题变复杂一些。

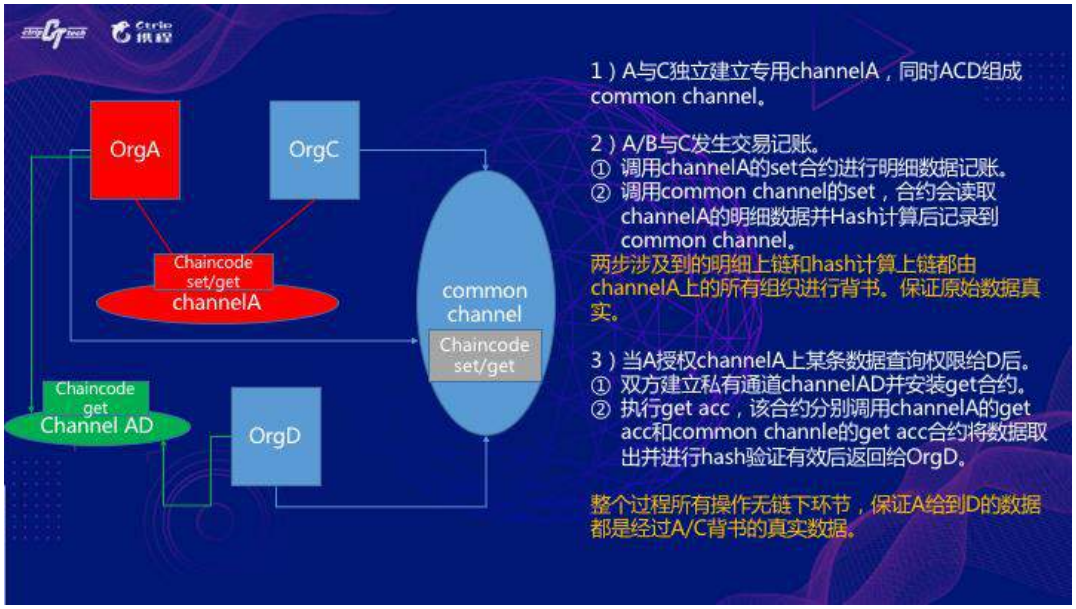


OrgD 假设是没有权利查看 AC、BC 所有交易的权限的。但是在实际业务中，他希望可以看到 AC、BC 的某些交易详情，且需要经过 A/B 的授权才可以查看，该怎么办?

这个时候有同学可能会讲，A 和 B 分别把他希望的数据给他不就可以了吗？我们这里有个前提，就是我们需要借助区块链这个技术，直接将数据在链上给过去，因为这样数据可以经过

交易对手方 C 组织的背书，假设 D 能够直接从 channelA、B 上拿到他想要的数据，那么这个数据是天然经过 C 组织背书的。

我们来看这样一个解决方案：



这张图已经完整的描述了整个方案的详情，以及这个方案的优点：整个交易过程全部上链，不引入链下的过程，实现数据、信任完全的链上流转。这个案例经常出现的业务场景想必大家已经猜到了，就是金融领域。

也许真实遇到的业务场景要比我上面举的例子要复杂的多，也许还有更好的解决方案，如很多人提出一个零知识证明的算法，但是之所以将这个例子拿出来交流，其实是希望每一个从业者能从“哈希上链”这样一个保护伞下中逐渐走出，去尝试克服真实数据上链的各种难题，从而使区块链技术真正地服务于实际业务，让业务数据能够真实的在链上互转，真正成为“信任机器”的主角。

希望在各位同道的一起努力下，区块链技术能够真正走出实验室、走出技术极客的圈子，发挥出技术应有的价值。

配置中心，让微服务『智能』

[作者简介] 宋顺，携程框架架构研发部技术专家，开源配置中心 Apollo 作者。2016 年初加入携程，主要负责中间件产品的相关研发工作。 毕业于复旦大学软件工程系，曾就职于大众点评，担任后台系统的技术负责人。本文来自宋顺在“2018 携程技术峰会”上的分享。

一、背景介绍

随着微服务的流行，应用和机器数量急剧增长，程序配置也愈加繁杂：各种功能的开关、参数的配置、服务器的地址等等。 同时，我们对程序配置的期望值也越来越高：配置修改后实时生效，灰度发布，分环境、分集群管理，完善的权限、审核机制等等。

在这样的大环境下，传统的通过配置文件、数据库等方式已经越来越无法满足我们对配置管理的需求。

配置中心，应运而生！

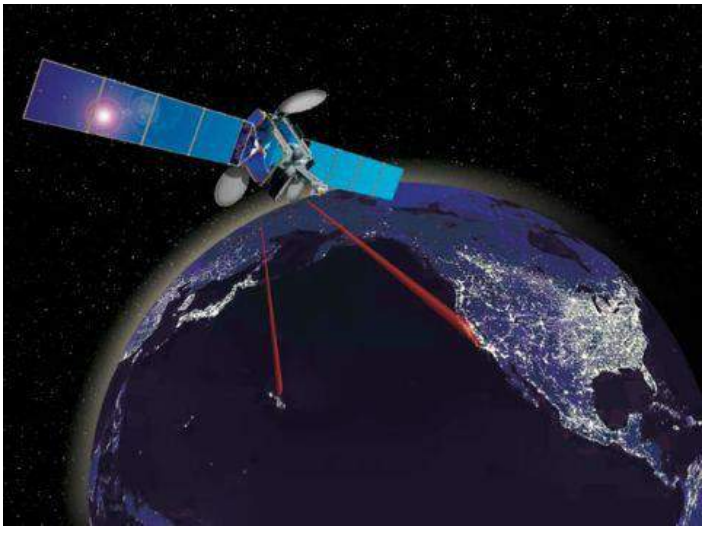
通过配置中心，我们可以方便地管理微服务在不同环境中的配置，从而可以在运行时动态调整服务行为，真正实现配置即『控制』的目标。 所以，在一定程度上，配置中心就成为了微服务的大脑，如何用好这个大脑，让微服务更『智能』，也就成为了一项比较重要的议题。

二、为什么需要配置中心?

2.1 配置即『控制』

程序的发布其实和卫星的发射有一些相似之处。

当卫星发射升天后，基本就处于自主驾驶状态了，一般情况下就是按照预设的轨道运行，间歇可能会收到一些来自地面的『控制』信号对运行姿态进行一定的调整。



程序发布其实也是这样，当程序发布到生产环境后，一般就是按照预设的逻辑运行，我们无法直接去干预程序的行为，不过可以通过调整配置参数来动态调整程序的行为。这些配置参数就代表着我们对程序的『控制』信号。

由此可见，配置对程序的运行非常重要，我们需要一种可靠性高、实时性好的手段，从而可以随时对程序『发号施令』。

2.2 配置需要治理

鉴于配置对程序正确运行的重要性，所以配置的治理就显得尤为重要了，比如：

权限控制、审计日志

由于配置能改变程序的行为，不正确的配置甚至能引起灾难，所以对配置的修改必须有比较完善的权限控制。同时也需要有一套完善的审计机制，能够方便地追溯是谁改的配置、改了什么、什么时候改的等等。

灰度发布、配置回滚

对于一些比较重要的配置变更，我们一般会倾向于先在少量机器上修改看看效果，如果没问题再推给所有机器。同时如果发现配置改得有问题的话，需要能够方便地回滚配置。

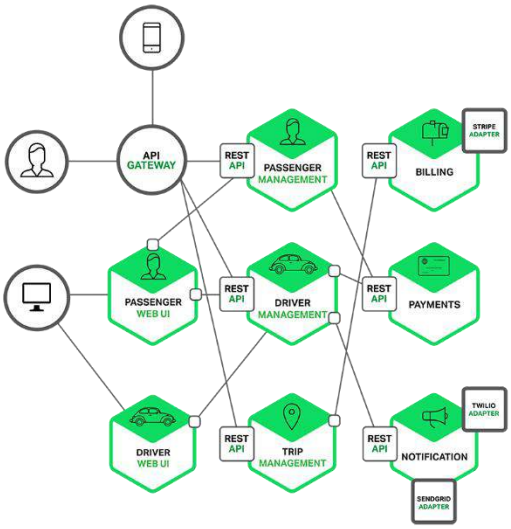
不同环境、集群管理

同一份程序在不同的环境（开发，测试，生产）、不同的集群（如不同的数据中心）经常需要有不同的配置，所以需要有完善的环境、集群配置管理。

2.3 微服务的复杂性

单体应用时代，应用数量比较少，配置也相对比较简单，还有可能让运维登上机器一台一台修改程序的配置文件。

随着微服务的流行，大应用拆成小应用，小应用拆成多个独立的服务，导致微服务的节点数量非常多，配置也随着服务数量增加而急剧增长，再让运维登上机器一台一台手工修改配置不仅效率低，而且还容易出错。如果碰到了紧急事件需要大规模迅速修改配置，估计运维人员也只能两手一摊了。



所以，综合以上几个要素，我们需要一个统一的配置中心来管理微服务的配置。

三、配置中心的一般模样

那么，我们应该需要一个什么样的配置中心呢？

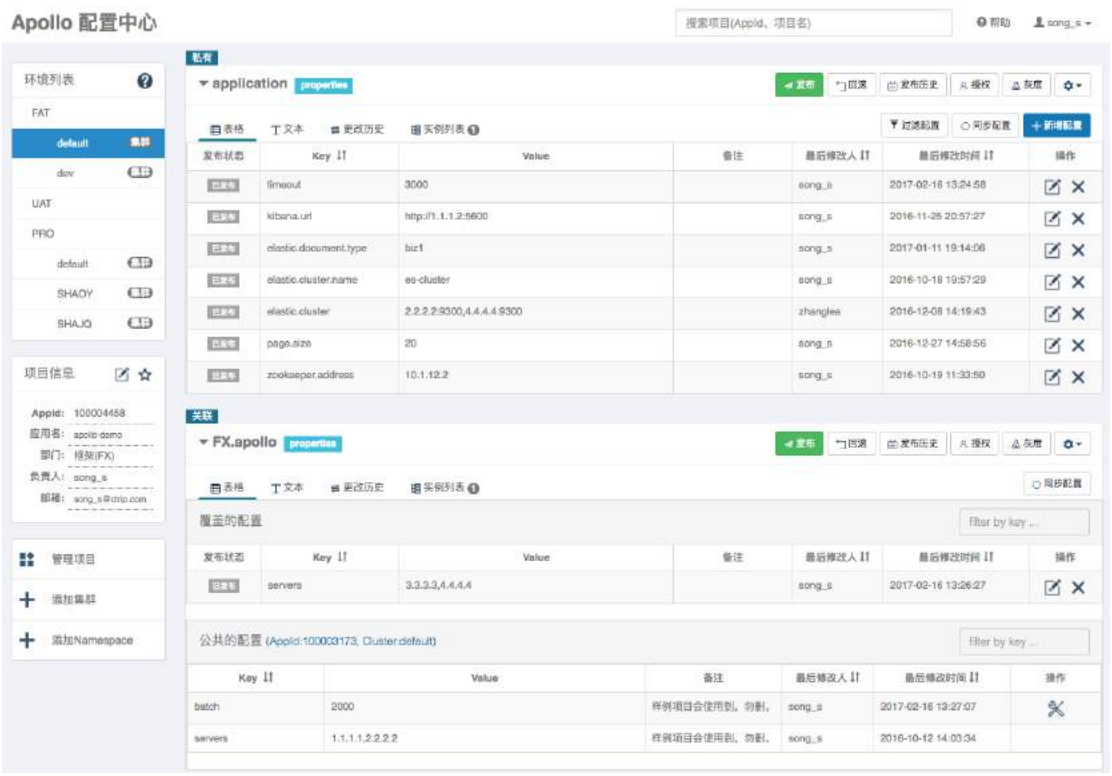
接下来就以开源配置中心 Apollo 为例，来看一下配置中心的一般模样。

3.1 治理能力

如前面所论述的：配置需要治理，所以配置中心需要具备完善的治理能力，比如：

1）统一管理不同环境、不同集群的配置
2）支持灰度发布
3）支持已发布的配置回滚
4）完善的权限管理、操作审计日志

Apollo 配置中心的管理界面如下图所示，可以发现相应的治理功能还是非常齐全的。



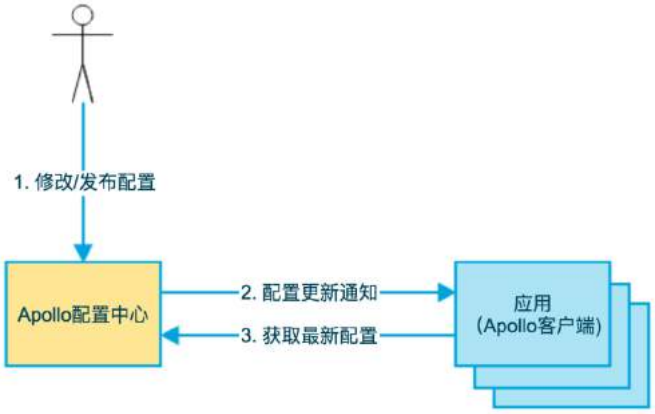
3.2 可用性

配置即『控制』，所以在一定程度上，配置中心已经成为了微服务的大脑，作为大脑，可用性显然是要求非常高的。

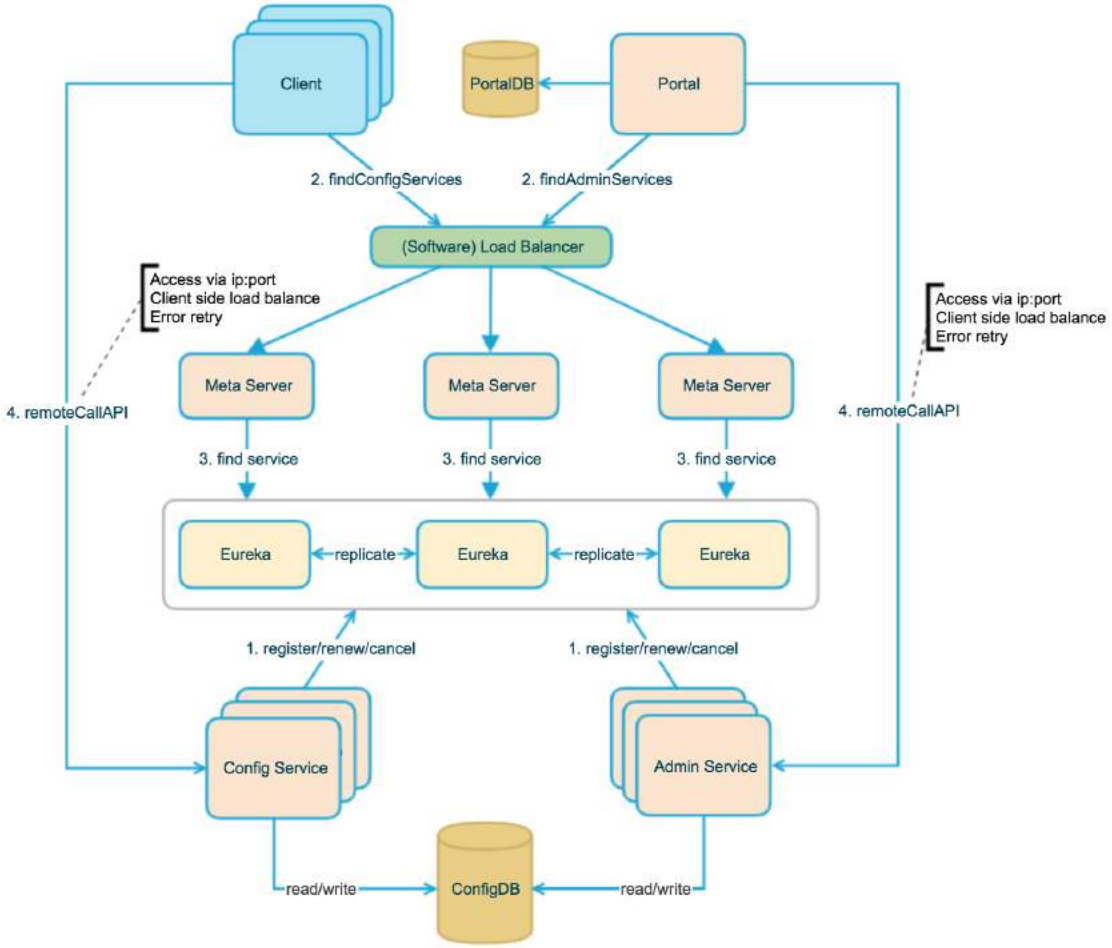
接下来我们一起看一下 Apollo 是怎么实现高可用的。

3.2.1 Apollo at a glance

如下即是 Apollo 的基础模型：
1）用户在配置中心对配置进行修改并发布
2）配置中心通知 Apollo 客户端有配置更新
3）Apollo 客户端从配置中心拉取最新的配置、更新本地配置并通知到应用



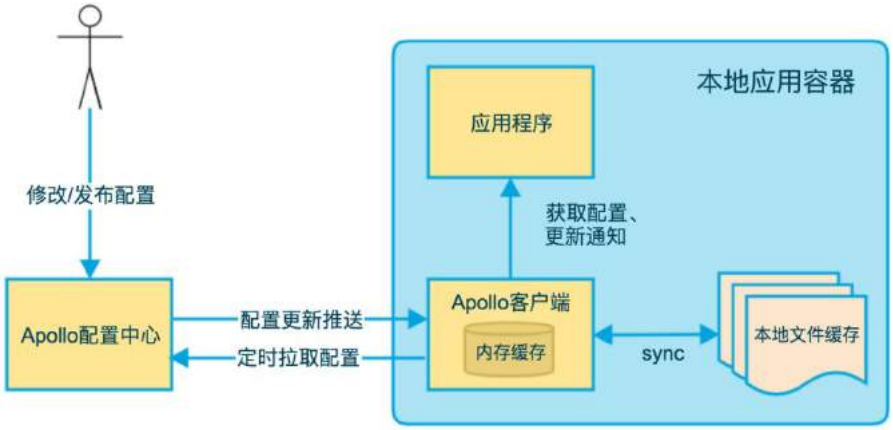
3.2.2 服务端高可用



上图简要描述了 Apollo 的服务端设计，我们可以从下往上看：

1）首先最下面是一个 DB，我们的配置是放在 DB 里的，然后在 DB 之上有两个服务：Config Service 和 Admin Service
2）Config Service 提供配置的读取、推送等功能，服务对象是 Apollo 客户端
3）Admin Service 提供配置的修改、发布等功能，服务对象是 Apollo Portal（管理界面）
4) Config Service 和 Admin Service 都是多实例、无状态部署，所以需要将自己注册到 Eureka 中并保持心跳
5）在 Eureka 之上我们架了一层 Meta Server 用于封装 Eureka 的服务发现接口，主要是为了让客户端和 Eureka 解耦
6）Client 通过域名访问 Meta Server 获取 Config Service 服务列表（IP+Port），而后直接通过 IP+Port 访问服务，同时在 Client 侧会做 load balance、错误重试
7）Portal 通过域名访问 Meta Server 获取 Admin Service 服务列表（IP+Port），而后直接通过 IP+Port 访问服务，同时在 Portal 侧会做 load balance、错误重试
8）为了简化部署，我们实际上会把 Config Service、Eureka 和 Meta Server 三个逻辑角色部署在同一个 JVM 进程中
9）通过上述的设计，可以看到整个服务端是无单点，有效地保证了服务端的可用性

3.2.3 客户端高可用



上图简要描述了 Apollo 客户端的实现原理:

1）客户端和服务端保持了一个长连接，从而能第一时间获得配置更新的推送。（通过 Http Long Polling 实现）

2）客户端还会定时从 Apollo 配置中心服务端拉取应用的最新配置。

这是一个 fallback 机制，为了防止推送机制失效导致配置不更新。

客户端定时拉取会上报本地版本，所以一般情况下，对于定时拉取的操作，服务端都会返回 304 - Not Modified。

3）客户端从 Apollo 配置中心服务端获取到应用的最新配置后，会保存在内存中，所以我们的应用程序来获取配置的时候其实始终是从内存中获取的。

4）客户端还会把从服务端获取到的配置在本地文件系统缓存一份。

这主要是为了容灾，假设应用程序重启的时候，恰好远端服务全挂了，或者网络有故障，应用程序依然能从本地恢复配置。

5）通过这种推拉结合的机制，以及内存和本地文件双缓存的方式，有效地保证了客户端的可用性。

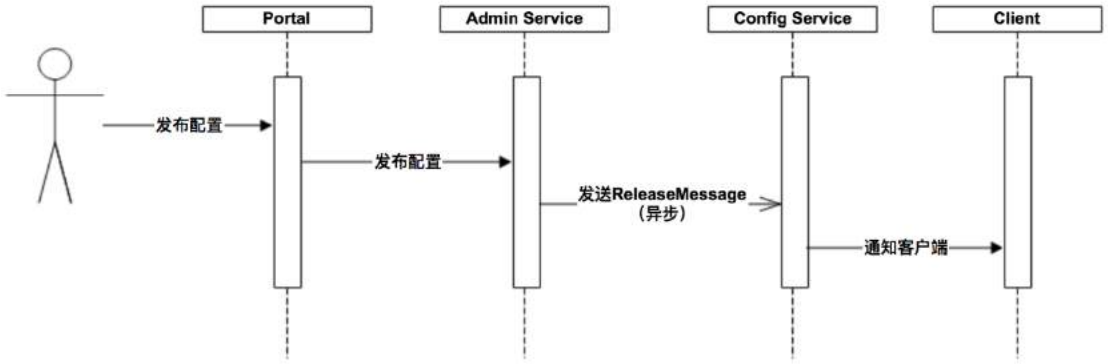
3.2.4 可用性场景举例

场景	影响	降级	原因
某台 Config Service 下线	无影响		Config Service 无状态，客户端重连其它 Config Service
所有 Config Service 下线	客户端无法读取最新配置，Portal 无影响	客户端重启时，可以读取本地缓存配置文件。如果是新扩容的机器，可以从其它机器上获取已缓存的配置文件	
某台 Admin Service 下线	无影响		Admin Service 无状态，Portal 重连其它 Admin Service
所有 Admin Service 下线	客户端无影响，Portal 无法更新配置		
某台 Portal 下线	无影响		Portal 域名通过 SLB 绑定多台服务器，重试后指向可用的服务器
全部 Portal 下线	客户端无影响，Portal 无法更新配置		
某个数据中心下线	无影响		多数据中心部署，数据完全同步，Meta Server/Portal 域名通过 SLB 自动切换到其它存活的数据中心

场景	影响	降级	原因
数据库全部宕机	客户端无影响，Portal 无法更新配置	Config Service 开启配置缓存后，对配置的读取不受数据库宕机影响	

3.3 实时性

配置即『控制』，所以我们希望我们的控制指令能迅速、准确地传达到应用程序，我们来看看 Apollo 是如何实现实时性的。



上图简要描述了配置发布的大致过程：

1）用户在 Portal 操作配置发布
2）Portal 调用 Admin Service 的接口操作发布
3）Admin Service 发布配置后，发送 ReleaseMessage 给各个 Config Service
4）Config Service 收到 ReleaseMessage 后，通知对应的客户端

3.3.1 发送 ReleaseMessage 的实现方式

Admin Service 在配置发布后，需要通知所有的 Config Service 有配置发布，从而 Config Service 可以通知对应的客户端来拉取最新的配置。

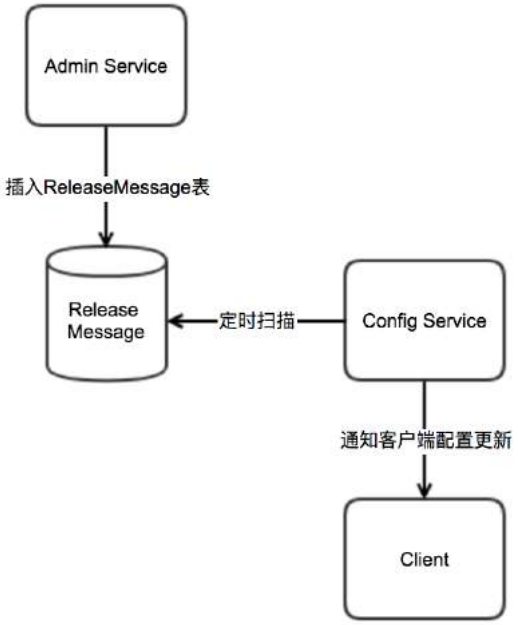
从概念上来看，这是一个典型的消息使用场景，Admin Service 作为 producer 发出消息，各个 Config Service 作为 consumer 消费消息。通过一个消息组件（Message Queue）就能很好的实现 Admin Service 和 Config Service 的解耦。

在实现上，考虑到 Apollo 的实际使用场景，以及为了尽可能减少外部依赖，我们没有采用外部的消息中间件，而是通过数据库实现了一个简单的消息队列。

实现方式如下：

1）Admin Service 在配置发布后会往 ReleaseMessage 表插入一条消息记录，消息内容就是配置发布的 AppId+Cluster+Namespace
2）Config Service 有一个线程会每秒扫描一次 ReleaseMessage 表，看看是否有新的消息记录
3）Config Service 如果发现有新的消息记录，解析得到配置发布的 AppId+Cluster+Namespace 后，通知到对应的客户端

示意图如下：



四、如何让微服务更『智能』？

接下来我们来看一下结合配置中心，我们能做哪些有趣的事情，让微服务更智能。

4.1 开关

4.1.1 发布开关

发布开关一般用于发布过程中，比如：

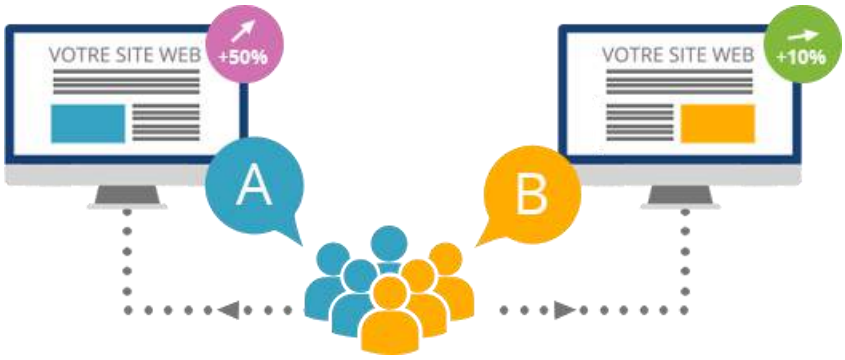
1）有些新功能依赖于其它系统的新接口，而其它系统的发布周期未必和自己的系统一致，可以加个发布开关，默认把该功能关闭，等依赖系统上线后再打开。
2）有些新功能有较大风险，可以加个发布开关，上线后一旦有问题可以迅速关闭。

需要注意的是，发布开关应该是短暂存在的（1-2 周），一旦功能稳定后需要及时清除开关代码。

4.1.2 实验开关

实验开关通常用于对比测试或功能测试，比如：

1）A/B 测试
针对特定用户应用新的推荐算法
针对特定百分比的用户使用新的下单流程



2）QA 测试
有些重大功能已经对外宣称在某年某日发布
可以事先发到生产环境，只对内部用户打开，测试没问题后按时对全部用户开放

实验开关应该也是短暂存在的，一旦实验结束了需要及时清除实验开关代码。

4.1.3 运维开关

运维开关通常用于提升系统稳定性，比如：

1）大促前可以把一些非关键功能关闭来提升系统容量
2）当系统出现问题时可以关闭非关键功能来保证核心功能正常工作

运维开关可能会长期存在，而且一般会涉及多个系统，所以需要提前规划。

4.2 服务治理

4.2.1 限流

服务就像高速公路一样，在正常情况下非常通畅，不过一旦流量突增（比如大促、遭受 DDOS

攻击）时，如果没有做好限流，就会导致系统整个被冲垮，所有用户都无法访问。

正常的高速公路

超出容量的高速公路

所以我们需要限流机制来应对此类问题，一般的做法是在网关或 RPC 框架层添加限流逻辑，结合配置中心的动态推送能力实现动态调整限流规则配置。

4.2.2 黑白名单

对于一些关键服务，哪怕是在内网环境中一般也会对调用方有所限制，比如：

1）有敏感信息的服务可以通过配置白名单来限制只有某些应用或 IP 才能调用
2）某个调用方代码有问题导致超大量调用，对服务稳定性产生了影响，可以通过配置黑名单来暂时屏蔽这个调用方或 IP

一般的做法是在 RPC 框架层添加校验逻辑，结合配置中心的动态推送能力来实现动态调整黑白名单配置。

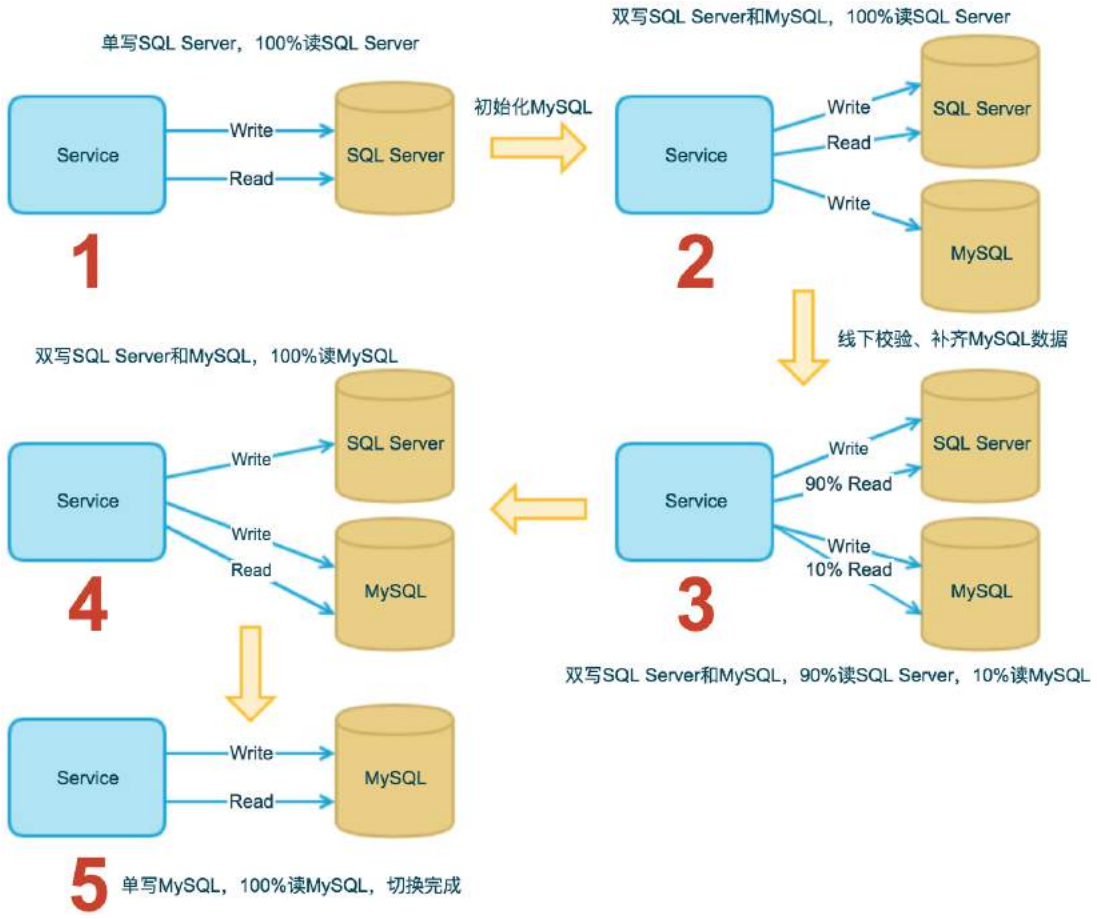
4.3 数据库迁移

数据库的迁移也是挺普遍的，比如：原来使用的 SQL Server，现在需要迁移到 MySQL，这种情况就可以结合配置中心来实现平滑迁移：

1）单写 SQL Server，100%读 SQL Server

2）初始化 MySQL
3）双写 SQL Server 和 MySQL，100%读 SQL Server
4）线下校验、补齐 MySQL 数据
5）双写 SQL Server 和 MySQL，90%读 SQL Server，10%读 MySQL
6）双写 SQL Server 和 MySQL，100%读 MySQL
7）单写 MySQL，100%读 MySQL
8）切换完成

上述的读写开关和比例配置都可以通过配置中心实现动态调整。



4.4 动态日志级别

服务运行过程中，经常会遇到需要通过日志来排查定位问题的情况，然而这里却有个两难：

1）如果日志级别很高（如：ERROR），可能对排查问题也不会有太大帮助；
2）如果日志级别很低（如：DEBUG），日常运行会带来非常大的日志量，造成系统性能下降；

为了兼顾性能和排查问题，我们可以借助于日志组件和配置中心实现日志级别动态调整。

以 Spring Boot 和 Apollo 结合为例：

```
@ApolloConfigChangeListener
private void onChange(ConfigChangeEvent changeEvent) {
    refreshLoggingLevels(changeEvent.changedKeys());
}

private void refreshLoggingLevels(Set<String> changedKeys) {
    boolean loggingLevelChanged = false;
    for (String changedKey : changedKeys) {
        if (changedKey.startsWith("logging.level.")) {
            loggingLevelChanged = true;
            break;
        }
    }

    if (loggingLevelChanged) {
        // refresh logging levels
        this.applicationContext.publishEvent(new EnvironmentChangeEvent(changedKeys));
    }
}
```

详细样例代码可以参考: https://github.com/ctripcorp/apollo-use-cases/tree/master/spring-cloud-logger

4.5 动态网关路由

网关的核心功能之一就是路由转发，而其中的路由信息也是经常会需要变化的，我们也可以结合配置中心实现动态更新路由信息。

以 Spring Cloud Zuul 和 Apollo 结合为例:

```
@ApolloConfigChangeListener
public void onChange(ConfigChangeEvent changeEvent) {
    boolean zuulPropertiesChanged = false;
    for (String changedKey : changeEvent.changedKeys()) {
        if (changedKey.startsWith("zuul.")) {
            zuulPropertiesChanged = true;
            break;
        }
    }

    if (zuulPropertiesChanged) {
        refreshZuulProperties(changeEvent);
    }
}
```

```
    private void refreshZuulProperties(ConfigChangeEvent changeEvent) {
        // rebind configuration beans, e.g. ZuulProperties
        this.applicationContext.publishEvent(new
EnvironmentChangeEvent(changeEvent.changedKeys()));

        // refresh routes
        this.applicationContext.publishEvent(new RoutesRefreshedEvent(routeLocator));
    }
```

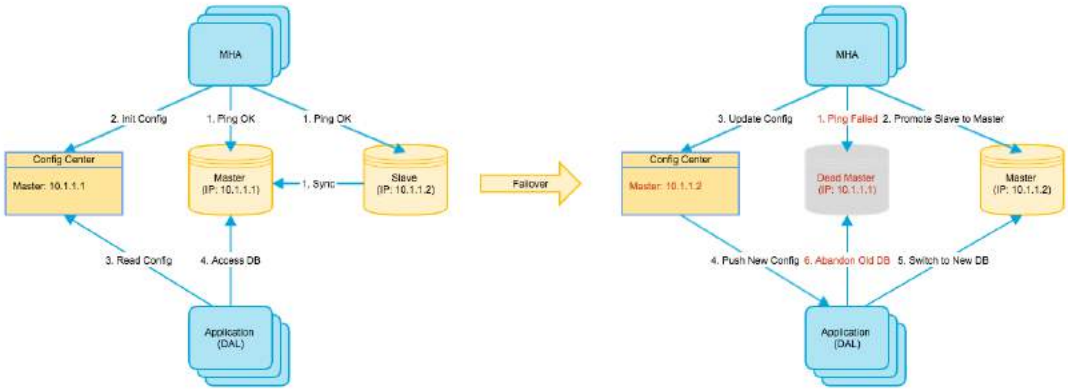
详细样例代码可以参考: https://github.com/ctripcorp/apollo-use-cases/tree/master/spring-cloud-zuul

4.6 动态数据源

数据库是应用运行过程中的一个非常重要的资源，承担了非常重要的角色。

在运行过程中，我们会遇到各种不同的场景需要让应用程序切换数据库连接，比如：数据库维护、数据库宕机主从切换等。

切换过程如下图所示:



以 Spring Boot 和 Apollo 结合为例:

```
@Configuration
public class RefreshableDataSourceConfiguration {

    @Bean
    public DynamicDataSource dataSource(DataSourceManager dataSourceManager) {
        DataSource actualDataSource = dataSourceManager.createDataSource();
        return new DynamicDataSource(actualDataSource);
    }
```

```
}

public class DynamicDataSource implements DataSource {
    private final AtomicReference<DataSource> dataSourceAtomicReference;

    public DynamicDataSource(DataSource dataSource) {
        dataSourceAtomicReference = new AtomicReference<>(dataSource);
    }

    // set the new data source and return the previous one
    public DataSource setDataSource(DataSource newDataSource){
        return dataSourceAtomicReference.getAndSet(newDataSource);
    }

    @Override
    public Connection getConnection() throws SQLException {
        return dataSourceAtomicReference.get().getConnection();
    }

    ...
}

    @ApolloConfigChangeListener
    public void onChange(ConfigChangeEvent changeEvent) {
        boolean dataSourceConfigChanged = false;
        for (String changedKey : changeEvent.changedKeys()) {
            if (changedKey.startsWith("spring.datasource.")) {
                dataSourceConfigChanged = true;
                break;
            }
        }

        if (dataSourceConfigChanged) {
            refreshDataSource(changeEvent.changedKeys());
        }
    }

    private synchronized void refreshDataSource(Set<String> changedKeys) {
        try {
            // rebind configuration beans, e.g. DataSourceProperties
            this.applicationContext.publishEvent(new EnvironmentChangeEvent(changedKeys));

            DataSource newDataSource = dataSourceManager.createAndTestDataSource();
            DataSource oldDataSource = dynamicDataSource.setDataSource(newDataSource);
```

```
            asyncTerminate(oldDataSource);
        } catch (Throwable ex) {
            logger.error("Refreshing data source failed", ex);
        }
    }
```

详细样例代码可以参考:
https://github.com/ctripcorp/apollo-use-cases/tree/master/dynamic-datasource

五、最佳实践

5.1 公共组件的配置

公共组件是指那些发布给其它应用使用的客户端代码，比如 RPC 客户端、DAL 客户端等。

这类组件一般是由单独的团队（如中间件团队）开发、维护，但是运行时是在业务实际应用内的，所以本质上可以认为是应用的一部分。

这类组件的特殊之处在于大部分的应用都会直接使用中间件团队提供的默认值，少部分的应用需要根据自己的实际情况对默认值进行调整。

比如数据库连接池的最小空闲连接数量（minimumIdle），出于对数据库资源的保护，DBA 要求将全公司默认的 minimumIdle 设为 1，对大部分的应用可能都适用，不过有些核心/高流量应用可能觉得太小，需要设为 10。

针对这种情况，可以借助于 Apollo 提供的 Namespace 实现:

1)中间件团队创建一个名为 dal 的公共 Namespace，设置全公司的数据库连接池默认配置;

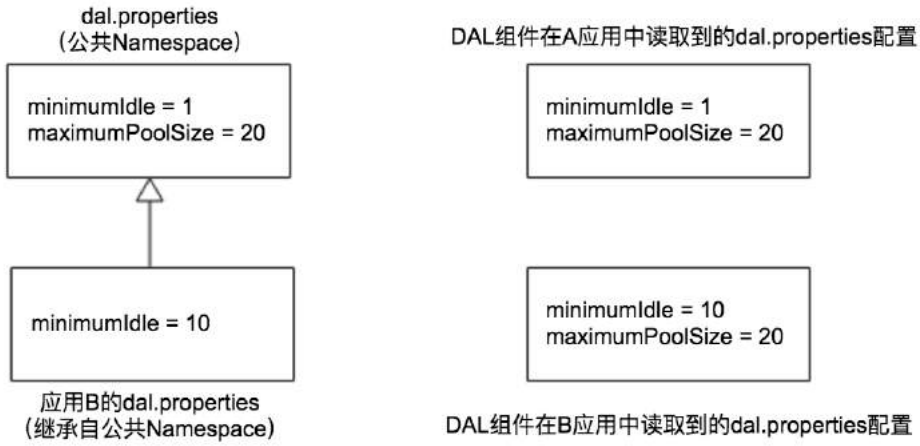
```
minimumIdle = 1
maximumPoolSize = 20
```

2）dal 组件的代码会读取 dal 公共 Namespace 的配置;

3）对大部分的应用由于默认配置已经适用，所以不用做任何事情;

4）对于少量核心/高流量应用如果需要调整 minimumIdle 的值，只需要关联 dal 公共 Namespace，然后对需要覆盖的配置做调整即可，调整后的配置仅对该应用自己生效;

```
minimumIdle = 10
```

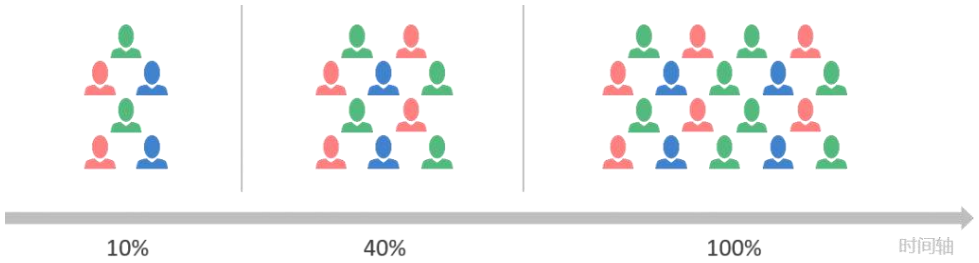


通过这种方式的好处是不管是中间件团队，还是应用开发，都可以灵活地动态调整公共组件的配置。

5.2 灰度发布

对于重要的配置一定要做灰度发布，先在一台或多台机器上生效后观察效果，如果没有问题再推给所有的机器。

对于公共组件的配置，建议先在一个或多个应用上生效后观察效果，没有问题再推给所有的应用。



5.3 发布审核

生产环境建议启用发布审核功能，简单而言就是如果某个人修改了配置，那么必须由另一个人审核后才可以发布，以避免由于头脑不清醒、手一抖之类的造成生产事故。

六、结语

本文主要介绍了以下几方面：

6.1 为什么需要配置中心？

配置即『控制』
配置需要治理
微服务带来的配置复杂性

6.2 配置中心的一般模样

以 Apollo 为例子，介绍了配置中心所具备的特征
介绍了 Apollo 是如何实现高可用和实时性的

6.3 如何让微服务更『智能』？

通过几个案例，分享了如何借助于配置中心使微服务更『智能』

6.4 配置中心的最佳实践

公共组件的配置
灰度发布
发布审核

最后，希望大家在平时工作中都能用好配置中心，更好地服务于业务场景，使微服务更『智能』，实现从青铜到王者的跨越！

携程基于云的软呼叫中心及客服平台架构实践

[作者简介]蒲成，携程云客服平台研发部资深研发经理。2015 年底加入携程从事呼叫中心相关产品的研发工作，主导建设了携程呼叫中心智能语音平台、统一配置中心，目前正在努力推进云客服平台的设计研发工作。本文来自蒲成在“2018 携程技术峰会”上的分享。

一、背景及设计理念

自携程创立以来，呼叫中心就一直伴随着公司业务一同发展壮大。经过近 20 年的迭代，目前携程的呼叫中心系统已经演进为第五代呼叫中心系统了，也就是我们完全自主研发的基于 FreeSwitch 的软交换与 IVR、微信 Server、邮件系统、无线 IM Server 的全渠道全媒体客服系统。

那么，基于现有可扩展架构的这套客服系统为携程的客服业务提供了什么样的支撑呢？我们可以从以下几个方面一窥全貌。

- 多渠道

目前支持传统电话、VOIP 电话、IM、微信公众号、邮件等通信渠道的接入。

- 多地域

目前携程的客服坐席分布在全国及海外各地，其中包括国内的上海、南通、合肥、如皋、信阳，以及海外的爱丁堡、韩国、日本等地。

- 多业务

本系统目前支撑着携程 400+场景以及 15000+坐席的服务业务落地。

- 多语种

目前提供中文、英语、日语、韩语、法语、俄语等多语种支持。

- 海量会话

目前电话日均通话量约 100 万通以上，而 IM 会话日均消息量约 1000 万条以上。

上述场景的背后是一套什么样的架构体系在提供服务支撑呢？我们又为何会选择建设这样一套架构体系呢？后文将给出答案。

传统的客服运营通常面临六大痛点，即沟通单一、信息碎片化、智能化程度低、效率低下、移动性不足、成本高昂。在企业发展壮大的过程中，传统的客服运营就逐渐成为制约企业业务发展的瓶颈。有鉴于此，我们研发了一套基于云和容器化的软呼叫中心及客服平台，并且引入了场景化的 AI 能力，从而在源头上消除了前面所说的六大痛点。

现在，我们的客服系统是这样的：

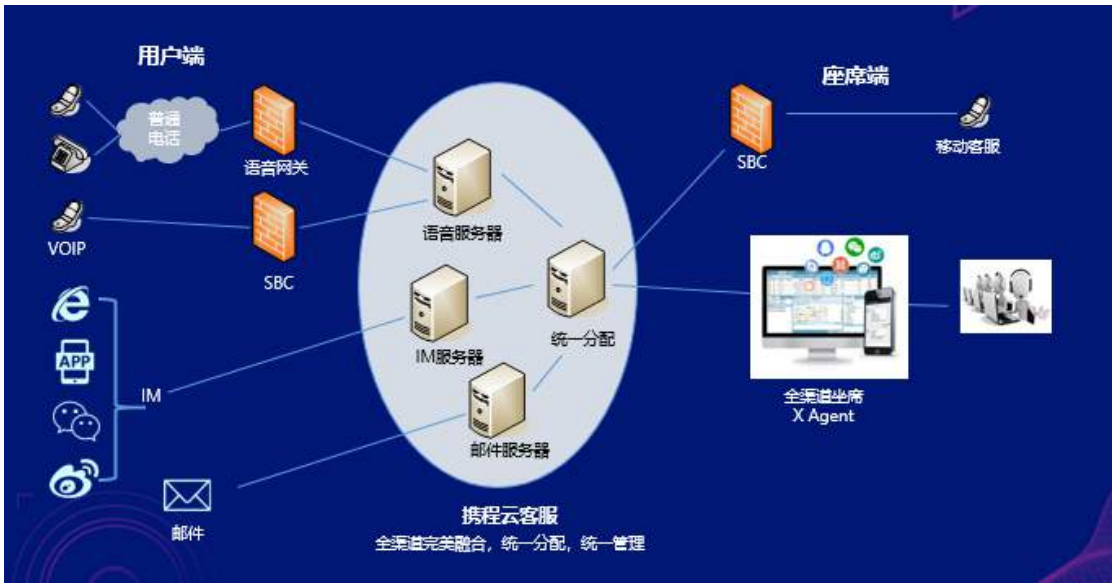
云客服平台 = 软交换云平台（公有云/私有云）

+全渠道座席（Call/Chat/IM/SNS)
+全媒体座席（Voice/Txt/Pic/Video）
+多模式（集中/在家/移动）
+AI 引擎（客服机器人/语义解析…）
+CRM、工单系统、知识库

二、核心架构

2.1 系统结构

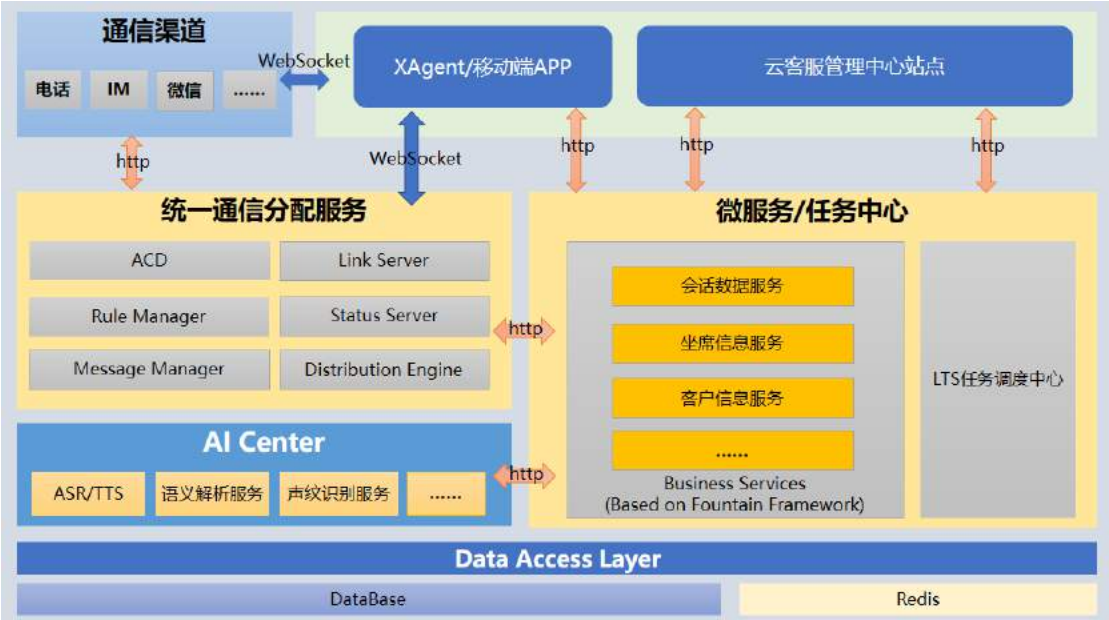
我们先来看看整体的系统结构，如下图所示。



从上图可以看出我们云客服平台的整体链路结构，其中最核心的就是中间的渠道服务和通信分配层，这一层中的每个节点都可按需进行水平扩展，从而支撑未来的业务发展。

通过这一中间层的转换，我们就将上图左侧来自各个渠道的客人服务请求整合为统一的服务请求，并通过右侧的全渠道坐席界面统一分配给客服人员进行服务响应。这样一来，也就实现了多个通信渠道融合的目的。下一节我们来看看其背后的处理逻辑。

2.2 逻辑架构



通信渠道由我们自研的各渠道 Server 构成，其中也包括无线平台研发部所研发的 IM Server。坐席所使用的全渠道通信端（XAgent/APP）使用 WebSocket 协议与这些渠道 Server 保持通信，同时也使用 WebSocket 协议与统一通信分配服务保持通信。

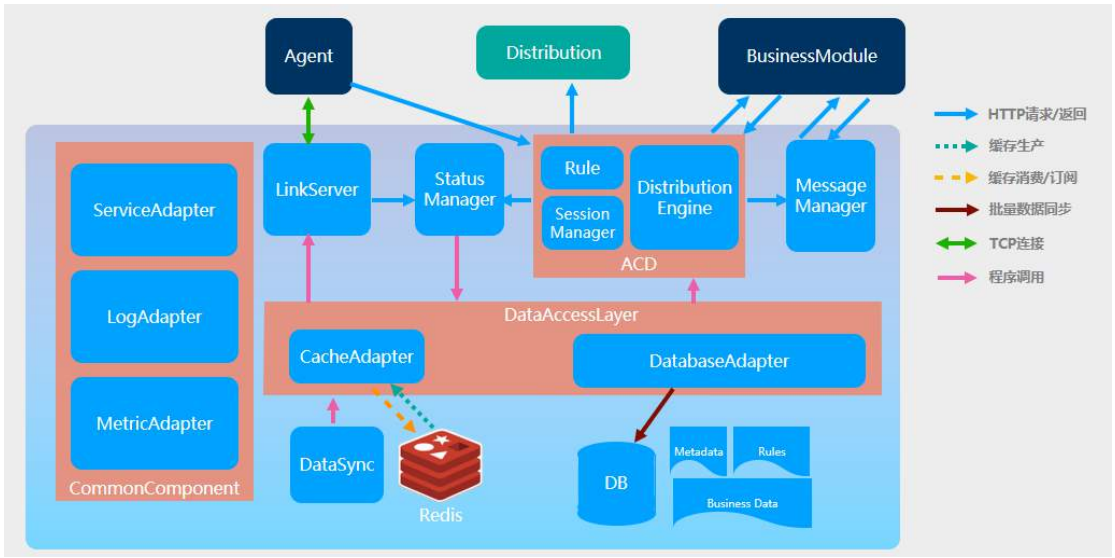
其余诸如分配服务、业务数据服务、AI 能力服务等，均以微服务 API 的方式在平台内部暴露。为此，我们搭建了一套名为方塔尖的微服务框架来提供基础设施的支持。

2.3 方塔尖微服务框架

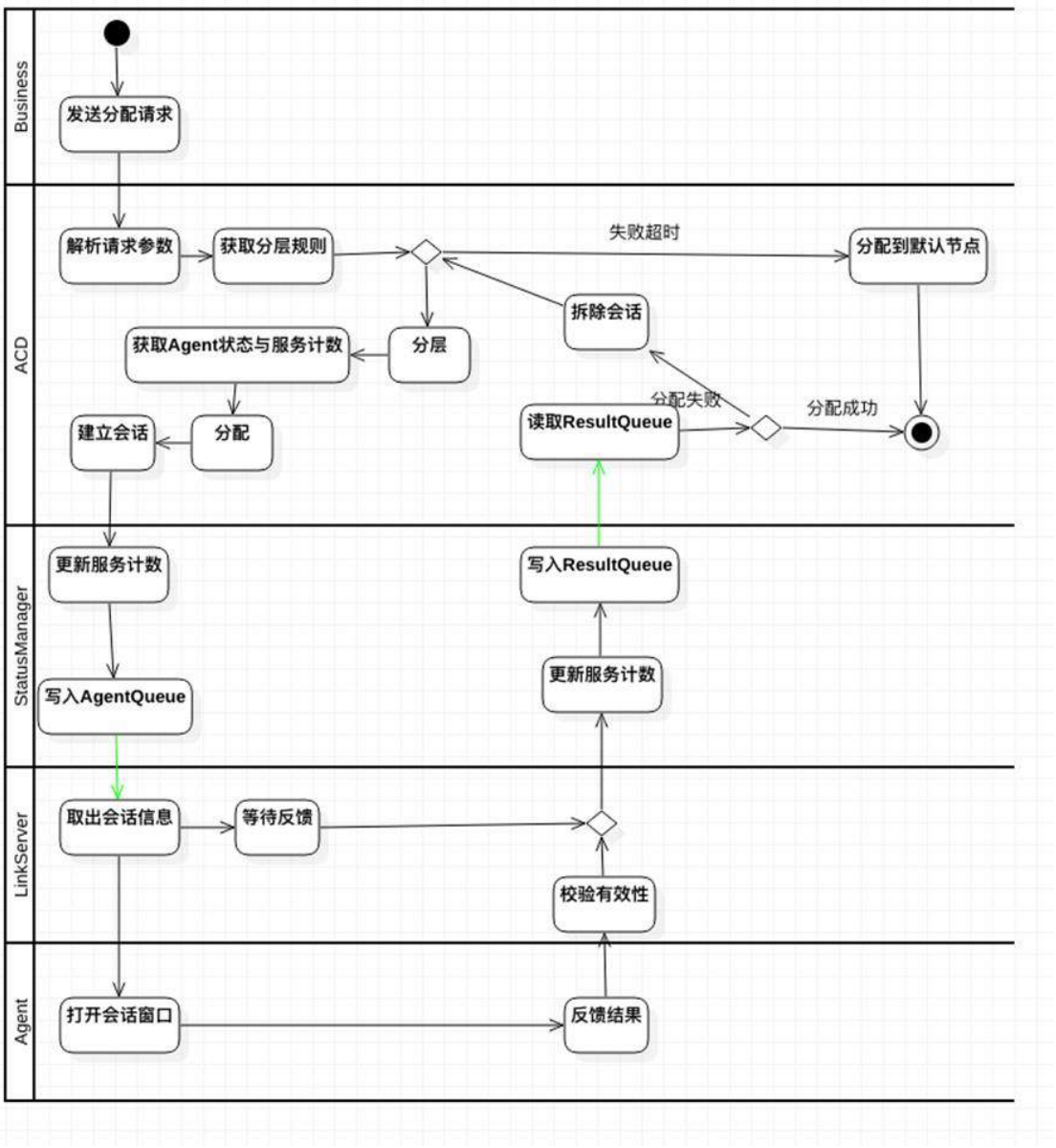
这套框架是基于 SpringCloud 搭建的，分别采用 consul、zuul 来实现服务发现和服务路由。此外，在方塔尖中我们还加入了一些功能级服务，比如用户/权限管理、短信验证码、数据加解密、数据访问层封装等等，以便让其上的逻辑层仅关注业务实现即可。

2.4 统一分配

下面我们来看看核心的统一通信分配服务的实现，其架构如下：

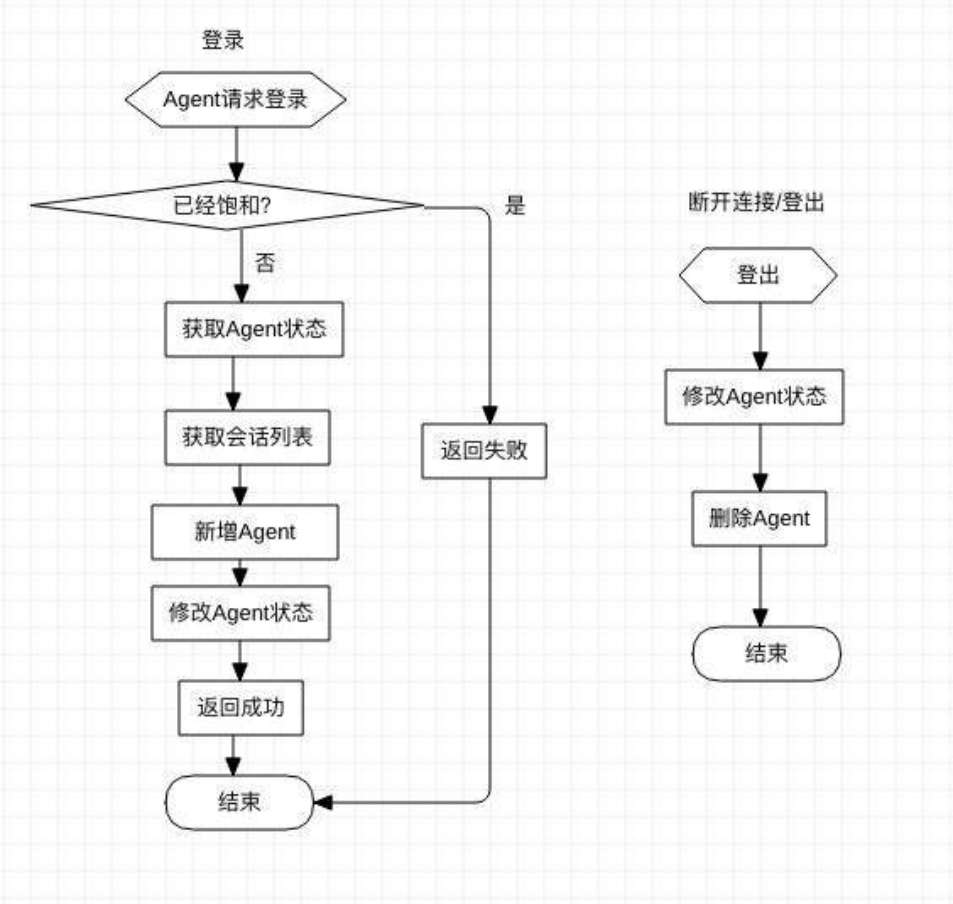


顾名思义，这个核心组件的目标就是实现各通信渠道的会话统一分配，其核心逻辑如下：

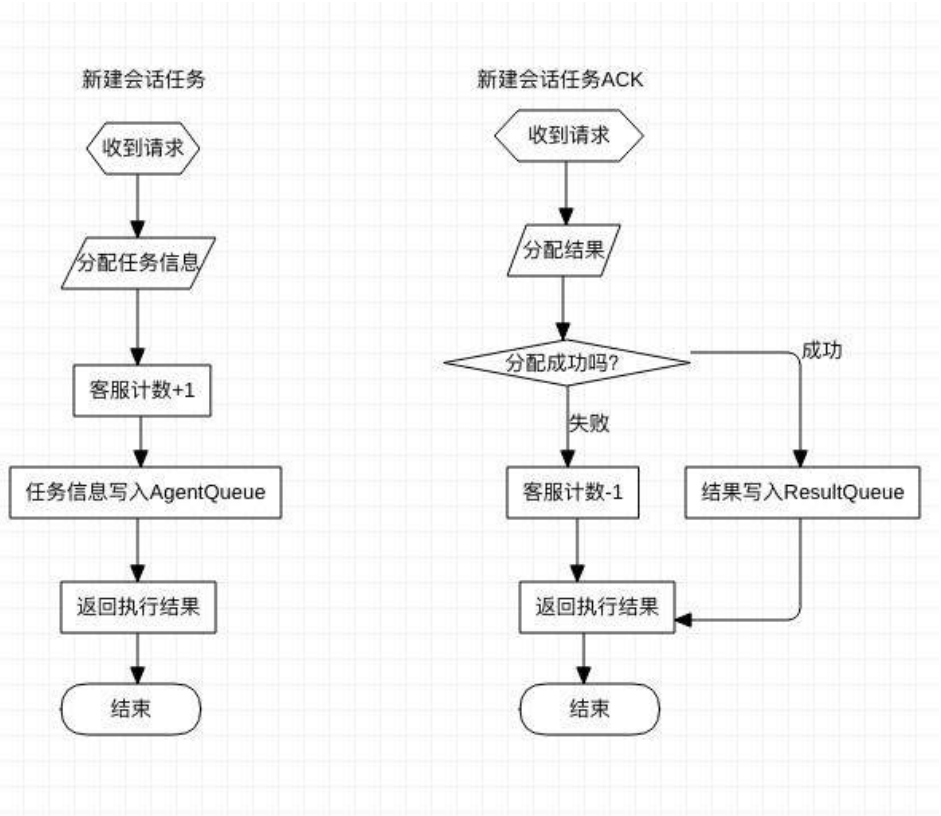


LinkServer 是坐席服务端，坐席端通过 WebSocket 连接到 LinkServer。

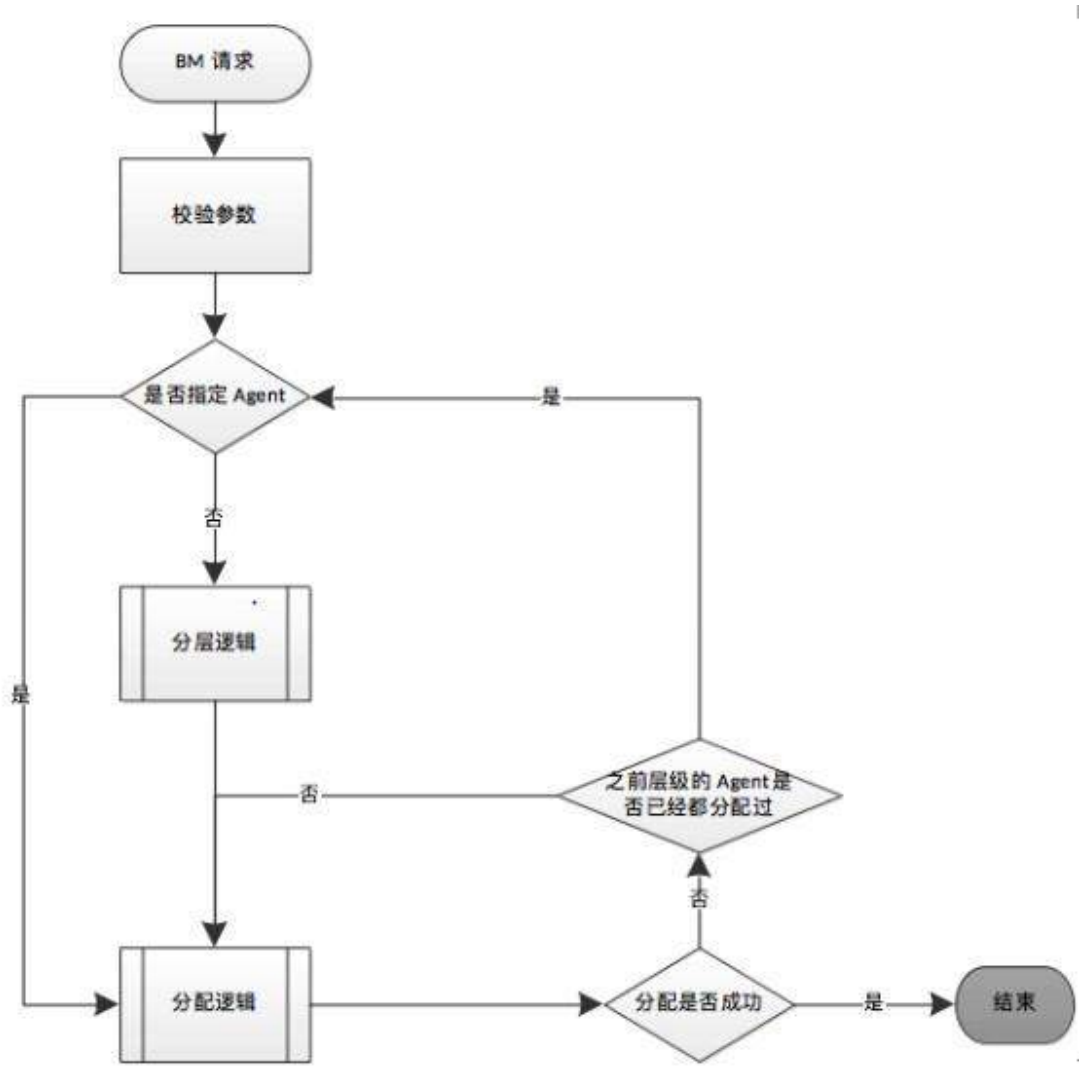
LinkServer 负责维护坐席连接、收发坐席请求和反馈、传递坐席状态。其处理流程如下:



StatusManager 是状态管理服务，负责处理 LinkServer 传递来的坐席状态变化，负责对外提供坐席状态查询。其处理流程如下：



ACD 是 IM+系统的核心模块，其主要功能是实现客人坐席分配，ACD 指令和消息的收发、ACD 会话管理等。其处理流程如下：



其中的分配逻辑是基于抽象的业务规则表达式来进行处理的，为此，我们采用了开源的表达式运算器 EvalEx，其好处在于：

- 使用 BigDecimal 进行计算和返回结果
- 不依赖于外部库
- 可以设置精度和舍入模式
- 支持变量
- 支持标准布尔和数学运算符
- 支持标准的基本数学和布尔函数
- 可以在运行时添加自定义函数和操作符
- 函数可以用变量数量的参数来定义(参见最小和最大函数)
- 支持十六进制数字和科学的数字符号
- 支持函数中的字符串文字
- 支持隐式乘法，例如(a+b)(a-b)或 2(x-y)，等于(a+b)*(a-b)或 2*(x-y)

基于此，我们提供了一些基础分配逻辑，并且也支持第三方分配逻辑的对接。

- 上次服务优先

优先分配给上次服务的客服

- 熟客优先

优先分配给为该客户服务次数最多的客服

- 均衡分配

按客服工作量平均分配

- 最闲分配

优先分配给空闲最长时间的客服

- 指定分配

指定分配给某几个客服

- 第三方分配

调用第三方接口分配

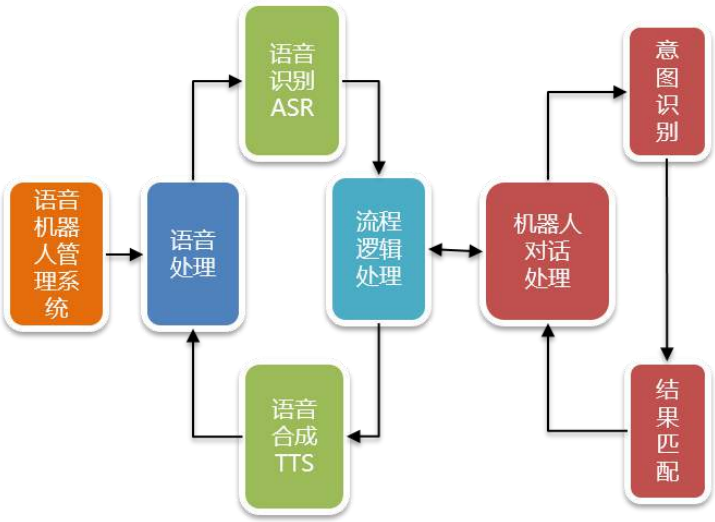
2.5 智能化

人工智能现在很火，但是在人工智能众多细分领域中，其实 NLP 技术的发展和应用才是人工智能“皇冠上的明珠”，它也是众多 AI 大厂持续投入的领域。

而就目前的市场环境和技术条件而言，客服业务的智能化是最有希望落地 NLP 技术的场景。因此，我们也着力构建了云客服平台的智能化应用框架。该框架结构如下：

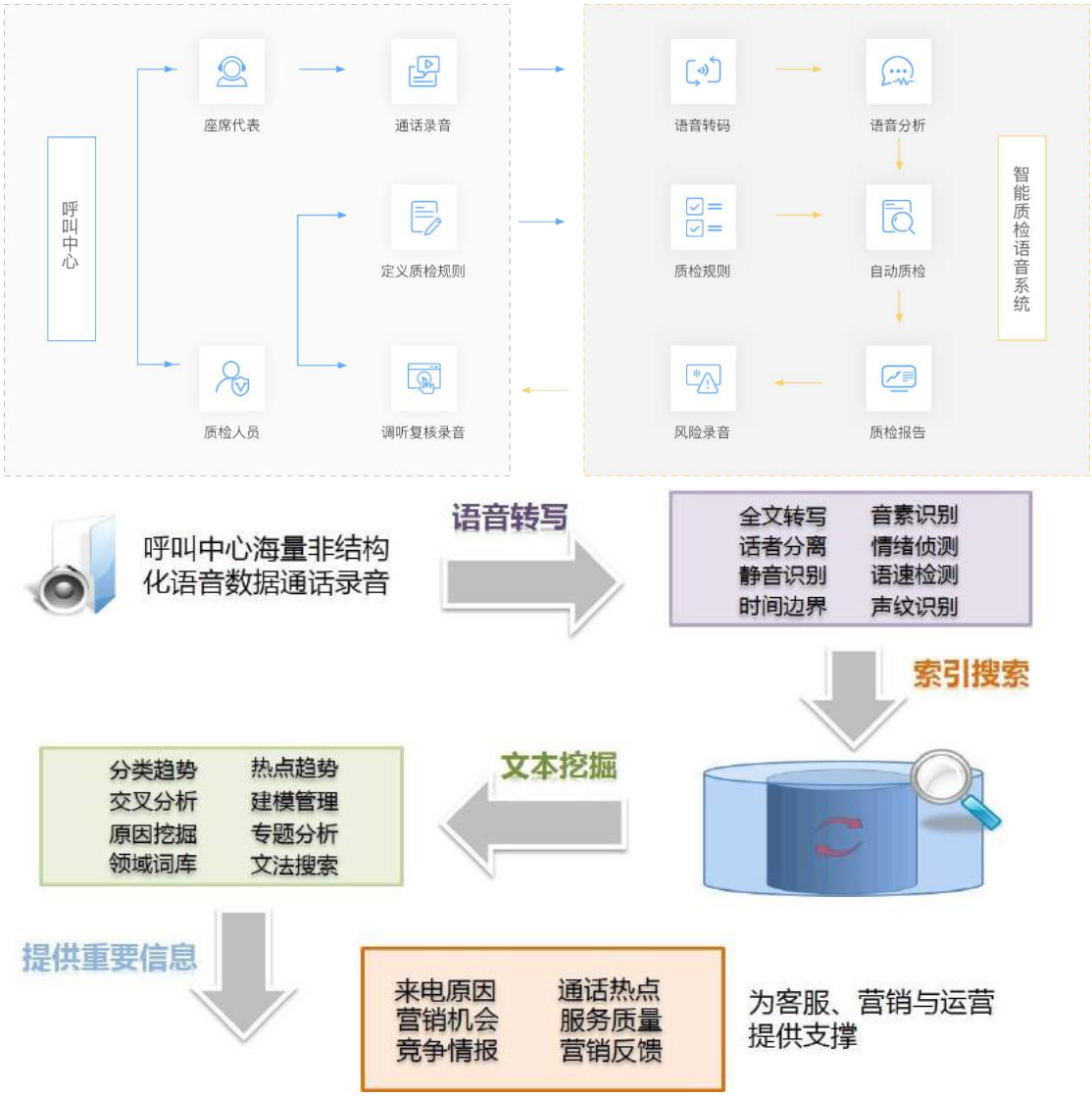
其中智能质检和对话机器人是两大重点应用场景，这两个场景的落地能够极大地提升客服业务运营效率并且显著降低运营成本。

对话机器人在我们的客服平台中分为语音机器人和在线 IM 机器人。语音机器人的服务对象是 IVR（交互式语音应答），即电话的呼入呼出 IVR 场景。其处理流程如下：



在线 IM 机器人主要对接的是 IM、微信等即时通信和社交媒体渠道，从广义上可以理解为我们常见的聊天机器人范畴，只不过在客服系统中，其模型是针对专有业务场景进行训练的。因此，相较于通用聊天机器人，在线 IM 机器人其实更容易达到比较好的智能交互效果。其整体模块结构如下：

智能质检对于客服运营管理而言是一项非常重要的功能，借助 ASR 语音转文字的能力，它能将非结构化的音频、文本数据转换成客服运营甚至企业运营统计分析所需的结构化数据，最终形成对业务管理运营的良性反馈闭环。下面两张图分别是我们云客服平台中智能质检的场景顺序图和处理流程图。



三、平台级能力输出

客服系统不同面向 C 端的应用，我们的目标并非寻求用户的长时间驻留。相反，在客服领域，我们希望能够以最快的时间去响应客户的需求，这样才能提升客户满意度并最大限度降低运营成本。所以，我们客服平台的每个模块、每项功能都是围绕这一主旨而设计构建的。

那么，基于前面的核心基础架构和上述考量，我们的客服平台能够对外输出哪些能力呢？

3.1 外呼

外呼通常是呼叫中心会高频使用的业务场景，传统的外呼都是坐席人工发起外呼，费时费力且成本高昂。因此，我们围绕外呼应用研发了四种外呼形态，以满足不同业务场景的需要。这四种外呼形态是，自动外呼、预测外呼、预览外呼和智能外呼。

篇幅所限，就不一一讲解每种形态的具体特性了，但它们的核心都是以自动外呼系统为基础的，其业务处理简图如下：

以此为基础，结合云端基础设施和容器特性，辅以我们自研的各种组件，我们的外呼系统就能提供以下特性：

- 支持高并发，吞吐能力可扩展
- 高可靠外呼平台，包含完善的保护机制
- 支持预测外呼、预约外呼、虚拟坐席外呼
- 提供智能呼叫算法，提升工作效率

3.2 中转

中转即号码埋名，也就是用虚拟号码替换真实号码的功能。这项功能的目的是让通话双方无法获悉对方的真实号码，从而实现隐私保护的目的。

作为该能力的配套，我们开发了配置界面、录音模块，以及对应的查询/统计报表等功能，用户可以基于浏览器操作完成实时的配置生效操作，并浏览话务中转结果。

3.3 VOIP

VOIP 也就是大家所熟知的 IP 网络电话。我们的平台提供了 VOIP SDK，方便第三方应用集成，并且自研了音频编解码和动态码率技术，能够满足弱网下的正常语音通信。其特性如下

图所示:

3.4 全渠道客服工作台

我们为客服人员提供了一站式全渠道的客服工作台，以便客服人员可以在统一的工作界面中为来自各个渠道的客人需求提供服务响应。其特性如下:

全渠道统一，一站式服务体验

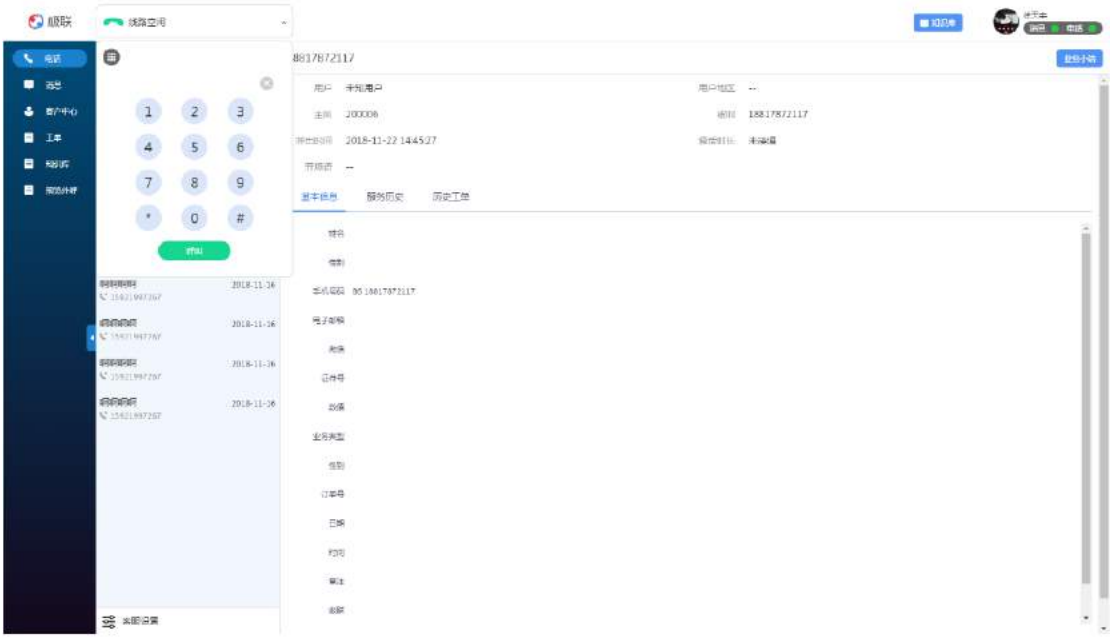
- 统一电话、网页 IM、APPIM、邮件、微信等多个渠道
- 快速响应用户咨询

全媒体融合，服务形式更丰富

- 文本、图片、电话、语音、视频等多媒体融合
- 服务形式多样化

数据整合，一目了然

- 完整用户信息与服务轨迹记录
- 提升服务质量



3.5 客户支持

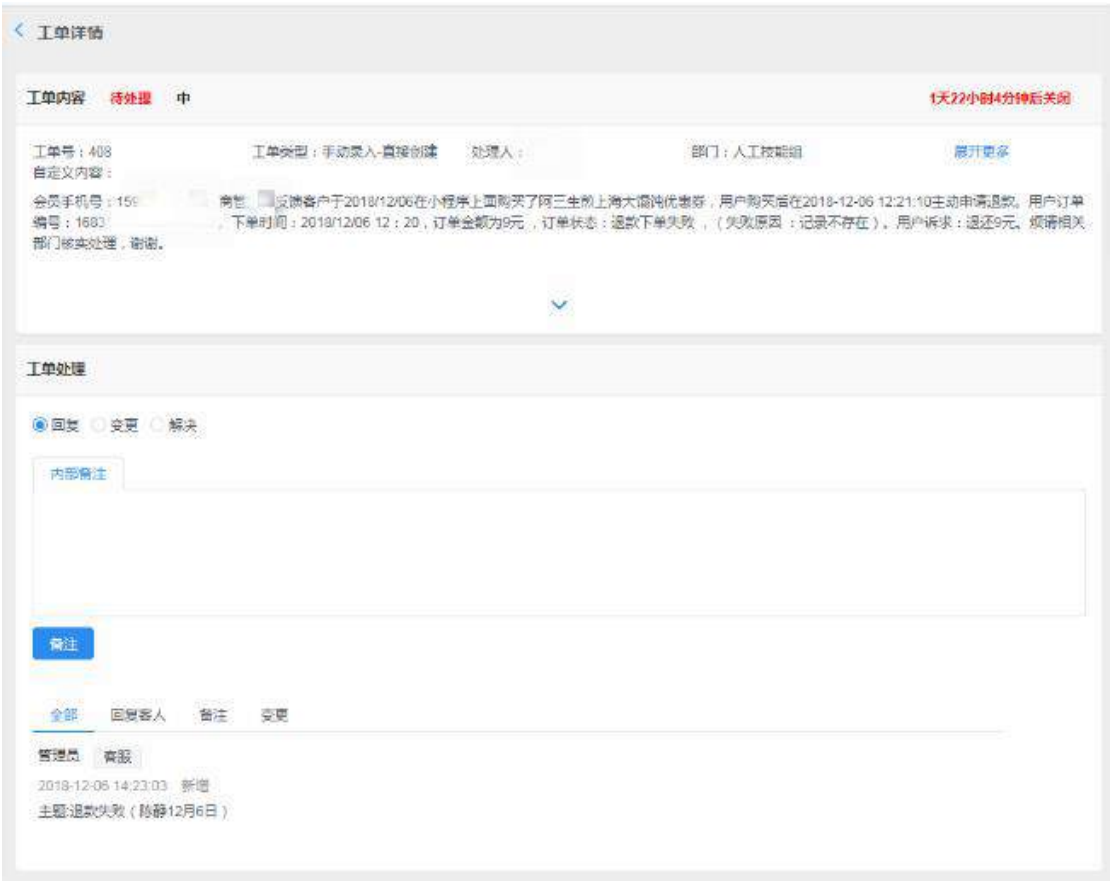
我们的客服平台在核心客服业务能力支持之外还提供了以客户为中心的服务周边配套模块，比如 CRM、工单、知识库等。这是为了给客服人员提供更为详尽的客户信息，以期为客户提供全方位的服务支持。

CRM

客户数据管理

- 完整记录用户数据
- 用户信息统一管理
- 提高客户留存率
- 提升企业获客能力

工单



速流转，协同处理

- 促进企业内外协作，共同处理用户咨询
- 企业服务更高效

知识库

座席服务规范高效

- 常用问题
- 常用语
- 客服话术标准化

3.6 报表监控

作为客服业务运营的日常管理手段，报表和监控是必不可少的支持方式。我们的客服平台自然也提供了相应的运营报表和监控界面。具体特性如下：

实时报表

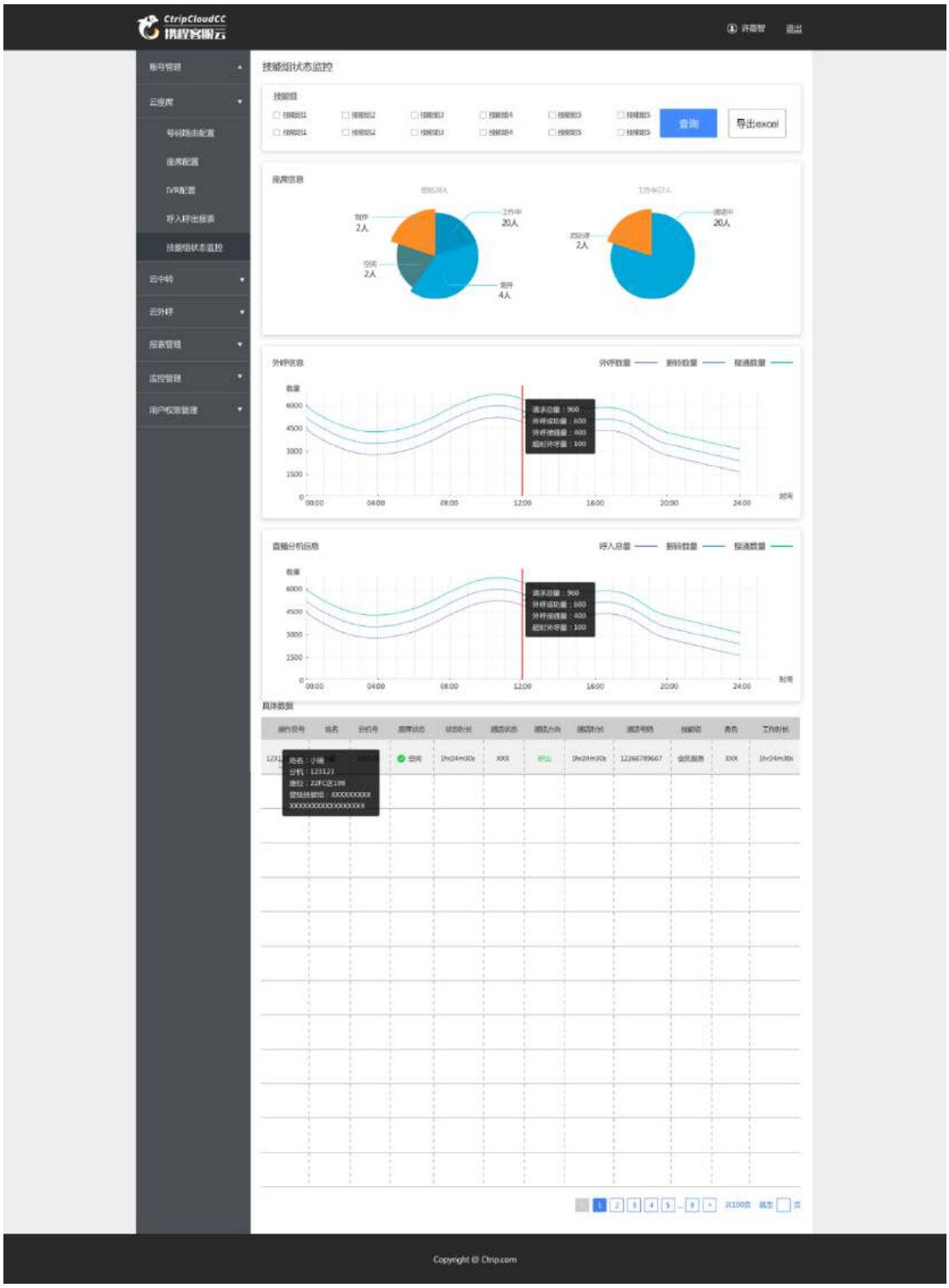
- 多维度、实时展示座席指标数据
- 可按日、周、月等多条件查询
- 图表展示方式支持自定义

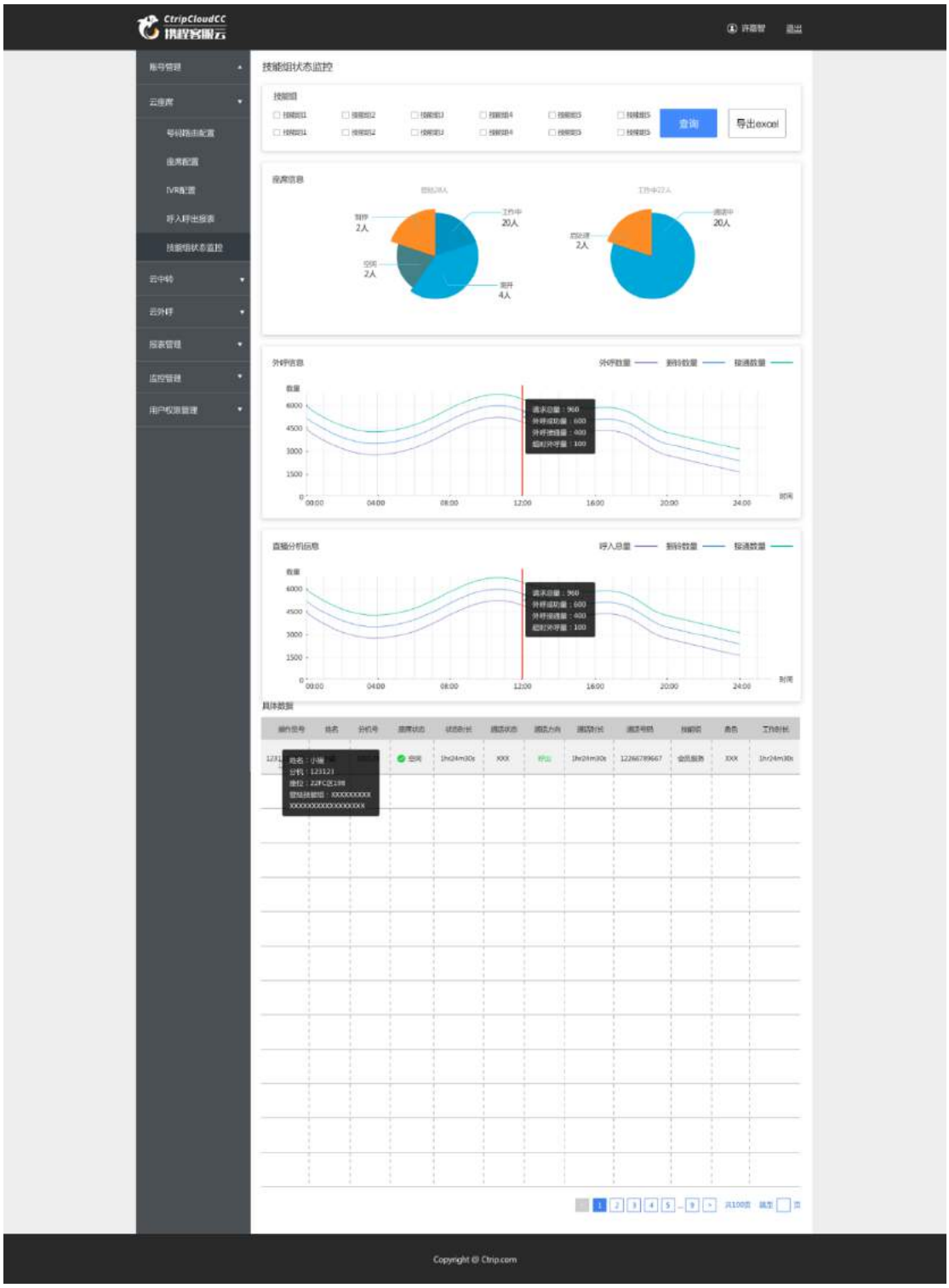
全面监控预警

- 系统、服务、座席，全方位监控
- 可设阈值告警与告警通知

话务预测

- 智能预测后续话务量
- 突发事件等因子对话务量的影响预测


CtripCloudCC
携程客服云
技能组状态监控
查询
导出excel
Copyright © Ctrip.com


CtripCloudCC
携程客服云
技能组状态监控
查询
导出excel
外呼信息
直播分析后息
具体数据
Copyright © Ctrip.com

四、结语

客服平台是异常复杂和庞大的结构化体系平台，要在一篇文章中全面论述其技术体系架构几乎是不可能完成的任务。受篇幅限制，本文仅摘取了部分核心架构和核心模块功能略作阐述。如果读者有兴趣了解更多与我们客服平台有关的信息，欢迎在技术中心微信公众号和我们互动。

携程 Redis 容器化实践

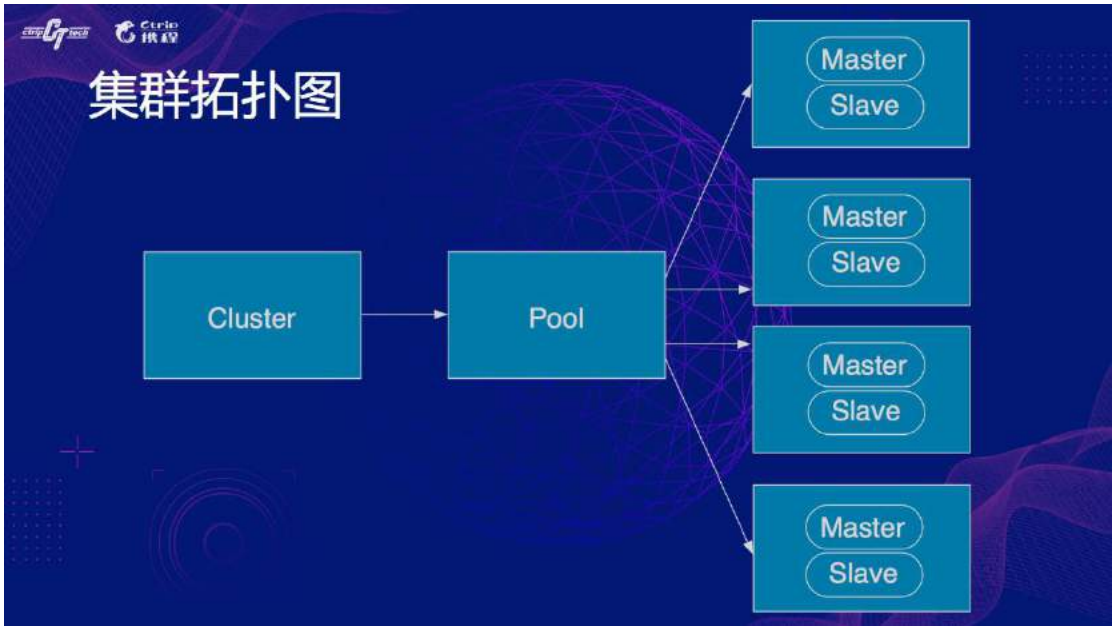
[作者简介]李剑，携程 CIS 资深软件工程师。加入携程之前主要从事音视频流媒体的开发，目前主要负责 Redis 和 Mysql 容器化和服务化的研发。本文来自李剑在“2018 携程技术峰会”上的分享。

携程的 Redis 使用规模有 200T+，并且每天有百万亿次的访问频率，如此大规模的 Redis 容器化对于我们来说是个不小的挑战，本文分享携程 Redis 容器化落地的一些实践经验。

一、背景

携程大部分应用是基于 CRedis 客户端通过集群来访问到实际的 Redis 的实例，集群是访问 Redis 的基本单位，多个集群对应一个 Pool，一个 Pool 对应一个 Group，每个 Group 对应一个或多个实例，Key 是通过一致性 hash 散列到每个 Group 上，集群拓扑图如截图所示。

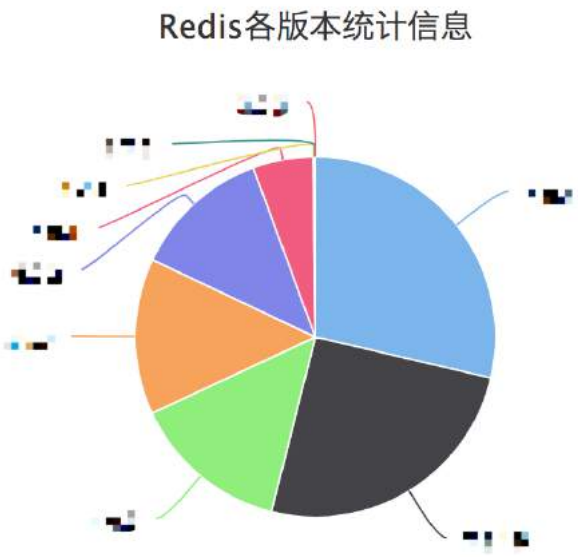
这个图里面我们可以看到集群，Pool，Group 还有里面的实例，这是携程 Redis 一个比较常见的拓扑图，如下图：



1.1 为什么要容器化

- 标准化和自动化

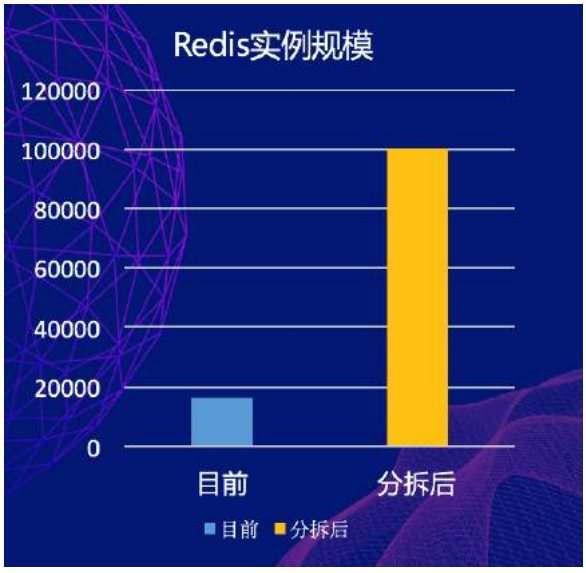
Redis 之前是直接部署在物理机上，而 DBA 是根据物理机上设定的 Redis 的版本来选择需要部署的物理机，携程的各个版本的 Redis 非常分散而且不容易维护，如下图所示，容器天然支持标准化，另外容器基于 K8S 自动化部署的效率，根据我们估算，相比人工部署提高了 59 倍。



- 规模化

有别于社区的方案比如官方 Redis Cluster 或代理方案而言，携程的技术演进方案需要对大的实例进行分拆（内部称为 CRedis 水平扩容），实例分拆后，单个实例的内存小了，QPS 降低，单个实例挂掉的影响小很多，可以说是利国利民的项目，但会带来一个问题，实例数急剧膨胀。容器化后我们能对分拆后的实例更好地管理和运维。

另外，分拆过程中需要大量中间状态的实例 Buffer 作为过渡，比如一对 60G 的实例分拆为 5G，中间状态的 Buffer 需要 24 个 60G 的实例，纯人工分拆异常艰难，而且容易出错，依靠容器自动调度生成实例会极大降低 DBA 分拆时的心智负担，极大提升了分拆的效率并减少出错的概率。



- 提高资源利用率

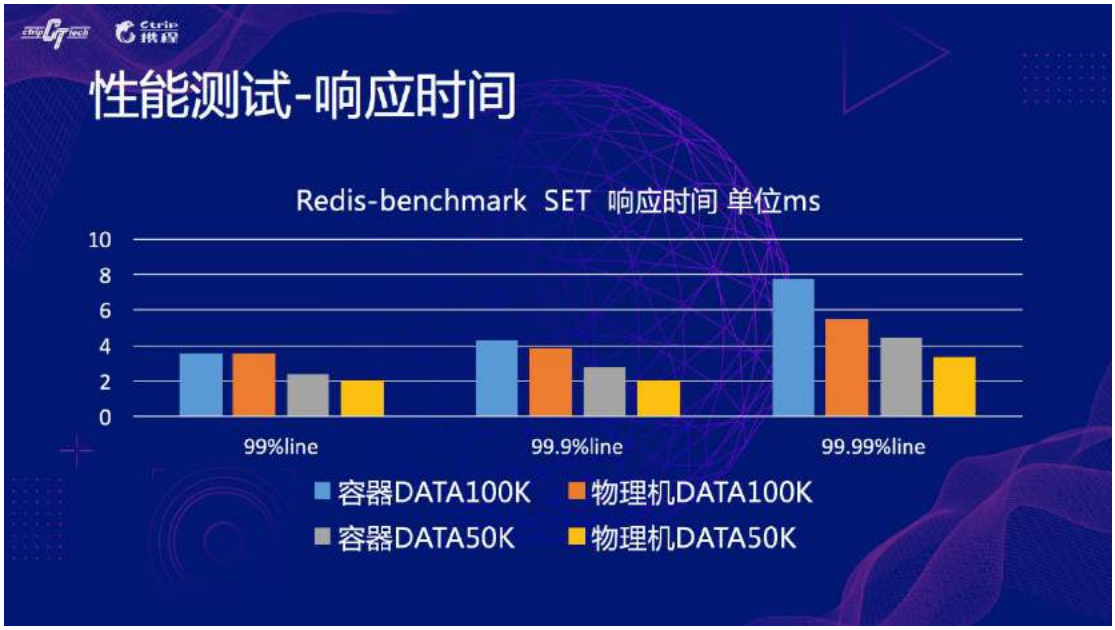
借助于容器化和上层的 K8S 的编排系统，我们很轻易的就可以做到资源利用率的提升，至于怎么做到的，后面细节部分会涉及。

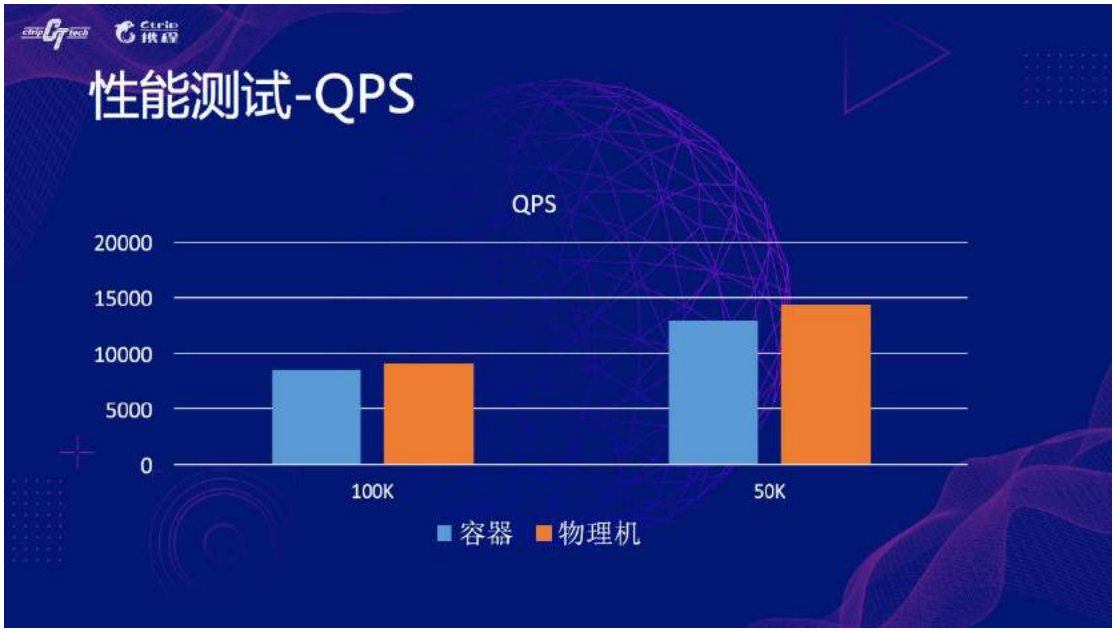
1.2 能不能容器化

既然 Redis 容器化后好处这么多，那么 Redis 能不能容器化呢？对比测试最能说明问题。

实际上我们在容器化前做了很多测试，甚至因为测试模式的细微差别在各个部门之间还有过长时间的争论，但最终下面这几张图的数据获得了大家的一致认可，容器化才得以继续推广下去。

我们 A/B 对比测试都是基于相同硬件的容器和物理机，不挂 slave，图上我们可以看到，Redis 的响应相比物理机要慢一点，QPS 也能看到差距很小，这些差异主要是容器化后经过多个虚拟网卡带来的性能损失。





这第三张图就更明显了，这是我们测试对比生产实际物理机的流量对比，我们测试的流量远高于生产实际运行的单台物理机的流量。

因此总结下来就是，容器与物理机的性能有细微的差别，大概 5-10%，并且携程的使用场景 Redis 完全可以容器化。

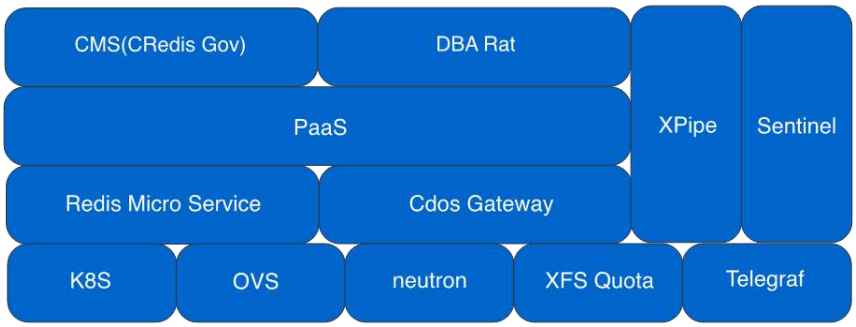
二、架构和细节

2.1 总体架构

以上介绍无非是容器化前的一些调研和可行性分析工作。

具体的架构如下图所示，首先最上层的是运维和治理工具 CRedis 和 Rat，这个在携程内部是属于框架和 DBA 两个部门，CRedis 不但提供应用访问 Redis 的客户端，本身也做 CMS 的工作，存储 Redis 实例最基本的元数据。

PaaS 层为 Credis/Rat 提供统一的 Redis Group/实例的创建删除接口，下面的 Redis 微服务提供实例申请具体的调度策略，基础设施有很多，这里其实只列举了一部分比较重要的，如网络相关的 ovs 和 neutron，与磁盘配额相关的 quota，以及监控相关的 telegraf 等。xpipe 是携程内部的跨 IDC 的 DR 方案，sentinel 就是官方的哨兵。



2.2 容器化遇到的一些问题

在我们容器化方案落地前遇到过一些具体的问题，例如：

1）Redis 实际上是被应用直连的，我们需要 IP 和宿主机固定，并且 master/slave 不能在一台宿主机上。

2）部署之前是在物理机上，通过端口来区分不同的实例，所有的监控通过端口来区分。

3）重启实例 Redis.conf 文件配置不能丢失，这个在容器之前甚至不算需求，但放在容器上就有点麻烦。

4）Master 挂了不希望 K8S 立刻把它拉起来，希望哨兵来感知到它，因为 K8S 如果在哨兵感知前拉起了它，导致哨兵还没切换 Master/Slave，Master 就活过来并且数据都丢失，这时候一同步到 Slave 上数据也全没有了，等于执行了一个清空操作，这对于业务和 DBA 来说是不能接受的。

5）实例几乎没有任何的内存控制，就是说实例不管写多大，都是得让 maxmemory 一直加上去，一直加到必须迁移走开始，再把实例迁移走，而不能控制 maxmemory，让应用那边直接写报错。这个是最大的问题，决定了容器化是否能进行下去。如果不控制内存，K8S 的某些功能形同虚设，但如果控制内存，与携程之前的运维习惯和流程不太相符，业务也无法接受。

以上都是我们遇到的一些主要问题，有些 K8S 的原生策略就可以很好地支持，有些则不行，

需自研策略来解决。

2.3 K8S 原生策略

首先，我们的容器基于 K8S 的 Statefulset，这个几乎没有任何疑问，毕竟 Redis 是有状态的。

其次，nodeAffinity 保证了调度到指定标签的宿主机，podAntiAffinity 保证同一个 Statefulset 的 Pod 不调度到同一台宿主机上，toleations 保证可以调度到 taint 的宿主机上，而该宿主机不会被其他资源类型调度到，如 Mysql，App 等，也就是说宿主机被 Redis 独占，只能调度 Redis 的实例。

上面提到的分拆其实也是基于 nodeAffinity，podAntiAffinity 等特性，我们内部划分出一块虚拟区域叫 slaughterhouse,专门用于分拆，分拆完成再迁到常规区域。

2.4 自研策略

宿主机固定，这个是自研的调度 sticky-scheduler 来提供支持，如下图所示，在创建实例的时候会看 annontation 有没有对应 host，有的话直接会跳过调度固化到该宿主机上，如果没有则进入默认的调度宿主机的流程。

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  annotations:
    redis-instance0: host0
    redis-instance1: host1
  creationTimestamp: 2018-06-14T11:38:17Z
  generation: 1
  labels:
    app: redis-instance
  name: redis-instance
  namespace: test-redis
  resourceVersion: "78632983"
  selfLink: /apis/apps/v1/namespaces/test-redis/statefulsets/redis-instance
  uid: 6e678f82-6fc7-11e8-aca4-567909392929
```

虽然 Redis 对磁盘需求不多，但我们还是得防止 log 或 rdb 文件过大将磁盘撑爆，自研的 chostpath 和 cemptydir 都是基于 xfs 的 quotas 很好的支持磁盘配额，并且我们将 Redis.conf 和 data 目录挂载出来，保证重启容器后配置文件不丢失，还可保证容器重启后可以读 rdb 数据。

比如我们在做风险操作升级 kubelet 时候可能会引起相关的 Pod 重启，但我们先对相关的 Redis bgsave 下，哪怕重启 pod 也会读取对应的 rdb 数据，不会导致完全没有数据的尴尬场面。

监控方面，之前 Redis 部署在物理机上，通过端口来区分不同的实例，所有的监控通过端口

来区分，但容器化后每个 Pod 都有一个 IP，自然监控策略要变。

我们的方案是每个 Pod 两个容器，一个是 Redis 本身的实例，一个是监控程序 telegraf，每 60 秒采集一次数据发送到公司的统一监控平台 Hickwall，所有的 telegraf 脚本固化在物理机上，一旦修改方便统一的推送，并且对于 Redis 实例没有任何影响。

实践证明这种监控方案最为理想，比如有一次我们生产迁移集群后，DBA 需要集群的聚合页面，也就是把所有的实例聚合在一起的按集群维度查看的页面，我们修改 telegraf 的脚本将集群的信息随着实例推送过去立刻就能显示在监控页面上，非常方便。

下面两张图清晰地展示出容器的监控页面和物理机完全没有区别。

为了解决上文提到的 Master 挂了不希望 K8S 立刻把它拉起来，希望哨兵来感知到它，我们用 Supervisord 作为容器的 1 号进程。当 Redis 挂了，Supervisord 默认不会拉起它，但容器还是活的，Redis 进程却不存在了，想让 Redis 活过来很简单，删除掉 Pod 即可。K8S 会自动重新拉起它。

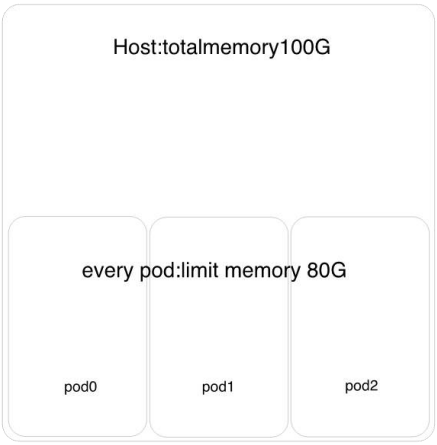
最后再来看看最困难的，实例几乎没有任何的内存限制。实际上在容器上我们对 CPU 和内存也几乎没有限制。

CPU 不限制主要是几个方面原因，首先，12 核的机器上 CPU quota/period =12， 按理是占满了整个机器，但压测时 CPU 居然有 throttle，这明显不符合我们的客观直觉，我们怀疑 Linux 的 cfs 是有问题的，而且很神奇的是我们设置一个很大的 quota 值后，也就是将 CPU 限额设置到 50 核，throttle 消失了。

其次 Redis 是单线程，最多能用一个 CPU，如果一个 CPU 跑一个 Redis 实例，肯定没问题，实际上我们设置两个实例分配到一个核也是完全可行的。

最后一个原因也是最主要的，Redis 在物理机上运行是没有任何 CPU 隔离的。基于上面三个原因，我们让 CPU 超分。

关于内存超分，下面这张图清晰地说明了问题所在，对于只有一个 100G 的宿主机，只要放上 2 个实例，每个实例 50G，它的内存就超了。内存超分好处很多，比如物理机迁移过来很平滑，用户也很能接受，运维工具几乎不需要修改就能套上去，但是，超分大法好，但 OOM 了怎么办?



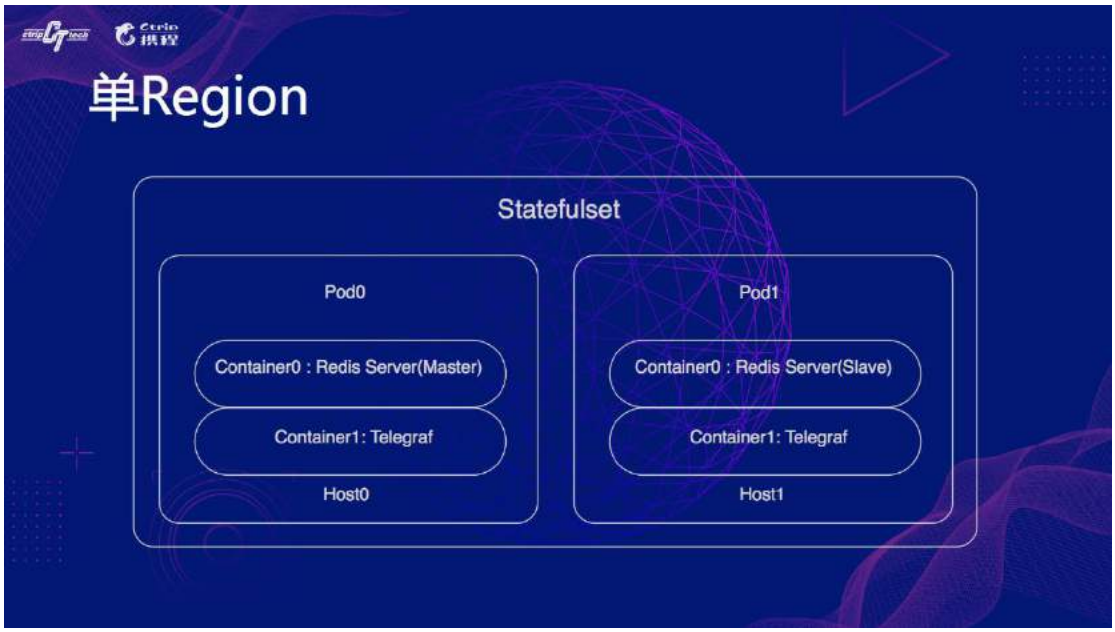
方案是不让 OOM 发生，只要策略合适，这显然是可以做到的，在说到杜绝 OOM 的策略之前，先看下普通的调度策略。

我们在调度时对集群重要性进行了划分，主要分为以下几种:
1）基础集群，比如账号相关的，登陆相关的，虽然订单无关但比订单相关都重要。
2）接入 XPIPE，订单相关的。
3）没有接入 XPIPE，订单相关的。

4）订单无关但相对重要的。
5）既订单无关的又不重要的。

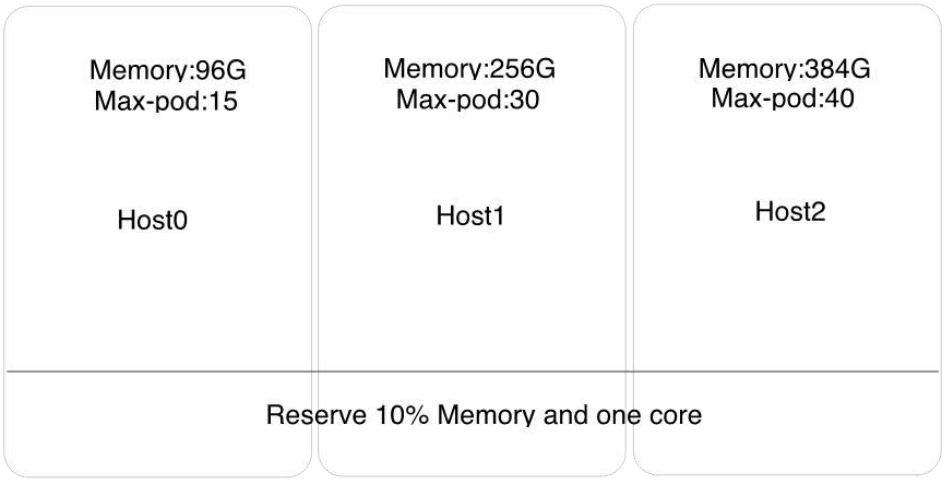
这样划分后，我们就可以很方便地让集群根据重要性按机器的高中低配来调度，并且让集群是否在多 Region 上打散。为了方便理解，这里一个 Region 可以简单等同于一个 K8S 集群。

单个 Region 如下图，一个 Statefulset 两个 Pod 分别是 Master/Slave，每个 Pod 里面有两个容器，一个是 Redis 本身，一个是监控程序 telegraf，部署在两个 Host 上。



多个 Region 如下图，这时候，其实是有 2 个 Statefulset，这种方案可以扩散到更多 Region，这样哪怕是某个 K8S 集群挂了，重要的集群仍然有对外提供服务的能力。

介绍完一般的调度策略后，接着说上文提到的杜绝 OOM 的策略。首先，调度之前，对于不同配置的宿主机限定不通的 Pod 数量，此外设定 10%的占位策略，如下图所示，并且设定 Pod 的 request == maxmemory。



调度中，我们会基于宿主机实际的可用内存进行打分，在 K8S 默认调度后，优选时我们会将实际剩余内存的打分赋值一个非常高的权重，当然基于其他策略的调度比如说 CPU，网络流量之类我们也在研究，但目前最优先考虑的是实际剩余的物理内存。

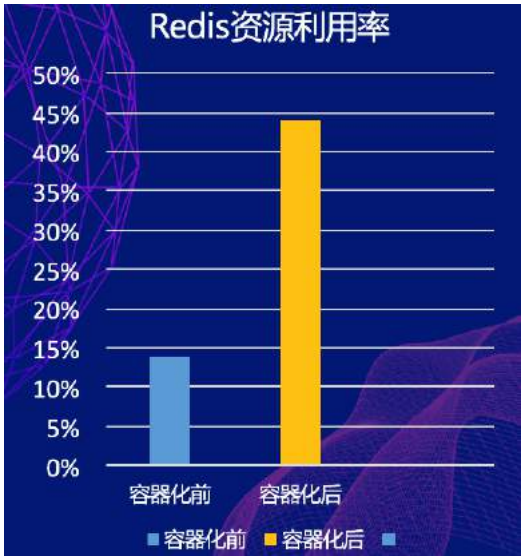
以上这些策略可以杜绝大部分 OOM，但还不够，因为 Redis 后续还是会自然增长的，所以在运维过程中，我们会有 Job 定时轮询宿主机，看可用内存和上面的 Pod 分配是否合理，对于不合理的 Pod，Job 会自动触发迁移任务，将一些 Pod 迁移到内存更空的机器上去，以达到宿主机整体可用内存方差最小。

还有一些其他的调度后的策略，比如动态调整 Redis 实例的 HZ，我们曾遇到一个情况就是，在物理机上跑着一个实例大小都是 10 多个 G，但跑到容器上后 2 天增加了 20 多个 G。

我们排查后发现 Redis 的 HZ 值设置的过小，导致大量过期的 Key 没时间来得及清理，清理完成后发现，usedmemory 是下来了，但 rss 还保持稳定，也就是碎片率很高，所以我们会动态打开自动碎片整理，整理一次完成后再关闭它，因为同时打开，消耗的 CPU 过高，目前情况下还不是很适合。

最后还有个保底的，基于宿主机内存告警，一般设置为 80%即可，这种保底策略到目前为止也就触发过一次。

小结下，Redis 跑在容器上，尤其在生产上大规模部署，需要多个组件共同协作才能达成。其次，携程的现状决定了我们必须超分，那么超分后如何不 OOM 是关键，我们从调度过程前中后容器层面和 Redis 层面分别都有相应的策略，调度上的闭环不但保证了 Redis 在容器上的平稳运行，而且资源利用率（如下图所示）也做到了非常大的提升。

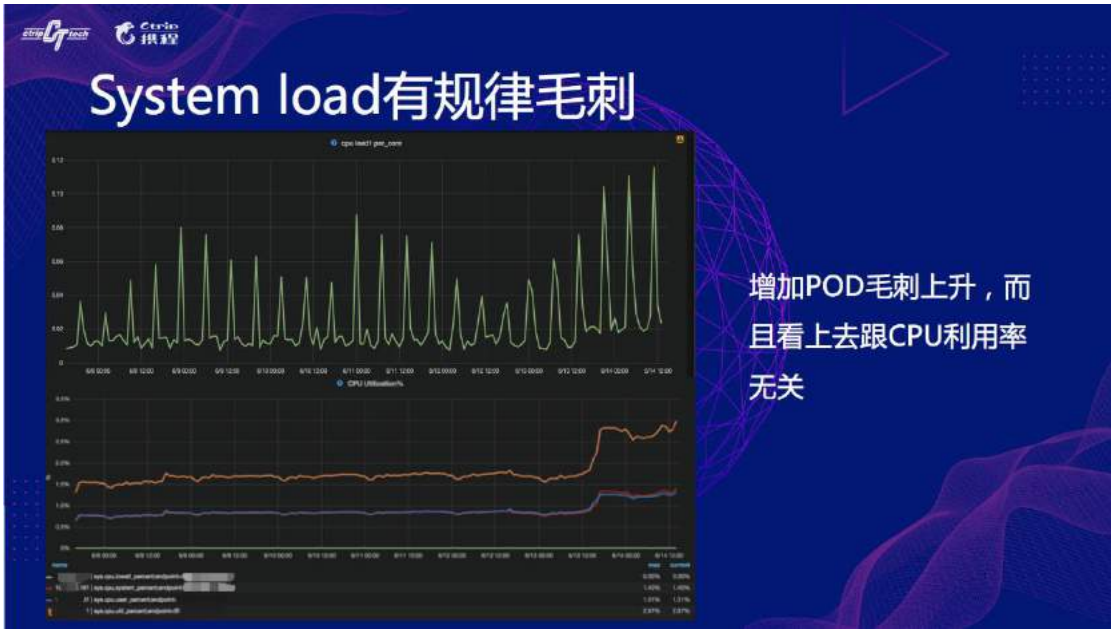


三、一些坑

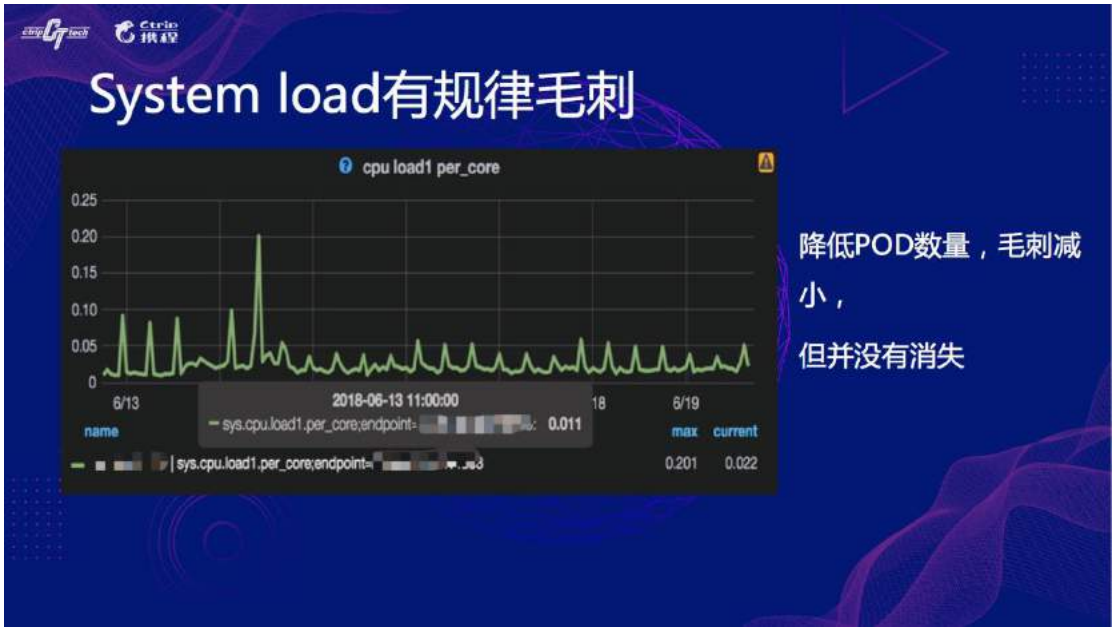
最后再分享一下实践过程中的一些坑，这些坑其实本身不是 Redis 的问题，但都是在 Redis 容器化过程中发现的。

3.1 System Load 有规模毛刺

首先是 System Load 有规模毛刺，每 7 小时一次，我们可以看到监控上，增加 Pod 后毛刺上升，但看上去跟 CPU 利用率没什么关系。



降低 Pod 数量，毛刺减小，但还存在，所以跟 Pod 数量正相关，



后来我们发现是 telegraf 监控脚本的问题。所有瞬间会产生很多进程的 Job 都会导致 System Load 升高。

对于 Redis 宿主机 load 异常情况，主要是因为监控程序每 1min 生成很多进程采集一次数据， System Load 采集则是每 5.001s 采集一次，当 telegraf 的第一次采集点命中 System Load 采集点后，第二次则需要 5s*(5/0.001)=25000s，导致 Load 有规律每 7 小时飙高。

我们修改 telegraf 中的 collection_jitter 值，用来设置一个随机的抖动来控制 telegraf 采集前的休眠时间，确保瞬间不会爆发上百个进程，修改后，毛刺消失了，如下图所示：

3.2 Slowlog 的异常

其次是 Slowlog 的异常，该问题根因在于 4.9-4.13 的内核的一个 bug，会导致 skylake 服务器的时钟变慢，而该时钟不断地被 NTP 修正，所以导致 Slowlog 的两次打点时间过长，升级内核到 4.14 即解决该问题。

详细的分析可见这篇文章：携程一次 Redis 迁移容器后 Slowlog“异常”分析

3.3 Xfs bugs

还有一个是 Xfs 的 bugs, Xfs 我们发现的有两个比较严重的问题, 第一个是字节对齐的问题, 这个比较隐蔽，简答地说就是内核态的 Xfs header 跟用户态的 Xfs header 里面定义不同, 导致内核在写 Xfs 的时候会越界。下图中就是很明显的症状，我们升级 4.14 的内核对内存对齐打了 patch 解决了该问题。

Xfs 第二个问题是 xfsaild 进入 D 状态缓慢导致宿主机大量 D 状态进程和僵尸进程，最终导致宿主机僵死，典型的现象如下图。

这个现象在 4.10 内核发现很多次，并且猜测与 khugepaged 有关系，我们升级到 4.14 并 Backport 4.15-4.19 的 Xfs bugfix，压测问题还是存在，但比 4.10 要难以复现，在 free 内存超过 3G 后不会再复现。目前升级到 4.14.67 Backport 的新内核实际运行中还没出现这个问题。

```
root     12971  0.0  0.0      0     0 ?        D    13:54   0:00 [kworker/u18:0]
root     12988  0.0  0.0      0     0 ?        D    13:54   0:13 [xfsaild/dm-2]
root     13095  0.0  0.0      0     0 ?        D    13:54   0:00 [kworker/4:0]
```

携程新一代监控告警平 Hickwall 架构演进

[作者简介]陈汉，携程网站运营中心研发工程师，从事 Hickwall 监控告警平台的研发工作。经历了 Hickwall 项目的雏形到交付生产再到不断改进，通过整个开发过程，对监控领域有了深入的了解。喜欢探究系统的底层原理，对分布式有浓厚的兴趣。本文来自陈汉在“2018 携程技术峰会”上的分享。

监控告警是网站可用性的第一道防线，为网站提供更加实时可靠高效的监控告警，对互联网企业具有非凡的意义。致力于这个目标，经过不断地改进，携程新一代监控告警平台 Hickwall 在存储效率、查询速度和告警可靠性方面都有了极大的改善。

本文将从存储、聚合、告警三个方面介绍 Hickwall 在核心架构方面的演进。

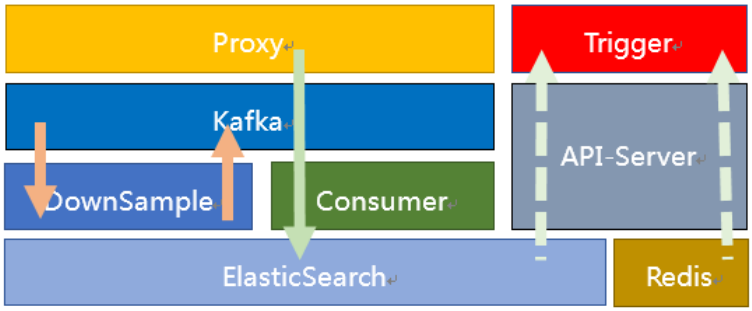
一、架构演进概述

为了更好地了解 Hickwall 在核心架构方面的设计，我们首先将 Hickwall 第一代的架构和现有架构进行比较。

Hickwall 最初的研发是在 2015-2016 年，当时我们调研了业界知名的开源监控系统。

比如 Graphite，拥有非常好的生态，但是集群配置复杂，每个指标都采用一个文件存储，导致小文件多，iowait 高，并且使用 python 实现，性能方面不太令人满意。

再比如 OpenTSDB，基于 HBase 天然就支持分布式，但是也受限于 HBase，多维查询的时候性能比较差。而其他的监控系统也并未非常成熟，最后我们决定使用 ElasticSearch 作为存储引擎。下图是第一代的核心架构图。



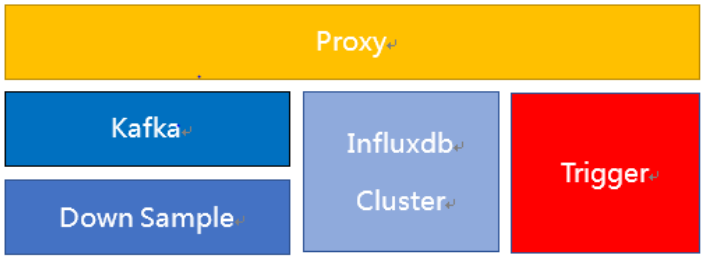
在这个架构中监控数据从 Proxy 进来，经过格式整理、数据补全、限流后发送到 Kafka。Donwsample 消费 Kafka 中的原始数据进行时间维度上的聚合，聚合成 5m、15m 等时间维度的数据点之后写入到 Kafka。Consumer 消费 Kafka 中的原始数据和聚合数据写入到 ES，

通过 API-Server 提供统一的接口给看图和告警。

因为 ES 的查询性能无法满足 Trigger 高频率的拉取需求，我们另外增加了 Redis 用来缓存最近一段时间的数据用于告警。这套架构初步实现了监控系统的功能，但是在使用过程中我们也发现以下几个问题：

- 组件过多。运维架构追求的是至简至稳，过多的组件会增加部署和维护的难度。另外在团队人员变动的情况下，新成员进来无法快速上手。
- 数据堆积。Consumer 消费 Kafka 出现问题，容易导致 Kafka 中数据堆积，用户将无法看到线上系统的当前实时状态，直到将堆积的数据消费完。按照我们的实践经验，数据堆积的时间往往会有几十分钟，这对于互联网企业来讲是个非常大的问题。
- 数据链条过长。监控数据从 Proxy 进来到 Trigger 告警需要依次经过 6 个组件，任何一个组件出现问题，都可能导致告警漏告或误告。

为了解决这些问题，我们研发了 Hickwall 的第二代架构，使用自研的 Influxdb 集群取代了 ES 作为存储引擎，如下图。



在这个架构中监控数据从 Proxy 进来分三路转发，第一路发送给 Influxdb 集群，确保无论发生任何故障，只要 Hickwall 恢复正常，用户就能立即看到线上系统的当前状态。

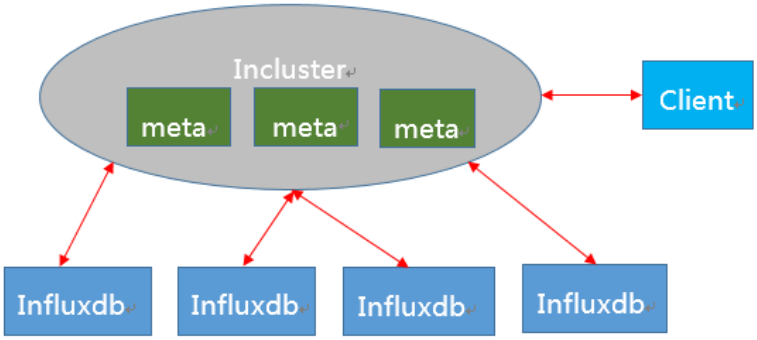
第二路发送给 Kafka，由 Downsample 完成数据聚合后将聚合数据直接写入到 Influxdb 集群。第三路发送给流式告警，这三路数据互不影响，即使存储和聚合都出现问题，告警依然可以正常工作，确保了告警的可靠稳定。

二、Influxdb 集群设计

ES 用于时间序列存储存在不少问题，例如磁盘使用空间大，磁盘 IO 使用多，索引维护复杂，写入和查询速度慢等。

而 Influxdb 是排名第一的时间序列数据库，能针对时间范围进行高效的查询，支持自动删除过时数据，较低的使用和维护成本。只是早期的 Influxdb 不够稳定，bug 比较多，直到 2017 年底。我们经过测试确认 Influxdb 已经足够稳定可以交付生产，就萌生了用 Influxdb 替换 ES 的想法。当然 Influxdb 存在单点问题，在 0.12 版本以后，官方的集群方案还闭源了。

为了解决 Influxdb 的单点问题，我们研发了 Influxdb 的集群方案 Incluster，如下图。

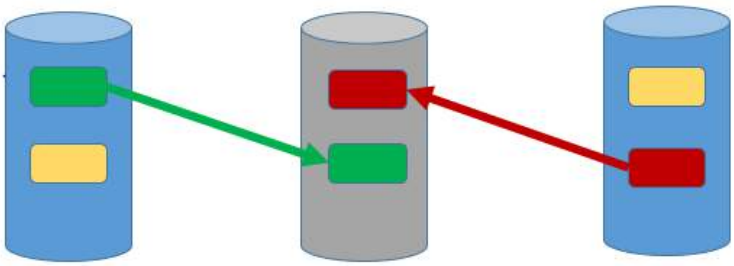


Incluster 并没有对 Influxdb 进行代码侵入式的修改，而是在上层维护关于数据分布和查询的元数据，因此当 Influxdb 有重大发布的时候 Incluster 能够及时更新数据节点。

客户端通过 Incluster 节点写入数据，Incluster 按照数据分布策略将写入请求转发到相关的 Influxdb 节点上，查询的时候按照数据分布策略从各个节点上读取数据并合并查询结果。在元数据这一层 Incluster 采用 raft 保证元数据的一致性和分区容错性，在具体数据节点上使用一致性 hash 保证数据的可用性和分区容错性。

Incluster 提供了三种数据分布策略 Series、Measurement 和 Measurement+Tag。通过调整数据分布策略，Incluster 能够尽量做到减少数据热点并在查询时减少查询节点。在实践过程中，我们使用 Measurement 策略来存储系统指标，如 CPU；使用 Measurement+Appid 策略来存储请求量。

作为一个分布式存储，磁盘损坏不可避免，灾备是必须考虑的问题。我们按照数据分布策略通过读取 Influxdb 底层的 TSM 数据文件，来恢复损坏的节点上面的数据。实践经验表明 Incluster 能够做到半个小时恢复一个损坏的节点。



在用户使用方面，Incluster 提供了对 InfluxQL 的透明支持，也提供了类 Graphite 语法用于配图。类 Graphite 语法可以简化配图语法，提供 InfluxQL 无法实现的功能，例如查询最近一段时间变化最剧烈的指标，除此之外还可以屏蔽底层存储细节，以后如果想使用比 Influxdb 更优秀的时间序列存储引擎，可以减少用户迁移成本。

三、数据聚合的探索

Influxdb 在数据存储和简单查询方面表现出色，但是在数据聚合上就存在一些问题。

Influxdb 提供了 Continuous Query Language(CQL)用于数据聚合，但是经过测试发现 CQL 内存占用较大。Influxdb 原本需要的内存就不小，在我们使用过程中 128G 内存已经使用了一半，如果再加上 CQL 的内存，容易造成节点不稳定。

另外 CQL 无法从不同的节点获取数据进行聚合，在 Incluster 集群方案中存在资源浪费维护复杂的问题。因此我们将数据聚合功能独立出来，在外部进行数据聚合后再将聚合数据写入到 Incluster。

时间维度的聚合是有状态的计算，我们面临两个问题。一个是中间状态如何减少内存的使用，另外一个是节点重启的时候中间状态如何恢复。

我们通过指定每个节点需要消费的 Kafka Partition，使得每个节点需要处理的数据可控，避免 KafkaPartition Rebalance 导致内存不必要的使用，另外通过对 Measurement 和 Tag 这些字符串的去重可以减少内存使用。中间状态恢复方面我们并没有使用保存 CheckPoint 的方法，而是通过提前一段时间消费来恢复中间状态。这种方式避免了保存 CheckPoint 带来的资源损耗。

业务场景聚合主要的挑战在于一次聚合涉及到的指标数太多，聚合逻辑复杂。例如某个应用的某个接口的请求成功率，涉及到的指标数目上千，这种聚合查询 Influxdb 无法支持的。

我们的解决方案是使用 ClickHouse 进行预聚合。ClickHouse 是俄罗斯开源的面向 OLAP 的分布式列式数据库，拥有极高的读写性能，并提供了强大的 SQL 语言和丰富的数据处理函数，可以完成很多指标的处理，例如 P95。

四、流式告警的实现

告警最简单的实现就是定时从数据库中拉取数据，然后检查一下数据是否有异常。但是这种

Pull 的方式对存储存在一定的压力，尤其是告警规则告警对象众多的时候，对存储的可靠性和响应时间有极高的要求。

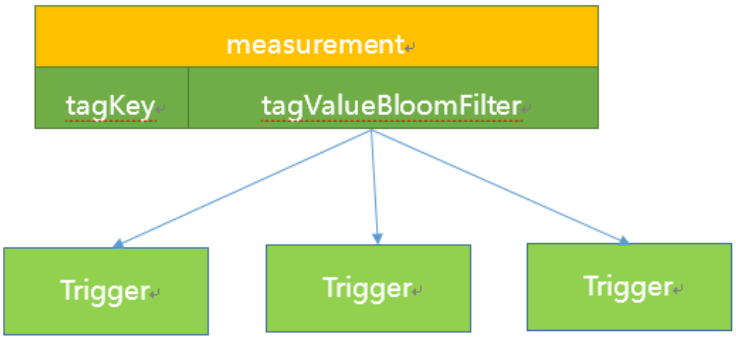
我们经过研究发现告警数据在所有监控数据中占比其实不大，以携程为例只占了 8%，而且需要的绝大部分都是最近几分钟的数据，如果我们能从数据流中直接获取所需要的数据，就能过滤掉大部分不必要的数据，避免对后台存储的依赖，让告警变得更加可靠实时。

实现流式告警最大的挑战是数据订阅。我们不可能让每一个告警规则都去消费一遍数据流，最好的方式是消费一遍数据流然后将告警数据准确的分发到告警上下文中。

在这里如何降低数据分发的时间复杂度和空间复杂度是最大的难点。

Hickwall 的实现思路是减治法，通过 Measurement 精确匹配减少下一步需要匹配的规则数量，通过 tagValue 的布隆过滤器判断是哪个 Trigger 节点需要的数据。Trigger 节点收到数据以后对数据点进行精确的匹配过滤，转发到具体的告警上下文中。

这个方案的优势在于时间复杂度不随规则数量告警对象而线性增长，空间复杂度不随 tagValue 的长度而增长。



Hickwall 使用 Akka 框架进行告警逻辑和告警数据的处理。Akka 是异步高并发的框架，提供了 Actor 编程模型，能够轻松实现并发地处理数据和执行告警逻辑。

生产系统是个时刻变化的系统，每时每刻都可能有机器上下线，每时每刻都可能有应用发布变更，随着这些变动告警系统需要随之增删告警对象和修改告警阈值。而 Actor 的创建删除是非常轻量的，为生产系统提供了非常友好的抽象，降低了开发成本。

Hickwall 使用了 RocksDB 来缓存告警数据，通过 JNI 直接嵌入到 Trigger 实例中。RocksDB 是 Facebook 开源的 KV 数据库，基于 Google 的 LevelDB 进行了二次开发，底层存储使用 LSM Tree，拥有极高的写入速率。

无停滞的处理数据在流式告警中是非常重要的，使用 RocksDB 能够减少 JVM 中的对象，减少内存的使用，进而减少了 JVM GC 的压力。

Init DSL:

```
T.require('m1', "sys.cpu.util_percent","endpoint",M.hostName, "5m", 4);
```

Run DSL:*

```
var ValueThreshold=M.ValueThreshold||90;
var CountThreshold=M.CountThreshold||3;
var m1=V.m1;

for(var endpoint in m1){
    if(m1[endpoint].len()<CountThreshold) continue;
    if(m1[endpoint].streak(">=",ValueThreshold) >= CountThreshold ){
        return WARNING("Processor(_Total)\\% Processor Time:Processor Time is overloaded");
    }else{
        return OK("");
    }
}
```

在用户使用方面，Hickwall 提供了基于 JS 语法的 DSL 语言，Init DSL 负责数据的订阅和接收到数据后的处理工作，提供了 groupBy、filter、exclude、summarize 等流式计算中常见的数据处理函数，Run DSL 负责具体的告警逻辑，判断是否有异常。

考虑到 DSL 书写有一定的难度，Hickwall 提供了语法检查、历史数据回测等功能，帮助用户书写出符合需求的告警逻辑。

无线大前端篇

携程 MTP 和 MCD 平台，如何支撑一年 10W+次无线集成和发布

【作者简介】陈浩然（Harry Chen），携程无线技术高级总监，负责无线委员会和无线基础工程团队。

2017 年，携程无线基础技术的研发重心有所变化，原先围绕无线技术体系维度（图 1），即性能、质量、开发框架、新技术、基础设施和工具这五个方面展开的，然而随着公司无线工程规模的扩大，原有的交付方式已不能满足工程需要。

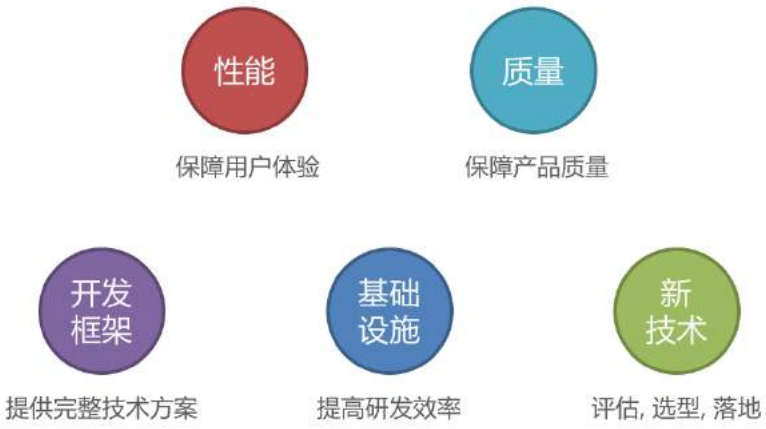


图 1 无线技术体系维度

我们转变思路从研发生命周期维度（图 2）出发，针对开发阶段、集成/测试/发布阶段、运营阶段来开发相应的平台化技术，以支持研发生命周期中的各类需求。2017 年我们重点建设的是图 3 所示三个平台，今天我们先简要分享其中 MTP (Mobile Tech Platform)无线技术平台和 MCD (Mobile Continuous Delivery)无线持续交付平台的设计思路及其成果。

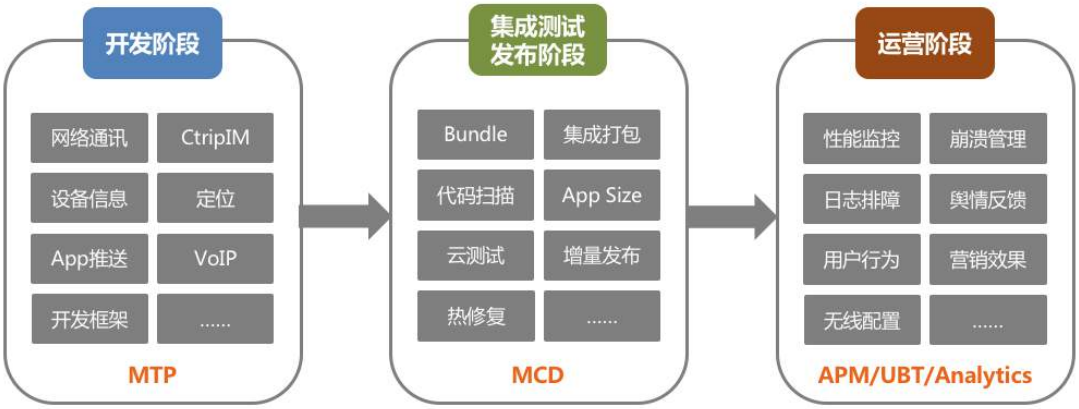


图 2 无线研发生命周期维度

图 3 平台化建设

一、MTP - 无线技术平台

MTP 的设计思路是由携程集团内的无线研发现状（图 4）决定的。虽然目前核心无线产品是携程旅行（针对国内市场）和 Trip.com（针对国际市场）两款 App，但是各类垂直业务和内部服务的众多 App 也都在快速发展。

图 4 携程 App 群

这些 App 所需的基础无线组件和服务（例如网络通讯、推送、开发框架等）需求是类似的，如果各自再独立重复开发会浪费研发资源，质量也无法保证，因此我们梳理了现有的基础技术产品，并对原本不支持多 App 应用的服务进行改造升级，面向集团内无线研发人员推出了 MTP 平台（图 5）。

图 5 MTP 技术平台

MTP 提供以下三类技术产品和服务：

1、 基础组件和服务：包括网络通讯、App 推送、IM 服务、定位、VoIP、设备信息、帐号登录、灰度配置、AB Testing、用户行为采集、增量更新、Hotfix 等十几类组件或服务。
2、 开发框架：React Native 框架（CRN）和 Hybrid 框架。
3、 研发支撑：以 MCD 平台为基础的打包集成、测试、发布和运营功能，以及以 APM 平台为基础的端到端性能监控功能。

MTP 不是简单用于展示技术产品的 Portal，其核心功能在于其控制台（图 6），在控制台中可以申请、配置、管理和运维各类组件和服务。

图 6 MTP 平台控制台

以 IM 服务举例（图 7-图 9），在控制台中具备实时查询、实时回调、状态接口、业务类型、消息类型等各种配置管理和日志查询功能，业务使用方可以方便的自助式管理，大幅降低了无线基础团队的技术支持工作量。

图 7-图 9 MTP 控制台 IM 服务

目前 MTP 平台已经在支撑携程集团内 20+的 App 产品研发需求，对于统一公司无线技术标准和保证研发质量具有决定性的作用。

二、MCD - 无线持续交付平台

MCD 平台在携程的研发已有三年历史，最初从淘宝的 Bundle 化技术获得启发，自研实现了自己的插件化技术，并逐步演化成支撑无线集成、测试、发布和运营阶段需求的平台产品（图 10），我们曾分享过相关技术细节，见《从零打造携程无线持续交付平台 MCD 实践》一文。

图 10 MCD 无线持续交付平台

MCD 的用户包括 Dev 开发、QA 测试、PM 产品经理和 PJM 项目经理，研发生命周期中每种角色所需的功能是不同的，所以不同用户的权限各不相同。MCD 中定义了一些基本概念来定义交付物：

1、Bundle：所有模块的中间二进制产物。以携程旅行 App 举例，目前已包含 100+ Native (iOS/Android) Bundle，70+ React Native Bundle，70+Hybrid (JS) Bundle，每个 Bundle 的开发和构建相互独立。

2、L 版本 Bundle：在开发自测阶段 Dev 提交代码之后在 MCD 平台上 Build 所开发的 Bundle，并会自动标记为 L 版本（表示 Latest）。Bundle 间如果有依赖关系，会自动触发相级联构建。

3、RC 版本 Bundle：如果 L 模式 Bundle 在测试正常后，Dev 或 QA 可将其标记为 RC 版本（表示 Release Candidate）（图 11）。

4、测试包：使用所有 Bundle 的 L 版本进行构建，作为日常测试使用（图 12）。

5、集成包：使用所有 Bundle 的 RC 版本进行构建，作为上架应用商店的候选版本用于集成测试。

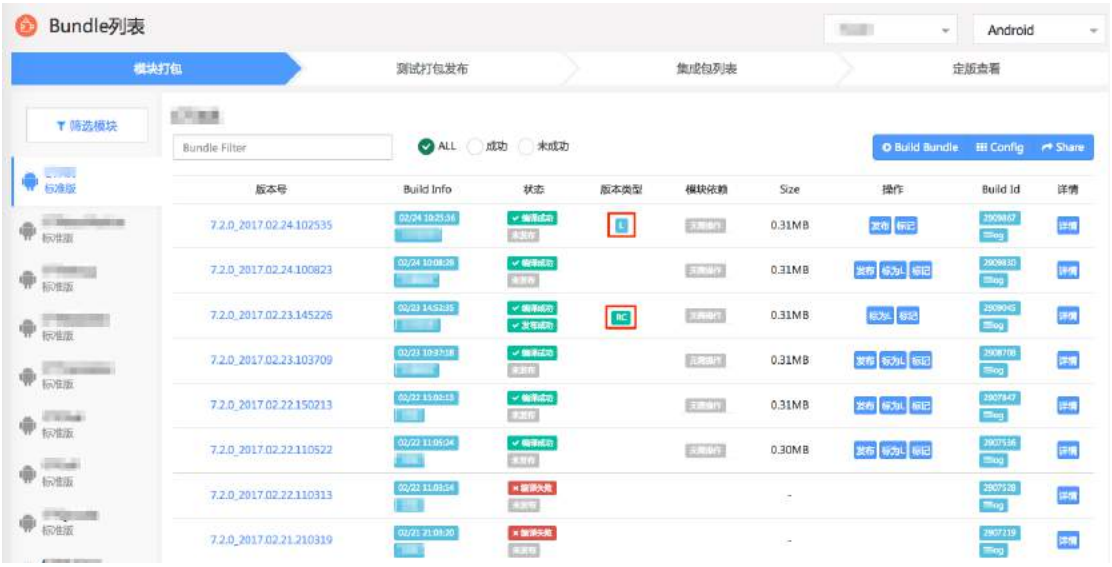


图 11 MCD Bundle 构建

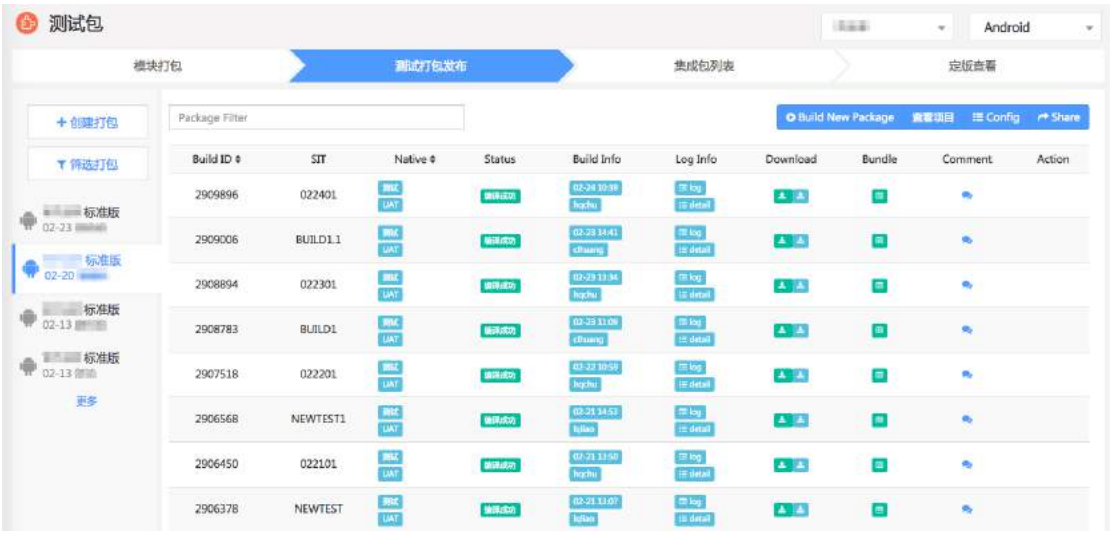


图 12 MCD 测试包打包

测试包和集成包都可以配置为是否需要 hourly build 或 daily build，并且可以和 MCD 测试阶段的的相关功能进行联动（代码扫描、白屏检测等），从而实现持续集成。

2017 全年，MCD 平台完成携程旅行中各种 Bundle 类型总计近 10 万次的构建，产出 iOS 和 Android 各 2.5 万个测试或集成包，平均每次打包构建都在 2-3 分钟内完成，稳定可靠地支撑了日常无线集成和测试工作。

MCD 在发布阶段提供了 Hotfix、Bundle (Android)、React Native 和 Hybrid 包的发布功能，从而支撑紧急问题的修复和业务逻辑的动态更新。不同类型的发布包使用统一的下发通道和多环境发布工单流程，支持差分、7z 压缩和灰度发布功能，Dev 或 QA 在 MCD 可以在发布后查看最新版本的下发情况和使用情况（图 13-14），从而做到想发就发，心中有数。

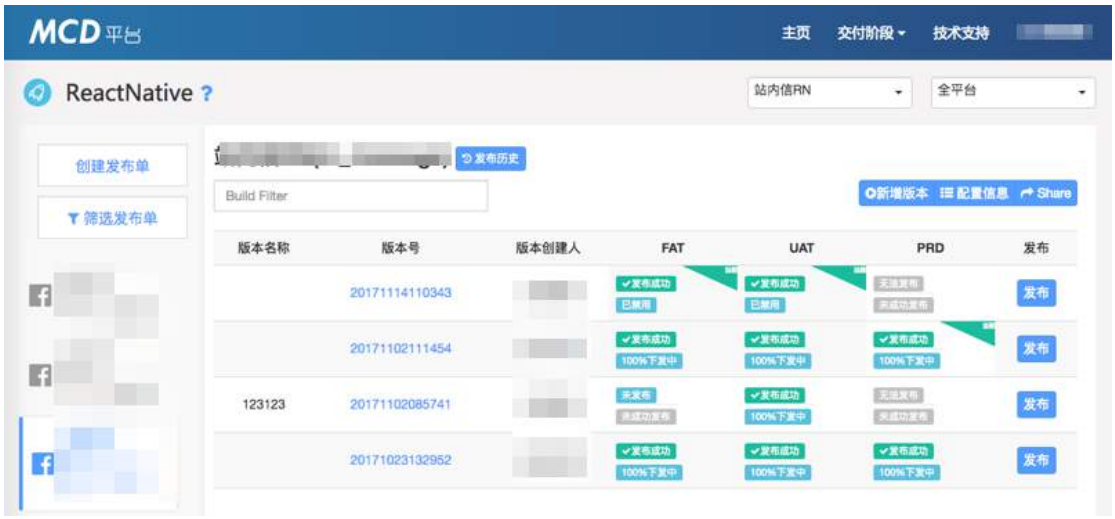


图 13 MCD 发布

图 14 MCD 发布结果

2017 全年，MCD 平台完成携程旅行近 300 次的 Hotfix 和 Bundle 发布（解决多次 PR 事件紧急变更）、各 5000 次的 React Native 和 Hybrid 发布，12 小时内用户更新率可达 95%以上，从发布频率上超出预期，并且无重大发布事故发生。

三、结束语

平台化和系统化解决问题的思路，对于无线研发生命周期的管理和产出至关重要。

从目前携程内 MTP 和 MCD 平台的使用情况看, MTP 技术成果的复用降低了研发成本, MCD 流程的自动化减少了沟通成本，无线相关人员日常都在使用这些平台。

未来我们会继续沿着平台化思路来演进携程无线技术和基础设施。

携程无线离线包增量更新方案实践

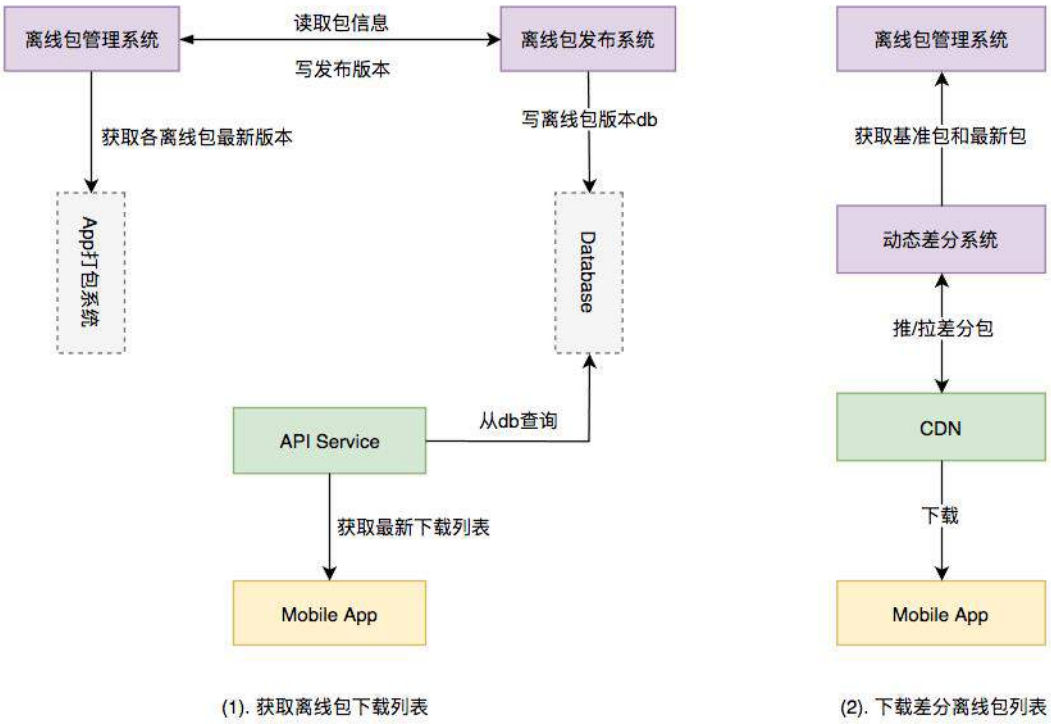
【作者简介】赵辛贵，携程技术中心基础业务研发部无线研发总监。 2013 年加入携程，主要负责 App 基础框架研发相关工作，主要关注 App 开发框架、性能、质量、效率和新技术。先后负责和参与携程 Native、Hybrid 和 React Native 框架设计、工程模块化拆分解耦、Android 插件化动态加载、无线持续交付平台等项目。目前重心主要在 React Native 框架在公司的推广和研发支持、以及公司内部其它独立 App 的框架和工程架构升级。

携程旅行 App 中近半数业务页面使用 H5 Hybrid 和 RN 技术开发，为了提高页面加载速度和成功率，我们在开发 Hybird 技术之初就采用了离线包方式，即将 RH5 Hybrid 或者 RN 开发的业务代码打包到 App 中，直接通过应用商店分发到用户终端。如果有业务功能有变更，就通过我们的无线发布系统，将新的业务离线包更新到 App 中，从而做到随时发布，动态更新。

当然如果都是全量发布，App 在启动时就需要下载更大的离线包，增加用户流量的同时加大了下载失败的概率，因此需要考虑好增量更新的方案。

一、离线包增量更新方案

下面这张简图，介绍了我们是如何设计离线包增量更新方案的：



(1). 获取离线包下载列表　　(2). 下载差分离线包列表

从客户端的角度，整个流程分为 2 部分，离线包下载列表获取和离线包文件下载。

现在以一个新的业务模块上线为例，说明下整个流程：

1）创建业务模块
在离线包管理系统里面，新增业务模块，并配置生效的 App、版本和环境(开发/测试/生产)。

2）发布业务模块
在离线包发布系统，选择业务代码仓库分支，然后 Build，发布。

3）App 打包，获取最新基准包
打包系统中，嵌入下载最新离线包功能的脚本，确保每次打包都能将最新的离线包打包到 App 中，并将每个包的版本信息，一并打包到 App 中。

4）App 启动，获取最新离线包列表
App 启动之后，发送本地 App 中离线包的版本号，以及 App 的 ID 到服务端，服务端返回最新离线包的下载路径。

5）App 根据离线包列表下载离线包
获取到离线包列表之后，在后台线程中按顺序逐个下载。

6）离线包安装
下载完成，解压，合并，安装；

其中 2 个系统的功能简单介绍下。

1）离线包管理系统主要负责以下功能：

a、离线包元数据信息管理
元数据包括唯一包名、适用的平台、优先级、负责人、以及业务频道描述。
b、离线包对应的 App 关系维护
所有的离线包，最终都要打包到 App 中，考虑到灵活性和扩展性，需要维护离线包和 App+环境的关系。
c、离线包的启停用控制
离线包在某个 App 版本中不在使用之后，可以修改相关配置。
d、App+版本+环境的最新离线包查询列表
打包 App 的时候，需要将最新的离线包打包进去，这个时候，需要离线包管理系统提供查询最新离线包列表的 API。

2）离线包发布系统包含以下功能：

a、拉取选择仓库分支的代码，然后 Build
b、发布 Build 完成的包（修改数据库中该离线包版本，修改离线包管理系统中最新包的版本）
c、灰度发布、回滚、停用支持
灰度切分流量下发离线包，发现有问题及时回滚，可以随时停止发布，避免影响更多用户。

d、发布数据查询与监控
发布效果监控，可以查看升级百分比，也支持特定用户的对某个发布的结果查询。

二、工程实践中的问题和解决方案

上面介绍了离线包增量更新方案，但在实际工程实践中还是会遇到了诸多问题，接下来逐个分析。

2.1 包依赖管理

携程旅行 App 有超过 100 个离线包，每个离线包，都是一个独立的功能或者业务模块，这么多的业务之间必定有相互依赖的问题，要严谨的解决该问题，需要引入类似 node 的包管理机制，但这样的解决方案对我们来说太重。

因此我们设计了一套简单的依赖管理规则，来解决这个问题：

1）使用数字标识离线包的优先级，数字越小，优先级越高；
2）优先级越高的包，先下载安装；
3）优先级相同的离线包，下载顺序和发布顺序一致；

实际使用过程中，我们只定义了 2 个优先级 0 和 100。0 为框架类，公共业务类的离线包，100 为业务功能的离线包。业务依赖框架正常，极少有业务之间的强依赖，偶尔有的时候，业务之间协调好发布顺序即可。

2.2 动态差分

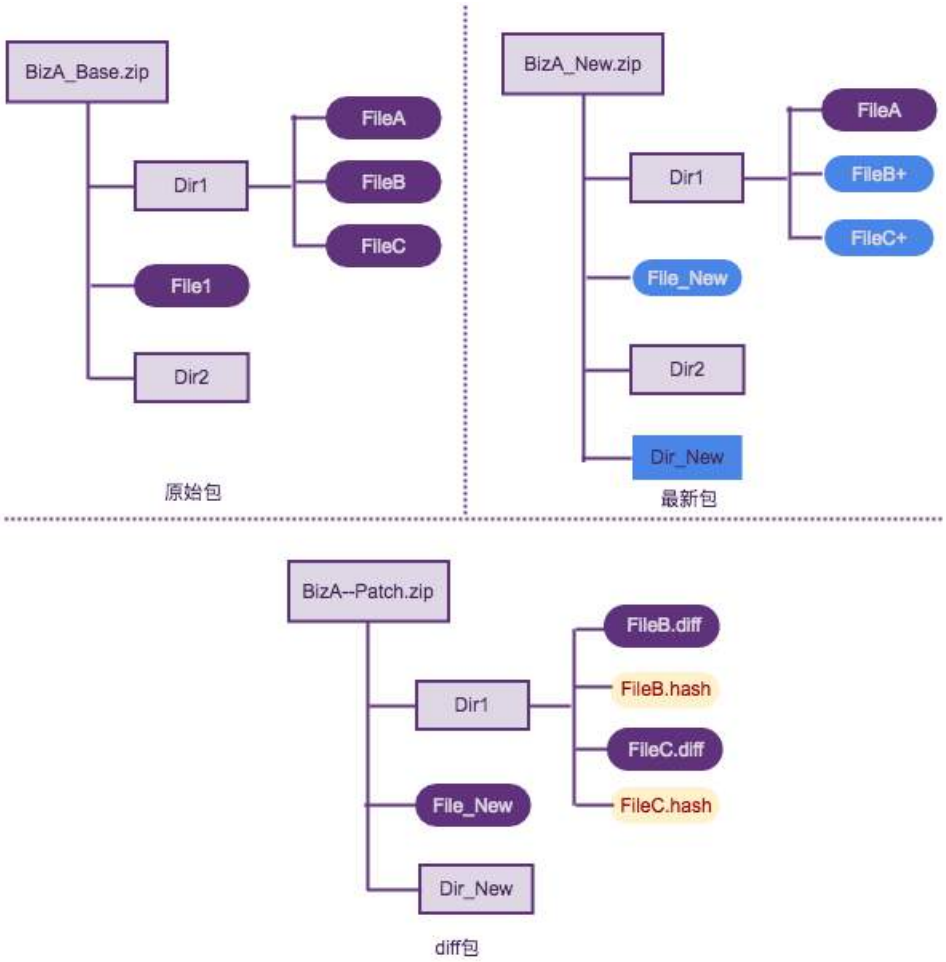
为了让用户能够尽快下载离线包，我们需要尽可能的减小每个离线包的大小。

这个时候，就需要采取差分算法，计算最新发布的包和原始打包到 app 中的基准包之间的差量，然后下发给 App。我们的方案是使用 bsdiff 做差分：

1）服务端拿最新包和打包到 App 里面的基准包计算差量，生成 patch；
2）客户端下载到该 patch 文件后，和打包到 app 里面的原始文件 merge，生成最新包；

看起来很完美的方案，并且业内大多做离线包的差分都是采取这种成熟的方案。但是实际效果并不完美，我们发现偶尔会出现 300 多 KB 大小的离线包在差分之后，生成的包有 100 多 KB。

经过反复测试，我们发现 zip 文件解压之后比较里面的变化文件，生成 diff 文件，然后将 diff 文件生成一个 zip 包，比直接 bsdiff 计算 2 个 zip 包生成的 diff，会小很多。基于这个测试，我们对 bspatch 做了一些改进：



上图可以看到，生成 patch 包的时候，只 zip 进去变更过和新增的文件，同时，对每个变化过的文件，生成了一个 hash 文件，这样可以确保客户端将 diff 文件 patch 到原始文件之后，能验证文件是否完整， 这一点非常重要，因为 bspatch 执行的时候，不会因为文件内容合并不正确而返回 patch 失败。

下图是某个版本中发布的 4 个差分包，传统 bsdiff 方案和我们的优化方案使用后，最终实际下载包的大小对比，可以看出优化效果非常明显。

App内原始包大小(KB)	更新离线包大小(KB)	传统bsdiff方案大小(KB)	优化方案大小(KB)
87.2	88.3	57.3	14.4
188.2	184.7	140.1	9.4
375.6	369.1	276.1	2.9
1021.5	970.57	611.8	10.1

另外，因为打包到不同 App 定制/渠道包里面的各个离线包版本不同，因此差分包需要动态生成。客户端获取到最新离线包列表之后，会先通到 CDN 下载，如果 CDN 没有，再回源到源站服务器，这个时候触发动态差分包生成，生成完成之后，再推到 CDN 上。

2.3 App 端离线包下载

主要由以下机制进行保障：

1）重试机制

离线包下载在网络状态不好时，会有下载失败的情况。为了减少网络因素导致的失败，需要增加重试策略，比如最多下载 3 次，第一次失败，隔 15s 重试一次，第二次 2 次失败，隔 30s 再试一次，3 次失败则会终止继续下载。

2）签名校验

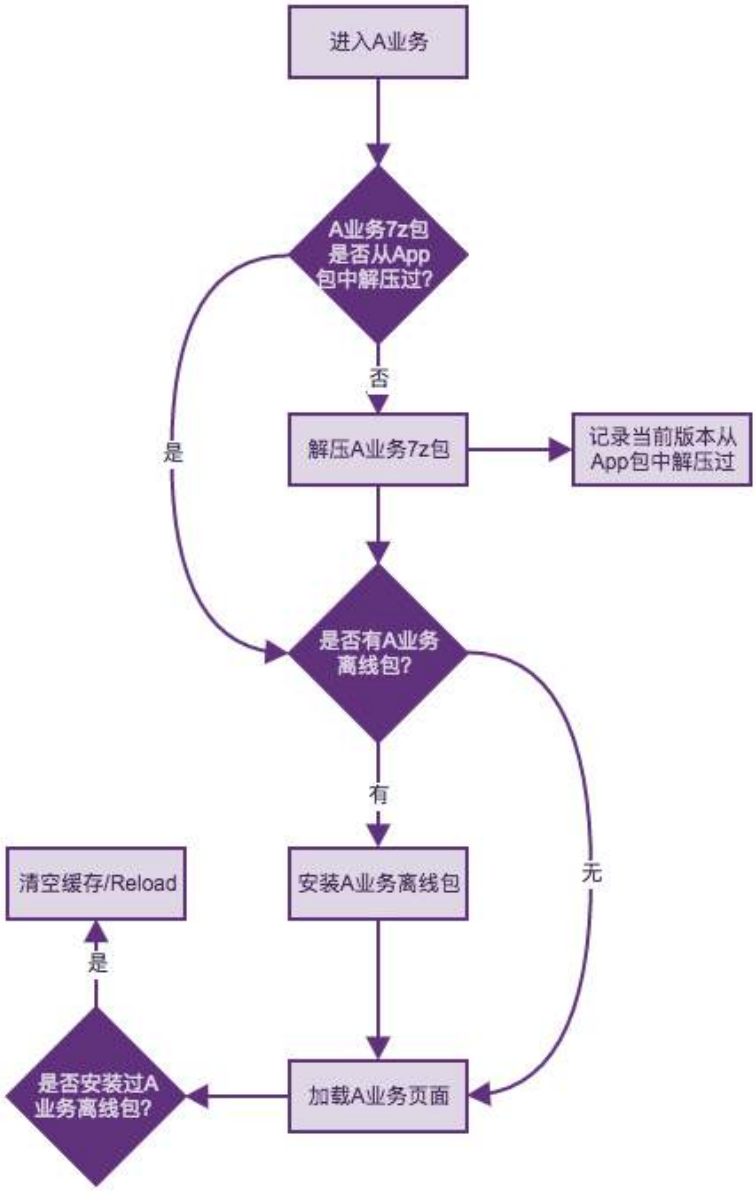
文件下载完成之后，需要检查文件是否被篡改过，因为离线包里面都是代码，必须保证代码的正确性，建议在下发离线包列表的时候，下发该文件的签名，下载完成之后，校验签名是否正确。

3）定时轮循

最初我们的离线包列表是在 App 启动之后，会获取一次，然后下载，当时经常会有反馈离线包下载不及时，不重启就下载不到最新版本。为了解决这个问题，也考虑过使用服务器推送的方案，但是成本较高。因此简单为第一次离线包下载完成之后，每隔 10 分钟再去服务器查询一次，是否有最新离线包列表，如果有，继续下载，这样保证了发布之后，用户网络正常情况下，最多间隔 10 分钟左右，离线包就可以被更新。

2.4 离线包的使用、安装和加载

离线包是打包到 App 中，发布到应用市场，用户下载安装的，因此在本地使用之前，需要先解压安装。从服务器端下载到的新离线包版本，也需要解压安装才能使用。下图是进入某个离线包业务的流程：



以业务 A 为例，简单说明离线包的下载安装过程：

1）业务 A 的离线包下载成功之后开启子线程合并下发的离线包 Anew.7z 和 App 包中的原始包 Abase.7z，合并成功之后保存在离线包的工作目录，例如 A 的工作目录为 Awork，将成功合并的目录命名为 Abak，不可直接覆盖该离线包的工作目录 A_work，因为 A 业务可能正在被使用；

2）进入 A 业务，如果发现有合并成功的离线包文件 Abak，先使用该文件覆盖 Awork，覆盖成功之后，加载页面时候，需要清空缓存，reload 页面，加载失败回滚 A_work 目录。

2.5 发布控制

灰度发布：离线包发布直接到达用户终端，为了确认发布的功能对用户带来的影响线，需要先观察一部分用户行为和数据。这样发布系统就需提供灰度发布功能。我们采用的默认设定规则是 10 分钟 10%，30 分钟 50% 1 小时后达到 100%。

发布回滚：发布的包如果有问题，可以回滚到先前版本的包。

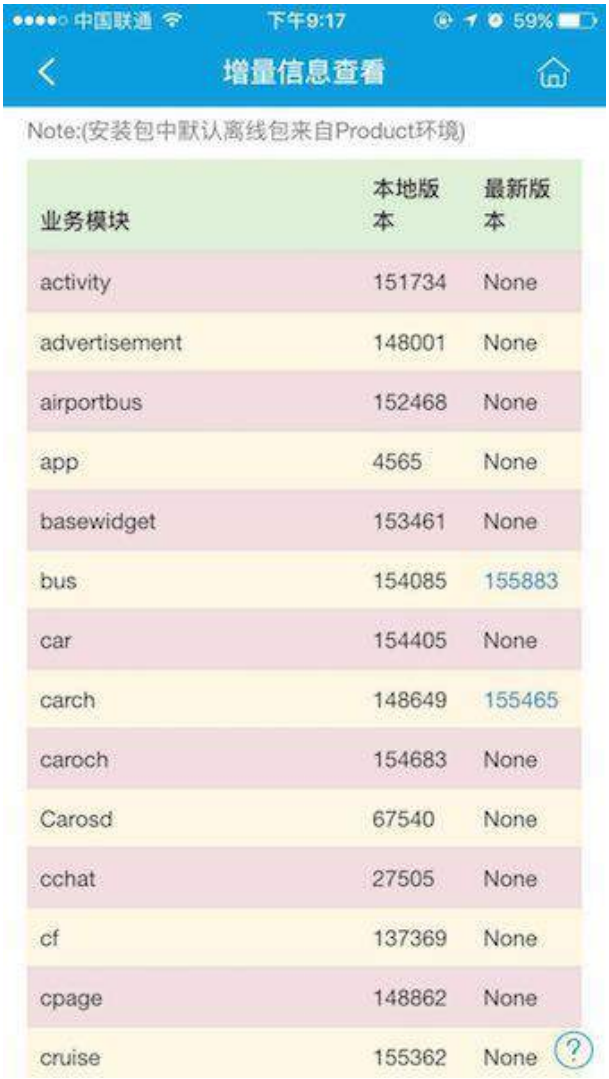
停止发布：发布的包如果没有生效，也没有副作用，为了尽可能的减少影响，可以直接停止这个包的发布。

2.6 端到端监控

离线包的下载、安装是一个复杂的过程，并且都是在 App 运行后台进行，用户并无感知，为了能了解发布的状态和结果，需要完整的数据采集和监控。

对于生产环境：可以以一次下载为起点，安装完成为终点，当做一个事务。每次事务完成，都记录一条日志。这些日志上报后，后端就可以根据这些日志进行监控，实现对端到端的离线包更新效果进行监控或告警。

对于测试环境：可以在测试包中保持完善的文件日志，记录离线包下载更新的每一步。另外，可以提供 Debug 工具，查看每个离线包业务的版本号，例如下图：



业务模块	本地版本	最新版本
activity	151734	None
advertisement	148001	None
airportbus	152468	None
app	4565	None
basewidget	153461	None
bus	154085	155883
car	154405	None
carch	148649	155465
caroch	154683	None
Carosd	67540	None
cchat	27505	None
cf	137369	None
cpage	148862	None
cruise	155362	None

三、小结

通过持续的迭代优化，这样一套无线离线包更新方案，已经稳定的运行在携程旅行主 App 和集团其它子 App 中。从目前生产统计数据看，我们的离线包在 iOS 和 Android 平台分别达到 99.8%和 99.5%的下载安装成功率，希望我们的方案可以对读者的类似业务有所参考。

携程无线 APM 平台，如何实现全球端到端性能监控

【作者简介】陈浩然，携程无线技术高级总监，负责无线委员会和无线基础工程团队。刘李丰，携程无线基础服务和平台总监。

作为 2017 年携程无线技术平台化的另一重点项目，全新 APM 平台改变了原有性能监控平台的设计思路，转为从全球化维度进行性能监控，以满足公司在新业务场景下的端到端性能监控需求。

一、携程无线 APM 历史

提到携程原有的无线 APM 平台，不得不提 UBT 平台。UBT (User Behaviour Tracking) 平台最初的目的是用于收集用户行为，其网络通道是自建的通道服务，和业务服务通道相隔离。刚开始一些性能相关的数据也通过 UBT 通道上传至后端处理，后期为了提高性能数据采集的实时性，把性能数据和普通用户行为数据进行区隔，所有性能数据都会实时上传至后端。

最初的性能监控是围绕网络性能展开，以服务为主维度进行监控（图 1），这是以当时对业务服务进行集中优化为背景的。

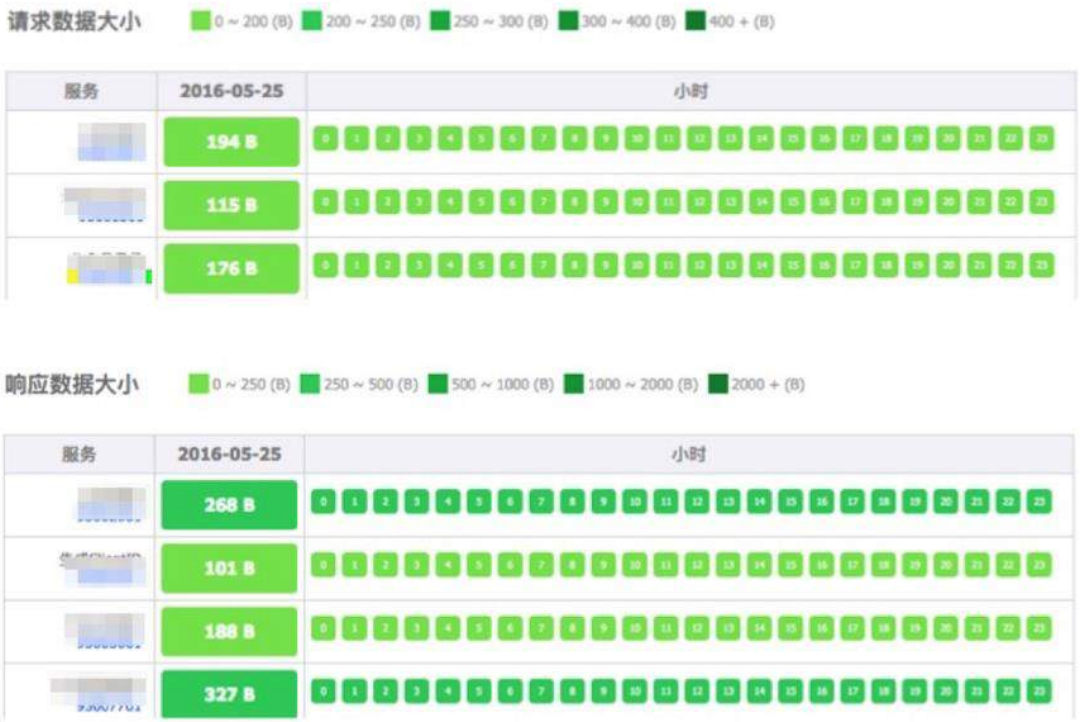
图 1 早期实时网络性能监控

为了方便 Dev 开发人员跟踪某个服务的性能数据，又按照版本和服务维度按照小时级聚合

网络性能数据（端到端成功率、平均耗时和耗时分布，见图 2），这样每个版本服务的性能优化效果很清晰：

图 2 小时级数据聚合

其他的监控数据还包括网络服务请求和响应数据大小、服务错误类型分布以及服务耗时细分（图 3-5），也是为具体服务的性能优化提供数据依据：



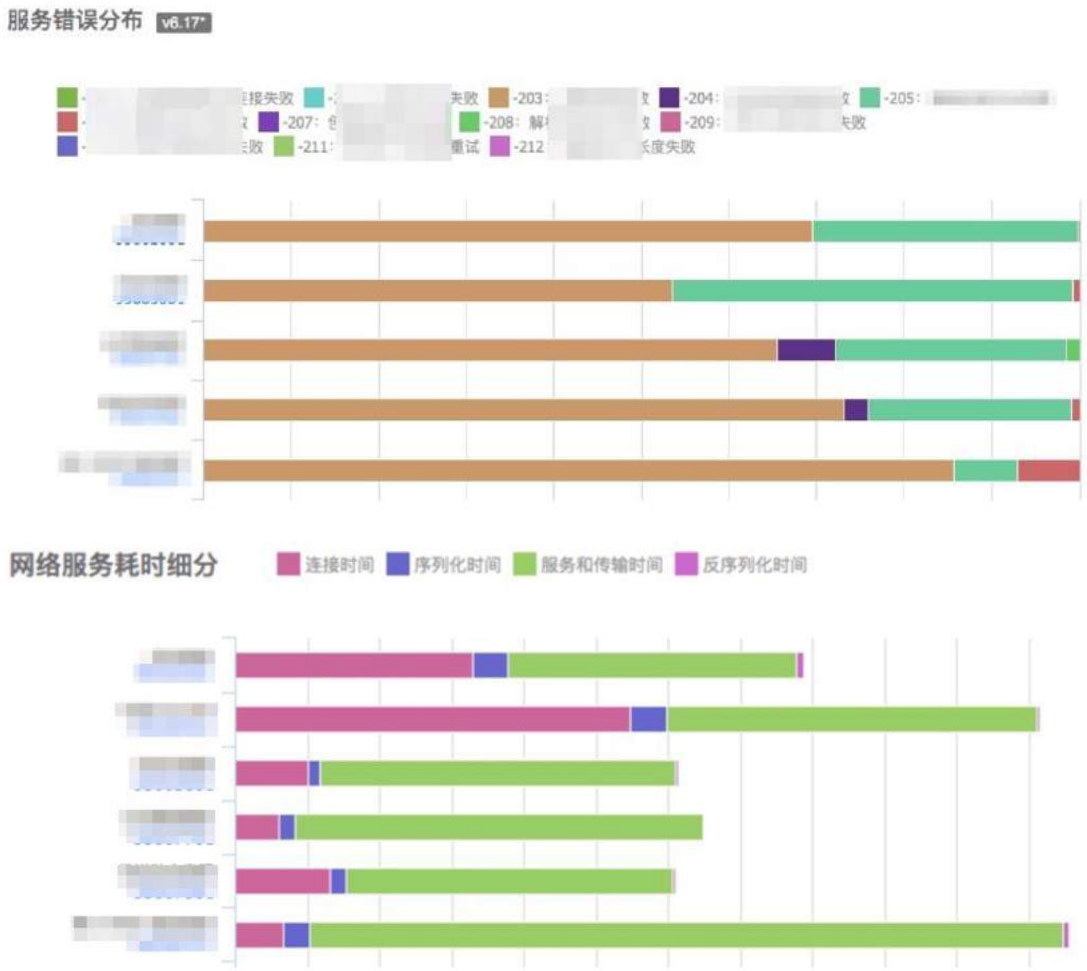


图 3-5 多种维度数据聚合

后期我们又增加了定位、启动、页面加载等各类性能指标的监控，对当时的性能优化和日常运维提供了充分依据。

二、新 APM 平台

2017 年下半年公司的国际化业务开始加速，原有的监控维度已不能满足技术场景的需要，我们调整了设计思路并参考了商业 APM 的设计，最终决定自研开发满足携程全球化业务场景的新 APM 平台。

我们将携程 App 核心性能指标集中为 8 类：网络性能、崩溃、启动加载、定位、图片、CRN、IM 和 VoIP（图 6），基础维度包括系统平台和 App 版本。

图 6 新 APM 平台

2.1 网络性能

网络性能是重中之重，所以需要从不同维度进行监控：

2.1.1 基础网络性能

对端到端网络服务成功率，平均耗时和访问量进行监控（图 7），并且提供全球热门国家和热门城市细分维度。

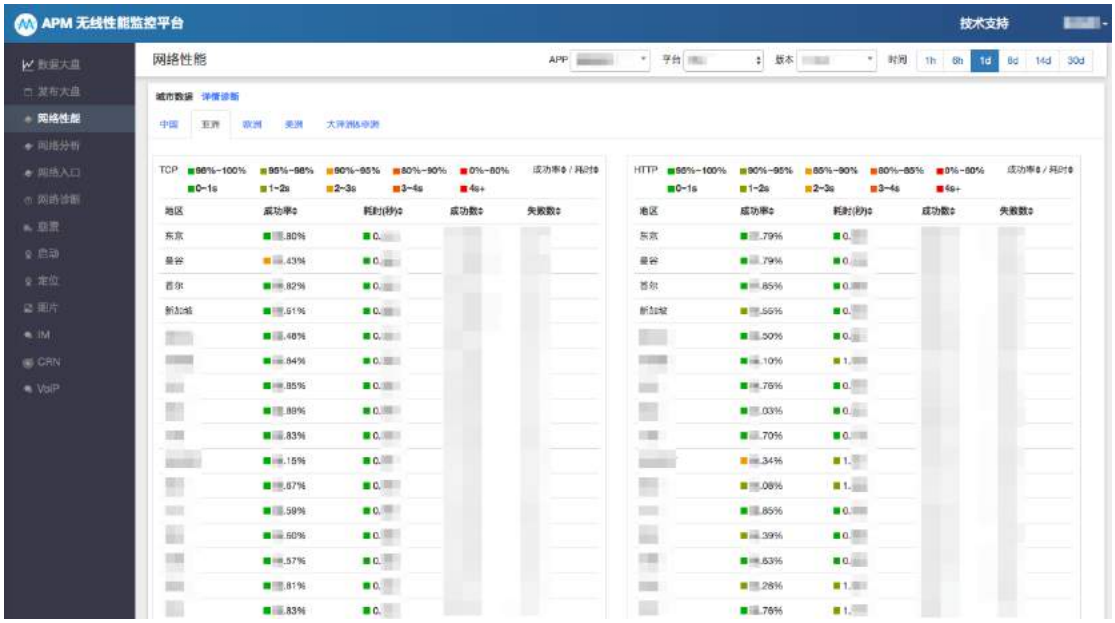


图 7 基础网络性能

2.1.2 网络性能组合分析

提供不同维度下的网络性能组合分析，可以从国家、城市、运营商、接入方式这几个方面实现组合监控，方便发现多种组合条件下的问题（图 8）。

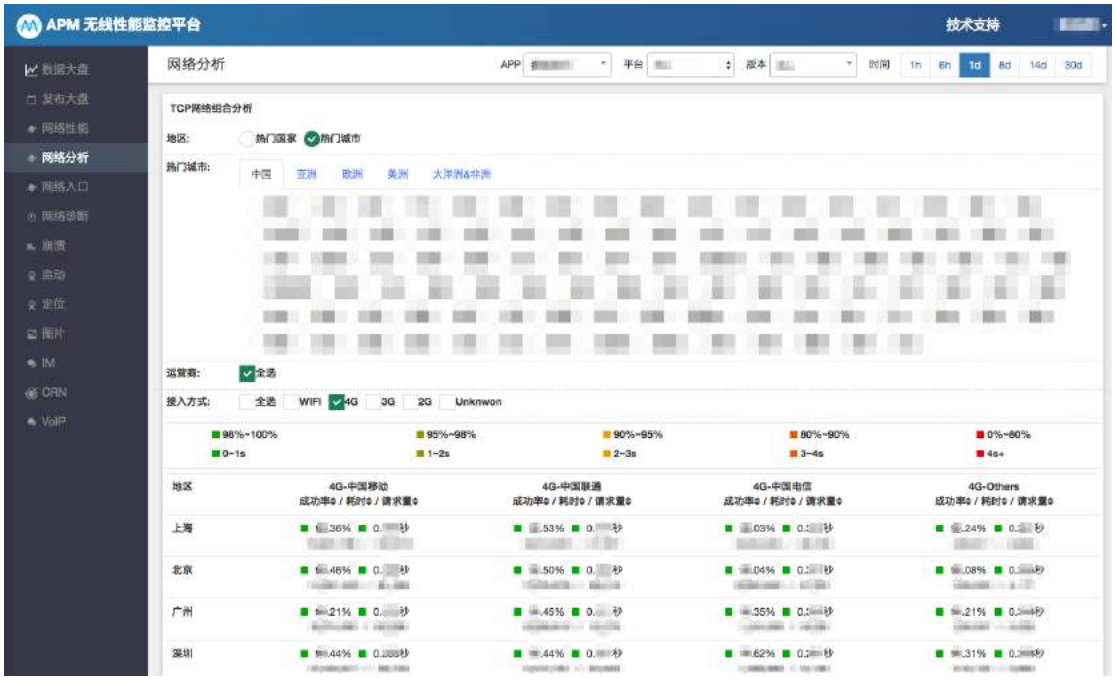


图 8 网络性能组合分析

2.1.3 网络入口性能

提供携程不同网络入口的性能聚合监控，为全球网络入口的优化提供依据（图 9）。

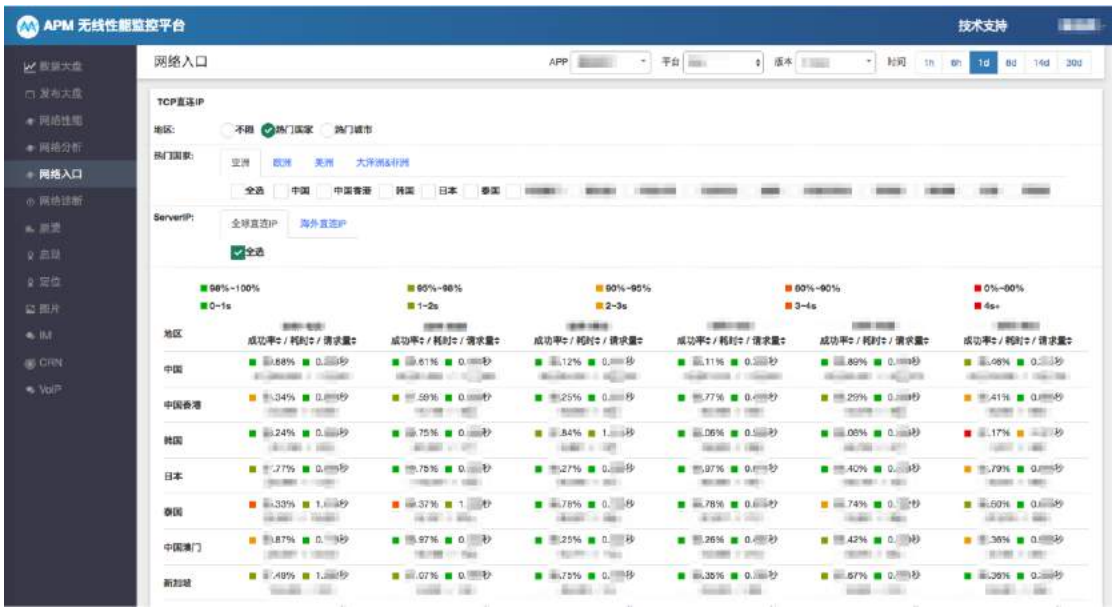


图 9 网络入口性能

2.1.4. 全球网络诊断（拨测功能）

通过 App 端实现对目标站点的性能拨测和诊断，包括 DNS 解析、TCP 连接、SSL 握手、Traceroute、Ping 类型数据都可以采集，由于涉及特殊功能，暂不公开更多细节。

2.2 崩溃

App 崩溃监控在业界已经非常成熟，难度在于崩溃数据的采集和分析能力（图 10）。

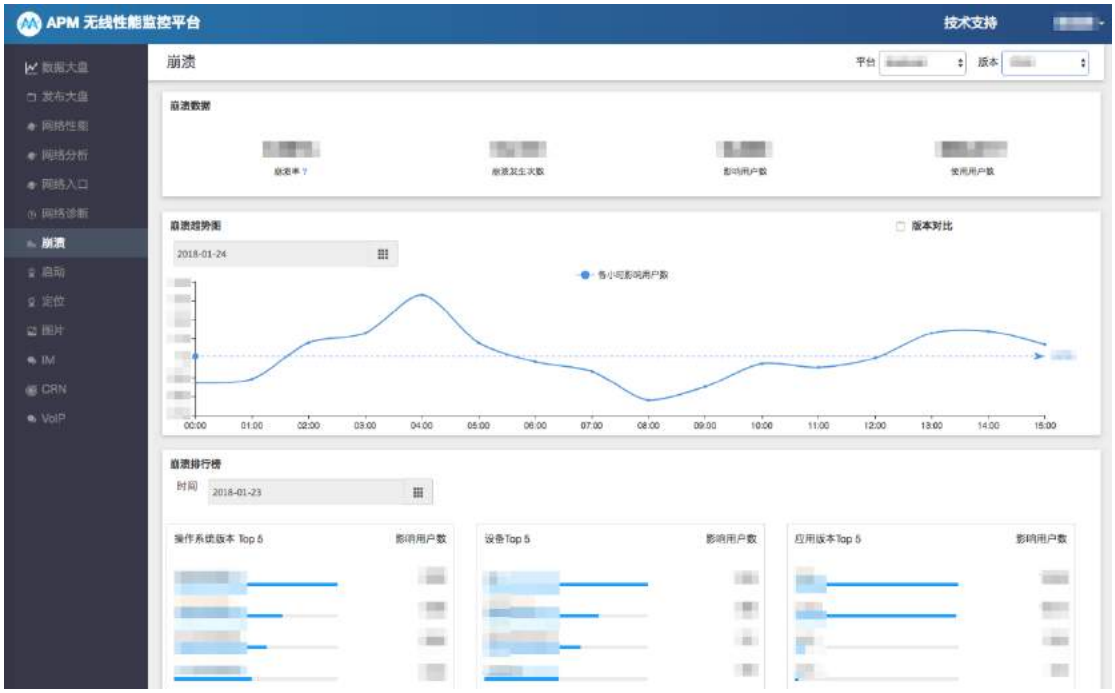


图 10 App 崩溃采集

2.3 启动加载

对 App 的冷启动时长、Android 安装后首次启动时长和 Android Bundle 启动加载时长进行监控。

2.4 定位

对 App 的经纬度定位和城市定位的成功率、平均耗时和请求量进行监控。

2.5 图片

对 App 的图片下载成功率、平均耗时和下载量进行监控，和网络性能类似提供了全球热门国家和城市的细分维度。

2.6 CRN

对使用携程 CRN 框架开发的 RN 模块加载数量和平均耗时进行监控，为 CRN 优化提供依据。

2.7 IM 和 VoIP

这两项都属于业务型技术指标的监控，例如对各类 IM 消息和 VoIP 通话的成功率、平均耗时和请求量进行监控，也提供全球热门国家和城市的细分维度。

从设计思路上可以看出新 APM 平台把重心放在全球化维度和 App 核心体验性能两方面，上线后已经通过 APM 发现一些热点地区的性能问题，未来我们会逐步分享相关优化实践。

三、APM 技术细节

目前新 APM 平台已实现 100+种的 Metric 监控，日处理数据量约 100+亿，日写入 ES(Elastic Search)数据量 100+G。其处理流程如下图所示：

图 11 APM 处理流程

1）App 客户端数据采集
2）Storm 流式计算，对数据进行滤噪和聚合
3）计算数据写入，以及进一步的归集
4）ES 数据存储
5）看板展示各种维度的组合数据

更多技术细节：

3.1 技术方案选型

APM 主要基于时序数据的存储和查询。我们没有采取目前比较流行的 InfluxDB，尽管在写入速度、数据存储以及查询响应上，都要优于 ES。我们采取 ES，主要是从携程 ES 平台运维成熟度、查询复杂度和系统数据量三个方面来综合考虑，ES 方案比较适合当前的技术场景。

3.2 数据量大情况下的 Trade Off

维度组合多，单单网络服务请求就可以划分为客户端版本、平台、网络类型、国家、城市等，按照 10 个版本，2 个平台，4 种网络类型，200 个国家，1000 个城市来计算维度，M = 10 x 2 x 4 x 200 x 1000 = 16,000,000，按照每分钟一个 Data Point，写入量就达到约 30W/秒，这个数据写入量量对系统压力是非常大的。

针对这个问题，我们采取 2 种方式进行数据处理：

1）降维：不是每一种监控都需要这样的模型，有的我仅仅是看国家，不关注客户端版本，那我们就可以把客户端版本这个维度去掉。

2）部分采样：比方说城市维度，全球有上万城市，对于监控来说，按照 80/20 原则，我们采集 20%热门城市进行监控。

3.3 噪音过滤

噪音数据会对监控造成比较大影响，比如 HTTP 请求时间，遇到奇点数据值会达到 30 秒以上（因客户端超时设置等影响），这类数据在进入 ES 前，先进行滤噪。对于各种需要滤噪的数据进行配置，在 Storm Extract 数据时进行数据处理。

四、结束语

技术平台自身需要生命力，相信 APM 平台也会随着业务重心的发展发生变化，希望我们的 APM 方案能够对大家有借鉴价值，也欢迎大家和我们交流自己的 APM 设计思路和技术方案。

Mvvm 前端数据流框架精讲

【作者简介】黄子毅，目前在阿里数据中台前端团队，负责数据产品相关业务。前端精读创办者、数据流框架 Dob 作者、可视化编辑器 gaea-editor 作者、react-native-image-viewer 作者、曾维护数套前端组件库。本文来自黄子毅在“携程技术沙龙——新一代前端技术实践”上的分享。

本文将带大家了解什么是 mvvm，mvvm 的原理，以及近几年产生了哪些演变。同时借 mvvm 话题，拓展到对各类前端数据流方案的思考，形成对前端数据流整体认知，帮助大家在团队中更好地做技术选型。

一、Mvvm 的概念与发展

1.1 Mvvm &单向数据流

Mvvm 是指双向数据流，即 View-Model 之间的双向通信，由 ViewModel 作桥接。如下图所示：

而单向数据流则去除了 View -> Model 这一步，需要由用户手动绑定。

1.2 生态 - 内置 & 解耦

许多前端框架都内置了 Mvvm 功能，比如 Knockout、Angular、Ember、Avalon、Vue、San 等等。

而就像 Redux 一样，Mvvm 框架中也出现了许多与框架解耦的库，比如 Mobx、Immer、Dob 等，这些库需要一个中间层与框架衔接，比如 mobx-react、redux-box、dob-react。解耦让框架更专注 View 层，实现了库与框架灵活搭配的能力。

解耦的数据流框架也诠释了更高抽象级别的 Mvvm 架构，即：View - 前端框架，Model - (mobx, dob)，ViewModel - (mobx-react, dob-react)。

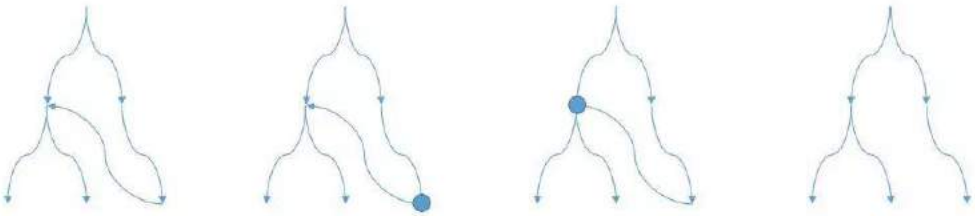
同时也实现了数据与框架分离，便于测试与维护。比如下面的例子，左边是框架无关的纯数据/数据操作定义，右边是 View + ViewModel：

```
import { observable, combineStores, inject } from 'dob'

@observable
class Store {
    name = 'bob'
}
class Action {
    @inject(Store) store

    setName () {
        this.store.name = 'lucy'
    }
}

export stores = combineStores({ Action, Store })
```

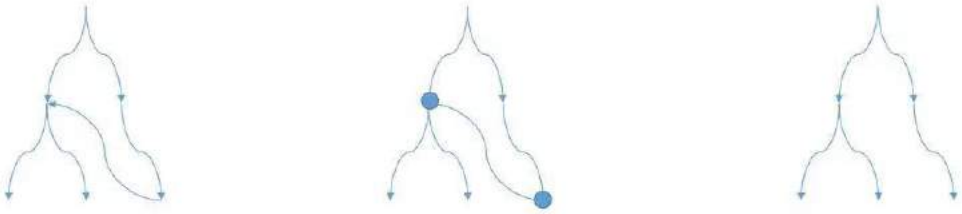
```
import * as React from "react"
import { Connect } from "dob-react"

@Connect(stores)
export default class Page extends React.PureComponent {
    render() {
        return (
            <div
                onClick={this.props.Action.setName}
            >{this.props.Store.name}</div>
        )
    }
}
```

1.3 运行效率 - 脏检测 & getter/setter 劫持

Angluar 早期的脏检测机制虽然开创了 mvvm 先河，但监听效率比较低，需要 N + 1 次确认数据是否有联动变化，就像下图所示：



现在几乎所有框架都改为 getter/setter 劫持实现监听，任何数据的变化都可以在一个事件循环周期内完成：



1.4 语法 - 特殊语法 & 原生语法

早期一些 Mvvm 框架需要手动触发视图刷新，现在这种做法几乎都被原生赋值语句取代。

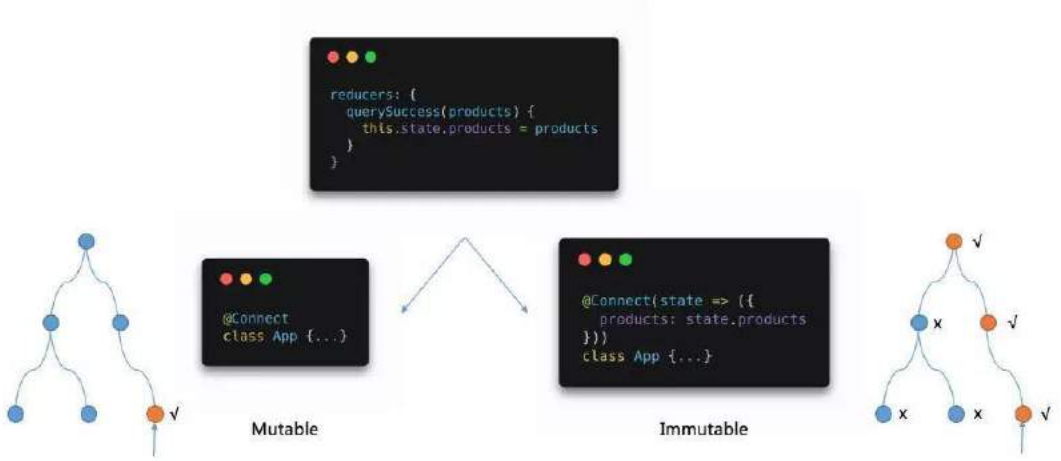
1.5 数据变更方式 - Mutable & Immutable

下图的代码语法虽为 mutable，但产生的结果可能是 mutable，也可能是 immutable，取决于 mvvm 框架内置实现机制：

```
const mutations = {
  SET_NAME: (state, action) => (state.name = 5),
  SET_NAME: (state, action) => (state.name ="bob")
}
```

1.6 Connect 的两种写法

由于 mvvm 支持了 mutable 与 immutable 两种写法，所以对于 mutable 的底层，我们使用左图的 connect 语法，对于 immutable 的底层，需要使用右图的 conenct 语法：



对左图而言，由于 mutable 驱动，所有数据改动会自动调用视图刷新，因此不但更新可以一步到位，而且可以数据全量注入，因为没用到的变量不会导致额外渲染。

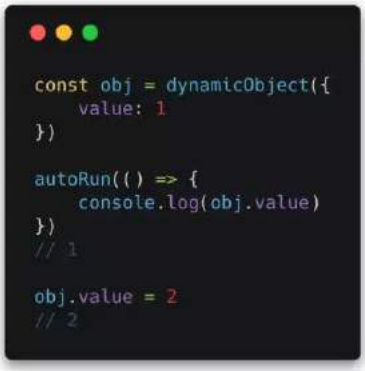
对右图，由于 immutable 驱动，本身并没有主动驱动视图刷新能力，所以当右下角节点变更时，会在整条链路产生新的对象，通过 view 更新机制一层层传导到要更新的视图。

二、从 TFRP 到 mvvm

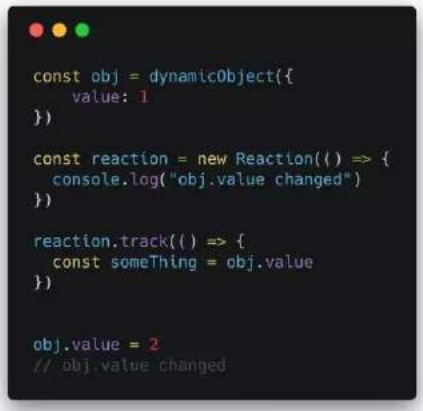
讲到 mvvm 的原理，先从 TFRP 说起，详细可以参考《dob-框架实现》，该文以 dob 框架为例子，一步步介绍了如何实现 mvvm。本文简单做个介绍。

2.1 autorun & reaction

autorun 是 TFRP 的函数效果，即集成了依赖收集与监听，autorun 背后由 reaction 实现。



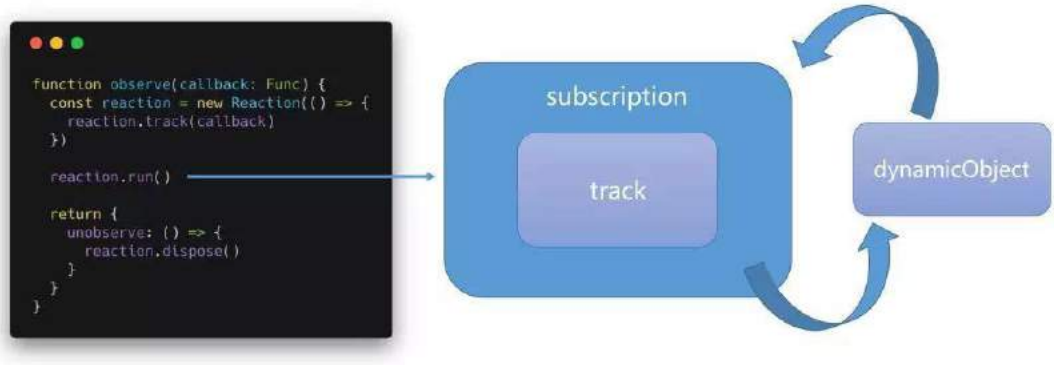
autorun



reaction

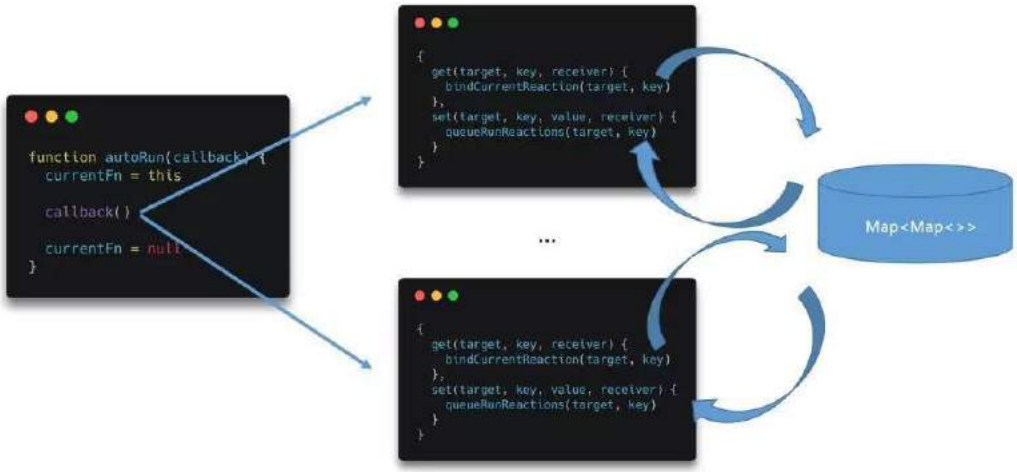
2.2 reaction 实现 autorun

如下图所示，autorun 是 subscription 套上 track 的 reaction，并且初始化时主动 dispatch，从入口（subscription）处激活循环，完成 subscription -> track -> 监听修改 -> subscription 完成闭环。



2.3 track 的实现

每个 track 在其执行期间会监听 callback 的 getter 事件，并将 target 与 properityKey 存储在二维 Map 中，当任何 getter 触发后，从这个二维表中查询依赖关系，即可找到对应的 callback 并执行。



2.4 View-Model 的实现

由于 autorun 与 view 的 render 函数很像，我们在 render 函数初始化执行时，使其包裹在 autorun 环境中，第 2 次 render 开始遍剥离外层的 autorun，保证只绑定一遍数据。

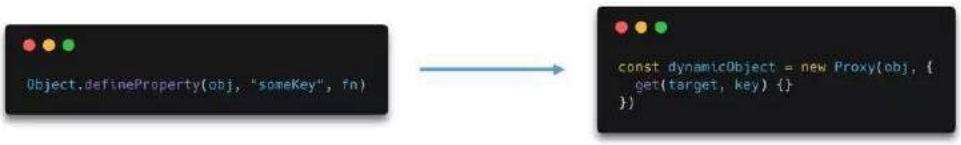
这样 view 层在原本 props 更新机制的基础上，增加了 autorun 的功能，实现修改任何数据自动更新对应 view 的效果。

三、Mvvm 的缺点与解法?

Mvvm 所有已知缺点几乎都有了解决方案。

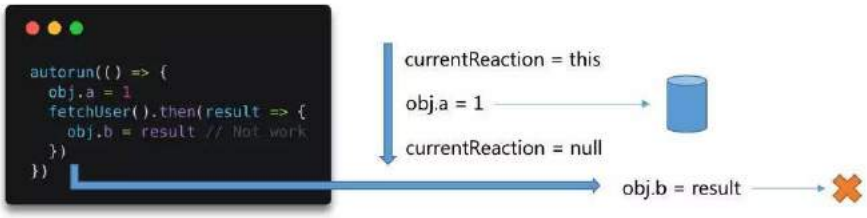
3.1 无法监听新增属性

用过 Mobx 的同学都知道，给 store 添加一个不存在的属性，需要使用 extendObservable 这个方法。这个问题在 Dob 与 Mobx4.0 中都得到了解决，解决方法就是使用 proxy 替代 Object.defineProperty：



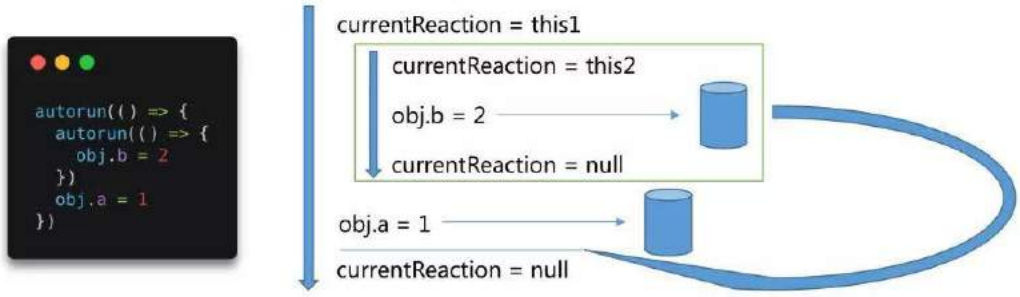
3.2 异步问题

由于 getter/setter 无法获得当前执行函数，只能通过全局变量方式解决，因此 autorun 的 callback 函数不支持异步：



3.3 嵌套问题

由于 reaction 特性，只支持同步 callback 函数，因此 autorun 发生嵌套时，很可能会打乱依赖绑定的顺序。解决方案是将嵌套的 autorun 放到执行队列尾部，如下图所示：



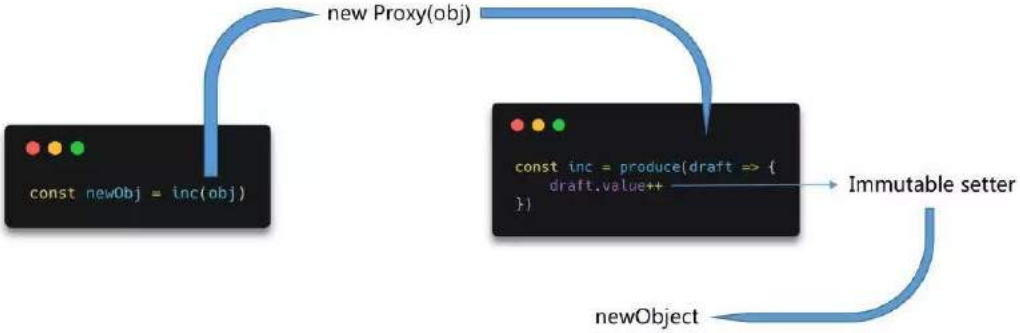
3.4 无数据快照

mutable 最被人诟病的一点就是无法做数据快照，不能像 redux 一样做时间回溯。有问题自然有人会解决，Mobx 作者的 Immer 库完美的解决了问题。

```
const inc = (obj) => {
  obj.value++
  return obj
}

const obj = { value: 1 }
const newObj = inc(obj)

// obj === newObj
```

```
const inc = produce(draft => {
    draft.value++
})

const obj = { value: 1 }
const newObj = inc(obj)

// obj !== newObj
```

原理是通过 proxy 返回代理对象，在内部通过浅拷贝替代对对象的 mutable 更改。具体原理可以参考我之前的一篇文章《精读 Immer.js 源码》。



3.5 无副作用隔离

mvvm 函数的 Action 由于支持异步，许多人会在 Action 中发请求，同时修改 store，这样就无法将请求副作用隔离到 store 之外。同时对 store 的 mutable 修改，本身也是一种副作用。

```
class Products {
  @inject(Store) store

  async query() {
    const result = await getProducts()
    this.store.products = result
  }
}
```

未隔离

```
class Products {
  @inject(Store) store

  async query() {
    const result = await getProducts()
    this.productReducer(result)
  }

  productReducer(products) {
    this.store.products = products
  }
}
```

隔离

虽然可以将请求函数拆分到另一个 Action 中，但人为因素无法完全避免。

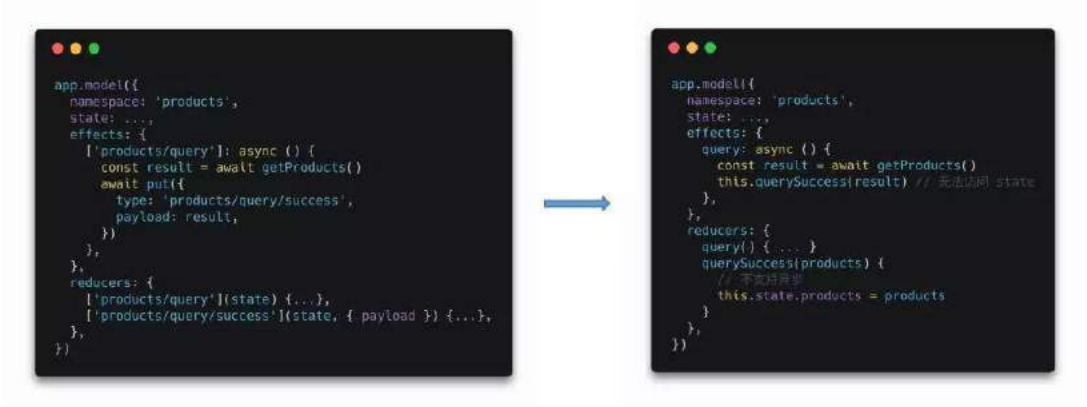
自从有了 Immer.js 之后，至少从支持元编程的角度来看，mutable 并不一定会产生副作用，它可以是零副作用的：

typescript function inc(obj) { return produce(obj => obj.count++) }

上面这种看似 mutable 的写法其实是零副作用的纯函数，和下面写法等价：

typescript function inc(obj) { return { count: obj.count + 1, ...obj } }

而对副作用的隔离，也可以做出类似 dva 的封装：



四、Mvvm store 组织形式

Mvvm 在项目中 stores 代码结构也千变万化，这里列出 4 种常见形式。

4.1 对象形式，代表框架 – mobx

mobx 开创了最基本的 mvvm store 组织形式，基本也是各内置 mvvm 框架的 store 组织形式。

4.2 Class + 注入，代表框架 – dob

dob 在 store 组织形式下了不少功夫，通过依赖注入增强了 store 之间的关联，实现

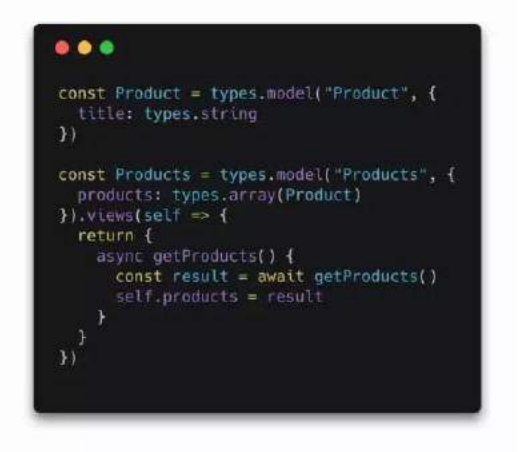
stores -> action 多对一的网状结构。

```
class Store {
  products = []
}

class Action {
  @inject(Store) store

  async getProducts() {
    const result = await fetchProducts()
    this.store.products = products
  }
}
```

- 灵活注入的能力
- Typescript 类型支持
- 无强制副作用隔离

4.3 数据结构化，代表框架 – mobx-state-tree

mobx-state-tree 是典型结构化 store 组织的代表，这种组织形式适合一体化 app 开发，比如很多页面之间细粒度数据需要联动。



```
const Product = types.model("Product", {
  title: types.string
})

const Products = types.model("Products", {
  products: types.array(Product)
}).views(self => {
  return {
    async getProducts() {
      const result = await getProducts()
      self.products = result
    }
  }
})
```

- Store 之间产生关联结构
- Typescript 类型支持
- 需要记住一套约定
- 无强制副作用隔离

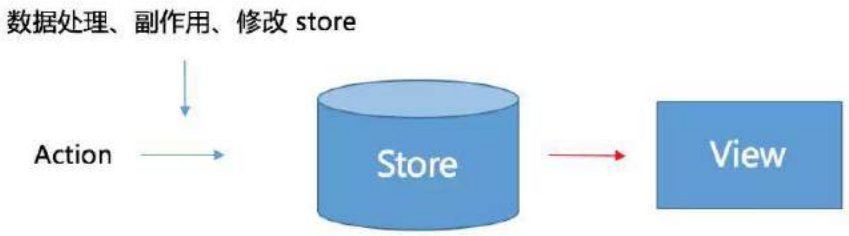
4.4 约定与集成，代表框架 – 类 dva

类 dva 是一种集成模式，是针对 redux 复杂的样板代码，思考形成的简化方案，自然集成与约定是简化的方向。

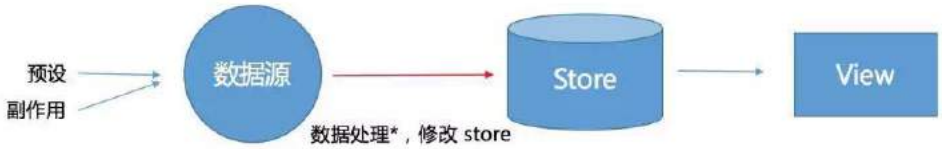
另外这种方案更像一层数据 dsl，得益于此，同一套代码可以拥有不同的底层实现。

五、Mvvm vs Reactive programming

Mvvm 与 Reactive programming 都拥有 observable 特性，通过下面两张图可以轻松区分：



上面红线是 mvvm 的 observable 部分，这里指的是数据变化的 autorun 动作。

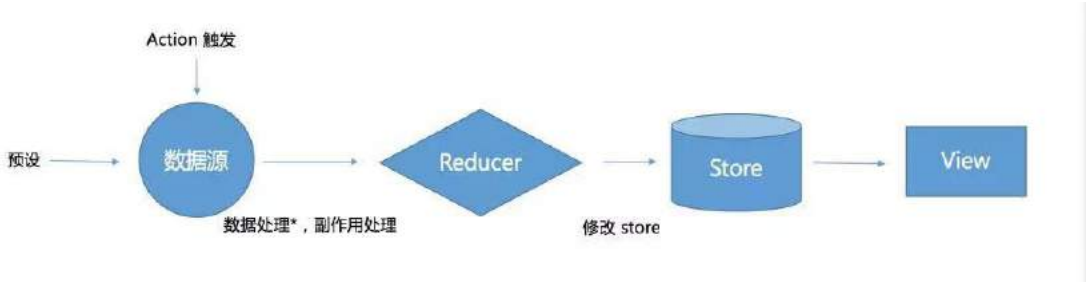


上面红线是 Reactive programming 的 observable 部分，指的是数据源派发流的过程。

Mvvm 与 Reactive programming 的结合

既然 redux 可以与 rxjs 结合（redux-observable），那么 mvvm 应该也可以如此。

下面是这种方案的构想：



rxjs 仅用来隔离副作用与数据处理，mvvm 拥有修改 store 的能力，并且精准更新使用的 View。

六、总结

根据业务场景指定数据流方案，数据流方案没有银弹，只有贴着场景走，才能找到最合适的方案。

了解到 mvvm 的发展与演进，让不同数据流方案组合，你会发现，数据流方案还有很多。

Android 工程模块化平台的设计

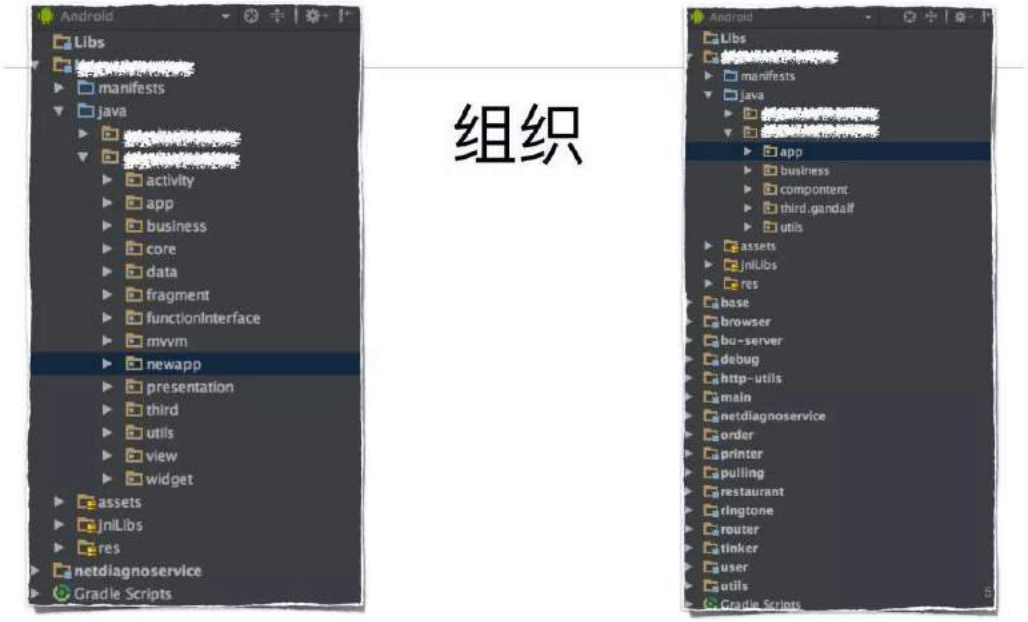
【作者简介】张涛，饿了么资深 Android 工程师，“开源实验室”博主，Kotlin 技术推广者。2013 年开始从事 Android 开发，带过团队，做过架构，写过应用，做过开源社区。目前在饿了么商户端负责应用模块化平台与插件化平台的设计和开发。本文来自张涛在“携程技术沙龙——无线技术工程化”上的分享。

本文的主题是基于项目模块化来说的，模块化其实跟项目重构很像，只是侧重点不同，分别是：删除、组织、降级、解耦。接下来将跟大家分享我是如何理解这四大块的。

一、模块化重构

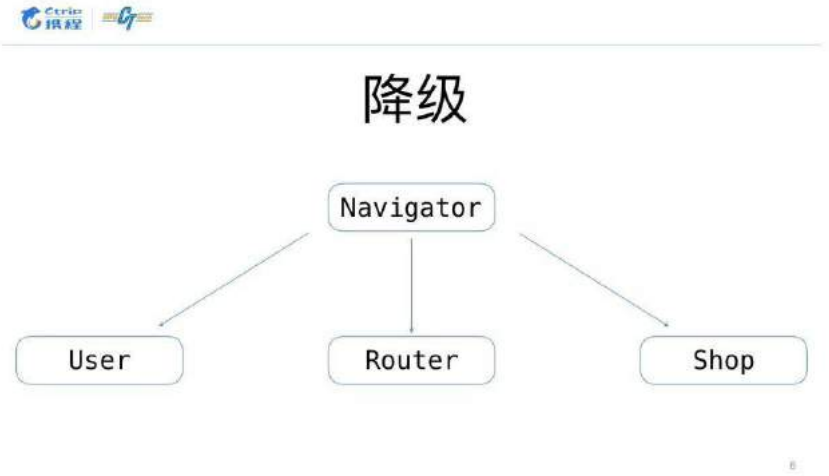
删除：删除不必要的文件，尽可能减小工程体积。这里有一组数据，是饿了么一款 APP 在模块化前后一些文件的数量。

可以看到，.java 文件从 1677 个减少到了 1543 个。其实这不是重点，重点是下面的 drawable，这里 drawable 只包含图片、和 xml 布局，当经过模块化重构后文件数从 693 减少到 538 个。图片资源减少接近 200 个，apk 的大小也会随之降低。



而组织呢，指的是：按照有意义的标准将代码分组。这其实也是 java 的包所存在的目的之一。

但是随着项目的不断迭代，需求很紧的情况下是很难有时间去真正规范的将类分组的。看到图中，我们之前的结构很乱，就是因为项目快速迭代和人员更替的过程中，难免会有这样的现象。所以这也是模块化重构时的一件大事。



接下来就是我们经常说的内聚和耦合了，降级。我们之前有一个类叫：Navigator，它负责几乎所有 Activity 直接跳转。我们会把所有的 startActivity()的跳转放到这个类里面去写。少的时候还好，等我看到这个类的时候，已经有 200 多个方法了，全是 Activity 跳转的方法。

在做模块化重构时，首先观察自己的项目，这是很重要的一步，要结合自身。把这个类拆分成三大部分，我们有两块业务是会频繁跳转的，但这两个业务跳转的页面又都是在自身的模

块内，分别是用户模块和商户模块。因此将这两个模块中分别建立两个用于模块自己内部的跳转叫 UserNavigator 和 ShopNavigator，而模块间的跳转或一些小模块内部的则使用 Router 去做。

之后解耦，如何优雅移除模块间的耦合。 到目前为止，我们能够做到让所有不包含业务状态接口的模块的增删，不需要改动任何一行代码。 一个示例：

或者，也可以是这样：

这两个段代码的区别，一个是手动管理 Debug 的状态，另一个是交给 Gradle 的编译任务去控制，原理上是一样的。

而这么做是如何实现的呢？其思路：一个模块就是一个功能，你想要让你的 apk 具备这个功能，就添加这个模块一起编译就可以了。这才是我们说的真正的组件化，模块之间零耦合，增减模块零改动。

例如图中：debug 这个模块，肯定不会用在正式的生产环境；而相反的 tinker 这个模块，热补丁肯定也不会用于调试阶段。所以在开发时就可以不使用这个模块相关的代码。

再举个使用的例子：我有一个订单模块，订单模块需要播放铃声，比如大家在饭店经常听到“您有新的饿了么订单，请及时处理”。但在开发订单模块的时候，如果已经确定铃声播放是没有问题的，那可以选择开发阶段不打铃声的包，直到发布到线上了，再去加上铃声的包。

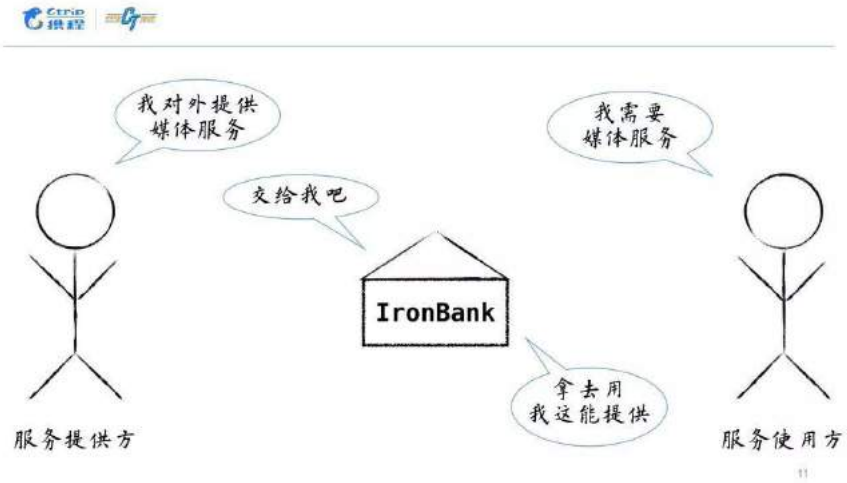
那没有添加铃声模块的时候，就默认不具备播放铃声的功能，但完全不影响其他的订单模块的业务功能，而这个铃声模块的增删，是不需要修改任何代码的。

听到这里，相信大家都很好奇是怎么实现的。接下来就跟大家分享下内部的原理。

二、铁金库解耦

所有的核心功能都来自我们自己写的一个库：IronBank。取《自冰与火之歌》中的【铁金库】，叫铁金库不容拖欠。

铁金库的内部实现，其实是使用了 APT 注解处理器，在编译时解析注解生成一个类，让这个类去生成跨模块的对象。铁金库使用了与后端 SOA 设计思路类似的方式：将模块之间的主动依赖倒置，变为功能的提供与使用。



例如图上左边有一个对外提供媒体功能的服务提供者，他告知 IronBank 我提供媒体服务：“嘿，老铁，我这有个媒体服务，你那边有谁要用的时候可以用我的。”

到了另一边，如果此刻有模块说是，我需要媒体服务：“老铁，你那有没有媒体服务，我这边需要播一个铃声啊！”。

“有的，给你。”

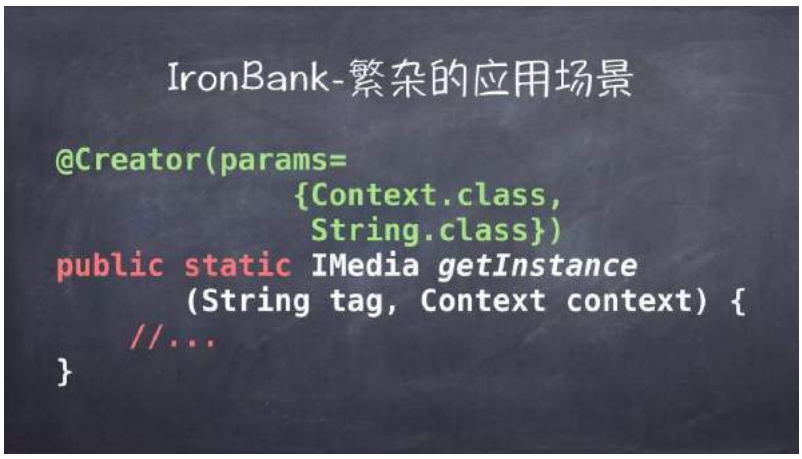
IronBank 就会将之前服务提供者提供给他的媒体对象交给服务使用者。

接下来我们来看具体到代码上是如何使用的：首先是作为服务使用方，也就是上一张图右半部分，传统的做法是先声明一个接口类型，然后 new 出接口的实现类给他赋值。

而使用了 IronBank 的时候，你是不需要关心接口的实现类到底是谁的。这就是 IronBank 唯一的用处，隐藏实现类，做到彻底的面相接口编程。

IronBank 将模块之间依赖倒置，由之前的服务提供方被动的接受调用方调用变为，服务方主动提供服务给调用方。

那作为服务提供方需要做些什么事呢？非常简单，你只需要给你的对象提供 public static 方法，并加上一个@Creator 注解，告诉 IronBank 这是一个创建器方法就可以了，其他任何事情，都不需要考虑。

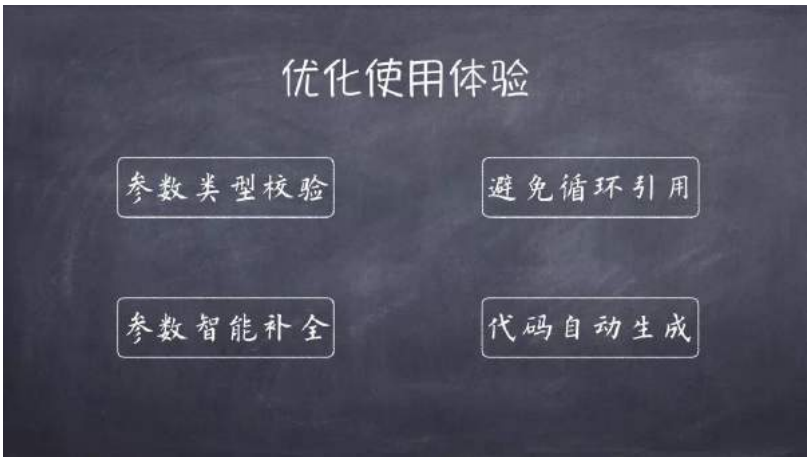


这里的创建器方法是可以有参数的，在接收时实际是使用另一个变长 Object 参数来接收。

而相对于繁杂的应用场景，也有对应的解决办法，例如这里的创建器方法是含参数的。看到示例第一个参数是 tag，第二个是 context 。但是你希望调用者在传的时候将 Context 作为第一个参数，tag 作为第二个参数。

那你在声明的时候，需要显示的声明参数，加一个 params，然后写上你希望的参数顺序。

这个@Creator 注解里面还有很多参数，比如这里返回的是 IMedia 类型的对象，如果 IMedia 接口还继承了一个 A 接口，这里我虽然返回的是 IMedia，但我不想外部知道，我就想外部知道我返回的只是个 A，这样也是可以显示的，在注解参数中声明就行了。以及还有方法的类所在文件自定义等等等等…… 就不一一列举了。



在使用上，为了接入方使用方便，我们也对 IronBank 做了非常多的体验优化。

我们通过自定义 lint 来使 IDE 可以检查参数类型是否正确。比如前面举的例子，如果声明的时候第一个参数是 String，第二个参数是 Context，如果你传错了，IDE 直接就报红了。

还有前面我们看到了，IronBank 提供了一种类似依赖注入的方式去创建对象，既然是类似依赖注入，一定会碰上循环引用问题。我们自定义的 Lint，也完美通过静态代码分析，在编译

前就避免了这个问题。

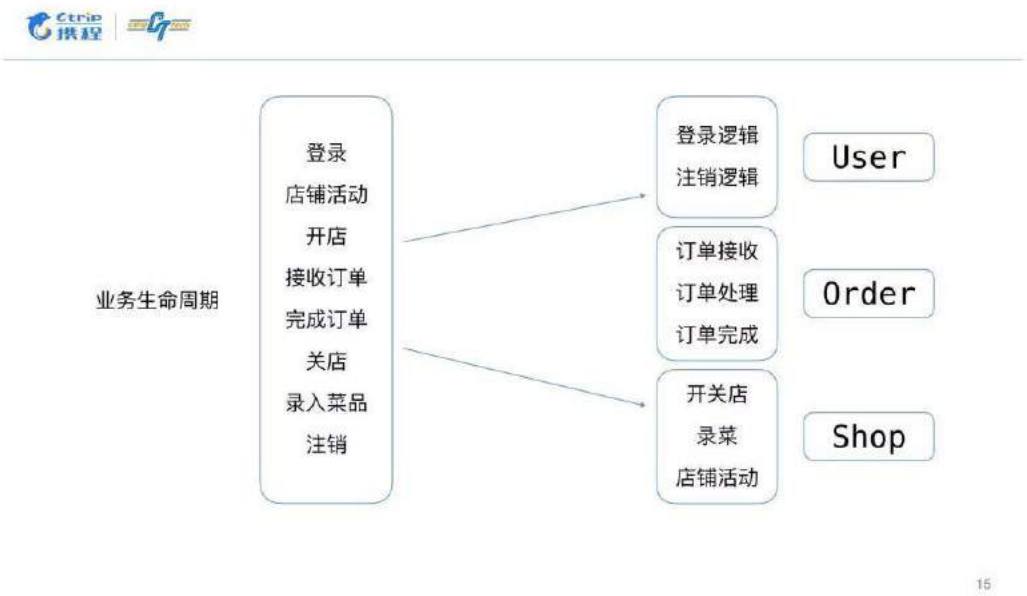
同时在开发的时候还提供了一个 Android Studio IDE 插件，可以用来帮你把参数智能补全，自动生成代码。前面看到，在写 IronBank.get()方法的时候得写很多字，如果有智能补全会少写很多。

三、业务状态解耦

前面讲的 IronBank 适用的场景是无状态的服务，而做业务 APP 开发的时候，更多的是有业务状态的对象。比方说通常长链与推送功能是等到用户登录了以后才会去启动，但具体到代码上，推送模块是根本不知道用户什么时候登录的，这就是一个业务状态的问题。

对此我们引入了 BizLifecycle 的接口，它与 Android 上的 Application 对象功能类似。只不过它用来管理的是业务的生命周期，而不是应用的。

在代码逻辑上，每个模块如果关心你所需要的业务生命周期，只需要注册一个 Lifecycle 就行了，同时注册的过程也只需要一个注解，由编译插件解决了。



可以看到，其实这样的一种能力用事件通知也可以做到，比方说广播或者 EventBus，但是我们刻意屏蔽了这种方式，就是因为事件通知这种功能你是很难去追踪的。你不知道一个消息发送了以后，它的接受者是在哪里。

相信大家也能够想象到，一个应用如果广播泛滥，到处都是事件接收事件发送，项目代码会变得多么吓人。

讲到这里，整个模块化解耦的全部能力就介绍完了。

接下来，我们再从宏观角度去看一下整个项目的结构，分为三级，最上层是业务模块，紧接着是一些可选的功能组件，最底层则是与项目无关的公共依赖。

最终，项目结构就是如图中所示的这样。但如果你真直接这么做，你一定是会烦死的。

为什么？

第一：这么多的模块，直接用源码依赖去编译，编译时间至少在 10 分钟以上；
第二：模块的隔离几乎为 0，任何一个人依旧可以修改任何一个模块的代码，并且很容易；
第三：在发版本以后，如果某一个模块有 BUG，再去修复，缺乏一个版本的概念，尤其是在跨团队的时候，最终一定会出现版本分裂问题。

四、平台化支持

解决办法我想大家都知道，就是将模块引用改为 aar 引用。aar 引用最大的优势就在于模块

版本的管理与跨团队的协作。

目前国内对 Android 领域的探索越来越深，应用规模也越来越大，为了降低大型项目的复杂性和耦合度，同时也为了适应模块重用、多团队并行开发测试等等需求，必须有一套合适的模块化平台。

这里是饿了么目前使用的模块化平台，大家可以从这张图中感受一下。

模块化平台，主要的功能是很明显的，就是用于构建模块，在这之上，还有隐含的功能，就是集中了构建模块的权限，可以更便于统一管理；

最重要的优势在于模块版本的管理，你可以很清晰的知道当前主应用所接入的模块的版本是哪个，当前最新构建的 SNAPSHOT 是哪个，以及每个版本的更新日志；

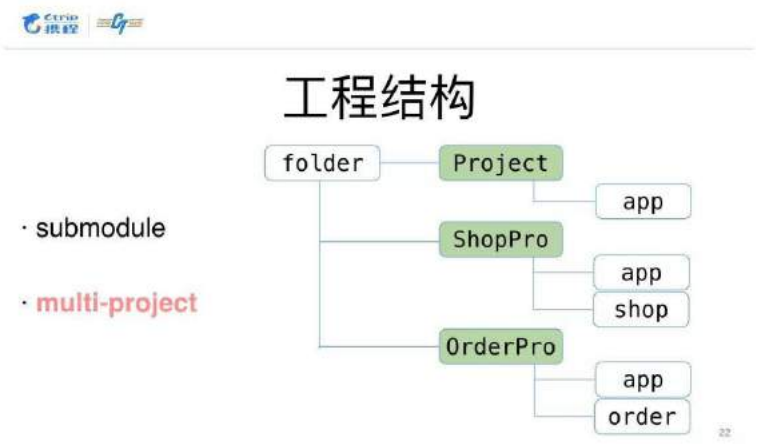
这样做了以后，在跨团队协作上的沟通就大大降低了，如果你已经接入或者即将接入的模块是另一个团队开发的模块组件，那你可以直接关注它，它的所有版本变动日志，最新版本全都一目了然；

并且可以通过平台简化模块的测试与模块发布的流程，比如提测的时候，如果是一次兼容版本的发布，你只需要告诉测试提测分支，测试可以自己根据现在线上应用的 tag，同时引入当前提测的模块替换老版本的模块重新编译，很容易就能控制变量。

引入了平台化以后，我们再从工程结构的角度看一下：就目前尝试下来，这两种结构是最合适 Android 工程模块化的。一种是 submodule，一种是 multi-project。

首先看 submodule：这种结构是 Android 默认的多模块结构，在一个工程下面有多个模块。图上每个绿色的方块都代表了一个 git 仓库，所有子模块都包含在主工程模块内。这种结构也是 git 默认支持的 submodule 结构，你只需要用最下面的这句 git 命令就可以将他们关联在一起。

它的好处就是所有都是默认的，任何一个人理解起来都是很直观。当然，它也有不适合的，就是协作开发的时候，所有人都在 app module 上测试自己的模块，很容易互相影响，主工程的 git 分支也会非常繁杂。



与之对应的，multi-project 能很好的解决这个问题：所有模块都是一个独立的工程，他们在文件系统上是并列关系，每个模块所在的工程才是一个 git 仓库。

对于单模块的操作达到最大化的遍历当然也是有牺牲的，就是这种结构很不利于全仓库整体的管理，对新人很不友好。比如我想所有模块同时初始化，同时切换到 develop 分支，对此，我们内部的处理方式是通过 shell 脚本达到全仓库批量处理。

同时还会对工程名会有一定的规范要求(非必须)，主要原因是在模块联调的时候。

我们看到这段代码是写在 setting.gradle 文件中的，他根据读取本地的 local.properties 文件，来 include 一个模块的源码，方便在模块联调的时候可以很容易的修改多模块的代码。

但是要求每个模块工程的文件夹名称是以模块名加上 Project 这样来命名，比如 order 模块所在的工程文件夹名就叫 OrderProject。 当然，你也可以不遵守，只不过不遵守就得写更多代码，我这里是直接用了循环，不遵守的话可能就需要把循环拆开手敲了。

以上两种工程结构各有各的好处，没有好坏，只有合不合适，我们内部两种结构也都有团队在用。

这里是模块联调的注意事项，如果你模块是以源码引入的，可能还有其他模块引用了同样模块的 aar，就会造成冲突，需要自己判断一下，加个自定义方法也好，用编译插件也可以，都能做到让源码引用与 aar 引用互斥。

模块化架构主要思路就是分而治之，在拆分的时候最重要的是，把依赖整理清楚，哪些是业务模块，哪些是可选的功能组件。最后为了团队方便以及更快的适应，还需要开发一些辅助

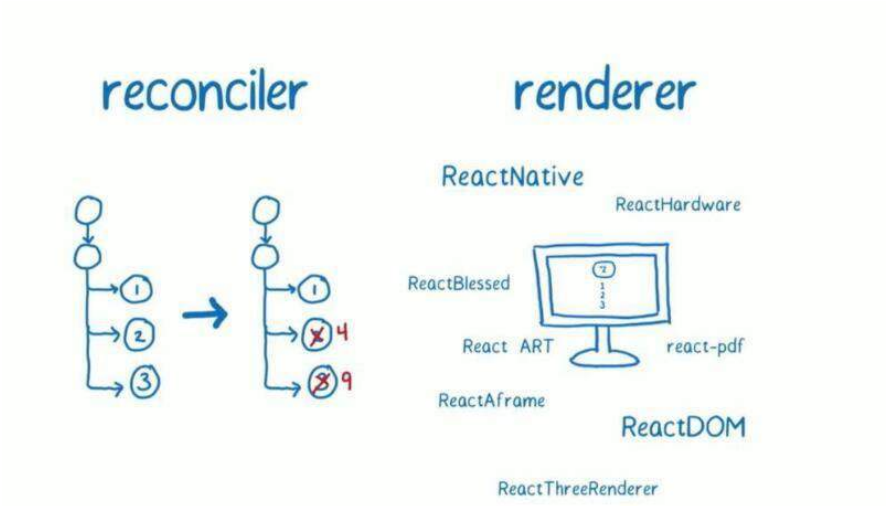
工具，比如前面说的 IronBank、BizLifecycle、初始化脚本等等，都是必不可少的。

React Fiber 初探

【作者简介】傅崇琛，携程境外专车研发部前端工程师，3 年前端相关工作经验，主要负责境外接送机业务部门前端相关框架/库的开发与维护。

React Fiber 是对 React 核心算法的重构，2 年重构的产物就是 Fiber Reconciler。

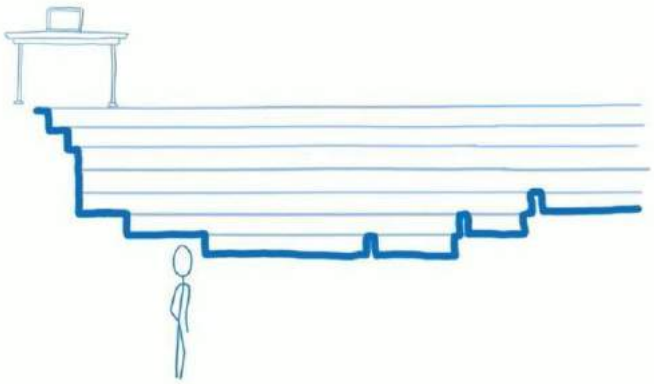
一、为什么需要 React Fiber



在 React Fiber 之前的版本，当 React 决定要加载或者更新组件树时，会做很多事，但主要是两个阶段：

调度阶段（Reconciler）：这个阶段 React 用新数据生成新的 Virtual DOM，遍历 Virtual DOM，然后通过 Diff 算法，快速找出需要更新的元素，放到更新队列中去。

渲染阶段（Renderer）：这个阶段 React 根据所在的渲染环境，遍历更新队列，将对应元素更新。在浏览器中，就是跟新对应的 DOM 元素。除浏览器外，渲染环境还可以是 Native、硬件、VR 、WebGL 等等。



表面上看，这种设计也是挺合理的，因为更新过程不会有任何 I/O 操作，完全是 CPU 计算，所以无需异步操作，执行到结束即可。

主要问题出现在，React 之前的调度策略 Stack Reconciler。这个策略像函数调用栈一样，会深度优先遍历所有的 Virtual DOM 节点，进行 Diff。它一定要等整棵 Virtual DOM 计算完成之后，才将任务出栈释放主线程。而浏览器中的渲染引擎是单线程的，除了网络操作,几乎所有的操作都在这个单线程中执行：解析渲染 DOM tree 和 CSS tree、解析执行 JavaScript，这个线程就是浏览器的主线程。

假设更新一个组件需要 1ms，如果有 200 个组件要更新，那就需要 200ms，在这 200ms 的更新过程中，浏览器唯一的主线程都在专心运行更新操作，无暇去做任何其他的事情。

想象一下，在这 200ms 内，用户往一个 input 元素中输入点什么，敲击键盘也不会获得响应，因为渲染输入按键结果也是浏览器主线程的工作，但是浏览器主线程被 React 占用，抽不出空，最后的结果就是用户敲了按键看不到反应，等 React 更新过程结束之后，那些按键会一下出现在 input 元素里，这就是所谓的界面卡顿。

React 这样的调度策略对动画的支持也不好。如果 React 更新一次状态，占用浏览器主线程的时间超过 16.6ms，就会被人眼发觉前后两帧不连续，呈现出动画卡顿。

过去的优化都是停留在 JavaScript 层面（Virtual DOM 的 create/diff）：如减少组件的复杂度（Stateless）、减少向下 diff 的规模（SCU）、减少 diff 的成本（immutable.js）...这些都并不能解决线程的问题。React 希望通过 Fiber 重构来改变这种现状，进一步提升交互体验。

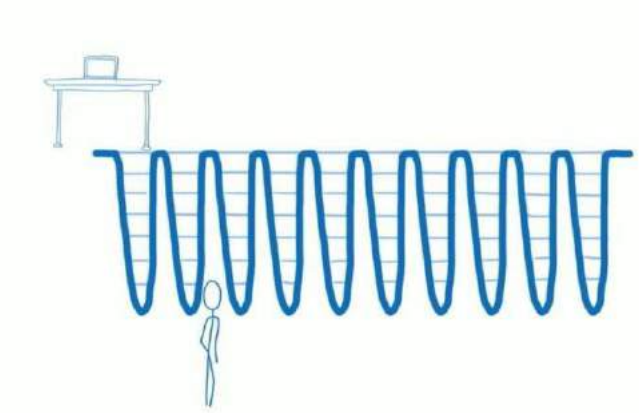
二、什么是 React Fiber

2.1 fiber tree

React Fiber 之前的 Stack Reconciler，在首次渲染过程中构建出 Virtual DOM tree，后续需要更新时，diff Virtual DOM tree 得到 DOM change，并把 DOM change 应用（patch）到 DOM 树。

其运行时存在以下三种实例：

- DOM： 真实的 DOM 节点。
- Elements：主要是描述 UI 长什么样子（type， props）。
- Instances： 根据 Elements 创建的，对组件及 DOM 节点的抽象表示，Virtual DOM tree 维护了组件状态以及组件与 DOM 树的关系。

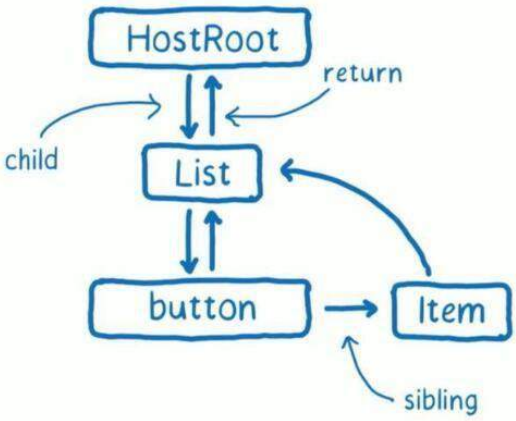


React Fiber 解决过去 Reconciler 存在的问题的思路是把渲染/更新过程（递归 diff）拆分成一系列小任务，每次检查树上的一小部分，完成后确认否还有时间继续下一个任务，存在时继续，不存在下一个任务时自己挂起，主线程不忙的时候再继续。

React Fiber 将组件的递归更新，改成链表的依次执行，扩展出了 fiber tree，即 Fiber 上下文的 Virtual DOM tree，更新过程根据输入数据以及现有的 fiber tree 构造出新的 fiber tree（workInProgress tree）。

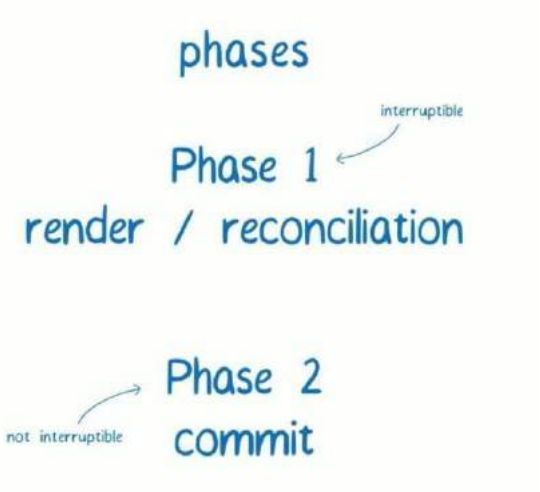
因此，运行时的实例变更为这种结构：

- DOM： 真实的 DOM 节点。
- effect： 每个 workInProgress tree 节点上都有一个 effect list 用来存放 diff 结果，当前节点更新完毕会 queue 收集 diff 结果，向上 merge effect list。
- workInProgress：workInProgress tree 是 reconcile 过程中从 fiber tree 建立的当前进度快照，用于断点恢复。
- fiber：fiber tree 与 Virtual DOM tree 类似，用来描述增量更新所需的上下文信息。
- Elements：主要是描述 UI 长什么样子（type， props）。



fiber tree 实际上是个单链表（Singly Linked List）树结构。Fiber 的拆分单位是 fiber tree 上的一个节点 fiber，按 Virtual DOM 节点拆，因为 fiber tree 是根据 Virtual DOM tree 构造出来的，树结构一模一样，只是节点携带的信息有差异。所以，实际上 Virtual DOM node 粒度的拆分以 fiber 为工作单元，每个组件实例和每个 DOM 节点抽象表示的实例都是一个工作单元。工作循环中，每次处理一个 fiber，处理完可以中断/挂起整个工作循环。

2.2 reconcile



React Fiber 把渲染/更新过程分为两个阶段：

1）可中断的 render/reconciliation 通过构造 workInProgress tree 得出 change。
2）不可中断的 commit 应用这些 DOM change。

第一阶段 render/reconciliation 具体实现为以 fiber tree 为蓝本，把每个 fiber 作为一个工作单元，自顶向下逐节点构造 workInProgress tree（构建中的新 fiber tree）。

以组件节点为例，具体过程如下：

1）如果当前节点不需要更新，直接把子节点 clone 过来，跳到 5；要更新的话打个 tag。
2）更新当前 props， state， context 等节点状态。
3）调用 should Component Update()，false 的话，跳到 5。
4）调用 render()获得新的子节点，并为子节点创建 fiber。创建过程会尽量复用现有 fiber，子节点增删也发生在这里。
5）如果没有产生 child fiber，该工作单元结束，把 effect list 归并到 return，并把当前节点的 sibling 作为下一个工作单元；否则把 child 作为下一个工作单元。
6）如果没有剩余可用时间了，等到下一次主线程空闲时才开始下一个工作单元；否则，立即开始做。
7）如果没有下一个工作单元了，回到了 workInProgress tree 的根节点，第 1 阶段结束，进入 pending Commit 状态。

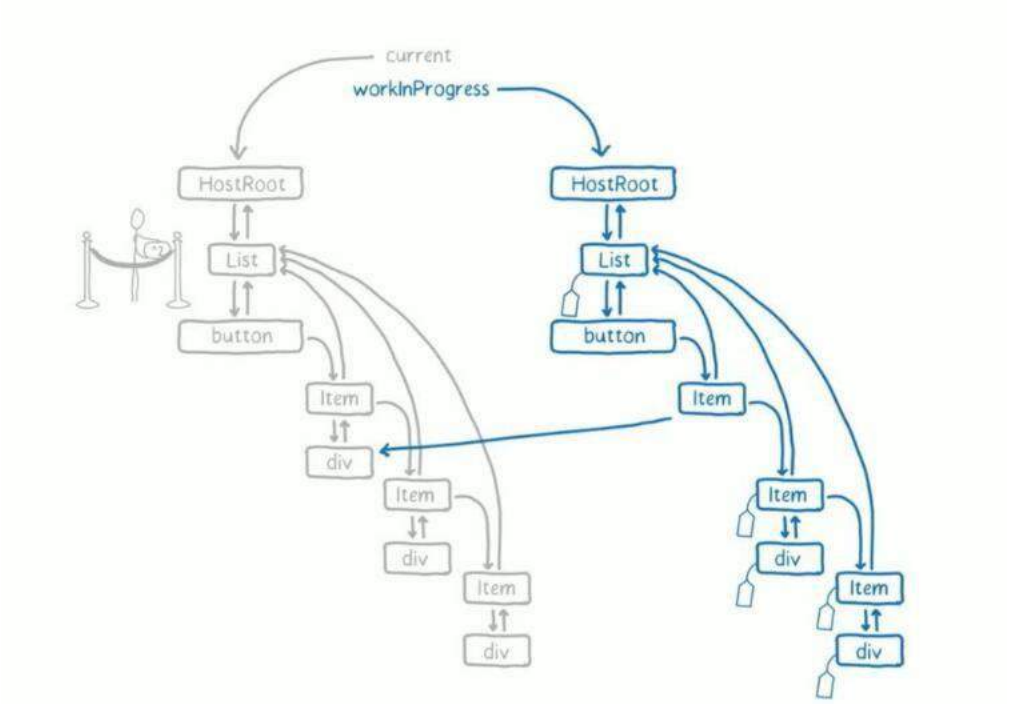
实际上是 1->6 的工作循环，7 是出口，工作循环每次只做一件事，做完看要不要休息。工作循环结束时，因为每做完一个都向上归并，workInProgress tree 根节点身上的 effect list 就是收集到的所有 side effect。

所以，构建 workInProgress tree 的过程就是 diff 的过程，通过 request Idle Callback 来调度执行一组任务，每完成一个任务后回来看看有没有插队的（更紧急的），每完成一组任务，把时间控制权交还给主线程，直到下一次 request Idle Callback 回调再继续构建 workInProgress tree。

这一阶段是没有副作用的，因此这个过程可以被打断，然后恢复执行。

第二阶段 commit：第一阶段产生的 effectlist 只有在 commit 之后才会生效，也就是真正应用到 DOM 中。这一阶段往往不会执行太长时间，因此是同步（所谓的一次性）的，这样也避免了组件内视图层结构和 DOM 不一致。

2.3 workInProgress tree



在 React Fiber 中使用了双缓冲技术（double buffering），像 redux 里的 nextListeners，以 fiber tree 为主，workInProgress tree 为辅。

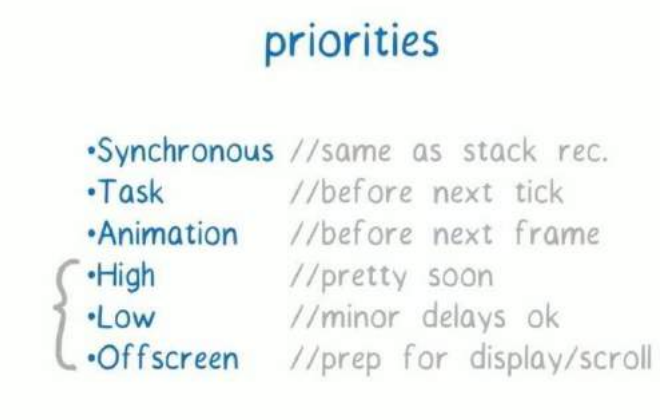
双缓冲具体指的是 workInProgress tree 构造完毕，得到的就是新的 fiber tree，每个 fiber 上都有个 alternate 属性，也指向一个 fiber，创建 workInProgress 节点时优先取 alternate，没有的话就创建一个。

fiber 与 workInProgress 互相持有引用，把 current 指针指向 workInProgress tree，丢掉旧的 fiber tree。旧 fiber 就作为新 fiber 更新的预留空间，达到复用 fiber 实例的目的。

2.4 优先级策略

React Fiber 为了更好的进行任务调度，会给不同的任务设置不同优先级。

React Fiber 切分任务并调用 requestIdleCallback 和 requestAnimationFrame API，保证渲染任务和其他任务，在不影响应用交互，不掉帧的前提下，稳定执行。而实现调度的方式正是给每一个 fiber 实例设置到期执行时间，不同时间即代表不同优先级，到期时间越短，则代表优先级越高，需要尽早执行。



synchronous 首屏（首次渲染）用，要求尽量快，不管会不会阻塞 UI 线程。animation 通过 requestAnimationFrame 来调度，这样在下一帧就能立即开始动画过程；后 3 个都是由 requestIdleCallback 回调执行的；offscreen 指的是当前隐藏的、屏幕外看不见的元素。高优先级的比如希望立即得到反馈的键盘输入，低优先级的比如网络请求，让评论显示出来等等。另外，紧急的事件允许插队。

2.5 生命周期

因为 render/reconciliation 阶段可能执行多次，会导致 willXXX 钩子执行多次。所以 getDerivedStateFromProps 取代了原本的 componentWillMount 与 componentWillReceiveProps 方法，而 componentWillUpdate 本来就是可有可无所以也被废弃了。

进入 commit 阶段时，组件多了一个新钩子叫 getSnapshotBeforeUpdate，它与 commit 阶段的钩子一样只执行一次。

出错时，在 componentDidMount/Update 后，可以使用 componentDidCatch 方法。

三、总结

React Fiber 最终提供的新功能主要是：

- 可切分，可中断任务。
- 可重用各分阶段任务，且可以设置优先级。
- 可以在父子组件任务间前进/后退切换任务。
- render 方法可以返回多元素（即可以返回数组）。
- 支持异常边界处理异常。

通过本文主要了解了 React Fiber 的基本实现和其产生的原因，其本身的还有更多的优化和实现细节，可以查看源码或参考其他文章进行更深入的了解。

参考文章

1）react-fiber-architecture
来源： https：//github.com/acdlite/react-fiber-architecture
2）Lin Clark
来源： https://conf.reactjs.org/speakers/lin

Kotlin 超棒的语言特性

【作者简介】何伦，携程度假 BU 移动端资深研发经理，负责 iOS、Android 平台上跟团游产品预订流程的前端页面的研发工作。对新技术有着浓厚的兴趣。

自从 2017 年 Google 宣布 Kotlin 成为 Android 官方开发语言之后，Kotlin 受到广大 Android 开发者的追捧。其强大的安全性，简洁性和与 Java 的互操作性，为开发者带来了耳目一新的开发体验，也极大提升了 Android 原生代码的开发效率。

不过大部分开发者对 Kotlin 的使用，仍然局限于把 Java 代码逻辑按照 Kotlin 语法进行转换的层面，其实 Kotlin 和 Java 虽然具有很强的互操作性，但本质上还是两种完全不同设计思想的语言。

本文在假定读者有一定 Kotlin 开发基础的前提下，详细讲解一些具有 Kotlin 特色的实用的语言特性，帮助开发者能够写出更加“具有 Kotlin 风格”的代码。这些语言特性包括空安全、Elvis 表达式、简洁字符串等等。

一、更加安全的指针操作

在 Kotlin 中，一切皆是对象。不存在 int, double 等关键字，只存在 Int, Double 等类。

所有的对象都通过一个指针所持有，而指针只有两种类型：var 表示指针可变，val 表示指针不可变。为了获得更好的空安全，Kotlin 中所有的对象都明确指明可空或者非空属性，即这个对象是否可能为 null。

```
//类型后面加?表示可为空
var age: String? = "23"
//不做处理返回 null
val age1 = age?.toInt()
//age为空返回-1
val age2 = age?.toInt() ?: -1
//抛出空指针异常
val age3 = age!!.toInt()
```

对于可空类型的对象，直接调用其方法，在编译阶段就会报错。这样就杜绝了空指针异常 NullPointerException 的可能性。

```
//类型后面加？表示可为空
val age:String? = "30"
//直接调用其方法，编译时即报错
val agei:Int = age.toInt()
```

如上图，编译器会报错
Error:Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String

二、？表达式和 Elvis 表达式

Kotlin 特有的？表达式和 Elvis 表达式可以在确保安全的情况下，写出更加简洁的代码。比如我们在 Android 页面开发中常见的删除子控件操作，用 Java 来写是这样的：

```
if (view != null) {
    if (view.getParent() != null) {
        if (view.getParent() instanceof ViewGroup) {
            ((ViewGroup) view.getParent()).removeView(view);
        }
    }
}
```

为了获得更加安全的代码，我们不得不加上很多 if else 判断语句，来确保不会产生空指针异常。但 Kotlin 的？操作符可以非常简洁地实现上述逻辑：

```
(view?.parent as? ViewGroup)?.removeView(view)
```

那么这个？表达式的内在逻辑是什么呢？以上述代码为例，若 view == null，则后续调用均不会走到，整个表达式直接返回 null，也不会抛出异常。也就是说，？表达式中，只要某个操作对象为 null，则整个表达式直接返回 null。

除了？表达式，Kotlin 还有个大杀器叫 Elvis 表达式，即？： 表达式，这两个表达式加在一起可以以超简洁的形式表述一个复杂逻辑。

```
val v = a?.b ?: c
```

以上面表达式为例，我们以红线把它划分成两个部分。若前面部分为 null，则整个表达式返回值等于 c 的值，否则等于前面部分的值。把它翻译成 Java 代码，是这样的

```
var temp = if(a != null) a.b else null
val v = if(temp != null) temp else c
```

同样等同于这样

```
val v = if(a == null || a.b == null) c else a.b
```

即 Elvis 表达式的含义在于为整个 ？表达式托底，即若整个表达式已经为 null 的情况下，Elvis 表达式能够让这个表达式有个自定义的默认值。这样进一步保证了空安全，同时代码也不失简洁性。

三、更简洁的字符串

同 Java 一样，Kotlin 也可以用字面量对字符串对象进行初始化，但 Kotlin 有个特别的地方是使用了三引号"""来方便长篇字符串的书写。而且这种方法还不需要使用转义符。做到了字符串的所见即所得。

```
val text = """
    for (c in "foo")
        print(c)
"""
```

同时，Kotlin 还引入了字符串模板，可以在字符串中直接访问变量和使用表达式：

```
/*
* kotlin字符串模版，可以用$符号拼接变量和表达式
* */
fun testString2() {
    val strings = arrayListOf("abc", "efd", "gfg")
    println("First content is $strings")
    println("First content is ${strings[0]}")
    println("First content is ${if (strings.size > 0) strings[0] else "null"}")
}
```

四、强大的 when 语句

Kotlin 中没有 switch 操作符，而是使用 when 语句来替代。同样的，when 将它的参数和所有的分支条件顺序比较，直到某个分支满足条件。如果其他分支都不满足条件将会进入 else 分支。

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // 注意这个块
        print("x is neither 1 nor 2")
    }
}
```

但功能上 when 语句要强大得多。首先第一点是，我们可以用任意表达式（而不只是常量）作为分支条件，这点 switch 就做不到。如下述代码，前面三个分支条件分别是：1、变量在[1, 10]区间内，2、变量 x 不在[10, 20]区间内，3、变量 x 是一个字符串。这个表达式用 switch 语句基本无法实现，只能用 if else 链来实现。

```
when (x) {
    in 1..10 -> print("x is in the range")
    !in 10..20 -> print("x is outside the range")
    is String -> print("x is a string")
    else -> print("none of the above")
}
```

说起 if else 链，我们可以直接用 when 语句把它给替换掉：

```
when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

五、对象比较

Java 的 == 操作符是比较引用值，但 Kotlin 的 == 操作符是比较内容， === 才是比较引用值。基于这点，我们可以写出逻辑更简洁合理的代码：

```
if (TextUtils.equals(day, today)) {
    viewHolder.mDay.setText("今天");
} else if (TextUtils.equals(day, tomorrom)) {
    viewHolder.mDay.setText("明天");
} else {
    viewHolder.mDay.setText(day);
}
```

上述代码可以直接用 when 语句实现

```
when (day) {
    today -> viewHolder.mDay.setText("今天")
    tomorrow -> viewHolder.mDay.setText("明天")
    else -> viewHolder.mDay.setText(day)
}
```

六、Nullable Receiver

NullableReceiver 我将其翻译成“可空接收者”，要理解接收者这个概念，我们先了解一下 Kotlin 中一个重要特性：扩展。Kotlin 能够扩展一个类的新功能，这个扩展是无痕的，即我们无需继承该类或使用像装饰者的设计模式，同时这个扩展对使用者来说也是透明的，即使用者在使用该类扩展功能时，就像使用这个类自身的功能一样的。

声明一个扩展函数，我们需要用一个接收者类型，也就是被扩展的类型来作为他的前缀，以下述代码为例：

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // “this”对应该列表
    this[index1] = this[index2]
    this[index2] = tmp
}
```

上述代码为 MutableList<Int> 添加一个 swap 函数， 我们可以对任意 MutableList<Int> 调用该函数了：

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // “swap()”内部的“this”得到“l”的值
```

其中 MutableList<Int>就是这个扩展函数的接收者。值得注意的是，Kotlin 允许这个接收者为 null，这样我们可以写出一些在 Java 里面看似不可思议的代码。比如我们要把一个对象转换成字符串，在 Kotlin 中可以直接这么写：

```
var number = null
var str = number.toString()
```

上述代码先定义了一个空指针对象，然后调用 toString 方法，会不会 Crash？其实不会发生 Crash，答案就在“可空接收者”，也就是 Nullable Receiver，我们可以看下这个扩展函数的定义：

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // 空检测之后，“this”会自动转换为非空类型，所以下面的 toString()
    // 解析为 Any 类的成员函数
    return toString()
}
```

扩展函数是可以拿到接收者对象的指针的，即 this 指针。从这个方法的定义我们可以看到，这个方法是对 Any 类进行扩展，而接收者类型后面加了个?号，所以准确来说，是对 Any?类进行扩展。我们看到，扩展函数一开始就对接收者进行判空，若为 null，则直接返回 “null” 字符串。所以无论对于什么对象，调用 toString 方法不会发生 Crash.

七、关键字 object

前面说过，Kotlin 中一切皆为对象，object 在 Kotlin 中是一个关键字，笼统来说是代表“对象”，在不同场景中有不同用法。

第一个是对象表达式，可以直接创建一个继承自某个（或某些）类型的匿名类的对象，而无须先创建这个对象的类。这一点跟 Java 是类似的：

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ……
    }

    override fun mouseEntered(e: MouseEvent) {
        // ……
    }
})
```

第二，对象字面量。这个特性将数字字面量，字符串字面量扩展到一般性对象中了。对应的

场景是如果我们只需要“一个对象而已”，并不需要特殊超类型。典型的场景是在某些地方，比如函数内部，我们需要零碎地使用一些一次性的对象时，非常有用。

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

第三，对象声明。这个特性类似于 Java 中的单例模式，但我们不需要写单例模式的样板代码即可以实现。

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ……
    }

    override fun mouseEntered(e: MouseEvent) {
        // ……
    }
}
```

请注意上述代码是声明了一个对象，而不是类，而我们想要使用这个对象，直接引用其名称即可：

```
DataProviderManager.registerDataProvider(……)
```

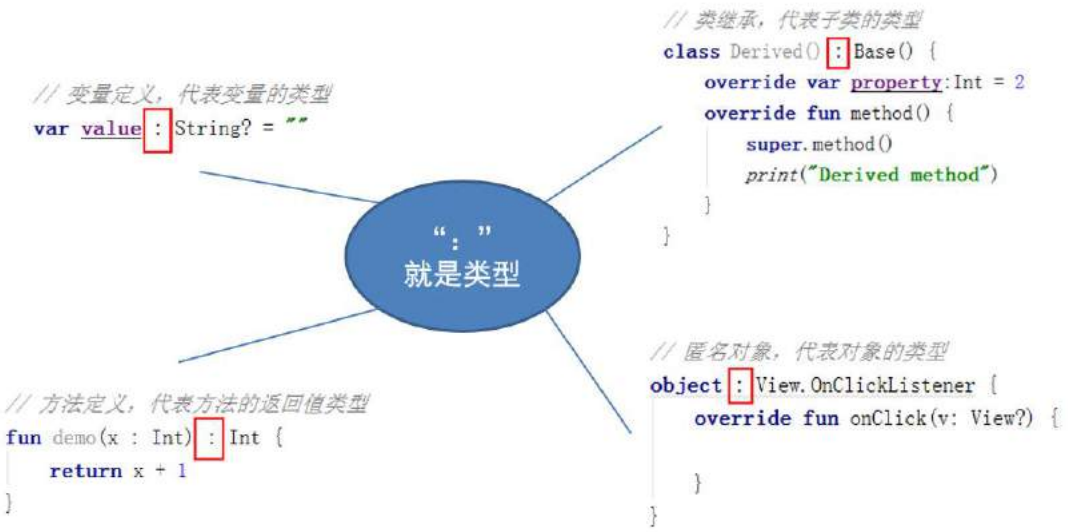
八、有趣的冒号

从语法上来看，Kotlin 大量使用了冒号（:）这一符号，我们可以总结一下，这个冒号在 Kotlin 中究竟代表什么。

考虑下面四种场景：

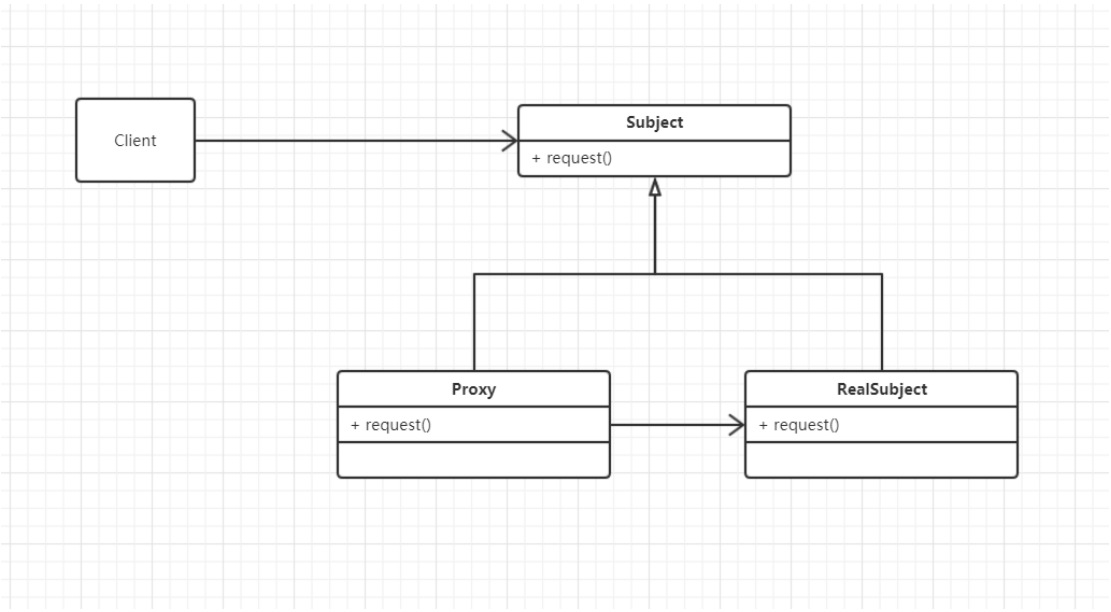
- 在变量定义中，代表变量的类型
- 在类定义中，代表基类的类型
- 在函数定义中，代表函数返回值的类型
- 在匿名对象中，代表对象的类型

笼统来说，Kotlin 的设计者应该就是想用冒号来笼统表示类型这一概念。



九、可观察属性

可观察属性，本质就是观察者模式，在 Java 中也可以实现这个设计模式，但 Kotlin 实现观察者模式不需要样板代码。在谈 Kotlin 的可观察属性前，先看下 Kotlin 里面的委托。同样的，委托也是一种设计模式，它的结构如下图所示：



Kotlin 在语言级别支持它，不需要任何样板代码。Kotlin 可以使用 by 关键字把子类的所有公有成员都委托给指定对象来实现基类的接口：

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b
```

上述代码中，Base 是一个接口，BaseImpl 是它的一个实现类，通过 by b 语句就可以把 Derived 类中的所有公有成员全部委托给 b 对象来实现。我们在创建 Derived 类时，在构造器中直接传入一个 BaseImpl 的实例，那么调用 Derived 的方法等同于调用 BaseImpl 的实例的方法，访问 Derived 的属性也等同于访问 BaseImpl 的实例的属性。

回到可观察属性这个概念，Kotlin 通过 Delegates.observable()实现可观察属性：

```
var name: String by Delegates.observable("wang", {
    kProperty, oldName, newName ->
    println("kProperty: ${kProperty.name} | oldName:$oldName | newName:$newName")
})


fun main(args: Array<String>) {
    println("name: $name") // Log: nam: wang

    name = "zhang" // Log: kProperty: name | oldName:wang | newName:zhang
    name = "li" // Log: kProperty: name | oldName:zhang | newName:li
}
```

上述代码中，name 是一个属性，改变它的值都会自动回调{ kProperty, oldName, newName -> }这个 lambda 表达式。简单来说，我们可以监听 name 这个属性的变化。

可观察属性有什么用处呢？ListView 中有一个经典的 Crash：在数据长度与 Adapter 中的 Cell 的长度不一致时，会报 IllegalStateException 异常。这个异常的根本原因是修改了数据之后，没有调用 notifyDataSetChanged，导致 ListView 没有及时刷新。如果我们把数据做成可观察属性，在观察回调方法中直接刷新 ListView，可以杜绝这个问题。

```
else if (mItemCount != mAdapter.getCount()) {
    throw new IllegalStateException("The content of the adapter has changed but "
            + "ListView did not receive a notification. Make sure the content of "
            + "your adapter is not modified from a background thread, but only from "
            + "the UI thread. Make sure your adapter calls notifyDataSetChanged() "
            + "when its content changes. [in ListView(" + getId() + ", " + getClass()
            + ") with Adapter(" + mAdapter.getClass() + ")]");
}
```

十、函数类型

Kotlin 中一切皆是对象，函数也不例外。在 Kotlin 中，函数本身也是对象，可以拥有类型并实例化。Kotlin 使用类似 (Int) -> String 的一系列函数类型来处理函数的声明，比如我们常见的点击回调函数：

```
val onClick: (View) -> Unit = { }
```

箭头表示法是右结合的，(Int) -> (Int) -> Unit 等价于(Int) ->((Int) -> Unit)，但不等于 ((Int) -> (Int)) -> Unit。可以通过使用类型别名给函数类型起一个别称：

```
typealias ClickHandler = (View) -> Unit
```

函数对象最大的作用是可以轻易地实现回调，而不需要像 Java 那样通过代理类才可以做到。我们以 ScrollView 滑动的回调为例，看一下使用 Java 编写一份 Callback 需要花费多大成本。对于主调方，即 MyScrollView 类而言，首先我们需要一个 Callback 的接口（OnScrollCallback），这个接口里面有一个待实现的 onScroll 方法。然后需要一个属性来保存回调对象。最后在 View 滑动的时候，我们调用这个回调对象的 onScroll 以实现回调。

```
public class MyScrollView {
    public interface OnScrollCallback {
        void onScroll();
    }
    private OnScrollCallback onScrollListener = null;
    public void setOnScrollCallback(OnScrollCallback onScrollListener) {
        this.onScrollListener = onScrollListener;
    }
    private void handleScroll() {
        onScrollListener.onScroll();
    }

}
```

对于被调方，即 MyScrollView 的使用者而言，我们需要一个实现 OnScrollCallback 接口的对象。然后设置成 MyScrollView 的回调对象，才能够实现滑动回调。

```
MyScrollView.OnScrollCallback myListener = new MyScrollView.OnScrollCallback() {
    @Override
    public void onScroll() {

    }
};

scrollView.setOnScrollCallback(myListener);
```

我们只是实现一个简单的回调而已，为什么还要这么复杂呢？本质上是因为 Java 里面函数并不是对象，所以要实现回调，必须要实现一个代理类来包装这个函数，否则我们无法传递这个函数给主调方。

Kotlin 实现回调就是完全不一样的方式了，因为 Kotlin 的函数也是对象，所以我们直接把函

数对象传递给主调方即可。

```
class MyScrollview {
    var onScrollListener: (Int, Int)->Unit = { _, _ -> }
}
```

```
scrollView.onScrollListener = { x, y ->

}
```

看一下上面的代码，就是这么简单！

再介绍下如何将函数类型实例化，有几种常见方式：

一是使用函数字面值的代码块，比如 lambda 表达式 { a, b -> a + b }，或者匿名函数 fun(s: String): Int { return s.toIntOrNull()?: 0 }

二是使用已有声明的可调用引用，包括顶层、局部、成员、扩展函数 ::isOdd String::toInt，或者顶层、成员、扩展属性 List<Int>::size，或者是构造函数 ::Regex

三是使用实现函数类型接口的自定义类的实例

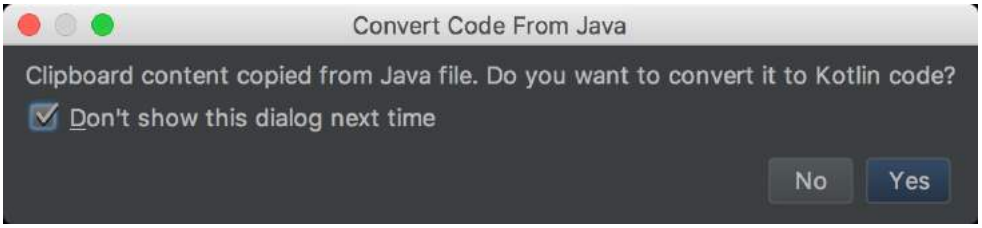
```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

四是编译器推断

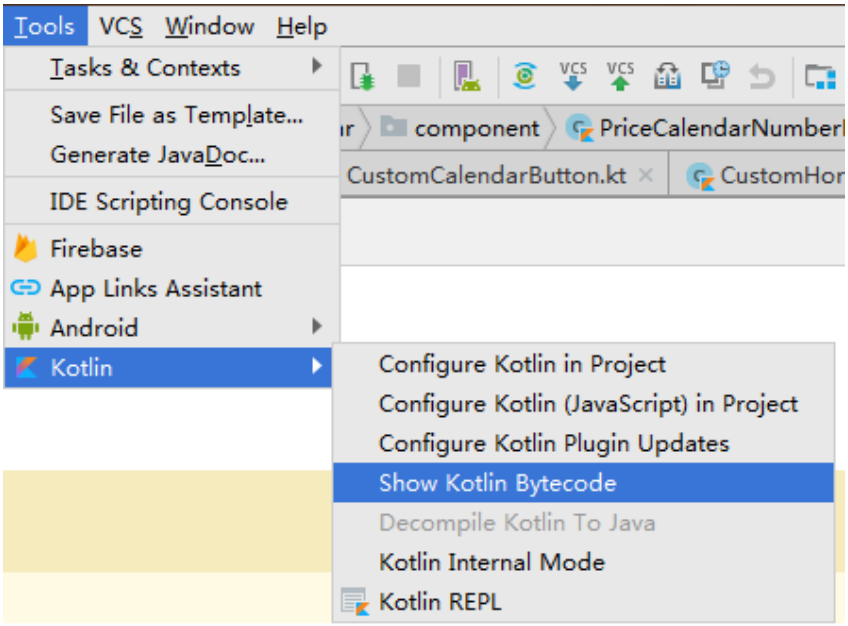
```
val a = { i: Int -> i + 1 } // 推断出的类型是 (Int) -> Int
```

十一、工具

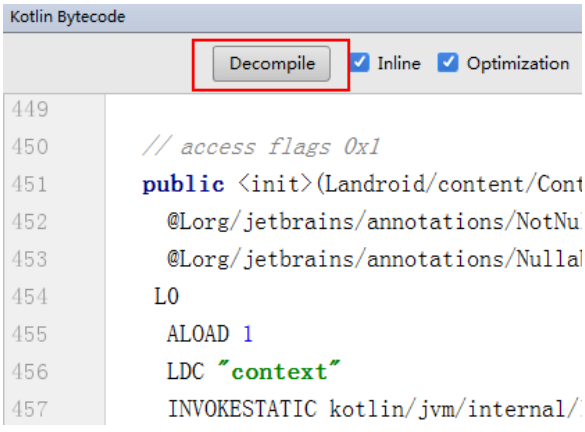
对于初学 Kotlin 的开发者而言，编译器提供了贴心的小工具，甚至可以直接把 Java 代码转换成 Kotlin 代码。直接把 Java 代码拷贝到.kt 文件中，编译器会弹出如下提示：



Kotlin 与 Java 是 100%兼容的，因为它最终会编译成 Java 字节码，我们可以通过 Android Studio 工具看到编译的 bytecode：



我们还可以把编译出来的 Java 字节码反编译成 Java 代码，这样可以窥探 Kotlin 的实现机理：



事实上，Kotlin 优秀的语言特性绝对不止本文提到的这几种，还有很多，比如函数默认参数、扩展属性、懒初始化、局部函数、数据类，等等。欢迎大家在学习的过程中一起交流。

关于 Apple Pay 接入和开发，看这一篇就够了

【作者简介】杨伟，携程金融支付中心支付 Native iOS 组 Leader，目前主要负责中文版携程 APP 和国际版携程 APP 的支付项目功能开发和团队管理。热爱生活，喜欢探索。

Apple Pay 是苹果 2014 年发布的 iOS8 开始支持的一项新功能，2016 年 iOS9 开始支持中国银联卡，Apple Pay 凭借其支付安全，简洁快速的用户体验，开发接入简单，在国内市场占据了一席之地。

携程是最早一批接入 Apple Pay 的中国应用商户，目前携程国际版 APP 中也支持 Apple Pay 支付。国际版的 Apple Pay 功能还在不断扩展中，支持接入新的国家币种和支付通道。

本篇主要从 iOS 前端客户端的角度出发，对 Apple Pay 的应用内接入和开发中遇到的一些问题，做一些总结和回顾，希望给开发人员带来启发和收获。

文章将主要包含三部分：

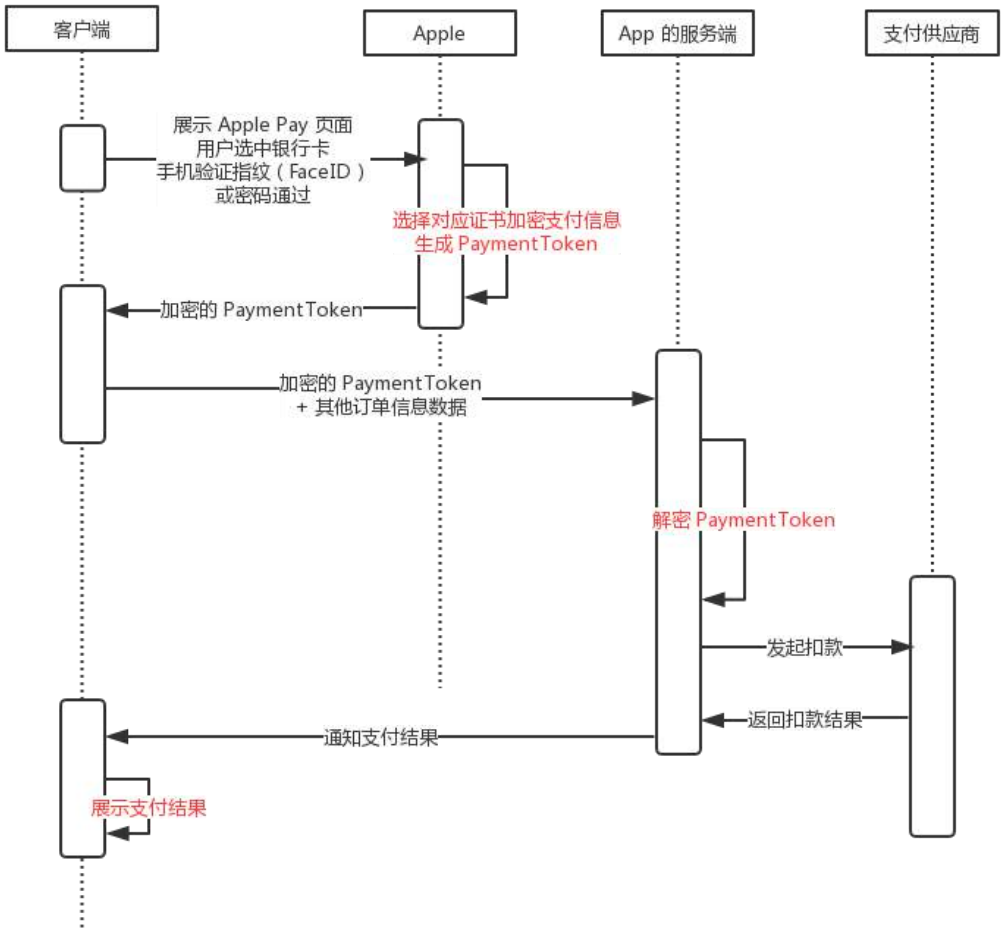
一是 Apple Pay 支付流程梳理，其中客户端、服务端、苹果、支付供应商分别担任什么样的角色；

二是客户端如何支持 Apple Pay：开发证书怎么配置，开发注意事项；

三是国内与国际 Apple Pay 接入的一些区别。

一、Apple Pay 整体流程

下图为目前国内 Apple Pay 支付接入的一个通用的流程（银联 API 模式），仅供参考：



整个流程中如下：

1、客户端通过苹果 API，在 APP 应用内展示 Apple Pay 支付控件。
2、用户在 Apple Pay 的支付控件上进行生物验证（指纹或者人脸识别）或者手机密码验证。
3、苹果在用户验证通过之后，会生成一个用户选中的银行卡相关的 PaymentToken 加密数据，Apple Pay 必须在有网情况下才能进行，苹果需要从开发者网站上使用证书的公钥进行加密，完成后通过 API 回调返回给客户端前端。
4、客户端获取到 PaymentToken 后，给服务端发送扣款请求，等待支付结果。
5、服务端收到客户端上送的 PaymentToken，解密 PaymentToken 取出一些关键字段信息，附带其他订单信息，再与支付供应商（如国内银联）进行通信发起扣款。
6、服务端收到扣款结果后，再返支付结果给手机客户端，最终通知用户支付结果。

二、苹果如何保证安全性

2.1 Apple Pay 绑卡

在 iOS 手机的 Wallet 应用中，用户可以绑定一张真实的银行卡。在绑定过程中，需要输入银行卡的安全字段，还需要进行手机号验证。

绑定成功后，我们可以在手机 Wallet 里面查看这张卡，在卡详情里面会看到一个设备卡号。这个设备卡号就是银行卡的虚拟卡号。这个虚拟卡号也就是苹果下发的 PaymentToken 里面解密后可以拿到的卡号，这个卡号最后要发给支付供应商（如国内银联）做扣款用的。

虚拟卡号不会固定，每次银行卡重新绑定都会重新生成，发生变化。这样就就确保之后所有 Apple Pay 支付过程中网络中不会传输真实的银行卡号，增强安全性。

苹果在绑卡过程中会与不同的发卡机构通信和交互，我们在 Apple Pay 刚进入国内时会看到有的银行卡可以绑定成功，有的不能绑定成功，就是因为有些通道还不稳定。

另外，我们可以在 Apple Pay 的开发者网站上找到特殊的 API ，通过这些 API 某些发卡机构的 APP 就可以直接在应用内向 Wallet 里面绑卡，如某些银行的 APP 就可以做到，这些功能需要专门向苹果提交申请，拿到专门的授权文件，配置到开发工程，才能调通那些 API。相关 API 都可以在网上查到，这里仅供了解，我们一般的 APP 开发用不到。

Apple Pay 的整个平台，苹果在与发卡机构、支付供应商、客户端前端各个流程都有着一整套安全机制。我们可以看出苹果在 Apple Pay 上的投入以及重视程度。 面向 iOS 客户端的 Apple Pay API，也是做到了简单和接入方便，具体使用可以在苹果开发者网站上查询到。

2.2 Apple Pay 数据加密

Apple Pay 中传输中的 PaymentToken 有着一套非常完善的加密安全机制。

国外统一用的是 ECC 加密方式，只有中国用的是 RSA 加密方式，详见官方 PaymentToken 说明。

PaymentToken 数据是 JSON 数据格式，其中包含了苹果加密后的支付信息数据。

PaymentToken 示例

ECC 就是椭圆加密算法，是一种非对称加密方式。

ECC 加密方式，PaymentToken 格式样例：

{

"data":"H22hDxsaFP8JZCuLf6AYu99IB2rLvRbQaZ/NtuQB/vS++ctYSCqPYWUH69eCV59eUdpEcHSOnJPX+95FpFuyRxryb+xzG0EIO0T7fPwDHPyZcA1gixgG/DD10oQgbTf6uWPWkB3z5E3ENsl4QKOTOpladzj4cLKPerlb28s9gjuJtXSylxbnSe5aRFBFwdqsus39hfXNdIlqRCJzYLiGrlFThxoKSv7kVK32u5UYjjAizMkH5tE0fvFbPjGvi3JLrGCozmJDytFWQUVjzba6ORLtd+ui0oe2PXzkVun8w0BNK2tsKh/Jfh4HdK/JD/TBbFLz06SkVa0T7OSjuyltln7NNh3PC+sAltllUyAtjYnY3X76t/DnJ23GVbwZ4t1bUjieZ2loBg+j1w0adcB0dR9UA1Eh1fexrdZW737sd6wp6Edd+PoEW4G5hY0RM/F280q6SD/wWwr80rOkQmE=",

"header": {
"ephemeralPublicKey":"MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEBZITXjr/xw6fsaVQ2bHCJVwgIJt9f9ikgFGtsCKun1Spbei/xi756rt/aTA13ugqNiI2ITk0QvivqV6PhwmrHQ==",
"publicKeyHash":"KjwaMOiXleoSDyLEz5TDPl3uXAVcwZ2f5/P/jGH+gMo=",
"transactionId":
"2e9d75661d1641d188d4e646901fe708836b5a5ac7c705c15915b27e23786488"
},

"signature":"MIAGCSqGSIb3DQEHAqCAMIACAQExDzANBglghkgBZQMEAgEFADCABgkqhkiG9w0BBwEAAKCAMIID4jCCA4igAwIBAgIIJEPyqAad9XcwCgYIKoZIzj0EAwIwejEuMCwGA1UEAwwlQXBwbGUgQXBwbGljYXRpb24gSW50ZWdyYXRpb24gQ0EgLSBHMzEmMCQGA1UECwwdQXBwbGUgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkxEzARBgNVBAoMCkFwcGxlIEluYy4xCzAJBgNVBAYTAlVTMB4XDTE0MDkyNTIyMDYxMVoXDTE5MDkyNDIyMDYxMVowXzElMCMGA1UEAwwcZWNjLXNtcC1icm9rZXItc2lnbl9VQzQtUFJPRDEUSBBcHBsaWNhdGlvbiBJbnRlZ3JhdGlvbiBDQSAtIEczMSYwJAYDVQQLDB1BcHBsZSBDZXJ0aWZpY2F0aW9uIEF1dGhvcml0eTETMBEGA1UECgwKQXBwbGUgSW5jLjELMAkGA1UEBhMCVVMCCCRD8qgGnfV3MA0GCWCGSAFlAwQCAQUAoGkwGAYJKoZIhvcNAQkDMQsGCSqGSIb3DQEHATAcBgkqhkiG9w0BCQUxDxcNMTUwNDA1MDQwOTE2WjAvBgkqhkiG9w0BCQQxIgQgmDfQVEEPlInbB9tQ/AQjni+COkvegoWmfwtpbwe7cuUwCgYIKoZIzj0EAwIERzBFAiB/4NnlUFbNkEqdJ6OGRtA78mZ30539P226OwsNRptycgIhAIzFEYsHH2xuMLVfxEiUmtQxcMdqgW1BgtL/AygaVaAhAAAAAAAA",
"version":"EC_v1"
}

解密 ECC 加密的 data 后看到：

{
"applicationExpirationDate": "190131",
"applicationPrimaryAccountNumber":"370295XXXXX5435",
"currencyCode":"840",
"deviceManufacturerIdentifier": "XXXXXXXXXX",

"paymentData":{ "emvData":"nycBgJ82AgDCnyYIG2vuQydGkMafEAcGhgEDoLABXzQBAJUFgAABAACCAhzAnwMGAAAAAAAAnxoCCECaAxQQBJwBAJ83BLnvab4="
},
"paymentDataType": "EMV",
"transactionAmount": 100
}

国内 RSA 加密方式，得到的 PaymentToken 格式：

{"data" :"juTLgE+0gQPg3RSe5DdJj1/w4amEvJOWSIka+nHNGFmFVd028omhkwMNNqG0exHgT39DAXnKqVNBh4ExGvIEgO7yi97JhDOmwq0nTyvPF/U393wgizrmoe8zX1FpUzI7e2co8PlYCJLkC6uTIuuumsE//503nDhvnz9frzmiMYVhdquf56couB028QBhJiQAupLM+NawVJ41i7e7WJlfyVhYEEn1Qw0TKZKy+Y65PkhAgdwlUGkKUI6r2IpHCc/l4EWvpn1tcVQvVeoG6qJxUdszgL6q

rJBLtaT+/8teg9/jfn0iQwipEgYfTjalgHwXx3nop0dK2ZxzcOzdclfTY3uguM6HBNK6rK3hL2B/LnidAuWE0EWMp5/kqNunDJsSXsUM4+g6zg7ceVjlndZ0YLrQozxRkhPkgHHGtjFxn01PpeGSMMdoAVc8iOgpGKrjx/AAlp2jNY1fZZ1G4cYzW+gsJo5PrmPV7gGxA+M+HG5xdwMVfa9/cxS0qVU2eMw92OB7hdgaIW6cwONETgrd0Y2vF/oHiw7AbRiX/3GxkOJfJ9/5S0P6cxJF4JsvVHkrgit6gZPMCMXFQxDQsK8DImQ9YxzDRIJCb9jxKE0=",
"header" : {
"publicKeyHash": "YcmdtTV209LCGVZV99T0INab3yo0KnudAyBKQPo+5e8=",

"transactionId":"41a6eb8dd8602def5f41463ad314e484289258df71ab17b50204ea4795a3db52",
"wrappedKey" :"l6w4oBmvLF/f/6Gj7idhO5aIFlwZ5qZrqSLxR+mLqsjJqHLf2OUTObn+UOO/laupc+nc6Kuz1ZTQbBMG2w6/KC8F07lZTPnCOcHVxxBP03UQn6gNkNV8DLNNqJ9GoBDzbGy/dWgKBBPNEdgA59jKY+8H/XQzvNuZqiFDM4LTr+O3/FdqGy6PxfFHRwRNV15WoKtsfQ91xPf++Ml4GoTZSdxpc63ewm/8l5Q81AqnZMH03PMRyu8POT92tl10tg8GRmQFCXMgBm7GM+6nz0tebOoLndspqHbe9xtAGmxzseuFCi4A2q8WaqzPgnQ917bvuUnxNHAkXoPTVlUCsXzxXA==",
},
"signature":"MIAGCSqGSIb3DQEHAqCAMIACAQExDzANBglghkgBZQMEAgEFADCABgkqhkiG9w0BBwEAAKCAMIIEtTCCBFugAwIBAgIIEVmL4CjCUF8wCgYIKoZIzj0EAwIwejEuMCwGA1UEAwwlQXBwbGUgQXBwbGIjYXRpb24gSW50ZWdyYXRpb24gQ0EgLSBHMzEmMCQGA1UECwwdQXBwbGUgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkxEzARBgNVBAoMCkFwcGxlIEluYy4xCzAJBgNVBAYTAlVTMB4XDTE0MTIwMzAxMDAyNloXDTE5MTIwMjAxMDAyNlowZzEtMCsGA1UEAwwkZWNjLXNtcC1icm9rZXItc2Inbl9VQzQtUFJPRF9LcnlwQ8XDTE2MTEwMjEyMzg0OFowLwYJKoZIhvcNAQkEMSIEIP+1yZ0K5+XsvQRPOcudpBEIpvl64hJ2pE0PryWNhFSlMA0GCSqGSIb3DQEBAQUABIIBAHuOnvaNlq/1oBGLGyi/+6BmUJLAZjtArsHnpMArgEcmW8qy1oObVeSF7puBjhsnnoWsGnSr+c0/jxYgxKk10RuzLdxdlTbx2DJab+3AzQ4gCatfKuzIzPAdqYaumZLnr/6pJ+CPLaTlWZJZhrPY7GNiqdN2v22tXZQae6fHLwXXqNzjdVOsHNYRRReZA9jbQ46haal/3HpYuQpRhLu5EAkReaYZZKuOhpyxxUeatFtRFwo/vH3yOU5eG07VASCGB33SZqgjJNMZWACIF3FRKK9W8yL50ywEZzDCpCV3/6Jk9yZB1h/jU7qu3iTBXo8jU6lGRcoKjI0DzkiIEQVHB+kAAAAAAAA=",
"version" :"RSA_v1"

中国区的 Apple Pay 加密方式是其实是 RSA + AES，RSA 是一种非对称加密，AES 是一种对称加密。上面的 data 实际由 AES 加密后得到。AES 的密钥藏在 Header 里面的 wrappedKey 中，被 RSA 加密保护着。

国内加密 data 解密后，如下：

{
"applicationPrimaryAccountNumber":"62583300888880215",
"applicationExpirationDate": "270101",
"currencyCode": "156",
"transactionAmount": 0,
"deviceManufacturerIdentifier": "062010011111",

"paymentDataType": "EMV",
"paymentData": {

"emvData":"nyYItis3L6CiQbufNgIACYECAE2DgZCgujJqvZh6gtCOicVyx2tOh1ncXHOQ9bhYMObxz+IHR5a4PD93thtwu7RKyIFb2zab3wkj0oMcra5Cf+J+JbXdk0FxxxxxxxxxxT56HVqNMBp4M/7Uh36IbIsiLkvW0H3rwLVWE/CV4/h0="
}
}

解密后我们都可以看到的 deviceManufacturerIdentifier 就是手机 Wallet 里面绑定银行卡的虚拟卡号，这个是要给到支付供应商发起扣款用到的。

三、支付供应商

支付供应商主要是提供支付扣款渠道的商户，比如中国的银联，国外的 Adyen，可以参考苹果支持的支付供应商。支付供应商主要负责与发卡机构交互，发起扣款。

四、客户端开发 Apple Pay

客户端开发的工作主要包括:

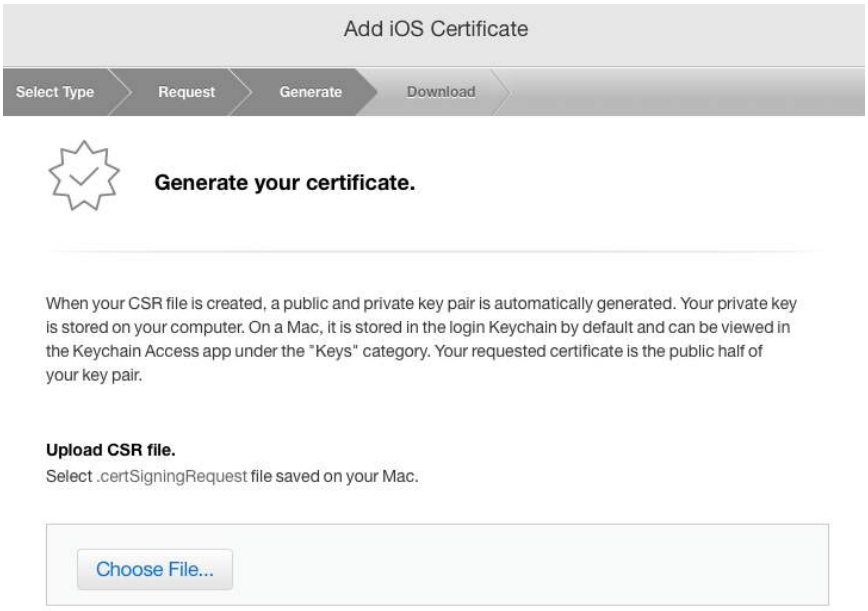
1、证书的准备;
2、APP 工程配置和证书使用;
3、API 调用;

4.1 证书准备

开发者需要在苹果开发者网站上提交 Apple Pay 的证书。

生成 Apple Pay 证书首先需要准备一个 CSR 文件，通过 mac 电脑的钥匙串应用可以快速生成，这个 CSR 文件就是一个文本文件 CertificateSigningRequest.certSigningRequest。

需要将这个 CSR 文件上传到苹果开发者网站上:



做 iOS 开发比较了解，苹果的各种开发证书都是通过这种形式生成。

生成 Apple Pay 证书的 CSR 文件时注意： 中国区生成 CSR 时需要使用 RSA 加密方式。非中国区生成 CSR 时需要使用 ECC 加密方式。

CSR 中包含公钥信息，生成同时，mac 电脑钥匙串应用里会生成对应的私钥。 上传 CSR 到苹果开发者网站生成证书完成后，下载安装到 mac 上，可以在钥匙串应用里面看到私钥，并可以导出私钥，用于解密。

ApplePay 的证书生成流程官方和网上都有流程介绍，这里不再赘述。

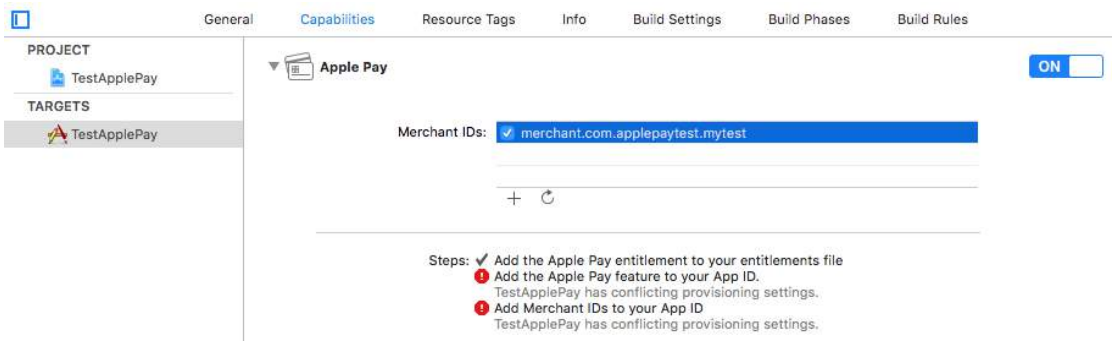
4.2 工程配置和证书使用

在苹果开发者账号网站上可以看到，每一个 Apple Pay 证书都对应和关联一个 MerchantId，每一个 Apple Pay 证书在实际使用过程中，对应一套密钥，对应一套支付扣款通道。苹果下发的 PaymentToken 加密时，就是根据 APP 调用 API 时传入的 MerchantId 找到对应的公钥进行加密处理。

客户端调用 ApplePay API 时，需要指定证书的 MerchantId，建议不要客户端写死在本地，最好由服务端下发，做成灵活可配置的，这样可以进行支付通道切换。

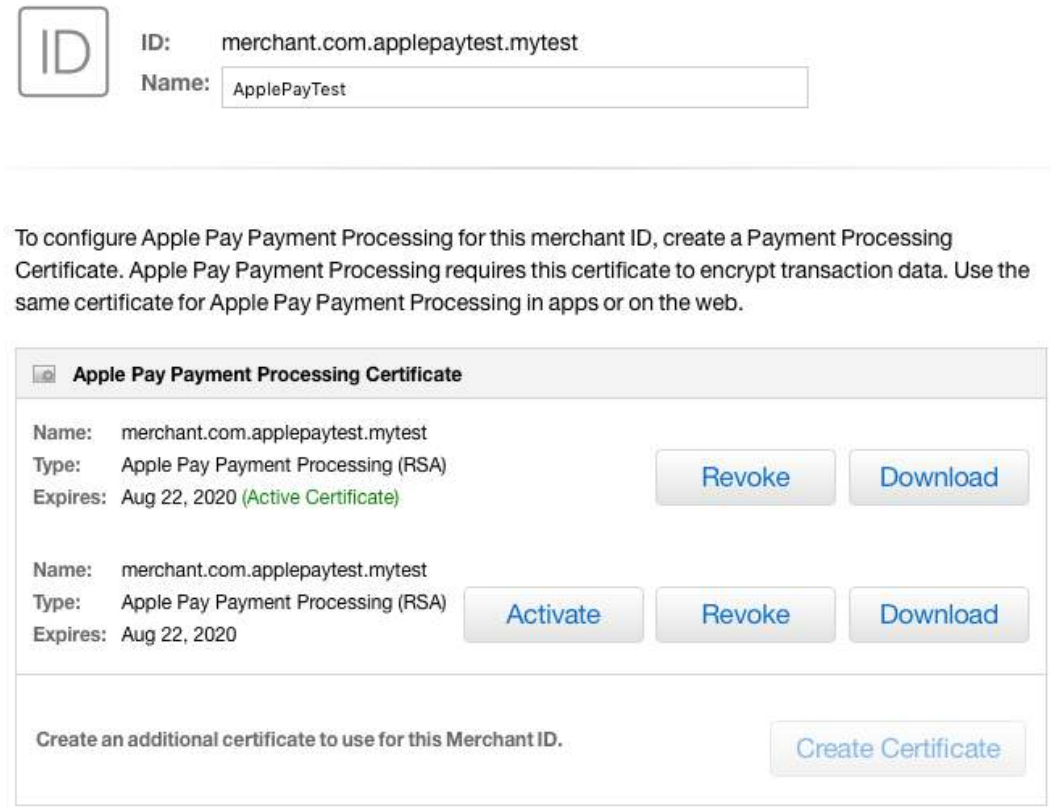
证书生成之后，需要在 Xcode 工程文件的 Targets 的 capability 里，打开 Apple Pay 的开关，并勾选证书对应的 MerchantId，否则打出的 APP 安装包 ipa 无法唤起 Apple Pay 的支付页面。

开启选项的位置，如图所示：



Apple Pay 证书有时效性，一般两年后会过期，需要更换密钥。证书过期的处理，具体操作可以在苹果开发者网站上执行。

在下图所示页面更换证书时，先绑定新的 CSR 生成新证书，Activate 之后，线上就会切换到新证书模式了。



线上验证过证书没有问题，之后就可以 Revoke 旧证书。

证书更新过程中，线上 APP 客户端不需要调整，证书的密钥都配置在服务端，为了不影响用户支付流程，APP 的服务端需要控制下，在证书更换期间，先短暂关闭 Apple Pay 支付方式，待证书新密钥生效后再开启。

五、国内与国际 ApplePay 接入的一些区别和注意事项

携程支付最早接入国内银联的 Apple Pay 渠道时，采用的是银联 SDK 的模式。

这种 SDK 模式就是中国银联以对外提供 SDK 的方式给到 APP 使用 Apple Pay 支付。 具体的 SDK 和使用说明都在可以在中国银联开发网站上下载到。

使用 SDK 的好处就是客户端接入简单，只管调用 SDK 的接口，处理支付结果回调即可，客户端不需要处理各种异常。银联 SDK 调用时需要传递一个 tn 号，这个 tn 号由银联生成，这个 tn 号对应到一笔交易，APP 调用银联 SDK 时，必须传递。

使用 SDK 模式的缺陷是：

1）对 APP 打包的 ipa 安装文件影响比较大，对于对 size 要求比较高的携程 APP 来说，当时银联 SDK 的 size 在支付这边占了不少比例。原因应该是银联的 SDK，内部本身又包含了自己的通信等其他框架。

2）当整个 APP 要求进行 Https，或者 ipv6 等类似特殊支持时，APP 对银联 SDK 的依赖比较重，需要与银联方沟通和确认，确保银联 SDK 支持。

3) SDK 模式下, 证书和密钥都是由银联生成, APP 开发用银联提供的 CSR 文件生成 Apple Pay 证书并绑定，证书和密钥更新麻烦。

4）Apple Pay 的页面展示完全由银联 SDK 控制，当需要增加展示项时，需要向外部寻求银联 SDK 的支持。

后来，携程支付改造接入方式，使用 API 模式，不再使用 SDK 方式接入银联 Apple Pay。这种方式，对于接入商户来说，证书和密钥都由接入商户自己管理，不再依赖支付供应商，客户端和服务端开发更加灵活。

这种方式，iOS 开发者需要自己控制和处理 Apple Pay 的 UI 展示和交互，并应对以下的一些异常：

1) 部分场景中, 用户验证通过后, 正在发送扣款请求时, 用户又点击了取消按钮, 取消 Apple Pay 操作，在这种场景下，支付需要采用一定的方案和策略避免多扣用户的钱。

2）同其他支付方式一样，要考虑如何处理异常情况下的订单重复提交问题。

更多的安全校验：在实际项目中，解密 Apple Pay 的数据后，并没有看到金额，我们尝试直接给到用户随机优惠，提交给银联服务端的支付金额与 APP 中展示给用户看到的金额不一致时，发现无法扣款成功，由此可以断定，苹果与支付供应商之间一定存在着金额校验机制。APP 端是无法越过用户进行任意金额扣款的，必须在 Apple Pay 的页面上展示明确的金额。

在后续的国际 Apple Pay 接入中，携程分别接入了 Adyen 和 SoftBank Payment Service，这两个渠道需要的 Apple Pay 证书都是 ECC 加密方式的，与国内银联不同。

另外 Adyen 接入时，证书由 Adyen 生成，加解密都在 Adyen 处理。而 SoftBank 的证书生成和加解密又都需要携程支付自己完成，所以携程支付在获取的 PaymentToken 之后需要针对不同的支付通道做不同的处理。

在国际版 Apple Pay 实际调用时，APP 服务端根据不同的币种，使用不同的支付通道，Apple Pay 的 API 支持指定证书的 MerchantId，进入支付时，APP 客户端根据服务端下发的 MerchantId 去调用 Apple Pay。通过这种方式就能实现，不同币种，支付使用不同的证书通道。

除了证书通道做成服务下发可配置之外，国际版的 Apple Pay 产品业务比中文版更加复杂，不同的业务场景，需要支持的银行通道又不一样。通过类似的方式，APP 根据服务端控制下发的卡通道，如 Visa、MasterCard 等支付方式限制，来控制 Apple Pay 的支持和展示。

六、总结

在 iOS 开发中，接入 Apple Pay 不仅仅是简单的 API 调用和展示，需要考虑用户的一些行为和交互，任何支付流程都一样，要为用户的体验和财产负责。要充分考虑各种可能存在的异常，如何避免和解决各种异常，需要从整体上做更加全面的设计。

Apple Pay 的正常运作需要客户端、服务端、发卡机构、支付供应商以及苹果各个环节紧密合作。通过参与 Apple Pay 开发以及对 Apple Pay 安全的不断深入了解，会发现苹果确实很注重细节问题，有很多值得借鉴和学习的思路和设计。在支付流程中，如何保证安全同时给用户做到极致的简单体验，苹果确实做到了。

手把手教你 iOS 自定义视频压缩

【作者简介】孙龙波，携程内容信息研发部 Native 开发 leader。目前主要负责携程攻略，行程，视频直播等项目的前端开发和团队管理。

前言

随着抖音，快手等 APP 的迅猛发展，短视频在移动端的地位越来越突出。而视频压缩是视频传输中很关键的一步。

本文会通过一个示例引入视频的一些基本概念并做稍微深入的介绍，最终给出在 iOS 上实现自定义码率和分辨率的视频压缩方案。这篇文章同时也是一个大杂烩，对于很多首次接触视频领域的同学是一个不错的入门文章。

一、传统视频压缩方式的缺陷

1.1 传统压缩的实现

产品需求：不管原视频的清晰度如何，压缩后的视频码率和分辨率是一样的。大家首先想到的应该是 iOS 在 AVFoundation 中已经提供了简单的视频压缩方法，示例代码如下：

```
- (void)compressVideo:(NSURL*) videoUrl witOutputUrl:(NSURL*)outputUrl{
    NSLog(@"The size of original video at %@ is %0.2f M", videoUrl.path, [self
        fileSize:videoUrl]);
    AVAsset* asset = [AVAsset assetWithURL:videoUrl];
    AVAssetExportSession* session = [[AVAssetExportSession alloc]initWithAsset:asset
        presetName:AVAssetExportPreset960x540];
    session.outputURL = outputUrl;
    session.outputFileType = AVFileTypeMPEG4;
    session.shouldOptimizeForNetworkUse = YES;
    [session exportAsynchronouslyWithCompletionHandler:^{
        switch (session.status) {
            case AVAssetExportSessionStatusCompleted:
                NSLog(@"The size of compressed video at %@ is %0.2f M",outputUrl.path,[self
                    fileSize:outputUrl]);
                break;
            case AVAssetExportSessionStatusFailed:
                NSLog(@"compress failed for reason:%@",session.error);
                break;

            default:
                break;
        }
    }];
}
```

正常情况下这段代码不会出现任何问题，但是大家可以用下面的视频做个测试，链接: https://pan.baidu.com/s/1wLVbDFtzROVPo8T1qw6MXA 密码: ij7d。

结果会发现原视频分辨率 1080x608，大小 90M，经过上面的代码压缩后变成了分辨率 960x540，大小 147M！分辨率降低但是文件大小增加了！ 输出的日志如下：

```
2018-07-24 16:52:43.546048+0800 CompressVideoStep1[8692:264200] The size of original video at /Code/test/IMG_0599.MOV
is 95.00 M
2018-07-24 16:55:56.544767+0800 CompressVideoStep1[8692:264436] The size of compressed video at /Code/test/result.mp4
is 147.00 M
```

1.2 压缩参数分析

问题出在哪里？这个视频有什么特殊的地方？我们尝试用 mediainfo 工具查看压缩前后两个视频的详细参数。

原视频参数信息

```
Video
ID                                       : 1
Format                                   : AVC
Format/Info                              : Advanced Video Codec
Format profile                           : High@L3.1
Format settings, CABAC                   : Yes
Format settings, ReFFrames               : 4 frames
Codec ID                                 : avc1
Codec ID/Info                            : Advanced Video Coding
Duration                                 : 3 min 42 s
Bit rate                                 : 3 293 kb/s
Width                                    : 1 080 pixels
Height                                   : 608 pixels
Display aspect ratio                     : 16:9
Frame rate mode                          : Constant
Frame rate                               : 30.000 FPS
Color space                              : YUV
Chroma subsampling                       : 4:2:0
Bit depth                                : 8 bits
Scan type                                : Progressive
Bits/(Pixel*Frame)                       : 0.167
Stream size                              : 87.2 MiB (96%)
Writing library                          : x264 core 148
Encoding settings                        : cabac=1 / ref=2 / deblock=1:0:0 / analyse=0x3:0x113 / me=hex /
Language                                 : English

Audio
ID                                       : 2
Format                                   : AAC
Format/Info                              : Advanced Audio Codec
Format profile                           : LC
Codec ID                                 : 40
Duration                                 : 3 min 42 s
Bit rate mode                            : Constant
Bit rate                                 : 129 kb/s
Channel(s)                               : 2 channels
Channel positions                        : Front: L R
Sampling rate                            : 48.0 kHz
Frame rate                               : 46.875 FPS (1024 SPF)
Compression mode                         : Lossy
Stream size                              : 3.41 MiB (4%)
Language                                 : English
Default                                  : Yes
Alternate group                          : 1
```

压缩后的视频参数:

```
Video
ID                                       : 2
Format                                   : AVC
Format/Info                              : Advanced Video Codec
Format profile                           : Main@L3.1
Format settings, CABAC                   : Yes
Format settings, ReFFrames               : 2 frames
Codec ID                                 : avc1
Codec ID/Info                            : Advanced Video Coding
Duration                                 : 3 min 42 s
Bit rate                                 : 5 177 kb/s
Width                                    : 960 pixels
Height                                   : 540 pixels
Display aspect ratio                     : 16:9
Frame rate mode                          : Constant
Frame rate                               : 30.000 FPS
Color space                              : YUV
Chroma subsampling                       : 4:2:0
Bit depth                                : 8 bits
Scan type                                : Progressive
Bits/(Pixel*Frame)                       : 0.333
Stream size                              : 137 MiB (97%)
Title                                    : Core Media Video
Language                                 : English
Encoded date                             : UTC 2018-07-24 08:52:43
Tagged date                              : UTC 2018-07-24 08:55:56
Color range                              : Limited
Color primaries                          : BT.601 NTSC
Transfer characteristics                 : BT.709
Matrix coefficients                      : BT.601

Audio
ID                                       : 1
Format                                   : AAC
Format/Info                              : Advanced Audio Codec
Format profile                           : LC
Codec ID                                 : 40
Duration                                 : 3 min 42 s
Source duration                          : 3 min 42 s
Bit rate mode                            : Constant
Bit rate                                 : 129 kb/s
Channel(s)                               : 2 channels
Channel positions                        : Front: L R
Sampling rate                            : 48.0 kHz
Frame rate                               : 46.875 FPS (1024 SPF)
Compression mode                         : Lossy
Stream size                              : 3.41 MiB (2%)
Source stream size                       : 3.41 MiB (2%)
Title                                    : Core Media Audio
Language                                 : English
Default                                  : Yes
Alternate group                          : 1
```

1.3 视频参数详解

首先我们要清楚截图中几个关键指标的含义。

1）码率（bit rate）是指数据传输时单位时间传送的数据位数，单位是 bit per second(bps)。简单的说码率=视频文件大小/视频时长。

2）帧率（frame rate）指每秒钟有多少个画面，单位 Frame Per Second 简称 FPS。视频实际是由一组连续的图片组成的，由于人眼有视觉暂留现象，画面帧率高于 16 的时候大脑就会把图片连贯成动画，高于 24 大脑就认为是非常流畅了。所以 24FPS 是视频行业的标准。

但这并不是人眼的极限，帧率继续提高能获取更好更流畅的体验，直到人眼无法区别（极限因人而异，美国空军曾做过一项测试，极限大概是 220FPS，正常人远低于这个数字）。所以游戏行业为了更逼真的效果获取更好的用户体验将标准定为 30FPS

3）分辨率：习惯上我们说的分辨率是指图像的高/宽像素值，严格意义上的分辨率是指单位长度内的有效像素值 ppi（每英寸像素 Pixel per inch）。差别是，图像的高/宽像素值和尺寸无关，但单位长度内的有效像素值 ppi 和尺寸就有关了。比如，同样 Width x Height 的图片，尺寸越大 ppi 越小。

1.4 视频编码标准

解释完上面几个概念大家可以得出比较直观的结论，帧率相同的情况下（压缩前后都是30FPS），分辨率越高码率越大，但是截图的中的参数显示码率大的分辨率低。仔细对比两个视频的参数会发现唯一有区别的是 Format profile，这个参数才是问题的根源。

在介绍这个参数之前需要了解另一个概念，H264 视频编码。所谓视频编码方式就是指通过特定的压缩技术，将某个视频格式的文件转换成另一种视频格式文件的方式。

如果视频不编码会出现什么后果？我们来计算一下，以原视频为例，每秒钟包含了 30 张 1080x600 的图片，那一秒钟的视频大小应该是 1080x600*30/(1024*1024*8) = 2.3 MB. 这个视频有 3 分 42 秒 ，文件大小应该是 2.3*222 = 510MB 远超过 95M。既然编码是必须的那编码标准有哪些呢？

国际上制定视频编解码技术的组织有两个，一个是“国际电联（ITU-T）”，它制定的标准有 H.261、H.263、H.263+等，另一个是“国际标准化组织（ISO）”它制定的标准有 MPEG-1、MPEG-2、MPEG-4 等。

而 H.264 则是由两个组织联合组建的联合视频组（JVT）共同制定的新数字视频编码标准，所以它既是 ITU-T 的 H.264，又是 ISO/IEC 的 MPEG-4 高级视频编码（Advanced Video Coding，AVC）的第 10 部分。所以大家在用不同工具查看同一个视频时有时候会显示 AVC 有时候会显示 H.264 实际是同一种编码方式。

1.5 H264 编码详解

而 H264 最大的优势就是低码率情况下提供高质量的视频图像。怎么做到的？这个问题比较复杂可以新开一篇文章来专门介绍了。有兴趣的大家可以看一下这篇介绍：http://read.pudn.com/downloads147/ebook/635957/%E6%96%B0%E4%B8%80%E4%BB%A3%E8%A7%86%E9%A2%91%E5%8E%8B%E7%BC%A9%E7%BC%96%E7%A0%81%E6%A0%87%E5%87%86H.264.pdf

总的来说编码流程可以分为五部分：帧间和帧内预测（Estimation）、变换（Transform）和反变换、量化（Quantization）和反量化、环路滤波（Loop Filter）、熵编码（Entropy Coding）。而 H264 为了满足不同设备不同场景的需要（比如直播注重实时性，存储注重压缩比）定义了多种编码层次也就是 Profile，官方给 Profile 的定义是：

The standarddefines a set of capabilities, which are referred to as profiles,targeting specific classes of applications. These are declared as a profilecode (profile_idc) and a set of constraints applied in the encoder. This allowsa decoder to recognize the requirements to

decode that specific stream.

Profiles 可以细分为十几种，实际使用的主要有以下四种，

1）BaslineProfile：支持 I/P 帧，只支持无交错（Progressive）和 CAVLC
2）Extended Profile：支持 I/P/B/SP/SI 帧，只支持无交错（Progressive）和 CAVLC
3) MainProfile：提供 I/P/B 帧，支持无交错（Progressive）和交错（Interlaced），也支持 CAVLC 和 CABAC
4）High Profile：在 mainProfile 的基础上增加了 8x8 内部预测、自定义量化、 无损视频编码和更多的 YUV 格式；

大家对里面的术语可能不太理解，简单介绍下。

视频压缩很重要的一个就是帧间预测，也就是视频相邻的几帧有很大的相关性，变化不会太大，所以存在很多冗余信息，压缩要做的就是去除这些冗余信息。帧类型主要有以下几种

1）I 帧表示关键帧，这一帧保留完整的画面数据，解码时只需要本帧数据就可以完成
2）P 帧，前向预测帧，表示的是这一帧跟之前的一个关键帧（或 P 帧）的差别，解码时需要用之前的画面叠加上本帧定义的差别，生成最终画面。
3）B 帧是双向预测帧，也就是 B 帧记录的是本帧与前后帧的差别，要解码 B 帧，不仅要取得之前的画面，还要解码之后的画面，通过前后画面的与本帧数据的叠加取得最终的画面。B 帧压缩率高，但是解码时比较耗费 CPU 。

总结起来就是 Profile 越高，压缩比就越高，但是编码、解码时要求的设备性能也就越高，编码、解码的效率也就越低。

1.6 Profile 与 Level 详解

回到之前的两个视频信息，profile 分别是 main@L3.1，high@L3.1， 现在我们搞清楚了 main 和 high 是 profile，L3.1 是什么？这个是 profile 的码流级别，同样给出官方的定义：

As the term is used in the standard, a "level" is aspecified set of constraints that indicate a degree of required decoderperformance for a profile. For example, a level of support within a profilespecifies the maximum picture resolution, frame rate, and bit rate that adecoder may use. A decoder that conforms to a given level must be able todecode all bitstreams encoded for that level and all lower levels.

简单的说 level 就是对每个 profile 的能力细分。

而 3.1 规定的最高标准如下：

Level	Max. decoding speed		Max. frame size		Max. video bit rate for video coding layer (VCL) kbit/s (Baseline, Extended and Main Profiles)	Examples for high resolution @ highest frame rate Toggle additional details
	Luma samples/s	Macroblocks/s	Luma samples	Macroblocks		
3.1	27,648,000	108,000	921,600	3,600	14,000	352x288@172 352x576@130 640x480@90 720×576@60 1,280×720@30

1.7 iOS 设备对 Profile 和 level 的支持情况

了解完视频的 profile 和 level 之后，大家会有疑问，既然每种 profile 对设备性能的要求不同，苹果的不同机型对各种 profile 支持程度是怎样的？可以参照下面的列表：

iPhone 3GS 和更早的设备支持 Baseline Profile level 3.0 及更低的级别
iPhone 4S 支持 High Profile level 4.1 及更低的级别
iPhone 5C 支持 High Profile level 4.1 及更低的级别
iPhone 5S 支持 High Profile level 4.1 及更低的级别
iPad 1 支持 Main Profile level 3.1 及更低的级别
iPad 2 支持 Main Profile level 3.1 及更低的级别
iPad with Retina display 支持 High Profile level 4.1 及更低的级别
iPad mini 支持 High Profile level 4.1 及更低的级别

二、 自定义视频压缩方案

2.1 实现思路

基本概念都搞清楚了，而我们只需要支持 iphone 5C 以上机型的背景下，要想获得最大的视频压缩率采取的最好办法就是：

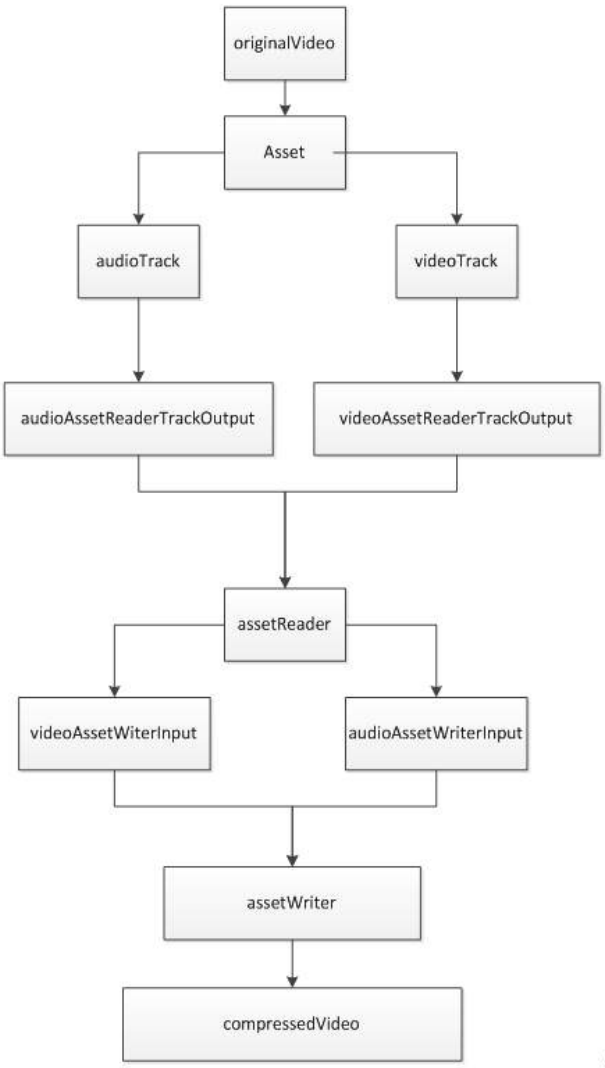
（1）指定 highprofile
（2）降低帧率
（3）适当降低分辨率

最终来获取更低的码率。 问题来了，文章最开始的压缩方式不支持指定 profile，帧率和码率。所以只有通过其他方式来实现。参照了括苹果官网的一个录像时自定义码率的实现

https://developer.apple.com/library/archive/samplecode/RosyWriter/Introduction/Intro.html

在录像时需要将拍摄的每一帧 sampleBuffer（音频或者视频）传给 assetWriter，并制定压缩参数。

而我们要实现的是视频压缩，不是录像，怎么办？思路很简单，先通过 assetReader 取出每一帧 sampleBuffer（音频或视频），然后指定压缩参数后将每一帧传给 assetWriter 最终实现自定义压缩的目的。具体的流程如下：



2.2 代码实现

1）初始化 reader，writer，video/audio track, video/audio input, video/audio output

```
- (void)compressVideoV2:(NSURL*)videoUrl witOutputUrl:(NSURL*)outputUrl{
    NSLog(@"The size of original video at %@ is %@.2f M", videoUrl.path, [self fileSize:videoUrl]);
    AVAsset* asset = [AVAsset assetWithURL:videoUrl];
    AVAssetReader* reader = [AVAssetReader assetReaderWithAsset:asset error:nil];
    AVAssetWriter* writer = [AVAssetWriter assetWriterWithURL:outputUrl fileType:AVFileTypeMPEG4 error:nil];

    //video part
    AVAssetTrack *videoTrack = [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];
    AVAssetReaderTrackOutput* videoOutput = [AVAssetReaderTrackOutput assetReaderTrackOutputWithTrack:videoTrack outputSettings:[self
        videoOutputSettings]];
    AVAssetWriterInput* videoInput = [AVAssetWriterInput assetWriterInputWithMediaType:AVMediaTypeVideo outputSettings:[self
        videoCompressSettings]];
    if ([reader canAddOutput:videoOutput]) {
        [reader addOutput:videoOutput];
    }
    if ([writer canAddInput:videoInput]) {
        [writer addInput:videoInput];
    }

    //audio part
    AVAssetTrack *audioTrack = [[asset tracksWithMediaType:AVMediaTypeAudio] firstObject];
    AVAssetReaderTrackOutput* audioOutput = [AVAssetReaderTrackOutput assetReaderTrackOutputWithTrack:audioTrack outputSettings:[self
        audioOutputSettings]];
    AVAssetWriterInput* audioInput = [AVAssetWriterInput assetWriterInputWithMediaType:AVMediaTypeAudio outputSettings:[self
        audioCompressSettings]];
    if ([reader canAddOutput:audioOutput]) {
        [reader addOutput:audioOutput];
    }
    if ([writer canAddInput:audioInput]) {
        [writer addInput:audioInput];
    }
```

2）指定音视频的压缩码率，profile，帧率等关键参数信息

```
- (NSDictionary*)videoCompressSettings{

    NSDictionary *compressionProperties = @{ AVVideoAverageBitRateKey : @(200 * 8 * 1024),
                                             AVVideoExpectedSourceFrameRateKey:@25,
                                             AVVideoProfileLevelKey : AVVideoProfileLevelH264HighAutoLevel };

    NSDictionary* videoCompressSettings = @{ AVVideoCodecKey                  : AVVideoCodecTypeH264,
                                             AVVideoWidthKey                  : @960,
                                             AVVideoHeightKey                 : @540,
                                             AVVideoCompressionPropertiesKey  : compressionProperties,
                                             AVVideoScalingModeKey : AVVideoScalingModeResizeAspectFill };
    return videoCompressSettings;
}
- (NSDictionary*)audioCompressSettings{
    AudioChannelLayout stereoChannelLayout = { .mChannelLayoutTag = kAudioChannelLayoutTag_Stereo,
        .mChannelBitmap = 0,
        .mNumberChannelDescriptions = 0 };
    NSData *channelLayoutAsData = [NSData dataWithBytes:&stereoChannelLayout length:offsetof(AudioChannelLayout, mChannelDes
    NSDictionary* audioCompressSettings = @{ AVFormatIDKey         : @(kAudioFormatMPEG4AAC),
                                             AVEncoderBitRateKey   : @96000,
                                             AVSampleRateKey       : @44100,
                                             AVChannelLayoutKey    : channelLayoutAsData,
                                             AVNumberOfChannelsKey : @2 };
    return audioCompressSettings;
}
```

3）开始读写

```
[reader startReading];
[writer startWriting];
[writer startSessionAtSourceTime:kCMTimeZero];
```

4）视频逐帧写入

```
dispatch_group_t group = dispatch_group_create();
dispatch_group_enter(group);
[videoInput requestMediaDataWhenReadyOnQueue:video_compression_queue usingBlock:^{
    while ([videoInput isReadyForMoreMediaData]) {
        CMSampleBufferRef sampleBuffer;
        if ([reader status] == AVAssetReaderStatusReading && (sampleBuffer = [videoOutput copyNextSampleBuffer])) {
            BOOL result = [videoInput appendSampleBuffer:sampleBuffer];
            CFRelease(sampleBuffer);
            if (!result) {
                [reader cancelReading];
                break;
            }
        }else{
            [videoInput markAsFinished];
            dispatch_group_leave(group);
            break;

        }
    }

}];
```

5）音频逐帧写入

```
dispatch_group_enter(group);
[audioInput requestMediaDataWhenReadyOnQueue:audio_compression_queue usingBlock:^{
    while ([audioInput isReadyForMoreMediaData]) {
        CMSampleBufferRef sampleBuffer;
        if ([reader status] == AVAssetReaderStatusReading && (sampleBuffer = [audioOutput copyNextSampleBuffer])) {
            BOOL result = [audioInput appendSampleBuffer:sampleBuffer];
            CFRelease(sampleBuffer);
            if (!result) {
                [reader cancelReading];
                break;
            }
        }else{
            [audioInput markAsFinished];
            dispatch_group_leave(group);

            break;
        }
    }
}];
```

6）完成压缩

```
dispatch_group_notify(group, dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    if ([reader status] == AVAssetReaderStatusReading ) {
        [reader cancelReading];
    }
    switch (writer.status) {
        case AVAssetWriterStatusWriting:
        {
            [writer finishWritingWithCompletionHandler:^{
                NSLog(@"The size of compressed video at %@ is %0.2f M",outputUrl.path,[self fileSize:outputUrl]);
            }];
            break;

        }
        default:
            break;
    }
});
```

压缩效果如下：

```
2018-07-30 15:29:21.823266+0800 CompressVideoStep1[29754:3937318] The size of original video at /Code/test/IMG_0599.MOV is 95.00 M
2018-07-30 15:32:17.220417+0800 CompressVideoStep1[29754:3937392] The size of compressed video at /Code/test/result1.mp4  is 47.00 M
```

最终自定义的视频压缩方案有了，其实逐帧写入还可以做添加水印，滤镜等动作，之后可以在后续的文章里进一步介绍。

携程国际 BU 酒店团队的大前端之路

【作者简介】王辛佳，携程 IBU 酒店技术负责人，从事前端工作 10 年。

当今互联网+高速崛起，大前端这个概念已经成为前端技术老生常谈的话题，但去做好“大前端”，并不容易。

一、什么是大前端

学术上目前没有明确的领域划分，在我看来，大致可以分为横向和纵向 2 个维度。

从纵向上来看，可以理解为浏览器端和 Node 服务端的。在过去的几年里，Node.js 的兴起，让前端不再局限于浏览器端，给前端人员一种从前端到后端包打天下之喜悦。

细细分析下来，由于受限于本身特性，无法得到大范围运行。不过 Node 确实为前后端分离指向明确的方向。

从横向上来看，可以理解为泛 UI，比如 PC、移动 H5、ReactNative、Weex、hybrid、小程序等等，凡是由 JavaScript 构成的视图层都可以理解为泛 UI。当然更广义的可以算上 iOS 与 Android，包括 Flash、Silverlight 等等。

无论是横向理解还是纵向认知，随着互联网的发展，大前端势必会越来越受到关注。

二、为什么要做大前端

在这个话题之前，需要明确：大前端为我们带来了什么？最为关键的，我觉得是降低成本。

1、降低沟通成本。产品一个需求，在 H5 上做要讲一遍，在其他平台上做也要讲一遍，而测试也需要在多个平台去测试一遍业务逻辑（这里讲的不是兼容性问题）。

2、降低研发成本。目前前端资源稀缺，成本高，如果我们可以用统一技术去实现“write once run anywhere”，那么就可以最大程度上降低研发成本。

从这个角度，我们做大前端是为了提高工作效率，解决产品、测试痛点，更好为用户服务，以提升用户体验为核心导向。

三、如何去做大前端

3.1 技术选型

前端技术很多，市面上前端框架少说也有几十种了。比较主流的 MVVM 架构就有 Vue、React、Angular 三大体系。除 MVVM 思想以外，Jquery 等也经久不衰。

到底使用哪个最为合适？

不少人会选择最为主流，使用最多的一个。当然，这没错，用的人多了，框架出错情况越少，而且错误的解决方案也越多。但是，往往今天的主流未必明天仍旧主流。就好比曾经 AMD 与 CMD 之争，sea.js 弃用是一回事。

我们怎么考虑呢？当前产线最大痛点是什么，需求是什么，要解决什么样的问题，从这个角度出发，去定型技术框架。

1） 我们需要使用 MVVM 架构，这样不仅提高开发效率，而且能吸引人才加盟。

2）我们需要在 Online 上支持 MVVM 架构，哪怕是在 IE7 上，而且不需要太大成本去支持。

有人会问，现在 PC 端浏览器的占比不是很少了么，为什么还要考虑 PC 端，甚至要考虑低版本浏览器呢？因为我们在做海外产品，不少国家仍旧在使用低版本浏览器。为了不抛弃哪怕百分之一的用户，在技术上我们尽量去满足，去 Support。

3）我们需要 SEO，也就需要考虑到服务端渲染。

4）我们需要 SPA（单页设计）。

5）我们需要 Size 压缩到最小。

按照这个标准，三大主流框架全部淘汰。有的无法满足这个，有的无法满足那个，最后选择来选择去，我们定型于 React。

有的人看到这里，会觉得你不是前言不搭后语么。这里指的 React 并非标准 React，而是 React 语法。

在 PC 端采用 ReactIE，在 H5 采用 Preact，在 iOS 或 Android 中用 ReactNative。同时搭配 Node+Koa2 做 SSR 服务端渲染，满足上述提出的所有要求。

3.2 架构设计

首先，前端职责是什么？

前端需要考虑用户交互行为，浏览器兼容性，代码扩展性，而不是大批量数据运算与转换。对于前端而言，最好能做到“所见即所得”。所以我们的目的是要把前端做轻做薄，把复杂业务逻辑，数据转换逻辑推向后处理。

其次，结构上剥离，让业务层和框架结构更加清晰。

再次，前端监控，最好能把所有错误都统计起来。包括而不限于前端 window.onerror。此外

我们最好能把前端用户轨迹能记录下来，以方便数据分析及排障。

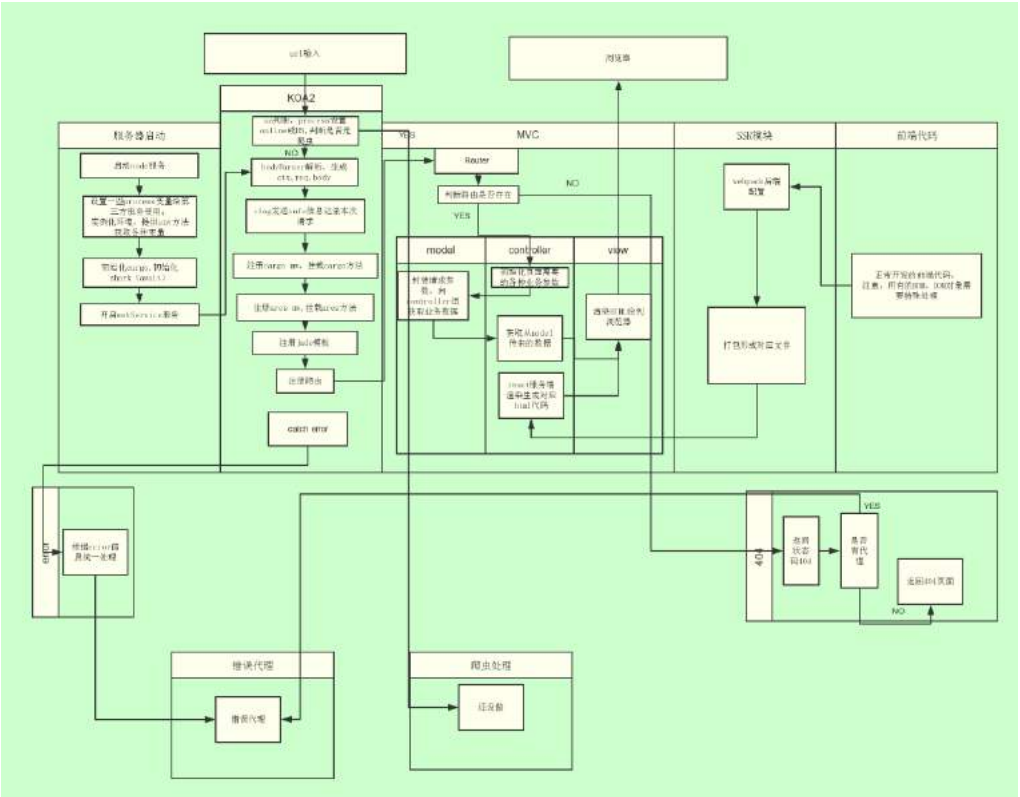
最后，Node 层如何来处理爬虫。

带着以上几个点简单，分享下我们设计的架构图。

1）代码仓库划分

采用 4git 仓库，分别存放 Online、H5 独有代码，Ares DB 存放前端共用业务组件和框架组件，Node+Koa 存放共用 Node 框架。这样好处在于通过 npm 包的方式共享代码，让业务和框架代码分离，职责更加明确。

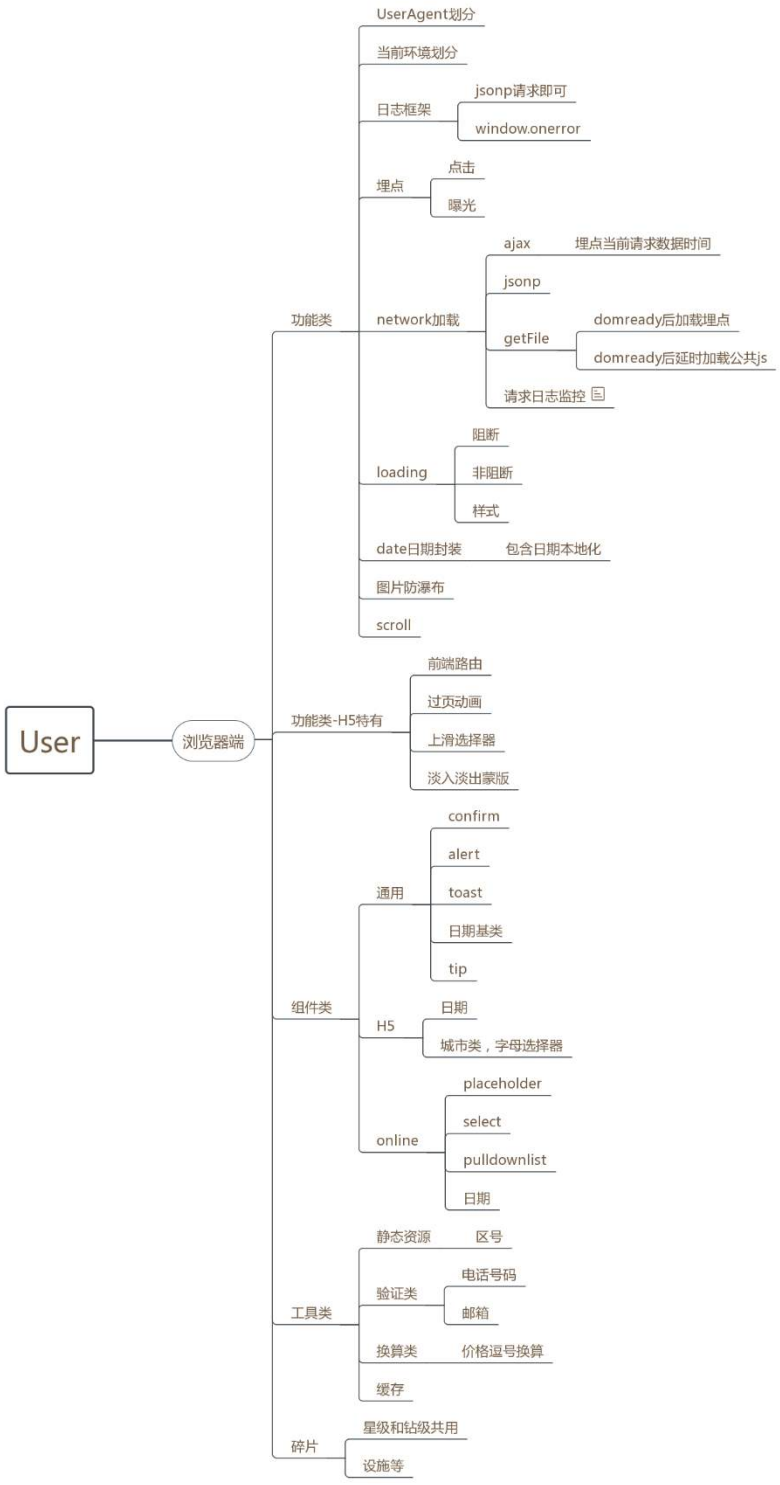
2） Node 架构设计



大致可以分为从服务启动注册、用户访问流程管控、React 服务端渲染 HTML 三大模块。 很

显然，哪怕在 Node 层也不会去做运算逻辑。除了监控日志外，就是做好服务端渲染。这里每一步流程，就不一一展开了。

3）前端组件架构设计

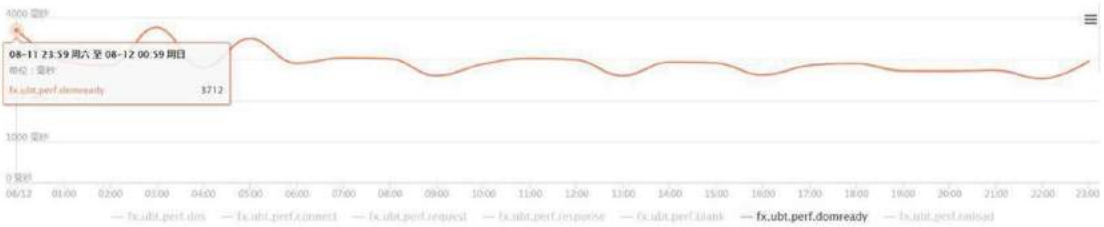


3.3 收益和效果

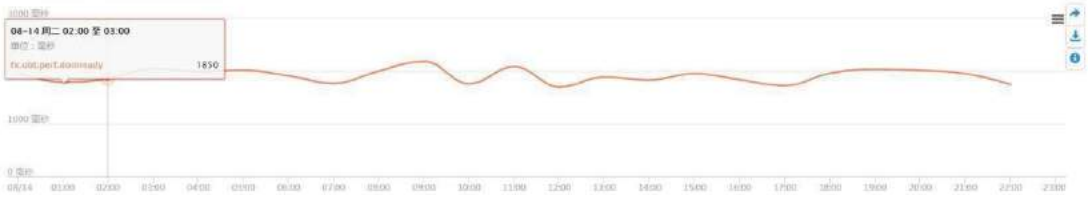
目前我们在 Online 和 H5 上共享组件，带来开发成本减少，改动一处逻辑两端收益效果。

拿已经上线的订单完成页来举例，与之前 Size 和请求数相比，少了将近 50%左右。domready 速度快了 1 倍多。同时采用服务端渲染，也减少白屏时间。

老版本



新版本



四、总结

大前端目前比较火，但还是有很多路需要去走，去探索。我们应该多思考，从痛点出发，来解决问题，而不是人云亦云。这里浅谈一下我们的大前端之路，欢迎各位给出不同意见和见解。

React 模块懒加载初探

【作者简介】常冉冉，携程租车高级前端开发工程师。拥有丰富的 React 技术栈及 Nodejs 工程实践经验，喜欢前端新技术。

2013 年 JSConf 大会上 Facebook 宣布 React 开源，其突破性的创新理念，声明式的代码风格，基于组件嵌套编码理念以及跨平台等优点，获得了越来越多前端工程师的热爱。同时将前端代码工程化提高到一个新的高度。
众所周知，React 的核心理念是模块的组合，但是如果首屏依赖模块过多，或者使用到一些大型模块等，将会显著拖累首屏渲染速度，影响用户体验。

我们尝试通过首次加载模块时仅渲染部分内容，然后在其他模块延迟加载完毕后再渲染剩余部分的方式，提高首屏加载（渲染）速度。

本文将分享一些关于模块延迟加载（懒加载）实现的探索和经验（Reactjs,React-Native 均适用，本文以 Reactjs 示例）。

比如现在我们有一个模块 Hello，demo 代码如下：

```
class Hello extends Component {
      constructor(props){
            super(props)
            this.state = {

            }
      }
   render() {
      return (
         <div className="container">
            <div>{this.props.title}</div>
         </div>
      );
   }
}
```

核心思路：懒加载的核心就是异步加载。可以先展现给用户一个简单内容，或者干脆空白内容。同时在后台异步加载模块，当模块异步加载完毕后，再重新渲染真正的模块。

我们以上述 Hello 模块为例，实现一个简单的异步加载

```
class FakeHello extends Component {
   constructor(props){
```

```
        super(props)
        this.state = {
            moduleLoaded:false
        }
        this._module = null
    }
    componentDidMount(){
        if(!this.state.moduleLoaded){
            setTimeout(()=>{
                this._module= require('../hello').default
                this.setState({moduleLoaded:true})
            },1000)
        }
    }
    render() {
        if(!this.state.moduleLoaded){
            return <div>loading</div>
        }else{
                let M = this._module
            return <M {...this.props} />
        }
    }
}
```

同时将添加一个 button，通过在点击事件回调中修改 state.show 值来控制 Hello 模块是否展示：

```
<btn onClick={this.load} > {this.state.show?'off':'on'}</btn>
{this.state.show && <FakeHello title={"I'm the content"}/>}
```

看下效果：

可以看到第一次点击，Hello 模块显示加载中，1 秒后显示实际模块内容。第二次渲染 Hello 模块时跳过 loading，直接显示模块内容。

实验初步达到了我们的预期。

我们尝试封装一个通用模块 LazyComponent，实现对任何 React 模块的懒加载：

```
let _module

class LazyComponent extends Component{
    constructor(props){
        super(props)
        this.state={
            show:!!_module
        }
    }
    componentDidMount(){
        setTimeout(()=>{
            _module = this.props.render()
            this.setState({show:true})
        },this.props.time)
    }
    render(){
        if(!this.state.show){
            return <div>will appear later</div>
        }else{
            return _module
        }
    }
}
```

LazyComponent 使用例子：

```
{
  this.state.show &&
  <LazyComponent time={1000} render={()=>{
    let M = require('./components/hello').default
    return <M title={this.state.title} />
  }} />
}
```

LazyComponent 有 2 个属性，time 用于控制何时开始加载模块，render 表示加载具体某个模块的方法，同时返回一个基于该模块的 react element 对象。

我们再给 LazyComponet 添加 default 属性，该属性接受任何 React element 类型，为模块未加载时的默认渲染内容。

```
let _module
```

```
class LazyComponent extends Component{
    constructor(props){
        super(props)
        this.state={
            show:!!_module
        }
    }
    componentDidMount(){
        setTimeout(()=>{
            _module = this.props.render()
            this.setState({show:true})
        },this.props.time)
    }
    render(){
        if(!this.state.show){
            if(this.props.default){
                return this.props.default
            }else{
                return <div>will appear later</div>
            }
        }else{
            return _module
        }
    }
}

{
    this.state.show &&
    <LazyComponent time={1000} default={<div>loading</div>} render={()=>{
        let M = require('./components/hello').default
        return <M title={this.state.title} />
    }} />
}
```

看下效果:

看上去完美了。

但是我们发现当父容器中 title 值发生改变时，LazyComponent 包裹的 Hello 模块并没有正确更新。

Why?

我们再来看 LazyComponet render 属性，其返回的是一个包含了 props 值的 element 对象。这样当 Hello 模块首次渲染时，可以正确渲染 title 内容。但是当 LazyComponent 所在的容器 state 改变时，由于 LazyComponet 的 props 未使用 state.title 变量，React 不会重新渲染 LazyComponent 组件，LazyComponent 包裹的 Hello 组件当然也不会重新渲染。

解决办法是将所有 Hello 组件所要依赖的 state 数据通过 LazyComponent 的 props 再传递给 Hello 组件。

```
{
    this.state.show &&
    <LazyComponent time={1000} default={<div>empty</div>} realProps={{title:'hello'}}
load={()=>require('./components/hello').default} />
}

let M

class LazyComponent extends Component{
    constructor(props){
        super(props)
        this.state={
            show:!!M
        }
    }
    componentDidMount(){
        if(!M){

            setTimeout(()=>{
                M = this.props.load()
                this.setState({show:true})
            },this.props.time)
        }
    }
    render(){
        if(!this.state.show){
            if(this.props.default){

                return this.props.default
            }else{
```

```
                return <div>will appear later</div>
            }
        }else{
            return <M {...this.props.realProps} />
        }
    }
}
```

再看下效果:

现在，我们已经实现了一个简单的 LazyComponent 组件。将懒加载组件代码同普通组件比较:

<LazyComponent time={1000} default={<div>loading</div>} realProps={{title:'hello'}} load={()=>require('./components/hello').default} />

<Hello title={"hello"}/>

显而易见，虽然我们实现了懒加载，但是代码明显臃肿了很多，而且限制只能通过 realProps 传递真实 props 参数，给工程师带来记忆负担，可维护性也变差。

那么,能否更优雅的实现懒加载？能否像写普通组件的方式写懒加载组件？或者说通过工具将普通组件转换为懒加载模块?

我们想到了高阶组件(HOC),将传入组件经过包装后返回一个新组件。

于是有了下面的代码:

```
function lazy(loadFun,defaultRender=()=><div>loading</div>,time=17){
    let _module
    return class extends Component{
        constructor(props){
            super(props)
            this.state={
                show:!!_module
            }
        }
        componentDidMount(){
```

```
                let that = this
                if(!_module){
                    setTimeout(()=>{
                        _module=loadFun()
                        that.setState({show:true})
                    },time)
                }
            }
            render(){
                if(!this.state.show){
                    return defaultRender()
                }else{
                    let M = _module
                    return <M {...this.props} />
                }
            }
        }
}
```

使用方法：

```
const LazyHello = lazy(()=>require('./components/hello').default,()=><Loading />,1000)

<LazyHello title={"I'm the content"}/>
```

总结

通过本次实践，我们得到了两种实现模块懒加载的解决方案：

A、使用 LazyComponent 组件，load 属性传入需要懒加载模块的加载方法；
B、使用高阶函数 lazy 包装原始组件，返回支持懒加载特性的新组件。

大数据篇

ALLUXIO 在携程大数据平台中的应用与实践

[作者简介]郭建华，携程技术中心软件研发工程师，2016 年加入携程，在大数据平台部门从事基础框架的研究与运维，主要负责 HDFS、Alluxio 等离线平台的研发运维工作。

进入大数据时代，实时作业有着越来越重要的地位，并且部分实时和离线作业存在数据共享。实践中使用统一的资源调度平台能够减少运维工作，但同时也会带来一些问题。

本文将介绍携程大数据平台是如何引入 Alluxio 来解决 HDFS 停机维护影响实时作业的问题，并在保证实时作业不中断的同时，减少对 HDFSNameNode 的压力，以及加快部分 Spark SQL 作业的处理效率。

一、背景

携程作为中国旅游业的龙头，早在 2003 年就在美国上市，发展到现在，携程对外提供酒店、机票、火车票、度假、旅游等线上服务产品，每天线上有上亿的访问量，与此同时，海量的用户和访问也产生了大量的数据，这些数据包括日志以及访问记录，这些数据都需要落地到大数据平台上。

为了对这些数据进行分析，我们在大数据方面有着大量的离线和实时作业。主集群已突破千台的规模，有着超过 50PB 的数据量，每日的增量大概在 400TB。巨大的数据量且每天的作业数达到了 30 万，给存储和计算带来了很大的挑战。

HDFS NameNode 在存储大量数据的同时，文件数和 block 数给单点的 NameNode 处理能力带来了压力。因为数据存在共享，所以只能使用一套 HDFS 来存储数据，实时落地的数据也必须写入 HDFS。

为了缓解和优化 NameNode 的压力，我们会对 NameNode 进行源码优化，并进行停机维护。而 HDFS 的停机会导致大量的需要数据落地到 HDFS 的 Spark Streaming 作业出错，对那些实时性要求比较高的作业，比如实时推荐系统，这种影响是需要极力避免的。

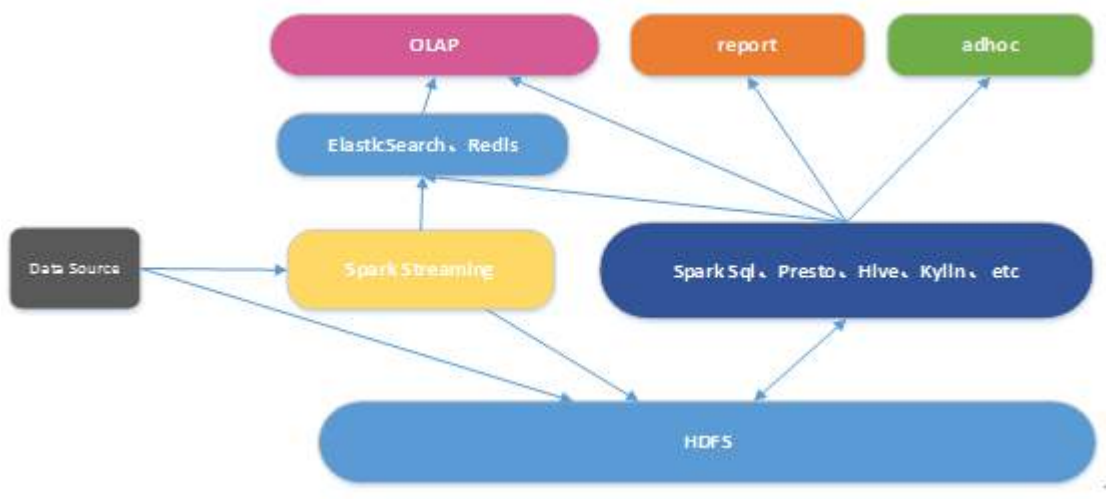


图 1　携程大数据平台架构图

图 1 为携程大数据平台架构图，DataSource 为不同的数据源，有日志信息、订单信息。它们通过携程自己研发的中间件或者直接落地到 HDFS 或者被 Spark Streaming 消费之后再落地到 HDFS。

Streaming 计算的结果有的直接落地到 Redis 或者 ElasticSearch 等快速查询平台，而有些 Streaming 计算的实时数据需要和历史数据进行再计算，则需要落地到 HDFS 上。

按照业务层不同的需求，我们提供了不同的执行引擎来对 HDFS 上的数据进行计算。执行快速的 Spark SQL 和 Kylin 主要用在 OLAP 上，Hive 和 Spark SQL 同时用在 ETL 作业上，Presto 主要用在 adhoc 查询。

上述架构能够满足大部分的工作要求，但是随着集群规模的增大，业务作业的增多，集群面临了很大的挑战，其中也存在着诸多不足。

上述架构存在以下几个问题:

1) SparkStreaming 依赖于 HDFS，当 HDFS 进行停机维护的时候，将会导致大量的 Streaming 作业出错。

2）SparkStreaming 在不进行小文件合并的情况下会生成大量的小文件，假设 Streaming 的 batch 时间为 10s，那么使用 Append 方式落地到 HDFS 的文件数在一天能达到 8640 个文件，如果用户没有进行 Repartition 来进行合并文件，那么文件数将会达到 Partition*8640。我们具有接近 400 个 Streaming 作业，每天落地的文件数量达到了 500 万，而目前我们集群的元数据已经达到了 6.4 亿，虽然每天会有合并小文件的作业进行文件合并，但太大的文件增量给 NameNode 造成了极大的压力。

3）SparkStreaming 长时间占用上千 VCores 会对高峰时期的 ETL 作业产生影响，同时，在高峰期如果 Streaming 出错，作业重试可能会出现长时间分配不到资源的情况。

为了解决上述问题，我们为 SparkStreaming 搭建了一套独立的 Hadoop 集群，包括独立的 HDFS、Yarn 等组件。

虽然上述问题得到了很好的解决，但这个方案仍然会带来一些问题。如果主集群想访问实时集群中的数据时，需要用户事先将数据 DistCp 到主集群，然后再进行数据分析。架构如图 2 所示。除了 DistCp 能够跨集群传输数据之外，我们第一个想到的就是 Alluxio。

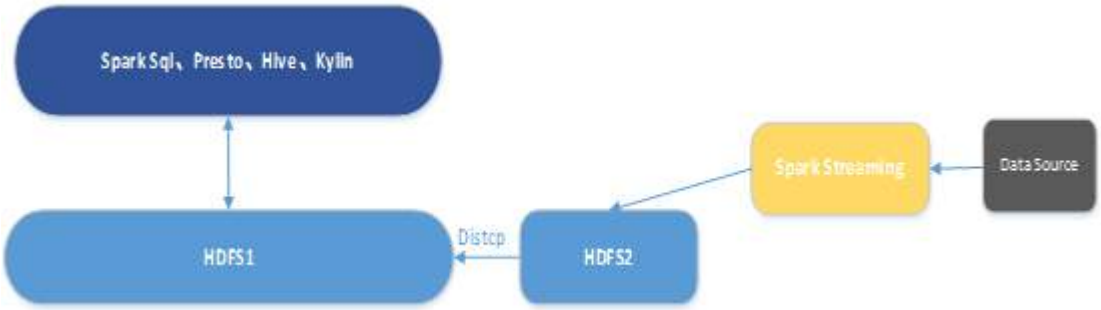


图 2　独立集群架构: HDFS2 独立与主集群 HDFS1 以提供资源隔离

Alluxio 作为全球第一个基于内存级别的文件系统，具有高效的读写性能，同时能够提供统一的 API 来访问不同的存储系统。它架构在传统分布式文件系统和分布式计算框架之间，为上层计算框架提供了内存级别的数据读写服务。

如图 3 所示，Alluxio 可以支持目前几乎所有的主流分布式存储系统，可以通过简单配置或者 Mount 的形式将 HDFS、S3 等挂载到 Alluxio 的一个路径下。这样我们就可以统一的通过 Alluxio 提供的 Schema 来访问不同存储系统的数据，极大的方便了客户端程序开发。

同时，对于存储在云端的数据或者计算与存储分离的场景，可以通过将热点数据 load 到 Alluxio，然后再使用计算引擎进行计算，这极大的提高了计算的效率，而且减少了每次计算需要从远程拉去数据的所导致的网络 IO。而我们利用 Alluxio 统一入口的特性，挂载了两个 HDFS 集群，从而实现了从 Alluxio 一个入口读取两个集群的功能，而具体访问哪个底层集群，完全由 Alluxio 帮我们实现了。

图 3　Alluxio 统一存储和抽象

二、解决方案

为了解决数据跨集群共享的问题，我们引入了国际知名并且开源的 Alluxio。部署的 Alluxio1.4 具有良好的稳定性和高效性，在引入 Alluxio 之后，架构如图 4 所示。

图 4　改进后架构图

从图 4 可以看到，Spark Streaming 数据直接落地到 Alluxio，Alluxio 通过将 HDFS1 和 HDFS2 分别挂载到两个路径下。简单的通过命令：
$ alluxiofs mount /path/on/alluxio hdfs://namenode:port/path/on/hdfs 就能分别挂载这两个 HDFS 集群。

HDFS-2 集群专门负责存储流计算的数据。数据收集到 Kafka 之后，Spark Streaming 对其进行消费，计算后的数据直接写挂载了 HDFS-2 集群的路径。

Alluxio 很友好的为 Client 提供了三种写策略，分别是：MUST_CACHE、CACHE_THROUGH、THROUGH，这三种策略分别是只写 Alluxio，同步到 HDFS，只写 HDFS。这里可以根据数据的重要性，采用不同的策略来写 Alluxio，重要的数据需要同步到 HDFS，允许数据丢失的可以采用只写 Alluxio 策略。

采用上述策略方案之后，我们发现 Alluxio 在一定程度上减少了 NameNode 的压力。部分热点数据并且多次使用的数据，我们会通过定时作业将该部分数据加载到 Alluxio，一方面加快了计算引擎加载数据的速度，另外一方面减少了对 NameNode 的数据访问请求数。

此外，Alluxio 自身实现了一个叫做 TTL（Time To Live）的功能，只要对一个路径设置了 TTL，Alluxio 内部会对这部分数据进行检测，当前时间减去路径的创建时间大于 TTL 数值的路径会触发 TTL 功能。

考虑到实用性，Alluxio 为我们提供了 Free 和 Delete 两种 Action。Delete 会将底层文件一同删除，Free 只删 Alluxio 而不删底层文件系统。为了减少 Alluxio 内存压力，我们要求写到 Alluxio 中的数据必须设置一个 TTL，这样 Alluxio 会自动将过期数据删除（通过设置 Free Action 策略，可以删除 Alluxio 而不删除 HDFS）。对于从 Alluxio 内存中加载数据的 Spark Sql 作业，我们拿取了线上的作业和从 HDFS 上读数据进行了对比，普遍提高了 30%的执行效率。

三、后记

从调研 Alluxio 到落地上线 Alluxio，整个过程下来，我们碰到过一系列的问题，针对这些问题以及业务需求，开发了一系列的功能并回馈了 Alluxio 社区。

1、Alluxio 在写 HDFS 的时候，需要使用 HDFS 的 Root 账号权限，对于带 Kerberos 的 HDFS 集群，会出现无权限写。为了解决这个问题，我们为 Alluxio 单独建了一个 Hadoop 账号，所有落地到 HDFS 的数据通过该账号来写。

2、1.4 版本的 Alluxio 不支持以文件夹的形式进行 TTL 的设置，我们进行了功能的完善并贡献给社区(出现在 1.5 以及后续版本中)。

3、1.4 版本不支持 TTL 使用 Free 策略来删除数据，我们对该功能进行了完善并贡献给社区(出现在 1.5 以及后续版本中)。

4、1.4 版本底层文件发生修改，对于 Alluxio 来说是不感知的，而通过 Alluxio 读取的数据可能出现不准确（1.7 版本得到了彻底解决），我们开发了一个 shell 命令 checkConsistency 和 repairConsistency 来解决这个问题。

携程机票实时数据处理实践及应用

[作者简介]张振华，携程旅行网机票研发部资深软件工程师，目前主要负责携程机票大数据基础平台的建设、运维、迭代，以及基于此的实时和非实时应用解决方案研发。

携程机票实时数据种类繁多，体量可观，主要包括携程机票用户访问、搜索、下单等行为日志数据；各种服务调用与被调用产生的请求响应数据；机票服务从外部系统(如 GDS)获取的机票产品及实时状态数据等等。这些实时数据可以精确反映用户与系统交互时每个服务模块的状态，完整刻画用户浏览操作轨迹，对生产问题排查、异常侦测、用户行为分析等方面至关重要。

回到数据本身，当我们处理数据的时候，往往会遇到两类数据，一类是已经存在的有界(bounded)数据(比如 hdfs 的某个数据文件)，一类是持续不断生成的无界(unbounded)数据(如某个活跃的 Kafka topic)，对应着这两类数据的处理也被分为批处理(batch processing)和流式处理(stream processing)。

批处理针对有界数据，处理完所有数据后释放计算资源；而流式处理则需要持续地处理不断产生的数据。随着大数据社区的发展，各类优秀成熟的计算框架不断涌现。众所周知，当前比较成熟的批处理计算框架主要包括 Hadoop、Spark 等，实时处理计算框架主要包括 Storm、Spark Streaming、Flink 等。

从大数据技术发展历史来看，海量历史数据的处理需求提出要早于实时流式数据，因此批处理计算框架出现和趋于成熟得更早。然而，互联网时代的来临，高吞吐的实时数据处理也成了在线平台的刚需，这也极大促进了实时计算框架的发展。

一、流数据处理框架

流数据处理框架按照其实现的方式，也可以分为逐条处理和微批量(micro-batching)处理两种(如图 1 所示)，Storm 和 Flink 属于前者，Spark Streaming 属于后者。

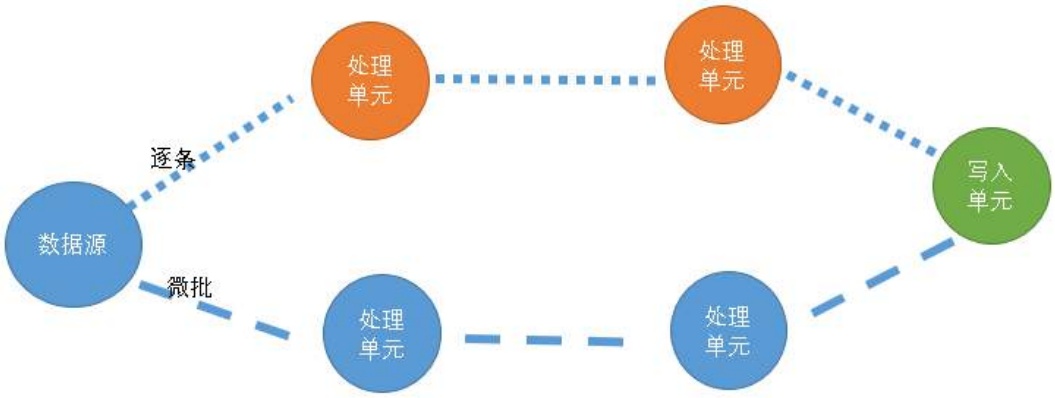


图 1 实时处理实现两种方式

介绍三个计算框架特性的文章很多，本文不再过多赘述。
在 Storm 里，通过定义 Spout 数据来源和后续一系列的 Bolt 处理流程组成完整拓扑 Topology，从而实现整套处理逻辑。利用 Topology 定义的 Storm 流程是无状态的，无法实现 exactly once 处理容错语义，如果应用场景中需求严格的一次处理，如统计一个小时内 IOS 用户的 PV，可以用 Storm Trident API，确保每条消息只会被处理一次。

Flink 和 Spark 则既可以支持批处理，也可以支持流处理，但两者对数据处理的设计似乎正好相反，Flink 会把所有数据处理当成流数据来处理，即使处理静态的有界数据；Spark 则将所有数据处理转化为批处理，即使在处理流式数据，也会将流数据切分成微批来进行计算。Flink 这种流处理优先的方式叫做 Kappa 架构，而 Spark 这种批处理优先的方式被称作为 Lambda 架构。

在大多数公开的性能测试报告中，Flink 吞吐、延时方面的性能指标最优，Spark Streaming 受限于 micro-batching 处理的机制，时延方面最好只能达到秒级，无法满足严苛的实时需求，Storm 的时延能达到亚秒，但吞吐指标稍显不足。在数据处理一致性方面，Flink 通过 state snapshot 优雅地实现了 exactly once 保证，Spark streaming 则利用 WAL(Write Ahead Log)和 RDD 本身特性保证了 exactly once，storm 由于其容错采用的 ack 机制只能保证 at least once，而其 Trident 则采用封装 tuple 到 batch 的方式，并保存元数据和中间状态，从而实现了 exactly once 的语义。

那么如何选择合适的流处理框架呢？主要看应用场景，如果应用需要亚秒级的时延，Storm 和 Flink 是不错的选择，特别是 Flink；如果对时延的要求不是特别高，可以选择 Spark Streaming，毕竟 Spark 目前社区的活跃度、产品成熟度、生态圈丰富度、SQL 支持这块都优于其余两者。

二、Kafka

在实时计算的很多场景中，消息队列扮演着绝对重要的角色，是解耦生产和 BI、复用生产数据的解决方案。Kafka 作为消息队列中最流行的代表之一，在各大互联网企业、数据巨头公司广泛使用。

Kafka 出身 LinkedIn，是一个分布式的发布/订阅系统。集群由多个 Broker 节点组成，通过 Zookeeper 维护元数据信息、选举 Partition 的 Leader、记录消费端状态。在 Broker 节点上，每个 Topic 的 Partition 对应着一个文件系统目录，并以 topic_name-partition_index 命名，最终数据会被写入到 Partition 对应的目录，并以 index 文件和 segment 文件成对出现的方式存储。众所周知，即使数据写入到普通的 SATA 硬盘上，Kafka 依然具有非常高的吞吐性能。究其本质，还是因为 Kafka 的顺序读写、系统级的 PageCache 利用，绕过用户态 buffer 数据拷贝的 sendfile 优化。

Kafka 作为生产环境和数据分析环境的数据枢纽环节，其稳定性至关重要。表 1 为一个可以作为生产环境 Kafka 的配置。

操作系统	CentOS 7.1
Open files	100000
文件系统	Ext4
JVM	1.8
GC	G1
Broker节点配置	12核/48G内存/4*4T硬盘/万兆网卡
JVM堆大小	4G
Zookeeper节点配置	12核/48G内存/2*4T硬盘

表 1 生产环境推荐 Kafka 配置

携程机票从 2015 年开始使用 Kafka，发生过多次大小故障，踩过的坑也不少，下面罗列些琐碎的经验。

1、Partition 的默认数目设置成与 Broker 节点数一致，这样在默认配置的情况下，不至于因为某些体量超大的 Topic 造成 Broker 节点硬盘负载和网络负载倾斜

2、做手工维护强制让某些体量大的 Topic 瘦身时(如 retention.ms 和 retention.bytes 变小)，要注意节奏，尽量不要同时修改多个，造成集群 IO 尖刺

3、某些写入端确实需要写入大报文数据并且超过默认设置(1MB)时，需要在 Topic 配置中增大 max.message.bytes，并且在写入端 Producer 侧增大 max.request.size

4、Producer 默认开启压缩，compression.codec=gzip/snappy，这样可以有效降低副本同步的网络 IO 开销，当然同样会带来消费解压引起的集群 CPU 开销

5、扩容新节点需要超过一台，并且应该尽快将已有 Topic 的 Partition 数目调整与扩容后节点数目一致

6、可以考虑独立出一个 SOA 写入服务供生产环境各服务使用，一方面减少生产环境对大数据组件的依赖，一方面可以让后续的版本升级，集群迁移等操作对调用端透明

7、启动 Kafka 进程时打开 JMX 参数，在 KafkaManager 里可以轻松观察各个节点的写入 qps

8、定期清理 dead Consumer，为 zookeeper 减负

9、设置 auto.leader.rebalance.enable=true，让 partitionLeader 的分布更均衡

10、num.io.threads 配置成 min(2*disk_num , cpu_core+1)，以达到较高的 IO 处理速率

三、携程机票实时数据处理架构实践及应用

携程机票实时数据主要来源于用户与机票系统交互产生的业务数据流和日志流，业务数据最

终会被写入关系型数据库 SQLServer 和 MySQL 中，日志数据则通过 SOA 服务写入消息队列 Kafka 中，目前机票 BI 实时应用使用的数据源主要来自于 Kafka 的日志消息数据。

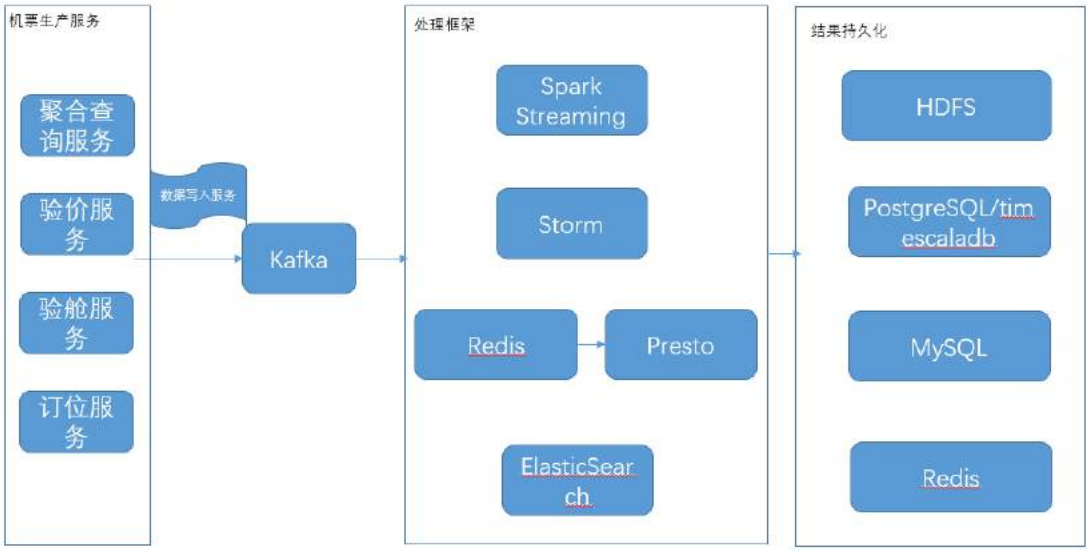


图 2　携程机票实时数据处理架构

图 2 为携程机票当前采用的实时数据处理技术栈。在实时处理框架选择上，我们采用了 Storm 和 Spark Streaming，主要针对不同时延需求的业务场景。没有采用 Flink，一方面由于公司暂时没有部署计划，另一方面没有高迭代并且对时延要求很高的应用场景。

Spark Streaming 目前主要用来实时解析机票查询日志，用户搜索呈现在机票 App/Online 界面上的航班价格列表在查询服务返回时其实是一个经过序列化压缩的报文，我们将 Kafka Direct Stream 接收到数据流 DStream，并经过计算处理，将大报文解析成航班价格列表，并存储至 Hive，进而支持机票价格监控、舱位实时分析、价格实时优劣势展现、各引擎优劣势实时分析等多个应用，每天解析出来的航班价格数据量大约 60 亿。另外 Spark Streaming 也被用来支持 ABT 实验的指标统计，从而近实时获悉新版接入流量后的表现。

Storm 主要用来实时识别舱位虚占，从下单服务推送到 Kafka 的明细下单日志，经过过滤、提取相应字段后，进行统计和虚占识别规则匹配，从而实时标记虚占用户，并将虚占识别结果写入 redis 供下单服务使用。

除了经典的 Spark Streaming 和 Storm 流计算框架外，为了支持机票数据监控系统灵活动态配置取数 SQL 的需求，我们采用了 Redis+Presto 这种方案，以分钟粒度的时间戳为 key，将 kafka 对应该时间戳的数据以 Json 列表的格式跟 key 作关联，并利用 Presto Redis Connector 通过 SQL 的方式聚合计算该 key 对应列表数据，并将聚合结果写入 DB 供监控系统前端调用，实时监控机票各项指标。

我们利用 Logstash 将 Kafka 内的各服务日志 topic 实时 ETL 至 ElasticSearch，并利用实时更新的二级索引(存储于 redis，记录唯一标识与 ElasticSearchindex 的对应关系)和并发查询支持唯一标识所有相关日志数据的串联，在展示界面里完整回溯用户在携程系统里的行为。

另外，相关前端埋点数据和后台访问日志被实时同步至 timescaledb 的超表中，通过灵活可配的 SQL 执行对应的反爬识别规则，并适用机器学习模型将爬虫 IP 尽快甄别出来，进而实施反爬策略。

四、总结

随着大数据开源社区的蓬勃发展，越来越多的计算框架涌现并趋于成熟，应用框架各自有各自的特性，只有最适合自身应用项目的框架才是最好的选择。当然，随时关注社区发展，调研了解新技术的特点也是非常有必要的。

参考：

1）Kafka 官方文档
2）Flink 官方文档
3）Storm 官方文档
4）三种流框架的对比分析
https://bigdata.163yun.com/product/article/5
5）大数据处理引擎 Spark 与 Flink 大比拼
https://www.douban.com/group/topic/95561683/

携程 Presto 技术演进之路

[作者简介]张巍，携程技术中心大数据资深研发工程师。2017 年加入携程，在大数据平台部门从事基础框架的研发和运维，目前主要负责 Presto，Kylin，StructedStreaming 等大数据组建的运维，优化，设计及调研工作。对资源调度，OLAP 引擎，存储引擎等大数据模块有浓厚的兴趣，对 hdfs，yarn，presto，kylin，carbondata 等大数据组建有相关优化和改造经验。

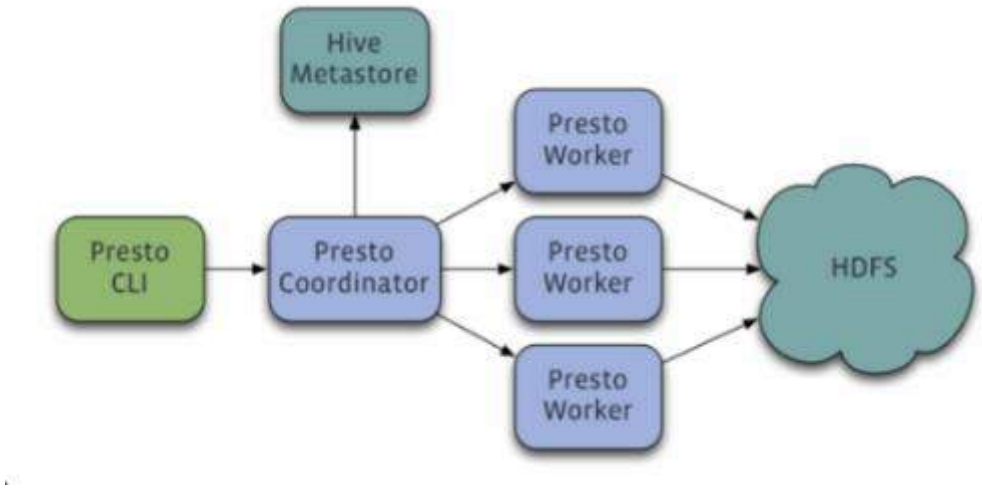
一、背景介绍

携程作为中国在线旅游的龙头，提供酒店，机票，度假等服务，这些服务的背后是基于各个部门每天对海量数据的分析。

随着业务的不断增长，用户体验的不断提升，每个部门对数据处理的响应时间要求越来越短，对各个部门报表的响应速度要求越来越快，对跨部门的数据交互和查询也越来越紧密，所以需要一个统一的快速查询引擎，且这个查询引擎需要从 GB 到 PB 以上的海量数据集中获取有价值的信息。

我们在技术选型上对比了 Presto，Spark，Impala 等 MPP 数据库。综合考量框架本身性能和社区活跃程度，最终选择了 Presto。

Presto 是 Facebook 开源的 MPP 数据库，先简单了解下架构图：



它是一个 Master-Slave 的架构，由下面三部分组成：

1）一个 Coordinator 节点

2）一个 Discovery Server 节点

3）多个 Worker 节点

Coordinator 负责解析 SQL 语句，生成执行计划，分发执行任务给 Worker 节点执行。Discovery Server 通常内嵌于 Coordinator 节点中。

Worker 节点负责实际执行查询任务以及负责与 HDFS 交互读取数据。

Worker 节点启动后向 DiscoveryServer 服务注册，Coordinator 从 DiscoveryServer 获得可以正常工作的 Worker 节点。如果配置了 HiveConnector，需要配置一个 Hive MetaStore 服务为 Presto 提供 Hive 元信息。

二、携程 Presto 使用的困境

首先来看一下我们 2018 年前遇到的一些问题。

携程在 2014 年探索使用 Presto 去满足用户快速即席查询和报表系统的需求。在 2017 年末，离线团队接手携程 Presto 后，发现当时的 Presto 本身存在一系列问题，那个时候 Presto 的版本是 0.159，也相对较低。

2.1 稳定性差

当时用户反馈最多的是，Presto 又 OOM 了。只能采取重启 Presto 恢复服务，实际上对于用户来说，系统挂掉是最恶劣的一种体验了。

Presto 严格的分区类型检查和表类型检查，导致大量用户在 Presto 上发起的查询以失败告终，对于那些使用老分区重新刷数据的用户简直就是灾难。

一些大数据量的查询经常占用着计算资源，有时运行了 2、3 个小时，累计生成上百万个 split，导致其他小的查询，响应速度受到严重影响。

2.2 认证不规范

很早以前，携程在 Presto 中内部嵌入一个 Mysql 的驱动，通过在 Mysql 表中存放用户账号和密码访问 Presto 的权限认证。实际上和大数据团队整体使用 Kerberos 的策略格格不入。

2.3 性能浪费

所有的 join 查询默认都是使用 Broadcast join，用户必须指定 join 模式才能做到 Broadcast join 和 Map join 的切换。

数据传输过程中并没有做压缩，从而带来网络资源的极大浪费。

2.4 没有监控

Presto 自身没有监控分析系统，只能通过 Presto 自身提供的短时监控页面看到最近几分钟的用户查询记录，对分析和追踪历史错误查询带来很大的不便。

无法知道用户的查询量和用户的查询习惯，从而无法反馈给上游用户有效的信息，以帮助应用层开发人员更合理的使用 Presto 引擎。

三、携程 Presto 引擎上所做的改进

为了提供稳定可靠的 Presto 服务，我们在性能，安全，资源管控，兼容性，监控方面都做了一些改动，以下列出一些主要的改进点。

3.1 性能方面

- 根据 Hive statistic 信息，在执行查询之前分析 hive 扫描的数据，决定 join 查询是否采用 Broadcast join 还是 map join。
- Presto Page 在多节点网络传输中开启压缩，减少 Network IO 的损耗，提高分布计算的性能。
- 通过优化 Datanode 的存储方式，减少 presto 扫描 Datanode 时磁盘 IO 带来的性能影响。
- Presto 自身参数方面的优化。

3.2 安全方面

- 启用 Presto Kerberos 模式，用户只能通过 https 安全协议访问 Presto。
- 实现 Hive Metastore Kerberos Impersonating 功能。
- 集成携程任务调度系统(宙斯)的授权规则。
- 实现 Presto 客户端 Kerberos cache 模式，简化 Kerberos 访问参数，同时减少和 KDC 交互。

3.3 资源管控方面

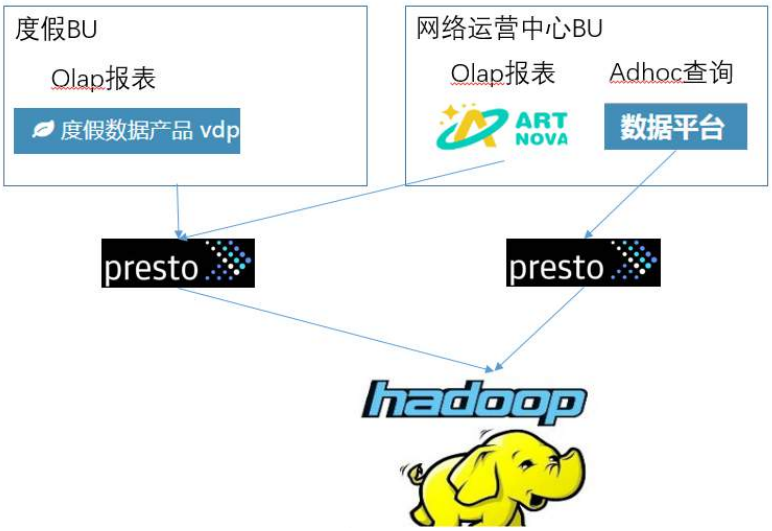
- 控制分区表最大查询分区数量限制。
- 控制单个查询生成 split 数量上限，防止计算资源被恶意消耗。
- 自动发现并杀死长时间运行的查询。

3.4 兼容性方面

- 修复对 Avro 格式文件读取时丢失字段的情况。
- 兼容通过 Hive 创建 view，在 Presto 上可以对 Hive view 做查询。(考虑到 Presto 和 Hive 语法的兼容性，目前能支持一些简单的 view)。
- 去除 Presto 对于表字段类型和分区字段类型需要严格匹配的检测。
- 修复 Alter table drop column xxx 时出现 ConcurrentModification 问题。

四、携程 Presto 升级之路

升级之初 Presto 的使用场景如图。



4.1 第一阶段，版本升级

对于版本选择，我们关心的几个问题：1）是否很好地解决各类内存泄漏的问题；2）对于查询的性能是否有一定提升。

综上考虑，决定使用 0.190 版本的 Presto 作为目标的升级版本。

通过这个版本的升级，结合对 Presto 的一部分改进，解决了几个主要问题：

- Presto 内存泄漏问题。
- Presto 读取 Avro 文件格式存在字段遗漏的问题。
- Presto 语法上无法支持整数类型相乘。

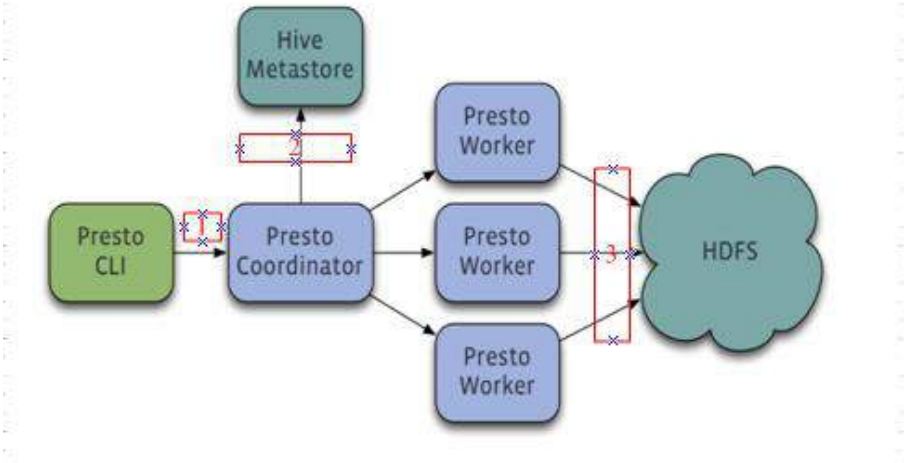
4.2 第二阶段，权限和性能优化

在第二个版本中，我们主要解决了以下问题：

- Kerberos 替换 Mysql
- Join 模式的自动感知和切换
- 限流（拒绝返回 100 万以上数据量的查询）

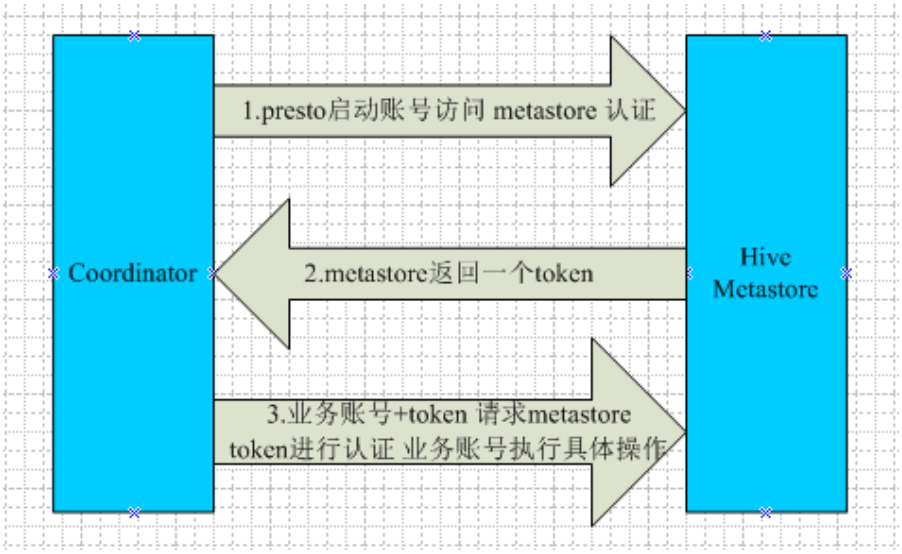
认证机制

这里简单介绍下 Kerberos 权限替换过程中的一些细节。Presto 的认证流程：



Presto 涉及到认证和权限的部分如上面红色框标注的 3 个部分，Coordinator, HDFS 和 Hive Metastore 这三块。Coordinator 和 HDFS 这两块是比较完善的，重点讲一下 Hive Metastore。

在 Kerberos 模式下，所有 SQL 都是用 Presto 的启动账号访问 Hive Metastore，比如使用 Hive 账号启动 Presto，不论是 flt 账户还是 htl 账户提交 SQL，最终到 Hive Metastore 层面都是 Hive 账号，这样权限太大，存在安全风险。我们增加了 Presto Hive MetastoreImpresonating 机制，这样 htl 在访问 Hive Metastore 时使用的是通过 Hive 账号伪装的 htl 账户。



新的问题又来了，在认证过程中需要获取 Hive 的 Token， 可是 Token 反复的获取都需要一次 Metastore 的交互，这样会给 Metastore 带来压力。于是我们对 Token 做了一个缓存，在其 Token 有效期内缓存在 Presto 内存中。

4.3 第三阶段，资源管控和监控平台

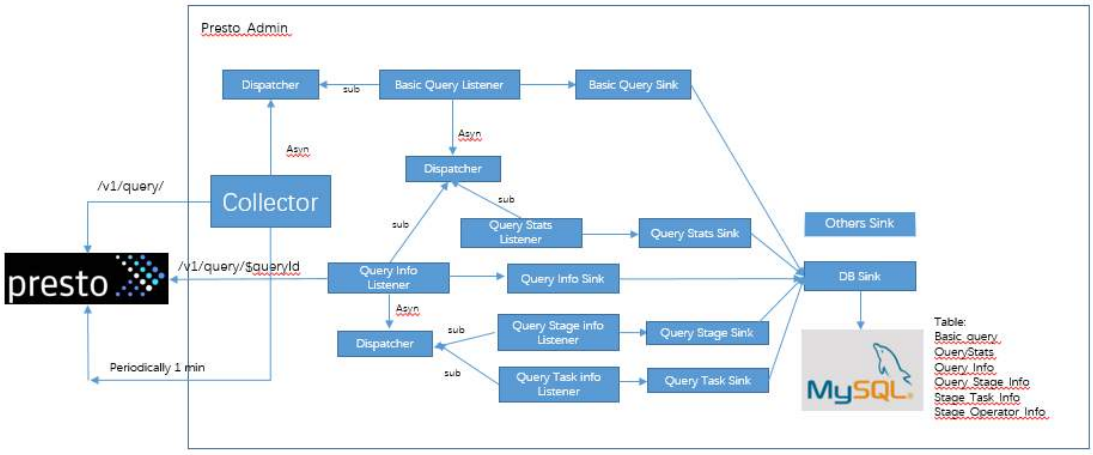
在第三个版本中，我们解决了以下问题：

- 拦截大量生成 split 的查询 SQL

- Presto 监控平台初步搭建
- 限制最大访问的分区数量

4.3.1 数据采集

流程图



程序每一分钟从 Presto Coordinator 采集数据， 分发到多个监听器，同时写入 Mysql 表。

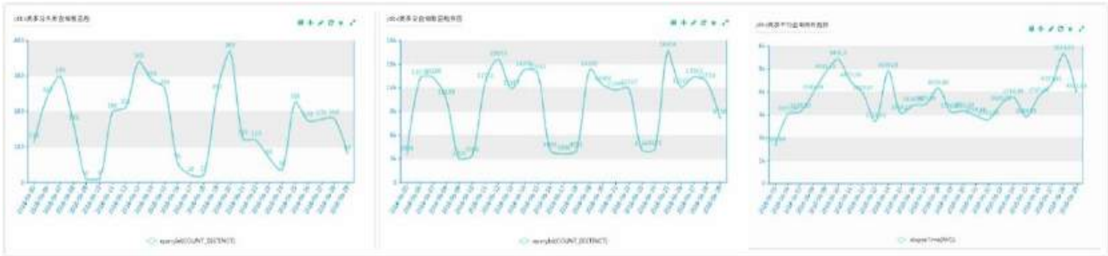
当前入库 5 张监控表。

Basic query：查询基本信息（状态，内存使用，总时间消耗，错误信息等）
Query stats：查询性能信息（每一步的时间消耗，数据输入输出量信息等）
Query info：查询客户端参数信息（发起客户的基本信息，参数信息等）
Query stage info：每个查询中所有 stage 的信息（输入输出量信息，内存使用情况，调用核的数量等）
Query task info：每个 stage 中所有 task 的信息（输入输出信息， 内存信息，调用核数等）

4.3.2 实时健康状况报告

基于以上采集的数据，我们会实时生成 presto 集群的健康报表以及历史运行趋势。这些数据可以用于：

- 集群容量的评估
- 集群健康状态的检测



4.3.3 问题追踪

除了健康报表之外，对于查询错误和性能问题，我们提供了详细的历史数据， 运维人员可以通过报表反应出的异常状况做进一步的排查。

通过报表能够发现某个用户查询时出现了外部异常

```
mysql> select queryId,errorType from basic_query where user='hotelcoupon' and date='2018-06-27' and errorType='EXTERNAL';
+-----------------------------+-----------+
| queryId                     | errorType |
+-----------------------------+-----------+
| 20180626_221631_11788_7adjh | EXTERNAL  |
+-----------------------------+-----------+
1 row in set (0.39 sec)
```

```
mysql> select b.stacktrace from basic_query a, QUERY_INFO b where a.queryId=b.queryId and a.user='hotelcoupon' and a.date='2018-06-27' and a.errorType='EXTERNAL' ;
```

```
| com.facebook.presto.spi.PrestoException : Failed connecting to Hive metastore: [SH02SVR5249.hadoop.sh2.ctripcorp.com:9083, SH02SVR5250.hadoop.sh2.ctripcorp.com:9083, SH02SVR5248.hadoop.sh2.ctripcorp.com:9083]
com.facebook.presto.hive.metastore.thrift.StaticHiveCluster.createMetastoreClient(StaticHiveCluster.java:99)
com.facebook.presto.hive.metastore.thrift.ThriftHiveMetastore.lambda$getTable$5(ThriftHiveMetastore.java:207)
com.facebook.presto.hive.metastore.thrift.HiveMetastoreApiStats.lambda$wrap$0(HiveMetastoreApiStats.java:42)
com.facebook.presto.hive.RetryDriver.run(RetryDriver.java:137)
com.facebook.presto.hive.metastore.thrift.ThriftHiveMetastore.getTable(ThriftHiveMetastore.java:206)
com.facebook.presto.hive.metastore.thrift.BridgingHiveMetastore.getTable(BridgingHiveMetastore.java:79)
com.facebook.presto.hive.metastore.CachingHiveMetastore.loadTable(CachingHiveMetastore.java:264)
```

通过查看异常堆栈，发现查询是由于 hive metastore 出现短暂重启引起的查询失败。

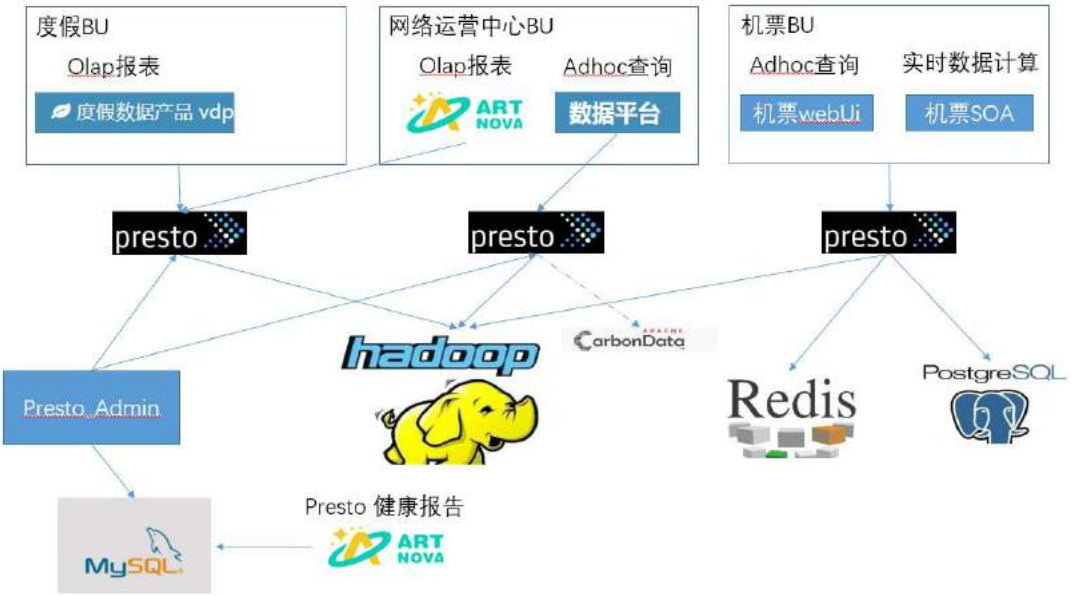
4.3.4 其他

在 Presto 升级改进的同时，我们也调研了 Presto on Carbondata 的使用场景。

当时 Carbondata 使用的是 1.3.0 版本。在此基础上：

- 修复了一系列 Presto on carbon data 的功能和性能的问题
- 对比了 Presto on carbon data 和 Presto on hive，得出结论：Presto on Carbon 整体性能和 Presto on orc 的整体性能相当。同样的数据量 Carbon snappy 的存储是 ORC zlib 的六倍，就目前存储吃紧的情况下，不适合使用。
- 目前仅在线上提供 Carbondata 连接器，暂未投入业务使用。

当前 Presto 的架构为：



五、携程 Presto 未来升级方向

5.1 架构完善和技术改进

启用 Presto 资源队列，规划通过 AppName 划分每个资源队列的最大查询并发数，避免某个应用大量的查询并发同时被 Presto 执行，从而影响其他的 App。

实时告警平台，对于错误的查询，Presto 能够实时的发送异常查询到告警平台，帮助运维人员快速响应和发现错误以便及时处理。

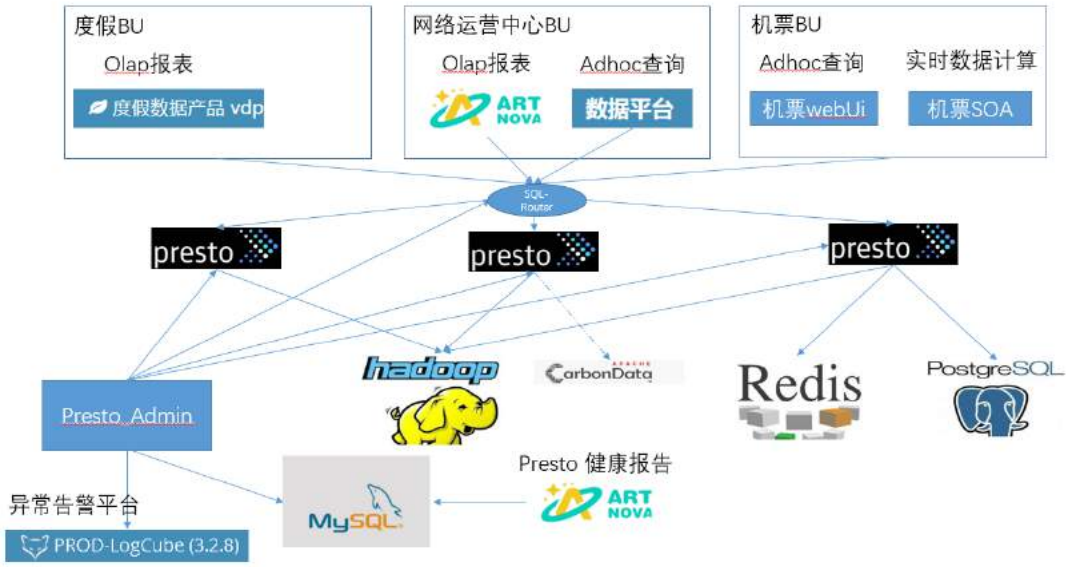
统一的查询引擎，统一的查询引擎可以在 presto，kylin，hive spark-sql 之间匹配最优的查询引擎，做语法转换后路由过去。

5.2 业务方向

未来携程内部 OLAP 报表系统(Art Nova) 会更大范围的采用携程 Presto 作为用户自定义报表的底层查询引擎，以用来提高报表的响应速度和用户体验。

部分业务部门的 vdp 也计划调研，实时数据计算采用 Presto 作为其默认的查询引擎的可能性。

下个阶段我们期望的 Presto 整体架构图：



六、结束语

随着 Presto 社区的蓬勃发展，最新版本为 0.203，其中包含了大量的优化和 Bug Fix，希望跟大家一起讨论。

一个数据分析师眼中的数据预测与监控

[作者简介]束开亮，携程大市场部 BI 团队，负责数据分析与挖掘。同济应用数学硕士，金融数学方向，法国统计学工程师，主修风险管理与金融工程。

一、前言

商业智能(BI)是企业级大数据分析必不可少的组成部分，除了传统的 ETL，数据仓库，可视化报表等应用和展示层技术，如今的 BI 更是依托大数据工具兼顾且发展了数据的策略和算法层，比如利用 R 和 Python 做数据分析和数据挖掘，基于 Spark 开发算法程序等。

数据科学家和算法工程师的日常工作，如动态监控与预测，搜索排序，推荐系统等位于数据分析金字塔的中上层，其研究结果对商业决策的影响则处于金字塔的顶端。

数据质量决定了算法模型效果的上限，而数据分析的方式和模型的搭建策略则又决定了成果优劣以及其被决策层采纳的可能性大小。

本文将以一个普通数据分析师的视角，阐述 BI 日常工作中的数据分析方法以及在统计模型搭建过程中的注意事项。鉴于篇幅限制，内容只涉及一些简单的统计模型，如预测和数据监控。

二、预测与监控

2.1 非时序预测

在机器学习和深度学习大行其道的当下，一个好的预测模型不在于应用了多么高深的算法，而在于如何从简单的模型开始进行尝试，兼顾业务逻辑，基于某个 baseline 来控制时间和应用成本。

对于非时序数据或是无明显趋势、季节特征的时序数据，回归和树模型是目前主流的预测方法。

广义线性回归，如线性最小二乘和 logistic 回归，因其模型的可解释性，从诞生之日至如今依旧发挥着其不可替代的作用，如金融风控中评分卡的开发，医学中对患者生存期限的研究等。

为了处理非线性问题，依托着分布式计算，又孕育出树模型和基于树的 boosting 模型，如 Decision Tree 和 Xgboost，以及后续的 LightGBM 和 CatBoost 等。

考虑到线性回归和 logistic 回归在处理非线性问题上的短板，以及为了适配模型需对数据做大量的预处理，如填补缺失，防止共线性等，我们自然偏向于树模型来做分类和回归预测，Xgboost 便是一个很好的选择。R 和 Python 都提供了 xgboost 的接口，Python 不仅拥有 xgb 的原生接口，更有适配 sklearn 的接口，便利了参数的网格搜索。

对于预测任务，我们的应用场景主要分成两类：

- 离线（T+1）预测，主要针对小批量数据，通过 shell 脚本调用 R 或 Python 的服务器，返回结果。
- 实时预测，由于线上预测需要实时响应，如在毫秒级内返回模型预测值，跨平台跑模型并不能满足要求。这就需要将模型文件打包成 PMML 文件供 Java 调用，响应速度极快。

作为一名数据科学家，不仅要保证数据处理的效率和质量，也要关注模型本身的应用规范。

比如应用 Xgboost 时，是否对分类变量做了正确的编码。首先，xgb 分类器只接受数值型变量，任何的字符型变量都需转换成数值型。

其次，分类器默认数据是连续且是有序的，2 一定比 1 大。 但无序分类变量的特征值之间是没有可比性地，比如变量“城市分类”，其特征值分为：一线城市，新一线城市，二线城市，其他城市。如果将其编码成（1，2，3，4），分类器便会误解为二线城市大于新一线城市，事实上特征值代表的仅是一个类别，不可相互比较。

可行的处理方式是对此类变量采取独热（one-hot）编码，每个特征值都作为一个新的衍生变量，每个衍生变量都是一个二元（0/1）互斥特征，这种编码方式充分考虑了分类变量每个特征值的独特性。当然，如果特征值过多，特征矩阵也会过于稀疏，此时可基于业务逻辑和数据分布对特征值进行分组处理。

模型调参，一个重复却又不可缺的步骤。可能有人觉得调参带来的提升并不明显，不值得费时费力。但这却是数据科学家严谨态度的体现，作者认为调参的目的不仅在于获得模型提升，更在于通过多次实验，基于概率确保模型参数的稳健性。

在实时预测模型中，打包的 PMML 文件不仅要包含模型文件，还要包含数据的预处理过程，这就需要借助管道（Pipe）将原始数据的处理过程（如编码，标准化，正则化等）和分类器的训练过程串联，再将管道本身打包成 PMML 文件。

恼人的是，管道一体化的过程限制了特征工程中的个性化发挥，接口提供了一些简单的数据转换函数和自定义函数功能，但这远远不够。此外，网格搜索过程中参数的赋值方式也略有改变。最后，特征重要性的可视化也并不友好，原因在于管道中的数据预处理掩盖了原有的特征名称。（读者如有个人见解，欢迎交流）。

管道中的网格搜索还需注意：假设通过 sklearn 接口预先定义了分类器，后利用管道包装了数据预处理过程和分类器，那么在网格搜索时，参数赋值相比传统方式将有所改变。

定义分类器：

```
xgb = XGBClassifier(...)
```

定义管道：

```
pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', xgb)
        ])
```

普通的网格搜索方式：

```
pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('classifier', xgb)
        ])
```

Pipe 中的搜索方式：

```
param_dist = {
    "classifier__n_estimators":  range(2, 10, 2)
}
```

Python 的 help 文档中指出了 Pipe 中的参数赋值采取二级结构：（分类器__参数：值），而非传统方式：（参数：值）。

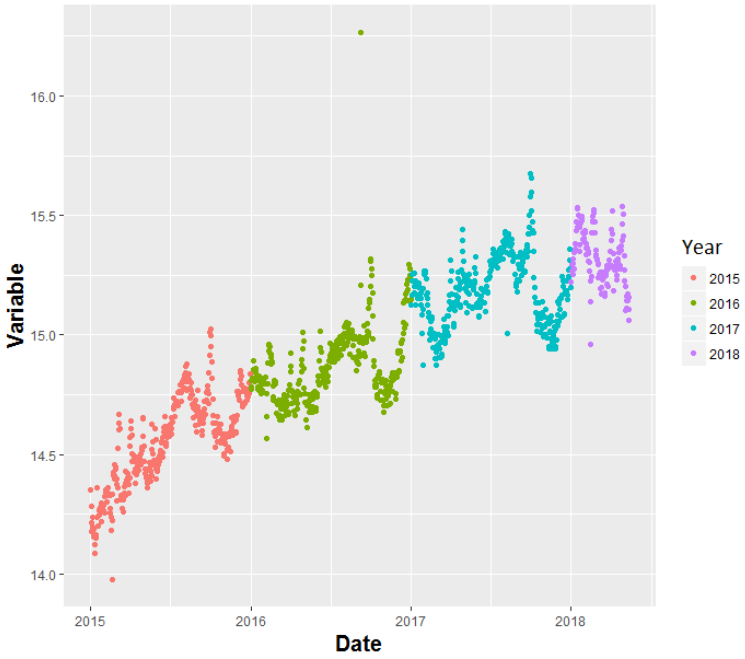
2.2 时序监控与预测

时序监控，主要还是针对各业务指标的异常值检测以及模型上线后的稳定性检测。

业务指标的异常检测多是单变量的检测，而提及单变量异常检测，首先想到的便是 3 sigma 原则。和线性回归中的极大似然估计类似，3 sigma 准则的应用前提需假设原始数据满足或近似满足正态分布，而实际数据往往具体一定的偏态性。作为数据科学家，切莫不假思索的去应用惯用的准则，而应从数据的分布出发找到合适的途径来分析数据，比如对偏态的数据采用 Box-Cox 转换。

其次，业务指标的监控是个双重任务，一是要及时发现数据中的异常，二是要对未来一天或是一段时间进行预测。如果能找到一个统计模型同时处理这二重任务，问题会显得简单多，可一个特定的模型往往很难适应多个场景。

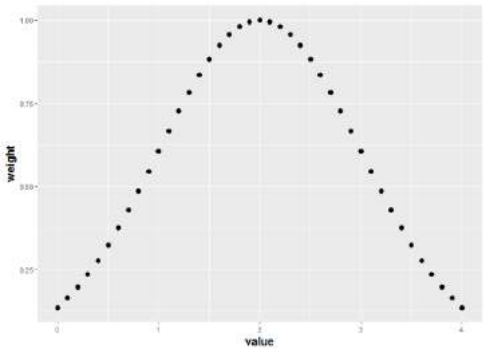
以某个业务指标为例，下图展示了指标近几年的走势，天的时序图有非常明显的季节效应和增长趋势，同时也可观察到一些节假日对业务指标的影响。



STL 时间序列分解法可针对此类数据做异常检测和时间序列预测。模型的核心由里外双重循环构成，内循环主要利用局部加权回归对季节效应和趋势做平滑处理，外循环将根据内循环的拟合效果重新调节观测值的权重，观测值偏离大的点权重低。

举例说明，在内循环中，预测点 $x_0 = 2$ 处的函数值：取 x_0 某一邻域（窗口）内所有点（支持缺失值处理）进行加权回归，假设邻近权重函数如下（仅为假设，非 STL 中的邻近权重设置）：

$$\omega(x) = \exp(-\frac{(x - x_0)^2}{2})$$



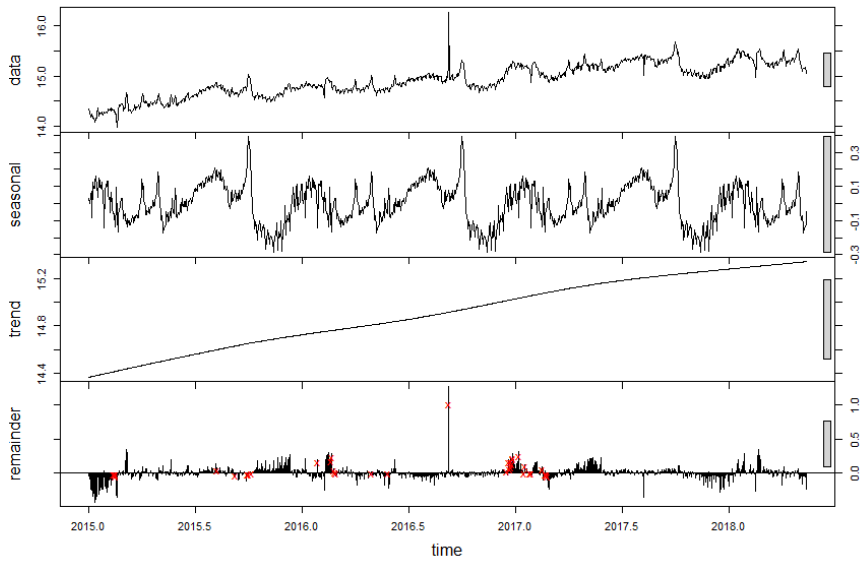
上图可以看出参与回归的点 x 离 x_0 越近，权重越高，x_0 自身的权重为 1。

内循环的局部加权回归属于非参数模型，可用来解决非线性问题，但是当数据量较大时，算法则需要更多的存储来重新计算各观测点的权重。

STL 中经过一轮内循环，得到趋势和季节项，那么每个点的余项可由观测值减去趋势和季节项得到。余项反应了观测点的稳定性，外循环将根据余项大小重新赋予各观测点一个稳健权重ρ(x)。新一轮的内循环将以 $\omega(x) \times \rho(x)$ 作为新的权重参与趋势和季节效应的平滑处理，而 $\rho(x) < \text{threshold}$ 则成为我们判断观测是否异常的准则，threshold 可由自己给定，权重

$\rho(x)$ 较小的观测被判定为异常。

下图是原始数据和 STL 模型中分解出的季节项，趋势项，和余项序列，红色×符号标记了数据中的异常值。



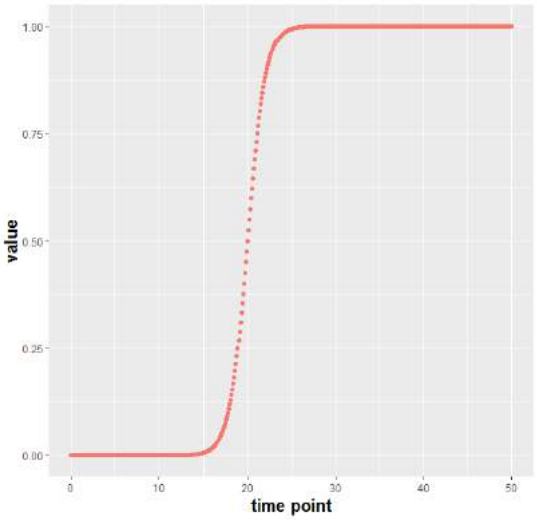
可以发现趋势项单调且具一定的线性性。按照此模式去预测，T+1 天的预测值不会有太大偏差，但 T+h 天的预测值必会受到趋势项的作用，偏离正常范围而显得过高。此外，模型也未考虑到节假日因素的影响。

如果专注于预测的高精度，Prophet 模型是一个不错的选择。模型通过 Fourier 转换可分解多周期序列，且在趋势处理上也有了新的创新，比如考虑环境和系统的承载力，提出阻滞增长和分段线性增长的想法。

阻滞增长模式：此模式中的趋势（增长）函数如下

$$g(t) = \frac{C}{1 + \exp(-k(t - m))}$$

其中，参数 C 是环境或系统的最大承载。函数 $g(x)$ 一阶导数大于零，二阶导数在$(x > m)$上大于零，所以 $g(x)$ 的增长速度会逐渐放缓趋于饱和状态。取 $C = 1$，$k = 1$，$x_0 = 2$ 增长趋势如下：



分段线性模式：不同时间段的增长速率不同，此模式中的趋势（增长）函数如下

$$g(t) = (k + \boldsymbol{a}(\boldsymbol{t})^{\mathrm{T}}\boldsymbol{\delta})t + (m + \boldsymbol{a}(\boldsymbol{t})^{\mathrm{T}}\boldsymbol{\gamma})$$

其中，k 是线性增长率，m 是漂移项，向量 δ δ的每个分量对应了不同时段的调节增长率，向量 $\boldsymbol{\gamma}$ 的每个分量对应了不同时段的调节漂移，示性函数向量 $\boldsymbol{a}(\boldsymbol{t})$ 指示了当前时刻是否属于某个时间分段。分段增长的模式也可和阻滞增长模式相结合产生更为复杂的增长模式。

在节假日问题的处理上，Prophet 假设各节假日影响相互独立，且根据历史节假日的影响归纳出一个先验分布，如 $\boldsymbol{k}$~$\mathrm{N}(0,\sigma^2)$。那么时间序列中节假日的影响函数可表示为：

$$h(t) = \boldsymbol{k}^{\mathrm{T}}\boldsymbol{Z}(\boldsymbol{t})$$

考虑 n 个节假日，$\boldsymbol{k}$ 就是一个 n 维向量，每个分量满足相同的正态分布，示性函数向量 $\boldsymbol{Z}(t)$ 指示了当前时刻 t 是否属于某个节假日。

笔者认为节假日影响的先验分布假设过于理想化，同方差意味着各节假日的影响强度是相同的，而实际上不同的节假日对业务指标的影响是有很大区别的，比如春节假期对火车票购买量的促进程度会远大于小长假。当然，也可尝试修改先验分布，但试错的方式也会带来工作成本增加和时间的消耗。

针对此问题，有人在时间序列中引入协变量来辅助预测，比如一个时间序列模型加上一个树模型。虽然这种方法没有太多的理论支持，但是实际应用中却十分有效，时间序列模型抓住了树模型很难解释的季节和趋势因素，而树模型又补充非线性的其他因素，二者的结合不失为一个很好的创意。一般时间序列模型的分解式：

$$y(t) = g(t) + s(t) + \epsilon(t),$$

$g(t)$ 为趋势，$s(t)$ 为季节，$\epsilon(t)$ ε为噪声。考虑节假日因素的序列分解式：

$$y(t) = g(t) + s(t) + h(t) + \varepsilon(t),$$

$h(t)$ 为节假日影响函数，如 Prophet 模型中 $h(t)$ 的设置。利用树模型如 xgb 解释节假日影响因素的序列分解式：

$$y(t) = g(t) + s(t) + \mathrm{xgb}(\boldsymbol{x_t}) + \varepsilon(t),$$

$\boldsymbol{x_t}$ 为 t 时刻的特征向量，特征工程中往往会对节假日做日期对齐处理以及通过日期变量衍生出其他子特征。建模时先通过时间序列模型拟合趋势和季节项，得到的余项再用 xgb 来拟合。

三、结束语

数据分析与挖掘离不开统计知识和算法设计，不同场景下的问题解决方案各有不同。数据分析师不仅要有良好的知识素养，也要深谙业务背景，更缺不了工程师团队的付出。

文章仅涉及作者工作中的轻任务，以及自身的一点思考，鉴于学识和经验的局限，文中措辞如有疏忽，恳请指出。同时也欢迎读者一起交流，分享自己在日常工作中遇到的难题和解决思路。

参考文献：

[1].Taylor S J, Letham B.Forecasting at Scale[J]. 2017.
[2].Livera A M D, Hyndman R J,Snyder R D. Forecasting Time Series With Complex Seasonal Patterns UsingExponential Smoothing[J]. Monash Econometrics & Business Statistics WorkingPapers, 2011, 106(496):1513-1527.
[3].Chen T, Guestrin C. XGBoost:AScalable Tree Boosting System[C]// ACM SIGKDD International Conference on KnowledgeDiscovery and Data Mining. ACM, 2016:785-794.
[4].Cleveland R B. STL : ASeasonal-Trend Decomposition Procedure Based on Loess[J]. Journal of OfficialStatistics, 1990, 6(1):3-33.

携程机票是如何准确预测未来一段时间话务量的?

[作者简介]侯淑芳，2016 年加入携程机票大数据团队，负责数据分析和挖掘项目，目前主要负责航变预测和话务预测及排班优化。

背景

客服中心是携程和用户之间重要的沟通渠道。出于对呼叫中心成本控制以及为用户提供更满意的服务这两方面的均衡考量，我们期望能够预测出呼叫中心未来一段时间的呼入量，以此来预先准确地进行客服人员的排班，在保证话务接通率的基础上缩减人力成本。

而在实际生产应用时，不同行业的话务中心都会受到不同行业相关外生因素干扰。比如营销策略、经济周期、自然时令季节以及天气等等。因此，除了关心话务呼入量的自然分布特征外，还需要结合自身行业属性，充分考虑可能影响话务异常趋势的外生因素，把这些外生因素融入到模型中去，尽可能提升模型预估的准确率。

一、技术方案实施

1.1 底层数据清洗和流转流程规范

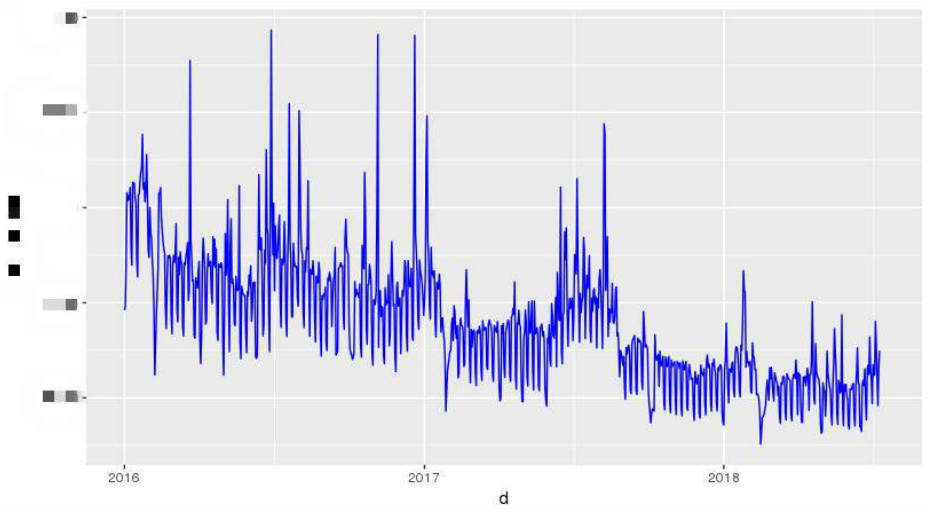
携程呼叫中心由于成立时间比较久，数据保存和传输方式比较传统，都是人工通过 EXCEL 和邮件手工记录和传输数据。这种现存的数据传输和保存方式不利于话务预测系统的自动化流程。因为，我们第一个需要解决的便是数据的自动化存储和传输。

针对此，我们为预订部的客服工作人员开发了网站，方便其上传历史过往每天的实际话务量，通过 MYSQL 落库，然后通过数据抽取流程把 MYSQL 的数据同步到 HIVE 数据库中，自动化的输入模型中。

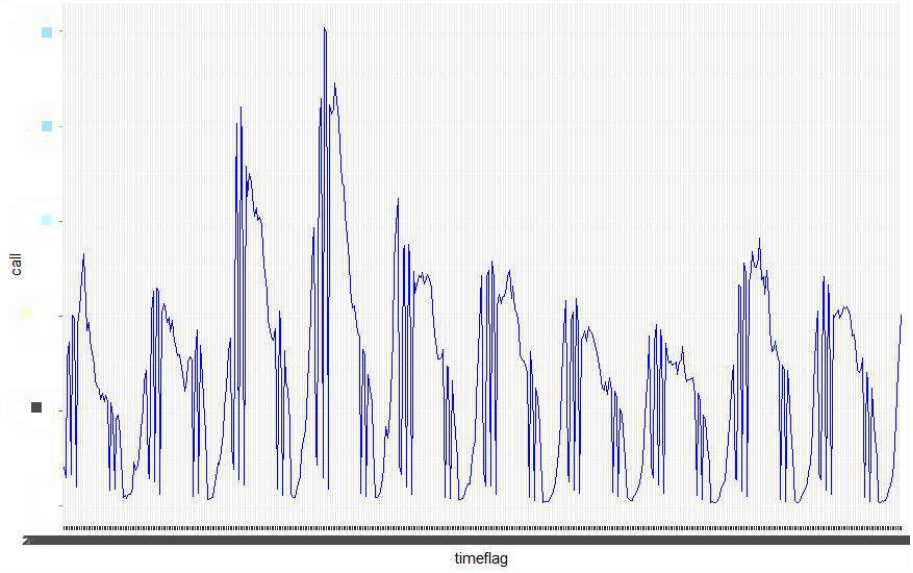
1.2 识别数据波动的趋势和规律

对数据波动的趋势和规律进行分析探索是数据挖掘的第一步工作。我们的业务场景是本周四要提前预测出下一周每天半小时粒度的携程客服中心呼入话量。

因此，我们需要观察携程客服中心历史话务数据的分布规律（包括年话务量分布、月话务量分布、日话务量分布、小时粒度分布以及半小时粒度分布），其中的日粒度和半小时粒度的分布规律如下图所示：



日话务量走势



半小时粒度话务走势

我们知道数据依附于商业活动的属性，每个行业都有其特定业务属性和商业周期。就携程这种具有代表性的旅游行业来说，其生产活动有着很强的季节性和周期性，很大程度上会受到旅游淡旺季、出行天气状况以及经济景气等等各种因素的影响。根据上图的数据，我们可以看出携程话务总体来说呈现出来以下几点特征：

1）时间序列特征明显
我们可以看到整个数据的走势呈现出很明显的时间特性。

2）周期特征多样化
通过数据我们可以看到，携程的话务呈现出明显的年周期波动、月周期波动、工作日与双休日这种周周期的波动以及一天之内小时级别的周期性波动。

3）逐年递减的发展趋势

这种人工接听话务逐年递减的趋势很大程度上要归功于携程 APP 技术功能的完善。由于携程历年来都很重视技术的研发和用户的操作体验，其 PC 端和 APP 的功能设计愈来愈智能化，能极大程度的满足用户的自动化操作需求，因此减少了大量的人工来电咨询量。

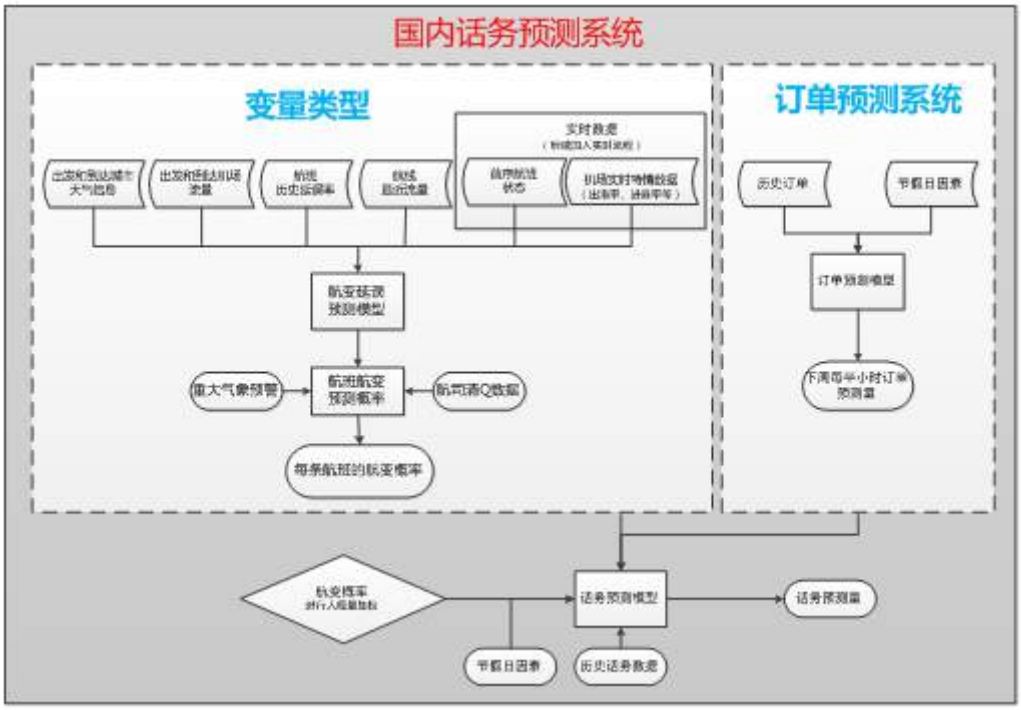
4）节假日波动
节假日的话务量明显要低于正常工作日以及正常双休日的话量。由于携程机票主要提供空中交通的出行服务，其出行热度会直接受制于受到旅游淡旺季的影响，而其咨询期一般都集中在用户的出行前（也就是节假日前），在节假日中，用户都已经处在目的地享受假期，因为呼入话量会急剧下降。

5）异常值波动
同于受制于携程机票的业务属性，用户能否顺利出行很大程度上掣肘于天气状况，如果发生了极端不适合飞行的天气状况，势必会出现航班延误甚至取消，而此时就会出现携程客户中心的话务暴增点，如果不能提前预估到这种爆点的话量，合理安排客户人员，那在这种时点就会严重影响到用户的体验。

1.3 特征工程准备

我们知道业内有句俗话“特征工程做不好，调参调到老”，由此可知，特征工程很大程度上决定了一个项目的成败。在预测一个对象时（Y），我们不仅要关注这个 Y 内部的运行规律，更要注重跟这个 Y 相关外生因素 X 的影响。

在上面的篇幅中，我们已经研究的 Y 的运行规律，接下来我们需要考虑的是 X，而 X 因素的构建很是依赖于业务经验，针对携程机票的商业属性，我们知道机票订单是影响来电量的一个很重要的因素。同时，在上面的篇幅中，我们也提到过极端天气状况是导致话量暴增的直接因素。另外，旅游淡旺季以及节假日这些时间特性也是直接导致话务呈现周期趋势的主要原因。因此，在已有的航变预测系统基础上，我们构建了话务预测 V1.0 系统:



1.4 V1.0 预测系统模型原理

基于工程部署上面简单、高效、快捷的需求，结合对模型预测准确率的考量，在 V1.0 系统的实现中，我们融合了外生变量、傅里叶项和 ARM 模型，也即广义上的 ARIMAX 模型：

$$y_t = a + \sum_{i=1}^{M}\sum_{k=1}^{K_i}\left[a\sin\left(\frac{2\pi kt}{P_i}\right) + \beta\cos\left(\frac{2\pi kt}{P_i}\right)\right] + ARMA + \text{其他因子} \quad ----(1)$$

$$\left(1 - \sum_{i=1}^{p}\emptyset_i L^i\right)(1-\mathrm{L})^d X_t = \left(1 + \sum_{i=1}^{q}\theta_i L^i\right)\varepsilon_t \quad --------(2)$$

1.5 V1.0 预测系统存在的问题

尽管 V1.0 系统能够满足日常的预测准确率需求，但是我们发现在节假日和重大天气的预测准确率远没有达到我们预想的准确率。而我们发现这个问题很大程度上要掣肘于模型的表现形式，由于傅里叶项和 ARMA 的周期只能抽取固定周期的频率，针对农历节假日这种变动周期的频率却没有办法进行精准的刻画，同时我们添加进去的外变量，比如航变和订单因子的影响，在 ARIMAX 模型中主要都是通过线性关系来影响 Y（也就是我们的话务量），其训练出来的模型在这些外生变量上面的系数都比较小，最后导致该种模型不能很好地拟合出这些 X 和 Y 之间的关系。

1.6 V2.0 预测系统的开发

为解决 V1.0 系统暴露出来的问题，我们又开发了 V2.0 的预测系统，想通过结合时间序列模型在周期性建模上面的优势以及树模型回归刻画变量之间非线性关系的特长，来提升整个系

统的预测精度。

出于此，我们首先对原始的话务数据进行了 BOX-COX 变化，然后对变化后的数据进行三阶指数平滑建模（tbats 模型），接着 tbats 的残差做 ARIMA 建模：

1) $y_t^{(w)} = \begin{cases} \frac{y_t^w - 1}{w}; & w! = 0 \\ \log(y_t); & w = 0 \end{cases}$

2) $y_t^{(w)} = l_{t-1} + \emptyset b_{t-1} + \sum_{i=1}^{T} S_{t-m_i}^i + d_t$

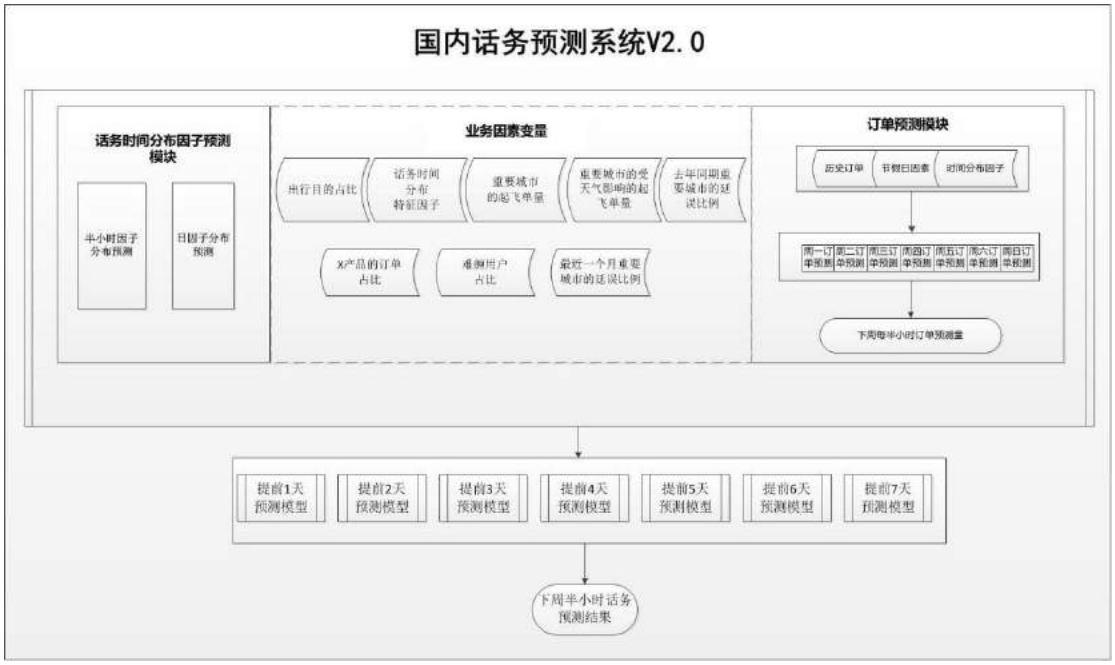
3) $l_t = l_{t-1} + \emptyset b_{t-1} + \alpha d_t$

4) $b_t = (1-\emptyset)b + \emptyset b_{t-1} + \beta d_t$

5) $s_t^{(i)} = s_{t-m_i}^{(i)} + r_i d_t$

6) $d_t = \sum_{i=1}^{p} \varphi_i d_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} + \varepsilon_t$

最后用 XGBOOST 回归树结合我们的 X 对 arima 模型的残差进行修正，最终用修正过后的残差来修正 tbats 模型的结果。



本项目从传统的时间序列模型开始尝试，逐步修正不同模型在项目应用中产生的问题，我们回测了不同模型在 2018-03-05 到 2018-07-10 这段期间的预测准确率，各模型最终的表现如下：

预测模型	平均预测准确率
STL	73.9%
ARIMAX+Fourier	78.9%
TBATS	79.6%
XGBOOST	82.7%
XGB+TBATS	89.5%

二、小结

本项目主要是针对携程呼叫中心的人力排班问题，提前一周进行话务预测。本项目中的方法同样适用于其他行业呼叫中心来电预测。

我们最终发现树模型回归优于传统的时间序列方法，但是能否充分发挥树模型的预测功效，很大程度上取决于特征工程的好坏。

尤其是在时间序列数据的预测上，传统的时间序列模型已然不能解决各行业所遭遇的业务因素对时间序列趋势的干扰，为了把业务因素纳入到预测系统中，我们必须充分构建能够很好的刻画出业务干扰因素的特征，如果你开始考虑树模型回归，请千万记住要把时间规律和趋势提取出来并作为特征放入你树模型中。

质量篇

携程 DARE 回归测试实施二三鉴

[作者简介]李艳秋，携程金融支付测试团队工具组负责人，专注于测试技术探索及测试工具研发。

正如我们所知，在整个项目的测试工作中，回归测试所占用的时间和资源消耗是整个测试周期的 70%-80%。在开发不断做 BUG 修复时，在系统不断维护过程中，都是回归测试必须出场的节点。

携程与众多“历史悠久”的 IT 企业一样，计算机系统历史悠久，同样“深不可测”。需求版本的叠加，人员的流动，难免在“传承”的过程中有所遗漏。

再加上近年来携程改造项目的逐渐增多，如：去 SP 改造类项目、Dot Net 转 Java 项目，对改善回归测试的技术或工具的研究探索是非常有必要的。

DARE 平台的使用，我们将一个手工回归需要 20 人日的项目降低至 5 人日。5 人日的工作中包含了对被测模块新旧版本的调研、配置、数据拉取和整理、环境搭建配置、测试执行、对比结果并完成报告。因为前面配置的可复用，后面的 bug 修复及多轮回归测试只需轻轻点一下按钮，即可等待最终的执行结果。发现的 bug 远比手工测试多。

在一个历史久远的通道上我们发现了一个特别出众的 bug。原因是该通道的存在对于手工测试和开发同学几乎是印象模糊的。另外还有各种特殊交易类型的未支持。让线上无数条银行通道，每个通道平均 10 种的交易类型的繁琐验证变得不再烧脑费时。

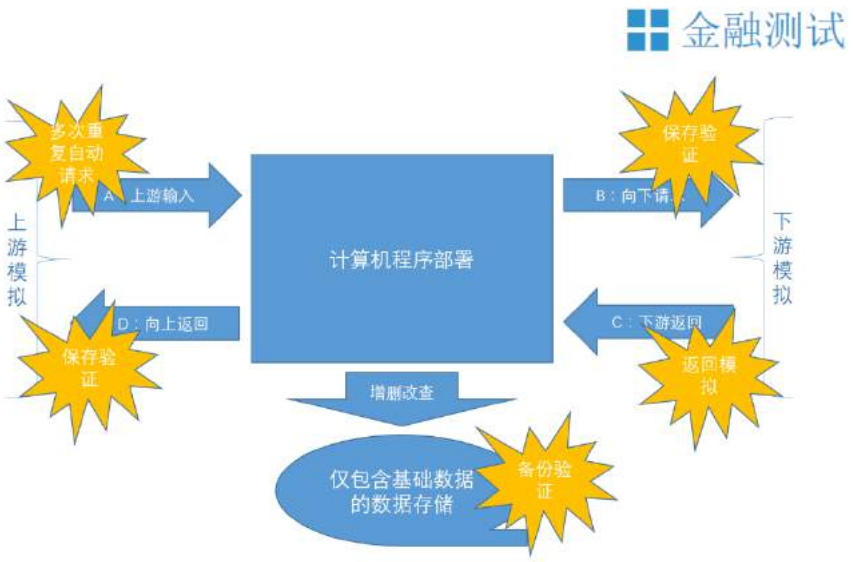
一、什么是 DARE 平台

DARE 策略是指利用生产上实际发生的用户请求，来完成新版本回归测试的软件测试方案。

为了 DARE 策略的落地使用，我们开发了 DARE 平台，用来自动拉取和整理数据，测试执行及结果验证过程。

DARE 适应的场景即那些系统出参、入参及 DB 结构改变不多，且内部逻辑有变化的系统变更。完全适合技改项目，及日常系统迭代过程中的回归测试。

DARE 平台的工作原理如下图：



在整个测试执行过程中，DARE 平台帮我们获取生产环境用户访问产生的中间数据；对新旧版本之间数据差异进行处理；对生产环境与测试环境之间的数据差异进行处理；模拟上游动作重复请求测试环境部署完成的应用；模拟下游为该请求返回相应的返回报文，以便让整个应用程序能够将一去一回的场景覆盖全面。

如上图，执行过程中需要验证的几个点，DARE 平台也同样设置了埋点。

首先，在这个模拟过程中，平台会保存 B 点向下请求的数据，以作为对 A-B 之间应用程序的验证数据。再次，平台会对 D 点向上返回的数据进行保存，以作为对 C-D 之间应用程序的验证。另外，应用程序运行过程中的数据存储我们也将保存并加以验证。

如何验证保存好的数据呢？为被测系统设置基准（基准版本+基准数据+基准 DB），测试产生数据与基准做对比。

基准版本可以是生产正在运行的版本，整理好的生产数据，生产应用程序版本在测试环境执行后产生的数据。除此之外，也可以拿某个比较出色的测试版本在测试环境执行后所产生的数据作为基准数据。DARE 平台可以直接将某个版本的执行结果设置成基准数据。

并且 B 点、D 点以及数据存储之间的对比工作平台也是可以一键完成的。

二、两个项目

下面分享两个主要采用 DARE 策略和 DARE 平台完成测试实施过程的试点项目。

2.1 项目 1

第一个是支付业务的 Dot Net 转 Java 项目。这个项目刚好符合出参、入参及 DB 结构变化不大的场景。

不过，出乎意料的是，这个项目的 B 点、C 点不是我们希望的 SOA1.0、SOA2.0、restful 等等中的任何一种接口。它是一个队列，对的 Rabbit MQ。当然这不是难题，我们增加了对 Rabbit MQ 的数据获取和抛入的支持。

前期的准备工作，首先了解老的系统架构，做新旧系统架构比较，确定系统回放的切入点。

过程中发现整个系统架构确实变化比较大的，切入点向外放大了一点点。这一点点对于整个测试来说，还是有差别的。不过差别仅在于需要增加应用程序部署的工作量，当链路中出现问题时，需要多验证几个节点。对测试结果没有任何影响。

与切入点同样重要的东西是每个切入点的日志记录及新旧版本之间的差异。分别与开发负责人沟通后，确认日志在 clog 里可以获取，并获取后确定了新旧日志间字段的转换规则。DARE 平台里配置起来。

与此同时单独搭建一套干净独立隔离的测试环境。使用环境预先尝试若干条处理好的数据，并尝试小规模执行。以此来确保拉取和整理的配置无误，期间的反复周折一笔略过。

确保没有问题之后，大批量的数据拉取开始了，clog 提供的 API 循环拉取 7 天的请求日志。拉取和整理的时间算下来每次大约 12 小时。验证对比出报告顺利搞定。

由此看出，创新改善工作。

2.2 项目 2

当第二个项目走进 DARE 的时候，我们总结了前一个项目经验教训。召集了相关干系人来了一个启动会，会议中搞定切入点选择、日志获取方法、新旧请求之间的差异问题。减少配置过程中的反复周折所耗费的资源和时间。

根据上一次 clog 拉取耗时及数据量表现，与大数据团队报告进行对比之后发现数据有所遗漏。再次与 clog 团队沟通，获得的支持是从 zeus 平台直接导出 hive DB 的方式来获得 clog 中的原始数据。将 12 小时的处理时间降低至 30 分钟，并且保证了数据的全面。

改进了上面两点后，效率再一次提升了一大截。本来 20 人日的测试任务，最后在 5 人日内完成。

在效率提升的同时，DARE 策略保证的场景回归的全面，发现手工回归很难发现的各种模糊问题。上线以来无事故。

通过第二次实践，可以看出：

第一，逐步的熟悉系统架构，可以更少的依赖开发同学。随着版本的推进，一些回归项目在 DARE 平台上的配置可以复用，更可以减少反复配置时间。

第二，环境搭建及应用的部署同样是一个占用时间资源的环节，有待解决。

分支集成加速器 Light Merge 在携程的应用

[作者简介]苏玲，5 年软件配置管理及 6 年持续集成经历。曾在苏州科达科技和大众点评任资深配置管理工程师，目前为携程代码中心负责人，专注于代码相关的平台建设，致力于提高研发效率与研发质量。

本文分析了 Ctrip 代码平台提供的集成加速器（Light Merge，简称 LM）产生的背景及其特点，并具体说明了 LM 在多特性分支上线流程中发挥的作用。LM 给不使用特性开关的项目的集成与上线，提供了一种高效便捷的解决方案。

我们期待更多的 Ctrip 团队能够享受到 LM 的服务，也希望 LM 能为同行的代码集成提供有效的参考，让研发的集成尽可能不那么繁琐，不那么低效。

一、Ctrip 研发状况和面临的问题

1）项目通常有多个功能一起开发，开发的初期为了减少彼此干扰，会为每个功能创建特性分支。

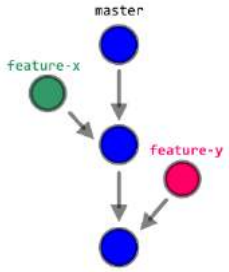
2）受研发不可控因素的影响，对于未来的某个时间点，多个特性功能存在是否能集成的问题。

3）各特性分支进入到稳定阶段，必然需要 merge 在一起，然后一起被编译、打包和测试。怎么让这个过程更加自动化呢？

4）多个特性分支，难免会修改公用文件，如果不及时关注公共文件的变化，以后各特性分支之间可能出现难 merge 的情况。

二、Ctrip 主流开发模式及分支模型

1）采用特性分支开发模式。
2）主推的分支模型：
master 分支为最核心的集成分支，用于上线。
master 分支的任何一个 commit，符合质量要求。
统一从 master 拉取新分支。
与 Paas 集成后，统一省略环境分支。



三、单个特性分支怎么合入 master 分支?

为了保证集成分支的质量，在 gitlab 上集成分支通常都被保护起来（protected），不允许直接 push 到被保护的分支。不过，我们可以通过发起 Merge Request 的方式把特性分支合入到集成分支 。借助 MergeRequest，我们可以完成 sonar 静态检查、代码 review 等质量管理的活动。

四、个特性分支会给集成带来哪些问题?

1) 不同分支可能会修改相同文件，集成时很可能出现代码冲突。
2) A、B 两个分支先后合入到集成分支，B 合入后导致 A 分支对应的功能发生故障。
3) A 合入到集成分支后可能需要一套测试环境；B 合入到集成分支后也可能再需要一套测试环境。多特性分支分别合入集成分支所需的测试环境也多。

五、靠什么快速发现多特性分支集成问题?

六、LightMerge 功能及特点

1) 合并冲突自动告警
2) 多分支自动 merge
3) 自由选择需集成的特性分支
4) 高效定位可集成的特性分支的最大集合
5) 设置非常简单
6) 支持 CI

7) 与 Merge Request 有机结合

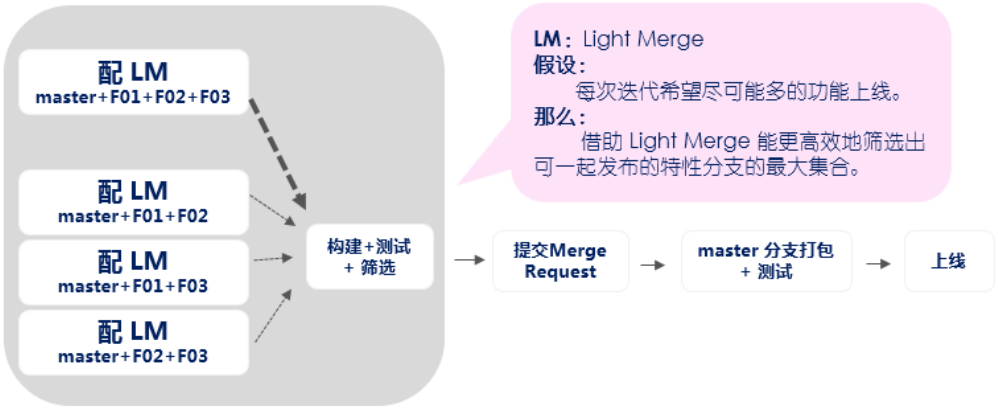
七、LightMerge 和 Merge Request 的关系

1) MergeRequest 是正式的 merge 请求，在开发的某个时机，用来把一个分支合入到另一个分支。

2) LightMerge 服务于开发的过程，在不影响任何一个分支的基础上，把>=两个分支做集成，发现冲突及时通知当事人；一旦某个分支发生变化，LightMerge 立即重新 merge、检查并通知。

3) 当它俩在一起工作时，可以大大提升团队集成的效率与质量。

八、多特性分支上线流程（推荐）



说明：

1) F01、F02 和 F03 三个功能，每个功能对应一个特性分支，并行开发。
2) 三个特性分支的开发人员通过自测后，各自发起了合入到 master 的 merge request。此时，团队可以做 codereview，sonar 静态扫描等检查活动。
3) 于此同时，负责集成的人员借助 Light Merge，构建、打包并测试后发现 F01 和 F03 的功能集成后可以一起上线，而集成 F02 后发现有问题。
4) 最后，F01→master 及 F03→master 的两个 Merge Request 被接纳，而 F02→master 的 Merge Request 被拒绝。

上面第 3）点在确定哪些特性分支可以一起上线的过程中，可以借助 Light Merge。

代码平台上 Light Merge 设置如图 A：

图 A

如果自动集成时代码发生冲突，则 web 上会提示冲突，也会 Email 通知给相关人员。如 F01 和 F02 分支修改了同文件的同一行，LightMerge 自动 merge 后 web 提示信息如图 B:

图 B

邮件提示信息如图 C:

图 C

文件冲突解决后，如果设置了 CI，则集成后的代码会自动编译、打包并部署到 FAT 测试环境。团队成员可根据测试结果主动调整 Light Merge 的特性分支的集合（也就是 Source Branches），直到筛选出最大的集合，该集合对应的功能通常就是下一个发布版本可发布的功能集。

至于这几个特性分支一起集成到 master 的方式，有两种做法：

Merge 方式	**优点**
直接把 Light Merge 的分支发 Merge Request 到 master	便捷
每个特性分支单独发 Merge Request 到 master	一旦需要 revert 某个特性，直接在对应的 Merge Request 上执行 revert 即可。

九、结束语

对于不使用特性开关的项目（上线后所有功能都会启用），我们必须保证上线后各个功能正确且有效，这对集成的效率和质量提出较高的要求。

在对多个特性分支做集成的时候，如果不借助 Light Merge 类似的工具，负责集成的人员需要做许多繁琐又重复的活动；反之，只需简单的调整需参加集成的特性分支的集合，就能靠 Light Merge 自动完成这些特性分支代码的集成、构建、部署，甚至自动化测试，从而筛选出用于上线的特性分支。因此，我们称 Light Merge 是分支集成的加速器。

Light Merge 功能目前已被 Ctrip 诸多项目团队作为分支集成的加速器，我们期待更多的团队能够享受到它带来的便利。

基于图像比对技术，低成本维护的携程机票前端测试平台 SnapDiff

[作者简介]陈亮，携程机票 BU 高级测试经理。在互联网服务端、前端的软件质量领域有多年的实战经验，喜欢钻研引入新技术，提升团队工作效率。

前言

前端由于直接和用户交互，在互联网公司具有天生的快速迭代特性，这使得测试代码的维护成本非常高，往往跟不上产品的迭代速度。

传统的测试方法是由测试人员根据测试用例，在测试代码中添加关键元素的校验点，随着测试代码的不断积累，维护成本不断上升。

每次页面改版都会带来大量的维护工作，维护工作量太高也带来投入产出比不高的问题。妥协方法往往是只维护少量核心用例，但随之产生的问题是测试覆盖不够全面，很可能会带着一些问题发布。

携程和大多数互联网公司一样，也遇到了上述问题。由于携程所处的旅游行业业务相对较复杂，业务场景众多，全面覆盖需要大量的测试场景。

以携程的机票业务举例，单一个订单详情页面的全面覆盖就需要上千个用例。在版本快速迭代的背景下，完全依靠手工测试肯定是不可行的，所以我们致力于通过一种技术方案去解决上述问题，SnapDiff 平台应运而生。

一、平台简介

平台的目标是替代按传统的维护检查点方式，降低测试代码的维护成本，提升测试用例的覆盖面，推动测试效率提升，并且能够真正帮助发现问题。

1.1 设计时考虑以下五个方面：

1）易用性：好的平台需要尽量减少学习成本，可快速上手。所以我们选择了前端最熟悉的 JavaScript 语言作为开发语言。为了方便不同的使用场景，支持手工触发和持续集成触发测试。

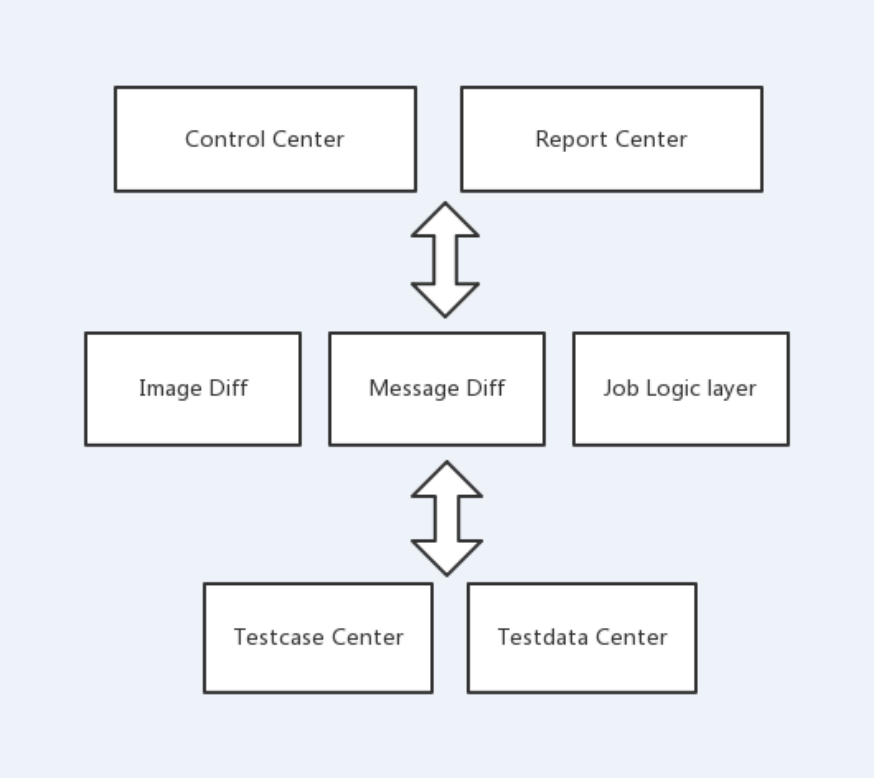
2）稳定性：平台支持两个层面的对比测试，前端展示层的图像对比和前端消息处理的监听对比，需要保证获取图像，消息监听，对比算法的稳定性。

3）高效性：前端测试相对接口测试耗时很长，需要降低用例耗时，采用分布式多并发的方

式。

4）扩展性：支持两个层面：测试数据和用例的快速扩展和测试页面的低成本接入扩展。

5）通用性：支持 H5，online，offline 多个使用渠道的 web&crn-web 端页面。



1.2 如何工作?

1）测试用例设计&测试数据准备

用户创建测试任务，在任务中配置测试用例和数据。测试数据由我们的数据中心 API 提供，通过关键字进行动态关联。为保证数据的稳定，我们针对数据都只做了数据镜像，在使用前服务会将数据恢复到初始状态。

2）配置任务执行模式

任务可以配置为持续集成执行和手工触发执行两种模式。持续集成执行主要对应回归测试用例，手工执行对应的是日常需求的测试用例。

3）测试任务执行

平台会从前端展示和消息监听两个层面去进行比对测试。

前端展示：根据不同的测试用例自动打开测试页面，访问不同的业务场景，截图保存，然后

自动去和上一个对照版本进行图像比对。比对是基于像素级别的，比对中如果有不同的区域，就会高亮标记。

消息监听：所有的页面被打开后，平台会记录页面的请求和返回消息，保存在服务端，然后自动去和上一个对照版本进行比对。报文中如果存在不同节点，就会记录并展示。

4）测试报告分析

测试比对结果会根据配置发送到收件人的邮箱，如比对无差异则显示通过，如存在差异则需要打开报告链接进行结果分析。

这样，测试人员只需要专注于设计测试用例，做测试结果分析，就可以很快确认系统是否存在 Bug。比对是基于整体页面的，所以不需要再针对每个元素单独去维护，维护工作量自然比原先大大减少。即使遇到页面大改版，我们的代码也基本不需要调整。真正做到了一次投入，长期使用。

二、使用场景

某一天，小王收到了一个需求，在界面上对某个元素的展示逻辑进行了调整。使用这个系统，小王需要设计好测试用例，在系统里配置好需要执行的测试数据，后续执行的工作就可以交给系统去自动执行完成，系统执行完成后会自动发送报告邮件给到小王。

报告里会汇总界面和消息处理与原先基准版本的对比区别点，然后小王要做的是仔细分析一下报告中内容，考虑下是否存在问题，是否后续还需要增加的哪些场景等。

三、遇到的问题和解决思路

3.1 提升报告准确性

1）前端的样式经常会有一些细微的调整，有时调整一个空格，一个换行就会导致页面中的元素位置产生变化。如果我们想忽略这种变化，只关心展示的内容是否正确，是否有一种方式可以快速实现呢？

我们想到了图像文字识别，通过这种技术，可以直接告知用户具体的不同点，用户不必看图，减少了报告分析者的分析工作量。

2）前端有的模块是已知的不同点，比如说：广告轮播模块，针对这类部分可以设置截图时自动忽略这个部分。

3）比对结果智能分类：针对不同数据发现的同一类问题，系统会根据不同点进行自动聚合分类，只在报告中展示一个样例，使用者可自行决定是否需要查看其它数据

4）智能忽略：使用者在分析报告过程中，如果发现一些不同点是正常的，可设置忽略，系统下次对比时就会自动将这类内容标识为可忽略。

3.2 提升执行效率

1）使用无头浏览器测试方案

在我们的 SnapDiff 平台中，我们使用了 Chrome 开发团队去年新推出的 PuppeteerAPI，Puppeteer 是一个 Nodejs 的库，支持调用 Chrome 的 API 来操纵 Web，相比较 Selenium 或是 PhantomJs,它最大的特点就是它的操作 Dom 可以完全在内存中进行模拟既在 V8 引擎中处理而不打开浏览器，而且关键是这个是 Chrome 团队在维护，会拥有更好的兼容性和前景。

2）分布式并发

平台首先会将任务分发到不同的测试服务器，然后在每台服务器上多线程并发执行，通过这种方式，整体耗时大幅减少。

3.3 降低接入成本

1）支持自助接入
对于非定制类的常规测试需求，用户可自助创建测试任务，通过在配置中心里配置对比页面的 url, div，运行规则，报告收件人等即可实现接入

2）支持 Job 自助启动
测试任务的执行控制可自助操作完成

3.4 测试数据稳定性问题

数据是测试的前提，这个问题如果不能保证稳定会导致测试结果不稳定，我们具体是通过下面两种方式去解决。

1） Mock
通过我们的 mock 平台，将测试用例和 mock 报文关联起来，用例开始执行前会首先自动配置好所有的 mock。

2） 数据镜像&恢复
将数据库中的测试数据创建数据镜像，保存在镜像仓库中，测试前通知数据服务做镜像恢复，这种方式比 mock 方式更贴近真实环境。

四、使用效果

不能帮助我们找到系统问题的测试框架都不完美，说明这个框架只是解决了一部分问题。所以我们在设计这个平台的时候，是抱着能够尽量多的发现问题的目的去做的。我们希望在测试的每个阶段都能够把工具引入进来，而不是只在某一个阶段。

工具可以支持开发同学自测，测试同学做新功能的测试，也可以做最后的回归测试，正因如此，它能够帮助我们发现更多的系统问题。

在我写这篇文章过去的 30 天内，这套比对系统帮助我们发现了 60 几个系统 bug，未来我们还会不断优化，让这个系统发挥更大的作用。

我们希望测试人员专注于测试用例的设计和测试数据的梳理，测试执行类的工作我们会尽量交给工具去完成，当然前端的特殊性决定了一些任务仍然还需要进行手工测试，但我们一直在努力尽量把这些执行类的工作自动化，让测试人员更专注于测试设计，测试报告分析，优化，形成了一套测试设计->工具执行->测试报告分析->测试设计再优化的闭环。

五、平台的不足

1）目前接入的主要是 H5&Web&Crn-web 页面，后续会逐步支持 Native&RN 接入。
2）测试报告的易读性还可以进一步提升。
3）稳定性和性能需要进一步提升。

携程 QA-流量回放系统揭秘

[作者简介]康猛，携程网站运营中心资深技术支持工程师。在互联网系统架构设计、后端开发、性能测试领域有多年实战经验。喜欢钻研新技术，善于转化研究成果，提升工作效率。

背景

众所周知，在产品迭代过程中，功能测试与性能测试是必不可少的两个环节。在产品上线的过程中，做容量预估离不开性能测试，在产品迭代过程中，测试 Case 覆盖率是功能测试必须要关注重要指标，甚至是一行代码的修改，没有经历过严格的功能与性能测试，都有可能导致大的生产故障。

近年来，携程生产环境应用改造项目逐渐增多，快速构造贴近生产的测试用例，压力可调的功能与性能测试场景，显得越来越不可或缺。

如何输出较大的压力源?

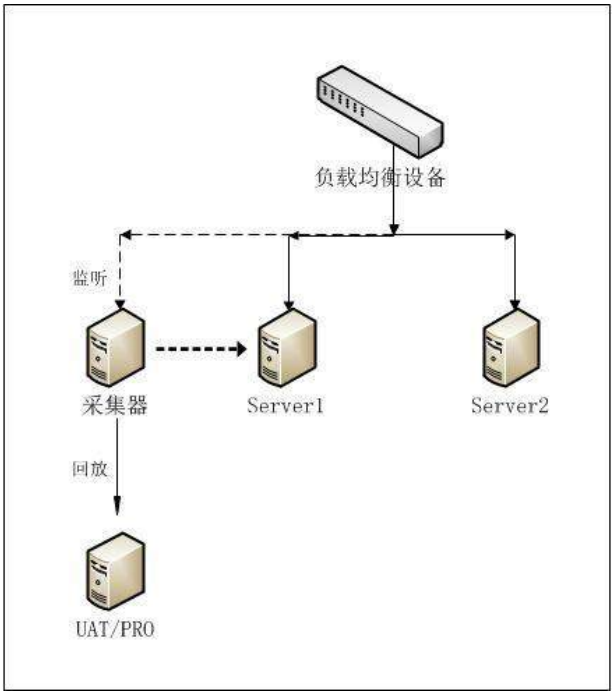
也许你心里已经涌现了很多现成方案，也许是 ApacheBench，可是想要用 ab 模拟生产的上下文顺序会不会有点痛苦？也许你想到了 JMeter，或者 LoadRunner？暂且不说 JMeter 和 LoadRunner 二次开发封装上下文逻辑的复杂度，单就想想需要人工构造海量贴近真实生产场景的测试 Case，就够让人头疼的了。

使用生产环境流量回放系统，可以使用生产环境的海量真实场景作为测试 Case，天然解决了海量测试 Case 的构造问题；系统支持录制多倍的生产流量，在回放过程中支持加压多倍进行回放，解决了较大压力源构造的问题；此外，系统天然支持 4 层与 7 层协议，可以快速支持各种特殊应用需求; 最后, 系统跨平台的特性, 也决定了系统具有较好的场景适配能力。

一、方案

流量回放系统，利用生产上现有真实流量进行镜像，原始流量依然回到生产环境的真实服务器，流量的镜像拷贝会分发到集群外的测试服务器上，在测试服务器上可以实现不同版本的功能测试，或者加压 10 倍进行性能压测。

流量回放系统的工作原理如下图:



一般极简的集群模式包含一套专用的负载均衡设备或者承担负载均衡角色的网络设备，以及一组提供具体服务的后端服务器，如上图 Server1、Server2。

首先，系统向集群扩容一台机器，我们称之为“采集器”，这台“采集器”实现的基本功能是转发集群正常流量，将全部流经自己的流量，原样输出回到原集群中指定的一台或者多台后端应用服务器，比如 Server1，实际上“采集器”可以监听到数倍于单台 Server 的流量。

同时，“采集器”将监听到的流量制作一份镜像拷贝，可以保存成离线文件的方式，也可以在线实时地将镜像流量转发到生产或者测试环境下任意网络可达的服务器上，保存的离线文件，可以在任意时间重新回放成离线的生产流量，直接转发或者修改后转发到任意网络可达的服务器上，在目标服务器（我们称之为“回放机”）上可以运行有版本差异的测试代码，以观察生产应用与测试应用的表现是否符合预期。

也许你会疑惑，这样难道不会导致 Server1 承担了两倍的流量吗？其实并不会，原因是在默认情况下，我们的生产环境中集群的 Server 的权重均为 5，机器的权重以及比例，共同决定了集群流量的分配，当流量复制任务开始时，系统向集群中扩容一台“采集器”，其权重默认为 5，同时，系统会调整 Server1 的权重为 1，如此一番，则采集器与生产 Server 之间的权重比例将变成：采集器:Server1:Server2 = 5:6:5，这样既保证了采集器能录制到足量的数据，也保证了生产机器之间流量不会因过分叠加导致容量问题。

既然是在生产环境操作流量拷贝，紧急情况下的容灾恢复就显得非常重要，特别是当宿主机故障，或者应用 Full GC 导致健康检测踢出等场景下，如果系统不能感知到后端成员已经宕机，而继续将生产流量转发到已经故障的机器，则会导致这部分请求失败，从而导致故障，这就需要一种自我保护机制，在后端应用故障的场景下，引流任务能够自动识别并终止。

但是从另一方面，由于一次偶然的 Full GC 或者网络抖动等，导致机器被踢出集群，整个任

务就立即终止又稍显敏感。在实际系统引流的过程中，当检测到引流目标机 Server1 宕机，会自动将任务暂停，连续 3 次探测 Server1 均失败，则判断 Server1 短时间内可能无法恢复，立即自动终止任务；这样既保证目标服务器宕机不会对生产环境造成持续影响，又照顾了偶发检测失败造成的任务终止过于敏感的问题。

二、系统

1) 系统总体上包含“引流任务设置”，“回放任务设置”，“任务查询三个模块”；

2) 设置引流任务，需要用户提供的信息主要包含 AppID、PoolID 以及引流的目标机，也就是要 Copy 哪台机器的流量；系统会根据 APP 信息和 Pool 信息进行各种关联校验，并自动获取服务器信息，供应用选择；
3) 最后设置开始与结束时间，即可快速开始引流任务；
4) 设置一个回放任务。设置回放任务，可以通过在任务列表页的最后一列，点击“去回放”，会自动跳转到回放任务设置页，并将流量复制任务的相关参数自动填充，需要用户填的只有回放系统的主机头 HOST，回放目标 IP，回放倍率等跟任务本身有关的信息；
5) 查看监控数据。复制/回放任务的全过程，都可以实时的获取到应用的各种维度监控数据，系统已经集成了一些常用的监控指标，比如 CPU、内存、连接数等信息，也将 ES 系统集成到系统中，并自动设置了一些常用参数，用户可以方便的查看在流量复制与回放阶段应用的请求/响应状态码、响应时间等数据。

三、项目

3.1 项目 1：内网某服务的流量复制

该项目中，系统为内网某服务集群扩容一台采集器，采集器将正常流量，转发回原集群中正常提供服务器的机器上，并将这份流量拷贝两份镜像，一份以抓包文件的方式，保存为 pcap 格式的抓包文件，另一份以请求报文的形式，保存成离线文本文件。

以应用服务器的视角，可以看到所有通过采集器进入的流量：

3.2 项目 2：内网某服务流量回放

该项目中，系统将事先录制好的离线流量文件，转换为请求的形式，回放到集群外用于多版本对比的测试服务器上：

四、总结与展望

当前流量回放系统能够比较便捷的获取真实生产环境海量用户请求的流量镜像数据，既有效保存了原集群流量无修改无损耗，又解决了人工构造的测试数据不能拟合生产真实场景的问题；除此之外，系统具有良好的跨平台支持特性，不管是 Windows 平台的应用还是 Linux 平台的应用，均能够镜像出完整生产数据；还有一点很重要，系统支持 7 层协议，支持自定义各种 HTTP Header 定制与修改。

技术平台的生命力需要不断的打磨才能变得更好。接下来我们也在寻求使用更少服务集群场景侵入的方式，获取生产流量镜像，比如将“采集器”下放到“目标回放机”上共存，以期完美解决直连应用场景。

如何利用Xcode实现线上代码覆盖率的检查

[作者简介]姜睿东，2009 年加入携程，从事无线研发，现在大住宿事业群担任酒店无线研发工作。

清理项目中的无用代码是日常开发中非常重要的一环，定期清理废代码既可以保持代码的简洁，也可以让代码逻辑变得更清晰，不给后人留坑。

比较传统的寻找无用代码的做法，一般是查找没有引用的方法或类，这个可以很容易的通过脚本来实现，甚至有的 IDE 自身就能提供这个功能，再进一步的话也可以在网上找到一些开源算法的脚本，来查找重复或相似的代码。

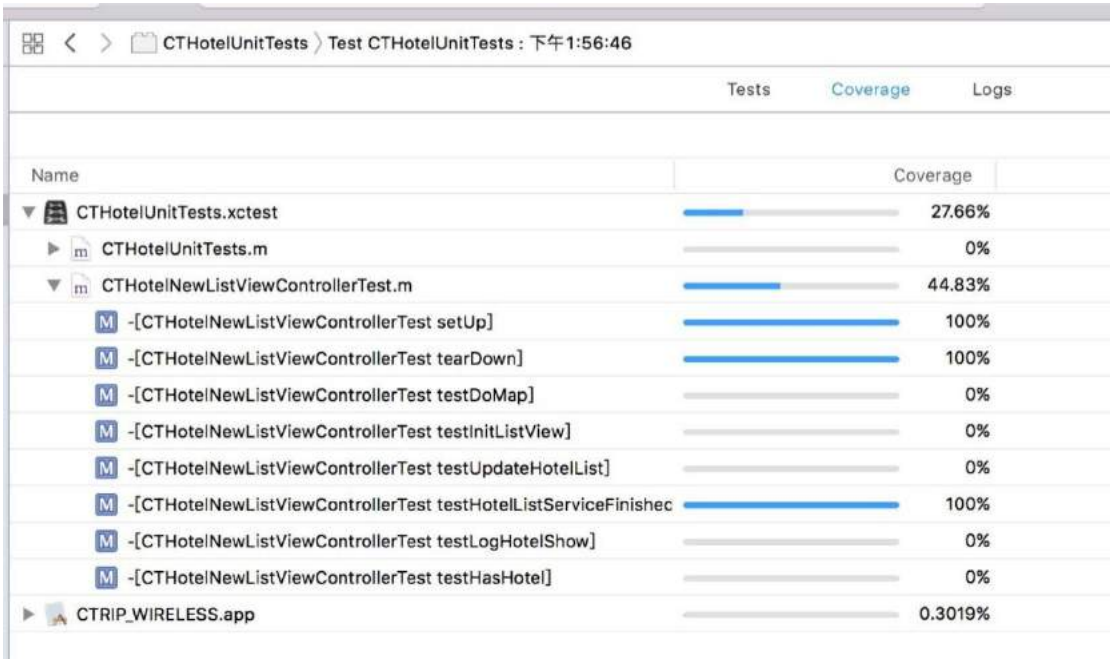
随着携程酒店业务的快速发展，线上版本的迭代频率越来越快，代码量开始急剧膨胀，以上这些方法已经不够用了。如何及时清理无用的代码，变得越来越困难。

大量的无用代码不是靠检查一下无引用就能发现的，因为我们有着数量庞大的服务端及客户端实验，以及频繁上线下线的业务，靠人肉很难发现哪些是无用代码，而 app 又对 size 有着极为苛刻的要求。所以怎么高效率的寻找无用的或利用率极低的代码，成为研究方向。

首先想到的是检查线上代码的覆盖率，没有覆盖到的部分，就是所谓的无用代码。

那么，怎么来检查线上代码的覆盖率呢？网上一般会采用“插桩”的方式，思路就是在代码的每一个函数中植入埋点代码，然后在后台利用一套算法来计算代码的覆盖率，用这种方式得出的结果相对比较精准。但是我们对代码有些洁癖，并不想对代码有任何的破坏，而且这种方式在后台的计算也是相对比较繁琐的。

我们想到的办法是利用 Xcode 自带的 Code Coverage 来检查代码的覆盖率。Xcode 的这个自带的工具非常的好用，不但可以方便的可视化的看到代码覆盖率，还可以看到代码被执行的频率，如下图所示：



通过这个报表，我们可以很清楚的看到代码被执行的情况，这样就可以针对那些没有被执行到的代码进行具体原因的分析。

但是 Code Coverage 只能在单元测试的 case 中才能使用，而单元测试一般用的都是 mock 数据，酒店业务极其复杂，各种真实数据不太容易造出来，很难真实反映线上代码的执行情况，并不能直接为我们所用。

于是我们把目光投向了我们的自动化测试平台，我们的自动化测试平台有一个流量回放的功能，可以回放线上的真实数据，平时用来自动回归服务端 case，存有千万条数据，足以覆盖绝大部分线上的 case。

我们设计的大概的流程图是这个样子的：

从图中可以看到，我们的 UI 测试用例往测试平台发出的是一个空的 request，然后由测试平台随机从日志数据库中抽取相应用例的 response 返回给客户端，如此循环足够多的次数基本上可以覆盖到这个用例的全部 case。

这样我们就有了一个理论上可行的应用框架，不过还需要解决一个问题，那就是我们的一个页面上往往有数十个小服务，而且互相之间都有数据依赖，自动化测试平台只能接受单个服

务的请求，且无法对应这个服务相关的其他小服务的数据，没有了联动性，就给我们的单元测试带来了麻烦，没有办法完整的测试一个页面。

对此我们又对 UI 测试和自动化测试平台双双进行了改造，首先在自动化测试服务的 api 前面，仿造生产环境一样搭建一个 Gateway，由这个 Gateway 来负责 json 和 pb 之间协议的转换，如下图所示：

这样的话，我们的单元测试无需在原来业务代码里做太多修改，只需要把原来指向生产 Gateway 的地址指向自动化测试平台 Gateway 就可以了，只要几行代码就可以实现一个列表页的测试。

由此我们得到了一个完整的自动化测试线上代码覆盖率的框架，通过不定期的跑自动化 UI Case，就可以得出线上代码的真实覆盖率。

带有业务逻辑的比对思想在接口测试中的应用

[作者简介]虞斌，携程机票 BU 资深测试开发工程师，主要负责携程机票测试工具以及基础组件的研发。对自动化测试领域有较深刻的认识，对新技术有着浓厚的兴趣。

前言

在互联网企业中，开发项目的快速迭代是必不可少的。这就导致了大多数情况下，很多测试人员的回归测试速度远远跟不上项目开发的迭代速度。

传统的接口自动化测试是测试人员事先编写好测试用例，写好相应的验证点，然后通过手动执行或者 Jenkins 自动执行用例，来达到接口回归测试的目的。

但是对于一些结构复杂度高、内容大的报文（如机票引擎的报文）来说，通过手动编写测试用例来做回归测试的成本很高，而且很难做到全面覆盖。

所以我们致力于寻找一种解决方案，使其能够低成本、高效率地解决上述问题，接口通用比对工具随之应运而生。

一、传统的比对思想

大家都会说，报文比对么，直接用 beyondcompare 等第三方的工具，把需要比对的文本粘贴进去，直接就能出结果了。确实，这么做也是比对的一种方法，但是这个只适用于结构比较简单的接口。

在实际的项目中，有一些接口的结构被设计的非常复杂，且自身结构还带有复杂的业务属性。这种情况下，传统的比对思想就变得不那么适用了。

二、什么是带有业务逻辑的比对思想

比对逻辑的本身其实很简单，就是同一层节点的“一对一”对应，然后分别进行比对，但是如何能找到这“一对一”的对应呢？

首先，我们可以把接口报文的节点分三种情况：

a）节点是叶子节点。（即：节点值的类型是 string、int、float 等基本类型，已经无法再遍历进去的节点）这种情况最简单，只需要通过节点 name 就可以找到对应关系。

b）节点是一个自定义的类型。这种情况需要对自定义类型的每个属性进行遍历，然后通过

属性名找到“一对一”的对应关系。

c）节点是一个数组集合。这种情况下的对应关系是最难确定的。因为集合中的元素通常并不是有序的，所以集合里面元素的对应关系就不能简单暴力地通过 index 的方式来确定了，所以我们需要另寻解决方案。

为了解决数组集合中“一对一”对应关系的确定，我们提出了一个业务逻辑 key 的概念。业务逻辑 key 是指在数组集合中某个元素的一个或者多个属性值的组合，并且在这个数组中可以唯一确定这个元素。

举一个机票的例子：在一个航班信息的无序数组中，航班号（flightNo）和日期能够唯一确定一个元素，那么 flightNo 和 date 的组合就是这个集合的业务逻辑 key。

通过业务逻辑 key，我们能够以更贴近业务的方式来确定集合中元素的对应关系。也能够很好地解决集合的乱序问题。以达到带有业务逻辑的比对思想的目的。

三、对于关联结构的比对思想

机票的一些核心报文体量是非常大的。为了能够进一步压缩报文，我们设计了一种我称之为 Agg 结构的报文结构。即把同一类可能会被重复使用的节点抽出放到另外的节点数组中进行统一管理并编号，在原来使用的地方引用该编号作为关联关系。

举个例子：在查询国际航班的时候，大多数情况下返回的是航班组合。这种情况下，同一个航班可能会有不同的组合而出现很多次，如果每出现一次就把该航班的所有信息都放进报文里，那么报文中就会出现很多重复冗余的航班节点，这大大增加了报文的体积。

而 Agg 结构的出现，则把所有航班节点都放在 FlightList 的节点中去重复，然后按顺序编号，原则上每个航班号只会在数组中出现一次。而在航班组合节点中只输出航班号对应的编号的组合，有点类似于关系型数据库。这么做的好处就是大大减小了报文的体积。

但是对于我们测试或者比对逻辑来说，这却是一个巨大的新的挑战：

a）如何处理编号。编号是在抽出重复节点过程中，为了能够唯一确定某个节点而顺序给的唯一编码，它本身并没有并不具备任何业务意义，且在重复请求中，同一个节点的编号可能会不同。所以，在比对过程中，我们不能简单的将它们直接进行值的比较，那样没有任何意义。

b）为了解决这一问题，我们引入了 reference 的概念。即在接口业务逻辑配置的时候，通过编号设置节点之间的关联关系，在比对之前通过该关联关系先计算出所有关联节点的业务逻辑 key，这样，在之后的比对过程中，通过已经计算出的业务逻辑 key 准确的找到需要比对的关联节点。再通过通用比对逻辑，递归遍历所有节点。

基于以上，我们设计开发了一个针对复杂报文接口的通用比对工具。该工具的特点是比对逻辑通用，业务逻辑可配置。通过这种方式，可以有效地降低后续的维护成本。

四、功能介绍

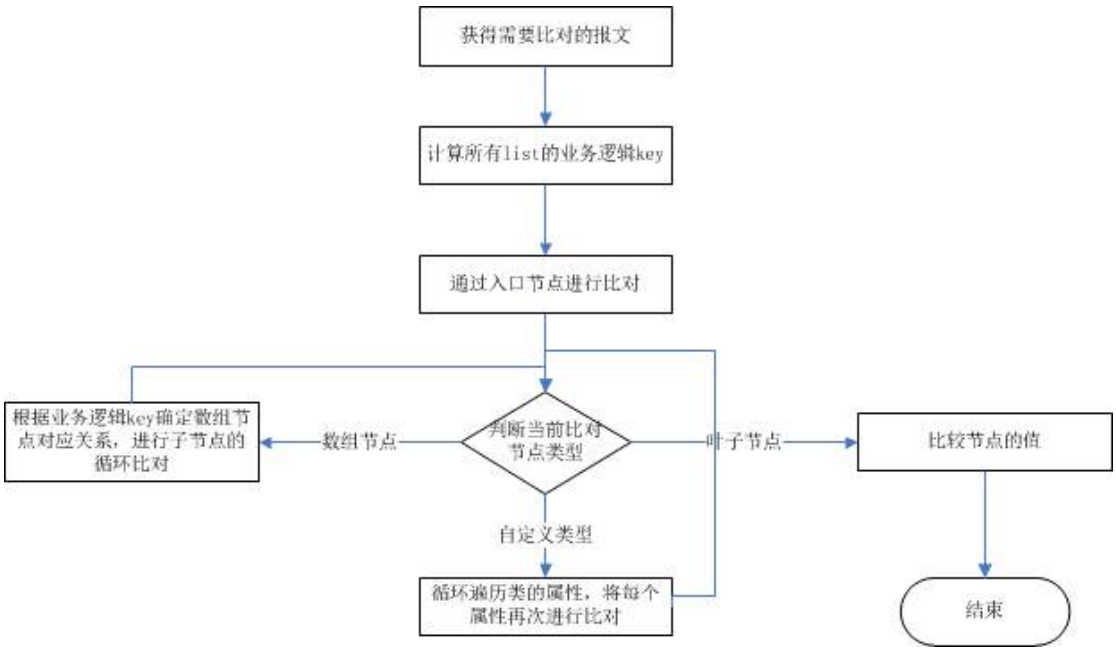
4.1 接口管理

a）接口信息配置——获得接口的请求、响应报文契约结构。
b）测试用例模板管理——该功能可以将测试用例绑定到接口上，方便测试套件添加测试用例。
c）响应报文业务逻辑 key 配置——可以为每个数组节点添加业务逻辑 key，不设置默认按顺序进行比较。
d）响应报文 Reference 配置——适用于 Agg 报文的结构。用于嵌套计算业务逻辑 key。

4.2 测试套件管理

a）基础配置——每个测试套件对应一个接口，里面可以有若干个测试用例，测试用例可以从接口模板克隆而来，也可以另外创建。
b）例外节点排除——可以排除一些不需要参与比对的节点，如时间戳等。
c）用例执行——并发的执行套件中所有的测试用例。
d）结果展示——展示比对结果，测试人员只需关注并分析执行错误的测试用例即可。

4.3 比对流程



特点：

a）业务逻辑配置——可以理解为把接口的业务属性翻译成一条一条的规则，录入系统，然后让系统能够理解报文的结构。

b）比对逻辑通用——针对任意一种报文结构，比对的逻辑是不会变的，即找到一对一的对应关系后，逐一比较对应节点的属性，直到最后的叶子节点。

c）降低复杂接口的测试门槛——所有接口的逻辑关系只需要在新建的时候配置一次，通常会由最熟悉该接口的开发人员来配置。然后使用方只需要执行用例，然后分析用例中不同点是否符合预期即可。这样的话，即使是新接手的开发或者不太熟悉接口结构的测试人员也能够很快上手并完成一轮接口的回归测试。

五、结束语

以上是我对复杂报文比对的一些理解，在这里和大家分享，希望文章能对大家有所启发，欢迎大家在学习的过程中一起交流讨论。

携程微服务架构下的测试浅谈

[作者简介]施赛花，携程机票 BU 测试工程师，主要负责携程机票聚合层服务的测试，以及自动化工具的开发。善于研究新技术，并转用于实践，提升测试工作效率。

前言

在现在这个互联网时代，多变的市场环境以及日益增长的客户需求，加速了产品的迭代和更新。传统的单体式应用已经因跟不上发展的步伐而逐渐退出历史的舞台，取而代之的是 SOA 的问世。

SOA 的出现使系统架构发生了跳跃式的转变，它提出了面向服务的设计思想。而现今流行的微服务架构，和 SOA 虽是一脉相承，但不再强调传统 SOA 架构里的 ESB（企业服务总线）。

使用微服务架构可以将系统划分成更细粒度的服务，每个微小服务实现单一业务功能，且可以根据自身特点选择更适用的编程语言和技术。每个微服务可以由不同的团队独立完成开发，互不影响，加速了产品的推出和迭代。

夸张一点说，如果将整个软件系统比作宇宙的话，那每个微服务就好比行星，它可以独立的运行在自己的进程中。各服务之间通过网络通信，看似独立，却又因业务联系关联在了一起，就像各个行星之间有着引力等物质将它们紧紧关联。一系列独立运行的微服务就共同构建起了整个系统。

一、微服务架构下测试的分类

以传统建筑行业为例，项目的完成需要设计院、施工方和监理单位协同合作。类比到软件系统中，架构师、开发和测试也是缺一不可。微服务架构下，更要求采用的测试策略能够为服务内部的完整性，以及每个服务之间的交互，提供全面的测试覆盖。

相比于常见的 UI 测试层、接口测试层、单元测试层，三层测试金字塔，在微服务架构，这个分类可以被扩展为 5 类。

下面将逐一介绍在微服务架构中主要采用的测试类型：

1.1 单元测试

单元测试是针对代码单元的测试，通常只测试一个函数和方法调用，验证其运行结果是否符合预期，是对代码质量最快速的反馈。高覆盖率、高质量的单元测试是保障代码质量的第一道保护伞。在掌握 TDD（Test-Driven Design，测试驱动开发）的前提下，单元测试更是对代码重构起到了非常关键的作用。

1.2 集成测试

虽然单独测试模块非常重要，但是测试各个模块之间交互是否正常，同样也占据了重要的地位。在微服务架构下，集成测试的目的是把一些子模块组合在一起，测试其作为子系统是否存在缺陷，检查模块之间的通信和交互是否通畅且准确，是否以预期的方式协作。

1.3 组件测试

在微服务架构中，组件实际上就代表着微服务本身，所以组件测试就是检查服务内部功能实现是否完整，内部逻辑是否正确，异常处理是否正常等。

1.4 契约测试

契约测试称之为消费者驱动的契约（Consumer-Driven Contracts，简称 CDC）测试。契约测试是为了测试服务之间连接的正确性，测试服务能否符合契约预期，即是否能真正满足服务消费者的需求。

1.5 端到端测试

端到端测试是从 UI 层开始执行，目的是检查整个软件系统是否符合用户的预期需求。一个常规页面功能展示的背后往往涉及多个服务，所以运行端到端测试需要部署多个服务。这样的测试能够达到更广的覆盖面，但是也面临测试不稳定，定位问题难等问题。

在微服务架构下，前端及后端各微服务之间都是分开研发及测试。保证每个微服务本身的质量成了测试中至关重要的一环。要想楼房造的高而稳，那基础必须扎实。微服务就好比那基石，高质量的微服务，是保障软件系统整体质量的关键。

相较于端到端测试在敏捷开发中的维护成本，服务在迭代研发过程中，和 UI 相比变化相对少，所以自动化更易维护，且能在迭代中不断复用。而且，服务测试自动化受外界因素的影响较少，不会受浏览器、手机型号及系统版本等影响。

二、基于微服务架构的服务测试工具

微服务架构带来了便利，随之也给测试工作增加了难度和新的挑战。测试人员必须跟上时代的步伐，调整自己的测试方法和工具。

怎样模拟多样的上层数据以及稳定的下层服务，高效、高覆盖的完成对服务的测试。此时，携程基于微服务架构的自动化比对工具就应运而生了。

该比对工具是以生产 Log 作为测试数据源，通过 Mock、缓存等实现生产流量在测试环境的回放，以达到对服务进行全面测试的目的。

2.1 设计初衷

1）缩减测试数据维护成本、增加测试覆盖度

因是对服务的测试，首先要做的就是模拟上层应用或系统调用被测服务。常规服务测试自动化中，测试数据都是由自动化来完成构建，痛点：数据维护成本高，Case 量级偏少。

解决方案：

拉取生产的 Log 数据作为数据源，这样原来 5%左右测试数据的维护成本就缩减为了 0%。数据每天定时拉取，增加了数据的新鲜度。且拉取的数据量大，这样就能保障数据的多样性，丰富的测试场景，以达到增加覆盖度的目的。

2）测试用例的持续运行成功

因测试环境的不稳定性，以及测试环境数据的缺失，且对于下层服务返回数据的强依赖性，做到 Case 持续运行成功并不是一件容易的事。

解决方案：

通过实现 Mock，摆脱对底层服务的依赖，且能高效还原当前 Case 在生产环境的测试场景。

3）实现对被测服务所有输入输出流的验证

很多时候被测试的服务内部需要去调用其他服务，且有非常复杂的逻辑判断，对调用其他服务这块中间输出流的验证也是非常重要的，所以常规只对服务返回结果的验证远远不够。

解决方案：

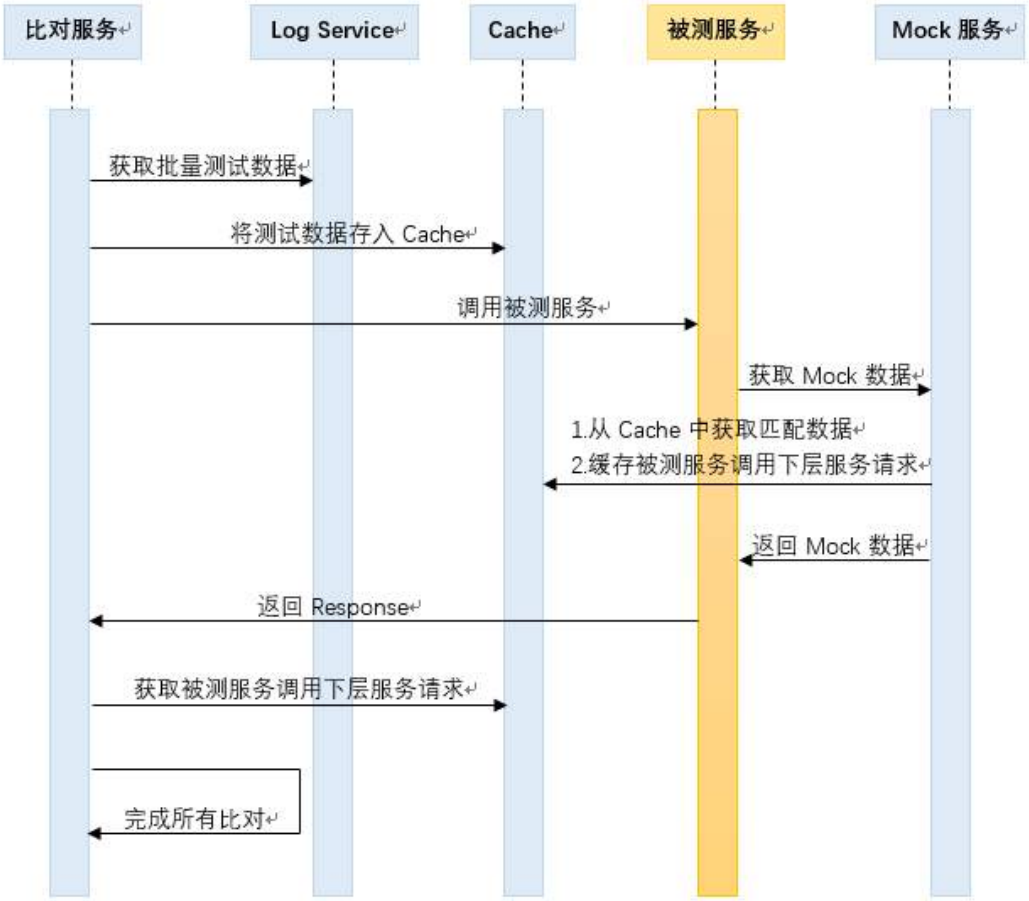
实现对被测试服务所有输入输出流的比对验证。

2.2 具体实现

比对工具中最关键的是实现两个服务：

比对服务：负责测试数据的拉取、被测服务的调用以及最后的比对；
Mock 服务：负责被测服务需要的返回结果的高效匹配，以及对调用其他服务请求报文的收集及缓存。

比对工具工作流图：



2.3 工具优势

1) 仿真生产流量

通过实现多线程，对生产上高 QPS 的服务，能够高效还原生产流量场景，并且通过模拟生产高并发时被测服务的负载量，使原本还有藏身之所的 Bug 更无所遁形。同时，每次运行 Case 的时间也有了量级上了缩减。

2) 灵活 Mock 下层服务

以携程机票一个聚合层服务为例，它需要调用超过 30 个下层服务，常规测试用的打桩服务需要对指定下层服务进行预设返回，在多线程的情况下，这种方式匹配效率较低且容易出错。我们 Mock 服务中对打桩功能进行了改良，不仅摆脱对指定服务的 Mock，并能快速准确的匹配返回结果。

3) 多种比对方式

工具除支持两个测试环境进行比对外，更支持直接与生产数据比对。不同的比对方式，满足更多的测试需求，且对于被测服务中已知有变化的节点，也支持除外不比较，大大缩减测试结果的分析时间。

三、总结

技术的发展不会停步，学习的脚步不会停止。在微服务架构下，除了对微服务技术本身的研究外，对更适用于微服务架构测试的探究，是每个测试人未来努力发展的方向。

千万级别数据 20 秒内反馈，携程酒店智能监控平台如何实现？

[作者简介]林晨曦，携程酒店研发部资深测试开发工程师，主要从事测试框架和平台的研发，现在负责监控系统与性能平台，热衷于研究技术提升测试工作效率。

前言

携程酒店业务量巨大，产生海量的埋点数据，以应用为单位接入公共日志平台；常规监控系统无法精确定位业务问题，测试人员花费大量时间查询与判断异常数据，低效且反应滞后。

为此我们根据测试痛点设计了一系列面向具体业务指标的监控，并在此基础上搭建了系统级的监控平台。本文将介绍酒店的智能监控系统，并详细阐述其中核心部分：针对酒店 Clog 和 ES 埋点数据的智能实时监控。

一、酒店监控系统

1.1 监控现状

酒店业务众多，关系链复杂，目前应用数量已经超过 2000 个，即使经过一系列线下测试，在上线以后还是要如履薄冰，业务人员要时刻关注各项指标，对波动及时查明原因，防止线上故障。

公共方面已有 Sitemon，Hickwall 等监控报表系统，我们结合了业务需求，在测试工作中构建了为测试工作服务的一些监控系统：

Smart：智能监控平台，基于主动健康监测和日志收集的智能保障系统

Mdata：性能埋点平台，针对具体业务指标在 ES&Dashboard 里埋点数据进行条件聚合用于性能预警与数据展示

Artemis：API 自动化监控平台，通过自动化运行收集数据进行监控，并对 ES 埋点涉及多个不同数据 Index 的业务指标进行逻辑处理的 ES 监控

1.2 监控工作简介

为什么要设计自己的监控系统？海量数据带来了一系列挑战：

1） 数据量大：

通过 CAT 系统采集到每天埋点数据
-2000+亿条日志
-100+T 业务日志通过 CAT 进入 ES

2）维度多：

如此海量数据，在数据大盘里即使已有过滤条件搜索仍如同大海捞针，以下为监控难点：

1）日志平台记录大量零散数据，查询不便
2）不能配置细粒度规则，用户难以定制符合业务需求预警
3） 无法判断是否遗留问题
4） 没有分析模块，不能跟踪用户分析行为
5）业务人员只关注生产环境，测试环境日志信息未起到先验效果

我们希望监控实现目标：

1） 监控为业务服务，以具体业务需求为切入点，持续集成，快速交付
2） 构建可扩展监控体系，避免为业务调整增加大量项目复杂度
3） 监控尽量实时，提供信息量全面
4） 覆盖面广，自动部署，减少人工值守时间
5） 集中式管理，方便配置

在搭建监控系统过程中，我们主要是通过两种方式：

1） 主动检测式监控：通过自动化手段模拟用户行为进行采点分析，统计 Badcase
2） 埋点收集方式：与开发配合，在代码里埋点关键数据，通过数据采集处理进行监控

监控系统以应用为维度，方便业务人员明确排查范围

埋点数据采集监控占了很大监控比重，数据真实性高，也方便做实时预警，处理埋点数据过程中采用了对已有 Clog&ES&Dashboard 埋点平台做二次挖据方式。

之所以不直接通过 Kafka 消费原始数据，因为 Clog&ES&Dashboard 已有可以配置条件的索引过滤掉用户无需关心数据，尽管数据落地有延迟，但是与需要过滤数据的量级相比，这种不利因素可以接受。以下介绍重点针对 Clog 与 ES 埋点监控内容。

1.3 监控效果

酒店监控平台目前已经配置了 30 多种主动式自动化监控，并有以下系统级监控：

1) Clog 监控
2) ES&Dashboard 规则监控
3) 机器状态监控
4) Job 监控
5) Badsql 监控
6) ART 报表监控
7) DB 数据库一致性监控
8) CAT 服务响应时间监控

监控获得收益：

1) 覆盖所有酒店核心应用
2) 系统灵敏度高，出现问题立即暴露，及时解决
3) 发现需要分析问题>60000，平台自动创建 CP4 Bug>4000

二、Clog 监控的前世今生

我们从 Clog 系统能获取到的信息：

1） 日志信息: AppID、服务器、标题、来源、信息、错误类型、CatId
2） 日志级别: DEBUG、INFO、WARN、ERROR、FATAL
3） 日志类型: APP、URL、WEB_SERVICE、SQL、MEMCACHED
4） 借助 CAT 获取详细信息

Clog 监控系统根据日志信息指标配置成规则，采用了两种监控方式:

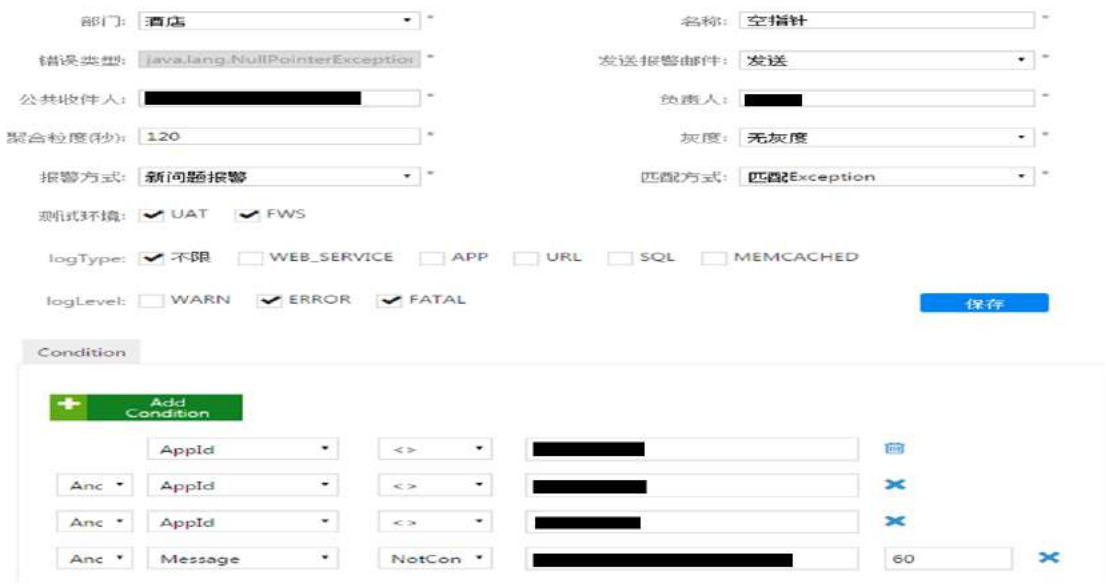
1）1.0 监控:

应用级日志监控，用户配置应用信息，设置阈值与过滤项，扫描应用日志根据历史比对发现新错误信息，超过阈值的错误量，需要关注错误内容预警到关注用户。

1.0 规则配制图

2) 2.0 监控:

重要 Issue 部门级智能监控,用户配置错误类型，设置需要关注应用信息，扫描所有关注应用里匹配该错误类型信息并预警到关注用户，生成缺陷到 CP4。



2.0 规则配置图

监控初期 1.0 的预警对象是错误量和新问题，从 1.0 到 2.0 的扩展是一次从问题收集到智能筛选的过程，如空指针引用、内存问题都是开发需要重点关注并解决的缺陷，这些处理也是可以明确为程序问题并直接开 Bug 的，从而减少了用户分析时间，而且部门级监控可以拉取所有酒店应用信息无需单独配置。

监控原理:

1) 用户在系统中配置监控规则
2) 主服务器根据用户配置自动生成执行任务，并调度分布式执行机执行，执行机分生产与测试环境，可收集不同环境数据

执行机配置管理图

3) 执行机上数据监视器通过配置规则批处理运行任务，该过程包括数据采集、算法过滤、历史比对、系统扫描，异常数据传至邮件服务器预警并推送到 CP4，目前日运行任务

数>30 万，最小时间粒度可至 20 秒

PRO-Clog 错误信息预警 配置 查看 Clog 暂停

开始时间	结束时间	每分钟错误数(总错误数)	阈值(次/分钟)	发布状态	发布时间
2018/1/24 18:51:53	2018/1/24 18:53:53	6(11)	200	堡垒拉入集群成功	2018-01-24 18:49:20

明细

title	exception	source	message	status	runNum	cat	analyze
[redacted]	[redacted]	[redacted]		new	1	链接	分析

明细(已分析-不计总阈值)

title	exception	source	message	status	分析标题	每分钟错误数(总错误数)	阈值(次/分钟)	cat	analyze
[redacted]	[redacted]	[redacted]		analyzed	[redacted]	1(1)	60	链接	分析

1.0 预警邮件

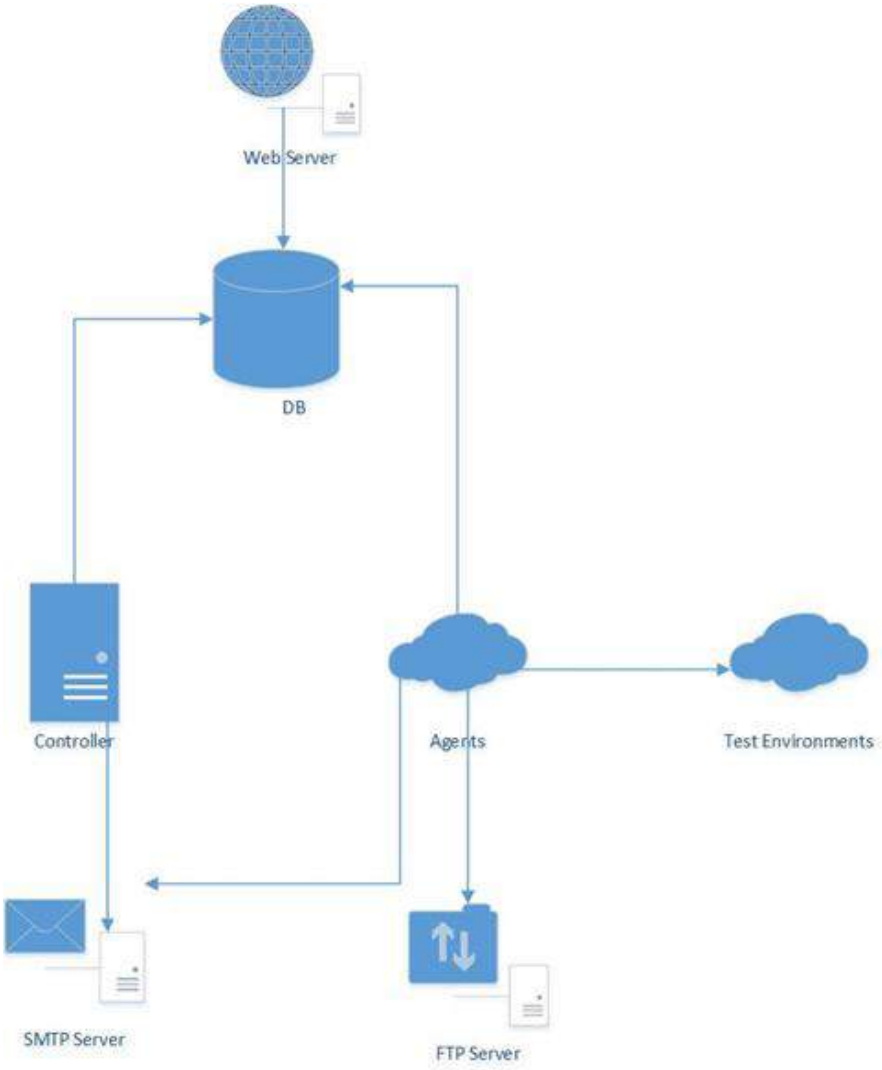
[redacted]酒店重要 Issue 预警(环境:PRO)-空指针,请[redacted]关注 查看

AppId	AppName	Error Type	Title	Exception	Source	Message	Timestamp	Link	应用最近发布时间(发布人)	应用状态	应用负责人	过去1小时有发布依赖应用	调用该应用最多的应用	CP4
配置	[redacted]	ERROR	[redacted]	java.lang.NullPointerException	过滤 Source(所有) 过滤 Source(仅本错误类型)	[redacted]	2018-07-03 16:41:29	cat clog	2018-06-27 16:11:03 [redacted]	SUCCESS(2018-06-27 16:11:03)	[redacted]	[redacted]-SMOKING(2018-07-03 16:29:18)	[redacted]:136517 [redacted]:51772 [redacted]:7123 [redacted]:4405	[redacted]

规则说明:
重要 Issue 预警自动创建 CP4Bug,报告人初始设为上一次发布人，经办人初始设为应用负责人，**Bug 未解决或解决结果为'无需修复'不再预警**
点击'查看'链接查看所有该类型错误报警
点击各 AppId'配置'链接设置该应用收件人

2.0 预警邮件

4) 监控缺陷数据处理环节，生成质量闭环报告，数据直接推送给质量平台，长期追踪解决结果。



Clog 监控系统架构图

指标监控->全链路监控:

为帮助业务人员分析问题，需要提供尽可能详细的信息，让用户从多维角度快速定位问题来源，预警信息集成了以下信息:

1) CAT 系统获取应用上下游调用关系与波动程度
2) NOC 系统获取到配置修改信息
3) 发布系统获取最新发布状态
4) SLB 系统获取机器状态信息

Clog 监控还提供了一些辅助功能：

1）集成 TTS 电话系统，重要预警可电话到负责人
2）集成 Trace 功能，有性能问题机器会被自动拉出集群采集 Trace 用于性能分析

对用户而言，使用 Clog 监控需要处理的内容：

1） 配置规则
2） 处理预警邮件，在平台及时完成分析
3） 查看个人未解决问题
4） 推动开发解决未完成 Bug

三、从 Mdata 到 Artemis：深度挖掘 ES

ES 数据量巨大，但是它自带的聚合功能可以大大减少数据量，用户关注的其实还是符合特征的数值指标。

根据特征聚合 ES 数据，我们设计了早期 Mdata 平台的 ES 监控，用户直接配置 Json 到规则，后台 Job 定时执行抓取数据源数据。平台有以下特征：

1） 配置简单，用户只需复制 ES 里的 Json 内容
2） 形式多样，可以配置百分比与数值形式
3） 监控采用系统默认与用户配置两种，系统默认采用环比增量预警方式
4） 数据采集时间<10 分钟
5） 半小时与按天预警同时设置
6） 集成系统发布信息与用户历史分析内容
7） 性能问题直接创建缺陷到 CP4 关联到性能负责人
8） 存储时间长，系统保留超过半年数据，可以查看长期趋势与历史分析内容

大量使用后的业务痛点：

1） 预警邮件多，对于波动较大指标很难界定
2） 性能问题成因复杂，定位困难
3） 预警阈值不好控制，业务变更造成前后不一致内容多，宽严两难

在进一步分析与调整中，采用了区别对待的处理方案，对于核心应用关键指标更多采用上下限阈值处理，并且预警时间达到<10 分钟。一般指标预警规则也更严格，达到连续增长趋势才会触发报警。

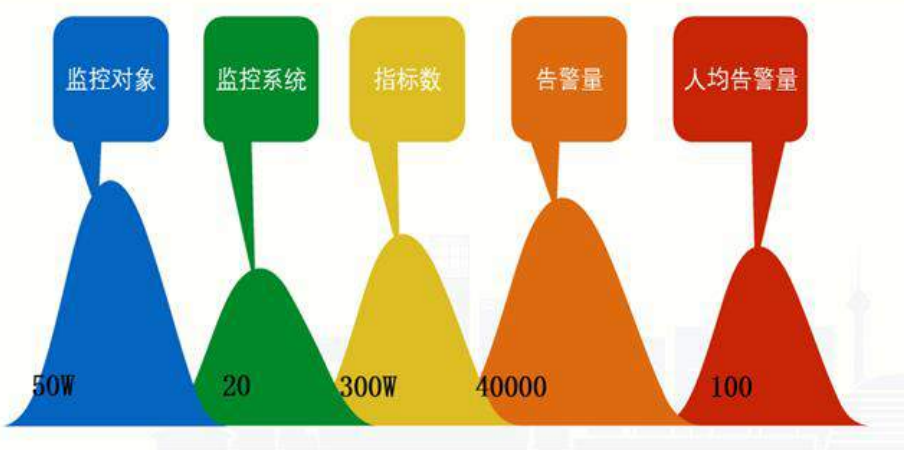
业务的发展带来很多难以界定的指标，数据埋点往往不能用单一规则来处理，需要对 ES 数据做更深的挖掘，在原来单指标监控基础上我们发展出了 Artemis ES 自定义指标监控，监控内容有以下扩展：

1) 对抓取 ES 数据做了逻辑处理，包括对不同 Index 的数据进行聚合，项目复杂度有所增加，更多应用于数量有限的关键业务指标
2) 配置规则多样化，提供了更多阈值检查
3) 维度更接近业务内容，用户有更多筛选方式
4) 实时度高，抓取数据时间<2 分钟

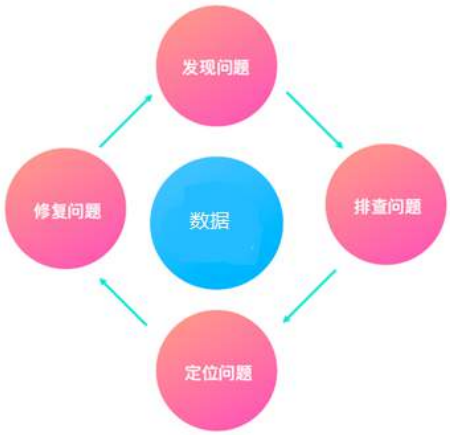
目前对 ES 数据的挖掘还有很多系统，公共 Sitemon，Hickwall 上也能配置 ES 规则与报警，我们的主动式自动化监控也有一些是对更细的包括未聚合具体埋点内容的分析，数据挖掘还有很多内容可以探索。

四、尾声

监控的目标是为了提高系统预警准确率与实时性，从而提高测试效率，减少人工值守时间。而监控全面化带来了一系列新的问题，目前公共 Sitemon 上已有超过 10000 个监控指标，大量告警需要更好的降噪处理。



从方法角度看，只有指标越细覆盖越广才能达到更高精度，而这需要大量逻辑处理提高了系统复杂度。现在流行的机器学习是个可以突破的方向，通过模型处理数据来提高报警精度，这也是监控下一步的工作目标，让预警更快更准，监控更智能化，在这个过程中需要更多思维突破与创新，让质量真正闭环。



发现问题：实时性、准确性

排查问题：错误分类细化、多维度

定位问题：全链路日志聚合、解析

修复问题：问题直达、结果反馈

基于信息论构建的测试解决方案——携程机票如何利用大数据提升测试效果?

[作者简介]陈亮，携程机票 BU 高级测试经理，在软件服务端、前端的软件质量领域有多年的实战经验，喜欢钻研引入新技术，提升团队工作效率。

前言

1948 年信息论创始人香农博士在他的论文中指出，要想消除一个系统的不确定性，就必须使用信息。当你没有收集到足够多的信息时，不确定性就是一种客观事实，无论采用什么方法，都不可能消除。

近些年来国内业界讨论自动化测试的内容比较多，另一块测试数据信息的讨论却较少，然而测试数据质量决定了自动化的效果。本文将分享我们团队是如何通过提升测试数据质量，进而提升数据的自动化处理速度，最终提升测试效果的实践。

基于信息论的理论基础，我们提炼出做好软件测试的两个原则:

1) 通过获取更多高质量的数据，提升测试覆盖率;
2) 通过提升工程处理效率，提升数据处理效率;

目标：在有限的资源条件下做到最好，降低交付时的不确定性。

一、案例分享：复杂功能 A

每次有大的功能点改动，上线后都会有不少问题。于是我找到这个功能的测试负责人，了解到每次发布会执行大约 300 条测试用例。

我：我们是否测试覆盖不足呢?

负责人：由于时间不足，我们只执行了大约一半的用例，如果下次给足够的时间，我确信能解决问题。

然后我们就这样尝试了一次。过了几天，他很沮丧的找到我。

负责人：上线仍然发现了一些问题，都是一些未考虑到的复杂场景，但是这些组合太多，很难全部测试一遍。

我：你说的这些复杂场景，我们必须要想办法解决，否则交付给用户时就会存在很大的不确定性。

接下来该怎么做？

二、首先要解决数据质量和数量的问题

2.1 扩展信息源&构建信息模型

首先，需要不断寻找新的数据信息，拓展自己的信息来源。

常用的信息源有 PRD，设计文档，测试用例库等，但是这些都不能解决复杂场景组合的问题。携程的产品业务逻辑复杂，影响数据构成的因素非常多。以文中提及的功能 A 举例，存在 10 种影响因素，平均每个因素中存在 3 种情况，那么理论上数据复杂度可达到 3 的 10 次方 59049 种。

很显然，这个功能依靠几百个用例，肯定无法保证测试的覆盖率。

真实情况下，很多理论上的组合也并不存在，所以需要过滤掉这些不存在的组合，并且为剩下的组合进行排序。

具体怎么做呢？

1）为这些因素设计埋点，构建测试信息模型。

2）通过我们设计的一个信息爬虫去采集这些组合的数据，包含数据报文，实际使用次数等。一段时间内无使用的可过滤，剩余的组合根据使用量进行优先级排序，确定测试的优先级。

3）测试人员利用这些信息辅助决策，生成覆盖率更全面的测试用例数据集。

2.2 数据持久化

测试需要持久化使用的数据通常有两类：报文数据和数据库数据。

针对报文数据的持久化很简单，比较麻烦的是关联表数据的持久化。针对这类数据，我们的方案是为之创建数据镜像。

以携程机票订单数据表举例，涉及到的数据表有上千张，复杂数据的构建成本相当高，通过这个工具，用户可以为数据创建多个还原点，需要使用时一键还原即可。

实际使用场景中还需要解决日期自动平移等问题，工具都一一进行了处理，确保数据恢复后实际可用。目前支持 SQLServer 和 MySQL 两种数据库。

2.3 数据搜索服务平台

随着数据量级的提升，需要有一个服务平台能够帮助用户快速在大量数据中精确获取结果，于是我们开发了一套 API 和前端网页，用户可通过输入一个或多个关键字，获取需要的信息，前端页面的使用模式类似于我们常见的搜索引擎。

实际使用中，还有以下问题需要解决。

1) 数据排序

我们为数据设计了一套打分机制，根据该数据的创建时间，使用次数，用户反馈，来源渠道进行综合评分，根据评分高低进行排序。

2) 数据锁定

为防止同一份数据多人使用互相影响，系统提供了数据锁定&解锁的功能。

3) 搜索关键字提示

为帮助用户输入更精准的关键字条件，系统支持关键字联想推荐，辅助用户确定搜索条件。

数据问题解决了，接下来就要考虑提升工程效率，应对数据量 100 倍的增长。

三、工程效率提升

具体在工程设计时考虑以下几个方面：

1） 高效性：降低单个用例耗时，并采用分布式多并发执行方式。
2） 可扩展性：支持两个层面：测试数据和用例的快速扩展和测试页面的低成本接入扩展。
3）低维护成本：维护成本不会随着使用量的增长而线性增加，应保持基本稳定。

具体来看下服务端和前端的解决方案：

3.1 服务端：流量回放测试

服务端的测试成本主要是两块：测试报文数据和测试验证点设计。

思路回到上文中的信息爬虫，针对各种场景组合，测试模式如下：

1） 爬虫可以帮助我们获取到服务和服务内部所依赖服务的所有报文信息（请求报文+返回报文）。
2） 准备两套测试环境，一套部署基线版本，一套部署待测试版本。
3） 为还原当时的场景，将依赖服务的返回报文动态配置到Mock服务器，保证环境的稳定。
4） 使用爬虫获取到的请求报文对部署好的两套环境分别发送请求，获取到返回报文和服务

内部调用依赖服务的请求报文。

5） 进行比对分析，生成测试报告。为提升报告分析效率，需要对报告内容进行聚合，并忽略设定可忽略的部分。

6） 集成到持续集成中。每当有代码签入时，即可触发一个小规模的流量回放测试，代码发布前，触发一个大规模组合的流量回放测试，实现问题的快速反馈。

3.2 前端：图像对比测试

前端的测试难度相对服务端复杂很多，主要是体现在依赖众多，检查点不易判断。根据传统的逐个元素获取的检查方式，成本太高，且很难判断样式，字体等问题。

为了做好前端的自动化测试，我们认为需要遵循以下几点：

1） 使用上文中信息爬虫获取的各种场景组合的数据，丰富测试覆盖面
2） 将前端页面展示锁依赖的服务端数据返回进行动态 mock，保证环境稳定
3） 流程性页面功能支持 schema url 直接访问，减少测试前序耗时
4） 使用图像比对的方式进行检查点判断，降低检查成本，增加检查覆盖面
5） 监听页面发送和接收的报文数据，进行报文层的检查
6） 提升单用例的执行速度
7） 支持分布式并发执行
8） 测试报告的可读性

参照以上几点，我们设计了一套图像比对系统，使用信息爬虫获取的数据，帮助测试人员进行前端测试。

四、图像比对工具的一些问题和解决思路

4.1 提升报告准确性

1）前端的样式经常会有一些调整，有时调整一下字体，一些样式就会导致页面中的元素展示产生很大变化。如果我们想忽略这种变化，只关心展示的内容是否正确，是否有一种方式可以快速实现呢？

我们想到了图像文字识别，通过这种技术，可以直接告知用户具体的不同点，用户不必看图，减少了报告分析者的分析工作量。

2）前端有的模块是已知的不同点，比如说：广告轮播模块，针对这类部分可以设置截图时自动忽略这个部分。

3）比对结果智能分类：针对不同数据发现的同一类问题，系统会根据不同点进行自动聚合分类，只在报告中展示一个样例，使用者可自行决定是否需要查看其它数据

4）智能忽略：使用者在分析报告过程中，如果发现一些不同点是正常的，可设置忽略，系统下次对比时就会自动将这类内容标识为可忽略。

4.2 提升执行效率

1）使用无头浏览器测试方案

在我们的 SnapDiff 平台中，我们使用了 Chrome 开发团队去年新推出的 PuppeteerAPI。

Puppeteer 是一个 Nodejs 的库，支持调用 Chrome 的 API 来操纵 Web，相比较 Selenium 或是 PhantomJs，它最大的特点就是它的操作 Dom 可以完全在内存中进行模拟既在 V8 引擎中处理而不打开浏览器，而且关键是这个是 Chrome 团队在维护，会拥有更好的兼容性和前景。

2）分布式并发

平台首先会将任务分发到不同的测试服务器，然后在每台服务器上多线程并发执行，通过这种方式，整体耗时大幅减少。

4.3 降低接入成本

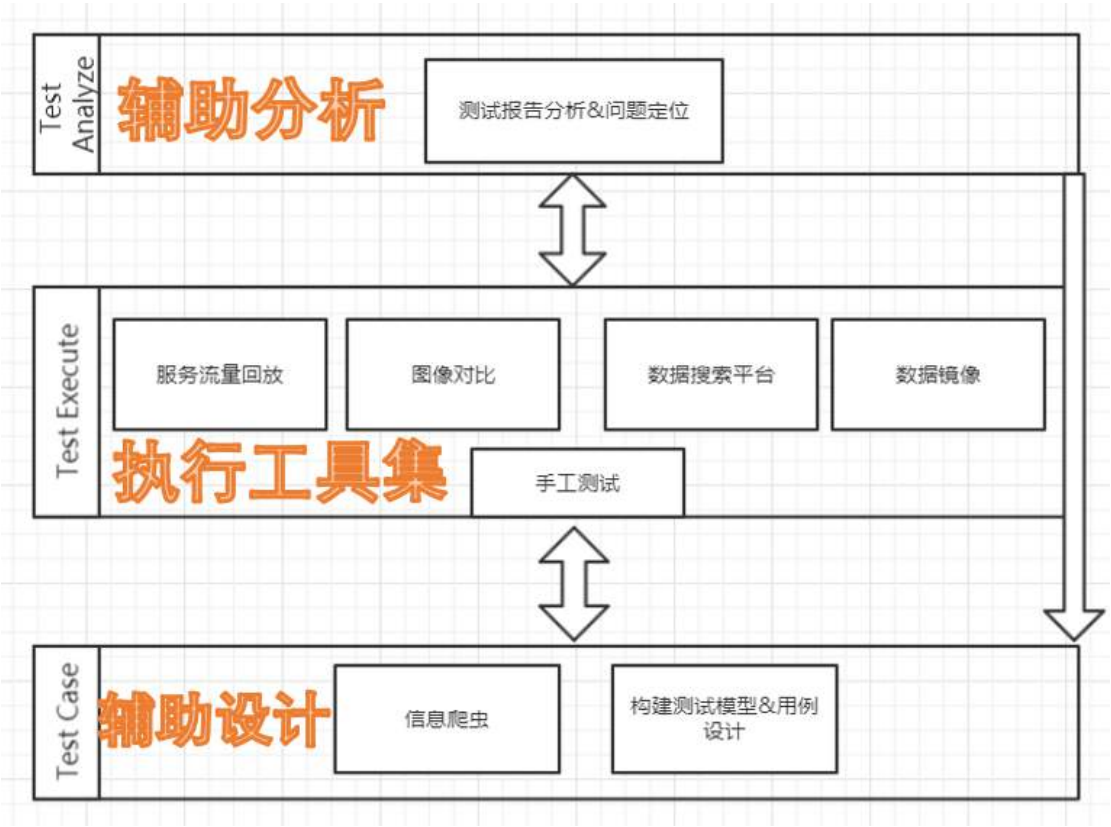
1）支持自助接入

对于非定制类的常规测试需求，用户可自助创建测试任务，通过在配置中心里配置对比页面的 url，div，运行规则，报告收件人等即可实现接入。

2）支持 Job 自助启动

测试任务的执行控制可自助操作完成

五、整体看下这套系统



不能帮助我们找到系统问题的测试框架都不完美，说明这个框架只是解决了一部分问题。所以我们在设计这套系统的时候，是抱着能够尽量多的发现问题的目的去做的。

我们希望在测试的每个阶段都能够把工具引入进来，而不是只在某一个阶段。

所以工具可以支持开发同学自测，测试同学做新功能的测试，也可以做最后的回归测试，正因如此，它能够帮助我们发现更多的系统问题，在我写这篇文章过去的 30 天内，这套系统帮助我们发现了 60+个系统 bug，未来我们还会不断优化，让这个系统发挥更大的作用。

使用这套系统，许多测试执行的工作都交给了计算机，测试人员更多的关注于构建测试数据模型，分析测试报告，定位问题发生原因。在测试设计和报告分析这两块工具更多的是起到了辅助的角色，未来我们也会在这两个领域进行更深入的探索。

一文带你了解携程第四代全链路测试系统

[作者简介]康猛，携程网站运营中心资深技术支持工程师，在互联网基础架构系统设计，后端开发，性能测试领域有多年实践经验。喜欢钻研新技术，转化研究成果，提升工作效率。

一、背景与意义

携程的应用性能测试和容量评估从技术思路上，历经了三代产品：

1.1 第一代，单接口性能测试

早期的应用性能测试主要依赖一些成熟的测试工具，如 ab、Jmeter 等，人工构造有限集合的简单报文，对测试环境或者生产环境的某一个接口，施加一定压力进行测试。用于分析在一定负载压力下，单一应用有限数量接口的 CPU、内存、响应时间等指标的性能表现。

该方法在一定程度上可以模拟较大的压力输入，但往往难以构造复杂的、高仿真生产的海量用户输入的场景，更别提需要模拟生产环境真实应用间相互依赖的场景了。

1.2 第二代，单应用生产环境压测

为了解决第一代产品中海量复杂用户输入的问题，携程开发了第二代单应用生产环境容量评估工具。

其基本原理是提高应用所在集群中某台机器的权重，使其承担远高于其他机器的真实流量负载，进而分析在该负载下，测试机应用的性能表现。

该方法在一定程度上解决了海量真实用户输入的问题，用真实的、复杂的业务场景对生产应用进行压力测试与容量评估，这对原本就有较高流量的应用也许有效，但对于正常情况下流量不大，仅在大促场景下负载爆发的应用场景，比较难以奏效。且测试机往往拥有较高的机器权重以承担较高流量，当单机故障或者冒烟，影响应用健康状态的比重就加大，收益与风险成正比。

此外，该方法通过调整单机权重使单机流量突增，整个集群的负载并没有变化，对集群上下游与中间件的负载也没有变化，这就很难测试出当前应用流量突增会不会对上下游依赖产生什么影响，应用容量的评估也很难准确。

1.3 第三代，生产环境流量回放

生产环境流量回放的基本原理是对应用所在集群中，流经一台或多台机器的流量进行拷贝，

并以实时的方式将流量的副本分发到测试机上，同时也可以将流量的镜像拷贝保存为副本文件，随时随地的对副本进行离线回放。

回放时支持对回放流量进行比例加压，比如加压 10 倍回放以模拟超高负载输入。此外，回放时还支持对回放流量进行过滤以及修改等操作，可以过滤仅回放某个固定 URL，也可以在回放时修改某接口的参数等高级功能。

该方法完美解决了海量用户输入的问题，使用真实的业务流量，不仅支持实时的 AB 流量对比，也支持离线的高压副本回放，解决了整个集群的流量输入的巨大变化，并且可以在一定程度上模拟单应用负载骤增下，上下游依赖应用的性能评估。

但是当生产链路过于复杂，使用流量回放思路去模拟也会变得复杂，且集中式的回放难以匹配超高并发，如数十万并发场景下的压测需求。

总结下来，三代技术产品思路在设计之初，都不可避免的需要回答一个基础问题：生产还是测试？

性能测试的设计初衷就是要探究生产应用的容量极限，在这一过程中，常常会伴随着应用冒烟或者进程奔溃，测试环境天然能够隔离故障，可以确保测试数据以及测试流量不会对生产业务产生影响。

但测试环境往往采用利旧设备，其设备多为过保机器，无法保证可用性，无法拟合生产机器的性能表现，这就导致测试数据往往不准，且测试环境上下游依赖数据往往缺失，无法真实模拟生产海量用户输入，可能导致测试数据与真实出现较大偏差。

生产环境天然有数据场景丰富，依赖齐全以及配置完整的优势，压测可以获取真实业务性能表现，具有极大的参考价值。基于此，我们最终选择了生产环境作为系统设计的基础环境。

但由于在生产环境做测试容易产生脏数据以及因极限测试导致的故障，设计时需要着重考虑脏数据清除以及应用测试冒烟点的识别与熔断。

正是基于对以上问题的分析，我们设计并实现了携程第四代全链路测试系统，该系统：

①采用多种方式构造测试报文，尽可能的拟合生产业务数据输入；
②使用生产环境的真实业务逻辑，拟合应用上下游与中间件依赖，能够测试出单应用负载变化场景下对上下游压力的变化；
③压测过程中实时追踪压测路径，获取压测调用链与真实核心调用链的区别对压测路径进行优化调整，使性能测试结果趋于准确；
④压测流量可渐进增长，避免瞬时集中压力对应用产生不可预知的影响；
⑤支持超高压力的并发，模拟应用在超高压力下的性能表现。

二、系统设计

2.1 系统总体设计

系统总体采用分层设计的思路，主要包括数据构造层、压测逻辑层、控制层、引擎层、应用层、中间件等。

数据构造层主要解决压测数据构造的问题，有多种数据构造的方式可供选择，支持人工构造、请求日志埋点、流量回放等；

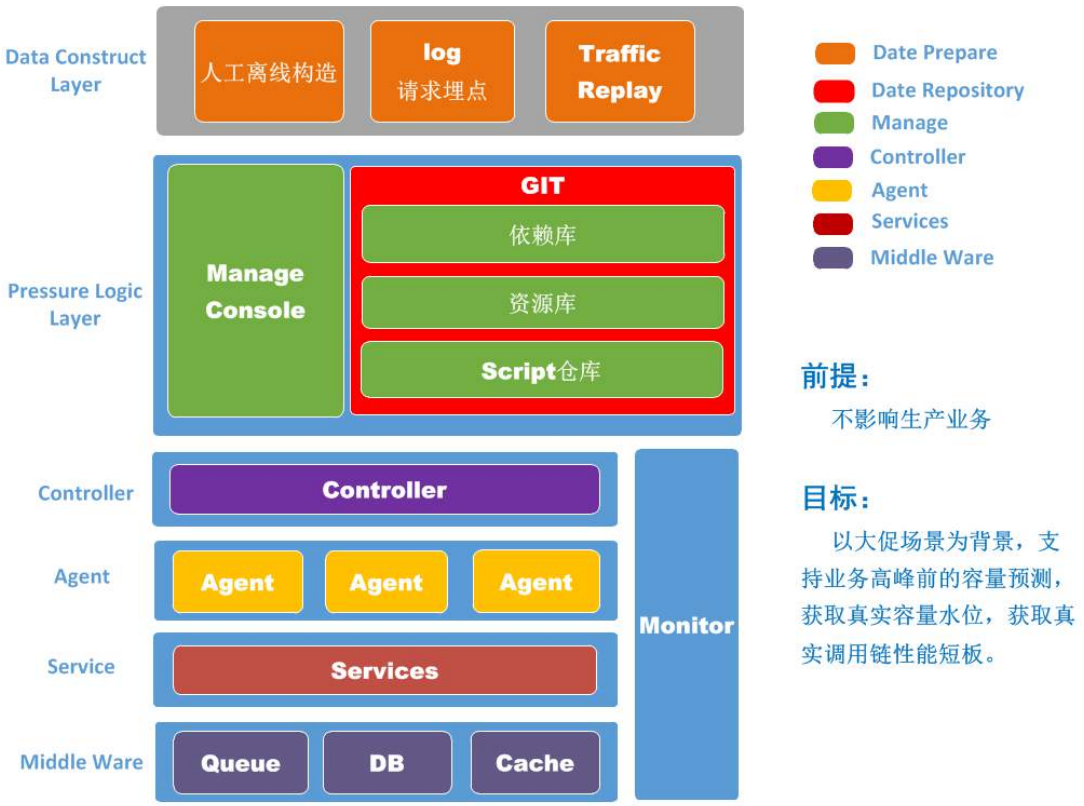
压测逻辑层主要解决压测任务设置以及压测脚本管理等问题，在这一层进行任务管理，如任务开始、终止、监控等，以及压测脚本的关联以及脚本库、依赖库的管理；

控制层的主要功能是任务下发、资源下发以及统计数据的回收等；

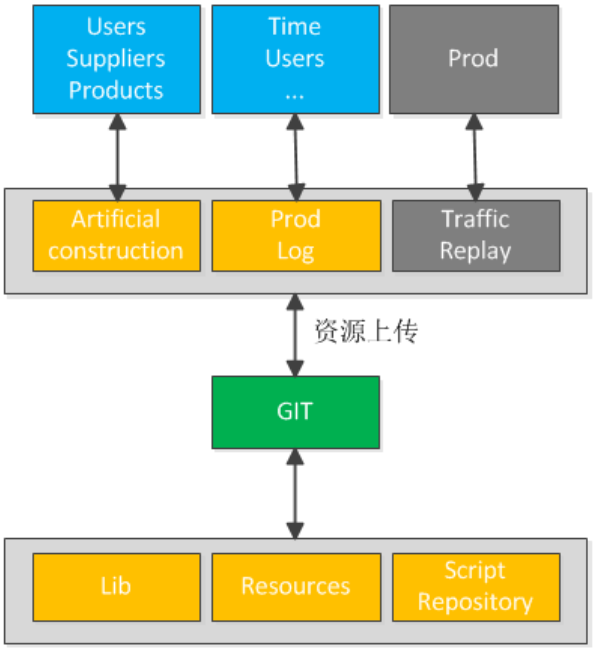
引擎层由数量众多的压测引擎构成，根据任务设置的并发开启相应数量的进程，每个进程中开启多线程的方式模拟高并发，对线上应用进行较大压力源输入；

应用层和中间件是我们生产环境真实有相互依赖关系的应用组以及其依赖的中间件；

整个系统设计主要以大促作为主要背景，模拟真实生产数据以及依赖发起较大压力的性能测试，支持进行高峰容量预测，获取真实业务容量以及核心调用链性能短板。



2.2 数据构造模块设计

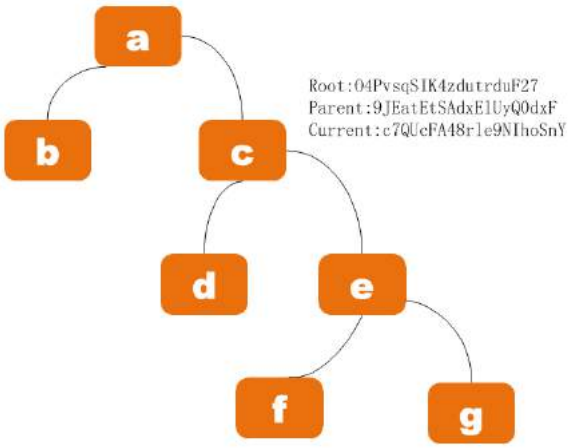


数据构造模块可以支持三种数据构造方法：人工构造、日志埋点及生产流量回放。

根据以往经验，人工构造数据适用于请求报文较简单的场景，构造报文中的用户、供应商、产品、订单等字段需要做一定的参数化替换；获取生产埋点数据可以获取海量真实用户输入，适用于较复杂的测试场景，其时间字段，用户字段等需要额外替换或随机，同理也适用于生产流量的回放，数据构造参数化的基本原则是构造尽可能分散的、足够数量的虚拟用户、商户等数据，避免热点过于集中的问题。

构造的数据作为资源依赖连同压测脚本通过 GIT 做版本管理，将压测数据与压测平台解耦，方便用户定制适用于特定场景的压测方案。

2.3 全链路请求识别



既然是全链路压测，必然涉及到全链路的请求追踪问题，我们的实现方案是，在请求的入口侧识别是否有根节点的标记。

这个标记是一个 http header，如果已经存在相应的 root 根节点标记，则透传这一标记，同时生成自己的 current 节点 ID，并将上层的节点 ID 赋予 Parent 父标记；如果未发现相应的 root 根节点标记，则当前节点生成 root 根节点标记，并在后续请求调用中透传。

如此一来，每个节点都会有 Root/Parent/Current 三种标记类型，其中 Root 根节点标记是全局唯一的，如果每个节点都可以这样自我描述，就可以画出一张基于请求粒度全网的调用拓扑图，正是通过这种方法完成全链路压测的请求识别。

2.4 压测数据清理

在生产环境做全链路压测，不可避免会产生一些脏数据，压测结束后，需要清理这些脏数据，否则可能会污染后续的订单预测，PV/UV 报表等，还可能会对生产环境的推荐算法产生误导。

通过压测前梳理生产核心链路，我们总结下来，需要清理的数据源主要有以下维度：BI 数据、UBT 数据、风控数据以及压测调用链上应用的落地数据等，一般情况下数据的清理都是基于

构造的压测数据进行的，如根据构造的用户，构造的订单等。

2.5、全链路压测平台设计

我们总结下来，全链路压测平台应包含以下基本功能：

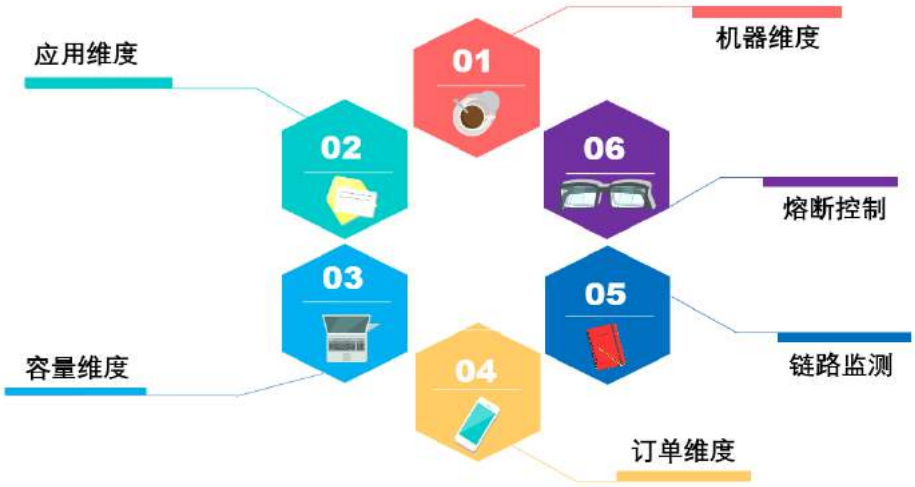
1）管理压测脚本
压测场景通过压测脚本表达，复杂的上下文逻辑以及参数化的随机可以通过脚本进行定制，平台支持查看/添加/删除/更新压测脚本，对脚本进行语法检查等，原始脚本全部存储在 GIT 上进行版本控制，平台可以添加 GIT 仓库地址批量管理脚本，也可以在脚本更新后刷新 GIT 仓库获取更新后的脚本。

2）管理压测主机
压测控制台分发压测任务到压测主机，压测主机通过开启多进程，每个进程开启多线程的方式构造足够数量的并发对目标应用进行压测。压测平台可以添加/删除压测主机，并接收压测主机回传的压测统计数据。

3）管理压测任务
通过平台可以创建/开始/终止压测任务，获取压测的真实链路，并查看本次压测的实时监控数据和统计数据等，当核心链路上出现应用冒烟，支持通过自动/人工方式终止压测任务。

4）生成压测报告
压测结束后，支持基于压测过程的统计和监控数据，自助填写压测报告，归档用于长期分析。

2.6、全链路监控设计



在生产环境做全链路压测，需要格外注意应用的实时的监控数据，我们总结，需包含以下维度：

1）机器维度
机器维度的监控数据主要包含：CPU 使用率、CPU Load、内存使用率、连接数、网络吞吐、GC 频率等指标，当出现应用内多台机器 CPU 等指标非线性变化，应格外注意任务的执行情况与应用的报错情况，及时的进行任务熔断。

2）应用维度
应用维度的监控数据主要包含应用请求量、报错量及响应时间等指标，当出现报错量的增加，或者响应时间的剧烈变化，应及时终止压测；

3）容量维度
容量维度的监控主要用于分析当前应用是否已经达到理论的容量上限，用于佐证压测效果、指导自动扩容等。

4）订单维度
订单维度的分析主要用于从订单角度进行故障发现，如果有核心链路存在监控遗漏的情况，则订单维度的监控是非常重要的补充，但是由于订单监控往往滞后于应用冒烟，所以当订单预测出现告警，则应及时有效的熔断压测任务。

5）链路检测
监控系统需要能够识别出压测请求所经过的应用调用链以及 Redis、DB、Queue 等中间件依赖，并汇总调用链上的监控数据，基本原则是“宁滥勿缺”，监控范围可以合理的扩展，避免出现监控盲区。

6）熔断控制
单一维度的监控很难反应生产应用的真实负载，全链路压测过程往往需要横跨多个维度，当出现异常点，应能够及时准确的判断问题所在，已经识别为异常点的，或者短时间不能快速定位异常情况，应及时有效的熔断压测任务，待异常分析确认结果后，决定是否继续进行压

测任务。

2.7、全链路压测重点与难点

我们结合以往生产环境进行的多次压测任务，总结生产环境进行全链路压测工作的重难点工作：

1）重点工作

①生产核心业务调用链的预梳理，并不一定要求非常精确，用于快速判断压测可能存在的异常，并进行重点监控部署，分析压测是否符合预期等。
②构造符合条件的压测数据，这直接影响压测的效果，以及后续清理数据的复杂度，也往往占用比较大的工作量，是整个任务的重点工作。
③隔离压测数据，生产环境压测不可避免会有脏数据产生，这些数据可能会影响 BI 报表与业务推荐算法，需要在压测后及时清理，这部分也是压测任务的重点工作。

2）难点工作

①压测数据的打标与识别，应用间调用需要将压测数据透传给后端，后端需要根据一定的标识识别出这是一次压测请求，并决定是否作出 Mock 等特殊的处理，不同应用间传递压测数据的方式并不完全一致，这可能涉及到一定的改造工作量。
②多部门协作保证脏数据不会对生产业务产生影响。

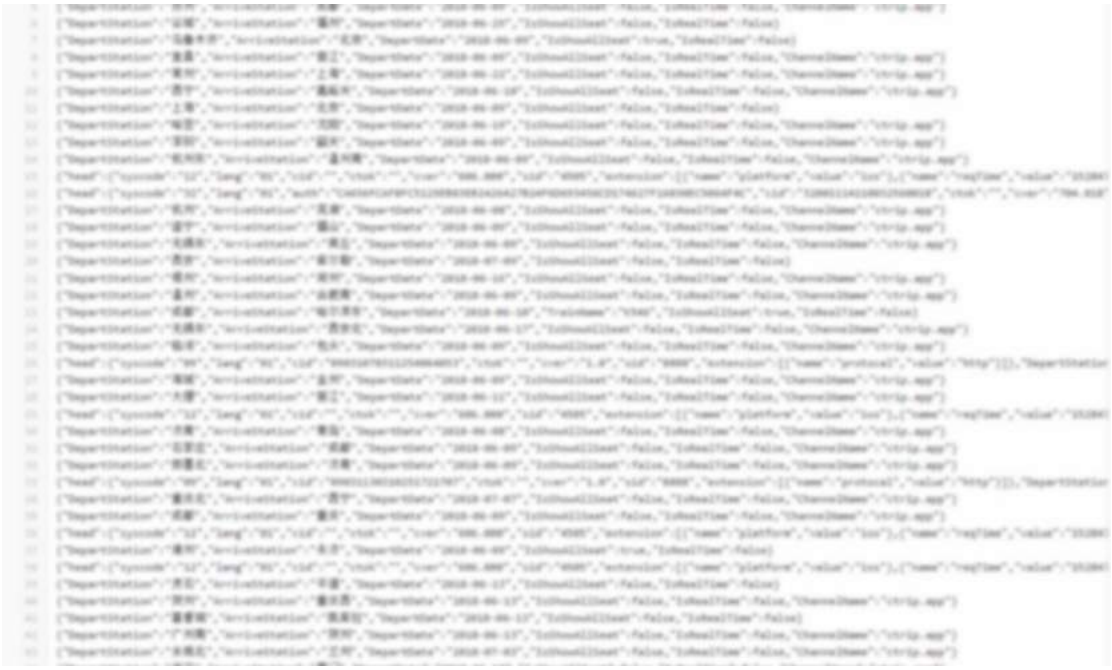
三、案例分享

3.1 压测目标应用准备

1） 链路梳理，获取压测核心链路；
2） 应用改造，识别/打标/透传压测流量；
3） 对外 Mock，外部供应商的调用，以及短信邮件等，需要统一 Mock。

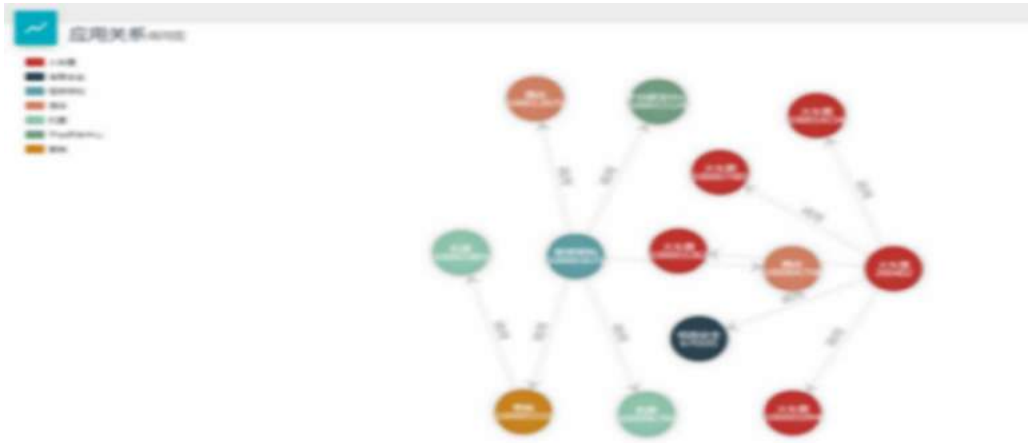
3.2 压测数据准备

结合业务的具体特点，构造相应的压测数据：



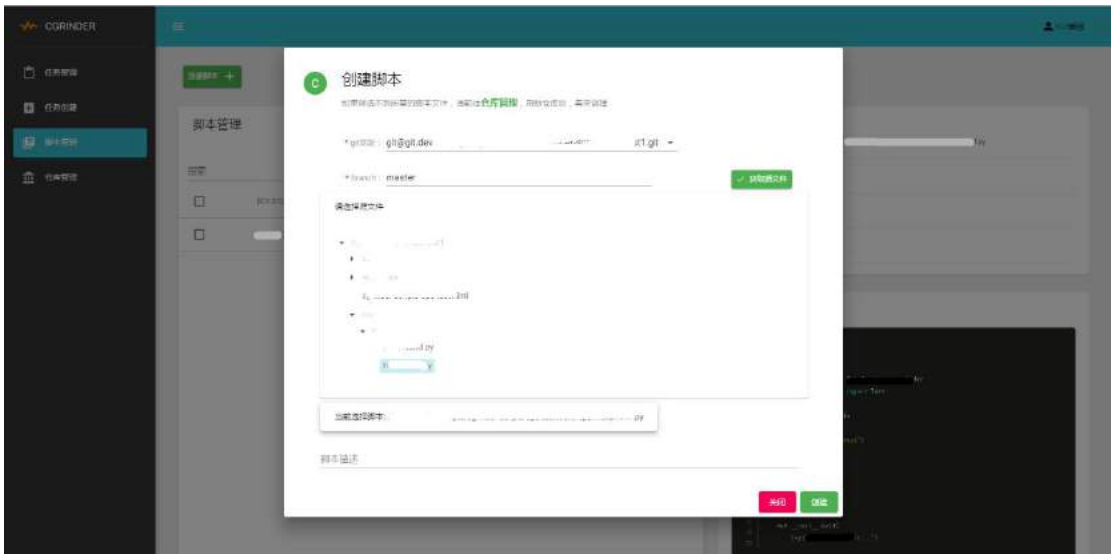
3.3 压测链路识别与抽象

压测过程中，需要能够根据打标，获取请求维度的调用链路，并以此对全链路进行监控。

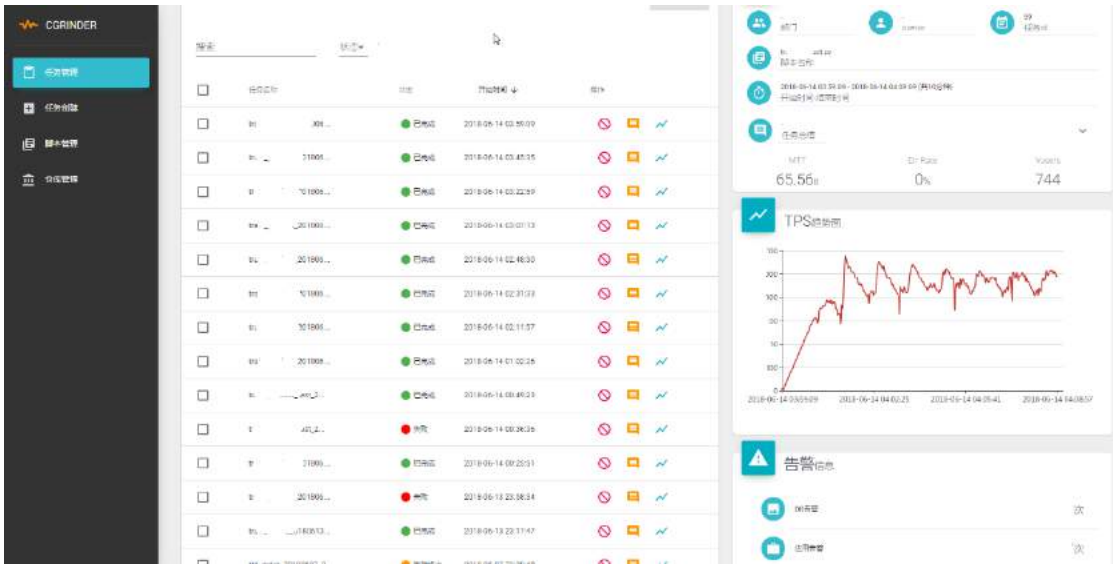


3.4 压测过程

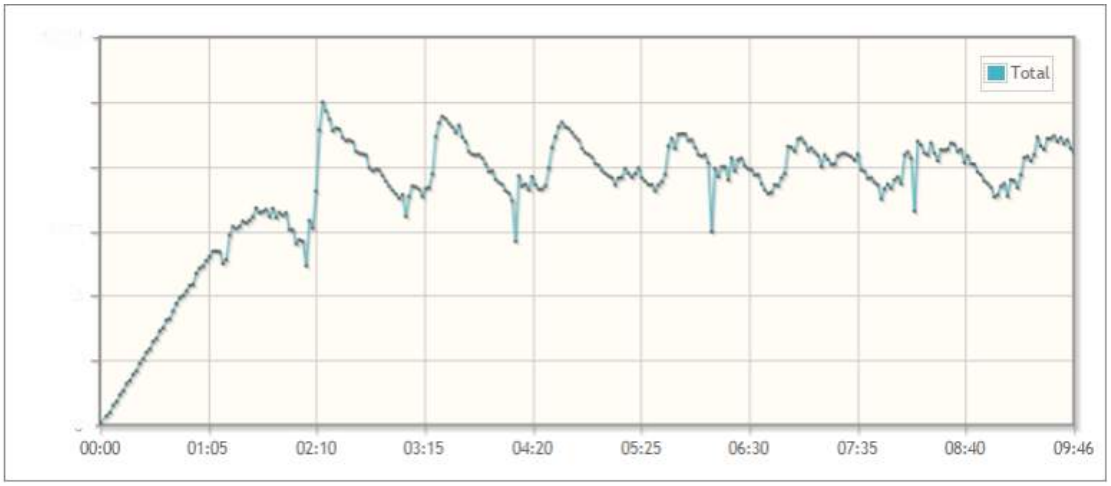
1） 创建压测场景

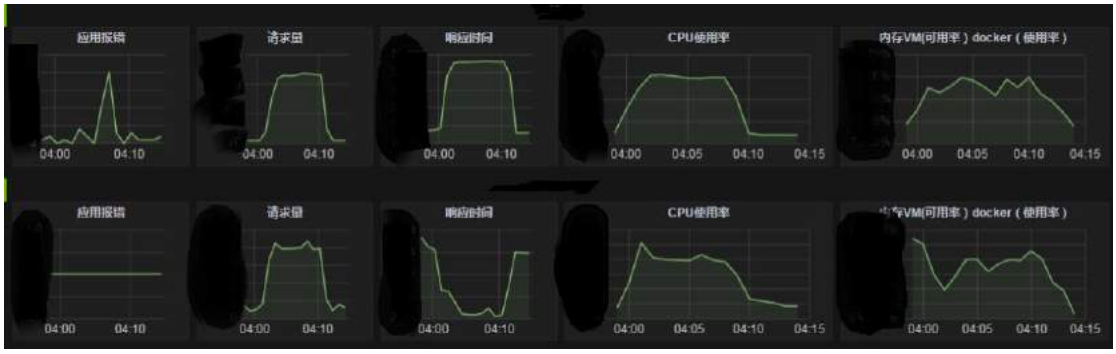


2）开始施压



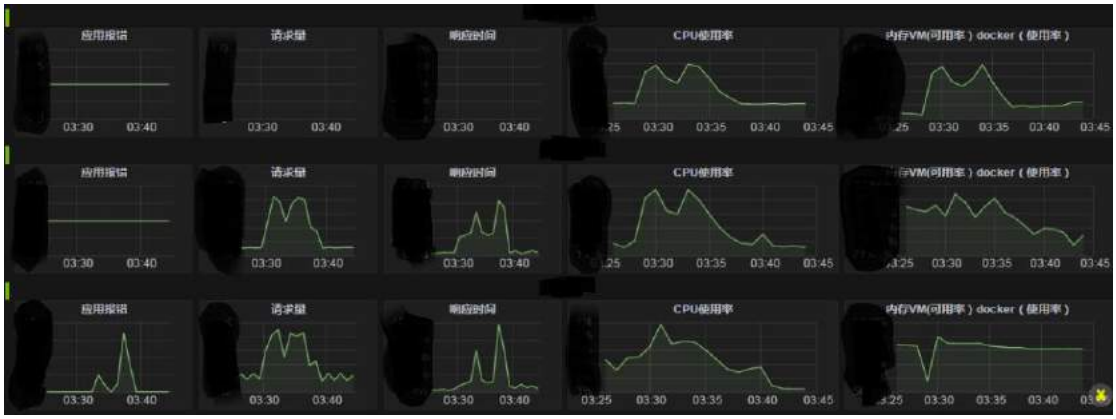
3）压测监控





4）冒烟举例

当应用出现瓶颈的一般表现是请求出现排队，应用响应变慢，此时从服务端能看到响应时间的增加以及队列长度的变化，从客户端看到的现象是请求并发的下降，当看到 TPS 指标剧烈变化，一般表示服务端此时应用已经冒烟，需要紧急介入排查冒烟点。



3.5 压测总结

至此，从开启一段压测的角度对全链路测试过程进行总结如下：

1）压测前：了解链路，准备数据；
2）压测中：下发任务，监控链路；
3）压测后：清理数据，整理报告。

四、总结与展望

当前，携程全链路压测平台已经投产，并历经了多次压测考验，这一过程中，我们也进行了多次优化改版，后续将在降低使用费力度以及监控集成方面持续发力，提供用户更便捷使用体验、更加强大的全链路性能压测产品。

携程 Hybrid 代码评审服务

[作者简介]苏玲，5 年软件配置管理及 7 年持续集成经历。曾在苏州科达科技和大众点评任资深配置管理工程师，目前为携程代码中心负责人，专注于代码相关的平台建设，致力于提高研发效率与研发质量。李海涛，目前为携程代码中心高级研发工程师，负责实现了 Gitlab sharding 和 hybrid 的代码评审服务。

一、综述

携程从 2013 年开始引入了 Gerrit 和 Gitlab 两款代码评审服务，开发团队可自行选择其一用来管理代码。

其中，Gitlab 开源项目经过四年的发展，持续优化了代码管理相关功能，还提供了很多 DevOps 的增值服务。而 Gerrit 提供的 pre-commit 的方式，也有其优势。

为了代码平台统一，携程代码中心团队在 Gitlab 上提供了类 Gerrit 的代码评审方式，推出了既有 change@Gerrit 方式又有 merge request@Gitlab 方式的 hybrid 的代码评审服务。

本文先分析两种 review 方式的优点，然后通过介绍几个场景来指出 Gitlab 中增加 change 功能的必要性，以及如何把 hybrid 的服务用到极致，并归纳出几类代码评审模式分别适合的场景，希望对开发团队有所帮助。

二、简称

本地开发每个 commit 都自动产生一个 change，没经过 review 的变更不能进入公共的 Git 仓库，这种方式我们简称为 CHANGE 。

每次 review 都要提交两个分支进行合并的 Merge Request，这种方式简称为 MR 。

三、Gerrit 与 Gitlab 的 PK

我们只比较两个平台在代码评审上的差异，然后提炼出优点。

Gerrit 提供了 pre-commit 的评审方式，通俗地说，就是没经过 review 的变更是不会进入到 Git 仓库里面。而 Gitlab 没有 pre-commit 的功能，只提供了 post-commit 的功能，也就是在同一个 Git 仓库中，任何开发人员必须向 Git 仓库推送自己的分支，然后发起 Merge Request 后才能请别人帮忙 review 代码。

Mr. Gitlab ： 你不想正式的仓库被坏代码弄脏，那你用 Gitlab fork 的方式好了，只要不接受 Merge Request，脏的代码只会存在 fork 出来的仓库里面，正式的仓库一样可以实现 pre-commit 的功能。

Mr. Gitlab：我们团队采用的是特性分支开发的分支模型，需求管理系统中新增一个需求就会自动创建一个分支，每个分支名就能看出特定的一个功能点，这个多好，想知道一个迭代周期有多少个功能要交付，看看有多少分支就行了，而且这些新建的分支就像计划任务一样提醒着开发人员。代码评审就用 Merge Request，当特性分支开发完毕，发个 Merge Requst 到 master 分支就行。

Mr. Gerrit：Gerrit 也可以为每个特性分支创建分支的，还能为特性分支上的每个 commit 建立 review 申请。另外，你们每次做 review，都得打开 Gitlab 的页面，手工发起一个 Merge Request，这个太麻烦了，大家看看 Gerrit 的做法吧，开发人员只要在自己的开发设备中，push 一个特殊的变更，Gerrit 上就能自动创建一个 change 了，根本不用人再登到 Gerrit 系统上去申请 review。

Mr. Gitlab：我不喜欢 Gerrit 对每个 commit 单独地做 review，用 MR 多好，一个分支合入另一个分支做个 review，这样虽然一次性 review 多一点，但不用在多个 changes 中跳来跳去，而且分支是可以多人共享的，我一次性可以 review 多个人的变更。

Mr. Gerrit：我觉得对单个 commit 做 review 挺好的，一个功能一个 commit，这样更容易发现问题。

Mr. Gerrit：CHANGE 的方式，很容易创建出 linear history 的分支，这样便于用 bisect 定位问题的出处。MR 行不行呢？

Mr. Gitlab：MR 当然也可以，虽然是两个分支之间发起 merge request，但是项目策略配置为 Fast-forward merge 就行啦。 ”

看了上面的讨论，我们发现有些所谓的“优点“并不明显，需要在特定场景下才拥有，有些优点则很明显。我们用表格形式做个归纳：

具体优点	是/否优点	Gerrit具备	Gitlab具备	PK结论
未经review的代码不会进到Git仓库	是	是	是（但要fork仓库，管理成本增加）	Gerrit 胜出
Push的时候自动发起了review的申请	是	是	否	Gerrit 胜出
Review一个分支，而不用review每个commit	是@某些情况	否	是	Gitlab 胜出
Review单个commit，保证review的质量	是@某些情况	是	否	Gerrit 胜出
能实现linear history	是	是	是	打平

上面 PK 的内容没有涉及 review 功能的所有特性，也不是用来说明 Gitlab 的 review 不如 Gerrit 的，而是告诉大家，某些情况下团队确实需要 Gerrit 的这种 pre-commit 的方式。我们不妨继续探讨一下，哪些情况下适合用 CHANGE 。

四、特别适合用 CHANGE 的场景

4.1 场景 1：主干分支开发的项目。

因为所有的变更都要求在第一时间提交到唯一的开发分支上，保持持续的集成，如此一来，特性分支就没必要存在了。这种情况下，用 CHANGE 最适合。

开发在本地始终基于主干分支做开发，开发完毕，直接向远端的 refs/for/主干分支 push 即可。一提交，远端自动创建一个 change，该 change 通过 review 后，其对应的 commit 就合入到主干分支；如果 review 没被通过，则变更的内容就不会进入到主干分支。

4.2 场景 2：特别重视代码质量的团队。

此类团队不允许质量低劣的 commit 存在仓库中，也不想用 fork 的方式，因为 fork 的方式不够简洁和直接，fork 出去的仓库需要经常 fetch 原仓库，需要维护多个 remote 。

4.3 场景 3：有较多开发新手的团队。

新手提交的变更先经过 review 然后才能进入到 Git 仓库。

五、CHANGE 和 MR 同时使用的场景

给 Gitlab 引入 CHANGE，很自然地会想起一些问题“难道仅仅是为了让 Gerrit 顺利下线，我们才把 CHANGE 引入到 gitlab 吗？”或者“现有采用 Gitlab MR 的团队是否也能享受 CHANGE 带来的好处呢？”

静下来想一想，还真的存在下面的场景，如果同时使用 CHANGE 和 MR，可以有效提高代码评审的效果。

团队特征：

1） 采用特性分支开发模式，每个功能对应一个分支。
2） 特性分支开发完毕，合入 master 分支后发布。
3） 有不少开发的新手。

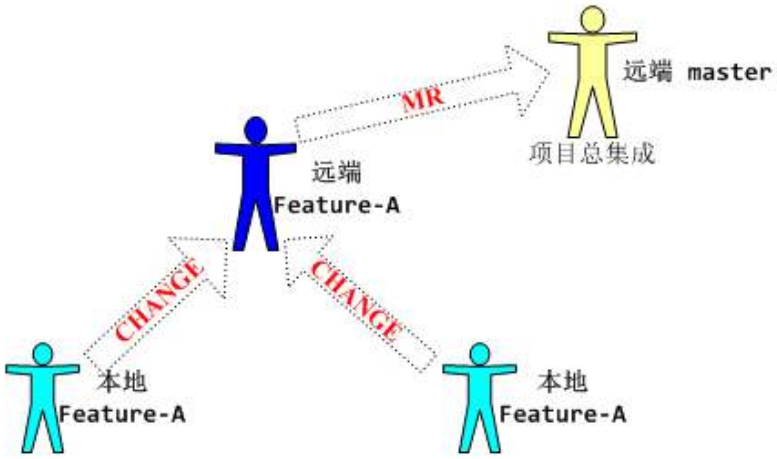
开发流程：

1） 甲、乙、丙三人同时负责 A 功能的研发，共同分享 feature-A 分支。
2） 甲负责 review 乙、丙的代码。
3） Feature-A 分支设为保护分支，变更前需 review 代码。
4） 乙和丙在本地基于 Feature-A 开发，自测完成后 push 到 refs/for/Feature-A。
5） 甲在 gitlab 上 review 后，乙、丙的变更被合入到 Feature-A 。
6） 然后甲向 master 发起了一个 Merge Request。
7） 由上一级集成人员 review 后，最终 Feature-A 被合入到了 master 。

“CHANGE 和 MR 同时使用”，比起“只使用 MR”，优点很明显：

以前只有 MR 的情况下，如果甲要 review 乙和丙的代码，除了结对编程外，还有一种方式，把 Feature-A 保护起来，不允许直接 push，然后，甲和乙基于 Feature-A 创建新的分支，开发完成后再 Feature-A 向发起 MR 。而“CHANGE 和 MR 同时使用”，可以省掉额外新建分支的麻烦，且省掉了发起 MR 的麻烦。

我们不妨用下面的简图呈现 CHANGE 和 MR 的关系：



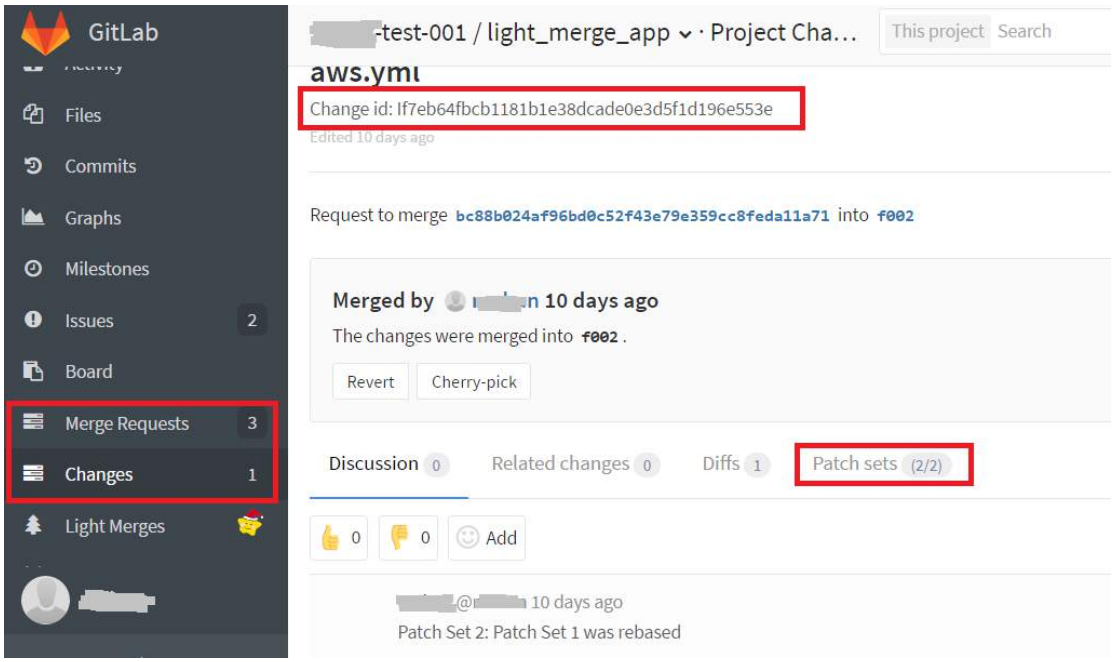
如此一来，在不增加远端仓库分支的情况下，基层 review 人员依赖 CHANGE，保证每个 commit 的代码质量，从而确保特性分支的质量；另一方面，主干分支的集成人员借助 MR，无需在个人环境上做分支的集成，待 review 人员完成评审后，他们就能一次性地在 Gitlab 界面上把特性分支合入到主干分支，从而保证 master 主干分支能被高效地集成。

六、Hybrid 代码评审服务的模样

1）尽可能保留 Gerrit 本地操作的方式：
git push origin HEAD:refs/for/目标分支 。

2）项目首页，提供 CHANGE 的 clone 方式：

3）CHANGE 和 MR 并存：



七、代码评审模式的类型与适用场景

MR 和 CHANGE 都提供代码评审的功能，并且携程 Gitlab 平台同时支持这两种功能。非常自然地，大家极有可能会问："什么时候该用 MR，什么时候用 CHANGE，什么时候同时使用，什么时候可以都不用呢？"

我们用下表做个归纳：

序号	模式	适合的场景
1	CHANGE	同时满足以下几个条件： • 产品有质量要求。 • 主干开发主干发布。
2	CHANGE+MR	同时满足以下几个条件： • 产品有质量要求。 • 非“主干开发主干发布”。 • 不少能力较弱的开发人员。
3	MR	同时满足以下几个条件： • 产品有质量要求。 • 非“主干开发主干发布”。 • 很少能力较弱的开发人员。
4	无代码评审	对质量没什么要求的项目。或者 每个成员都是神一样的人，永远交付高质量的代码。

结合上面的分析与归纳，大家不难发现：这套既有 CHANGE 又有 MR 的 hybrid 的代码评审服务，为那些分支策略多样性的公司的代码评审提供了灵活性和高效性。

作为一款代码评审服务，它不仅仅适合携程，同样也适合与携程有类似特征的公司。

八、结束语

hybrid 代码评审服务的推出，预示着携程即将全面下线 Gerrit，统一代码平台为 Gitlab，期待携程的研发能从中获益，从而进一步提高 review 效率。

我们也希望 hybrid 的方式，能给代码平台的建设提供一种新的思路。当崭新的事物出现的时候，我们不妨听听用户的心声，看看如何能把旧事物里面好的一些设计理念有机地结合到新事物中。如果 hybrid 得非常巧妙的话，很可能会有非常棒的效果。

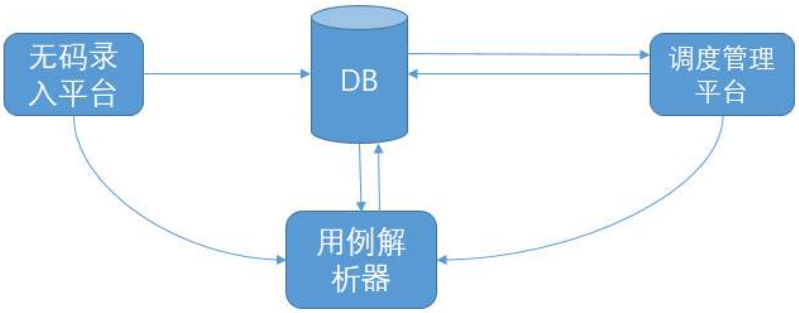
携程用户中心接口自动化实践

[作者简介]高峰，携程用户平台研发部高级测试工程师，负责业务功能测试和相关测试工具的开发。目前主要负责的测试产品有用户积分系统、用户画像、首页、OCR 及相关项目。

为了适应敏捷开发模式，同时解决接口测试遇到的问题，如冒烟测试不及时、人工回归成本高、自动化用例编写维护成本高、case 依赖数据破坏等等，携程用户平台研发部做了接口自动化测试项目。

本文将从接口自动化的痛点解决实践方面，分享携程用户中心的接口自动化案例，希望能给遇到同样问题的小伙伴一些启发和借鉴。

一、整体介绍



携程接口自动化平台由三部分组成：无码录入平台、用例解析器、任务调度执行平台，数据均由 DB 存储。

无码录入平台：可以进行用例的编写、调试，实现接口服务管理、用例管理、接口用例数量图表展示；

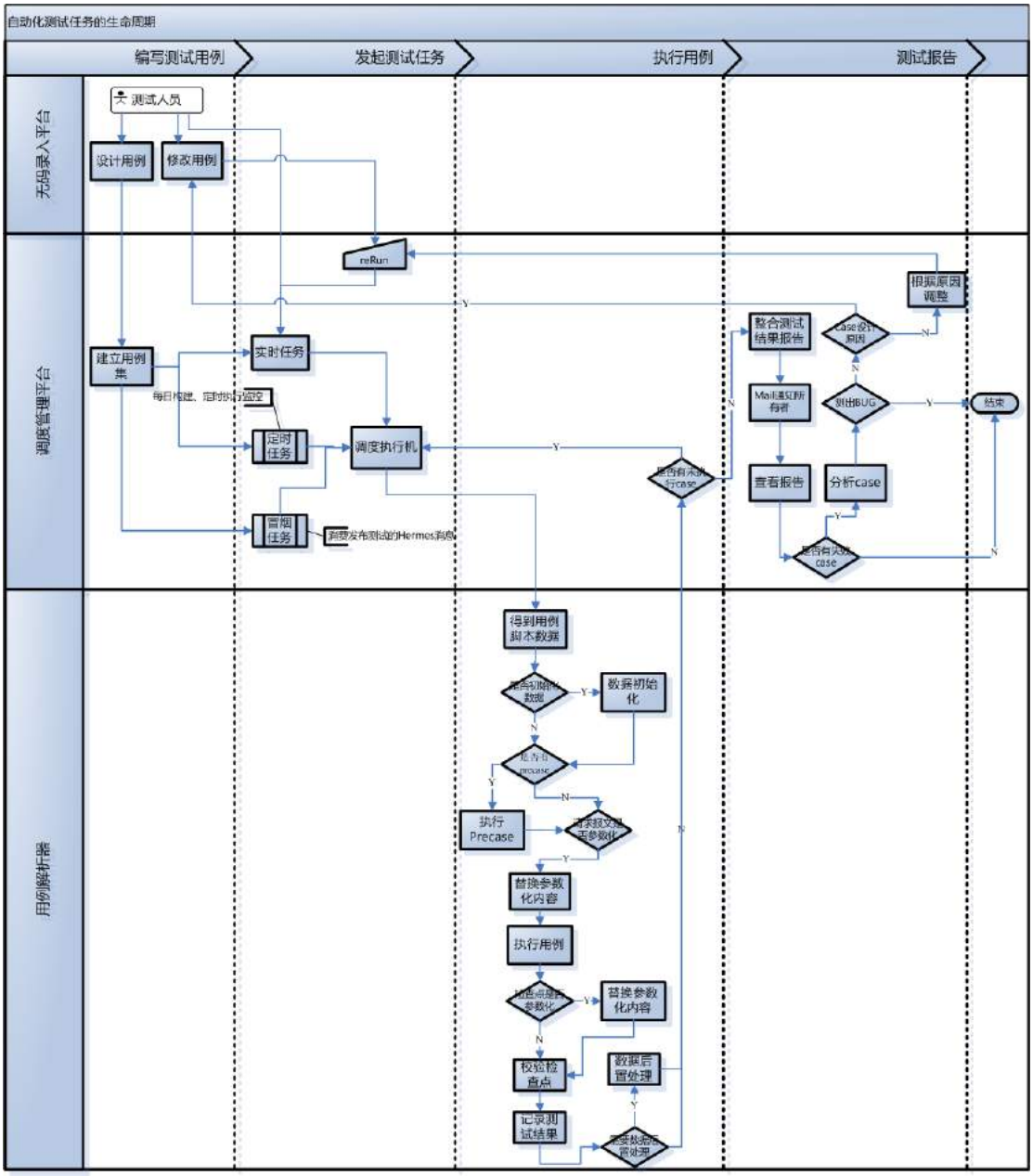
用例解析器：无码录入用例后执行脚本和测试数据分离，用例解析器执行一系列的动作包括数据初始化、参数化数据替换、发送请求、检查点检验、日志报告记录、用例后置处理等；

调度管理平台：用例集管理、任务管理、执行机调度、测试报告管理，测试任务发起后，调度管理平台任务要求选择满足条件的执行机、调用用例解析器执行自动化用例并输出测试报告、邮件系统将测试报告通知任务归属人员，测试人员进行失败 case 分析；

DB：测试数据都存在 db 中，且测试脚本和数据相互分离，db 存储解决了以往的 excel 参数化难以管理的难题；

二、生命周期

了解了各个组成部分的作用后，我们来看下平台中一个自动化测试任务的生命周期。



如图我们可以清晰地看出自动化测试的生命周期由四个阶段构成，细看下每个阶段的中间过程。

2.1 编写测试用例

执行自动化测试任务的前提是需要有自动化用例，自动化用例的编写工作不局限于自动化测试人员，功能测试人员在手工测试的同时可以完成。在无码自动化录入平台上 web 化操作，所有数据存储 DB。用例完成后可以在调度管理平台创建用例集。

2.2 发起测试任务

调度管理系统目前支持三种测试任务，实时任务、定时任务、冒烟任务。

实时任务是手工创建任务，选定空闲的执行机随即执行；

定时任务是每天定时触发，调度选定的执行机用来监控测试代码的稳定性；

冒烟任务是消费发布系统发送的 Hermes 消息来触发自动化测试任务，当应用在测试环境更新立即触发冒烟。

2.3 执行测试

调度管理系统分配好执行机给自动化测试任务后，任务中每个 case 按照调度规则调用用例解析器执行，解析器依次初始化测试脚本数据、递归依赖的 precase、替换参数化的内容、执行校验检查点、记录报告日志、数据后置处理，循环执行直到测试任务中所有 case 执行完成。

2.4 测试报告

用例全部执行结束后，调度管理平台的日志系统整合结果生成在线测试报告，mail 模块将在线报告的链接发送给相关人员；到这里还没结束，测试人员查看并分析测试报告，也支持自动分析。失败 case 修复后可直接 reRun。

三、实践解决方案

上述的自动化测试项目并非一蹴而就，在此之前经历了 2 个大版本的迭代。

初版测试脚本和数据全部用代码维护，第二版 excel 数据驱动、测试脚本与数据隔离，测试工程师在使用前两个版本时遇到些问题，如：测试用例需要 code 能力的测试工程师、写自动化用例需要额外的用例编写时间、用例编写维护麻烦、excel 数据驱动多人操作难管理、测试用例执行过再次执行通常会因为数据原因失败等等。

以接口自动化测试项目投入产出比为核心评价指标，并针对这些问题我们给出了以下解决方案。

3.1 降低自动化用例编写难度

1）无码自动化

通常编写自动化测试用例的工程师都需要有 code 的能力，这在一定程度上对功能测试人员设定了门槛。若招聘专门的自动化工程师做自动化测试用例的编写，一方面会使企业成本提高，另一方面专门的自动化测试人员，可能没有功能测试工程师对系统功能的熟悉程度高，而导致自动化测试用例覆盖率低。

为解决这个问题，我们给出的解决方案是无码自动化，不需要用例编写人员具备 code 能力，

你会简单的使用市场上流行的接口测试工具（如：jmeter、postman），就会使用我们的无码自动化平台。

测试用例的主体就是请求报文的编写和预期结果的设置。

2）用例可复制

同一个接口的用例通常大部分是入参的不同，我们提供了用例复制功能，测试用例的一些要素包括参数化内容不用每次都手工填写，只要修改不同点，这样达到减少用例编写的时间的目的；

3）自动生成用例

一些固定的校验某些入参空值的用例，我们提供了用例批量生成工具；

3.2 确保自动化用例的健壮性

测试用例执行过一次，就因为依赖环境、数据的变化导致不能重复使用，那是不可取的。为了保证测试用例的健壮性就必须保证测试数据的鲜活度。

1）测试数据参数化

同一个用例在不同环境执行，不同环境可以设置不同的参数化数据；
请求报文和预期结果都可以进行参数；
参数化的 value 可以通过 sql 取 DB 中的值；

2）数据初始化

可以通过操作 db、redis 将用例依赖数据初始化；

3）定义 precase

当一个接口的执行参数依赖另外一个接口的返回，我们是无法通过操作 db 或 redis 获得 case 的入参数的，我们可以在本来的 case（称为 case A）中定义一个 precase(称为 case B)，case A 会自动获取 caseB 的结果进行参数化替换，当然如果依赖的接口较多，我们可以设定 case C、case D、设置更多；如此也覆盖了复杂调用场景的用例；

4）用例后置处理

case 执行结束后，可以通过操作 db、redis 将数据还原；

5）提供常用函数 API

case 的请求报文入参有时候需要一些特定数据，如随机数、uuid、加密签名算法、当前时间

等，定义 API 的关键字 FUN_APIName 并以参数化的形式输入；

3.3 手工测试完成自动化用例即完成

我们在用例编写页加入了实时调试功能，这个功能不仅可以在自动化用例执行有问题时进行调试，而且可以利用这个功能进行手工测试，保存每个手工测试用例测试完成即自动化测试用例编写完成，节省了额外编写自动化用例的时间。

3.4 失败自动分析

自动化任务执行后查看报告，或多或少有些失败的 case，分析失败 case 通常会占据测试工程师大部分时间，平台提供了自动化分析功能，可以自动分类出一部分非接口功能问题，这样可以节省分析时间让测试工程师把关注的重点放在接口功能上；

四、其他

4.1 三种用例类型

自动化测试项目实践运行过程中，通用的校验响应报文的方式（caseType-O）满足不了 100%的接口测试需求，我们又新增支持另外两种 case 类型。

1） caseType-N

与 O 型不一样的是这种类型面向的为新增、删除类功能的接口，这种类型的接口通常响应报文只会返回 success 或 fail，只校验响应报文显示是不行，N 型 caseType 可以对 db 数据变化进行校验；

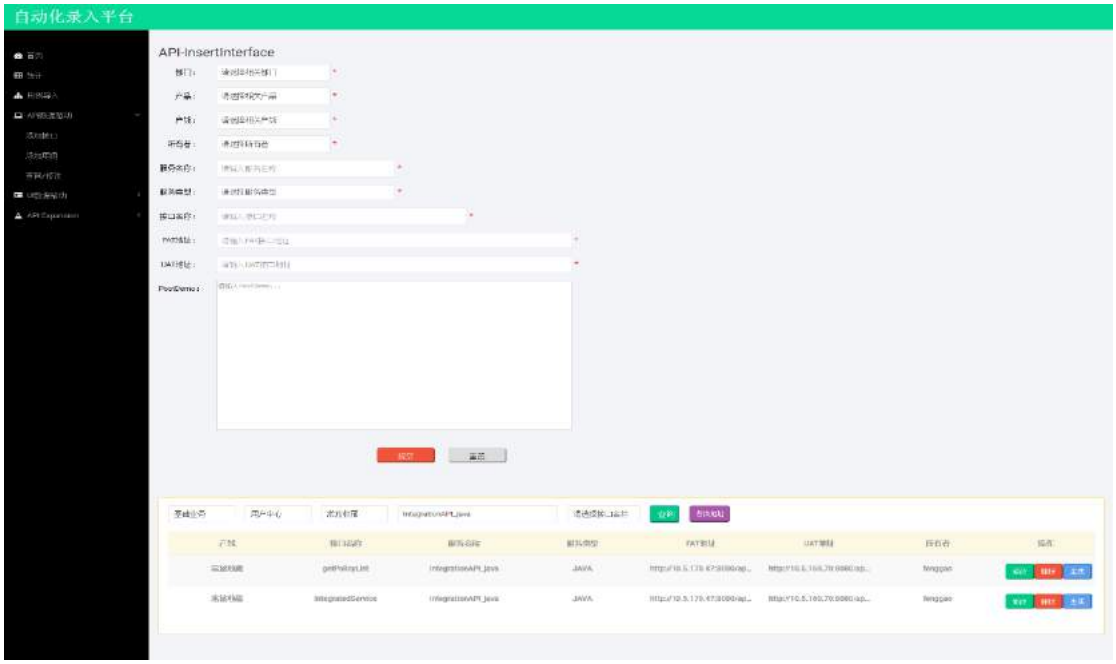
2）compareCase

顾名思义是比较型 case，现在试用的是两种场景：一种是聚合型的接口，比如我们用户中心的订单查询的接口，该接口下发的数据来源基本都是调用 BU 的订单接口下发的内容，通过比较两接口的响应关键字段值来达到校验 case 预期值，compareCase 就是为第一种类型设计；第二种是版本比较，可以将同一 case 的两个不同测试版本进行比较校验预期值；

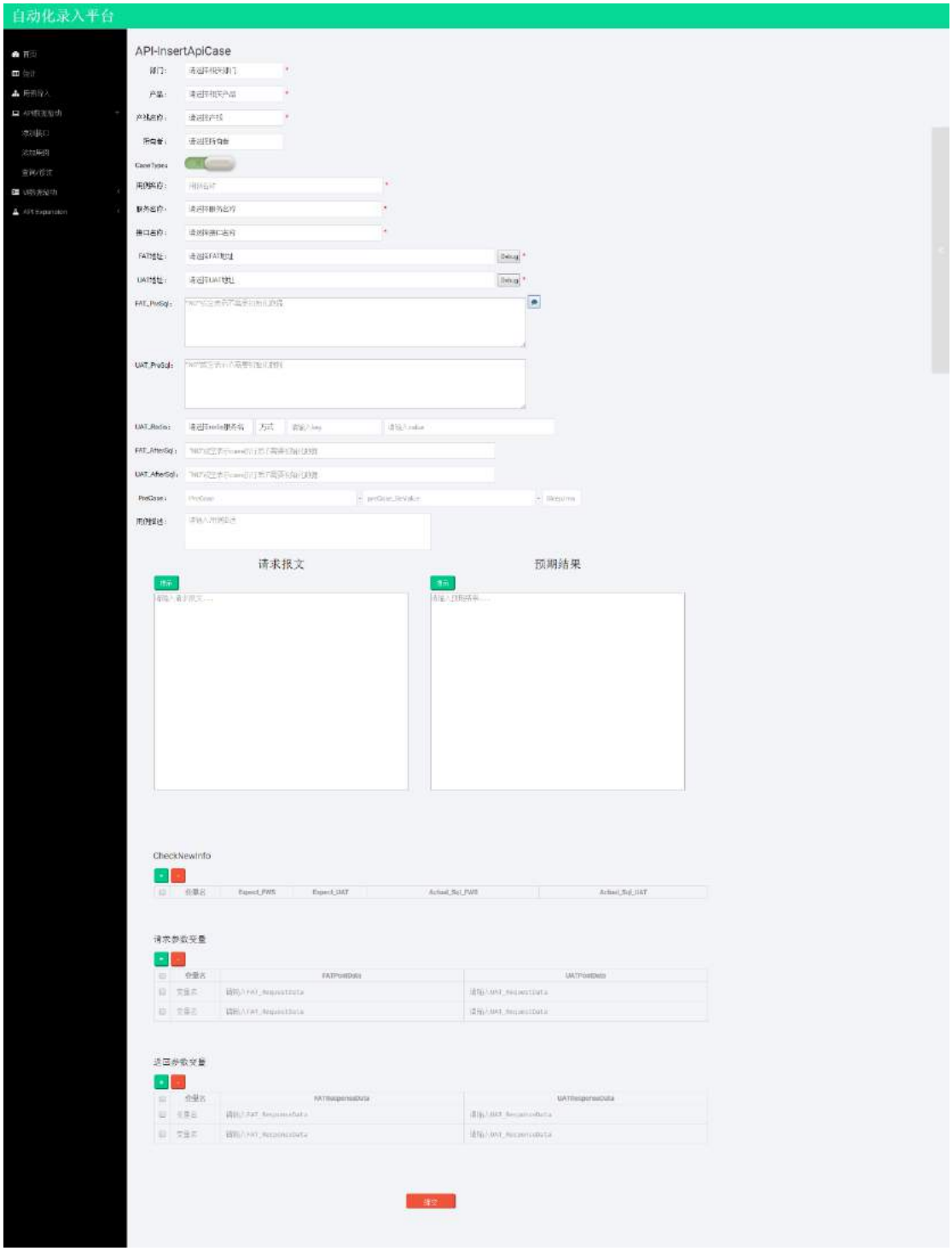
4.2 平台部分页面展示

无码录入平台的首页以柱状图展示了接口数量和自动化用例数量，按类型、业务区分一目了然目前的用例情况；

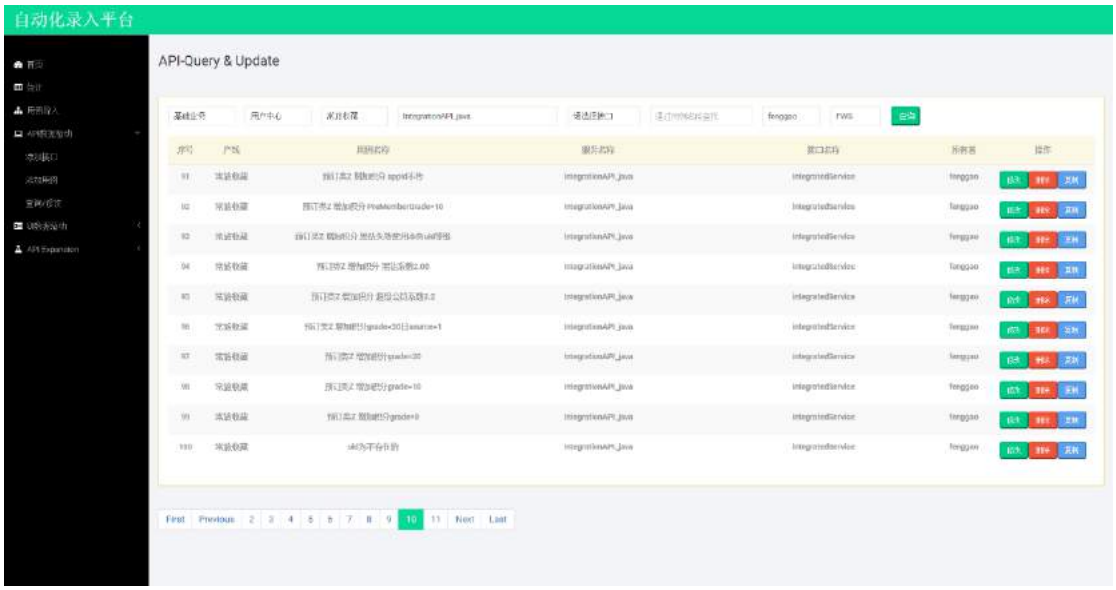
1）接口管理页面。可以增删改查接口信息，有一键生成特殊自动化用例功能；



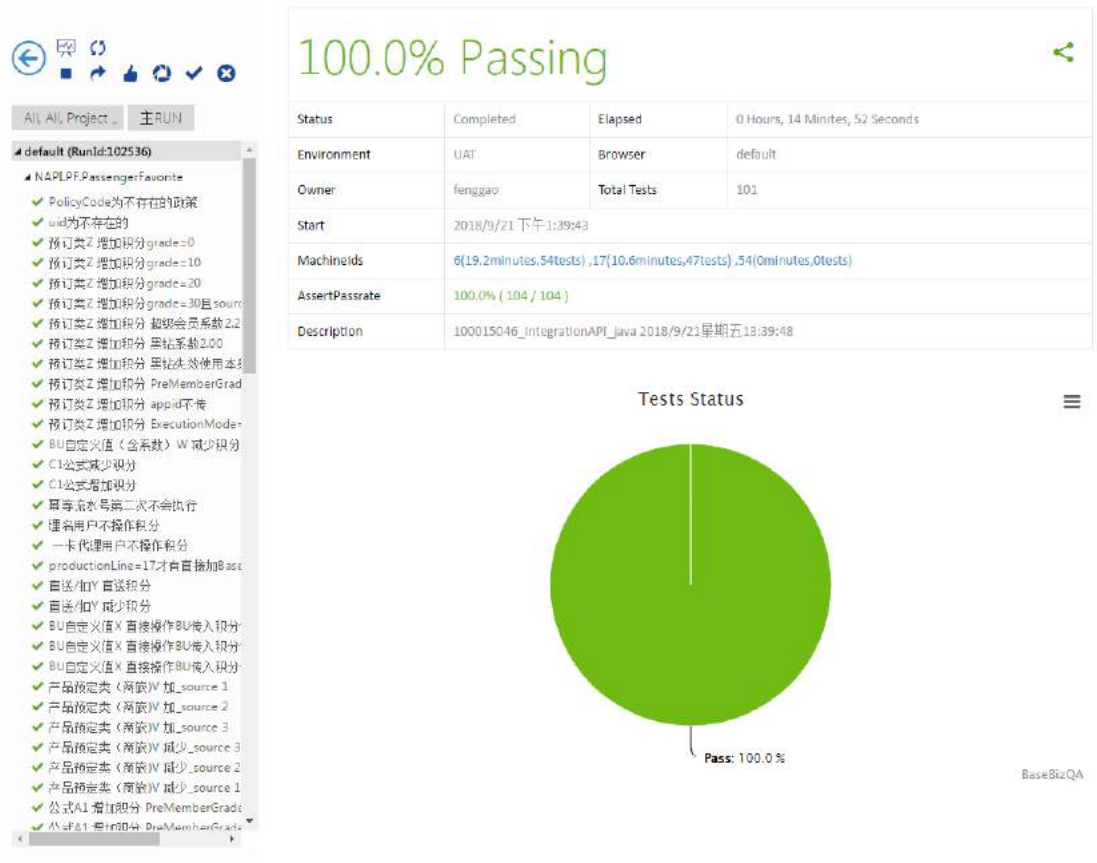
2）测试用例编写页面，无需编码，参数化，数据初始化、precase、校验点、debug 等都在该页面设置；

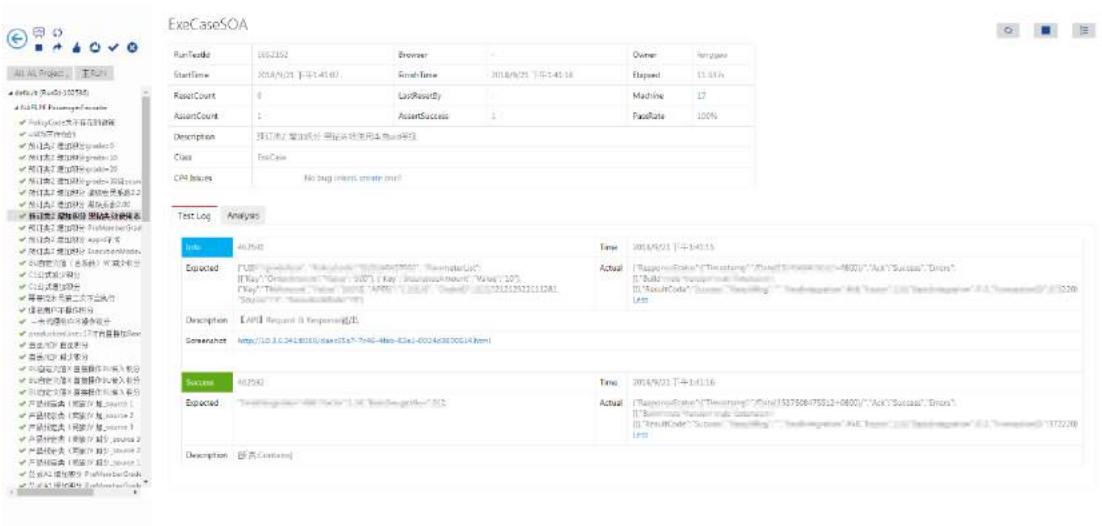


3）用例查询管理页面，可以查询、复制、删除用例；

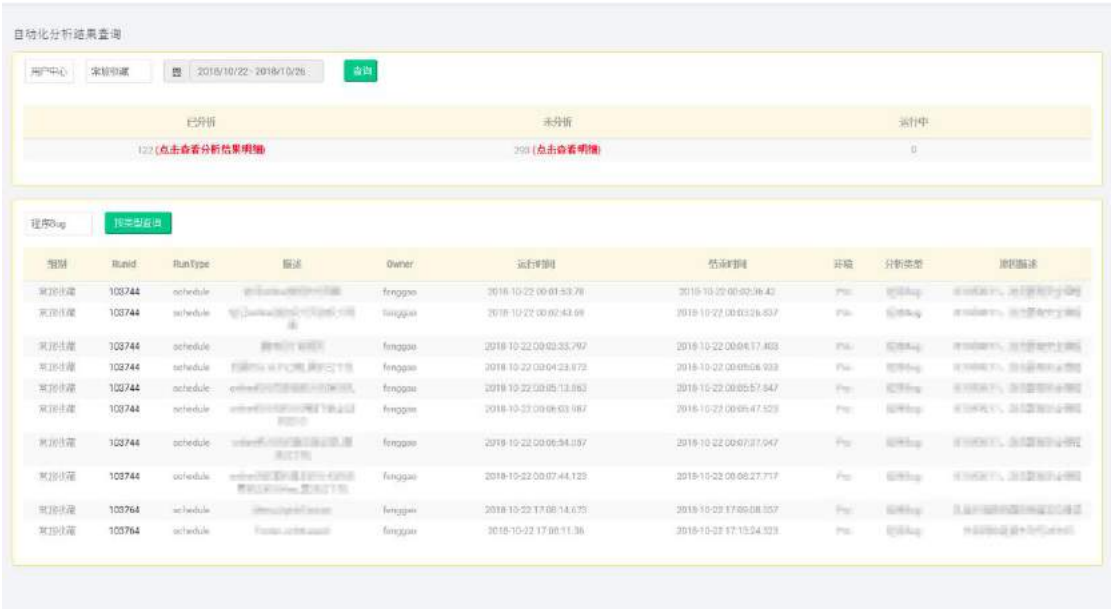


4）自动化结果在线报告页，展示一次测试任务的测试报告，包括通过率、执行记录、用例集信息、分析功能；





5）自动化分析结果查询页，失败用例的分析结果整体查询；



五、总结

以上为用户中心接口自动化测试项目的实践介绍，总结如下：

目前用户中心的接口自动化已稳定运行两年半，为了更好的满足用户需求，仍在持续迭代中。

运维篇

AIOps 在携程的践行

[作者简介]徐新龙，携程技术保障中心应用管理团队高级工程师，负责多个 AIOps 项目的设计与研发。信号处理专业硕士毕业，对人工智能、机器学习、神经网络及数学有浓厚的兴趣，对人工智能技术结合运维场景的实践有深入研究。

随着人工智能时代的到来，携程生产环境运维进入了新的运维时代——AIOps。通过两年多时间的技术投入与实践，AIOps 在效率提升、可用性保障、成本优化等运维场景取得了显著的成果。

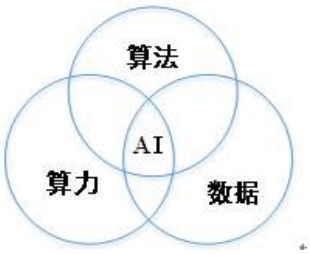
本文选取了几种典型的运维场景对 AIOps 在携程的践行展开了介绍，首先让我们从概念认识下 AIOps。

一、AIOps 的概念

2016 年 3 月，AlphaGo 与围棋世界冠军、职业九段棋手李世石进行围棋人机大战，以 4 比 1 的总比分获胜，人工智能技术又重新进入大众视野，备受关注。

通常人工智能技术分为“弱人工智能”和“强人工智能”，其中弱人工智能是能够和人一样，甚至比人更好地执行特定任务的技术；强人工智除了更好的执行特定任务之外，有着人类所有的感知和理性思考(甚至比人类更多)。目前我们提到的人工智能技术都是弱人工智能。

人工智能技术从概念落地到实践需要具备三个必要条件，即算法、算力和数据。



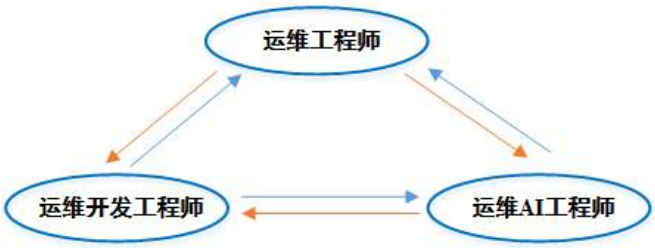
其中算法和算力，随着机器学习算法理论和工程技术体系的成熟，已经得到很好的攻克，关键点在于数据和应用场景。有过算法经验的人应该都知道，算法效果的上限取决于数据质量。

运维行业因为积累了大量生产环境数据，其中包括各种指标的监控数据、告警数据等，特别是对于携程这样体量庞大的网站，这些数据每分钟正以惊人的速度在不断增长，具备了 AI 技术落地得天独厚的条件。

2016 年 Gartner 报告中提出了 AIOps 概念，也就是 Algorithmic IT Operations—基于算法的 IT 运维，主要指用大数据、机器学习驱动自动化、服务台、监控场景下的能力提升。

从某种意义上讲，AIOps 也可以称之为数据运维。

AIOps 人员组成上由三大主体构成，即运维工程师、运维开发工程师和运维 AI 工程师。



运维工程师作为最前线的人员，对生产运维场景了如指掌，提炼出相应的需求点；运维开发工程师则是协助运维工程师，将其提出的需求加以自动化实现，从而避免重复性手动劳动，解放双手；随着运维规模和场景变的越来越复杂，仅仅靠以往的经验知识已经难以应对，于是 AIOps 应运而生，用基于统计、学习替代传统的基于规则，而代表这种生产力的就是运维 AI 工程师。

二、携程 AIOps 几种典型应用场景介绍

目前 AIOps 在业界还属于应用探索阶段，其比较成熟的场景主要包括以下两个领域：

可用性保障：异常指标检测、故障智能诊断、故障预测、故障自动修复等；
成本优化：容量规划、资源利用率提升、性能优化等。

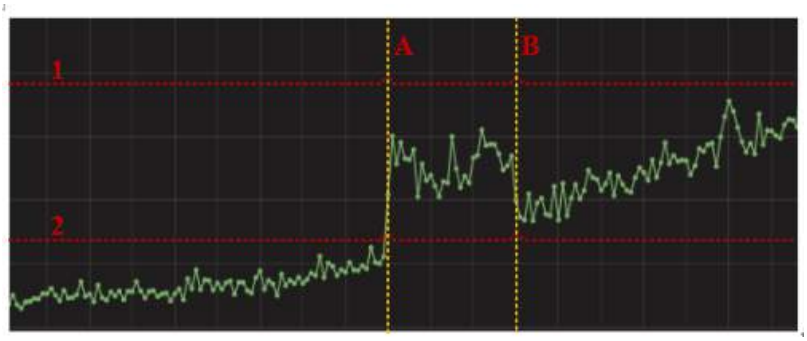
接下来介绍下携程在这两个领域下的部分典型场景，以及相关算法简介；

2.1 应用异常指标检测

应用的埋点信息是最能反映应用健康状况的指标，这些埋点指标被监控系统采集并设置一定的告警规则(一般为固定的阈值)，一旦某个指标连续达到设定的阈值时，就会通知相应的负责人处理。传统基于固定阈值的告警技术存在以下的问题：

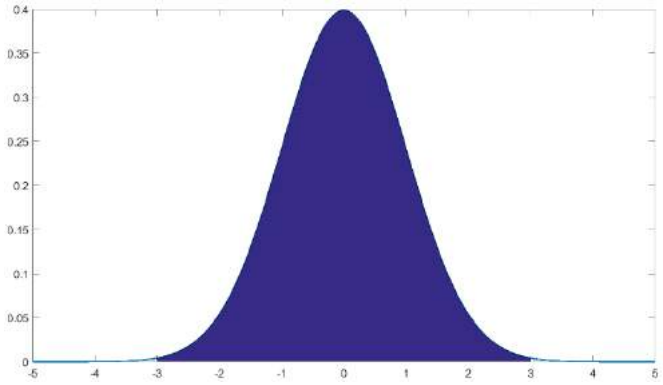
1）固定阈值的设定依赖人为经验，和实际情况可能存在较大的偏差；
2）一旦设置不合理，即存在大量的漏报误报；
3）无法检测异常序列的冒烟特征，例如缓慢爬升等；
4）携程应用成千上万，数量众多，为每个应用维护固定阈值，成本极高；
5）牺牲告警及时性换取有效性，出现应用告警滞后于订单告警。

下图给出某个应用的一段监控指标(已经脱敏)，其中 A、B 两条换黄线之间存在异常，如采用固定阈值就可能造成：使用红线 2 作为阈值可以发现异常点，但造成了大量的误报(黄线 B 的有部分)；使用红线 1 作为阈值虽然减少了误告，但无法检测出异常时序。



AB 之间存在异常的某段时间序列

为了解决以上存在的问题，通过对时间序列建立 ARMA 模型，利用时序的统计特性、时频分析，再结合工业常用的 3sigma 准则确定动态阈值，自动识别时序中的异常点。同时根据告警时序之间的相关性、专家知识库等区分告警类型。相比传统告警规则，智能告警系统在准确率和召回率都有巨大的提升。针对应用异常指标检测这种场景，抽取一定的样本统计，在基于专家经验标注下的准确率可达到 90%以上，召回率接近 100%。



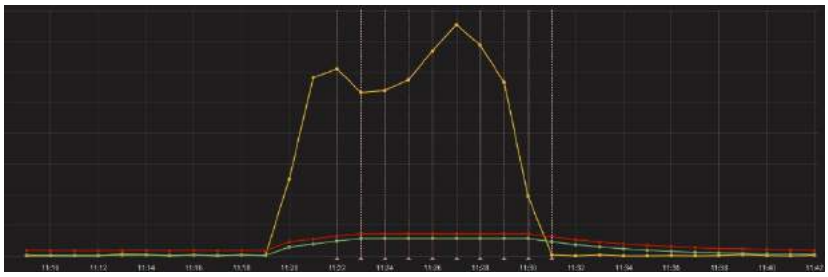
对正态分布而言，样本 99.74%都分布在中心位置$\pm 3\sigma$之处

针对应用异常指标检测这个场景，为了更具体说明算法步骤，我们使用动态阈值告警和周期性检测来进一步阐述。

2.1.1 动态阈值告警:

在创建 ARMA 模型之后，利用频域特性设计模型参数，待模型参数调节得当后，对原始时间序列进行平滑滤波，确定时序每一时刻的统计中心，中心位置确定后，再结合 3sigma 准则给出时序每一时刻的上限，以此达到动态阈值的功能。

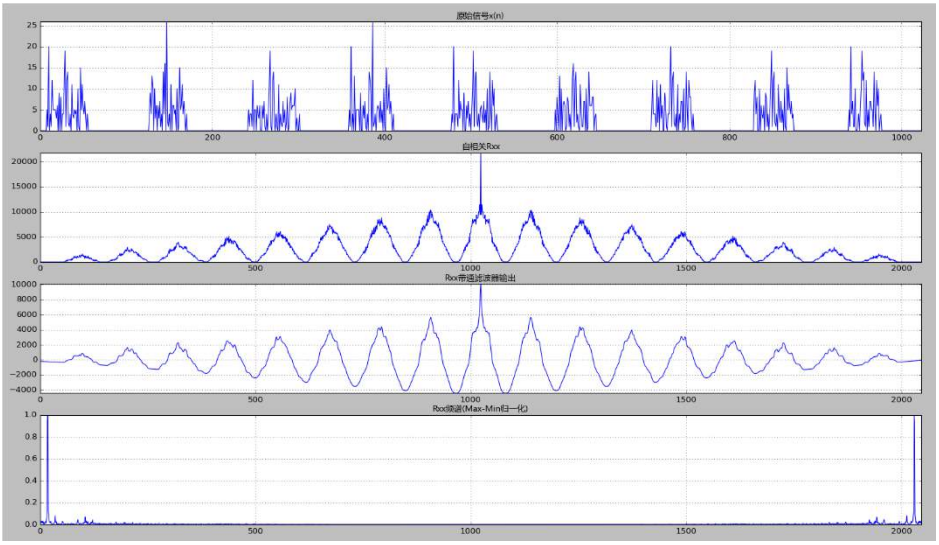
以下示例为生产某个应用一段包含异常情况的监控时间序列，黄色曲线代表了原始时序数据，绿色的曲线为 ARMA 平滑滤波的输出，红色曲线是在绿色曲线基础上叠加 3sigma 之后的动态阈值(根据实际数据的统计特性，如果满足超高斯分布，则可选择 2sigma 或更小；对于亚高斯分布，可选择 5sigma 或更大等)。借助动态阈值，异常点(离群较远的监控点)很容易被识别出来。



某段包含异常的时间序列检测，红色竖直虚线位置为检测到的异常点

2.1.2 周期性检测：

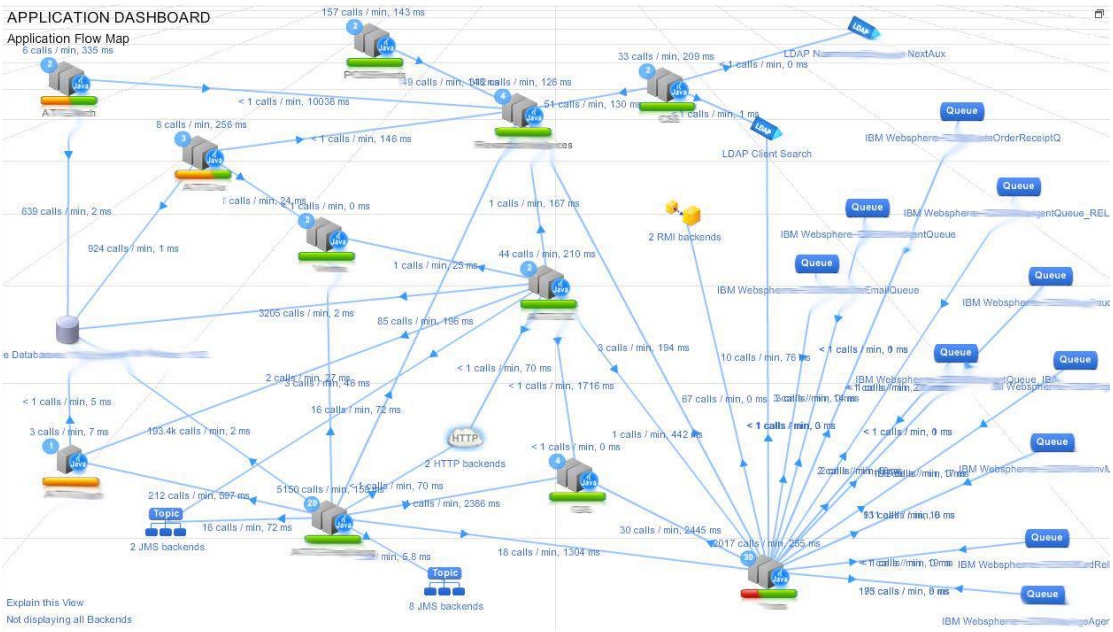
自动周期发现主要是通过对监控指标一段时间的观察，找出其中蕴含的规律信息，例如检查网站是否存在定期的爬虫抓取房价、票价现象，自动发现某些监控指标的季节性指数等场景。通常的办法是将时间域的时序信号经过傅里叶变换映射到频率域加以分析。以下示例是对生产某个应用一段监控时间序列先做自相关降噪，滤波器剔除高频干扰，然后借助快速傅里叶变换 FFT 确定时序周期的场景。



自上而下分为原始时序、自相关函数输出、滤波器滤波后的时序及 FFT 变换之后的幅频响应，从最下面一幅图可以清楚的看到一条谱线，通过适当的变换即可得出该时序发生的周期

2.2 故障智能诊断

便随着携程业务的不断扩大，网站架构也在朝着越来越复杂的结构演化。一旦出现订单或应用异常，靠人力已经很难快速定位到故障根源，即便是经验丰富的资深运维人员很多时候也需要花费很长时间来定位。对携程这样一个在 OTA 行业的领军企业来说，长时间的网站不可用，损失的不仅仅是收入，更是用户体验和社会信任，因而能够快速定位故障源和止损，至关重要。



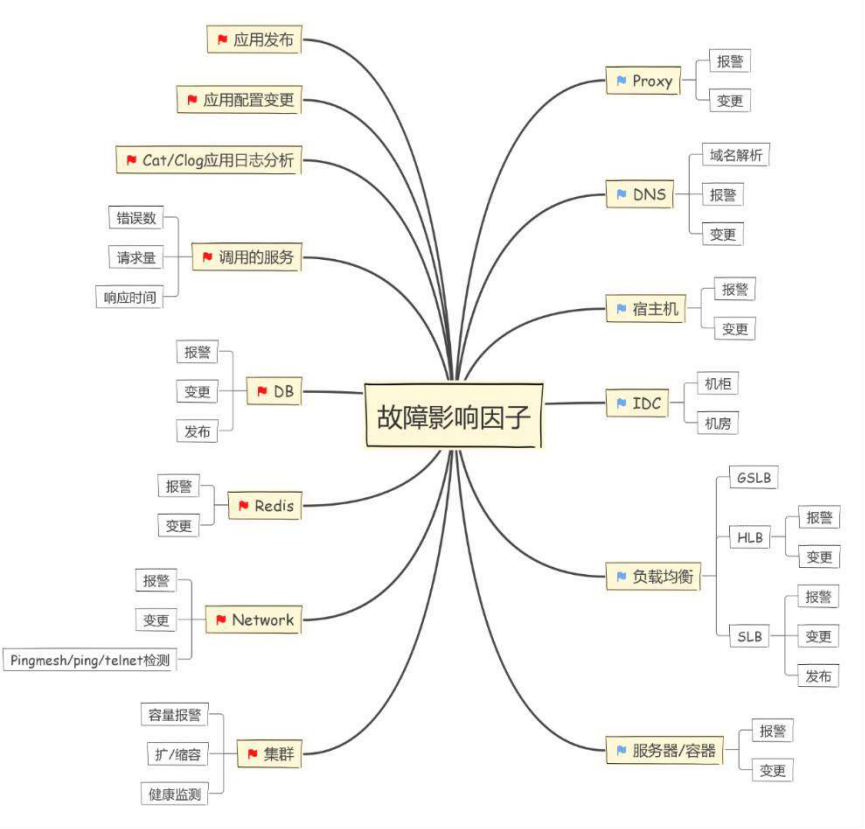
实际的网站架构拓扑更加复杂

正因网站架构非常复杂，故障点就可能存在于网络线路、路由器、交换机、机架、服务器、负载均衡设备、代理、DNS、CDN、数据库、Redis、应用程序、外部供应商接口等各个环节。

而且对于大部分的网站故障，往往环节相扣。例如，上游的故障源，通过调用链，一层层传遍所有依赖方，即使每个环节都有着非常完善的监控告警系统，但面对大量的告警风暴，定位故障根源绝非易事。另外，定位到故障根源之后，也需要手动恢复，如此一来也是增加了网站恢复的总时间。

为了解决这个难题，我们收集来自各个告警系统、发布系统、变更系统、配置中心等多个数据源的消息，包括网络告警、基础系统告警、应用告警、DB 告警、Redis 告警、代理告警、应用发布、变更、配置修改等，通过因子分析、相关性分析、决策树分析、马尔科夫链分析、调用链分析(面积算法)、专家知识库分析等方法，对每个候选的故障、发布或变更利用相关系数或贝叶斯公式给出一定的分数(分数代表了可信度)，分数最高的确定为故障源。

通过不断地实践优化，未来花费在排障中的时间将大大减少，由原来数十分钟、乃至小时级别的排障时间缩短至分钟级，智能故障诊断将成为提升网站可用性最重要的保障之一。



所有潜在故障因子展示

选取其中调用链故障诊断的部分逻辑做算法说明。

故障应用相关性诊断:

所谓牵一发而动全身，某个应用发生异常时，往往会殃及到整个调用链。故障期间如果可以归类这些相关联的告警，对于收敛告警或判断是否大面积故障，将变得非常有帮助。在众多判断相关性算法中，最简单的相关系数是一个不错的衡量标准。计算两个告警时序之间的皮尔逊相关系数，再结合应用之间的调用关系，即可给出故障应用相关性的诊断报告。

以下是某个时间点的故障应用相关性部分诊断报告内容，通过这个报告，可以比较快速的判断出是局部异常还是全局故障。如果仅为调用链上的相关告警，则认为是局部异常；反之则可能为全局或大面积故障导致。

AppId(1)	AppId(2)	存在调用	相关系数	是否相关
10[redacted]	120	是	0.9676	是
10[redacted]	121	是	0.9644	是
10[redacted]	116	是	0.9557	是
1[redacted]	120	否	0.9909	是
9[redacted]	120	否	0.9906	是
1[redacted]	121	否	0.9884	是
9[redacted]	120	否	0.9869	是
9[redacted]	121	否	0.9743	是
100[redacted]	121	否	0.959	是
9[redacted]	121	否	0.9588	是

告警时序相关性分析显示应该为工具组件或大面积异常导致

2.3 资源利用率提升

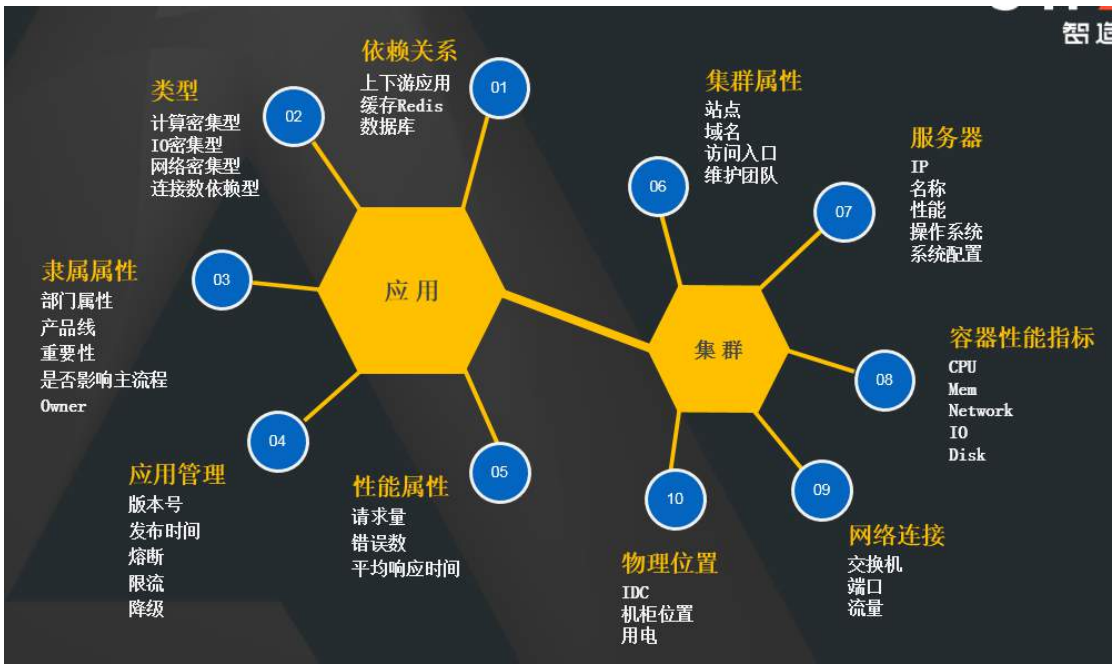
如何在满足业务需求、安全的前提下，更加经济的运营网站是每个互联网公司都在不断探索的课题。降低网站成本一个很不错的主意就是提升对资源的利用率。接下来通过应用画像、Online/Offline 混部以及智能弹性扩缩容来展示携程在提升资源利用率方面借助机器学习算法的成功实践。

2.3.1 应用画像

作为国内最大的在线旅游 OTA，携程拥有成千上万的应用，每个应用对资源的使用情况也各不相同，哪些是 CPU 计算型应用、哪些是内存消耗型应用、哪些是高 IO 应用等，为了提高应用对资源的整体利用率，需要我们有能力对其进行区分。

针对这个问题，我们设计了应用画像，通过利用 K-means、EM 等聚类算法对应用基础监控的各项监控指标、应用监控指标、发布历史数据做分类，区分出：CPU 密集型、内存密集型、网络 IO 密集型、请求耗时型、频繁发布型等应用，同时给出每个应用属于某个标签的置信度。

对比以往宿主机上随机分配应用的资源，借助应用画像，将亲和性高的应用部署在同一宿主机上可以有效提高整体资源的利用率。

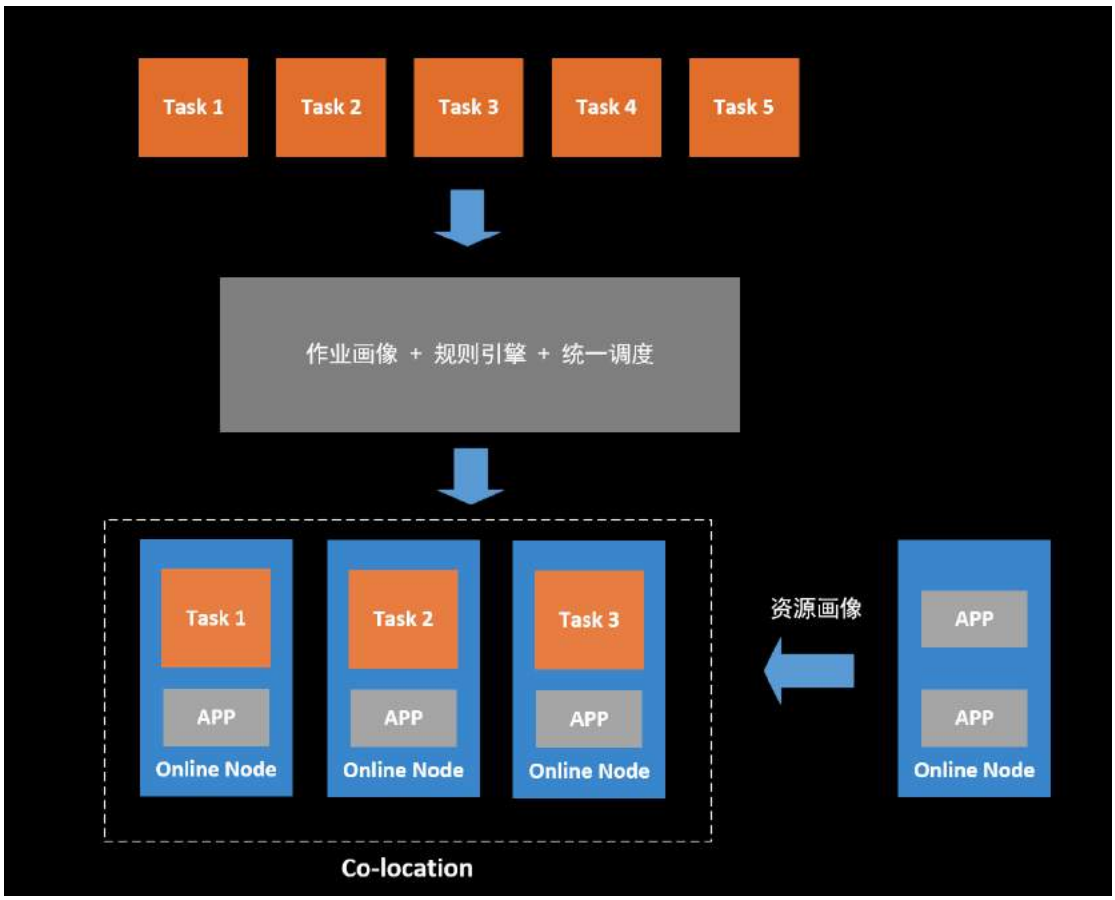


应用画像各个维度标签展示，除帮助提升资源利用率外，在其他场景也有广泛使用

2.3.2 Online/Offline 混部

携程的业务特点决定了大部分的 Online 应用资源在夜间非常的空闲，而 Offline 的一些 Hadoop、Spark 作业在这段时间对资源的利用率又非常高。

借助应用画像筛选合适的 Online 应用资源，在其利用率低峰期间部署实例调度 Offline 作业使用，即填补了 Online 资源利用率的低谷，又为 Offline 作业减少了离线资源采购(离线资源采购费用非常昂贵)。Online/Offline 混部在提升数倍资源利用率、分担 Offline 作业的同时，也提供了一种对资源互补使用的常态调度模式。



Offline 作业调度到 Online 计算节点

2.3.3 智能弹性扩缩容

虽然携程已经具备了虚拟机分钟级交付、容器秒级交付的能力，但在传统运维中，资源的申请与回收均需要有相关人员手动触发才可以完成。如遇到突发流量飙升造成的容量不足，且相关人员不能及时就位，就可能引发生产故障；另外，因业务逻辑下线但服务器并未下线时，长此以往就会堆积大量的空闲服务器，造成了对资源的浪费以及网站成本的增加。

针对这个场景，设计资源的利用率模型，使用 SVM(支撑向量机)等算法对 CPU、内存、网络 IO 等多个维度加以训练，定期扫描生产资源，产出容量富裕和不足的容量报告。针对这些容量报告，通过携程内部的弹簧系统(Spring，弹性扩缩容平台)对资源补给和裁剪。整个过程不需要人工干预，在节省人力成本的同事，大大提高了资源的利用率，降低了运营成本。

三、结束语

Dev 和 Ops 的碰撞，产生了 DevOps，DevOps 的出现避免了重复性的工作，减少了人力成本，显著地提升了运维效率；AI 和 Ops 的邂逅，诞生了 AIOps，作为 DevOps 更高阶的实现，以机器学习、统计替代基于规则的传统运维模式，通过对海量运维数据的分析，更加智能的做出分析、决策。AIOps 目前还属于不断完善和探索阶段，未来会有更多的场景被挖掘和应用。可以预见，面对未来越来越复杂的运维场景和挑战，非 AIOps 而不能胜任。

记一个真实的排障案例：携程 Redis 偶发连接失败案例分析

[作者简介]张延俊，携程技术保障中心资深 DBA，参与携程 MySQL 与 Redis 的运维工作。在数据库 HA，自动化运维建设，数据库架构和疑难问题分析排查方面有浓厚的兴趣。
寿向晨，携程技术保障中心高级 DBA，参与携程 Redis 及 DB 的运维工作。在自动化运维，流程化及监控排障等方面有较多的实践经验，喜欢深入分析问题，提高团队运维效率。

Redis 是使用非常广泛的开源缓存数据库，在携程几乎所有业务线都有使用。本文来源于线上真实案例，记录了一次偶发 Redis 访问错误的排障过程，从网络和内核深入解析此次报错的前因后果，希望对各位有所帮助。

一、问题描述

生产环境有一个 Redis 会偶尔发生连接失败的报错。报错的时间点，客户端 IP 并没有特别明显的规律。以下是客户端报错信息。过一会儿，报错会自动恢复。

```
CRedis.Client.RExceptions.ExcuteCommandException:Unable to Connect redis server: ---> CRedis.Third.Redis.RedisException: Unable to Connect redis server:
在 CRedis.Third.Redis.RedisNativeClient.CreateConnectionError()
在 CRedis.Third.Redis.RedisNativeClient.SendExpectData(Byte[][]cmdWithBinaryArgs)
在 CRedis.Client.Entities.RedisServer.<>c__DisplayClassd`1.<Get>b__c(RedisClientclien)
在 CRedis.Client.Logic.ClientPool.ExcuteAction[T](Func`2 action,String methodName)
在 CRedis.Client.Logic.ClientPool.Excute[T](Func`2 action, StringmethodName)
```

从报错的信息来看，应该是连接不上 Redis 所致。Redis 的版本是 2.8.19。虽然版本有点老，但基本运行稳定。

线上环境只有这个集群有偶尔报错。这个集群的一个比较明显的特征是客户端服务器比较多，有上百台。

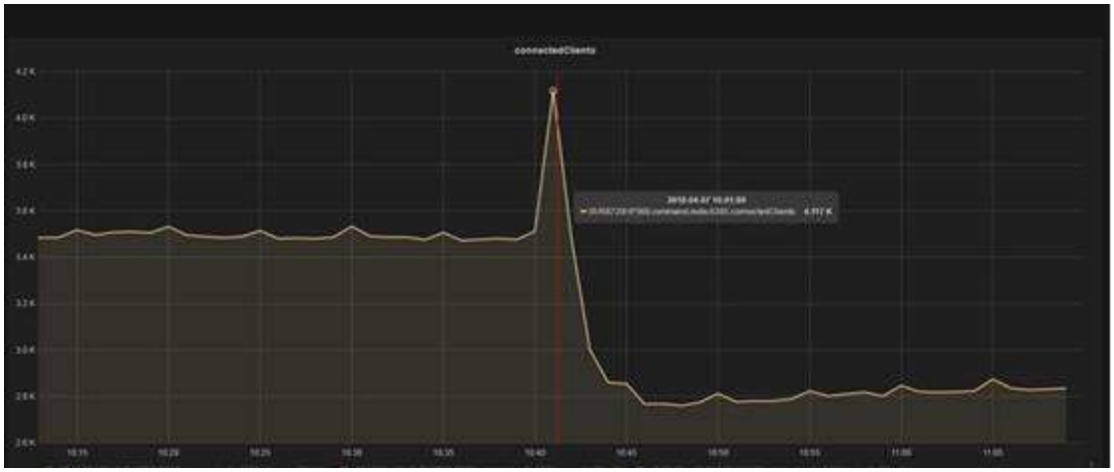
二、问题分析

从报错的信息来看，是客户端连接不到服务端。

一个常见的原因是端口耗尽，于是我们对网络连接进行排查，在出问题的点上，TCP 连接数远没有达到端口耗尽的场景。因此这个不是 Redis 连接不上的根因。

另外一种常见的场景是，在服务端有慢查询，导致 Redis 服务阻塞。在 Redis 服务端，我们把运行超过 10 毫秒的语句进行抓取，也没有抓到运行慢的语句。

从服务端部署的监控来看，出问题的点上，连接数有一个突然飙升，从 3500 个链接突然飙升至 4100 个链接。如下图显示。



同时间，服务器端显示 Redis 服务端有丢包现象。345539 – 344683 = 856 个包。

```
Sat Apr  710:41:40 CST 2018
    1699 outgoing packets dropped
    92 dropped because of missing route
344683 SYNs to LISTEN sockets dropped
344683 times the listen queue of a socket overflowed

Sat Apr  710:41:41 CST 2018
    1699 outgoing packets dropped
    92 dropped because of missing route
    345539 SYNs to LISTEN sockets dropped
    345539 times the listenqueue of a socket overflowed
```

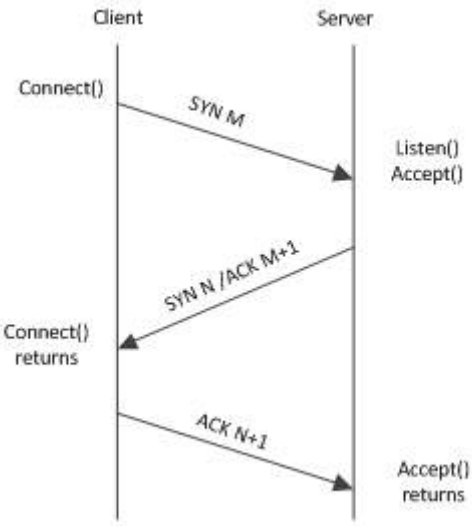
客户端报错的原因基本确定，是因为建连速度太快，导致服务端 backlog 队列溢出，连接被 server 端 reset。

三、关于 backlogoverflow

在高并发的短连接服务中，这是一种很常见的 tcp 报错类型。一个正常的 tcp 建连过程如下：

1）client 发送一个(SYN)给 server；
2）server 返回一个(SYN,ACK)给 client；
3）client 返回一个(ACK)，三次握手结束。

对 client 来说建连成功，client 可以继续发送数据包给 server，但是这个时候 server 端未必 ready，画成图的话大概是这个样子：



在 BSD 版本内核实现的 tcp 协议中，server 端建连过程需要两个队列，一个是 SYN queue，一个是 accept queue。

前者叫半开连接（或者半连接）队列，在接收到 client 发送的 SYN 时加入队列。(一种常见的网络攻击方式就是不断发送 SYN，但是不发送 ACK，从而导致 server 端的半开队列撑爆，server 端拒绝服务)。

后者叫全连接队列，server 返回（SYN,ACK)，在接收到 client 发送 ACK 后（此时 client 会认为建连已经完成，会开始发送 PSH 包)，如果 accept queue 没有满，那么 server 从 SYN queue 把连接信息移到 accept queue；如果此时 accept queue 溢出的话，server 的行为要看配置。

如果 tcp_abort_on_overflow 为 0 (默认)，那么直接 drop 掉 client 发送的 PSH 包，此时 client 会进入重发过程，一段时间后 server 端重新发送 SYN,ACK，重新从建连的第二步开始；如果 tcp_abort_on_overflow 为 1，那么 server 端发现 accept queue 满之后直接发送 reset。

通过 wireshark 搜索发现，在一秒内有超过 2000 次对 Redis Server 端发起建连请求。我们尝试修改 tcp backlog 大小，从 511 调整到 2048，问题并没有得到解决。所以此类微调，并不能彻底的解决问题。

四、网络包分析

我们用 wireshark 来识别网络拥塞的准确时间点和原因。已经有了准确的报错时间点，先用 editcap 把超大的 tcp 包裁剪一下，裁成 30 秒间隔，并通过 wireshark I/O 100ms 间隔分析网络阻塞的准确时间点。

根据图标可以明显看到 tcp 的 packets 来往存在 block。

对该 block 前后的网络包进行明细分析，网络包来往情况如下：

Time	Source	Dest	Description
12:01:54.6536050	Redis-Server	Clients	TCP:Flags=...AP...
12:01:54.6538580	Redis-Server	Clients	TCP:Flags=...AP...
12:01:54.6539770	Redis-Server	Clients	TCP:Flags=...AP...
12:01:54.6720580	Redis-Server	Clients	TCP:Flags=...A..S..
12:01:54.6727200	Redis-Server	Clients	TCP:Flags=...A.....
12:01:54.6808480	Redis-Server	Clients	TCP:Flags=...AP.....
12:01:54.6910840	Redis-Server	Clients	TCP:Flags=...A...S.,
12:01:54.6911950	Redis-Server	Clients	TCP:Flags=...A.....
...	...	...	...
12:01:56.1181350	Redis-Server	Clients	TCP:Flags=...AP....

12:01:54.6808480,Redis Server 端向客户端发送了一个 Push 包，也就是对于查询请求的一个结果返回。后面的包都是在做连接处理，包括 Ack 包，Ack 确认包，以及重置的 RST 包，紧接着下面一个 Push 包是在 12:01:56.1181350 发出的。中间的间隔是 1.4372870 秒。

也就是说，在这 1.4372870 秒期间，Redis 的服务器端，除了做一个查询，其他的操作都是在做建连，或拒绝连接。

客户端报错的前后逻辑已经清楚了，redis-server 卡了 1.43 秒，client 的 connection pool 被打满，疯狂新建连接，server 的 accept queue 满，直接拒绝服务，client 报错。

开始怀疑 client 发送了特殊命令，这时需要确认一下 client 的最后几个命令是什么，找到 redis-server 卡死前的第一个包。装一个 wireshark 的 redis 插件，看到最后几个命令是简单的 get，并且 key-value 都很小，不至于需要耗费 1.43 秒才能完成。服务端也没有 slow log，此时排障再次陷入僵局。

五、进一步分析

为了了解这 1.43 秒之内，Redis Server 在做什么事情，我们用 pstack 来抓取信息。Pstack 本质上是 gdb attach，高频率的抓取会影响 redis 的吞吐。死循环 0.5 秒一次无脑抓，在 redis-server 卡死的时候抓到堆栈如下（过滤了没用的栈信息）：

```
Thu May 31 11:29:18 CST 2018
Thread 1 (Thread 0x7ff2db6de720 (LWP 8378)):
#0  0x000000000048cec4 in ?? ()
#1  0x00000000004914a4 in je_arena_ralloc ()
#2  0x00000000004836a1 in je_realloc ()
#3  0x0000000000422cc5 in zrealloc ()
#4  0x00000000004213d7 in sdsRemoveFreeSpace ()
#5  0x000000000041ef3c inclientsCronResizeQueryBuffer ()
#6  0x00000000004205de in clientsCron ()
#7  0x0000000000420784 in serverCron ()
```

#8 0x0000000000418542 in aeProcessEvents ()
#9 0x000000000041873b in aeMain ()
#100x0000000000420fce in main ()
Thu May 31 11:29:19 CST 2018
Thread 1 (Thread 0x7ff2db6de720 (LWP8378)):
#0 0x0000003729ee5407 in madvise () from/lib64/libc.so.6
#1 0x0000000000493a4e in je_pages_purge ()
#2 0x000000000048cf70 in ?? ()
#3 0x00000000004914a4 in je_arena_ralloc ()
#4 0x00000000004836a1 in je_realloc ()
#5 0x0000000000422cc5 in zrealloc ()
#6 0x00000000004213d7 in sdsRemoveFreeSpace ()
#7 0x000000000041ef3c inclientsCronResizeQueryBuffer ()
#8 0x00000000004205de in clientsCron ()
#9 0x0000000000420784 in serverCron ()
#100x0000000000418542 in aeProcessEvents ()
#110x000000000041873b in aeMain ()
#12 0x0000000000420fcein main ()
Thu May 31 11:29:19 CST 2018
Thread 1 (Thread 0x7ff2db6de720 (LWP8378)):
#0 0x000000000048108c in je_malloc_usable_size()
#1 0x0000000000422be6 in zmalloc ()
#2 0x00000000004220bc in sdsnewlen ()
#3 0x000000000042c409 in createStringObject ()
#4 0x000000000042918e in processMultibulkBuffer()
#5 0x0000000000429662 in processInputBuffer ()
#6 0x0000000000429762 in readQueryFromClient ()
#7 0x000000000041847c in aeProcessEvents ()
#8 0x000000000041873b in aeMain ()
#9 0x0000000000420fce in main ()
Thu May 31 11:29:20 CST 2018
Thread 1 (Thread 0x7ff2db6de720 (LWP8378)):
#0 0x000000372a60e7cd in write () from/lib64/libpthread.so.0
#1 0x0000000000428833 in sendReplyToClient ()
#2 0x0000000000418435 in aeProcessEvents ()
#3 0x000000000041873b in aeMain ()
#4 0x0000000000420fce in main ()

重复多次抓取后，从堆栈中发现可疑堆栈 clientsCronResizeQueryBuffer 位置，属于 serverCron()函数下，这个 redis-server 内部的定时调度，并不在用户线程下，这个解释了为什么卡死的时候没有出现慢查询。

查看 redis 源码，确认到底 redis-server 在做什么：

```
clientsCron(server.h):
#define CLIENTS_CRON_MIN_ITERATIONS 5
voidclientsCron(void){
    /*Makesuretoprocessatleastnumclients/server.hzofclients
     *percall.Sincethisfunctioniscalledserver.hztimespersecond
     *wearesurethatintheworstcaseweprocessalltheclientsin1
     *second.*/
    intnumclients=listLength(server.clients);
    intiterations=numclients/server.hz;
    mstime_tnow=mstime();

    /*Processatleastafewclientswhilewareatit,evenifweneed
     *toprocesslessthanCLIENTS_CRON_MIN_ITERATIONStomeetourcontract
     *ofprocessingeachclientoncepersecond.*/
    if(iterations<CLIENTS_CRON_MIN_ITERATIONS)
        iterations=(numclients<CLIENTS_CRON_MIN_ITERATIONS)?
                    numclients:CLIENTS_CRON_MIN_ITERATIONS;

    while(listLength(server.clients)&&iterations--){
        client*c;
        listNode*head;

        /*Rotatethelist,takethecurrenthead,process.
         *Thiswayiftheclientmustberemovedfromthelistit'sthe
         *firstelementandwedon'tincurintoO(N)computation.*/
        listRotate(server.clients);
        head=listFirst(server.clients);
        c=listNodeValue(head);
        /*Thefollowingfunctionsdodifferentservicechecksontheclient.
         *Theprotocolisthattheyreturnnon-zeroiftheclientwas
         *terminated.*/
        if(clientsCronHandleTimeout(c,now))continue;
        if(clientsCronResizeQueryBuffer(c))continue;
    }
}
```

clientsCron 首先判断当前 client 的数量，用于控制一次清理连接的数量，生产服务器单实例的连接数量在 5000 不到，也就是一次清理的连接数是 50 个。

clientsCronResizeQueryBuffer(server.h):

```
/*Theclientquerybufferisansds.cstringthatcanendwithalotof
 *freespacenotused,thisfunctionreclaimsspaceifneeded.
 *
```

```
 *Thefunctionalwaysreturns0asitneverterminatestheclient.*/
intclientsCronResizeQueryBuffer(client*c){
    size_tquerybuf_size=sdsAllocSize(c->querybuf);
    time_tidletime=server.unixtime-c->lastinteraction;

    /*只在以下两种情况下会 Resize querybuffer:
     *1)Querybuffer>BIG_ARG( 在 server.h 中 定 义 #definePROTO_MBULK_BIG_ARG
(1024*32))且这个 Buffer 的小于一段时间的客户端使用的峰值.
     *2)客户端空闲超过 2s 且 Buffer size 大于 1k.*/
    if(((querybuf_size>PROTO_MBULK_BIG_ARG)&&
          (querybuf_size/(c->querybuf_peak+1))>2)||
          (querybuf_size>1024&&idletime>2))
    {
        /*Onlyresizethequerybufferifitisactuallywastingspace.*/
        if(sdsavail(c->querybuf)>1024){
            c->querybuf=sdsRemoveFreeSpace(c->querybuf);
        }
    }
    /*Resetthepeakagaintocapturethepeakmemoryusageinthenext
     *cycle.*/
    c->querybuf_peak=0;
    return0;
}
```

如果 redisClient 对象的 query buffer 满足条件，那么就直接 resize 掉。满足条件的连接分成两种，一种是真的很大的，比该客户端一段时间内使用的峰值还大；还有一种是很闲（idle>2）的，这两种都要满足一个条件，就是 buffer free 的部分超过 1k。

那么 redis-server 卡住的原因就是，正好有那么 50 个很大的或者空闲的并且 free size 超过了 1k 大小连接的同时，循环做了 resize。由于 redis 都属于单线程工作的程序，所以 block 了 client。

那么解决这个问题办法就很明朗了，让 resize 的频率变低或者 resize 的执行速度变快。

既然问题出在 querybuffer 上，我们先看一下这个东西被修改的位置：

readQueryFromClient（networking.c）：

```
redisClient*createClient(intfd){
    redisClient*c=zmalloc(sizeof(redisClient));

    /*passing-1asfditispossibletocreateanonconnectedclient.
     *ThisisusefulsinceallltheRediscommandsneedstobeexecuted
     *inthecontextofaclient.Whencommandsareexecutedinother
     *contexts(forinstanceaLuascript)weneedanonconnectedclient.*/
```

```
    if(fd!=-1){
        anetNonBlock(NULL,fd);
        anetEnableTcpNoDelay(NULL,fd);
        if(server.tcpkeepalive)
            anetKeepAlive(NULL,fd,server.tcpkeepalive);
        if(aeCreateFileEvent(server.el,fd,AE_READABLE,
            readQueryFromClient,c)==AE_ERR)
        {
            close(fd);
            zfree(c);
            returnNULL;
        }
    }

    selectDb(c,0);
    c->id=server.next_client_id++;
    c->fd=fd;
    c->name=NULL;
    c->bufpos=0;
c->querybuf=sdsempty();初始化是 0

readQueryFromClient(networking.c):
voidreadQueryFromClient(aeEventLoop*el,intfd,void*privdata,intmask){
    redisClient*c=(redisClient*)privdata;
    intnread,readlen;
    size_tqblen;
    REDIS_NOTUSED(el);
    REDIS_NOTUSED(mask);

    server.current_client=c;
    readlen=REDIS_IOBUF_LEN;
    /*Ifthisisamultibulkrequest,andweareprocessingabulkreply
     *thatislargeenough,trytomaximizetheprobabilitythatthequery
     *buffercontainsexactlytheSDSstringrepresentingtheobject,even
     *attheriskofrequiringmoreread(2)calls.Thiswaythefunction
     *processMultiBulkBuffer()canavoidcopyingbufferstocreatethe
     *RedisObjectrepresentingtheargument.*/
    if(c->reqtype==REDIS_REQ_MULTIBULK&&c->multibulklen&&c->bulklen!=-1
        &&c->bulklen>=REDIS_MBULK_BIG_ARG)
    {
        intremaining=(unsigned)(c->bulklen+2)-sdslen(c->querybuf);

        if(remaining<readlen)readlen=remaining;
    }
```

```
qblen=sdslen(c->querybuf);
if(c->querybuf_peak<qblen)c->querybuf_peak=qblen;
c->querybuf=sdsMakeRoomFor(c->querybuf,readlen);在这里会被扩大
```

由此可见 c->querybuf 在连接第一次读取命令后的大小就会被分配至少 1024*32，所以回过头再去看 resize 的清理逻辑就明显存在问题。

每个被使用到的 query buffer 的大小至少就是 1024*32，但是清理的时候判断条件是>1024，也就是说，所有的 idle>2 的被使用过的连接都会被 resize 掉，下次接收到请求的时候再重新分配到 1024*32。

这个其实是没有必要的，在访问比较频繁的群集，内存会被频繁得回收重分配，所以我们尝试将清理的判断条件改造为如下，就可以避免大部分没有必要的 resize 操作。

```
if (((querybuf_size >REDIS_MBULK_BIG_ARG) &&
        (querybuf_size/(c->querybuf_peak+1)) > 2) ||
        (querybuf_size >1024*32 && idletime > 2))
    {
        /* Only resize the query buffer if it is actually wasting space. */
        if (sdsavail(c->querybuf) > 1024*32) {
            c->querybuf = sdsRemoveFreeSpace(c->querybuf);
        }
    }
```

这个改造的副作用是内存的开销，按照一个实例 5k 连接计算。5000*1024*32=160M，这点内存消耗对于上百 G 内存的服务器完全可以接受。

六、问题重现

在使用修改过源码的 Redisserver 后，问题仍然重现了，客户端还是会报同类型的错误，且报错的时候，服务器内存依然会出现抖动。抓取内存堆栈信息如下：

```
Thu Jun 14 21:56:54 CST 2018
#3 0x0000003729ee893d in clone () from /lib64/libc.so.6
Thread 1 (Thread 0x7f2dc108d720 (LWP27851)):
#0 0x0000003729ee5400 in madvise () from /lib64/libc.so.6
#1 0x0000000000493a1e in je_pages_purge ()
#2  0x000000000048cf40 in arena_purge ()
#3 0x00000000004a7dad in je_tcache_bin_flush_large ()
#4 0x00000000004a85e9 in je_tcache_event_hard ()
#5 0x000000000042c0b5 in decrRefCount ()
#6 0x000000000042744d in resetClient ()
#7  0x000000000042963bin processInputBuffer ()
```

```
#8 0x0000000000429762 in readQueryFromClient ()
#9 0x000000000041847c in aeProcessEvents ()
#10 0x000000000041873b in aeMain ()
#11 0x0000000000420fce in main ()
Thu Jun 14 21:56:54 CST 2018
Thread 1 (Thread 0x7f2dc108d720 (LWP27851)):
#0 0x0000003729ee5400 in madvise () from /lib64/libc.so.6
#1 0x0000000000493a1e in je_pages_purge ()
#2  0x000000000048cf40 in arena_purge ()
#3 0x00000000004a7dad in je_tcache_bin_flush_large ()
#4 0x00000000004a85e9 in je_tcache_event_hard ()
#5 0x000000000042c0b5 in decrRefCount ()
#6 0x000000000042744d in resetClient ()
#7 0x000000000042963b in processInputBuffer ()
#8 0x0000000000429762 in readQueryFromClient ()
#9 0x000000000041847c in aeProcessEvents ()
#10 0x000000000041873b in aeMain ()
#11 0x0000000000420fce in main ()
```

显然，Querybuffer 被频繁 resize 的问题已经得到了优化，但是还是会出现客户端报错。这就又陷入了僵局。难道还有其他因素导致 query buffer resize 变慢?

我们再次抓取 pstack。但这时，jemalloc 引起了我们的注意。

此时回想 Redis 的内存分配机制, Redis 为避免 libc 内存不被释放导致大量内存碎片的问题，默认使用的是 jemalloc 用作内存分配管理，这次报错的堆栈信息中都是 je_pages_purge() redis 在调用 jemalloc 回收脏页。

我们线上的 Redis 版本大多是 2.8.19 版本，原生代码中使用的是 jemalloc 3.6 的版本，我们看下 jemalloc 做了些什么。

```
arena_purge(arena.c)
staticvoid
arena_purge(arena_t*arena,boolall)
{
                        arena_chunk_t*chunk;
                        size_tnpurgatory;
                        if(config_debug){
                                                size_tndirty=0;

arena_chunk_dirty_iter(&arena->chunks_dirty,NULL,
                                                        chunks_dirty_iter_cb,(void*)&ndirty);
                                                assert(ndirty==arena->ndirty);
                        }
```

```
                assert(arena->ndirty>arena->npurgatory||all);
                assert((arena->nactive>>opt_lg_dirty_mult)<(arena->ndirty-
                    arena->npurgatory)||all);

                if(config_stats)
                                arena->stats.npurge++;
                npurgatory=arena_compute_npurgatory(arena,all);
                arena->npurgatory+=npurgatory;

                while(npurgatory>0){
                                size_tnpurgeable,npurged,nunpurged;

                                /*Getnextchunkwithdirtypages.*/

chunk=arena_chunk_dirty_first(&arena->chunks_dirty);
                                if(chunk==NULL){
                                                arena->npurgatory-
=npurgatory;
                                                return;
                                }
                                npurgeable=chunk->ndirty;
                                assert(npurgeable!=0);


if(npurgeable>npurgatory&&chunk->nruns_adjac==0){


arena->npurgatory+=npurgeable-npurgatory;

npurgatory=npurgeable;
                                }
                                arena->npurgatory-=npurgeable;
                                npurgatory-=npurgeable;

npurged=arena_chunk_purge(arena,chunk,all);
                                nunpurged=npurgeable-npurged;
                                arena->npurgatory+=nunpurged;
                                npurgatory+=nunpurged;
                }
}
```

Jemalloc 每次回收都会判断所有实际应该清理的 chunck 并对清理做 count，这个操作对于高响应要求的系统是很奢侈的，所以我们考虑通过升级 jemalloc 的版本来优化 purge 的性能。

Redis 4.0 版本发布后，性能有很大的改进，并可以通过命令回收内存，我们线上也正准备进行升级，跟随 4.0 发布的 jemalloc 版本为 4.1。

jemalloc 的版本使用的在 jemalloc 的 4.0 之后版本的 arena_purge()做了很多优化，去掉了计数器的调用，简化了很多判断逻辑，增加了 arena_stash_dirty()方法合并了之前的计算和判断逻辑，增加了 purge_runs_sentinel，用保持脏块在每个 arena LRU 中的方式替代之前的保持脏块在 arena 树的 dirty-run-containing chunck 中的方式，大幅度减少了脏块 purge 的体积，并且在内存回收过程中不再移动内存块。

代码如下：

```
arena_purge(arena.c)
staticvoid
arena_purge(arena_t*arena,boolall)
{
                chunk_hooks_tchunk_hooks=chunk_hooks_get(arena);
                size_tnpurge,npurgeable,npurged;
                arena_runs_dirty_link_tpurge_runs_sentinel;
                extent_node_tpurge_chunks_sentinel;

                arena->purging=true;

                /*
                 *Callstoarena_dirty_count()aredisabledevenfordebugbuilds
                 *becauseoverheadgrowsnonlinearlyasmemoryusageincreases.
                 */
                if(false&&config_debug){
                                size_tndirty=arena_dirty_count(arena);
                                assert(ndirty==arena->ndirty);
                }
                assert((arena->nactive>>arena->lg_dirty_mult)<arena->ndirty||all);

                if(config_stats)
                                arena->stats.npurge++;

                npurge=arena_compute_npurge(arena,all);
                qr_new(&purge_runs_sentinel,rd_link);
                extent_node_dirty_linkage_init(&purge_chunks_sentinel);

                npurgeable=arena_stash_dirty(arena,&chunk_hooks,all,npurge,
                        &purge_runs_sentinel,&purge_chunks_sentinel);
                assert(npurgeable>=npurge);
```

```
npurged=arena_purge_stashed(arena,&chunk_hooks,&purge_runs_sentinel,
                            &purge_chunks_sentinel);
                        assert(npurged==npurgeable);
                        arena_unstash_purged(arena,&chunk_hooks,&purge_runs_sentinel,
                            &purge_chunks_sentinel);

                        arena->purging=false;
}
```

七、解决问题

实际上我们有多个选项。可以使用 Google 的 tcmalloc 来代替 jemalloc，可以升级 jemalloc 的版本等等。

根据上面的分析，我们尝试通过升级 jemalloc 版本，实际操作为升级 Redis 版本来解决。将 Redis 的版本升级到 4.0.9 之后观察，线上客户端连接超时这个棘手的问题得到了解决。

八、问题总结

Redis 在生产环境中因其支持高并发，响应快，易操作被广泛使用，对于运维人员而言，其响应时间的要求带来了各种各样的问题，

Redis 的连接超时问题是其中比较典型的一种，从发现问题，客户端连接超时，到通过抓取客户端与服务端的网络包，内存堆栈定位问题，我们也被其中一些假象所迷惑，最终通过升级 jemalloc（Redis）的版本解决问题。

希望本文能给遇到同样问题的小伙伴们一些启发。

携程一次 Redis 迁移容器后 Slowlog “异常”分析

[作者简介]李剑，携程技术保障中心系统研发部资深软件工程师，负责 Redis 和 Mysql 的容器化和服务化工作，以及维护容器宿主机的内核版本，喜欢深入分析系统疑难杂症。

容器化对于 Redis 自动化运维效率、资源利用率方面都有巨大提升，携程在对 Redis 在容器上性能和稳定性进行充分验证后，启动了生产 Redis 迁移容器化的项目。其中第一批次两台宿主机，第二批次五台宿主机。

本次“异常”是第二批次迁移过程中发现的，排查过程一波三折，最终得出让人吃惊的结论。

希望本次结论能给遇到同样问题的小伙伴以启发，另外本次分析问题的思路对于分析其他疑难杂症也有一定借鉴作用。

一、问题描述

在某次 Redis 迁移容器后，DBA 发来告警邮件，slowlog>500ms，同时在 DBA 的慢日志查询里可以看到有 1800ms 左右的日志，如下图 1 所示：

图 1

二、分析过程

2.1 什么是 slowlog

在分析问题之前，先简单解释下 Redis 的 slowlog。阅读 Redis 源码（图 2）不难发现，当某次 Redis 的操作大于配置中 slowlog-log-slower-than 设置的值时，Redis 就会将该值记录到内存中，通过 slowlog get 可以获取该次 slowlog 发生的时间和耗时，图 1 的监控数据也是从此获得。

```
    start = ustime();//call gettimeofday()
    c->cmd->proc(c);
    duration = ustime()-start;

//ustime function
/* Return the UNIX time in microseconds */
long long ustime(void) {
    struct timeval tv;
    long long ust;

    gettimeofday(&tv, NULL);
    ust = ((long long)tv.tv_sec)*1000000;
    ust += tv.tv_usec;
    return ust;
}
```

图 2

也就是说，slowlog 只是单纯的计算 Redis 执行的耗时时间，与其他因素如网络之类的都没关系。

2.2 矛盾的日志

每次 slowlog 都是 1800+ms 并且都随机出现，在第一批次 Redis 容器化的宿主机上完全没有这种现象，而 QPS 远小于第一批次迁移的某些集群，按常理很难解释，这时候翻看 CAT 记录，更加加重了我们的疑惑，见图 3:

图 3

CAT 是携程根据开源软件(https://github.com/dianping/cat)的定制版本，用于客户端记录打点的耗时，从图中可以很清晰的看到，Redis 打点的最大值 367ms 也远小于 1800ms，它等于是说下面这张自相矛盾图，见图 4:

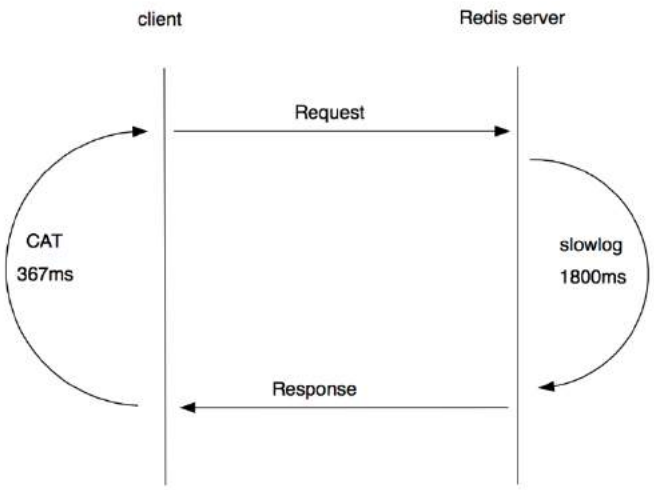


图 4

2.3 求助社区

所以说，slowlog 问题要么是 CAT 误报，要么是 Redis 误报，但 Redis 使用如此广泛，并且经过询问 CAT 的维护者说 CAT 有一定的消息丢弃率，而 Redis 在官方 github issue 中并没有发现类似的 slowlog 情形，因此我们第一感觉是 CAT 误报，并在官方 Redis issue 中提问，试图获取社区的帮助。

很快社区有人回复，可能是 NUMA 架构导致的问题，但也同时表示 NUMA 导致 slowlog 高达 1800ms 很不可思议。关于 NUMA 的资料网上有很多，这里不再赘述，我们在查阅相关 NUMA 资料后也发现，NUMA 架构导致如此大的 slowlog 不太可能，因此放弃了这条路径的尝试。

2.4 豁然开朗

看上去每个方面好像都没有问题，而且找不到突破口，排障至此陷入了僵局。

重新阅读 Redis 源代码，直觉发现 gettimeofday()可能有问题，模仿 Redis 获取 slowlog 的代码，写了一个简答的死循环，每次 Sleep 一秒，看看打印出来的差值是否正好 1 秒多点，如图 5 所示：

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>

static long long ustime(void) {
    struct  timeval tv;
    long long ust;

    gettimeofday(&tv, NULL);
    ust = ((long long)tv.tv_sec)*1000000;
    ust += tv.tv_usec;
    return ust;
}
int main(){
       FILE *ff=fopen("log", "wb");
       for   (;;){
             long long start = ustime();
             sleep(1);
             long long end = ustime();
             int duration = end -start;
             fprintf(ff, "start:%lld,end:%lld, duration:%d\n", start, end,duration);
             fflush(ff);
        }
        fclose(ff);
        return 0;

}
```

图 5

图 5 的程序大概运行了 20 分钟后，奇迹出现了，gettimeofday 果然有问题，下面是上面程序测试时间打印出来的 LOG，如图 6:

```
start:1533721454818576,end:1533721455818700, duration:1000124
start:1533721455818746,end:1533721456818878, duration:1000132
start:1533721456818903,end:1533721457819029, duration:1000126
start:1533721457819068,end:1533721458819145, duration:1000077
start:1533721458819178,end:1533721459819249, duration:1000071
start:1533721459819273,end:1533721460819409, duration:1000136
start:1533721460819448,end:1533721463633084, duration:2813636
start:1533721463633120,end:1533721464633194, duration:1000074
start:1533721464633237,end:1533721465633319, duration:1000082
start:1533721465633353,end:1533721466633426, duration:1000073
start:1533721466633468,end:1533721467633600, duration:1000132
start:1533721467633634,end:1533721468633758, duration:1000124
start:1533721468633802,end:1533721469633932, duration:1000130
```

图 6

图 6 中标红的时间减去 1 秒等于 1813ms，与 slowlog 时间如此相近！在容器所在的物理机上也测试一遍，发现有同样的现象，排除因容器导致 slowlog，希望的曙光似乎就在眼前了，那么问题又来了:

1、到底为什么会相差 1800ms+呢?
2、为什么第一批机器没有这种现象呢?
3、为什么之前跑在物理机上的 Redis 没有这种现象呢?

带着这三个问题，重新审视系统调用 gettimeofday 获取当前时间背后的原理，发现一番新天地。

三、系统时钟

系统时钟的实现非常复杂，并且参考资料非常多。

简单来说 我们可以通过
cat /sys/devices/system/clocksource/clocksource0/current_clocksource
命令来获取当前系统的时钟源，携程的宿主机上都是统一 Time Stamp Counter(TSC)：80x86 微处理器包括一个时钟输入插口，用来接收来自外部振荡器的时钟信号，从奔腾 80x86 微处理器开始，增加了一个计数器。

随着每增加一个时钟信号而加一，通过 rdtsc 汇编指令也可以去读 TSC 寄存器，这样如果 CPU 的频率是 1GHz，TSC 寄存器就能提供纳秒级别的计时精度，并且现代 CPU 通过 FLAG constant_tsc 来保证即使 CPU 休眠也不影响 TSC 的频率。

当选定 TSC 为时钟源后，gettimeofday 获取墙上时钟（wall-clock）正是从 TSC 寄存器读出来的值转换而来，所谓墙上时钟主要是参照现实世界人们通过墙上时钟获取当前时间，但是用来计时并不准确，可能会被 NTP 或者管理员修改。

那么问题又来了，宿主机的时间没有被管理员修改，难道是被 NTP 修改？即使是 NTP 来同步，每次相差也不该有 1800ms 这么久，它的意思是说难道宿主机的时钟每次都在变慢然后被 NTP 拉回到正常时间？我们手工执行了下 NTP 同步，发现的确是有很大偏差，如图 7 所示：

图 7

按常识时钟正常的物理机与 NTP 服务器时钟差异都在 1ms 以内，相差 1s+绝对有问题，而且还是那个老问题，为什么第一批次的机器上没有问题？

四、内核 BUG

两个批次宿主机一样的内核版本，第一批没问题而第二批有问题，差异只可能在硬件上，非常有可能在计时上，翻看内核的 commit log 终于让我们发现了这样的 commit，如图 8 所示：

```
Linux-4.9 added INTEL_FAM6_SKYLAKE_X to native_calibrate_tsc():

commit 6baf3d61821f
("x86/tsc: Add additional Intel CPU models to the crystal quirk list")

There are several problems with doing this.

The first is that while SKX servers use a 25 MHz crystal,
SKX workstations (with same model #) use a 24 MHz crystal.
This results in a -4.0% time drift rate on SKX workstations.

While SKX servers do have a 25  MHz crystal, but they too have a problem.
All SKX subject the crystal to an EMI reduction circuit that
reduces its actual frequency by (approximately) -0.25%.
This results in -1 second per 10 minute time drift
as compared to network time.

This issue can also trigger a timer and power problem,
on configurations that use the LAPIC timer (versus the TSC deadline timer).
Clock ticks scheduled with the LAPIC timer arrive a few usec
before the time they are expected (according to the slow TSC).
This causes Linux to poll-idle, when it should be in an idle
power saving state.  The idle and clock code do not graciously
recover from this error, sometimes resulting in significant polling
and measurable power impact.

So stop using native_calibrate_tsc() for INTEL_FAM6_SKYLAKE_X.
native_calibrate_tsc() will return 0, boot will run with
tsc_khz = cpu_khz, and the TSC refined calibration will
update tsc_khz to correct for the difference.

This patch restores correctness.  Without it, all three of the
issues above occur.
```

图 8

该 commit 非常清楚指出，在 4.9 以后添加了一个宏定义 INTEL_FAM6_SKYLAKE_X，但因为搞错了该类型 CPU 的 crystal frequency 会导致该类型的 CPU 每 10 分钟慢 1 秒钟。

这时再看看我们的出问题的第二批宿主机 xeon bronze 3104 正好是 skylake-x 的服务器，影响 4.9-4.13 的内核版本，宿主机内核 4.10 正好中招。

并且 NTP 每次同步间隔 1024 秒约慢 1700ms，与 slowlog 异常完全吻合，而第一批次的机器 CPU 都不是 SKYLAKE-X 平台的，避开了这个 BUG，迁移之前 Redis 所在的物理机内核是 3.10 版本，自然也不存在这个问题。至此，终于解开上面三个疑惑。

五、总结

5.1 问题根因

通过上面的分析可以看出，问题根因在于内核 4.9-4.13 之间 skylake-x 平台 TSC 晶振频率的代码 BUG，也就是说同时触发这两个因素都会导致系统时钟变慢，叠加上 Redis 计时使用的 gettimeofday 会容易被 NTP 修改导致了本文开头诡异的 slowlog“异常”。有问题的宿主机内

核升级到 4.14 版本后，时钟变慢的 BUG 得到了修复。

5.2 怎么获取时钟

对于应用需要打点记录当前时间的场景，也就是说获取 Wall-Clock，可以使用 clock_gettime 传入 CLOCK_REALTIME 参数，虽然 gettimeofday 也可以实现同样的功能，但不建议继续使用，因为在新的 POSIX 标准中该函数已经被废弃。

对于应用需要记录某个方法耗时的场景，必须使用 clock_gettime 传入 CLOCK_MONOTONIC 参数，该参数获得的是自系统开机起单调递增的纳秒级别精度时钟，相比 gettimeofday 精度提高不少，并且不受 NTP 等外部服务影响，能准确更准确来统计耗时（java 中对应的是 System.nanoTime），也就是说所有使用 gettimeofday 来统计耗时（java 中是 System.currenttimemillis）的做法本质上都是错误的。

数据库篇

MySQL 锁之源码探索

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

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

一、MySQL 的架构与锁

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

二、元数据锁

2.1 元数据锁类型

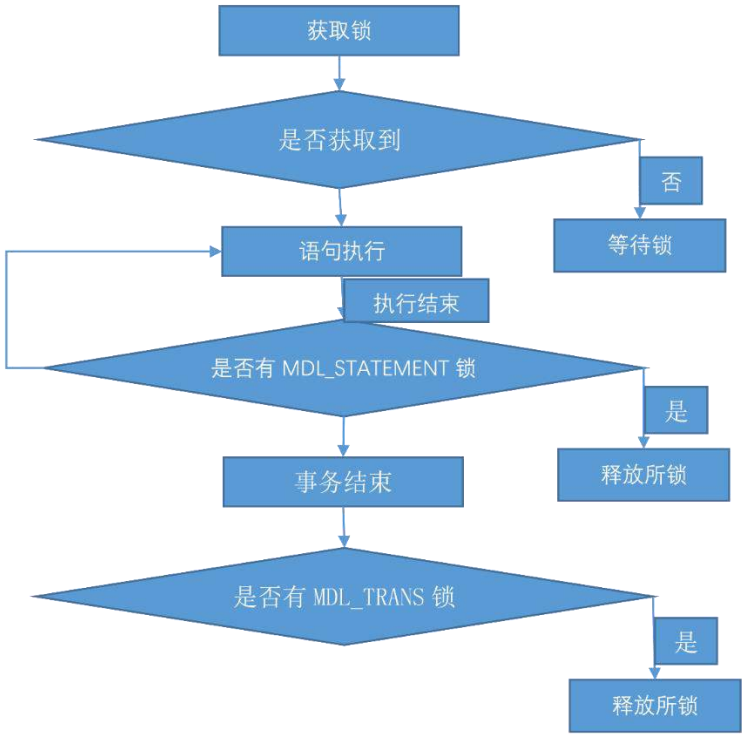
元数据类型	说明
GLOBAL	全局锁
TABLESPACE	表空间锁
SCHEMA	数据库锁
TABLE	表锁
FUNCTION	函数锁
PROCEDURE	存储过程锁
TRIGGER	触发器锁
EVENT	事件锁
COMMIT	事务锁，在服务器层进行提交事务时进行上锁
USER_LEVEL_LOCK	用户锁，通过GET_LOCK/RELEASE_LOCK进行获取和释放
LOCKING_SERVICE	安装locking_service.so插件后使用，为用户提供所服务

2.2 元数据锁申请与释放

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

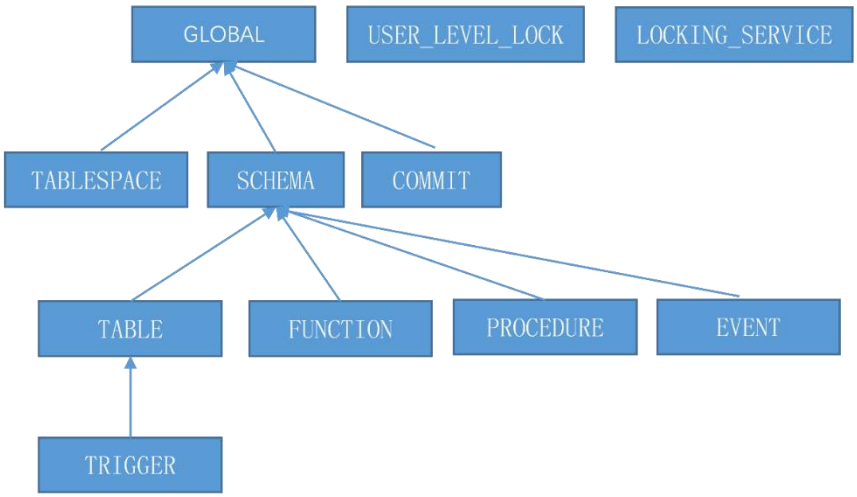
释放类型	说明
MDL_STATEMENT	语句执行结束时释放
MDL_TRANSACTION	事务结束时释放
MDL_EXPLICIT	显式释放

抽象元数据锁的上锁和释放的过程，整理为如下流程图



2.3 元数据锁关系

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



2.4 元数据锁级别

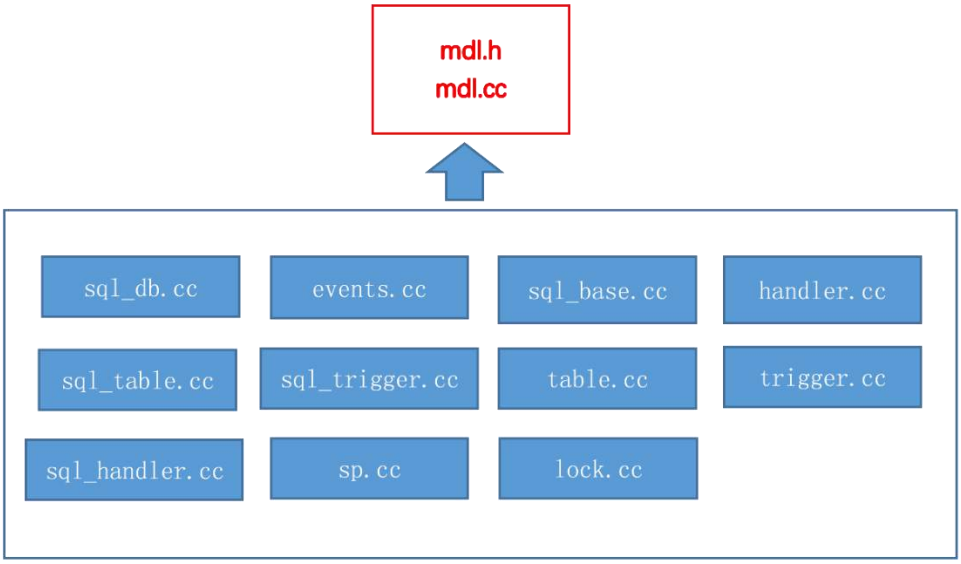
由于对元数据的访问存在不同的需求，因此设置不同级别锁级别，用于对元数据及数据的访问控制。

级别	说明
MDL_INTENTION_EXCLUSIVE（IX）	意向排它锁，可升级为排它锁。可与其他连接的意向排它锁兼容，但不兼容于共享锁。
MDL_SHARED（S）	共享锁
MDL_SHARED_HIGH_PRIO（SH）	高优先级共享锁。
MDL_SHARED_READ（SR）	共享读锁，对表数据存在意向读
MDL_SHARED_WRITE（SW）	共享写锁，对表数据存在意向写
MDL_SHARED_WRITE_LOW_PRIO	低优先级共享写锁
MDL_SHARED_UPGRADABLE（SU）	可升级共享锁
MDL_SHARED_READ_ONLY（SRO）	共享只读锁，该锁将阻塞对表的元信息和数据的更新
MDL_SHARED_NO_WRITE（SNW）	可升级的表锁，该锁将阻塞对表数据的更新，但允许进行表数据的读取
MDL_SHARED_NO_READ_WRITE（SNRW）	可升级的表锁，该锁将阻塞对表数据的读和更新。
MDL_EXCLUSIVE（X）	排它锁

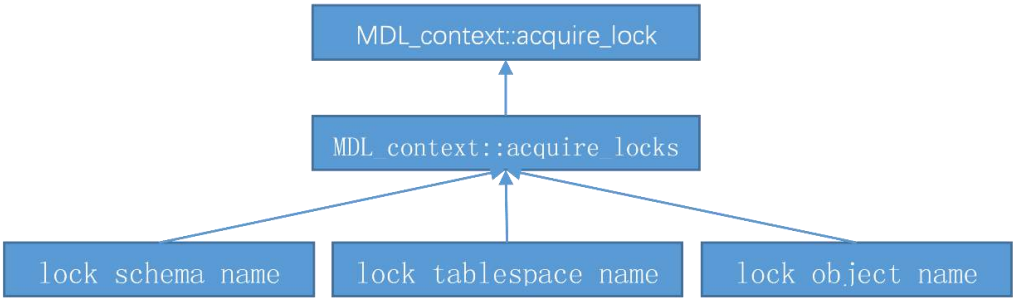
2.5 元数据锁源码

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

主要源文件及其关系如下，蓝框内所列源文件依赖于红框内的源文件所定义的内容。



主要函数如下图关系所列



三、InnoDB 锁

3.1 事务隔离级简介

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

InnoDB 事务采用不同的锁机制，会产生不同的现象。

现象	说明
脏读	此时事务忽略其他事务的任何锁，因此可以读取其他事务未提交的修改数据，当其他事务回滚数据时所读取的数据为错误数据。
不可重复读	在同一事务中，对同一行读取不同的值。此时读取数据时未对数据进行读取保护，故其他事务可修改该事务。
幻读	在同一事务中，使用相同的过滤条件，获取不同结果集。此时事务读取数据时，未对查询数据进行范围保护。

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

事务隔离级	脏读	不可重复读	幻读
读未提交	X	X	X
读提交	--	X	X
可重复读	--	--	X
串行化	--	--	--

3.2 InnoDB 的锁类型

InnoDB 为保护并发访问下的数据，根据不同的粒度对数据进行。

锁类型	说明
表锁	全表上锁，此锁对表中所有数据进行保护
行锁	单行数据进行保护
间隙锁	和行锁结合，对行数据进行范围上锁，对该范围数据进行保护

3.3 InnoDB 的锁级别

InnoDB 的锁

锁级别	说明
意向共享锁（IS）	用于对表锁，不能用于行锁
意向排它锁（IX）	用于对表锁，不能用于行锁
共享锁（S）	主要用于行锁，只有在“lock tables for read”时用于表锁
排它锁（X）	主要用于行锁，只有在“lock tables for write”时用于表锁
自增锁（AI）	表级锁，用于语句级的MySQL binlog

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

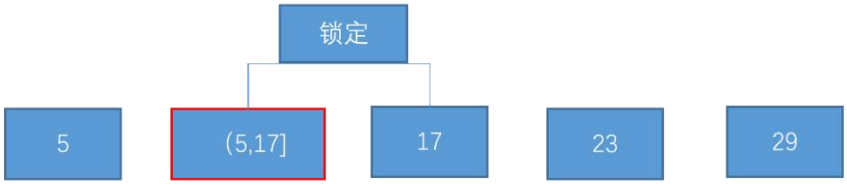
“O”表示兼容。

	IS	IX	S	X	AI
IS	O	O	O	X	O
IX	O	O	X	X	O
S	O	X	O	X	X
X	X	X	X	X	X
AI	O	O	X	X	X

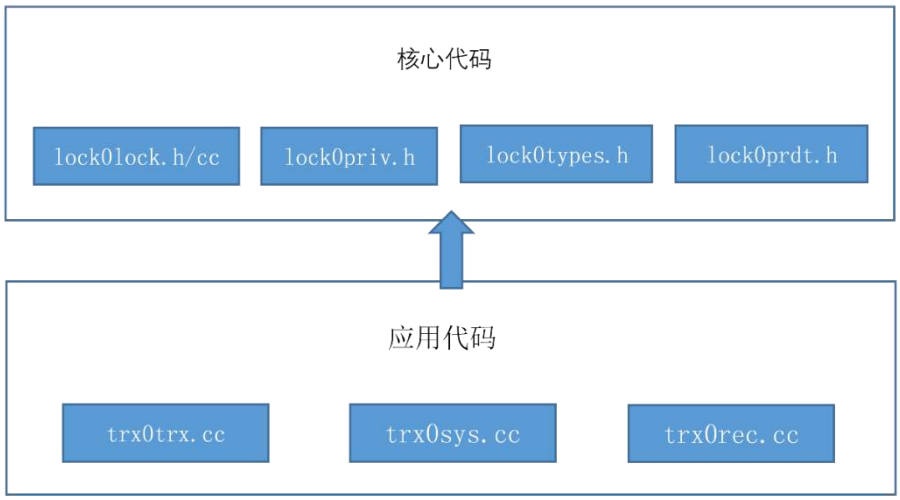
3.4 间隙锁

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

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



3.5 源码结构



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

携程酒店订单 Elastic Search 实战

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

一、业务场景

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

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

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

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

二、索引技术的选型

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

三、为什么没有使用数据库索引

数据库索引是一张表的所选列的数据备份。

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

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

四、开源搜索引擎的选择

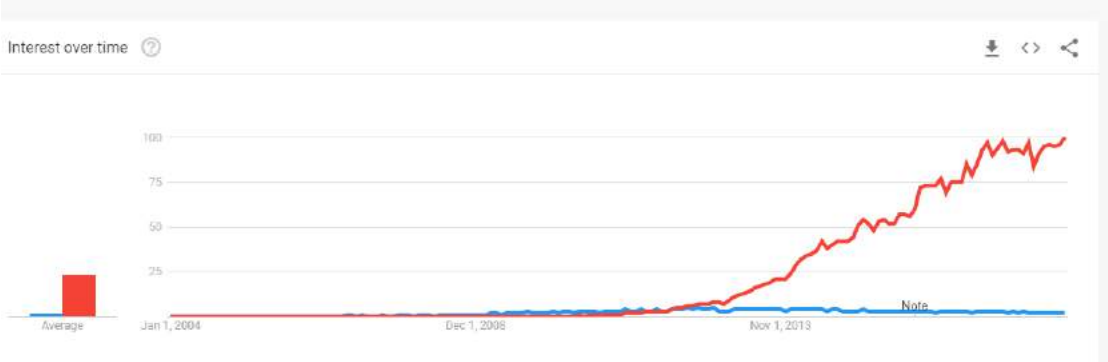
当时闪现在我们脑中的是开源搜索引擎 Apache Solr 和 Elastic Search。

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

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

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

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



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

五、复制分发的实现

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

六、为什么没有使用数据访问层复制分发

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

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

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

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

七、实时扫描数据库

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

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

八、提高 Elastic Search 写的吞吐量

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

- 使用 upsert 替代 select + insert/update。类似于 MySQL 的 replace into，避免多次请求，成倍节省多次请求带来的性能消耗。

- 使用 bulkrequest，把多个请求合并在一个请求里面。Elastic Search 的工作机制对批量请求有较好的性能，例如 translog 的持久化默认是 request 级别的，这样写硬盘的次数就会大大降低提高写的性能。至于具体一个批次多少个请求，这个与服务器配置、索引结构、数据量都有关系。可以使用动态配置的方式，在生产上面调试。

- 对于实时性要求不高的索引，把 index.refresh_interval 设置为 30 秒（默认是 1 秒）。这样可以让 Elastic Search 每 30 秒创建一个新的 segment，减轻后面的 flush 和 merge 压力。

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

据具体查询的场景，选择其中一个即可。

```
PUT index
{
  "mappings": {
    "type": {
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
      ]
    }
  }
}
```

对于不关心查询结果评分的字段，可以设置为 norms:false。

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "text",
          "norms": false
        }
      }
    }
  }
}
```

对于不会使用 phrase query 的字段，设置 index_options: freqs。

```
PUT index
{
  "mappings": {
    "type": {
      "properties": {
        "foo": {
          "type": "text",
          "index_options": "freqs"
        }
      }
    }
  }
}
```

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

九、提高 Elastic Search 读的性能

为了提高查询的性能，我们做了以下优化：

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

写：

```
PUT my_index/_doc/1?routing=user1&refresh=true ❶
{
  "title": "This is a document"
}

GET my_index/_doc/1?routing=user1 ❷
```

查：

```
GET my_index/_search
{
  "query": {
    "terms": {
      "_routing": [ "user1" ]
    }
  }
}
```

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

十、系统监控的实现

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

十一、灾备的实现

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

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

十二、总结

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

参考

1）Elastic Search 官方文档；
2）https://en.wikipedia.org/wiki/Database_index
3）https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B4%A2%E5%BC%95
4）https://logz.io/blog/solr-vs-elasticsearch/

携程数据库高可用和容灾架构演进

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

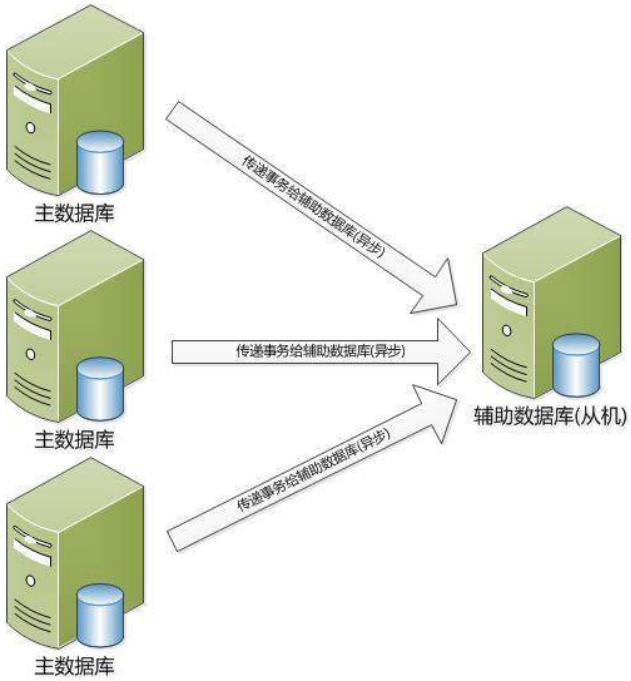
网站稳定性对任何一家公司来说都至关重要。业务长时间中断，不仅仅意味着收入的损失，更可能会失去客户。数据库的稳定性和 HA/DR 建设是网站高可用建设非常重要的一个环节。

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

一、1.0 时代【1999~2008】

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

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



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

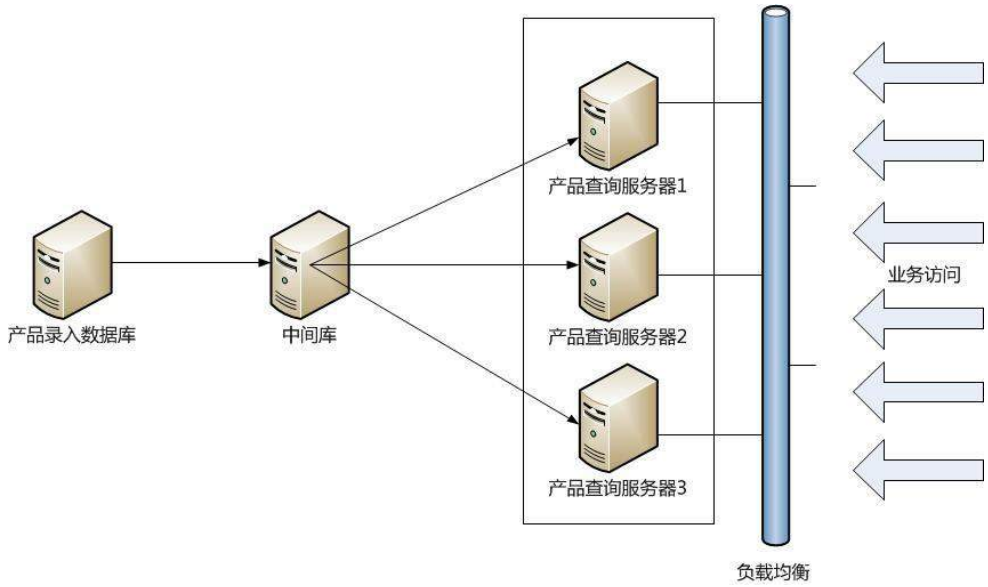
这种 HA 架构比较简单、粗糙，优点是不需要群集和共享存储等资源，成本低。缺点是主机故障时没法自动转移，业务恢复速度较慢，严格来讲还不具备 HA 的能力。

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

二、2.0 时代【2008~2012】

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

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

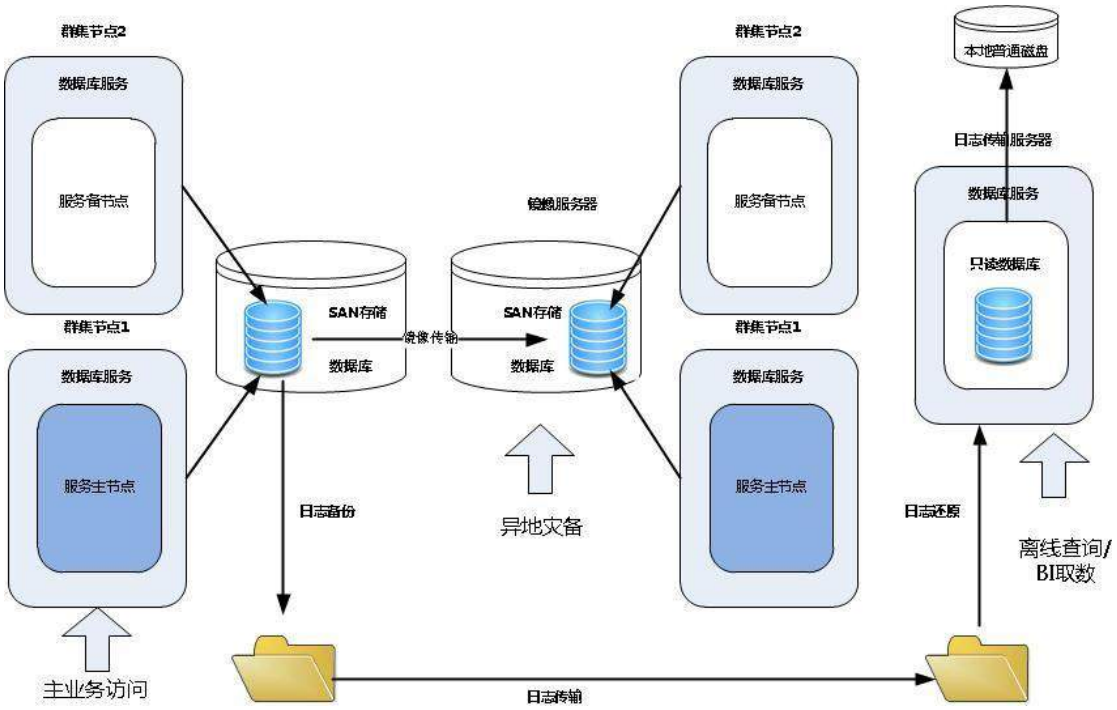


对于非读写分离场景的数据库，高可用是通过 Failover Cluster 群集方式。DR 依旧采用数据库镜像的方式。

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

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

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



由于业务的需求越来越复杂，数据库的架构开始逐渐复杂起来，主要体现在：

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

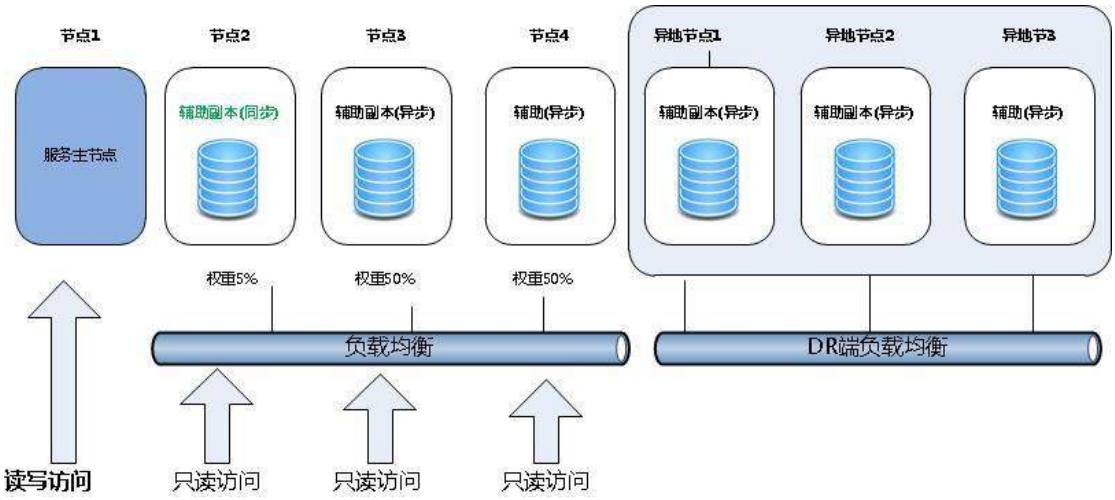
2）过于依赖底层的 SAN 存储。SAN 存储内含相当复杂的存储技术和网络技术，复杂一点的问题就需要依赖供应商来解决。

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

三、3.0 时代【2012~2014】

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

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



纯 AG 架构

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

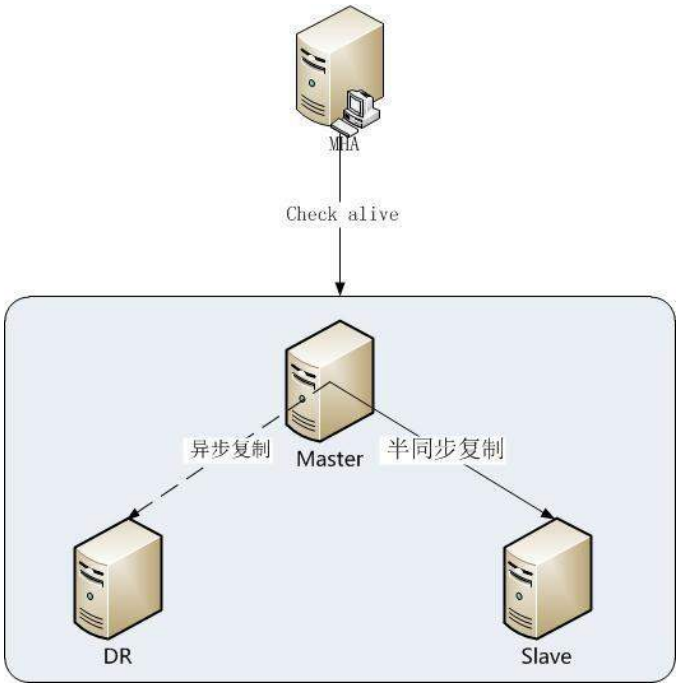
在推进 Always ON 新架构过程中，我们也逐步用 SSD 来取代原有的 SAN。除了 SSD 的价格越来越便宜容量越来越大，相比 SSD，SAN 存储对 DBA 基本是黑盒子，需要专业的运维团队支持，运维成本更高。

四、4.0 时代【2014~2018】

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

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

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



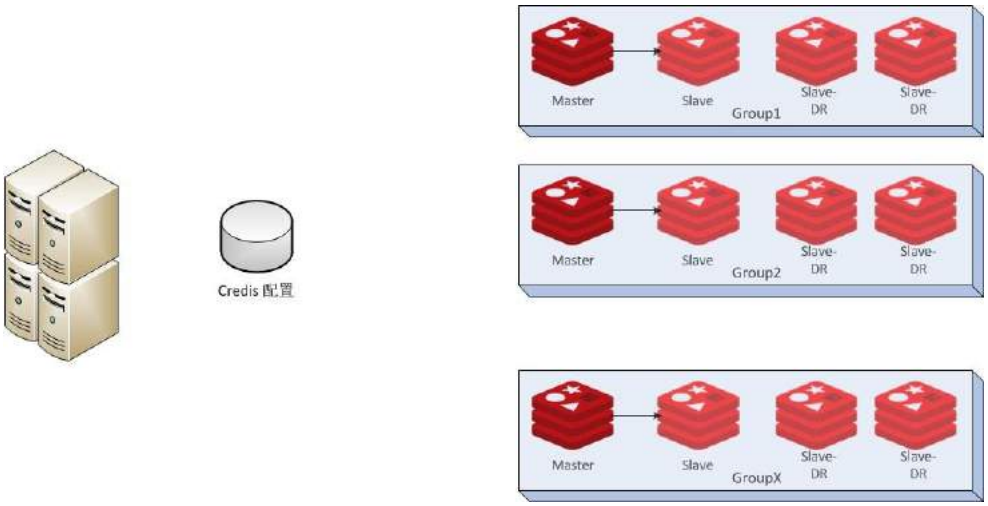
通过 MHA 管理节点，来监听主节点的可用性情况，一旦发现主机不可访问，则切换到 slave 节点。MHA 的高可用模式，既兼顾到高可用，也兼顾到 DR。对于读写分离的需求，主要是通过分库分表方案来进行的。部分有采用读写分离。

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

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

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

简要架构如下图所示：



说明如下：

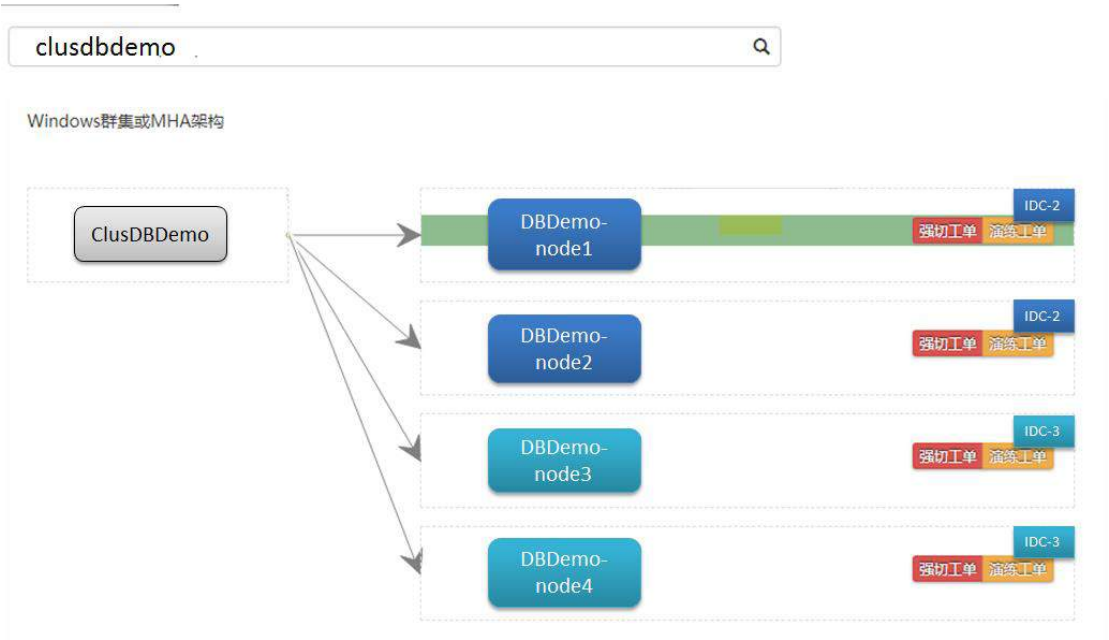
1）业务访问的时候，通过引用框架的客户端组件读取配置信息。Redis 切换由哨兵发起，并且 Credis 配置能及时捕获，并通知客户端 IP 和端口的变化。
2）对 Redis 拆分多个 Group，以避免访问热点。每个 Group 包含多个 Redis 实例。
3）Redis 实例 Master/Slave 在一个机房， 另外两个实例 Slave-DR 在另个机房。
4）访问请求通过 CRedis 配置路由到指定的分片。
5）Slave-DR 实例和主实例的数据同步通过框架同步工具 XPipe 来实现。XPipe 已经开源，更多信息请访问：https://github.com/ctripcorp/x-pipe。

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

DR 切换工具支持不同的切换维度，覆盖了所有的场景：

1）单个或多个数据库群集，应对单机故障或日常维护等场景；
2）单个业务线下所有数据库群集，应对 DR 切换演练场景；
3）IDC 下所有数据库群集，应对主 IDC 故障场景；

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



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

每个节点提供 2 种切换方式：

1）强切工单，强制切换。主机或主站 down 时，可能有数据丢失；
2）演练工单，正常切换。主机可用时，演练或计划内主机维护时使用；

生成的工单后续可以自动执行。

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

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

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

希望以上的分享对大家有所帮助。

风险控制篇

基于红黑树的高效 IP 归属地查询方案

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

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

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

实现方案：

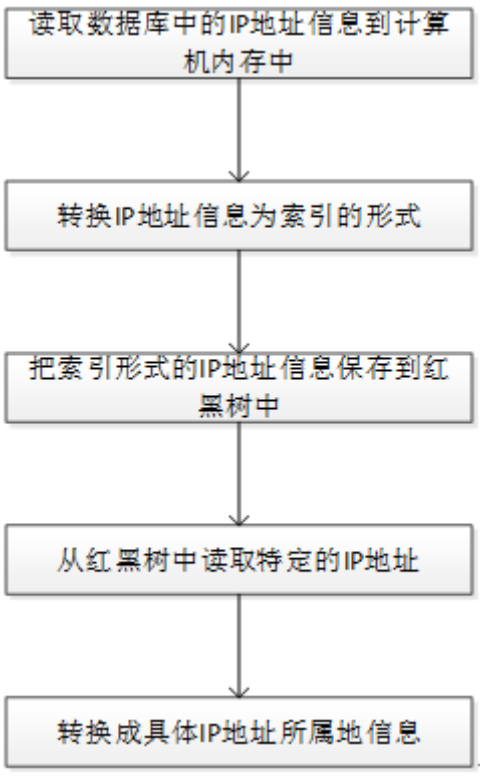


图 1

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

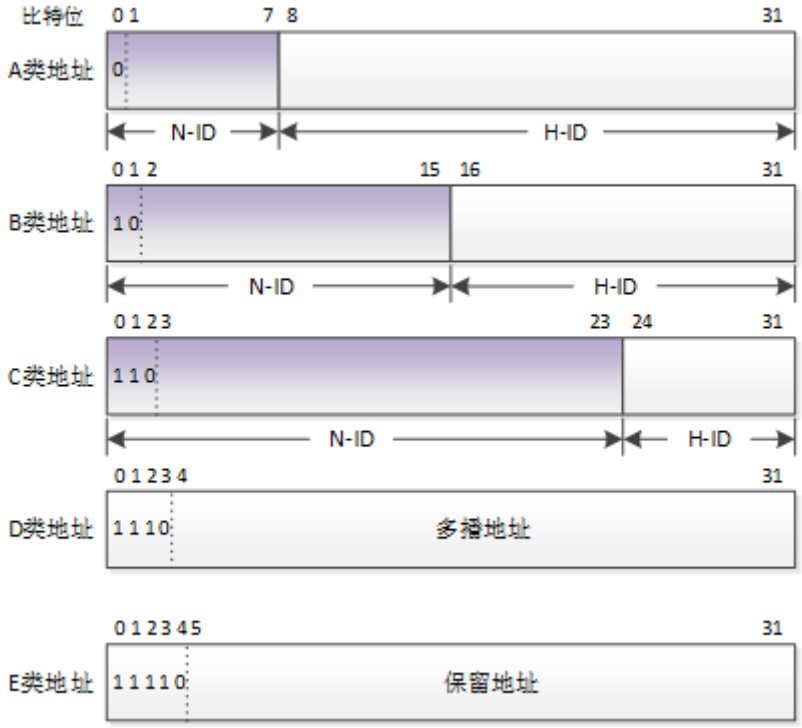


图 2

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

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

IP 地址是由 4 段组成的，如 1.15.168.0，其对应的 4 字节二进制形式为 00000001.00001111.10101000.00000000，根据计算机的计算特性，可以把第一段左移 24 位、第二段左移 16 位、第三段左移 8 位、第四段不移动，得到整数相加就是 IP 地址转换后的整数值，即 16777216 + 983040 + 43008 + 0 =17803264。

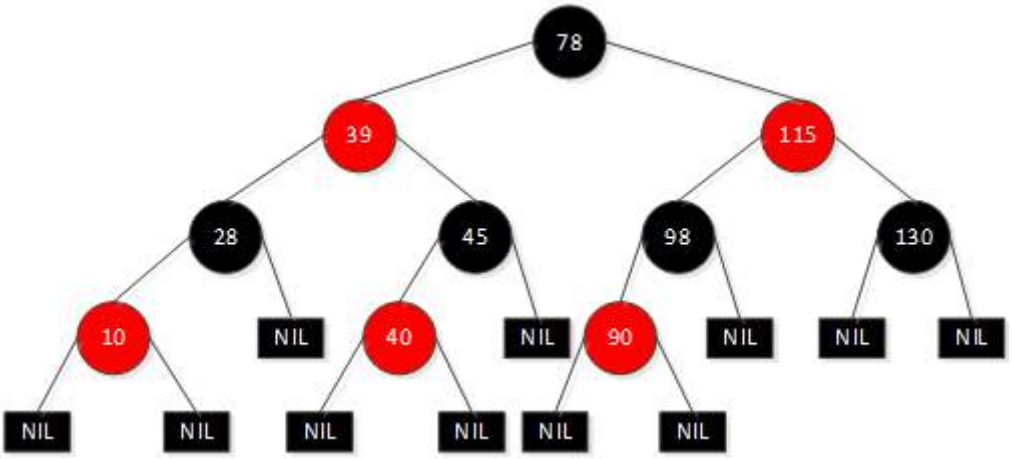


图 3

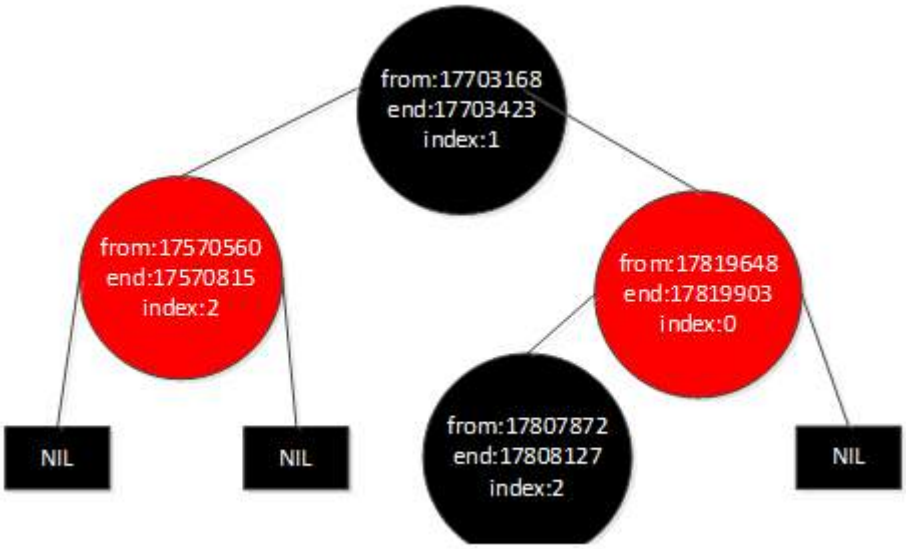


图 4

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

1、节点是红色或黑色。
2、根是黑色。
3、所有叶子都是黑色（叶子是 NIL 节点）。
4、每个红色节点必须有两个黑色的子节点。(从每个叶子到根的所有路径上不能有两个连续的红色节点。)
5、从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

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

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

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

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

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

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

表 1

type	name
1	中国
6	天津
2	辽宁
4	大连
6	北京

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

表 2

type	name	index
1	中国	0
6	天津	1
2	辽宁	2
4	大连	3
6	北京	4

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

表 3

国家索引	地区索引	城市索引	开始IP	结束IP
0	1	1	1.15.232.0	1.15.232.255
0	2	3	1.14.33.0	1.14.33.255
0	4	4	1.15.186.0	1.15.186.255
0	4	4	1.12.27.0	1.12.27.255

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

表 4

国家索引	地区索引	城市索引	fomatted-raw-ip索引
0	1	1	0
0	2	3	1
0	4	4	2

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

表 5

segment-regions-ip的索引	开始IP	结束IP
0	1.15.232.0	1.15.232.255
1	1.14.33.0	1.14.33.255
2	1.12.27.0	1.12.27.255
2	1.15.186.0	1.15.186.255

表 5 的 IP 地址转换成 long 整数。

表 6

segment-regions-ip的索引	开始IP	结束IP
0	17819648	17819903
1	17703168	17703423
2	17570560	17570815
2	17807872	17808127

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

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

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

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

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

该方案不仅适用于 IP 归属地查询，也适用于其他相对静态的数据的快速解析。

携程基于大数据分析的实时风控体系

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

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

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

一、性能和复杂度可以兼得

携程的风控系统，和大部分第三方支付平台一样，也是以实时风控系统为主：

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

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

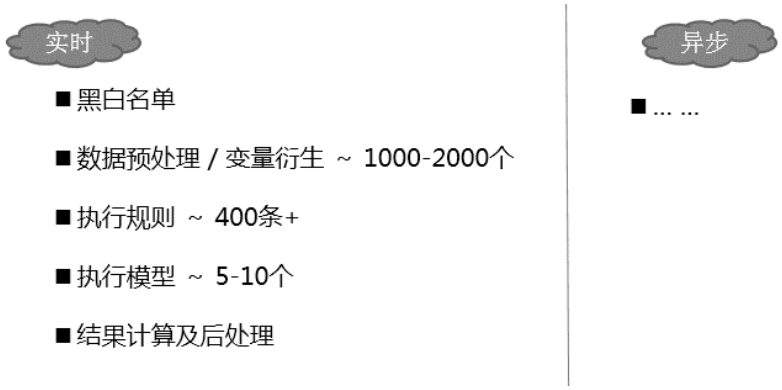
3、规则数量两年翻了五倍，同时规则使用更多的数据不再仅限于产品信息、支付信息、账号信息，行为数据等弱关联数据开始大量的应用于规则分析。

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

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

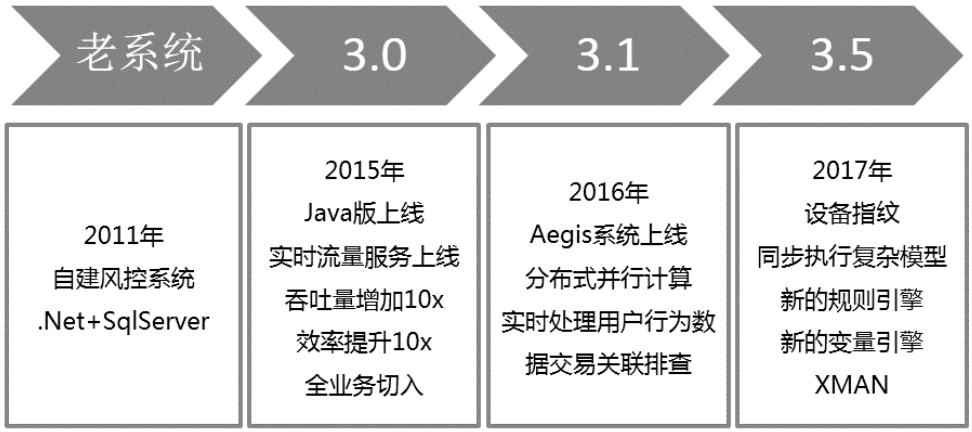
给大家看一些数据：

一笔支付请求背后，携程风控的规则计算复杂度：



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

二、携程风控架构变迁简史

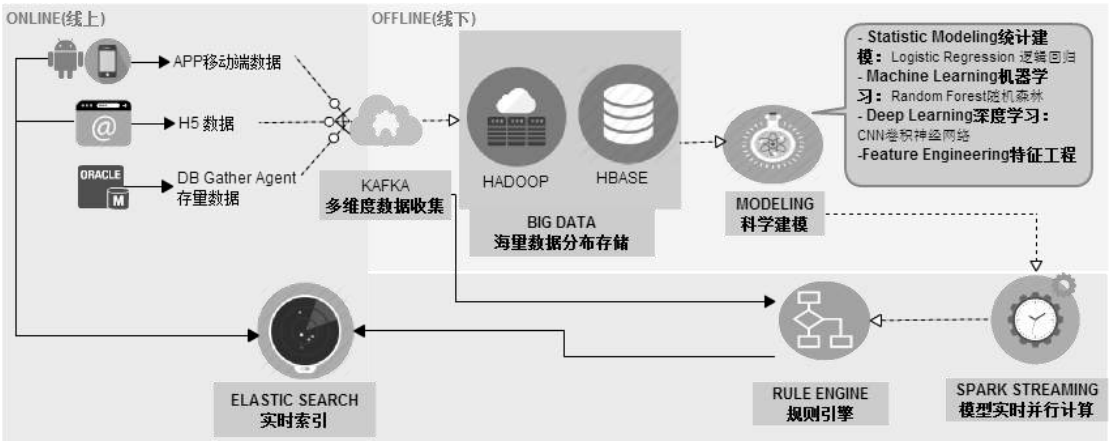


携程自建风控系统开始于 2011 年左右，直到 2015 年正好赶上公司技术栈从.Net 往 Java 平台转变，风控系统也迎来了一次完全的重写。

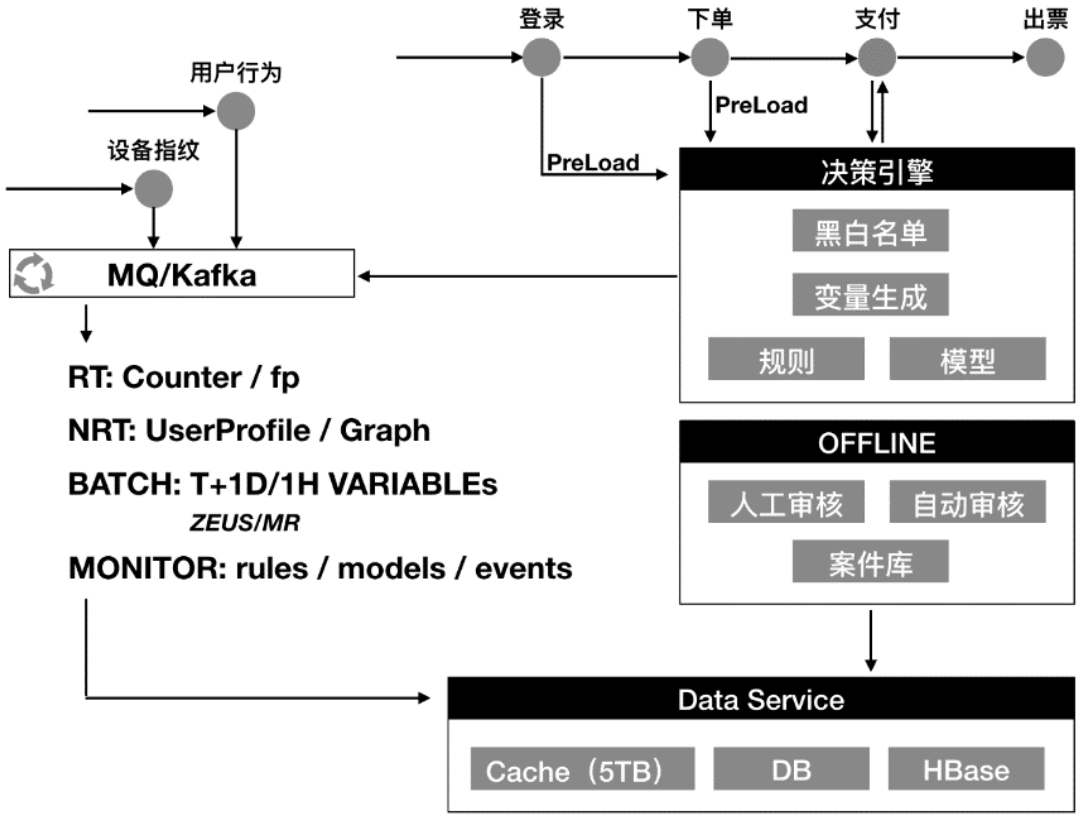
新系统的架构、设计复杂度、预计的处理能力也充分考虑了公司的业务发展预期，第一次让技术走在了业务到来之前。经过每年一个大版本的迭代，到目前为止，携程风控的技术水平已经处于行业第一梯队。

三、架构概述及核心服务

下面我们看看携程风控的架构实现：



上图可能有点抽象，我们看一个具体的例子：



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

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

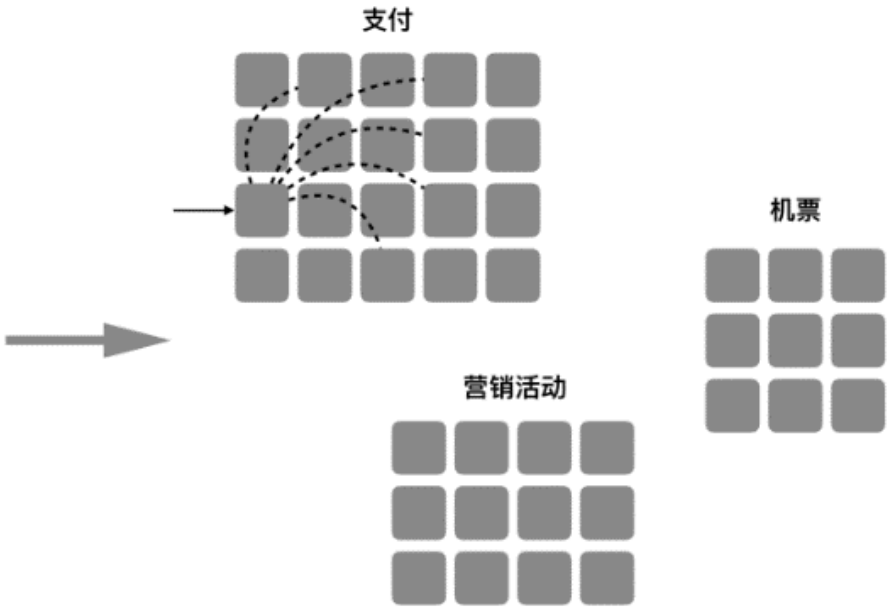
每天风控收集上来的数据超过 50 亿条，其中超过 1 亿左右的请求需要风控实时校验风险并返回给业务系统当前操作是否可以继续。

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

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

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

3.1 风控引擎



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

3.2 规则引擎

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

3.3 模型执行引擎

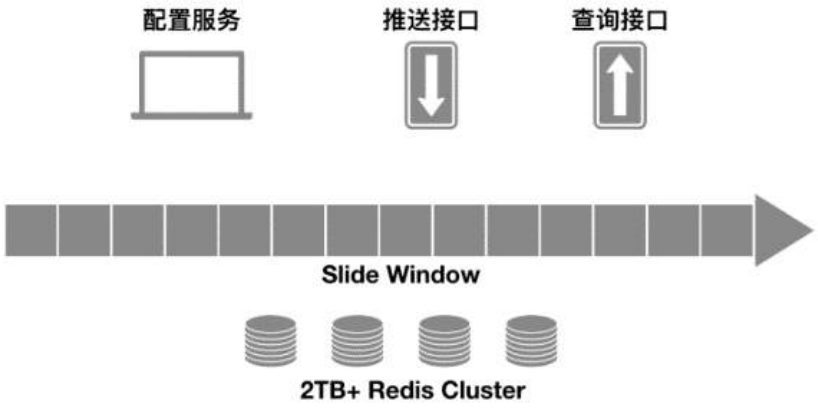
风控引擎支持把 SAS 或 SPARK 等工具训练出来的模型直接在风控系统中部署，支持 DOT 和 PMML 等多种格式。

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

	Python	JPMML	自主研发
特点	标准、开源， 兼容性好	标准、开源， 兼容性好。	使用Java解析并执行.dot模型文件， 支持随机森林和逻辑回归算法， 算法可扩展
性能	10-100ms， 因需要独立部署，有网络开销	性能和Python执行.dot接近，只是可以嵌入式运行，所以稳定性比Python高	0-10ms， 嵌入式执行，性能高，稳定性高

3.4 实时流量服务

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



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

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

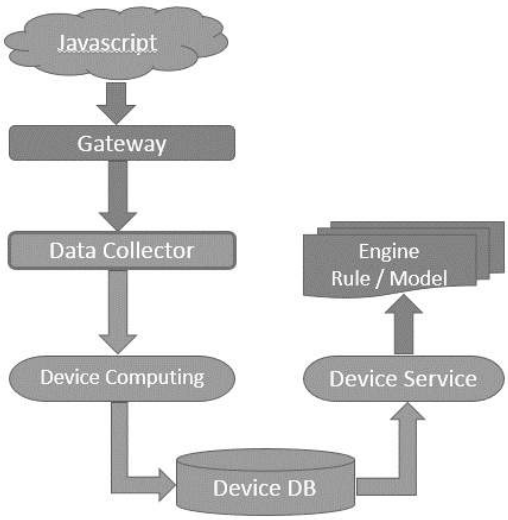
3.5 设备指纹

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

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

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

设备指纹的架构及关键指标：



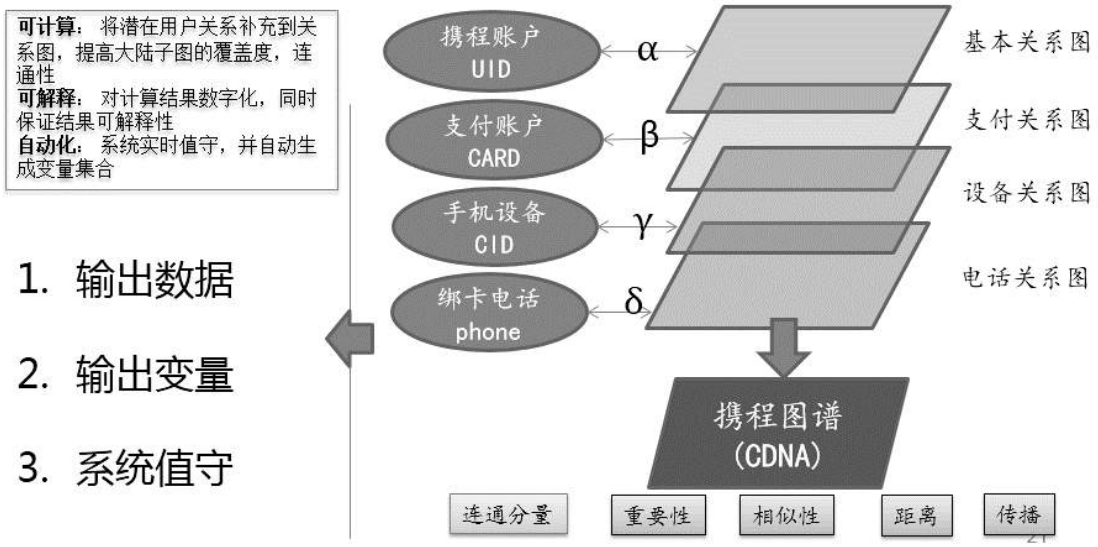
	错误率	准确率	稳定性
线上	1.2%	98.8%	1.52
实验	2.5%	97.5%	1.12

3.6 CDNA

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

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

通过 CDNA 对于发现新的欺诈特征很有帮助，让规则抓取更准确。



3.7 代理和模拟器识别

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

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

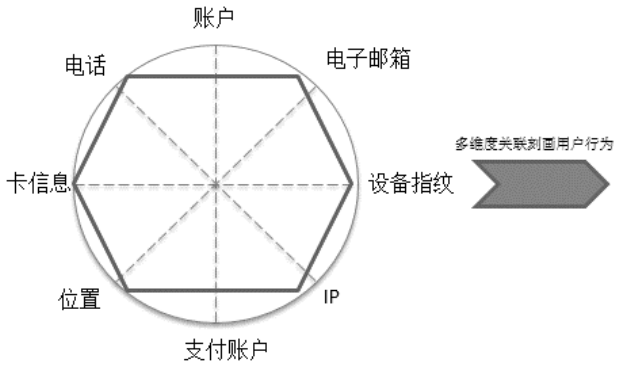
四、人工规则 vs 模型

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

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

简单介绍我们的特征变量提取方法：

变量衍生方法：



衍生方法	刻画pattern
基础衍生	高金额、快速起飞/入住等
冲突变量	信息不一致，例如发卡国和ip国
recency	账户年龄，最近一次交易
velocity（单、双主体）	频繁交易\换卡等
过滤条件velocity	频繁高危行为
ratio	高危行为占比，短期交易集中
个体异常	个体行为发生变化/异常
群体异常	行为相对于同地域人群异常
躲闪行为	行为有躲避风控规则的嫌疑
跳跃行为	小额试卡的行为
risktable	历史案件信息的利用

五、结束语

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

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