高德地图 amap.com　高德技术 amap_tech　高德技术　阿里云 开发者社区

高德技术 -2019年刊合辑

```
Programer *you;
while(you.believe) {
[you see];}
//因为相信，所以看见
```

六大篇章　人工智能/前端&移动/汽车工程
架构/数据/质量

高德技术出品

序

春节渐近，年味逾浓。

回首 2019 年，作为首个日活过亿的国民出行平台，高德地图 to C 和 to B 的用户数都再攀新高。老业务稳健增长，新业务突飞猛进。

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高德技术人。坚持技术高德，通过有温度的科技，为用户的出行生活带来改变。

2019 年是高德技术人脚踏实地“加油干事”的一年。

人工智能技术在高德地图全面落地，在视觉、搜索、导航、定位等业务场景发挥了越来越大的作用；客户端&移动、汽车技术、服务架构、数据研发、质量等技术领域，也实现了深化、融合、智能、创新和突破，为用户提供更准确、更高效的地图服务和交互体验。

- 视觉技术：快速发展和突破，不仅使自动化地图制作的质量和效率提升到了崭新水平，还打造了业界最好的自动化高精地图制作系统。积极探索新产品形态，开发和发布了全新体验的高德 AR 导航系统，在普通硬件上实现了精准直观的实景导航以及多项实用的辅助驾驶功能。

- 搜索推荐：聚焦用户。利用海量的时空出行及开放平台数据，建模用户的出行习惯，让目的地预测、需求预测更准确，搜索、推荐、语音在不同场景下更有人性化及精准。引入刻画真实世界的评价体系——POI 指数，完善 POI 深度知识图谱，初步建立吃住购行的百科全书，提升用户生活决策效率。

- 机器学习：强化数据和服务中台的能力建设，深耕数据和算法在业务的应用。构建地图领域特有的空间+时序深度模型，实现了数据质量和更新效率的绝对领先；出行服务领域，深度结合时空信息，引入高时效的交通事件，提升路况预测准确性；规划算法上利用多元化的出行数据及深度学习算法，开拓建成了全新的技术体系，使得路线规划具备了基于“人、时空、事件”的场景定制化能力。

- 定位技术：强化能力中台的角色和技术驱动的解题思路，在场景化的网络定位、DR 位置推算、GNSS 的质量识别等领域不断提升能力建设。

- 前端&移动：全面结合阿里经济体资源，打造高德独有的客户端分层架构，解决业务快速落地，重点强化云与端配合框架，端容器能力增强，共性业务有效识别。

- 汽车技术：深化工程服务能力，通过灵活高效的更新能力实现需求的快速迭代，攻克车载复杂的软/硬件环境。AutoSDK 对外开放全面启动，开启与行业共享、共创的新形态。汽车新业务领域持续探索，赋能车厂。在 EHP 高精地图应用、智能硬件等领域摸索出合适的打法，开始体系化建设汽车相关领域的创新和专业布局。

- 服务架构：在系统性能、工程效能、服务稳定性等方面开展架构升级，为业务的持续发展夯实基础。同时，共享出行业务聚合了全国各平台的出行运力，基于亿级活跃用户的位置及交通实时离线大数据联动，提供智能最优的全国运力调度，为国民提供智能便捷的出行服务。

- 地图数据研发：打造在线化、自动化、智能化的地图数据生产平台。数据驱动的生产管理能力已初见雏形。POI 数据，围绕生态化的创新模式，打通数据与用户和商家的直接关系，让 POI 变成“活”数据。高精地图，实现生产平台和高精采集车从零到一，场景从高速到普通路逐步覆盖，数据精度水平业界最高。

- 质量：坚持数据驱动、技术赋能的质量理念，实现了绝大部分测试工作平台化和自动化，全面向智能化前进。完成了线上性能数据采集、Metrics 监控、智能指标、全链路回放等多项技术的突破和落地。

这本书作为 2019 年高德技术的结晶，集合了全年度的重要技术文章，总计 45 篇，涵盖了人工智能、前端&移动、汽车工程、架构、数据、质量技术 6 个篇章，约 400 页。

大家在阅读时如果发现内容中有 BUG，或者希望对书中的技术问题深入探讨，或者有更好的建议想要交流，欢迎扫描文末二维码，及时通过高德技术微信公众号与我们联系。

如果这本书对你有帮助，欢迎分享给身边感兴趣的同事、朋友，一起切磋，共同成长。

衷心感谢大家一直以来的支持和陪伴！

未来，我们希望与业界展开更多技术分享和交流，推动交通出行科技的进一步发展。

凡是过往，皆为序章。2020，积蓄能量，一起冲！

最后，提前祝大家，新春快乐，阖家幸福。

高德技术微信公众号（amap_tech）

分享来自于高德技术的原创文章，发布技术活动、组织文化、热招岗位信息，和技术圈小伙伴一起学习成长。**关注公众号回复“年刊”获取电子书**

阿里云开发者社区

访问开发者社区，扫码领取更多免费电子书

阿里技术微信公众号（ali_tech）

关注阿里技术，扫码关注「阿里技术」获取更多资讯

目录

人工智能篇

机器学习在高德起点抓路中的应用实践

作者：安宁

导读

高德地图作为中国领先的出行领域解决方案提供商，导航是其核心用户场景。路线规划作为导航的前提，是根据起点、终点以及路径策略设置，为用户量身定制出行方案。

起点抓路，作为路线规划的初始必备环节，其准确率对于路线规划质量及用户体验至关重要。本文将介绍高德地图针对起点抓路准确率的提升，尤其是在引入机器学习算法模型方面所进行的一些探索与实践。

什么是起点抓路

首先，我们来简单介绍一下什么是起点抓路。起点抓路是指针对用户发起的路线规划请求，通过获取到的用户定位信息，将其起点位置绑定至实际所在的道路。

从高德地图 App 可以看到，用户进行路线规划时选择起点的方式有以下三种：

1.手动选点（用户在地图上手动标注所处位置）。

2.POI 选点（Point of Interest，兴趣点，在地理信息系统中可以是商铺、小区、公交站等地

理位置标注信息）。

3.自动定位（通过 GPS、基站或 WiFi 等方式自动定位所在位置）。

三种方式中，用户手动选点及 POI 选点这两种方式的位置信息相对准确，起点抓路准确率相对较高。

而自动定位起点的方式，由于受 GPS、基站、网络定位精度影响，定位坐标易发生漂移，定位设备抓取的位置与用户实际所处道路可能相差几米、几十米甚至几百米。如何在有限信息下，将用户准确定位到真实所在道路，就是我们所要解决的主要问题。

为什么要引入机器学习

引入机器学习之前，起点抓路对候选道路的排序采用了人工规则。核心思路是：以距离为主要特征，结合角度、速度等特征，加权计算得分，进而影响排序，人工规则中所涉及到的权重及阈值等是经综合实战经验人工拍定而成。

随着高德地图业务的不断增长，规划请求数量及场景的增多，人工规则的局限性越来越明显，具体表现在以下方面：

- 即使包含了众多经验在内，人工设定的阈值、权重仍不够完善，易发生偏移或存在盲区是不可改变的事实。
- 策略维护方面，面对上游数据的更新，新特征无法用最快速度加入到策略中。
- 人工规则拍定对经验要求较高，对于人员的更迭，很难做出最敏捷的响应。

在大数据和人工智能时代，利用数据的力量代替部分人力工作，实现流程的自动化，提高工作效率是必然趋势。

因此，基于起点抓路人工规则的现状及问题，我们引入了机器学习模型，自动学习特征与抓路结果之间的关系。一方面，拥有大量规划及实走数据，对于机器学习模型的训练数据获取，高德有天然优势；另一方面，机器学习模型有更强的表达力，能够学习到特征之间的复杂关系，提高抓路准确率。

如何实现机器学习化

回归机器学习本身，下面来介绍我们如何建立起点抓路机器学习模型。一般来讲，运用机器学习方法解决实际问题分为以下几个方面：

- 目标问题的定义
- 数据获取与特征工程
- 模型选择
- 模型训练及效果评估

1.目标问题定义

在引入机器学习模型之前，需要将待解决问题进行数学抽象。

分析起点抓路问题，如上图所示，我们可以看到当用户在 A 点发起路线规划请求时，其定位位置 A 所对应的周边道路是一个独立的集合 B，而用户所在的实际道路是这个集合中的唯一一个元素 C。

这样，起点抓路问题转化为在定位点周边道路集合中选出一条最有可能是用户实际所在的道路。

整个过程类似搜索排序，因此，我们在制定建模方案时也采用了搜索排序的方式。
1.提取用户路线规划请求中的定位信息 A。
2.对定位点周边一定范围内的道路进行召回，组成备选集合 B。
3.对备选道路进行排序，最终排在首条的备选道路为模型输出结果，即用户实际所在道路 C。

最终，我们将起点抓路定义为一个有监督的搜索排序问题。明确了需要达到的目标，我们开始考虑数据获取及特征工程问题。

2.数据获取与特征工程

业界常言，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。可见对于项目最终效果，数据和特征至关重要。

训练起点抓路机器学习模型，我们需要从原始数据中获取两类数据：

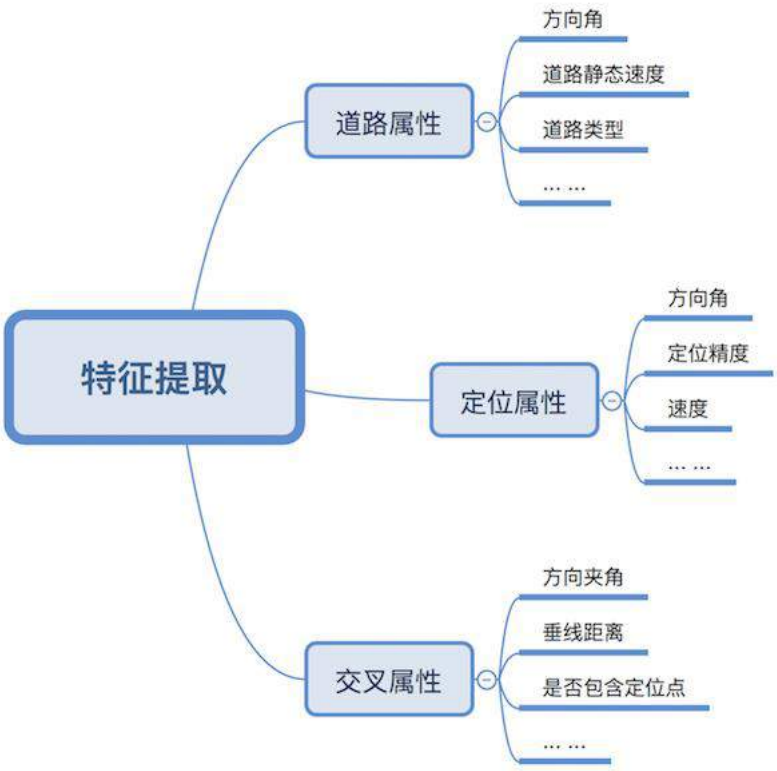
- 真值数据，即用户发送路线规划请求时实际所处道路信息。

机器学习应用于起点抓路项目，第一个问题就是真值数据的获取。用户在某个位置 A 发起路线规划请求，由于定位精度限制，我们无法确认其实际所在位置，但如果用户在发起规划请求附近有实走信息，可以将实走信息匹配到路网生成一条运动轨迹，通过这条轨迹我们就可以获取到请求定位点所处的实际道路。
我们针对高德地图的导航请求数据进行相关挖掘，将用户实走与路线规划信息相结合，得到

了请求与真值一一映射的数据集。

- 特征数据

在起点抓路模型中，我们提取了三大类特征用于构建样本集，分别是定位点相关特征、道路自身特征以及定位点与道路之间的组合特征。



特征处理是特征工程的核心部分，不同项目在进行特征预处理时会有不同，需要根据实际业务场景进行特殊化处理，往往依赖于专业领域经验。起点抓路项目中，我们针对定位特征进行了样本去重、异常值处理、错误值修正及映射等数据清洗工作。

3.模型选择

在目标问题定义中，我们将起点抓路剖析为搜索排序问题，而机器学习的 ranking 技术，主要包括 point-wise、pair-wise、list-wise 三大类。

根据起点抓路业务特点，我们采用了 list-wise，其 learning to rank 框架具有以下特征：

- 输入信息是同一路线规划请求对应的所有道路构成的多特征向量（即一个 query）。
- 输出信息是对应请求（即同一 query）特征向量的打分序列。
- 对于打分函数，我们采用了树模型。

我们选择 NDCG（Normalized Discounted Cumulative Gain 归一化累积折算信息增益值）作为模型评价指标，NDCG 是一种综合考虑模型排序结果和真实序列之间关系的指标，也是常用的衡量排序结果的指标。

4.模型训练及效果评估

我们抽取了一定时间段内的请求信息，按照步骤 2 中描述的方式获取到对应真值及特征数据，打标构建了样本集，将其划分为训练集与测试集，训练模型并查看结果是否符合预期。

评估模型效果，我们将测试集的请求分别用人工规则及机器学习模型进行抓路，并分别与真值进行对比，统计准确率。

对比结果，针对随机抽取的请求，模型与人工规则抓路结果差异率为 10%，这 10%的差异群体中，模型抓路准确率比人工规则提升 40%，效果显著。

写在最后

以上我们介绍了大数据和机器学习在起点抓路方面的一些应用，项目的成功上线也验证了机器学习在提升准确率、优化流程等方面可以发挥重要作用。

未来，我们希望能够将现有模型场景继续细化，寻找新的收益点，从数据和模型两个角度共同探索，持续优化机器学习抓路效果。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

深度学习在高德驾车导航历史速度预测中的探索与实践

作者：沐亦

1.导读

驾车导航服务是数字地图提供的核心功能。通常而言，用户在发起导航之前会对比高德前端展示的三条路线（如下图），以决定按照哪条路线行驶。

而预估到达时间是用户参考的最为重要的指标之一。给定一条路线，对应的预估到达时间的计算需要两组信息输入，分别是实时路况信息和历史速度信息（历史速度信息指的是对应的平均通行时间）。其中实时路况信息，对短时（例如 60 分钟以内）路况预测帮助较大；而历史速度信息对长时（例如 60 分钟以上）路况预测帮助较大。一般来说，对于未来 2 小时以上的路况预测而言，当前时刻的路况信息帮助十分有限，也可以理解为历史速度信息基本处于绝对主导的地位。

因此在长距离路线的预估到达时间计算中，历史速度至关重要，其预测的准确性直接影响预估到达时间，进而影响用户选择及体验。我们希望能够通过建模的方式提高历史速度的

预测准确率。

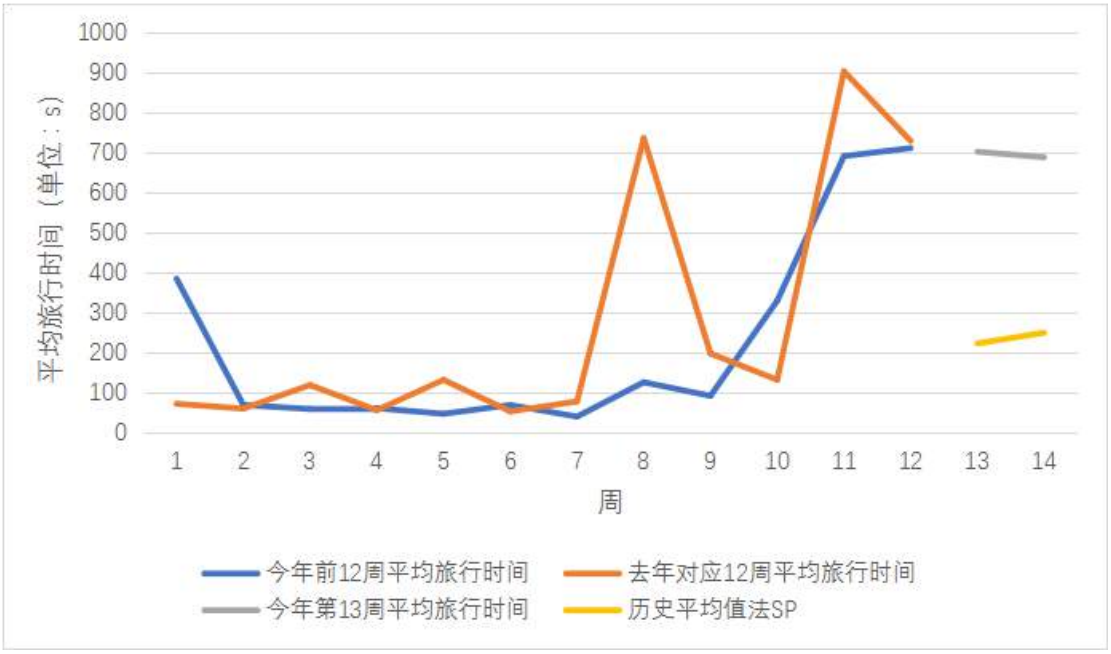
2.历史平均法的不足

以往预测历史速度的方式是历史平均值法，主要是将历史上过去某时间段同一特征日同一时间段（例如 8:00-8:10）经过同一条路段的所有车辆的用时求平均，这种方法的假设是“历史即未来”。该方法对于 3 个月内常发性震荡走势比较适合，但针对有特定趋势的走势（如上升走势），效果不会太好。

该方法不足之处有以下三点：

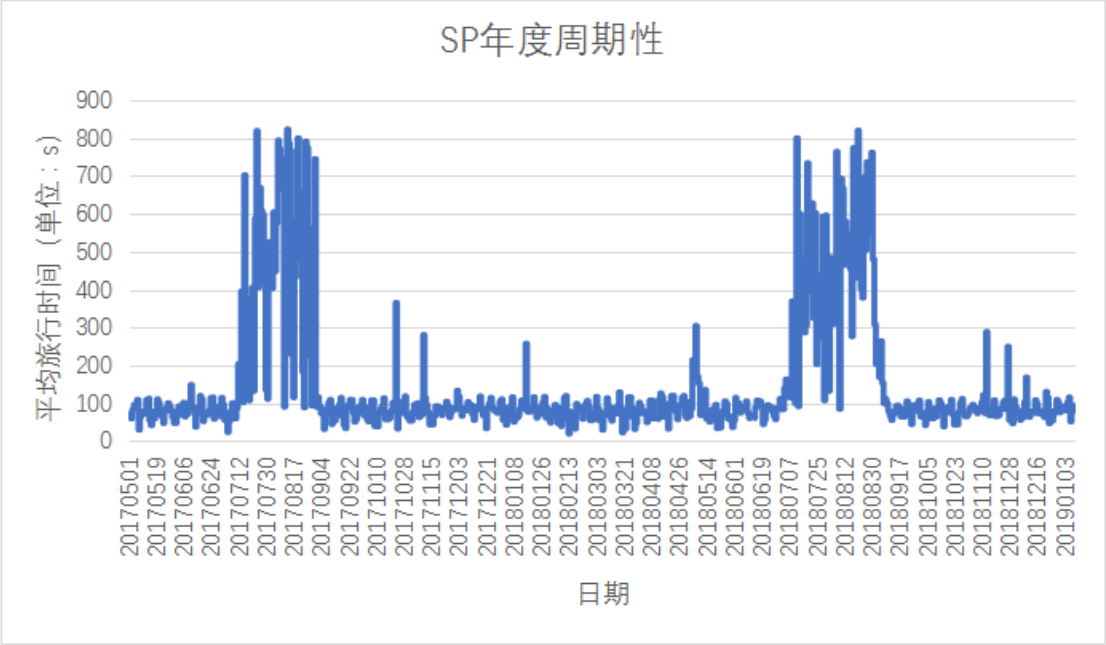
- 对于异常点敏感
- 无法利用时域序列的演化趋势（trend）信息
- 无法利用去年同期的车辆行驶规律

下面我们用一个分析过的 badcase 来具体说明，如下图：



上图显示了北京市某路段在过去连续若干周里确定特征日（周五）及确定时间批次（12:10-12:20）上各自的平均旅行时间。如图所示，近期旅行时间已逐渐升高，且去年同期旅行时间也已经升高，但是使用历史平均值法计算的历史速度信息却显著偏小，与未来一周对应时间段的真实旅行时间偏差近三倍。

通过前期的 case 调研分析，发现部分路段的历史速度曲线呈现出时效性、年度周期性特征。



上图为某段道路从 20170501 到 20190103 期间某个时间批次的旅行时间变化，可以发现，每到暑假期间，该路段旅行时间显著增大，呈现出明显的年度周期性。

本项目旨在解决历史平均值法存在的不足，其中，由于年度周期性问题导致的恶劣 badcase 占比较大，故着重解决年度周期性问题。

3.机器学习解题

3.1 基于 TCN 模型的历史速度问题建模

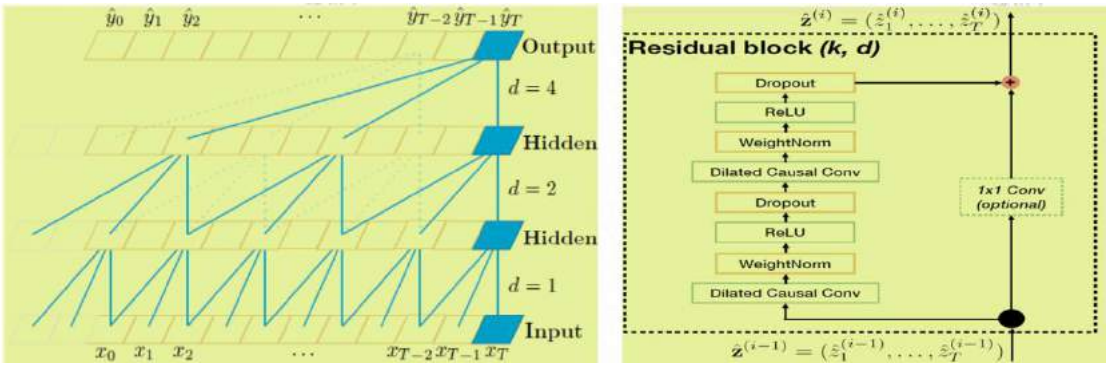
历史平均值方法简单粗暴，但也能取得相当不错的效果，对于具有年度周期性&时效性的路段，仅仅通过统计近期的信息会损失一定的精度，这时把去年的相关信息融合进来就显得特别重要，该问题是个典型的时序建模问题，本文基于 TCN 构建历史速度周期性问题解决方案。

我们的目标是构建一个基于历史信息（某时间段&去年同期：同一段道路、确定特征日、确定时间批次）和道路属性来预测未来一周历史速度的机器学习模型，解决历史平均值法存在的问题，从历史速度信息维度提高预估到达时间的准确率，解决恶劣 bacase。

3.2 TCN 简介

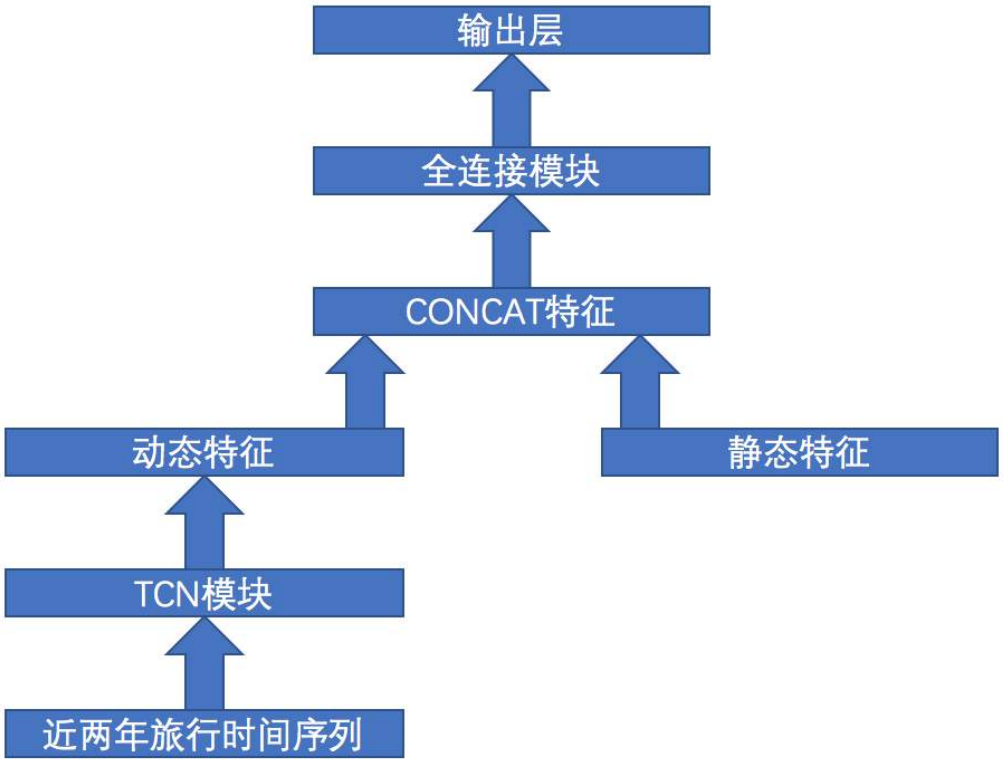
时间卷积网络（TCN）可以作为一般的序列建模架构，且拥有非常好的效果。TCN 显著的特点有如下几点：

- 架构中的卷积存在因果关系，这意味着从未来到过去不会存在信息泄漏。
- 卷积架构可以将任意长度的序列映射到固定长度的序列。
- 利用残差模块和空洞卷积来构建长期依赖关系。



TCN 论文图：TCN 架构的组成元素。左图为空洞系数 d=1, 2, 4、卷积核大小 k=3 的空洞因果卷积，感受野能覆盖输入序列中的所有值。右图为 TCN 残差块，当残差输入和输出有不同的维度，我们会添加一个 1x1 的卷积。

1.3 网络架构



上图为整个模型的框架图，主要分为动态特征提取模块和静态特征模块，其中动态特征的提取基于 TCN 模型实现，而静态特征则直接和提取出的动态特征进行连接之后使用。具体说明见下文。

3.4 动态特征提取

该模块的主要目标是通过 TCN 模型去学习旅行时间的曲线走势特征，这里的动态特征指的是从今年和去年对应的一定数量的平均旅行时间构成的序列中提取出的走势特征。

本文将今年和去年对应的平均旅行时间序列作为一个双通道序列放进 TCN 模型中学习，旨在利用 TCN 强大的时序建模能力，同时结合今年和去年的走势特征，更加准确的预测未来一周的走势（上升、下降或震荡）。

针对该类序列建模问题，已有成熟的 RNN 技术，而且目前更新的 TCN 技术也已出现。在项目开展过程中，分别使用了 RNN、LSTM 和 TCN 来做序列建模，实验结果表明，使用 TCN 进行序列特征提取效果最好，相对于 RNN 约有 1.39%的效果收益，相对于 LSTM 约有 0.83%的效果收益，而且由于 TCN 模型是基于卷积网络实现的，训练速度更快，所以本项目中采用 TCN 进行动态特征提取。

3.5 静态特征

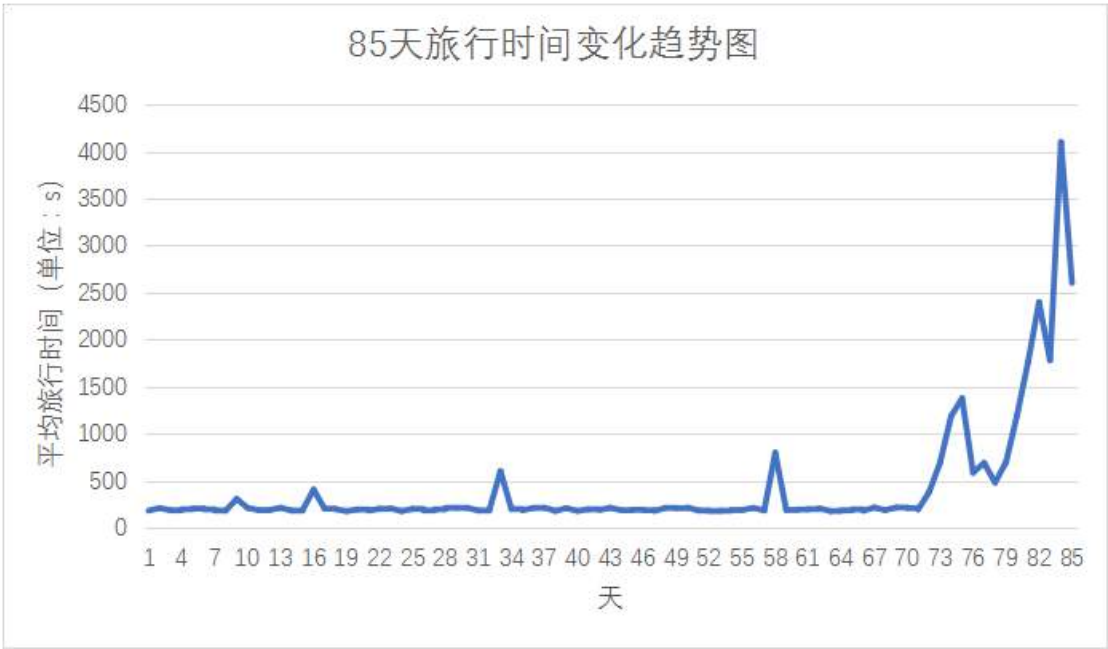
这里的静态特征主要指一些人工提取的特征，用以加强模型表达能力。具体如下：

道路属性特征	路长、路宽、车道数、车道宽度、最大限速等
时间属性特征	前三天对应时间批次旅行时间、前七天对应时间批次旅行时间均值、去年同期前后两个平均旅行时间（同一特征日&同一时间批次）

道路属性特征主要考虑不同的道路通行能力不一样，会在一定程度上影响车辆通行速度。

时间属性特征主要分三方面：

- 前三天对应时间批次旅行时间-考虑到时效性问题，越靠近预测天则可靠性越大，例如，最近几天道路施工，车道速度受到影响；但该特征也会对基于特征日的建模产生一定影响，因为车辆出行规律大部分情况下跟周几特征日有关，而与非特征日的关联较小（除特别情况，如活动举办、连续数天施工）。案例见下图。



图中为某段道路连续 85 天某时间批次上的旅行时间变化趋势图，从图中可以看到，该曲线呈现先平稳后上升的趋势，第 85 天的旅行时间与前面的关系不大，而与近期关系较大（近期一直较拥堵），尤其是近三天。

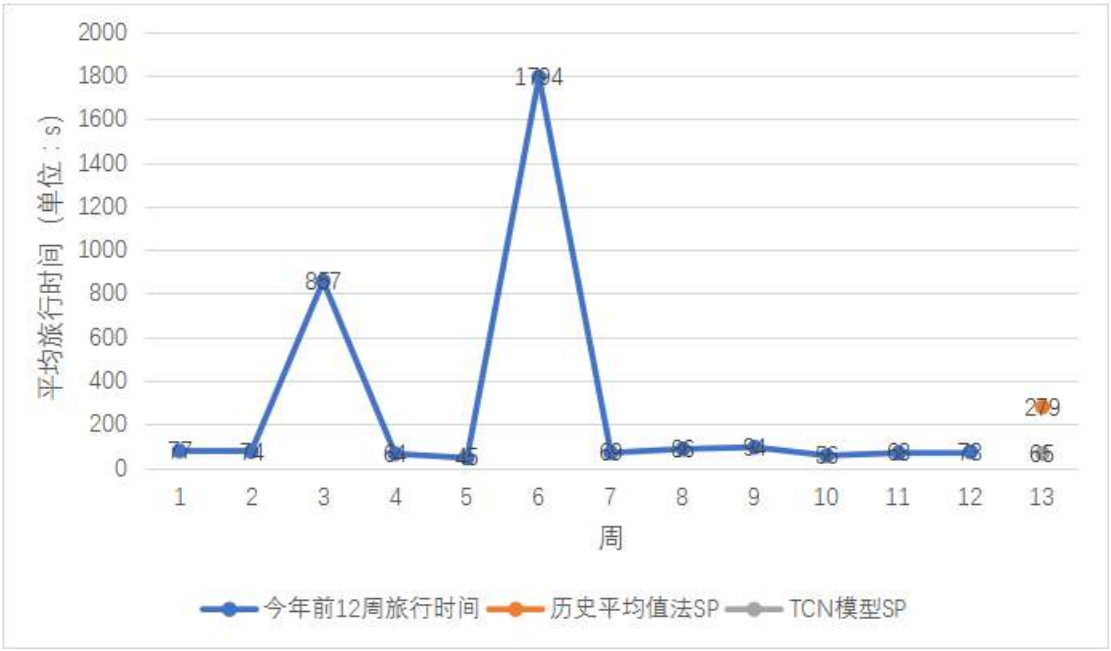
- 前七天对应时间批次旅行时间均值-该特征为近期旅行时间的一个统计指标，旨在反映近一周的通行状况，作用同上。

- 去年同期前后两个平均旅行时间（同一特征日&同一时间批次）。

4.模型效果

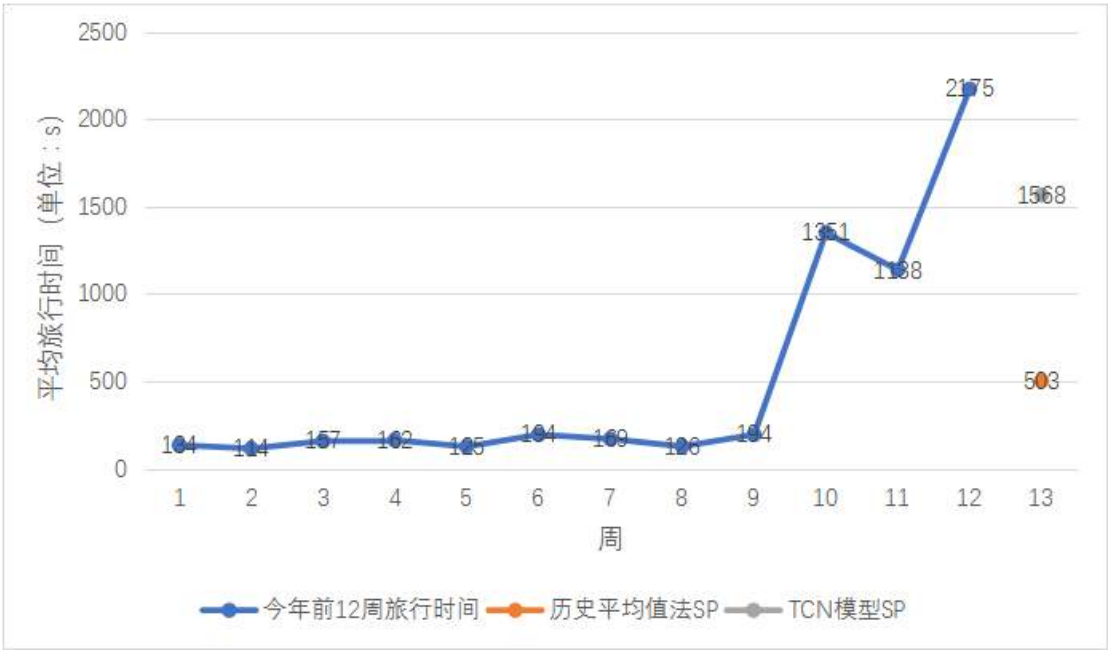
本文所采用的基于 TCN 建模方法，能够从动态和静态特征中提取出驾驶规律信息，包括异常点的识别过滤、旅行时间的趋势变化信息和年度周期信息，给出更符合预期的预测值，较好的解决当前历史平均值法的弊端，可以部分解决恶劣 badcase 发生的问题。具体效果说明如下：

4.1 异常值自动过滤



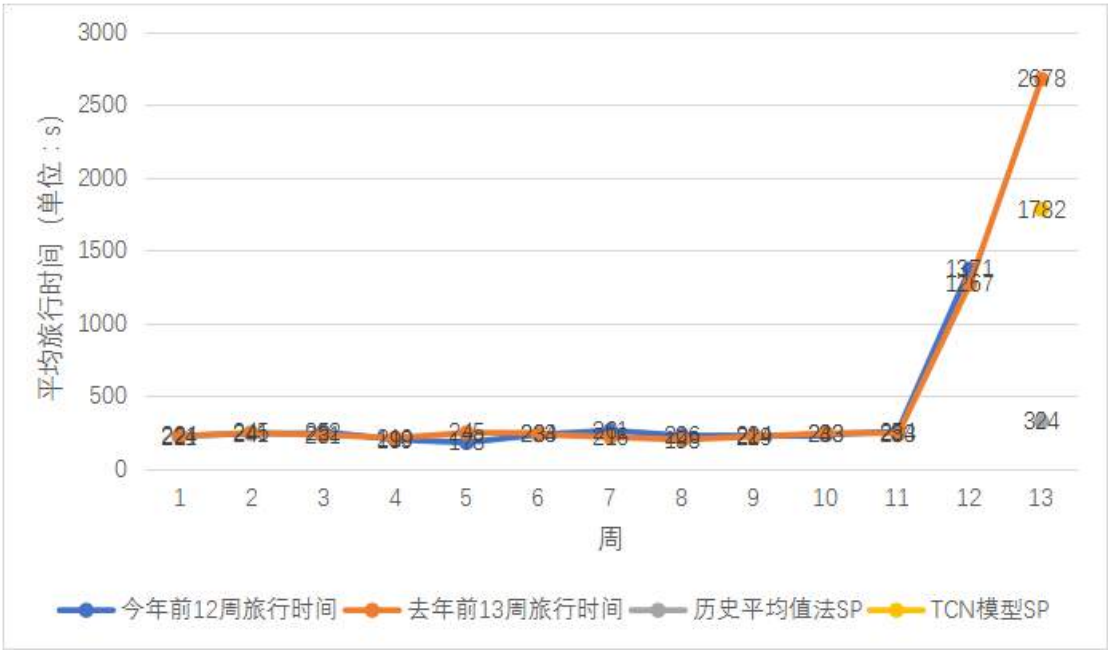
上图中 case，从整体上来看，旅行时间在 70s 上下浮动，而局部某些点是异常偏大的（当天可能发生了交通事故等），属于异常点，在预测未来历史速度信息走势的时候，应该忽略，TCN 模型成功的忽略了这些异常点，但历史均值法则会将其纳入计算，导致算出的平均旅行时间偏大。

4.2 趋势信息提取



上图中 case，从曲线走势来看，旅行时间近期有所上升，所以下周的平均旅行时间大概率还是会延续这种趋势，可以看出，TCN 模型比较好的学习到了这种趋势信息，预测效果较好，但历史平均值法，由于前期很长一段时间旅行时间都很小，导致算出的平均旅行时间也偏小。

4.3 年度周期性的引入



上图中 case，从图中可以看出，今年前 11 周都比较平稳，第 12 周旅行时间突然上升，但只从今年信息中我们无法得知第 12 周这天是否是异常值，从而模型无法准确给出第 13 周的预测值，但从去年对应的 13 周的数据中可以发现，去年对应的时间在第 12 和 13 周都上升了，从而模型可以确定今年第 13 周大概率还是会继续升高（根据年度周期性），但使用历史平均值法给出的平均旅行时间则明显偏小。

5.评测结果

该项目在某一周的 case 集合上的评测效果：

- 基线-历史平均值法的 badcase 率为 11.0‰；
- 对照-基于 TCN 的方法的 badcase 率为 10.1‰。

可以看出，本文所采用的方法相对基线恶劣 badcase 率下降幅度较大，说明引入年度周期性可以解决部分恶劣 badcase。

6.小结

本文将 TCN 模型进行工业化实践，帮助建模历史速度问题，并结合特征工程(提取动态、静态特征，引入年度周期性等)，成功的解决了现有模型的不足，并在实际应用中取得了不错的效果，为将来的时序性问题探索了一条可行的路径。对现有 TCN 模型框架所做改动较小，后续可进一步探索，针对特定问题做一些定制化的改进。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

机器学习在高德搜索建议中的应用优化实践

作者：雪糯，星泉

导读

高德的愿景是：连接真实世界，让出行更美好。为了实现愿景，我们要处理好 LBS 大数据和用户之间的智能链接。信息检索是其中的关键技术，而搜索建议又是检索服务不可或缺的组成部分。

本文将主要介绍机器学习在高德搜索建议的具体应用，尤其是在模型优化方面进行的一些尝试，这些探索和实践都已历经验证，取得了不错的效果，并且为后来几年个性化、深度学习、向量索引的应用奠定了基础。

1.对搜索排序模块做重构

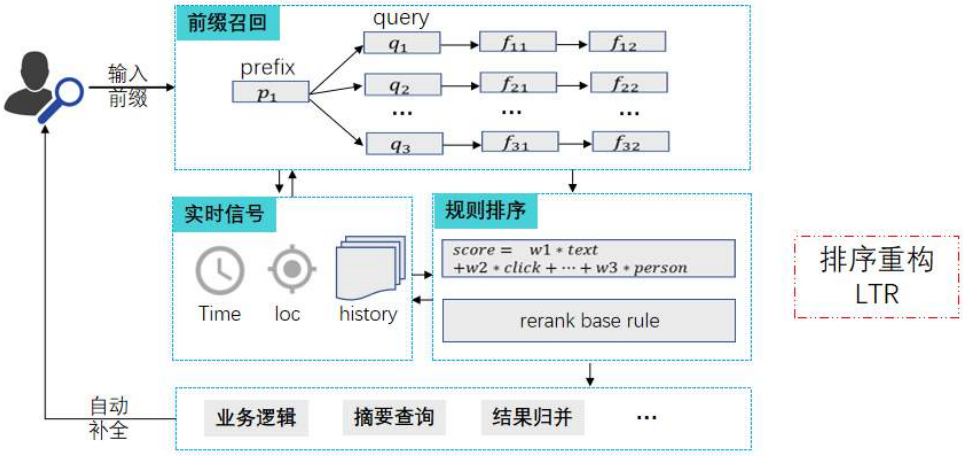
搜索建议（suggest 服务）是指：用户在输入框输入 query 的过程中，为用户自动补全 query 或 POI（Point of Interest，兴趣点，地理信息系统中可以是商铺、小区、公交站等地理位置标注信息），罗列出补全后的所有候选项，并进行智能排序。

我们希望通过 suggest 服务：智能提示，降低用户的输入成本。它的特点是：响应快、不承

担复杂 query 的检索，可以把它理解为一个简化版的 LBS 领域信息检索服务。

和通用 IR 系统一样，suggest 也分为 doc(LBS 中的 doc 即为 POI)的召回和排序两个阶段。其中，排序阶段主要使用 query 和 doc 的文本相关性，以及 doc 本身的特征（weight、click），进行加权算分排序。

但随着业务的不断发展、特征规模越来越大，人工调参逐渐困难，基于规则的排序方式已经很难得到满意的效果。这种情况下，为了解决业务问题，将不得不打上各种补丁，导致代码难以维护。



因此，我们决定对排序模块进行重构，Learning to Rank 无疑是一个好的选择。

2.面临的挑战：样本构造、模型调优

Learning to Rank（LTR）是用机器学习的方法来解决检索系统中的排序问题。业界比较常用的模型是 gbrank，loss 方案用的最多的是 pair wise，这里也保持一致。一般应用 LTR 解决实际问题，最重要的问题之一就是如何获得样本。

首先，高德地图每天的访问量巨大，这背后隐藏的候选 POI 更是一个天文数字，想要使用人工标注的方法去获得样本明显不现实。

其次，如果想要使用一些样本自动构造的方法，比如基于用户对 POI 的点击情况构建样本 pair <click, no-click>，也会遇到如下的问题：

- 容易出现点击过拟合，以前点击什么，以后都给什么结果。
- 有时，用户点击行为也无法衡量真实满意度。
- suggest 前端只展示排序 top10 结果，更多的结果没有机会展现给用户，自然没有点击。
- 部分用户习惯自己输入完整 query 进行搜索，而不使用搜索建议的补全结果，统计不到这部分用户的需求。

对于这几个问题总结起来就是：无点击数据时，建模很迷茫。但就算有某个 POI 的点击，却也无法代表用户实际是满意的。

最后，在模型学习中，也面临了特征稀疏性的挑战。统计学习的目标是全局误差的一个最小化。稀疏特征由于影响的样本较少，在全局上影响不大，常常被模型忽略。但是实际中一些中长尾 case 的解决却往往要依靠这些特征。因此，如何在模型学习过程中进行调优是很重要。

3.系统建模过程详解

上一节，我们描述了建模的两个难题，一个是样本如何构造，另一个是模型学习如何调优。先看下怎么解决样本构造难题，我们的解决方案是：

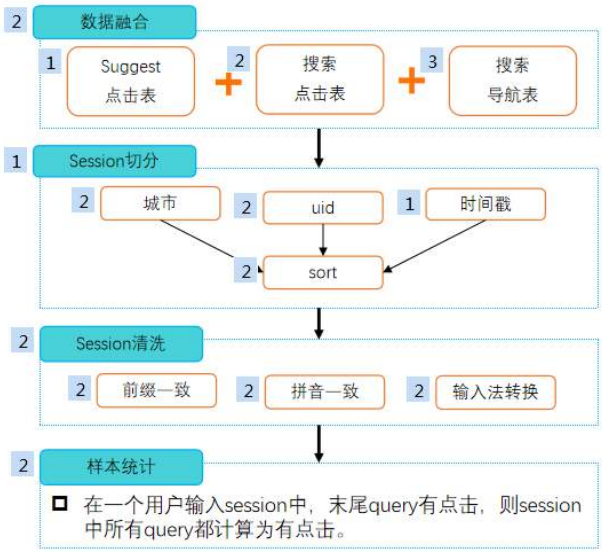
- 考量用户在出行场景的行为 session，不光看在 suggest 的某次点击行为，更重要的是，考察用户在出行场景下的行为序列。比如 suggest 给出搜索建议后，继续搜索的是什么词，出行的地点是去哪里，等等。

- 不是统计某个 query 下的点击， 而是把 session 看作一个整体，用户在 session 最后的点击行为，会泛化到 session 中的所有 query 上。

详细方案

第一步，融合服务端多张日志表，包括搜索建议、搜索、导航等。接着，进行 session 的切分和清洗。最后，通过把输入 session 中，末尾 query 的点击计算到 session 中所有 query 上，以此满足实现用户输入 session 最短的优化目标。

如下图所示：



最终，抽取线上点击日志超过百万条的随机 query，每条 query 召回前 N 条候选 POI。利用上述样本构造方案，最终生成千万级别的有效样本作为 gbrank 的训练样本。
特征方面，主要考虑了 4 种建模需求，每种需求都有对应的特征设计方案：

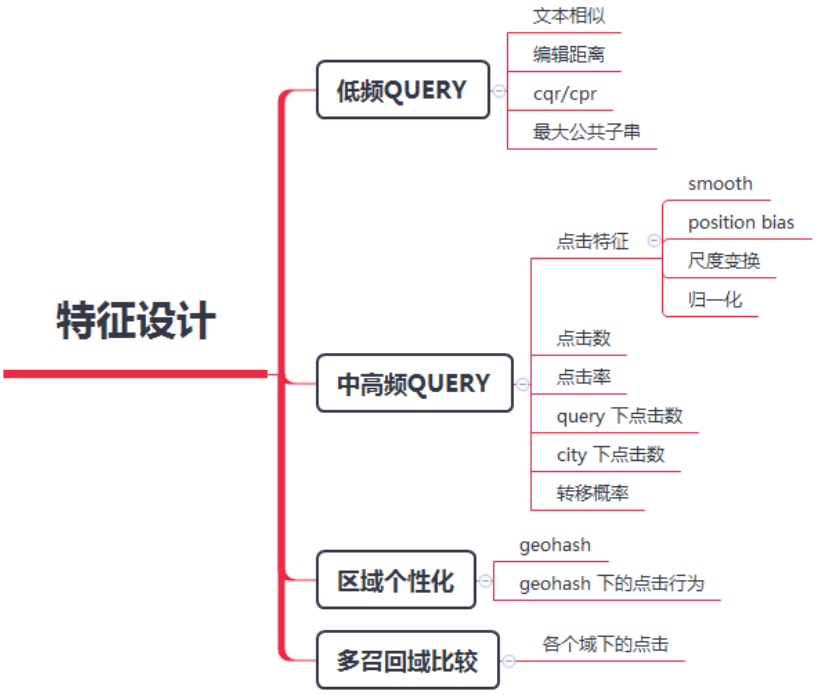
- 有多个召回链路，包括：不同城市、拼音召回。因此，需要一种特征设计，解决不同召回链路间的可比性。

- 随着用户的不断输入，目标 POI 不是静态的，而是动态变化的。需要一种特征能够表示不同 query 下的动态需求。

- 低频长尾 query，无点击等后验特征，需要补充先验特征。

- LBS 服务，有很强的区域个性化需求。不同区域用户的需求有很大不同。为实现区域个性化，做到千域千面，首先利用 geohash 算法对地理空间进行分片，每个分片都得到一串唯一的标识符。从而可以在这个标识符（分片）上分别统计特征。

详细的特征设计，如下表所示：



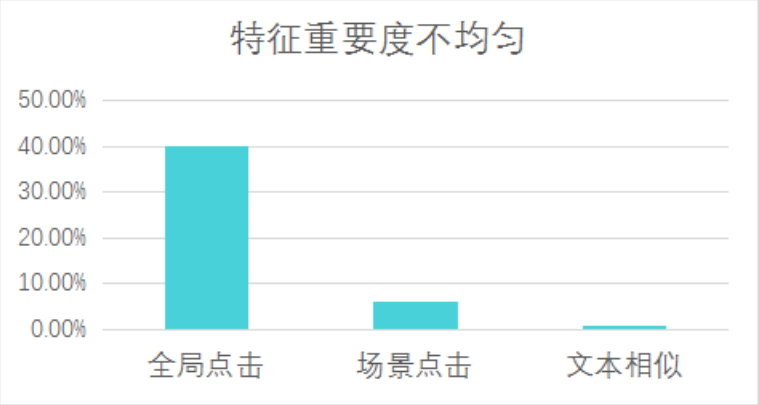
完成特征设计后，为了更好发挥特征的作用，进行必要的特征工程，包括尺度缩放、特征平

滑、去 position bias、归一化等。这里不做过多解释。

初版模型，下掉所有规则，在测试集上 MRR 有 5 个点左右的提升，但模型学习也存在一些问题，gbrank 特征学习的非常不均匀。树节点分裂时只选择了少数特征，其他特征没有发挥作用。

以上就是前面提到的，建模的第二个难题：模型学习的调优问题。具体来就是如何解决 gbrank 特征选择不均匀的问题。接下来，我们详细解释下。

先看下，模型的特征重要度。如下图所示：



经过分析，造成特征学习不均衡的原因主要有：

- 交叉特征 query-click 的缺失程度较高，60%的样本该特征值为 0。该特征的树节点分裂收益较小，特征无法被选择。然而，事实上，在点击充分的情况下，query-click 的点击比 city-click 更接近用户的真实意图。

- 对于文本相似特征，虽然不会缺失，但是它的正逆序比较低，因此节点分裂收益也比 city-click 低，同样无法被选择。

综上，由于各种原因，导致树模型学习过程中，特征选择时，不停选择同一个特征（city-click）作为树节点，使得其他特征未起到应有的作用。解决这个问题，方案有两种：

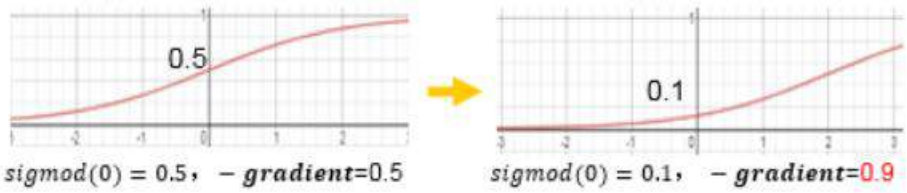
- 方法一：对稀疏特征的样本、低频 query 的样本进行过采样，从而增大分裂收益。优点是实现简单，但缺点也很明显：改变了样本的真实分布，并且过采样对所有特征生效，无法灵活的实现调整目标。我们选择了方法二来解决。

- 方法二： 调 loss function。按两个样本的特征差值，修改负梯度（残差），从而修改该特征的下一轮分裂收益。例如，对于 query-click 特征非缺失的样本，学习错误时会产生 loss，调 loss 就是给这个 loss 增加惩罚项 loss_diff。随着 loss 的增加，下一棵树的分裂收益随之增加，这时 query-click 特征被选作分裂节点的概率就增加了。

具体的计算公式如下式：

$$-\boldsymbol{gradient} = y_{ij} - \frac{1}{1+\exp\left(-\left(h(x_i)-h(x_j)\right)+loss_{diff}\right)}$$

以上公式是交叉熵损失函数的负梯度，loss_diff 相当于对 sigmod 函数的一个平移。



差值越大，loss_diff 越大，惩罚力度越大，相应的下一轮迭代该特征的分裂收益也就越大。

调 loss 后，重新训练模型，测试集 MRR 在初版模型的基础又提升了 2 个点。同时历史排序 case 的解决比例从 40%提升到 70%，效果明显。

4.写在最后

Learning to Rank 技术在高德搜索建议应用后，使系统摆脱了策略耦合、依靠补丁的规则排序方式，取得了明显的效果收益。gbrank 模型上线后，效果基本覆盖了各频次 query 的排序需求。

目前，我们已经完成了人群个性化、个体个性化的建模上线，并且正在积极推进深度学习、向量索引、用户行为序列预测在高德搜索建议上的应用。

招聘

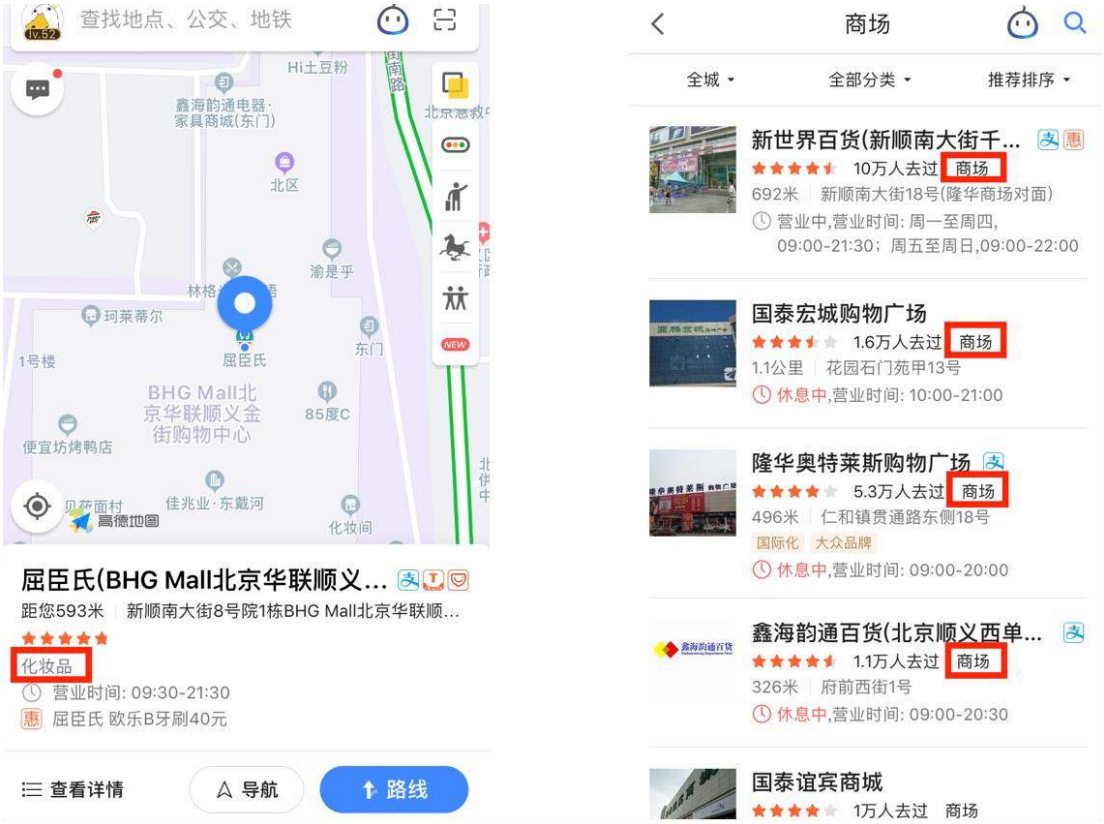
高德信息研发部诚招 NLP、机器学习、搜索推荐算法专家，Java、PHP、C++高级工程师/专家。职位地点：北京，欢迎有兴趣的同学投递简历到 xi.yang@alibaba-inc.com

地图 POI 类别标签体系建设实践

作者：李璋

1.导读

POI 是“Point of interest”的缩写，中文可以翻译为“兴趣点”。在地图上，一个 POI 可以是一栋房子、一个商铺、一个公交站、一个湖泊、一条道路等。在地图搜索场景，POI 是检索对象，等同于网页搜索中的网页。在地图客户端上，用户选中一个 POI，会有一个悬浮的气球指向这个 POI。



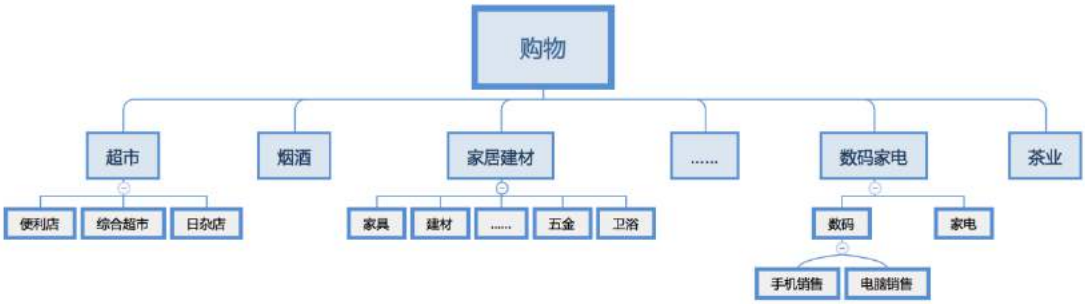
如上图左边，这家商场内的屈臣氏是一个 POI；而所谓类别标签，就是在类别维度对 POI 属性的一种概括，比如，屈臣氏的类别标签化妆品，而屈臣氏所坐落的凯德 mall，类别标签是商场；右侧则是商场 query 搜索召回的一系列 POI，都具有和 query 相匹配的类别属性。

上图也展示了类别标签的两种主要使用场景：为用户提供丰富信息和支持决策，一方面在前端为用户显示更丰富的信息，另一方面支持搜索的类别搜索需求，主要是在地图场景 query 和 POI 双方都具有丰富的多义表达，通过传统的文本匹配引擎或者简单的同义词泛化是难以达到目的的，因此挖掘标签作为召回和排序依据。

我们的类目体系建设主要依据以下几点：

- 用户实际的 query 表达，主要为了支持用户的搜索需求。
- 真实世界的客观类目分布，以及 pm 对该分布的认知。
- 不同标签间的从属、并列关系。

最终每个大类将构建一个多层的多叉树体系，比如购物类别的划分：



2.类别标签建设的难点

我们的目标是打标，就是将 POI 映射到上面类目树体系的各个节点上，很显然这是一个分类问题，但又不是一个单纯的分类问题：

- **多标签问题**：屈臣氏打上化妆品的标签，是一个一对一的映射；而部分 POI，可能同时具有多个标签，比如汤泉良子，可以洗浴、按摩、足疗；xx 家具店，打上家具店标签同时，必须打上其父节点家居建材标签。整体上，这是一个多标签问题，而不是多分类问题。

- **文本相关问题**：大多数的 POI 具有比较直观的文本标题，比如小牛电动车、海尔专卖店、东英茗茶、熙妍精衣、新生贵族，通过名称文本分析，可以预测出比较正确的结果。另一方面，又不是纯文本问题，比如苹果专卖，仅从文本无法确认是一个手机店，还是一个水果店；还有一些表达，比如老五批发，低频表达或者不含类别信息，则需要引入其他特征来进行解决。

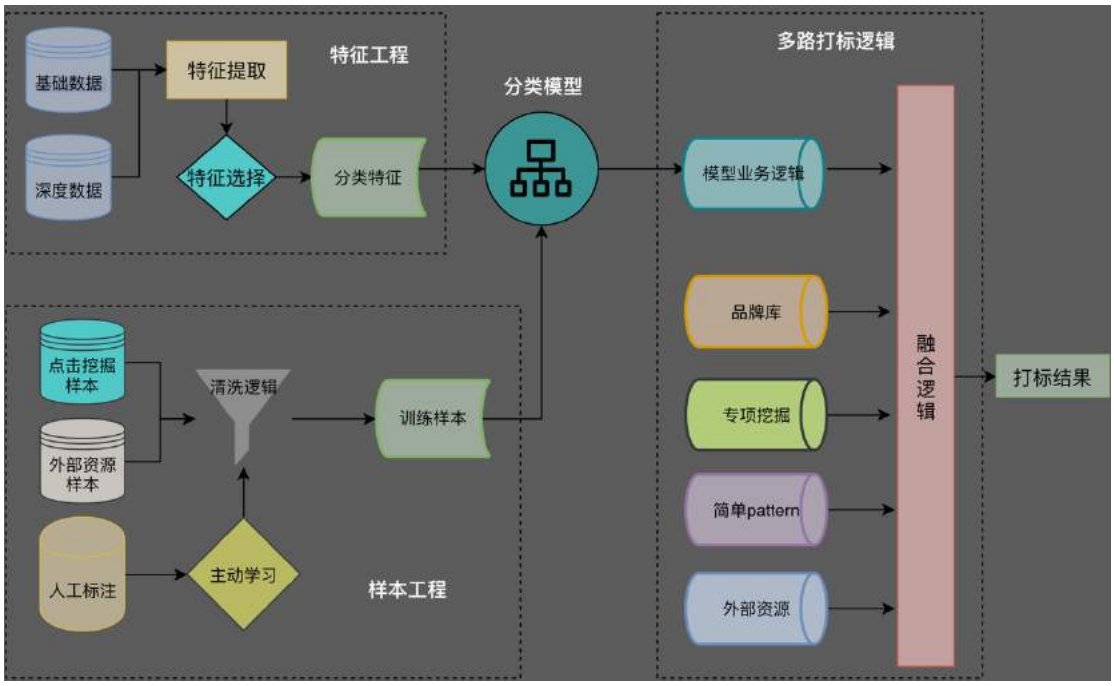
- **综合性问题**：算法可能解决主要问题，但现实世界的复杂，通过单纯的算法是难以完全覆盖的，比如酒吧中夜店和清吧的区分，三甲医院、汽车 4S 店的打标，低频品牌的识别等，通过受限的样本和特征无法尽数解决，但又无法置之不理。

此外，应用方对于标签的准召和产出速率也有较高的要求：打标准确率低，则可能导致用户搜索时召回错误 POI；覆盖率低，则可能导致用户期待的结果被漏掉；而待建设的大分类有 20+，同时每个大分类有数十个子标签，大小标签总量上千。则必须使用高速高效、准召均有保障的方法进行打标，才能有效落地收益。

综上，我们要解决的类别标签打标的主要问题，是一个多标签分类问题，主要使用文本进行识别，但有必要引入其他非文本特征或手段，才能比较完满的解决。

3.技术方案

3.1 整体方案设计



如图，为了高效完成打标，我们设计了主要的流程模块，具体描述如下：

- **特征工程**：文本特征解决最主要的打标问题，但同时地图场景下 POI 文本偏短，长尾分布广泛，具有较多的低频文本或者完全不含类目信息的低频品牌等，而评论、简介等长文本描述往往偏于高频，而难点在于解决低频。因此特征设计上，尽可能使用一些通用特征，比如 POI 名称、typecode(生产方维护的另一套分类体系)、来源类别(数据提供方的原始零件类别)、品牌等通用特征；对于高频专有特征或数据，一般不在通用模型中进行识别。

- **样本工程**：样本的挖掘和清洗、以及模型的设计也是旨在解决通用性的打标问题，POI 表达的多样性，而同时标签数量极多，导致需要的样本量也极大，标注成本较高时，必须考虑人工标注之外的样本来源。

- **分类模型**：单纯的文本分类模型，不能解决非文本的问题，同时多标签的问题，也不能单纯使用多分类模型来解决；我们设计了多种贴合业务的模型改造工作。

- **多路融合**：分类模型能解决主要的问题，但不是全部问题。在地图场景下，总有些算法之外的待解问题，如 5A 景点、三甲医院，以及各类品牌，模型难以尽数处理。我们设计多路打标，如品牌库对品牌效果进行兜底，引入外部资源批量解决非算法问题，引入专项挖掘解决非通用的打标类别等等……总体上对模型之外的问题引入外部知识辅助进行打标，从整体上收敛问题。

后面将重点介绍业务的主要难点，在样本和模型上的主要工作。

3.2 样本工程

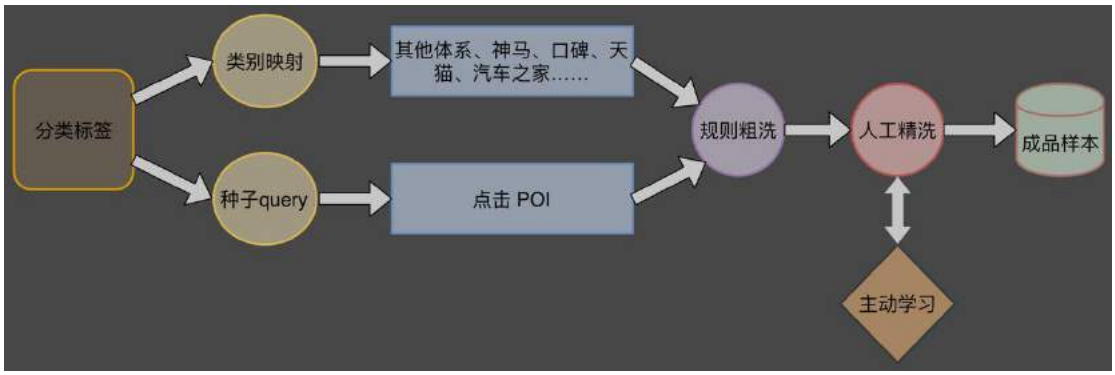
3.2.1 样本来源&清洗

样本方面，经过一些实验论证，标签数量多，每个标签需要的样本量大，人工标注几乎不可能满足要求，因此考虑主要使用点击日志和一些现成的外部资源：

- 点击日志数据量大，能够循环产生，同时反映了用户最直接的意图；缺点是含有的噪声大，同时用户点击往往偏向于高频，低频表达较为稀缺。

- 外部资源数据量小，但多样性较好，能够弥补点击数据在低频表达不足的问题。

通过引入这两方面的样本，我们很快得到了数百万的原始样本，这么大量级的样本，即使清洗依然是一个及其巨大的工作量，为了高效地清洗样本，我们设计了结合主动学习的两级模式：



在两方面的初始样本引入后：

- 首先对数据进行抽样清洗，并将清洗过程抽象为业务规则，进行全局清洗和迭代。

- 当整体系统且明显的噪声趋于收敛后，我们通过对剩余数据进行划分，每次抽取部分数据建模，对另一部分数据进行识别，然后对识别不好的一部分数据进行人工标注；如此反复迭代多轮。

通过一种类似主动学习的方式，使人工标注的价值最大化，避免低信息量重复样本的反复标注造成人力浪费。

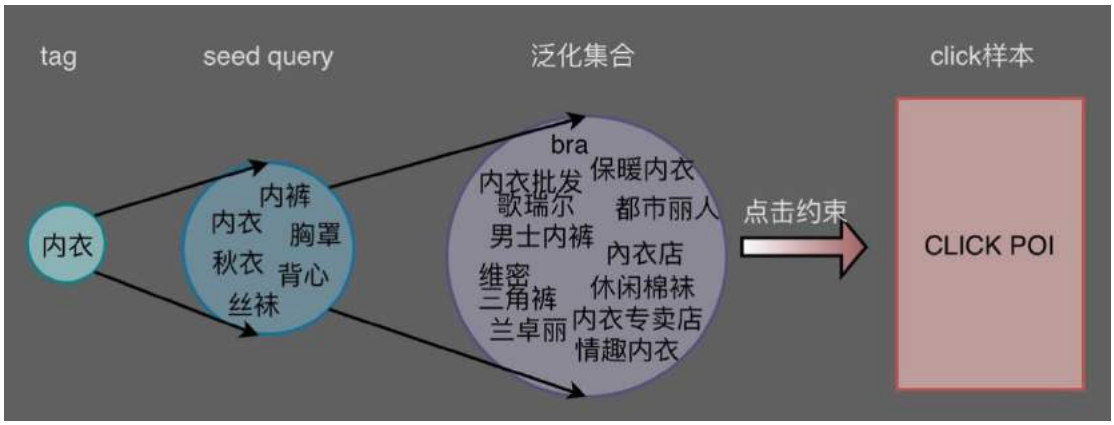
下面具体介绍点击样本的挖掘思路。

3.2.2 点击样本挖掘

搜索点击日志凝聚了无数用户的需求与智慧，大多数的搜索业务都能从中挖掘最原始的训练样本。具体到当前的挖掘业务，首先要解决的问题是样本表达形式的不一致问题。具体描述为：

点击数据：query -> POI
需要样本：tag -> POI
解决方案：tag -> query -> POI

如下图，要挖掘内衣的样本集，人工定义了该标签的映射的 query 集合 seed query，再通过这个 query 集合去召回对应的 click 样本，就可以直接作为标签内衣的样本。



在实际操作中，我们增加了 seed query 到泛化集合的映射，即由人工定义的高频 query 集合泛化到一个更大的同义集合后，再由同义集合进行 click 样本的召回，其出发点在于：

高频的 query 主要点击集中于高频的样本，要解决的问题难点在于低频表达的挖掘，因此对 query 进行从高频到低频的泛化，以期通过低频 query 召回低频的样本表达，比如丝袜到休闲棉袜，内衣到维密、都市丽人等方面的扩展。

query 泛化过程

query 的泛化，需要通过高频集合获得近义的低频表达，同时又要保证不会过度的语义扩散，导致泛化集合偏离了标签原本的语义。我们主要尝试了以下方案：

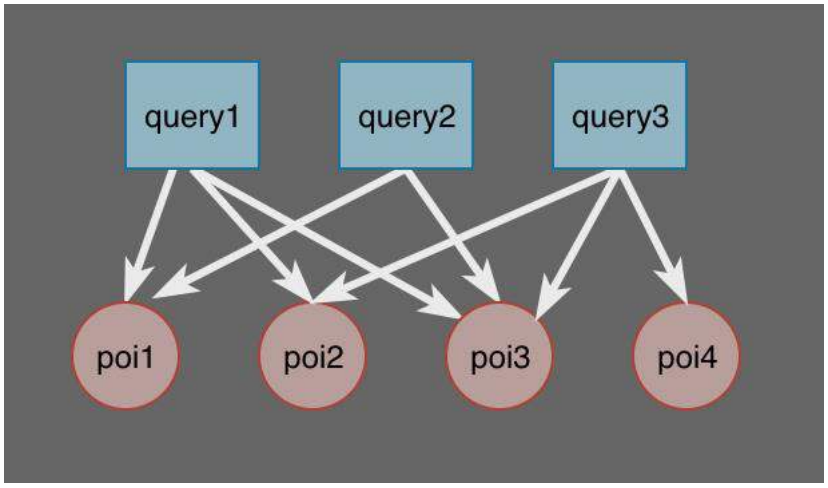
- **word2vec** :通过对点击或外部语料学习 word 粒度的向量表达，对 query 中的多个 word 进行 maxpooling，得到 query 的向量表达，再通过 query 向量去全量集合中搜索向量距离近的其他 query。该方法的主要问题是通过 word 粒度的 embedding 刻画的 query 表达，在泛化过程中不太受控，容易引入大量的噪声，清洗难度大，同时存在显著 case 的漏召。

- **同义词**：该方法召回非常受限，且得到的 query 不一定符合用户的自然表达。

- **Session 上下文**：地图场景的 session 普遍较短，召回受限且语义有偏离，准召均不高。

- **推荐方法**：继续考虑点击日志的挖掘，将各个 query 比作 user，点击的 POI 作为 item，考虑引入推荐的思想进行相似 query 挖掘——即点击相同或相似 POI 的 query 具有某种程度上的相似性，而 query 和 POI 点击关系天然构成一个社交网络，由此参考了两种基于图的推荐方式：

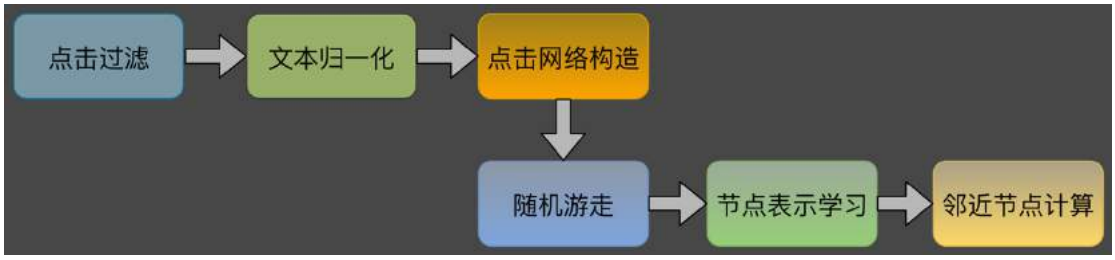
SimRank，通过 query 与 POI 间的点击关系形成二部图，两个节点间的相似度由他们共同关联的其他所有点加权平均得到，通过反复迭代多轮后，相似度趋于稳定，得到两两 query 间相似度；



a，b 表示图中的两个节点，定义自相关度为 1，即 s(a, a)=1，I(a)、I(b)表示与 ab 相连的节点集合，对于不同的 a、b，其相似度描述为：

$$s(a,b)=\frac{C}{|I(a)||I(b)|}\sum_{i=1}^{|I_i(a)|}\sum_{j=1}^{|I_i(b)|}s(I_i(a),I_j(b))$$

DeepWalk，是一种学习网络中节点的表示的新的方法，是把 language modeling 的方法用在了网络结构上，从而使用深度学习方式学习网络中节点间的关系表示。类似 simrank 方式，通过 query 与 POI 间的点击关系构成点击网络，在点击网络上进行随机游走，通过对游走片段的学习，得到 query 的向量表达，再通过向量表达计算 query 的相似召回。



推荐方法的两种方案中，simrank 计算量极大，在小数据量上实验经过了较长时间的迭代，而数千万点击数据计算资源和计算时间都将变得极大，成本上不合算；而 DeepWalk 随机游走方式在实际测试中取得了较好的效果：

query	泛化结果
涂料	硅藻泥、乳胶漆、多乐士、巴德士漆
ETC	苏通卡、鲁通卡、中原通

比如，原始 query 为涂料，泛化得到其子分类、品牌等，ETC 可以得到一些地方性的命名表达。

不同于将 query 分词后学习其 word 粒度 embedding 表达，DeepWalk 在这里直接学习整个 query，即 Sentence 的表达，避免了将 query 分词为多个 word 再 pooling 过程中语义偏移；

而且直接 query 粒度的表示学习，得到的挖掘结果更加符合线上的实际表达，便于我们后续召回点击 POI 样本的操作；同时粗粒度的学习对网络节点间社会关系会有更优的保留。

整体上，通过 query 泛化步骤，样本集合的低频表达比例明显提升。

3.3 模型设计与迭代

使用分类的方式解决打标问题，是业类相似场景的通用做法。具体到业务角度，我们需要模型解决的打标数据范畴主要分为四类：

- 高频后缀识别
- 低频后缀识别
- 品牌识别
- 非文本识别(typecode、来源类别等)

前三类可以使用纯文本分类模型，但非文本特征无法直接用文本模型进行整合；同时，样本数据在前期具有噪声大、分布不均衡的特点，都需要兼顾。

1.3.1 分层多分类

前期使用多分类模型解决分类问题，对于每一个非叶节点构建一个多分类器，从根部开始构建，分到最后一级输出为叶节点。其特点是直观，和标签体系比较匹配。

这种方式有比较明显的缺陷：方案结构复杂不稳定，维护难，样本组织相当繁杂；上级分类模型的错漏，将层层影响下级的分类效果。

3.3.2 每个标签二分类

业务前期受限于优化前的搜索效果，样本集有一定量的噪声，同时体系在调研期间也会时常迭代，不断优化，需要一个解释性强、兼容变化，同时又能直接支持多标签的模型。

因此，我们尝试了对每个标签训练二分类模型，待识别的 POI 经过所有的二分类模型预测后结果合并使用，再进行后续的业务逻辑建设，该方式较好地解决了业务多标签的场景，避免同一 POI 多个标签间的冲突。

在样本方面，使用 ovr 方式组织样本，样本比例可调整（正负例比例、负例来源比例等），兼容了业务样本不均衡问题。

3.3.3 统一模型

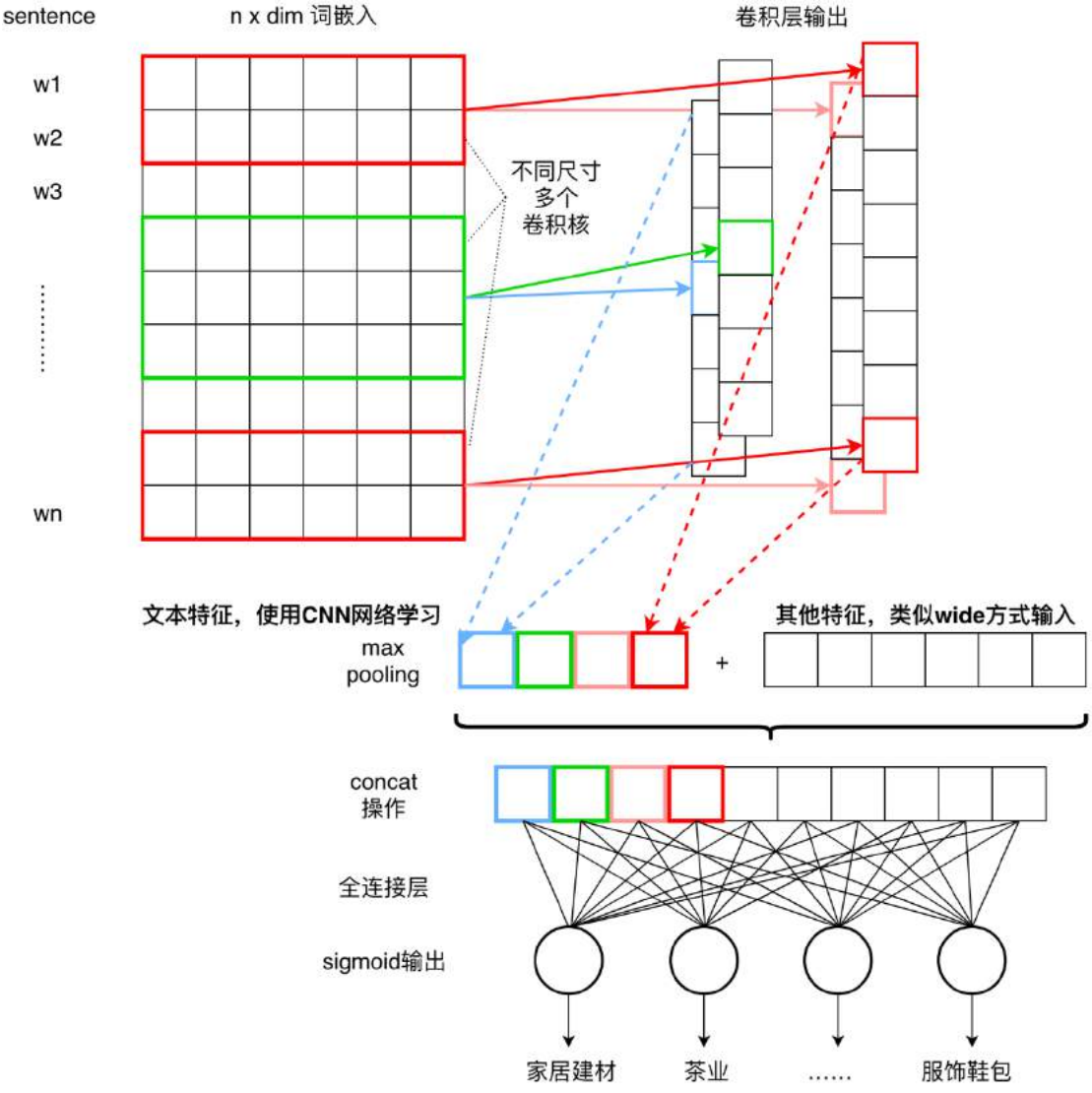
单标签的二分类模型具有良好的解释性，以及每个标签建模带来的灵活性，很好匹配了业务的前期需求，迅速推进标签效果落地，线上恶劣 case 大量收敛，同时较原来的人工专家规则方式，召回明显提升的同时，效率也提升十倍以上。

前期为了短平快地推进业务，使用简单二分类模型迅速解决了主要问题，随着搜索效果的不断提升，业务上也有了更高的质量要求，而 LR 二分类模型依然存在一些问题：

- 使用词袋特征，维度膨胀
- 特征选择造成低频特征损失
- 泛化性能一般，需要大量建设业务规则
- 独立建模，标签间的关系识别不充分(父子、共现、互斥)
- 负样本大量降采样，浪费样本
- 模型数量极多，维护成本高

为了达到更好的效果，深度模型的升级有了必要，深度文本分类，常用的模型有 cnn、rnn 以及基于 attention 的各种模型，而 attention 模型主要解决的是长文本的深层语义理解问题，我们业务场景需要解决的是短文本简单语义的泛化问题，同时考虑到需要和非文本特征结合，以及分类效率方面的考虑，我们选择 textCNN 模型。

而 textCNN 是一个纯文本的多分类模型，不能解决多标签情况，也无法兼容非文本特征，同时样本不均衡问题也导致小样本类别学习不充分。为了匹配实际的业务，我们对原始的 textCNN 进行了业务改造，如图：



原始 textCNN 对文本进行词嵌入、卷积核池化后，直接进行 softmax 多分类，而在当前业务下：

- 将文本表示经过池化后，拼接了外部的其他非文本特征，对文本使用深度的特征提取与表示，而简单的非文本特征使用类似 wide&deep 的方式接入，共同参与预测。
- softmax 只能输出归一化的预测结果，不适用与多标签场景；通过改造输出层，使用多路的 sigmoid 输出，每一路输出对应一个标签的预测结果。
- 对同一样本多标签的情况进行了压缩，多个 label 合并为一个向量。
- 重新设计损失函数，多分类交叉熵不适用于多个二分类输出未归一化的场景。

如此，解决了多标签场景&非文本特征接入的问题，同时使用深度模型取得更好的泛化效果。

另一方面，所有样本在同一模型中训练，原本样本不均衡问题造成了一些负面影响。比如大类家居建材、服饰鞋包有数十万的样本，而劳保用品、美容美发用品等标签只有几千样本，在训练时很容易直接丢弃对整体准确率影响较小的小样本集。因此对每个样本集计算权重：

$$w_k = max\left(exp\left(-{}^{c_k}/_{h}\right),\ 1\right)$$

ck 为 k 类别的样本数，h 为平滑超参，效果上类别样本数量越少，权重越大。将类别权重加入到损失函数：

$$L = -\sum_{i=1}^{N}\sum_{k=1}^{C} w_k \cdot [y \cdot \log \hat{y} + (1-y) \cdot \log(1-\hat{y})] + \lambda_1 \cdot \|w_{text}\|_2^2 + \lambda_1 \cdot \|w_{wide}\|_2^2$$

λ1、λ2 分别代表对文本特征和非文本特征的正则化约束比重因子。

经过改造和优化后：

以上几个标签间样本数量差距极大，但分类效果都维持在一个比较高的准召率上。此外，我们还根据改造后的模型，开发了模型解释工具。对于一些不符合预期的 case，通过解释工具，可以直接观察：

- 分类时起主要作用的因子，是文本特征，还是非文本特征，以及具体是哪个特征及其 value 显示。

- 如果是文本特征，显示权重最大的若干个卷积窗口位置，以及具体窗口中每个 term 的权重。

通过该工具可以在使用深度模型预测时，保持如同 LR 一般的解释性，方便定位与迭代。

总体上经过以上 textCNN 的业务改造与迭代，随机数据准确率提升 5%以上，同时业务规则减少 66%，同时在长尾 case(低频文本、品牌)上取得比较明显的效果。

1.4 相关工作

前面主要介绍了我们在类别标签建设方面的一些工作。

在另一方面，类别标签在地图搜索中生效，则需要识别哪些 query 需要使用标签进行召回。在前期，我们人工标注了高德地图搜索头部 query 和标签的对应关系。但人工的标注方式始终覆盖有限：

- 不能取得中低频 query 的标签召回收益，从而标签利用率不够高。

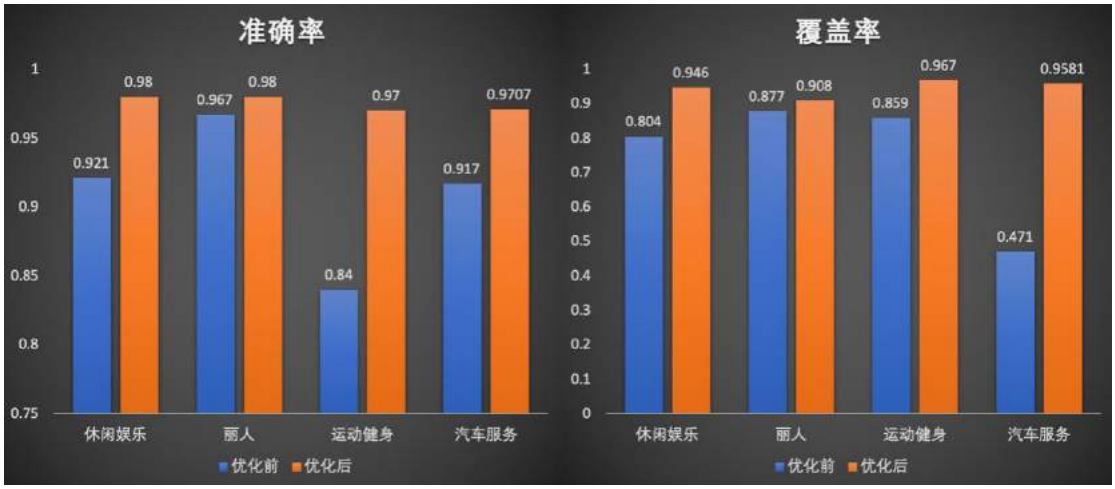
- 对于应该使用标签召回的 query，如果因为未能识别而使用文本召回，也影响了线上的效果，降低用户体验，造成 badcase。

因此，在完成体系建设后，我们另外建模，主要使用文本语义和点击方面的特征，识别 query 和标签的对应召回关系，即识别哪些 query 可以使用类别标签召回，通过这一系列的工作，对中低频的标签召回进行了深挖，高德地图中类别搜索流量中使用标签召回比例高达 94%。

4.收益

标签数据产出后，需要进行两方面的评估：

- 从输出数据的质量上，即本身标签的准确率和覆盖率：



高质量的数据，不光利于搜索的业务支撑，还利于我们的类别标签体系在整个 BU 的推广和落地。

- 上线后对线上泛搜实际 query 的搜索效果提升评估：

除了数据质量的评估，还会结合新的标签数据对搜索效果进行评估，即对新标签体系和旧标签体系的对照效果评估。在人工效果评估中，gsb 效果上，每个大类的数据上线，都带来了非常明显的类别搜索质量提升，从而让搜索更准确、更全面的辅助用户决策。

5.小结

当前工作的重点在于使用通用特征解决了主体的类别打标问题，对于通用特征不可解的问题，往往通过外部知识、资源的建设方式解决，如品牌库建设、A 级景区资源收集等方式。

而实际上，使用通用特征外，不通用特征对于部分数据的分类效果提升应用并不充分，后续应该安排一系列的专项优化，比如：

- 评论、简介等特征，应用一些 Attention 方法，可能取得比较好的补充效果。
- POI 图片中往往隐含一些类别相关信息，对图片识别可以充分利用这些信息。
- 外部百科、知识图谱等知识的引入，辅助中低频品牌库的建设等。

在业务闭环建设上还要持续，比如恶劣 case 的流转与自动修复机制建设，新品牌、新标签的发现等问题，以避免打标系统长期运行后的效果退化等。

无论机器学习还是外部资源辅助的方式，对于海量的长尾数据往往乏力，实际线上很多的 POI 特征相当匮乏，只有一个简单的名称，从中很难准确预测其类别。如何引导用户自己提交类别信息，或者众包方式完成类别标签的标注，也是我们后续需要着重考虑的解决方案。

招聘

高德信息研发部诚招 NLP、机器学习、搜索推荐算法专家，Java、PHP、C++高级工程师/专家。职位地点：北京，欢迎有兴趣的同学投递简历到 xi.yang@alibaba-inc.com

地理文本处理技术在高德的演进(上)

作者：暮兮

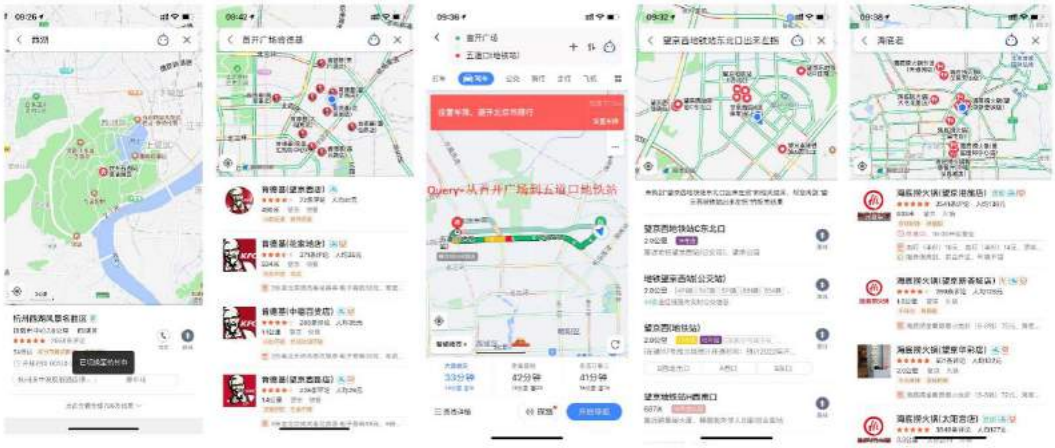
1.背景

地图 App 的功能可以简单概括为定位，搜索，导航三部分，分别解决在哪里，去哪里，和怎么去的问题。高德地图的搜索场景下，输入的是，地理相关的检索 query，用户位置，App 图面等信息，输出的是，用户想要的 POI。如何能够更加精准地找到用户想要的 POI，提高满意度，是评价搜索效果的最关键指标。

一个搜索引擎通常可以拆分成 query 分析、召回、排序三个部分，query 分析主要是尝试理解 query 表达的含义，为召回和排序给予指导。

地图搜索的 query 分析不仅包括通用搜索下的分词，成分分析，同义词，纠错等通用 NLP 技术，还包括城市分析，wherewhat 分析，路径规划分析等特定的意图理解方式。

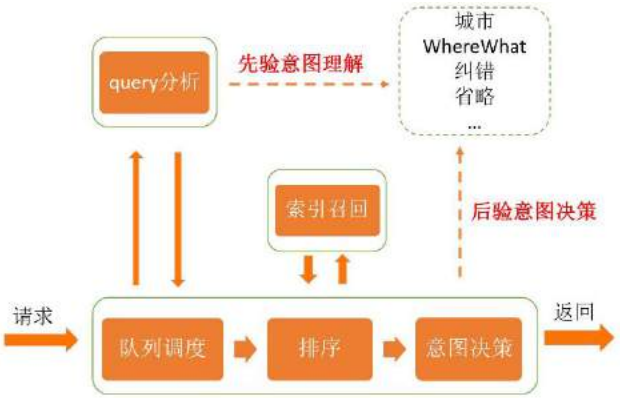
常见的一些地图场景下的 query 意图表达如下：



异地城市意图　wherewhat意图　路径规划意图　省略意图　纠错意图

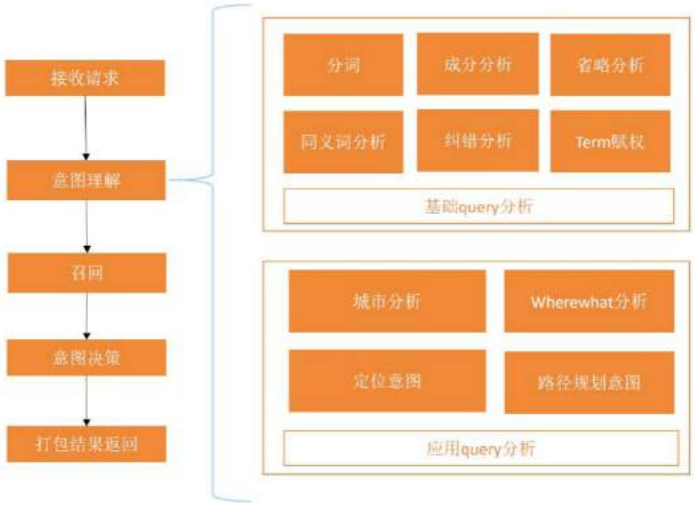
query 分析是搜索引擎中策略密集的场景，通常会应用 NLP 领域的各种技术。地图场景下的 query 分析，只需要处理地理相关的文本，多样性不如网页搜索，看起来会简单一些。但是，地理文本通常比较短，并且用户大部分的需求是唯一少量结果，要求精准度非常高，如何能够做好地图场景下的文本分析，并提升搜索结果的质量，是充满挑战的。

2.整体技术架构



搜索架构

类似于通用检索的架构，地图的检索架构包括 query 分析，召回，排序三个主要部分。先验的，用户的输入信息可以理解为多种意图的表达，同时下发请求尝试获取检索结果。后验的，拿到每种意图的检索结果时，进行综合判断，选择效果最好的那个。



query 分析流程

具体的意图理解可分为基础 query 分析和应用 query 分析两部分，基础 query 分析主要是使用一些通用的 NLP 技术对 query 进行理解，包括分析，成分分析，省略，同义词，纠错等。应用 query 分析主要是针对地图场景里的特定问题，包括分析用户目标城市，是否是 where+what 表达，是否是从 A 到 B 的路径规划需求表达等。

整体技术演进

在地里文本处理上整体的技术演进经历了规则为主，到逐步引入机器学习，到机器学习全面应用的过程。由于搜索模块是一个高并发的线上服务，对于深度模型的引入有比较苛刻的条

件，但随着性能问题逐渐被解决，我们从各个子方向逐步引入深度学习的技术，进行新一轮的效果提升。

NLP 领域技术在最近几年取得了日新月异的发展，bert，XLNet 等模型相继霸榜，我们逐步统一化各个 query 分析子任务，使用统一的向量表示对进行用户需求进行表达，同时进行 seq2seq 的多任务学习，在效果进一步提升的基础上，也能够保证系统不会过于臃肿。

本文就高德地图搜索的地理文本处理，介绍相关的技术在过去几年的演进。我们将选取一些点分上下两篇进行介绍，上篇主要介绍搜索引擎中一些通用的 query 分析技术，包括纠错，改写和省略。下篇着重介绍地图场景中特有 query 分析技术，包括城市分析，wherewhat 分析，路径规划。

3.通用 query 分析技术演进

3.1 纠错

在搜索引擎中，用户输入的检索词（query）经常会出现拼写错误。如果直接对错误的 query 进行检索，往往不会得到用户想要的结果。因此不管是通用搜索引擎还是垂直搜索引擎，都会对用户的 query 进行纠错，最大概率获得用户想搜的 query。

在目前的地图搜索中，约有 6%-10%的用户请求会输入错误，所以 query 纠错在地图搜索中是一个很重要的模块，能够极大的提升用户搜索体验。

在搜索引擎中，低频和中长尾问题往往比较难解决，也是纠错模块面临的主要问题。另外，地图搜索和通用搜索，存在一个明显的差异，地图搜索 query 结构化比较突出，query 中的片段往往包含一定的位置信息，如何利用好 query 中的结构化信息，更好地识别用户意图，是地图纠错独有的挑战。

- 常见错误分类

(1) 拼音相同或者相近，例如: 盘桥物流园-潘桥物流园

(2) 字形相近，例如: 河北冒黎-河北昌黎

(3) 多字或者漏字，例如: 泉州州顶街-泉州顶街

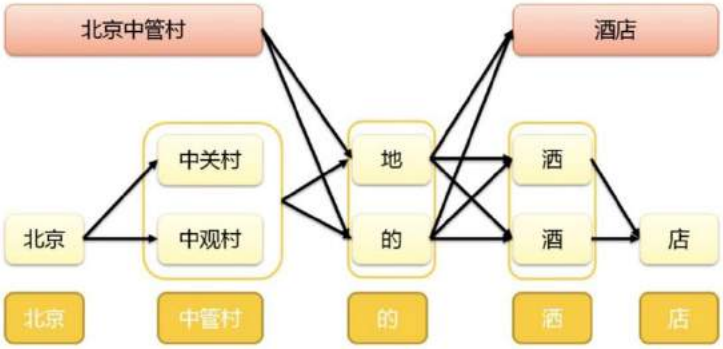
- 纠错现状

原始纠错模块包括多种召回方式，如：

拼音纠错：主要解决短 query 的拼音纠错问题，拼音完全相同或者模糊音作为纠错候选。

拼写纠错：也叫形近字纠错，通过遍历替换形近字，用 query 热度过滤，加入候选。

组合纠错：通过翻译模型进行纠错替换，资源主要是通过 query 对齐挖掘的各种替换资源。



组合纠错翻译模型计算公式：

$$C(f) \propto P(f)P(f|e)$$

其中 p(f)是语言模型，p(f|e)是替换模型。

问题 1 :召回方式存在缺陷。目前 query 纠错模块主要召回策略包括拼音召回、形近字召回，以及替换资源召回。对于低频 case，解决能力有限。

问题 2：排序方式不合理。纠错按照召回方式分为几个独立的模块，分别完成相应的召回和排序，不合理。

技术改造

- 改造 1：基于空间关系的实体纠错

原始的纠错主要是基于用户 session 挖掘片段替换资源，所以对于低频问题解决能力有限。但是长尾问题往往集中在低频，所以低频问题是当前的痛点。

地图搜索与通用搜索引擎有个很大的区别在于，地图搜索 query 比较结构化，例如北京市朝阳区阜荣街 10 号首开广场。我们可以对 query 进行结构化切分（也就是地图中成分分析的工作），得到这样一种带有类别的结构化描述，北京市【城市】朝阳区【区县】阜荣街【道路】10 号【门址后缀】首开广场【通用实体】。

同时，我们拥有权威的地理知识数据，利用权威化的地理实体库进行前缀树+后缀树的索引建库，提取疑似纠错的部分在索引库中进行拉链召回，同时利用实体库中的逻辑隶属关系对纠错结果进行过滤。实践表明，这种方式对低频的区划或者实体的错误有着明显的作用。

基于字根的字形相似度计算

上文提到的排序策略里面通过字形的编辑距离作为排序的重要特征，这里我们开发了一个基于字根的字形相似度计算策略，对于编辑距离的计算更为细化和准确。汉字信息有汉字的字根拆分词表和汉字的笔画数。

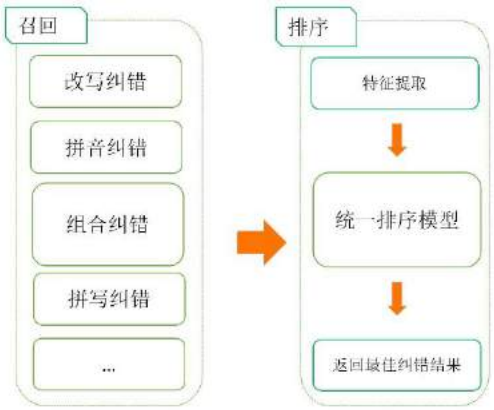
洲 氵--州 9

州 州 6

将一个汉字拆分成多个字根，寻找两个字的公共字根，根据公共字根笔画数来计算连个字的相似度。

- 改造 2：排序策略重构

原始的策略召回和排序策略耦合，导致不同的召回链路，存在顾此失彼的情况。为了能够充分发挥各种召回方式的优势，急需要对召回和排序进行解耦并进行全局排序优化。为此我们增加了排序模块，将流程分为召回和排序两阶段。



模型选择

对于这个排序问题，这里我们参考业界的实践，使用了基于 pair-wise 的 gbrank 进行模型训练。

样本建设

通过线上输出结合人工 review 的方式构造样本。

特征建设

(1) 语义特征。如统计语言模型。

(2) 热度特征。pv，点击等。

(3) 基础特征。编辑距离，切词和成分特征，累积分布特征等。

这里解决了纠错模块两个痛点问题，一个是在地图场景下的大部分低频纠错问题。另一个是重构了模块流程，将召回和排序解耦，充分发挥各个召回链路的作用，召回方式更新后只需要重训排序模型即可，使得模块更加合理，为后面的深度模型升级打下良好的基础。后面在这个框架下，我们通过深度模型进行 seq2seq 的纠错召回，取得了进一步的收益。

3.2 改写

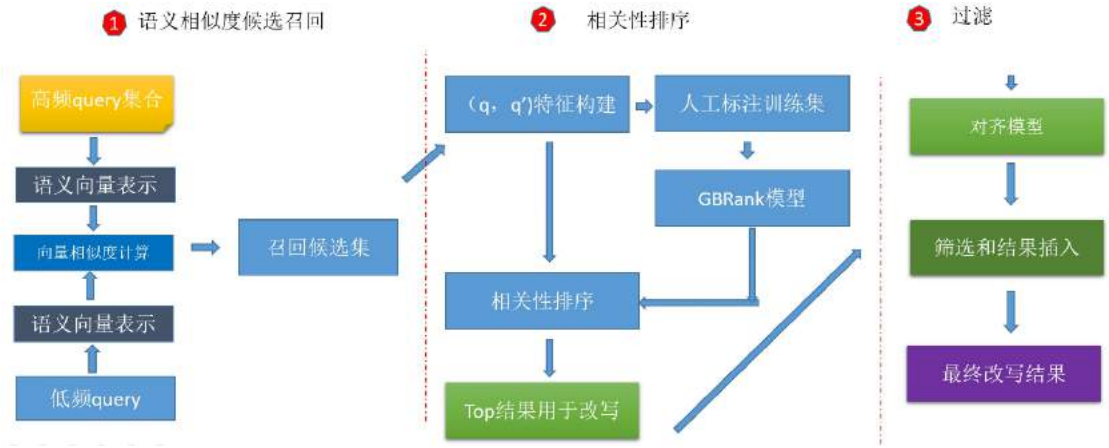
纠错作为 query 变换的一种方式的召回策略存在诸多限制，对于一些非典型的 query 变换表达，存在策略的空白。比如 query=永城市新农合办，目标 POI 是永城市新农合服务大厅。用户的低频 query，往往得不到较好搜索效果，但其实用户描述的语义与主 poi 的高频 query

是相似的。

这里我们提出一种 query 改写的思路，可以将低频 query 改写成语义相似的高频 query，以更好地满足用户需求多样性的表达。

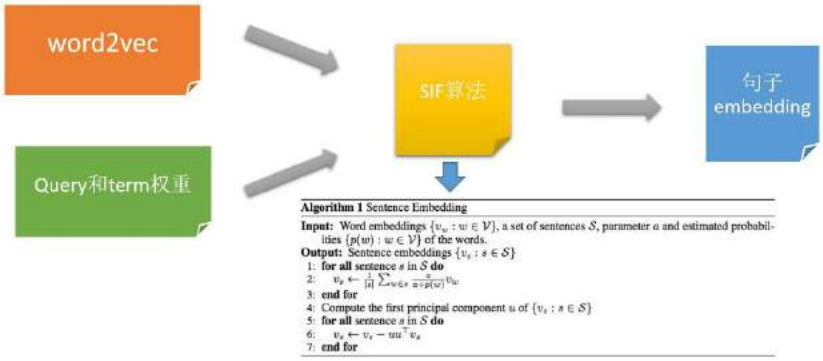
这是一个从无到有的实现。用户表达的 query 是多样的，使用规则表达显然是难以穷尽的，直观的思路是通过向量的方式召回，但是向量召回的方式很可能出现泛化过多，不适应地图场景的检索的问题，这些都是要在实践过程中需要考虑的问题。

方案



整体看，方案包括召回，排序，过滤，三个阶段。

- 召回阶段



我们调研了句子向量表示的几种方法，选择了算法简单，效果和性能可以和 CNN，RNN 媲美的 SIF（Smooth Inverse Frequency）。向量召回可以使用开源的 Faiss 向量搜索引擎，这里我们使用了阿里内部的性能更好的的向量检索引擎。

- 排序阶段

样本构建

原 query 与高频 query 候选集合，计算语义相似度，选取语义相似度的 TOPK，人工标注的训练样本。

特征建设

（1）基础文本特征
（2）编辑距离
（3）组合特征

模型选择

使用 xgboost 进行分数回归

- 过滤阶段

通过向量召回的 query 过度泛化非常严重，为了能够在地图场景下进行应用，增加了对齐模型。使用了两种统计对齐模型 giza 和 fastalign，实验证明二者效果几乎一致，但 fastalign 在性能上好于 giza，所以选择 fastalign。

通过对齐概率和非对齐概率，对召回的结果进行进一步过滤，得到精度比较高的结果。

query 改写填补了原始 query 分析模块中一些低频表达无法满足的空白，区别于同义词或者纠错的显式 query 变换表达，句子的向量表示是相似 query 的一种隐式的表达，有其相应的优势。

向量表示和召回也是深度学习模型逐步开始应用的尝试。同义词，改写，纠错，作为地图中 query 变换主要的三种方式，以往在地图模块里比较分散，各司其职，也会有互相重叠的部分。在后续的迭代升级中，我们引入了统一的 query 变换模型进行改造，在取得收益的同时，也摆脱掉了过去很多规则以及模型耦合造成的历史包袱。

3.2 省略

在地图搜索场景里，有很多 query 包含无效词，如果用全部 query 尝试去召回很可能不能召回有效结果。如厦门市搜"湖里区县后高新技术园新捷创运营中心 11 楼 1101 室 县后 brt 站"。这就需要一种检索意图，在不明显转义下，使用核心 term 进行召回目标 poi 候选集合，当搜索结果无果或者召回较差时起到补充召回的作用。

在省略判断的过程中存在先验后验平衡的问题。省略意图是一个先验的判断，但是期望的结果是能够进行 POI 有效召回，和 POI 的召回字段的现状密切相关。如何能够在策略设计的过程中保持先验的一致性，同时能够在后验 POI 中拿到相对好的效果，是做好省略模块比较困难的地方。

原始的省略模块主要是基于规则进行的，规则依赖的主要特征是上游的成分分析特征。由于基于规则拟合，模型效果存在比较大的优化空间。另外，由于强依赖成分分析，模型的鲁棒性并不好。

技术改造

省略模块的改造主要完成了规则到 crf 模型的升级，其中也离线应用了深度学习模型辅助样本生成。

- 模型选择

识别出来 query 哪些部分是核心哪些部分是可以省略的，是一个序列标注问题。在浅层模型的选型中，显而易见地，我们使用了 crf 模型。

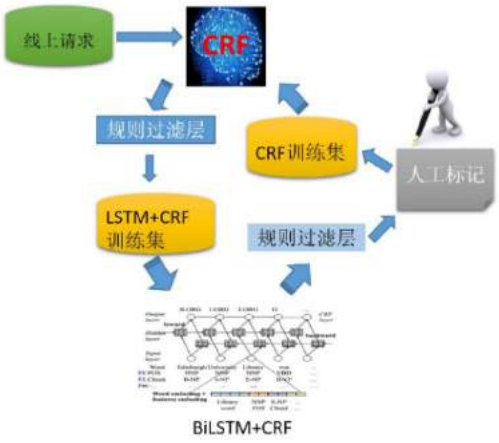
特征建设

term 特征：使用了赋权特征，词性，先验词典特征等。
成分特征：仍然使用成分分析的特征。
统计特征：统计片段的左右边界熵，城市分布熵等，通过分箱进行离散化。

样本建设

项目一期使用了使用线上策略粗标，外包细标的方式，构造了万级的样本供 crf 模型训练。

但是省略 query 的多样性很高，使用万级的样本是不够的，在线上模型无法快速应用深度模型的情况下，我们使用了 boostraping 的方式，借助深度模型的泛化能力，离线构造了大量样本。



使用了这种方式，样本从万级很容易扩充到百万级，我们仍然使用 crf 模型进行训练和线上应用。

在省略模块，我们完成了规则到机器学习的升级，引入了成分以外的其他特征，提升了模型的鲁棒性。同时并且利用离线深度学习的方式进行样本构造的循环，提升了样本的多样性，使得模型能够更加接近 crf 的天花板。

在后续深度模型的建模中，我们逐步摆脱了对成分分析特征的依赖，对 query 到命中 POI 核心直接进行建模，构建大量样本，取得了进一步的收益。

招聘

高德信息研发部诚招 NLP、机器学习、搜索推荐算法专家，Java、PHP、C++高级工程师/专家。职位地点：北京，欢迎有兴趣的同学投递简历到 xi.yang@alibaba-inc.com

地理文本处理技术在高德的演进(下)

作者：暮兮

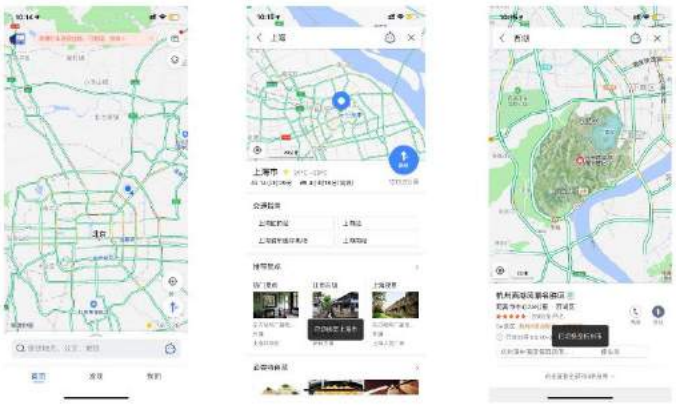
在【上篇】里，我们介绍了地理文本处理技术在高德的整体演进，选取了几个通用 query 分析的点进行了介绍。下篇中，我们会选取几个地图搜索文本处理中特有的文本分析技术做出分析，包括城市分析，wherewhat 分析，路径规划，并对未来做一下展望。

4.query 分析技术演进

4.1 城市分析

在高德地图的检索场景下，从基础的地图数据索引、到在线召回、最终产品展示，均以市级别行政单位为基础粒度。一次完整的检索需求除了用户输入的 query 外，还会包含用户的图面城市以及用户位置城市两个城市信息。

通常，大多数的搜索意图都是在图面或者用户位置城市下，但是仍存在部分检索意图需要在其他城市中进行，准确的识别出用户请求的目标城市，是满足用户需求的第一步，也是极其重要的一步。



用户位置城市　　图面城市　　异地城市

在 query 分析策略流程中，部分策略会在城市分析的多个结果下并发执行，所以在架构上，城市分析的结果需要做到**少而精**。同时用户位置城市，图面城市，异地城市三个城市的信息存在明显差异性，不论是先验输出置信度，还是用后验特征做选择，都存在**特征不可比**的问题。

在**后验意图决策**中，多个城市都有相关结果时，单一特征存在说服力不足的问题，如何结合先验置信度和后验的 POI 特征等多维度进行刻画，都是我们要考虑的问题。

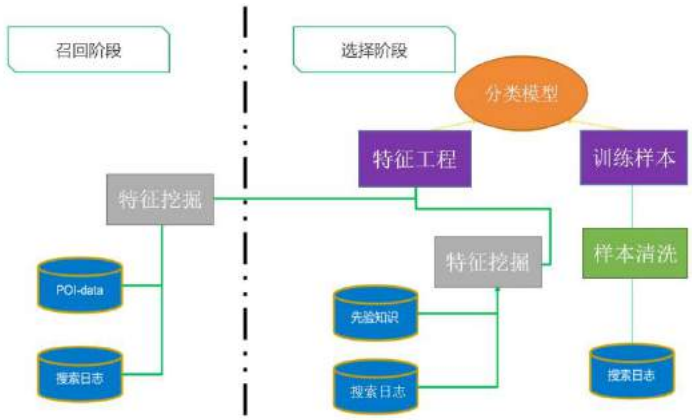
原始的城市分析模块已经采用先验城市分析和后验城市选择的总体流程

但是原始的策略比较简陋，存在以下问题：

- 问题 1：先验和后验两部分均基于规则，效果不好并且可维护性差；
- 问题 2：特征体系存在缺陷。原始的城市分析仅使用 query 级的特征，包括点击，session 改写，query 和城市共现等，对于低频 query 处理得不好。

技术改造

改造 1：城市分析



方案

城市分析是一个**轻召回重选择**的问题，我们将城市分析设计为召回+选择的两阶段任务。

召回阶段，我们主要从 query 和 phrase 两种粒度挖掘特征资源，而后进行候选城市归并。

排序阶段，需要对候选城市进行判断，识别是否应为目标城市，用 gbdt 进行二分类拟合。

样本构建

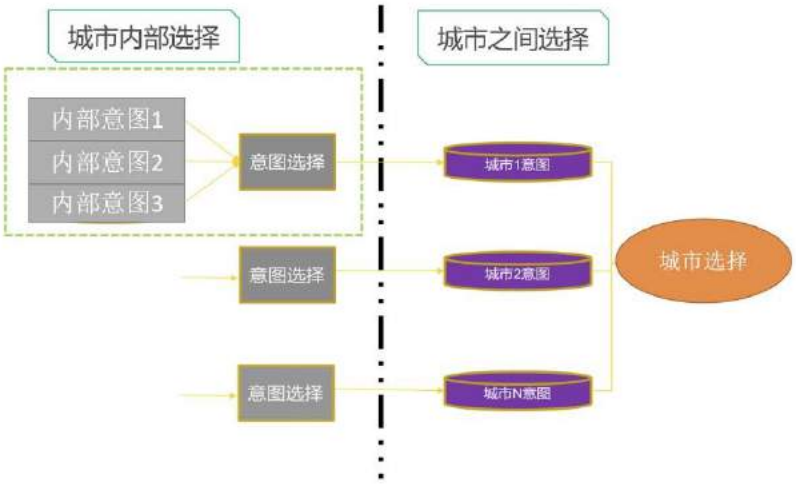
样本方面，我们选择从搜索日志中随机抽取，简单清洗后，进行人工标注。构造样本时存在本异地分样本分布不均的问题，本地需求远远多于异地，这里需要剔除本地和异地相关的特征，避免模型学偏。

特征体系

主要特征包括：

- query 级特征：如用户在特定 query&city 下的点击。
- phrase 级特征：类比于 query 级的特征，在更细粒度下进行统计。
- 组合特征：为了克服单一特征表征能力不足的问题，这里我们进行了一些人工特征组合。

改造 2：城市选择



方案

城市选择在整体的意图决策中处于下游，多种意图先在城市内部 PK，然后城市互相 PK。城市选择问题可以理解为多个城市间的排序问题，这里我们使用 ltr 进行城市选择的建模。

样本构建

使用随机 query 的多个城市意图结果作为样本，每次检索只有 1 个展示城市，因而每次只需要从候选城市中选择目标作为正样本，其他候选城市均作为负样本，与目标城市构成 pair 对。

特征构建

主要特征包括：

- 先验特征。如城市分析部分输出的置信度。
- 文本特征。一些基础的文本相关性特征。
- 点击特征。如不同意图城市中的点击强度。
- 意图特征。一些特征可能影响用户的城市倾向，如用户位置与首位 POI 距离。

相比原始的城市分析和城市选择，两个模块全部实现机器学习化，在恶劣 badcase 显著降低的同时，可维护性大幅提高。在后续的建模中，我们将城市分析作为一个上层应用任务，通过多任务的方式接入到 query 分析的统一模型中来，降低了特征间的耦合，同时实现进一步的效果提升。

4.2 wherewhat 分析

地图场景下的 query 经常包含多个空间语义片段的描述，只有正确识别 query 中的核心部分做 what 用于召回，同时用空间描述部分做 where 进行限定，才能够得到用户想要的 POI。如 query=北京市海淀区五道口肯德基，what=肯德基，是泛需求。query=南京市雨花台区板桥街道新亭大街与新湖大道交界口湾景，what=湾景，是精确需求。这种在 A 附近找 B 或者在 A 范围内找 B 的需求我们把它称作 wherewhat 需求，简称 ww。

query=北京市海淀区五道口肯德基

query=南京市雨花台区板桥街道新亭大街与新湖大道交界口湾景

wherewhat 意图分析主要包括先验和后验两个部分。先验要做 wherewhat 切分，是一个序列标注问题，标注出 query 中的哪些部分是 where 哪些部分是 what，同时给出 where 的空间位置。后验要做意图选择，选择是否展示 wherewhat 意图的结果，可以转化成分类或者排序问题。

wherewhat 体系的主要难点在 ww 切分上，不仅要应对其他 query 分析模块都要面对的低频和中长尾问题，同时还要应对 ww 意图理解独特的问题。像语义模糊，比如北京欢乐谷公交站，涌边村牌坊这样的 query 该切还是不该切，结果差异很大。像语序变换，如 query=嘉年华西草田，善兴寺小寨，逆序的表达如果不能正确识别，效果可能很差。

现状与问题

现状

切分：wherewhat 模块在成分分析模块下游，主要依靠了成分分析的特征。对于一些比较规整的 query，通过成分分析组合的 pattern 来解决，对于一些中长尾的 query，通过再接入一个 crf 模型进行 ww 标注。

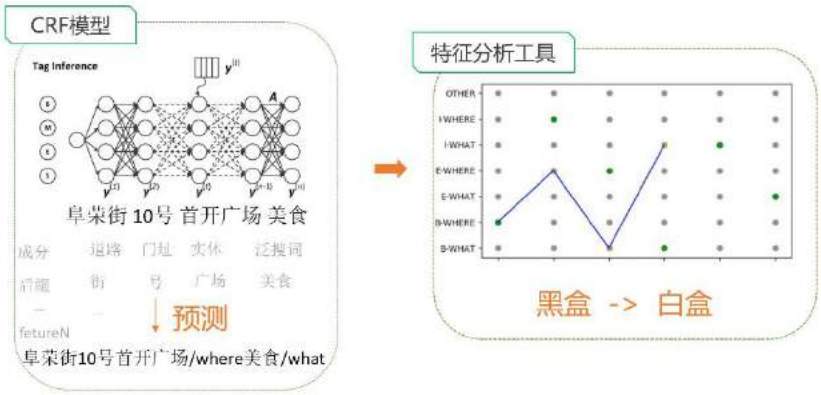
选择：基于人工规则进行意图的判断和选择。

- 问题 1：切分模型简陋。基于 crf 的切分模型使用特征单一，对成分特征依赖严重，处于黑盒状态无法分析。
- 问题 2：后验意图判断，规则堆砌，不好维护。
- 问题 3：逆序问题表现不好。由于 query 的语料大部分是正序的，模型在小比例的逆序 query 上表现不好。

技术改造

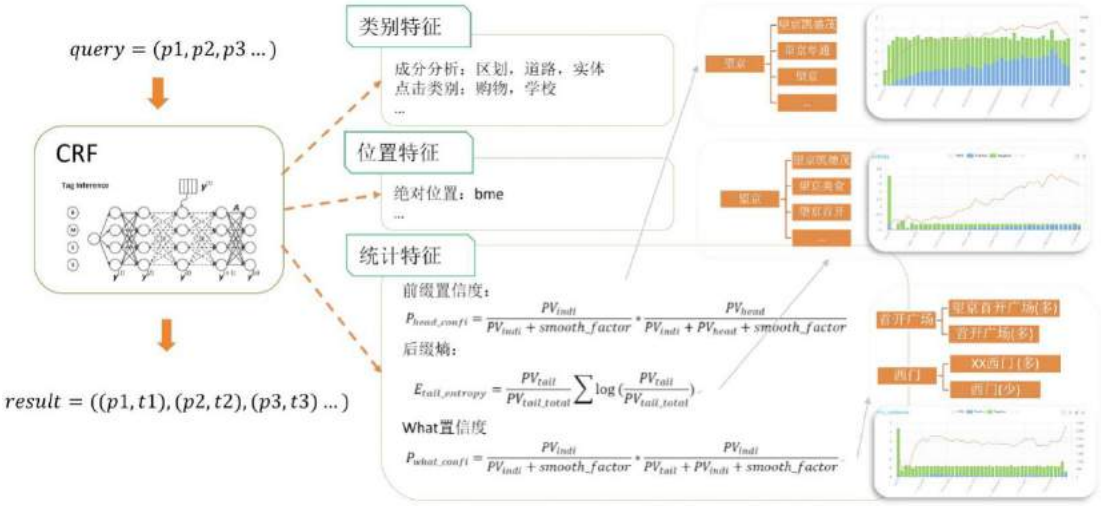
crf 问题分析工具

为了能够分析原始 crf 切分模型的问题，我们基于 crf++源码开发了一个 crf 模型的分析工具，能够将基于维特比算法的预测过程交互式的展示出来，将模型的黑盒分析变成白盒，分析出了一系列问题。



同时，crf 问题分析工具也应用在了其他 query 分析模块。

- 改造 1：特征建设和模型优化



原始的模型依赖成分分析比较严重，由于成分分析特征本身存在准确率的问题，我们从用户的 query 中总结了一些更加可靠的统计特征。

前缀置信度

描述了片段做前缀的占比。比如望京凯德茂，望京阜通，也有望京单独搜，如果望京在前缀中出现的占比很高，那说明这个片段做 where 的能力比较强。

后缀熵

描述了后缀的离散程度，如望京凯德茂，望京美食，望京首开，后缀很乱，也可以说明片段做 where 的能力。

what 置信度

片段单独搜占比，比如西门经常是在片段结尾出现，但是单独搜比较少，那片段做 what 的能力比较弱。

以特征值域做横轴，where 和 what，label 作为纵轴，就得到了特征-label 曲线。从这几个特征的特征-label 曲线来看，在某些区间下区分度还是很好的。由于 crf 模型只接受离散特征，特征-label 的曲线也指导了特征离散化的阈值选择。对于低频 query，我们通过低频

query 中的高频片段的统计信息，也可以使我们的模型在低频问题上表现更好。

- 改造 2：后验意图选择升级

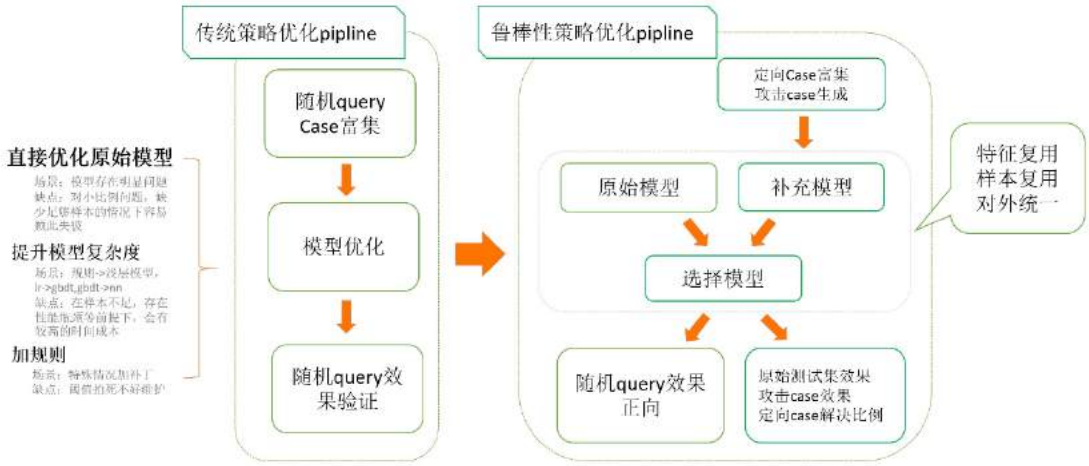
将原始的堆砌的规则升级到 gbdt 机器学习模型，引入了先验特征，在拿到一定收益的同时也使得整个体系更加合理。

- 改造 3：鲁棒性优化

将 ww 中逆序的问题抽象出来，可以归纳为 query 分析中的鲁棒性问题。随着策略优化进入深水区，存在策略提升用户不可感知，攻击 case 容易把系统打穿的问题。

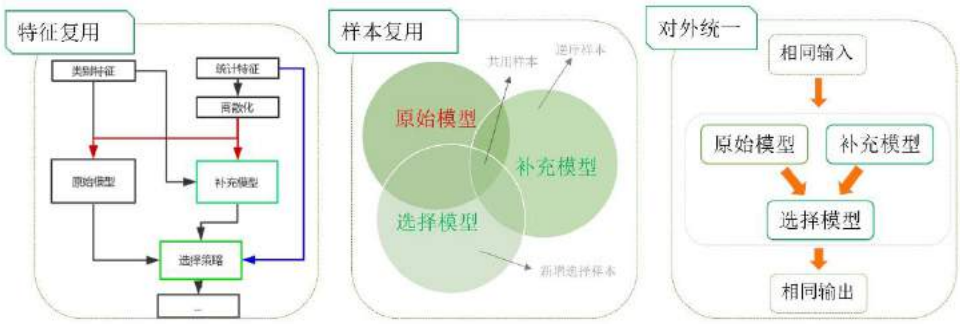
如上图，用户变换 query 中 where 和 what 的顺序，效果可能变差。变换下检索城市，对于同一个知名景点，跳转逻辑不一致。用户区划输错，纠错不能识别，效果变很差。这种模块对非预期的 query 变换，或者更通用地叫做，对上下游的特征扰动的承载能力，我们可以把它叫做模块的**鲁棒性**。

这里我们设计了对于通用鲁棒性问题解决的思路



在不引入复杂模型的前提下，通过构建 ensemble 的浅层模型来优化特定问题，降低了问题解决的成本。

对于 ww 逆序这个特定问题进行了特征复用用本复用，并且模型对外统一



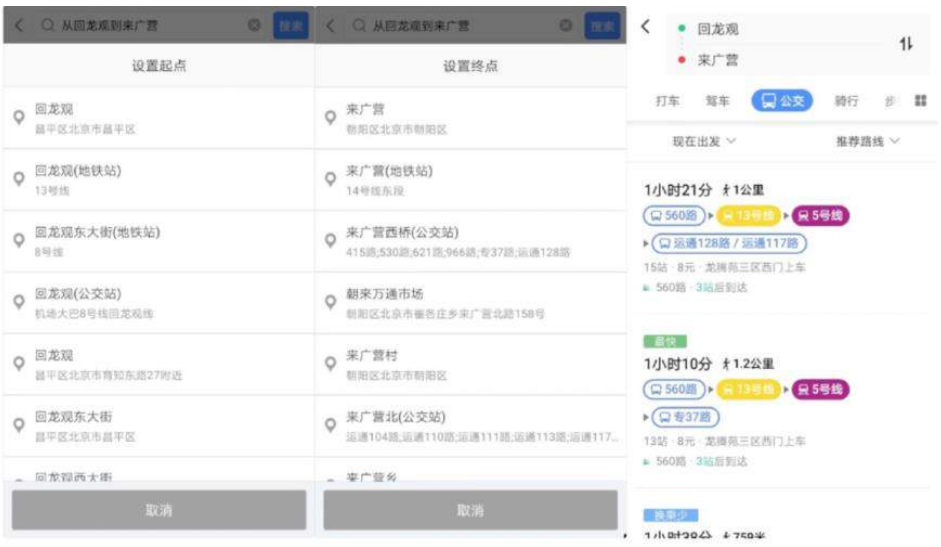
效果上，新的模型在原始测试集效果持平，人工构造攻击 case 集合准召明显提升，目标 case 集合具有可观的解决比例，验证了鲁棒性优化的思路是有效的。

这里基本完成了对于 ww 体系的一个完整的升级，体系中的痛点基本都得到了解决。优化了切分模型，流程更加合理。意图决策上完成了规则到机器学习模型的升级。在优化或者引入浅层机器学习模型的同时尽可能发挥浅层模型的潜力，为从浅层模型升级为深度模型打下基础。

在后续的建模中我们使用字粒度 lstm+crf 模型代替现有的 crf 模型，彻底摆脱了对成分分析特征的依赖，同时通过融合知识信息到 lstm+crf 模型，进一步提升效果。

4.3 路径规划

在高德地图的搜索场景中，一类用户搜索意图为路径规划意图。例如，当用户在高德地图 App 的搜索框中输入“从回龙观到来广营”，点击搜索按钮后，搜索服务能识别出用户的搜索意图为路径规划，并识别出用户描述的起点为“回龙观”，终点为“来广营”，进而检索到对应的 POI 点给下游服务做出路线的规划。



从用户输入中识别路径规划意图，并提取出对应的起终点，这是一个典型的 NLP 任务。早

期的路径规划模块使用的是模板匹配的方式，这种方式开发成本低，能解决大部分常见的路径规划问题，如上面这种“从 A 到 B”类的问题。

但随着业务的不断发展，模块需要解决的问题越来越复杂，比如“坐地铁从西直门到大兴狼垡坐到哪里下车”，“广东到安徽经过哪几个城市”，“去往青岛的公交车有吗”等等各种非“从 A 到 B”模式的问题。由于模板匹配方式没有泛化能力，只能通过不断增加模板来解决，使得模块越来越沉重难以维护。

优化

由于线上所有的搜索 query 都会经过路径规划模块，若是让模型去处理所有的 query，那么模型不仅要解决意图识别问题（召回类问题），又要解决槽位提取问题（准确类问题），对于模型来说是很难同时将这两个任务学好的。因此，我们采取了以下三段式：

模型前使用关键字匹配策略进行简单意图识别，过滤掉大部分非路径规划 query；模型处理疑似路径规划的 query，进行槽位提取；模型后再对模型结果进行进一步检验。

样本和特征

机器学习的样本一般来源于人工标注，但人工标注耗时长成本高。因此我们采取的是自动标注样本方式。通过富集路径规划模式，如“从 A 怎么乘公交到 B”，再用清洗后的随机 query 按照实际起终点的长度分布进行起终点替换，生成大量标注样本。

特征方面，我们使用了成分分析特征及含有关键字的 POI 词典特征。它们主要在如“从这里到 58 到家”这类起终点 中含有关键字的 query 上起着区分关键字的作用。

模型训练

crf 算法是业界常用的为序列标注任务建立概率图模型的算法。我们选取的也是 crf 算法。

效果评估

在验证集准确率召回率，以及随机 query 效果评比上，指标都有了明显的提升。

对于路径规划这样一个定向的 NLP 任务，使用 crf 模型完成了从规则到机器学习模型的升级。作为一个应用层任务，路径规划也很容易被迁移到 seq2seq 的多任务学习模型中来。

5.展望

过去两年随着机器学习的全面应用，以及基于合理性进行的多次效果迭代，目前的地理文本处理的效果优化已经进入深水区。我们认为将来的优化重点在攻和防两方面。

攻主要针对低频和中长尾问题。在中高频问题已经基本解决的前提下，如何能够利用深度学习的技术进行地理文本处理 seq2seq 的统一建模，在低频和中长尾问题上进行进一步优化，获得新一轮的效果提升，是我们目前需要思考的问题。另外，如何更好地融合知识信

息到模型中来，让模型能够具有接近人的先验判断能力，也是我们亟待提升的能力。

防主要针对系统的鲁棒性。如用户的非典型表达，变换 query 等定向的问题，如何能够通过定向优化解决这些策略的死角，提高系统的容错能力，也是我们目前需要考虑的问题。

地图搜索虽然是个垂类搜索，但是麻雀虽小五脏俱全，并且有地图场景下很多有特色的的难点。未来我们需要继续使用业界先进的技术，并且结合地理文本的特点进行优化，理解将更加智能化。

招聘

高德信息研发部诚招 NLP、机器学习、搜索推荐算法专家，Java、PHP、C++高级工程师/专家。职位地点：北京，欢迎有兴趣的同学投递简历到 xi.yang@alibaba-inc.com

机器学习在高德用户反馈信息处理中的实践

作者：沉虑

1.背景

作为国内领先的出行大数据公司，高德地图拥有众多的用户和合作厂商，这为高德带来了海量的出行数据，同时通过各个渠道，这些用户也在主动地为我们提供大量的反馈信息，这些信息是需要我们深入挖掘并作用于产品的，是高德地图不断进步和持续提升服务质量的重要手段。

本文将主要介绍针对用户反馈的文本情报，如何利用机器学习的方法来提高大量用户数据的处理效率、尽可能实现自动化的解题思路。

先来解释一下重点名词。

情报：是一种文本、图片或视频等信息，用来解决高德地图生产或者导航中的具体问题，本质上是指与道路或交通相关的知识或事实，通过一定空间和时间通知给特定用户。

用户反馈：是指用户借助一定的媒介，对所使用的软件等提供一些反馈信息，包括情报、建议和投诉等。

典型的用户反馈类型和选项如下图所示:

2.问题及解法

用户反馈的方式可以通过手机的 Amap 端、PC 端等进行上报，上报时选择一些选择项以及文本描述来报告问题，以下是一个用户反馈的示例，其中问题来源、大类型、子类型和道路名称是选择项，用户描述是填写项，一般为比较短的文本。这些也是我们可以使用的主要特征。

工单号	序号	问题时间	问题来源	大类型	子类型	用户描述	道路名称
	30262	2017/3/30 7:34	导航中页面			回城南路断桥已修复，道路已通	

每个用户在上报了问题之后，均希望在第一时间内问题能够得到解决并及时收到反馈。但是高德每天的用户反馈量级在几十万，要想达到及时反馈这个目标非常的不容易。

针对这些用户反馈信息，当前的整体流程是先采用规则进行分类，其中与道路相关的每条反馈都要经过人工核实，找到用户上报的问题类型和问题发生的地点，及时更新道路数据，作用于导航。

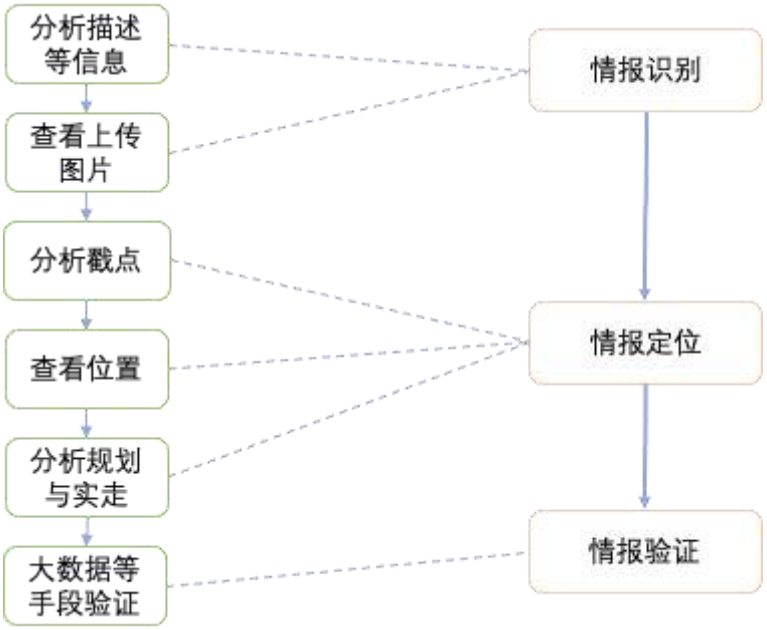
具体一条反馈的操作需要经过情报识别、情报定位、情报验证等环节：

1) 情报识别主要是判断问题类型即给情报打标签：①分析用户上报的信息包括问题来源、大类型、子类型和用户描述等，②查看上传的图片资料，包括手机自动截图和用户拍照。

2) 情报定位主要是找到问题发生的位置信息即定位坐标：①分析用户反馈问题时戳的位置点即戳点的有效性，②查看用户上报问题时车辆行驶的位置即自车位置，③分析用户使用高德软件过程中的规划和实走轨迹等日志信息。

3) 情报验证：通过以上两步确定了情报标签和位置坐标，此环节需要验证情报标签(含道路名称)：①分析影像和大数据热力图或路网基础数据，②查看用户上传的资料和采集的多媒体图片资料。

整个业务处理流程如下图所示：



在处理用户反馈问题整个过程秉持的原则是完全相信用户的问题存在。若用户上报的信息不足以判断问题类型和问题发生地点，则会尽量通过用户规划和实走轨迹等日志信息进行推理得出偏向用户的结论。

目前整个用户反馈问题处理流程存在的主要问题有：规则分发准确率低，人工核实流程复杂、技能要求高且效率低，去无效误杀严重等。

为了解决以上问题，我们希望引入机器学习的方法，以数据驱动的方式提高作业能力。在目标具体实现的探索过程中，我们首先对业务进行拆解及层级化分类，其次使用算法来替代规则进行情报分类，再次工程化拆解人工核实作业流程为情报识别、情报定位和情报验证等步骤，实现单人单技能快速作业，最后将工程化拆解后的情报识别步骤使用算法实现其自动化。

3.机器学习解题

3.1 业务梳理与流程层级化拆解

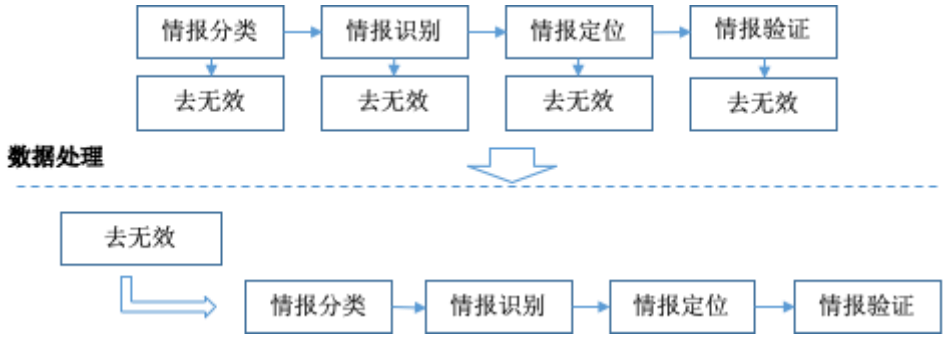
原始的用户反馈问题经由规则分类后，再进行人工情报识别、定位和验证，最终确认问题及其所在是属于近百种小分类项中的哪一个，进而确定上一级分类以及整个层级的对应关系。

由此可以看出，整个问题处理流程只有一个步骤，处理过程相当复杂，对人工的技能要求很高，且效率低下。而且一千个人眼中就有一千个哈姆雷特，个人的主观性也会影响对问题的判断。

针对这种情况，我们对原有业务流程进行梳理和拆解，希望能够利用机器学习和流程自动

化等方式解决其中某些环节，提升整体问题处理的效率。

首先进行有效情报和无效情报的分类即去无效，接着将整个流程拆解为六个层级，包括业务一级、业务二级、业务三级、情报识别、情报定位和情报验证。



如上图所示，拆解后的前三个级别为情报分类环节，只有后三个级别需要部分人工干预，其他级别均直接自动化处理。这样通过层级化、自动化和专人专职等方法极大地简化了问题同时提高了效率。

3.2 业务与模型适配

我们可以看到用户反馈中既有选择项又有输入项，其中选择项如问题来源等都是有默认值的，需要点击后选择相应细分项，用户不一定有耐心仔细选择，有耐心的用户可能会由于不知道具体分类标准而无法选择正确的分类。而用户描述，是需要用户手动输入的内容，是用户表达真实意图的主要途径，是一条用户反馈当中最有价值的内容。

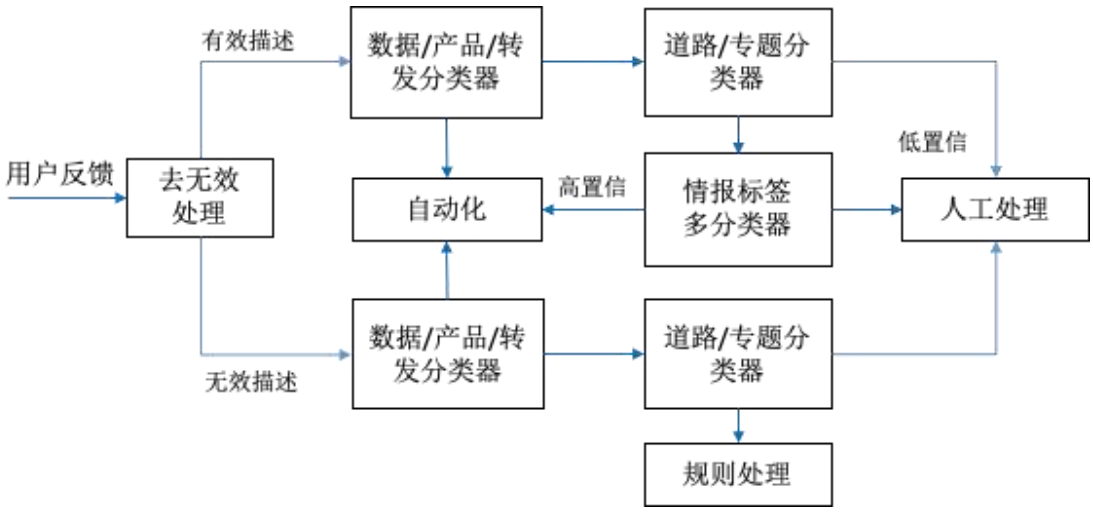
用户描述一般分为三种情况：无描述、有描述但无意义的、有描述且有意义的。前两种称之为无效描述，后一种称之为有效描述。

根据业务拆解结果，业务流程第一步即为去无效，在这之后，我们将有效、无效描述的用户反馈进行区分，分别建立相应的流程进行处理。

1) 有效描述的用户反馈，逐级分类，第一级分为数据、产品、转发三类，其中产品和转发两类直接进行自动化处理，数据类别会在第二级中分为道路和专题，专题是指非道路类的限行、步导、骑行等。

2) 无效描述的用户反馈，进行同样的分类，并走一样的流程，但是样本集和模型是不同的，并且最后没有算法处理的步骤，直接走人工或者规则处理。

3) 最终根据实际业务需要进行层层拆解后形成了下图所示的业务与模型适配的结构。



由以上分析可见，情报分类和情报识别均为多分类的文本分类问题，我们针对各自不同的数据特点，进行相应的操作：

情报分类，每一级类别虽不同，但是模型架构却是可以复用的，只需要有针对性的做微小改动即可。且有以前人工核实过(包含情报识别、情报定位、情报验证等过程)具有最终结果作为分类标签的历史数据集作为真值，样本集获得相对容易。

情报识别，其分类标签是在情报验证之前的中间结果，只能进行人工标注，并且需要在保证线上正常生产的前提下，尽量分配人力进行标注，资源非常有限。所以我们先在情报分类数据集上做 Finetuning 来训练模型。然后等人工标注样本量积累到一定量级后再进行情报识别上的应用。

3.3 模型选择

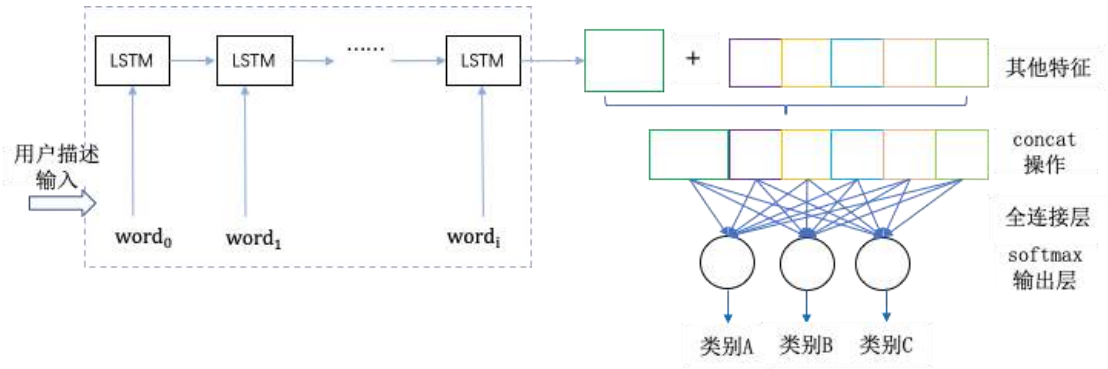
首先，将非结构化的文本用户描述表示成向量形式即向量空间模型，传统的做法是直接使用离散特征 one-hot 表示，即用 tf-idf 值表示词，维度为词典大小。但是这种表示方式当统计样本数量比较大时就会出现数据稀疏和维度爆炸的问题。

为了避免类似问题，以及更好的体现词语之间的关系如语义相近、语序相邻等，我们使用 word embedding 的方式表示，即 Mikolov 提出的 word2vec 模型，此模型可以通过词的上下文结构信息，将词的语义映射到一个固定的向量空间中，其在向量空间上的相似度可以表示出文本语义上的相似度，本质上可以看作是语境特征的一种抽象表示。

其次，也是最重要的就是模型选择，相对于传统的统计学习方法复杂的特征工程步骤，深度学习方法更受青睐，NLP 中最常用的是循环神经网络 RNN，RNN 将状态在自身网络中循环传递，相对于前馈神经网络可以接受更广泛的时间序列结构输入，更好的表达上下文信息，但是其在训练过程中会出现梯度消失或梯度爆炸等问题，而长短时记忆网络 LSTM 可以很好的解决这个问题。

3.4 模型架构

将每个用户反馈情报的词向量结果作为 LSTM 的输入，接着将 LSTM 的最后一个单元的结果作为文本特征，与其他用户选择项问题一起 merge 后作为模型输入，然后经过全连接层后使用 softmax 作为输出层进行分类，得到的 0~1 之间的实数即为分类的依据。多分类的网络架构如下图所示：



4.实战经验总结

理清业务逻辑、确定解题步骤、确认样本标注排期并跑通了初版的模型后，我们觉得终于可以松一口气，问题应该已经解决过半了，剩下的就是做做模型调参和优化、坐等样本积累，训练完模型就可以轻松上线了。

但实际情况却是面临着比预想更多的问题和困难，训练数据量不够、单个模型效果不好、超参设置不理想等问题接踵而至，漫长而艰难的优化和迭代过程才刚刚开始。

4.1 Fine-tuning

选定了模型之后，情报识别首先面临的问题是样本量严重不足，我们采用 Fine-tuning 的办法将网络上已经训练过的模型略加修改后再进行训练，用以提升模型的效果，随着人工标注样本逐渐增加，在不同大小的数据集上都可以取得大约 3 个百分点的提升。

4.2 调参

模型的调参是个修炼内功炼制金丹的过程，实际上取得的效果却不一定好。我们一共进行了近 30 组的调参实验，得出了以下饱含血泪的宝贵经验：

1)初始化，一定要做的，我们选择 SVD 初始化。

2) dropout 也是一定要用的，有效防止过拟合，还有 Ensemble 的作用。 对于 LSTM，dropout 的位置要放到 LSTM 之前，尤其是 bidirectional LSTM 是一定要这么做的，否则直接过拟合。

3) 关于优化算法的选择，我们尝试了 Adam、RMSprop、SGD、AdaDelta 等，实际上 RMSprop 和 Adam 效果相差不多，但基于 Adam 可以认为是 RMSprop 和 Momentum 的结合，最终选择了 Adam。

4) batch size 一般从 128 左右开始调整，但并不是越大越好。对于不同的数据集一定也要试试 batch size 为 64 的情况，没准儿会有惊喜。

5) 最后一条，一定要记住的一条，尽量对数据做 shuffle。

4.3 Ensemble

针对单个模型精度不够的问题，我们采用 Ensemble 方式解决，进行了多组试验后，最终选定了不同参数设定时训练得到的最好模型中的 5 个通过投票的方式做 Ensemble，整体准确率比单个最优模型提高 1.5 个百分点。

另外为了优化模型效果，后续还尝试了模型方面的调整比如双向 LSTM 和不同的 Padding 方式，经过对比发现在情报识别中差异不大，经分析是每个用户描述问题的方式不同且分布差异不明显所致。

4.4 置信度区分

当情报识别多分类模型本身的结构优化和调参都达到一定瓶颈后，发现模型最终的效果离自动化有一定的差距，原因是特征不全且某些特征工程化提取的准确率有限、类别不均衡、单个类别的样本数量不多等。

为了更好的实现算法落地，我们尝试进行类别内的置信度区分，主要使用了置信度模型和按类别设定阈值两种办法，最终选择了简单高效的按类别设定阈值的方法。

置信度模型是利用分类模型的标签输出结果作为输入，每个标签的样本集重新分为训练集和验证集做二分类，训练后得到置信度模型，应用高置信的结果。

在置信度模型实验中，尝试了 Binary 和 Weighted Crossentropy、Ensemble 的方式进行置信度模型实验，Weighted Crossentropy 的公式为：

$$t * -log\left(sigmoid\left(y\right)\right) * pos\_weight + \left(1 - t\right) * -log\left(1 - sigmoid\left(y\right)\right)$$

为了避免溢出，将公式改为：

$$\left(1 - \mathrm{t}\right) * \mathrm{y} + l * \left(log\left(1 + exp\left(-abs\left(y\right)\right)\right) + max\left(-y, 0\right)\right)$$

其中，表示：

$$l = \left(1 + \left(\mathrm{pos\_weight} - 1\right) * \mathrm{t}\right)$$

实验的结果是 Binary 方式没有明显效果提升，Ensemble 在 95%置信度上取得了较高的召回率，但是没有达到 98%置信度的模型。

借鉴了情报分类算法模型落地时按照各个类别设定不同 softmax 阈值的方式做高置信判断即按类别设定阈值的方式，在情报识别中也使用类似的方法，取得的效果超过了之前做的高置信模型效果，所以最终选择了此种方式，这部分可以很大地提高作业员的作业效率。同时为了减少作业员的操作复杂性，我们还提供了低置信部分的 top N 推荐，最大程度节

省作业时间。

5. 算法效果及应用成果

5.1 情报分类

算法效果：根据实际的应用需求，情报分类算法的最终效果产品类准确率 96%以上、数据类召回率可达 99%。

应用成果：与其他策略共同作用，整体自动化率大幅提升。在通过规则优化后实际应用中取得的效果，作业人员大幅度减少，单位作业成本降低 4/5，解决了用户反馈后端处理的瓶颈。

5.2 情报识别

算法效果：根据使用时高置信部分走自动化，低置信走人工进行标注的策略，情报识别算法的最终效果是有效描述准确率 96%以上。

应用成果：完成情报标签分类模型接入平台后通过对高低置信标签的不同处理，最终提升作业人员效率 30%以上。

6. 总结与展望

通过此项目我们形成了一套有效解决复杂业务问题的方法论，同时积累了关于 NLP 算法与业务紧密结合解题的实战经验。目前这些方法与经验已在其他项目中很好的付诸实施，并且在持续的积累和完善中。在不断提升用户满意度的前提下，尽可能的高效自动化的处理问题，将产品的每一个细节争取做到极致， 是我们前进的原动力和坚持不懈的目标。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

高德网络定位之“移动 WiFi 识别”

作者：佳郎

1.导读

随着时代的发展，近 10 年来位置产业蓬勃发展，定位能力逐渐从低精度走向高精度，从部分场景走向泛在定位。设备和场景的丰富，使得定位技术和能力也不断的优化更新。定位能力包括 GNSS、DR（航迹推算）、MM（地图匹配）、视觉定位和网络定位等。

其中网络定位是通过客户端扫描到的 WiFi 和基站信息来进行定位的一种定位方式。网络定位能力是 GNSS 定位的有力补充，在 GNSS 无法定位或者定位较慢的时候，网络定位都可以快速给出位置。网络定位能力也是高德能够深植于各类手机厂商（提供系统级网络定位能力）和 APP（出行、社交、O2O、P2P、旅游、新闻、天气等诸多领域）的原因之一。

要做到通过 WiFi 和基站来定位，我们需要通过亿级数据来挖掘出 WiFi 和基站的类型、位置、指纹等各种信息。这些信息的挖掘，历史上是通过一系列的人工经验策略来进行的，人工规则的历史局限带来了所挖掘信息较低的准召率，为了进一步提升高德网络定位能力，我们需要卸下以往的包袱，从方法上进行改变。

2.如何定义“网络定位”

网络定位分为离线训练和在线定位两个过程：

- **离线训练**：是在用户有 GPS 位置时采集周边的 WiFi 和基站（以下统称为 AP）信息，通过对采集数据进行聚类和关联，得到两类数据产品：AP 库和指纹库。

- **在线定位**：与离线训练的过程正好相反，当用户没有 GPS 定位时，可以通过扫描到的周边 WiFi 和基站信号，结合离线训练出的 AP 库和指纹库来进行实时定位。

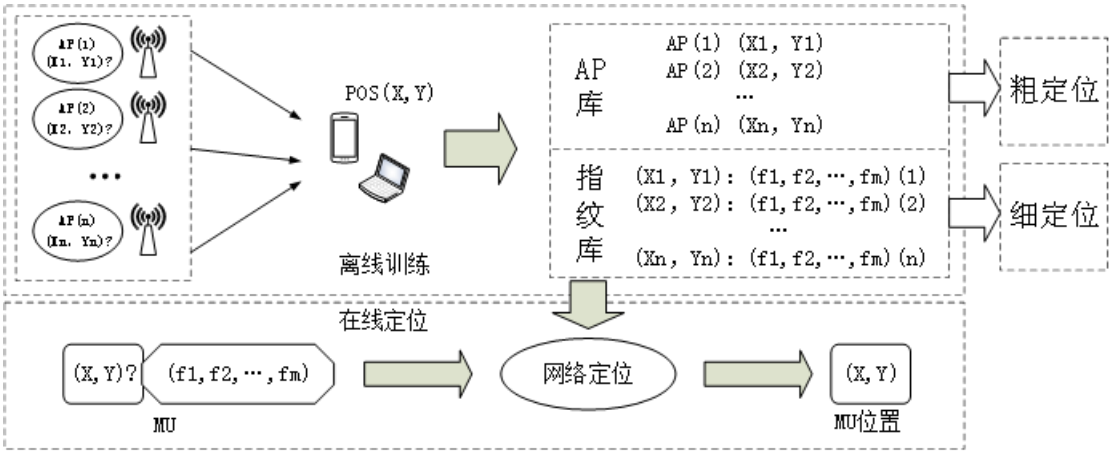
AP 库和指纹库这两类数据产品中：

- **指纹库**：以物理坐标位置对应的特征指纹信息为内容，这些特征指纹信息可以包括扫描到的 WiFi 或者基站的信号强度分布，采集点频次等统计信息，也可以是通过神经网络提取出的特征信息。

- **AP 库**：以 WiFi 的 mac 地址或者基站的 ID（gsm 基站为 mcc_mnc_lac_cid，cdma 基站为 mcc_sid _bsid_nid ）为主键，以 WiFi 或者基站的物理坐标信息（经纬度或者地理栅格坐标信息）为内容。

典型的 AP 库数据只包含挖掘出的物理坐标信息和覆盖半径，这种“点圆模型”是对 AP 发射信号的一种理想化，没有考虑任何实际场景中的信号遮挡、反射等情况，所以 AP 库大多用

来进行粗略定位。而指纹库直接与位置相关，可以刻画比“点圆模型”更细致的分布信息，所以指纹库可以用来进行精细定位。

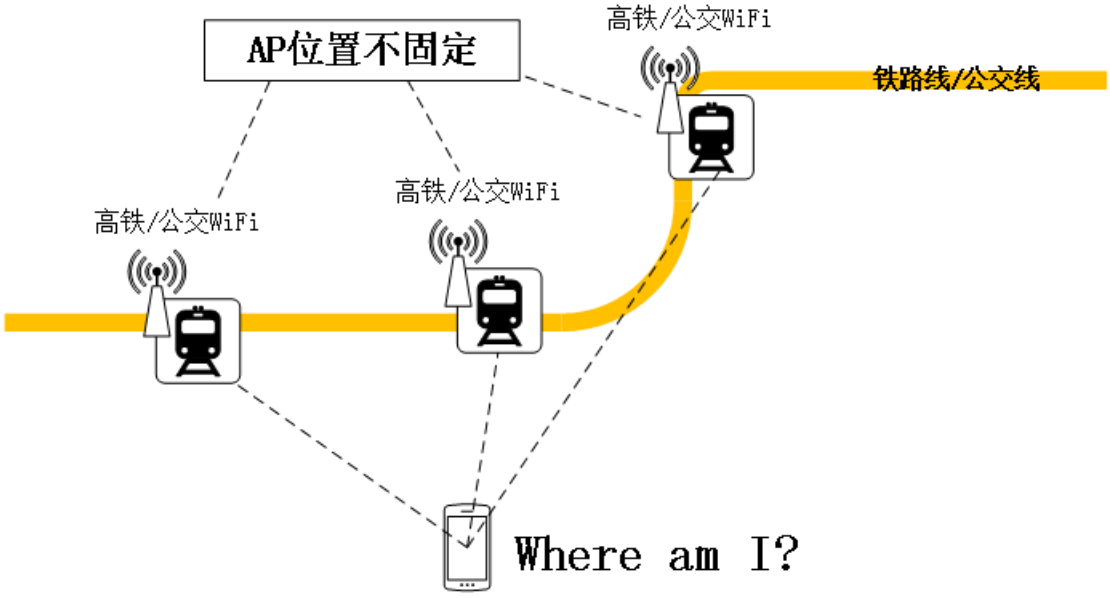


高德的指纹库主要包括特有的室内指纹和全场景指纹信息两种。

3.“网络定位”的问题

网络定位的基本思路类似聚类，假设用户手机扫描到的 AP 列表中的 AP 的位置均比较固定，则我们可以以这些 AP 位置为锚点，来确定用户位置。现实世界中，锚点（即 AP 库中的 AP）的位置通过大数据来进行挖掘，并不一定完全准确，甚至出现严重错误。

针对 WiFi 而言，移动 WiFi、克隆 WiFi、搬家 WiFi 等都可能造成 AP 位置的错误。移动 WiFi 包含手机热点，4g 移动路由器，公交车/地铁/高铁上的 WiFi 热点等，这些 WiFi 的移动属性较强，位置频繁变化，如下图所示。



如果以移动 WiFi 作为锚点，因为这些锚点的位置不固定，极可能会导致用户的定位出现极大误差。克隆 WiFi 指不同的 WiFi 设备使用了同一个 mac 地址，国内的腾达和斐讯等路由器厂商制造了大量这样的 WiFi 设备（例如大部分 mac 前缀为“c8:3a:35”的即为腾达的克隆 WiFi），克隆 WiFi 导致 AP 库中同一个 mac 地址对应的锚点位置有多个。搬家 WiFi 指某些因为搬家

而发生位置变化的 WiFi，数据挖掘存在一定的滞后性，搬家后 AP 库中的位置未及时更新，也会造成定位错误。

因为大误差的 badcase 严重损害用户体验，我们必须要将这些非固定 WiFi 的属性在 AP 库中标记出来。

历史上，高德是通过一系列简单的人工规则对这些 WiFi 的属性进行分类的。例如，通过采集点覆盖范围较大来判定移动 WiFi，通过 mac 前缀来判定克隆 WiFi 等。人工规则的缺点是准召率不高，训练分类模型就成了一个自然的选择。

鉴于 badcase 中最严重的问题是移动 WiFi 的准召率不高，下面我们就尝试使用监督学习的方法来进行“移动 WiFi 识别”。

4.如何实现“移动 Wifi 识别”

4.1 样本提取

AP 库中的 WiFi 数量十分庞大，如果我们在 AP 库中随机抽取样本进行人工标注，那大部分标注的结果可能是人工规则判定正确的样本，为了尽可能低成本获取有效的标注样本，我们借鉴主动学习的思路，不断抽取模糊样本进行标注，快速迭代使得模型稳定。

我们根据人工规则的判定结果提取了一批确定性较高的样本，使用人工强特征训练第一版模型，之后将第一版模型的预测结果与线上人工规则的结果进行全量比较，提取出模糊样本进行人工标注。在标注样本的过程中发现问题，持续特征工程，不断迭代模型。

这里模糊样本的定义包含三种 :预测结果与上一版模型的结果不同，预测概率值在 0.5 附近，预测结果在不同训练周期内存在波动（例如昨天识别是移动 WiFi，今天识别是非移动）。

4.2 特征提取

4.2.1 移动 WiFi vs 克隆/搬家 WiFi

第一版模型中，我们使用了一些采集聚集程度相关的特征。

名称	描述
ratioX	聚簇寻找中心点，中心点向外 X 米圆形范围内的定位点占总定位数的比例
areaSqare	定位点覆盖的矩形围栏范围的面积

模型迭代过程中，我们遇到的第一个问题是移动 WiFi 与克隆 WiFi 或搬家 WiFi 比较容易混淆。下面几幅图分别画出了固定 WiFi、移动 WiFi、克隆 WiFi、搬家 WiFi 的定位点散布的实例。

类型	采集点
固定 WiFi	
移动 WiFi	
克隆 WiFi	
搬家 WiFi	

可以看到，如果仅仅使用定位点的聚集程度来分类，那克隆 WiFi 和搬家 WiFi 的定位点也比较分散，极易与移动 WiFi 混淆。所以我们先使用聚簇算法，将采集点局部聚集的点集合成

不同的簇，在每个簇中计算定位点的散布程度，再将所有簇的散布程度求平均值等，获取平均意义上的聚集程度。

4.2.2 多维度提取特征

为了进一步提升分类的准召率，我们不仅从定位点的聚集维度来提取特征，还增加了信号强度、关联特征、IP 特征、时间特征等，以下进行简要介绍：

信号强度信息：（和上节中的聚集特征一起，统称为采集特征）移动设备与非移动设备采集点的信号强度在去除设备差异性之后，分布存在差异性。

关联特征：关联信息是指当设备扫描到的一次 WiFi 列表中，列表中所有 WiFi 两两之间就算产生了一次关联（或称邻居）关系，统计 WiFi 周边关联的 WiFi 和基站信息，可以描述出 WiFi 的移动属性。

IP 特征：固网 IP 和移动网的 IP 存在一定隔离，移动 WiFi 设备的上游一般是通过基站连接的移动网，固定 WiFi 设备的上游一般是通过 ADSL 等连接的固网。

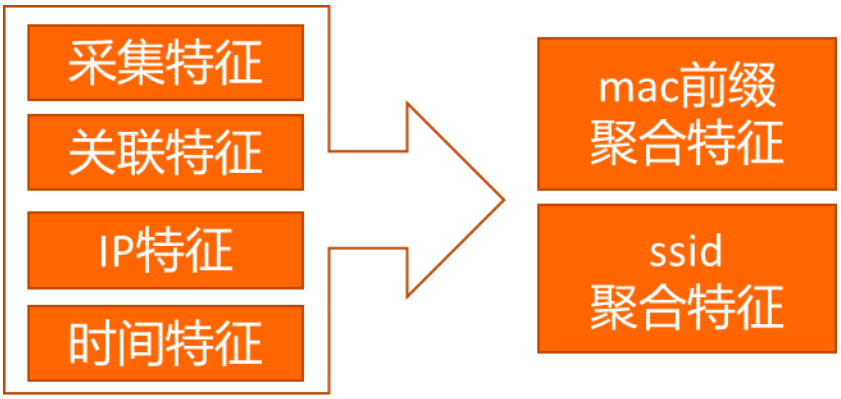
时间特征：固定 WiFi 一般是长时间连接电源的，而移动 WiFi 一般是临时在某些地方和时间短暂出现的。

4.2.3 聚合特征

在 AP 库中，存在一部分 WiFi 定位信息不够充分，即使是人工标注也存在着非常大的不确定性，这些定位信息不够充分的 WiFi，我们称之为“弱信息 WiFi”。

对于这类 WiFi，我们只有通过 ssid 和 mac 前缀来进行辅助判断。因为 ssid 中包含了一些诸如“iPhone”、“个人热点”、“oppo”、“shouqiyueche”（首汽约车）、“往返免费”、“tp-link”等能够表明设备属性的信息。另一方面，mac 前缀（mac 信息的前半部分）代表了厂商信息。基于这些辅助信息，我们可以在其他信息不够充分的情况下辅助推测 WiFi 的类别属性。

我们将基础特征（采集特征、关联特征、IP 特征、时间特征）中较为重要的 TOP_N 维特征按照 ssid 和 mac 前缀进行聚合，聚合函数为中位数（median）和总体标准差（stddev）。这样，聚合特征体现了一类 WiFi 共有的特征，针对弱信息 WiFi，我们就可以通过集体的特征来推测出个体的属性。



5.应用场景

除了提升网络定位能力，移动 WiFi 的识别还有更多用武之地，例如手机热点的识别，室内外的判断，建筑物和 POI 级别的定位等等。其中一个例子就是判断当前设备所连接的 WiFi 是否为移动热点（如 4g 路由器，手机热点等），在视频类的 APP 中，可以通过判别当前用户连接的 WiFi 是否为移动热点，从而控制是否进行视频的自动播放或缓存，给予用户提示性信息。

6.小结

最终我们使用随机森林来训练分类模型，经过特征选择和模型参数调整之后，最终得到的模型，移动 WiFi 的准召率均优于 99.8%。高德网络定位的精度也因此得到了较大提升，尾部大误差 badcase 降低了 18%左右。

网络定位作为一种低功耗的定位手段，不仅在 GNSS 无法触达的地区（例如地铁、室内等场景）为普通用户带来辅助的定位信息，而且在某些急救和寻人的场景中发挥了重要的作用。未来，随着 5G 通信技术的开展，将迎来更加精准的网络定位能力。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

车载多传感器融合定位方案：GPS +IMU+MM

作者：箕裘

1.导读

高德定位业务包括云上定位和端上定位两大模块。其中，云上定位主要解决 Wifi 指纹库、AGPS 定位、轨迹挖掘和聚类等问题；端上定位解决手机端和车机端的实时定位问题。近年来，随着定位业务的发展，用户对在城市峡谷（高楼、高架等）的定位精度提出了更高的要求。

特别是车机端定位，由于定位设备安装在车上，一方面，它可以搭载更丰富的定位传感器来解决特殊场景的问题，另一方面，各个传感器之间相互固连，有利于高精度的算法设计。这两点为车机端进一步提高导航精度的提供了可能。

城市峡谷一直是车机端定位的痛点。原因是城市峡谷的环境使用户无法接收到 GPS 信号或 GPS 信号受干扰，导致 GPS 无定位结果或定位精度差。这是“有源定位”固有的缺点，无法从算法上来克服。

针对这个问题，以 GPS+IMU 的多传感器融合方案越来越受到重视，因为“无源定位”的 IMU 恰好可以弥补 GPS 的短板。此外，车机还可以搭载里程计、视觉设备形成更丰富的多传感器融合方案。

对高德而言，地图数据是定位业务的灵魂。多传感器融合只是定位业务中的一部分，如何把多传感器与地图数据结合起来，始终是我们在思考的问题。

针对车机应用，我们使用 GPS、IMU、里程计等传感器，结合高德地图的地图优势，提出了一种结合地图匹配（Map Matching）的多传感器融合算法——GPS/IMU/MM 融合（软件+硬件的解决方案）。

本文概述了车载多传感器融合定位项目背景，该项目确立是为了向用户提供好的导航定位服务。为了解决**用户反馈的三大痛点问题：偏航重算、无法定位和抓路错误**，结合算法和数据，提出了一套软件+硬件的解决方案。最后，用实测数据验证对用户反馈问题的改善程度。

2.车载应用的痛点

- 偏航重算：是指在高架或城市峡谷，信号遮挡引起位置点漂移。
- 无法定位：是指在无信号区域（停车场、隧道）推算的精度低，导致出口误差大。
- 抓路错误：是指主辅路、高架上下抓路错误。

其中，导致偏航重算和无法定位直接原因是 GPS 定位精度差和 DR 航位推算精度差。GPS 定

位精度由观测环境决定，难以改善；DR 航位推算精度与 DR 算法性能有关，尤其是里程计系统误差和陀螺零偏的标定精度。对于抓路错误，直接原因是正确道路与误抓道路相隔太近，受定位精度限制无法区分 ;根本原因是只使用位置信息进行抓路，没有发挥其它数据的价值。

图 1 用户痛点问题

3.相关名词

GPS(GlobalPositioning System)：指美国国防部研制的全球定位系统。用户设备通过接收 GPS 信号，得到用户设备和卫星的距离观测值，经过特定算法处理得到用户设备的三维坐标、航向等信息。使用不同类型的观测值和算法，定位精度为厘米级到 10 米级不等。GPS 的优点是精度高、误差不随时间发散，缺点是要求通视，定位范围无法覆盖到室内。

IMU(Inertial measurementunit)：指惯性测量单元。包括陀螺仪和加速度计。陀螺仪测量物体三轴的角速率，用于计算载体姿态；加速度计测量物体三轴的线加速度，可用于计算载体速度和位置。IMU 的优点是不要求通视，定位范围为全场景；缺点是定位精度不高，且误差随时间发散。GPS 和 IMU 是两个互补的定位技术。

MM(Map matching)：指地图匹配。该技术结合用户位置信息和地图数据，推算用户位于地图数据中的哪条道路及道路上的位置。

4.技术方案

车机融合定位项目解决的是道路级的定位问题，受限于硬件性能，目前市场上通用的技术方案有两种，如下表 1 所示：

方案	代表	偏航重算	无法定位	抓路错误
软件 (GNSS+MM)	Apple Google	部分可解	不可解	不可解
硬件 (GNSS+IMU)	奔驰 Trimble Ublox	部分可解	可解	不可解

表 1 通用方案

这两种技术方案涉及到 3 种技术手段，在场景覆盖和精度上，它们各有所长，互相补充。如表 2 所示：

车载多传感器融合定位方案：GPS +IMU+MM

技术	优势	局限
卫星定位（GNSS）	➢ 全局、绝对定位 ➢ 低成本	➢ 信号易受干扰 ➢ 不能解决头部问题
地图匹配（MM）	➢ 位置约束 ➢ 场景化	➢ 提升定位精度，本身无定位能力 ➢ 解决部分偏航重算问题
惯性导航（IMU）	➢ 输出连续可靠 ➢ 无需外部依赖	➢ 误差累积发散 ➢ 解决无法定位问题

表 2 技术手段

表 1 表明，目前市面上存在的通用方案并不能完全解决偏航重算、无法定位和抓路错误这三个问题，尤其是抓路错误。为此，在技术层面上，我们将两套通用方案进行融合，提出了一套软+硬（GNSS+MM+DR）方案；在算法层面上，依靠高德的数据优势，以数据融合模块为核心，一方面提高定位结果可靠性，弥补硬件性能上的不足，另一方面对抓路错误问题进行专门的算法设计。

更进一步，将用户反馈的三个问题解构为算法上解决的三个问题：器件误差标定、场景识别和数据融合。如图 2 所示：

图 2 业务问题解构图

5.功能模块

车机融合定位包括数据适配层（DataAdaptive Layer）、算法支撑层(Aided Navigation Layer)和融合层(NavigationLayer)。数据适配层负责将不同输入标准化、将信号同步；算法支撑层计算中间结果，为融合层服务；融合层是整个系统的核心，它负责融合算法支撑层输出的数据，得到可靠的导航信息。图 3 列出了各层所处位置及每个层的具体功能模块：

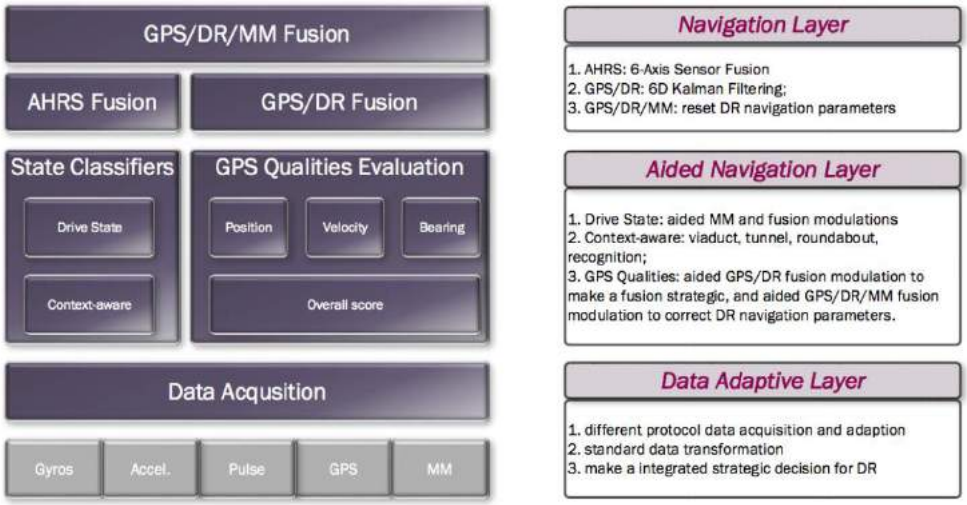


图 3 算法框架

下面，将功能模块分为**基础模块**和**特色功能**两个部分分别进行介绍。

5.1 基础模块

5.1.1 GPS 质量评估

GPS 质量评估模块的功能是计算 GPS 位置、速度、航向角和全局可靠性指标。根据可靠性指标的大小将其投影到状态空间（GOOD、DOUBT、BAD、ABNORMAL）中，状态空间的值表征 GPS 数据质量的好坏。如图 4 所示：

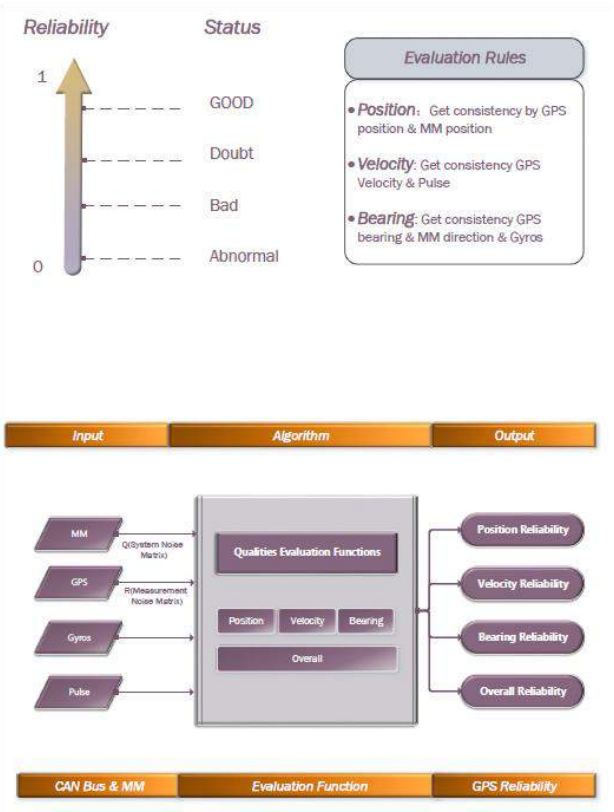


图 4 GPS 质量评估模块

评估 GPS 质量有两个目的：第一，决定是否使用 GPS 数据进行器件误差标定或某些状态的判断（如转弯行为、动静状态等）；第二，在数据融合模块，为设定 GPS 观测量的方差—协方差阵提供参考。

5.1.2 器件补偿

无 GPS 信号环境时，定位只能依靠 DR 算法。DR 算法精度主要取决于 IMU（陀螺仪和加速度计）和测速仪的误差，陀螺仪误差将引起位置误差随时间的二次方增长，测速仪误差将引起位置误差随时间线性增长，如图 5 所示：

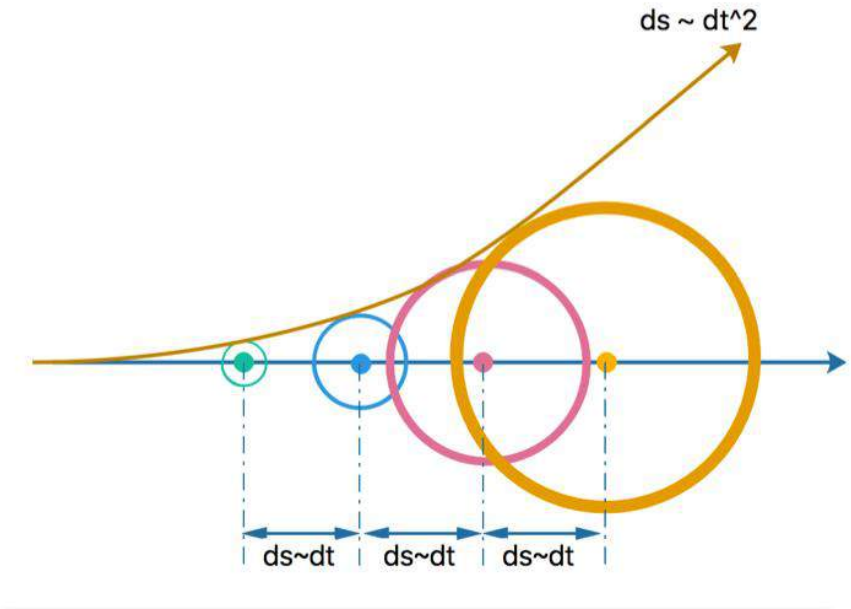


图 5 位置误差与陀螺仪误差的关系

为改善无 GPS 信号环境的定位精度，必须进行器件误差补偿。

补偿模块的主要功能是利用 GPS 数据来补偿速度敏感器误差参数（比例因子）和 IMU 的误差参数（陀螺仪天向比例因子和陀螺仪三轴零偏）。补偿的目的是在无 GPS 信号或弱 GPS 信号的场景，仅靠 DR 算法也能得到较为可靠的导航信息。

5.1.3 DR 算法

DR(DeadReckoning，航位推算)算法是指已知上一时刻导航状态（状态、速度和位置），根据传感器观测值推算到下一时刻的导航状态。DR 算法包括姿态编排和位置编排两个部分。

姿态编排使用的是 AHRS（Attitude andheading reference system ）融合算法，处理后输出车机姿态信息。姿态编排流程如图 6 所示：

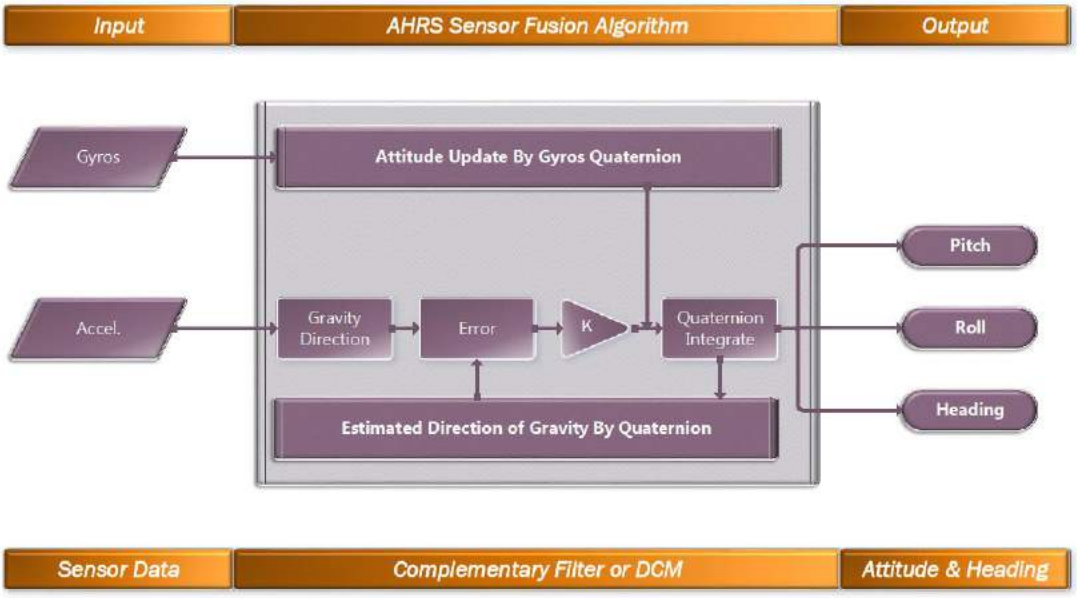


图 6 AHRS 融合算法

位置编排是指结合姿态编排结果，对测速仪观测值进行积分后得到车机位置。

5.1.4 融合算法

车机融合定位项目使用了 GNSS、MM 和 DR 三项技术，每项技术能够提供全部或部分车机导航信息，如表 3 所示。以位置信息为例，GNSS、MM 和 DR 都输出车机位置，但由于不同技术手段有各自的误差源，致使不同技术的定位结果并不相同。

技术	位置	速度	姿态
GNSS	提供	提供	部分提供（航向角）
MM	提供	——	——
DR	提供	提供	提供

表 3 GNSS/MM/DR 提供的导航信息

因此，融合算法有两个目的：第一，将不同技术的导航信息融合成唯一导航信息，使之可靠性高于未融合前的；第二，估计器件误差（陀螺仪零偏、测速仪尺度误差和导航误差等）。

融合算法基于 Kalman 滤波实现，其关键在于模型建立和模型参数设置。Kalman 滤波模型由状态转移方程和观测方程构成。状态转移方程表示相邻导航状态之间的转移关系，它通过构建导航误差微分方程实现；模型参数是指状态转移噪声和观测噪声，观测噪声的设置与 GPS 质量评估模块相关。

经 Kalman 滤波处理后，得到导航误差的最优估值，如图 7 所示。即经过补偿得到了导航信息的最优估值。

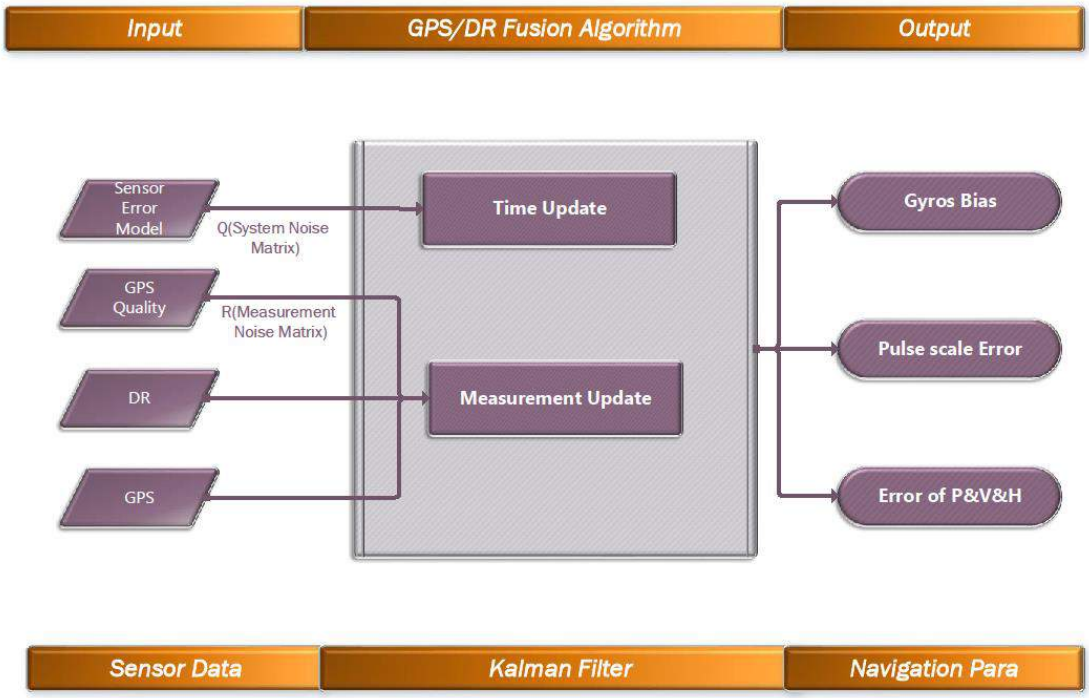


图 7 融合算法模块

5.2 特色功能

5.2.1 主辅路识别

以往的主辅路识别策略是通过 DR 输出的位置和方向与两条候选路的关系，选择最接近的候选路作为输出。但对于道路级定位系统而言，DR 输出误差与两条路的差异在同一量级，误判的概率较高，所以，需要从一些驾驶特征来解决此问题，例如，路口附近的转弯、变道等。

图 8 主辅路街景

如图 9 所示，具体步骤为：

- 提取驾驶行为特征，求特征信息的转移概率。

- 根据 DR 精度分类计算卷积和，求最终概率。

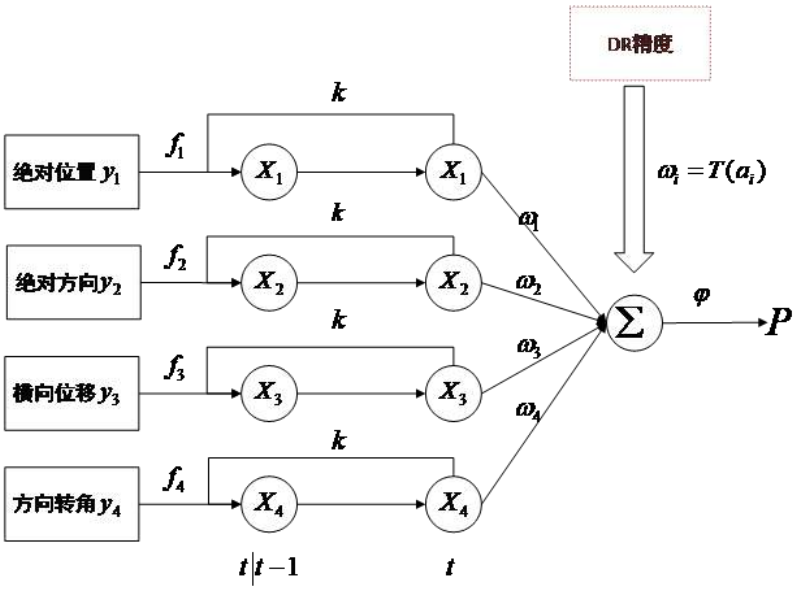


图 9 主辅路识别算法

5.2.2 高架识别

过去，高架识别策略是通过高程积分和阈值法来判断，识别效果受坡角误差和速度误差的影响。其中，速度误差与高程积分误差成正比，是影响高架识别准确率的主要原因。为克服这一缺点，我们结合 MM 技术，计算道路坡度与输出 pitch 角的接近程度（如图 11 所示），以避免引入速度误差。高架识别流程如图 12 所示：

图 10 高架街景

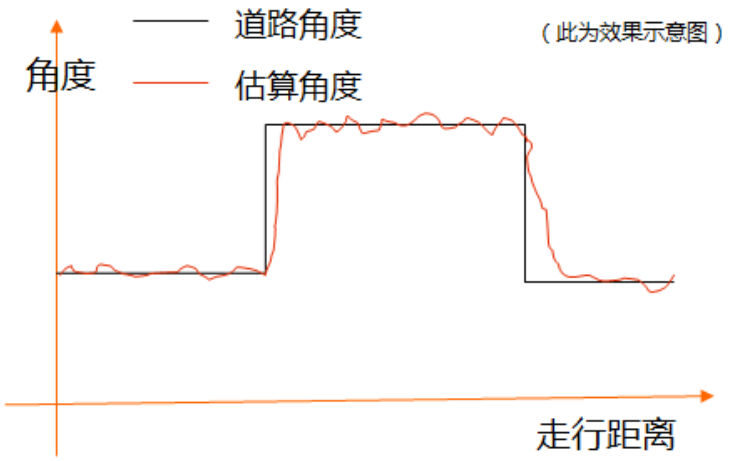


图 11 估算角度与道路角度匹配

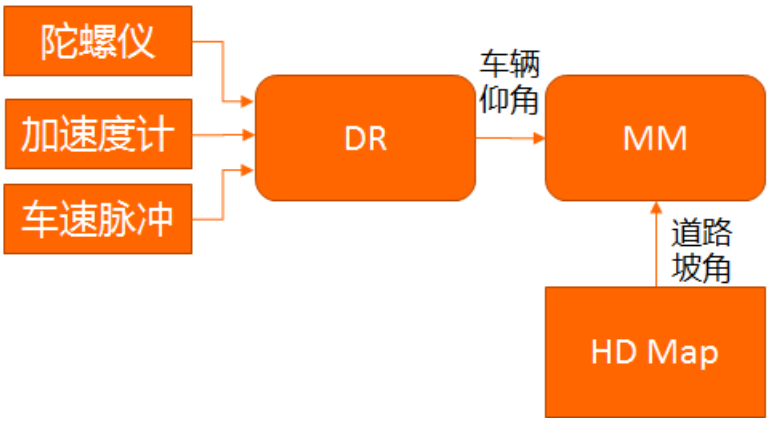


图 12 高架识别流程

5.2.2 停车场识别

停车场识别是新增模块，是停车场定位导航的前置工作。停车场定位导航的主要目的是将车机用户导航到指定的停车位，其中涉及到室内外场景地图切换、层与层地图切换和导航等一系列问题。停车场识别的目的就是为这地图切换提供支持。

停车场内容包括进出停车场识别和跨层识别。进出停车场识别是指利用停车场无 GPS 信号、上下坡、低速、高程变化等一系列特征判断车机是否进出停车场。停车场跨层识别是指利用上下坡、高程变化等特征判断车机是否在停车场内有跨层行为。识别流程如图 13 所示：

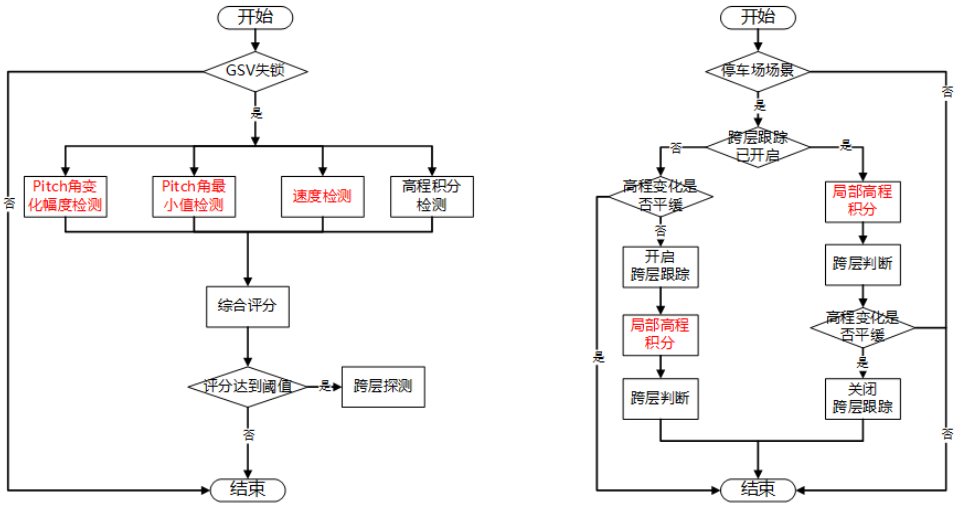


图 13 停车场识别流程图 （左：进出停车场识别，右：跨层识别）

6.效果

为验证本项目算法的效果，我们采集了实测数据，并从以下两方面验证：

- 验证算法对用户痛点问题（偏航重算、无法定位和）的改善程度
- 与竞品及高德手机定位端产品性能的比较。

6.1 融合抗漂

针对高架和城市峡谷的偏航重算（位置漂移）问题，车机算法做了以下两点改进：

- 多元参考：结合运动趋势、传感器信息和地图数据，将 GPS 可靠性归一化。
- 场景分类：参考地图道路属性和 GPS 信号分布判断是否有遮挡。

在高架下采集两圈数据，使用车机软件和市场某款同类软件进行处理，效果如图 14 所示。从近半年的测试来看，在 GPS 受遮挡的场景下，本项目的抗漂能力明显优于传统方案。

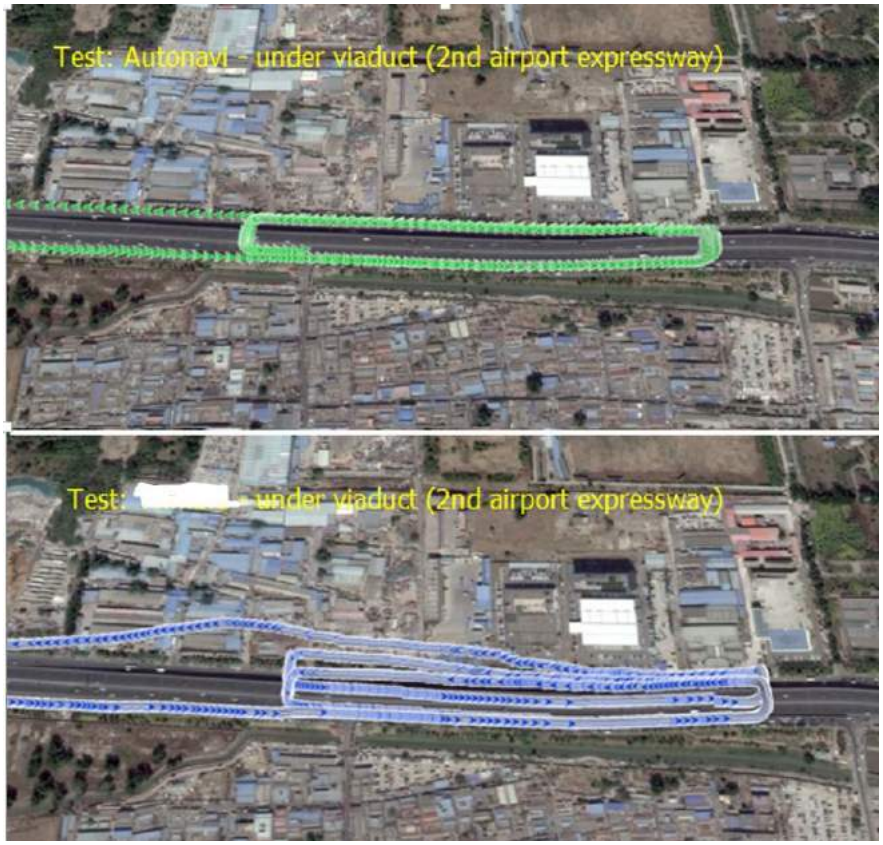


图 14 融合抗漂效果（上图为车机算法，下图为市场某同类产品）

6.2 器件标定

为验证有无陀螺仪动态零偏估计对 DR 方向和位置解算精度的影响, 本项目采集了望京 soho 停车场的数据，经解算，效果如下。测试表明，经动态零偏补偿后，DR 精度明显提高：

零偏：动态零偏估计保证陀螺仪误差量级为 0.01 度/s。

方向：停车场出口出的方向误差减小至 40%以内，方向精度提升 2 倍以上。

位置：停车场出口处的位置误差减小至 25%以内，位置精度提升 4 倍以上。

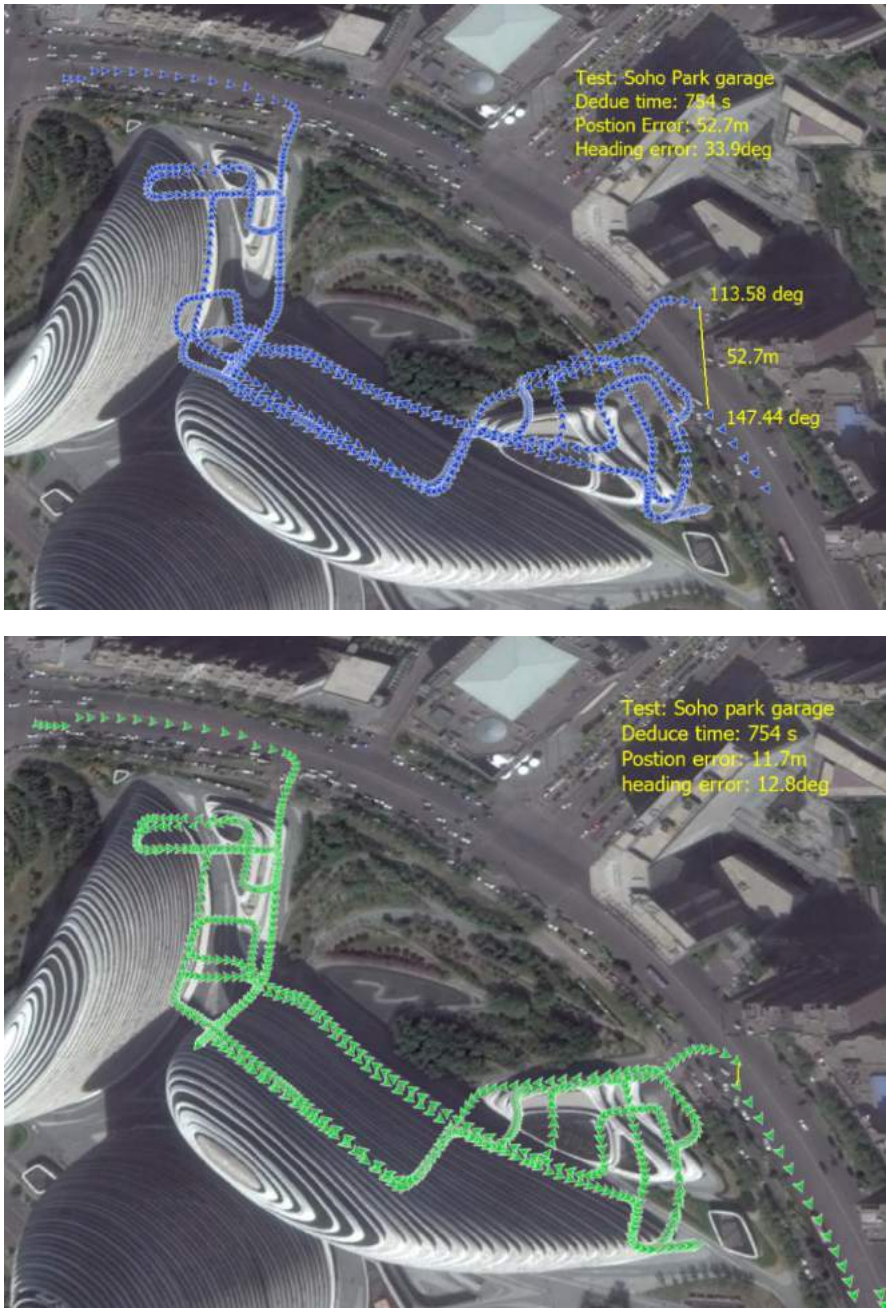


图 15 soho 停车场（左：无动态零偏估计，右：有动态零偏估计）

6.3 主辅路识别

为计算主辅路识别的成功率，统计了近千条主辅路的识别效果，识别率达到 90%以上，大于某厂商产品的 75%。

6.4 横/纵向对比

最后，我们与市面的中高端竞品进行了横向对比，与高德手机端定位产品进行了纵向对比。横向对比结果表明，在器件成本不到竞品成本 10%的情况下，不超过某一阈值的位置误差、方向误差和速度误差的占比均在 90%以上，相对竞品，提高了 1%~5%。主辅助路识别准确率

优于 90%，相对于竞品提高了 15%。

纵向对比结果表明，在不同场景（高架，城市峡谷，环岛，停车场出口等）下，不超过某一阈值的位置误差占比提升 15%~60%不等，这是因为车机算法对特殊场景（无 GPS 或弱 GPS 场景）进行了专门的算法设计和优化。全场景下的位置误差占比提升约 20%。

7.小结

针对用户提出的三大痛点问题，本文结合多传感器融合和地图匹配，提出了一套车载多传感器融合定位方案，并应用于实际，提高了在城市峡谷中的定位精度，并且取得了不错的效果。

然而，城市峡谷的定位精度问题很难彻底解决，它似乎是一个没有终点的难题。为此，站在用户的角度，我们需要不停思考：需要什么样的传感器技术、应该设计什么样的算法、如何挖掘数据的最大价值。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

深度学习在道路封闭挖掘方案的探索与实践

作者：洛德

1.导读

还原真实世界是每个地图技术人的追求，愿意为此付出不懈努力。随着地图静态路网的完善，道路上动态发生的事件，对用户出行的影响逐渐突显出来，尤其是道路上发生的封闭事件。

为了挖掘道路上的封闭事件，高德技术团队设计了一套半监督的深度学习方案。下面通过业务背景、解决方案、建模方法以及业务落地四个方面展开说明。

2.业务背景

动态事件是道路通行能力的变化进而影响用户出行的事件。通过动态事件的描述，可以了解动态事件包含两个要素，第一个是通行能力的变化，第二个是影响用户出行。

动态事件基本类型是封闭、施工、事故，如图 1 所示。其中封闭是道路通行能力极弱，正常车辆不能通行，特殊车辆才可能通行；封闭影响用户出行，需要用户掉头并绕路才能到达目的地，严重影响用户的出行。

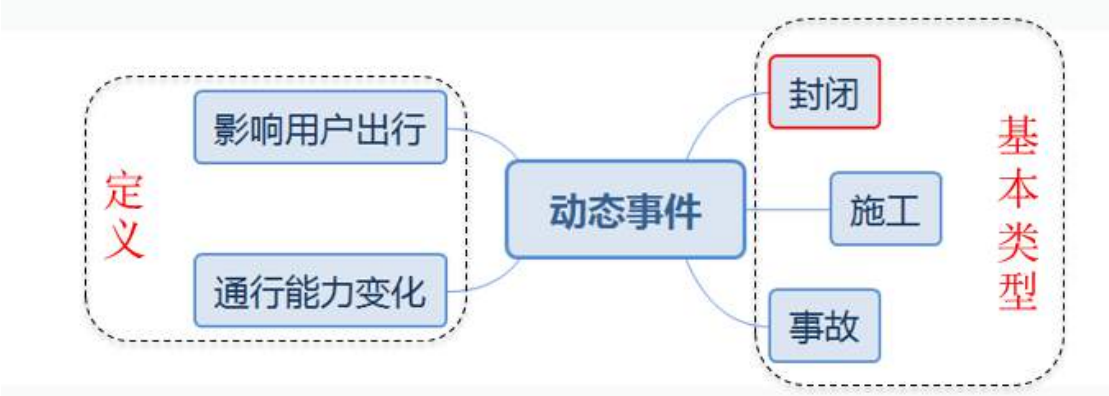


图 1 动态事件定义及基本类型

为了对动态事件有形象的理解，图 2 展示了动态事件的常见情况。第一张图展示了天气类的封路，雨雪雾等均可能引起道路封闭。第二张图展示了管制类封路，如道路要进行马拉松比赛，所以管制性封路。第三张图像展示了施工类封路，第四张图展示了施工但未封闭的情况。

图 2 动态事件示例

高德有多种发现封路事件的方法，本文主要介绍基于用户轨迹数据的动态事件挖掘算法。

图 3 中第一张图片展示了道路封闭发生后，流量从 100 左右跌到了 0；第二张图片展示了车辆的轨迹不能正常通过某一段道路，需要掉头并绕路通过；第三张展示了一条道路不能通行，道路上没有车辆的 GPS 点。热力用来描述 GPS 点的密度，GPS 点密度越高，热力越明显，颜色越深。

图 3 封闭事件的大数据线索

封路问题主要拆分为新增和消散两部分工作来展开的。新增和消散对应封闭事件的上线和下线。封闭问题之所以分为新增和消散，主要原因是新增和消散在业务分布上有着很大的差异。

新增问题是面向全路网的数据，封闭事件是小概率事件，发现封闭事件前会进行导航规划。消散面向的是线上事件，绝大部分为封闭事件，不进行导航规划。本文主要介绍封路新增问题。下面开始介绍封路挖掘的解决方案。

3.解决方案

高德在处理动态事件时，基本逻辑是利用已知数据，找出疑似封闭事件，之后再进行提纯，产出封闭事件并进行上线。按照此逻辑，产线处理过程分为三个层次：

- 数据层
- 发现层
- 验证层

大数据的解决方案也是基于此三层架构来设计的。经过系统化设计最终确定了分层化、半监督的深度学习方案，该方案可用于离线挖掘，也可以用于实时挖掘。整体方案如图 4 所

示：

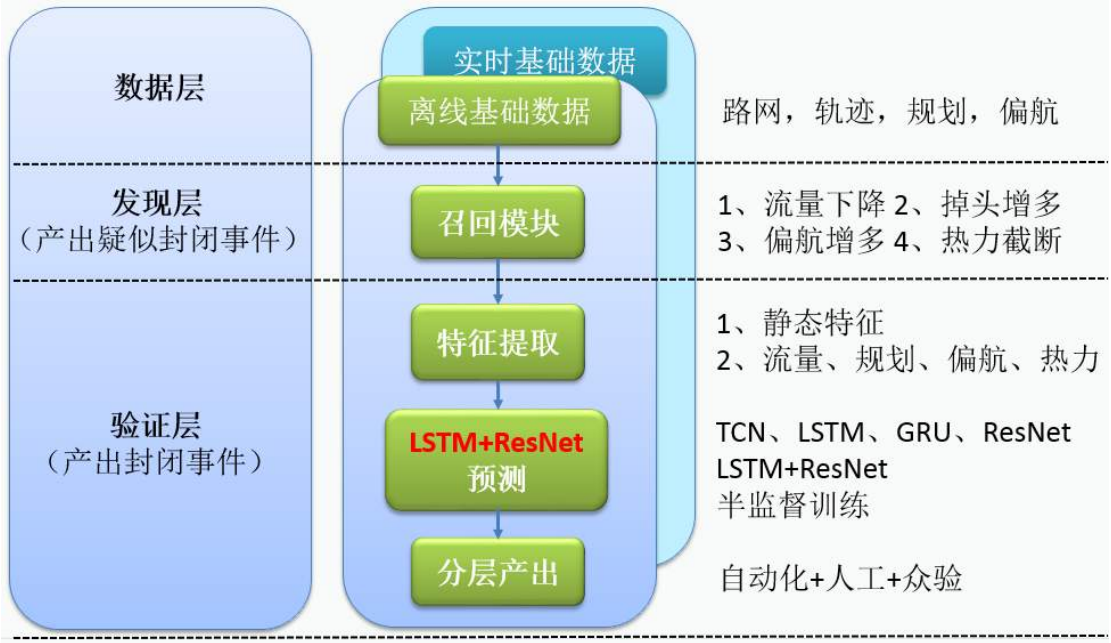


图 4 整体解决方案

本方案主要包括如下 5 个部分：

- 基础数据：基础数据主要用到了静态数据及动态数据，静态数据主要包括路网及其属性，动态数据主要是用户的轨迹、规划、偏航等。

- 召回模块：为了发现疑似封闭事件，设计了召回模块。召回模块在搜索、广告等任务中均会使用。流量下降、掉头增多、偏航增多、热力截断是典型的召回策略。

- 特征提取：业务建模过程中会将流量、规划、偏航、热力的数据在拓扑结构及时序上进行建模，产出相应的特征。

- LSTM+ResNet 预测：模型部分围绕时序模型及卷积模型进行了探索，如 TCN、LSTM、GRU 等。最终设计了 LSTMResNet 组合模型用于线上业务。

- 分层产出：模型置信度越高，封闭准确率越高。不同的置信度可以分层化产出，高置信的产出自动化上线的同时，中低置信度的产出人工协助上线，低置信度的产出能够赋能产线，大数据协同其他事件源一起挖掘封闭事件。

4.建模方法

4.1 路网建模

路网是一张有向图，每一条边，也就是路网中的一条路，被称为一条 link。路网建模分为空间建模、业务数据建模、时序建模三个步骤，如图 5 所示。将路网三步建模展开描述，

分别是：

- 空间建模：路网按拓扑结构拆分，分为上游 links、当前 link、下游 links。
- 业务数据在道路空间上的建模：基于拆分后的拓扑结构，对当前 link 及上下游 links 在规划、流量、偏航、热力几方面进行建模，形成一个 39 维的特征向量。
- 时序建模：我们的业务是典型的时序问题。以流量下降为例说明,道路封闭前，流量在 100 左右波动；道路封闭过程中，流量是逐渐下降的过程；道路封闭后，流量在 0 附近波动，基本无车辆通行。道路从非封闭到封闭的过程，是流量在时序上逐渐下降到 0 附近的过程。我们选取了四周的时间序列，每一天的数据是上一步提取的对应日期的 39 维特征向量。

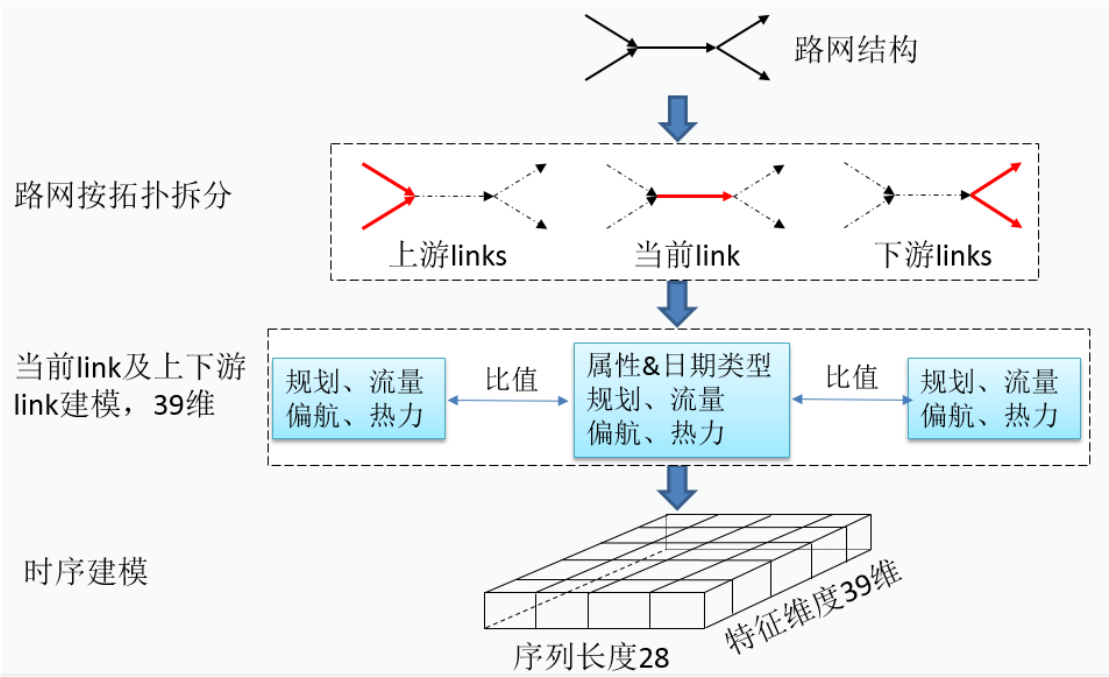


图 5 路网建模方法

4.2 算法建模

从时序建模开始，到最终选定时序和卷积的组合模型，LSTMResNet 模型，我们经历了一系列的探索：

- 鉴于我们的业务是典型的时序问题，所以从经典的时序模型 LSTM、GRU 进行实验。
- 有了经典的模型，就希望在“state of the art”的时序模型上实验，所以调研了 TCN 并进行实验。最终 TCN 实验表现优于 LSTM、GRU。
- 本着“他山之石可以攻玉”的想法，我们也实验了 CNN 经典模型 ResNet，ResNet 表现虽不如 TCN，但与 GRU 相当，优于 LSTM。重要的是 TCN 表现优秀的原因之一就是内

部运用了 ResNet Block。

- 鉴于 ResNet 表现优秀，所以有了时序+ResNet 的想法。于是我们试验了 LSTM+ResNet 的模型，称为 LSTMResNet 模型。

快、准、稳是我们选取模型的主要考虑因素。“快”指的是挖掘周期短，LSTM 比 TCN 需要的序列更短；“准”指的是挖掘的准确率高，LSTMResNet 模型的准确率最高；“稳”指的是模型潜在的恶劣 badcase 更少，越是经典常用的模型，一般认为模型潜在问题更少。

基于快、准、稳的考虑，我们选取了 LSTMResNet，并进行后续的业务迭代、落地。

LSTMResNet 网络结构如图 6 所示，输入特征向量经过 LSTM 网络层，LSTM 的输出作为 ResNet 的输入，ResNet 的输出连接全连接层，最后全连接层与只有两个节点的网络层连接，这两个节点就是二分类的置信度。输入向量是长度为 28，表示 28 天，每天特征是 39 维的特征向量；LSTM 输出向量是长度为 28，有 5 个隐层的网络层。

ResNet 是由 7 个 ResNet Block 组成。每个 ResNet Block 内部都会进行卷积、归一化、ReLU 运算，ResNet Block 运算结果与 ResNet Block 的输入向量进行相加。

LSTMResNet 模型参数整体较少，LSTM 只有 5 个隐层；ResNet 只有七个 Block，包含 14 个网络层。这是因为模型复杂的情况下，非常容易过拟合，所以模型参数配置时没有使用更多的神经元。

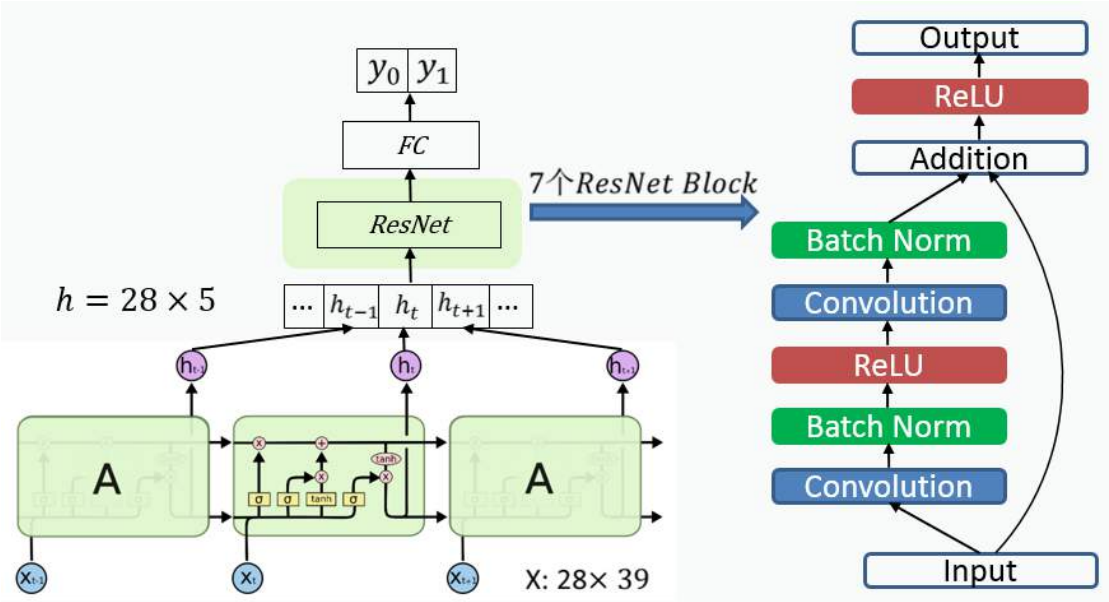


图 6 LSTMResNet 网络结构

为了克服过拟合问题，我们在 ResNet 中使用 Batch Normalization 的同时，还使用了 dropout，图 7 是 dropout 取值不同时的表现（数据来自中间实验过程）：

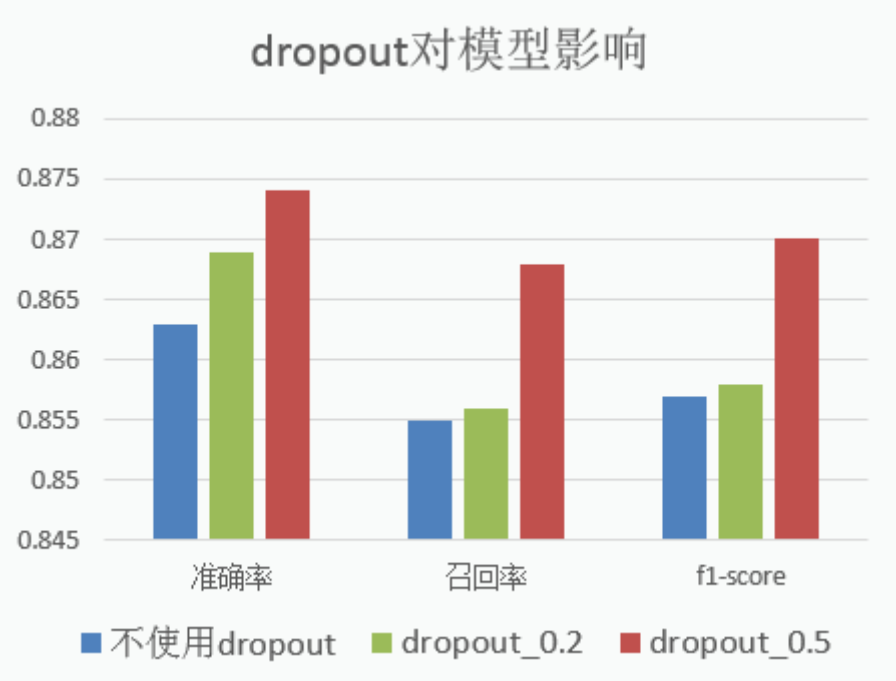


图 7 dropout 调参

5.业务落地

业务建模之后进行业务落地的工作，主要考虑两个方面：

- 模型落地方式：当前业务的主要需求是高置信的产出能够达到人工标注的准确率，这部分产出将自动化上线，要模型产出准确率不比人工标注准确率差，这是非常高的一个标准。基于高置信产出必须高准确率的要求，我们采用半监督的方法提升了高置信产出的准确率。

- 业务风险预防：为了防止模型上线后出现一些影响面较大、明显背离业务常识的 badcase，我们对模型进行了可解释性分析，分析模型的产出是否符合业务常识。

5.1 半监督助力业务落地

半监督方法是一种介于监督和非监督的方法，本文半监督实现的主要思路是：首先，用数量较少的高精样本数据学习模型，其次，用该模型对线上差分样本预测，最后，将预测的高置信部分样本作为带标签数据，重新训练模型，得到最终的模型。实验过程如图 8 所示：

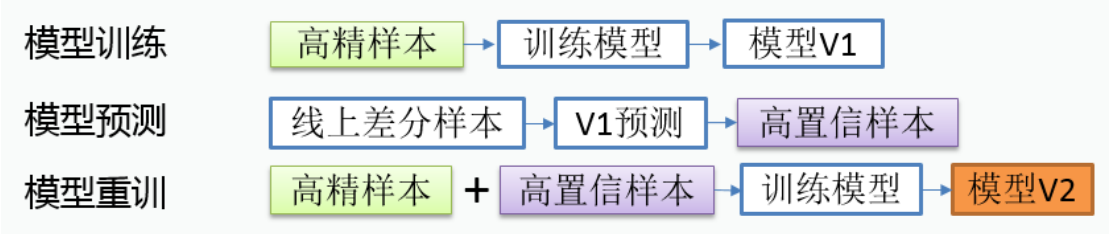


图 8 半监督实验流程

为了评测半监督训练的模型的高置信部分的准确率，分别评测模型 V1 和模型 V2 在业务数据上产出的 topN 准确率，模型 V2 比模型 V1 准确率高 10 个百分点，由此可见，半监督方法非常明显的提升了高置信样本的准确率。

5.2 业务数据验证

业务数据验证，主要是通过分析流量、规划、偏航、热力这四类主要特征是否符合业务常识，来解释模型对封闭事件的刻画是否符合业务预期。模型在流量、规划、偏航、热力上符合业务预期，则模型产出恶劣 badcase 的可能较小。

实验方法是，首先提取北京市某天的业务数据，其次使用模型进行预测，最后按置信度统计分析。业务数据验证结论如下：

- 模型置信度在流量、规划、偏航、热力截断这四方面均符合业务常识。
- 置信度能够刻画事件有无。
- 置信度越高封闭可能性越大

6.小结

本文介绍了动态事件和封闭事件的概念。为了挖掘封闭事件，我们设计了一套半监督的深度学习方案，较为详细的介绍了路网建模、TCN 及 LSTM 等深度学习建模。为了防止模型产出背离业务常识，进行了业务数据验证，实验表明模型挖出的封闭事件符合业务常识。封闭事件的挖掘能够更好帮助用户合理的规划路线、提高用户体验。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

深度学习在商户挂牌语义理解的实践

作者：毅青

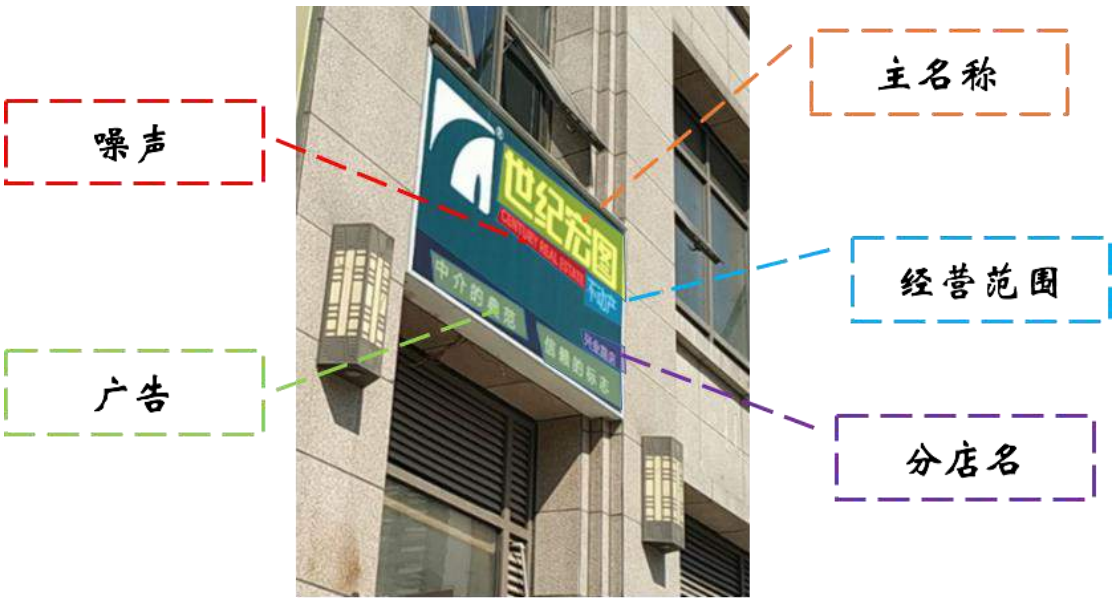
导读

高德地图拥有几千万的 POI 兴趣点，例如大厦、底商、学校等数据，而且每天不断有新的 POI 出现。为了维持 POI 数据的鲜度，高德会通过大量的数据采集来覆盖和更新。现实中 POI 名称复杂，多变，同时，名称制作工艺要求严格，通过人工来制作 POI 名称，需要花费大量的人力成本。

因此，POI 名称的自动生成就显得格外重要，而机器对商户挂牌的语义理解又是其中关键的一环。本文主要介绍相关技术方案在高德的实践和业务效果。

1.背景

现实世界中，商户的挂牌各式各样，千奇百怪，如何让机器正确的理解牌匾语义是一个难点。商户挂牌的文本种类有很多，如下图所示，我们可以看到一个商户牌匾的构成。



POI牌匾属性分类

结合 POI 的名称制作工艺，我们目前将 POI 的牌匾的文本行分为 4 大类：主名称、经营性质（包括经营范围，具体的进行项目）、分店名、噪声（包括非 POI 文字，地址，联系方式），前面 3 个类别会参与到 POI 名称制作中。如上图所示的牌匾，它输出的规范名称应该是“世纪宏图不动产（兴业路店）”。其中“世纪宏图”是主名称，“不动产”是经营范围，而“兴业路店”是分店名。

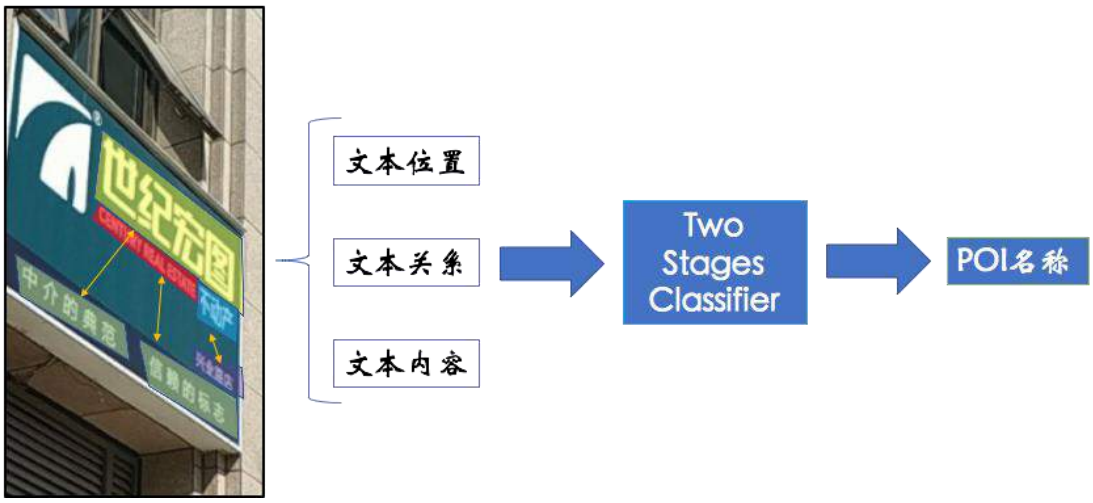
从牌匾中找出制作名称所需要的文字，不仅仅需要文本行自身的一些特征，还需要通过结合牌匾上下文，以及图像的信息进行分析。单纯的文本行识别会遇到下面的问题，如下

图，在两个牌匾中都提到了“中国电信”，但是它们的意义是不一样的，这时必须结合上下文的理解。

左边中国电信是营业范围，右边是主名称

2.技术方案

单纯从文本的语义理解的角度出发，那么这个应该是一个文本分类问题。但是直接的分类效果不佳。现实中在理解牌匾文本行语义的时候，需要结合图形，位置，内容，以及上下文关系综合来判断。为此，我们将商户挂牌理解的这个问题分解成两个子问题来解决，1.如何结合图像、文本、以及空间位置；2.如何结合上下文关系。因此，我们提出了 Two-Stages 级联模型。

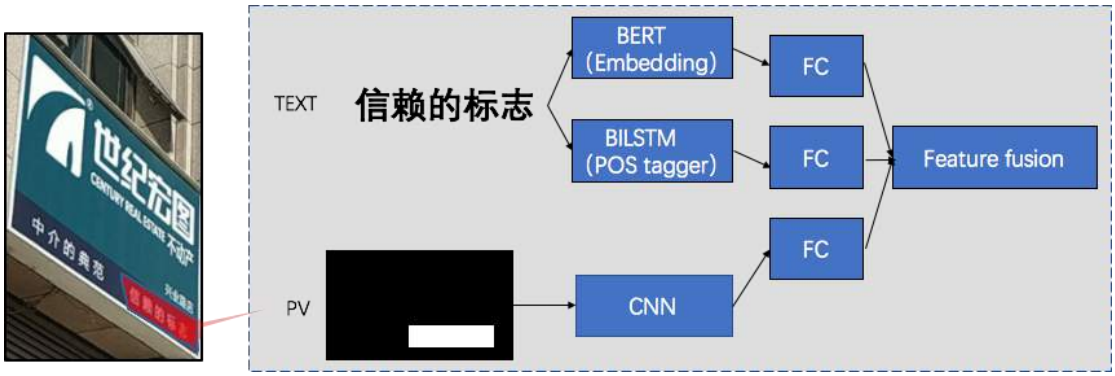


2.1 Two-Stages 级联模型

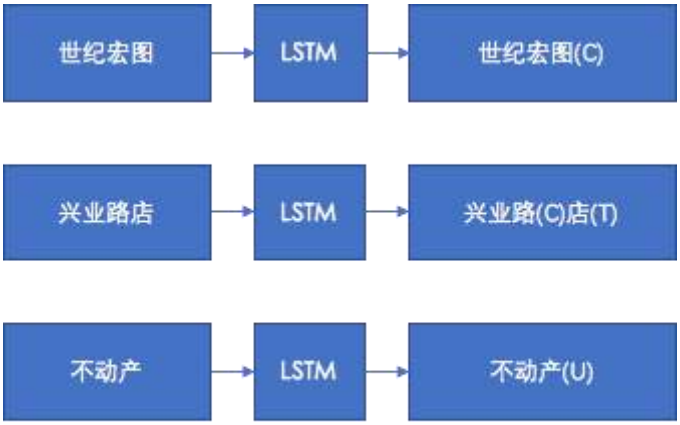
Two-stages 级联模型分为两个主要的阶段：第一阶段提取单文本信息特征，包括文本位置和文本内容等，第二阶段提取牌匾中文本行上下文关系特征，消除只用单个文本识别容易造成的歧义，准确识别出该文本属性。

2.1.1 Stage One 单文本行特征提取

单文本行特征可以分为词性结构（token level）特征和句子语义（sentence level）特征。除此之外，位置信息（PV）也是比较重要的信息，需要进行特征提取和编码。将以上特征进行融合，得到了单文本行特征。



token level 层的特征提取方面，结合名称的构成以及名称工艺，我们定义了三种词性：核心词(C)、通用词(U)、结尾词(T)。在这里我们使用 LSTM 网络来学习名称的词性序列。



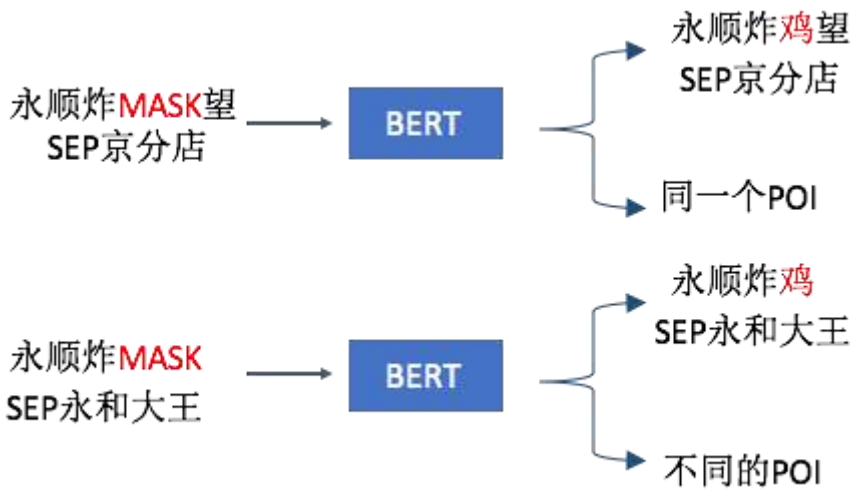
BILSTM提取token level特征用于词性标注

sentence leve 层的特征提取方面，由于我们的标注量相对比较少，采用了具有大量先验知识的 BERT 模型。同时，为了更好的符合当前业务场景的需求，我们结合业务中 POI 的数据集合，在原来 Google 官方提供的预训练模型基础上继续 pre-training，得到新的模型 BERT-POI。

预训练的 POI 文本语料没有太多的上下文环境，在构造样本时，我们将两个 POI 名称串起来或是同一个 POI 随机切分，中间都用 SEP 隔开，进行多任务学习：缺字补全和预测两个文本行是否属于同一 POI。经过实验发现，在 POI 数据上预训练模型 BERT-POI 比 Google 发布模型 BERT-Google，缺字补全和同一 POI 判定两项任务上名，正确率高 20%左右。

此外，将预训练的模型用于下游属性识别任务上，BERT-POI 与 BERT-Google 相比，提升主名称，分店名，营业范围的召回 3%~6%。

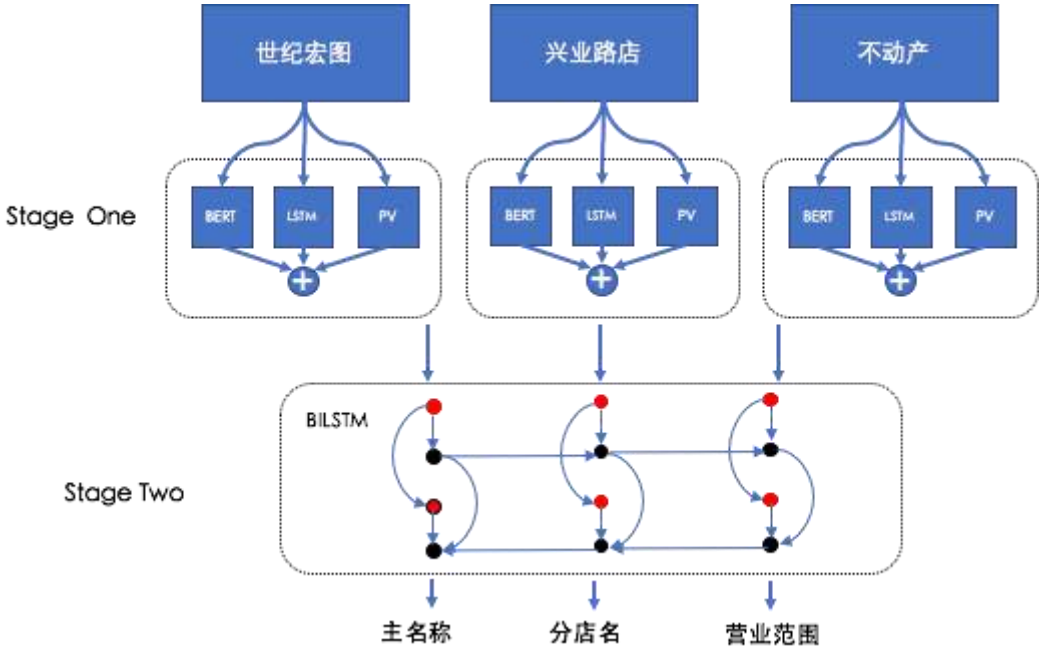
下图展示了我们预训练的过程图:



随后，我们对预训练好的 BERT-POI 在进行了 finetune，提取出 sentence leve 层的特征。

2.1.2 Stage-Two 文本相互关系提取

Stage One 提取到了单文本行的特征，那如何去实现上下文的关联，我们加入了 Stage Two 的模块，模型结构如下：



Stage Two 最主要是用 BILSTM(Bidirectional LSTM)处理 stage one 输出特征，能够将当前文本特征和牌匾内其他文本特征进行学习，消除歧义。

3.业务效果

牌匾通过语义理解后，会根据具体的输出类型来制定名称生成的策略。例如：对于单主+噪声牌匾，我们直接将主名称作为 POI 名称，而对于单主+分店名+经营性质+噪声的牌匾，我们会分析主名称的结构，看是否需要拼接经营性质。

下图展示了当前我们牌匾语义理解和名称的部分拼接策略：

图 3.1 单主+噪声场景

图 3.2 单主+分店名场景

图 3.3 单主+经营性质场景(主名称中有经营性质)

图 3.4 单主+经营性质场景(主名称中无经营性质)

4.小结

目前商户牌匾语义理解模块的准确率在 95%以上，在 POI 的名称自动生成中起到的重要的作用。商户牌匾的语义理解模块只是 POI 名称自动化的一部分内容，在 POI 名称自动化中还会涉及到噪声牌匾过滤、牌匾是否依附建筑物、敏感类别、文本的缺失、名称生成、名称纠错等模块。我们会在图文多模态这块更深入的探索，更多地应用于我们现实场景中，生产更多、更高质量的数据。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

高德在提升定位精度方面的探索和实践

作者：方兴

2019 杭州云栖大会上，高德地图技术团队向与会者分享了包括视觉与机器智能、路线规划、场景化/精细化定位时空数据应用、亿级流量架构演进等多个出行技术领域的热门话题。现场火爆，听众反响强烈。

阿里巴巴高级地图技术专家方兴在高德技术专场做了题为**《向场景化、精细化演进的定位技术》**的演讲，主要分享了高德在提升定位精度方面的探索和实践，本文根据现场内容整理而成（在不影响原意的情况下对文字略作编辑）。

以下为方兴演讲内容的简版实录：

今天要分享的主题是关于定位的场景化、精细化。高德定位，并不只是服务于高德地图本身，而是面向所有的应用开发者和手机设备厂商提供定位服务。目前已有 30 万以上的 APP 在使用高德的定位服务。

用户每天会大量使用定位服务，比如看新闻、打车、订外卖，甚至是购物，首先都是要获得位置信息，有了更精准的位置信息，才可能获得更好的服务体验。

高德地图有超过 1 亿的日活用户，但是使用定位的有好几个亿，每天的定位请求数量有一千亿次。如此大的数据量，高德定位服务可以保持毫秒级的响应速度，我们在这里面做了很多工作。此外，我们还提供全场景的定位能力，不管为手机、车机还是任何厂家，都能提供位置服务。

我今天从四个方面介绍，分别是：

- 定位面临的挑战
- 高德地图全场景定位
- 分场景提升定位精度
- 未来机遇

定位面临的挑战

大家可能都知道 GPS，GPS 在大部分情况下可以提供很好的精度，但是对于某些场景还是不够，比如驾车，GPS 给出的精度大概是 10 米，如果仅靠 GPS 定位甚至无法区分出在马路的哪一侧。

第二个场景是在室内，室内收不到 GPS 信号，这样的场景下如何实现比较准确的定位？第三个场景是如何在精度和成本之间取得平衡，因为不可能为了追求一个很好的精度去无限投入成本。只有通过海量大数据挖掘，算法和数据质量的提升，达到效果的持续优化，才能达

到最终对各种场景的全覆盖。

有很多技术可以选择，除了 GPS 定位，还有基于网络定位、Wifi 基站，原理就是通过扫描到的 Wifi 和基站列表、信号强度，进行数据库查找，找到 Wifi 位置，定位。

除此之外还有惯性导航定位，惯性导航是一种相对定位的方式，可以不断计算跟上次定位的偏移量，有了初始定位之后，根据连续计算可以获得最终的位置。

还有根据地图匹配定位，比如 GPS 的点落在一个湖里，显然是有问题的，可以通过地图匹配，找到最近的一条路，这时候精度就得到了提升。

还有一些定位方式最近几年变得很热门，例如视觉、雷达、激光，自动驾驶的概念推动了这些技术的发展，这些方式各有各的定位精度和差异性。例如视觉，在实践中往往需要大量计算和存储的开销。

很多时候，还是要基于 Wifi 的定位，获得初始定位，然后在不同场景下不断的优化，通过不同的数据源提升精度。

高德地图如何实现全场景定位

高德主要分为两个业务场景，手机和车机。在手机上主要是 GPS+网络定位。驾车的场景下，我们还会做一些根据地图的匹配，实现对特殊道路的支持。

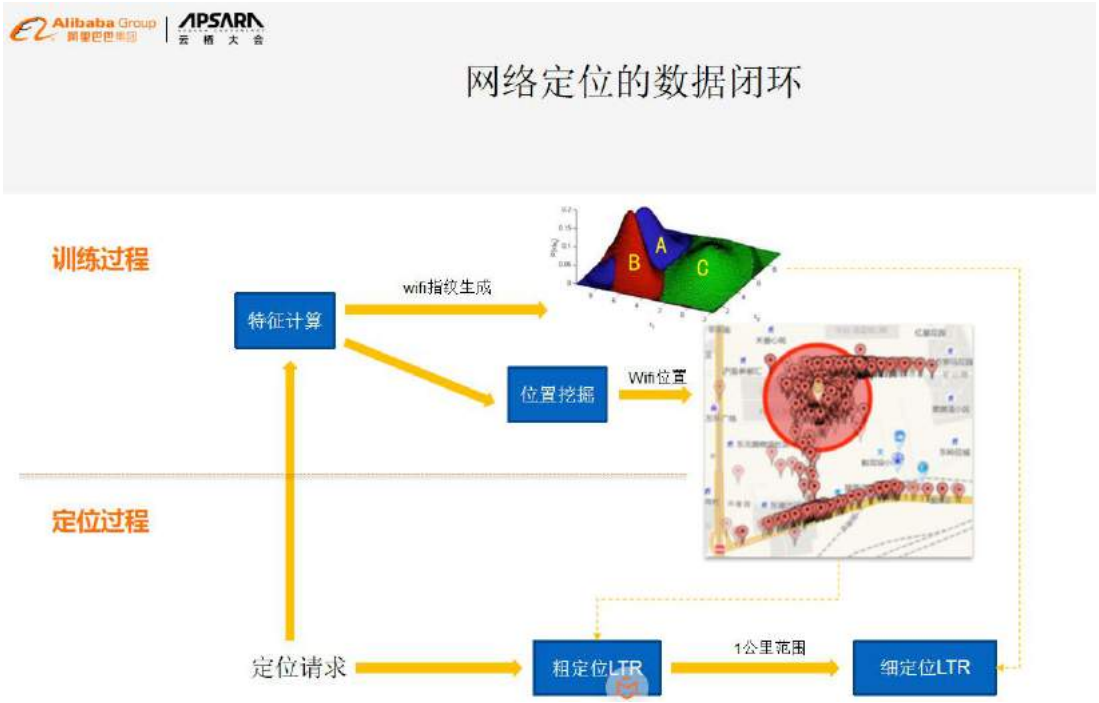
以往，很多用户会反馈说会遇到 GPS 信号不好，导致无法定位、无法导航的情形。约有 60%的情况是因为用户位于地下停车场或者在隧道里，约 30%的情况是附近有严重的遮挡，比如在高架桥下，或者在很高的高楼旁。这些都会造成对 GPS 比较严重的遮挡。

我们打电话的时候，连接的基站可能就在一公里范围内，这样短的距离传输信号还时常会出现信号中断，如果 GPS 信号距离两万多米的高度，出现问题的可能性还是存在的。所以必须通过其他方式，例如地图匹配或者惯性导航来对 GPS 进行补充。

在室内的场景，需要解决的是如何去挖掘 Wifi 基站的位置，提升精度。

在车机的场景，我们会结合更多来自于汽车的数据输入来帮助我们。

定位的基础能力



网络定位本质上是一个数据闭环，每个人在定位的时候，实际上是发送了本身的基站和 Wifi 列表，发送的数据一方面可以用来定位，另一方面也可以用做数据训练。数据训练主要产出两种数据，一个是 Wifi 基站的位置，通过数据挖掘，我们就可以获得大概的位置（初始定位），但是精度比较差。第二个是产生更详细的空间信号强度分布图。有了这个图以后，就可以进行比较精准的定位了，根据信号强度判断我距离这个基站和 Wifi 有多远，从而对精度进行改进。

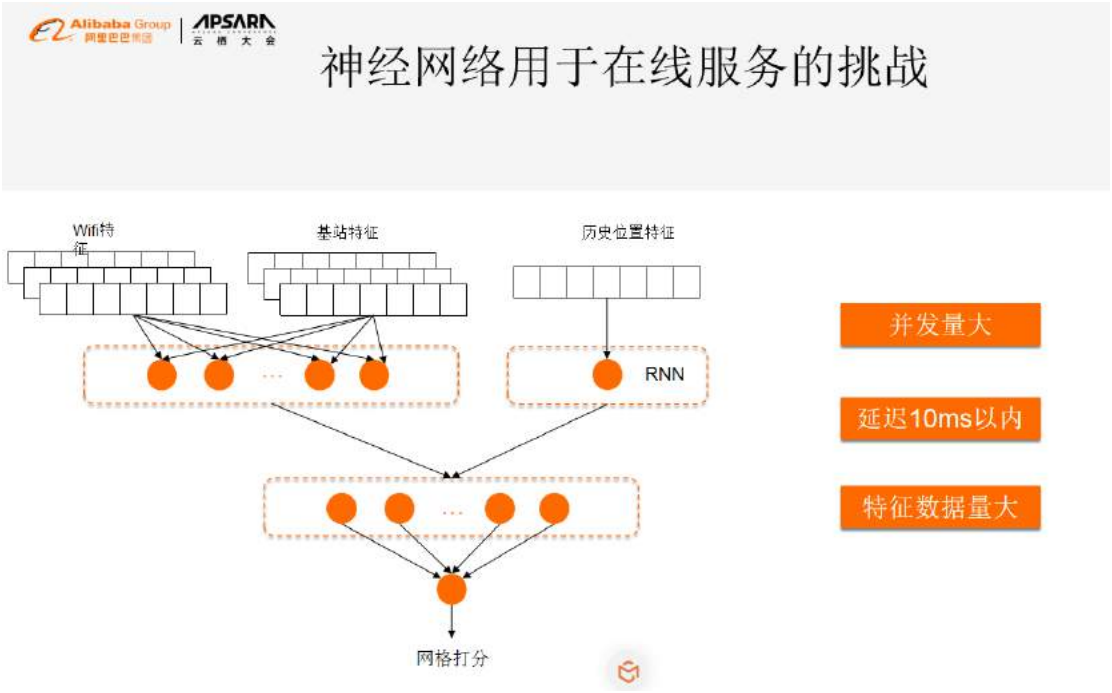
数据闭环完成以后，就是一个正向的反馈，数据越多，训练结果越多，定位结果就越准确，从而吸引更多的用户来使用（产生数据）。这就是通过数据挖掘，不断提升精度的闭环。

算法部分，我们也经过了不断的迭代。最早是基于经典的聚类模型，就是扫描基站 Wifi 列表，聚类以后选择其中一处作为我的位置，这个方法效率比较高，很快可以得到结果，但是精度很差。

第二步，我们把空间进行了精细的划分，在每个网格内统计一些基础的特征，比如历史上的点定位的数量、定位的次数、Wifi 的数量等等，计算出一个网格的打分，再对网格进行排序，最后你的定位点就是这个网格。通过这种方法，30 米精度的占比提升了 15%。

这种方法也有局限性，人工调参带来的收益是有限的，调到一定程度就没办法再提升了。所以，第三步就是把机器学习算法引入这个过程，利用监督的学习提升到最佳的模型和参数，这样可以在特定场景下获得显著提升。主要的场景就是解决大误差的 Case。

一个比较典型的问题就是，扫描到的基站 Wifi 可能只有一个基站、一个 Wifi，没有别的信息了。这个基站 Wifi 又离的特别远，无论选择基站还是 Wifi，都有 50%的概率是算错了。有监督学习，就可以把海量的配送拿出来，精细化的挖掘细微的差异，达到全局最优的效果，在某一情况下选基站，某一情况下选 Wifi。把犯错的比例降低了 50%。



上图就是我们的线上神经网络的模型，神经网络用于在线服务现在是比较流行的方式，我们在这里实际上是利用基站和 Wifi 的信号强度和混合特征作为特征输入，同时把历史位置也作为序列放进来，这个历史位置特征会放入一个 RNN 模型，预测现在的位置，使用预测的结果和基站 Wifi 列表特征，再往下预测，最后是网格的打分。最终输出一个概率最高的网格作为输出。

这个方法最大的挑战并不是在算法，而是算法效果和工程上的可实现性，如何能够达到最优。高德每天有上千亿次的调用，延时要在 10 毫秒以内。

另外，数据量很大，所有的数据，每条都有很多特征，在线的数据存储大概有几十个 TB，这个数据量也不可能放在在线服务里做，所以要做相应的优化。

我们做了三个方面的优化，第一是**分级排序**。把定位过程变成一个显微镜步骤，先做一个很粗略的定位，然后逐步收敛到很精确的位置。粗略定位的时候可以用很大的网格，用很少的特征，快速过滤掉一些不可能的位置。

然后，在很精细的网格里，用更多的特征、更多的网格进行排序。通过这种方法，就可以极大提升计算的效率，把一些不必要的计算过滤掉。

第二是**模型简化**。虽然深度学习的效果很好，但是不可能在线上用很复杂的模型，我们通过减少层数和节点数，把浮点数精度降低。

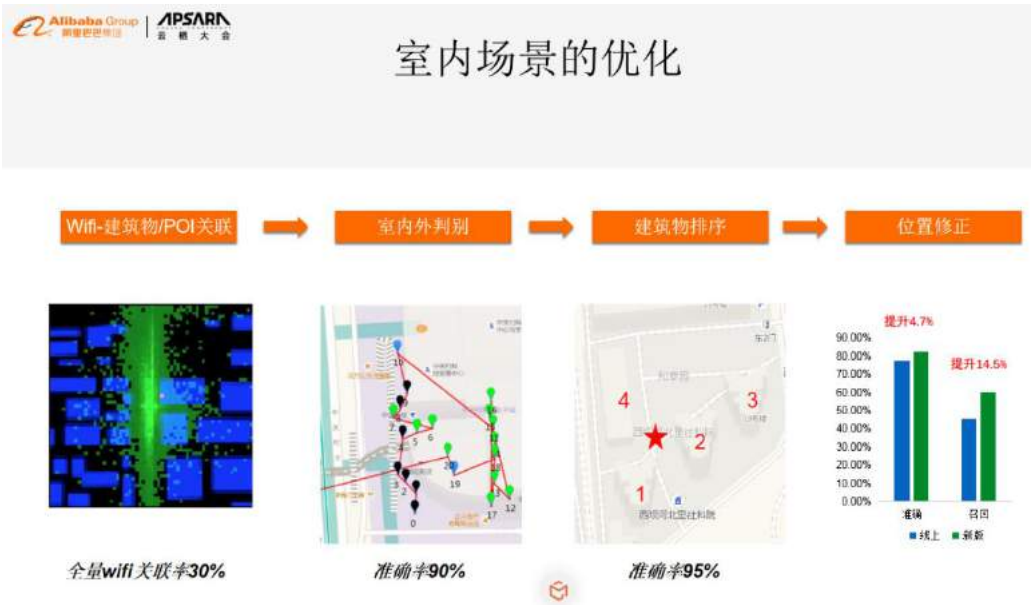
第三是**特征压缩**。这里面有特色的一点是我们根据模型进行的压缩，原始特征的输入的数量是很大的，我们增加一个编码层，输入的特征经过编码层以后，只输出两个字节的特征。我们把在线、离线的数据处理好以后，最后在线只存储两个字节。通过这种方法，在线特征的数据量降低了 10 倍，降低到 1 个 TB 以内。以上是解决的几个主要问题。

不同场景下的精度提升

在室内场景，经常会定位到室外去，这跟刚才介绍的序列流程是有关系的，因为采集过程更大概率是在室外，序列后的 Wifi 位置都在马路上，所以定位最后的概率也是在马路上，但是这对用户体验是很差的。比如打车，可能在室内叫车，定位在对面的马路上，但这条马路可能是不对的，需要找到我在哪个楼里，离哪个道路比较近。

怎么解决这个问题？一种方法是通过数据采集，就是在室内进行人工的采集，使训练数据的数据分布跟实际的预测数据分布保持一致，这种方法当然精度比较好，但是主要缺陷是成本非常高，目前也只是在热门商场和交通枢纽进行这样的数据采集，这肯定不是一个可扩展的方法。

我们的方法是想通过引入更多的数据优化定位过程。如果能基于地图数据挖掘出 Wifi 和 POI 的关系，就可以用数据关联提升精度。比如扫到一个 Wifi，名字叫 KFC，有一个可能就是你在肯德基里，这个方法比较简单。实际用的方法会更加复杂。



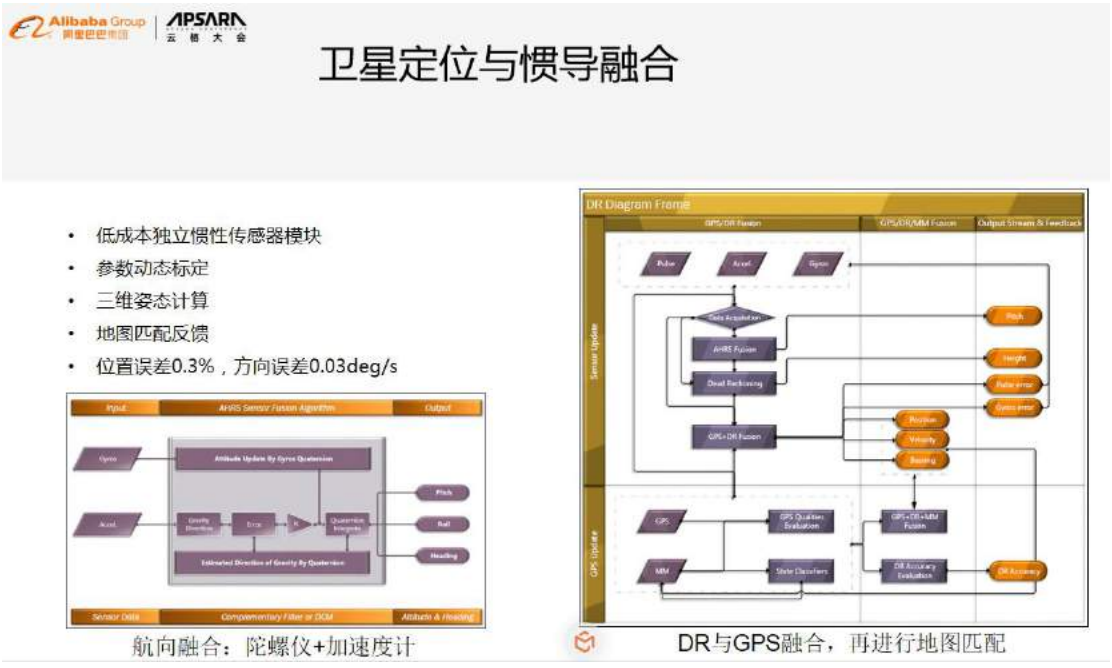
我们是利用 Wifi 信号的分布反向挖掘出位置，上图里蓝色的部分就是楼块的位置，红色的点是 Wifi 的真实位置，绿色的点是采集到 Wifi 的位置，绿色越亮，代表这个地方的信号强度越强，通过这个图放入图像学习，比如用 CNN 挖掘出它的位置以后，我们就可以建立一个 Wifi 跟楼块或者跟 POI 的关联，通过这个方法可以使全量 Wifi 的 30%都能关联上相应的 POI 或者楼块。

在线的时候需要知道用户什么时候在室内，什么时候在室外。我们用的是利用信号强度特征做区分的算法，在室内室外扫描到的 Wifi 列表和强度会有很大差别，通过这个差别可以训练出模型。绿色的点预测为室内的点，蓝色的点是室外的点。通过这种方法，定位精度提升了 15%。

驾车场景，导航过程中可能会遇到的常见问题。第一个问题是无法定位，开到停车场或者有遮挡的地方，第二个场景是点会有漂移，因为 GPS 受到建筑或者其他遮挡的时候，会产生精度下降的情况。第三种情况是无法区分主路，可能会错过路口。

对于以上问题，我们采用的是“软+硬”融合定位，软的部分包括两部分，一个是基于移动定位，第二个是根据地图匹配。经过两个“软+硬”结合之后，我们在 GPS 10 米精度做到 90%以上，可以实现高架主路和停车场的持续导航。

这里面关键的就是如何实现融合定位，比较有特色的一点就是我们做车机的传感器模块是低成本的，成本不到 100 元，其他类似产品成本是比较高的，可能需要几千块钱。使用低成本的器件，能够更容易得到普及。缺点是精度比较差，定位准确性差一些。要通过软件的方法弥补硬件上的缺点。

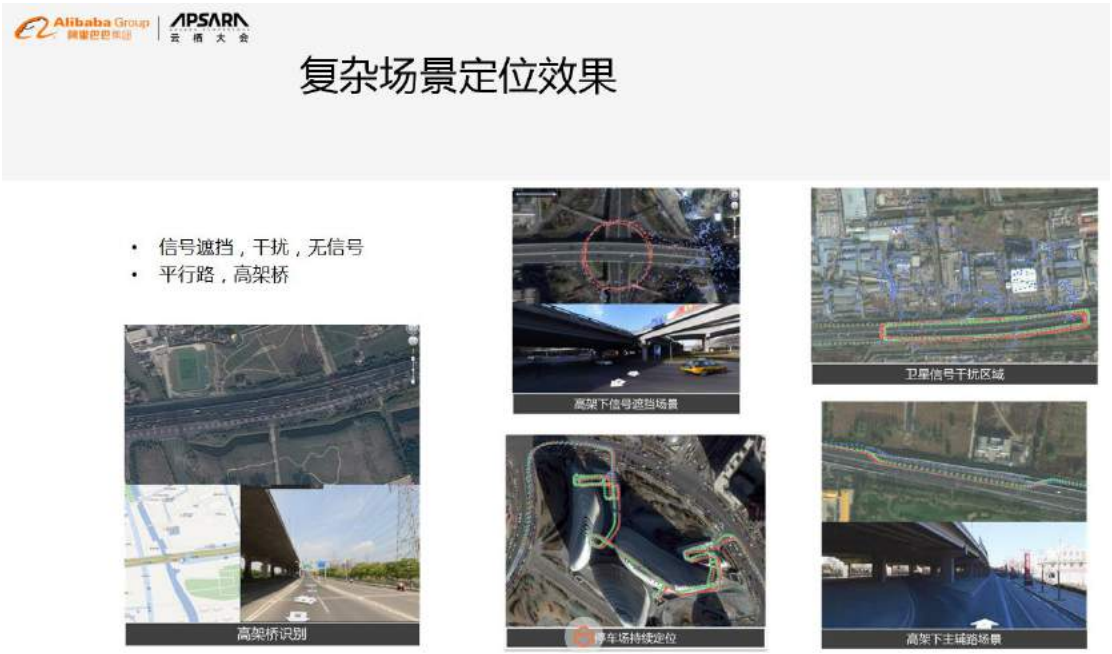


我们的解决办法分成三个步骤，首先是航向融合。陀螺仪有相对的角度可以算出来，加速器可以算出地球引力的方向，这两个结合以后就可以建立一个滤波方程，把真实的方向持续不断的输出。第二，把三维的方向和 GPS 的结果进行一次融合，就可以计算出修正后的位置。第三步，再和地图匹配做对比，因为我们知道它的方向、位置以后，就知道它是在上坡还是下坡，是在高架上还是高架下。还有一点，匹配后的位置跟 GPS 原始位置做对比，如果差

别很大，GPS 可能发生了偏移，我们就把 GPS 舍弃掉，只用惯性导航推算。

这里面有三个特点，第一，**参数动态标定**，不需要对器件有初始的标的，我们通过三维的计算出方向，用**地图匹配反馈**。关于地图匹配的部分，核心是我们利用 HMM 的算法进行位置的匹配，推算每个点的道路。这里面比较关键的概率，一个是发射概率，一个是位置转移概率。

第二，我们把角度也考虑进来，角度的变化同样用于决策转移概率，这里面跟位置转移概率的区别就是引入了速度做变量，不同的速度下，发生转角的概率是不一样的，速度慢了可能会转向，速度快也可能转向，所以我们针对每个速率都有一个曲线。



上图是定位效果，红色的点是实际修正后的轨迹，蓝色的点是原始的 GPS 点，下面是在高架下的效果，可以看到高架下 GPS 点已经非常发散了，飘的到处都是，但是修正之后跟绿色的点是重叠的。下面的图是在停车场里，在停车场进去的时候，蓝色的点就已经消失了，但是红色的点可以很好的还原出用户在停车场里持续的轨迹。

高精定位方面，高德主要建立两种定位能力，一种是基于图像定位，一种是基于融合定位。图像定位是只用图像就可以形成比较好的分米级精度，融合定位主要是引入了两个新的定位技术，一个是 VSLAM，一个是差分 GPS。这两个方法分别应用于有 GPS 和没有 GPS 的情况，可以提供很好的精度。VSLAM 可以做到误差很小，因为可以有图像的方法进行修正。

自动驾驶是一个方向，并且需要从辅助驾驶过渡到自动驾驶，但系统性变化到来之前会有阶段性的变化，就是服务于人的导航服务的精细化，即车道级导航。车道级导航需要高精地图，至少是分米级的精度。

对未来定位技术发展的理解。基础能力部分，我们认为 5G 的出现会为定位提供一种新的可能性，因为 5G 的频率比 4G 更高，波长会更短。它可以测距，以前基于基站和 Wifi 的定位都是基于信号强度的。但是 5G 支持了测距以后，它就可以提供一个很好的精度，所以可能会出现一种方式，基于 5G 的定位可以达到类似 GPS 的效果。

第二是融合定位，随着各种新的数据源不断出现，用新的算法去发挥不同数据源的特点，从而达到整体效果的提升。驾车部分，视觉定位和差分 GPS 技术的逐渐普及。室内部分，有超宽带的定位，除此之外还有蓝牙和 Wifi 的精准定位。在最新的技术标准里，也都支持了测距和测角的技术，也就是未来新的蓝牙或者 Wifi 的 APP，可能就能提供一部分的定位能力。

所以，未来 10 年内，我们可能会看到这几种方式相互融合，精度会得到质的改变。以上就是我介绍的内容，谢谢大家！

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

高德网络定位算法的演进

作者：方兴

1.导读

GPS 定位精度高，且早已成为移动设备标配，但 GPS 也具有一些难以克服的缺陷，包括：

- 冷启动时间长。GPS 启动时，需要进行搜星，锁定卫星信号，然后再进行位置技术，这个过程可能会达到几十秒，即使采用诸如 AGPS 等技术，仍然有秒级的时间无法定位。
- 室内或有遮挡的场景。GPS 信号弱，无法有效定位。

用户需要持续的有效定位，因此需要另一个技术对 GPS 进行补充，这就是网络定位技术。

网络定位是将手机设备收到的信号（主要是基站、Wifi、蓝牙）发送到网络服务器，获得位置。之所以要将信号数据发送到网络上，是因为网络定位是利用信号指纹进行定位，需要一个庞大的且持续更新的指纹数据库，这个数据库难以同步到移动设备上。为了进行定位，需要事先建立每个位置的指纹特征，然后在定位时用实时指纹比对每个位置的历史指纹，确定位置。

高德网络定位不仅承担着高德地图用户的定位请求，还面向国内所有主流手机厂商，以及国内 30 万以上 App 提供服务，日均处理请求千亿次，峰值 QPS 百万级。

在过去的几年中，高德网络定位算法经历了从无监督算法向有监督算法的演进，从定位精度、定位能力透出等方面都有了显著的提升。

注：高德网络定位只存在于安卓平台上，在 iOS 上由于苹果公司未开放任何定位相关的指纹数据（Wifi、基站列表等），定位结果全部来自于 iOS 自身。

2.基于聚类的无监督算法

经典的指纹定位算法是无监督算法，其核心是计算指纹的相似性，用指纹确定位置。下图是一个例子，AP 代表手机扫描到的基站和 Wifi 设备编号，纵轴代表不同的位置，二者交点的数值代表该位置扫描到该 AP 的信号强度，为空代表该位置没有扫描到该 AP。

	AP1	AP2	AP3	AP4	AP5	AP6	
Loc1	-50	-40		-80			
Loc2	-30		-80	-70		-40	
Loc3	-100	-30			-60	-80	
Loc4			-60		-30		
Loc5		-90					

要对一个新定期请求进行定位（比如 AP1:-30,AP2:-50,AP3:-90），一个最简单的方法，是用 KNN 逐一计算该指纹与历史指纹的相似度（比如用 L2 距离或者余弦相似度），取相似度最大的历史位置作为用户位置。

这有两个问题，第一是计算量太大（AP 是 10 亿量级，loc 是千亿量级），无法满足实时定位的要求，第二是历史指纹在局部可能比较稀疏，对于用户指纹无法精确匹配。

于是需要对历史数据进行预处理，提取出 AP 和网格的通用指纹，这样在定位时只需要比对一次即可。下图是利用一个 AP 的历史采集位置进行聚类，获得 AP 实际位置和覆盖半径的过程，有了每个 AP 的位置，在定位时将多个 AP 的位置进行加权平均即可获得最终位置。

这种方法需要解决的一个挑战是当有多个候选位置时如何选择，如下图，有两个候选位置。

此时需要设计一个策略进行簇选择，基于每个簇的特征进行打分，找出最有可能的一个簇作为用户位置。

基于加权平均的定位，速度很快，但精度比较差，原因是指纹在空间上的分布并不是连续的，而可能受到建筑、地形、道路的影响，呈现一种不规则的分布，于是在上面定位方式的基础上，发展出一种基于格子排序的算法，可以更精准的定位。

首先将地球划分为 25*25 的网格，然后统计每个网格内的指纹特征，最后进行格子排序。设候选网格为 l，信号向量是 S，则定位过程就是计算

$$\text{argmax}_l[P(l|S=S_0)]$$

根据贝叶斯公式，有

$$\text{argmax}_l[P(l|S=S_0)] = \text{argmax}_l[P(S=S_0|l)*P(l)/P(S=S_0)] \qquad 1\text{-}1$$

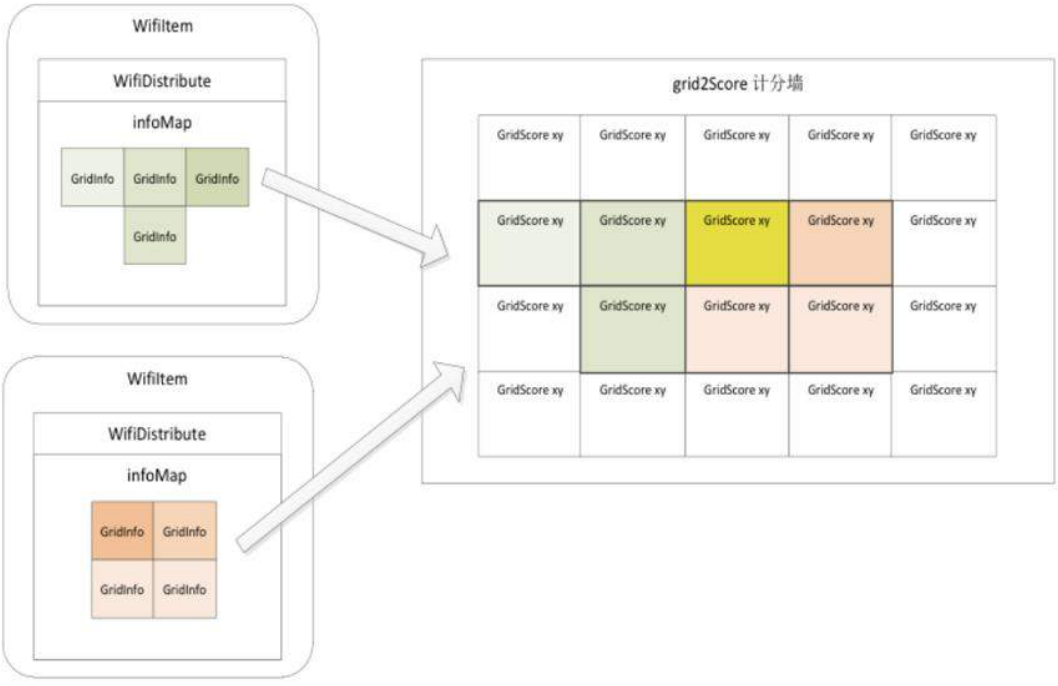
根据 1-1，由于所有候选网格的分母相同，只需要计算分子，即：

$$\text{argmax}_l[P(S=S_0|l)*P(l)] \qquad 1\text{-}2$$

其中 P(l)是某个位置在全量用户位置中出现的概率，可以用定位 PV 表示，而 P(S=S0|l)则需要计算在每个网格内出现某种信号向量的概率，由于向量维数高，概率难以计算，因此对不同维进行独立假设，认为每个信号出现的概率是独立的。有：

$$P(S=S_0|l) = \prod P(s=s_0|l) \qquad 1\text{-}3$$

这样，可以基于历史指纹对每个网格内的每个 AP 的信号强度进行直方图统计，即可计算出概率，最后对所有格子的概率进行排序，获得概率最高的那一个，如下图：

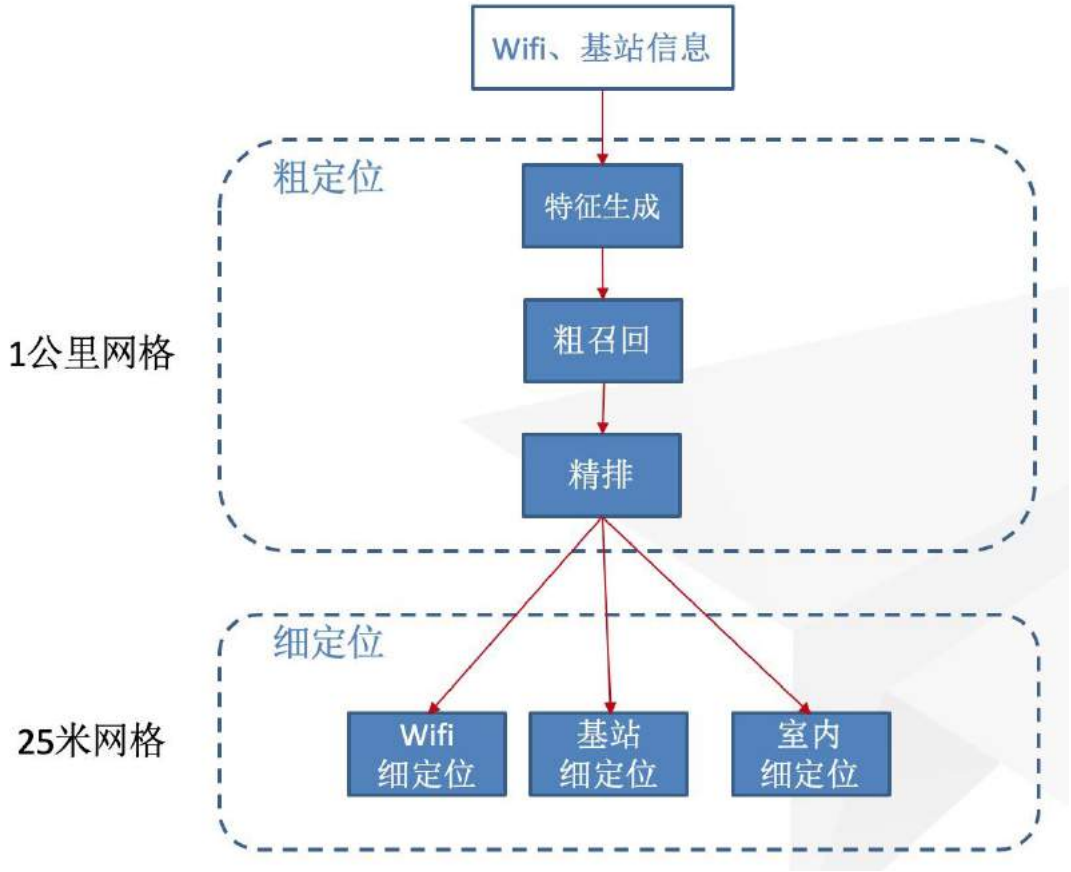


3.基于分层排序的有监督算法

无监督算法的一个问题，是难以迭代，对于 badcase 无法进行有效优化，一旦调整策略就会影响到其他 case，无法获得全局最优。

因此，有监督学习就变得很有必要，高德定位从近两年开始全面转向有监督学习，持续进行特征和模型设计，提升效果，取得了不错的收益，解决了 50%以上的大误差问题（5 公里以上），在移动 Wifi 识别上获得了 99%以上的识别准确率。

有监督学习需要使用大量的特征，特征的计算需要消耗较多资源，考虑到定位服务要承受 10 万以上的 QPS，模型的复杂性与效果同等重要，因此我们首先将定位服务进行了分层，上面的层级针对大网格，计算粗略的位置，下面的层级针对小网格，逐步细化位置。这样可以极大减少不必要的计算，在性能和效果间取得平衡。



对于每一个单独的算法模块，都采用类似下面的神经网络模型对每个候选网格进行打分，再使用 LTR 损失函数作为目标进行训练，从而获得神经网络的参数。在特征方面，同时考虑以下三类：

- AP 的动态特征，比如信号强度。
- 网格特征，比如 PV、UV、AP 数、周边候选网格数等。
- AP 在网格上的特征，比如信号强度分布、PV、UV 等。

采用这种方法可以解决绝大部分格子选择不准确的问题，遗留的一个问题是当定位依据特别少的时候，比如只有一个基站和一个 Wifi，二者分别位于距离较远的两个网格，此时无论选择哪个都有 50%的错误概率。为了解决这个问题，我们引入了用户历史定位点辅助进行各自

选择。

在特征部分加入历史定位点序列，输出一个历史位置特征（可以看成是一个预测的新位置），让这个预测位置参与网格打分。当有两个距离较远但打分接近的网格进行对比时，通过预测位置进行加权。这样模型应该可以学出这样的规律：如果网格距离预测位置比较远，打分就降低，如果比较近，分就高。通过这个方法，大误差 case 的比例可以降低 20%。

4.场景化定位

用户在不同场景下对定位的要求是不同的，比如用户在旅途中可能只需要知道大致的位置，不需要很精确，但是在导航时就需要精确的知道自己在哪条道路上，距离出口多远。

因此，除了在整体算法架构上进行优化，高德还在不同特定场景上进行针对性的优化，满足用户不同场景下的定位需求。

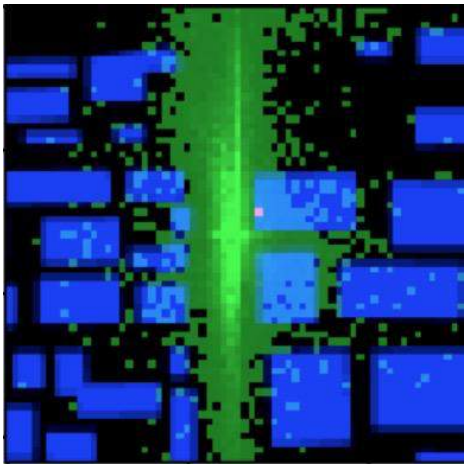
- 室内场景

指纹定位的一个局限，是需要采集带 GPS 的样本作为真值进行训练，由于 GPS 只能在室外被采集到，即使用户在室内，其定位结果有很大概率在室外，这会对用户造成不少困扰，特别是在用户准备出行的时候，其定位点的漂移会导致起点偏离真实位置较大。

为了解决这个问题，有两个解决办法，一是采集室内真值，但这种方法需要大量人工采集工作，工作量巨大，目前高德在一些热门商场和交通枢纽进行人工指纹采集（除了基站 Wifi 还支持蓝牙、传感器定位）。第二个办法是借助大数据，无需人工干预，对 Wifi 进行建筑/POI 关联，用建筑/POI 位置去修正定位结果。

Wifi-POI 关联有多种方法，一个简单的方法是用 POI 名字与 Wifi 名字的相似度判断是否有关联，比如麦当劳的 Wifi 名字就是 McDonald，关联的时候需要考虑中英文、大小写、中英文缩写等。从名称能分析出关联关系的 Wifi 毕竟是少数。另外一种覆盖能力更强的方法是利用 Wifi 信号分布规律去挖掘 Wifi 的真实位置，毕竟绝大部分 Wifi 都是部署在室内的。

这里我们采用的是 CNN 的方法，将楼块数据、POI 数据、采集真值数据绘制为二维图像，然后进行多层卷积计算，label 为 Wifi 所在的真实楼块区域。下图中蓝色块为楼块，绿色为采集点，颜色越亮代表信号强度越高，红色点代表 Wifi 真实位置。



目前算法能挖掘出 30%Wifi 对应的真实位置，在最终定位效果上，用户在室内时，能正确定位到室内的样本比例提升了 15%

- 高铁场景

从用户报错情况看，有大量报错是用户乘坐高铁时定位异常。高铁在近两年开通了车载 Wifi，这些 Wifi 都是移动 Wifi，因此这些 AP 是没有一个固定位置的，如果不进行任何处理，算法训练获得的 Wifi 位置一定是错误的，很大概率会在沿途的某个车站（用户集中，采集量高）。

针对这种场景，需要将移动 Wifi 全部去除再进行定位。我们开发了针对高铁和普通场景的移动 Wifi 挖掘算法，利用采集点时空分布等特征判断某个 Wifi 是否移动，挖掘准确率和召回率均超过 99%，可以解决绝大部分高铁定位错误的问题。

- 地铁场景

地铁场景有点类似高铁，用户扫到的 Wifi 基本都是移动 Wifi（少量车站有固定 Wifi），因此只能借助基站进行定位。但基站深埋地下，缺乏采集数据，如何获得基站的真实位置呢？我们采用了两种策略，第一个策略是利用相邻基站信息，当用户在一个请求里或者在短暂时间段内同时扫描到地铁基站（无 GPS 采集）和非地铁基站（有 GPS 采集）时，我们可以用后者的位置去推算前者位置，当然这种方式得到的基站位置不太准确。于是我们进行了进一步优化，利用用户轨迹去精准挖掘出每个请求对应的地铁站，从而构建出指纹对应的真值。

基于以上方法，地铁内的定位精度可达到 90%以上，实现地铁报站和换乘提醒。

5.未来演进

在未来，定位技术特别是移动设备的定位技术还将快速发展，主要突破可能来自以下方面：

图像定位：谷歌已经发布了基于街景的 AR 定位，可以解决在城市峡谷区域内的精准定位。这种定位利用了更丰富的数据源，对用户体验的提升也会非常显著。

5G 定位：5G 相比 4G，频率更高，频带更宽，用于测距时精度更高（比如利用相位差进行传输时间计算），行业协会也在孵化 5G 定位相关的标准，运营商在未来可能会支持基于 5G 网络的定位，届时在 5G 覆盖区将会有类似 GPS 精度的定位效果。

IOT 定位：随着物联网的普及，基于 NB-IOT 的定位技术也会应运而生，它可以使用类似基站定位的方法，或者使用 P2P 定位的方法为物联网设备进行定位。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

视觉智能在高德地图的应用

作者：任小枫

2019 杭州云栖大会上，高德地图技术团队向与会者分享了包括视觉与机器智能、路线规划、场景化/精细化定位、时空数据应用、亿级流量架构演进等多个出行技术领域的热门话题。现场火爆，听众反响强烈。

高德地图首席科学家任小枫在高德技术专场分享了题为**《视觉智能连接真实世界》**的演讲，本文根据现场内容整理而成（在不影响原意的情况下对文字略作编辑）。

以下为演讲内容的简版实录：

我今天主要给大家介绍视觉及相关技术如何在高德落地，如何帮助连接真实世界。连接真实世界这句话并不只是我个人的想法，而是高德地图的使命，我们的使命是“**连接真实世界，让出行更美好**”。

首先，简单介绍下高德地图，有超过 1 亿的日活用户，超过 4 亿的月活用户，高德地图不光提供导航，也提供出行相关的其他服务，涵盖了信息服务、驾车导航、共享出行、智慧公交、智慧景区、骑行、步行、长途出行等应用场景。

高德地图做的事情是建立人和真实世界的关系，人要跟真实世界建立联系，地图是基础，地图之上还有更多的信息可以获取。

视觉是连接真实世界的桥梁

视觉是连接真实世界的桥梁。为什么？从人的信息获取角度来看，80%的内容是通过视觉获取到的。从人的信息处理来看，人的大脑 30%-60%用于视觉感知。从机器的角度，视觉是非常重要的通用感知手段。

人类感知真实世界的方法，还有很多其他方式，例如传感器、LT...但是，作为通用的手段，我一直觉得视觉是第一选择，通用，信息量非常大，可以远距感知，也可以做到实时。

还有一个原因，人类真实世界里（各种元素）80%以上是为了视觉而设计。有的时候，我们对真实世界太过于熟悉，可能不会太在意。但是看一下周围的标志和信息，包括认识的事物，都是根据视觉设计和获取。

因为人类获取信息的主要方式是通过视觉，所以真实世界的设计也是基于视觉。大家可以想象下，如果获取信息的主要方式是通过嗅觉，那这个世界会非常不一样。基于这些，回到我们在做的事情，大家一定不会奇怪，地图信息的获取和建立，绝大部分也是来自于视觉。

视觉技术@高德地图-地图制作

视觉技术在高德地图的应用有很多不同的方式，如下图所示：

左边是地图制作，有常规地图和高精地图，高精地图对应于未来的无人驾驶。右边是跟导航体验相关的，我们在做的一些跟定位相关的工作，也在利用视觉技术希望使导航变得更加便利。因为时间关系，今天只给大家介绍常规地图和导航相关的部分。

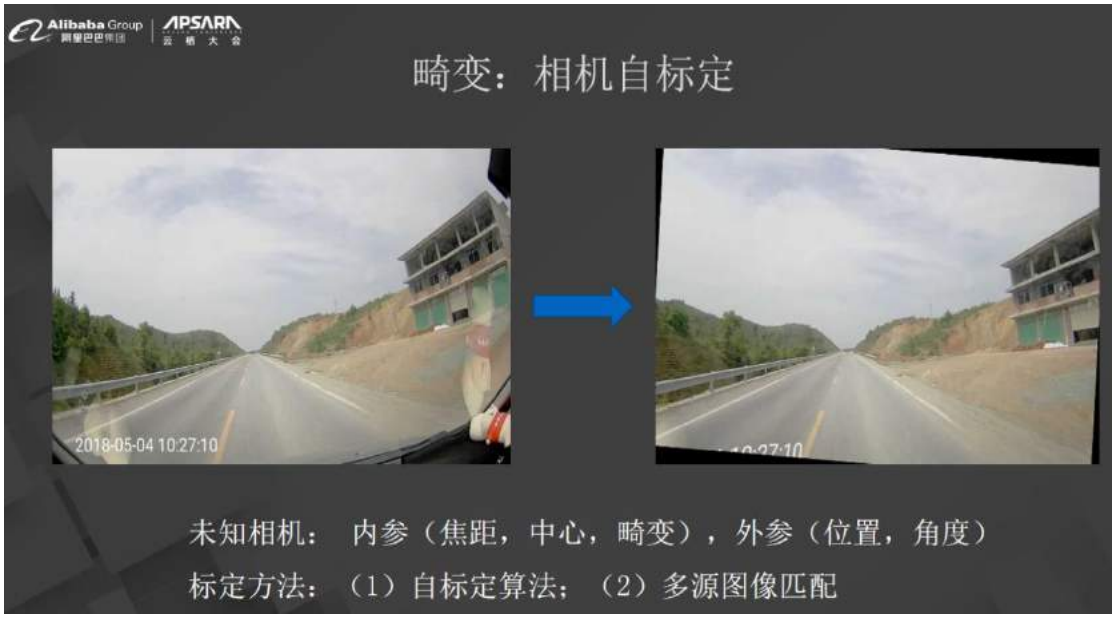
地图服务从哪里来，首先要采集资料，目前绝大部分是通过相机和视觉的方式采集信息。真实世界很大，全国有几百万公里道路，再加上其他信息，人工方式目前是处理不过来的，很大程度上需要用自动识别，通过算法识别资料。当然有时候算法没办法做到 100%，还需要人工修正，从而制作成地图数据库，来支持地图数据服务。

地图制作任务，常规地图任务通常分为两大类，一类是道路相关，一类是 POI 挂牌识别。这两类任务都需要较多的视觉技术。例如，在道路标志识别上，算法要做的就是把道路上的标志一个一个全部找出来，同时识别标志的类型和内容。

道路标志有 100 多种。如果只是处理这些标志，其实并不是那么复杂。现实中，有时候需要用**低成本**的方式采集数据，这时如何保证图像质量就是需要考虑和解决的问题。

采集信息的时候，有时候图片会有畸变、反光、遮挡等情况，先不说分辨率压缩的问题，成像本身取决于镜头的质量和成本、天气条件、光线等因素，有时候采集回来的图像中差的图很多。这时候就不只是单纯去解决一个理想当中的算法问题，也需要处理很多实际情况。

给大家举几个例子，下面左边的图是实际采集的图像，会有各种各样的问题。大家对相机有些了解的话，知道相机有内参和外参，内参是焦距、中心、畸变。外参是位置、角度，这些都会影响成像效果。



对于识别问题来说，这些相机参数不会造成太大问题，但是如果需要做一些跟几何、位置相关的计算，这时候相机畸变和内外参不准就会造成很大的问题。我们通过把多源数据放在一起做匹配，基本可以解决这个问题。右边是一个实际例子，相机的畸变纠正角度，有一些斜

的被纠正过来了，很大的提高了后面的算法处理。

另一个例子，图像质量。有的图质量比较差，但是没办法丢掉，还是有有用的信息。有的原始图像，放大之后非常模糊。如果这时采用图像增强的方法，可以把这张图变得更清楚。改善原始数据的质量，有很多可用的方法。比如提高识别算法精度，提高人工效率，也可以用它做模糊的检测，对比一下增强前后，可以知道哪些是模糊，哪些是不模糊。

刚才举的只是交通标志的例子。还有一个有趣的问题，就是感知电子眼。电子眼很小，而**小目标的检测**是一个有挑战的问题，在研究领域大家也比较关注。大家可以感受下，拿一张图，如果是太小的东西，放大之后就看不清了，还不如远景。那怎么能比较精确的找到这么小的电子眼呢？

通常方式就是放大区域，因为这个东西太小了，光找这个目标比较难，找到区域放大，引入周边的信息。这些信息可以帮助更好的找到这个小目标，放的再大一点，才能看到其他相关的信息来帮助电子眼的智能检测。

但是放的太大也会有问题，放的太大会引入很多无关的信息。从技术上来说有一些解决方法，现在视觉技术上用的比较多的有一个注意力机制，画一个大框，机器自己会学哪块重要哪块不重要，帮助更好的聚焦到目标本身。当然，尽量会用一些先验信息，比如本身的分布、高度、大小。

光检测还不够，很多时候真实世界在变。很多时候要分辨出哪些变了哪些没变。以前检测出一个电子眼，新的资料又检测出一个电子眼，需要知道这两个是否是同一个。

如何判断？因为这张图表达的不一样，如果仔细看，确实可以看到背景的建筑、架设类型都差不多。需要用算法来判断到底是不是，这就牵涉到目标检测、车道归属、架设类型分析，还要做场景匹配。通过这些，很大程度上可以判断这是一个什么场景，从而判断两张图的元素是不是同一个。

刚才说的是道路，下面是几个跟 POI 相关的例子。POI 的牌子，可以分成好多不同类型，有牌坊式、挂牌式、门脸式等。不仅 POI 各种各样，非 POI 其实也各种各样。如果只是检测文

字的话，你会发现真实世界里的很多不是 POI，有的只是标牌、标语、广告、对联、交通标志等。所以，要区分出 POI 和非 POI。

有很多其他的复杂场景，这里不一一举例了，有些可能平时也不太能想到，比如三维挂牌，它不是一个平的牌子，在街角，可能是一个水果超市，沿着街角弯曲过来。这类牌子很难在一张图里完全检测出来，即使检测出来，一不小心就会分成两块牌子，所以真实世界的复杂性还是会造成更多的问题。

面对这么多复杂性，需要去分析具体场景的情况。很多时候最后的结果往往不是一个算法就能解决所有的问题，需要各种算法的融合。比方说，如果是文字，需要做检测，文字本身也需要做检测和识别。位置的话，需要做一些三维方面的推断。很多时候资料获取到以后也有模糊和遮挡的部分，也要做判断。

每一个判断不是单一办法就可以解决，不是光靠一个模型就能够做到最好的效果，需要的是两个甚至更多的模型，从不同的角度去解决问题，才能够达到更好的效果，这是在数据积累的基础之上。

上面列举的一些问题有一定的复杂性，跟所有的问题一样，越做到后面越难，我们现在还在做，这些算法很大程度上决定了地图制作的效率和触达到用户的地图质量，这些是非常重要的核心问题。

POI 也不光是以上介绍的只需要判断是不是 POI 或者文字识别，很多时候还需要做版面的内容理解。如果一个牌子，需要知道这个牌子上的信息，有时候会有主名称，有时候会有分店，有时候没有，有没有联系方式、营业范围，这些都需要用算法去做。

视觉技术@高德地图-导航

以上介绍的是在地图制作方面有很多的复杂性，需要用视觉算法或者其他算法来处理。接下来分享下在导航方面的。

先说下自己的一个体会。前段时间在西班牙休假，欧洲的环岛特别多，谷歌（地图）导航经常提示我，进了弯道之后从第三个出口出去，我当时特别郁闷，因为要数口子，经常你也不知道那个到底算不算出口，所以走错了好几次。我在国内没开过车，国内的交通更复杂，例如在北京的西直门，有时候可以直接右拐，有时候需要转一个 810 度的圈。

我们希望对导航的方式做一个比较大的变化，让它变成所见即所得的场景。如果有算法能够直接告诉人们往哪边走，对人来说是更加有用的，能够让开车更加简单，导航变得更加直接。

很多汽车现在都会有摄像头，不管是前端还是后端，很多时候可以获取到视频数据。我们把 AI 算法计算出的效果叠加在视频上，告诉人们到底该怎么走。

高德在今年 4 月份发布了 AR 导航产品，这个产品里有一项是实景增强，它会告诉你应该保持在这条线上继续往前开或者转弯，会有压线的提示，会有箭头告诉你前面右转。

这个产品中，除了引导之外，还有别的功能。例如，也加入了前车的碰撞预警功能，会估计前车的距离和速度，这将帮助大家安全驾驶。其他事物也可以用更加直观的方式展示，例如限速，电子眼，跟斑马线相关的，如果看到前方有人，也会做出提示。

以上的功能看起来可能不那么难，但要实现起来很难。为什么？因为我们希望这是每个人马上就能实用的功能，所以要做到很低的成本。这和自动驾驶系统不一样。从传感器的角度，我们要做的是**单个传感器**，而且是**低成本**的相机。从计算的角度来说，**自动驾驶系统可能会用一个几百瓦的专用芯片，而对于我们来说，所需要的算力大概只是普通手机的五分之一**。

给大家看一个 AR 导航的例子，这是实际算法的输出，这个例子里面有车辆的检测，车道线的分割，和引导线的计算等。刚才提到了，高性能（低算力）是一个主要挑战，那我们在开发算法的时候就要充分考虑计算效率，包括各种手段，比如模型压缩，小模型训练优化，检测和跟踪的结合，多目标的联合模型，和传统 GPS 导航的融合，等等，需要几件事情在一个模型里做。

真实世界是非常复杂的，要做到高质量、高效的地图制作，或者做到精准的定位导航，在视觉方面还有很多工作要做。希望通过以上介绍，大家对视觉技术在高德地图中的应用，在出行领域的应用，有了更多的了解，也对高德的使命有了更多了解。

我们在很多时候需要去连接真实世界或者是理解真实世界，才能够让出行更美好。希望能够尽快的把这些做好，让大家实际应用高德 APP 的时候，能够感受到技术进步带来的体验变化。我今天就讲到这里，谢谢大家。

招聘

高德地图视觉攻坚小组火热招聘中：计算机视觉、图像识别、点云、三维、AR、视频等算法类职位正在向你招手~~职位地点 :北京, 西雅图, 湾区。简历投递到 ruby.JL@alibaba-inc.com

自动驾驶中高精地图的大规模生产：视觉惯导技术在高德的应用

作者：you3649

导读

导航、驾驶辅助、自动驾驶等技术的不断发展对地图的精细程度提出了更高的要求。常规的道路级地图对于智能交通系统存在很多不足，针对自动驾驶应用的需求，我们提出了利用视觉惯导技术制作高精地图的方法。

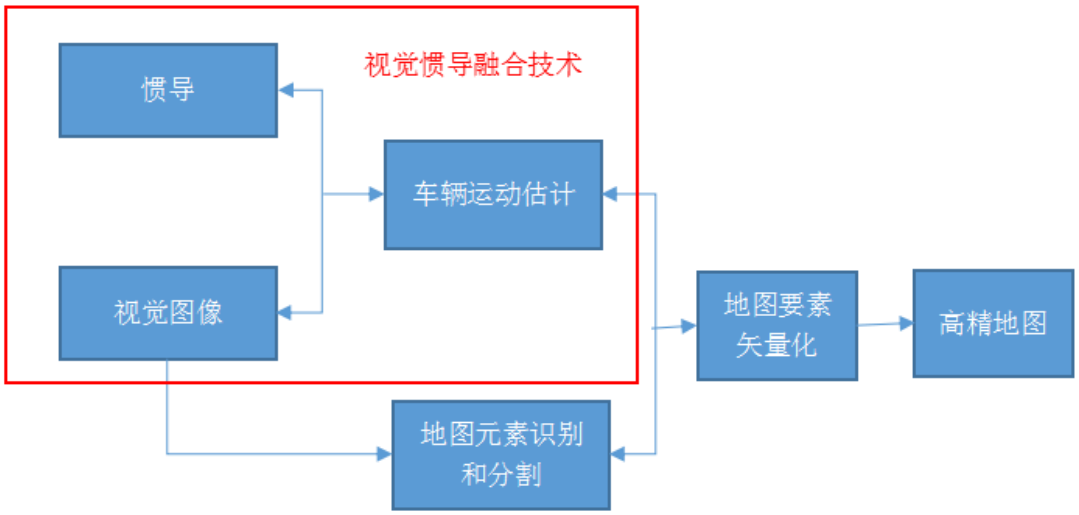
本文将首先介绍视觉和惯导的主流设备，视觉惯导融合的框架和关键技术，高德在基于视觉方式生成高精地图道路标志和地面标识要素的计算方案，最后总结了这项技术在高精地图精度上所面临的挑战和未来发展方向。

视觉惯导技术具有广泛前景

高精地图是自动驾驶的核心技术之一，精准的地图对无人车定位、导航与控制，以及安全至关重要。随着自动驾驶的不断发展，越来越多的车企选择和地图供应商合作。高精度地图需要考虑规模和实时的问题，高德能面向不同品牌车型提供大规模的数据服务，在高精地图行业具有领先优势。

目前，高德完成了全国超过 **32 万公里**高等级道路的高精地图数据，采用了搭配激光雷达采集、图像视觉惯导融合两种方式。

通过图像视觉惯导结合的方式采集数据，一方面能大大降低成本。另一方面，基于图像视觉的高精地图在识别上具有一定优势，能提高车道级别要素作业的效率。因此，这项技术在大规模高精地图生产中具有广泛的前景。



高精地图由高精度的地图要素矢量信息组成，获取这些高精度的地图要素信息，一方面是通过识别视觉图像获取地图要素目标，另一方面通过惯导信息获取车辆高精度的位置和姿态，两方面融合得到对应的矢量地图要素。

视觉惯导硬件工具篇

视觉设备

主流视觉设备按照工作方式的不同，相机可以分为单目相机（Monocular）、双目相机（Stereo）和深度相机（RGB-D）三大类。

单目相机结构简单，成本低，劣势在于照片是三维到二维的映射平面，缺少深度信息，无法通过单张图片来计算场景中物体与我们之间的距离，只有运动才能估计深度。

双目相机由两个单目相机组成，但彼此之间的距离（基线）是已知的。我们通过基线来估计每个像素的空间位置。双目相机测量到的深度范围与基线相关，基线距离越大，能够测量到的就越远。

所以，无人车上搭载的双目相机通常会是个很大的家伙。它的缺点是配置与标定均较为复杂，其深度量程和精度受双目的基线与分辨率所限，而且视差的计算非常消耗计算资源。

深度相机原理是通过红外结构光，类似激光传感器，主动向物体发射光并接收返回的光，测出物体与相机之间的距离。这部分并不像双目相机那样通过软件计算来解决，而是通过物理的测量手段，所以相比于双目相机可节省大量的计算。

深度相机缺点是可能存在测量范围窄、噪声大、视野小、易受日光干扰、无法测量透射材质等诸多问题，室外场景较难应用。

针对高精地图需要大规模生产的需求，单目相机因其成本低，安装简单的特点是目前主流的高精地图视觉设备。

单目相机

深度相机

双目相机

惯导设备

惯性导航系统（简称惯导）是一种不依赖于外部信息、也不向外部辐射能量的自主式导航系统。工作环境不仅包括空中、地面，还可以在水下。

惯导的基本工作原理是以牛顿力学定律为基础，通过测量载体在惯性参考系的加速度，将它对时间进行积分，且把它变换到导航坐标系中，就能够得到在导航坐标系中的速度、偏航角和位置等信息，被广泛应用在军事、测绘、资源勘探、机器人、自动驾驶等领域。

惯导系统具有抗干扰、自主性强、数据频率高、稳定性好等优点。按漂移率从小到大可分为导航级、战术级、工业级、车载级和消费级。目前自动驾驶和高精地图制作领域多选用战术级的惯导设备，以满足高精定位需求。

此外，惯导系统已发展出挠性惯导、光纤惯导、激光惯导、微机电系统惯导等多种方式。其中微机电系统（Micro-electromechanical Systems, MEMS）具有体积小、重量轻、功耗低、价格便宜、抗冲击等优点，被广泛应用，目前已拓展至中低精度的战术级应用领域。

惯导系统单独使用时会有累计误差，实际应用中多与以 GPS 和北斗为代表的全球导航卫星系统（Global Navigation Satellite System, GNSS）等辅助系统构成组合系统，得到载体的全局位置。

当卫星信号丢失时，通过惯导积分可以获取较为准确的实时位姿推算。对于不要求实时性的测绘应用，通过平滑算法能获取更高的定位精度。

在移动测绘领域，惯导的另一个作用是配合激光和相机等外部传感器。通过与 GNSS 耦合得到的载体位姿，可为图片姿态及激光脉冲发射姿态提供高精高频定位，经过传感器间的外部标定，将对应的信息投射到全局三维坐标系。

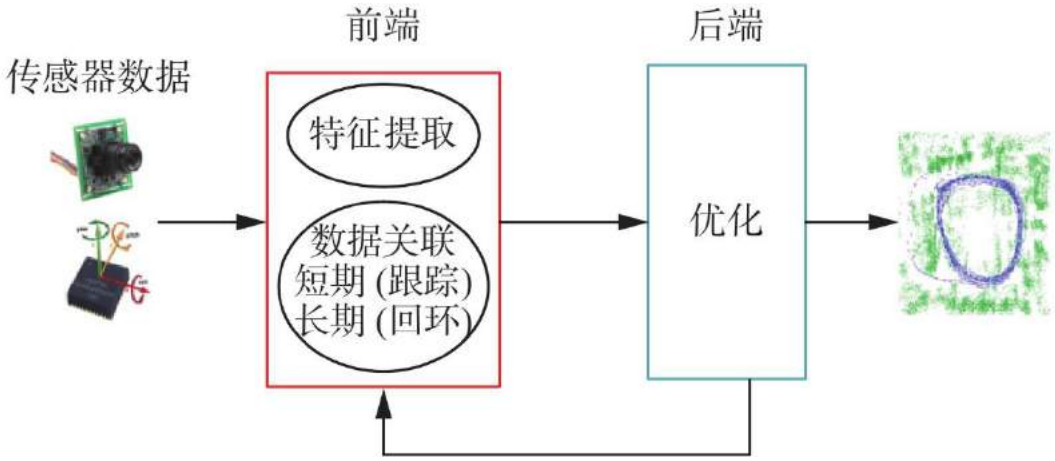
惯导的另一种组合方式是与视觉传感器耦合构成视觉惯性里程计（Visual Inertial Odometry, VIO）。视觉传感器在纹理丰富的场景中 SLAM 效果较好，但是如果遇到移动物体占据照片主体或者特征较少的场景，视觉传感器会失效。

融合惯导数据能提高整体定位精度和连续性。MEMS 惯导单元广泛存在于智能手机当中，苹果公司推出的 ARkit 和谷歌公司推出的 ARcore 框架都提供了相应的 VIO 实现，以支持增强现实应用。

多传感器融合的定位导航方案已经成为趋势， 惯导系统首先与 GNSS 组合，再结合图像、激光雷达等传感器构成的组合导航系统是目前自动驾驶及高精地图制作领域的研究热点和发展方向。

视觉惯导框架及关键技术

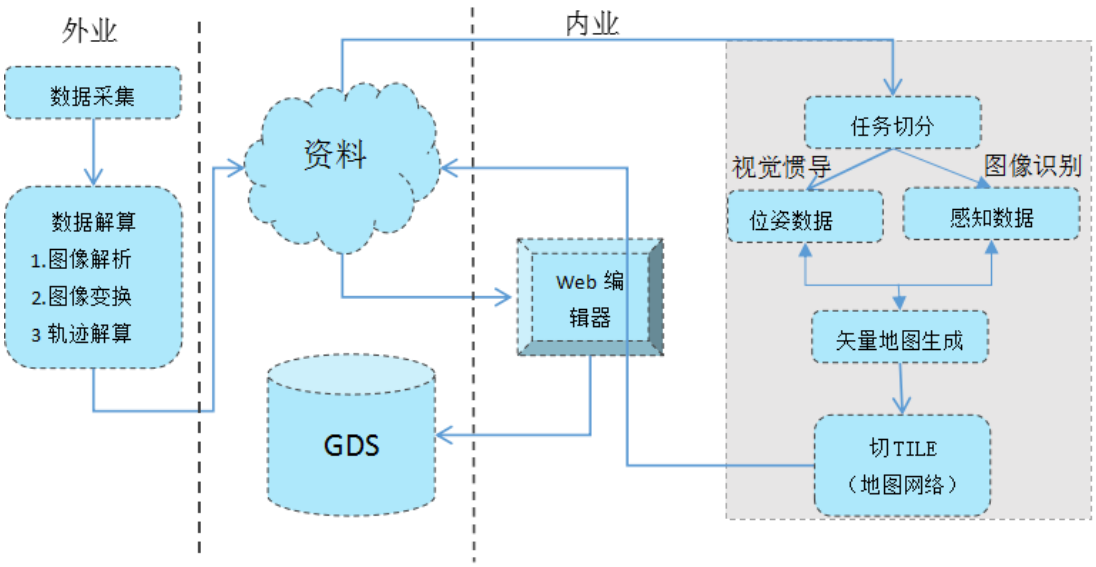
目前主流的视觉惯导融合框架分为两部分：前端和后端。前端提取传感器数据构建模型用于状态估计，后端根据前端提供的数据进行优化，最后输出相机的位置、姿态和全局地图，架构如图所示：



视觉惯导技术框架中前端和后端的优化是关键技术，本文介绍的是采用滑动窗口的模式进行视觉融合惯导的局部相对优化，当初始化失败的时候考虑融入纯视觉 SFM 加惯导对齐的方式进行初始化，相对优化之后会有一个全局的优化，最后对整个地图做绝对的优化。

高德高精地图生产技术方案

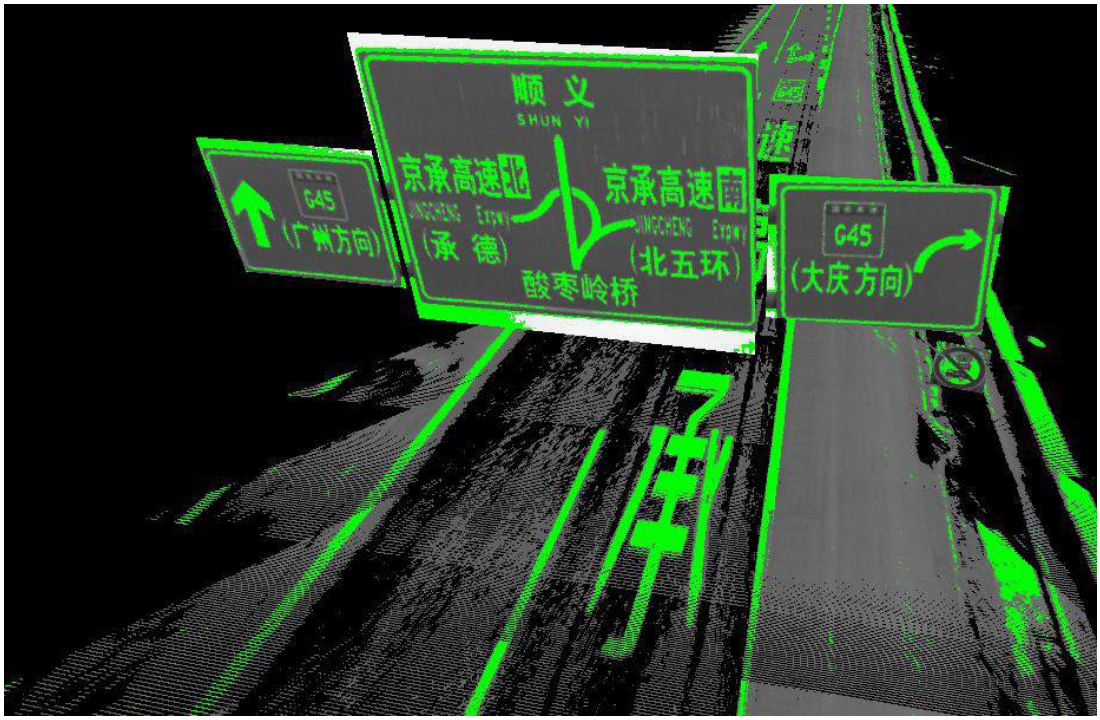
高精地图的生产主要从两类要素进行，一类是道路标志牌，例如路面导向指示牌，红绿灯等；一类是地面标识，例如车道分割线，导向箭头等。两种类别的地图要素均要先计算出位置，然后把要素和路网关联，得到要素的属性信息和几何信息。



地图要素的生产把人工作业和自动化提取融为一体。首先，通过外业采集的数据进行图像和轨迹的解算，获取自动化所需的视觉惯导信息，根据视觉惯导融合技术生成地图要素，在自动化地图的基础上采用人工进行 Web 编辑的模型，提高地图要素的精度，最后存储到对应的数据库中去。

感知结果示例：

生成地图示例：



展望

基于惯导视觉的高精地图生产方案有很多，国内外公司像 Moment、宽凳科技，lvl5 等都在研究，但是从目前市面上看，由于设备成本限制，基于视觉的高精地图精度极限在 10cm。

后续，基于视觉的高精地图发展可能是朝着多源数据融合的方向，即同一道路多次采集，不同设备多次采集获取的数据源融合在一起，提高精度的同时提高地图更新的时效。

高德扎根于地图行业，有丰富的地图数据源，有行业领先的自动化生产技术和成熟的工艺流程，为未来基于多元视觉惯导融合的高精地图生产打下了坚实的基础，这些都会进一步推动自动驾驶的发展。

招聘

高德地图视觉攻坚小组火热招聘中：计算机视觉、图像识别、点云、三维、AR、视频等算法类职位正在向你招手~~职位地点：北京，西雅图，湾区。简历投递到 ruby.JL@alibaba-inc.com

深度学习在交通标志检测与精细分类中的应用

作者：亮黄

1.导读

数据对于地图来说十分重要，没有数据，就没有地图服务。用户在使用地图服务时，不太会想到数据就像冰山一样，用户可见只是最直接、最显性的产品功能部分，而支撑显性部分所需要的根基，往往更庞大。

地图数据最先是从专业采集来的，采集工具就是车、自行车、飞机和卫星影像等，近两年有了利用智能硬件的众包采集。采集之后，就是把数据更新的速度和精准度都无限提升。因为地面上变化太快了，用户越来越依赖于地图应用。所以数据更新的速度和数据的准确度就是在乎用户体验的地图公司的第一要务了。而数据更新的第一步，就是交通标志检测。

本文将主要介绍机器学习技术在高德的地图数据生产的具体应用，这些技术方案和设计都已经过验证，取得了不错的效果，并且为高德地图数据的快速更新提供了基础的技术保证。

2.“交通标志检测”定义

交通标志检测，特指在普通街景图像上通过自动化手段检测出各种类型的交通标志，如限速、禁止掉头、人行横道和电子眼等。这些检测结果将作为生产数据交付给地图数据制作流程，最终演变为服务于广大用户的地图数据。

3.难点与挑战

交通标志检测的主要难点有样式繁杂，且在拍摄过程中受自然环境的影响较大。此外，为满足数据更新的速度和数据准确度的要求，对于算法的性能要求也格外严格。

3.1 样本形态差异大

交通标志的形态差异主要体现在：

- 类型繁多：国标定义的交通标志有几百个类型。
- 形状多样：常见交通标志的形状有三角形、圆形、方形、菱形、八边形等，还有地面标线、电子眼、信号灯，以及限高杆、栅栏等物理设施。
- 颜色分布广泛：常见的有黄色、红色、蓝色、绿色、黑色、白色等。

- 图像内尺寸差异大：从几百像素（如方牌、人行横道等）到十几像素（如电子眼）不均匀分布。

图 1 常见道路交通标志(标牌类)

3.2 自然场景下变化多端

在自然场景下，交通标志存在树木或车辆遮挡、磨损等情况；天气、季节等也会影响到图像采集过程中，造成图像模糊、颜色失真等。

图 2 自然场景下拍摄的交通标志

一些外形与交通标志相似的标牌，如商户的招牌、交通公益广告牌等，对算法的准确率造成极大的挑战。

图 3 类似交通标志的噪声示例

3.3 性能要求

- 准召率：我们的应用场景中对于召回率和准确率的要求极高，任何未召回都会导致数据更新的延迟，而错召回则会影响作业效率与作业周期，最终对数据的快速更新造成影响。

- 吞吐量：高德每天需要处理上亿张图片，这就要求我们的算法不仅效果要好，处理速度也必须够快，以免造成数据积压，影响地图数据的更新时效。

- 扩展性：交通标志的类型不是一成不变的(国标会存在调整，不同国家和地区之间各有特色)，因此需要算法环节具有非常好的扩展性，能够快速适应新增的各种交通标志类型。

4.高德地图中的交通标志检测方案

当前学术界针对目标检测任务常用的深度学习模型一般都采用 End2End 的方式进行训练，以得到全局最优的检测效果。这个方案在使用时非常简单，只需要标注好“几百类物体的样本”，然后放到深度学习的框架里进行迭代训练，就可以获得最终模型，主要可以分为 Two Stage(FasterRCNN[1])和 One Stage(YOLO[2],SSD[3])两大类。

但是在实际使用过程中，需要应对如下问题：

- 样本标注成本高：所有训练样本都需要进行全类别标注，当有新增类别时需要将历史训练样本全量补标，成本极高。

- 无法单类迭代：由于交通标志出现的频率和重要性不等，业务上对于部分类型(如电子

眼、限速牌等)的准召率要求更高。但是 End2End 的模型必须针对所有类型全量迭代，无法优化单一类型，导致算法迭代和测试成本极高。

- 模型训练难度大：我们需要处理的交通标志有几百类，且各自出现频率差异很大，使用单一目标检测模型完成如此巨大的分类任务，模型训练难度太大，收敛缓慢，召回率、准确率上难以平衡。

结合通用目标检测技术的发展以及高德地图对于交通标志检测的需要，我们最终选择了 Faster-RCNN 作为基础检测框架，它的检测效果更好(尤其是针对小目标)，独立的 RPN 网路也可以满足扩展性要求。速度方面，我们也进行了针对性的优化调整。

在实际使用时，我们将检测框架分为目标检测与精细分类两阶段：

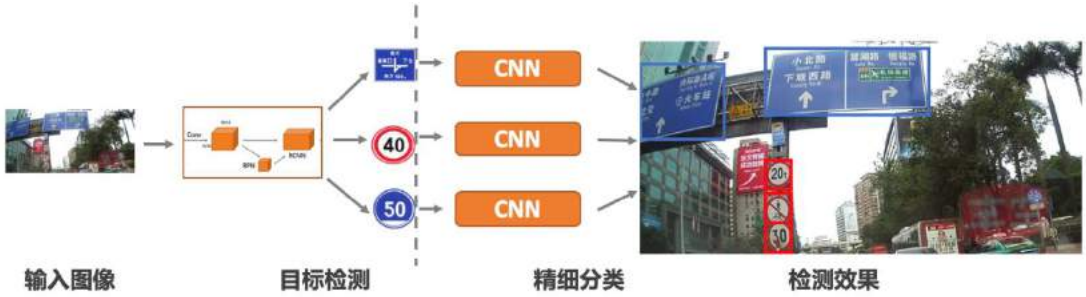


图 4 交通标志检测的目标检测和精细分类阶段

4.1 目标检测阶段

目标检测阶段的目的是通过 Faster-RCNN 在图片中检测所有的交通标志，并进行粗分类，要求极高的召回率和执行速度。在实际使用时，我们采用了如下策略来提升算法能力：

- 效果方面：将检测目标根据外形特征分为 N 大类(如圆形、三角形、方形，以及高宽比异常的人行横道等)，再为每一类配置专属的 RPN 网络，各个 RPN 根据对应的尺寸特性设计 Anchor 的 Ratio 和 Scale；不同 RPN 根据需要使用不同层的特征图，设计更有针对性。

- 效果方面：针对各个类型样本分布不均匀问题，使用多种样本增强手段，并在训练过程中使用 OHEM 等方式进一步调整样本分布。

- 效果方面：借鉴了 IoU-Net、Soft-NMS 等方案，进一步提升检测效果。

- 性能方面：各个大类之间共享基础卷积层，保证检测时间不会过分增长。

- 扩展性方面：对于新增类型，理想情况下只需要新增一个 RPN 网络单独迭代，可以不对其他类型的效果造成任何影响(如下图，RPN1 和 RPN2 完全独立)。

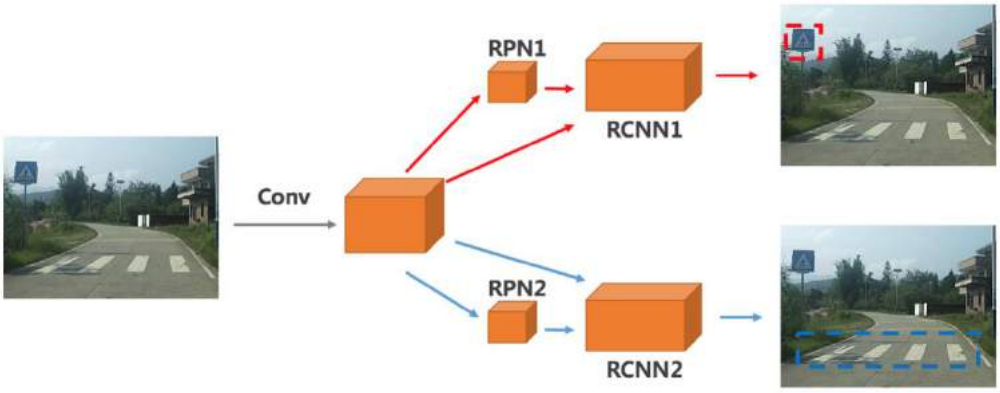


图 5 多 RPN 设计示意图

4.2 精细分类阶段

精细分类阶段的目的是对目标检测阶段得到候选框进行精细分类并滤除噪声，保证极高的召回率和准确率。在实际实现中，还使用以下策略来提升效果：

- 为每个大类配置独立的精细分类网络，互相之间不干扰；各个大类的迭代完全独立和并行，可以多人并行研发，有效缩短研发周期。

- 针对各个大类的难易程度，选择不同计算复杂度的网络来完成精细分类和噪声抑制，避免因为某些类型复杂度过高产生效率瓶颈。

- 样本方面，各个大类可以独立收集样本，可以针对特定类型进行收集和标注，训练和测试集合的构建效率大幅提升。

如下图，针对圆形标牌，其差异比较明确，可以使用简单网络；针对方牌，需要根据文字布局和内容来区分正负样本，分类难度大，因此必须使用较深的网络：

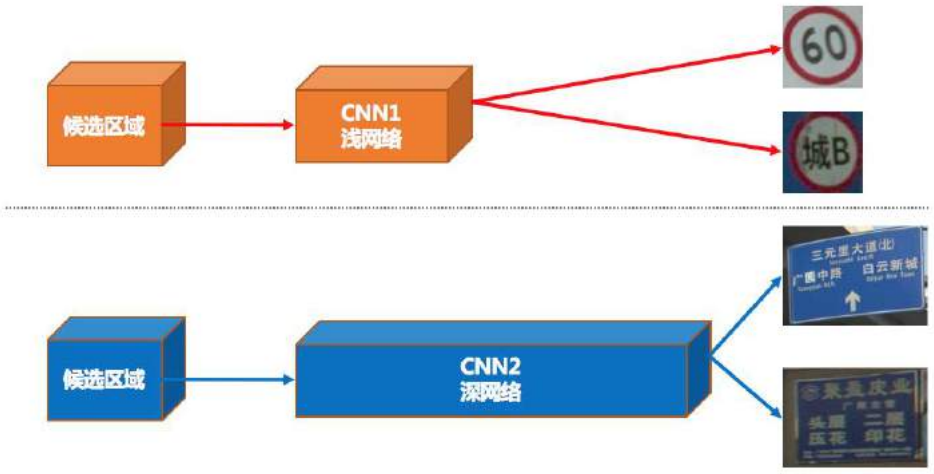


图 6 精细分类模块示意图

由于同时使用了多个模型，上述方案会导致服务器的显存占用显著增加，对计算资源产生额外要求。针对该问题，我们针对深度学习框架进行优化，动态分配并在各模型间共享临时缓冲区，并裁剪框架的反向传播功能，最终使得显存占用降低 50%以上。

5.效果与收益

上述方案已经正式上线，准召率都达到了生产作业的要求，日均图片吞吐量在千万以上。以下是部分效果图(不同框代表不同检测结果)：

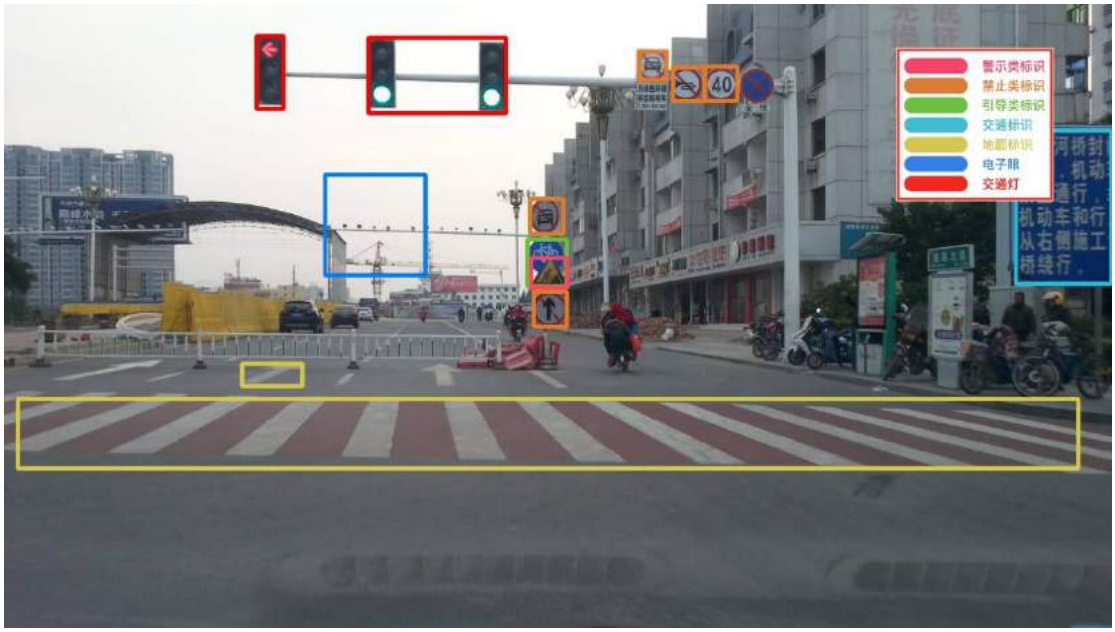


图 7 交通标志检测效果图

6.小结

交通标志检测技术已经在高德地图内部得到应用，有效提升了高德地图的数据制作效率，达成地图数据更新速度接近 T+0（时间差为零）的目标。

目前我们也在把机器学习技术用于数据的自动化制作，进一步减少现实世界和地图数据之间的差异，做到“连接真实世界，让出行更美好”。

招聘

高德地图视觉攻坚小组火热招聘中：计算机视觉、图像识别、点云、三维、AR、视频等算法类职位正在向你招手~~职位地点：北京，西雅图，湾区。简历投递到 ruby.JL@alibaba-inc.com

车道线检测在 AR 导航中的应用与挑战

作者：达瑟

1.导读

现代社会中，随着车辆的普及，人的活动范围在逐步扩大，单单依靠人类记忆引导行驶到达目的地已经越来越不切实际，因此车载导航就扮演了越来越重要的角色。

传统车载导航根据 GPS 将用户车辆位置定位于地图上，导航软件根据设定的目的地规划行驶路径，通过屏幕显示和语音播报的形式指引用户行驶至目的地。这样的展示方式使得用户在使用导航的过程中，需要将地图指引信息和语音播报信息与当前自车所处的真实世界连接起来，才能理解引导信息的具体含义，之后做出相应驾驶动作。

总结导航信息处理的整个过程，可以分为三个部分：接收(看/听)、理解、行动。

然而，试想这样一个场景，行驶前方有个岔路，由于没有听清楚需要走左前方还是右前方，这个时候就很容易错过变道时机。

因此，AR 导航利用视觉技术，综合导航信息和真实场景信息，优化了引导信息展现形式，降低了用户的接收和理解成本，所见即所得，用户只用根据导航信息行动即可。

2.AR 导航的定义

车载 AR 导航是指通过摄像头将车辆前方道路的真实世界实时地捕捉下来，利用移动端视觉识别技术及算法，实时地识别车辆行驶场景中的各种重要导航要素，例如车道线、前方车辆、车道相对位置等，再结合自车 GPS 定位、地图导航信息，通过多元信息的融合以及计算，生成虚拟的导航指引模型，并渲染叠加到真实场景上，从而创建出更贴近驾驶者真实视野的导航画面（图 1）。

图 1 AR 导航

在驾驶车辆过程中，车道线的重要性不言而喻，它通过不同的属性，例如虚、实、黄、白等，来指引车辆的行驶方向，规范驾驶员的驾驶行为，避免车辆之间的碰撞，最终实现更加高效和流畅的交通。

在 AR 导航中车道线同样重要，实时车道线检测能够为 AR 导航引擎提供当前车道宽度、车道线属性等信息，从而提前对用户做出正确的引导，指引驶入正确的车道。

如上图所示，通过车道级定位将自车定位在当前路左数第二车道，这时根据导航信息前方将要左转，与此同时通过车道线检测获知左侧车道线为白色虚线，允许变道行驶，再通过车道宽度和自车在车道内距左右车道线的距离，渲染出正确的引导线，指引用户提前变道驶入左数第一车道，避免错过路口，导致偏航行驶，浪费时间精力。

此外，车道线检测还能提供 ADAS 功能，如车道保持、车道偏离预警（LDW）等。

3.车道线检测背景介绍与方法回顾

有关车道线检测的研究已经持续了比较长的时间，虽说已经取得了一定的成果，但是想要实际应用仍然具有非常大的挑战，导致这种现状主要有以下几个方面的原因：

- 图像质量问题：摄像头采集得到的图像由于车辆遮挡、树木和建筑的阴影、车辆移动带来的光照剧烈变化等原因而变化巨大。
- 光照场景受天气因素影响较大：雨天、雪天、雾天、黄昏、夜晚等。
- 车道线磨损程度不一：高速和城市快速路车道线较清晰，低等级道路磨损较严重，与轮胎划痕难以区分。
- 车道线宽度不一：通常来说车道线宽度在 2.3m-3.75m 之间，但在现实世界中，特别是低等级道路，车道线宽度变化较大。

多样丰富的场景给基于视觉的算法带来了巨大的挑战，本文将对车道线检测方案进行讲述，并以是否使用深度学习为界，分为以下两个大的体系来介绍：**基于特征工程的传统视觉方案和基于图像分割的深度学习方案**。

3.1 基于特征工程的传统视觉方案

传统视觉方案里，车道线检测过程较少使用机器学习方法，主要是利用车道线的视觉特征和空间位置关系实现车道线检测提取，通常来说分为以下几个步骤：

- **预处理**：对图像进行预处理，通常包括车辆等障碍物去除、阴影处理、划定感兴趣区域(ROI)、前视图转为俯视图等。
- **车道线候选点提取**：基于颜色和纹理，针对车道线特点设计特有的图像特征提取方式，常用的方法有灰度阈值处理、颜色阈值处理、边缘提取、特定滤波器滤波等。

- **车道线拟合**：在获取到候选点之后，通过一些先验知识(如车道线在俯视图上是平行关系)设置规则，去除一部分 outlier 车道线候选点，之后可采用参数方程的方式，进行车道线的拟合。常见参数方程有直线、二次曲线、样条曲线等，不同的拟合算法对性能也有很大的影响，其中 RANSAC 算法能够较好区分 outlier 和 inlier，所以在车道线拟合过程中得到了广泛的关注。

- **后处理策略**：通过坐标映射，利用车辆行驶过程中时间和空间的连续性，实现车道线跟踪和滤波，从而提高车道线检测的稳定性和准确性。

传统视觉方案检测车道线过程依赖较多的先验假设，特征设计依赖经验阈值的调整，在实际应用中无法应对复杂的道路情况，因此鲁棒性较差，尤其是在光照条件变化、车道线磨损程度不同的影响下，经验阈值非常容易失效，导致较差的检测结果。

但是由于其计算量较小，在算力受限且路况单一(高速/城市快速路)的场景下，仍然可以发挥它的价值。

3.2 基于图像分割的深度学习方案

深度学习图像分割自 2014 年 FCN 提出以来发展迅速，在众多的图像任务中取得了不凡的结果。并且由于车道线在 ADAS 和自动驾驶任务中的特殊地位，可以将车道线检测逐步从通用的分割任务中独立出来，并且取得了较大的进展。

基于图像分割的车道线检测方案处理流程与传统视觉方案类似，主要区别在于车道线候选点的提取方式上，车道线图像分割不需要关于车道线的纹理/颜色/宽度/形状等先验假设，而是通过机器学习从训练样本中获取车道线的相关信息，自动地学习车道线的特征，具体应用时便可以通过学习得到的信息，来预测单个像素位置是否为车道线。

深度学习车道线检测方案基于通用深度学习分割方案，但是针对车道线场景进行特别的改进。

接下来，先介绍近期深度学习车道线检测的相关工作，而后描述 AR 场景下的车道线检测方案。

由于在较多的应用场景中只关注当前车道车道线，大多数方案是先识别全部的车道线，然后通过空间位置关系进行后处理，提取出当前车道车道线，但这个过程中容易出错，稳定性较差，Jiman Kim 在 2017 年提出在图像分割时赋予当前车道左右车道线不同的类别，把单条车道线当做分割的一个实例，通过 End-to-End 方式，直接从图像中提取出当前车道的左右车道线，从而避免了后处理过程中区分左右车道线，降低出错概率。

商汤科技 Xingang Pan 针对车道线细长的结构特点，提出 Spatial CNN 替换 MRF/CRF 结构，在高度和宽度方向（从上至下，从下至上，从右至左，从左至右）逐层进行卷积，以增强信息在空间上的流转，实例分割以当前车道为中心的 4 条车道线，与此同时输出单条车道线的置信度，可以为实际使用的多元信息融合提供依据。该方案在图森未来举办的 Lane Detection

Challenge 上取得了第一名。

从前视视角看，车道线最终都交汇于消失点，为了让网络学习图像中的结构信息，Seokju Lee 提出了一个多任务网络，在检测车道线的同时检测地面标志和消失点。针对消失点的特点，作者设计了一个精巧的结构用于检测消失点，并通过实验证明了消失点任务的加入提高了车道线的检测效果。

相比于检测指定数量的车道线，Davy Neven 在 2018 年提出将所有车道线当做一类进行语义分割得到 Binary lane segmentation，与此同时网络输出 Pixel embeddings 结果，提取 Binary lane segmentation 前景区域像素点在 Pixel embeddings 输出上的特征值，使用聚类算法，确定每个像素点的车道线类别，从而实现不定数量的车道线实例分割，以适应不同的车道线场景。

综上，相比于通用图像分割，车道线分割方案主要在利用车道线之间的位置结构关系，针对车道线细长的特点，优化深度网络的空间信息提取能力，并将更多的后处理工作融入至网络中，减少后处理难度和出错概率。

4.AR 导航中的车道线检测方法探索和实践

车载 AR 导航要求将引导要素实时迭加到真实场景中，这对于 AR 导航中的车道线检测实时性和稳定性提出了极高的要求，与此同时，由于车载设备(车机/车镜)的硬件算力较差，一般落后于手机芯片 3-5 年，所以 AR 导航中的车道线检测必须做到又快又好。

为了在车载设备上实现快速高效的车道线检测算法，我们在多个方面进行了尝试：

4.1 高效的多任务模型

由于交通图像中车辆和车道线有一定的相关性（车辆一般情况在两条车道线中间），为了充分的利用深度学习网络能力，共用网络主干部分提取到的图像特征，我们设计了一个高效的多任务网络（图 2 所示），单个模型完成车辆检测、车道线检测任务和其他任务，并在此基础上，实现了一套多任务权重自学习的机制，保证各个任务高效充分的学习，最终使得多任务学习模型达到独立单任务模型持平的识别效果，部分任务甚至略有超出。

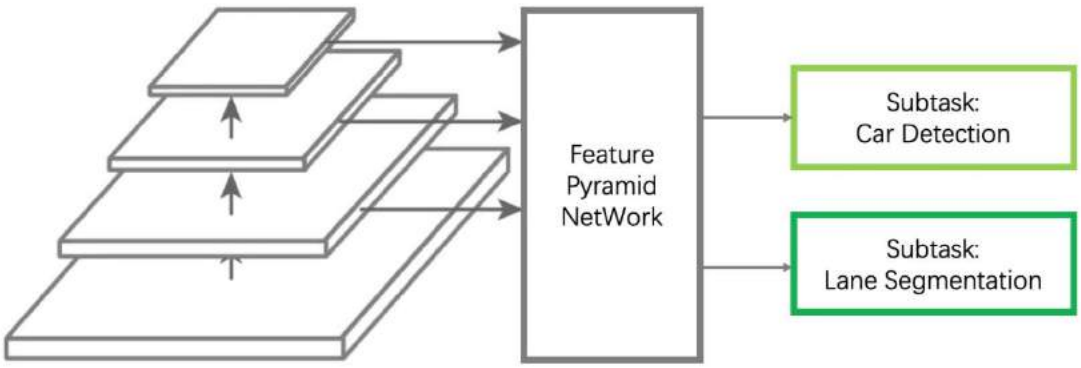


图 2 多任务网络

在多任务模型迭代的过程中，多任务样本的标注成本较高，如果需要补充某个任务的数据集，则必须在该数据集上对所有的任务进行标注，耗费较多的标注资源。从迭代模型和利用已有的独立任务标注数据角度出发，我们研发了一套以任务为粒度的训练框架，允许单张图片有任意个任务进行了标注（比如只标注了车道线，或者同时标注了车辆和车道线等）。

由于和车辆共享主干网络，车道线的 ROI 设置从图像输入转移至车道线分割分支的特征层，进一步降低了车道线检测分支的计算复杂度，最终车道线检测分支仅占用原有车道线检测网络 15%的计算量。

4.2 消失点优化车道线检测

一般来说在车道线分割方案中有两种真值标注方式：

- 第一种是常规的道路场景分割标注：在这种情况下，标注规则按照车道线的物理含义，随着车道线由近及远、由粗变细，如 KITTI、CamVid 等数据集。在这种标注规则下能够较精确的获取车道线内侧距离车辆的距离，提高横向车道定位精度，但由于远处车道线较细，特征不明显，车道线分割精度随着距离的增加逐步降低。

- 第二种标注方式常见于单纯的车道线检测任务：如 CULane 等，这种标注方式将车道线定义为白色区域的中心线，以固定的图像像素宽度生成分割真值标注。相比第一种方式，该方法能够在一定程度上提高远处车道线的识别效果，得到较完整的车道线。但在进行车道线扩宽时，远处不同车道线的真值容易互相压盖，造成堆叠部分车道线识别效果较差，增加了车道线后处理难度。

在 AR 导航中，我们采用第二种车道线标注方式，并在车道线检测模型中增加消失点识别分支，在车道线后处理中以消失点为锚点，优化车道线识别精度。

4.3 深度学习神经网络量化

深度学习神经网络在训练过程中，为了接收反向传播的梯度，实现对模型权重的细微调整，一般采用高精度的数据格式进行计算和权重的更新，最终完成模型训练，保存模型结构和权重。

神经网络量化就是将高精度的模型权重量化为低比特的数据，以使用更少的数据位宽来实现神经网络的存储和计算，这样既能减少运算过程中的带宽，又能降低计算量。由于移动端 CPU 带宽资源有限，通过神经网络量化，可以较大地提高模型运算的速度、降低模型空间占用，以 TensorFlow 为例，量化后的 uint8 模型与量化前的 float 模型相比，速度提高 1.2 到 1.4 倍，模型空间占用降低 3/4。

为了进一步优化神经网络量化后的速度，我们在 tflite-uint8(基于 TensorFlow r1.9)量化基础上进行二次开发，实现了一套 tflite-int8 量化框架，其中包含了量化模型训练、模型转化和自研的 int8 矩阵运算库，相比官方 tflite-uint8，tflite-int8 在移动端 A53 架构上有 30%的提速，在 A57 架构上有 10%的提速，与此同时，量化前后多任务模型的精度几乎保持不变。

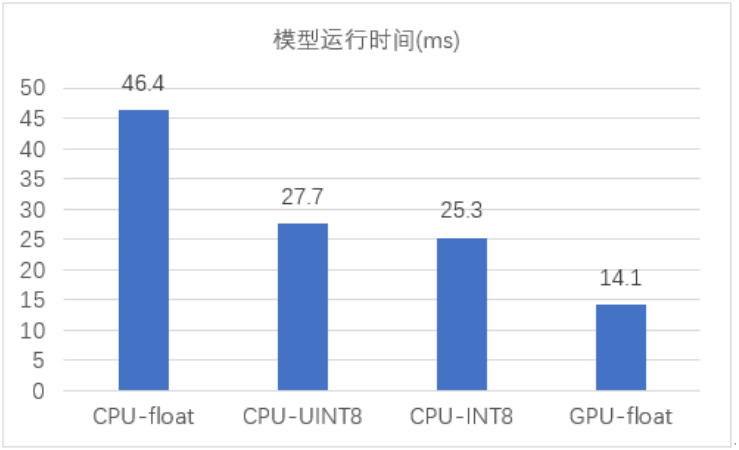


图 3 小米 5s 上单线程模型运行时间对比

最终通过上述方法，在较低算力的车镜/车机芯片上实现了实时稳定的车道线检测，骨干提取后效果图如下图：

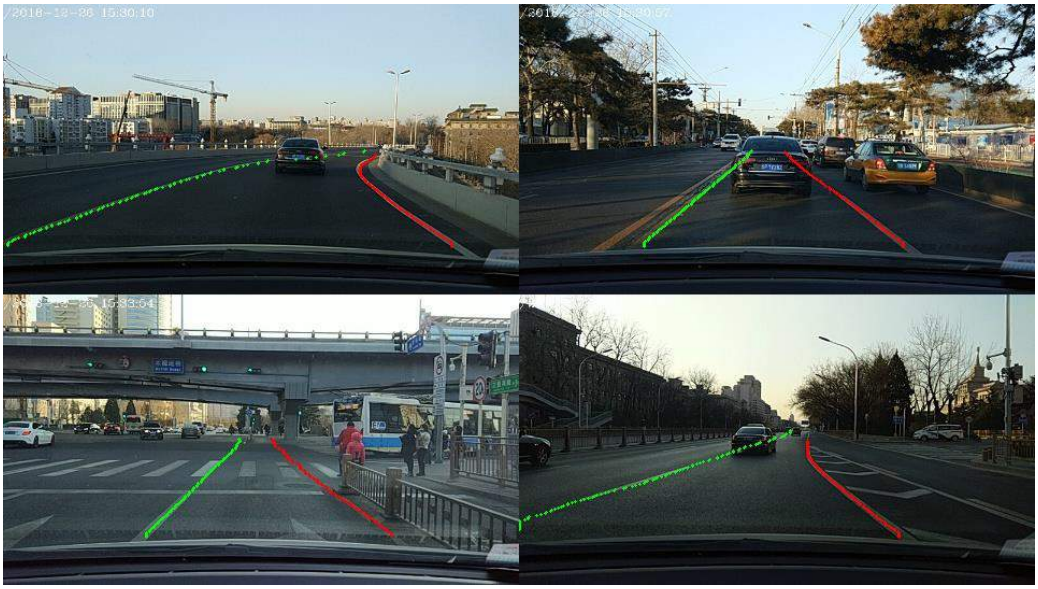


图 4 车道线骨干提取效果

5.挑战与展望

在 AR 导航中，车道线有着举足轻重的地位，作为 AR 导航的基础，搭建在其上的一系列导航功能的好坏都与它的检测精度息息相关。然而实际道路场景千变万化、天气光照也四季不同，这些都给车道线的检测识别带来了较大的难度。

因此，在后续的研发过程中，我们将不断扩大覆盖范围，充分利用高德自采的大量道路数据的优势，结合相关传感器和导航数据不断完善和优化，来进一步地提高车道线的检测精度，以更好地服务 AR 导航项目，最终向用户提供更直观高效的导航服务。

招聘

高德地图视觉攻坚小组火热招聘中：计算机视觉、图像识别、点云、三维、AR、视频等算法类职位正在向你招手~~职位地点 :北京, 西雅图, 湾区。简历投递到 ruby.JL@alibaba-inc.com

高精地图中地面标识识别技术历程与实践

作者：渔晓

导读

本文将主要介绍高德在高精地图地面标识识别上的技术演进，这些技术手段在不同时期服务了高精地图产线需求，为高德地图构建高精度地图提供了基础的技术保证。

1.面标识识别

地面标识识别，指在地图道路中识别出各种类型的地面标识元素，如地面箭头、地面文字、时间、地面数字、减速带、车距确认线、减速丘、人行横道、停止让行线、减速让行线等。这些自动化识别结果将作为生产数据交付给地图生产产线，经过制作后演变成服务于自动驾驶、车载导航、移动导航的地图。

高精地图一般对各个地图要素精度至少有着厘米级的要求，所以相对于普通地图来说需要更高的位置精度，这也是与普通地图识别的最大不同，所以探索如何将地面标识识别得又全又准是我们一直努力的方向。

地面标识识别有两大难点：一是地面标识本身的种类、大小繁多，二是地面标识易被磨损遮挡，清晰度参差不齐，这给高精度识别带来了巨大的挑战。

地面标识种类繁多：实际场景中地面标识种类繁多，在内容、颜色、形状、尺寸等方面均有不同分布。

- 颜色：比如黄色、红色、白色等
- 形状：箭头形、各种文字数字形状、条形、多条形、面状、丘状等
- 尺寸：国标定义的标准箭头长度为 9m，但也存在 1m~2m 甚至 1m 以下的地面标识元素，尤其减速带以及人行道等尺寸差异会更大，反映到图像中像素个数以及长宽比均会有较大差异。

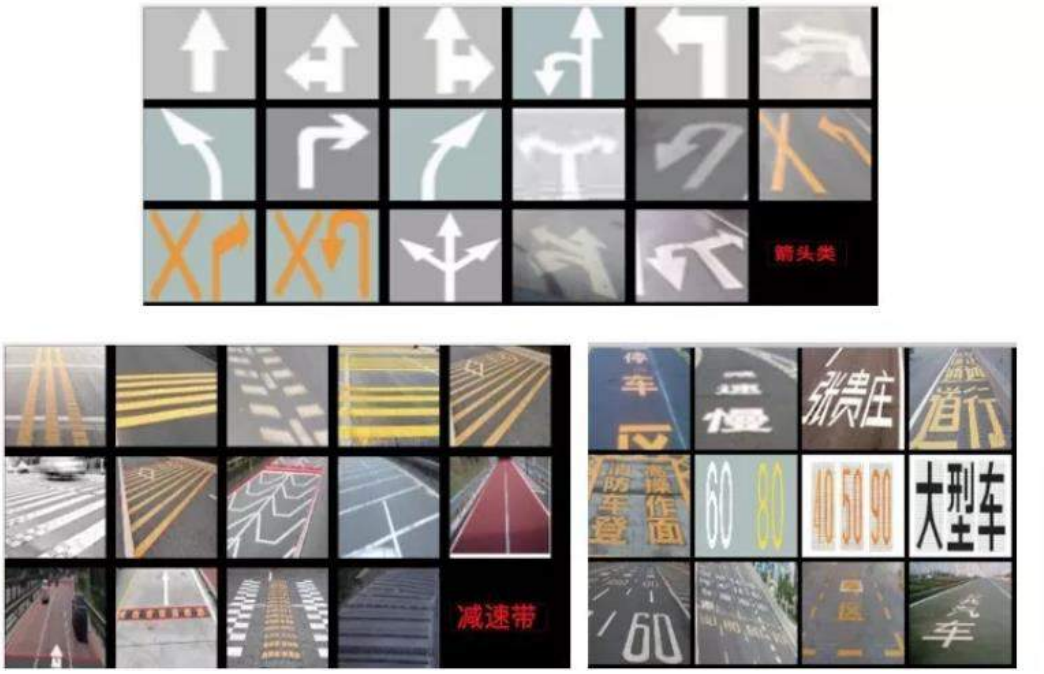


图 1. 部分地面标识

磨损压盖多：地面元素长年累月受车辆、行人等碾压会造成磨损，以及经常存在的堵车等场景更是加大了地面要素被遮挡的可能。所以从激光雷达获取的点云数据和由相机获取的可见光图像数据的质量均参差不齐，对地面标志识别带来了极大的挑战。

常见的问题如下所示，示例如图 2 所示。

- 地面标识磨损：地面标志由于磨损褪色、掉漆导致不完整或者严重不清晰
- 采集环境问题：遮挡(施工、车辆)、由于环境改变引起的材料激光反射率差异以及可见光不清晰（雨天、逆光等）

图 2. 自然场景下拍摄的地面标识

2. 识别起步

地面标识识别需要做的是将地面标识这部分区域提取出来，则最直观的是对其进行阈值分割、骨架提取、连通域分析等传统方法。首先获取点云中地面点集合，接着获取集合中高反射率部分的骨架集合，然后对每个局部骨架区域计算强度截断阈值，最后对区域进行连通区域搜索以及附加降噪措施等。

另外我们也尝试了 GrabCut 等算法在地面标志上的提取，GrabCut 算法对前景和背景分别聚类，得到 k 组类似的像素集合，然后对前景和背景分别进行高斯混合模型（GMM）建模，判断像素属于地面标志还是背景。在提取疑似地面标识区域后，再经过机器学习模型（SVM 等）进行细分类以获得更好的识别效果。

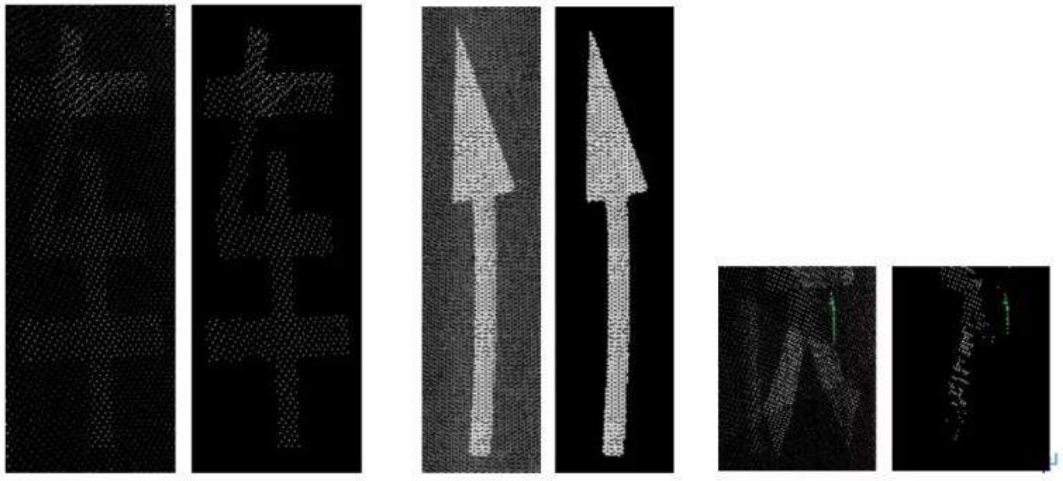


图 3. 传统提取方法识别结果

由上图可以看到，对于一些前后景区分比较好的地面标识提取的比较好，但是针对有磨损、模糊、前景背景相似、背景复杂等情况均效果欠佳，容易漏召回且位置精度不高，鲁棒性不强。

3.深度学习时代

2012 年 Hinton 团队提出的 Alexnet 网络赢得了 2012 年图像识别大赛冠军，相比传统方法获得显著提升，CNN 在图像领域有了明显优势。近几年，基于深度学习的检测识别技术也得到了很大发展。

深度学习时代是数据和硬件驱动的时代，结合部分人工标注以及自动化生成，我们拥有百万级的数据，而且各种场景的数据还在不断丰富，结合算法探索与创新，我们取得了越来越好的技术与业务效果。

目前检测识别技术主要分为两大方向：Two-Stage(如 RCNN 系列)和 One-stage(SSD、YOLO 等)。Two-Stage 网络优势在于效果整体较好，识别位置较精确，对小目标检测也有一定的竞争力。

One-stage 检测识别方法优势在于处理速度较快。高精地图不仅需要较高的识别性能、也需要有足够高的识别位置精度，所以我们选择了准确率较高的 Two-stage 大方向。

1）R-FCN 检测

结合位置敏感得分图（position-sensitive score map）和位置敏感降采样（position-sensitive roi pooling）等操作，R-FCN 算法在目标检测识别上获得了较高的性能和位置精度，我们选择了 R-FCN 检测算法实现对地面标识的检测识别。

R-FCN 算法基于深度学习的方法，通过学习大量实际场景样本，所以在泛化性上取得了比较大的提升，自动化识别对于不同场景的识别能力有所提高，地面标识召回率得到了较大的改善。算法示意图如下所示：

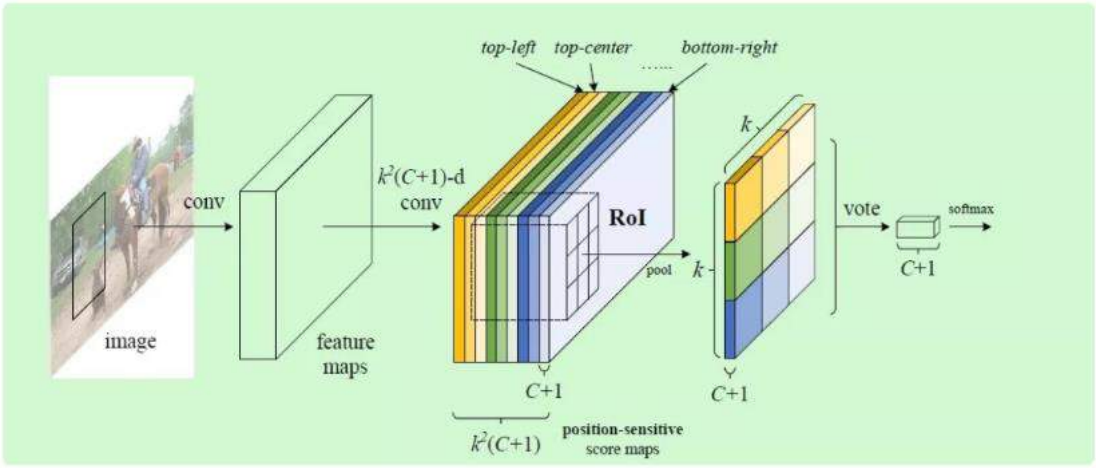


图 4. R-FCN 算法示意图

以下为一些地面标识检测识别示例：

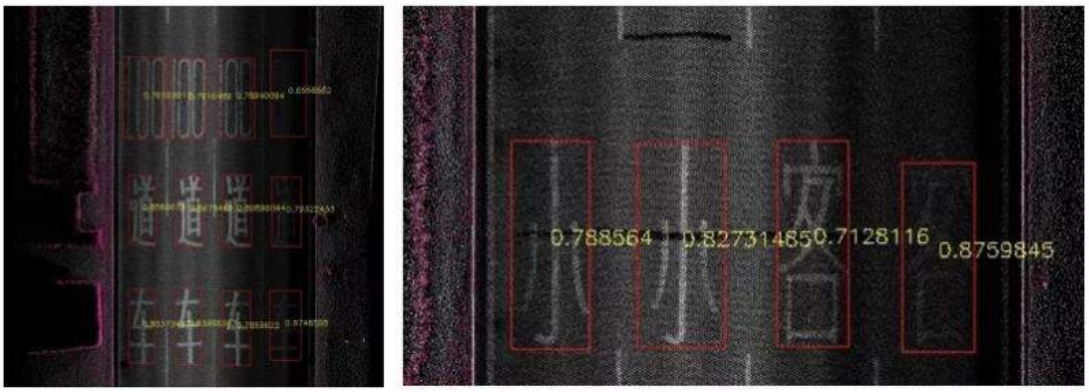


图 5. R-FCN 算法地面标识识别示例

引入深度学习极大的改善了高精地图地面标识自动识别的性能，地面标识召回得到了很大提升，美中不足的是 R-FCN 存在着一个弊端就是其输出的最终检测位置是基于地面标识类别的得分，但往往得分最高的位置并不一定跟实际位置最贴合，所以在位置预测精度上，R-FCN 并不完美。

2）级联检测器

随着深度学习的发展以及业界对目标检测识别位置精度的要求不断提高，更多高精度检测识别算法被提了出来，如 Iou-Net 等。

我们适时采用了更加先进的识别算法，以期获得更加精准的位置精度来满足产线业务需求，结合级联检测，利用 Deformable-Conv 自适应感受野等技术提升算法识别精度。

该算法不同于传统算法对 roi 进行一次预测回归得到最终位置，而是通过级联的形式不断修正预测的位置和实际位置的偏差，每经过一个级联回归器，算法识别结果均会更加贴合真值，这非常有利于提高识别精度，契合高精地图对目标位置精度的高要求，最后在召回和位置精度上都达到更好的效果。

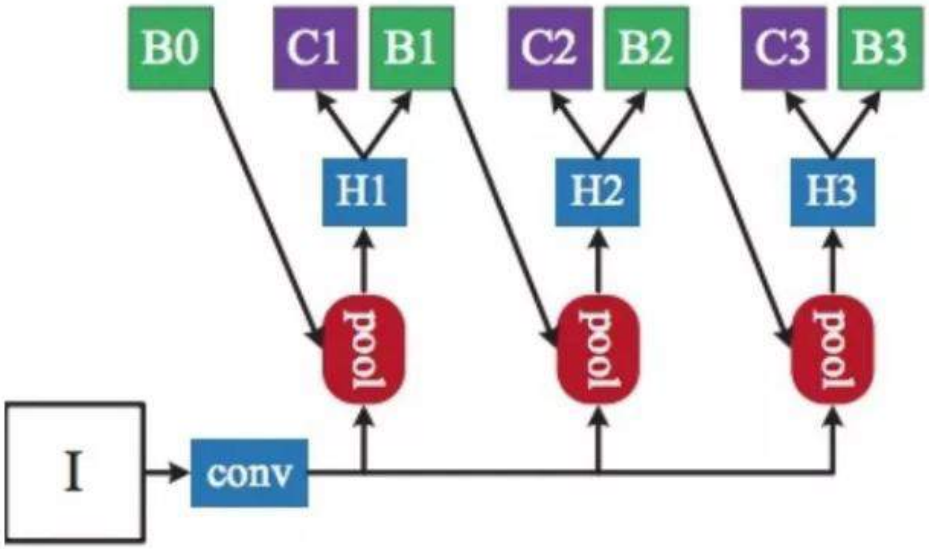


图 6.级联检测算法原理图

以下为一些算法识别结果示例：

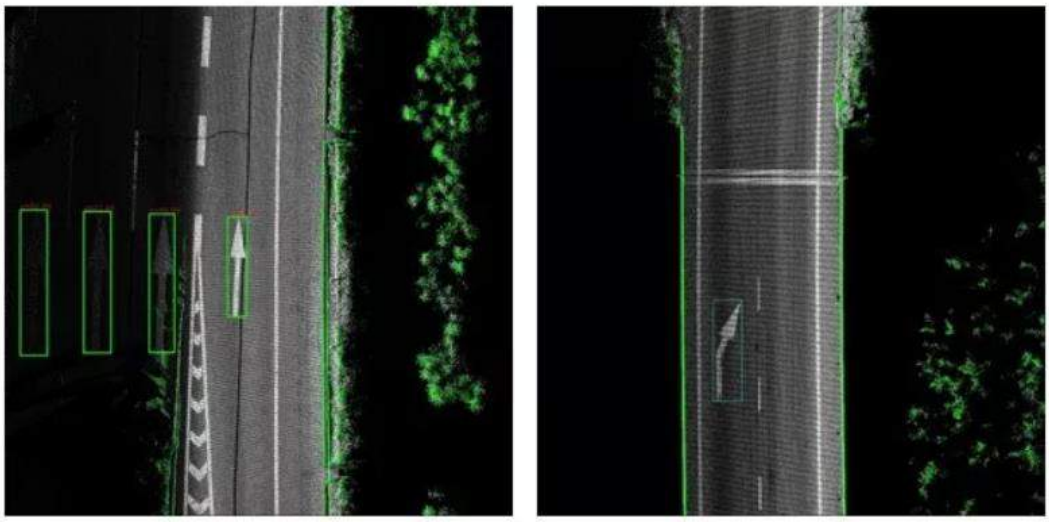


图 7. 级联检测算法识别示例

通过引入级联形式的检测识别模型令高精产线自动识别能力在识别精度上得到了不错的提升，但我们对自动识别位置精度提升的挖掘是无止尽的，所以有了以下的方案。

3）级联检测 + 局部回归

设想一下，如果我们在地面标识区域进行局部的位置回归，那么网络就能够聚焦到更加细微的地面标识区域，最终得到更加接近边界的位置。结合实际在做地面标志识别时，我们将容易造成精度问题的部分单独做位置精修，得到了更加精细的位置。

以下为部分算法识别结果示例：

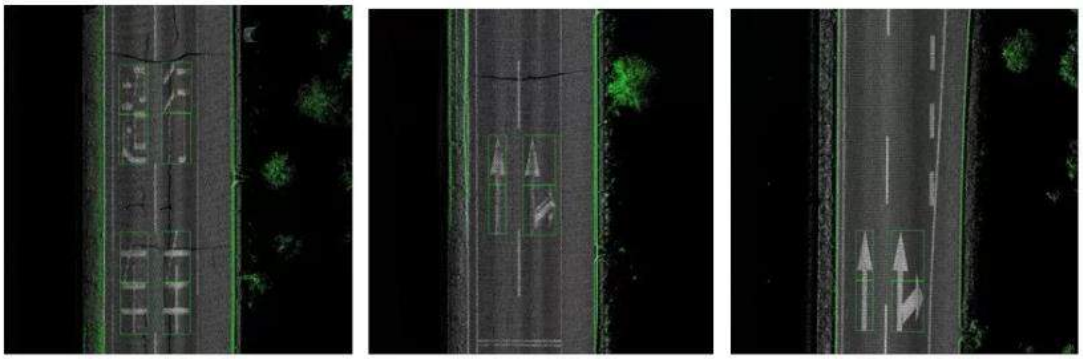


图 8. 算法识别示意图

采用检测+回归技术方案实现了更加好的位置检测精度，让我们离“真实世界”更进了一步。其缺点是技术方案流程较长，不够简洁美观。

4）基于角点的检测

基于角点回归的目标检测方法，使用单个卷积神经网络预测两组热力图来表示不同物体类别的角的位置，即将目标边界框检测为一对关键点（即边界框的左上角和右下角），以及每个检测到的角点的嵌入向量。其中角点用于确定目标位置，嵌入向量用于对属于同一目标的一对角点进行分组。

此种方法简化了网络的输出，通过将目标检测为成对关键点，消除了现有的检测器设计中对特征层需要大量 anchors 的弊端，因为大量 anchors 造成了大量的重叠以及正负样本不均衡。同时为了产生更紧密的边界框，网络还预测偏移以精细调整角点的位置。通过预测热力图、嵌入向量、以及偏移最终得到了精确的边界框。

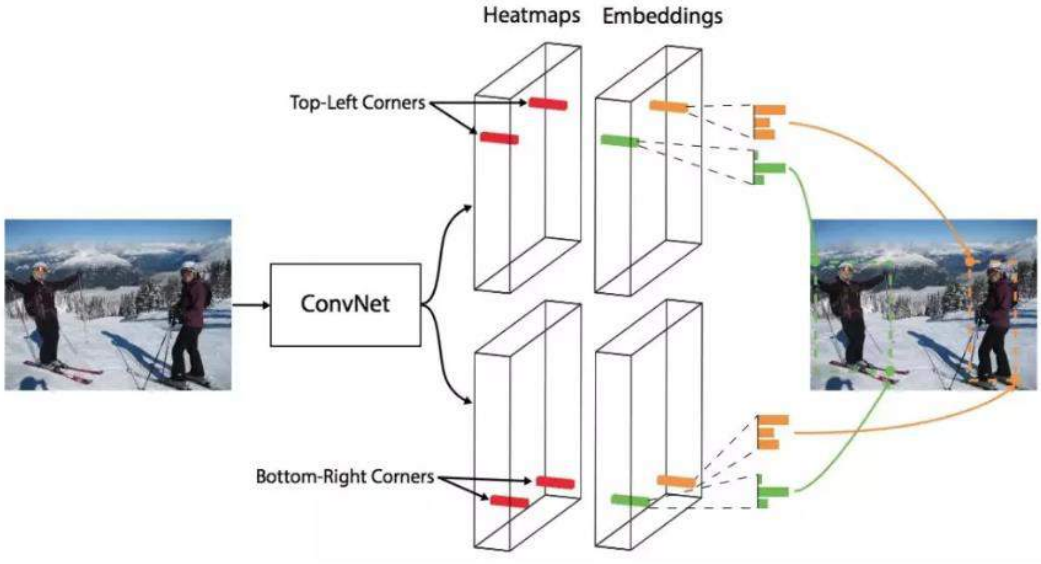


图 9 角点检测示意图

由于在检测任务中需要获取相同尺寸的特征图对目标进行位置回归、类别分类等，算法会对其进行量化以及降采样等操作，不可避免会有精度上的损失。这个弊端带来的最大影响就是经由检测回归出的位置不够鲁棒，在某些情况下会出现或多或少的偏移。

5）级联检测 + 分割精修

随着语意分割技术的不断成熟，基于深度学习的语意分割已经能够将输入图像进行像素级的分类，而且其精度也越来越高，也就是图片中要素的轮廓越来越精细。

我们采用以 resnet 为主干的分割模型，并结合了自适应感受野、多尺度融合、Coarse-Fine 融合、感兴趣区域注意力机制等技术实现了对地面标识的像素级分割。

为了获取地面标识的实体信息，我们仍然用检测来确定地面标识大致位置，但是不同的是最终由对应区域的地面标识分割语义信息获取最终精确的地面标识位置。

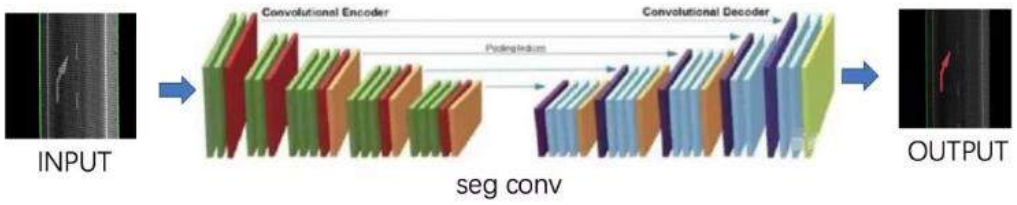


图 10.地面标识分割示意图

以下为部分检测结合精修示例图：

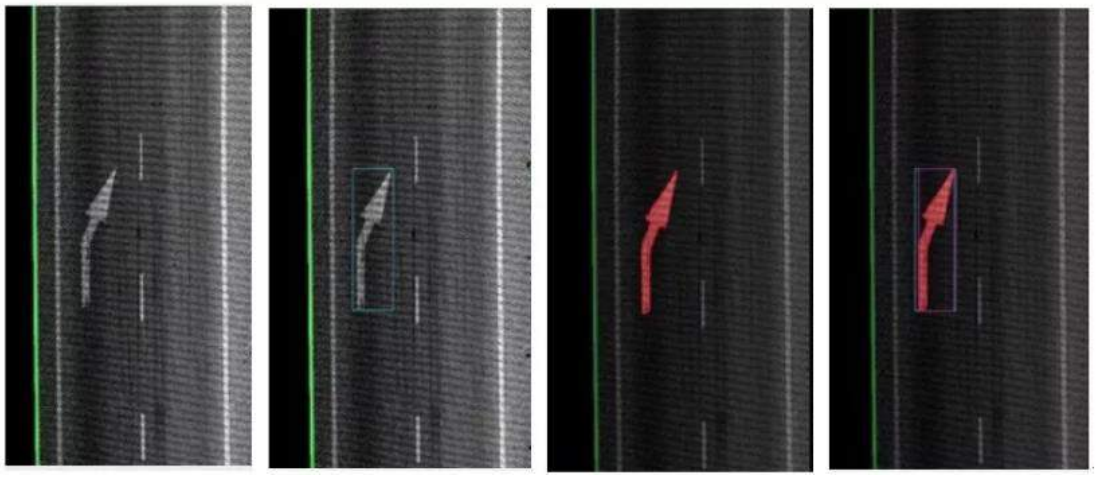


图 11. 分割精修示例

语义分割的引入使得地面标识的识别位置精度得到了改善，解决了由检测带来的识别位置精度不鲁棒的问题，使得高精地图地面标识自动化效果上了一个新的台阶。

但是这种方法稍显繁琐，而且检测和分割任务都需要耗费大量 GPU 资源，也就是说一张图片需要同时多次 GPU 运算加上后续的 CPU 后处理融合才能够得到最终的结果，如果能够将这些步骤优化，那么必然能够简化流程同时节省大量运算资源。

6）PAnet

基于以上考虑，我们采用了基于 PAnet 的检测识别算法。传统的实例分割模型各层中的信息传播不够充分。PAnet 较好的解决了这些问题，充分融合了 coarse、fine 特征，不仅有自顶向下的特征融合还结合了自底向上的特征融合，在高层特征中充分融合进了底层的强定位特征，解决了浅层特征信息丢失的问题。

另外还结合了自适应特征降采样将不同特征层进行融合提取 roi 特征做预测，以及添加额外 mask 前景背景分类分支，使得预测 mask 更加精确，这些手段结合对于目标检测位置精度有比较大的收益。同时，分割和检测任务结合能够互相促进取得更好的结果。

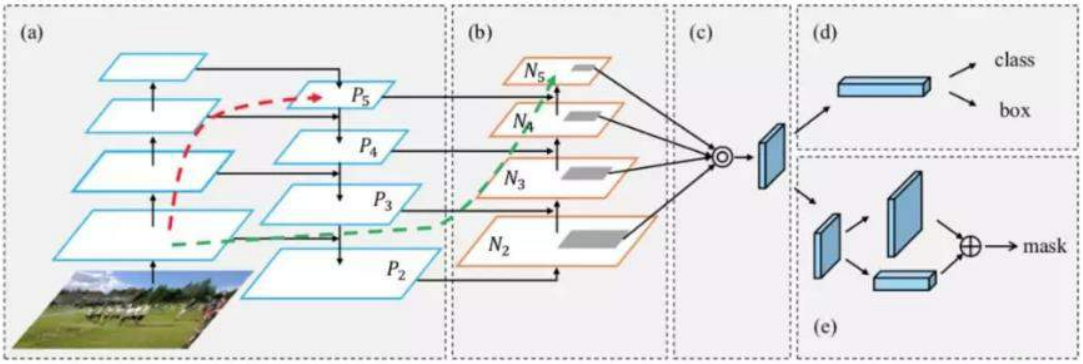


图 12 PAnet 示意图

以下为一些算法的识别结果示例。可以看到算法对部分磨损模糊的地面标识也有了一定的宽容度，其位置精度有了巨大的改善。（图中地面标识外框为检测得到的大概位置，内框为根

据像素级分割得到的位置，取内框为地面标识最终位置）。

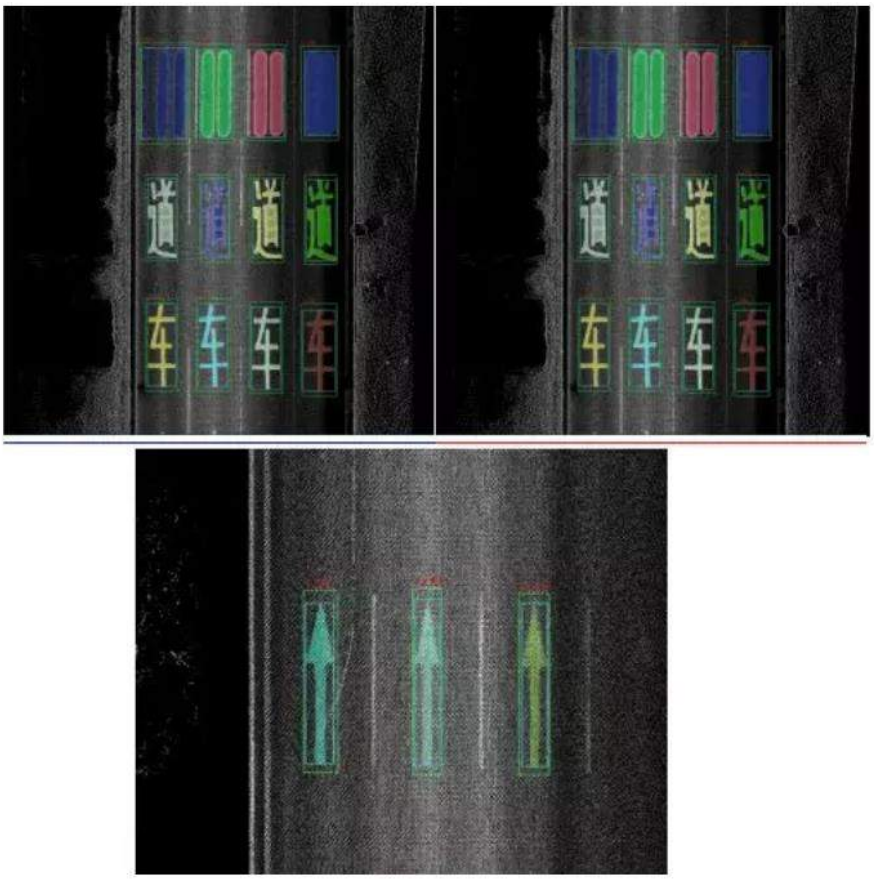


图 13 检测识别实例

采用上述方案需要将点云投影为 2D 空间，中间有一定的归一化量化操作，不可避免的会损失一些信息，最直观的是在一些点云反射率较低的地方容易造成目标丢失。如果能够在原始 3 维点云上提取那么这些问题就迎刃而解。

7）基于 3 维点云的目标检测

基于上面的考虑，我们探索原始点云上的 3D 物体检测，3D 点云识别是各种真实世界应用的一个重要组成部分，如自主导航、重建、VR/AR 等。与基于图像的检测相比，激光雷达提供可靠的深度信息，可以用于精确定位物体并表征它们的形状。

我们探索了多种 3 维点云识别算法，比如基于 bird-view、voxel 等的 3 维点云识别。由于 PointRCNN 在原始 3 维点云目标检测上的良好表现，我们采用基于 PointRCNN 的方法提取地面标识，整个检测框架包括两个阶段 :第一阶段将整个场景的点云分割为前景点和背景点，以自下而上的方式直接从点云生成少量高质量的 3D proposal。

第二阶段在规范坐标中修改候选区域获得最终的检测结果，将每个 proposal 经池化后转换为规范坐标，以便更好地学习局部空间特征，同时与第一阶段中全局语义特征相结合，用于预测 Box 优化和置信度预测。

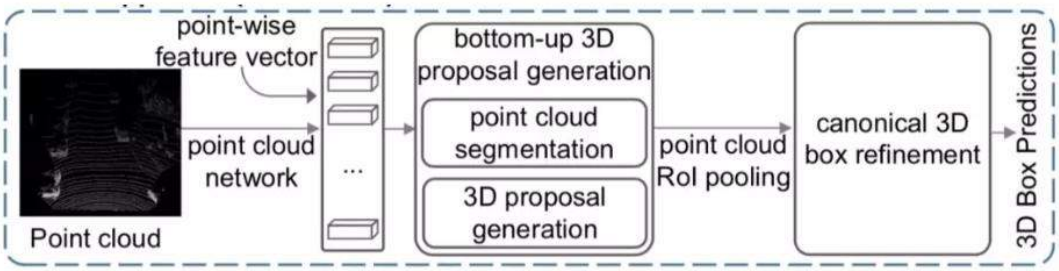


图 14 3 维点云检测

4. 效果与收益

大数据的支撑使得我们的算法拥有更好的鲁棒性与识别能力。结合算法中各种策略以及多种数据源（点云、可见光等），我们在不断提升地面标识识别精度，其位置精度在 Ground Truth 5cm 范围区间内达到 99%以上，召回也达到了 99.99%以上，各项指标都得到了稳步提升。

上述方案已经正式上线，并处理了大量数据，准召率都达到了生产作业的要求，同时算法对人工作业产线的效率提升作用日益提高。以下是部分效果图：

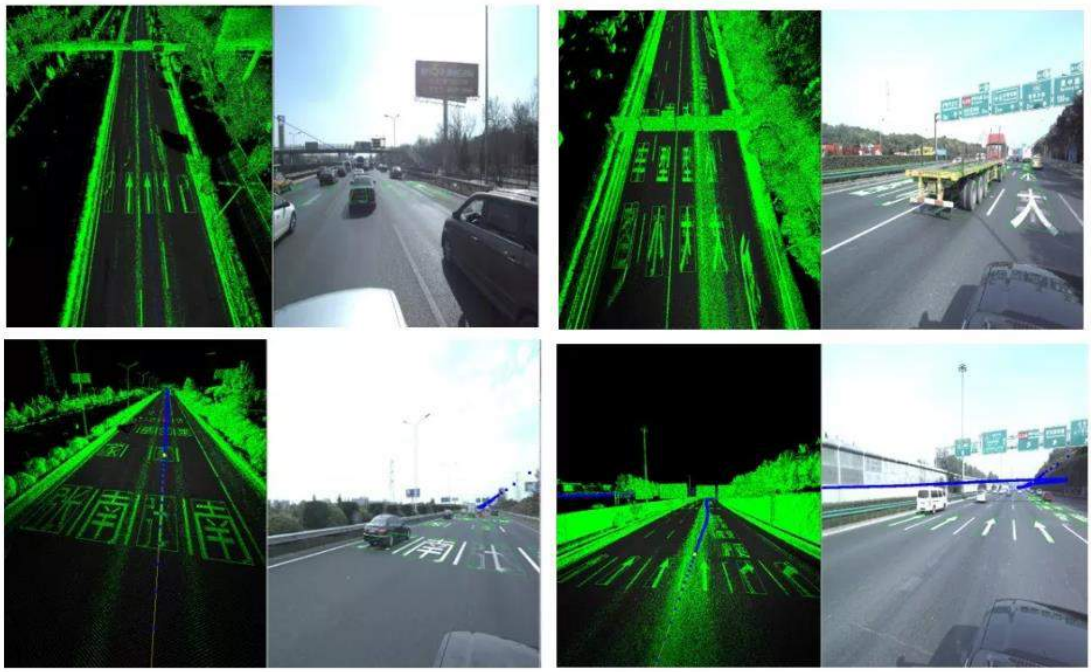


图 15. 地面标识检测效果图

5. 写在最后

高精地图被称作自动驾驶系统的“眼睛“，与普通地图最大的不同点在于使用主体不同。普通导航地图的使用者是人，用于导航、搜索，而高精地图的使用者是计算机，用于高精度定位、辅助环境感知、规划与决策。因而高精地图对地图要素不仅需要极高的召回率，还需要非常高的位置精度。

高精地图中要素的识别对技术提出了比较高的要求，纵观整个高精地图产业发展，地图制作逐渐从纯人工过渡到半自动乃至全自动。期间识别技术也不断得到发展与完善，从手动构造特征到自动特征、从 2 维识别到 3 维以及更高维识别、从单源识别到多源融合等。

目前，高精地图多采用人工作业，人工作业质量和效率始终是一个矛盾点，相比之下，机器自动识别有着更高的效率、更低的作业成本以及不亚于人工的作业质量。自动识别的应用必将加速高精地图构建，推动高精地图产业发展。高精度地面标识识别技术已经在高德高精地图内部得到应用，有效提升了数据制作效率与制作质量，为高德构建高精地图提供坚实的技术支撑。

招聘

高德地图视觉攻坚小组火热招聘中：计算机视觉、图像识别、点云、三维、AR、视频等算法类职位正在向你招手~~职位地点：北京，西雅图，湾区。简历投递到 ruby.JL@alibaba-inc.com

基于深度学习的图像分割在高德的实践

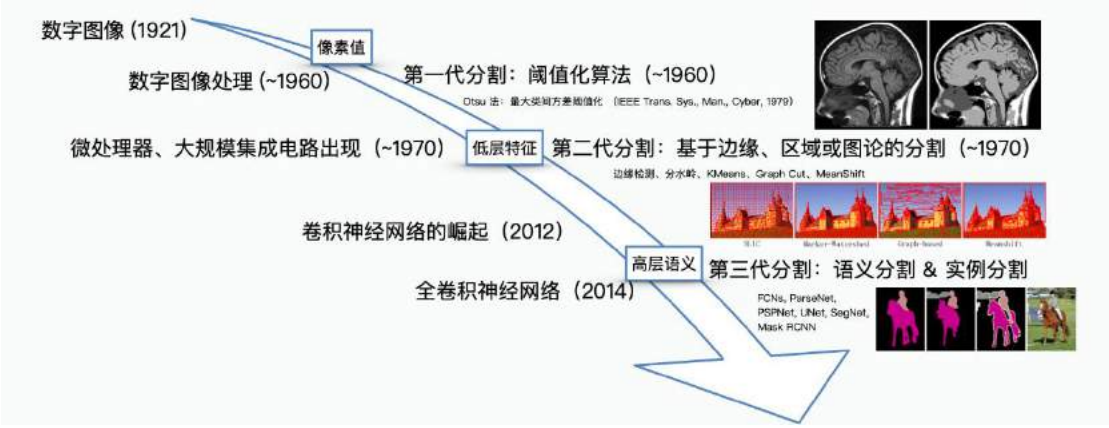
作者：匀瀚

1.前言

图像分割（Image Segmentation）是计算机视觉领域中的一项重要基础技术，是图像理解中的重要一环。图像分割是将数字图像细分为多个图像子区域的过程，通过简化或改变图像的表示形式，让图像能够更加容易被理解。更简单地说，图像分割就是为数字图像中的每一个像素附加标签，使得具有相同标签的像素具有某种共同的视觉特性。

图像分割技术自 60 年代数字图像处理诞生开始便有了研究，随着近年来深度学习研究的逐步深入，图像分割技术也随之有了巨大的发展。早期的图像分割算法不能很好地分割一些具有抽象语义的目标，比如文字、动物、行人、车辆。这是因为早期的图像分割算法基于简单的像素值或一些低层的特征，如边缘、纹理等，人工设计的一些描述很难准确描述这些语义，这一经典问题被称之为“语义鸿沟”。

得益于深度学习能够“自动学习特征”的这一特点，第三代图像分割很好地避免了人工设计特征带来的“语义鸿沟”，从最初只能基于像素值以及低层特征进行分割，到现在能够完成一些根据高层语义的分割需求。



（图像分割的发展历史）

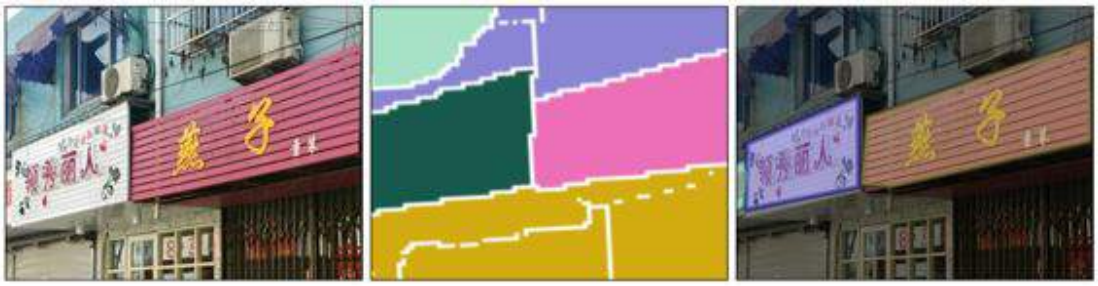
高德地图拥有图像/视频大数据，在众多业务场景上都需要理解图像中的内容。例如，在数据的自动化生产中，通常需要寻找文字、路面、房屋、桥梁、指示牌、路面标线等目标。这些数据里有些是通过采集车辆或卫星拍摄，也有些数据则是通过用户手机拍摄，如下图所示：

复杂多样的图像/视频数据

面对这些场景语义复杂、内容差异巨大的图像，高德是如何通过图像分割对其进行理解的？本文介绍了图像分割在高德地图从解决一些小问题的“手段”，逐步成长为高度自动化数据生产线的强大技术助力。

2.探索期：一些早期的尝试

在街边的数据采集中，我们需要自动化生产出采集到的小区、店铺等 POI（Point of Interest）数据。我们通过 OCR 算法识别其中文字，但苦恼于无法确定采集图像中到底有几个 POI。例如，下图中“领秀丽人”与“燕子童装”两家店铺，人眼可以很容易区分，但是对于机器则不然。一些简单的策略，比如背景颜色，容易带来很多的错误。



商户挂牌检测的逐步迭代

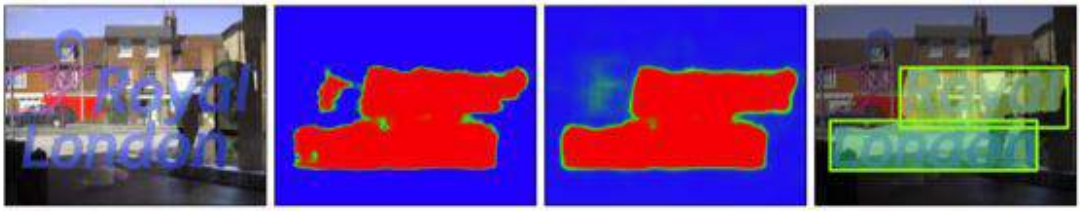
例如，遇到两个样式十分相近的挂牌的时候，我们利用无监督的 gPb-owt-ucm 算法 [1] 在检测多级轮廓的基础上结合改进的分水岭算法将图像切分为多个区域，并利用 Cascade Boosting 的文字检测结果将图中带有文字的区域进行了分割。

3.成长期：自然场景下的语义分割

于 2014 年底问世的全卷积神经网络 [2]（FCNs, Fully Convolutional Networks）无疑是继 2012 年问鼎 ImageNet 大赛以来深度学习发展的又一里程碑。FCNs 提供了第一个端到端式的深度学习图像分割解决方案。FCNs 在 CNN 的基础上可以从任意尺寸的输入进行逐像素的分类。我们也在第一时间将其落地到高德自身的应用场景中，例如文字区域的分割。自然场景下的文字由于其背景、光照复杂，文字朝向、字体多样，使得人工构建特征十分困难。

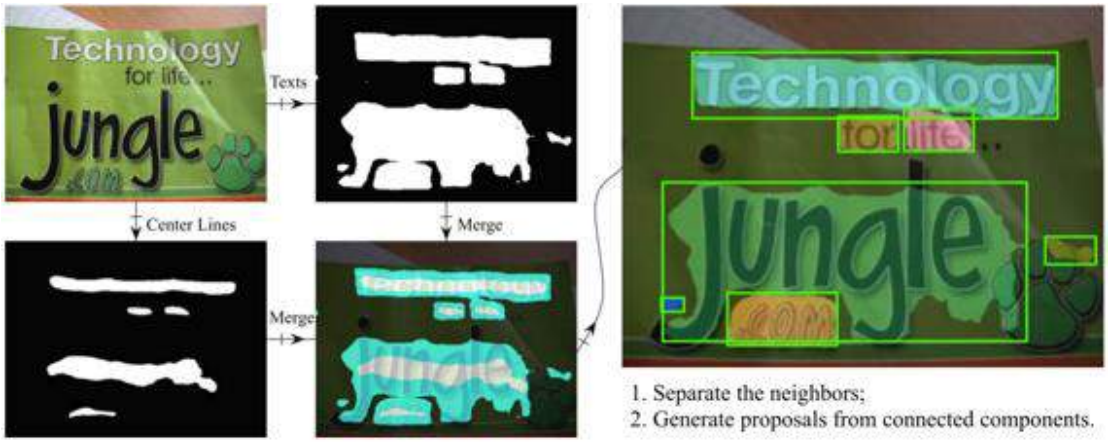
很快地，我们发现 FCNs 还并不能很好地满足我们的需求。虽然 FCNs 在解决语义鸿沟问题上提供了解决方案，但在一般情况下只能给出一个“粗糙”的区域分割结果，不能实现很好

的“实例分割”，对于目标虚警、目标粘连、目标多尺度、边缘精度等问题上也没有很好地解决。一个典型的例子就是在分割文字区域时，“挨得近”的文字区域特别容易粘在一起，导致在计算图像中的文本行数时造成计数错误。



基于 FCNs 的图像区域分割

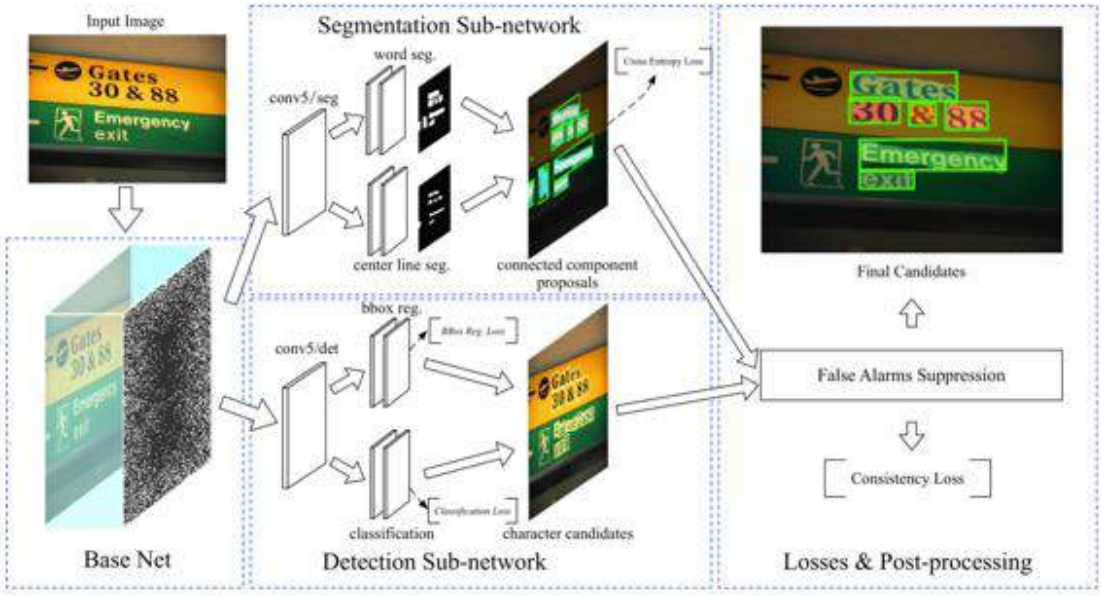
因此，我们提出了一个多任务网络来实现自己的实例分割框架。针对目标粘连的问题，我们在原始网络中追加了一个分割任务，其目标是分割出每个文本行的“中轴线”，然后通过中轴线区域来拆分粘连的文本行区域。拆分的方法则是一个类似于 Dijkstra 的算法求解每个文本区域像素到区域内中轴线的距离，并以最短距离的中轴线作为像素归属。



实例分割的尝试：通过中轴线区分多个不同文本行

另外一个比较困扰的问题是 FCNs 结果中的虚警，即非文字区域被分割为文字区域。虽然相较于一些传统方法，FCNs 结果中的虚警已经少了很多，但为了达到更好的分割正确率，我们在原有网络基础上增加了一个并行的 R-CNN 子网络进行文字的检测，并利用这些检测结果抑制虚警的产生（False Alarms Suppression）。

为了通过端到端的学习使得网络达到更好的效果，我们设计了一个一致性损失函数（Consistency Loss Function），来保证网络主干下分割子网络和检测子网络能够相互指导、调优。从优化后分割网络输出的能量图可以看到，虚警的概率明显降低了。若想要了解详细细节，可以参考我们 17 年公布在 arxiv 上的文章[3]。



高德自然场景文字区域分割/检测网络

4.成熟期：分割的精细化与实例化

得益于 Mask R-CNN 框架 [4] 的提出，实例化的图像分割变得更加容易。以之前提到的商户挂牌的分割为例，挂牌区域的分割也十分容易出现粘连，且挂牌样式多样，不存在文本行这样明显的“中轴线”。目标检测方法可以对提取挂牌的外包矩形。但问题在于，自然场景下挂牌的拍摄往往存在非垂直视角，因此在图像上并不是一个矩形，通常的检测算法则会带来不准确的边缘估计。Mask R-CNN 通过良好地整合检测与分割两个分支，实现了通用的实例化图像分割框架。其中目标检测分支通过 RPN 提取目标区域，并对其进行分类实现目标的实例化；然后在这些目标区域中进行分割，从而提取出精准的边缘。

一些更加复杂的场景理解需求，也对图像分割分割精细程度提出了更高的要求。这主要体现在两个方面：（1）边缘的准确度（2）不同尺度目标的召回能力。

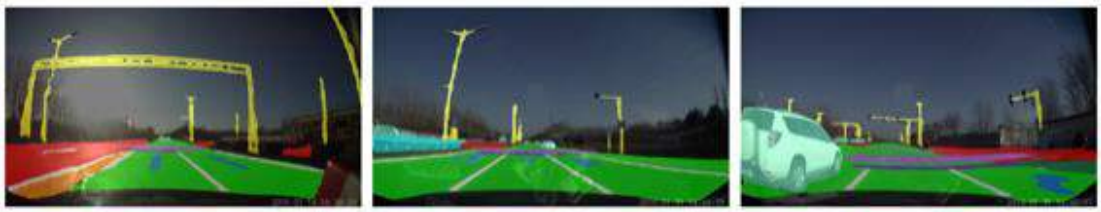
在高精地图的数据数据生产需要分割出图像中的路面，然而高精地图对于精度的要求在厘米级，换算到图像上误差仅在 1~2 个像素点。观察原始分割的结果不难发现，分割的不准确位置一般都是出现在区域边缘上，区域内部是比较容易学习的。

因此，我们设计了一个特殊的损失函数，人为地增大真值边缘区域产生的惩罚性误差，从而加强对边缘的学习效果，如图所示，左侧为可行驶路面区域分割，右侧是路面及地面标线分割。

高精地图下可行驶区域分割　　路面及地面标线

高精地图路面的分割

道路场景下需要理解的目标种类繁多，一方面其本身有大有小，另一方面由于拍摄的景深变化，呈现在图像上的尺度也大小各异。特别的是，有些特殊目标，例如灯杆、车道线等目标是“细长”的，在图像上具有较大长度，但宽度很小。这些目标的特性都使得精细的图像分割变得困难。



道路场景下的全要素图像分割

首先，由于受到网络感受野的限制，过大和过小的目标都不容易准确分割，比如道路场景下的路面与灯杆，卫星影像中的道路与建筑群。针对该问题，目前的 PSPNet [5], DeepLab [6], FPN [7] 等网络结构都能在不同程度上解决。

其次，由于目标尺度不同，导致分割网络样本数量的比例极不均衡（每一个像素可以认为是一个样本），我们将原先用于目标检测任务的 Focal Loss [8] 迁移到图像分割网络中来。Focal Loss 的特点在于可以让误差集中在训练的不好的数据上。这一特性使得难以学习的小尺度目标能够被更加准确地分割出来。

多尺度目标的分割

5.未来的展望

图像分割技术目前朝着越来越精确的方向上发展，例如 Mask Scoring R-CNN [9]、Hybrid Task Cascade [10] 的提出，在 Mask R-CNN 的基础上持续优化了其分割的精确程度。然而站在应用角度，基于深度学习的图像分割相较于当量的分类任务则显得“笨重”。

出于图像分割任务对精度的要求，输入图像不会像分类任务一样被压缩至一个很小的尺寸，带来的则是计算量的指数级增加，使得图像分割任务的实时性较难保证。针对这个问题，ICNet, Mobile 等网络结构通过快速下采样减少了卷积初期的计算量，但也带来了效果上的折损。基于知识蒸馏（Knowledge Distillation）的训练方法，则像个更好的优化方案，通过大网络指导小网络学习，使得小网络的训练效果优于单独训练。知识蒸馏在训练过程中规避了网络剪枝所需要的经验与技巧，直接使用较低开销的小网络完成原先只能大网络实现的复杂任务。

对于高德地图来说，图像分割已经是一个不可或缺的基础技术，并在各个数据自动化生产线中得到了广泛应用，助力高德地图的高度自动化数据生产。未来，我们也将持续在地图应用场景下打造更加精准、轻量的图像分割技术方案。

6.参考文献

[1] Arbelaez, Pablo, et al. "Contour detection and hierarchical image segmentation." IEEE transactions on pattern analysis and machine intelligence 33.5 (2010): 898-916.

[2] Long, Jonathan, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[3] Jiang, Fan, Zhihui Hao, and Xinran Liu. "Deep scene text detection with connected component proposals." arXiv preprint arXiv:1708.05133 (2017).

[4] He, Kaiming, et al. "Mask r-cnn." Proceedings of the IEEE international conference on computer vision. 2017.

[5] Zhao, Hengshuang, et al. "Pyramid scene parsing network." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[6] Chen, Liang-Chieh, et al. "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs." IEEE transactions on pattern analysis and machine intelligence 40.4 (2017): 834-848.

[7] Lin, Tsung-Yi, et al. "Feature pyramid networks for object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[8] Lin, Tsung-Yi, et al. "Focal loss for dense object detection." Proceedings of the IEEE

international conference on computer vision. 2017.

[9] Huang, Zhaojin, et al. "Mask scoring r-cnn." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.

[10] Chen, Kai, et al. "Hybrid task cascade for instance segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.

招聘

高德地图视觉攻坚小组火热招聘中：计算机视觉、图像识别、点云、三维、AR、视频等算法类职位正在向你招手~~职位地点：北京，西雅图，湾区。简历投递到 ruby.JL@alibaba-inc.com

前端&移动篇

高德客户端及引擎技术架构演进与思考

作者：无鄙

2019 杭州云栖大会上，高德地图技术团队向与会者分享了包括视觉与机器智能、路线规划、场景化/精细化定位、时空数据应用、亿级流量架构演进等多个出行技术领域的热门话题。现场火爆，听众反响强烈。

阿里巴巴高级无线开发专家宋照春在高德技术专场做了题为《高德客户端及引擎技术架构演进与思考》的演讲，主要分享了高德地图客户端技术架构沿着「上漂下沉」、「模块化、Bundle 化」的思路演进所做的一系列架构升级中的经验和思考。

以下为宋照春演讲内容的简版实录：

主要分享三个方面的内容：

- 融合
- 架构治理
- 动态化

1.三管齐下 深度融合

高德最初有两个端，车机版的高德导航，手机版的高德地图，两个团队，一个是 2B，一个是 2C，分别是汽车业务和手机业务。当时在引擎/技术上，分为离线引擎和在线引擎，但两个团队之间交流比较少，各自有自己的研发、产品和测试，而作为一款端上的 APP，两块业务都需要有地图渲染、路线规划、导航以及定位等通用能力。从公司层面看，存在较大的重复建设，整体研发效率较低。

于是我们做了一件事：利用技术手段，打通端上引擎，打造一套能同时支撑多端的 APP 能力。具体到执行层面，先从 A 团队拉一部分人到 B 团队一起建设，建设完之后再从 B 团队

拉到 A 团队。在同时支撑好主线业务发展的情况下，通过一年左右时间，完成了引擎上的融合，做到同时支撑手机、车机以及开放平台。

这样就从引擎的维度，实现了渲染、定位、规划和引导的统一。具体来说，我们的各大引擎有好多套代码，好几个开发团队，每个团队有各自的开发方式和开发环境（Linux, Windows, Mac OS）。各种开发环境，工程配置文件大量重复，修改非常繁琐。

为此，我们通过两种方法：

- 建立了一套构建系统 Abtor，通过一个配置系统实现统一构建，能够同时支持多个子引擎，在构建集成效率上得到了很大的提升。

- 对基础库进行了整体重构，形成了一套涵盖了文件 I/O、KV 存储、多线程框架&异步框架、归档、基础容器等一系列标准能力的基础库，同时也做了引擎核心架构的统一。

2.架构治理

通过引擎的融合同时支持多端，在研发效率上实现比较大的收益。而通过技术的抓手来实现团队的融合，对公司发展而言，这其实是更大的收益，团队融合的意义在于人才拉通和复用，组织效率得到了较大提升。
随着高德业务的快速发展，业务上持续扩品类，需求量激增，高德地图从最初的驾车导航，到后来的步行、骑行、摩托车导航等等，App 所承载的业务发展非常快，而原有的架构治理

模式的问题也逐渐暴露出来。

首先就是 App 的代码规模变得特别大。当时一个仓库达到了 10G 以上，由此导致的一个典型的问题就是**编译慢**，编译出一次安装包需要一个小时。伴随代码规模的另一个问题是团队规模快速增长。代码增长和大团队并行开发，最终导致**合版慢**，每次迭代，客户端合版需要 2 天。

代码膨胀导致的架构腐化问题特别突出，所以测试质量以及线上的质量有段时间也比较差。此外，从产品提出需求到上线，平均需要 45 天，版本迭代周期很长。

为解决以上架构问题，我们采取了三个手段：**升级 Native 基础组件**，**搭建 Native 容器和页面框架**，**Bundle 化分拆（微应用）**。

下面重点介绍下页面框架和微应用。

页面框架主要借鉴和融合了 Android 和 iOS 的生命期管理机制。从高德地图 App 架构看，下层模块是一套标准地图，所有上层业务都要基于地图模块开发。为确保上层业务低耦合、一致性，我们设计了一个页面框架。

如上图，左边的 Activity 是 Android 的系统页面控制器，右边的 UIViewController 是 iOS 的系统页面控制器，通过虚线连接比较，我们发现两端的页面状态设计基本相同。

所以，我们在设计自己的页面框架时沿用了这些系统页面状态，同时从命名上也保持一致，这样可以让 Android 和 iOS 原生开发的同学更容易理解和上手。

我们吸取了双端各自的优点。比如，Android 端页面有四种启动模式，但是 iOS 端并没有这些，我们就把 Android 的四种启动模式运用到了 iOS 端；iOS 端有 Present 特性，但是 Android 端没有，那么也把这种特性融合到 Android 端的页面框架中；最后，还有一些小设计，比如 Android 的 onResult 设计，也可以借鉴融合到 iOS 端。

此外，我们还做了微应用，所谓微应用，首先是模块化，就是把大模块仓库大模块拆成一个

个小的 Bundle，除了实现模块化，还主要实现以下几个目标：

- 粒度：以业务为单位，以业务线为分组
- 编译：二进制级别的产物，可独立编译、出包时链接
- 依赖：松耦合，以“服务”为导向，不关心模块归属

而 Native 容器层面，要实现四个核心目标：路由管理、服务管理、UI 生命期管理、微应用管理。

通过一年时间的 Bundle 化改造，高德地图单端 App 完成了 300 多个页面的建设，拆分了 100 多个 Bundle。

从收益来看，总编译时间从原来的 60 分钟降低到了 8 分钟，合版周期从原来的 3 天降到 1 天，需求上线周期降到了 1 个月以内，线上质量和测试质量都得到了极大的提升，崩溃率从万分之八降低到十万分之八。

3.动态化

随着高德地图业务发展沿着扩品类、在垂直品类做精做细，景区、酒店、银行商铺、充电桩等个性化定制需求凸显，对前端展现提出了更高的要求，对“快速应变”要求也更高了。

实际上，在 2015 年，高德就开始做动态化。最早的时候业内就有 React Native，团队做了技术调研，发现不能完全满足业务上的需要，尤其是性能方面。最后我们决定自研一套动态化技术。

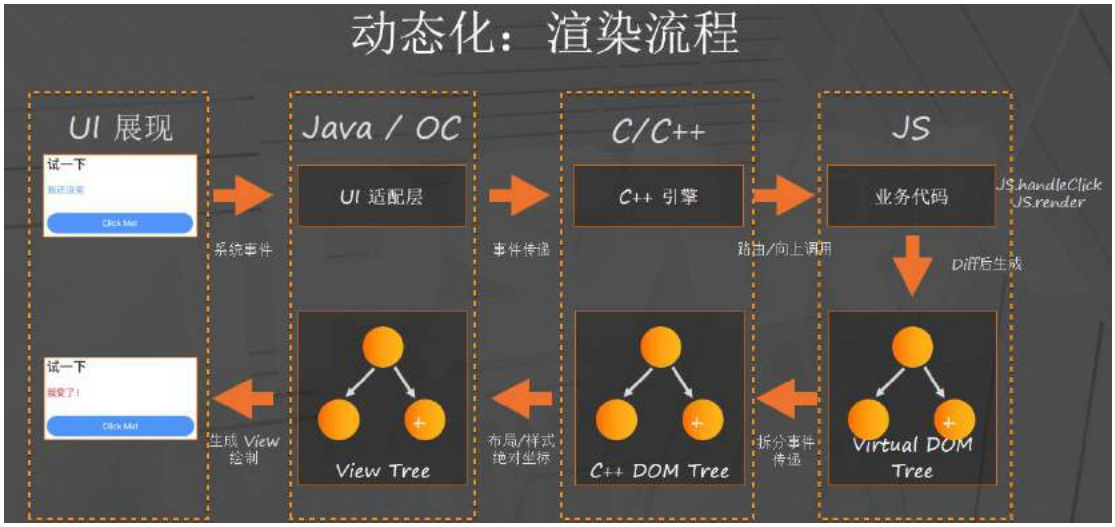
具体来说，就是通过一个核心 C++引擎，把两端业务（Android、iOS）用一套 JavaScript 代码解决，实现双端归一，Android 实现业务动态化发布。

架构层面，最下面是高德 App 核心的地图引擎，我们在上面搭建了一套动态化应用引擎，通过 C++来实现。应用引擎的作用是为了承上启下，上面承载动态化业务，下层完成地图引擎的直接打通。众所周知，GUI 的核心是 DOM 树，所以应用引擎不但要实现和 JavaScript 引擎的整合，还要负责 DOM 树的核心逻辑计算。

其次，动态化的技术和前端 Web 技术一致：样式、布局。应用引擎负责完成样式的布局计算、DOM 树 Diff、事件生成。而 GUI 的绘制，通过 Diff 事件，交由原生的 Android 以及 iOS 去完成。这样，所有的 GUI 都是原生的组件。

在之上，我们搭建了一套前端框架，前端框架采用当前前端响应式框架做，前端框架之上又搭建了一套前端的 UI 卡片库和 UI 组件库，让上层业务能够更高效的开发。

而对于一些通过动态化的技术无法实现，或者性能上存在卡点的功能，我们就通过 Native 扩展能力来支撑，这样，完整的动态化的业务能够直接运行在 Android 以及 iOS 上。



JS 去执行代码之后，前端框架会产生虚拟的 DOM 树，最后提交到 C++引擎，形成 C++的 DOM 树。C++引擎去完成布局、样式计算，Diff 计算，将每个节点的属性和坐标交给 Android 以及 iOS，由 Native 来完成最终 UI 的渲染。

总体来说，动态化的特点 :首先是它**与主流前端框架融合**，充分融合了大前端的生态 ;第二，**性能、扩展性较好**。因为采用 C++实现整个核心逻辑，静态和动态的语言绑定技术，能够保证地图引擎的能力能够直接透出到上层，或者从上层能够直接 call 底层的 C++能力 ;第三，**多端归一和动态化**，充分利用 Native 优势，接近原生 Native 体验。

动态化技术改造完成之后，双端不一致的问题降低了 90%，开发、测试成本降低 30%，发版周期从 T+30 到 T+0。

最后，总结下高德客户端及引擎技术架构演进的几个重要阶段：第一个阶段，通过在线&离线引擎的融合拉通，让高德最核心的导航能力提到提升；第二阶段，在客户端发展成为“巨型”APP，代码量发展到超大规模的时候，通过架构治理，满足业务快速增长的诉求，解决大规模业务体量下的架构合理性问题，消除架构瓶颈；第三个阶段通过动态化的技术，实现多端归一，以及动态发版能力，为业务发展提供更大的助力。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

高德地图：崩溃率从万分之 8 降到十万分之 8 的架构奥秘

嘉宾：郝仁杰（本文由 infoQ 根据速记整理而成）

近几年来，高德地图业务发展迅猛，团队规模迅速扩张，代码体量急剧增加，为了提高团队的高效并行作战能力，端上做了一系列的架构升级。2018 年通过双端融合、组件化、研发平台搭建等技术实践，使得发版效率提升 50%， App 崩溃率从万分之八降到十万分之八。本文整理自 ArchSummit 全球架构师峰会（深圳站）2019 峰会演讲，主要分享在一系列架构升级改进中，高德地图的具体做法、经验和思考。

以下为演讲速记整理内容：

大家好，我是来自高德地图的郝仁杰，本次分享的主题是“**高德地图 App 架构演化与实践**”。2018 年，我们通过架构的演进将发版周期缩短了一半，整个 App 的崩溃率从万分之八降低至十万分之八。在正式开始介绍之前，我先简单介绍下高德。

背景介绍

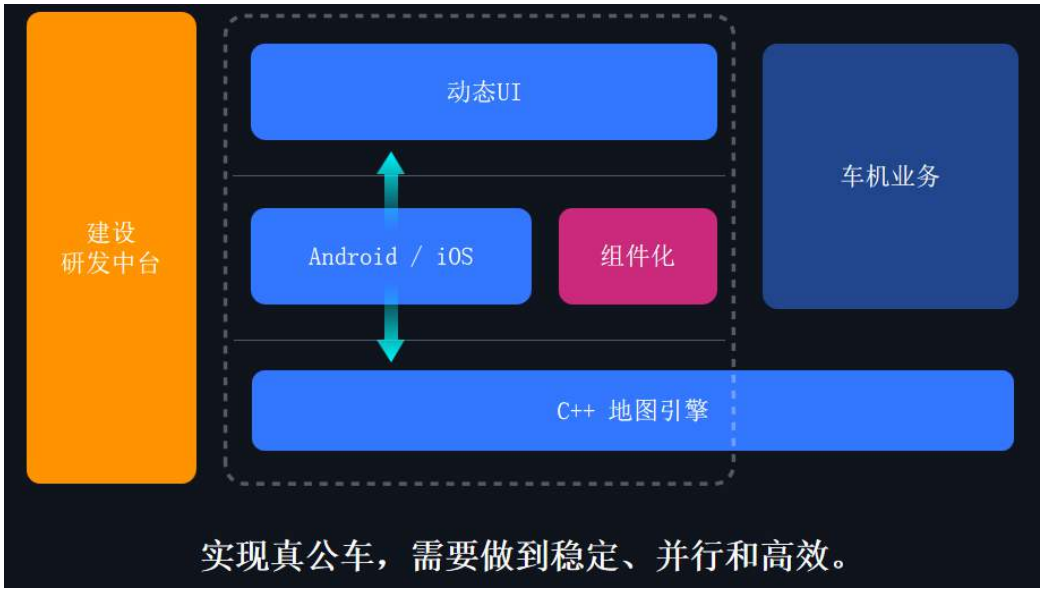
高德是国内领先的数字地图内容、导航和位置服务解决方案提供商。目前在端上，分为手机和车机两条主线。近年来，高德业务迅猛发展，人员规模迅速扩张，代码体量急剧增加，为了提高团队高效并行作战的能力，端上做了 C++ 多端融合和动态化能力建设。

回顾近几来高德地图 App 架构的演进历程：2014 年，手机端上只有几十个研发，Android 和 iOS 端由原生单体架构实现；2015 年，地图引擎下沉 C++，实现了手机和车机的多端融合；2016 年，端上启动了动态 UI 框架的开发，为未来业务的动态化铺路；2017 年，动态 UI 框架建设完成，具备了运行静态页面的能力；到了 2018 年，手机端已经成长为拥有数百研发规模的团队，双端代码量也已经达到了数百万行，架构要如何继续演化来提高团队高效并行作战的能力，来支撑并赋能业务快速发展呢？

问题现状

为了让业务开发有节奏的进行，项目上每年会制定一些公车计划。公车就是每个 App 版本，版本里带的产品功能就是公车上的货物，公车计划即每年的发版计划。按照计划，公车会在指定的时间把组装好的货物拉走。

2018 年初，由于双端代码差异较大、耦合严重、复用率低、职责不清晰、平台工具简陋等问题，公车无法按照计划将拉走货物。工具落后，货物组装慢且质量差，无法如期交货，迫使公车等待，导致整个发版周期长达 3 个月，崩溃率高达万分之八，公车变成了伪公车。



为了解决这些问题，使伪公车变为真公车，需要做到稳定、并行和高效。端上通过以下三种方式达到该目的，一是双端融合，如上图，蓝色部分上漂动态 UI，下沉 C++，以及 Android、iOS 双端拉齐，减少差异，提高可维护性；二是选择组件化方案，分而治之，解除耦合，提高复用率，做到并行、高效；三是搭建研发中台，工具升级，流程自动化以及风险质量管控，提升效率和稳定性。

执行方案

双端融合

2015 年，我们通过地图引擎下沉 C++，实现了手机、车机的多端融合，同理，可将部分功能下沉 C++；通过 2017 年建成的动态 UI 框架，可将部分业务上漂到动态 UI；对于既不能上漂也不能下沉的，通过双端拉齐做到融合。

那么，什么样的场景适合下沉到 C++ 呢？一，需要有稳定的逻辑，不经常变化；二是不强依赖原生；三，对性能要求较高。举例来说，导航逻辑，地图从开始建立到现在已经打磨出一套非常核心稳固的逻辑，这部分逻辑可以下沉到 C++。

哪些场景又适合上漂到动态 UI 呢？一，对性能要求不高；二，经常易变的业务代码，比如产品的 UI 需求；三，不强依赖于原生能力。

对于既不能下沉，也不能上漂的功能，选择双端拉齐：对性能有一定要求；强依赖原生的能力；需要支撑一些原生业务。例如，高德地图的页面框架，虽然 Android 和 iOS 端有原生的页面框架，但地图类应用和普通应用不太一样，地图类应用的主要功能都是围绕着一张地图进行，这张地图上面的元素非常丰富，数据量非常庞大，内存占用较大，如果采用原生页面框架进行开发，就意味着每切换一个页面就得创建一张新地图，这对手机端这种资源紧缺的环境来说是非常浪费的，对于低端机型来说是不可接受的。

另外，地图应用从一个页面切换到另一个页面，或者从一个场景切换到另一个场景，并不

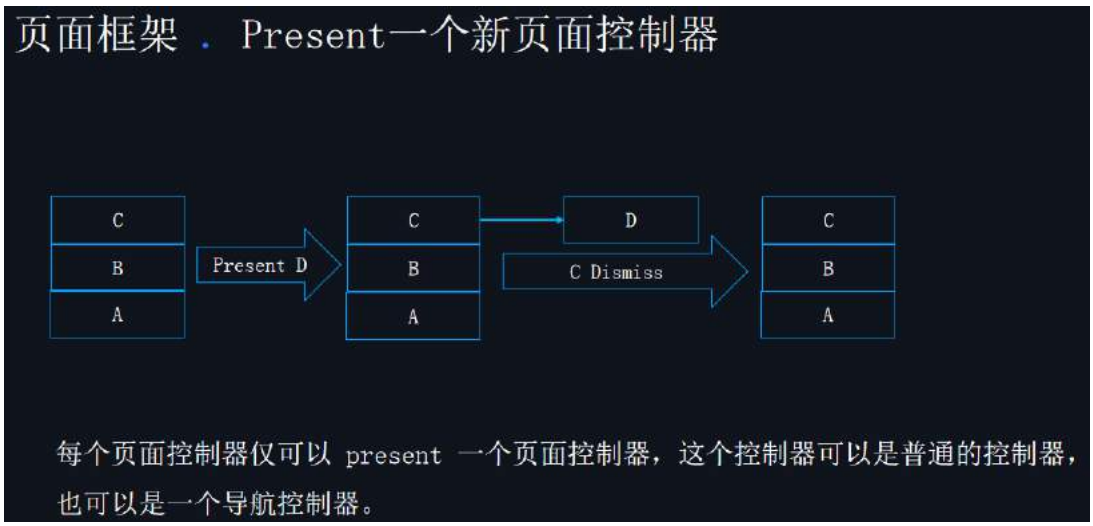
是完全不同的两张图切换，而仅仅是一张地图的不同状态转换，此时，如果额外创建一张新的地图，显然是极大的浪费，所以，对于地图类应用，我们建设了自己的页面框架：以单系统页面控制器多视图切换的方式实现。由于原来都是单体开发，Android 和 iOS 只关注自身特性，两边的实现不太一样，跳转规则、功能特性均有差异，我们通过分析双端的规则、特性，借鉴双端各自的优点，设计了一套统一地规则、特性，实现了双端的融合。

下面简单介绍下高德地图页面框架的融合方案：

如上图，左边的 Activity 是 Android 的系统页面控制器，右边的 UIViewController 是 iOS 的系统页面控制器，通过虚线连接比较，我们发现两端的页面状态设计基本相同。所以，我们在设计自己的页面框架时沿用了这些系统页面状态，同时从命名上也保持一致，这样可以让 Android 和 iOS 原生开发的同学更容易理解和上手。

此外，我们吸取了双端各自的优点。比如，Android 端页面有四种启动模式，但是 iOS 端并没有这些，我们就把 Android 的四种启动模式运用到了 iOS 端；iOS 端有 Present 特性，但是 Android 端没有，那么也把这种特性融合到 Android 端的页面框架中；最后，还有一些小设计，比如 Android 的 onResult 设计，也可以借鉴融合到 iOS 端。

首先，介绍下四种启动模式，这是安卓特有的。第一种是 Standard 模式，这个模式和栈的行为是一样的，就是标准的 Push 和 Pop；第二种是 SingleTop 模式，当向一个页面栈压入另一个页面时，如果该页面已经在栈顶，那么将不会创建一个新的页面实例 C 放到栈顶，不会变成 ABCC 这样的方式，而仅仅是通知当前栈顶的 C 页面做一个数据更新；第三种是 SingleInstance 模式，当以 SingleInstance 模式 Push 一个页面时，如果该页面已经在栈中，那么就把它从栈中带到栈顶；最后一种是 SingleTask 的模式，这和原生系统略有差别，因为我们目前是基于单页面控制器的方式实现的，当以 SingleTask 的方式 Push 一个页面到页面栈时，如果该页面已经在栈中，页面框架会把其之上的所有页面全部清除出栈，使其成为新的栈顶，这就是四种启动模式。



接下来，简单介绍下 iOS 的 Present 特性。当页面栈顶的 C 页面 Present D 页面时，D 页面并没有被加入到 ABC 页面栈中，而是变成了 C 页面的一个附属，当 D 页面要消失时，同样也是通过 C 页面的 dismiss 移除掉。这里有些限制，每个页面仅可以 Present

一个页面，这个页面可以是一个普通页面，也可以是一个导航页面，那么导航页面是什么呢？大家可以理解成一个新的页面栈（功能类似 UINavigationController），其上可以添加其它页面。如果 D 页面是导航页面，就可以在其上 Push 其它的页面，如果业务流程有一个主流程，一个分支流程，就可以采用这种方式实现。

最后是 Android 的 onResult 特性，实现了页面间数据返回的解耦，如上示例代码就是大致的实现原理。具体来说，从 A 页面跳转到 B 页面，那么 B 页面执行了一段逻辑之后，A 希望得到执行结果，如果按照原来 iOS 的实现方式，只能通过监听 Listener 或 Delegate 等方式将 B 页面的执行结果返回给 A 页面。当 iOS 的页面框架实现了这个特性， A 页面就不需要额外注册 B 页面的 Listener 或 Delegate 了，只需重写自己的 onResult 方法并处理结果即可，这样既可以实现页面解耦，又方便了业务同学开发。

接下来，举个高德地图手机端上的具体实例。

有这样一个搜索场景，从一个具体地理位置详情页可以跳转到以它为中心的搜周边页，在搜周边页中又可以跳转到另一个具体地理位置详情页，接着可以跳转到新的搜周边页，以此递归循环，但是返回时，产品希望仅返回到之前搜索过的具体地理位置详情页，略去搜周边页。如上视频展示，查询顺序是：7 天优品酒店详情页 -> 7 天优品酒店搜周边页 -> 火驴火烧肉亭详情页 -> 火驴火烧肉亭搜周边页；返回顺序是：火驴火烧肉亭搜周边页 -> 火驴火烧肉亭详情页 -> 7 天优品酒店详情页。中间的 7 天优品酒店搜周边页被去掉了。

在 iOS 页面框架未实现 launch mode 前，火驴火烧肉亭详情页在跳转到火驴火烧肉亭搜周边页前，需要自行遍历当前页面栈，将 7 天优品酒店搜周边页从页面栈中移出后，再跳转到火驴火烧肉亭搜周边页，以此保证产品逻辑的正确性。在实现了 launch mode 之后，火驴火烧肉亭详情页仅需以 SingleInstance 的方式打开火驴火烧肉亭搜周边页即可，页面框架会自动将之前的 7 天优品酒店搜周边页调到栈顶，并将该搜周边页的内容刷新为火驴火烧肉亭搜周边。极大简化了 iOS 端业务同学的开发成本，规范了 iOS 页面跳转的规范，结束了由业务自行操作页面栈的混乱时代，同时双端技术能力的融合也为上层动态 UI 业务提供了一致性的体验。

上面，我们介绍了双端融合方案的三种方式，也举例说明了其带来的效果。下沉 C++，实现两套代码合一，解决了一致性问题，提高了性能，但同时也提高了开发门槛，适用于多年沉淀的核心逻辑；上漂动态 UI，同样解决了双端一致性问题，性能会稍有损失，但降低了开发门槛，使得开发速度得到提升，适用于频繁变动的业务场景；双端拉齐则是借鉴了双端优势，做到互相融合。

组件化

我们做了一些团队组件化方案的选型和参考，例如手淘的 Atlas、Beehive，网易的 LDBusMediator 等，由于这些组件化方案都比较成熟，这里不再赘述。它们都包含五个概念：容器、模块、生命周期、页面路由和对外服务（通信），我们重新命名了这些概念使其更加形象化。

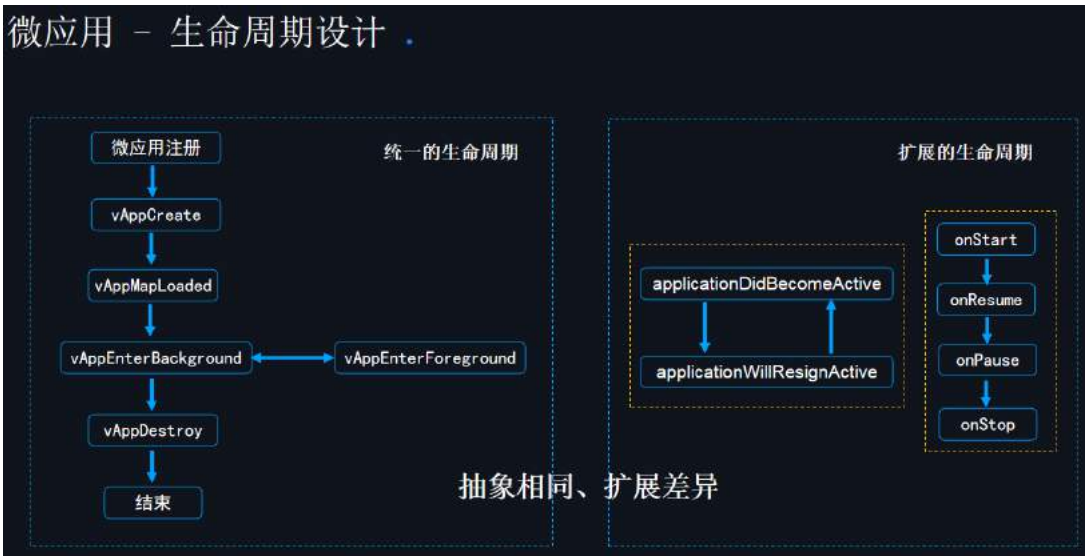
容器，负责管理模块；模块，是一个独立的功能单元，可以独立编译；微应用，管理模块的生命周期，对于一个手机操作系统，是为每个应用派发生命周期，对于一个单独的应

用，是为每个模块派发生命周期，就像一个应用管理着很多微应用一样，因此我们取了这个形象的名字；页面路由，负责进行 URL 的解析和页面的跳转；微服务，模块中的逻辑功能，同时提供对外服务。

我们对容器在设计进行了一些改造，如上右半边图，模块被虚化了，被定义成了一个物理概念（即一个独立代码仓库），逻辑上拆分为微应用、微服务和页面路由，容器不再管理模块，而是直接管理这三个元素。之所以这样做，是因为我们希望业务更关注自身需要的服务是什么，而不是它在哪个模块，这些也是借鉴了安卓的组件化思想。

接下来，我们详细介绍下微应用生命周期的设计，如上图，微应用在 iOS 端参考的是 UIApplicationDelegate 的生命周期，而在 Android 端参考的是 Activity 的生命周期。做这样的参考选择，原因有三：一，高德地图内的应用场景大都依赖前后台切换的事件做一些逻辑处理；二，iOS 的 UIApplicationDelegate 作为应用的生命周期，同时支持前后台切换，完全吻合高德地图的场景；三，Android 选择 Activity 是因其组件化的思想，在 Android 的设计中，Application 已经弱化成了一个特殊进程的概念，并不能代表一个应用，且高德地图是基于单 Activity 实现的（上面介绍页面框架时提到过），通过 Activity 的 onStop，onRestart 生命周期中做些逻辑处理，即可判断出应用是否为前后台切换。

这样，去除图中虚线框中的生命周期后，双端得到了统一的生命周期，如下左半部分图：



对于虚线中差异化的部分（如上右半部分图），设计为扩展的生命周期，做到抽象相同、扩展差异，即统一了通用的生命周期，也支持了双端各自的特性。

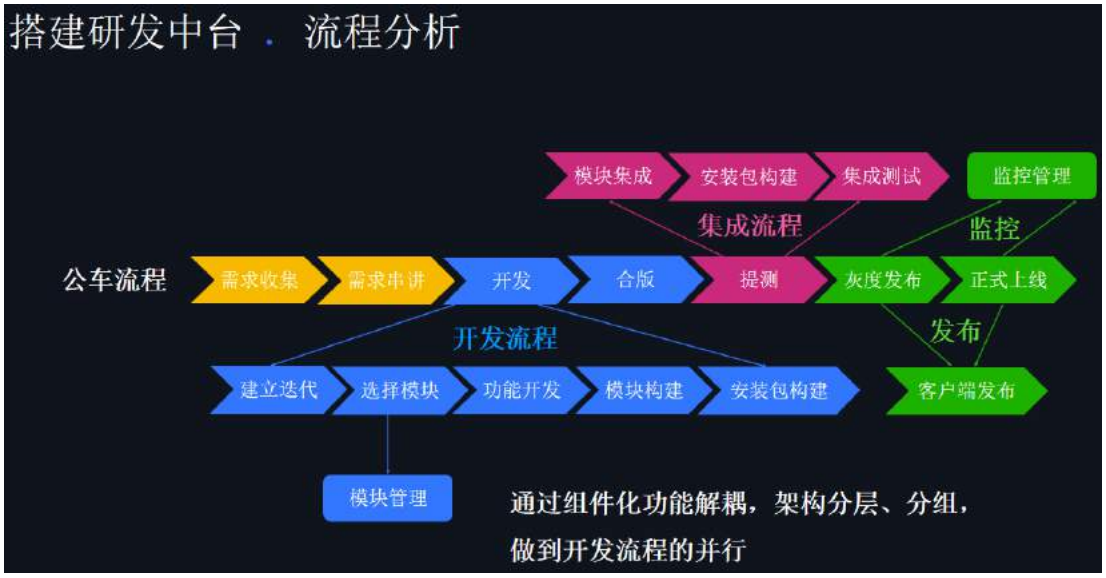
对于微服务，我们定义了一个通信规范，只能通过接口方法，不能直接调用实现。定义微服务主要是希望 UI 展现与业务逻辑能够分离，并让业务逻辑服务化，不仅服务于当前页面，也能够服务更多页面，提高代码的复用率，降低维护成本。

有了容器框架，代码便可以抽成一个个独立的模块单元，但模块应该放在那里，上下依赖关系是什么，还需要对模块进行分层、分组，下图为分层、分组后的整体架构：

通过容器建设，架构分层、分组，我们实现了组件化，解除了模块间耦合，提高了代码复用率，为后面的高效并行打好基础。分而治之的思想，组件化的“分”也是为后面的“治”做好铺垫。

搭建研发中台

研发中台应该有哪些功能，可以结合组件化和公车流程来分解，如下图：



主流程是公车流程，分为：需求收集、需求串讲、开发、合版、提测、灰度发布和正式上线。开发流程可以分解为更细的建立迭代、选择模块、功能开发、模块构建和安装包构建。这里解释下迭代的概念，即是一个发版周期内的功能开发。组件化实现了功能解耦，使得不同业务团队可以在开发阶段创建自己的迭代并行开发，开发完成后在规定的时间段进行合版。提测流程可以分成模块集成、安装包构建和集成测试，其中模块集成是以产物的方式进行集成。测试通过后，通过客户端发布流程，进行灰度发布验证，灰度通过后，再进行正式上线，上线之后，我们会对崩溃、性能等维度进行监控。通过流程拆解，我们整理出了研发中台的完整功能：

研发中台建设完成后，我们实现了研发流程、测试流程以及发布流程的自动化，提高了人效。另外，通过质量管控，提高了稳定性；通过流程管控，约束了可能产生的风险。

主副收益

首先，通过双端融合、组件化、中台建设提升了代码稳定性，实现了流程自动化，做到了

开发阶段的并行，使发版周期缩短到原来的一半，从伪公车变成真公车。

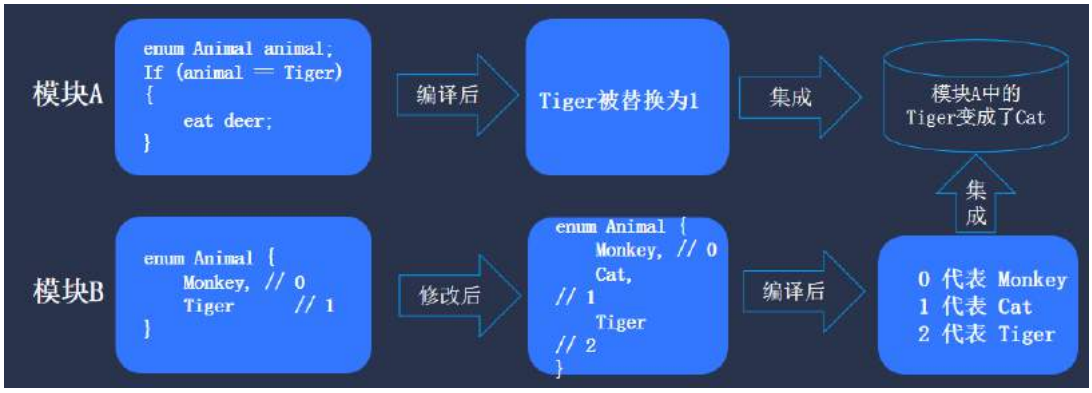
其次，通过质量优化，让崩溃率从万分之八降低到十万分之八：双端融合减少了一致性问题；架构合理化提高了可维护性；关键流程管控，减少了风险源头；通过质量扫描，解决了头部质量问题，通过崩溃监控，解决头部崩溃问题。

然后，通过升级编译脚本，支持并行编译；通过模块化，基于产物构建安装包，大大降低编译时长，从原来的 40 多分钟降至现在的 8 分钟。

最后是包大小优化，iOS 端从 146M 减到 123M，纯减量达 48M，这主要是通过编译优化，资源云化，功能合并（分层、分组），svg 替代 png 小图标，删除无用图片和代码实现等手段实现。资源云化主要是指将启动时的非必要资源放在云端，需要时再进行动态加载。

经验教训

在组件化以后，编译模式发生了一些变化，模块在集成前提前生成了产物，这些变化同时带来了一些问题，比如二进制兼容问题。以枚举功能为例，在模块化后，A 模块依赖 B 模块中定义的枚举，在 A 模块生成产物后，B 模块的枚举定义发生了变化，A 中使用的枚举值含义可能发生变化，如下图：



为了解决该问题，我们制定了一些开发规范：对于枚举的定义，不允许删除任何已定义的枚举值，不允许从中间插入任何枚举值，如果一定要添加，只能在末尾添加，以此来解决二进制兼容性问题。当然，除了枚举的问题，还有宏定义等引起的二进制兼容性问题，此处不一一详述。

此外，Android 端还可能出现代码注解丢失问题。编译期注解仅存在于编译阶段，模块化后，产物中无法保存注解信息，导致产物集成时，由于找不到注解信息而无法进行全局注册。为此，我们做了一些自定义 APT 插件，在注解处 阶段生成 Java 数据类的同时也存储一份注解信息，这样在集成阶段就可以根据注解信息进行全局注册。

未来展望

2018 年，高德客户端通过一系列架构治理，从伪公车变成了真公车，但这只是近几年架构

演进的一个阶段性成果。未来，我们要发挥动态 UI 的优势，让业务真正动态化起来，从公车时代跨入到 Feature Team 时代，让公车变成一条条公路，每个 Feature Team 就是一个小汽车，按照自己的节奏装好货物后，就可以在修好的公路上自由的行驶，更好地做到灵活、并行和高效！

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

Android Native 内存泄漏系统化解决方案

作者：布强

导读

C++内存泄漏问题的分析、定位一直是 Android 平台上困扰开发人员的难题。因为地图渲染、导航等核心功能对性能要求很高，高德地图 APP 中存在大量的 C++代码。解决这个问题对于产品质量尤为重要和关键，我们在实践中形成了一套自己的解决方案。

分析和定位内存泄漏问题的**核心在于分配函数的统计和栈回溯**。如果只知道内存分配点不知道调用栈会使问题变得格外复杂，增加解决成本，因此两者缺一不可。

Android 中 Bionic 的 malloc_debug 模块对内存分配函数的监控及统计是比较完善的，但是栈回溯在 Android 体系下缺乏高效的方式。随着 Android 的发展，Google 也提供了栈回溯的一些分析方法，但是这些方案存在下面几个问题：

- 栈回溯的环节都使用的 libunwind，这种获取方式消耗较大，在 Native 代码较多的情况下，频繁调用会导致应用很卡，而监控所有内存操作函数的调用栈正需要高频的调用 libunwind 的相关功能。
- 有 ROM 要求限制，给日常开发测试带来不便。
- 用命令行或者 DDMS 进行操作，每排查一次需准备一次环境，手动操作，最终结果也不够直观，同时缺少对比分析。

因此，如何进行高效的栈回溯、搭建系统化的 Android Native 内存分析体系显得格外重要。

高德地图基于这两点做了一些改进和扩展，经过这些改进，通过自动化测试可及时发现并解决这些问题，大幅提升开发效率，降低问题排查成本。

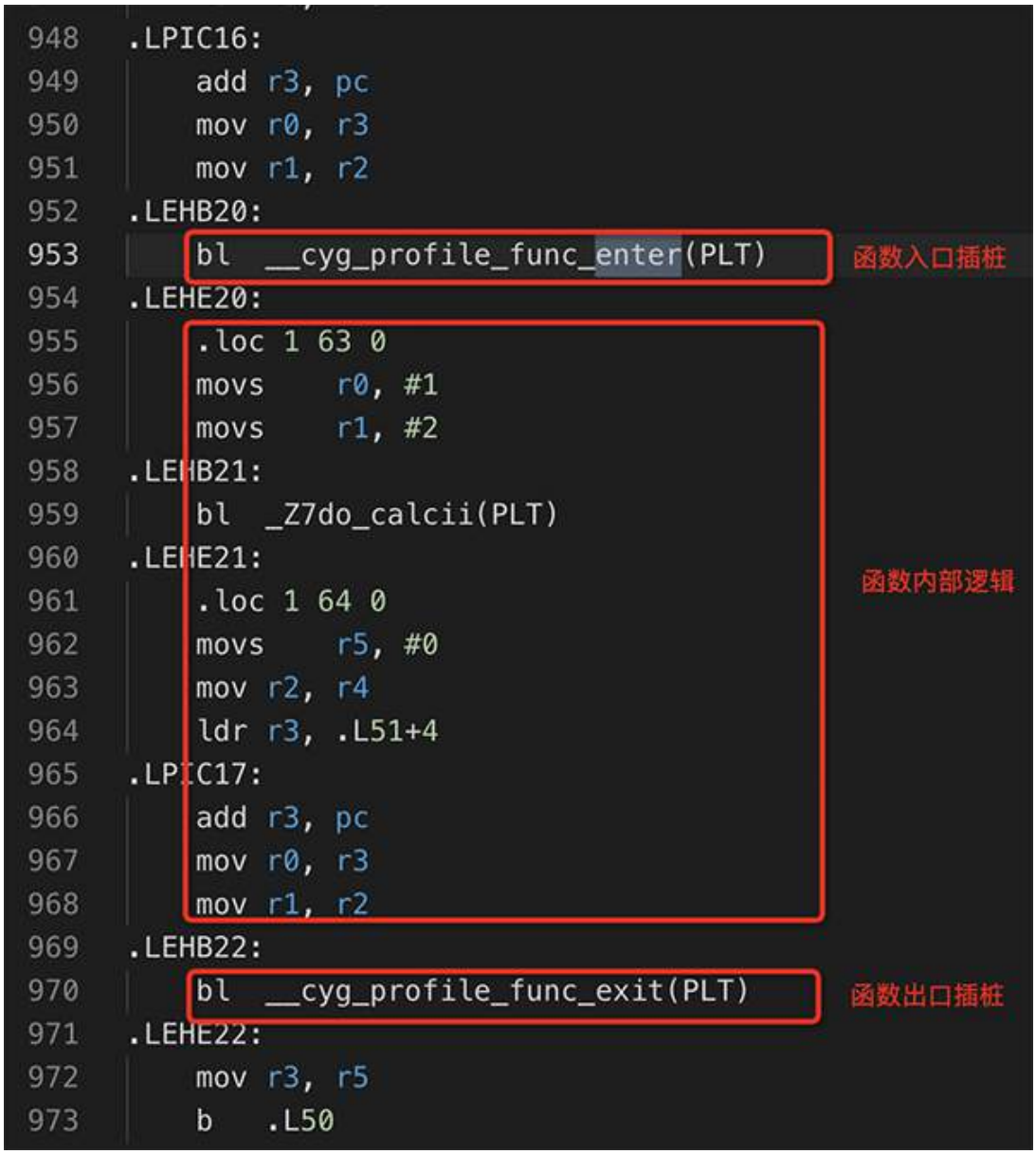
栈回溯加速

Android 平台上主要采用 libunwind 来进行栈回溯，可以满足绝大多数情况。但是 libunwind 实现中的全局锁及 unwind table 解析，会有性能损耗，在多线程频繁调用情况下会导致应用变卡，无法使用。

加速原理

编译器的-finstrument-functions 编译选项支持编译期在函数开始和结尾插入自定义函数，在每个函数开始插入对__cyg_profile_func_enter 的调用，在结尾插入对__cyg_profile_func_exit 的调用。这两个函数中可以获取到调用点地址，通过对这些地址的记录就可以随时获取函数调用栈了。

插桩后效果示例：



这里需要格外注意，某些不需要插桩的函数可以使用__attribute__((no_instrument_function))来向编译器声明。

如何记录这些调用信息？我们想要实现这些信息在不同的线程之间读取，而且不受影响。一种办法是采用线程的同步机制，比如在这个变量的读写之处加临界区或者互斥量，但是这样又会影响效率了。

能不能不加锁？这时就想到了线程本地存储，简称 TLS。TLS 是一个专用存储区域，只能由自己线程访问，同时不存在线程安全问题，符合这里的场景。

于是采用编译器插桩记录调用栈，并将其存储在线程局部存储中的方案来实现栈回溯加速。具体实现如下：

- 利用编译器的-finstrument-functions 编译选项在编译阶段插入相关代码。
- TLS 中对调用地址的记录采用数组+游标的形式，实现最快速度的插入、删除及获取。

定义数组+游标的数据结构：

```
typedef struct {
    void* stack[MAX_TRACE_DEEP];
    int current;
} thread_stack_t;
```

初始化 TLS 中 thread_stack_t 的存储 key：

```
static pthread_key_t sBackTraceKey;
static pthread_once_t sBackTraceOnce = PTHREAD_ONCE_INIT;

static void __attribute__((no_instrument_function))
destructor(void* ptr) {
    if (ptr) {
        free(ptr);
    }
}

static void __attribute__((no_instrument_function))
init_once(void) {
    pthread_key_create(&sBackTraceKey, destructor);
}
```

初始化 thread_stack_t 放入 TLS 中：

```
thread_stack_t* __attribute__((no_instrument_function))
get_backtrace_info() {
    thread_stack_t* ptr = (thread_stack_t*) pthread_getspecific(sBackTraceKe
y);
    if (ptr)
        return ptr;

    ptr = (thread_stack_t*)malloc(sizeof(thread_stack_t));
    ptr->current = MAX_TRACE_DEEP - 1;
```

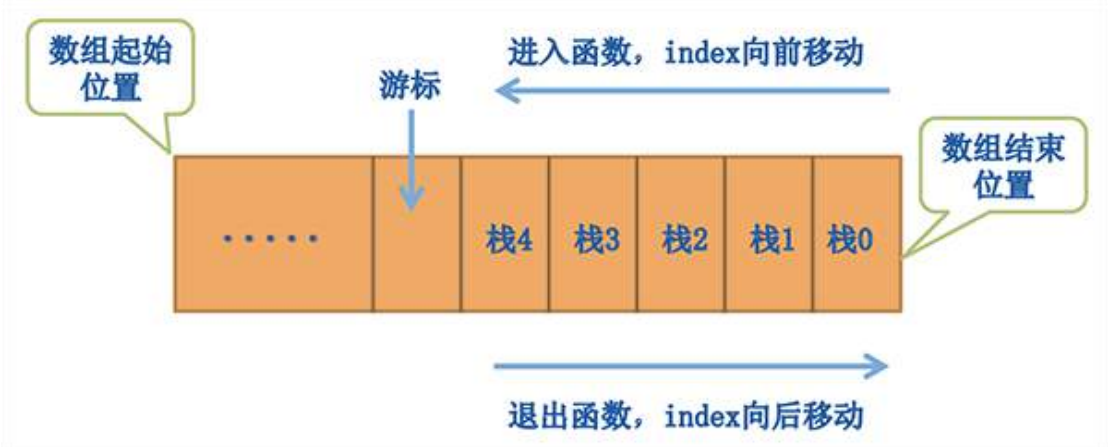
```
9.      pthread_setspecific(sBackTraceKey, ptr);
10.     return ptr;
11. }
```

- 实现__cyg_profile_func_enter 和__cyg_profile_func_exit，记录调用地址到 TLS 中。

```
1.  extern "C" {
2.  void __attribute__((no_instrument_function))
3.  __cyg_profile_func_enter(void* this_func, void* call_site) {
4.      pthread_once(&sBackTraceOnce, init_once);
5.      thread_stack_t* ptr = get_backtrace_info();
6.      if (ptr->current > 0)
7.          ptr->stack[ptr->current--] = (void*)((long)call_site - 4);
8.  }
9.
10. void __attribute__((no_instrument_function))
11. __cyg_profile_func_exit(void* this_func, void* call_site) {
12.     pthread_once(&sBackTraceOnce, init_once);
13.     thread_stack_t* ptr = get_backtrace_info();
14.     if (++ptr->current >= MAX_TRACE_DEEP)
15.         ptr->current = MAX_TRACE_DEEP - 1;
16. }
17. }
```

__cyg_profile_func_enter 的第二个参数 call_site 就是调用点的代码段地址，函数进入的时候将它记录到已经在 TLS 中分配好的数组中，游标 ptr->current 左移，待函数退出游标 ptr->current 右移即可。

逻辑示意图：



记录方向和数组增长方向不一致是为了对外提供的获取栈信息接口更简洁高效，可以直接进行内存 copy 以获取最近调用点的地址在前、最远调用点的地址在后的调用栈。

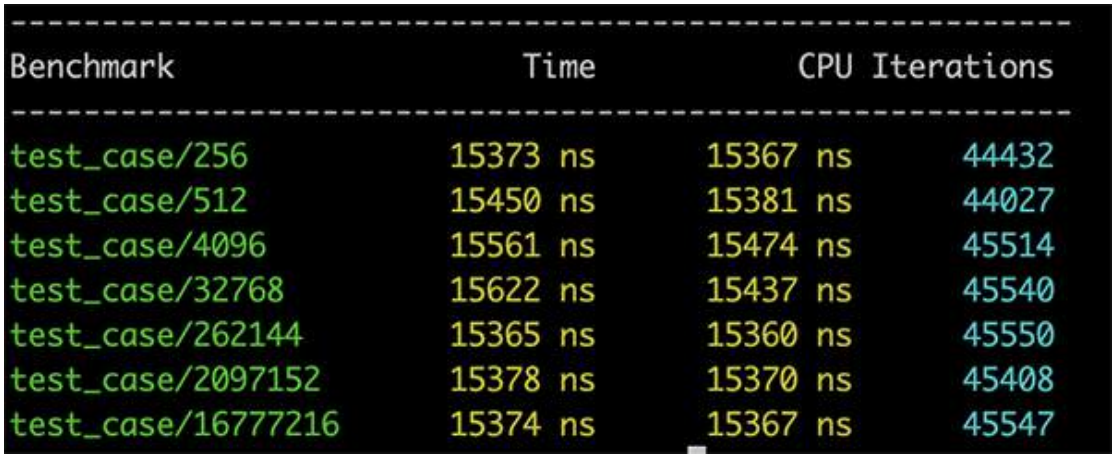
- 提供接口获取栈信息。

```
1.  int __attribute__((no_instrument_function))
2.  get_tls_backtrace(void** backtrace, int max) {
3.      pthread_once(&sBackTraceOnce, init_once);
4.      int count = max;
5.      thread_stack_t* ptr = get_backtrace_info();
6.      if (MAX_TRACE_DEEP - 1 - ptr->current < count) {
7.          count = MAX_TRACE_DEEP - 1 - ptr->current;
8.      }
9.      if (count > 0) {
10.         memcpy(backtrace, &ptr->stack[ptr->current + 1], sizeof(void *) * co
    unt);
11.     }
12.     return count;
13. }
```

- 将上面逻辑编译为动态库，其他业务模块都依赖于该动态库编译，同时编译 flag 中添加-finstrument-functions 进行插桩，进而所有函数的调用都被记录在 TLS 中了，使用者可以在任何地方调用 get_tls_backtrace(void** backtrace, int max)来获取调用栈。

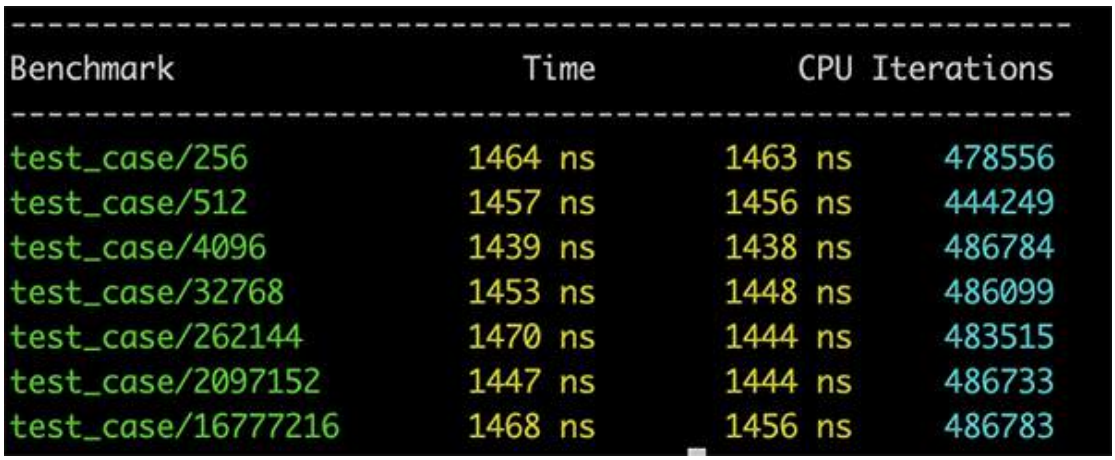
效果对比（采用 Google 的 benchmark 做性能测试，手机型号：华为畅想 5S，5.1 系统）：

libunwind 单线程



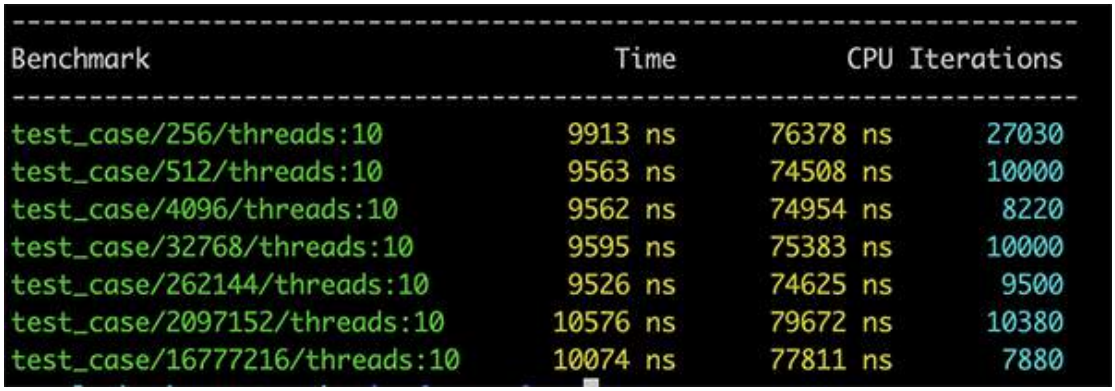
Benchmark	Time	CPU	Iterations
test_case/256	15373 ns	15367 ns	44432
test_case/512	15450 ns	15381 ns	44027
test_case/4096	15561 ns	15474 ns	45514
test_case/32768	15622 ns	15437 ns	45540
test_case/262144	15365 ns	15360 ns	45550
test_case/2097152	15378 ns	15370 ns	45408
test_case/16777216	15374 ns	15367 ns	45547

TLS 方式单线程获取



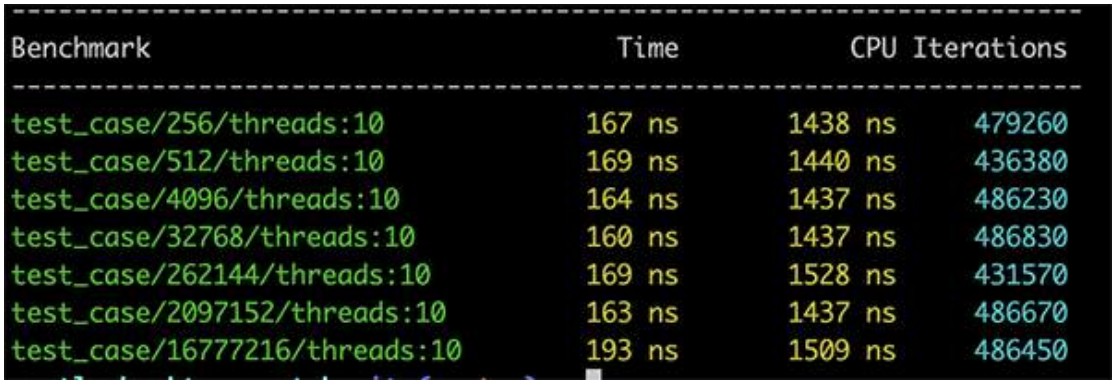
Benchmark	Time	CPU	Iterations
test_case/256	1464 ns	1463 ns	478556
test_case/512	1457 ns	1456 ns	444249
test_case/4096	1439 ns	1438 ns	486784
test_case/32768	1453 ns	1448 ns	486099
test_case/262144	1470 ns	1444 ns	483515
test_case/2097152	1447 ns	1444 ns	486733
test_case/16777216	1468 ns	1456 ns	486783

libunwind 10 个线程



Benchmark	Time	CPU	Iterations
test_case/256/threads:10	9913 ns	76378 ns	27030
test_case/512/threads:10	9563 ns	74508 ns	10000
test_case/4096/threads:10	9562 ns	74954 ns	8220
test_case/32768/threads:10	9595 ns	75383 ns	10000
test_case/262144/threads:10	9526 ns	74625 ns	9500
test_case/2097152/threads:10	10576 ns	79672 ns	10380
test_case/16777216/threads:10	10074 ns	77811 ns	7880

TLS 方式 10 个线程



Benchmark	Time	CPU	Iterations
test_case/256/threads:10	167 ns	1438 ns	479260
test_case/512/threads:10	169 ns	1440 ns	436380
test_case/4096/threads:10	164 ns	1437 ns	486230
test_case/32768/threads:10	160 ns	1437 ns	486830
test_case/262144/threads:10	169 ns	1528 ns	431570
test_case/2097152/threads:10	163 ns	1437 ns	486670
test_case/16777216/threads:10	193 ns	1509 ns	486450

从上面几个统计图可以看出单线程模式下该方式是 libunwind 栈获取速度的 10 倍，10 个线程情况下是 libunwind 栈获取速度的 50-60 倍，速度大幅提升。

优缺点

- 优点：速度大幅提升，满足更频繁栈回溯的速度需求。
- 缺点：编译器插桩，体积变大，不能直接作为线上产品使用，只用于内存测试包。这个问题可以通过持续集成的手段解决，每次项目出库将 C++项目产出普通库及对应的

内存测试库。

体系化

经过以上步骤可以解决获取内存分配栈慢的痛点问题，再结合 Google 提供的工具，如 DDMS、adb shell am dumpheap -n pid /data/local/tmp/heap.txt 命令等方式可以实现 Native 内存泄漏问题的排查，不过排查效率较低，需要一定的手机环境准备。

于是，我们决定搭建一整套体系化系统，可以更便捷的解决此类问题，下面介绍下整体思路：

- 内存监控沿用 LIBC 的 malloc_debug 模块。不使用官方方式开启该功能，比较麻烦，不利于自动化测试，可以编译一份放到自己的项目中，hook 所有内存函数，跳转到 malloc_debug 的监控函数 leak_xxx 执行，这样 malloc_debug 就监控了所有的内存申请/释放，并进行了相应统计。

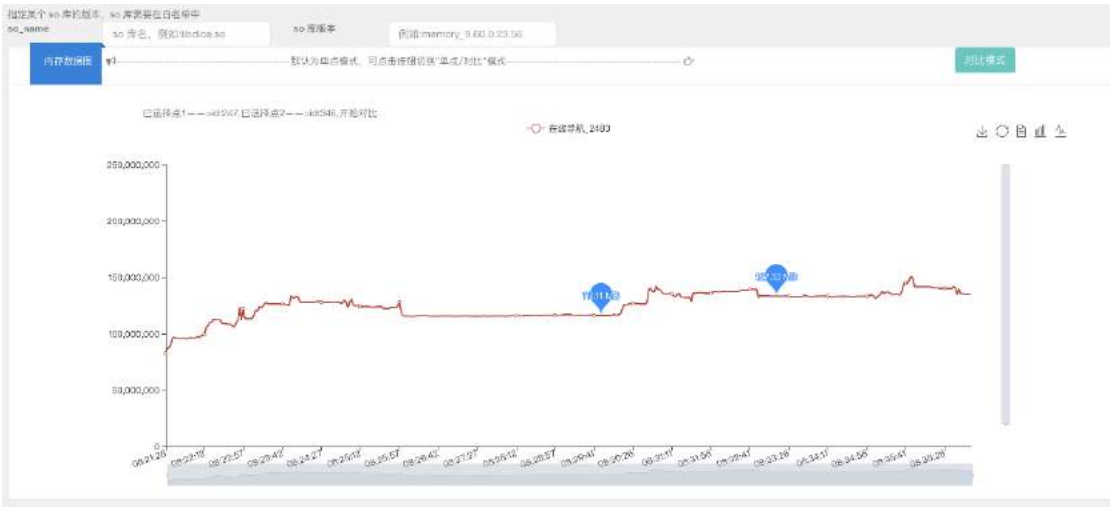
- 用 get_tls_backtrace 实现 malloc_debug 模块中用到的__LIBC_HIDDEN__ int32_t get_backtrace_external(uintptr_t* frames, size_t max_depth)，刚好同上面说的栈回溯加速方式结合。

- 建立 Socket 通信，支持外部程序经由 Socket 进行数据交换，以便更方便获取内存数据。

- 搭建 Web 端，获取到内存数据上传后可以被解析显示，这里要将地址用 addr2line 进行反解。

- 编写测试 Case，同自动化测试结合。测试开始时通过 Socket 收集内存信息并存储，测试结束将信息上传至平台解析，并发送评估邮件。碰到有问题的报警，研发同学就可以直接在 Web 端通过内存曲线及调用栈信息来排查问题了。

系统效果示例：



对比信息汇总

内存大小比：点①内存大小为：58.53 MB　点②内存大小为：69.63 MB　差值为：11.10 MB　差比为：18.97 %

对象个数比：点①对象个数为：0　点②对象个数为：0　差值为：0　差比为：0.00 %

Show 10 entries　Search:

库	标签	点①占用大小(/KB)	点②占用大小(/KB)	差值(/KB)	差比
libdice.so (memory_10.10.0.127)		27833.21	39246.81	11413.60	41.01
libhwui.so		3006.93	3111.03	104.10	3.46
libsqlite.so		948.00	1031.08	82.07	8.75
libc++.so		3085.33	3149.67	64.34	2.09
libajx_v3.so (memory_10.10.0.59.80)		11187.54	11232.33	44.79	0.40
libGLESv2_adreno.so		3927.22	3957.25	30.02	0.76
libutils.so		401.43	404.93	3.49	0.87
[illegible]	[illegible]	88.97	90.97	1.00	1.11
libart.so		33.75	34.00	0.25	0.74
libsgmainso-6.4.112.so		481.87	481.87	0.20	0.04

Previous 1 2 3 4 Next

Show 20 entries　Search:

库名	点1对象数	点2对象数	差值	差比%	点1大小(KB)	点2大小(KB)	差值	差比%	点1总大小(KB)	点2总大小(KB)	差值	差比%	栈信息
libdice.so (渲染)	0	53	53	-	0.00	3017.58	3017.58	-	0.00	3019.96	3019.95	-	CAnRoadWidthLineRenderObj::CollectGpuInfo()
libdice.so	0	12	12	-	0.00	1339.13	1339.13	-	0.00	1339.13	1339.13	-	sgi::__malloc_alloc_template<0>::allocate(unsigned int)
libdice.so (渲染)	0	4	4	-	0.00	192.00	192.00	-	0.00	768.00	768.00	-	rct::CModelParse::InitObjectsBuffer(unsigned char const*, rct::GModelInfoHead*, rct::GModelData*)
libdice.so (基础库)	79	98	19	24.05	1427.74	2128.67	700.93	49.09	1427.74	2128.67	700.93	49.09	dice::OnlineDataPacker::packTileData(dice::OnlineTileDataInfo const*, unsigned char*&, int&)
libdice.so	0	1	1	-	0.00	426.41	426.41	-	0.00	426.41	426.41	-	b6e938b1
libdice.so	66	56	-10	-15.15	182.84	587.13	404.29	221.11	255.94	660.08	404.14	157.91	b69a7337
libdice.so (渲染)	0	15	15	-	0.00	382.22	382.22	-	0.00	382.22	382.22	-	CAnRoadWidthLineRenderObj::CollectGpuInfo()
libdice.so (渲染)	0	16	16	-	0.00	288.23	288.23	-	0.00	288.23	288.23	-	CAnBuildingRenderObj::CopyVertice()

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

字节码技术在模块依赖分析中的应用

作者：昭鱼

背景

近年来，随着手机业务的快速发展，为满足手机端用户诉求和业务功能的迅速增长，移动端的技术架构也从单一的大工程应用，逐步向模块化、组件化方向发展。以高德地图为例，Android 端的代码已突破百万行级别，超过 100 个模块参与最终构建。

试想一下，如果没有一套标准的依赖检测和监控工具，用不了多久，模块的依赖关系就可能会乱成一锅粥。

从模块 Owner 的角度看，为什么依赖分析这么重要？

- 作为模块 Owner，我首先想知道“谁依赖了我？依赖了哪些接口”。唯有如此才能评估本模块改动的影响范围，以及暴露的接口的合理性。

- 我还想知道“我依赖了谁？调用了哪些外部接口”，对所需要的外部能力做到心中有数。

从全局视角看，一个健康的依赖结构，要防止“下层模块”直接依赖“上层模块”，更要杜绝循环依赖。通过分析全局的依赖关系，可以快速定位不合理的依赖，提前暴露业务问题。

因此，依赖分析是研发过程中非常重要的一环。本文将介绍 Java 字节码技术在 Android 模块依赖分析中的应用。文中的“字节码”特指“Java 字节码”。

常见的依赖分析方式

提到 Android 依赖分析，首先浮现在脑海中的可能是以下这些方案：

- 分析 Gradle 依赖树。
- 扫描代码中的 `import` 声明。
- 使用 Android Studio 自带的分析功能。

我们逐个来分析这几个方案：

1.Gradle 依赖树

使用 `./gradlew :<module>:dependencies --configuration releaseCompileClasspath -q` 命令，很容易就可以得到模块的依赖树，如图：

```
------------------------------------------------------------
Project :buildType
------------------------------------------------------------

releaseCompileClasspath - Resolved configuration for compilation for variant: release
+--- com.squareup.leakcanary:leakcanary-android:1.3.1
|    \--- com.squareup.leakcanary:leakcanary-analyzer:1.3.1
|         +--- com.squareup.haha:haha:1.3
|         \--- com.squareup.leakcanary:leakcanary-watcher:1.3.1
\--- com.autonavi.amap:appcontext:1.0.5
```

不难发现，这种方式有两个问题：

- 声明即依赖，即使代码中没有使用的库，也会输出到结果中。
- 只能分析到模块级别，无法精确到方法级别。

2.扫描 `import` 声明

扫描 Java 文件中的 import 语句，可以得到文件(类)之间的调用关系。

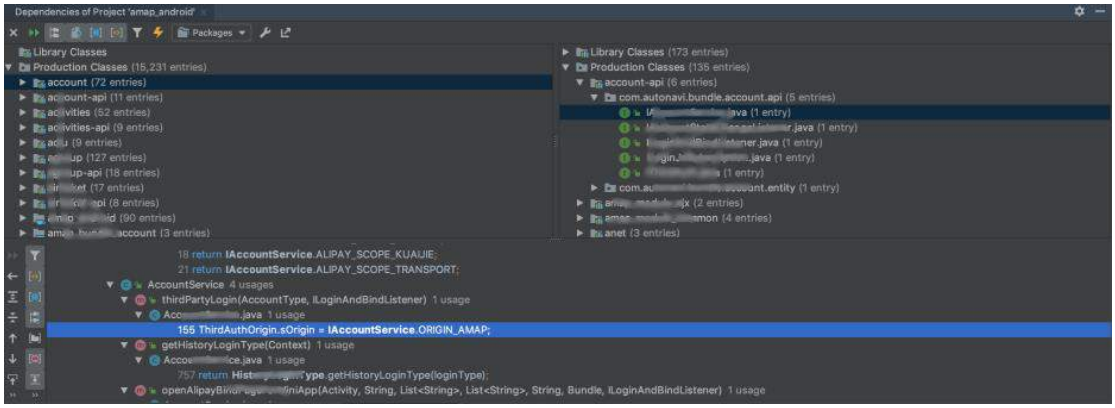
因为模块与文件(类)的对应关系非常容易得到(扫描目录)。所以，得到了文件(类)之间的依赖关系，即是得到了模块之间文件(类)级别的依赖关系。

这个方案相比 Gradle 依赖扫描提升了结果维度，可以分析到文件(类)级别。但是它也存在一些缺点：

- 无法处理 import * 的情况。
- 扫描“有 import 但未使用对应类”的场景效率太低(需要做源码字符串查找)。

3.使用 IDE 自带的分析功能

触发 Android Studio 菜单 「Analyze」 -> 「Analyze Dependencies」，可以得到模块间方法级别的依赖关系数据。如图：



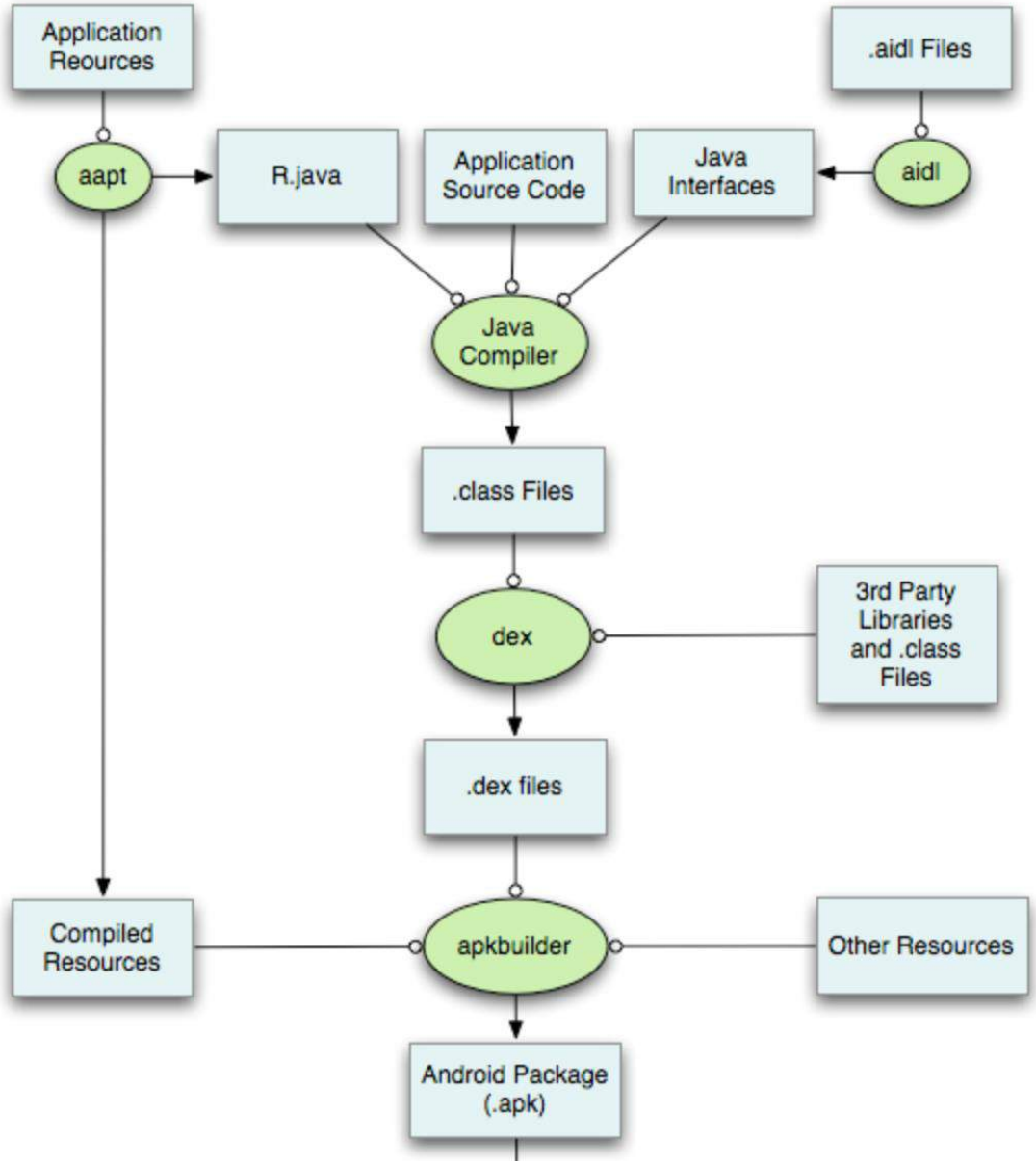
Android Studio 能准确分析到模块之间“方法级别”的引用关系，支持在 IDE 中跳转查看，也能扫描到对 Android SDK 的引用。

这个方案比前面两个都优秀，主要是准确。但是它也有几个问题：

- 耗时较长：全面分析 AMap 全源码，大约需要 10 分钟。
- 分析结果无法为第三方复用，无法生成可视化的依赖关系图。
- 分析正向依赖和逆向依赖，需要扫描两次。

总结一下上述三种方案：Gralde 依赖基于工程配置，粒度太粗且结果不准。“Import 扫描方案”能拿到文件级别依赖但数据不全。IDE 扫描虽然结果精准，但是数据复用困难，不便于工程化。

为什么要使用字节码来分析？



参考 Android 构建流程图，所有的 Java 源代码和 aapt 生成的 R.java 文件，都会被编译成 .class 文件，再被编译为 dex 文件，最终通过 apkbuilder 生成到 apk 文件中。图中的 .class 文件即是我们所说的 Java 字节码，它是对 Java 源码的二进制转义。

在 Android 端，常见的字节码应用场景包括：

- 字节码插桩：用于实现对 UI 、内存、网络等模块的性能监控。
- 修改 jar 包：针对无源码的库，通过编辑字节码来实现一些简单的逻辑修改。

回到本文的主题，为什么要分析字节码，而不是 Java 代码或者 dex 文件？

不使用 Java 代码是因为有些库以 jar 或者 aar 的方式提供，我们获取不到源码。不使用 dex 文件是因为它没有好用的语法分析工具。所以解析字节码几乎是我们唯一的选择。

如何使用字节码分析依赖关系？

要得到模块之间的依赖关系，其实就是要得到“模块间类与类”之间的依赖关系。而要确定类之间的关系，分析类字节码的语句即可。

1.在什么时机来分析？

了解 Android 构建流程的同学，应该对 transform 这个任务不陌生。它是 Android Gradle 插件提供的一个字节码 Hook 入口。

在 transform 这个任务中，所有的字节码文件(包括三方库) 以 Input 的格式输入。

以 JarInput 为例，分析其 file 字段，可得到模块的名称。解析 file 文件，即可得到此模块所有的字节码文件。

```
-----------------------------------------------------------------------------------------------
JarInput.file                                                                  | Module
-------------------------------------------------------------------------------- | ----------
~/.gradle/caches/transforms-1/files-1.1/verifysdk-1.0.0.aar/xxx/jars/classes.jar  | verifysdk
~/.gradle/caches/modules-2/files-2.1/com.amap/pages/1.0.1/xxx/pages-1.0.1.jar     | pages
-----------------------------------------------------------------------------------------------
```

有了模块名称和对应路径下的 class 文件，就建立了模块与类的对应关系，这是我们拿到的第一个关键数据。

2.使用什么工具分析？

解析 Java 字节码的工具，最常用的包括 Javassit，ASM，CGLib。ASM 是一个轻量级的类库，性能较好，但需要直接操作 JVM 指令。CGLib 是对 ASM 的封装，提供了更高级的接口。

相比而言，Javassist 要简单的多，它基于 Java 的 API ，无需操作 JVM 指令，但其性能要差一些(因为 Javassit 增加了一层抽象)。在工程原型阶段，为了快速验证结果，我们优先选择了 Javassit 。

3.具体方案是怎样的？

先看一个简单的示例，如何分析下面这段代码的调用关系：

```
1: package com.account;
2: import com.account.B;
3: public class A {
4:     void methodA() {
5:         B b = new B(); // 初始化了 Class B 的实例 b
6:         b.methodB();   // 调用了 b 的 methodB 方法
7:     }
8: }
```

第 1 步：初始化环境，加载字节码 A.class，注册语句分析器。

```
// 初始化 ClassPool，将字节码文件目录注册到 Pool 中。
ClassPool pool = ClassPool.getDefault();
pool.insertClassPath('<class 文件所在目录>')
// 加载类 A
CtClass cls = pool.get("com.account.A");
// 注册表达式分析器到类 A
MyExprEditor editor = new MyExprEditor(ctCls)
ctCls.instrument(editor)
```

第 2 步：自定义表达式解析器，分析类 A(以解析语句调用为例)。

```
class MyExprEditor extends ExprEditor {
    @Override
    void edit(MethodCall m) {
        // 语句所在类的名称
        def clsAName = ctCls.name
        // 语句在哪个方法被调用
        def where = m.where().methodInfo.getName()
        // 语句在哪一行被调用
        def line = m.lineNumber
        // 被调用类的名称
        def clsBName = m.className
        // 被调用的方法
        def methodBName = m.methodName
    }
    // 省略其它解析函数 ...
}
```

ExprEditor 的 edit(MethodCall m) 回调能拦截 Class A 中所有的方法调用(MethodCall)。

除了本例中对 MethodCall 的解析，它还支持解析 new，new Array，ConstructorCall，FieldAccess，InstanceOf，强制类型转换，try-catch 语句。

解析完 Class A，我们得到了 A 对 B 的依赖信息 ：

```
1. ------------------------------------------------------------------------
2. | Class1        | Class2        | Expr       | method1 | method2 | lineNo |
3. | ------------- | ------------- | ---------- | ------- | ------- | ------ |
4. | com.account.A | com.account.B | NewExpr    | methodA | <init>  | 5      |
5. | com.account.A | com.account.B | methodCall | methodA | methodB | 6      |
6. ------------------- ------------------------------------------------------
7. 简单解释如下：
8. 类 com.account.A 的第 5 行(methodA 方法内)，调用了 com.account.B 的构造函数；
9. 类 com.account.A 的第 6 行(methodA 方法内)，调用了 com.account.B 的 methodB 函
   数；
```

这便是“类和类之间方法级”的依赖数据。结合第 1 步得到的“模块和类”的对应关系，最终我们便获得了“模块间方法级的依赖数据”。

基于这些基础数据，我们还可以自定义依赖检测规则、生成全局的模块依赖关系图等，本文就不展开了。

小结

本文主要介绍了模块依赖分析在研发过程中的重要性，分析了 Android 常见的依赖分析方案，从 Gradle 依赖树分析， Import 扫描，使用 IDE 分析，到最后的字节码解析，方案逐步递进。越是接近源头的解法，才是越根本的解法。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

离屏渲染在车载导航中的应用

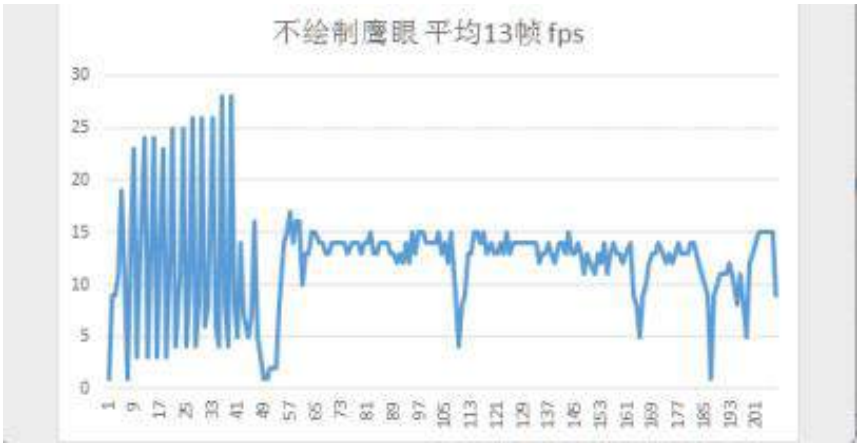
作者：吴朝良

1.导读

与手机导航不同，高德地图的车机版（AMAP AUTO）直接面对各大车厂和众多设备商。这些 B 端用户采用的硬件参数参差不齐，提出的业务需求涉及到渲染中诸多复杂技术的应用，这对渲染性能提出了极高的要求。

最初车机版沿用手机版的当前屏渲染模式，每一帧都需要实时的将地图元素渲染出来。但在业务实践过程中，我们发现在多屏渲染和多视图渲染场景下，CPU 负载急剧增高。以鹰眼图场景为例，在鹰眼图场景下，地图存在多视图渲染的状态：一张是主地图，一张是鹰眼小地图，因此渲染引擎同时渲染了两个地图实例对象，下图右下角即为鹰眼图：

鹰眼图绘制后，平均帧率下降了 2 帧，如下图所示：



针对上述情况，除了对渲染细节、批次和纹理等进行常规优化外，我们还需要寻找一种全局性的技术优化手段，大幅度提升引擎的渲染性能。为此，我们深入地研究了离屏渲染技术，并结合导航业务，提出了一种基于离屏渲染技术对特定地图的视图进行性能优化的方法。

2.优化原理

在 OpenGL 的渲染管线中，几何数据和纹理通过一系列变换和测试，最终被渲染成屏幕上的二维像素。那些用于存储颜色值和测试结果的二维数组被称为帧缓冲区。当我们创建了一个供 OpenGL 绘制用的窗体后，窗体系统会生成一个默认的帧缓冲区，这个帧缓冲区完全由窗体系统管理，且仅用于将渲染后的图像输出到窗口的显示区域。我们也可以使用在当前屏幕缓冲区以外开辟一个缓冲区进渲染操作。前者即为当前屏渲染，后者为离屏渲染。

与当前屏渲染相比，离屏渲染：

- 在变化的场景下，因为离屏渲染需要创建一个新的缓冲区，且需要多次切换上下文环境，所以代价很高。

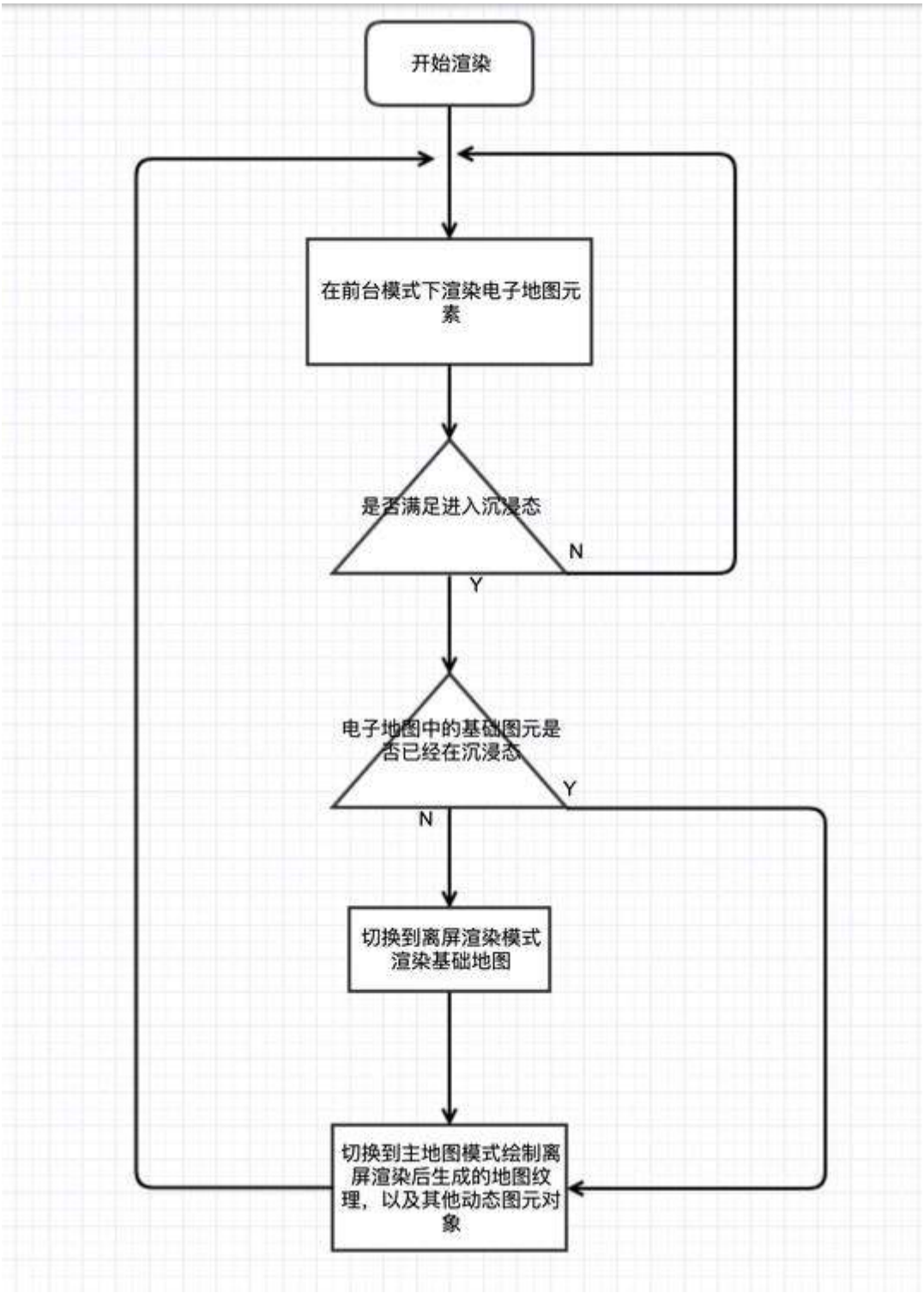
- 在稳定的场景下，离屏渲染可以采用一张纹理进行渲染，所以性能较当前屏渲染有较大提升。

从上述对比可以看出，在稳定场景下使用离屏渲染的优势较大。但因为地图状态随时都在变化，所以地图渲染通常处于前台动态渲染状态。那么有没有相对稳定的场景呢？答案是肯定的，我们将地图的状态分为沉浸态和非沉浸态。顾名思义，在地图处于变化状态的称为非沉浸态，进入稳定状态称为沉浸态。

进入沉浸态的地图，为我们使用离屏渲染提供了条件。经过统计，地图处于前台状态的场景下，沉浸态时间基本上和非沉浸态时间相当，这样我们采用一张纹理，即可将处于非沉浸态场景下的地图渲染出来，大大降低了系统开销。在鹰眼图，矢量路口大图等特定的视图场景下，地图基本上均处于沉浸态。所以这些视图下采用离屏渲染技术进行优化，取得的收益将是巨大的。

3.工程实践

将以上的技术优化原理，代入到实际的导航应用中，流程如下：

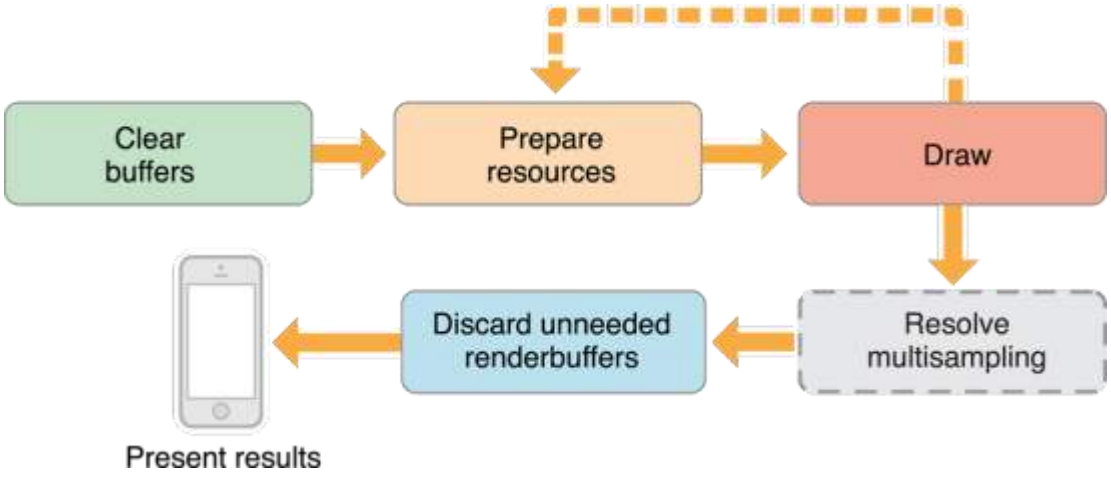


离屏渲染通常使用 FBO 实现。FBO 就是 Frame Buffer Object，它可以让我们的渲染不渲染到屏幕上，而是渲染到离屏 Buffer 中。但是通常的离屏渲染 FBO 对象不具备抗锯齿能力，因此开启了全屏抗锯齿能力的 OpenGL 应用程序，如果采用离屏渲染 FBO 对象进行离屏渲染，会出现锯齿现象。而在非沉浸态地图的状态下，是开启全屏抗锯齿能力的，所以我们

必须使用具备抗锯齿能力的离屏渲染技术来进行地图渲染技术优化。

4.抗锯齿离屏渲染技术简述

本节以 iOS 系统为例，对抗锯齿能力的离屏渲染技术进行简述。iOS 系统对 OpenGL 进行了深度定制，其抗锯齿能力就是建立在 FBO 基础上的。如下图所示，IOS 基于对抗锯齿的帧缓存（FBO）对象进行操作，从而达到全屏抗锯齿的目的：



接下来具体介绍抗锯齿 FBO 的创建步骤：

- 创建 FBO 并绑定：

```
GLuint sampleFramebuffer;
glGenFramebuffers(1, &sampleFramebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, sampleFramebuffer);
```

- 创建一个颜色帧缓冲区，在显存中开辟一个具有抗锯齿能力的对象，并将颜色缓冲区挂载到开辟的对象上。创建一个深度模版渲染缓冲区，开辟具有抗锯齿能力的显存空间，并和帧缓冲区进行绑定：

```
GLuint sampleColorRenderbuffer, sampleDepthRenderbuffer;
glGenRenderbuffers(1, &sampleColorRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, sampleColorRenderbuffer);
glRenderbufferStorageMultisampleAPPLE(GL_RENDERBUFFER, 4, GL_RGBA8_OES, width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER, sampleColorRenderbuffer);
glGenRenderbuffers(1, &sampleDepthRenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, sampleDepthRenderbuffer);
glRenderbufferStorageMultisampleAPPLE(GL_RENDERBUFFER, 4, GL_DEPTH_COMPONENT16, width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, sampleDepthRenderbuffer);
```

- 测试创建的环境是否正确，避免如显存空间不足等造成创建失败的可能：

```
1.  GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER) ;
2.  if(status != GL_FRAMEBUFFER_COMPLETE) {
3.    return false;
4.  }
```

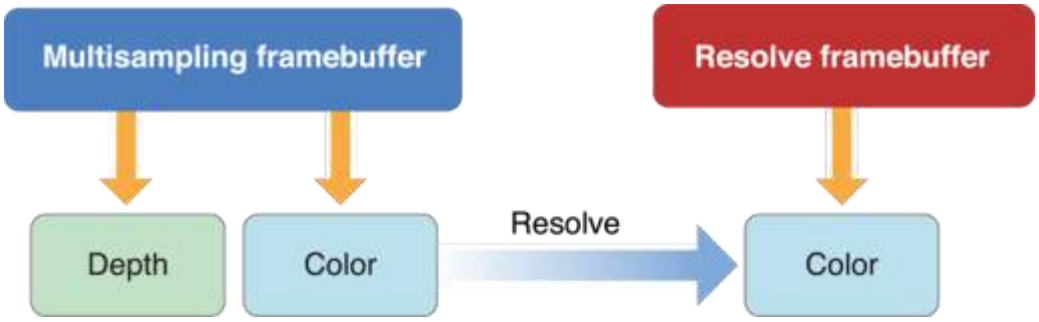
至此，一个具备抗锯齿能力的离屏 FBO 已创建好，下面将应用这个 FBO，步骤如下：

- 先清除抗锯齿帧缓存空间重的内容：

```
1.  glBindFramebuffer(GL_FRAMEBUFFER, sampleFramebuffer);
2.  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
3.  glViewport(0, 0, framebufferWidth, framebufferHeight);
```

开始进行一系列的渲染函数操作，比如准备顶点数据，纹理数据，VBO，IBO，矩阵，状态等，并执行一系列的渲染指令，选择指定的 shader，及其传输数据状态；

FBO 不是一个具备直接渲染能力的帧缓存空间，在执行完 2 的操作之后，需要将抗锯齿的 FBO 内渲染的内容通过合并每个像素，转换到屏幕渲染所在的帧缓存空间去。原理如下图所示：



代码如下：

```
1.  glBindFramebuffer(GL_DRAW_FRAMEBUFFER_APPLE, resolveFrameBuffer);
2.  glResolveMultisampleFramebufferAPPLE();
3.  glBindFramebuffer(GL_READ_FRAMEBUFFER_APPLE, sampleFramebuffer);
```

以上操作完成后，需要进行一些 Discard 步骤，将一些原先在当前帧缓存中的内容忽略掉：

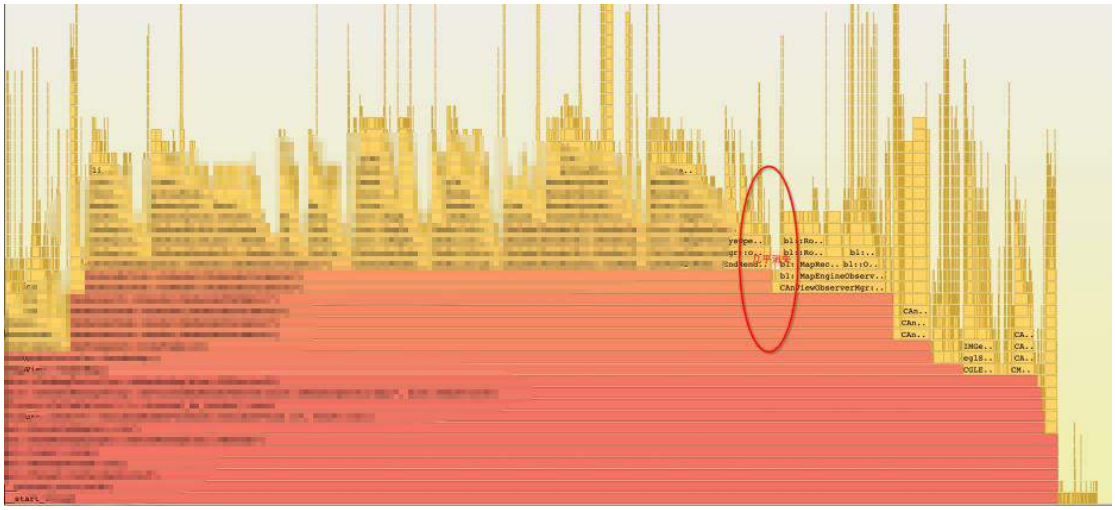
```
1.  glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
2.  [context presentRenderbuffer:GL_RENDERBUFFER];
```

Android 系统基本思路一致，需要采用 gles3.0 接口提供的抗锯齿能力来进行渲染，在此不做展开。

5.优化对比

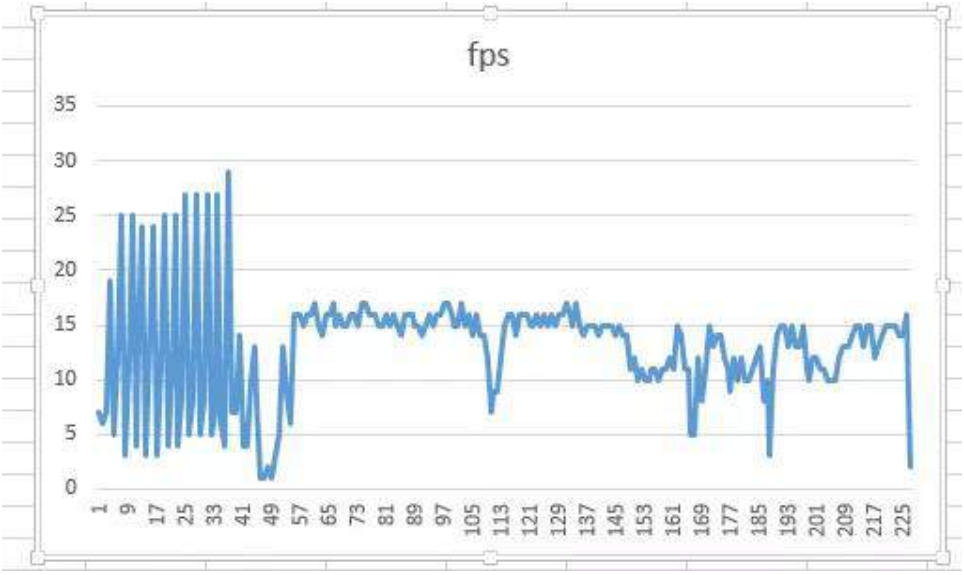
优化前的鹰眼图渲染耗时火焰图如下：

优化后的鹰眼图渲染耗时火焰图如下：



从前后对比图可以看出，鹰眼图渲染的耗时，几乎已经消失不见。

从系统的渲染帧率上进一步得到验证。从下图可以看出帧率已经恢复到与不显示鹰眼图的情况相当：



需要注意的是，全屏抗锯齿损耗资源，除了增加额外的显存空间，抗锯齿过程中也会产生一定的耗时。所以在取得收益的同时，也需要衡量其产生的代价，需要具体问题具体分析。在本案例中，如对比结果所示，采用抗锯齿离屏渲染技术的优化产生的收益远远高于付出的代价。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

高德引擎构建及持续集成技术演进之路

作者：陈志承

1.背景

由于导航应用中的地图渲染、导航等核心功能对性能要求很高，所以高德地图客户端中大量功能采用 C++ 实现。随着业务的飞速发展，仅地图引擎库就有 40 多个模块，工程配置极其复杂，原有的构建及持续集成技术已无法满足日益增长的需求变化。

除了以百万计的代码行数带来的复杂度外，高德地图客户端中的 C++ 引擎库工程（以下简称引擎库）的构建和持续集成还面临以下几个挑战：

- **支持多团队协作**：多团队意味着多操作系统多 IDE ，降低不同操作系统和不同 IDE 下的工程配置的难度是重点要解决的难题之一。

- **支持多业务线定制**：引擎库为手机、车机、开放平台等业务线提供支持，而各个业务线的诉求不同，所以需要具备按功能构建的能力。

- **支持车机环境**：在诸多业务线中，高德地图有一个非常特殊的业务线，即车机（AMAP AUTO）。车机直接面对各大车厂和众多设备商，环境多为定制化，构建工具链各式各样。如果针对每个车机环境都定制一套构建配置文件，那么其维护成本将非常高，所以如何用一套构建配置满足车机的多样化构建需求成为亟需解决的问题。

此外，由于历史原因，引擎库中源码和依赖库混杂，都存放于 Git 仓库中，这样会带来两个问题：

- 随着构建次数不断增加，Git 仓库越来越大，代码与依赖库检出越来越慢，极大影响本地开发以及打包效率。

- 缺乏统一管理，依赖关系混乱，经常出现因为依赖问题而导致的构建失败，或者虽然构建成功但运行时发生错误的情况。

上述的挑战和历史遗留问题严重阻碍了研发效能的提升。为此，我们对现有的构建及持续集成工具进行了深入的研究和分析，并结合自身的业务特性，最终发展出高德地图 C++ 本地构建工具 Abtor 和持续集成工具 Amap CI 。

2.本地构建

现有工具分析

C++ 是一门靠近底层的语言。不同的硬件、操作系统、编译器，再加上交叉编译，导致 C++ 构建的难度非常高。针对这些问题，C++ 社区涌现出许多优秀的构建工具，比如大名鼎鼎的 Make 和 CMake 。

Make，即 GNU Make ，于 1988 年发布，是一个用来执行 Makefile 的工具。Makefile 的基本语法包括目标、依赖和命令等。使用过程中，当某些文件变了，只有直接或者间接依赖这些文件的目标才需要重新构建，这样大大提升了编译速度。

Make 和 Makefile 的组合可以看作项目管理工具，但它们过于基础，在跨平台的使用方面有很高的门槛和较多的限制，此外大项目的构建还会遇到 Makefile 严重膨胀的问题。

CMake 产生于 2000 年，是一个跨平台的编译、测试以及打包工具。它将配置文件转化为 Makefile ，并运行 Make 命令将源码编译成可执行程序或库。CMake 属于 Make 系列，配置文件比 Makefile 具有可读性，支持跨平台构建，构建性能高。

但是 CMake 也有两项明显不足，一是配置文件的复杂度远高于其它现代语言，对于 CMake 语法初学者有一定的学习成本，二是与不同 IDE 的配合使用不够友好。

可以看出 Make 和 CMake 的抽象度还是比较低，从而对构建人员的要求过高。为了降低构建成本，C++ 社区又出现了一些新的 C++ 构建工具，现在使用较广泛的包括 Google 的 Bazel 和 Ninja ，以及 SCons 。这些工具的特点和不足如下：

	特点	不足
Bazel	•完整的C++构建解决方案； •适用于庞大代码库的项目； •支持多平台； •构建性能高；	•没有完全实现跨平台，目前支持Linux和macOS，要移植其它平台代价较高；
Ninja	•专注构建速度的小型构建系统；	•构建文件编写难度较大，需要借助一些更高级别的构建工具来生成，使用成本高；
SCons	•采用Python语言编写，内置函数和简单的扩展函数的功能，配置文件编写简单；	•对于大型项目有点力不从心，构建性能较慢；

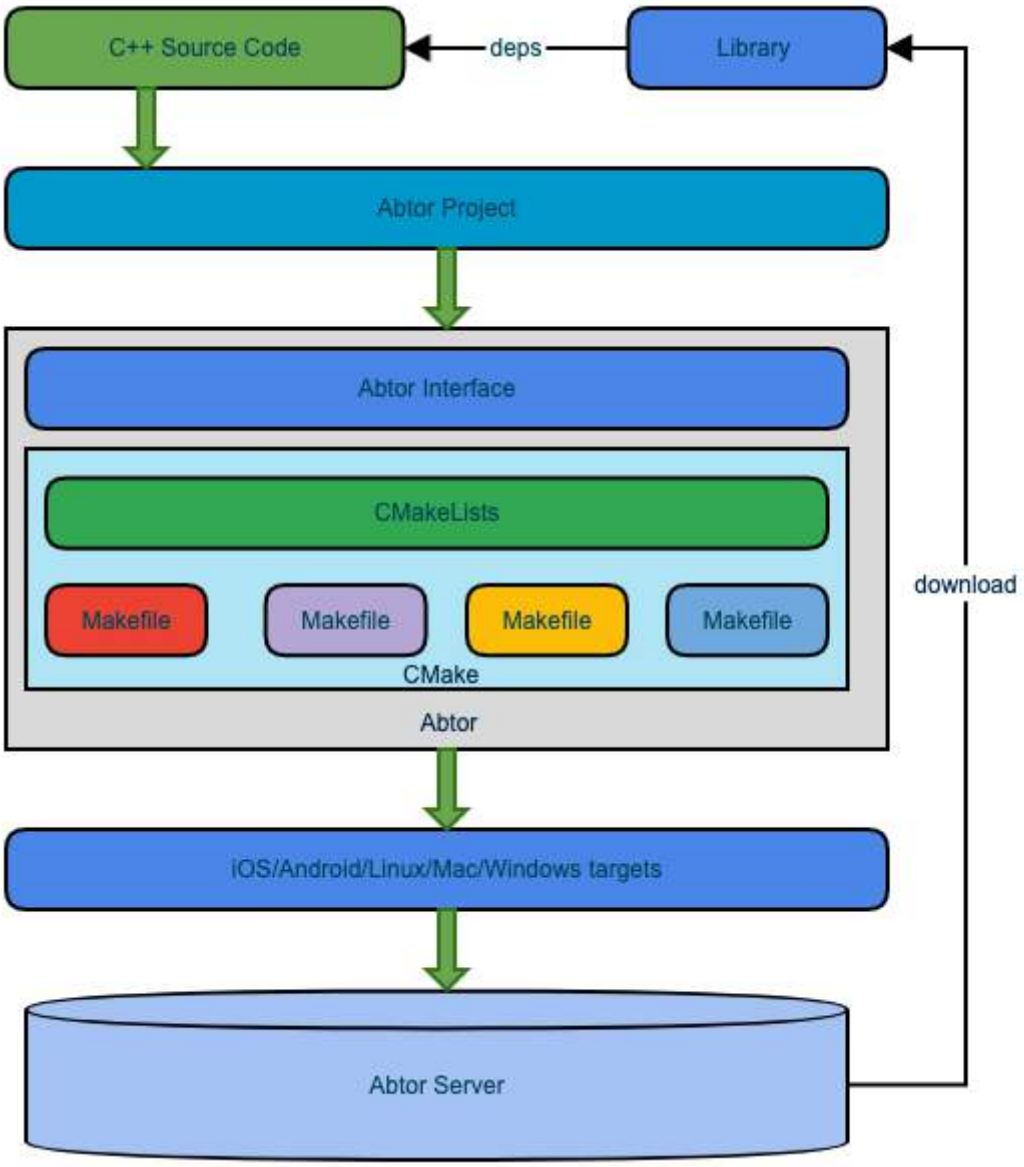
经过上述对现有 C++ 构建工具的研究和分析，可以得出每个工具既有所长又有不足的结论。再考虑到高德地图引擎库工程面临的挑战和历史遗留问题，我们发现以上工具没有一个可以完美契合业务需求，且改造成本非常高，所以我们决定基于 CMake 自建 C++ 本地构建工具，即现在引擎库工程使用的 Abtor 。

Abtor

首先，我们需要解释一个问题，即 **Abtor 是什么？**

Abtor 是一个 C++ 跨平台构建工具。Abtor 采用 Python 编写构建脚本，生成 CMake 配置文件，并通过内置 CMake 组件生成构建文件，最终产出可执行程序或库。它抽象出构建描述，使得复杂的编译器和连接器对开发者透明；它提供强大的内置功能，从而有效的降低开发者编写构建脚本的难度。

其次，我们需要阐述一个问题，即 **Abtor 的构建流程是什么？**



如上图所示，Abtor 构建的整个流程为：

- 编写 Abtor 构建脚本
- 解析 Abtor 构建脚本
- 检测依赖关系，识别冲突，并从阿里 OSS 上下载所需依赖
- 生成 CMakeLists.txt，并通过内置的 CMake 生成 Makefile 文件
- 编译，链接，生成对应平台的目标文件
- 将目标文件发布到阿里 OSS

除此之外，还增加了控制访问发布库权限的功能，用于保证发布库的安全。

最后，我们需要探讨一个问题，即 **Abtor 解决了什么？**

在开篇背景中，我们提到阻碍研发效能的一些挑战和问题，这就是 Abtor 需要解决的，所以 Abtor 具备以下特点：

- 更广泛的跨平台：支持 MacOS 、iOS、Android、 Linux、Windows、QNX 等平台。
- 有效的多团队协作：较好得与 IDE 结合，并支持一套配置生成不同项目工程，从而达到工程配置一致化。
- 高定制化：支持工具链及构建参数的灵活定制，并通过内置工具链配置为车机复杂的构建提供强有力的支持。
- 源码与依赖分离：支持源码依赖与库依赖，源码通过 Git 管理，构建库存放于阿里云，源码与产物完全分离。
- 良好的构建性能：快速构建大型项目，从而提高开发效率。

从上述特点可看到，Abtor 有效地解决了已有的构建工具在高德业务中面临的痛点。但是冰冻三尺，非一日之寒，Abtor 也是在不断地完善中，下面重点介绍一下 Abtor 发展过程中遇到的三个问题。

工程配置一致化

在日常开发过程中，工程项目的调试工作尤为重要。高德地图客户端中的 C++ 引擎库工程的开发人员涉及几个部门和诸多小组。这些组擅长的技术栈，使用的平台和习惯的开发工具都大为不同。如果针对每一个平台都单独建立相应的工程配置，那么工作量及后续维护成本可想而知。

基于以上原因，Abtor 内置与 IDE 结合的功能，即开发者可以通过一套配置并结合 Abtor 命令一键生成工程配置，实现在不同平台的工程配置的一致化。工程配置一致化为引擎库开发带来以下几个收益：

- 命令简单，降低学习成本，开发者只需熟记 abtorw project [IDE name]。
- 配置文件不会因为 IDE 的增加而迅速膨胀，开发者更换构建命令，比如 abtorw project xcode 或者 abtorw project vs2015，即可生成对应的项目工程。
- 有利于部门间的协作及新人的快速融入，开发者可以根据喜好选择 IDE 进行开发，大大提高开发效率。

目前 Abtor 支持的 IDE 有 Xcode、Android Studio、Visual Studio、Qt Creator、CLion 等。

复杂车机环境的构建

作为高德地图一条非常重要的业务线，车机面对的构建环境复杂多变，厂商往往会自行定制工具链。如果每接入一个设备，所有工程项目都需要修改配置文件，那么这个成本还是非常高的。为了解决这个问题，Abtor 提供两种做法：

- **内置工具链配置**：对于开发者完全透明，他不需要修改任何配置即可构建相应平台的产物。

- **支持自定义配置插件**：开发者按照规则编写配置插件，构建时 Abtor 会检测插件，并根据设置的工具链及构建参数进行构建。

除此之外，我们对所有的车机环境进行了 Docker 化处理，并通过 Docker 控制中心统一管理车机 Docker 环境的上线与下线，再利用上述 Abtor 的内置工具链配置功能内置车机构建参数，实现开发者无感知的环境切换等操作，有效地解决了复杂车机环境的构建问题。

基于 Docker 的车机构建主要步骤如下：

- 工具链安装：一般由厂商提供，我们会将该工具链安装到基础 Docker 镜像中。

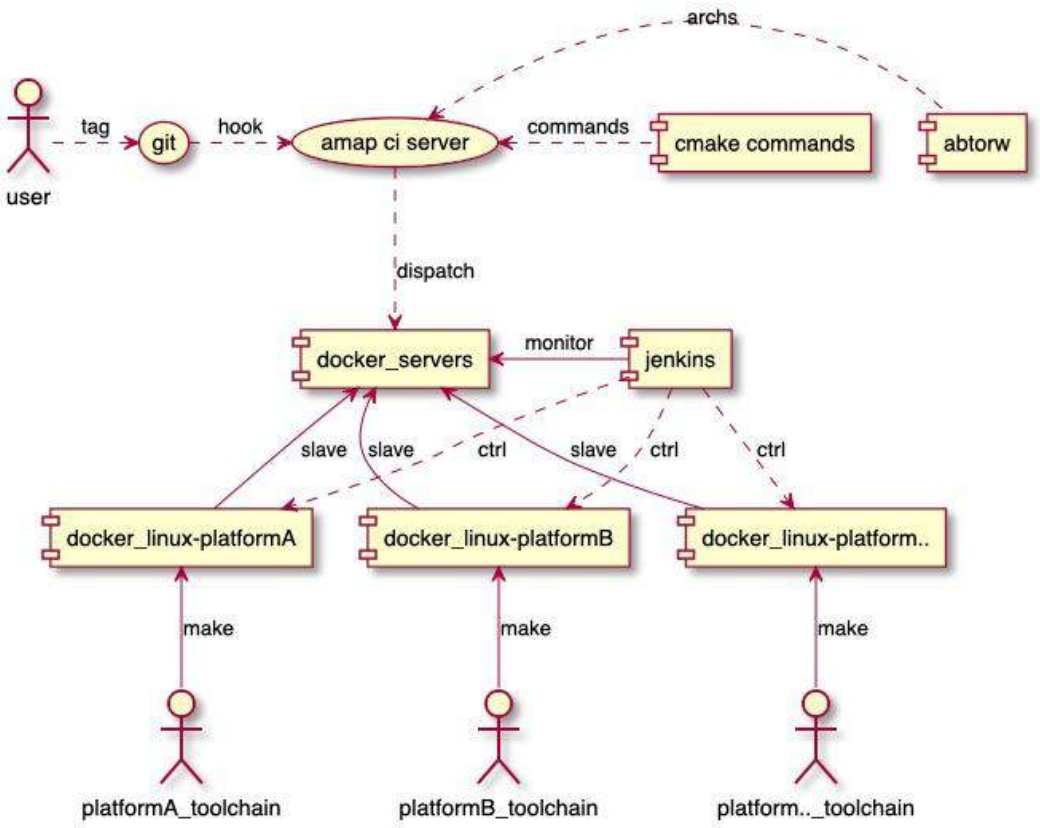
- Docker 发布：将镜像发布到 Docker 仓库。

- Abtor 适配：一次性适配工具链，并内置配置，开发者可通过 Abtor 版本升级使用该配置。

- 服务配置更新：由 Jenkins 管理，支持分批更新 Abtor 版本，不影响当下编译需求。

- 服务监控: 由 Jenkins 管理, 定时检测服务状态, 异常态的 Docker 服务将自动被重启。

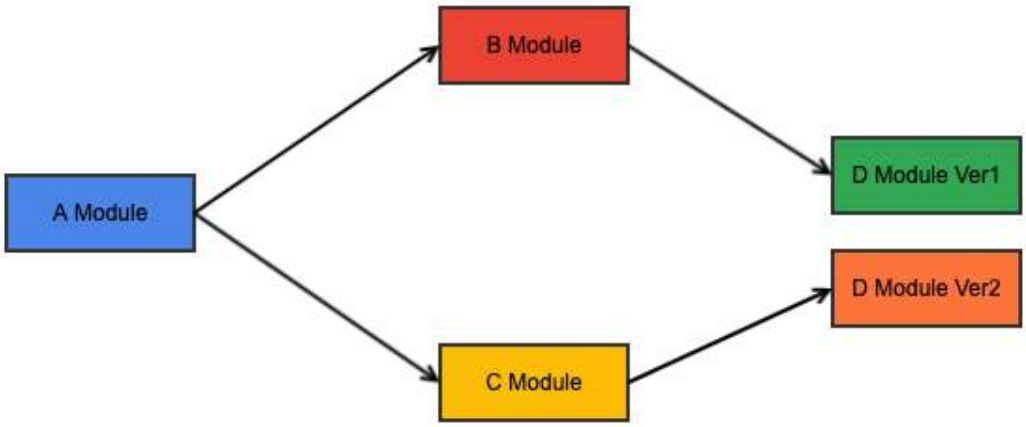
基于 Docker 的车机构建关系图如下:



依赖管理

依赖问题是所有构建工具都避免不了的问题，在这其中，菱形依赖问题尤为常见。如下图所示，假设 A 依赖了 B 和 C，B 和 C 又分别依赖了不同版本的 D，而 D 之间只存在很小的差异，这是可以编译通过的，但最终在运行时可能会出现意想不到的问题。

如果没有一种机制来检测，菱形依赖是很难被发现，而产生的后果又可能是非常严重的，比如导致线上出现大面积的崩溃等。所以依赖问题的分析与解决非常重要。



当下，市面上 Java 有比较成熟的依赖管理解决方案，如 Maven 等，但 C++ 并没有。为此 Abtor 专门建立依赖管理的机制来确保编译的正确性。

Abtor 的依赖管理是怎么做的呢？这里提供一个思路供大家参考：

- 建立 Abtor 服务端，用做库发布，以及处理依赖关系。
- 每个库在云端构建完，都会把库依赖的版本信息存放于云端数据库中。
- 本地/云端构建前 Abtor 会解析出所有依赖库的版本信息。
- 递归查找这些子库对应的依赖信息，即可罗列出所有依赖库的信息。
- 检测依赖库列表中是否存在不同版本号的相同库名：

如果没有相同库名，则继续执行构建；如果有相同库名，则说明依赖库之间存在冲突问题，此时中断构建，并显示冲突的库信息，待开发者解决完冲突后方可继续执行构建。

根据上述思路，我们保证了库依赖的一致性，避免了菱形依赖问题。另外，如果某个库被其它库所依赖且有更新，那么依赖它的库也应当随之构建，以确保依赖的一致性。这种对依赖构建的触发更新我们放到 Amap CI 上实现，在第三节会进行详细介绍。

工程实践

在介绍完 Abtor 的一些基本原理后，我们将介绍 Abtor 在日常开发中是如何使用的。

下图是 Abtor 工程项目的目录结构，其中有两类文件是开发者需要关心的，一类是源文件目录（src），一类是 Abtor 核心配置文件（abtor.proj）。

```
abtor_demo
├── ABTOR
│   └── wrapper
│       ├── abtor-wrapper.properties # 配置文件,可指定 Abtor 版本信息
│       └── abtor-wrapper.py         # 下载 Abtor 版本并调用 Abtor 入口函数
├── abtor.proj                       # Abtor 核心配置文件
├── abtorw                           # Linux/Mac 下的初始执行脚本
├── abtorw.bat                       # Windows 下的初始执行脚本
└── src
    └── main.c                       # 要编译的源文件
```

源文件目录的组织形式与 Make 系列构建工具没有太大区别。下面重点看一下 Abtor 核心配置文件：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 以下内容为 python 语法

# 指定编译的源码
```

```
header_dirs_list = [abtor_path("include")]    # 依赖的头文件目录
binary_src_list = [abtor_path("src/main.c")]  # 源码

cflags = " -std=c99 -W -Wall "
cxxflags = " -W -Wall "

# 指定编译二进制
abtor_ccxx_binary(
  name = 'demo',
  c_flags = cflags,
  cxx_flags = cxxflags,
  deps = ["add:1.0.0.0"],                     # 指定依赖的库信息
  include_dirs = header_dirs_list;
  srcs = binary_src_list
)
```

从上图可以看出，Abtor 核心配置文件具有以下几个特点：

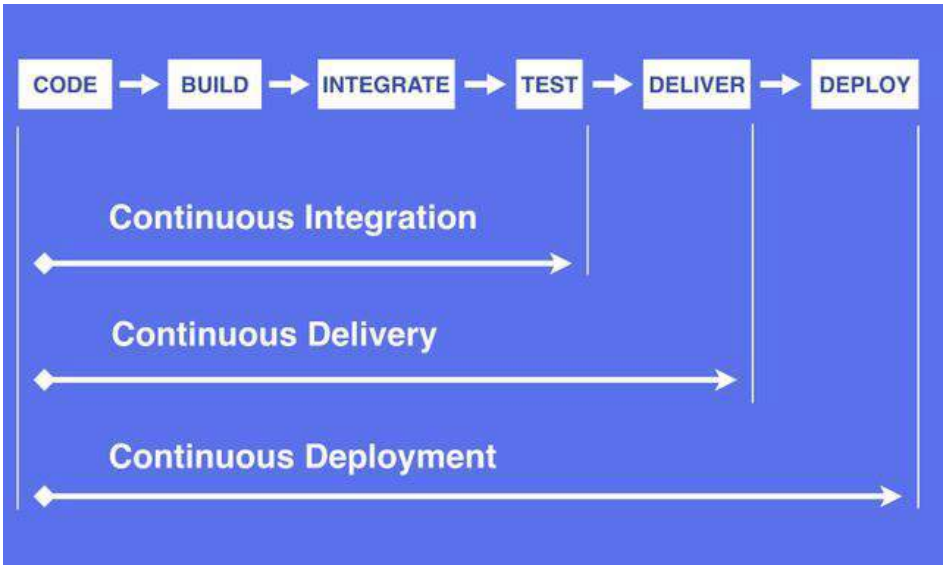
- 采用 Python 编写，易上手。
- 抽象类似 abtor_ccxx_binary 等的构建描述，降低使用门槛。
- 提供诸如 abtor_path 等的内置功能，提高开发效率。

通过以上的对源文件目录组织及 Abtor 核心配置文件编写，我们就完成了项目的 Abtor 配置化，接着可以通过 Abtor 内置的命令构建、发布或直接生成项目工程。我们相信，即使开发者不是很精通构建原理，依然可以无障碍地使用 Abtor 进行构建与发布。

3.持续集成

面临的问题

如下图所示，整个开发工作流程可分为几个阶段 :编码->构建->集成->测试->交付->部署。在使用 Abtor 解决本地构建遇到的一系列挑战与问题后，我们开始将目光转移到了整个持续集成阶段。



持续集成是指软件个人研发的部分向软件整体部分交付，频繁进行集成以便更快地发现其中的错误。它源自极限编程（XP），是 XP 最初的 12 种实践之一。对于引擎库来说，持续集成方案应该具备一次性批量构建不同平台不同架构目标文件的能力，同时也应当具备运维管理和消息管理的能力等。

最初高德引擎库使用 Jenkins 进行持续集成。因为引擎库开发采用在 Git 仓库上拉取分支的方式进行版本管理，所以每次版本迭代都需要手动建立 Jenkins Job，修改相应脚本，另外还需要额外搭建一个依赖库关系的 Jenkins Job 做联动编译。

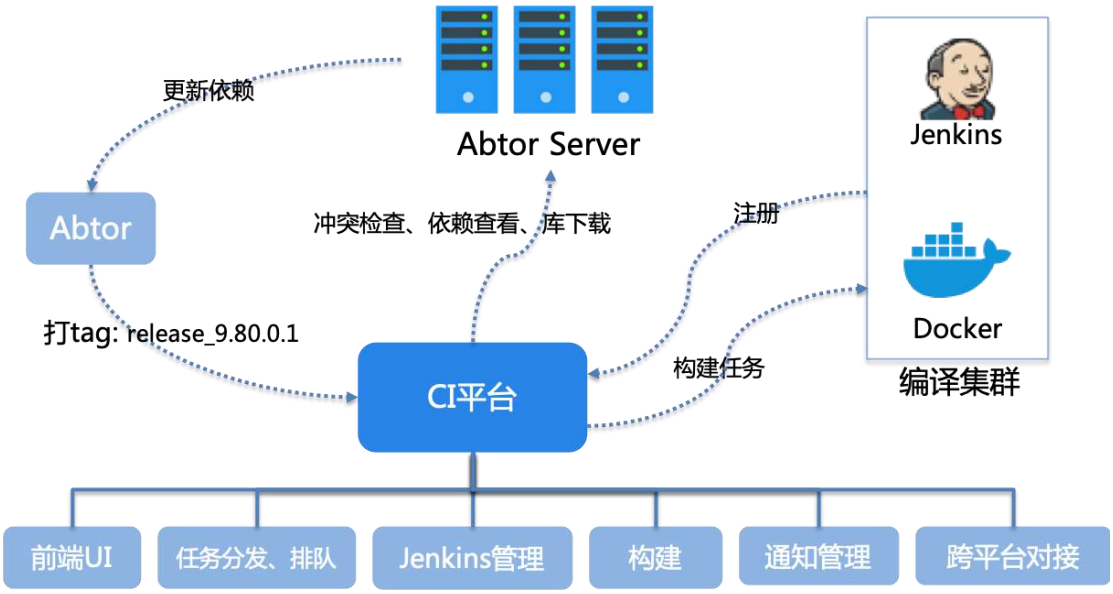
假设有 100 个项目，那么每个版本迭代都需要手动创建 101 个 Jenkins Job 。每次版本迭代都重复类似的操作，中间需要大量的协调工作，随着迭代版本越来越多，这些 Jenkins Job 变得不可维护。这是 Jenkins 持续集成方案在高德引擎库开发过程中遇到的非常严重的问题。

基于上述原因，我们迫切得需要这样一个持续集成系统：开发者不用维护 Jenkins，不需要部署构建环境，可以不了解构建细节，只需要通过某个触发事件就能够构建出所有平台的目标文件。于是我们决定自建持续集成平台，即 Amap CI。

Amap CI

Amap CI 平台使用 Gitlab 的 Git Webhook 实现持续集成。其中，Gitlab 接收开发者的 tag push 事件，回调 CI 平台的后台服务，然后后台服务根据构建机器的运行情况进行任务的分发。当构建任务较多时，CI 平台会等待直到有构建资源才进行任务的再分配。

Amap CI 平台由任务管理、Jenkins 管理、构建管理、通知管理、网页前端展示等几部分组成，整体架构图如下：



通过 Amap CI 平台，我们达到了以下几个目的：

- **可扩容**：所有构建机器通过注册的方式接入，构建机器扩容变得非常容易，减轻构建峰值带来的压力。
- **可视化**：Abtor Server 对于开发者是透明的。CI 平台与 Abtor Server 交互，为开发者提供冲突检查、依赖查看及库下载等可视化功能。
- **智能化**： CI 平台内置标准的 Jenkins Job 构建模板。开发者不感知这些模板，也无须做任何的修改。他们只需要通过 Git 提交一个 tag 信息即可实现全平台的构建，从而实现一键打 tag 构建。
- **自动化**：服务分析 Gitlab hook tag 的 push 信息并拉取代码，然后解析对应的配置文件和要构建的所有平台信息。根据这些信息 CI 平台分配构建机器，并执行 Abtor 命令进行构建与发布。所有这些皆自动完成。
- **即时性**：构建启动后会发送钉钉消息，消息除了概要信息外还附加了构建的链接等，开发者可以点击链接跟踪进度情况。构建成功或失败也都会发送消息，从而使得开发者可以及时进行下一步工作或处理构建错误。
- **可扩展**：CI 平台提供可扩展的对接方式，方便高德或阿里的其它平台对接，比如泰坦平台、CT 平台、Aone 等，从而实现编码、构建、测试和发布的开发闭环。

在上述目的中，对 Amap CI 平台最重要的是自动化，下面我们重点介绍一下自动化中的整树联动编译。

整树联动编译

在第二部分中我们提到了一个问题，即如果某个库被其它库所依赖且有更新，那么依赖它的

库也应当随之构建，以确保依赖的一致性，这是构建自动化的关键点之一。Amap CI 采用整树联动编译的方案来解决这个问题。

开发者在 CI 平台上建立对应的版本构建树，构建树中罗列了各个库之间的构建顺序，如下图所示。CI 平台会根据这棵构建树进行构建，被依赖的库优先构建，完成后再自动触发其上级的库构建，以此类推，最终形成一棵多叉树。在这棵多叉树上，从叶子节点开始按层级顺序逐级并发构建对应的库，这就是整树联动编译。

根据上述思路，我们保证了持续集成时的依赖一致性。开发者只需关心自己负责的库，打个 tag ，即可触发生成所有依赖该库的库，从而避免了依赖不一致的问题。

工程实践

在介绍完 Amap CI 的一些基本原理后，我们将介绍日常开发中应该如何使用 Amap CI。

一个新的工程项目在集成到 Amap CI 平台时，首先需要将 CI 平台的 web hook 网址增加到 Gitlab 的配置中，然后编写配置文件 CI_CONFIG.json ，至此一个新的项目已集成完成，非常简单。下面我们重点介绍一下 CI_CONFIG.json 。

CI_CONFIG.json 是核心配置文件，一次编写，无需再修改。它的结构如下：

```
CI_CONFIG.json DEMO:(json)
{
    "mail":"name@alibaba-inc.com",                  # 邮件通知
    "arch":"Android,iOS,Mac,Ubuntu64,Windows",      # 构建的平台
    "build_vars":"-v -V",                           # 构建参数
    "modules":{                                     # 构建的模块列表
        "amap":{                                    # 模块名为 amap
            "features":[                            # 功能列表
                {
                    "name":"feature1",              # 设置功能名为
feature1
```

```
                    "macro":"-DFEATURE_DEMO1=True"     # 宏控：FEATURE_DEMO1
                },
                {
                    "name":"feature2",                 # 设置功能名为 feature2
                    "macro":"-DFEATURE_DEMO2=True"     # 宏控：FEATURE_DEMO2
                }
            ]
        },
        "auto":{                                       # 模块名为 auto
            "features":[                               # 功能列表
                {
                    "name":"feature1",                 # 设置功能名为 feature1
                    "macro":"-DFEATURE_DEMO1=True"   # 宏控：FEATURE_DEMO1
                },
                {
                    "name":"feature3",               # 设置功能名为 feature3
                    "macro":"-DFEATURE_DEMO3=True" # 宏控：FEATURE_DEMO3
                }
            ]
        }
    }
}
```

从上图可以看出，配置文件描述了邮件通知、构建的平台、构建参数等信息，同时还为多业务线定制提供了良好的支持。

Amap CI 构建时读取上述文件，解析不同项目中配置的宏，并通过参数传递给 Abtor ，另一方面开发者在代码中利用这些宏进行代码隔离，构建时会根据这些宏选择对应的源码进行编译，从而支持多条业务线不同的需求，达到代码层面的最大复用。

目前 Amap CI 接入的项目数有几百个，编译的次数达到几十万次级别，同时在构建性能和构建成功率方面相比之前都有了大幅度的提高，现在仍旧不断有新的项目接入到构建平台上。可以说 Amap CI 平台是高德地图客户端 C++ 工程快速迭代开发的坚实保障。

4.未来展望

从 2016 年年中调研现有构建工具算起，到现在三年有余。三年很长，足以让我们将构想变成现实，足以让我们不断完善 Abtor ，足以让我们发展出 Amap CI 。三年又很短，对于一个系统开发生命周期而言，这仅仅是萌芽阶段，我们的征途才刚刚开始。

关于未来，我们的规划是向开发闭环方向发展，即打通编码、构建、集成、测试、交付和部署等各个环节中的链路，解决业务开发闭环的问题，实现整个开发流程自动化，进一步把开发者从繁琐的流程中解放出来，使得这些人员有精力去做更有价值的事情。

基于 LLVM 的源码级依赖分析方案的设计与实现

作者：瑞兽

1.导读

随着业务快速发展，移动客户端技术架构也从单一的工程配置，转向模块化、组件化、动态化方向发展。越来越多的业务模块被拆分成独立组件 bundle，进行独立开发、构建、测试、发布、运营，但这也面临着许多挑战：

- 如何保证众多的独立组件 bundle 能够准确无误快速集成到主工程、打包、提测、发布审核？
- 如果删除或更新某个独立组件 bundle，将会对剩余的哪些 bundle 有影响？
- 架构或产品优化时，哪些独立组件 bundle 可以删除/下线？

这就需要确定这些独立组件 bundle 之间的依赖关系。

2.依赖分析的定义

简单地说，通过某种技术手段获取到某个复杂系统中各个子系统之间相互关系，并将这种关系数据化、图像化处理的过程，即依赖分析。

3.常见的依赖分析方案

3.1 基于 Cocoapods 的依赖包分析

Cocoapods 是 iOS 业界提供，开源的、事实上的依赖管理标准工具，其 Podfile.lock 及 podspec 文件中均有显式的记录各个组件之间的依赖关系，因此只需要分析这些文件即可获取到依赖关系。

3.2 基于#include 和#import 头文件的依赖分析

众所周知，当某个源码文件 A 依赖另一个源码文件 B 时，必定会在 A 文件头部显式的添加上#include 和#importB。因此只需要扫描所有源码文件中的头文件引用关系即可获取到依赖关系。

3.3 基于 nm、otool 等命令行工具的符号依赖分析

nm 和 otool 常用于分析二进制文件中的符号信息，通过符号建立依赖关系。

3.4 三种符号依赖分析比较

三种方案各有优缺点：

方案	优点	缺点	分析时机	难度
Cocoapods	简单直观，业内基础方案	分析粒度大(以 bundle 为单位)	编译前	简单
头文件引用	简单直观	分析粒度中(以文件为单位)，存在无效、循环依赖问题	编译前	简单
nm/otool	简单直观	分析粒度细(以符号为单位)，编译混淆或优化(strip)的库查不到符号信息	编译后	简单

本文从编译原理角度，设计一种新的源码级别依赖分析方案。

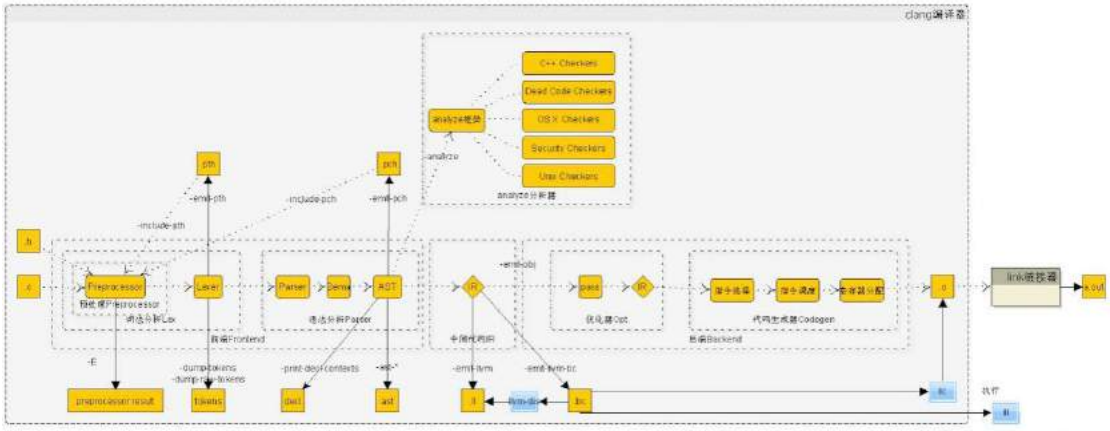
4.基于 LLVM 的依赖分析方案

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

LLVM 项目是一系列分模块、可重用的编译工具链。它提供了一种代码编写良好的中间表示(IR)，可以作为多种语言的后端，还可以提供与编程语言无关的优化和针对多种 CPU 架构的代码生成功能，举个例子来说明整个 LLVM 的编译过程：

```
1.  // main.m
2.  #include <stdio.h>
3.  #define kPeer 3
4.  int main(int argc, const char * argv[]) {
5.      int a = 1;
6.      int b = 2;
7.      int c = a + b + kPeer;
8.      printf("%d",c);
9.      return 0;
10. }
11.
12. // 执行命令 clang -ccc-print-phases main.m 输出
13. 0: input, "main.m", objective-c
14. 1: preprocessor, {0}, objective-c-cpp-output
15. 2: compiler, {1}, ir
16. 3: backend, {2}, assembler
17. 4: assembler, {3}, object
18. 5: linker, {4}, image
19. 6: bind-arch, "x86_64", {5}, image
```

整体流程如图示：



4.1 预处理(Preprocessor)阶段

预处理包括：条件编译、源文件包含、宏替换、行控制、抛错、杂注和空指令。

```
clang -E main.m
```

4.2 词法分析(Lexer)阶段

行词法分析：将预处理过的代码转化成一个个 Token，比如左括号、右括号、等于、字符串等等。

```
clang -fmodules -fsyntax-only -Xclang -dump-tokens main.m
```

4.3 语法分析(AST)阶段

行语法分析：根据当前语言的语法，验证语法是否正确，并将所有节点组合成抽象语法树(AST)。

```
clang -fmodules -fsyntax-only -Xclang -ast-dump main.m
```

4.4 中间代码(IR)生成阶段

CodeGen 负责将语法树从顶至下遍历，翻译成中间代码 IR，IR 是 LLVM Frontend 的输出，也是 LLVM Backerend 的输入，桥接前后端。

```
clang -S -fobjc-arc -emit-llvm main.m -o main.ll
```

4.5 代码优化(Opt)阶段

例如 Xcode 中开启了 bitcode, 那么苹果后台拿到的就是这种中间代码, 苹果可以对 bitcode 做进一步的优化。

`clang-emit-llvm-c main.m-o main.bc`

4.6 代码生成器(CodeGen)阶段

```
// 生成汇编代码
clang-S-fobjc-arc main.m-o main.s

// 生成目标文件
clang-fmodules-c main.m-o main.o
```

4.7 链接成可执行文件

`clang main.o-o main`

其中 IR 代码生成(CodeGen)阶段，会遍历整个 AST 语法树，在此处插桩记录下函数名 + 行号 + 文件路径 + 源码 hash 值等信息，即可生成依赖分析的元数据。

5.如何进行 LLVM 插桩

针对 iOS 端的代码编译，LLVM 前端使用 Clang 编译器，要在中间代码(IR)阶段插桩即要进行 Clang Plugin 开发。

5.1 准备 Clang 开发工具链

可以选择自行编译的 Clang 开发工具链，如下操作：

```
#!/bin/sh
cd /opt
sudo mkdir llvm
pushd llvm &&
git clone -b release_80 git@github.com:llvm-mirror/llvm.git llvm &&
git clone -b release_80 git@github.com:llvm-
mirror/clang.git llvm/tools/clang &&
git clone -b release_80 git@github.com:llvm-mirror/clang-tools-
extra.git llvm/tools/clang/tools/extra &&
git clone -b release_80 git@github.com:llvm-mirror/compiler-
rt.git llvm/projects/compiler-rt &&
popd &&
sudo mkdir -v llvm_build &&
pushd llvm_build &&
cmake -DCMAKE_INSTALL_PREFIX=/opt/llvm_release  \
      -DLLVM_TARGETS_TO_BUILD="X86;ARM;Mips;AArch64;WebAssembly" \
      -DCMAKE_BUILD_TYPE=Release                \
      -DLLVM_ENABLE_FFI=ON                      \
```

```
16.       -DLLVM_ENABLE_RTTI=ON                \
17.       -DLLVM_BUILD_TESTS=OFF               \
18.       -DLLVM_INCLUDE_TESTS=OFF             \
19.       -Wno-dev -G Ninja ../llvm            &&
20. ninja && ninja install && popd
```

也可以选择已编译好的 Clang 开发工具链，下载地址：http://releases.llvm.org/

5.2 编写 Clang 插件

Clang 插件实际上一个动态链接库，因此使用 Xcode 创建一个 dylib 工程，将编译器指定到准备好的 Clang 工具链上即可开始，如下图示：

Clang Plugin 通常的入口点是 FrontendAction。FrontendAction 是一个接口，它允许用户指定的 actions 作为编译的一部分来执行。为了在 AST clang 上运行工具，AST clang 提供了方便的接口 ASTFrontendAction，它负责执行 action。剩下的唯一部分是实现 CreateASTConsumer 方法，该方法为每个翻译单元返回一个 ASTConsumer。继承它们即可实现遍历 AST 语法树的功能：

类	功能
clang::RecursiveASTVisitor	遍历 AST 语法树的抽象基类
clang::PluginASTAction	基于 consumer 的 AST 前端 Action 抽象基类
clang::ASTConsumer	读取 AST 的抽象基类

识别 AST 语法树中的类名、方法名、调用关系，需使用 AST 中的以下类：

类	功能
clang::ObjCInterfaceDecl	记录 Object-C 类声明信息

clang::ObjCCategoryDecl	记录 Object-C 扩展类名信息
clang::ObjCMethodDecl	记录 Object-C 类方法声明信息
clang::ObjCImplDecl	记录 Object-C 类方法实现声明信息
clang::ObjCImplementationDecl	记录 Object-C 类方法实现信息
clang::ObjCPropertyDecl	记录 Object-C 类的属性声明信息
clang::ObjCProtocolDecl	记录 Object-C 协议声明信息
clang::ObjCMessageExpr	记录 Object-C 表达式信息

5.3 加载 Clang 插件

在编译参数 Other C/C++ Flag 中添加

`-Xclang -load -Xclang /opt/llvm_release/plugins/libXXXPlugin.dylib -Xclang -add-plugin -Xclang XXXPlugin`

5.4 举个例子

以下代码实现遍历 AST 语法树中的所有 C++类名，并打印出来的功能：

```
1.  #include "clang/AST/ASTConsumer.h"
2.  #include "clang/AST/RecursiveASTVisitor.h"
3.  #include "clang/Frontend/CompilerInstance.h"
4.  #include "clang/Frontend/FrontendAction.h"
5.  #include "clang/Tooling/Tooling.h"
6.
7.  using namespace clang;
8.
9.  class FindNamedClassVisitor
10.   : public RecursiveASTVisitor<FindNamedClassVisitor> {
11. public:
12.   explicit FindNamedClassVisitor(ASTContext *Context)
13.     : Context(Context) {}
14.
15.   bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
16.     llvm::outs() << "Found class: " << Declaration->getNameAsString() << "\n
    ";
17.     return true;
18.   }
19.
```

```
20. private:
21.   ASTContext *Context;
22. };
23. 
24. class FindNamedClassConsumer : public clang::ASTConsumer {
25. public:
26.   explicit FindNamedClassConsumer(ASTContext *Context)
27.     : Visitor(Context) {}
28. 
29.   virtual void HandleTranslationUnit(clang::ASTContext &Context) {
30.     Visitor.TraverseDecl(Context.getTranslationUnitDecl());
31.   }
32. private:
33.   FindNamedClassVisitor Visitor;
34. };
35. 
36. class FindNamedClassAction : public clang::ASTFrontendAction {
37. public:
38.   virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
39.     clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
40.     return std::unique_ptr<clang::ASTConsumer>(
41.         new FindNamedClassConsumer(&Compiler.getASTContext()));
42.   }
43. };
```

编译参数可使用 LLVM 为我们提供的 llvm-config 工具自动生成，执行

```
llvm-config --cxxflags --ldflags --system-libs --libs core
```

其余额外依赖库自行根据功能添加。

6.建立依赖关系元数据

通过加载定制化开发的 Clang Plugin，经过编译即可生成如下面格式的数据结构：

```
1.  {
2.      "+[GTMBase64 decodeBytes:length:]": {
3.          "call": [
4.              "+[GTMBase64 baseDecode:length:charset:requirePadding:]"
5.          ],
6.          "class": "GTMBase64",
7.          "filename": "/Sources/Internal/Encode/GTMBase64.m",
8.          "range": "11401-11553",
9.          "sourceCode": "{return [self baseDecode:bytes length:length charset:
    kBase64DecodeChars requirePadding:YES];}"
```

```
10.     }
11. }
```

其中：

key 值	描述
call	标识调用链上的方法列表
class	标识类名
filename	标识编译单元文件名
range	标识方法所在行号
sourceCode	标识方法的实现源码

基于这些依赖元数据，经过后台系统加工处理，就可以准确地知道某个组件 bundle 与其他组件之间的关系，实现一套基于 LLVM 的依赖分析方案。

7.小结

本文主要介绍了业内常见的依赖分析方案，并分享了一种基于 LLVM 的，从细粒度方法级别来实现依赖分析的方案，它能更准确反馈出各个独立组件 bundle 之间的关系，指导开发人员优化架构设计，可以应对未来“五独”技术进化带来的挑战。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

高德 JS 依赖分析工程及关键原理

作者：达诺

1.背景

高德 App 进行 Bundle 化后，由于业务的复杂性，Bundle 的数量非常多。而这带来了一个新的问题——Bundle 之间的依赖关系错综复杂，需要进行管控，使 Bundle 之间的依赖保持在架构设计之下。

并且，为了保证 Bundle 能实现独立运转，在业务持续迭代的过程中，需要逆向的依赖关系来迅速确定迭代的影响范围。同时，对于切面 API（即对容器提供的系统 API，类似浏览器中的 BOM API），也需要确定每个切面 API 的影响范围以及使用趋势，来作为修改或下线某个 API 的依据。

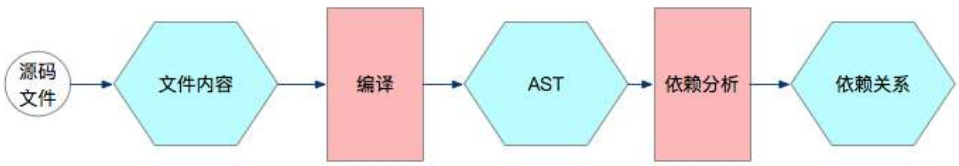
以组件库为例，由于组件会被若干业务项目所使用，我们对组件的修改会影响这些业务项目。在计划修改前，需要根据正向的依赖关系（业务依赖组件）来算出逆向的依赖关系——该组件被哪些地方所依赖，从而确定这个组件修改的影响范围。

比文件更高的维度，是 Bundle 间的依赖。我们有业务 Bundle，也有公共 Bundle。公共 Bundle 也分为不同层级的 Bundle。

对于公用 Bundle，业务 Bundle 可以依赖它，但公用 Bundle 不能反过来依赖业务 Bundle；同样的，底层的 Bundle 也禁止依赖上层封装的 Bundle。我们需要通过依赖分析，来确保这些依赖按照上述规则进行设计。

2.实现关键步骤

实现 JS 依赖分析，整个实现过程大致如下图所示：



下面挑一些关键步骤来展开介绍。

使用 AST 提取依赖路径

要做文件级别的依赖分析，就需要提取每个文件中的依赖路径，提取依赖路径有 2 个方法：

- 使用正则表达式，优点是方便实现，缺点是难以剔除注释，灵活度也受限。
- 先进行词法分析和语法分析，得到 AST（抽象语法树）后，遍历每个语法树节点，此方案的优点是分析精确，缺点是实现起来要比纯正则麻烦，如果对应语言没有提供 parser API（如 Less），那就不好实现。

一般为了保证准确性，能用第 2 个方案的都会用第 2 个方案。

以类 JS（.js、.jsx、.ts、.tsx）文件为例，我们可以通过 TypeScript 提供的 API ts.createSourceFile 来对类 JS 文件进行词法分析和语法分析，得到 AST：

```
const ast = ts.createSourceFile(
  abPath,
  content,
  ts.ScriptTarget.Latest,
  false,
  SCRIPT_KIND[path.extname(abPath)]
);
```

得到 AST 后，就可以开始遍历 AST 找到所有我们需要的依赖路径了。遍历时，可以通过使用 typeScript 模块提供的 ts.forEachChild 来遍历一个语法树节点的所有子节点，从而实现一个遍历函数 walk：

```
function walk (node: ts.Node) {
  ts.forEachChild(node, walk); // 深度优先遍历

  // 根据不同类型的语法树节点，进行不同的处理
  // 目的是找到 import、require 和 require.resolve 中的路径
  // 上面 3 种写法分为两类——import 声明和函数调用表达式
  // 其中函数调用表达式又分为直接调用（require）和属性调用（require.resolve）
  switch (node.kind) {
    // import 声明处理
    case ts.SyntaxKind.ImportDeclaration:
      // 省略细节……
      break;

    // 函数调用表达式处理
    case ts.SyntaxKind.CallExpression:
      // 省略细节
      break;
  }
}
```

通过这种方式，我们就可以精确地找到类 JS 文件中所有直接引用的依赖文件了。

当然了，在 case 具体实现中，除了用户显式地写依赖路径的情况，用户还有可能通过变量的方式动态地进行依赖加载，这种情况就需要进行基于上下文的语义分析，使得一些常量可以替换成字符串。

但并不是所有的动态依赖都有办法提取到，比如如果这个动态依赖路径是 Ajax 返回的，那就没有办法了。不过无需过度考虑这些情况，直接写字符串字面量的方式已经能满足绝大多数场景了，之后计划通过流程管控+编译器检验对这类写法进行限制，同时在运行时进行收集报警，要求必需显式引用，以 100% 确保对切面 API 的引用是可以被静态分析的。

建立文件地图进行寻路

我们对于依赖路径的写法，有一套自己的规则：

- 引用类 JS 文件支持不写扩展名。
- 引用本 Bundle 文件，可直接只写文件名。
- 使用相对路径。
- 引用公用 Bundle 文件，通过 @${bundleName}/${fileName} 的方式引用，fileName 同样是直接只写该 Bundle 内的文件名。

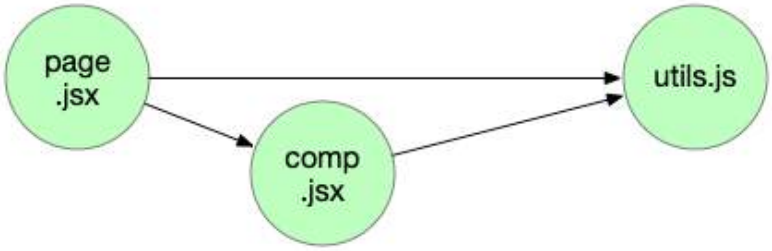
这些方式要比 CommonJS 或 ECMAScript Module 的规划要稍复杂一些，尤其是「直接只写文件名」这个规则。对于我们来说，需要找到这个文件对应的真实路径，才能继续进行依赖分析。

要实现这个，做法是先构建一个文件地图，其数据结构为 { [fileName]: 'relative/path/to/file' } 。我使用了 glob 来得到整个 Bundle 目录下的所有文件树节点，筛选出所有文件节点，将文件名作为 key，相对于 Bundle 根目录的路径作为 value，生成文件地图。在使用时，「直接只写文件名」的情况就可以直接根据文件名以 O(1) 的时间复杂度找到对应的相对路径。

此外，对于「引用类 JS 文件支持不写扩展名」这个规则，需要遍历每个可能的扩展名，对路径进行补充后查找对应路径，复杂度会高一些。

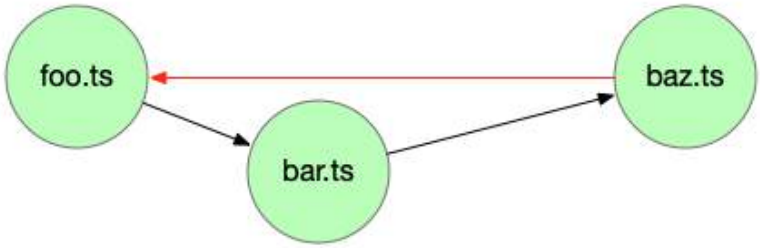
依赖是图的关系，需先建节点后建关系

在最开始实现依赖关系时，由于作为前端的惯性思维，会认为「一个文件依赖另一些文件」是一个树的关系，在数据结构上就会自然地使用类似文件树中 children: Node[] 的方式——链式树结构。而实际上，依赖是会出现这种情况的：



如果使用树的方式来维护，那么 utils.js 节点就会分别出现在 page.jsx 和 comp.jsx 的 children 中，出现冗余数据，在实际项目中这种情况会非常多。

但如果仅仅是体积的问题，可能还没那么严重，顶多费点空间成本。但我们又会发现，文件依赖还会出现这种循环依赖情况：



写 TypeScript 时在进行类型声明的时候，就经常会有这样循环依赖的情况。甚至两个文件之间也会循环依赖。这是合理的写法。

```
const fooTs = new Node({
  name: 'foo.ts',
  children: [
    new Node({
      name: 'bar.ts',
      children: [
        new Node({
          name: 'baz.ts',
          children: [
            new Node({
              name: 'foo.ts', // 和最顶的 foo.ts 是同一个
              children: [...] // 无限循环……
            })
          ]
        })
      ]
    })
  ]
})
```

但是，这种写法对于直接使用链式树结构来说，如果创建链式树的算法是「在创建节点时，先创建子节点，待子节点创建返回后再完成自身的创建」的话，就不可能实现了，因为我们会发现，假如这样写就会出现无限依赖：

此问题的根本原因是，这个关系是图的关系，而不是树的关系，所以在创建这个数据结构时，不能使用「在创建节点时，先创建子节点，待子节点创建返回后再完成自身的创建」算法，必须把思路切换回图的思路——先创建节点，再创建关系。

采用这种做法后，就相当于使用的是图的邻接链表结构了。我们来看看换成「先创建节点，再创建关系」后的写法：

```
// 先创建各节点，并且将 children 置为空数组
const fooTs = new Node({
  name: 'foo.ts',
  children: []
});

const barTs = new Node({
  name: 'bar.ts',
  children: []
});

const bazTs = new Node({
  name: 'baz.ts',
  children: []
});


// 然后再创建关系
fooTs.children.push(barTs);
barTs.children.push(bazTs);
bazTs.children.push(fooTs);
```

使用这种写法，就可以完成图的创建了。

但是，这种数据结构只能存在于内存当中，无法进行序列化，因为它是循环引用的。而无法进行序列化就意味着无法进行储存或传输，只能在自己进程里玩这样子，这显然是不行的。

所以还需要对数据结构进行改造，将邻接链表中的引用换成子指针表，也就是为每个节点添加一个索引，在 children 里使用索引来进行对应：

```
const graph = {
  nodes: [
    { id: 0, name: 'foo.ts', children: [1] },
```

```
    { id: 1, name: 'bar.ts', children: [2] },
    { id: 2, name: 'baz.ts', children: [0] }
  ]
}
```

这里会有同学问：为什么我们不直接用 nodes 的下标，而要再添加一个跟下标数字一样的 id 字段？原因很简单，因为下标是依赖数组本身的顺序的，如果一旦打乱了这个顺序——比如使用 filter 过滤出一部分节点出来，那这些下标就会发生变化。而添加一个 id 字段看起来有点冗余，但却为后面的算法降低了很多复杂度，更加具备可扩展性。

用栈来解决循环引用（有环有向图）的问题

当我们需要使用上面生成的这个依赖关系数据时，如果需要进行 DFS（深度遍历）或 BFS（广度遍历）算法进行遍历，就会发现由于这个依赖关系是循环依赖的，所以这些递归遍历算法是会死循环的。要解决这个问题很简单，有三个办法：

- 在已有图上添加一个字段来进行标记

每次进入遍历一个新节点时，先检查之前是否遍历过。但这种做法会污染这个图。

- 创建一个新的同样依赖关系的图，在这个新图中进行标记

这种做法虽然能实现，但比较麻烦，也浪费空间。

- 使用栈来记录遍历路径

我们创建一个数组作为栈，用以下规则执行：

每遍历一个节点，就往栈里压入新节点的索引（push）；

每从一个节点中返回，则移除栈中的顶部索引（pop）；

每次进入新节点前，先检测这个索引值是否已经在栈中存在（使用 includes），若存在则回退。

这种方式适用于 DFS 算法。

3.总结

依赖关系是源代码的另一种表达方式，也是把控巨型项目质量极为有利的工具。我们可以利用依赖关系挖掘出无数的想象空间，比如无用文件查找、版本间变动测试范围精确化等场景。若结合 Android、iOS、C++ 等底层依赖关系，就可以计算出更多的分析结果。

目前，依赖关系扫描工程是迭代式进行的，我们采用敏捷开发模式，从一些简单、粗略的 Bundle 级依赖关系，逐渐精确化到文件级甚至标识符级，在落地的过程中根据不同的精确度来逐渐满足对精度要求不同的需求，使得整个过程都可获得不同程度的收益和反馈，驱使我们不断持续迭代和优化。

关于作者

叶俊星（花名达诺），网名 Jasin Yip，高级前端工程师，长期从事大前端技术架构优化等工作。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

高德 APP 全链路源码依赖分析工程

作者：达诺

1.背景

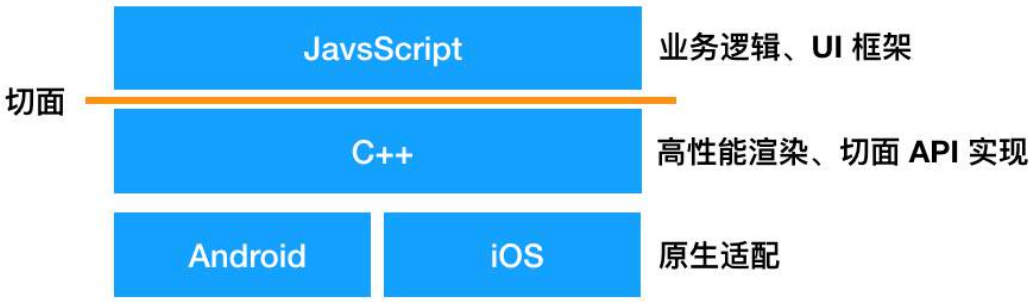
高德 App 经过多年的发展，其代码量已达到数百万行级别，支撑了高德地图复杂的业务功能。但与此同时，随着团队的扩张和业务的复杂化，越来越碎片化的代码以及代码之间复杂的依赖关系带来诸多维护性问题，较为突出的问题包括：

- 不敢轻易修改或下线对外暴露的接口或组件，因为不知道有什么地方对自己有依赖、会受到影响，于是代码变得臃肿，包大小也变得越来越大；
- 模块在没有变动的情况下，发布到新版本的客户端时，需要全量回归测试整个功能，因为不知道所依赖的模块是否有变动；
- 难以判断 Native 从业务实现转变为底层支撑的趋势是否合理，治理是否有效；

这些问题已经达到了我们必须开始治理的程度了，而解决此类问题的关键在于需要了解代码间的依赖关系。

2.高德 APP 平台架构

为了消除一些疑惑，在讨论依赖分析的实现前，先简单说明一下高德 APP 的平台架构，以便对一些名词和场景有一些背景了解。



高德 APP 从语言平台上可以分为 4 个部分，JS 层主要负责业务逻辑和 UI 框架；中间有 C++层做高性能渲染（主要是地图渲染），同时实现了一些切面 API，这样可以在双端只维护一套逻辑了；Android 和 iOS 层主要作为适配层，做一些操作系统接口的对接和双端差异化的（尽可能）抹平。

这里的切面是指 JS 层与 Native/C++ 层的分界线，这里会实现一些切面 API，也就是 JS 层与 Native/C++ 层交互的一系列接口，如蓝牙接口、系统信息接口等，由 Native/C++ 层来实现接口，然后往 JS 层暴露，由 JS 层调用。

3.基础实现原理

整个项目最基本也是最重要的数据就是依赖关系。所谓依赖关系，最简单的例子就是文件 A 依赖文件 B 的某个方法。

要将这个关系查出来，一般来说需要经过两个步骤。

第一步：编译源码，获得 AST

遍历所有源码，通过语法分析，生成抽象语法树（Abstract Syntax Tree, AST）。以 JS 扫描器为例，我采用了 typeScript 模块作为编译器，它同时支持 JS(X)、TS(X)，通过 ts.createSourceFile 来生成 AST。除 JS 外，iOS 采用的是 CLang，Android 采用的是字节码分析，C++ 采用的是符号表分析。

第二步：路径提取，依赖寻路

从 AST 上我们可以找到所有的引用和暴露表达式，以 JS 为例就是 import/ require 和 export/ module.exports。寻找表达式的方法就是递归地遍历所有语法节点，在 JS 中我采用了 TypeScript 编译器提供的 ts.forEachChild 来进行遍历，通过 ts.SyntaxKind 进行语法节点类型的识别。

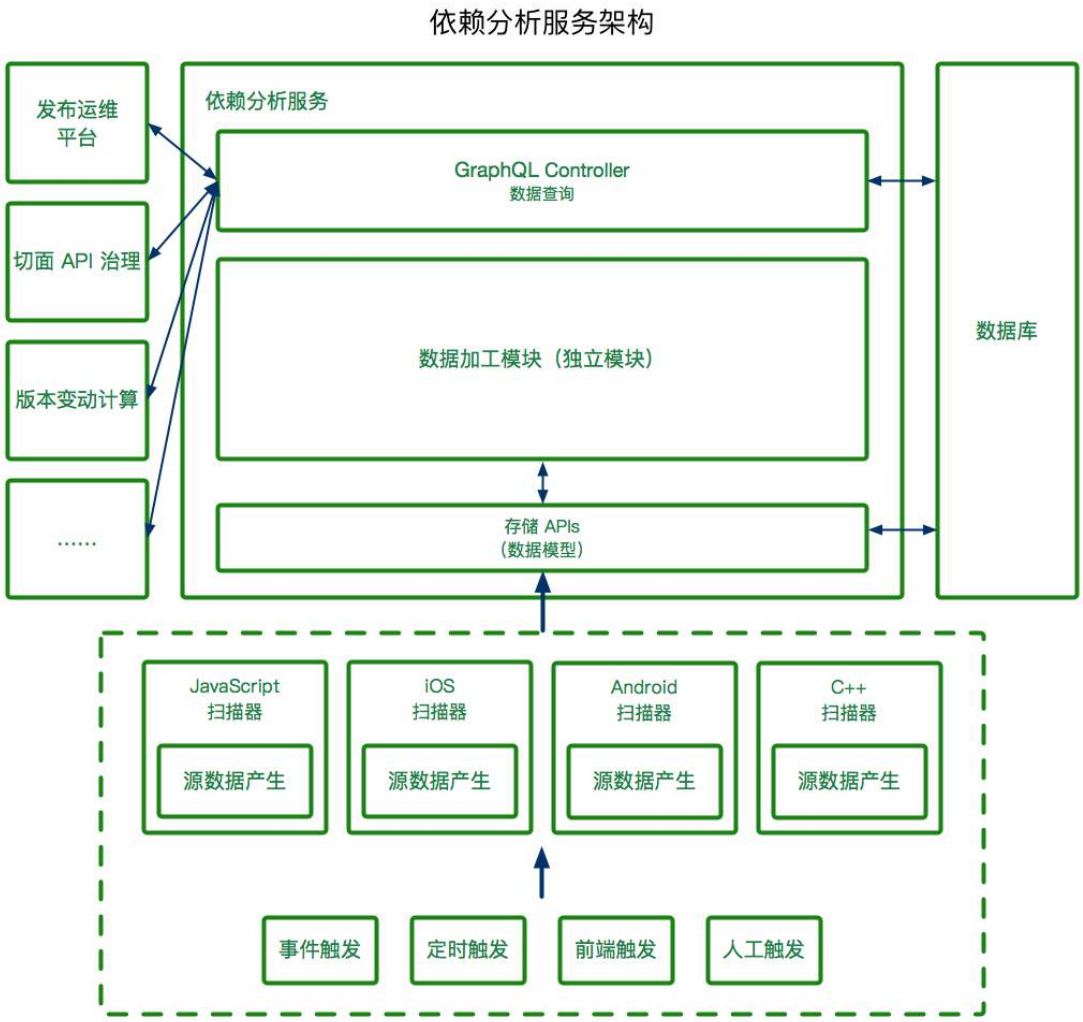
找到表达式后，通过依赖路径找到具体的依赖文件。以 JS 为例，我们可以通过 const { identifierName } = require('@bundleName/fileName') 的方式引用其它模块（bundleName）的某个文件（fileName）的某些标识符（identifierName），我们就需要根据这表达式来定位到具体的标识符。

跨切面的依赖会需要多做一步，需要将切面 API 分为调用侧和声明侧，在 JS 层通过 AST 分析出调用侧数据，在 Native/C++ 层分析出声明侧数据（对应到具体实现切面 API 的标识符），将调用侧和声明侧数据通过版本号关联到一起，即可实现全依赖链路贯通。

我们把这个关系以及一些元数据保存下来，就可以作为源数据来作数据分析了。

4.项目架构

整体项目架构如下：



我们使用 Node.js 和集团的 egg.js 框架搭建了本依赖分析工程服务，并且考虑到数据使用场景的多变性和多样性，我选用了 GraphQL 作为查询接口，输出我们定义的数据类型，由上层应用自行封装，如果出现多个上层应用同时需要类似的数据，我们也会进行整合复用。

其中数据加工模块是独立模块，由 Node.js 编写，支持其它项目复用，未来会计划在 IDE 等项目复用。

左侧是我们的数据消费方，这里只列举了几个；右侧是我们的数据库，用于储存分析结果；下侧是四端扫描器和触发器，四端分别对自己平台的源码进行源数据生产，触发器支持发布流程触发事件触发、定时触发、前端触发（应用侧前端，不是 Web 前端）和人工触发等。

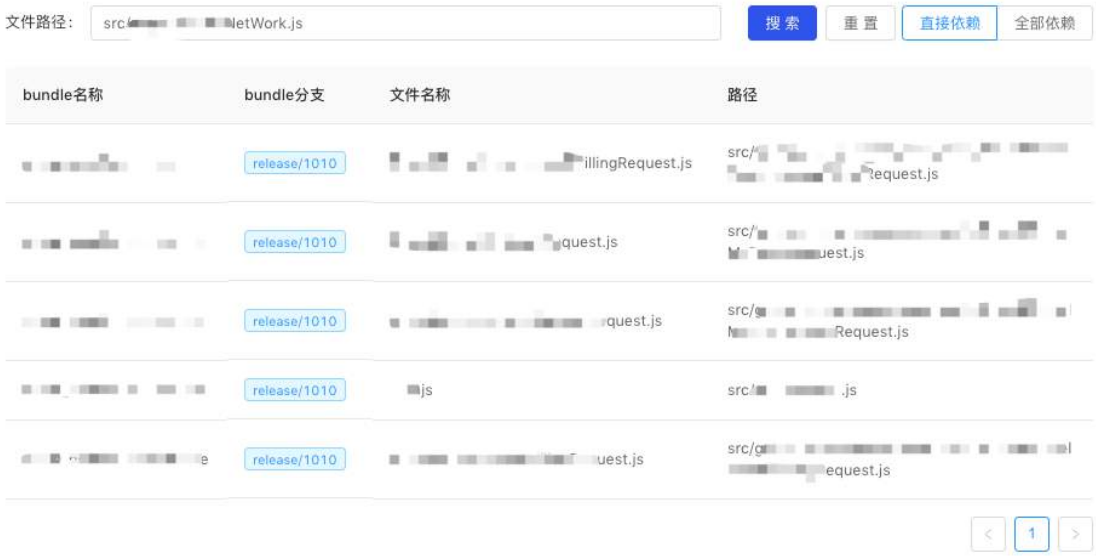
5.应用场景及实现原理

全链路依赖关系的使用场景有无穷的想象力，这里挑几个来举例。

影响范围判断（逆向依赖分析）

第一个我们能想到的应用场景就是影响范围判断，这也是我们这个项目的第一个抓手。大家都能想到，如果维护一个接口（或组件），我们会发现当越来越多地方用的时候，迭代它的风险会随之而越来越高，我们需要明确地知道到底有哪些地方调用了这个接口，以确定到底要回归测试多少功能、要怎么做发布、怎么做兼容等。而这就需要进行逆向依赖分析了。

逆向依赖是相对扫描器中分析出来的依赖关系的，扫描器分析出来的我们称之为正向依赖，它主要表示「此模块依赖了哪些别的模块」；而逆向依赖则指的是「此模块被哪些模块依赖了」。所以很自然地，我们的逆向依赖就是基于正向依赖关系做的数据加工。



（逆向依赖查询页面）

基于逆向依赖数据，结合多个版本的数据，我们还能算出「连续未被引用的版本数」，以衡量下线接口的安全性。

（一些切面 API 的连续未被使用的版本数）

组件库、框架和切面 API 的维护者是这个能力的重度用户，这个能力为他们带来了数据支

撑，明确了自己的修改将会影响多少的其它模块，从而进行变更、发布决策和回归测试。

版本间变动分析

版本提测时，我们可以对两个版本进行依赖链比对，分析出文件的变动及其整个影响链路，为 QA 提供一些数据支持，能更精确地知道有哪些功能要进行回归测试，有哪些不需要。

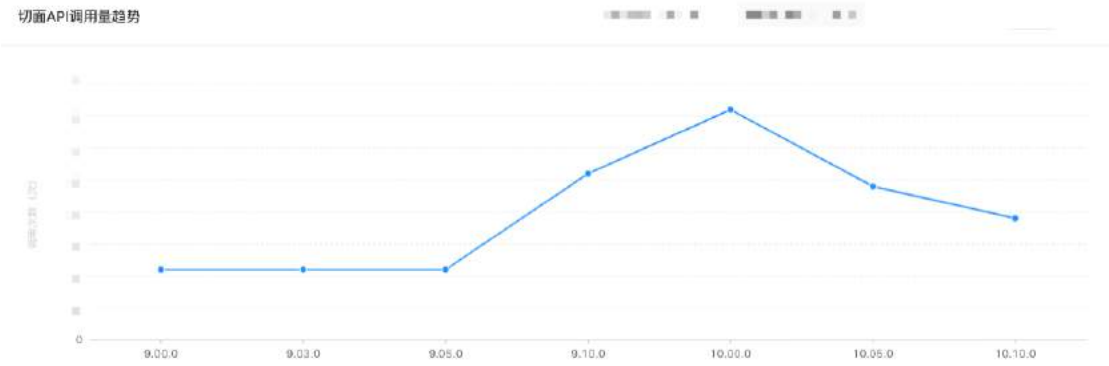
版本间变动分析有很多场景，除了正常的版本迭代的场景之外，还有一个常见的场景：模块在未变动的情况下被集成到新版本的高德 APP 中，那就会出现「发布代码不变，而所依赖的其它模块有变动」的情况，尤其有是 Native/C++ 和公用模块。测试环境需要知道的是，当前模块所依赖的其它模块到底有哪些变动、这些变动对此模块的影响是什么、需要回归测试哪些功能点等。

这个数据的主要消费方是 QA 同学，他们利用这个数据可以提高测试效率，也能发现漏考虑的回归点。

趋势变化判断

前面也提到过，由于高德 APP 时间跨度很大，以及之前未进行限制，所以我们有部分业务逻辑代码仍然是通过 Native 来实现的，我们希望逐渐迁移到 JS 或 C++ 层实现，Native 仅作适配。

而要判断这个治理的进度和效果，需要从两个方面的数据来支撑，一是各平台代码行数，这个我们另有专门的服务做，暂且不提；二是接口趋势。接口趋势也分为调用侧和声明侧两种，按照我们治理的方向，我们期望的效果应该是：一条 Native 业务切面 API 的调用量按版本/时间不断减少的曲线，当一些 API 的调用量为 0 后就可以把 API 下线掉，这样就会随之出现另一条曲线——Native 业务切面 API 的声明量也不断减少。



（从某版本开始就不断减少调用的切面 API）

（某版本未被使用的切面 API）

进行架构治理、切面 API 治理的同学是这些数据的主要消费方，有了这些数据他们就能确定架构治理的趋势是否合理、是否能下线某切面 API 等。

包大小优化——无用、重复文件查找

我们也为包大小优化作了贡献。根据依赖关系数据，我们可以找出一些没有被引用或者内容完全一样（md5 值相同）的文件，这些文件也占用了不少体积。

我们利用依赖分析工程找出了上千张这样的图片，@1x @2x @3x 文件是重灾区，有很多假装自己是另一个清晰度的图片被我们揪出来了（我们甚至因此推动了设计师出图标准化和增加了检验工具）。

6.写在最后

以上便是高德全链路依赖分析工程的基本概述，在具体的实现当中，会有无数的细节需要处理，如各种历史遗留问题、多级版本处理产生指数级的代码快照、变动分析产生指数级的分析结果等，其中也涉及到不少编译原理、数据结构与算法（尤其是图结构）等知识，非常考验编程能力和权衡能力，以及最重要的——韧性。欢迎大家一起讨论，一起迸发新的想法、新的场景！

关于作者

叶俊星（花名达诺），网名 Jasin Yip，高级前端工程师，长期从事大前端技术架构优化等工作。

前端内存优化的探索与实践

作者：见秋

1.导读

标注是地图最基本的元素之一，标明了地图每个位置或线路的名称。在地图 JSAPI 中，标注的展示效果及性能也是需要重点解决的问题。

新版地图标注的设计中，引入了 SDF （ signed distance field）重构了整个标注部分的代码。新的方式需要把标注的位置偏移，避让，三角拆分等全部由前端进行计算，不仅计算量激增，内存的消耗也成了重点关注的问题之一。

例如，3D 场景下需要构建大量的顶点坐标，一万左右的带文字的标注，数据量大约会达到 8 （attributes）* 5 (1 个图标 + 4 个字)* 6（个顶点） * 1E4 ，约为 250w 个顶点，使用 Float32Array 存储，需要的空间约为 2.5E6 * 4（byte）空间(海量地图标注 DEMO)。

前端这样大量的存储消耗，需要对内存的使用十分小心谨慎。于是借此机会研究了一下前端内存相关的问题，以便在开发过程中做出更优的选择，减少内存消耗，提高程序性能。

2.前端内存使用概述

首先我们来了解一下内存的结构。

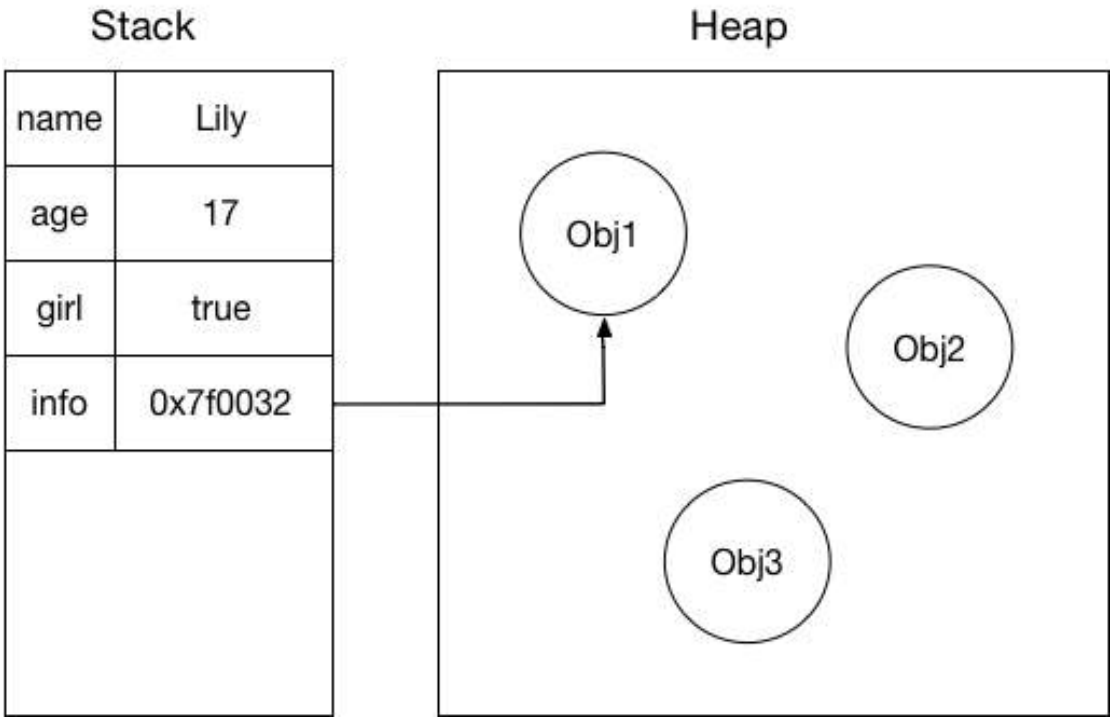
内存结构

内存分为堆（heap）和栈（stack），堆内存存储复杂的数据类型，栈内存则存储简单数据类型，方便快速写入和读取数据。在访问数据时，先从栈内寻找相应数据的存储地址，再根据获得的地址，找到堆内该变量真正存储的内容读取出来。

在前端中，被存储在栈内的数据包括小数值型，string ， boolean 和复杂类型的地址索引。

所谓小数值数据(small number)，即长度短于 32 位存储空间的 number 型数据。

一些复杂的数据类型，诸如 Array，Object 等，是被存在堆中的。如果我们要获取一个已存储的对象 A，会先从栈中找到这个变量存储的地址，再根据该地址找到堆中相应的数据。如图：



简单的数据类型由于存储在栈中，读取写入速度相对复杂类型（存在堆中）会更快些。下面的 Demo 对比了存在堆中和栈中的写入性能：

```
function inStack(){
    let number = 1E5;
    var a;

    while(number--){
        a = 1;
    }
}

var obj = {};
function inHeap(){
    let number = 1E5;

    while(number--){
        obj.key = 1;
    }
}
```

实验环境 1：mac OS/firefox v66.0.2。对比结果：

实验环境 2：mac OS/safari v11.1(13605.1.33.1.2)。对比结果：

在每个函数运行 10w 次的数据量下，可以看出在栈中的写入操作是快于堆的。

对象及数组的存储

在 JS 中，一个对象可以任意添加和移除属性，似乎没有限制（实际上需要不能大于 2^32 个属性）。而 JS 中的数组，不仅是变长的，可以随意添加删除数组元素，每个元素的数据类型也可以完全不一样，更不一般的是，这个数组还可以像普通的对象一样，在上面挂载任意属性，这都是为什么呢？

Object 存储

首先了解一下，JS 是如何存储一个对象的。

JS 在设计复杂类型存储的时候面临的最直观的问题就是，选择一种数据结构，需要在读取，插入和删除三个方面都有较高的性能。

数组形式的结构，读取和顺序写入的速度最快，但插入和删除的效率都非常低下；

链表结构，移除和插入的效率非常高，但是读取效率过低，也不可取；

复杂一些的树结构等等，虽然不同的树结构有不同的优点，但都绕不过建树时较复杂，导致初始化效率低下；

综上所属，JS 选择了一个**初始化，查询和插入删除都能有较好，但不是最好的性能的数据结构** -- 哈希表。

- **哈希表**

哈希表存储是一种常见的数据结构。所谓哈希映射，是把任意长度的输入通过散列算法变换成固定长度的输出。

对于一个 JS 对象，每一个属性，都按照一定的哈希映射规则，映射到不同的存储地址

上。在我们寻找该属性时，也是通过这个映射方式，找到存储位置。当然，这个映射算法一定不能过于复杂，这会使映射效率低下；但也不能太简单，过于简单的映射方式，会导致无法将变量均匀的映射到一片连续的存储空间内，而造成频繁的哈希碰撞。

关于哈希的映射算法有很多著名的解决方案，此处不再展开。

- **哈希碰撞**

所谓哈希碰撞，指的是在经过哈希映射计算后，被映射到了相同的地址，这样就形成了哈希碰撞。想要解决哈希碰撞，则需要对同样被映射过来的新变量进行处理。

众所周知，JS 的对象是可变的，属性可在任意时候（大部分情况下）添加和删除。在最开始给一个对象分配内存时，如果不想出现哈希碰撞问题，则需要分配巨大的连续存储空间。但大部分的对象所包含的属性一般都不会很长，这就导致了极大的空间浪费。

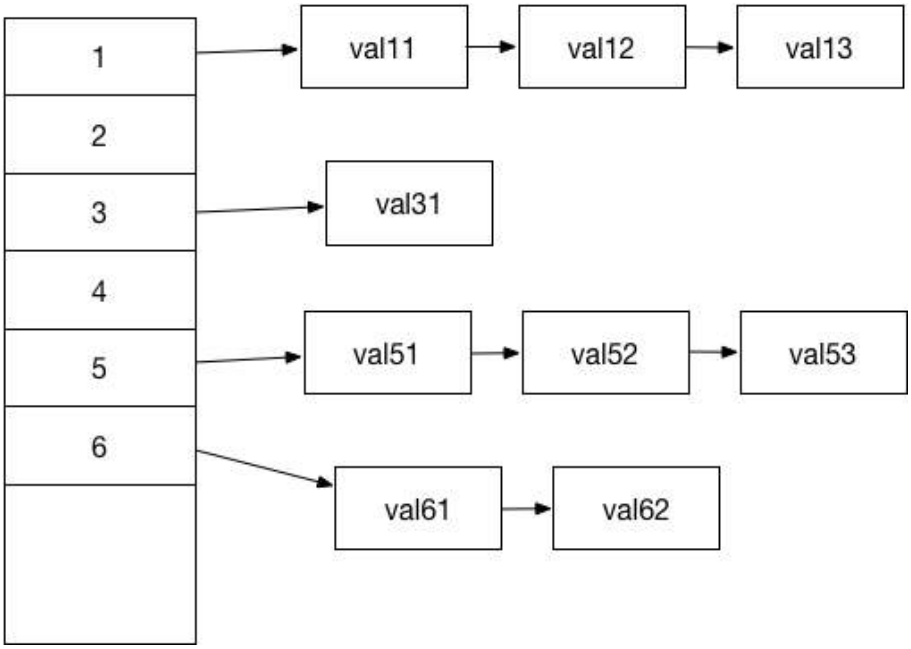
但是如果一开始分配的内存较少，随着属性数量的增加，必定会出现哈希碰撞，那如何解决哈希碰撞问题呢？

对于哈希碰撞问题，比较经典的解决方法有如下几种：

- 开放寻址法
- 再哈希法
- 拉链法

这几种方式均各有优略，由于本文不是重点讲述哈希碰撞便不再缀余。

在 JS 中，选择的是拉链法解决哈希碰撞。所谓**拉链法，是将通过一定算法得到的相同映射地址的值，用链表的形式存储起来。**如图所示（以倾斜的箭头表明链表动态分配，并非连续的内存空间）：



映射后的地址空间存储的是一个链表的指针，一个链表的每个单元，存储着该属性的 key, value 和下一个元素的指针；

这种存储的方式的好处是，最开始不需要分配较大的存储空间，**新添加的属性只要动态分配内存即可**；

对于索引，添加和移除都有相对较好的性能；

通过上述介绍，也就解释了这个小节最开始提出的为何 JS 的对象如此灵活的疑问。

Array 存储

JS 的数组为何也比其他语言的数组更加灵活呢？**因为 JS 的 Array 的对象，就是一种特殊类型的数组！**

所谓特殊类型，就是指在 Array 中，每一个属性的 key 就是这个属性的 index；而这个对象还有 .length 属性；还有 concat, slice, push, pop 等方法；

于是这就解释了：

为何 JS 的数组每个数据类型都可以不一样？
因为他就是个对象，每条数据都是一个新分配的类型连入链表中；

为何 JS 的数组无需提前设置长度，是可变数组？
答案同上；

为何数组可以像 Object 一样挂载任意属性？
因为他就是个对象；

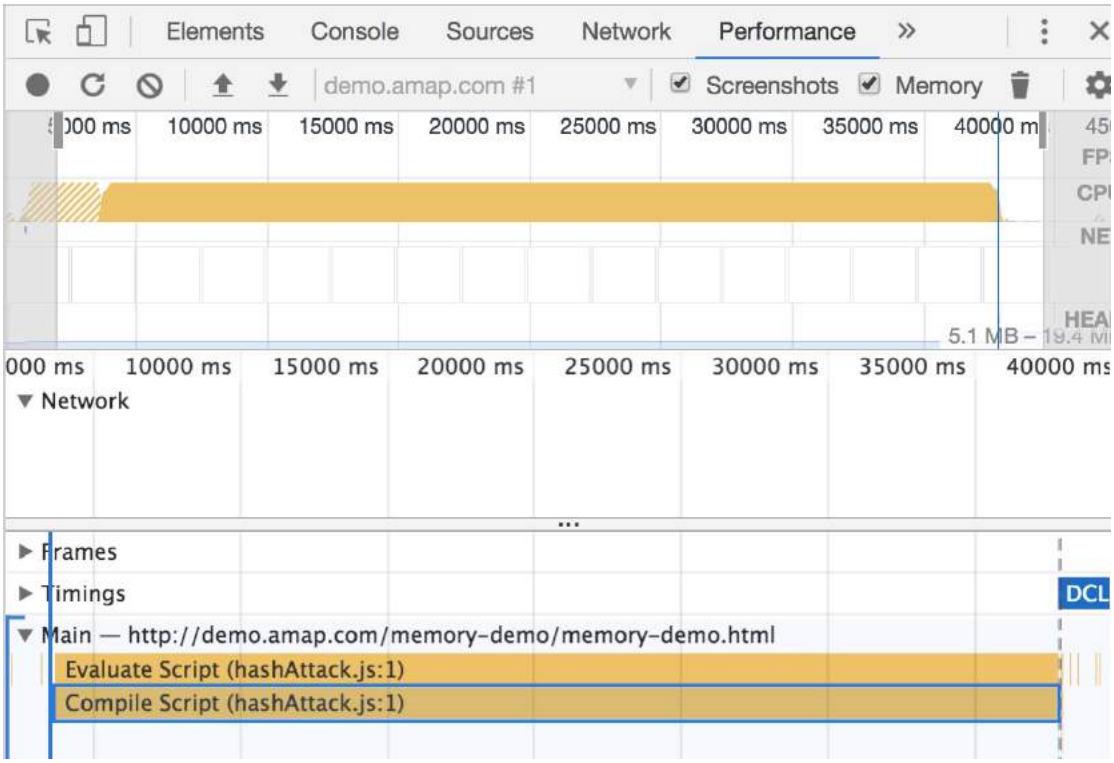
等等一系列的问题。

- **内存攻击**

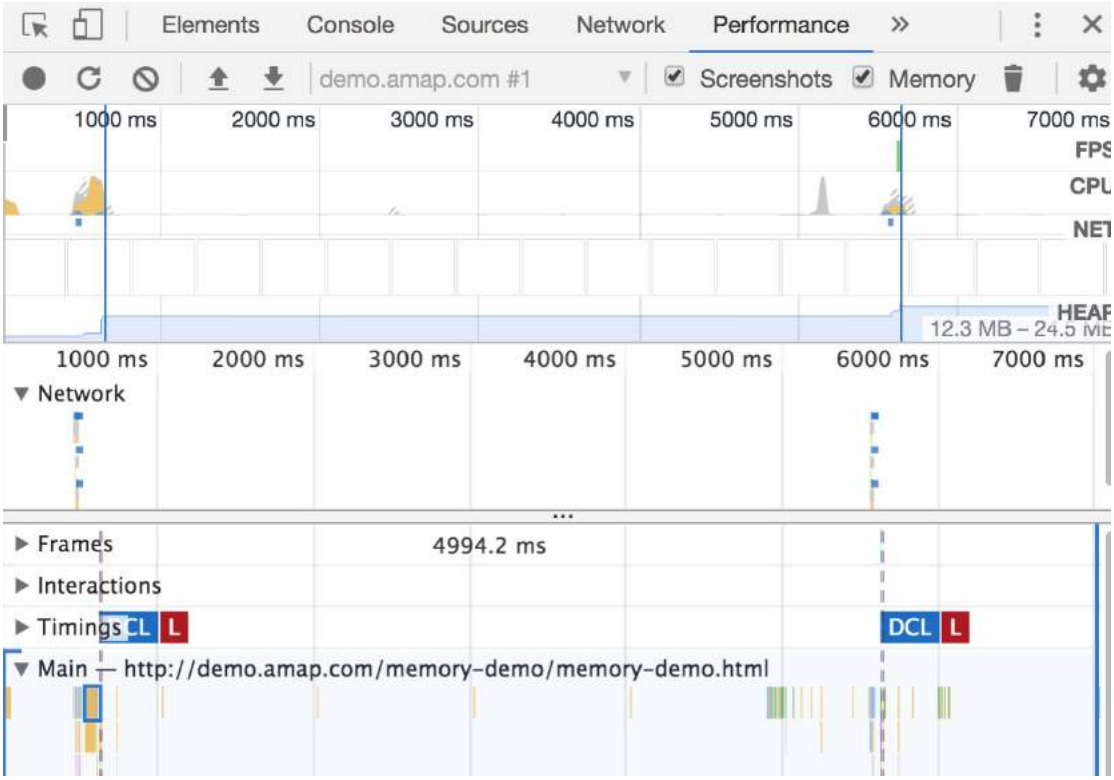
当然，选择任何一种数据存储方式，都会有其不利的一面。这种哈希的拉链算法在极端情况下也会造成严重的内存消耗。

我们知道，良好的散列映射算法，可以讲数据均匀的映射到不同的地址。但如果我们掌握了这种映射规律而将不同的数据都映射到相同的地址所对应的链表中去，并且数据量足够大，将造成内存的严重损耗。读取和插入一条数据会中了链表的缺陷，从而变得异常的慢，最终拖垮内存。这就是我们所说的内存攻击。

构造一个 JSON 对象，使该对象的 key 大量命中同一个地址指向的列表，附件为 JS 代码，只包含了一个特意构造的对象，图二为利用 Performance 查看的性能截图：



相同 size 对象的 Performance 对比图：



根据 Performance 的截图来看，仅仅是 load 一个 size 为 65535 的对象，竟然足足花费了 40 s！而相同大小的非共计数据的运行时间可忽略不计。

如果被用户利用了这个漏洞，构建更长的 JSON 数据，可以直接把服务端的内存打满，导致服务不可用。这些地方都需要开发者有意识的避免。

但从本文的来看，这个示例也很好的验证了我们上面所说的对象的存储形式。

3.视图类型（连续内存）

通过上面的介绍与实验可以知道，我们使用的数组实际上是伪数组。这种伪数组给我们的操作带来了极大的方便性，但这种实现方式也带来了另一个问题，及无法达到数组快速索引的极致，像文章开头时所说的上百万的数据量的情况下，每次新添加一条数据都需要动态分配内存空间，数据索引时都要遍历链表索引造成的性能浪费会变得异常的明显。

好在 ES6 中，JS 新提供了一种获得真正数组的方式：ArrayBuffer，TypedArray 和 DataView

ArrayBuffer

ArrayBuffer 代表分配的一段定长的连续内存块。但是我们无法直接对该内存块进行操作，只能通过 TypedArray 和 DataView 来对其操作。

TypedArray

TypeArray 是一个统称，他包含 Int8Array / Int16Array / Int32Array / Float32Array 等等。详细请见：

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray

拿 Int8Array 来举例，这个对象可拆分为三个部分：Int、8、Array

首先这是一个数组，这个数据里存储的是有符号的整形数据，每条数据占 8 个比特位，及该数据里的每个元素可表示的最大数值是 2^7 = 128，最高位为符号位。

```
// TypedArray
var typedArray = new Int8Array(10);

typedArray[0] = 8;
typedArray[1] = 127;
typedArray[2] = 128;
typedArray[3] = 256;

console.log("typedArray","   -- ", typedArray );
//Int8Array(10) [8, 127, -128, 0, 0, 0, 0, 0, 0, 0]
```

其他类型也都以此类推，可以存储的数据越长，所占的内存空间也就越大。这也要求在使用 TypedArray 时，对你的数据非常了解，在满足条件的情况下尽量使用占较少内存的类型。

DataView

DataView 相对 TypedArray 来说更加的灵活。每一个 TypedArray 数组的元素都是定长的数据类型，如 Int8Array 只能存储 Int8 类型；但是 DataView 却可以在传递一个 ArrayBuffer 后，动态分配每一个元素的长度，即存不同长度及类型的数据。

```
// DataView
var arrayBuffer = new ArrayBuffer(8 * 10);

var dataView = new DataView(arrayBuffer);

dataView.setInt8(0, 2);
dataView.setFloat32(8, 65535);

// 从偏移位置开始获取不同数据
dataView.getInt8(0);
// 2
dataView.getFloat32(8);
// 65535
```

TypedArray 与 DataView 性能对比

DataView 在提供了更加灵活的数据存储的同时，最大限度的节省了内存，但也牺牲了一部分性能，同样的 DataView 和 TypedArray 性能对比如下：

```
// 普通数组
function arrayFunc(){
    var length = 2E6;
    var array = [];
    var index = 0;

    while(length--){
        array[index] = 10;
        index ++;
    }
}

// dataView
function dataViewFunc(){
    var length = 2E6;
```

```
    var arrayBuffer = new ArrayBuffer(length);
    var dataView = new DataView(arrayBuffer);
    var index = 0;

    while(length--){
        dataView.setInt8(index, 10);
        index ++;
    }
}

// typedArray
function typedArrayFunc(){
    var length = 2E6;
    var typedArray = new Int8Array(length);
    var index = 0;

    while(length--){
        typedArray[index++] = 10;
    }
}
```

实验环境 1：mac OS/safari v11.1(13605.1.33.1.2)。对比结果：

实验环境 2：mac OS/firefox v66.0.2。对比结果：

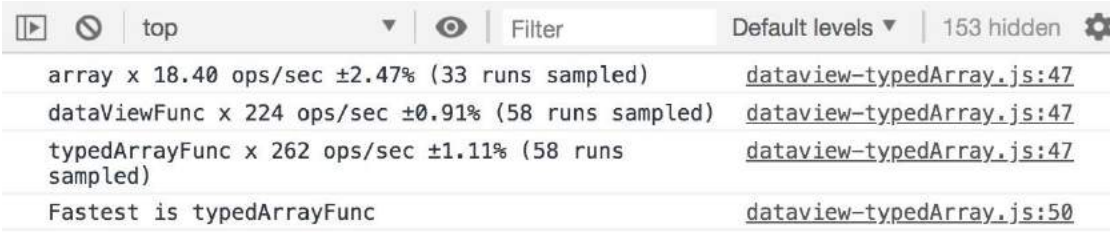
在 Safari 和 firefox 下，DataView 的性能还不如普通数组快。所以在条件允许的情况下，开发者还是尽量使用 TypedArray 来达到更好的性能效果。

当然，这种对比并不是一成不变的。比如，谷歌的 V8 引擎已经在最近的升级版本中，解决了 DataView 在操作时的性能问题。

DataView 最大的性能问题在于将 JS 转成 C++ 过程的性能浪费。而谷歌将该部分使用 CSA（ CodeStubAssembler）语言重写后，可以直接操作 TurboFan（V8 引擎）来避免转

换时带来的性能损耗。

实验环境 3:mac OS / chrome v73.0.3683.86。对比结果：



可见在 chrome 的优化下，DataView 与 TypedArray 性能差距已经不大了，在需求需要变长数据保存的情况下，DataView 会比 TypedArray 节省更多内存。

具体性能对比：
https://v8.dev/blog/dataview

4.共享内存（多线程通讯）

共享内存介绍

说到内存还不得不提的一部分内容则是共享内存机制。

JS 的所有任务都是运行在主线程内的，通过上面的视图，我们可以获得一定性能上的提升。但是当程序变得过于复杂时，我们希望通过 webworker 来开启新的独立线程，完成独立计算。

开启新的线程伴随而来的问题就是通讯问题。webworker 的 postMessage 可以帮助我们完成通信，但是这种通信机制是将数据从一部分内存空间复制到主线程的内存下。这个赋值过程就会造成性能的消耗。

而共享内存，顾名思义，可以让我们在不同的线程间，共享一块内存，这些现成都可以对内存进行操作，也可以读取这块内存。省去了赋值数据的过程，不言而喻，整个性能会有较大幅度的提升。

使用原始的 postMessage 方法进行数据传输

- main.js

```
1.  // main
2.  var worker = new Worker('./worker.js');
3.
4.  worker.onmessage = function getMessageFromWorker(e){
5.      // 被改造后的数据，与原数据对比，表明数据是被克隆了一份
6.      console.log("e.data","   -- ", e.data );
7.      // [2, 3, 4]
```

```
8.
9.      // msg 依旧是原本的 msg，没有任何改变
10.     console.log("msg","   -- ", msg );
11.     // [1, 2, 3]
12. };
13.
14. var msg = [1, 2, 3];
15.
16.  worker.postMessage(msg);
```

- worker.js

```
1.  // worker
2.  onmessage = function(e){
3.      var newData = increaseData(e.data);
4.      postMessage(newData);
5.  };
6.
7.  function increaseData(data){
8.
9.      for(let i = 0; i < data.length; i++){
10.         data[i] += 1;
11.     }
12.
13.     return data;
14. }
```

由上述代码可知，每一个消息内的数据在不同的线程中，都是被克隆一份以后再传输的。数据量越大，数据传输速度越慢。

使用 sharedBufferArray 的消息传递

- main.js

```
1.  var worker = new Worker('./sharedArrayBufferWorker.js');
2.
3.  worker.onmessage = function(e){
4.    // 传回到主线程已经被计算过的数据
5.      console.log("e.data","   -- ", e.data );
6.      // SharedArrayBuffer(3) {}
7.
8.    // 和传统的 postMessage 方式对比，发现主线程的原始数据发生了改变
9.      console.log("int8Array-outer","   -- ", int8Array );
10.     // Int8Array(3) [2, 3, 4]
```

```
};

var sharedArrayBuffer = new SharedArrayBuffer(3);
var int8Array = new Int8Array(sharedArrayBuffer);

int8Array[0] = 1;
int8Array[1] = 2;
int8Array[2] = 3;

worker.postMessage(sharedArrayBuffer);
```

- **worker.js**

```
onmessage = function(e){
    var arrayData = increaseData(e.data);
    postMessage(arrayData);
};

function increaseData(arrayData){
    var int8Array = new Int8Array(arrayData);
    for(let i = 0; i < int8Array.length; i++){
        int8Array[i] += 1;
    }

    return arrayData;
}
```

通过共享内存传递的数据，在 worker 中改变了数据以后，主线程的原始数据也被改变了。

性能对比

实验环境 1：mac OS/chrome v73.0.3683.86, 10w 条数据。对比结果：

实验环境 2：mac OS/chrome v73.0.3683.86,100w 条数据。对比结果：

从对比图中来看，10w 数量级的数据量，sharedArrayBuffer 并没有太明显的优势，但在百万数据量时，差异变得异常的明显了。

SharedArrayBuffer 不仅可以在 webworker 中使用，在 wasm 中，也能使用共享内存进行通信。在这项技术使我们的性能得到大幅度的提升时，也没有让数据传输成为性能瓶颈。

但比较可惜的一点是，SharedArrayBuffer 的兼容性比较差，只有 chrome 68 以上支持，firefox 在最新版本中虽然支持，但需要用户主动开启；在 safari 中甚至还不支持该对象。

5.内存检测及垃圾回收机制

为了保证内存相关问题的完整性，不能拉下内存检测及垃圾回收机制。不过这两个内容都有非常多介绍的文章，这里不再详细介绍。

内存检测

介绍了前端内存及相关性能及使用优化后。最重要的一个环节就是如何检测我们的内存占用了。chrome 中通常都是使用控制台的 Memory 来进行内存检测及分析。

使用内存检测的方式参见：

https://developers.google.com/web/tools/chrome-devtools/memory-problems/heap-snapshots?hl=zh-cn

垃圾回收机制

JS 语言并不像诸如 C++ 一样需要手动分配内存和释放内存，而是有自己一套动态 GC 策略的。通常的垃圾回收机制有很多种。

前端用到的方式为标记清除法，可以解决循环引用的问题：
https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Memory_Management#垃圾回收

6.结束语

在了解了前端内存相关机制后，创建任意数据类型时，我们可以在贴近场景的情况下去选

择更合适的方式保有数据。例如：

- 在数据量不是很大的情况下，选择操作更加灵活的普通数组。
- 在大数据量下，选择一次性分配连续内存块的类型数组或者 DataView。
- 不同线程间通讯，数据量较大时采用 sharedBufferArray 共享数组。
- 使用 Memory 来检测是否存在内存问题，了解了垃圾回收机制，减少不必要的 GC 触发的 CPU 消耗。

再结合我们的地图标注改版来说，为了节省内存动态分配造成的消耗，量级巨大的数据均采用的 TypedArray 来存储。另外，大部分的数据处理，也都在 worker 内进行。为了减少 GC，将大量的循环内变量声明全部改成外部一次性的声明等等，这些都对我们的性能提升有了很大的帮助。

最后，这些性能测试的最终结果并非一成不变（如上面 chrome 做的优化），但原理基本相同。所以，如果在不同的时期和不同的平台上想要得到相对准确的性能分析，还是自己手动写测试用例来得靠谱。

汽车工程篇

NDS 中车道连接关系的制作方法

作者：闻竹

1.导读

在高精度导航数据中，车道级数据，取代了原来标精数据中的道路级数据。从原来的道路 A 连接道路 B，变为车道 A 连接车道 B。高精数据，极大的提高了导航的精确度，是自动驾驶不可缺少的一个模块。

本文着重介绍基于 NDS 的高精导航数据中，车道连接关系是如何制作的。

2.如何定义“车道连接关系”

2.1 定义

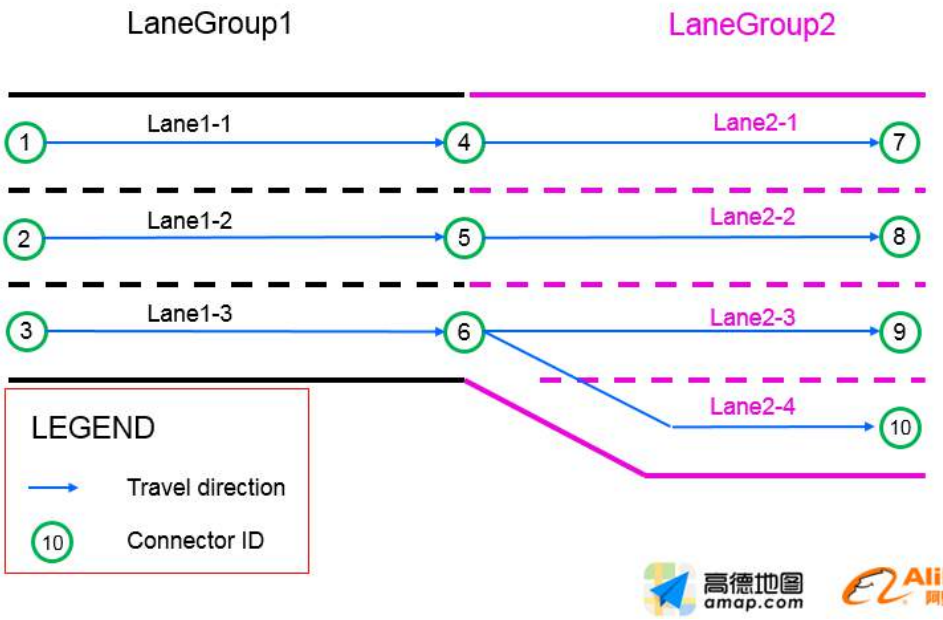
个人认为，在高精数据中，车道是导航数据中的最小单元。每一个车道等同于标精数据中的每一根道路。而车道连接关系表达了车道之间的连通性。导航应用通过数据中存储的车道连接关系，可以知道当前车道的前序和后继车道有哪些，从而利用这些信息完成车道引导、车道计算等功能。

2.2 NDS 中表达方法

NDS 规格中定义了 LANE BuildingBlock（以下简称为 Lane BB），是用于表达高精的车道级数据。Lane BB 中 Lane Group 的定义是：一组位置平行或属性相同的车道组合。在 Lane Group 中，定义了很多的属性，比如：车道数、车道类型、车道边线类型、车道中心线形点、车道连接关系等等。

而 NDS 中是使用 connector ID 来表达车道连接关系的。我们可以这样理解 connector ID：一个车道在入口处和出口处各有一个 ID，如果两个车道是相互连接的，那么它们连接处的 connector ID 必须要相同。

如下图 LaneGroup 1 的 Lane1-1 和 LaneGroup 2 的 Lane2-1 是连通的，所以 Lane1-1 出口的 connector ID 和 Lane2-1 的入口的 connector ID 都等于 4。这就表达了两个车道的相互连接关系。



connector ID 的取值分为 NDS 2.5.2 和 NDS2.5.4：

- NDS 2.5.2：

定义：connector ID 是一个 varuint16 类型的变量，范围是 0~32639。Tile 内唯一，但是 Tile 边界的 connector ID 必须在周围 9 tile 内唯一。如果不唯一会引发车道连接错误的 bug。

缺点：可用范围小。

- NDS 2.5.4：

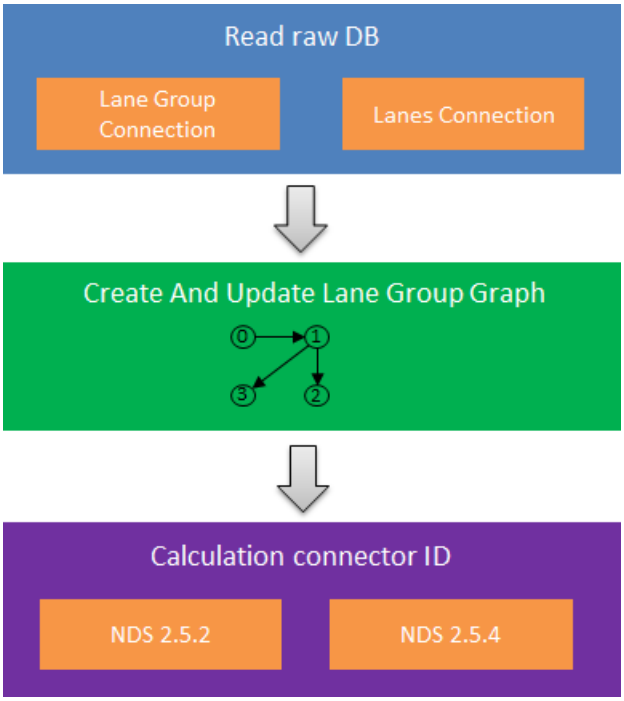
定义：connector ID 是一个 varuint32 类型的变量，范围是 0~536870911。周围 9 Tile 内唯一。

3.基于 NDS 的制作方案

3.1 基本思路

基于原有的代码逻辑，connector ID 的制作思路分为三步：

- 读取原始数据的车道连接关系
- 构建 Lane group graph， 根据 Tiling 的结果，更新 Lane group graph
- 遍历 Lane group graph 中每个节点，计算 connector ID



3.2 两个难点及解决方案

3.2.1 如何快速找到 Lane Group 和 Lane 的连接关系？

- 采用图（十字链表）的方式解决
- 采用图的优点：

i.图可以快速的找出当前节点的前序和后继节点，便于后续的 connector ID 的计算。

ii.相比其他数据结构，图可以快速的更新图中的拓扑，适用于车道数据更新后连接关系的维护。

3.2.2 如何计算可用的 connector ID？

在不同的 NDS 版本中，使用了不同的解决方案：

- NDS 2.5.2：

问题：由于可用的 ID 区间（0~32639）有限，如何既保证 Tile 内 ID 唯一，又保证 Tile 边界处 9 Tile 唯一？

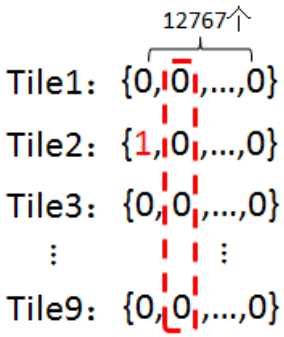
思路：最简单的保证唯一值的方案就是获取最大值，但是可用的 ID 区间很小，如果单纯的获取最大值，很容易出现越界的情况，所以此方案不可行。对于 NDS252 来说，我认为规格设计是存在缺陷的。connector id 的可用范围设计的太小了。日后该版本的 NDS 数据肯定会被淘汰，所以此处的解法仅供参考。

解法：将 varuint16 划分为 2 个区间

i.Tile 内的 ID 区间（目前分配 0~19999）

ii.Tile 边界处的 ID 区间（20000~32639）

Tile 内 ID 采用自增的逻辑分配，Tile 边界处采用 9 Tile 内垂直比较方法获取可分配的 ID（如下图）。



- NDS 2.5.4：

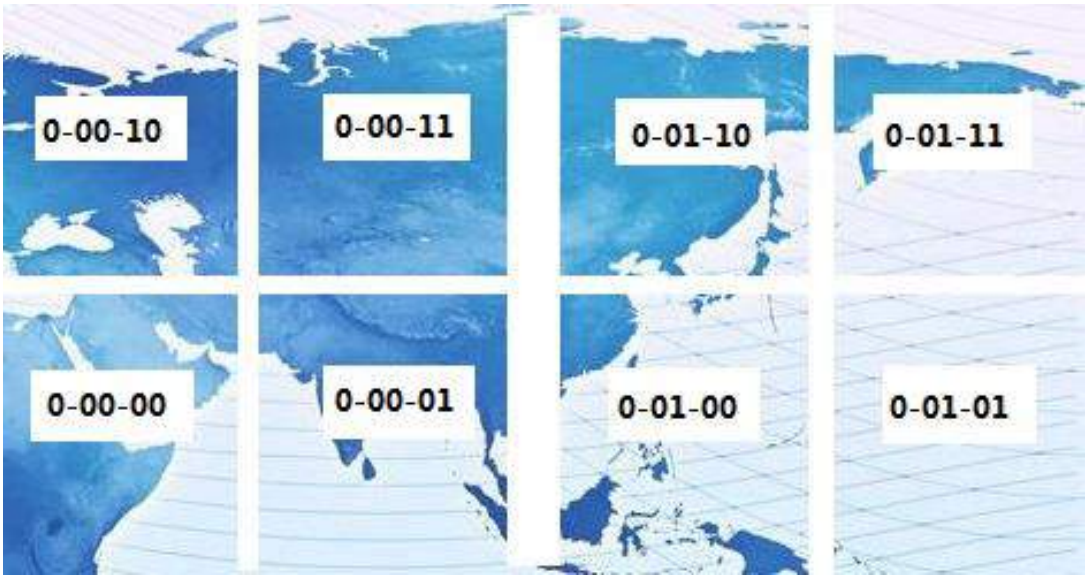
问题：ID 可用空间扩充，如果还采用 2.5.2 的垂直比较方法，内存撑不住。

思路：到了 NDS254，从规格上补足了 252 设计的缺陷，扩充了 connector ID 的可用范围。所以我们要寻求一种新的方案来分为 9 Tile 内唯一 ID。如果我们能保证在任意位置，可以给以当前位置为中心的 9 宫格，分配 1~9 不同的索引号，这样就能保证任意位置的 9 宫格内都是唯一 ID 了（有点类似数独）。顺着这样的思路便有了下面的解法。

解法：根据 Tile ID 划分不同的 connector ID 可用区间。

i.NDS 中 Tile ID 是一个 int 变量。虽然它只是一个数，但是是根据一定的规则计算出来的。

如下图，Tile 实际就是把地球按照 2 的 n 次方分割下去。如果把 Tile ID 转换为 2 进制，我们可以发现，其偶数位是所在的行号，奇数位是所在的列号。我们有了行列号，就可以把其转为 9 宫格内的唯一编号了（行列号余 3）。



ii.将 connector ID 划分为 9 个区间，目前一个区间分配了 100000 个 connector ID，如果后续不够可以修改区间范围。根据上面计算出的 Tile 的唯一编号，乘以对应的区间 ID，即是当前 Tile 可用的 connector ID 区间。

iii.Connector ID 在区间中采用自增的逻辑即可。

4.小结

以上就是基于 NDS 的车道连接关系的制作方案。NDS 外其他导航数据规格中表达车道连接关系的形式各不相同。但是本文中的思路同样适用，只需要根据不同规格要求进行适当调整即可。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

云控平台的双向音频解决方案

作者：东滔

1.导读

随着移动互联网的发展，行业内衍生了基于移动平台的各类解决方案。其中，设备规模化管理的云控能力是各互联网公司在设备集群控制背景下的诉求。因此涌现了大批提供类似解决方案的平台。如：阿里系的阿里云 MQC、阿里无线和菜鸟 Nimitz 等，阿里之外的有 Testin、百度 MTC、腾讯 WeTest、华为、三星等等。

目前以上平台在云真机的使用上，都存在一个已知的短板 —— 声音。用户看的到画面，能够响应操作，但是涉及到声音播报、语音交互的场景时则无能为力。尤其对于音乐、视听、短视频、直播客户端等这类多媒体属性强的 App，在云真机的使用场景上是受限最大的。

现在回到我们自己的产品。高德地图车机/镜版（后面统称 Auto）。其中最常见的导航播报、与系统的多媒体混音交互、以及语音助手多轮对话的交互场景中，这些与声音相关的场景占比高达 25%以上。所以解决远程场景下的声音双向交互问题，是云真机要成为一个日常化的生产工具之前必须迈过的坎。

2.挑战

在远程音频的双向通讯解决方案的背景下，满足基本用户体验的方面也存在以下挑战：

- 能力：满足所有车载设备的声音场景的双向交互能力（因为车载设备在声音部分比手

机具有更高的定制性，在覆盖车载场景后，手机基本可以无缝适配）。

- 延迟：传输延迟低于 500ms（基于一定的网络条件）。
- 体验：无明显卡顿、杂音问题。

3.思考

首先通过下面的一张图来了解一下我们的需求是什么：

- 将声音通过电脑传输到远端的车机设备（车机系统能正常解析处理）。
- 将车机通过喇叭播报出的声音传输到用户端。

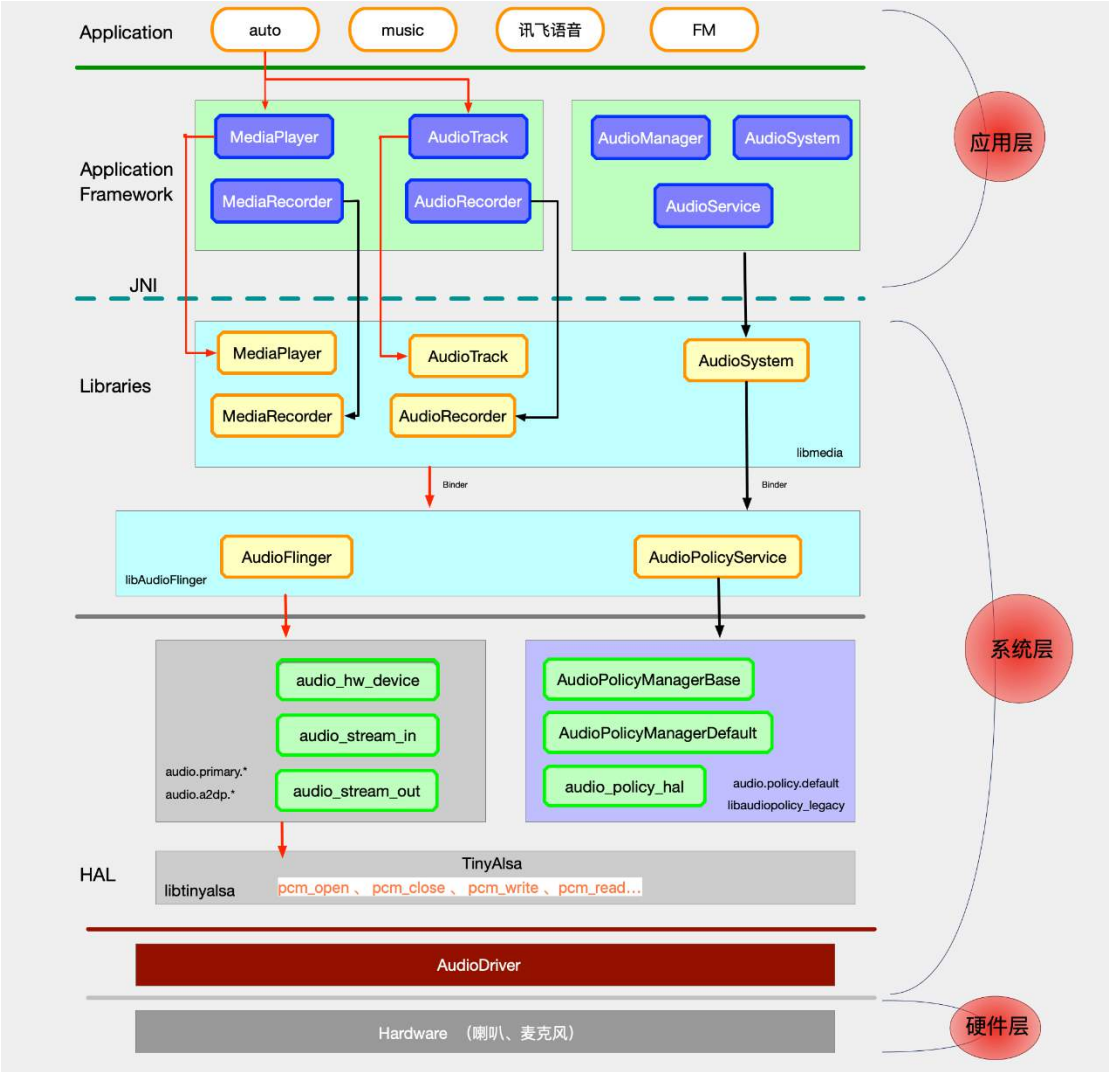
而实现这两条链路，关键核心的两个因素是：

- 如何获取和写入音频数据。
- 如何实现实时的音频数据在车机和用户设备间的传输链路。

4.音频获取和写入

4.1 Android 系统音频概要

在思考如何进行设备的音频获取前，我们先来了解下 Android 的音频系统架构：



上图描述了音频通信从应用层、Libraries、HAL、到 Driver，最后到硬件模块各层主要实现。而我们也需要从这条链路中去挖掘获取和写入音频数据的思路。

首先，我们考虑的是 Android 对应的音频链路中是否有成熟的支持双向音频的能力。即音频数据在 OS 内部获取到对外传输。

4.2 REMOTE_SUBMIX

API 19 新加的 MediaRecorder. AudioSource. REMOTE_ SUBMIX，用于传输系统混音的音频流到远端（在 API 18 也存在，只是属于隐藏属性）。

由于要生效 REMOTE_SUBMIX，需要 Manifest. permission. CAPTURE_ AUDIO_ OUTPUT 权限，而该权限只有系统组件才具备。也就是如果第三方 App 需要的话，需要进行系统签名或者在烧写 OS 版本时就修改对应的权限。作为系统方是可以这么操作的，但显然对于要适配所有系统方的我们来说不适用。

4.3 软件 hook

考虑到我们拿到的车载设备中，root 比例高达 80%以上。因此我们想从在音频数据传输到底层硬件驱动前进行“截胡”。也就是 hook HAL 定义的往驱动写入和读取对应音频数据的方法，来达到音频数据的双向获取。

4.3.1 hook HAL hardware

hw hook 的是 struct audio_hw_device 的
音频输出：open_output_stream、close_output_stream
音频输入：open_input_stream、close_input_stream

system/lib/hw/audio.primary.*.so (不同的设备有后缀部分差异)

4.3.2 hook tinyalsa

在实际的调研测试中，我们发现并不是每台设备都能通过 hook hw 来获取到对应的声音数据，尤其是车载设备。于是我们又调研了 ALSA（Advanced Linux Sound Architecture）高级 Linux 声音架构。根据官方的推荐，我们选择了具备 GPL-licensed 的 external/tinyalsa

hook tinyalsa.so
音频输出：pcm_open、pcm_close、pcm_write、pcm_mmap_write
音频输入：pcm_open、pcm_close、pcm_read、pcm_mmap_read

system/lib/libtinyalsa.so

4.3.3 问题

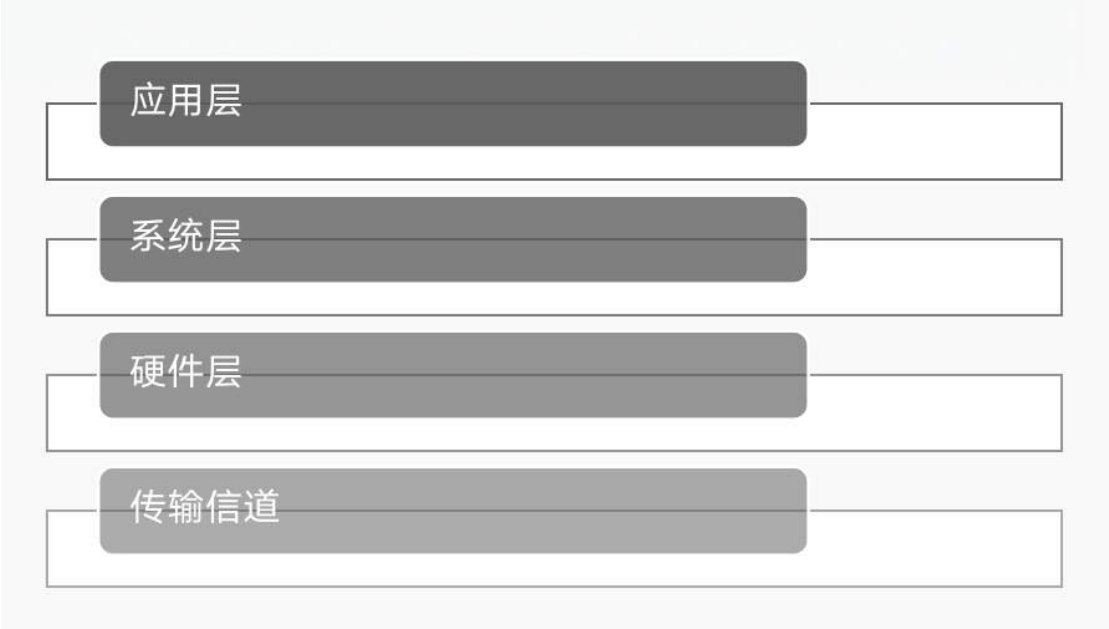
在实际摸底验证中，我们发现车机比手机还复杂的原因在于多了功放的概念，而部分车厂选择在设备的 DSP 模块去处理混音。带来的问题就是部分设备如果单纯的通过 hook 播报，对应听到的声音与设备真实通过喇叭播报的效果不同，这也导致我们对于该场景的还原并不真实。

因此，在于 root 设备覆盖不完全且部分设备存在硬件功放处理混音问题的情况下，软件 hook 的方案只能适用于部分设备。

4.3.4 成本

hook 自身也会带来一个问题，即针对不同的车机需要每台都进行 hook 处理，使得 hook 带来的成本过高。需要批量一键 hook 来解决这个问题。

分析到这里，我们回顾下音频传输的链路：



基于以上我们对音频获取的这条传输链路上的分析，现在理论上可行的获取途径，就只有硬件的对接或者具体的接收端（喇叭、蓝牙）。

4.4 USB 音频

硬件对接部分，在云控场景下，我们的设备通常是通过 USB 线束与我们的节点 PC 连接的。因此音频通过 USB 进行传输的链路，也是一个值得探索的方向。

我们知道，Android 设备在连接 USB 时有三种模式：Host、Development、Accessory Mode：

- 主机模式：可以传输音频，但是 Android 设备作为主机，无法使用 adb 的能力。
- 开发者模式：具备 adb 的能力，但是没有现成的 USB 音频能力。
- 配件模式：既保留了 adb 的能力，在 Android4.1 后的配件模式下，Android 也能自动将其音频输出导向到 USB。

思路：通过实现 AOA 协议，作为主机角色的设备，必须具有能够将 Android 设备从开发模式切换到配件模式的主机控制能力，然后主机从适当的端点传输音频数据

该方案的局限性在于：1、单向传输；2、配件模式取决于设备硬件，但并非所有设备都支持。实测过程中，车机支持配件模式的比例很低，绝大多数都被“阉割”了。

综上，我们无法靠单纯的某种 USB 模式来实现音频的双向交互。但如果是手机集群的场景中，这个方案倒是可以作为单向音频传输的一个优选方案。

4.5 蓝牙接收

声音除了可以通过 usb 传输以外，常见的方式还有蓝牙耳机、有线耳机。（这里由于车载设备不存在 3.5mm 孔，所以我们先不讨论有线方式，具体可参考后面「硬件转发」的方案）。

关于蓝牙接收的基本思路就是 PC 端通过安装蓝牙接收器与车机通信。其中蓝牙接收器起到类似于蓝牙耳机的作用。然后对蓝牙接收器的收发数据在 PC 端进行编码处理。

蓝牙耳机：具备了可以听说的能力，也就是双向的音频通信。

摸底验证：部分车机对蓝牙驱动进行了定制，使得蓝牙设备只能作为从设备，无法接入蓝牙耳机功能。我们测试了 35 台，其中 5 台可以用，成功率 14%，收益太低，成本过高。这个方案如果是面向手机集群，倒是一个不错的选择，理论上成功率应该会大大高于车载设备。

4.6 硬件转发

上面提到的有线耳机的思路。在车载设备上，不存在 3.5mm 孔或者 type C 口，而是通过主机与功放、音箱外放装置进行连接。在车辆量产上市前的研发阶段，只是一个主机通过线束连接着喇叭的一个过渡状态。所以我们实际是通过将原本接到喇叭上的音频数据通过一种转换装置转接到 PC 上，在 PC 端进行音频编码处理。

大致的参考示意效果如下：

上述方案的优势在于：

- 跨平台，不管是 Android、Linux、QNX 或者 iOS 的设备都适用。
- 解决了混音问题，由于对接的是最终播报出的声音效果，就不存在软件 hook 可能还原不真实的问题。
- 支持双向音频通信。

缺点：

- 部分智能车镜设备，由于集成封装性太强，没有暴露出可对接的线束。这类设备不容

易通过该方案覆盖。

- 需要针对车机进行线束定制来实现整体的动态封装性。
- 需要配套的硬件成本。

硬件转发方案中存在几个方面的问题需要注意，比如失真问题、声卡识别问题、USB 兼容上限、声卡与 ehci、xhci 的兼容性问题以及整体封装设计等等。

4.7 小结

综合以上音频传输的整条链路的所有方案，我们列举对比下这些方案的优劣（特指在车载场景下）：

方案	优势	劣势	覆盖率
REMOTE_SUBMIX	1、现有 API 2、同时利于音视频同步编码 3、无硬件成本	1、需要系统签名或 Root 权限 2、只有音频获取，无音频输入接口 3、需要 Android API 18 以上 4、录音时本地无法播放声音，也就是不能同时本地和远程都有声音	低
软件 hook	1、无硬件成本 2、同时利于音视频同步编码 3、满足目前所有车载的系统版本	1、需要 Root 权限 2、需要适配不同版本的 Android 系统，成本较高 3、开发难度较大 4、存在混音	理论 100% 实际摸底：真实还原的混音效果的占比 52%
USB 音频	1、现有的技术方案 2、无硬件成本	1、只有音频输出，没有音频输入 2、只有在配件模式下支持 adb 和音频通信 3、车机大部分不支持配件模式	低
蓝牙接收	1、同时支持输入音频输入和输出	1、PC 端需要加个 USB hub 接入多个蓝牙接收器，PC 端需要开发蓝牙接收转发程序 2、部分车机对蓝牙驱动进行了定制，使蓝牙设备只能作为从设备，无法接入蓝牙耳机功能	低
硬件转发	1、跨平台（Android、Linux、QNX） 2、同时支持输入和输出 3、不存在混音还原的问题	1、部分车镜设备，由于集成封装性太强，没有暴露出可对接的线束。这类设备无法覆盖 2、需要针对车机进行线束定制来实现整体的动态封装性 3、需要配套的硬件成本	摸底 前装 100% 后装 58%

基于上述情况，考虑到车载的应用场景。最终我们的选型是 「软件 hook」 +「硬件转发」的组合方案。

5.音频编码传输

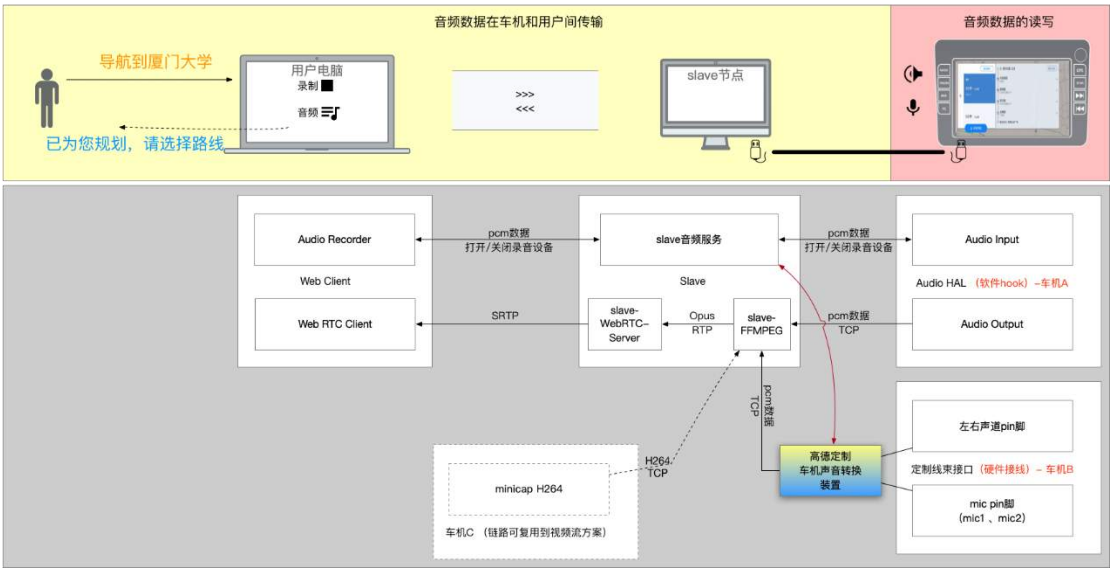
关于音频编码传输这部分的内容，行业中已经有较成熟的解决方案，因此，这部分不展开篇幅讨论，我们仅针对一些方案做选型评估：

推流方案	音频格式	丢包情况	延迟情况	问题
rmtp	aac/mp3	无丢包	有叠加延迟,延迟高 2-5s	
httpflv	aac/mp3	无丢包	有叠加延迟,延迟高 2-5s	
rtp	aac/mp3	存在丢包可能	测试环境 vlc 软件：1s 延迟为 vlc 网络缓冲 1s。通过 vlc 设置 ctrl+p --> 全部 --> 输入/编解码器 --> 网络缓存(ms) 改为 100， 太小容易卡顿	浏览器不支持
webrtc	opus	存在丢包可能	500ms	需要配合 webrtc 服务端，少数部分浏览器不兼容

综上，从我们的应用场景以及高实时性要求考虑，最终选取了 webRtc 的方案。

6.最终选型

结合音频的获取和写入以及整体编码传输的方案，最终的技术方案选型图如下：



对应流程图中，也顺带涵盖了远程画面传输的视频流优化的参考链路。

7.总结

通过软硬件组合的方案来实现音频数据读写的能力，是一种基于特定背景条件下解决方案。但其基本推演的思路和策略，也是适用于手机平台的。而其中硬件的解决方案，理论上是适用于 Android、iOS、Linux、QNX 等平台设备的。

相较来说，手机的硬件转发成本更低。而对于软件的方案，实际的播报效果上仍会有很多细节问题，比如播报声音太小，需要对应设备去调节播报音量比例；出现延迟的场景，可能需要修改采样率；或是需要 hook 的自动化来降低成本等等。最终落地到项目中时，还需要考虑各方面的适配成本，确保整体的投入产出比。

招聘

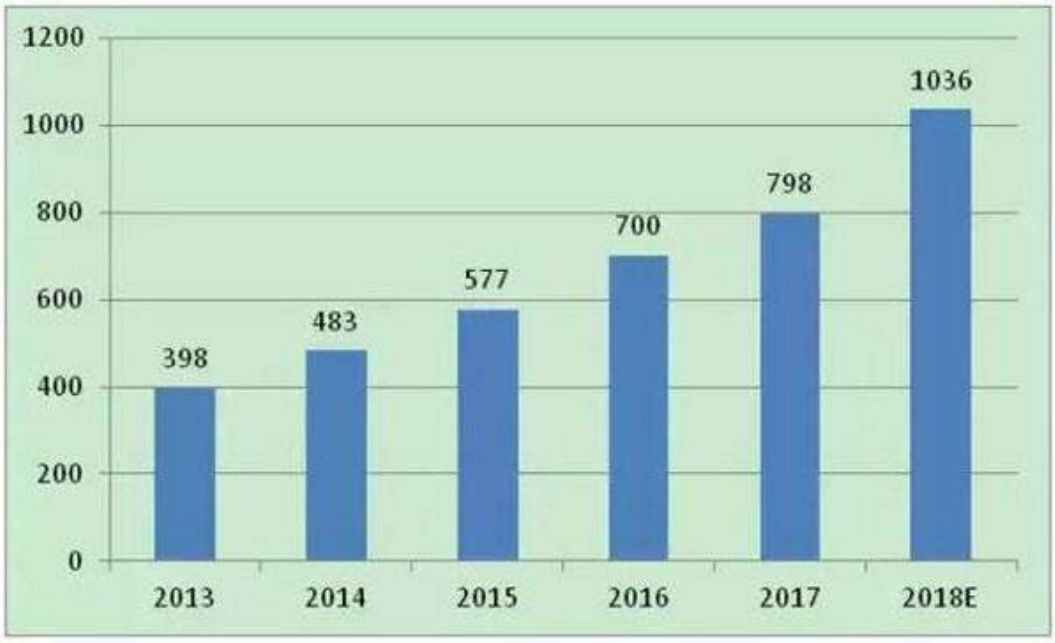
高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

IoT 时代：Wi-Fi“配网”技术剖析总结

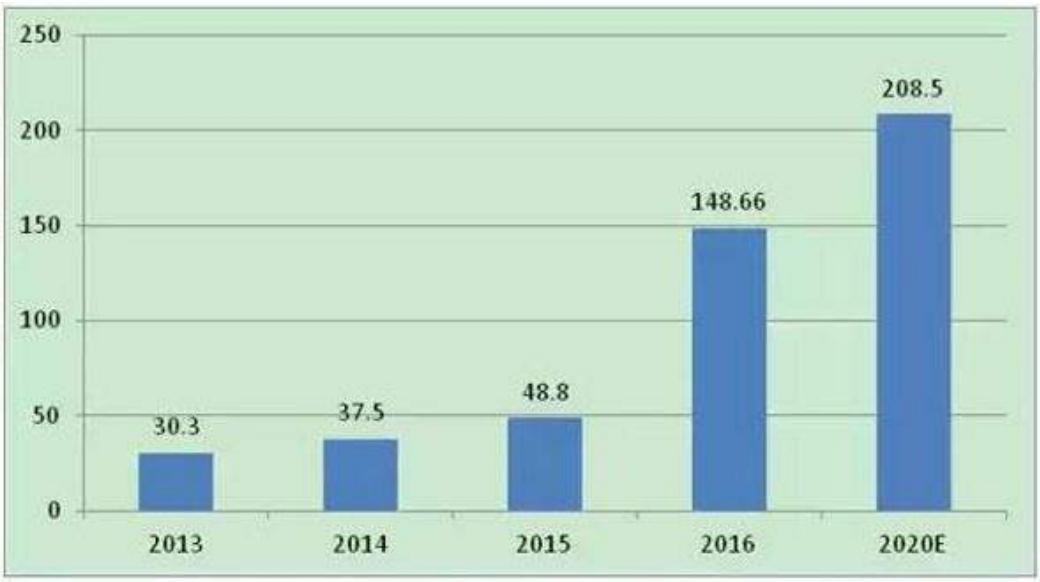
作者：若之

1.导读

近年来，物联网市场竞争激烈，从物联网平台厂商，设备生产商，到服务提供商，都在涌入这片红海。预计到 2020 年，全球联网设备数量将达到 260 亿个，年复合增长率达到 20%；全球联网设备带来的数据将达到 44ZB，这一数据将是 2012 年的 22 倍，年复合增长率 48%。



2013-2018 年全球物联网市场规模预测图(单位：亿美元)

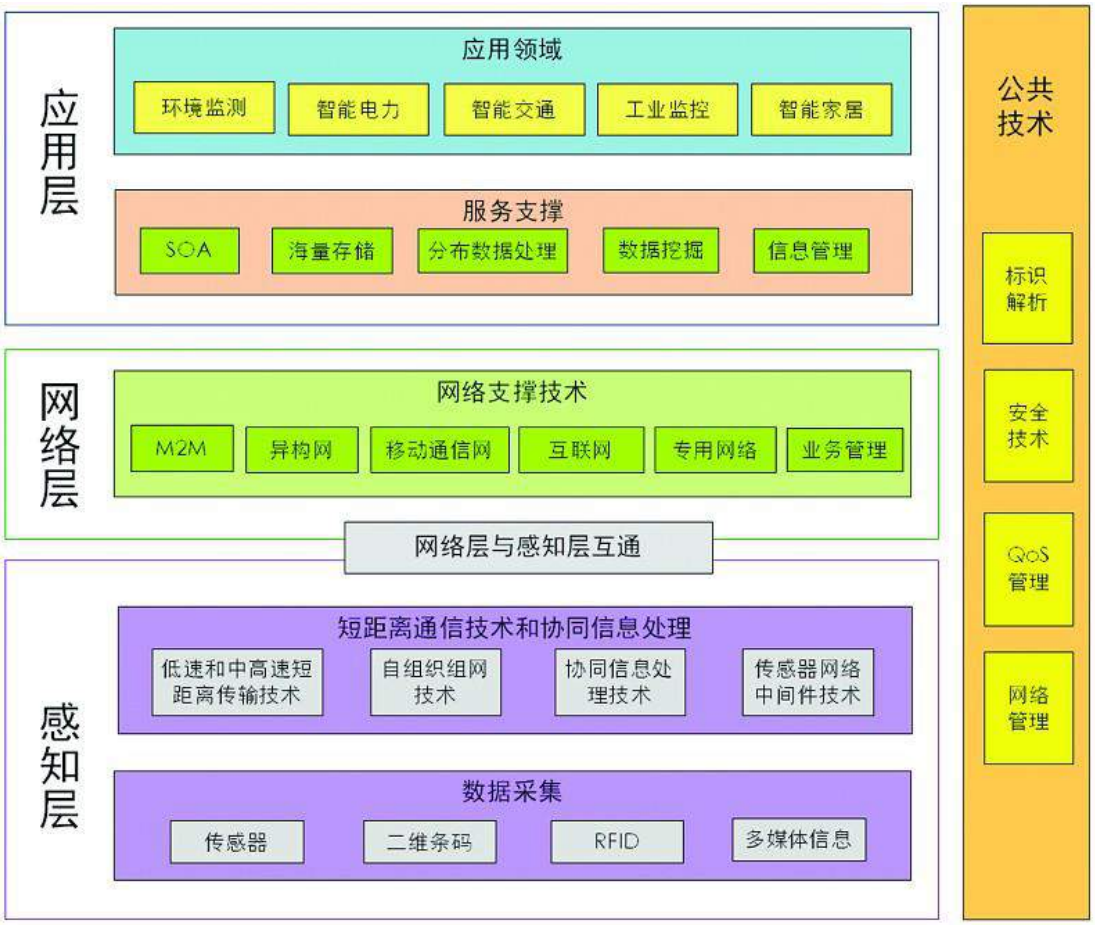


2013 年-2020 年全球物联网设备数量及预测图(亿台)

2.物联网时代对网络的需求

2.1 物联网系统层次

物联网系统从架构上划分为三个层次：感知层、网络层、应用层：



感知层：解决的是人类世界和物理世界的数据获取问题，由各种传感器以及传感器网关构成。该层被认为是物联网的核心层，主要是物品标识和信息的智能采集，它由基本的感应器件（例如 RFID 标签和读写器、各类传感器、摄像头、GPS、二维码标签和识读器等基本标识和传感器件组成）以及感应器组成的网络（例如 RFID 网络、传感器网络等）两大部分组成。该层的核心技术包括低速和中高速短距离传输技术、自组织组网技术、协同信息处理技术，传感器网络中间件技术等，涉及的核心产品包括传感器、电子标签、传感器节点、无线路由器、无线网关等。

传输层：也被称为网络层，解决的是感知层所获得的数据的接入和传输功能，是进行信息交换、传递的数据通路。物联网传输层分为有线通信传输层和无线通信传输层。有线通信技术包括中长距离的广域网络和短距离的现场总线；无线通信层分为长距离的无线局域网、中短距离的无线局域网和超短距离的无线局域网。而由于物联网的网络层承担着巨大的数据量，并且面临更高的服务质量要求，物联网需要对现有网络进行融合和扩展，利用

新技术以实现更加广泛和高效的互联功能。

应用层：也可称为处理层，解决的是信息处理和人机界面的问题。网络层传输而来的数据在这一层里进入各类信息系统进行处理，并通过各种设备与人进行交互。处理层由业务支撑平台（中间件平台）、网络管理平台（例如 M2M 管理平台）、信息处理平台、信息安全平台、服务支撑平台等组成，完成协同、管理、计算、存储、分析、挖掘、以及提供面向行业和大众用户的服务等功能，典型技术包括 SOA 技术、海量存储、分布数据处理、数据挖掘、信息管理等先进技术可被广泛采用。

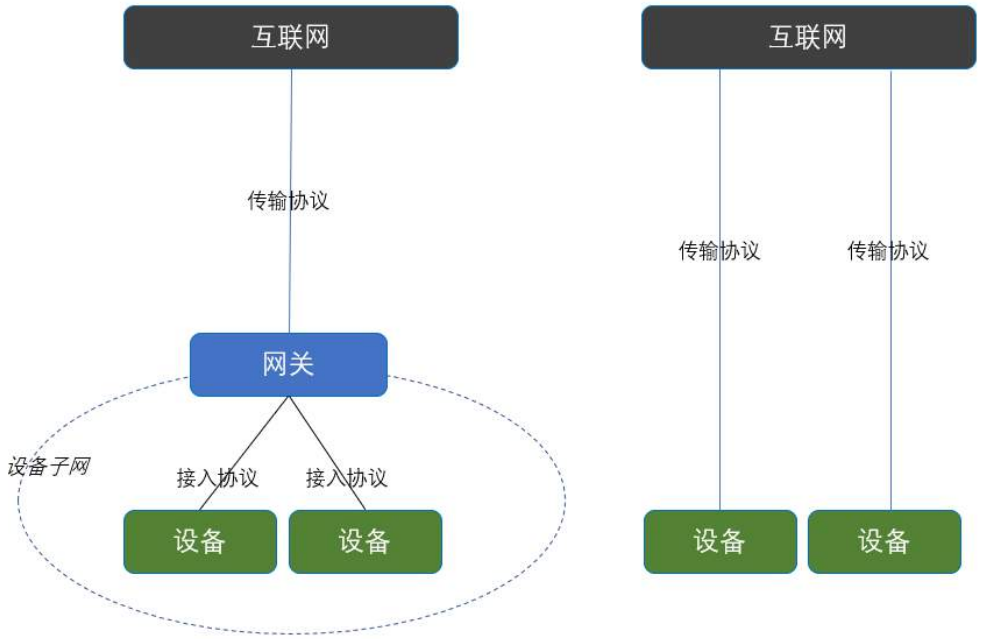
在各层之间，信息不是单向传递的，可有交互、控制等，所传递的信息多种多样，包括在特定应用系统范围内能唯一标识物品的识别码和物品的静态与动态信息。

尽管物联网在环境监测、智能电力、智能交通、工业监控、智能家居等经济和社会各个领域的应用特点千差万别，但是每个应用的基本架构都包括感知、传输和应用三个层次，各种行业和各种领域的专业应用子网都是基于三层基本架构构建的。

2.2 物联网接入协议与传输协议的区别

我们将物联网通信协议分为两大类，一类是接入协议，一类是传输协议：

物联网在设备连接方面的两种架构



接入协议一般负责子网内设备间的组网及通信，接入协议大多都不属于 TCP/IP 协议族，只能用于设备子网（设备与网关组成的局域网）内的通讯；传输协议主要是运行在传统互联网 TCP/IP 协议之上的设备通讯协议，负责设备通过互联网进行数据交换及通信。

采用接入协议的物联网设备，需要通过网关进行协议转换，转换成通讯协议才能接入互联网。而采用通讯协议的物联网设备，则可以直接接入互联网。

常用的接入协议包括 Wi-Fi、RFID、NFC、ZigBee、Bluetooth、LoRa、NB-IoT、GSM、GPRS、3/4/5G 网络、Ethernet、RS232、RS485、USB 等等；常用的通讯协议包括 HTTP、CoAP、MQTT、XMPP、AMQP、JMS 等。接入协议位于网络层次架构中的物理/数链层，通讯协议位于应用层。

物联网接入协议和通讯协议区别如下：

类别	接入协议	通讯协议
传输范围	子网通讯（设备+网关）	互联网通信（TCP/IP协议族）
接入互联网方式	需要网关	直接接入
层次	物理/数链层	应用层

那么，既然有了可以直接接入互联网的通讯协议，那么接入协议的意义何在呢？接入协议的优势说起了，相对于通讯协议，接入协议所依赖的硬件资源要求更低，功耗更低，网络传输的数据量也更小，因此，在控制领域等一些场景中更具优势。

这些场景中，物联网设备往往没有外接电源，因此要求功耗尽可能低，比如，一节纽扣电池能够供电一年左右。这样的要求是 HTTP 等协议的所需的硬件环境难以胜任的。

2.3 常用的几种物联网接入协议

目前市场上常见的接入协议有 ZigBee、蓝牙以及 Wi-Fi 协议等：

- ZigBee 目前在工业控制领域应用广泛，在智能家居领域也有一定应用。它有以下主要优势：

①低成本：ZigBee 协议数据传输速率低，协议简单，所以开发成本也比较低。并且 zigbee 协议还免收专利费用。

②低功耗：由于 ZigBee 协议传输速率低，节点所需的发射功率仅 1mW，并采用休眠+唤醒模式，功耗极低。

③自组网：通过 ZigBee 协议自带的 mesh 功能，一个子网络内可以支持多达 65000 个节点连接，可以快速实现一个大规模的传感网络。

④安全性：使用 crc 校验数据包的完整性，支持鉴权和认证，并且采用 aes-128 对传输数据进行加密。

ZigBee 协议的最佳应用场景是无线传感网络，比如水质监测、环境控制等节点之间需要自组网以相互之间传输数据的工业场景中。在这些场景中 ZigBee 协议的优势发挥的非常明显。目前国内外很多厂商也将 ZigBee 运用在智能家居方案中。

- 蓝牙协议大家都非常熟悉了，特别是随着蓝牙 4.0 协议推出后发展迅速，目前已经成为智能手机的标配通信组件。蓝牙 4.0 之所以在近几年发展迅速，主要有以下两点原因：

①低功耗 :我认为这个是蓝牙 4.0 的大杀器,使用纽扣电池的蓝牙 4.0 设备可运行一年以上，这对不希望频繁充电的可穿戴设备具有十分大的吸引力。当前基本世面上的可穿戴设备基本都选用蓝牙 4.0 方案。

②可手机接入 :近年来支持蓝牙协议基本成为智能手机的标配，用户无需购买额外的接入模块。

- Bluetooth 最大的优点是不依赖于外部网络、便携、低功耗。只要有手机和智能设备，就能保持稳定的连接，走到哪连到哪。所以大部分运动和户外使用的设备都会优先考虑 Bluetooth。它的主要不足是：不能直接连接云端，传输速度比较慢，组网能力比较弱。

- Wi-Fi 协议和蓝牙协议一样，目前也得到了非常大的发展。由于前几年家用 Wi-Fi 路由器以及智能手机的迅速普及，Wi-Fi 协议在智能家居领域也得到了广泛应用：

①Wi-Fi 可以直接接入互联网：相对于 ZigBee，采用 Wi-Fi 协议的智能家居方案省去了额外的网关，相对于蓝牙协议，省去了对手机等移动终端的依赖。

②Wi-Fi 最大的优点是连接快速、持久、稳定，它是 IoT 设备端连接的首选方案，唯一需要考虑的是智能设备对 Wi-Fi 覆盖范围的依赖导致 smart devices 的活动范围比较小，不适合随时携带和户外场景。

相当于蓝牙和 ZigBee，Wi-Fi 协议的功耗成为其在物联网领域应用的一大瓶颈。但是随着现在各大芯片厂商陆续推出低功耗、低成本的 Wi-Fi soc（如 esp8266），这个问题也在逐渐被解决。

种类	ZigBee	蓝牙	Wi-Fi
标准	IEEE802.15.4	IEEE802.15.1x	IEEE802.11b/n/g
单点覆盖距离	50米	10米	50米
功耗	5mA	20mA	10-50mA
复杂性	简单	复杂	复杂
传输速率	250Kbps	1Mbps	10Mbps
频段	2.4GHz	2.4GHz	2.4GHz
网络节点数	65000	8	50
联网所需时间	30ms	10s	3s
安全性	128bit AES	64bit 128bit	SSID
成本	低	低	一般
安装难度	简单	一般	一般
优点	可自组网	体积小，受众群体广	容易实现，受众范围大
缺点	难兼容，其搭载的设备普及率极低	连接能力有限	连接能力有限
主要应用	无线传感器（汽车轮胎）、医疗等	通信、汽车、工业、医疗等	多媒体、无线上网、PAD、PC等

2.4 何谓“配网”

2.4.1 WIFI 的 “联网”和“自动联网”

- 连网：一般指的是 Wi-Fi 设备通过 SSID 和密码来连接热点 AP 或路由器，以加入后者所建立的网络的过程。

- 自动连网：一般指的是 Wi-Fi 设备在启动、掉线、或扫描到特定的 SSID 后，会使用之前保存的 SSID 与密码，自动连接热点 AP 或路由器，而不需要手工重新输入。其中，WIFI 设备掉线后的“自动连网”，又常常被称为“自动重连”。

- 自动连网：一般需要在之前配网成功后，将 SSID 和密码进行保存，以便在需要“自动连网”时可以从保存的地址读取出来使用。

2.4.2 Wi-Fi 的“配网”

“配网”指的是，外部向 Wi-Fi 模块提供 SSID 和密码，以便 Wi-Fi 模块可以连接指定的热点或路由器并加入后者所建立的相关 Wi-Fi 网络。

Wi-Fi 模块一般不像电脑手机或平板等设备，有丰富的人机交互界面，可以方便的实现配网，因此，Wi-Fi 模块的“配网”方式支持，会成为 Wi-Fi 模块特性的一个基本话题。

能提供方便、灵活多样、条件约束少的配网方式，常常成为 Wi-Fi 模块的卖点之一，更是 Wi-Fi 模块的使用者，在选型时需要慎重考虑评估的一个重要方面。

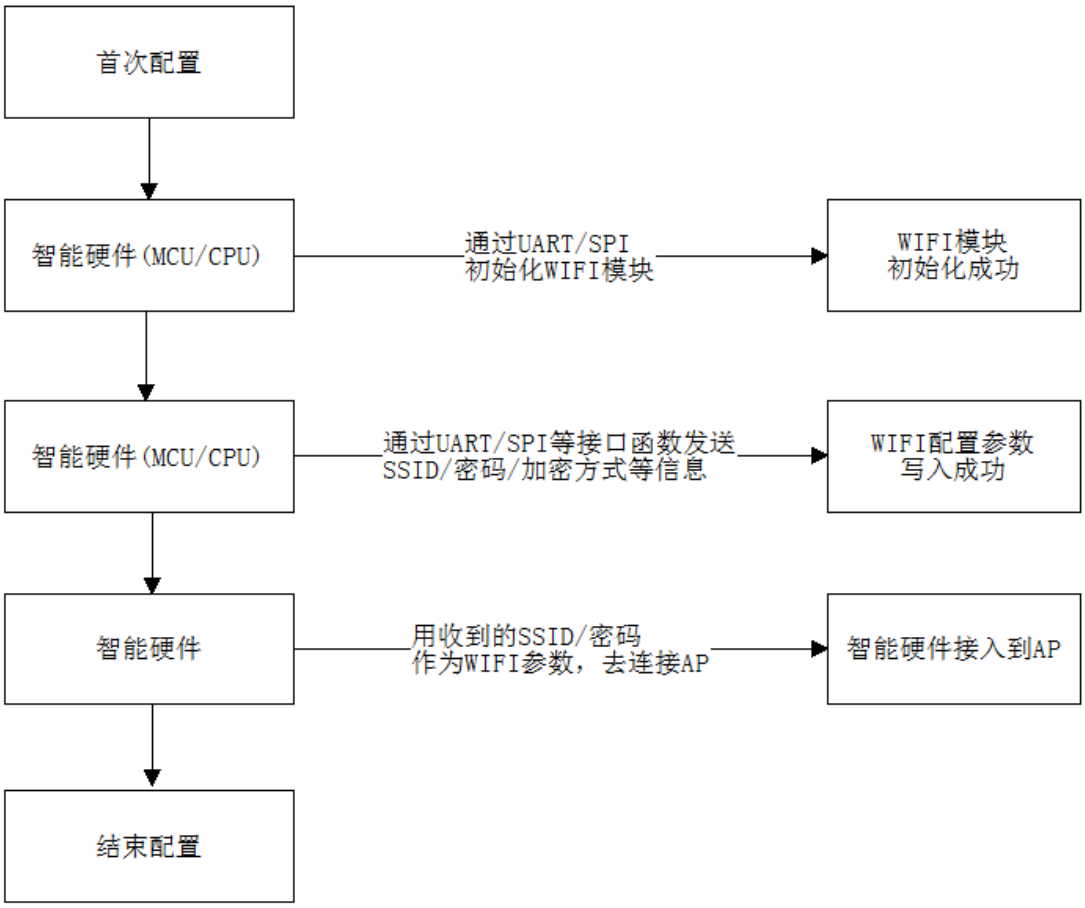
3.Wi-Fi 常用配网方式及原理实现

常见的配网方式，可归为如下几大类：直接配网、WPS 配网、WEB 配网、SoftAP 配网、智能配网配网、声波配网。用户可以根据具体的使用场合选择各种最适合的配网方式。

3.1 直接配网

所谓直接配网，就是通过 UART 串口、SPI 口、SDIO 口、I2C 等主机接口，按照一定的通信协议，将 SSID 和密码，直接传递给 WIFI 模块。Wi-Fi 模块在收到 SSID 和密码后去连接热点或路由器，并将连接的结果从主机接口返回。目前斑马车机采用的这种方式连接盯盯拍。

例如，常见的通过 UART 串口 AT 指令配网、SPI API 函数配网、SDIO API 函数配网、I2C API 函数配网等。

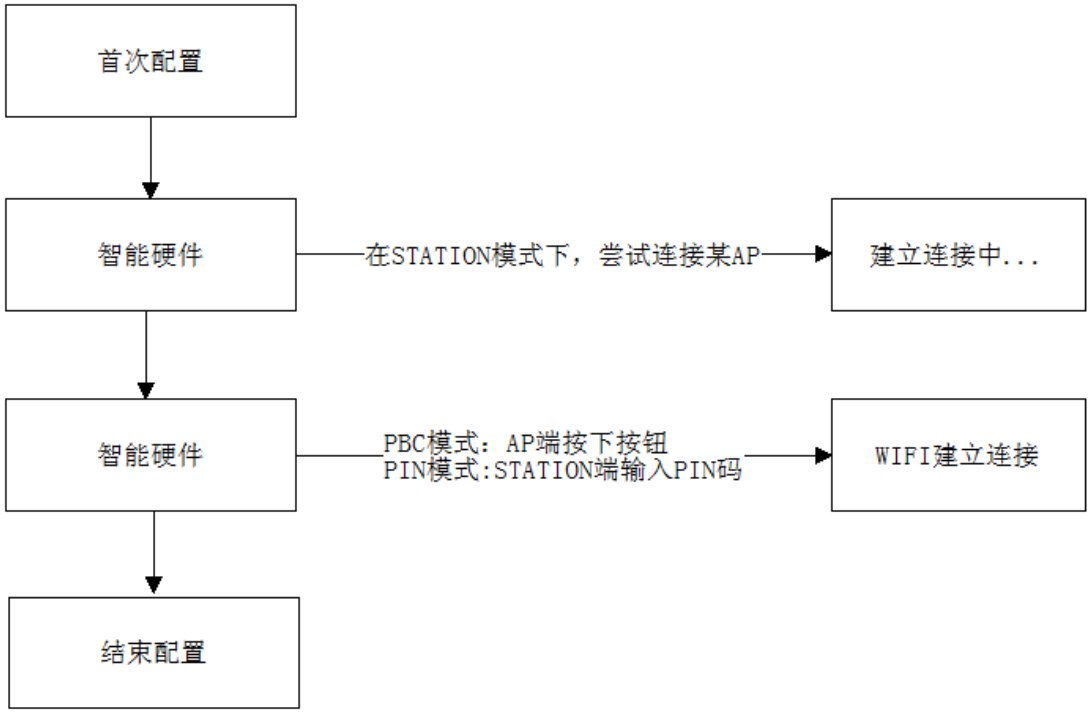


直接配网方式软件方案实现简单，但需要铺设其他的通信线路，比较适合于板载 WIFI 模块，或有其他协议传输线连接的设备间。因此对于环境要求比较高，需要在系统间有其它的通信链路存在。

3.2 WPS 配网

路由器中 WPS 是由 Wi-Fi 联盟所推出的全新 Wi-Fi 安全防护设定(Wi-Fi Protected Setup)标准，该标准推出的主要原因是为了解决长久以来无线网络加密认证设定的步骤过于繁杂艰难之弊病。WPS 用于简化 Wi-Fi 无线的安全设置和网络管理。它支持两种模式：个人识别码

(PIN)模式和按钮(PBC)模式。



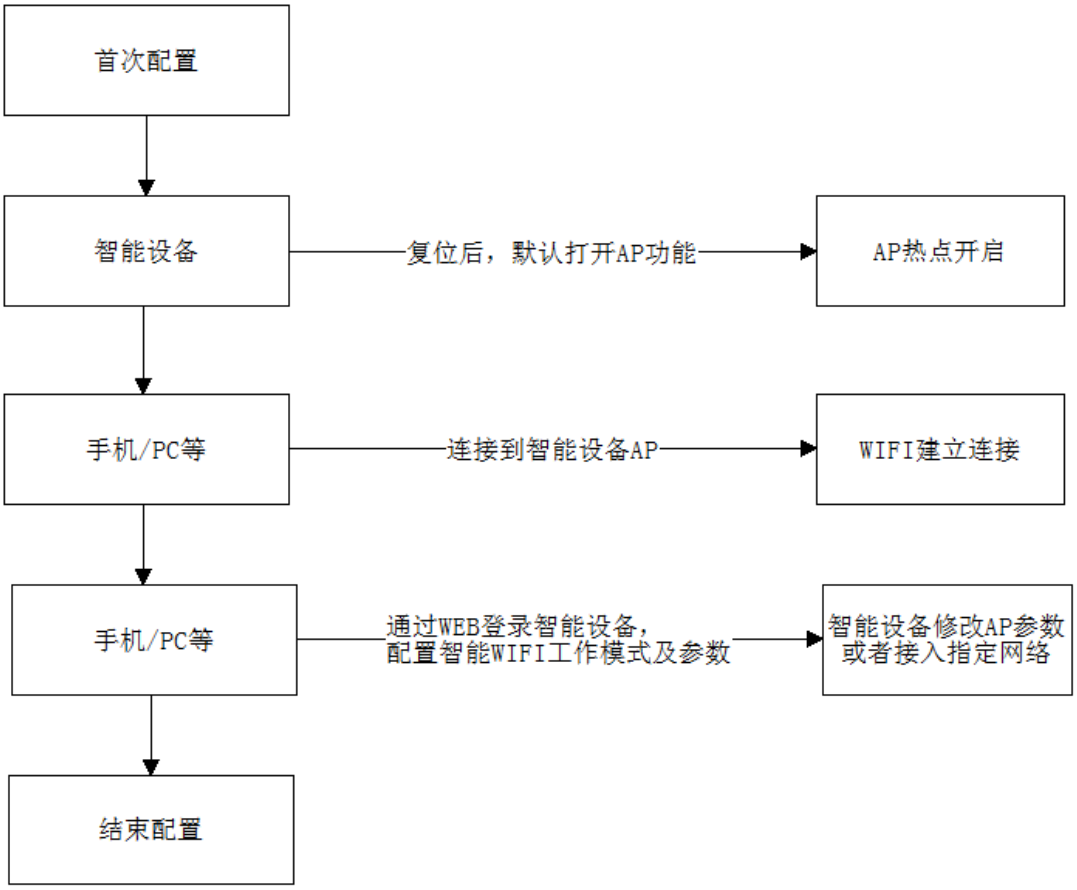
这种方式需要模块支持 WPS 功能。使用者往往会因为步骤太过麻烦，以致干脆不做任何加密安全设定，因而引发许多安全上的问题。因为安全性的缘故，近几年已经逐步被放弃，越来越多的路由器开始放弃或者自动关闭对这种方式的支持。

3.3 WEB 配网

在支持 AP 模式的 Wi-Fi 模块上内嵌一个简易的 WEB 服务器，在 WEB 网页里提供了配网的交互接口。其他网络设备（例如手机、平板、电脑等）直接连接上 Wi-Fi 模块的 AP 热点，在浏览器上打开该 WEB 网页，在 WEB 网页里配置该 Wi-Fi 模块去连接其他的 AP 或路由器。

归因于近年来越来越多的 Wi-Fi 芯片解决方案都开始支持 STA+AP 混合模式（即 WIFI 模块不仅可以作为工作站 STA 使用去连接其他路由器或热点，同时本身也可以作为一个热点 AP 供其他 WIFI 设备节点来连接），也归因于近年来许多 Wi-Fi 芯片解决方案越来越高的集成度可以将 TCP IP 协议栈直接集成在 Wi-Fi 模块上，因此，可以简单地在 Wi-Fi 模块上直接实现一个 WEB 服务器，且这个服务器可以通过 Wi-Fi 模块的 AP 模式直接访问（不需要依赖其他网络，手机等设备直接访问 WIFI 模块自建立的 Wi-Fi 网络和 WEB 网页，进行配置）。

这种配网方式的基本思想是，Wi-Fi 模块工作在 STA+AP 混合模式并启动内嵌的 WEB 服务器，电脑手机或平板等 Wi-Fi 设备连接 WIFI 模块所建立的 AP 热点，并获取得到一个 IP 地址（即：加入了这个 Wi-Fi 模块的热点 AP 模式所建立的 Wi-Fi 局域网），然后电脑手机或平板等 Wi-Fi 设备通过其上标配的浏览器访问 Wi-Fi 模块上的 WEB 服务器，在打开的 WEB 网页中，完成各种配置，包括设置 Wi-Fi 模块在 STA 模式下去链接第三方热点或路由器的 SSID 和密码，让 WIFI 模块作为 STA 去连接其他热点 AP 或路由器。



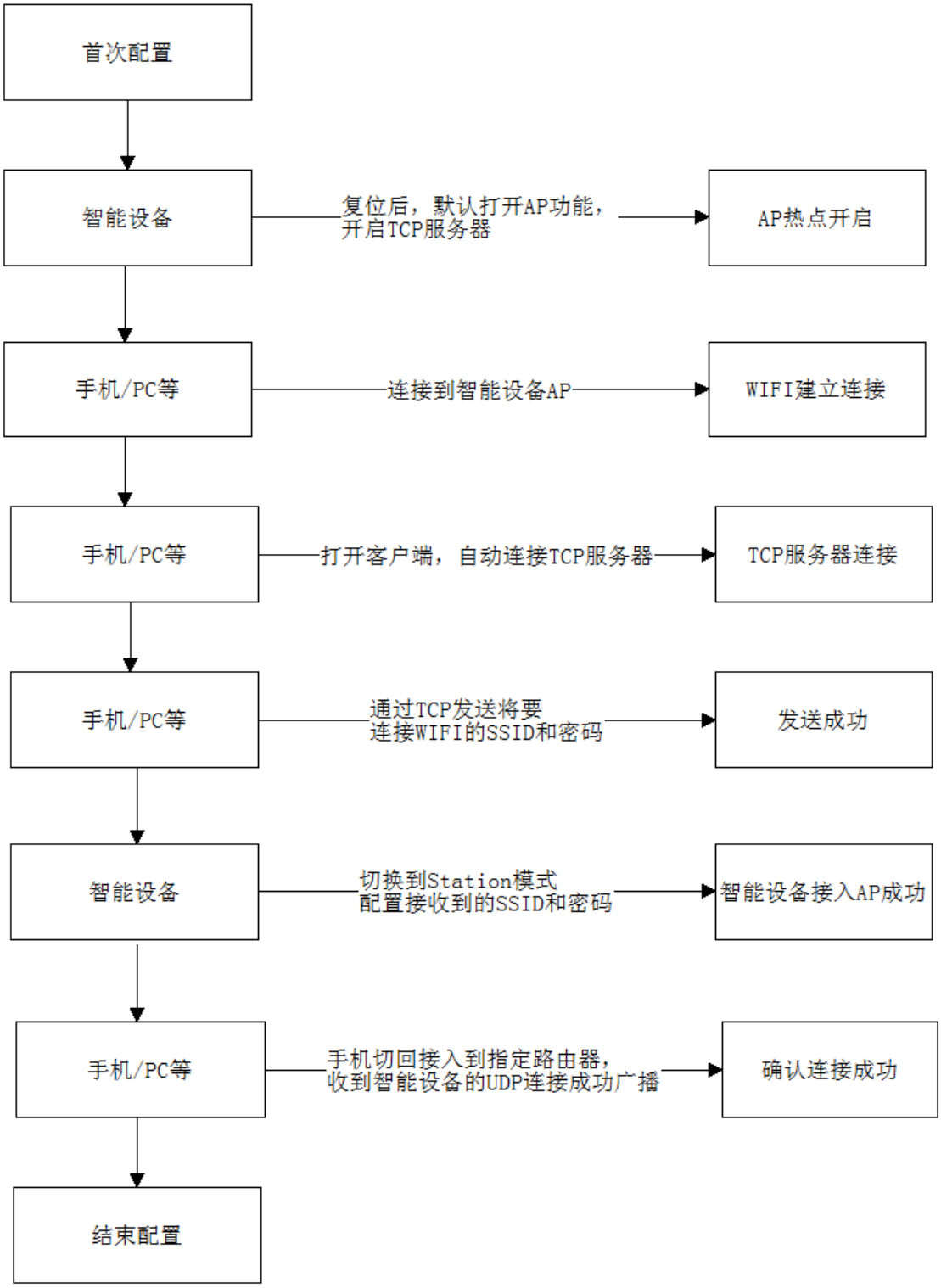
3.4 SoftAP 配网

SoftAP 配网方式在小米智能家居产品中被广泛应用。其原理是在 Wi-Fi 网络中另外启动 TCP 服务，通过 TCP 进行 SSID 和密码的配置，使智能硬件接入到指定的路由器。

在机器复位后，首先智能硬件会工作在 Wi-Fi 的 AP 模式，且开启 TCP 服务器，进入监听状态。

在此时，使用手机接入该 AP 热点，连接成功后，打开客户端，手机会去连接 TCP 服务器，三次握手连接成功后，则传输协议数据，内容包括指定智能硬件将要连接的 Wi-Fi 的 SSID 和密码。

硬件成功接收到手机发来的数据包解析得到 Wi-Fi 名字和密码。回复手机正在尝试连接了。关闭 AP 模式，开启 station 模式连接路由器，成功连接到指定的路由器。然后手机切回到指定路由器并开启 UDP 通讯，智能硬件用 UDP 协议广播配网成功数据。



3.5 智能配网（SmartConfig/SmartConnection….）

所谓智能配网，就是使用 Wi-Fi 设备本身自带的 WIFI 信号，在 MAC 层将 SSID 和密码按照一定的协议格式填充在 MAC 包中不加密的包头部分，采用广播和抓包方式，从手机等设备将 SSID 和密码分段多次传递给 WIFI 模块。

目前市面上常见的多种 SmartConfig/SmartConnection 技术，虽然各个 Wi-Fi 芯片方案会取不同的英文名字，但是基本原理则基本相同，只是填充的数据协议格式稍有区别。

厂商	芯片方案	技术名称	发包方式
TI	CC3200	Smartconfig	往某一固定IP发udp包
高通	QCA4004/QCA4002	SmartConnection	组播地址编码
MTK	MTK7681	Smartconnection	组播地址编码
Marvell	MC200+8801/MW300	Easyconnect	组播地址编码
Reltek	AMEBA	Simpleconfig	组播地址编码
乐鑫	Esp8266	Smartconfig	组播，通过长度编码
新案线	NL6621	Smartconfig	组播地址编码
微信	多家	Airkiss	全网广播，通过长度编码

智能配网一般需要在发送 SSID 和密码的设备（例如手机）上安装一个 APP，该 APP 实现了和 Wi-Fi 模块之间的协议交互（发送 SSID 和密码）。

这个功能最早是 TI 提出并应用于 CC3200 上;不过从原理上讲,只要芯片驱动支持开启混杂模式(Wi-Fi Promiscuous),就可以支持一键配网功能,只是各个厂家叫法及实现编码方式不同而已。

字段	DA	SA	LENGTH	LLC	SNAP	DATA	FCS
长度bytes	6	6	2	3	5	X	4

SNAP：格式数据包
DA:目标 MAC 地址
SA:源 MAC 地址
LENGTH:表示后面数据的长度
LLC:表示 LLC 头
SNAP:表示 3byte 的厂商代码和 2byte 的协议类型表示
DATA:载荷数据
FCS:帧检验序列

由于无线数据传播必定是广播的,所以必然可以被监听到；如果 AP 没有加密的话,UDP 直接可以把相关的信息发送出来.但是路由器 AP 一般都是加密的,而且加密方式不固定。

Wi-Fi 模块在无法直接解析出数据包。从 802.11 的 MAC 层帧格式中可以看到，链路层载荷数据(即网络层头部及网络层数)在数据帧中是清晰可辨的，只要接收到 802.11 帧就可以立刻提取出载荷数据，计算载荷数据的长度自不用说，而这里的载荷数据，通常就是密文。
在发送端，可以采用 2 种不同的编码发送方式：

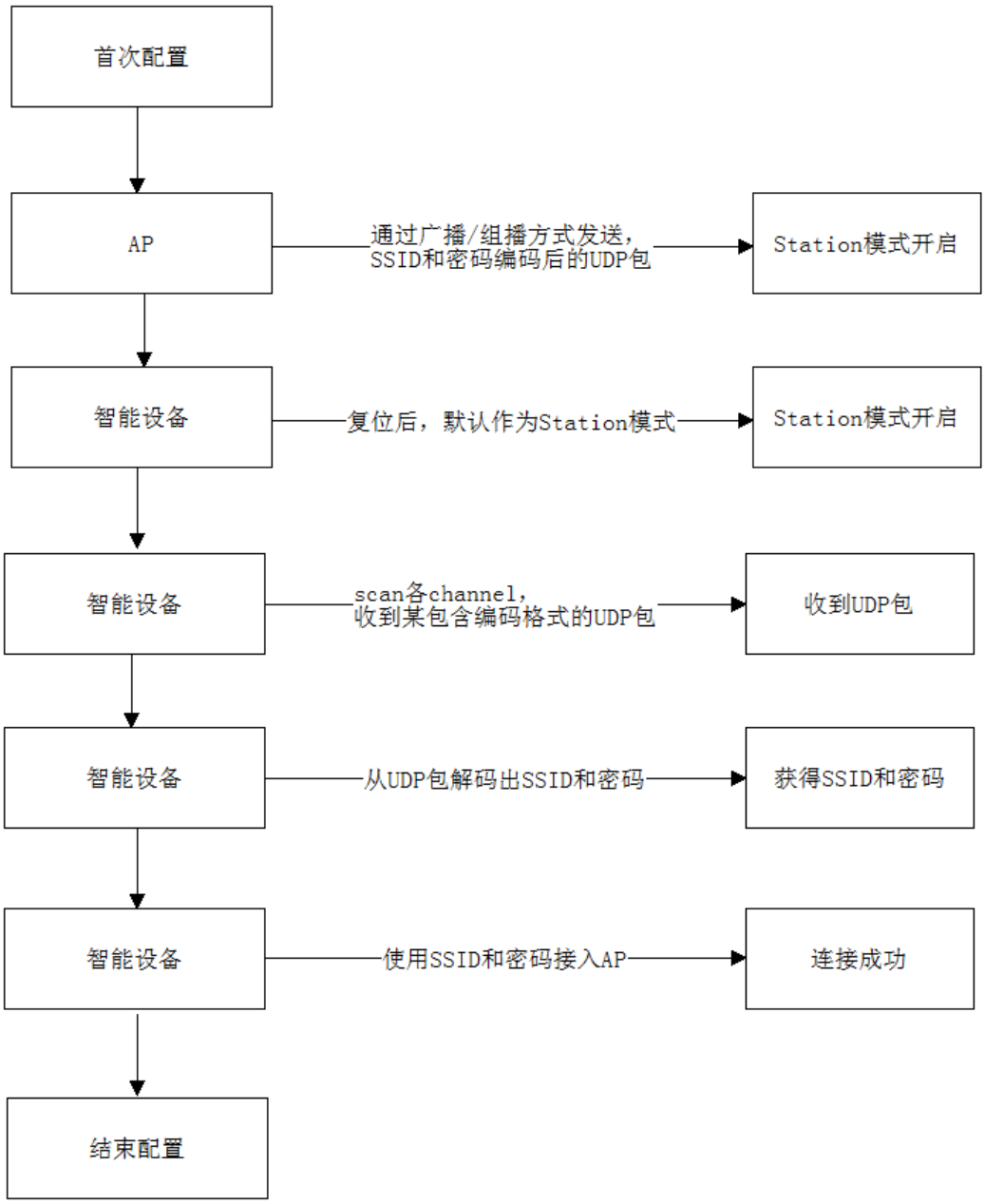
- UDP 广播：从 802.11 帧格式分析中获知,无线信号监听方的角度来说,不管无线信道有没有加密,DA、SA、LENGTH、LLC、SNAP、FCS 字段总是暴露的，因此信号监听方可以从这 6 个字段获取有效信息.从发送方讲,由于操作系统的限制,如果采用广播只剩下 LENGTH 发送方可通过改变其所需要发送数据包的长度进行控制.所以只要指定出一套

利用长度编码的通讯协议,就可利用数据包的 Length 字段进行数据传递。

- UDP 组播：组播地址是保留的 D 类地址从 224.0.0.0-239.255.255.255，IP 地址与 MAC 地址映射关系为:将 MAC 地址的前 25 位设定为 01.00.5e,而 MAC 地址的后 23 位对应 IP 地址的位;故发送端可以将数据编码在组播 ip 的后 23bit 中,通过组播包发送,接收端进行解码即可。

接收端进入一键配置功能后,Wi-Fi 智能硬件从信道 1 开始监听路由上的数据,如当前监听信道有符合规则的数据包,就停止信道切换,停留在当前信道接收完全部数据.否则就依次切换至信道 2.3.4....直到信道 14 后又从信道 1 开始继续监听依次循环;

当然,Wi-Fi 智能硬件可以在开启混杂模式之前,先行扫描当前环境下存在的 AP 获取所有当前 AP 的信道,然后只对当前扫描到的信道进行依次监听,如当前环境下只存在 2 个路由,分别在 1.6 信道,只需轮流扫描 channel1 和 channel6,这样可以提高配置效率。



3.6 声波配网

声波配网，即通过手机发出声波，将 SSID、password 等信息传给设备的一种配网方式。通过手机播放声波把 Wi-Fi 的初始化连接信息传递给智能设备，让设备识别完成 Wi-Fi 初始化流程建立网络连接。

一定程度上，声波传输可以理解为类似 NFC 的一种近场通讯技术。适用于没有触屏或触屏较小不易于信息输入，但是拥有麦克风的智能设备，如对话机器人，智能音响等。其优点是配网速度快、可人耳感知，缺点是受环境干扰较大。

实现声波配网，首先需要一套特定的算法库，算法库分手机端和设备端两部分。手机端算法库将 ssid 信息由字符串转化为声音信号（PCM），然后将声音信号通过音频模块播放出来。

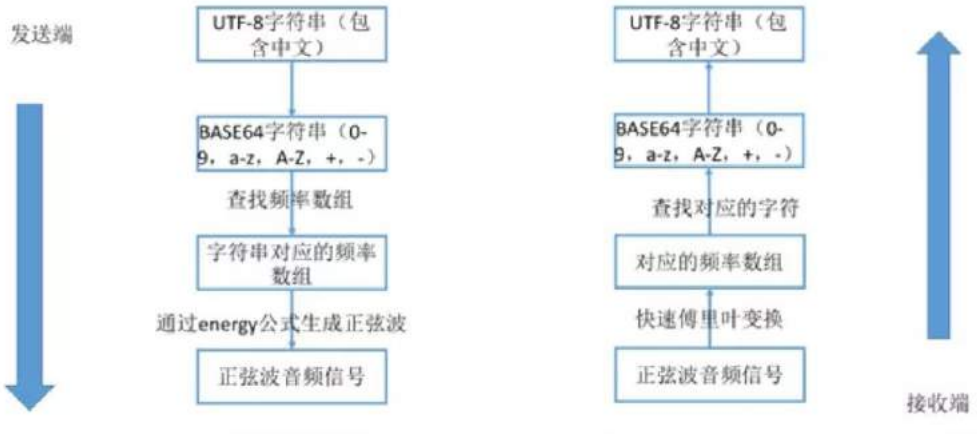
同时，设备端录下这一段声音，然后用同一套算法库将声音信息解析出来，还原成原来的 ssid 信息（字符串），最后用解析到的 ssid 信息用于连接 WIFI。

编解码可选择范围分为低频、中频、高频三种，其中低频的频率范围为 2K~5K，中频的范围为 8K~12K，高频的范围为 16K~20K。频率越高，声音越尖锐，抗噪性能越强。

显然声波配网技术中的技术难点就是声波传输技术。而声波传输的应用其实已经很广啦：支付宝的声波支付，QQ 音乐中的歌曲的声波分享，茄子快传，蛐蛐儿等。

其实原理很简单，可以近似理解为对称加密，加解密的过程大概如下所示：

仅传输 ASCII 可打印字符。



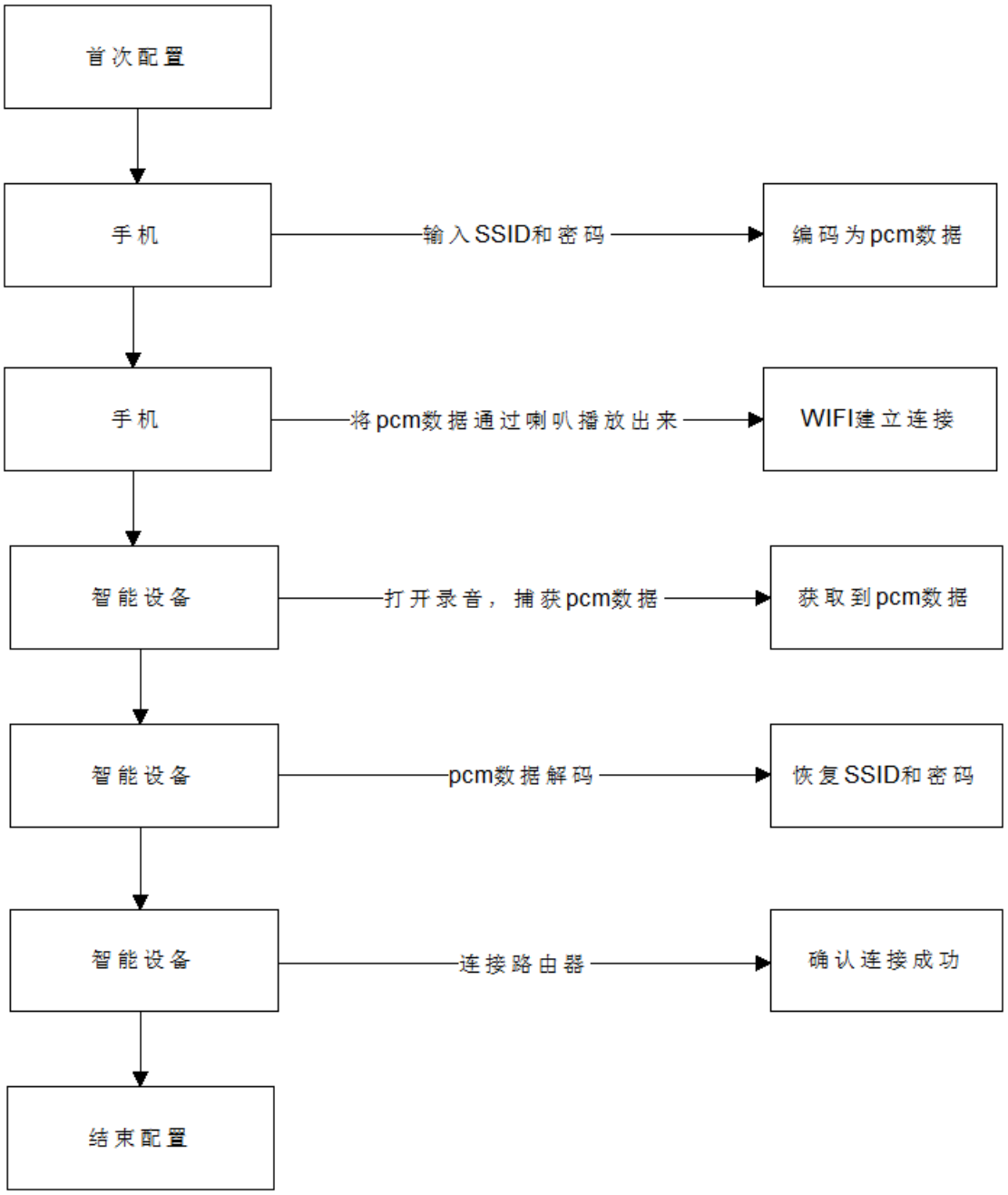
传输 UTF-8 字符串。

简单的说就是在发送端把你把要识别的字符映射成频率，然后把一个频率映射成一个音节信号（单频率的正弦波）编码成音频播放；在接收端接收到音频信号后，解析出频率，然后根据两边共同的码表找到频率对应的字符，从而解码出数据。

具体来说就是我们可以将 700HZ 的正弦波对应成字符'a'，800HZ 的正弦波对应数字'b'，900HZ 的正弦波对应数字'c'，以此类推。那么数字串"abc"就对应成频率串就是{700，800，900}，然后把这个频率串变成 3 个音节的正弦波音频。如果规定每个音节持续 100ms，则{700，800，900}对应 300 毫秒的音频段。接收方录制声音，对收到的声音进行解析，识别出 700HZ，800HZ，900HZ 三段正弦波频率，然后查找码表，解码出的字符串就是"abc"。

声波配网主要流程如下：

- 首先，在手机（或平板等其它一代设备）输入 ssid 信息（或获取当前或系统保存的 ssid 信息），将信息由 buffer 编码为 pcm 数据。
- 将使用算法库编码出来的 pcm 数据通过喇叭播放出来，同时，设备端打开录音，捕获 pcm 数据。
- 设备端将 pcm 数据通过算法库解码回原来的 buffer 数据。
- 从数据中解析出 ssid、password 等信息，并将其用于连接路由器。



4.IOT 场景下的 Wi-Fi 配网选择

上个章节中各配网方式对比汇总：

配网方式		优势	局限
直接配网	串口AT指令	直接输入，配网简单，过程明了，成功率高。	需提供额外的人机交互接口来输入SSID和密码，例如串口线等。
	SPI接口API		
WPS配网	个人识别码(PIN)模式	解决长久以来无线网络加密认证设定的步骤过于繁杂艰难之弊病。	不安全，没有做任何加密认证。
	按钮(PBC)模式		
SoftAP配网	AP->Station切换	跟智能配网难度类似，但是更加稳定	需要手机参与配置，过程较为繁琐
WEB配网	WEB网页配网	可以通过任意支持WIFI和浏览器的设备来配网，非常灵活实用。	需要一个支持Wi-Fi功能和浏览器功能的设备(例如，手机或电脑PC)来配合配网。
智能配网	智能配网	直接用手机配网，不需要提供额外的人机交互接口，也不占用额外的单片机主机资源。	1、手机需安装APP，iOS或Android版。 2、配网原理和过程复杂，成功率较低。 3、需要提供额外的按键或其他接口引导模块在智能配网和正常工作模式间切换。
声波配网	PCM编码	配网速度快、可人耳感知	目前方案大都收费，落地还需要时间；受环境影响大

Wi-Fi 作为最适合物联网连接的技术，它可以作为物联网的粘合剂。随着连接节点的无限激增，联网设备的覆盖面和总量也将随着快速增长。其内部集成了射频收发、MAC、基带处理、Wi-Fi 协议和配置信息及网络协议栈，用户利用它可以轻松实现串口设备的无线网络功能。用户在实际使用中可根据表中各配网方式的优劣选择。

高德今年发布天猫精灵高德版套装。车盒借助盯盯拍 mini3 行车记录仪的摄像头实现 AR 导航功能。在该案例中，充分考虑产品功能，论证其安全性，采用 SoftAP 的配网方式。

车联网服务 non-RESTful 架构改造实践

作者：云勤

1.导读

在构建面向企业项目、多端的内容聚合类在线服务 API 设计的过程中，由于其定制特点，采用常规的 restful 开发模式，通常会导致大量雷同 API 重复开发的窘境，本文介绍一种 GraphQL 查询语言+网关编排联合的实践，解决大量重复定制的问题。

早期与车厂合作过程中，基于高德已有的数据、引擎能力和一些较为重要的相关 CP 服务（如停车场、加油站、天气等），形成的在线服务协作模式是针对客户需求，采用 REST API 提供针对每个车厂、每个项目以及每个终端提供不同的 API 实现，然而数据核心独立服务实际上就有十余种，然而由于车线业务维护周期长，定制多，2-3 年下来，API 规模已达几百个，而且持续发散级增长，这给持续开发和维护带来不小挑战。

分解业务开发过程，无非两类工作，业务需求能力数据的获取和非业务诉求但是必不可少的如鉴权等通用化能力，当前来看，其实这两个问题是几乎所有业务团队都会遇到的问题，因此解决方案也基本类似，如服务聚合、流程编排、API 网关等。

本文简要介绍下车联网在线服务改造旧架构的一些实践。

2.有关名词

- **GraphQL**：GraphQL 既是一种用于 API 的查询语言也是一个满足数据查询的运行时。GraphQL 对 API 中的数据提供了一套易于理解的完整描述，使得客户端能够准确地获得它需要的数据，而且没有任何冗余，也让 API 更容易地随着时间推移而演进，还能用于构建强大的开发者工具。

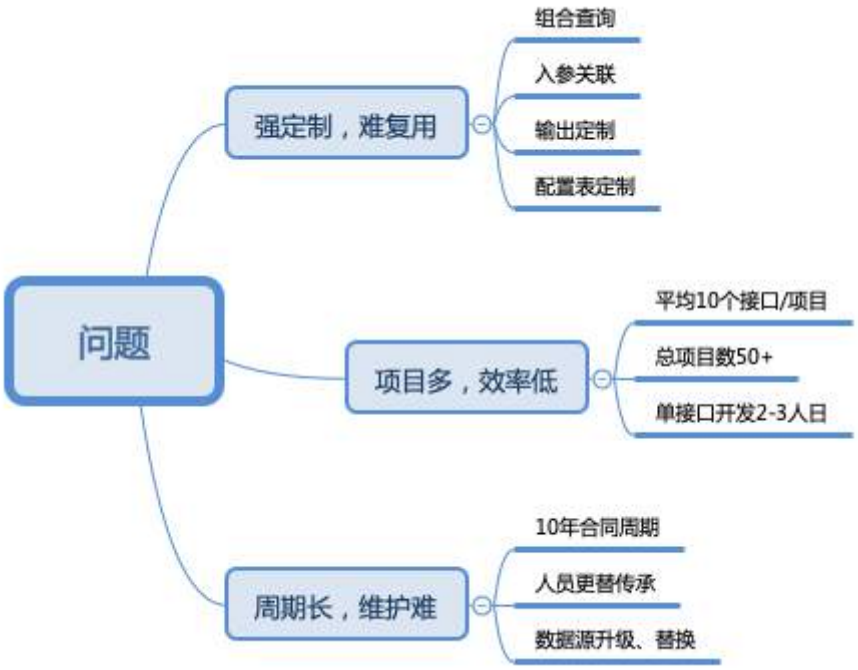
- **DSL**：指的是专注于某个应用程序领域的计算机语言。又译作领域专用语言。不同于普通的跨领域通用计算机语言(GPL)，领域特定语言只用在某些特定的领域。 比如用来显示网页的 HTML，以及 Emacs 所使用的 Emac LISP 语言。

- **API 网关**：API 网关是一个服务器，是系统的唯一入口。从面向对象设计的角度看，它与外观模式类似。API 网关封装了系统内部架构，为每个客户端提供一个定制的 API。它可能还具有其它职责，如身份验证、监控、负载均衡、缓存、请求分片与管理、静态响应处理。

3.存在的问题

车线业务在线服务旧架构如下：

面临以下问题：



4.思考与改进

针对上述问题，主要从以下几个方面思考改进：

- 服务能力原子化：目标是做稳，让上层通过组合实现业务需求。
- 构建查询引擎：支持强大的查询组合能力，实现原子服务能力任意聚合和定制。
- API 网关：对非业务数据能力需求进行抽象提供插件，实现插件编排。

下面分别介绍。

4.1 实现稳定、独立演进的原子能力服务

对已有的服务进行梳理，抽象出不同应该独立开发、部署演进的核心能力，对于引擎能力没有什么工作，重点是对于一些历史对接的外部 CP，主要实现以下目标：

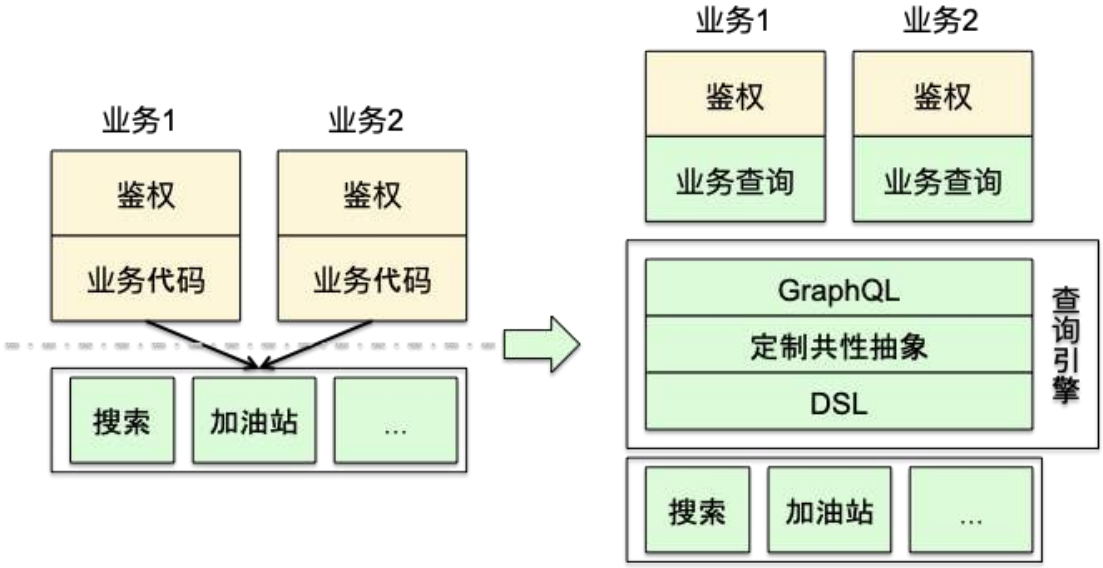
- 向上提供稳定接口，向下屏蔽底层复杂性（数据访问，多源差异）。
- 以位置为中心有机整合，构建完备原子化能力集合。

这部分工作主要是解决历史遗留的一些服务组合不合理，跟随业务过度定制的问题。

4.2 定制代码开发转换为定义查询语句

这里主要目的就是将服务聚合、定制逻辑等原来需要的代码开发转换为编写查询语言的方式实现，只需要编写出声明式的查询语句即完成服务发布，特性如下：

- 向上提供标准化查询语言
- 向下实现原子能力组合
- 归纳业务共性，提炼定制模式，提升复用



本文选择 GraphQL 作为查询语言基础，然而，直接采用 GraphQL 有这样两个主要问题需要解决：

- 数据查询 N+1 放大问题，直接采用 Fackbook 提出的 dataloader 来解决，原理是批量加缓存；
- GraphQL 规范限制，一些定制难以实现，如：

入参定制：如参数关联，类型转换等。

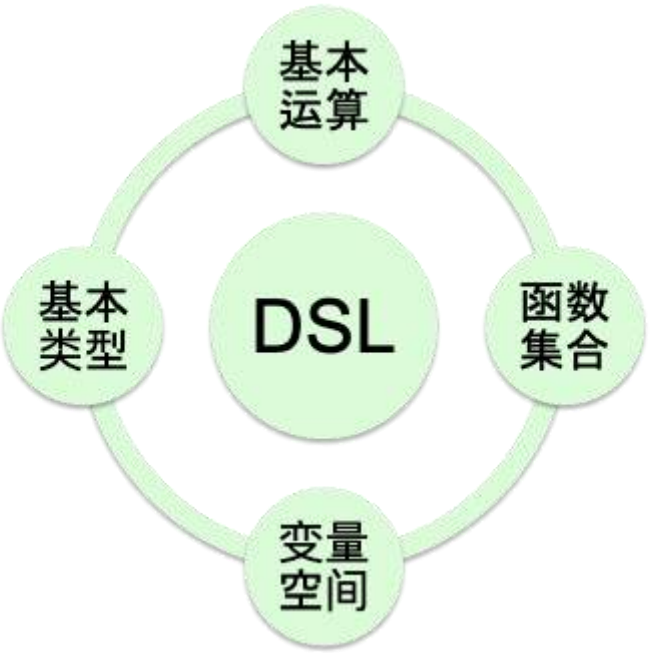
输出格式：字段展现形式，如时间、经纬度等。

配置表定制：主要是部分业务逻辑需要根据配置表定制，如深度返回字段等。

模型连接：原子能力服务尽可能独立，同时也无法枚举定义模型关系，但是定制业务需求需要大量关联透出，减少业务请求降低延时，所以模型自由关联能力是必要的，由于本方案最终的查询控制在内部，对外暴露 REST API，因此不会关联自由度造成的难理解性并不是一个问题。

需要通过嵌入简单的 DSL 实现：

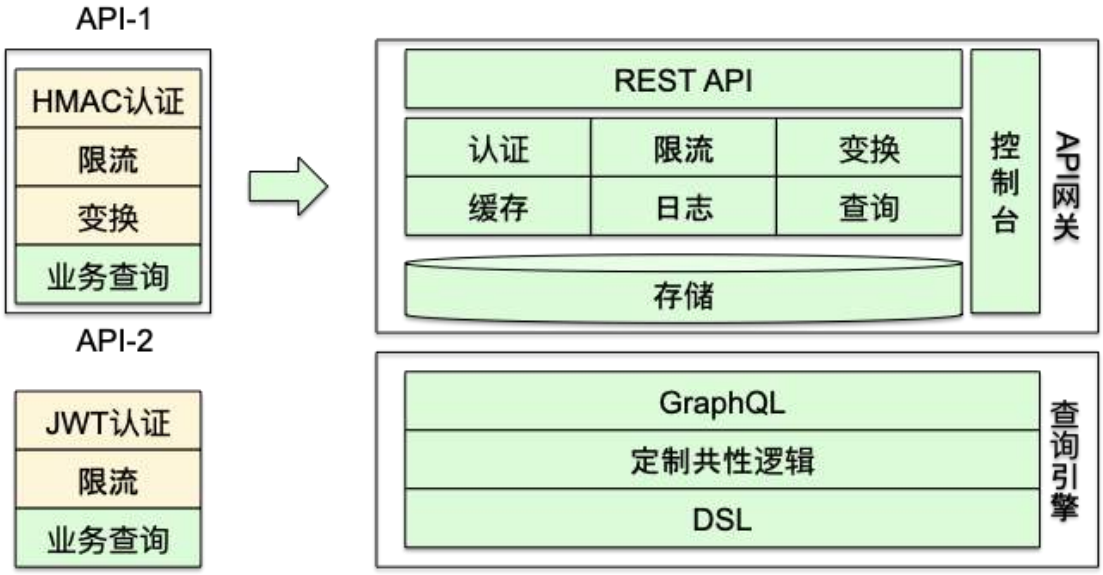
- 内置和自定义函数功能。
- 模型动态关联查询，上下文参数获取。
- 可以方便扩展自定义函数。



这里嵌入 DSL 需要控制好度，因为 DSL 如果过于复杂，那么，使用者或者发布者无法快速写出查询的话，对比写代码提效就会打折扣，偏离本来的价值，所以基本原则是简单、可扩展。

5.业务无关功能通过 API 网关插件配置化

由于之前每个 API 的定制开发基本所有功能混合在一起，能复用部分就是鉴权提供装饰器，常规性的响应格式定制提供一些工具函数，任何需求变更都需要变更代码，走发布流程，有了上面第一步的改造，这个步骤期望将非业务数据部分的定制功能抽象出处理链，每个处理节点提供多实现（包含通用和定制），通过数据库存储插件链实现编排。



车线业务由于鉴权方式需要根据客户定制，因此存在多样性，实现上是通过 Web 中间件实现多种鉴权插件：

- HTTP 签名，参考这里：主要面向 ToB（车厂后台、合作方）的请求。
- JWT 认证：主要面向车机、手机等终端。
- API Key。

对于 API 网关来说，这些鉴权插件并没有什么不同之处，只是工程要处理一些定制场景，比如对于不同车厂的 JWK 管理刷新策略，JWT 验证策略等，具体需要根据业务诉求抽象建模，通过插件属性来实现配置控制。

另外，网关还实现了一些变换器，主要用于将 GraphQL 的输出变换为 REST API 接口透出，这一方面由于一些旧接口要做兼容支持，另外，一些重点客户的全球化架构背景下自己已经完全定义好了接口式样，目前主要实现了：

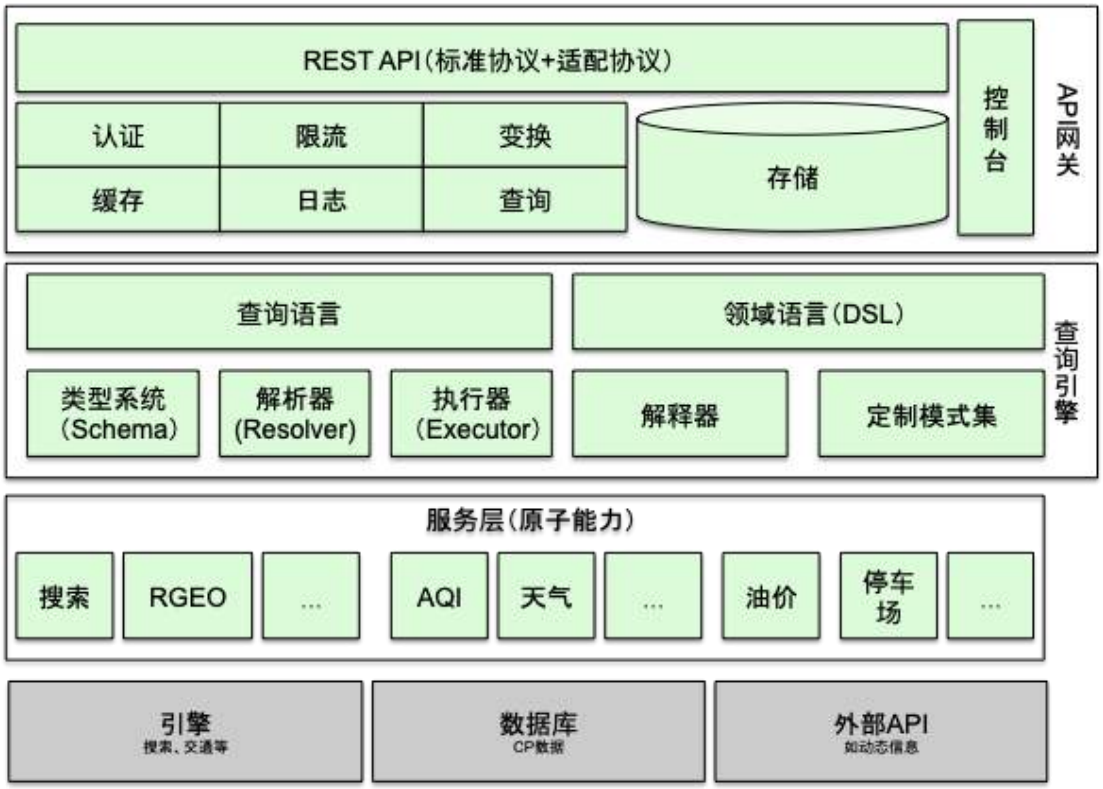
- 入参变换：使用 REST API 参数填充 GraphQL 查询模板。
- Header 变换：主要用于适配不同客户规范。
- JSON 变换，使用场景如下：

可复用标准接口，但是不同客户的响应结构规范不一致

定制非标接口，需要对 GraphQL 输出进行转换

而插件的使用则通过控制台或 API 实现将插件配置信息存储于数据库中进行管理，使用时根据请求特征从 DB 中提取并缓存起来使用。

改造后的新架构如下：



6.小结

通过上述改造，将车联网在线服务开发模式进行了升级，实现 API 控制台动态发布，大幅提升定制开发效率：

- 提效开发：正交化原子能力编排，通过轻量级定义取代定制化代码开发。

定制化开发占比下降 60%。

单接口开发从 2-3 人日→2-3 人时。

- 协议兼容：混合 REST 方案，对外提供标准协议、支持既有适配协议。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

车载导航应用中基于 Sketch UI 主题定制方案的实现

作者：龙枫

1.导读

关于应用的主题定制，相信大家或多或少都有接触，基本上，实现思路可以分为两类:

- 内置主题（应用内自定义 style）
- 外部加载方式(资源 apk 形式、压缩资源、插件等)

其实，针对不同的主题定制实现思路，没有绝对的好坏，每种实现方案都有其利弊，重要的是如何去权衡、选择，根据实际的项目需求，痛点，制定一个符合实际项目需求，能够解决主题定制过程中痛点的方案才是好的方案。

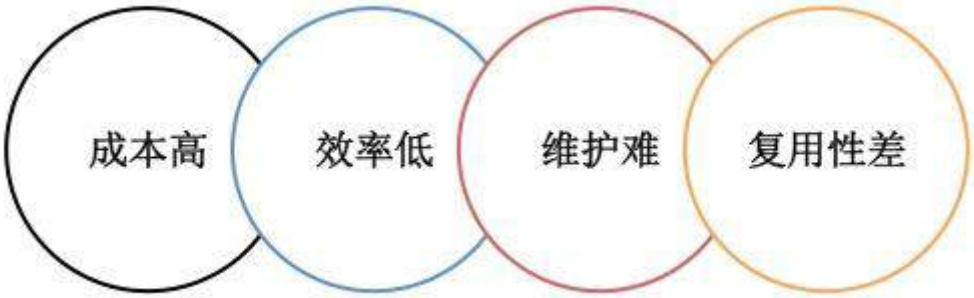
由于我和团队一直是做车载导航应用开发，面向的对象是客户。不同的客户对于应用的 UI 或者主题是有不同需求的，也就是说针对不同客户，不同渠道的版本，需要有不同的应用主题。

随着项目的增多，如果没有一个灵活，易管理，低成本的主题定制方案，那么实现将变得非常困难。

2.过去主题定制的实现方案

- 针对需要定制的 UI，研发增加对应的自定义主题控件。
- 布局由这些自定义主题控件搭建而成。
- UED 修改对应主题控件 xml 中对应的色值实现主题的定制。
- 研发集成 UED 配置的文件，实现主题定制。
- 研发出包，UED 进行主题定制还原度验证。

基本思路就是界面中需要主题定制的 UI，通过自定义主题控件实现，而自定义主题控件可以通过替换布局 xml 的方式实现主题色值等的替换，从而实现主题定制，但这种方案的缺点非常明显：



- 成本高：主题定制需要 UED 手动去修改 xml，但一般 UED 对于 xml 格式内容不是很了解，这样无疑学习及修改成本都非常高，其次也容易产生错误的修改，导致稳定性很差。

- 效率低：主题定制需要 UED 修改完 xml 后，发给研发，然后由研发替换 xml，流程多，效率低。

- 维护难：之前的主题定制，没有统一的东西管理，主题对于 UED 来说就是一堆的 xml 文件，很难进行维护。

- 复用性差：由于主题定制是通过 xml，如果主题控件修改了，xml 也就跟着修改，主题的复用就很难实现。

3.新方案的设计与实现

从上图中我们可以看出在主题定制的整个过程中，UED 参与了大部分的流程，只有在主题集成的时候，研发需要参与，其实说到底，UED 才是主题定制的 Owner，对于主题定制最理想的状态是研发提供一定的工具平台，UED 设计完主题后，可以直接集成到应用内验证回归，中间无需研发干预处理，整个主题定制流程都由 UED 走完。

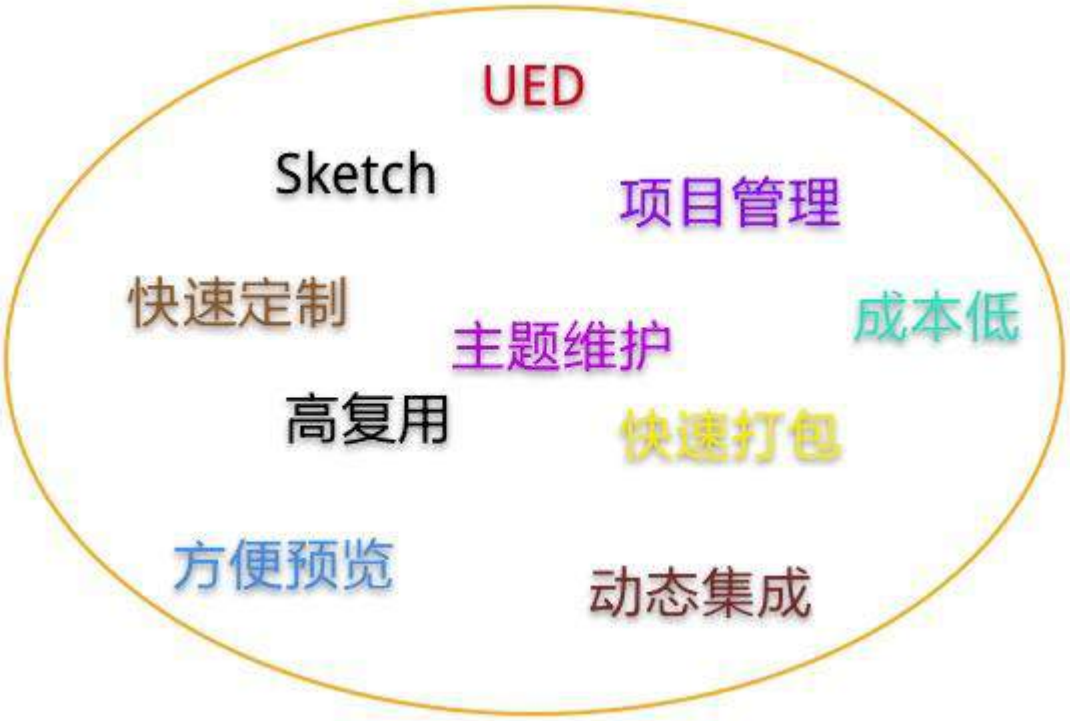
3.1 方案要素

基于 UED 参与就能完成主题定制的理念，同时考虑项目对于主题定制的需求，在新主题定

制方案的设计中，应当围绕核心角色，服务好核心角色，以最大限度降低核心角色的成本，提高整个定制过程的效率，同时兼顾项目核心需求为目标来设计、搭建新方案，以下是方案中应该关注的角色及要解决的关键问题：

- 主题定制主角：UED
- UED、UI 设计工具：Sketch
- 项目需求：

不同项目的不同渠道有不同的主题定制需求。
出包时就有对应主题，无需下载。
项目周期短，需要能快速定制。
项目需要维护，主题也需要方便维护、管理。



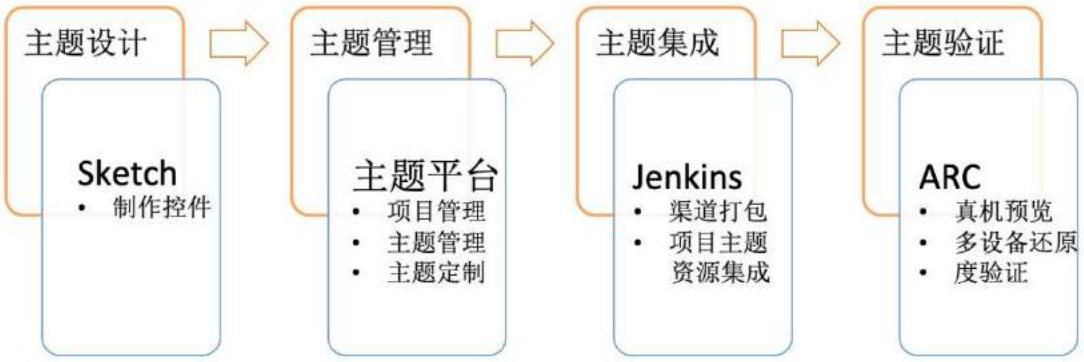
3.2 方案雏形

基于以上分析，主题定制的大体流程，方案如下：

- 首先，方案应当围绕着 UED 建设，由于 UED 是基于 Sketch 进行 UI 界面设计，因此如果主题定制也能基于 Sketch，UED 就能很方便的基于 Sketch 设计 UI 的同时也设计主题。
- 同时，不同的渠道会有不同的主题定制，并且需要快速定制，方便管理、维护，同时出包时就需内置主题，因此需要有一个主题管理平台能管理，维护主题资源，并且支持主题复制，使得主题资源能基于现有项目的主题进行少量修改，就能形成一个新的主题。

- 接着，需要在 Jenkins 打包时，能根据不同的渠道打包不同的主题资源到 apk 中。
- 最后，需要将打包好的 apk 快速安装到真机上进行还原度验收。

最终主题定制方案也就基本成型，基于 Sketch 的 UI 主题定制方案：



4.方案详解

基于 Sketch 的 UI 主题定制，主要分成四大步骤：

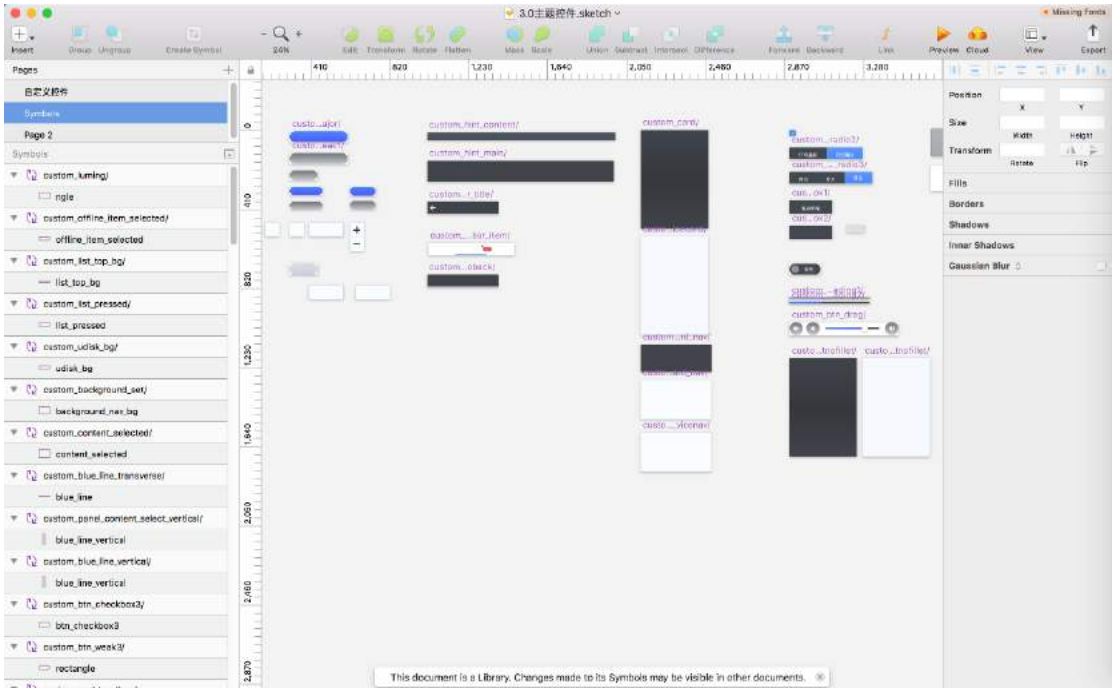
- UED 在 Sketch 中制作主题资源及上传云端（主题制作）
- UED 在 WEB 主题平台管理主题资源（主题管理）
- Jenkins 拉取主题资源打包到应用（主题集成）
- 真机预览效果（主题验证）

以上流程主要针对主分支，第一次界面开发，针对后续不同项目的主题定制，只需要在 WEB 平台中 copy 主题资源，然后进行对应的主题修改，即可快速定制出一套对应的主题出来。

以下详细介绍主要关键步骤：

4.1 主题资源制作

主题资源制作一般在界面设计前就需要提前设计好，UED 在 Sketch 中将主题控件设计好之后，我们提供了一个快捷的主题输入界面，方便 UED 能快速编辑对应控件的主题属性，并一键上传到云端，如下所示，在 Sketch 中进行主题控件制作：



4.2 WEB 主题平台管理主题资源

WEB 主题平台，保存了项目中所有可定制的主题资源，包含主题控件、插画、iconfont、文字大小等，在这里可以快速基于母版主题，copy 出新的主题资源，然后进行个性化定制，或者基于其他项目快速 copy，主题的定制不再是繁琐、耗时的操作，而变成了直观、易操作。

4.3 真机预览

在 WEB 主题平台编辑完主题资源后，通过 Jenkins 或服务器将主题资源打包到 apk 中，最后通过车机设备集群管理平台，将 apk 安装到不同的真机上，实现真机预览主题定制效果，UED 可快速回归还原度，发现问题，并快速在主题平台上修复。

5.方案对比

针对业内几种比较常用的主题定制方案与基于 Sketch 的 UI 主题定制方案进行了几个方面的对比：

	内置主题	外部加载方案（apk）	基于Sketch的UI主题定制方案
原理	在values文件夹中定义若干种style，通过setTheme方法设置主题	将主题包做成APK的形式，使用远程Context的方式访问主题包中的资源	通过sketch插件的能力生成主题资源，在apk编译时将对应主题资源打包进apk中
适用范围	主题定制需求、变更少的项目	单一渠道主题样式多，拓展性要求高的项目	针对定制项目，apk的不同渠道有不同主题，主题定制需求多，样式多的项目
灵活性	差，内置主题，新增、修改主题都需要修改代码	高，可在运行时新增	中等，打包时确定主题，无法在运行时加载新的主题
成本	较高，新增主题需要UED提供主题资源，研发编码	中等，需要制作并上线主题包	低，后台配置，打包签名包即可生效
稳定性	较高	较差，需要在代码中设置所有的可变资源，编码容易出错	较高，编译时替换资源
apk包大小	较大，需要内置几套主题资源	较小，主题资源以其他apk的形式存在，不影响主apk	较小，编译时替换主题，只有一套主题资源

以上三个方案的对比:

内置主题：优点在于实现简单、配置方便，缺点是主题定制不灵活。

外部加载方案(apk)：优点在于扩展性很高，但由于该方案需要在代码中设置所有的可变资源，软件实现周期较长，写代码时容易出错。而且第一版耗时较长，一旦界面布局改变，需要较长的时间进行代码的编写。

基于 Sketch 的 UI 主题定制方案，它的优势在于：

- 基于 UED 的 UI 设计工具 Sketch，能在设计 UI 的时候，同时设计主题资源，并且能快捷的预览主题资源在整体界面上的显示效果。
- 可以通过 Sketch 插件的能力，快速将主题资源上传到后台，方便主题资源的统一管理及维护。
- 在后台配置修改完主题资源还能回流 Sketch 中进行查看。

这个方案的最大优势在于与 UED 的 UI 设计工具 Sketch 无缝的衔接在一起，极大的提高了主题的制作效率及成本，使得 UED 能够独立完成主题资源制作、管理、集成、应用、验证，形成一个主题小闭环。

6.小结

该方案的特点在于与 Sketch 很好的结合在一起，通过插件的能力，无缝的与云端进行资源相互同步，能力的实现主要通过 WEB 主题平台进行主题的管理、编辑，实现主题的快速复制，方便修改，然后再通过 Jenkins 的打包能力，实现主题的快速应用，最终实现 UI 主题定制。

能力建设完成后，以上的每个环节，只需要 UED 参与就能完成不同项目的主题定制，极大减少了 UED 及研发的成本，大大提高了效率，同时在定制能力上，从只能定制颜色、圆角、大小等，到能支持插画、iconfont 定制，以及其他各项拓展定制能力等。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

UI 自动化技术在高德的实践

作者：贯众

1.背景

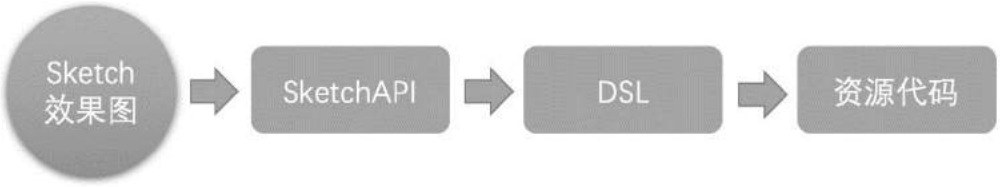
汽车导航作为 ToB 业务，需要满足不同汽车厂商在功能和风格上体现各自特色的需求。针对这种情况，传统的 UI 开发方式，基本上是一对一的特别定制。但是这种方式动辄就要 500~600 人日的工作量投入，成为业务发展的重要瓶颈。因此，能够对导航 UI 进行快速定制开发，成为汽车导航业务 UI 开发的必解课题。

高德地图技术团队希望打造一套快速精准的 UI 解决方案，通过自动化的方式生产 UI 代码，解放研发生产力的同时满足客户需求。

2.方案调研

为了避免重复造轮子，我们调研了行业上现有的 UI 自动化生成方案。主要分为两种：

Sketch 插件方案：

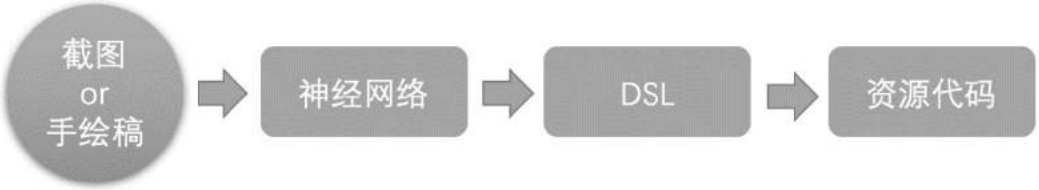


该方案是基于 Sketch 开发插件，利用 SketchAPI 读取出图层信息转换 DSL，主要代表作有 imgcook、Dapollo 等。

优势：从 SketchAPI 可以读取到非常详细的信息，足以生成高质量的界面代码。

劣势：要求效果图必须使用 Sketch 制作，并且对效果图会有一定的制图要求。

图片转代码方案：



该方案是通过经训练的深度神经网络，从截图或手稿直接生成 UI 代码，主要代表作有 pix2code、Sketch2Code 等。

优势：简单粗暴，通过截图或手绘就可以生成界面资源代码。

劣势：图层细节丢失，识别准确率欠佳、自适应不好。

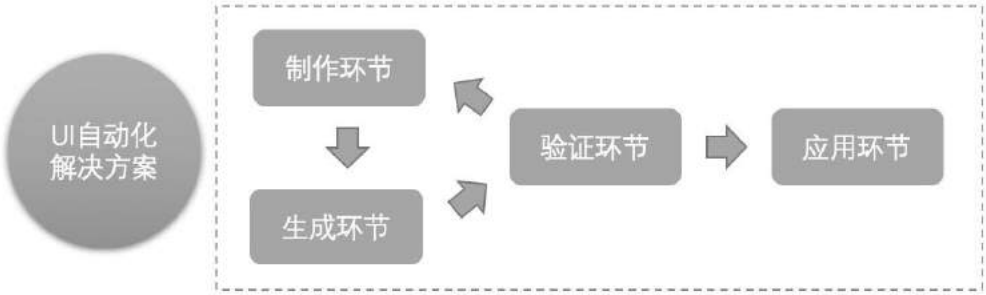
调研总结：

- Sketch 插件方案更加适合工程化使用，图片转代码方案更加适合用于快速生产原型。
- 目前行业上并没有能完全满足车载导航业务使用的 UI 自动化解决方案。

基于以上调研结果，我们决定基于 Sketch 插件方案，自研适合高德汽车业务需求的 UI 自动化方案。

3.技术方案与实践

结合 Sketch 插件方案的工作流以及高德内部的人员体制，我们将 UI 自动化解决方案在高德内部的使用过程拆分成 4 大环节，如下图所示：



制作环节

提供效果图编辑的能力：

- 制作带有主题信息的控件库供设计师拖拽使用。
- 对生成环节需要的信息进行输入（比如布局、控件、动画等）。

生成环节

提供资源的生成能力：

- 生成 xml、drawable、png、asvg 等资源，同时打包成可执行程序，用于验证环节。
- 对生成资源进行性能优化（如控件智能合并、png 图片无损压缩，asvg 生成）。

验证环节

提供多设备、多分辨率的还原度精准验证能力：

- 效果图质量检测能力，提前发现效果图像素偏差，不符合设计规范等问题，降低后期成本。
- 布局意图标注能力，通过简单的布局意图标注后，能实现把效果图拉伸成任意分辨率，解决设计师和开发人员沟通不流畅问题，降低沟通成本。
- 对比验证能力，通过坐标对比、截图和效果图差分对比等方式，实现还原度的像素级验证，保障资源质量。

应用环节

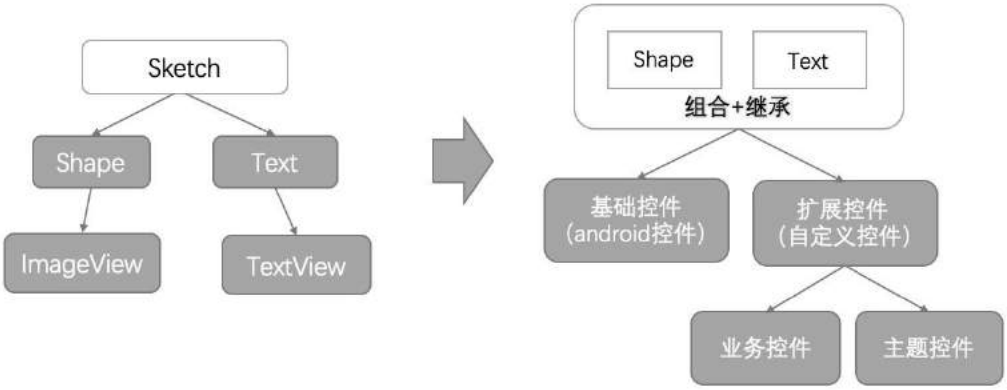
客户端工程资源管理能力：

- 通过一系列工具链，简化资源对接成本（如资源导入工具、重复资源清理工具等）。
- 开发 DHMI 主题定制平台，提供给设计师或客户，实现可视化的快速修改客户端主题，自主验证的能力。满足客户不断增长的主题定制需求，实现千人千面。

4.技术难点

控件体系

Sketch 的图层只有 text、shape 两种类型，分别可以对应上 Android 的 TextView 和 ImageView，但是只有这两种控件无法满足业务需求。参考 Android 控件体系中基础控件+自定义控件的方式，我们对这两种类型图层进行组合+继承，得到两种新的控件类型。



基础控件：Android 原生控件集合，如 TextView、EditText、ProgressBar 等，能满足界面搭建的基本需求。

扩展控件： Android 自定义控件，又分为以下两类：

业务控件：用于解决 Sketch 静态设计无法满足的部分，如需要 canvas paint 的控件。设计师只需画出静态部分，让开发人员自由发挥，能满足界面设计上动态元素、复杂元素的需求；同时也能形成控件库积累。

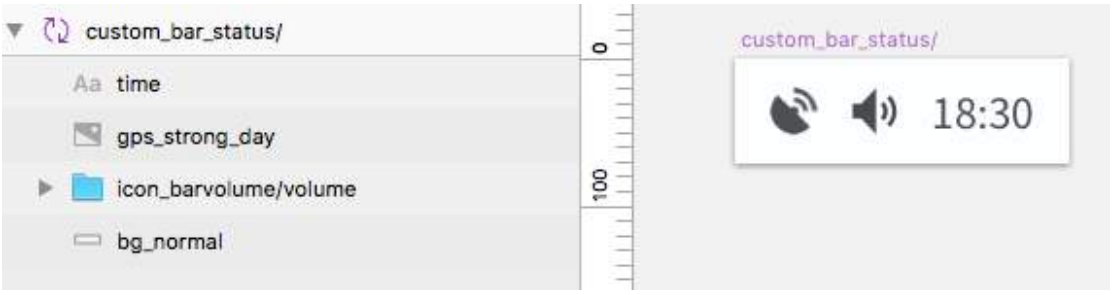
主题控件：大多数情况作为底色色块，实现 App 换肤的能力，支持在 DHMI 主题平台上做主题样式编辑。控件之间也支持互相组合，如多个主题控件和基础控件可以组合成一个新的业务控件。

通过基础控件和扩展控件的搭配使用，在实际生产中证实，这套控件体系可以无限扩展，做到全覆盖，满足任意界面的搭建需求。

示例

基础控件：

业务控件：



布局体系

布局的选择

布局采用的是约束布局 ConstraintLayout。

优势：

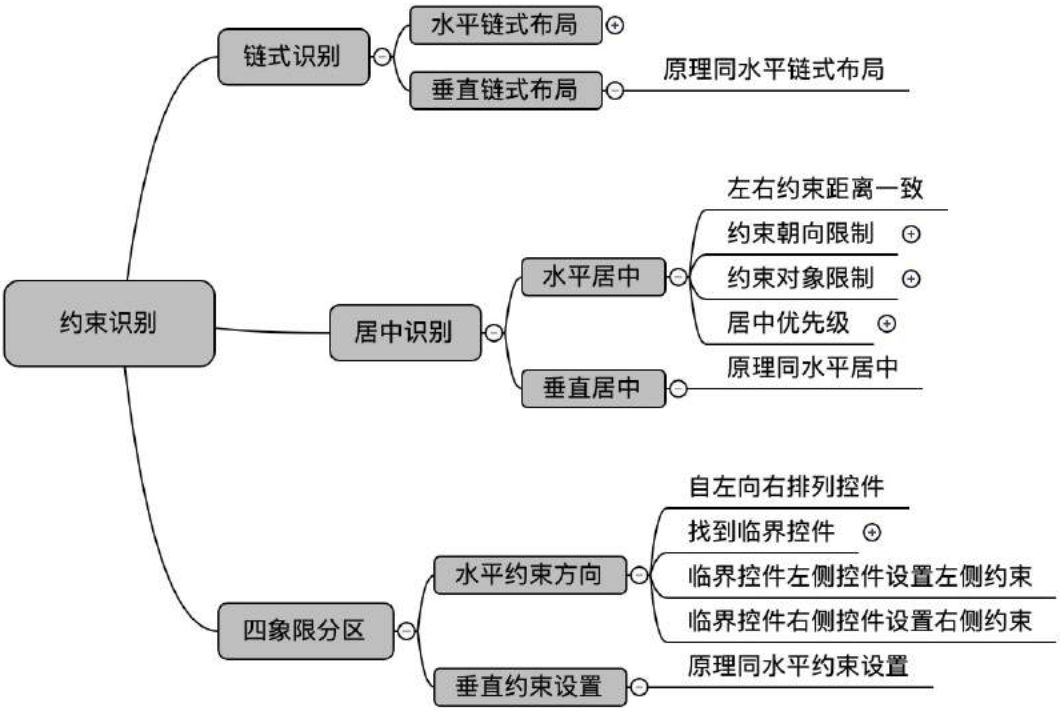
- 扁平化，理论上一个层级可以完成复杂界面的设计，相比传统布局，性能会有很大的提升。
- 简单易懂，贴近 UED 界面设计思路。
- 通过正向推导与反向逆推，证明该布局可以替代 Android 平台目前所有的布局，支持搭建任意界面。

布局识别算法

布局识别算法是在设计稿上基于位置关系推算布局约束关系的一种算法。

布局识别的难点：

- 主观性太强，容易出现误识别，没有绝对的规律。
- 交互动态性如何识别。
- 识别出的布局需要能支持多分辨率适配（横屏、竖屏、宽屏、方屏）。

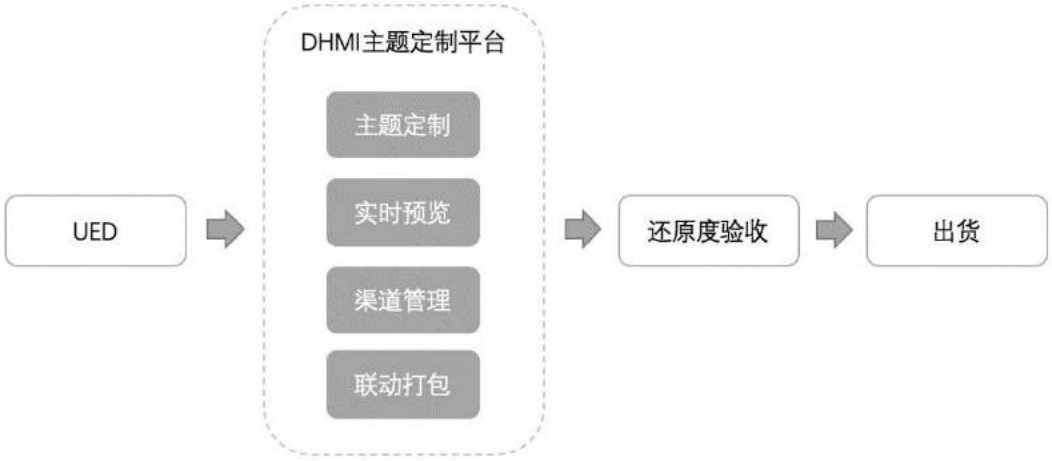


主题定制

由于车载导航面向的对象是车厂客户，不同的客户对于应用的 UI 或者主题是有不同需求的，也就是说针对不同客户，不同渠道的版本，需要有不同的应用主题。随着项目的增多，如果没有一个灵活，易管理，低成本的主题定制方案，那么这将是一个噩梦的开始。

传统的实现方案

DHMI 主题定制方案



具体实现如下

- 设计师在界面设计时，通过主题控件库拖拽的方式搭建界面。
- UI 自动化生成环节生成主题相关资源集成到客户端。
- DHMI 平台部署主要界面的关键场景。
- 设计师通过点哪改哪，实时预览的方式定制界面主题。
- 设计师自主出包还原度验收，全程无需研发参与。

5.小结

能力建设完成后，设计师和开发人员效率都得到极大提升。对于设计师，整体成本降低50%以上，有助于设计规范更好的落地，省去标注和切图环节，精准的还原度验收，快速的主题定制。对于开发人员，UI 开发成本降低 80%以上，不用再操心如何开发 UI，只需关注资源如何对接，完善的资源管理工具链，低成本的版本迭代，主题定制 0 成本。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

地图数据赋能 ADAS 的探索与实践

作者：李燕东

1.导读

作为汽车智能化的“眼睛”，传感器在其中扮演的角色举足轻重。本文将探讨，作为一种优质的“数字化传感器”，地图数据如何赋能汽车 ADAS 系统，如何在汽车智能化的变革中发挥作用。

2.背景

2.1 ADAS 及其发展

ADAS（AdvancedDriver Assistance System），即高级驾驶辅助系统，通过安装在汽车上的各种传感器，如单双目摄像头、毫米波雷达、激光雷达等，不断感知车身周围环境并收集数据，进行静、动态物体辨识、侦测与追踪，并结合地图数据，进行实时的系统的融合运算和分析，从而在车辆行驶中能够主动提醒驾驶员，或者某种程度上接管车辆的部分控制，让驾驶员预先察觉到可能发生的危险，有效增加汽车驾驶的舒适性和安全性。

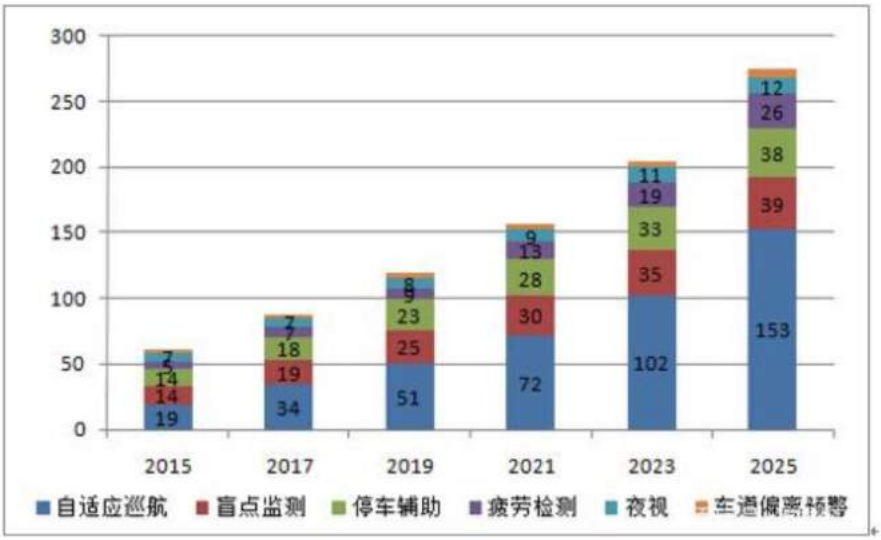
2.1.1 ADAS 发展

ADAS 通常由多个配套系统协作而成，常见的系统有：

- 前碰撞预警（FCW）
- 自动紧急制动（AEB）
- 车道偏移报警（LDW）
- 车道保持辅助（LKA）
- 自适应巡航（ACC）
- 盲点探测系统（BSD）
- 全景泊车系统（SVC）

根据公开资料估计，2025 年全球 ADAS 市场规模将达 275 亿欧元，2015~2025 年均复合增长率高达 17%。

全球ADAS市场规模预测（亿欧元）



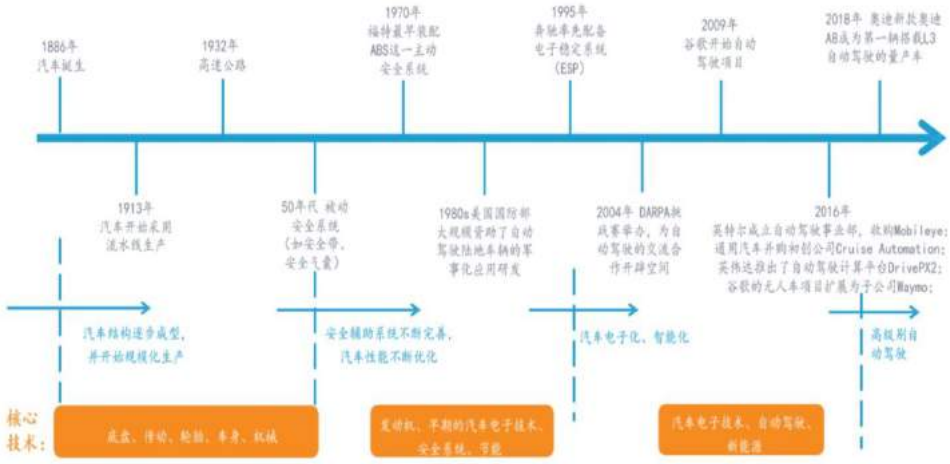
（数据来源：公开资料整理）

欧美日将标配 AEB 功能，ADAS 渗透率提升正在路上。2016 年 3 月，美国主流车厂承诺将于 2022 年 9 月前在所销售新车全部标配 AEB 功能。2019 年 2 月，欧盟地区和日本、澳大利亚共约 40 个国家达成协议，所有乘用车和轻型商用车(载客少于 9 人的货车及小型巴士)都必须配备 AEBS (AEB+ABS)，日本将于 2020 年起执行，欧盟将于 2022 年起执行。2016 年底出台的交通部文件规定，中国自 2018 年 4 月起大客车强制安装 FCW、自 2019 年 4 月起强制安装 AEBS，但实际执行状况较差。

2.1.2 ADAS 终局：自动驾驶（Autonomous Driving, AD）

ADAS 被视为汽车实现完全自动驾驶的前提，对于终局的自动驾驶，目前处在海外先行、国内紧跟的情况下。从国际来看，美国在 80 年代初开始自动驾驶军事化应用，而欧洲从 80 年代中期开始研发自动驾驶车辆，更多强调单车自动化、智能化的研究，日本的自动驾驶研发略晚于欧美，更多关注于采用智能安全系统降低事故发生率，以及采用车间通信方式辅助驾驶。

进入 21 世纪后，随着美国国防高级研究计划（DARPA）挑战赛的开启，提高了自动驾驶的社会关注度，从 2009 年起陆续开启产业热潮，谷歌首先布局自动驾驶，随后 Intel 成立自动驾驶事业部，收购 Mobileye，通用汽车并购 Cruise Automation，谷歌无人驾驶项目扩展为子公司 Waymo。



（资料来源：亿欧智库）

国内的自动驾驶起步依托于高校。2009 年起，国家自然科学基金委员会举办“中国智能车未来挑战赛”，为自动驾驶技术的交流和发展起到了良好的促进作用。国家战略的推进刺激了行业快速发展，自国务院在 2015 年发布《中国制造 2025》起，以自动驾驶技术为重点的智能网联汽车成为未来汽车发展的重要战略方向，大批初创企业投身自动驾驶领域。

（资料来源：亿欧智库）

2.2 传统传感器的使用

2.2.1 传统传感器的类型

ADAS 的核心技术是环境感知，作为 ADAS 工作的数据来源，传感器是 ADAS 系统的核心部件。传统的 ADAS 传感器主要包括毫米波雷达、激光雷达、摄像头、超声波雷达等。

传统传感器的主要特征如下：

	摄像头	毫米波雷达	激光雷达	超声波雷达
距离	短（50m）	短/中/长（250m）	长（> 100m）	很短（5m）
精度	一般	较高	极高	高
成本	中	高	极高	低
功能	利用计算机视觉判别周边环境与物体、判断前车距离	感知大范围内车辆的运行情况，多用于自适应巡航系统	动静态障碍检测识别与跟踪、路面检测、定位和导航、环境建模	探测低速环境，如自动泊车系统
优势	成本低、硬件技术成熟、可识别物体属性	全天候全天时工作、探测距离远、性能稳定、分辨率较高、测速精确	测量精度极高、分辨率高、抗干扰能力强、测距范围大，响应速度快	成本低，近距离探测精度高，且不受光线条件的影响
劣势	依赖光线、易受恶劣天气影响、难以精确测距	部分场景下易受信号干扰、无法识别物体属性、探测角度小	易受恶劣天气影响、成本高昂、制造工艺复杂	只适用于近距离、低速场景，易受信号干扰
类别	包括单目、双目摄像头，按照芯片类型又可分为 CCD 摄像头、CMOS 摄像头	依据测距原理不同可分为脉冲测距雷达、连续波测距雷达	可分为机械激光雷达、固态激光雷达，根据探测原理也能够区分为单线激光雷达和多线激光雷达	

2.2.2 传统传感器的局限

- 感知范围和识别距离限制：

传统传感器都存在识别距离、高度、范围方面的限制，最长的识别范围不过数百米。

- 受天气、环境等因素影响：

识别传感器容易受雨、雪、雾、霾等异常天气影响而失效，然而越发在这种天气，对驾驶安全的要求就越高，相应的 ADAS 和 AD 系统发挥的作用也会越发重要，如果此时传感器失效，对驾驶安全和体验带来的影响也越大。

同时，车身传感器还容易受到树木、其他车辆/行人、广告牌、建筑物等遮挡物和光照、角度等不可控因素的影响，导致误识别或者无法识别。

- 识别准确率，情景和上下文识别难度：

受限于国内道路的复杂状况和交通设施的安装位置等因素，即使能够准确识别到目标，也可能错误的关联到所在道路上，比如一个实际作用在主路上的限速牌可能安装在与匝道相邻的岔路口上，进而可能被误识别为匝道的限速，给后续决策控制系统带来错误的信息，更严重的是，系统可能根本不知道出错了。

- 成本：

识别效果好的毫米波雷达和激光雷达，成本高昂，毫米波雷达单体价格在千元级别，而激光雷达则更高，单体价格在数万到数十万人民币。成本适中且技术更加成熟的摄像头，则更容易受到上述局限的影响。

2.3 地图数据的价值

基于以上传统传感器的局限，地图数据可作为感知源的有效补充。

- 超“视”距：

不受探测距离限制，应用方可根据需要获取数米、数公里、甚至上千公里的地图数据。同时，地图数据的获取还不受环境、光照、天气、遮挡物等不可控因素的影响，真正的全路况全天候全天时。

- 包括动态信息在内的丰富的地图要素和道路拓扑信息：

地图数据中，包括丰富的道路相关数据，比如道路等级、车道信息及车道线、交通标志、交通设施、交叉口、道路形状点、坡度、曲率、连接关系等静态道路数据，同时还包括比如天气、拥堵或畅通等交通信息、事故或封路等动态事件信息。这些数据的使用可以大幅减小感知融合的处理复杂度，同时可以有效的补充道路复杂场景或者上下文复杂情景识别等传统传感器识别效果不满足要求或无法胜任等情况。

- 高精度定位：

借助于高精度带有地理位置的道路车道线形状点、交通标志、交通设施等静态数据，结合 GNSS、车身传感器甚至视觉信号等其他辅助定位手段，能够达到厘米级别的定位精度，满足 ADAS 或 AD 功能对定位的要求。

- 局部路线引导：

通过地图数据拓扑结构，能够描述车辆最有可能行走的路径（Most Probable Path, MPP），该 MPP 带有道路级甚至车道级行驶引导信息，可以提供给车辆决策控制系统，用于引导驾驶者操控车辆（ADAS 功能），或者决策系统接管车辆动力、转向、制动控制进行半自动或全自动的引导、变道等操作（AD 功能）。

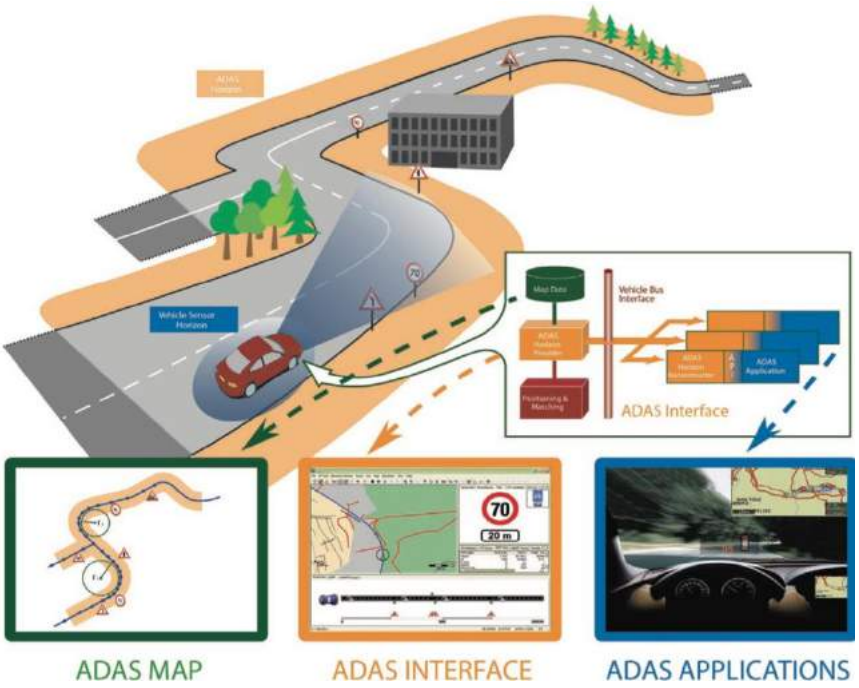
- 全局路线规划：

同样借助于全局 MPP，车辆可以时刻了解全局道路的静态和动态属性信息，一方面可以根据这些信息合理的调整车辆动力系统或电源管理模块，减少燃油消耗或节省电量，达到最佳的经济性；另一方面，可以提前识别车辆行驶路径上的事故或拥堵等实时动态信息，适时变更行驶路线或者提前预警，提高驾驶的安全性和舒适性。

- 安全冗余：

尤其对于面向 AD 场景的功能，安全是首要考虑的前提，地图数据作为数字化的感知源，可以跟其他传感器一起，为感知融合算法提供多重信息输入，联合校验，在减少融合算法复杂度的同时，进一步增加感知的准确性和安全性。

2.4 地图数据的表达



（图片来源：ADASIS 官方资料）

- ADASIS AISBL：

为了利用主要用于导航的地图数据来提高主动安全和 ADAS 应用的性能，弥补 ADAS 传感器的局限和不足，2002 年由 Navtech 联合欧洲汽车行业的汽车制造商、ADAS 供应商和图商成立 ADASIS 论坛（ADASIS Forum），随后该论坛被划入欧洲智能交通协会（ETRICO）。该组织的主要目标是开发、测试和验证在 ADAS 应用程序和地图数据源之间交换地图数据的行业标准接口（ADASIS 标准）。

因为降低了开发成本和个体的风险，符合行业的共同利益，这种标准化协议接口规范的概念受到了业内汽车制造商和各供应商的欢迎。成立至今共推出 3 个版本的 ADASIS 协议标准，标准详情将在下文再行展开。

2018 年 5 月，ETRICO 将 ADAS 论坛重新定义为一个官方的非营利性国际组织，并命名为 ADASIS AISBL，总部位于比利时首都布鲁塞尔。

ADAIS AISBL 从创立开始即采用会员制，按照 ADASIS 最初的归属，会员分别归类到汽车制造商、ADAS 系统制造商、导航系统供应商和地图和数据供应商四个角色中。截至 2019 年 6 月，协会共有 57 家会员，具体参考下表。

Vehicle manufacturers (13)	ADAS manufacturers (16)	Navigation system manufacturers (16)	Map & data providers (12)
BMW	Aptiv (former Delphi)	AISIN AW	AutoNavi (Alibaba Group)
Daimler	Continental Automotive	ALPINE ELECTRONICS	Baidu
Ford Forschungszentrum Aachen	CTAG	Banma Network Technology	eMapgo
Honda	DENSO	Bosch SoftTec	EnGis
Hyundai Mnsoft	Denso Ten (Europe)	CarLink Software Co.	HERE
Jaguar Land Rover	Hitachi Automotive Systems	Elektrobit Automotive	Kuandeng
Nissan Motor Co.	Huawei	Garmin	MOMENTA
Opel Automobile	Huizhou Desay SV Automotive	Harman/Becker Automotive	NavInfo
Renault	Ibeo Automotive Systems	Mappers Co.	Tencent
Toyota Motor Europe	Knorr-Bremse	Mitsubishi Electric Automotive	TomTom
Volkswagen	LG Electronics	MXNavi	Wuhan Kotei
Volvo Car	MAGNA	Neusoft	Zenrin
Volvo Group Trucks Technology	Valeo Comfort and Driving Assistance	NNG	
	Visteon	Panasonic Automotive	
	Zenuity	TeleNav	
	ZF	Veoneer (Autoliv)	

（数据来源：ADASIS 官方资料）

3.技术名词解释

- ADAS：

Advanced DriverAssistance System

即高级驾驶辅助系统，利用车载传感器感知车辆环境，并融合计算，预先让驾驶者察觉可能发生的危险，有效提升车辆驾驶的安全性、经济性和舒适性。

- ADASIS：

Advanced DriverAssistance System Interface Specification

ADAS 论坛制定的行业国际标准，用于规范地图数据和车辆 ADAS 应用之间交换地图数据的标准接口协议，目前有 v1、v2、v3 三个版本。

- EHP/AHP：

Electronic HorizonProvider / ADAS Horizon Provider

即电子地平线，为 ADAS 应用提供超视距的前方道路和数据信息。

- EHR/AHR

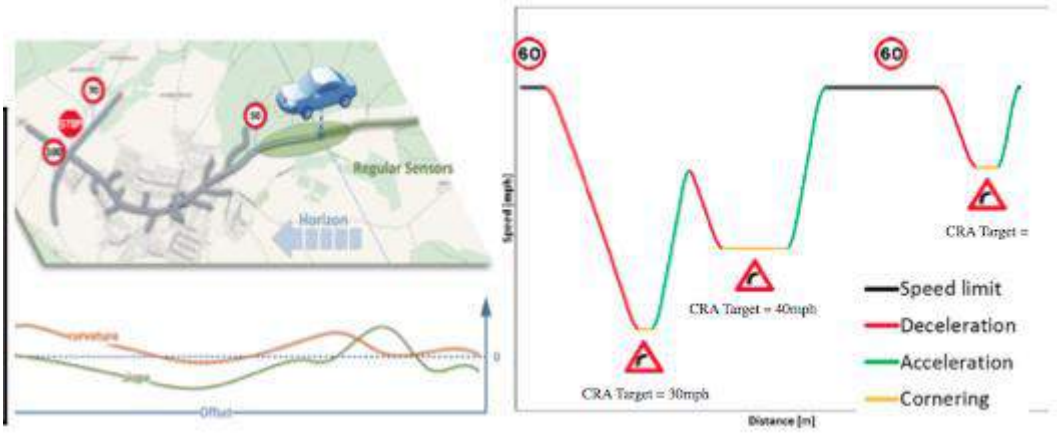
Electronic HorizonReconstructor / ADAS Horizon Reconstructor

用于解析 EHP 发出的消息并重建地图数据，供终端 ADAS 应用模块使用。

4.ADAS 应用场景

应用一：智能限速提醒

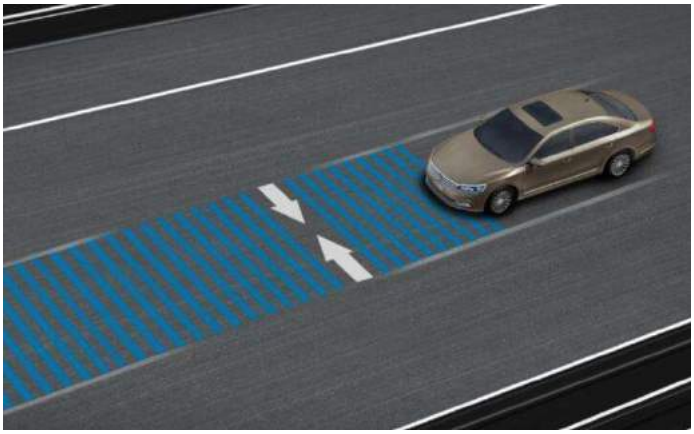
应用二：自适应巡航



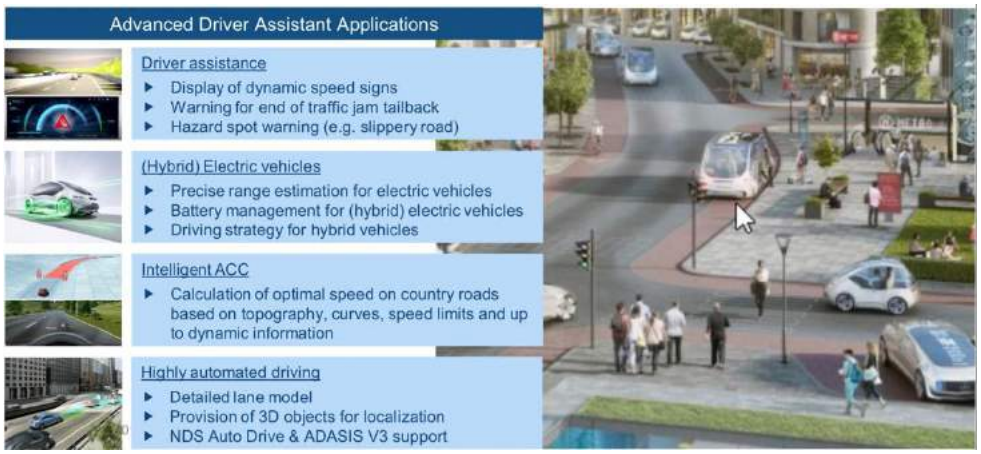
应用三：绿色出行

道路铺设、坡度、曲率、前方路况等因素，调整车辆动力系统或电源管理模块，达到节能经济、绿色出行的目的。

应用四：车道保持

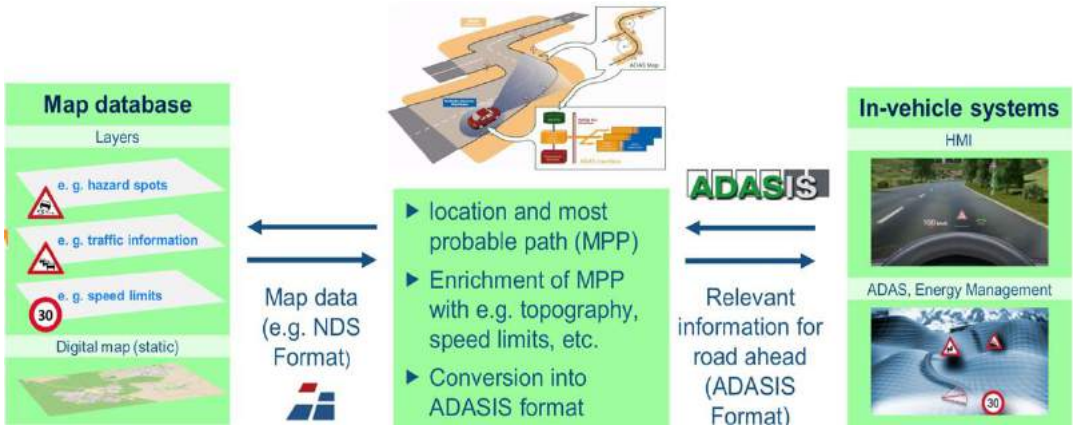


应用五：自动驾驶

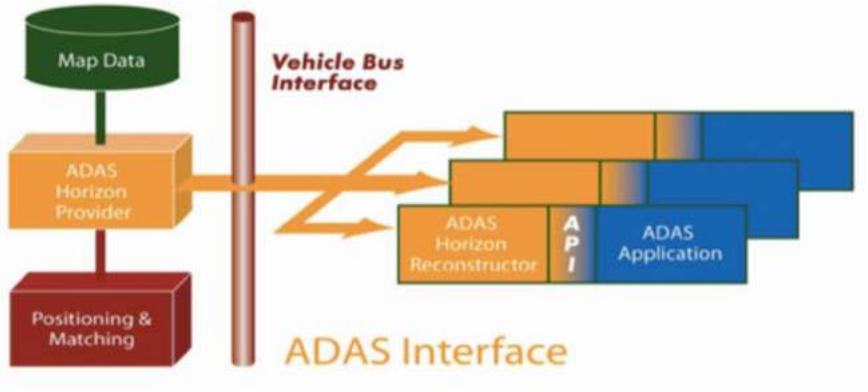


（图片来源：ADASIS 官方资料）

5.ADASIS 协议的前世今生



5.1 原理



（ADASIS 系统架构 图片来源：ADASIS 官方资料）

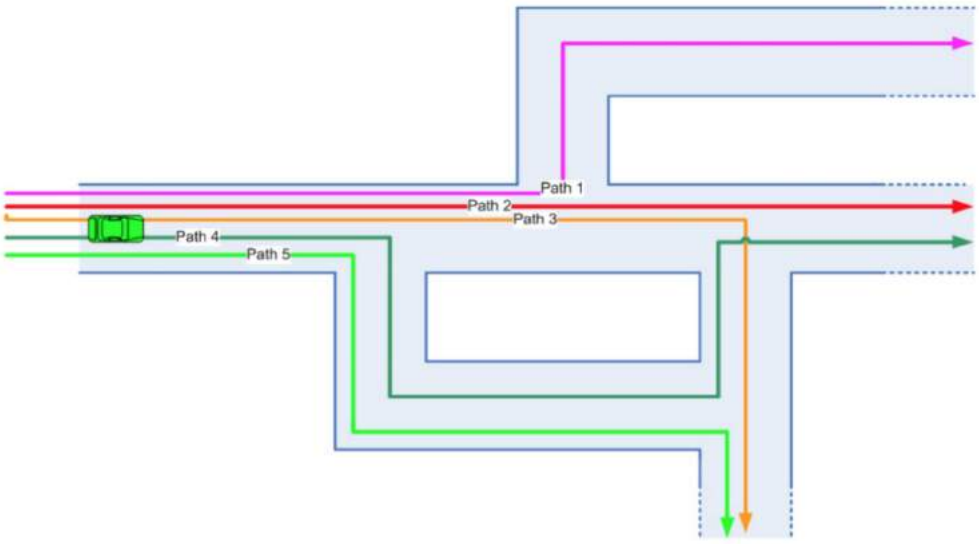
如前文所述，ADASIS 论坛成立主要目的是制定统一的地图数据交换接口，在制定具体的标准接口前，ADASIS 论坛首先提出了上述的系统架构。从架构图可以看到，地图数据并不是直接传递给 ADAS 系统的，而是需要进行一个分解并重组的过程。

首先，要有一个数据提取单元。数据提取单元会提取详细的地形与道路相关数据，以及车辆的位置信息，将这些信息生成 ADASHorizon 所需要的数据，然后通过车辆的相关总线进行传输。

其次，要有一个接收数据的重构单元。重构单元接收到传输数据之后，把它变成不同的 ADAS 系统能够看得懂的数据，然后再根据需求把不同的数据传递到不同的 ADAS 应用中。有了这个过程后，ADASIS 论坛的工作就集中在了数据如何提取、如何传输、如何重构并分发了。

5.2 ADASIS v1

2004 年初，ADASIS 论坛的成员开始了一个名叫 PReVENTMAPS&ADAS 的项目，项目的目标就是明确相关标准的细节并开发出相应的测试方法。ADASIS 论坛借着这个项目验证了 ADASIS 的概念以及前述系统架构和拆分过程是可行的，并在项目结束之后，发布了首款标准协议 ADASISv1。

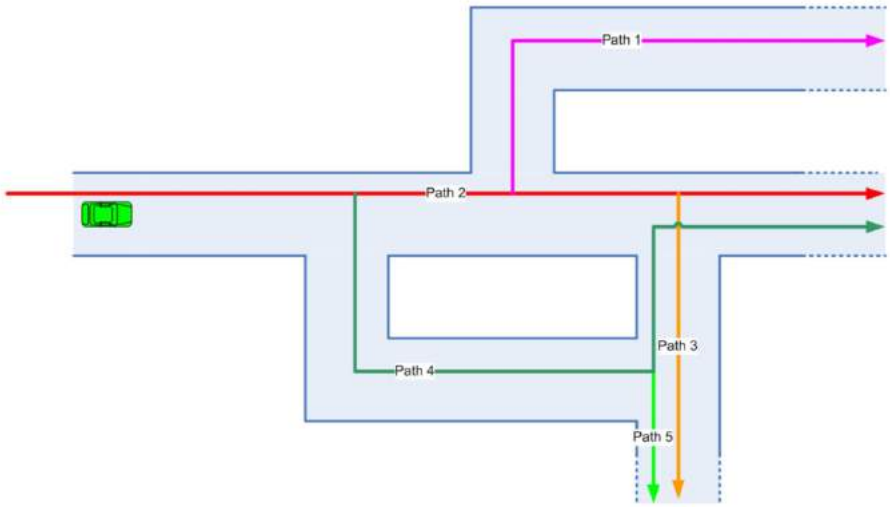


在 v1 中，系统会根据车辆当前位置以及最终目的地，提取所有可能的路径规划方案，将其中最为优先的路径称定义为最有可能路径（MPP），其他方案为备选方案。所有的方案都会被提取并进行重构，并最终给出选择哪条路径以及如何驾驶通过的建议，传递到不同的 ADAS 系统中。

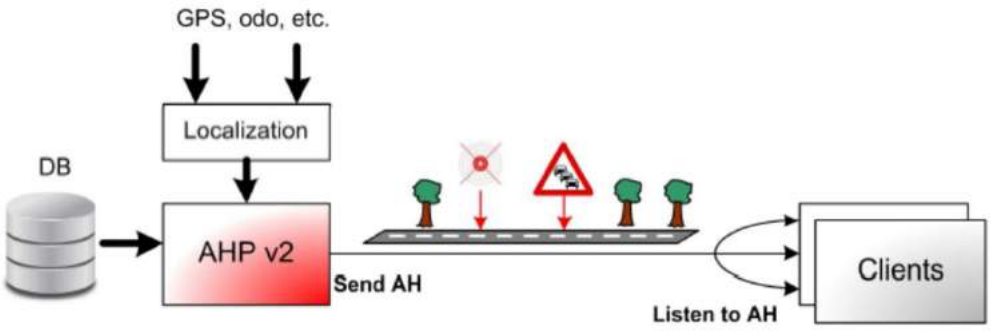
由于上述过程过于复杂，且传输的数据量过多，给系统总线和提取与重构单元带来大量的时间与成本，因此并没有在任何一家汽车 OEM 厂商的量产车型上使用，相反，越来越多的企业各自推出了自己的解决方案（并没有使用统一标准），严重影响了 ADASIS 标准的推行。

5.3 ADASIS v2

因为 v1 版本的推行失败，ADASIS 论坛开始准备第二版标准，着重在降低系统占用的总线资源，以及使用最小原则提取并重构数据。



经过优化后的 ADASISv2 版本于 2010 年内部发布，其精简的结构和良好的扩展性也得到会员们的支持，并于 2012 年正式对外发布。此后在欧洲，乃至全球范围内得到广泛的使用。



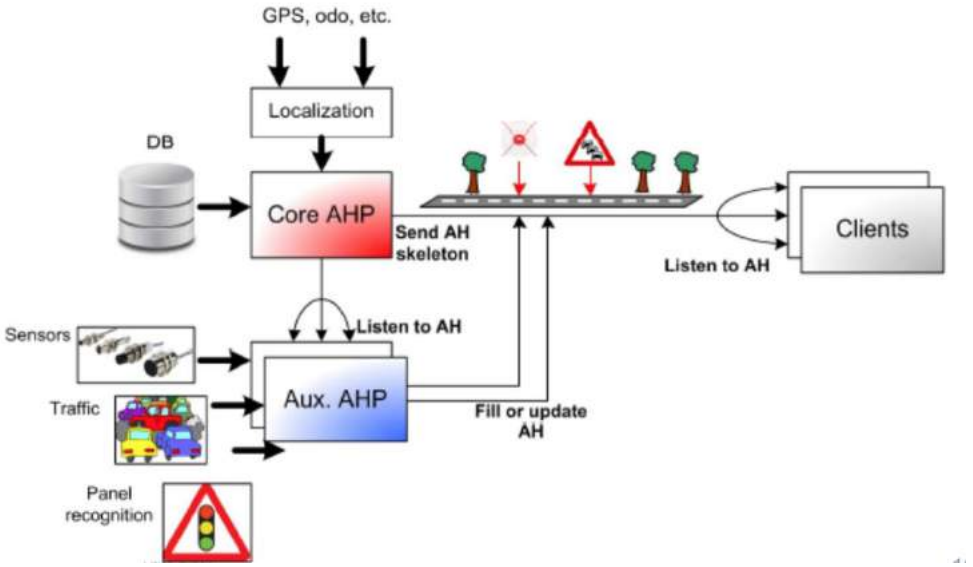
（图片来源：ADASIS 官方资料）

5.4 ADASIS v3

随着地图数据的日趋丰富，地图数据也开始进入高精度地图时代，同时功能更多的 ADAS 系统或者自动驾驶 AD 系统的出现，也对传输数量和传输带宽提出了更高的要求，这时 ADASIS v2 开始显现出了不足。

为解决这个问题，ADASIS 论坛自 2015 年 6 月开始研究基于车身以太网传输的下一代协议即 ADASIS v3，并于 2018 年 7 月内部发布了 ADASIS v3 RC 版。

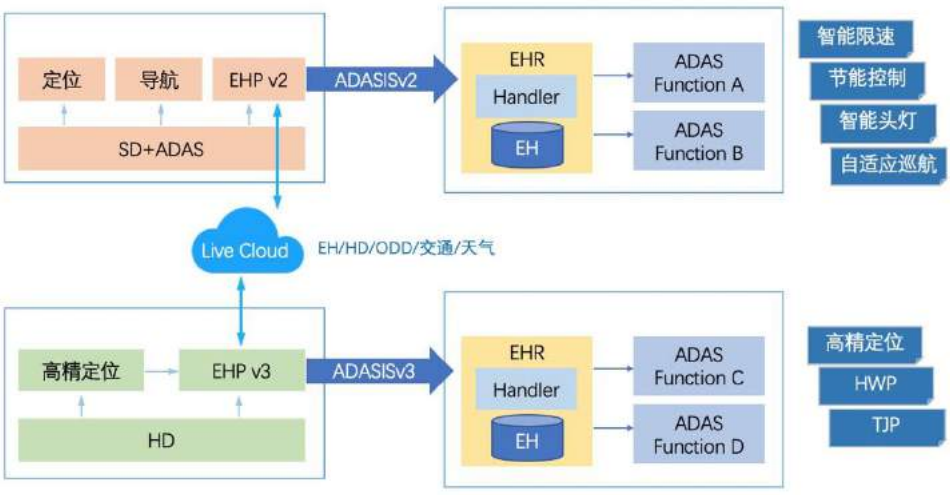
ADASIS v3 版本支持包括车道数据在内的高精度地图数据，并且添加了多条 Horizon 数据的支持，根据 ADASIS AISBL 公布的信息，正式版本的 v3 协议预计在 2019 年 11 月对外发布。



（图片来源：ADASIS 官方资料）

6.高德商业项目应用实践

结合自身的积累和优势，高德面向商业量产项目推出了基于导航 SD 数据的 EHPv2 和基于 HD 数据的 EHPv3+高精定位的完整解决方案。该方案采用灵活的组合方式，具备 SD、HD 数据的全量数据要素透出能力，支持 Android、Linux、QNX 等多种平台，同时支持 EHP 云+端和数据 OTA 能力，目前已在多个商业项目中落地。



车联技术在高德的演进和实践

作者：歌白

导读

伴随汽车行业的发展和汽车智能化的演变，车联技术应运而生。本文从主流车联技术和协议的诞生开始，讲述高德在车联技术演进与赋能中的探索与实践，自有硬件产品迭代升级的过程，以便更好的总结过去，展望未来。

将手机与车载系统连接，在车和手机之间交换数据，这正是车联技术当今最成熟的应用场景。除了拥有国民级地图和导航产品，高德在车载领域同样有持续的积累，在人、车、路协同要求越来越高的今天，高德也在致力于车联网的探索，利用高德的数据和车载应用经验，推动和赋能车载车联技术的演进和变革。

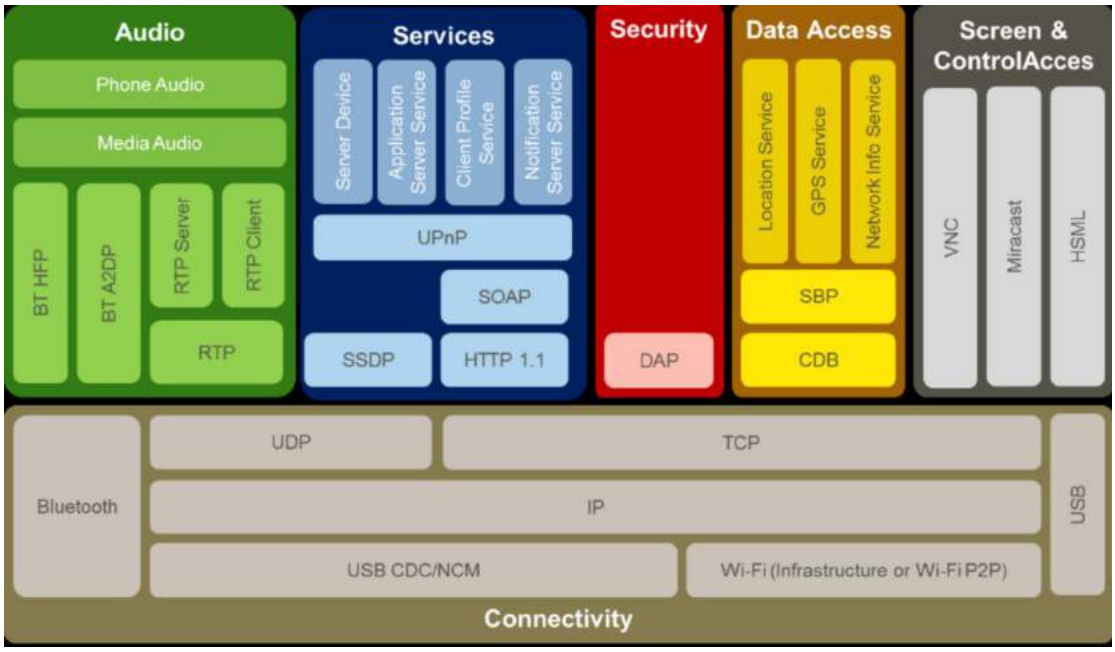
探索

探索的开始是为 1.0，那是 MirrorLink 第一次在高德落地的时代。

MirrorLink 是一个智能手机与车载信息系统交互集成的标准。通过 MirrorLink，某些应用装载并运行在智能手机之上，汽车的司乘人员便可以通过车载娱乐系统在方向盘、中控面板上的按钮以及触控屏幕与它们交互。

MirrorLink 源于 2009 年诺基亚和玛涅蒂马瑞利联合演示的一个叫做 Terminal Mode 的概念模型。尽管在那之前，汽车行业也有其他形形色色的车联协议，但随着当时如日中天的诺基亚与汽车消费电子组织合作，成立了汽车连接联合体（Car Connectivity Consortium，CCC 组织），并将众多汽车行业和移动行业的玩家拉来为 MirrorLink 站台，MirrorLink 协议便隐隐然成为行业中不可忽视的存在：全球超过 90 款车型支持手机用 MirrorLink 与其互联。

1.0 之前，高德并没有多少 MirrorLink 相关产品经验，机缘巧合，在某个 T-Box 项目中，我们有机会切入这一领域。MirrorLink 实际上并不是一个单一协议，而是由众多协议组成的协议族。其复杂程度，从其架构上就可见一斑。



面对从没有做过的协议栈，团队最初的一批同学没有退缩，迎难而上，奋勇争先。在技术同学尚未到位的情况下，一位产品同学独撑大局，打通 T-Box 到车盒的 USB 连接，实现 MirrorLink 协议的核心视频触控交互，完成首版演示，成为团队后来者膜拜的“大神”。1.0 时代的产品也由此渐入正轨。而回顾当时，高德在车联协议这条道路上的探索不过刚刚开始。

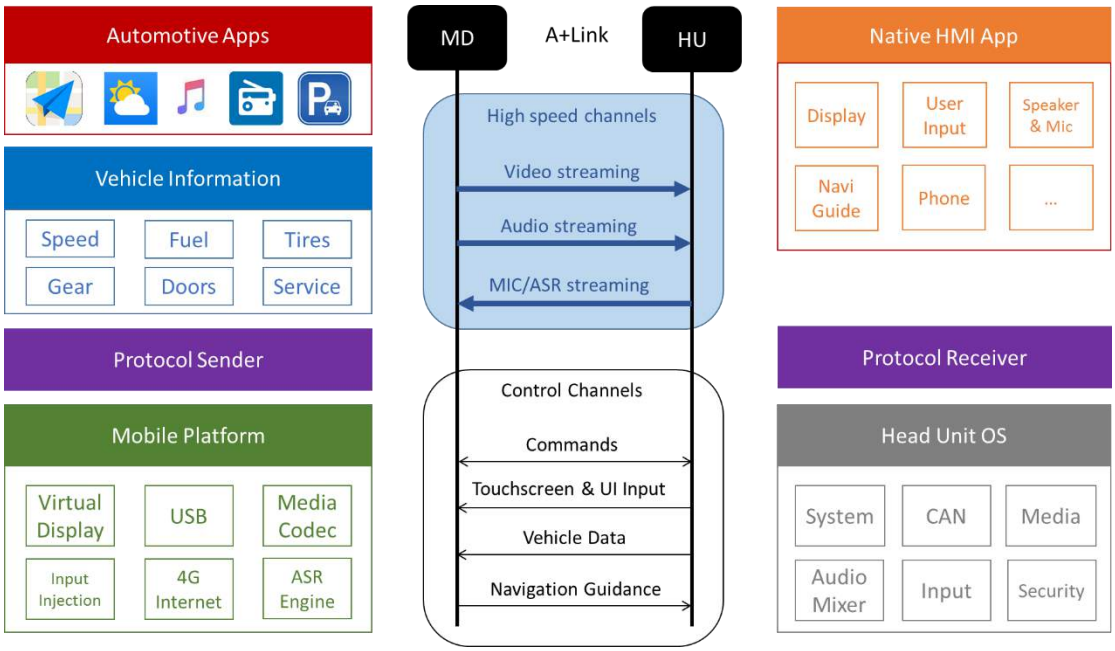
MirrorLink 虽然是主流车联协议里发展最早的，但它的征途并不顺利。诺基亚曾是

MirrorLink 最主要的技术参与方，但随着它在 IT 浪潮中的浮沉，MirrorLink 的手机侧技术方案已经不见诺基亚的身影，相对的，目前使用最广的手机系统 Android 成为了 MirrorLink 连接场景下的主要参照环境。然而，Android 最重要的玩家谷歌，非但不是 MirrorLink 联盟 CCC 组织的成员，而且还一度力推与 MirrorLink 相似的车联协议 AndroidAuto。同时，移动行业的另一个大玩家苹果公司也没有加入 CCC 组织，并且也有自己的车联协议 Carplay。没有了技术发起方诺基亚，又缺少移动操作系统方支持，还要与谷歌苹果去竞争车联协议，MirrorLink 发展的困难可想而知。

沉淀

尽管 MirrorLink 的发展举步维艰，但连接汽车与手机的技术实践却从没被冷落过。MirrorLink 在 2009 年面世之后，苹果公司在 2011 年推出了 iPhone 手机专用的车联协议 Carplay；之后，谷歌在 2014 年也推出绑定自家应用的 AndroidAuto；而在国内，百度在 2015 年也推出了超级应用形态的 Carlife。

在高德这边，经过 1.0 在 MirrorLink 协议上的实战锤炼，我们在分析了主要的车联协议之后，开启了 2.0 的阶段，推出了高德自有知识产权的 ALink 协议。ALink 优化了车联协议的架构，能快速建立连接，采用 H.264 压缩视频，支持 QoS，大幅提升传输能力，降低带宽消耗，显著提升连接体验。同时，ALink 不限于手机和车机之间的连接，它支持一般的具有 USB/WiFi 连接功能的移动设备连接汽车。在此场景下，移动设备与车机之间的连接，被划分为多个逻辑信道：



- 行令信道

负责 ALink 连接的建立和维护。加密的信令信道保证了汽车和外部移动设备连接时的私密和安全。为数不多的信令，优化了协议的连接过程，保证了协议状态切换时的及时响应。

- 高速视频信道

负责 H.264 编码的视频传输。结合底层驱动的流控，压缩过的视频传输能力可轻松应对

60FPS 的帧率。

- 合成音频信道

负责合成音频的传输。可根据配置提供合成过的 MD 声音，方便适配只接受一路声音，缺乏多路音源合成能力的车机。

- 媒体音信道

负责媒体类应用的声音传输。可根据配置提供 MD 上音乐、电台、电子书等应用产生的声音，供有多路音源合成能力的车机做声音合成。

- 导航播报信道

负责导航播报音的传输。可根据配置提供导航播报音，供有多路音源合成能力的车机做声音合成。

- 语音输出信道

负责语音的输出。可根据配置提供语音助手类应用的输出声音，供有多路音源合成能力的车机做声音合成。

- 语音输入信道

负责语音的输入。可根据配置传输车机侧采集的语音，供语音识别模块做语义识别。

演进

2.0 阶段，随着 ALink 协议在同国际著名车厂的合作中落地，我们不仅有了自主知识产权的车联协议，而且拓展了实现车联协议较为关键的底层驱动能力，保障了协议实现的性能和质量。

在推出 ALink 协议之后，我们 3.0 的目标，着眼点就不仅仅是车联协议，而是放大到了软硬一体的车联产品。基于这些积累，我们进一步把视野放大，在车联协议上同业内伙伴增

强合作的同时，将着眼点放在了软硬一体的车联产品。在 3.0 阶段，我们在集团内跨 BU 与 AliOS、天猫精灵团队合作，从底层做起，定义了崭新的车联产品：天猫精灵高德版智能车盒。

3.0 的车盒，采用 AliOS 移动操作系统，支持多种车联协议，并通过车联协议，把天猫精灵语音助手、AR 导航等先进应用带入了车内环境。同时，在 3.0 阶段，我们团队的技术能力进一步成长，除了在已掌握的技术上进一步深耕之外，还涉足了新领域，学到了新技能。其中尤为重要的是，我们建成了软硬一体的技术团队，演进出从结构到电路，从内核驱动到上层应用，从端到云的全栈能力。

展望

车联技术发展了十年，是车内场景出车的实践，也是车外场景进车的实践。在传统的车联技术应用上，拉通车内外的主要媒介是手机。然而手机并不是唯一的选择，随着广域无线通信技术的日益普及，车盒形态的智能硬件提供了拉通车内外场景的又一个选择。

展望未来，汽车行业正经历技术革新，5G、IoT、AI 等技术在汽车应用领域也是方兴未艾。以移动智能设备赋能汽车，融合汽车行业与移动行业的机遇就在眼前。我们的车联技术和产品，正可以先行一步，为新型汽车生态的应用寻找入口，担当新汽车战略落地的先锋。

架构篇

高德亿级流量接入层服务的演化之路

作者：孙蔚

2019 杭州云栖大会上，高德地图技术团队向与会者分享了包括视觉与机器智能、路线规划、场景化/精细化定位时空数据应用、亿级流量架构演进等多个出行技术领域的热门话题。现场火爆，听众反响强烈。

阿里巴巴资深技术专家孙蔚在高德技术专场做了题为**《高德亿级流量接入层服务的演化之路》**的演讲，主要分享了接入层服务在高德业务飞速发展过程中，为应对系统和业务的各方面挑战所做的相关系统架构设计，以及系统在赋能业务方面的思考和未来规划。

以下为孙蔚演讲内容的简版实录：

高德地图的 DAU（日活）已经过亿，服务量级是数百亿级。高德的应用场景，比如实时公交、实时路况、导航、司乘位置的同时展示等，对延迟非常敏感。如何做到高可用、高性能的架构设计，高德在实践中总结了一套解决方案。

今天主要分享三个方面的内容：

- 接入层定位思考与挑战
- 高可用、高性能的架构设计
- 高德服务端的思考及规划

1.接入层定位思考与挑战

首先介绍下 Gateway，从架构上看，Gateway 在中间位置，上层是应用端，下层是引擎，例如驾车引擎、步导引擎等等。目前已接入 80+应用，500 多个 API 透出，QPS 峰值 60W+。

从 Gateway 的定位来思考，**作为网关，最重要的就是稳定，同时能提效和赋能。**一句话概括：如何在资源最少的情况下，在保证稳定的前提下，以最快速度帮助业务的达成，这就是服务端的定位。

高德的网关设计挑战在于每天数百亿级的流量过来，场景对延迟又特别敏感。举个例子，很多开发者和应用都在使用高德定位服务，定位服务架构挑战 5 毫秒内需返回。

为了解决这些问题，高德做过一次比较大的系统架构升级，主要做了几方面的工作。首先是**流式、全异步化改造**。机器数量减少一半，性能提升一倍，通过这个架构升级达到了。

其次是**加强基础支撑能力建设**，为配合引擎提效，做了接口聚合、数据编排和流量打标与

分流。
此外，为了提供服务稳定性，同时提升单元性能，做了**高德单元化网关解决方案**。最主要是方便其他业务快速实现单元化。

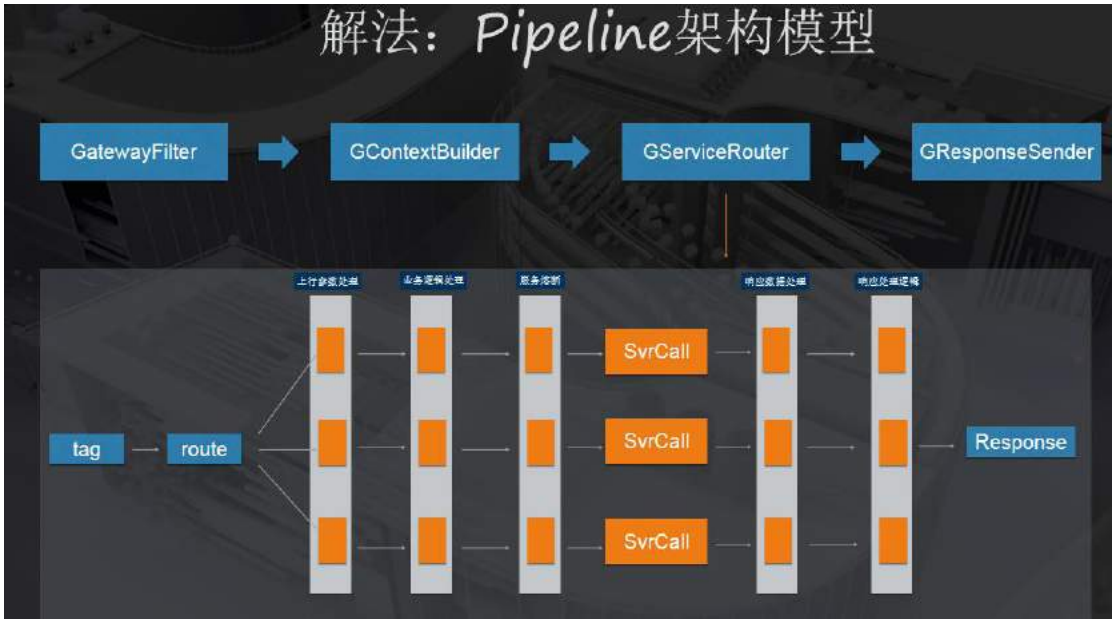
2.高可用、高性能的架构设计

重构前比较严重的问题是服务性能低，BC 服务器综合性能在 1200QPS。稳定性风险比较高，特别是网络抖动，如何保证整个系统的稳定性，这可能是最大的挑战。所以，对于**整个架构的思考，最主要的事情是做异步化**。

高德接入层网关演进过程主要经历了 3 个阶段：

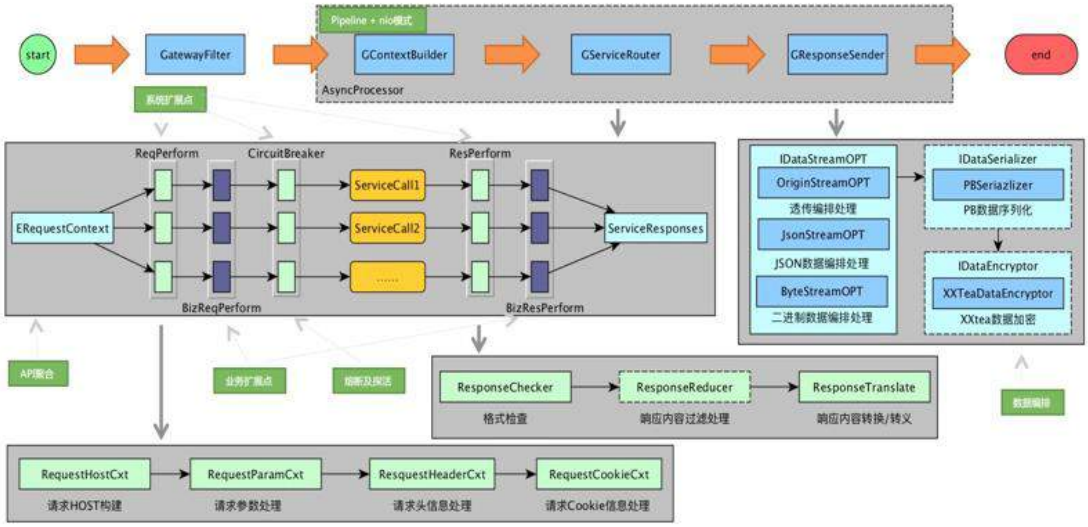
- **异步+Pipeline 架构改造**

1）流式、全异步化架构



如上图 Pipeline 的架构模型，我们在 2016 年开始做，那时候还没有很流行，我们自己实现

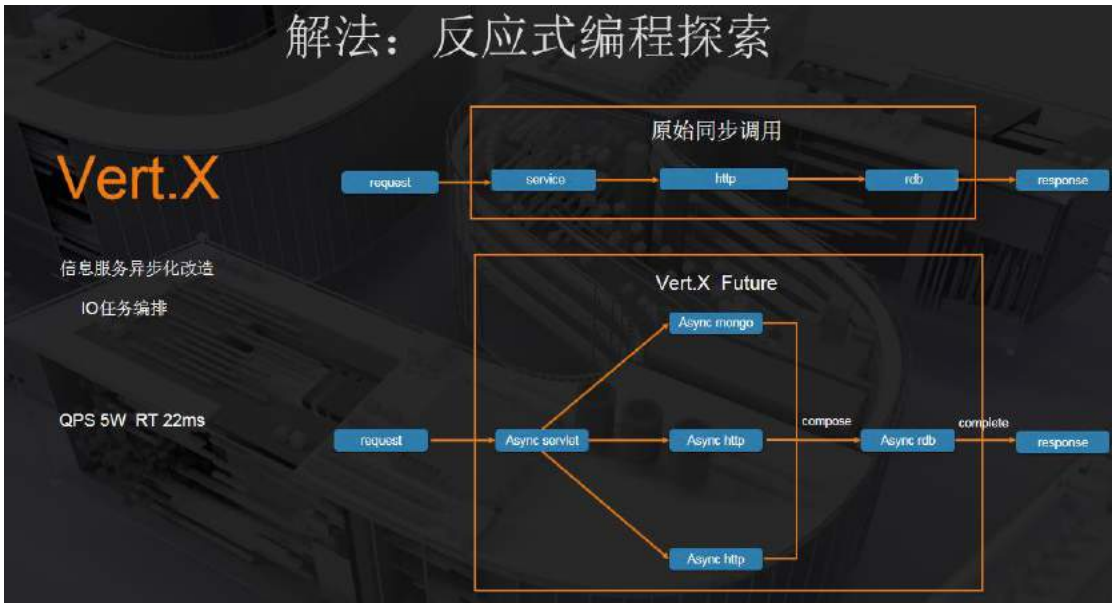
了异步认知，再加入 Pipeline 架构模型。
采用流式、全异步化的架构模型，使用 Tomcat nio+Async Servlet + AsyncHttpClient。
Gateway QPS 峰值 60W，服务 rt 控制在 1ms 左右。



整体服务是 Pipeline 架构，在服务的上行和下行关键节点进行了扩展点设计，利用该扩展点设计，解决了接口的历史包袱问题。使用到的相关工具类库也要注意异步性能问题，在全链路异步化的时候，最核心的是相关的工具，也必须解决异步化的问题。要不然就是内部有阻塞，基本上会带来整个链路的阻塞。

收益：单机性能提升了 400%，服务延迟低于 2 毫秒，现在基本上都是在 1 毫秒左右。

2）反应式编程探索：Vert.X && Webflux



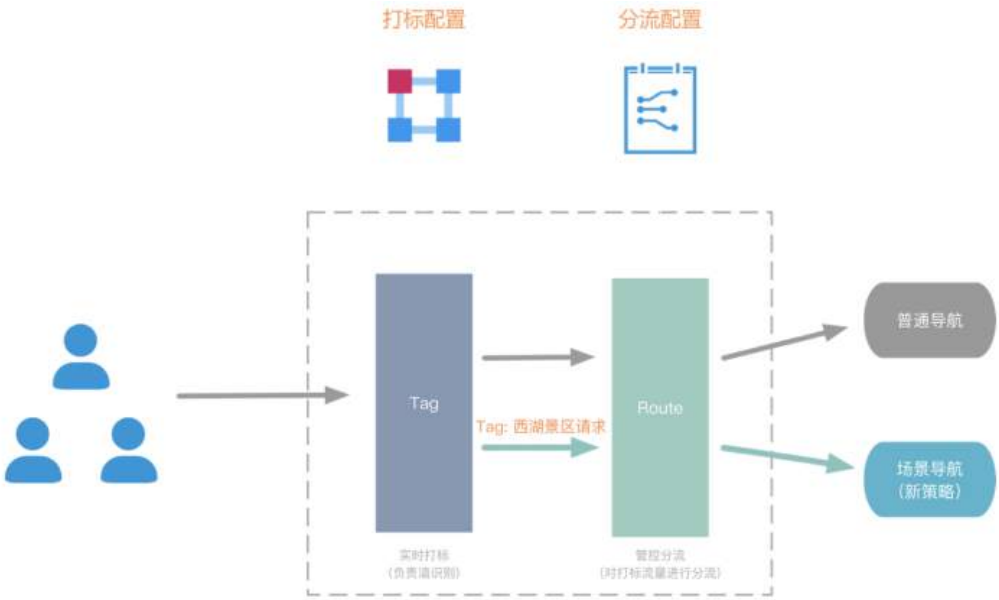
我们也做了反应式编程，主要用 Vert.X。我们一些同步调用的场景需要修改为异步，他比较特殊，RPC 的依赖比较少，主要是同步依赖 RDB、Mongodb、Http 接口等，这时候我们用 Vert.X 来做 IO 任务及数据编排，Http 异步调用还是用的 AsyncHttpClient。最后的效

果，QPS 大概在 5 万左右，RT 是 22 毫秒左右。
高德现在的打车业务中有一个业务场景，服务里要调服务 A、服务 B、服务 C、服务 D、服务 E、服务 F，最多的时候要调 27 个服务，还要做业务逻辑。用 Webflux 更合适一些，不仅可以做到异步化改造，还可以用它做复杂业务逻辑编排。使用 Webflux 可以直接使用 Netty 处理链接、业务层用 Reactor 交互，全反应式编程，IO 线程与业务线程互不阻塞，最大限度压榨 CPU 资源。

在这个项目里，反应式编程最终达到的效果，QPS 提升了 3 倍，RT 降低 30%。

- **API 聚合、数据编排与打标分流**

面对新的业务，压力越来越大，并且每次迭代的速度要求越来越快。目前 API 数量超过 500+，接口数据项超过 400。对于 **API 的定制化、复用**，怎么解？就是通过 API 聚合和数据编排。



打标分流是另外一个挑战，随着业务的发展，很多服务都需要做架构升级，需要做重构，

算法和模型也需要不断的调优，这时候对于业务或者研发来说，对业务参数进行打标和分流，可以降低风险。

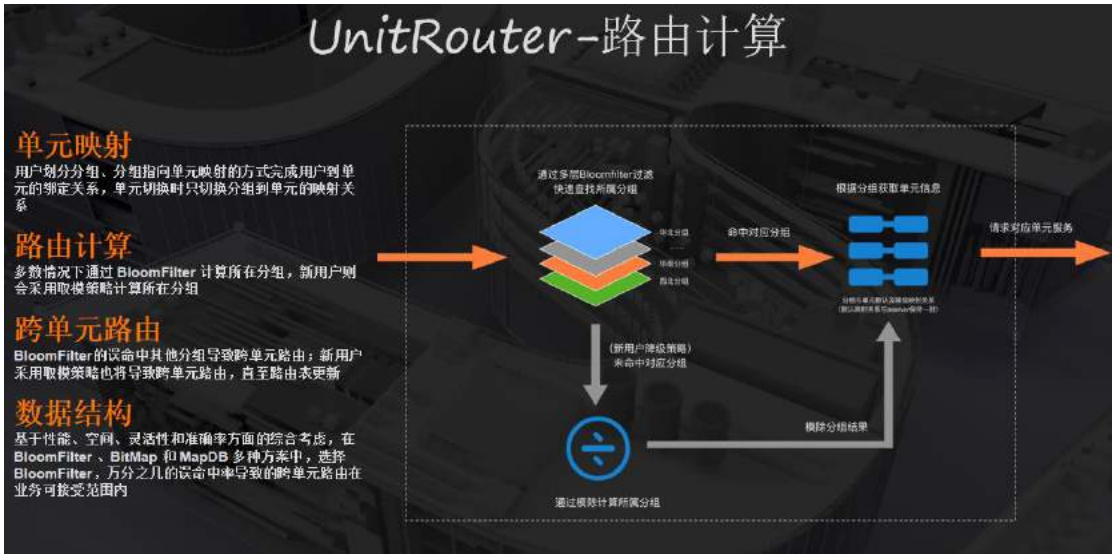
- **高德单元化网关**

1）高德单元化网关：路由策略

对于**业务异地多活、单元化需求**，我们做了单元化路由的解决方案，这里最核心的，给业务提供的能力是：当有用户请求过来时，能够实现就近接入能力，尽量减少跨单元调用。

单元路由主要帮助业务解决异地多活的能力，我们支持的路由策略，主要分为两种：第一种是基于路由表，第二种是基于取模策略。如果你的应用对就近接入需求比较强烈，对延迟敏感，就可以用基于路由表策略。如果是对多单元同写敏感度高的场景，用取模策略更合适。两种我们都支持。

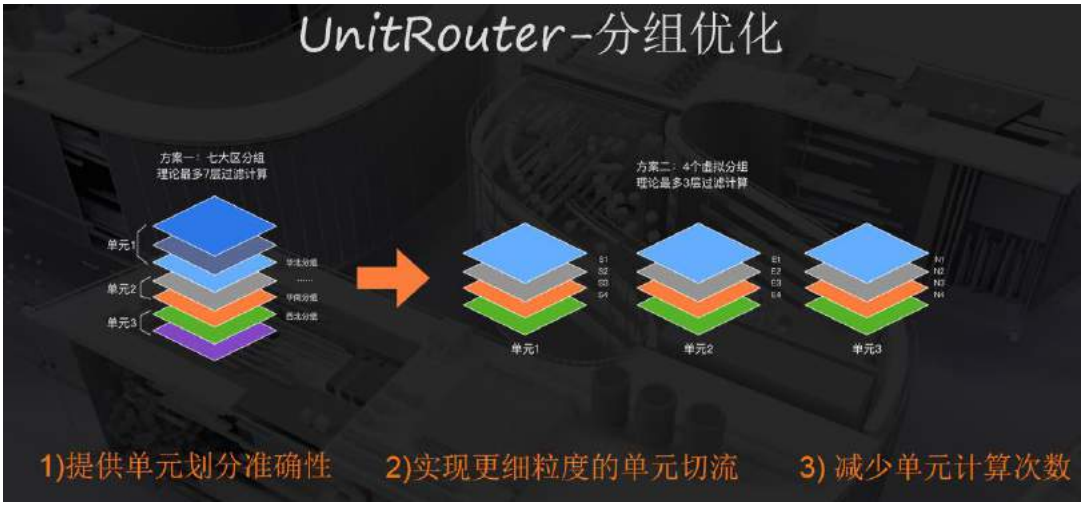
2）高德单元化网关：路由计算



上图是我们做的路由计算核心逻辑图。具体而言注意以下几点：1）**单元映射**，用户划分分组、分组指向单元映射的方式完成用户到单元的绑定关系，单元切换时只切换分组到单元

的映射关系；2）**路由计算**，多数情况下通过 BloomFilter 计算所在分组，新用户则会采用取模策略计算所在分组；3）**跨单元路由**，BloomFilter 的误命中会导致跨单元路由；新用户采用取模策略也将导致跨单元路由，直至路由表更新；4）**数据结构**，基于性能、空间、灵活性和准确率方面的综合考虑，在 BloomFilter 、BitMap 和 MapDB 多种方案中，选择 BloomFilter，万分之几的误命中率导致的跨单元路由在业务可接受范围内。

3）高德单元化网关：分组优化



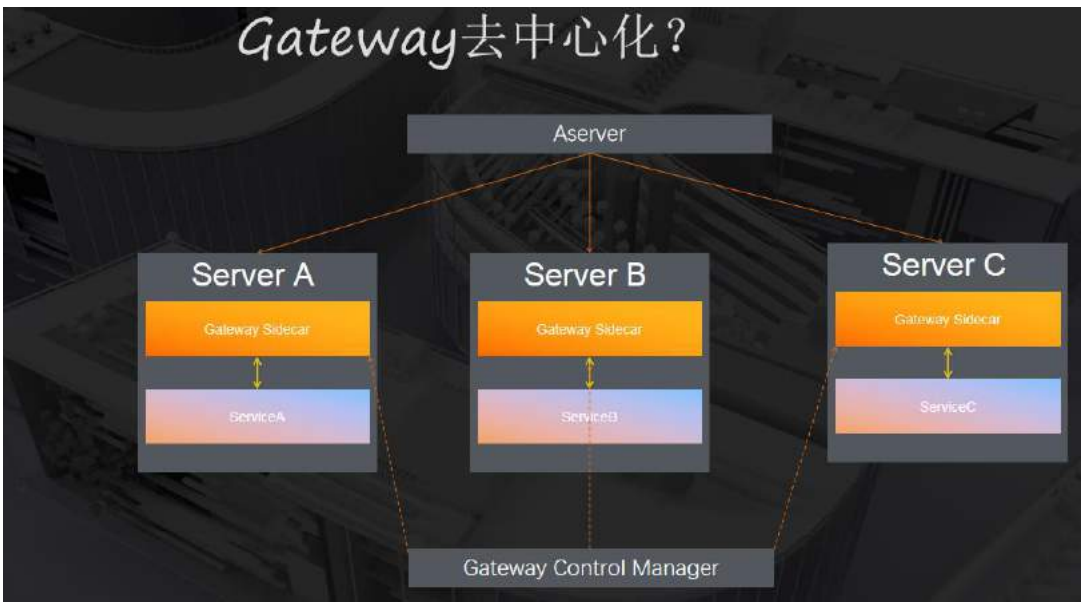
这个是目前正在迭代做的网关虚拟分组优化，分为 3 单元*4 片，每个单元分成四个片。

目标提高单元划分的准确性，同时每次访问需要 7 次计算优化为 3 次，同时解决以前如果发现单元出现问题流量只能全切，现在可灰度切量。

目前使用的案例有云同步、用户等。用户单元化的案例，最终的收益是，整个单元计算耗时小于 2 毫秒，跨单元路由比例低于 3%。

3.思考及规划

Gateway 现在是集中化的场景，怎么变成分布式的解决方案？



这方面我们也做了尝试。分布式网关一般有两种实现路径：第一种是做 SDK，第二种是做边车或服务网格的方式。SDK 方式的分布式网关我们已经在部分场景使用，缺点是对异构支撑困难，和应用的隔离性不好，好处是开发比较快，目前每天也有过百亿的请求在访问。

边车或者服务网格其实是我们架构的终局，他能解决异构、应用系统隔离性等问题。因为：

- Gateway Sidecar 与业务应用运行于同服务器的独立进程，既具有分布式部署优势又具备较好的隔离性。

- Gateway Control Manager 负责管理分布式 Gateway Sidecar，相当于 Service Mesh 的控制面，主要负责网关配置和元数据管理、服务高可用以及统计打点、异常监控和报警等。

服务网格优势是去中心化的分布式部署方式，天然就具备高可用性和水平扩展性，无单点和性能瓶颈问题，缺点是不太适合实现聚合 API 的实现。服务网格我们目前是基于蚂蚁 SOFA 来做，主要用来解决异构 RPC 调用的问题。

最后给个建议，根据实际经验，大家如果在做服务或 Gateway 相关的事，如果你面临的挑战是机器数量减少一半，性能提升一倍，全链路异步化架构可能会对你有所帮助。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

高德服务单元化方案和架构实践

作者：韦王

1.导读

本文主要介绍了高德在服务单元化建设方面的一些实践经验，服务单元化建设面临很多共性问题，如请求路由、单元封闭、数据同步，有的有成熟方案可以借鉴和使用，但不同公司的业务不尽相同，要尽可能的结合业务特点，做相应的设计和处理。

2.为什么要做单元化

- 单机房资源瓶颈

随着业务体量和服务用户群体的增长，单机房或同城双机房无法支持服务的持续扩容。

- 服务异地容灾

异地容灾已经成为核心服务的标配，有的服务虽然进行了多地多机房部署，但数据还是只在中心机房，实现真正意义上的异地多活，就需要对服务进行单元化改造。

3.高德单元化的特点

在做高德单元化项目时，我们首先要考虑的是结合高德的业务特点，看高德的单元化有什么不一样的诉求，这样就清楚哪些经验和方案是可以直接拿来用的，哪些又是需要我们去解决的。

高德业务和传统的在线交易业务还是不太一样，高德为用户提供以导航为代表的出行服务，很多业务场景**对服务的 RT 要求会很高**，所以在做单元化方案时，尽可能减少对整体服务 RT 的影响就是我们需要重点考虑的问题，尽量做到数据离用户近一些。转换到单元化技术层面需要解决两个问题:

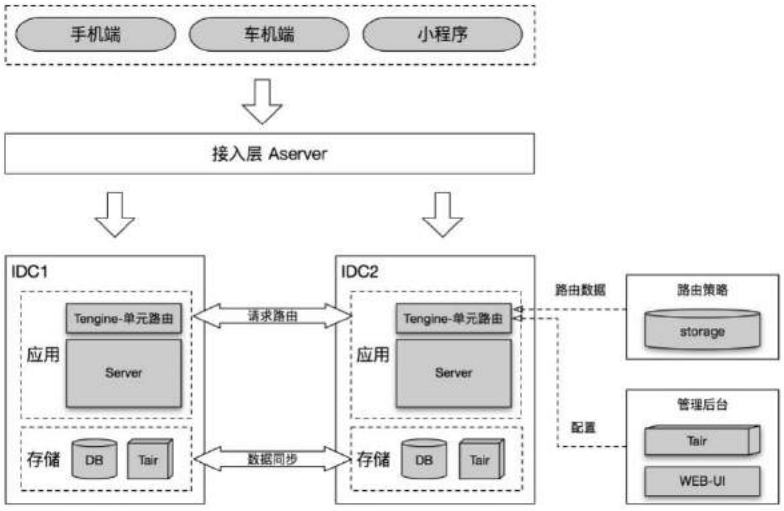
- 用户设备的单元接入需要尽可能的做到就近接入，用户真实地理位置接近哪个单元就接入哪个单元，如华北用户接入到张北，华南接入到深圳。

- 用户的单元划分最好能与就近接入的单元保持一致，减少单元间的跨单元路由。如用户请求从深圳进来，用户的单元划分最好就在深圳单元，如果划到张北单元就会造成跨单元路由。

另外一个区别就是高德很多业务是无须登录的，所以我们的单元化方案除了用户 ID 也要支持基于设备 ID。

4.高德单元化实践

服务的单元化架构改造需要一个至上而下的系统性设计，核心要解决**请求路由、单元封闭、数据同步**三方面问题。

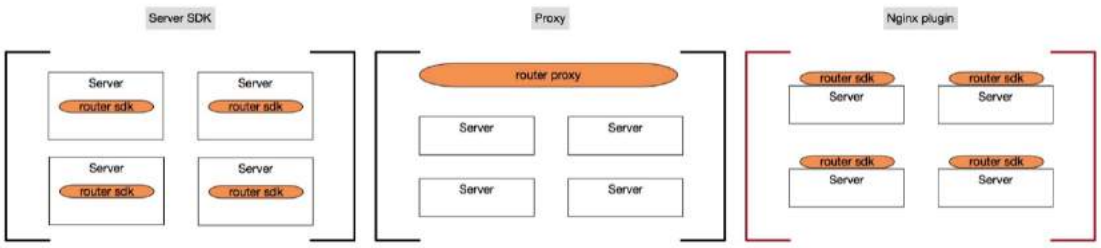


请求路由：根据高德业务的特点，我们提供了取模路由和路由表路由两种策略，目前上线应用使用较多的是路由表路由策略。

单元封闭：得益于集团的基础设施建设，我们使用 vipserver、hsf 等服务治理能力保证服务同机房调用，从而实现单元封闭(hsf unit 模式也是一种可行的方案，但个人认为同机房调用的架构和模式更简洁且易于维护)。

数据同步：数据部分使用的是集团 DB 产品提供的 DRC 数据同步。

单元路由服务采用什么样的部署方案是我们另一个要面临的问题，考虑过以下三种方案：

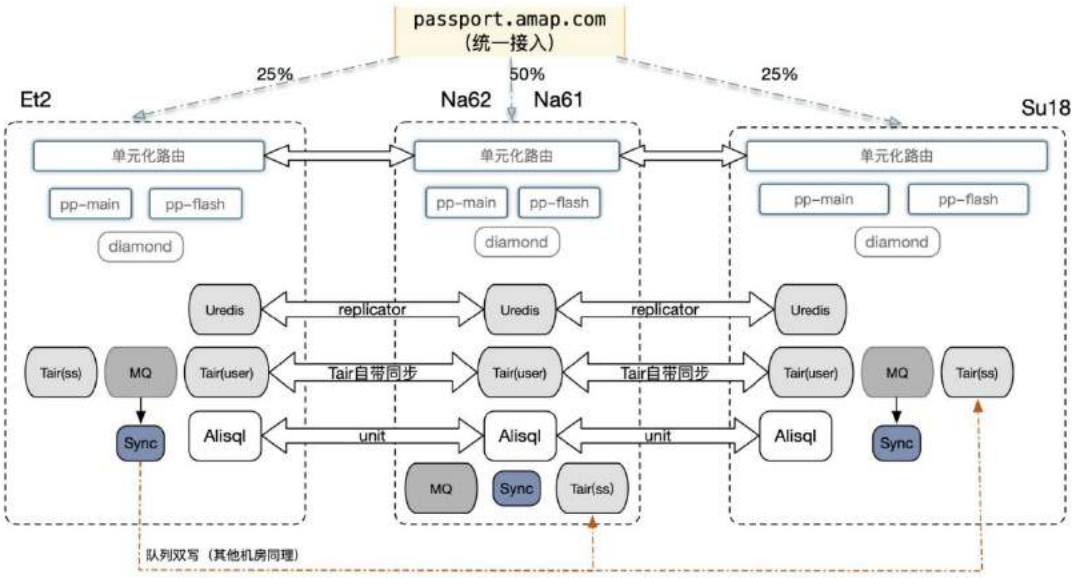


第一种 SDK 的方式因为对业务的强侵入性是首先被排除的，统一接入层进行代理和去中心化插件集成两种方案各有利弊，但当时首批要接入单元化架构的服务很多都还没有统一接入到 gateway，所以基于现状的考虑使用了去中心化插件集成的方式，通过在应用的 nginx 集成 UnitRouter。

服务单元化架构

目前高德账号，云同步、用户评论系统都完成了单元化改造，采用三地四机房部署，写入量较高的云同步服务，单元写高峰能达到数 w+QPS (存储是 mongodb 集群)。

以账号系统为例介绍下高德单元化应用的整体架构。



账号系统服务是三地四机房部署，数据分别存储在 tair 为代表的缓存和 XDB 里，数据存储三地集群部署、全量同步。账号系统服务器的 Tengine 上安装 UntiRouter，它请求的负责单元识别和路由，用户单元划分是通过记录用户与单元关系的路由表来控制。

PS：因历史原因缓存使用了 tair 和自建的 uredis(在 redis 基础上添加了基于 log 的数据同步功能)，目前已经在逐步统一到 tair。数据同步依赖 tair 和 alisql 的数据同步方案，以及自建的 uredis 数据同步能力。

就近接入实现方案

为满足高德业务低延时要求，就要想办法做到数据(单元)离用户更近，其中有两个关键链路，一个是通过 aserver 接入的外网连接，另一个是服务内部路由(尽可能不产生跨单元路由)。

措施 1：客户端的外网接入通过 aserver 上的配置，将不同地理区域(七个大区)的设备划分到对应近的单元，如华北用户接入张北单元。

措施 2：通过记录用户和单元关系的路由表来划分用户所属单元，这个关系是通过系统日志分析出来的，用户经常从哪个单元入口进来，就会把用户划分到哪个单元，从而保证请求入口和单元划分的相对一致，从而减少跨单元路由。

所以，在最终的单元路由实现上我们提供了传统的取模路由，和为降延时而设计的基于路

由表路由两种策略。同时，为解无须登录的业务场景问题，上述两种策略除了支持用户 ID，我们同时也支持设备 ID。

路由表设计

路由表分为两部分，一个是用户-分组的关系映射表，另一个是分组-单元的关系映射表。在使用时，通过路由表查对应的分组，再通过分组看用户所属单元。分组对应中国大陆的七个大区。

先看“用户-(大区)分组”：

路由表是定期通过系统日志分析出来的，看用户最近 IP 属于哪个大区就划分进哪个分组，同时也对应上了具体单元。当一个北京的用户长期去了深圳，因 IP 的变化路由表更新后将划进新大区分组，从而完成用户从张北单元到深圳单元的迁移。

再看“分组-单元”：

分组与单元的映射有一个默认关系，这是按地理就近来配置的，比如华南对应深圳。除了默认的映射关系，还有几个用于切流预案的关系映射。

单元分组	默认配置	单元一切流预案	单元二切流预案
东北	单元一	单元二	单元一
华北	单元一	单元三	单元一
西北	单元二	单元二	单元三
.....	单元三	单元三	单元三
模1分组	单元一	单元二	单元二
模2分组	单元二	单元二	单元二
模3分组	单元三	单元三	单元三

老用户可以通过路由表来查找单元，新用户怎么办？对于新用户的处理我们会降级成取模的策略进行单元路由，直至下次路由表的更新。所以整体上看新用户跨单元路由比例肯定是比老用户大的多，但因为新用户是一个相对稳定的增量，所以整体比例在可接受范围内。

路由计算

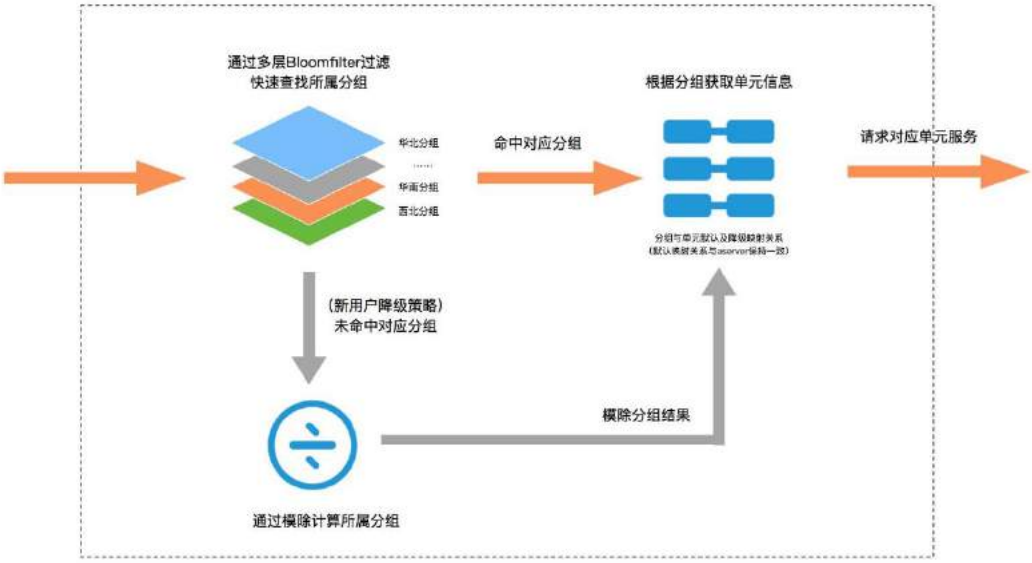
有了路由表，接下来就要解工程化应用的问题，性能、空间、灵活性和准确率，以及对服务稳定性的影响这几个方面是要进行综合考虑的，首先考虑外部存储会增加服务的稳定性风险，后面我们在 BloomFilter 、BitMap 和 MapDB 多种方案中选择 BloomFilter，万分之几的误命中率导致的跨单元路由在业务可接受范围内。

	BitMap	MapDB	Bloom Filter
支持模型	Uid	Uid、Tid、...	Uid、Tid、...
空间占用	小	大	适中
效率	良好	一般	良好
准确率	100%	100%	>99%
实时修改	是	是	否

通过日志分析出用户所属大区后，我们将不同分组做成多个布隆过滤器，计算时逐层过滤。这个计算有两种特殊情况：

1) 因为 BloomFilter 存在误算率，有可能存在一种情况，华南分组的用户被计算到华北了，这种情况比例在万分之 3 (生成 BloomFilter 时可调整)，它对业务上没有什么影响，这类用户相当于被划分到一个非所在大区的分组里，但这个关系是稳定的，不会影响到业务，只是存在跨单元路由，是可接受的。

2) 新用户不在分组信息里，所以经过逐层的计算也没有匹配到对应大区分组，此时会使用取模进行模除分组的计算。



如果业务使用的是取模路由而非路由表路由策略，则直接根据 tid 或 uid 计算对应的模除分组，原理简单不详表了。

单元切流

在发生单元故障进行切流时，主要分为四步骤

- 打开单元禁写 (跨单元写不敏感业务可以不配置)；
- 检查业务延时。
- 切换预案。
- 解除单元禁写。

PS：更新路由表时，也需要上述操作，只是第 3 步的切换预案变成切换新版本路由表；单元禁写主要是了等待数据同步，避免数据不一致导致的业务问题。

核心指标

- 单元计算耗时 1~2ms。
- 跨单元路由比例底于 5%。

除了性能外，因就近接入的诉求，跨单元路由比例也是我们比较关心的重要指标。从线上观察看，路由表策略单元计算基本上在 1、2ms 内完成，跨单元路由比例 3%左右，整体底于 5%。

5.后续优化

统一接入集成单元化能力

目前大部分服务都接入了统一接入网关服务，在网关集成单元化能力将大大减少服务单元化部署的成本，通过简单的配置就可以实现单元路由，服务可将更多的精力放在业务的单元封闭和数据同步上。

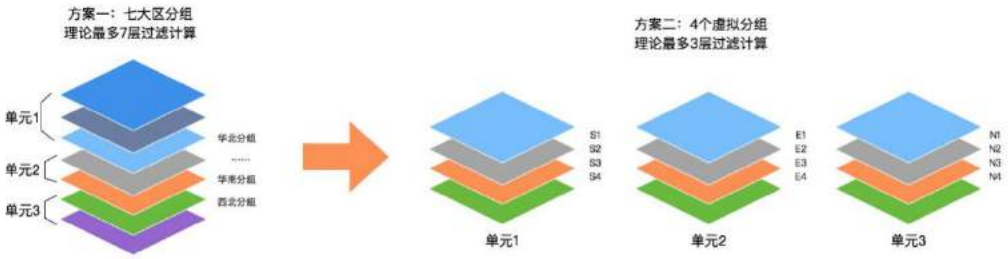
分组机制的优化

按大区分组存在三个问题：

- 通过 IP 计算大区有一定的误算率，会导致部分用户划分错误分组。
- 分组粒度太大，单元切流时流量不好分配。举例，假如华东是我们用户集中的大区，切流时把这个分组切到任意一个指定单元，都会造成单元服务压力过大。
- 计算次数多，分多少个大区，理论最大计算次数是有多少次，最后采取取模策略。

针对上述几个问题我们计划对分组机制做如下改进

- 通过用户进入单元的记录来确认用户所属单元，而非根据用户 IP 所在大区来判断，解上述问题 1。
- 每个单元划分 4 个虚拟分组，支持更细粒度单元切流，解上述问题 2。
- 用户确实单元后，通过取模来划分到不同的虚拟分组。每个单元只要一次计算就能完成，新用户只需经过 3 次计算，解上述问题 3。



热更时的双表计算

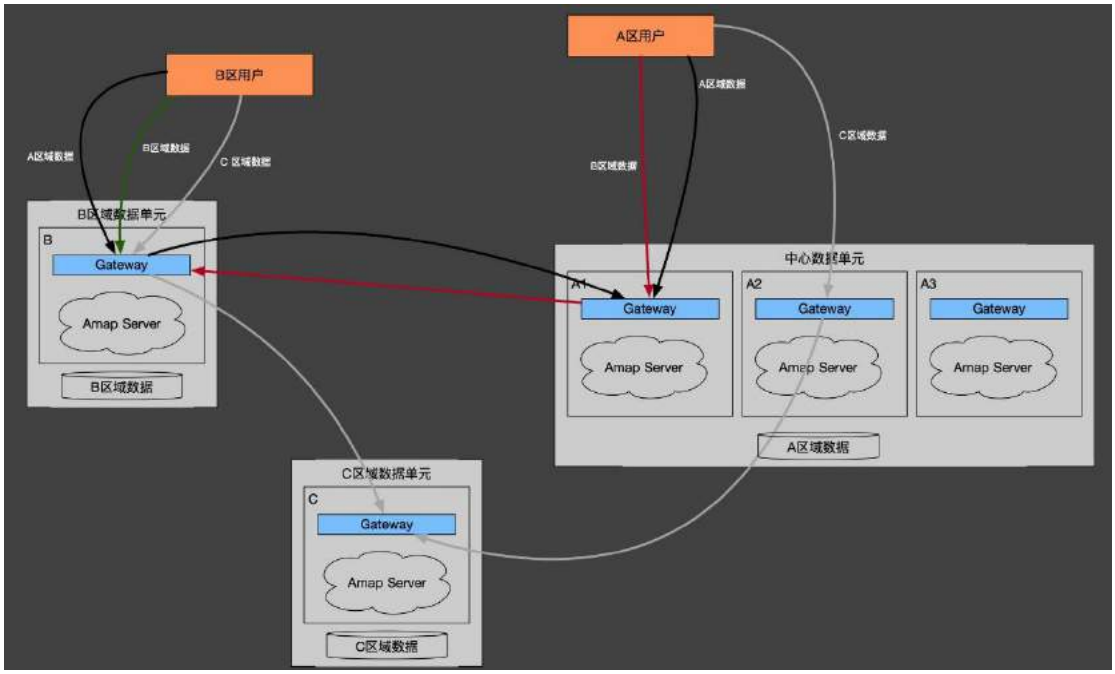
与取模路由策略不同，路由表策略为了把跨单元路由控制在一个较好的水平需要定期更新，目前更新时需要一个短暂的单元禁写，这对于很多业务来说是不太能接受的。

为优化这个问题，系统将在路由表更新时做双(路由)表计算，即将新老路由表同时加载进内存，更新时不再对业务做完全的禁写，我们会分别计算当前用户(或设备)在新老路由表的单元结果，如果单元一致，则说明路由表的更新没有导致该用户(或设备)变更单元，所以请求会被放行，相反如果计算结果是不同单元，说明发生了单元变更，该请求会被拦截，直至到达新路由表的一个完全起用时间。

优化前服务会完全禁写比如 10 秒(时间取决于数据同步时间)，优化后会变成触发禁写的是这 10 秒内路由发生变更的用户，这将大大减少对业务的影响。

服务端数据驱动的单元化场景

前面提到高德在路由策略上结合业务的特别设计，但整体单元划分还是以用户(或设备)为维度来进行的，但高德业务还有一个大的场景是我们未来要面对和解决的，就是以数据维度驱动的单元设计，基于终端的服务路由会变成基于数据域的服务路由。



高德很多服务是以服务数据为核心的，像地图数据等它并非由用户直接产生。业务的发展数据存储也将不断增加，包括 5G 和自动驾驶，对应数据的爆发式增长单点全量存储并不实现，以服务端数据驱动的服务单元化设计，是我们接下来要考虑的重要应用场景。

6.写在最后

不同的业务场景对单元化会有不同的诉求，我们提供不同的策略和能力供业务进行选择，对于多数据服务我们建议使用业务取模路由，简单且易于维护；对于 RT 敏感的服务使用路由表的策略来尽可能的降低服务响应时长的影响。另外，要注意的是强依赖性的服务要采用相同的路由策略。

招聘

高德工程技术中心长期招聘 Java、Golang、Python、Android、iOS 前端资深工程师和技术专家，职位地点：北京，欢迎有兴趣的同学投递简历到 tongxian.wxx@alibaba-inc.com

系统重构的道与术

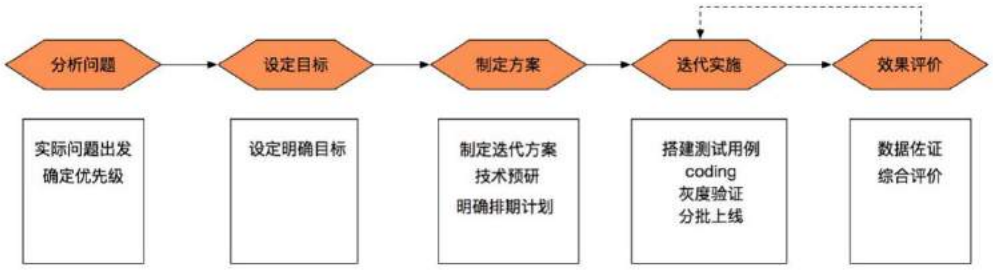
作者：韦王

最近参与了很多重构项目，有以提高服务器资源利用率为目标的 Gateway 网关、AMAPS 等服务的重构，也有以提升架构合理性和研发效率为目标的共享业务服务化拆分，借此机会把相关内容梳理一下，是分享更是自我总结和学习。准备以重构工作中容易产生误区的地方或容易被忽视的重点来聊聊，既不重复网上千篇一律的各种方案资料，也对重构工作有参考价值。

什么是“道和术”？个人简单的理解，道就是思想，术是方法。可谓有道无术，术尚可求也；有术无道，止于术。分别从重构的基本思路和原则，以及常见重构方案的应用来分别讲讲系统重构的“道与术”。

1.系统重构之道

现在是进行重构的恰当时机吗？重构前需要做什么准备？如何保障重构工作顺利完成并达成预期目标？从这几个大家都关心的问题，来谈谈重构工作遵循的基本思路和原则。



从实际问题出发

“不能解决实际问题的重构就是耍流氓 ”，从实际问题出发，切勿为了重构而重构，看似简单的道理，但现实中确实存在为了重构而发起的重构，或许是想应用诱人的新技术，或许是为了跟上流行趋势，甚至有自己主动 YY 需求而发起的重构。作为工程师我们需要谨记系统稳定高于一切，任何重构都存在风险，没有业务收益的重构相当于平白让业务承担非必要的风险，这是一种极不负责的表现。

所以，发起重构项目时，先想明白要解决什么实际问题，是为了提升性能？还是加强安全？或是为了快速的持续集成和发布？想明白再行动。

设定明确目标

目标是否明确很大程度上决定了事情的最终效果，重构项目也是如此。在组织管理、目标管理课程上经常会提及目标设定的 SMART 原则，同样，重构项目也要有具体的、可衡

量、可执行、可实现、且有时间限制的目标，可执行、可实现、且有时间限制这三者好理解，重点讲讲具体可衡量，上面提到的待解决问题可不可以作为重构项目的目标吗？答案不可以，问题就出在具体可衡量上，就拿以解决性能问题的重构项目为例，目标应该是服务响应 RT 要降多少？或是单核 QPS 承载量提升多少？甚至也可以是服务器资源减少多少？这才是具体可衡量的目标。

那有些不好量化的目标怎么做到可衡量呢？拿提升服务高可用性为目的的重构项目为例，目标确实不好量化，针对这样的问题可以以具体事件为衡量标准，比如实现机房故障用户无感知，或底层故障自动降级和恢复等 （系统高可用经常使用几个 9 的指标来评定，但它是一个事后采集指标，用作指导中短周期项目目标并不适合）。

阿里内部经常会提到工作抓手，而这个具体可衡量的目标就是我们重构工作的抓手。

设计要有度

“设计不足”和“过度设计”一样都是设计失误，设计不足是因为缺乏必要的抽象思维和前瞻性思考，使得系统存在设计缺陷；而过度设计往往是对系统问题把脉不清，偏离实际需求过度追求扩展性而引入了多余的设计，过度设计的结果并不只是并没什么用处的扩展功能，更多时候它会带来一些新的问题，比如增加系统维护和迭代的成本、增加线上问题排查难度等等。

设计要有度除了设计不足和过度设计外，还有一个成本收益的层面需要考虑，可能有些设计的引入能解决一部分问题，但它方案过于复杂，实施成本过高，这时候就需要从收益产出比上去权衡是否要采纳。

设计不足几乎没有捷径，需要不断的学习和经验积累。而过度设计，需要我们在做设计方案时多想一想相关设计的必要性以及成本收益问题。

小步快走

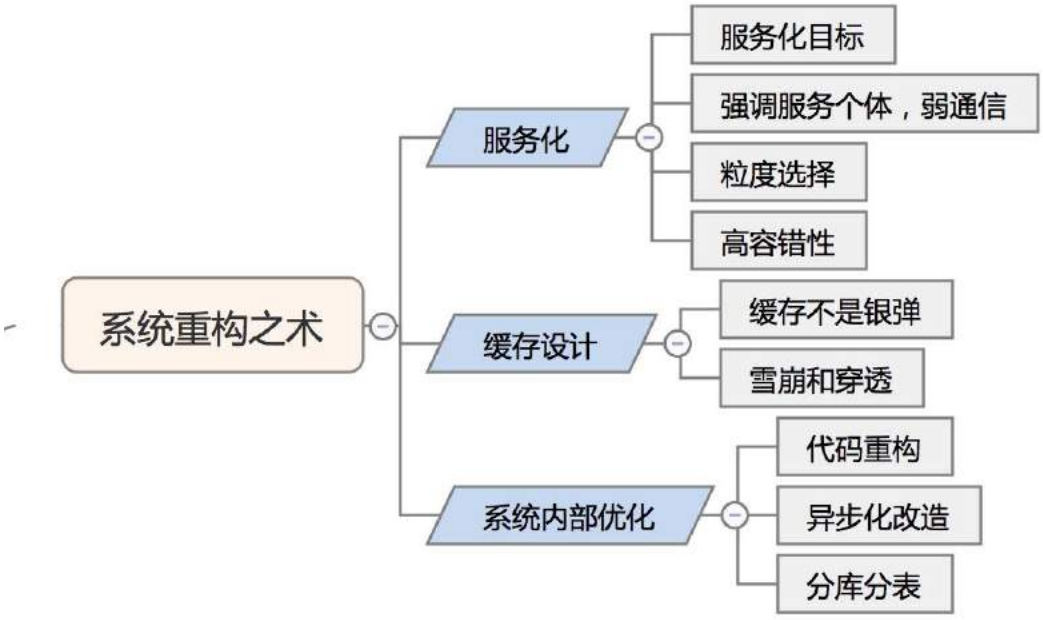
提前做好迭代计划是在重构工作中容易被忽略的重要事项，重构方案设计之初就要考虑如何分阶段实施，甚至为了达到分阶段目的有时需要在设计方案上做一些妥协。如果把重构比作建筑施工，小步快走层层分离的策略就相当于搭建施工现场的脚手架，是一种把风险控制在可接受范围的有效手段。

举一个实际的重构经历，是一个订单服务，订单量不大但业务种类很多(酒店、门票、火车票等等)。最终设计按订单处理流程将系统划分四个模块：下单模块、CP 订单同步模块、订单处理模块、统计模块。有同学问过订单量不大拆多个模块合适吗？其实除了设计本身的考虑因素外，按订单流程拆多个模块很重要的原因是为了能够分阶段上线和验证，只有这样风险才真正可控 (因一些特定原因没按业务垂直拆分，这里暂不详表)。

所以系统重构尽可能采用迭代实施方案，而且是从一开始就要考虑。

2.系统重构之术

在系统重构工作中，会使用一些具体的手段来解决所面对的特定问题，在术的部分聊聊重构中经常使用的一些方案，方案的具体内容资料很多就不写，我们这里重点聊聊相关方案应用时要注意考虑哪些问题。



服务化

服务化在很多重构项目中被提及，抽象、解耦、分治、统一是系统设计和重构的重要思想，服务化是该思想的重要实践，在运用服务化设计时，需要注意哪些问题？

服务化目标

做服务化设计或重构工作的时候，首先要想清楚服务化带来的价值，它也是我们做服务化工作的目标。

- **需求层面**：支持快速迭代
- **开发层面**：代码解耦，独立开发，降低维护成本
- **运维层面**：独立部署，单独扩容，降级控制

上面提的是服务化带来的价值，有意思的是，如果是一个不好的服务化设计，上面也会是服务化带来的问题，比如经常有同学抱怨服务化设计比之前开发上线更麻烦了。所以清晰的认识服务化目标是服务化工作的第一步，如果目标没有达成甚至带来的是负面效果，就要重新审视相关设计是否合理了。

- 强调服务个体，弱通信

在参与服务化工作的时候经常遇到同学上来就聊各种 RPC 框架或各种消息中间件，服务化是一种服务设计模式或者说一种设计思想。服务化工作强调的是服务个体设计，具体的通信方式至少在开始阶段不是那么重要。在不同阶段尽量聚焦核心问题。

- 粒度选择

服务化工作最难最依赖经验的是服务粒度的选择，如何结合系统实际特点正确定义子系统边界，一方面有相关设计原则可以参照(比如单一原则、无状态等等，网上资料很多)，更多的还是经验的积累，如果是系统重构工作，建议从优先级重要的模块进行提炼，粒度可大可小不好把控时，可以考虑先实施较大粒度的方案，这样即使有问题可以进一步优化拆分，但如果一下子过细导致过度设计，再想回去就比较难了。

- 高容错性

前面提到服务化强调服务个体，而作为独立的服务其容错性设计应该是要被重点考虑的，它也决定了整个服务体系的稳定性。所以服务化不仅仅是把服务拆分出来，拆分后要分析各服务之间的依赖，区别强弱关系，进行相关容错设计。

缓存设计

如果说数据缓存是重构工作中被使用最多的手段，估计不会有太大的歧义，使用数据缓存方案以下几点需要特别注意。

- 缓存不是银弹

涉及到性能优化经常会听到“那我们加个缓存”吧，确实数据缓存对性能提升的效果立竿见影，几乎成了很多同学在解决系统性能问题时条件反射式的选择。

无论是新系统设计还是老系统重构，在面对性能问题时不要总把数据缓存作为第一选择，它会蒙蔽你的眼睛，使你无法看到其他层面的问题。记得在之前一家企业软件公司工作时，跟着公司首席架构师做基础服务设计，他有一个要求就是系统初期设计不能考虑任何缓存设计，这让我印象深刻。性能提升就加缓存，其实是在用战术上的勤奋掩盖战略上的懒惰，解决问题时我们需要多角度多维度的全面评估，这样才有可能系统性的解决问题。

- 雪崩和穿透

数据缓存解决了数据读取性能问题，但同时也在系统架构里引入了新的故障点。

雪崩和穿透是引入数据缓存时需要考虑的问题。首先从业务层面要考虑相关情况下的降级策略和具体降级方案，从技术层面，缓存自身的高可用、缓存数据是否持久化、是否加入缓存预热机制、expire time 否要进行离散设计等等这些细节都是要考虑的。

系统内部优化

- 代码重构

代码优化分为代码结构优化和代码内容优化，后者重点在于如何识别代码中的 bad smell，有很多具体指导方法，这里就不提了。而前者更多的是对代码设计的调整，考验的是设计抽象能力，需要有较好的领域模型(DDD)知识。所以说一个好的程序员，他/她一定是个领域专家。

- 异步化改造

对于 IO 密集型的应用，异步化改造是很有效的手段，但实施起困难还是挺大的，体现在两方面，一是要有完整技术解决方案的支持；幸运的是已经有同事给我们趟好了路，解决了公司常用中间件（Hsf、Tair、Metaq、Tddl、Sentinel 等）异步化的问题，给大家重点推荐淘宝架构升级项目相关信息（羡慕无比，这种规模的重构可遇不可求），项目核心就是微服务化和基于响应式编程的异步化改造，从中可见零起步做系统异步化改造是多大的一个工程。另一方面是团队自身学习成本，需要所有参与者对异步化、响应式编程模型要有很好的认知，这点也尤其重要。

- 分库分表

数据量级快速增长单库无法满足业务需求时，分库分表是常用的应对方法。

如果是开发新系统，除非业务本身依赖海量数据，否则不建议在开发初期就实施分库分表，因为这会在一定程度上加大系统设计和开发的难度。而且一开始就让业务开发人员关切数据分库分表，如果他经验不足容易带来额外的困惑，将原本简单的问题处理复杂化。

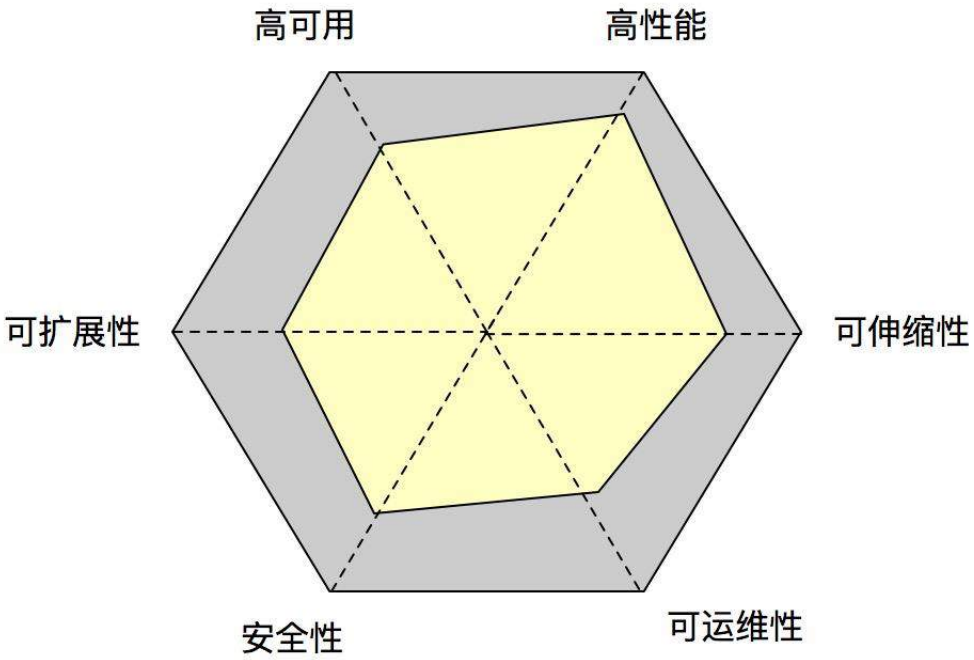
关于主键生成策略，有不同机制供大家选择，目前被使用和讨论较多的是 Snowflake，它有一个系统时间上的要求，另一个是 Tddl 的生成策略，兼顾了性能、全局唯一、单库递增的核心诉求。

分库分表后需要考虑跨库跨表查询的问题，首先业务上要尽可能的避免，但像订单业务就需要针对用户、卖家进行不同维度的数据拆分(可考虑主库从库分别使用用户、卖家作为分表键) 。如果是运营管理后台类多条件的复杂查询，不管是不是分库分表，单库同样支持不好，非海量数据可以考虑使用 ES，海量数据使用 ES+HBASE。

3.说说系统评估

最后简单的聊下系统评估，从研发角度，可以从高性能、高可用、可扩展性、可伸缩性、安全性、可运维性这六个维度来考量，没有一个标尺，也不建议使用所谓的标准，系统评估结果一定是结合业务实际情况的结论，比如高可用，一个运营管理类平台多实例部署可能就是良，但如果是线上交易系统，多机房只是起步要求。

做系统评估时结合业务实际情况，从上述六个维度分别按 严重缺失、不足、满足 三档进行评估，初步分析系统短板。另外特别需要注意的是，这些维度不是独立存在的，在针对某一方面进行重构优化时，要深入考虑对系统其他层面的影响。



最后想说的是做好系统重构并不容易，其困难不在具体问题的解决上，它是一个系统性工程，如何能做到全面考量以及考虑多方面因素后选出相对最优解，才是最大的挑战。

数据篇

漫话 | 地图数据处理之道路匹配篇

作者：晶阳

导读

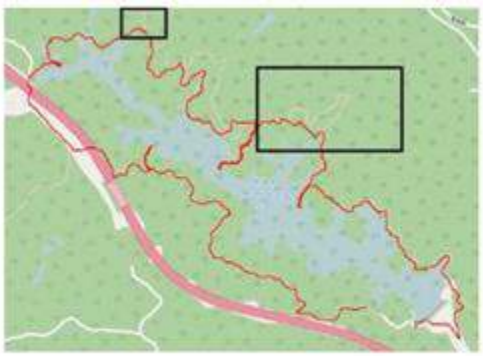
道路匹配是地图数据处理方面非常基础且重要的理论，特别是道路相关业务，一定避不开道路匹配的应用，这也是业务中普遍会碰到的痛点。

本文属于「漫话地图」系列，我们将结合地图数据业务的特点，持续介绍地图行业一些有趣的知识点，希望能抛砖引玉，为大家带来一定的启思和裨益，欢迎长期关注。

道路匹配定义及应用场景

定义

道路匹配是地图匹配理论的子集，通俗讲就是两幅地图 A 和 B，在没有唯一 ID 关联的情况下，如何确定地图 A 上的道路是 B 上一条道路的过程。如果做交通轨迹或者地图数据融合方面的研究，那么就一定会遇到地图匹配的问题。



匹配前

匹配后

地图匹配 Map Matching:不同条件下获取同一物景的地图之间的配准关系。

道路匹配是刨除了点和面状匹配之外的线状要素理论，道路的话就是路网，也是实际应用中研究最多、应用最广的一部分。

利用路网数据，采用适当算法，将目标定位映射到实际道路上的过程，具体来说道路匹配是：

- 地图匹配理论的首要子集

- 针对矢量拓扑道路数据的匹配模式
- 异源道路数据融合的关键

- 导航定位精度改善的重要手段

应用

导航位置预测

最常见的应用场景

道路匹配最直观的应用就是地图导航。手机自带 GPS 的定位精度在 10 米上下，单车道的宽度一般是 2-3 米。实际上，手机 GPS 定位不足以精确判断车辆行驶的实际道路。但大家会发现，通常情况下高德导航的道路定位都是很准确的，导航过程中地图会知道用户在某某道路而不是附近小区或者沟河中。

究其原因，用户导航过程中，系统一直在计算 GPS 位置和导航路线&路网间的配准关系，从而进行一定程度的纠偏，这也是提高定位精度的重要手段。

大家也会经常发现同一手机第三方客户 API 定位效果比高德地图差了不少，刨除硬件因素，实际上这里有算法水平的差异。

空间距离和评价曲线相似性的一般方法

离散点集匹配

路网匹配的两个方面应用：第一个是离散点集匹配，相对简单，随机离散点没有形状和拓扑关系，用欧氏距离作吸附即可，典型应用如离散热力图。

曲线拟合

实际中更有应用价值的是曲线拟合匹配关系，比如轨迹和路网，GPS 序列和导航路的相似性。

曲线信息更多，这方面比离散点集有更多的评价要素，也有更高的复杂度。评价曲线相似性的一般要素有长度、形状、曲率、拓扑关系、方向比如正向逆向、距离、属性例如交通规则左转右转禁行等信息。

算法分类

曲线匹配方法分类

基于几何信息的匹配算法考虑形状、角度等常规要素，属于早期的一些算法，实现最简单，准确度最低。基于拓扑信息的算法，准确度比几何方法大大提升，应用最广。基于概率预测的算法，实现比较困难，实际上应用不多。

目前有一些比较高级的算法理论，包括隐马模型等等，在实际应用中准确度是相对最高的。

实时算法主要用于在线导航，时间和空间复杂度低，离线算法用于数据处理的离线计算，算法复杂，追求最高准确度。

空间距离

线要素的匹配，主要通过几何、拓扑或语义相似度来进行识别，其中通过空间距离来进行要素匹配的常用方式有：

- 闵可夫斯基距离(Minkowski Distance)
- 欧氏距离(Euclidean Distance)
- 曼哈顿距离(Manhattan Distance)
- 切比雪夫距离(Chebyshev Distance)
- 汉明距离(Hamming distance)
- 杰卡德相似系数(Jaccard similarity coefficient)
- 豪斯多夫距离(Hausdorff Distance)
- 弗雷歇距离(Fréchet 距离)

评价曲线相似性-弗雷歇距离

什么是弗雷歇距离?

Fréchet distance（弗雷歇距离）是法国数学家 Maurice RenéFréchet 在 1906 年提出的一种路径空间相似形描述定义。

狗绳距离

弗雷歇距离通俗的讲就是狗绳距离，人和狗之间有一条狗绳约束。主人走路径 A，狗走路径 B，各自走完这两条路径过程中所需要的最短狗绳长度就是弗雷歇距离。

最大距离最小化

设定 t 是时间点，该时刻曲线 A 上的采样点为 A(t)，曲线 B 上采样点为 B(t)。如果使用欧氏距离，则容易定义 d (A(t),B(t)) 。在每次采样中 t 离散的遍历区间[0,1]，得到该种采样下的最大距离。弗雷歇距离就是使该最大距离最小化的采样方式下的值。

K-WALK 和弗雷歇排列

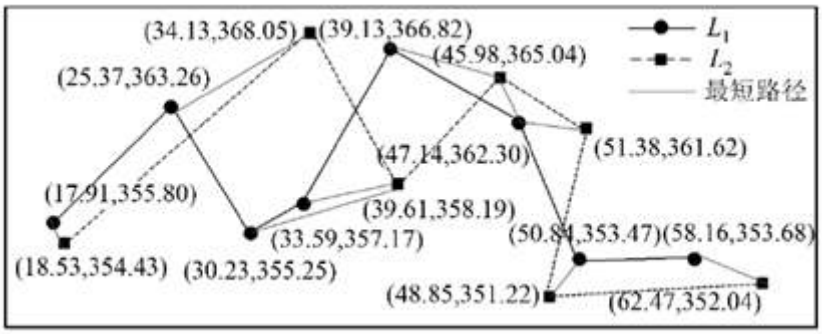
给定一个有 n 点的路链 P=〈p1 , p2 , … , pn 〉， 一个沿着 P 的 k 步分割成为 k 个不相交的非空子集，称为 K-WALK。

给定两个路链 A =〈a1 , …, am 〉, B =〈b1 , …, bn 〉， 一个沿着 A 和 B 的组合步(Paired Work)是 A 的 k-walk {Ai}i =1 …k 和一个沿着 B 的 k-walk {B i}i =1 …k 组成(1 ≤i ≤k) 。

链 A 和 B 间的离散 Fréchet 距离(discrete Fréchet distance)就是一个沿着链 A 和 B 的组合步 W={(Ai ,Bi)}的最小花费，这个组合步称为链 A 和 B 的 Fréchet 排列(Fréchet alignment)，也称为最佳组合步。弗雷歇距离实际上就是不断的遍历计算，尝试找出最佳组合步的过程。

利用平均弗雷歇距离评价曲线相似性

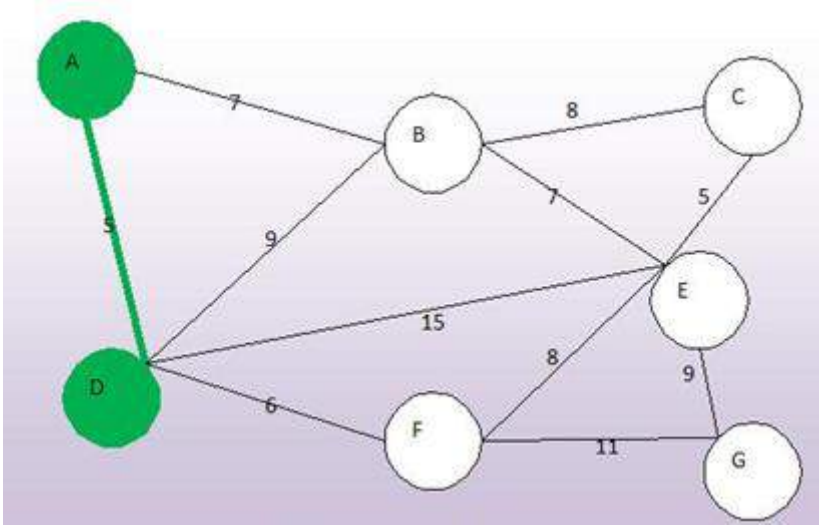
采用平均 Fréchet 距离代替离散 Fréchet 距离，因为前者是从顶点距离集合中选取的一个最大距离，易受到局部变形较大点的影响。



基于离散 Fréchet 距离识别曲线上点与点之间最短路径的方法，平均 Fréchet 距离通过计算离散要素点集之间的最短距离的平均值，来度量线要素间的相似性。

全局算法

两条曲线之间的匹配，研究的是 1:1 的关系，实际应用中 GPS 轨迹比较长的时候面临 M:N 全局择优的问题。



Dijkstra 求解全局最短路径，Frechet Distance 计算局部相似度

进行全局路线匹配时，需要考虑 M:N 的情况来确定整体路径，代表性的算法是使用弗雷歇距离来衡量待匹配序列和候选路段序列的匹配度，并作为路段的权重，由此构建网络图，通过计算最短路径得到最佳匹配结果。

最准确的匹配模型-隐式马尔科夫模型 HMM

除了弗雷歇距离外，再介绍一种高级算法，也是目前应用中准确度最高的一种算法(和最通用解决方案)—隐式马尔科夫模型 HMM。

20 世纪 60 年代，Leonard E. Baum 和其它作者在一系列的统计学论文中描述了隐式马尔科夫模型。它最初的应用之一是语音识别，80 年代成为信号处理的研究重点，现已成功用于故障诊断、行为识别、文字识别、自然语言处理以及生物信息等领域。

核心特征

- 隐式马尔科夫模型五要素：2 个状态集合和 3 个概率矩阵，Viterbi 算法。
- 隐含状态 S：马尔科夫模型中实际所隐含的状态，通常无法通过直接观测得到，这些状态之间满足马尔科夫性质。
- 可观测状态 O：通过直接观测而得到的状态，在隐式马尔科夫模型中与隐含状态相关联。
- 状态转移概率矩阵 A：描述隐式马尔科夫模型中各个状态之间的转移概率。

- 观测状态概率矩阵 B：表示在 t 时刻隐含状态是 Sj 条件下，其可观测状态为 Ok 的概率。

- 初始状态概率矩阵π：表示隐含状态在初始时刻 t=1 的概率矩阵。

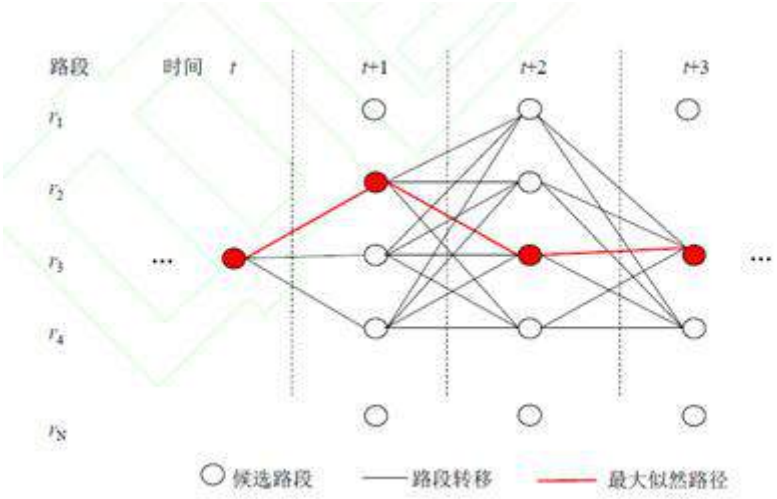
维特比算法详见:
https://blog.csdn.net/xueyingxue001/article/details/52396494

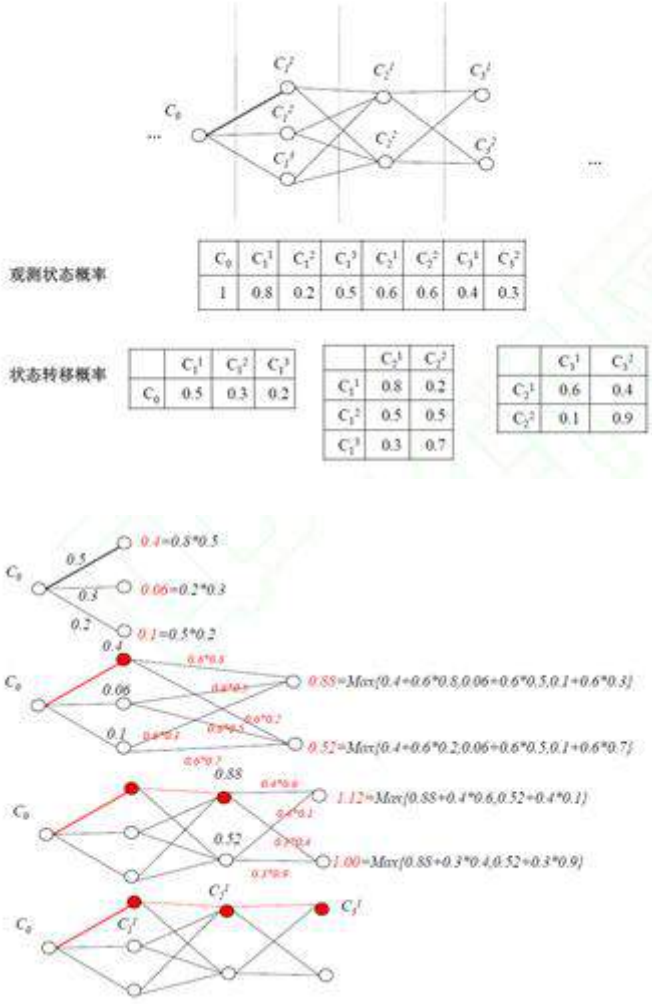
开源实现 Graphhopper-mapmatching，Java 实现的地图匹配项目，作为开源导航引擎 graphhopper 的子项目提供，最新实现用的是隐式马尔科夫模型，GitHub 地址：https://github.com/graphhopper/map-matching

解决三类问题

路网匹配实际是一个解码问题，基于 HMM 的路网匹配算法是在一系列观察的前提下，寻找最有可能产生这个观察序列的隐含状态序列。一系列 GPS 位置点集合是可观测状态，寻找最有可能产生位置点集合的路网隐藏序列。

基于隐马尔科夫模型的路网匹配过程





衍生算法集合

算法	观测概率矩阵	状态转移矩阵	初始状态概率
ST-Matching	均值 μ=0，标准差 σ=20 米的高斯分布。考虑道路网络的几何性质和局部特性。	空间分析函数和时间分析函数的乘积，考虑了道路网络的拓扑结构和前后两个时间点之间的速率信息	-
IVMM	μ=5 米，σ=10 米的高斯分布。	空间分析函数和时间分析函数的乘积，考虑了道路网络的拓扑结构和前后两个时间点之间的速率信息	-
HMMM	μ=0，σ= 4.07 米的高斯分布。σ 使用绝对中位差进行鲁棒估计。	用指数函数来拟合前后两个相邻 GPS 观测点的距离与两个候选点的距离之差的绝对值	在初始时刻时的观测概率矩阵
Simplified HMM	μ=0，σ=1.0 的高斯分布。	假设状态转移概率与两个交叉点间的距离正相关，也考虑行驶时间、成本等距离之外的其他因素	使用相应交叉点的观测概率来表示移动
OHMM	μ=0，σ= 6.86 米的高斯分布。σ 使用绝对中位差进行鲁棒估计，考虑了路段的宽度信息，引入一个超速处罚因子。	使用支持向量机来探悉状态转移概率，两个特征为距离差异函数和动量变化函数	-
QMM	μ=0，σ= 6.86 米的高斯分布。加入路段的速度限制。	用指数函数来拟合前后两个相邻 GPS 观测点的距离与两个候选点的距离之差的绝对值	-

其中 STM 算法，稳定健壮，实用性强，有成熟的研究和开源实现。

ACM SIGSPATIAL Cup

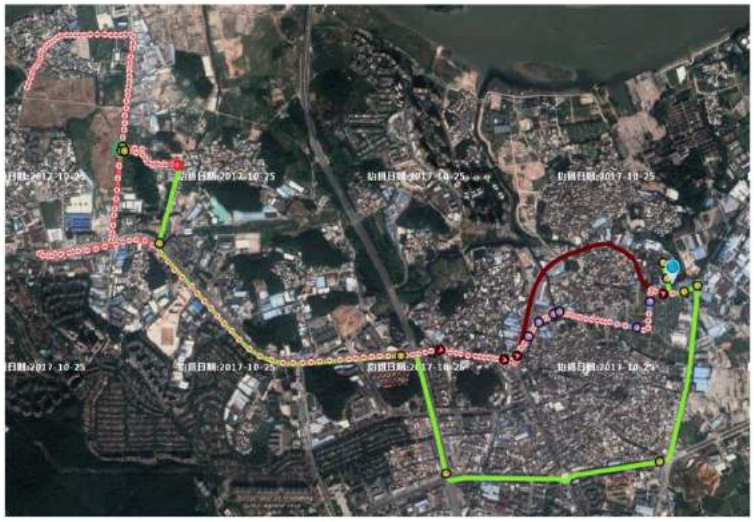
2012 年 ACM SIGSPATIAL Cup 是由 ACM 主办的全球范围内关于地图匹配算法的科技竞赛，竞赛吸引了来自全球 31 支专业级的参赛队伍。所有算法当中匹配准确率最高的两个

都是基于 HMM 的匹配算法。

道路匹配在业务中的应用

道路匹配在自动化项目中的应用，包括交通轨迹拟合度计算和道路自动识别等。

拟合度计算



自动识别

更多场景，比如异源数据融合、轨迹数据挖掘、交通数据分析、城市规划等领域，道路匹配都有广泛的应用前景。

招聘

高德地图数据研发团队热招 Java、Andorid、前端高级工程师/专家，数据仓库专家，定位算法及激光点云技术高级工程师/专家。职位地点：北京，欢迎有兴趣的同学投递简历到 xi.yang@alibaba-inc.com

系统性能提升利刃 缓存技术使用的实践与思考

作者：禅空

1.导读

按照现在流行的互联网分层架构模型，最简单的架构当属 Web 响应层+DB 存储层的架构。从最开始的单机混合部署 Web 和 DB，到后来将二者拆分到不同物理机以避免共享机器硬件带来的性能瓶颈，再随着流量的增长，Web 应用变为集群部署模式，而 DB 则衍生出主从机来保证高可用，同时便于实现读写分离。这一连串系统架构的升级，本质上是为了追求更高的性能，达到更低的延时。

高德作为一款国民级别的导航软件，导航路线的数据质量是由数据中心统一管理的。为了保证数据的鲜度，数据中心需要对不断变化的现实道路数据进行收集，将这些变化的信息保存到数据库中，从而保证导航数据的鲜度；另一方面数据中心内部多部门协调生产数据的时候，会产生海量请求查询最新生产的数据，这就要求数据的管理者要控制数据库连接数，降低请求的响应耗时，同时也需要保证返回数据的实时性。

在平衡数据鲜度和性能之间，高德数据中心针对不同的业务场景使用了不同的策略，达到了数据变更和缓存同步低延迟的目标，同时保障了系统的稳定性。

本文将提及的缓存技术则是提升性能的另一把利刃。然而任何技术都是有可为有可不为，没有最好的技术只有最适合的技术，因此在使用缓存之前，我们也需要了解下引入缓存模块所带来的好处和坏处。

2.缘起：为何使用缓存

在应用对外提供服务时，其稳定性受到诸多因素影响，其中比较重要的有 CPU、内存、IO(磁盘 IO、网络 IO)等，这些硬件资源十分宝贵，因此对于那些需要经过复杂计算才能得到结果的，或者需要频繁读取磁盘数据的，最好将结果缓存起来，避免资源的重复消耗。

2.1 CPU 瓶颈

如果项目中有很多正则表达式计算，或者某个计算结果是多次中间结果合并后才得出的，且 CPU 的使用率一直居高不下，那么就可以考虑是否应该将这些结果缓存起来，根据特定 Key 直接获取 Value 结果，减少中间链路的传递过程，减少 CPU 的使用率。

2.2 IO 瓶颈

众所周知，从磁盘获取数据受到磁盘转速、寻道速度、磁盘缓冲区大小等诸多因素影响，这些因素决定了磁盘的 IOPS，同时我们也知道对于数据的读写来说，CPU 的缓存读写速度> 内存的读写速度 >磁盘的读写速度。虽然磁盘内部也配备了缓存以匹配内存的读写速度，但其容量毕竟是有限的，那么当磁盘的 IOPS 无法进一步提升的时候，便会想到将数据缓存到内存中，从而降低磁盘的访问压力。这一策略常被应用于缓解 DB 数据库的数据访问压力。

3.选择本地缓存和分布式缓存的考量点

既然可以使用缓存来提升系统吞吐能力，那么紧接着遇到的问题就是选择本地缓存，还是分布式缓存？什么时候需要使用多级缓存呢？接下来，让我们聊一聊在使用缓存优化项目的过程中，本地缓存和分布式缓存的应用场景和优缺点。

3.1 本地缓存的优缺点和应用场景

统一进程带来了以下优势：

- 由于本地缓存和应用在同一个进程中，因而其稳定性很高，达到了和应用同生共死的境界。
- 由于在同一进程中，避免了网络数据传输带来的消耗，所有缓存数据直接从进程所在的内存区域获取即可。

强耦合性也会导致以下这些劣势：

- 本地缓存和应用共享一片 JVM 内存，争抢内存资源，无法水平扩展，且可能造成频繁的 GC，影响线上应用的稳定性。
- 由于没有持久化机制，在项目重启后缓存内数据就会丢失，对于高频访问数据，需要对数据进行预热操作。
- 多份进程内缓存存储着同样的数据内容，造成内存使用浪费。
- 同样的数据存储在不同的本地机器，数据变化后，很难保证数据的一致性。

结合以上优缺点，我们就会想到，如果有一种数据需要频繁访问，但一旦创建后就轻易不会改变，而且初始创建时就能预估占用的内存空间，那么这种类型的数据无疑是最适合用本地缓存存储了。

既然有了上述的应用场景，我们反观技术开发中的诉求，发现其实很多优秀的框架已经在这样使用了，比如缓存类 class 的反射信息，包括 field、method 等。因为 class 的数量是有限的，且内容不会轻易改变，在使用时无需再使用反射机制，而只需要从本地缓存读取数据即可。

3.2 分布式缓存的优缺点和应用场景

优势：

- 数据集中存储，消除冗余数据，解决整体内存的占用率，易于维护集群建缓存数据的一致性。
- 缓存中间件可以对缓存进行统一管理，便于水平扩容。

劣势：

- 依赖分布式缓存中间件稳定性，一旦挂了，容易造成缓存雪崩。
- 由于是跨机器获取缓存数据，因此会造成数据传输的网络消耗，以及一些序列化/反序列化的时间开销。

对于上述缺点中，网络耗时等开销是难免的，而且这些操作耗费的时间在可接受范围内，而对于中间件的稳定性则可以通过服务降级、限流或者多级缓存思路来保证。我们主要看中的是它的优点，既然分布式缓存天然能保证缓存一致性，那么我们倾向于将需要频繁访问却又经常变化的数据存放于此。

4.选择缓存框架的衡量标准

在了解了何时使用缓存以及缓存的优缺点后，我们就准备大刀阔斧开始升级系统了，可紧接着的问题也随之出现，对于本地缓存和分布式缓存，到底应该使用什么框架才是最适用的呢？

现在的技术百花齐放，不同的技术解决的问题侧重点也不同，对于本地缓存来说，如果无资源竞争的代码逻辑，可以使用 HashMap，而对于有资源竞争的多线程程序来说，则可以使用 ConcurrentHashMap。但以上二者有个通病就是缓存占用只增不减，没有缓存过期机制、也没有缓存淘汰机制。

那么本地缓存是否有更高性能的框架呢？而对于分布式缓存，领域内常用的 Redis 和 Memcache 又应该怎样取舍呢？本小节期望通过横向对比的方式，分别给出一个比较通用的缓存框架方案，当然如果有个性化需求的，也可以根据不同缓存框架的特性来取舍。

不同本地缓存框架的横向对比，如下表所示：

比较项	HashMap	ConcurrentHashMap	Guava Cache	Caffeine
读写性能	很好，但限于单线程	很好，分段锁	较好，掺杂元素淘汰操作，影响读写性能	很好，Disruptor异步队列解耦对元素淘汰
淘汰算法	无	无	LRU淘汰算法，性能一般	Window-TinyLFU算法
功能丰富度	单一	单一	丰富	和Guava Cache一样
持久化	无	无	无	无

总结：如果不需要淘汰算法则选择 ConcurrentHashMap，如果需要淘汰算法和一些丰富的

API，推荐选择 Caffeine。

不同分布式缓存框架的横向对比，如下表所示：

比较项	Memcache	Redis
读写性能	很高	很高
数据结构	Value支持类对象，纯KV结构	Value支持字符串、列表、哈希、集合、有序集合
存储方式	纯物理内存	物理内存+swap虚拟内存机制
持久化	无	支持

对于存储容量而言，Memcache 采用预先分配不同固定大小存储单元的方式，内存空间使用并不紧凑。如果存储 Value 对象大小最大为 1MB，那么当一个对象有 1000KB，那么会存储到大小最匹配 1MB 的单元中，因此会浪费 24KB 的内存；而 Redis 是使用之前才去申请空间，内存使用紧凑，但频繁对内存的扩容和收缩，可能造成内存碎片。

总结：由于 Redis 具有丰富的数据结构能满足不同的业务场景需求，同时 Redis 支持持久化，能有效地解决缓存中间件重启后的数据预加载问题，因此大多数应用场景中还是推荐使用 Redis。

5.缓存框架使用过程的知识点

不论是本地缓存还是分布式缓存，在使用缓存提升性能的时候，必然会考虑缓存命中率的高低，考虑缓存数据的更新和删除策略，考虑数据一致性如何维护，本小节主要针对以上的问题来分析不同实现方案的优缺点。

5.1 缓存命中率

缓存命中率不仅是系统性能的一个侧面指标，也是优化缓存使用方案的一个重要依据。缓存命中率=请求命中数/请求总数。接下来的若干缓存使用策略所围绕的核心考量点就是在保证系统稳定性的同时，旨在提升缓存命中率。

5.2 缓存更新策略

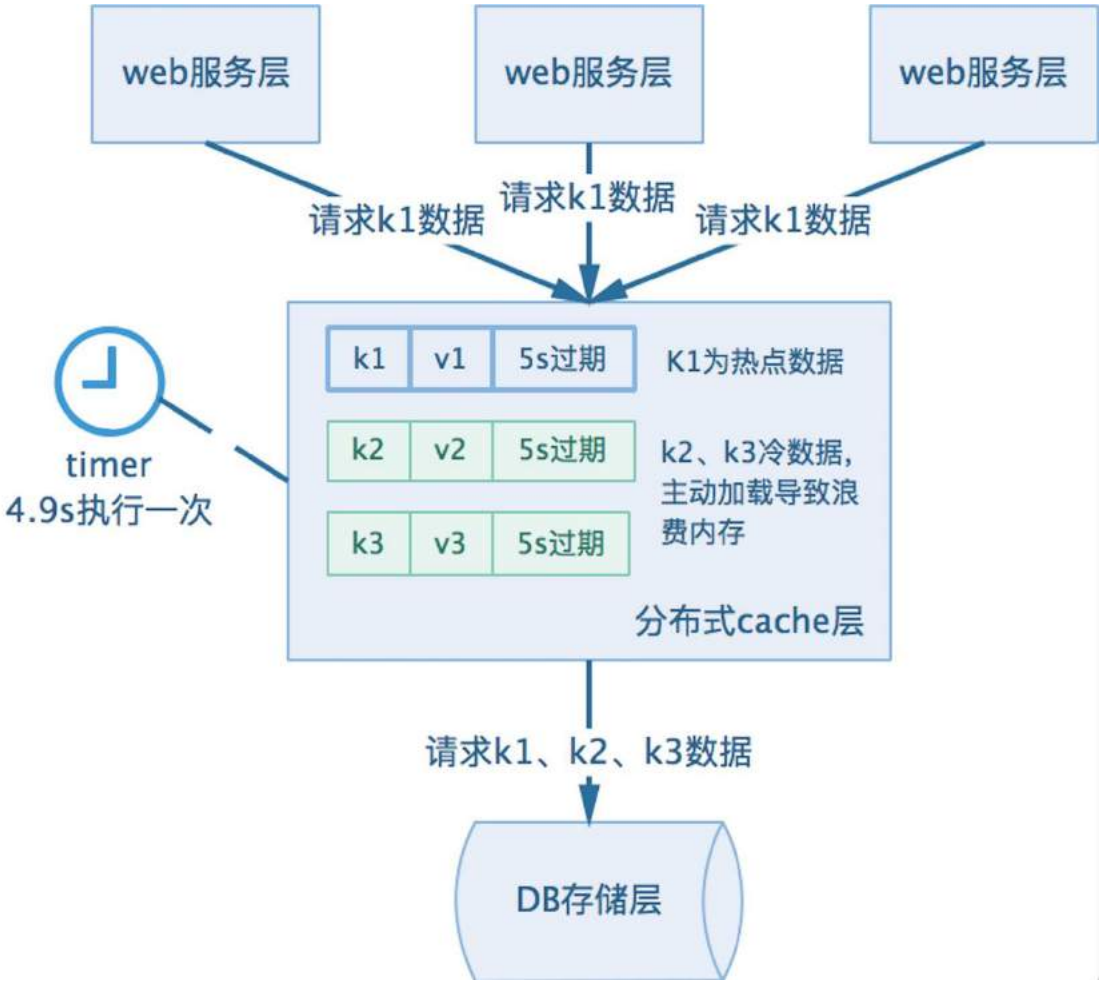
5.2.1 主动请求 DB 数据，更新缓存

通过在集群中的每台机器都部署一套定时任务，每隔一段时间就主动向数据库 DB 请求最新数据，然后更新缓存。这样做的好处是可以避免缓存击穿的风险，在缓存失效前就主动请求加载 DB 数据，完成缓存数据更新的无缝连接。

但这样做也增加了机器的 CPU 和内存的占用率，因为即使有若干 Key 的缓存始终不被访问，可还是会被主动加载加载到内存中。也就是说，提高了业务抗风险能力，但对 CPU 和内存资源并不友好。

详情可参见下图，分布式缓存中存储着 DB 中的数据，每隔 4.9s 就会有定时任务执行去更新缓存，而缓存数据失效时间为 5s，从而保证缓存中的数据永远存在，避免缓存击穿的风

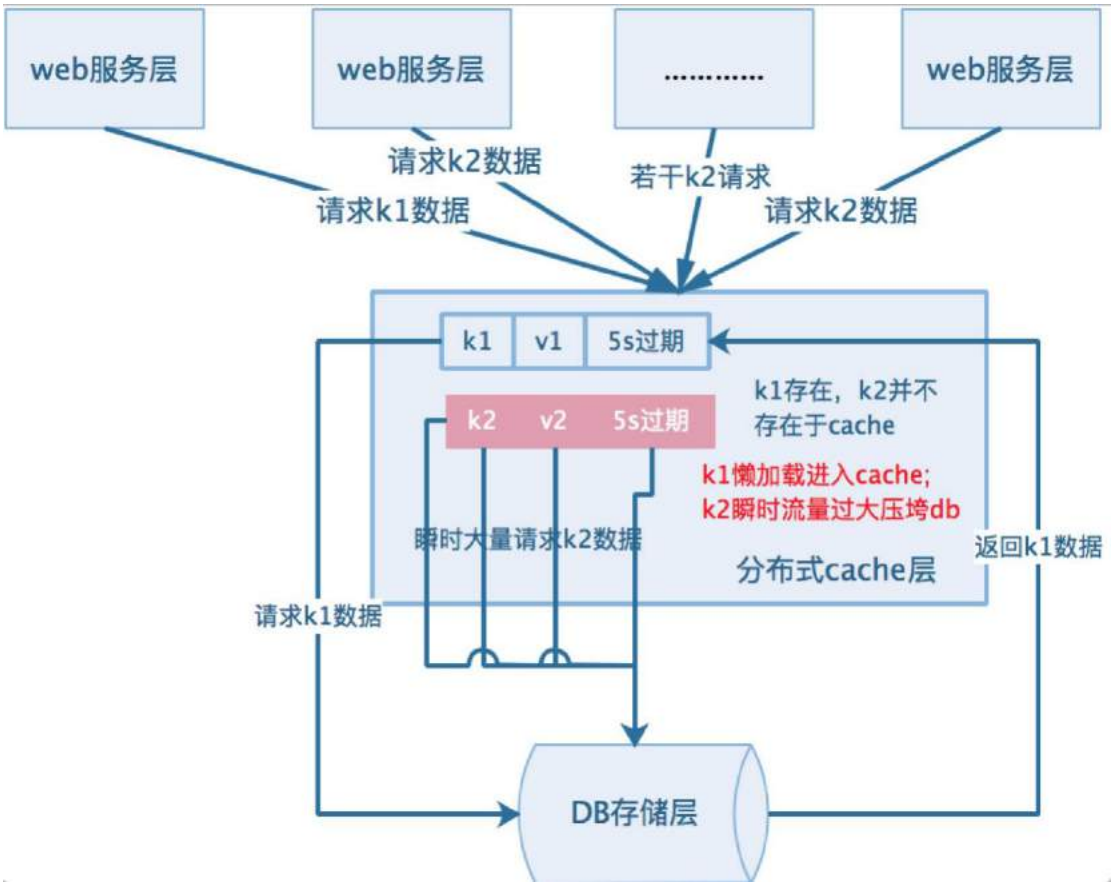
险。但对于 Web 请求来说，只会访问 k1 的缓存数据，也即对于 k2 和 k3 数据来说，是无效缓存。



5.2.2 被动请求 DB 数据，更新缓存

当有请求到达且发现缓存没数据时，就向 DB 请求最新数据并更新缓存。这种方案完全可以看做是方案一的互斥方案，它解决的是机器 CPU 和内存浪费的问题，内存中存储的数据始终是有用的，但却无法避免缓存失效的瞬间又突然流量峰值带来的缓存击穿问题，在业务上会有一定的风险。

详情见下图，缓存不会主动加载数据，而是根据 Web 请求懒加载数据。对于请求 k1 数据来说，发现缓存没有对应数据，到 DB 查询，然后放入 Cache，这是常规流程；但如果有突发流量，大量请求同时访问 k2 数据，但 Cache 中没有数据时，请求就会同时落到 DB 上，可能压垮数据库。



5.3 缓存过期策略

5.3.1 依赖时间的过期策略

- 定时删除

对于需要删除的每个 Key 都配备一个定时器，元素超时时间一到就删除元素，释放元素占用的内存，同时释放定时器自身资源。其优点是元素的删除很及时，但缺点也很明显，比如为每个 Key 配备定时器肯定会消耗 CPU 和内存资源，严重影响性能。这种策略只适合在小数据量且对过期时间又严格要求的场景能使用，一般生产环境都不会使用。

- 惰性删除

元素过期后并不会立马删除，而是等到该元素的下一次操作（如：访问、更新等）才会判断是否过期，执行过期删除操作。这样的好处是节约 CPU 资源，因为只有当元素真的过期了，才会将其删除，而不用单独管理元素的生命周期。但其对内存不友好，因为如果若干已经过期的元素一直不被访问的话，那就会一直占用内存，造成内存泄漏。

- 定期删除

以上两种元素删除策略各有优缺点，无非是对 CPU 友好，还是对内存友好。为了结合两者的优点，一方面减少了元素定时器的配备，只使用一个定时器来统一扫描过期元素；另一

方面加速了判断元素过期的时间间隔，不是被动等待检测过期，而是间隔一段时间就主动执行元素过期检测任务。正是由于以上的改进点，此方案是元素过期检测的惯常手段。

我们假设一个场景，为了保护用户隐私，通常在用户电话和商家电话之间，会使用一个虚拟电话作为沟通的桥梁。业务使用中，往往同一个虚拟号码在一定时间内是可以对相同的用户和商家建立连接的，而当超出这个时间后，这个虚拟号码就不再维护映射关系了。

虚拟电话号码的资源是有限的，自然会想到创建一个虚拟号码资源池，管理虚拟号码的创建和释放。比如规定一个虚拟号码维持的关系每次能使用 15 分钟，那么过期后要释放虚拟号码，我们有什么方案呢？

A. 方案一：全量数据扫描，依次遍历判断过期时间

字段名称	字段含义
vr_phone	虚拟号码
expire_seconds	相对超时时间
create_time	虚拟号码创建时间
other_fields	其它扩展字段

对于 DB 中存储的以上内容，每天记录都存储着虚拟号码的创建时间，以及经过 expire_seconds 就会删除此记录。那么需要配备一个定时任务扫描表中的所有记录，再判断 current_time - create_time >expire_seconds，才会删除记录。

如果数据量很大的情况，就会导致数据删除延迟时间很长，这并不是可取的方案。那是否有方案能直接获取到需要过期的 vr_phone，然后批量过期来解决上述痛点呢？来看看方案二吧。

B. 方案二：存储绝对过期时间+BTree 索引，批量获取过期的 vr_phone 列表

字段名称	字段含义
vr_phone	虚拟号码
expire_timestamp	删除记录时间戳
other_fields	其它扩展字段

将相对过期时间 expire_seconds 改为记录过期的时间戳 expire_timestamp，同时将其添加 BTree 索引提高检索效率。仍然使用一个定时器，在获取待删除 vr_phone 列表时只需要 select vr_phone from table where now()>=expire_timestamp 即可。

对于空间复杂度增加了一个 BTree 数据结构，而基于 BTree 来考虑时间复杂度的话，对于元素的新增、修改、删除、查询的平均时间复杂度都是 O(logN)。

此方案已经能满足业务使用需求了，那是否还有性能更好的方案呢？

d)单层定时轮算法

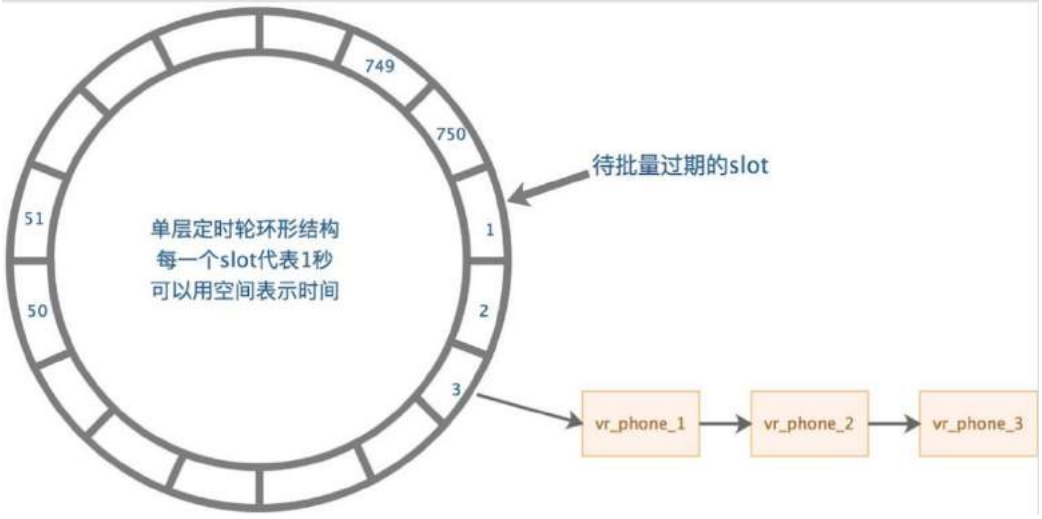
我们继续讨论上面的案例，寻找更优的解题思路。下表是 DB 存储元素：

字段名称	字段含义
vr_phone	虚拟号码
other_fields	其它扩展字段

此时 DB 中不再存储和过期时间相关的数据，而专注于业务数据本身。对于过期的功能我们交给单层定时轮来解决。其本质是一个环形数组，数组每一格代表 1 秒，每次新加入的元素放在游标的上一格，而游标所指向的位置就是需要过期的 vr_phone 列表。

执行过程：

- 初始化：启动一个 timer，每隔 1s，在上述环形队列中移动一格，1->2->3...->29->750->1...有一个指针来标识有待过期的 slot 数据。
- 新增数据：当有一个新的 vr_phone 创建时，存储到指针的上一个 slot 中。对于有 slot 冲突的场景，可以利用链表解决冲突，也可以利用数组解决冲突。链表和数组的考量标准还是依赖于单个 slot 的数据长度，如果数据过长，那么存储的数组会很长，则需要很大的内存空间才能满足，无法利用内存碎片的空间。
- 过期数据：指针每隔 1 秒移动一个 slot，那么指针指向的 slot 就是需要过期的数据，因为新增的数据在环形 slot 转完一圈后，才会被指向到。



这样一种算法结构，将时间和空间巧妙地结合在了一起。新增元素的时间复杂度为 O(1)，直接插入待批量过期的 slot 的上一个位置即可；获取待删除元素列表时间复杂度也是 O(1)，就是待批量过期的 slot 位置。流行框架 Netty、Kafka 都有定时轮的影子。

当然，单层定时轮只适用于固定时间过期的场景，如果需要管理不同过期时间的元素，那么可以参考"多层定时轮算法"，其实就是模拟现实世界的时针、分针、秒针的概念，建立多个单层定时轮，采用进位和退位的思想来管理元素的过期时间。

以上各种元素过期策略各有优缺点，可以根据业务的诉求来取舍。比如 Memcache 只是用了惰性删除，而 Redis 则同时使用了惰性删除和定期删除以结合二者的优点。

5.3.2 依赖空间的过期策略

此处只探讨最经典的三种策略 FIFO、LRU、LFU 的原理及实现方案，对于其它改进算法，感兴趣的同学可以自行查找。

a) FIFO：先进先出，当空间不足时，先进入的元素将会被移除。此方案并没有考虑元素的使用特性，可能最近频繁访问的一个元素会被移除，从而降低了缓存命中率。实现：基于 LinkedHashMap 的钩子函数实现 FIFOMap。

```
1.  // 链表头部是最近最少被访问的元素，需要被删除
2.  public class FIFOMap<K, V> extends LinkedHashMap<K, V> {
3.      private int maxSize;
4.
5.      //LinkedHashMap 每次插入数据，默认都是链表 tail；当 accessOrder=false，元素被访问不会移动位置
6.      public FIFOMap(int maxSize) {
7.          super(maxSize, 0.75f, false);
8.          this.maxSize = maxSize;
9.      }
10.
11.     //每次 put 和 putAll 新增元素的时候都会触发判断;当下面函数=true 时，就删除链表 head 元素
12.     @Override
13.     protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
14.         return size() > maxSize;
15.     }
16. }
```

b) LRU：最近最少使用算法，当下多次被访问的数据在以后被访问的概率会很大，因此保留最近访问的元素，提高命中率。可以应对流量突发峰值，因为存储的池子大小是固定的，因此内存占用不可能过多。但也有缺点：如果一个元素访问存在间歇规律，1 分钟前访问 1 万次，后面 30 秒无访问，然后再访问一万次，这样就会导致被删除，降低了命中率。实现：基于 LinkedHashMap 的钩子函数实现 LRUHashMap。

```
1.  // 链表头部是最近最少被访问的元素，需要被删除
2.  public class LRUMap<K, V> extends LinkedHashMap<K, V> {
3.      private int maxSize;
4.
5.      //LinkedHashMap 每次插入数据，默认都是链表 tail；当 accessOrder=true 时，被访
    问的元素也会放到链表 tail
6.      public LRUMap(int maxSize) {
7.          super(maxSize, 0.75f, true);
8.          this.maxSize = maxSize;
9.      }
10.
11.     //每次 put 和 putAll 新增元素的时候都会触发判断;当下面函数=true 时，就删除链表
    head 元素
12.     @Override
13.     protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
14.         return size() >= maxSize;
15.     }
16. }
```

c) LFU：最近最少频率使用，根据数据的历史访问频率来淘汰数据，其核心思想是"如果数据过去被访问多次，那么将来被访问的频率也更高"。这种算法针对 LRU 的缺点进行了优化，记录了元素访问的总次数，选出访问次数最小的元素进行删除。原本的 LFU 算法要求记录所有元素的访问次数，但考虑到内存成本，改进后的 LFU 是在有限队列中进行淘汰。

实现：Redis 的优先级队列 Zset 实现，Zset 存储元素的数量固定，Value 是访问次数，超过 size 就删除访问次数最小的即可。但这种删除策略对于有时效性的数据却并不合适，对于排行榜类的数据，如果某个历史剧点击量特别高，那么就始终不会被淘汰，新剧就没有展示的机会。改进方案，可以将 Value 存储为入库时间戳+访问次数的值，这样随着时间流逝，历史老剧就可能被淘汰。

6.其他影响命中率的因素

6.1 缓存穿透

对于数据库中本就不存在的值，缓存中肯定也不会存在，此类数据的查询一定会落到 DB 上。为了减少 DB 访问压力，我们期望将这些数据都可以在缓存中 cover 住，以下是两种解法。

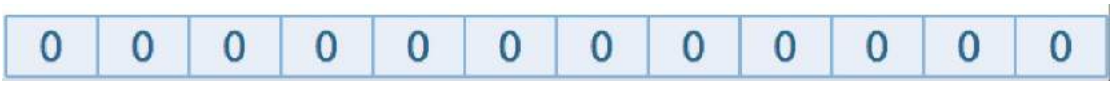
- 解法一：缓存 null 值：

该方法对于元素是否存在于 DB 有精准的判断，可如果存在海量 null 值的数据，则会对内存过度占用。

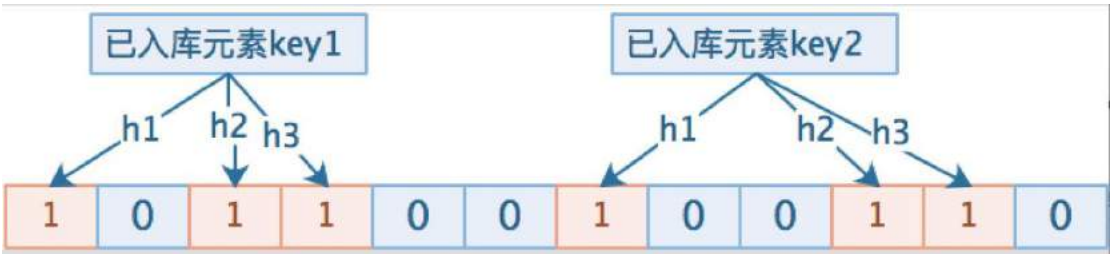
- 解法二：布隆过滤：

使用场景是海量数据，且不要求精准判断和过滤数据。其思路是借助 Hash 和 bit 位思想，将 Key 值映射成若干 Hash 值存储到 bit 数组中。

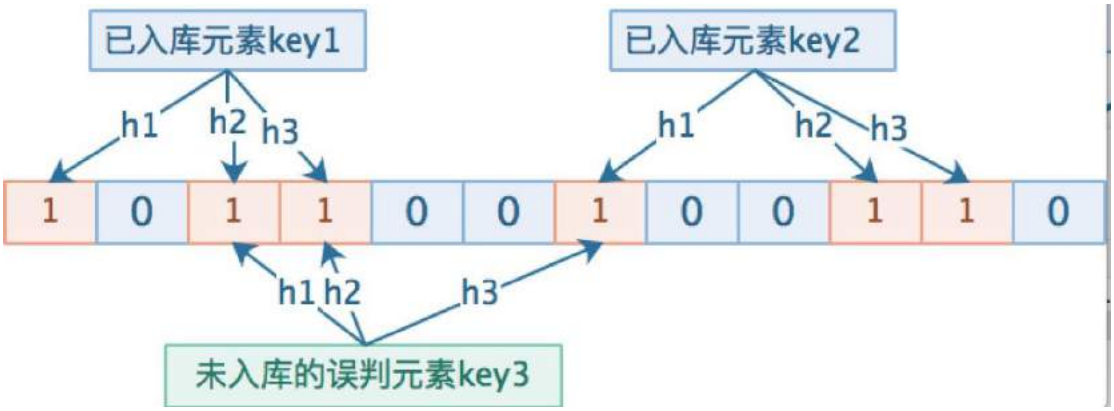
A. 初始创建 bit 位数组，所有 bit 位全都为 0。



B. 新增元素时，将元素的 Key 根据预设的若干 Hash 函数解析成若干整数，然后定位到 bit 位数组中，将对应的 bit 位都改为 1。



C. 判断元素是否存在，也是将元素的 Key 根据 Hash 函数解析成整数，查询若干 bit 位的值。只要有一个 bit 位是 0，那么这个 Key 肯定是新元素，不存在；如果所有 bit 位全都是 1，那么这个 Key 很大概率是已经存在的元素，但也有极小的概率是 Key3 经过若干 Hash 函数定位到 bit 数组后都是 Hash 冲突的，可能造成误判。



6.2 缓存击穿

缓存中原本一批数据有值，但恰好都同时过期了，此时有大量请求过来就都会落到 DB 上。避免这种风险也有两种解法。

- 解法一：随机缓存失效时间：

对缓存中不同的 Key 设置不同的缓存失效时间，避免缓存同时失效带来大量请求都落到 DB 上的情况。

- 解法二：主动加载更新缓存策略，替代缓存过期删除策略：

在缓存失效之前就主动到 DB 中加载最新的数据放到缓存中，从而避免大量请求落到 DB 的情况。

6.3 缓存雪崩

大量缓存同时过期，或者缓存中间件不可用，导致大量请求落到 DB，系统停止响应。解法是对缓存设置随机失效时间，同时增加缓存中间件健康度监测。

7.保证业务数据一致性的策略

在分析了影响缓存命中率的若干策略和方案后，我们结合实际开发诉求，来分析下缓存是如何降低 DB 的访问压力，以及 DB 和缓存中业务数据的一致性如何保证？

维护数据一致性常用的方案有两种：先操作 DB，再操作 Cache；先操作 Cache，再操作 DB。而以上两步操作都期望是全部成功，才能保证操作是原子性的。如果不依赖事务，那么对数据怎样操作才能保证即使流程异常中断，对业务影响也是最小呢？

7.1 对于读取操作

因为只是读取，不涉及数据修改，因此先读缓存，Cache miss 后，读 DB 数据，然后 set cache 就足够通用。

7.2 对于写入操作

7.2.1 先操作 DB，再操作(delete/update)缓存

当 DB 数据操作成功，但缓存数据（不论是 delete 还是 update）操作失败，就会导致在未来一段时间内，缓存中的数据都是历史旧数据，并没有保证操作的原子性，无法接受。

7.2.2 先操作(delete/update)缓存，再操作 DB

- 第一种方案：当 update 缓存成功，但操作 DB 失败，虽然缓存中的数据是最新的了，但这个最新的数据最终并没有更新到 DB 中，当缓存失效后，还是会从 DB 中读取到旧的数据，这样就会导致上下游依赖的数据出现错误，无法接受。

- 第二种方案：先 delete 缓存，再操作 DB 数据，我们详细讨论下这种方案：

如果 delete 就失败了，整体操作失败，相当于事务回滚；

如果 delete 成功，但 DB 操作失败，此时会引起一次 cache miss，紧接着还是会从 DB 加载旧数据，相当于整体无操作，事务回滚，代价只是一次 cache miss；

如果 delete 成功，且 DB 操作成功，那么整体成功。

结论：先 delete 缓存，再操作 DB，能尽可能达到两步处理的原子性效果，即使流程中断对业务影响也是最小的。

8.小结

对于缓存的使用没有绝对的黄金标准，都是根据业务的使用场景来决定什么缓存框架或者缓存策略是最适合的。但对于通用的业务场景来说，以下的缓存框架选择方法应该可以满足大部分场景。

对于本地缓存，如果缓存的数量是可估计的，且不会变化的，那么可使用 JDK 自带的 HashMap 或 ConcurrentHashMap 来存储。

对于有按时间过期、自动刷新需求的本地缓存可以使用 Caffeine。

对于分布式缓存且要求有丰富数据结构的，推荐使用 Redis。

招聘

高德地图数据序列化的探索与实践

作者：宋峰

1.导读

高德既有上半身（导航服务、交通服务、步导服务、渲染服务、离线包等），为亿级用户提供基础的地图展示、搜索、导航、路况服务；又有下半身（数据采集、制作），源源不断地采集最新的数据，为用户提供最新最准的数据。

下半身采集的数据，需要经过处理，序列化为二进制数据，输送给上半身的各个应用。同时，上半身的各个服务间也需要数据传输，这些都需要将地图数据序列化为二进制数据。

地图数据包括道路、POI、水系、绿地、建筑等，可抽象为三类：点、线、面数据。再进一步抽象，可分为两类：几何数据和属性数据。

本文将分享高德技术团队在地图数据序列化领域的探索和实践。

2.序列化的关键因素

序列化主要考虑以下几个因素：

- 数据量：

针对客户端 App，流量消耗情况非常重要。高德地图对数据的新鲜度要求非常高，如京港澳高速开通，需要实时上线，让用户第一时间用最新的数据进行导航。数据更新快，会导致客户端缓存很快失效，需要实时从服务端请求最新数据。这就要求地图数据要尽量小，以减少服务带宽占用、用户流量消耗。

针对离线数据，离线包的大小也非常重要。地图数据全国离线包动辄上 G，对用户流量、下载速度和手机存储空间占用都是不小的消耗。

- 扩展性：

客户端 App 一方面无法控制用户软件更新，一方面又需要不停迭代为用户提供新功能。这就要求数据需要做到向前兼容和向后兼容。

各服务间的数据传递最好也做到数据兼容。如无法兼容，则需要各服务同步修改、同时上线，二者强耦合。强耦合就带来开发以及沟通的复杂性。

- 编解码效率：

毋庸置疑，二进制数据的解码效率，对服务和客户端来说，都是很重要的指标。

3.序列化方案选择

3.1 开源序列化库的优势与弊端

序列化成本最低的选择是使用开源的序列化库。目前流行的开源序列化库有 protobuf，flatbuffers 等。它们的共同特点是都提供一种数据描述语言，只需定义好协议，即可自动生成编解码器，编码器负责将内存数据序列化为二进制数据，解码器负责将二进制数据反序列化到内存。

使用第三方库的好处是无需自己设计数据规格，自己实现编解码器，将这些工作交给第三方库，自己专注于业务逻辑即可。缺点是无法根据业务特点，充分定制化，设计出最适合地图业务的数据编码规格。

开源序列化库在扩展性方面都做的很好，支持协议的升级，并且在升级的同时保证向前兼容和向后兼容。flatbuffers 通过其特有的数据结构 table 实现，protobuf 通过 optional 关键字来实现。

数据量和编解码效率是相互矛盾的两个方面。protobuf 更倾向于降低数据量，针对数值类型的存储进行优化，使用 varint 存储整形，使用 zigzag 编码存储负数；可节省整形数据所占存储空间。

图 1 展示了使用 varint 表示整数 300 的原理，zigzag 原理如图 2 所示。flatbuffers 更倾向于使解码效率最优。它通过 mmap，zero-copy，random-access 等手段使得使用方解码、以及读取解码后的属性值都效率极高。

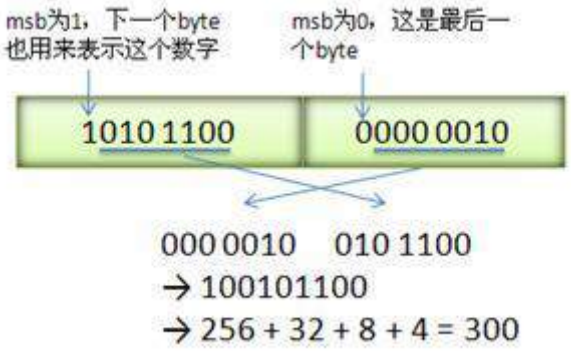


图 1 varint 表示

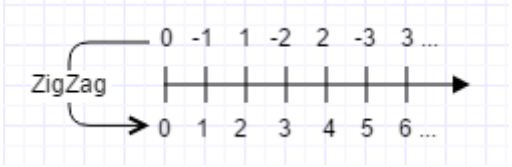


图 2 zigzag 编码

3.2 自定义序列化规格

由于地图数据有其独有特点，且全国数据量巨大，开源库序列化后的数据量大小都无法满足数据传输的要求。

所以，我们需要在保留开源库优点的前提下，设计自己的序列化规格。保证扩展性的同时，做到数据量最小，同时尽量优化解码效率。

地图数据功能扩展有两类：扩展一种新的要素，或者针对现有要素扩展其属性。如果仅考虑扩展性，flatbuffers 和 protobuf 都可满足要求。但地图数据使用方在不同场景下，希望可以只解析部分数据即可工作，即需要支持随机读取功能。

所以综合扩展性和随机读取这两类需求，我们针对性地设计了两种支持协议扩展的模式：章节式存储、可变属性。通过这两种模式，可支持任意的数据扩展，并且保证向前兼容和向后兼容。同时，这两种模式还可以带来解码效率的优化，使用方对其不关心的章节或者可变属性，可直接跳过，提高解码效率。

针对地图数据序列化后数据量的优化，我们无须拘泥于一种压缩形式，多种压缩方法综合使用，才能达到最好的压缩效果。shannon 的信息论告诉我们，对信息的先验知识越多，我们就可以把信息压缩的越小。

换句话说，如果压缩算法的设计目标不是任意的数据源，而是基本属性已知的特种数据，压缩的效果会进一步提高。这提醒我们，在使用通用压缩算法之余，还可以针对地图数据开发其特有的数据压缩算法。

我们使用的压缩手段有：

- 针对整形使用 varint 编码。
- 针对负数使用 zigzag 编码。
- 将 double 转为整形存储。
- 针对几何数据在不同比例尺做简化，经典的算法有 Douglas-Peucker，Li-Openshaw 等。
- 针对几何数据做曲线拟合，如贝塞尔曲线拟合，CLOTHOID 曲线拟合等。（如图 3 所示）。
- 针对连续几何数据进行差值存储，减少每个坐标点所需 bit 数。
- 使用指定 bit 位存储数值类型，而非整数个字节。

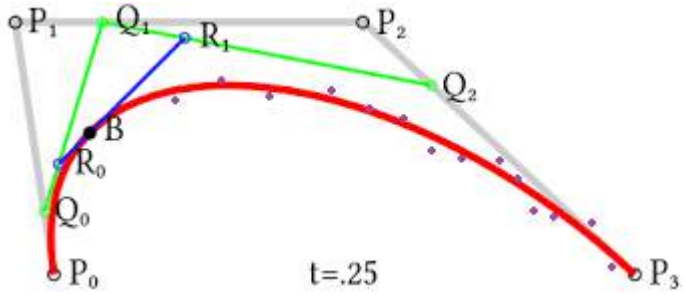


图 3 贝塞尔曲线拟合

除了上述针对地图数据特有的压缩算法外，我们还可使用通用的无损压缩算法。目前通用的压缩库使用的算法基本分为两类：基于字典的压缩算法，基于统计的压缩算法。基于字典的压缩算法代表是 lz 系列算法（lz77, lz78, lzw，lzss），基于统计的压缩算法代表是 haffman 编码和算术编码等。

二者的基本原理都是找到输入串中冗余的信息，用更少的字节来表示冗余信息，以达到压缩的效果。基于字典的压缩算法以类似查字典的方法进行编码，它的基本原理是使用索引来表示字典中出现过的字符串。基于统计的压缩算法实质是统计字符的出现频率来对字符本身进行重新编码，属于熵编码类，与原始数据的排列次序无关而与其出现频率有关。

地图数据中的几何信息经过我们针对地图数据特有的压缩算法处理后，冗余信息已经很少了，所以使用通用压缩算法的效果不明显。针对属性信息，通用压缩算法有一定效果。所以我们的序列化协议支持对不同类型数据选择性地使用通用无损压缩算法。

综上，我们设计的序列化规格和 protobuf，flatbuffers 综合对比如下：

	Protobuf	Flatbuffers	自定义规格
扩展性	支持	支持	支持
数据量	中	高	低
解码效率	中	高	低
随机读取	不支持	支持	支持
Zero-copy	不支持	支持	不支持

4.小结

可以看出，各种序列化方法各有优缺点。扩展性方面，三者都做的很好。如果注重解码效率，则 flatbuffers 最优；如果注重数据量，则我们自定义的序列化规格最优。当然，没有最好，只有最适合。选择时，需要根据业务特点来选择适合自己的序列化规格。

招聘

高德地图在线引擎中心团队长期招聘机器学习算法、C++、Java 资深工程师/技术专家/高级专家，职位地点：北京，欢迎有兴趣的同学投递简历到 Lenka@alibaba-inc.com

质量篇

高德全链路压测平台 TestPG 的架构与实践

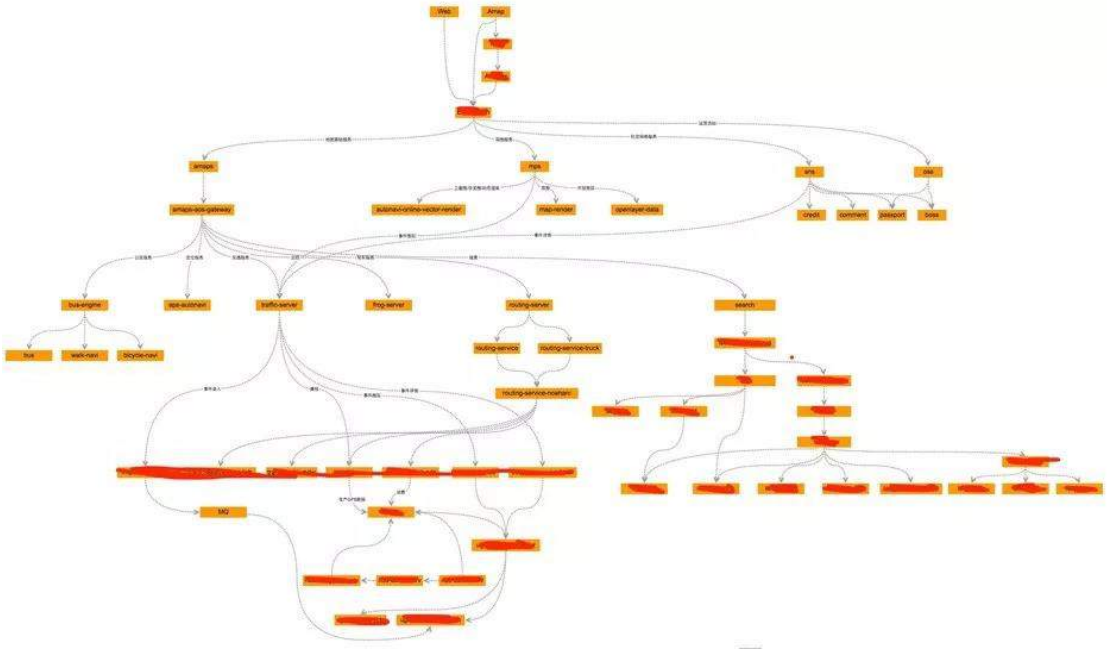
作者：才溅

1.导读

2018 年十一当天，高德 DAU 突破一个亿，不断增长的日活带来喜悦的同时，也给支撑高德业务的技术人带来了挑战。如何保障系统的稳定性，如何保证系统能持续的为用户提供可靠的服务？是所有高德技术人面临的问题，也是需要大家一起解决的问题。

2.高德业务规模

支撑一亿 DAU 的高德服务是什么体量？可能每个人的答案都不相同，这里从基础设施的角度给大家做个简单的介绍，我们有数千个线上应用，分别部署在全国各地多个机房中的数万台机器上。



这张图是高德业务核心链路的架构，从图中可以看出高德业务具有相当高的复杂性。当然，真实系统远远要比图表示的复杂，如果用这张图来代表高德整体业务形态，无异于管中窥豹，太过于片面。

对于如此大规模，高复杂度的系统，如何保障系统的稳定性，是高德技术人长期面临和解决的问题。

3.保障稳定性的手段

如何保障系统稳定性是几乎所有互联网企业都需要面对的问题。通常来讲，有五种手段来从理论上保障系统的稳定性，分别是：

- **容量规划**：根据以往业务的流量，估算出未来（通常是即将来临的大促，节假日）的流量。以整体流量为基础，估算出每个子系统需要满足的容量大小。然后根据子系统的容量来计算出需要的资源数量，对系统进行适当的扩容。计算方式可以简单的表示为如下公式：

机器数量 = 预估容量 / 单机能力 + Buffer （一定数量的冗余）

- **流量控制**：系统需要防止流量超过设计的容量，对超出设计的流量进行限流。各业务也需要对超出子系统服务能力的流量进行限流，对超负荷的服务进行降级。

- **灾备**：一旦系统发生灾难性故障，需要将流量切换到容灾机房，避免对大量用户造成损失。

- **监控**：对服务进行全方面的监控，实时掌控系统的状态，对系统中出现的问题及时预警，做到早发现，早治理。

- **预案演练**：对系统可能面临的问题要进行全面的预演，结合断网，断电等等灾难模拟的手段来检验系统在灾难面前的表现。

有了稳定性保障的五大法宝，我们是否就可以高枕无忧了呢？答案是令人遗憾的，这里有两个残酷的现实例子，告诫我们不要太乐观。

多年前的某年春节前夕，我们对高德核心链路进行了压测，压测设计的流量要高于预估的春节流量，系统在当时表现良好，各项指标都满足要求。可是春节期间，服务因某种原因发出告警，而此刻线上流量的水位并没有超过我们的预期。

还有一次在某年五一期间，该服务再次发出预警，而且和春节的那次预警的原因不一样。

我们的稳定性保障手段是基于对于系统的认知来实现的，而认知往往是真实世界在头脑中的映射，是一种模型，或是真实系统的快照。系统的真实状态往往和我们观测到的不太一致，基于观测到的模型对系统进行的测量也往往会不够准确，对于系统，我们要保持敬畏。对系统进行不厌其烦的模拟，无限的接近真实系统的状态，是永恒不变的追求。

上述稳定性保障的工作，只能在理论上保证系统的抗压能力，但是不能确定在真实流量到来的时候，系统的表现如何！

因此，我们需要演习，需要让真实的流量提前到来！

4.全链路压测

如何让真实的流量提前到来？我们需要借助全链路压测，那么什么是全链路压测呢？

我的理解是：把全链路压测拆分为两个部分来看。一是全链路，一是压测，分别来理解：

- **全链路**：分为两层意思；一是自顶向下，一个请求在系统中经过的完整路径。二是一系列动作的集合，比如从用户登录，到浏览商品，到选择商品，到加入购物车，到支付等等这整个环节。对于高德业务而言，我们关注的是第一种全链路。

- **压测**：通过对海量用户行为模拟的方式，对系统逐步施压，用于检验系统的稳定性。

集团的战友们把全链路压测比作 "终极武器"，非常的形象。既然是 "终极武器"，那就需要有足够的威慑力，对于高德来说，目标是：提供真实的流量，在真实的环境中来检验系统的稳定性。

这里面包含三个关键点：

- **真实的流量**：流量的量级和特征贴近真实的世界。

- **真实的环境**：直接在线上环境进行。

- **提前进行**：在流量洪峰到来之前。

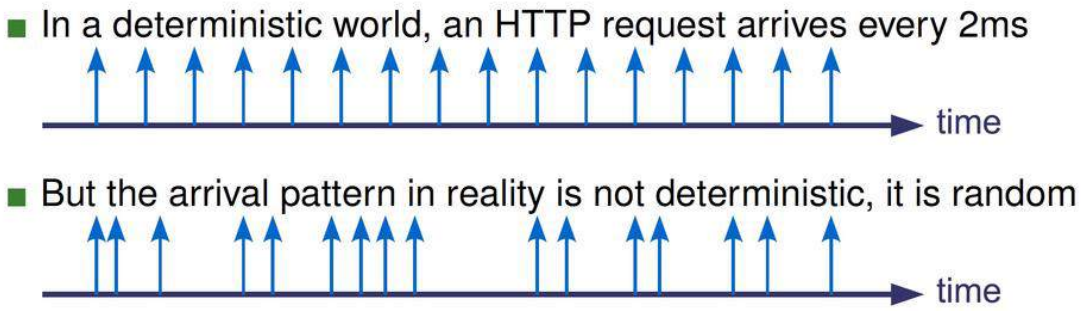
做到这三点，才能称得上是一次真正意义上的演习，才可以叫做全链路压测。

5.高德全链路压测面临的挑战

高德全链路压测面临众多方面的挑战，除了分布式系统固有的挑战外，高德全链路压测还面临着业务特殊性的挑战。

5.1 分布式系统的特性

5.1.1 不确定性



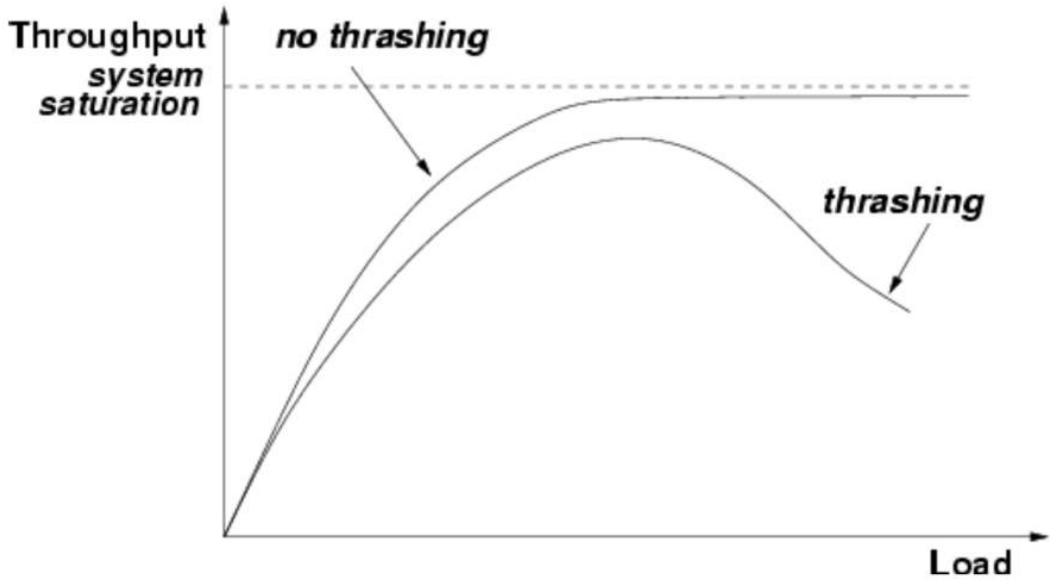
我们认为的流量有序的，而真实的流量却是随机的，不确定的。

5.1.2 抖动性

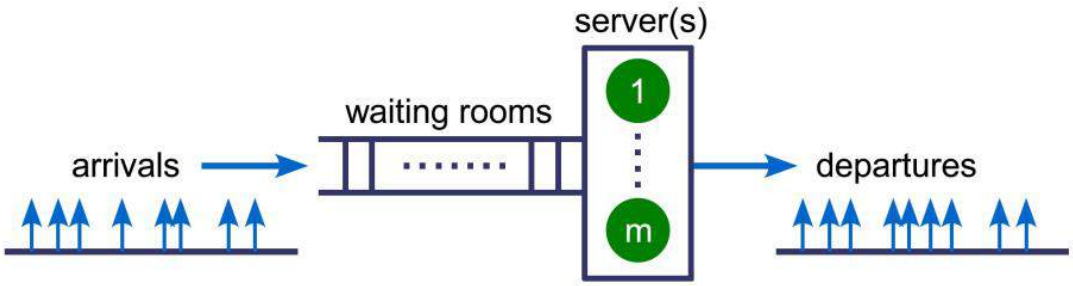
在理论的世界里面，吞吐量是请求量的函数，可以表示为如下的公式：

Throughput=f(load)

如果没有其他因子的干扰，在系统达到饱和之前，吞吐量和请求量的关系应该是线性的。而现实世界里面，这种理论模型几乎是不可能出现的。为什么？因为抖动。为什么会出现抖动？因为网络，磁盘等等的不确定性。



5.1.3 排队系统的特性



我们的业务可以简单的抽象成为一个排队系统，请求从左边随机的进入系统，等待被处理，处理完成之后，从右边离开队列。在系统未达到饱和状态时，系统可以很好的处理用户的请求，而一旦队列接近饱和，那么队列的长度可能会显著的增加，而且请求的平均响应时间也会出现增加，甚至会出现部分请求超时的情况。对于一个理论上能处理 1000q/s 的系统，在不同流量的情况下，可能的状态如下（特定系统的测量结果）：

Normalised arrival rate	Average input	Average output	Average queue
0.5	500	500	43
0.95	950	907	1859
0.99	990	942	2583

从图中可以看出：

- 请求达到率增加 2 倍，队列的长度会增加约 60 倍。
- 当系统接近饱和时，请求端极小的变化将会对系统造成很大的影响。请求达到率从 0.95 ~ 0.99，队列的长度将增加 40%。

排队系统的特征是：系统会在接近饱和时出现拥堵，拥堵会导致服务的时间变长，而这反过来又会加重拥堵的程度。随着情况的恶化，系统的资源（cpu，mem）最终会被耗尽。拥堵和服务质量下降表现为正相关性，这种特性会随着时间急剧的恶化，最终拖垮系统，出现故障。

5.2 高德业务特点

出行业务有其特有的特殊性，会受到诸多因素的影响。具体到高德业务而言，系统的行为会受到区域，地形，路况，路网密度，季节，天气，政府活动等等因素的影响。

以驾车导航为例，导航系统会受到如下因素的影响：

- **区域**：不同的区域经济发展水平不一致，人们选择出行的交通工具也会不一样，经济发达地区的人们汽车拥有率会更高，使用汽车出行的频率也会更高。

- **地形**：山区，城市繁华地区会因 gps 信号遮挡导致定位不准确，可能被系统认为偏航，从而引发路径重新规划。

- **路况**：事故，拥堵，施工，限行，管制 这些路况都对导航服务造成影响。

- **路网密度**：导航算法所要求的计算资源和路网的密度有很强的关联，路网越密，路径规划所消耗的 cpu 资源，内存资源就会越大，同时计算的时间也会越长。

- **距离**：路径规划的距离越长，导航算法对计算资源的要求就越高。

- **季节&天气**：人们的出行行为和季节也会呈现相关性，如 五一，端午 人们可能集中前往景区旅游。十一，春节 人们可能会集中返乡。这样在导航行为上就会出现导航距离分布不同的情况，不同的导航距离对服务的要求会不一样，越长距离的导航对服务资源的要求越高。

- **政府活动**：交通管制，修路，限行等等。

对于高德而言不能单纯的通过放大流量来对系统进行压测，在流量构造阶段我们需要考虑到流量的特征，考虑到所有影响服务行为的因素。

6.高德全链路压测平台的自建起因

身在阿里，说起全链路压测，首先想到的肯定是大名鼎鼎的 Amazon 平台，Amazon 诞生于 2013 年，自 2013 年起，Amazon 一直作为淘宝、天猫稳定性保障体系中神一般的存在。经过多年的发展和演进，Amazon 平台已经日臻稳定和成熟，在施压能力，链路治理，数据构造，用户模拟方面已经做到极致。

所以，在落地高德全链路压测的时候，首先考虑的就是借助 Amazon 平台。经过充分的调研和部分项目的试用，我们发现 Amazon 平台并不能满足我们的要求。

Amazon 平台诞生于淘宝，作为淘宝，天猫稳定性保障体系中重要的成员。Amazon 追求的是超高的施压能力和极强的平台稳定性。另外，淘宝的双 11，双 12 全链路压测是多团队合作的项目，一次全链路压测可能需要数百人的参与，对于这种超大规模的全链路压测项目，成败的关键在于团队的合作。平台需要搞定的是人无法搞定的事情，对于 Amazon 来讲，要做的就是把事情做到极致。
高德的全链路压测流量的要求远远不及淘宝的全链路压测，并且通常全链路压测的周期都不会太长（通常在 2 周左右，这个时间还需要缩减），所以我们比较关注压测成本的问题，例如压测数据的构造成本，压测资源申请的成本，错误排查成本，以及便捷性方面的问题，例如压测过程的可视化，压测报告等。

另外，高德的日常环境的压测需求比较旺盛，除了全链路压测外，我们还需要借助别的平

台（如 PAP，PAS）来满足日常的压测需求。

从成本收益的角度出发，最终我们选择自研压测平台来满足高德的全链路压测需求，以及日常的压测需求。

7.高德压测平台的自建思路

如何打造一款全新的压测平台？

回答这个问题，先要回答平台的目的是什么。高德压测平台的目的是什么？一定是为了解决业务的问题！结合上文对全链路压测的描述，对高德业务特点的描述，我们建设压测平台就需要回答这几个问题：

- 如何保证场景的真实性？
- 线上压测，如何保证压测流量不影响线上用户，如果保证压测数据不污染线上数据？
- 如何构建超高流量？
- 如何降低使用成本？
- 如何降低资源成本？

7.1 如何保证场景的真实性

压测要保证真实，需要在压测场景上做到全覆盖，需要从协议支持和用户行为两个方面来满足场景的真实性。

协议支持：
对于高德服务而言，不同的用户场景使用到的通信协议不一样，例如 PC 端主要是 http 协议，而移动端则是 accs 协议。因此全链路压测的场景设计上首先需要满足对全协议的支持。

用户行为：
除协议外，场景构造还需要考虑到真实场景的用户行为，目前的做法是使用线上日志作为原材料，对日志进行过滤，整理，最终形成标准化语料文件，这样可以在一定程度上保证压测数据的真实性。这种低成本的做法，只能借助业务同学的经验去保证。人总会出错，因此未来在场景真实性保障方面不能仅仅依靠人的经验，平台需要通过技术的手段，通过模型，依靠机器去保证。

7.2 线上压测，如何保证压测流量不影响线上用户，如何保证压测数据不污染线上数据？

为了保证压测流量不影响线上用户，不对污染线上环境，首先要做的是：链路上的各系统，中间件需要做到对压测流量进行识别，具体而言需要如下步骤：

- 接入鹰眼
- 使用集团的中间件（集团的中间件都支持压测流量识别）
- 业务改造（结合鹰眼对压测标进行透传，对于某些业务可能需要在业务层面对压测流量进行过滤，如对三方系统的调用需要替换成 mock 方式）

除此之外，还需要有完备的监控手段，在发现服务问题（如系统中发生限流，降级，RT 突然增高等等）时，及时的止损。

7.3 如何构建超高流量

对线上服务的压测，需要构造出超大规模级别的压力。这需要大量的施压机器共同努力才能达到。要达到验证服务稳定性的目标，全链路压测就需要具备构造超大规模压力的能力。我们目前的策略是自建分布式 Jmeter 施压集群，依托于 Aone 的快速部署和便捷的扩容能力来迅速的满足压测流量的需求。

7.4 如何降低使用成本

压测引擎：以往高德线下的压测属于工具形式，大家使用 Jmeter 对被测服务进行压力测试，负责压测的同学都对 Jmeter 比较熟悉。考虑到平台使用的成本，和工具转平台的便捷性，我们使用 Jmeter 作为 TestPG 的压测引擎。

快速压测：高德有非常多的单链路压测需求，而且服务的接口数量比较少，也比较简单。对于此类型的压测需求，平台需要提供开速开始的能力，简化语料->场景->任务的标准化流程。

压测数据管理：全链路压测使用的压测数据（语料文件）的大小达到数十 G，文件的存储和分发对系统的设计而言都是不小的挑战。目前采取的做法是使用 OSS 来存储压测数据，通过一定的策略和算法在施压机器上对语料进行预加载。

7.5 如何降低资源成本

高德每年会进行两次大型的压测（春节和十一），其他时间如：清明，五一，端午视各业务线的情况而定。在大型压测时，需要大量的压测资源（几百台，4C16G），而平时少量的压测资源（几十台）就足以满足日常的压测需求，因此在压测资源的管理上要保证灵活，需要时快速满足，不需要时释放，避免资源浪费。

目前 TesgPG 提供两种施压集群扩容的方式：一是使用 Aone 部署，通过 Normandy 平台进行快速扩容（pouch）；二是支持用户自主提供压测机，技术方案是使用 Docker + StarAgent 方式来实现施压机的自动部署。

压测资源调度：除了支持全链路压测外，还需要支持日常的压测。对于不同的压测，系统在设计上要满足多租户，多任务的需求，在压测资源的管理方面做到按需分配（目前是人

工评估）。

8.高德压测平台的自建目标

短期目标：
支撑高德全链路压测。
为高德所有服务提供线下压测的能力。

中期目标：
成为高德线上系统稳定性的试金石。

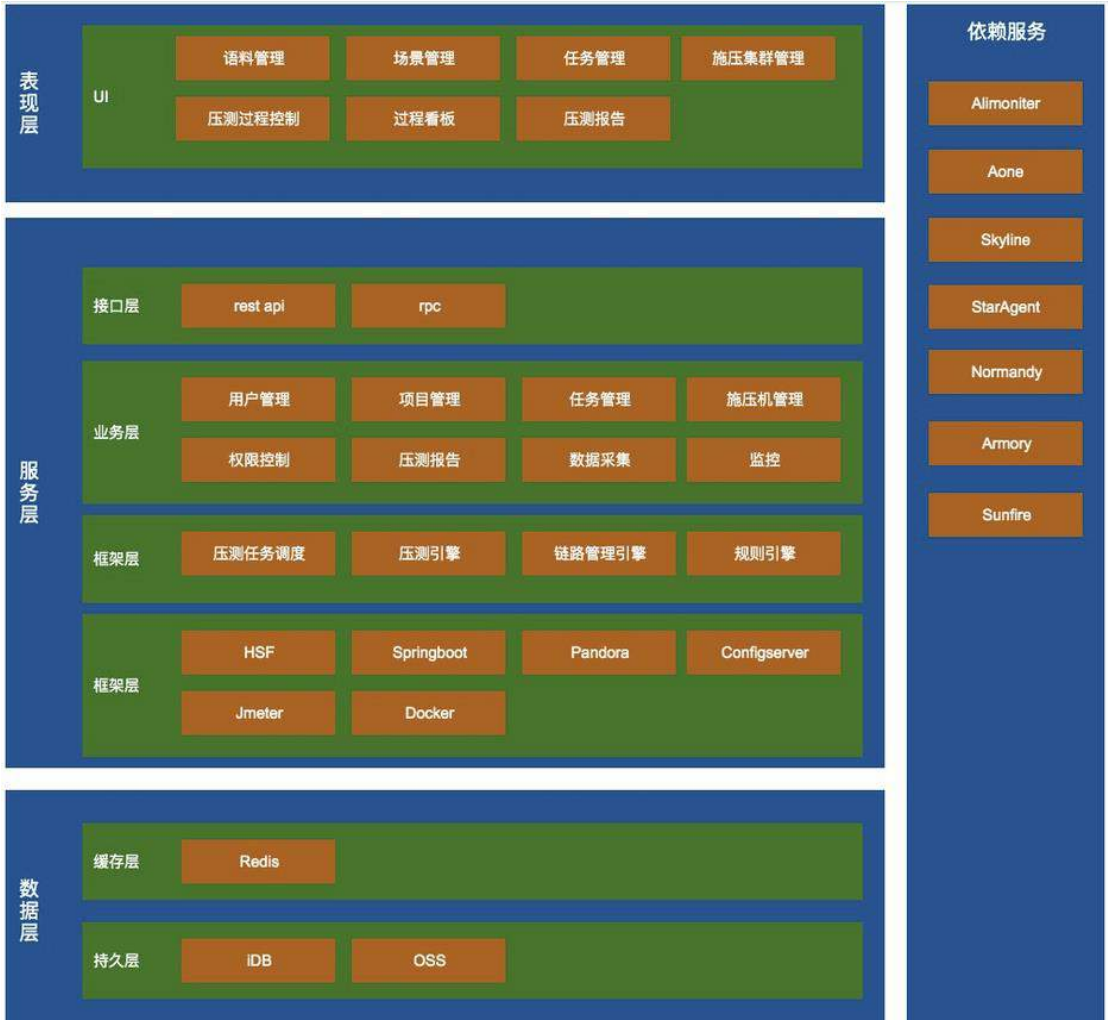
长期目标：
产品化，服务于集团的更多业务。

9.高德压测平台架构与特点

9.1 高德压测平台架构

- 高德压测平台业务架构

- 高德压测平台技术架构



9.2 高德压测平台特点

极速压测：
对于大部分简单压测需求，TestPG 提供极速压测的能力，只需要填写被测服务地址，设定压测必须的 url，请求类型，请求字段，QPS，压测时长等信息，就可以开始压测。对于初次使用平台的用户，平均 15 分钟即能上手压测。

调试能力：

TestPG 平台提供两种调试手段：挡板调试和服务调试。

- **挡板调试**：请求不会发往真实的服务，施压机作为调试挡板，对语料格式，url 格式，请求参数进行校验。

- **服务调试**：平台将压测请求（默认 20 条请求）发往真实的被测服务，并详细记录请求的上行数据和下行数据，格式化后，通过平台展示给用户。

调试的目的式帮助用户调整压测的参数，为正式的压测做好准备。

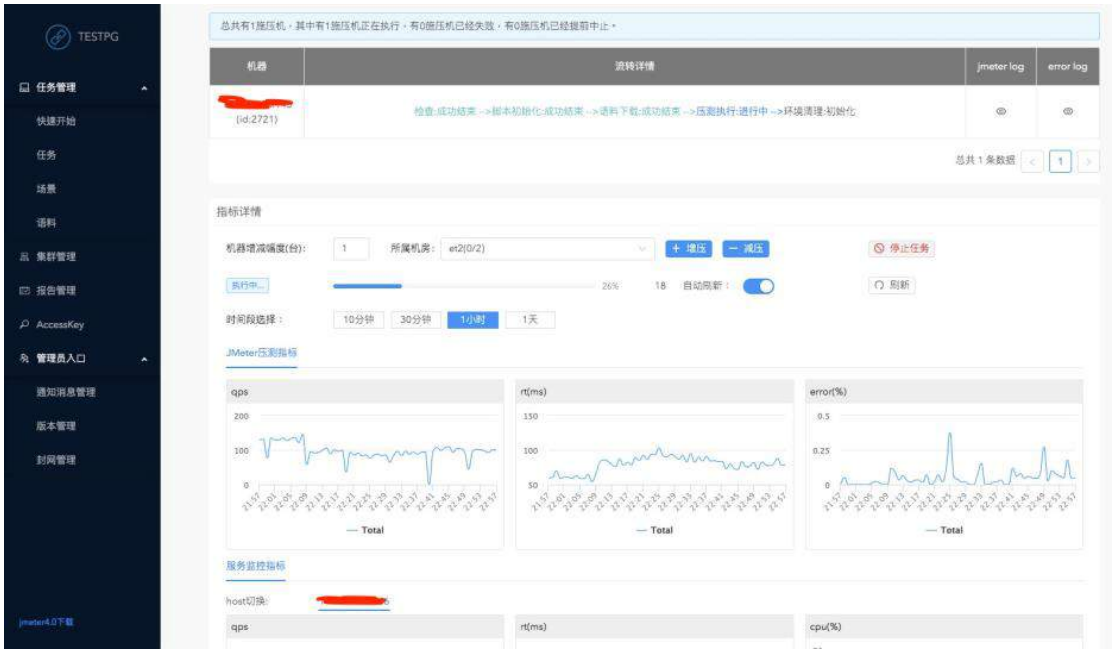
错误定位辅助：
TestPG 平台会详细记录压测过程中出现异常的请求，对请求的上行数据和下行数据进行格式化保存，用户可以在压测过程中通过平台查看异常日志，定位错误原因。

详细的压测报告&基线对比：
TestPG 平台在压测完成后自动产出压测报告，压测报告详细包括本次压测的整体、各场景、所有接口的 QPS，RT，最大、最小 RT，百分统计 RT，错误请求数量，错误请求率。并且还包含压测时的实时 QPS ，RT 图表，以及被测服务的基础监控数据。

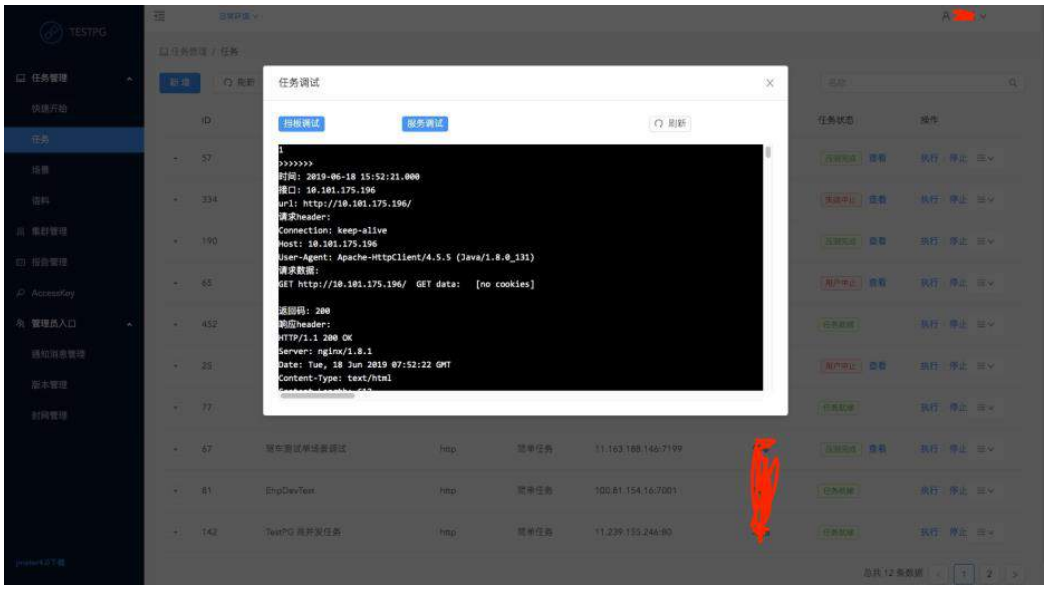
此外，压测报告还支持基线对别，若设定了基线报告，后续产出的压测报告都会和基线进行对比，在报告中就会体现出服务性能的变化。压测报告的详细信息可以查看下文平台展示中压测报告部分。

10.高德压测平台展示

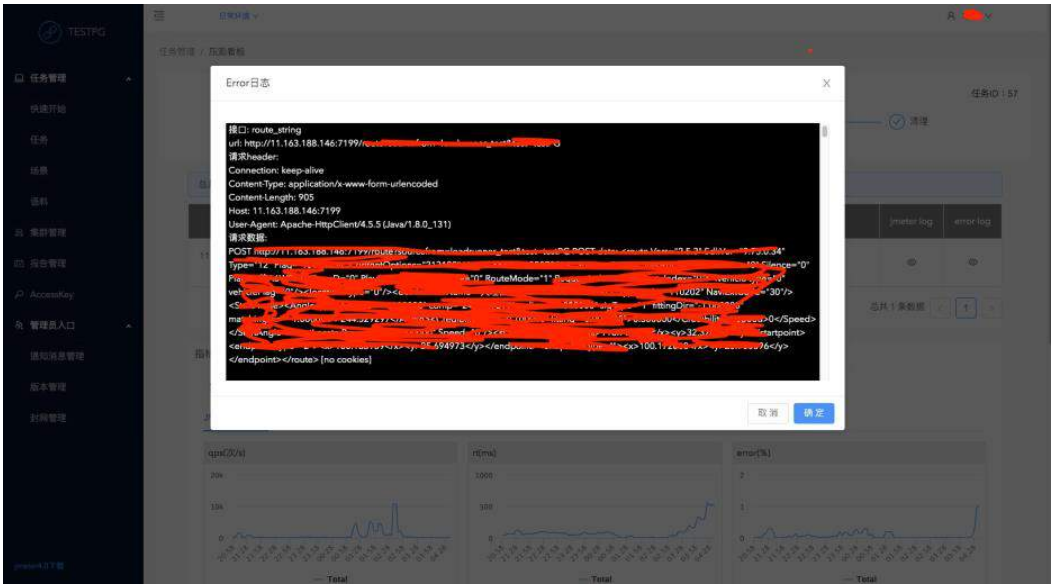
- 压测看板



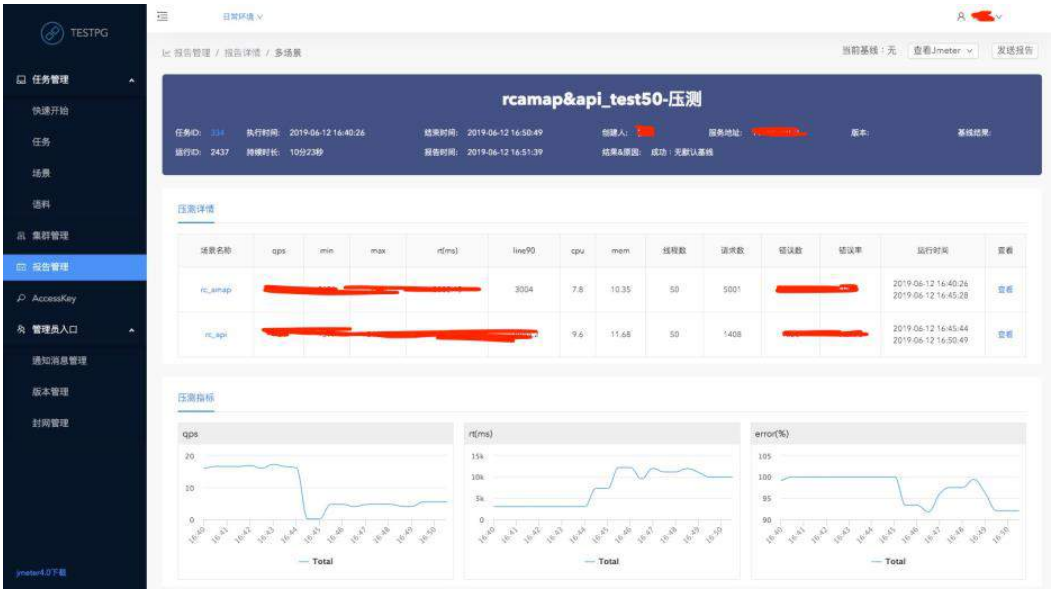
- 调试


任务调试
指标调试
服务调试
刷新
接口: 10.101.175.196
url: http://10.101.175.196/
请求header:
Connection: keep-alive
Host: 10.101.175.196
User-Agent: Apache-HttpClient/4.5.5 (Java/1.8.0_131)
请求数据:
GET http://10.101.175.196/ GET data: [no cookies]
返回码: 200
响应header:
HTTP/1.1 200 OK
Server: nginx/1.8.1
Date: Tue, 18 Jun 2019 07:52:22 GMT
Content-Type: text/html

- 错误排查


Error日志
接口: route_string
请求header:
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 905
Host: 11.163.188.146:7199
User-Agent: Apache-HttpClient/4.5.5 (Java/1.8.0_131)
请求数据:
</endpoint></route> [no cookies]
取消
确定

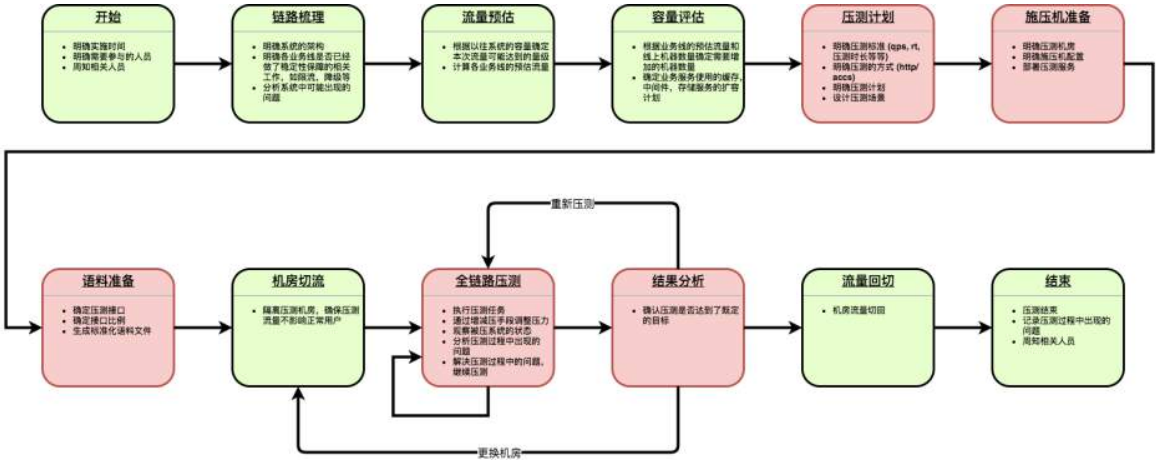
- 压测报告



11.高德全链路压测的实践

2018 年十一是我参加了高德全链路压测，当时团队的同学一起聚在稳定性保障会议室里，对高德的核心链路进行了为期两周的全链路压测。

高德的全链路压测流程是比较传统的方式，下图是历年全链路压测的大致流程：



12.高德压测平台的现状

高德压测平台从 2018 年开始启动，经过大半年的发展，已经成功支持了高德 2019 年春节全链路压测，2019 年五一全链路压测，在施压能力方面达到 百万 级别。

目前高德压测平台包括语料生产，场景构造，压测资源管理，压测任务调度，压测过程监控，压测调试，压测错误排查，压测报告生成，压测报告评估（对比基线，从 QPS，rt，cpu，mem 这几个维度进行压测结论的评估）等功能。

平台从 2018 年底投入使用至今，共执行几千次的压测。通过开放 api 的方式，平台开放压测能力，现已成功和持续交付平台打通，为持续交付提供可靠的性能指标。

自春节后，平台主要致力于降低压测门槛，提升压测效率。对于大部分简单的压测需求，从原有的语料->场景->任务逐级构造模式简化为一键压测模式，在初次使用平台成本方面，简单的压测工作预计能从平均 3 小时，减小为平均 30 分钟。

虽然高德压测平台在近一年的时间里，取得了一些成果，我们也支撑了几次全链路压测。但是我们的全链路压测还不能完全满足上文的三个要求，其中最关键的场景真实性还无法得到保证，"全链路压测" 对于高德来说，还是远方，我们还有很多路要走，还有很多困难要克服。

13.高德压测平台的未来

在可见未来里，我们期望在以下几个方面去探索和实践：

13.1 全链路监控

TestPG 目前的监控是基于用户自定义链路的监控，在链路真实性上无法得到保障。未来我们将会与运维团队合作，打造基于 EagleEye 链路自动梳理的全链路监控。除了基本的系统指标监控功能外，TestPG 平台还会在下面的几个方面进行探索：

- 监控大盘，全面的展示系统的状态
- 短板机器，短板服务的识别
- 基于时间序列的性能问题挖掘
- 结合邮件，钉钉等手段，对发现的问题进行及时的报警

13.2 简化语料生成流程

目前压测语料的生成采用的是用户自定义脚本的方式。平台定义好语料目录格式，语料文件格式，用户编写语料处理（一般以日志文件为基础）的程序，然后上传到平台。平台会执行用户提供的程序，然后在将程序的输出文件存储在 OSS 上，以备压测之用。

这种方式降低了平台的实现成本，也给用户提供了足够灵活度，但是增加了语料的处理门槛。未来我们会将这部分工作全部交由平台来处理，用户只需要提供原材料，配置生成规则即可。

13.3 丰富压力模型

TestPG 平台目前的压力模型是 Jmeter 原生的，为了最大限度的接近真实场景，在压力模拟上，我们还需要具备如下几种压力模型：

- 步进施压

- 抖动施压
- 脉冲施压

13.4 压测置信度评估

压测场景能否代表真实的用户场景？这是一个 TestPG 目前还无法回答的问题。场景的真实性现在依托于业务同学的经验，业务同学按照自己抽象出来的业务模型对原材料（通常是线上日志）进行处理，生成平台规定格式的预料文件。语料对平台和用户来讲，都是黑盒子，除了被测系统外，没人知道语料的模型是否接近真实世界的用户场景。

因此，我们期望通过一些技术手段来对压测场景的真实性进行一个科学的评估，这便是压测置信度评估。置信度评估的目标是评估压测的场景和我们期望的场景相似度，下面是压测置信度评估需要探索的方向：

- 建设场景特征库（例如五一，十一，春节）
- 基于特征库的压测场景置信度评估
- 基于地理位置的压测场景置信度评估
- 基于链路覆盖度的压测场景置信度评估
- 基于流量模型的压测场景置信度评估

结合上述维度的评估数据，我们可以给出一个具有价值的评估报告，作为一个重要的参考，可以帮助用户评估压测的效度。

13.5 链路改造

TestPG 平台的中期目标是要支撑高德业务的全链路压测，成为高德系统稳定性的试金石。但现在我们离真正的全链路压测的距离还很远，我们的压测场景只覆盖了读请求，还不具备支持写请求压测的能力，原因是系统还不具备全链路流量识别的能力，还不具备隔离压测流量的能力。未来，全链路压测，全场景覆盖是必经之路。

招聘

高德质量部热招测试开发高级工程师/专家，Java、C++高级工程师/专家，算法岗位。职位地点：北京、厦门，欢迎有兴趣的同学投递简历到 xi.yang@alibaba-inc.com

持续交付体系在高德的实践历程

作者：韫辉

1.前序

对于工程团队来说，构建一套具有可持续性的、多方面质量保证的交付体系建设，能够为业务价值的快速交付搭建起高速公路，也能为交付过程中的质量起到保驾护航的作用。本文为大家介绍持续交付体系在高德的演进与落地。

2.持续交付

正如前序中所总结的，我们需要构建一套持续交付体系，从而保证在质量不下降的前提下，在业务价值交付上有更进一步的突破。那么我们先了解一下什么是持续交付以及集团在持续交付的建设上有哪些指引。

2.1 持续交付概念

引用 Martin Fowler 大师在 2013 年时发表的文章，对于持续交付的概念有如下的解释：Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.

在上述文中，可以提取几个关键词：

- 软件开发的标准化准则
- 可以做到随时随地的发布

什么情况下就可以算是团队达到了持续发布的状态呢？Martin Fowler 大师也给出了标准的答案：

- Your software is deployable throughout its lifecycle
- Your team prioritizes keeping the software deployable over working on new features
- Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them
- You can perform push-button deployments of any version of the software to any environment on demand

那么基于以上的观点，我们在建立自身的持续交付体系时，需要抓住以下几个重点：

- 标准化流程流转
- 当有变更进入时，能够快速、准确且自动的得到反馈
- 解决部署问题的优先级高于功能开发
- 一键发布

2.2 集团的持续交付建设

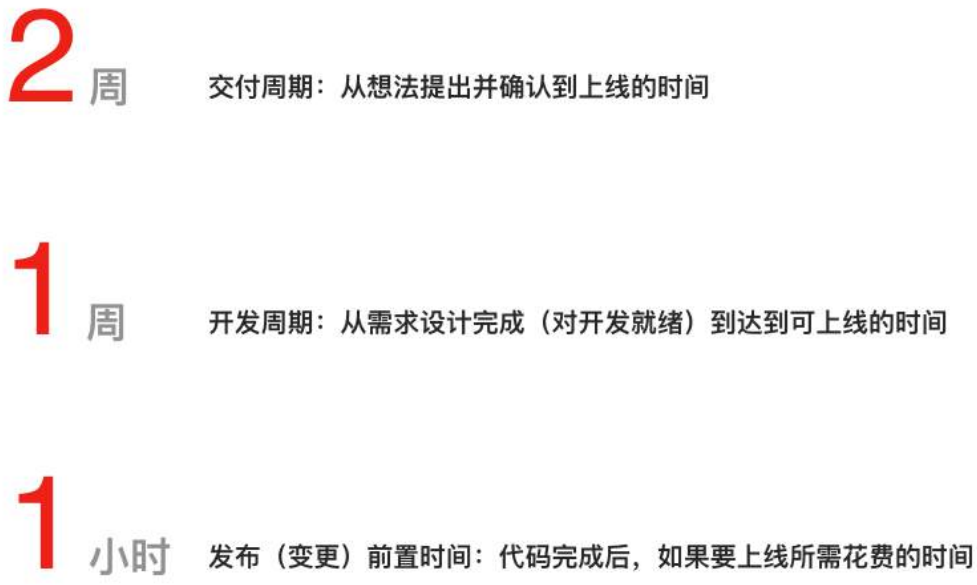
从理论基础上对于持续交付有了初步了解后，我们从集团层面了解一下是如何定义持续交付的能力，并且对于持续交付提出了哪些效能改进目标。

文章中将持续价值交付的能力拆分为 3 个层面的 5 组指标，从不同角度对持续价值交付能力进行了衡量。

有了上面专业层面的衡量指标，那我们是如何定义一个优秀的持续交付衡量目标呢？

管理学之父德鲁克说：“如果你不能度量它，就无法改进它”。度量帮助我们更深刻认识研发效能，设定改进方向，并衡量改进效果，所以想要进行效能提升的前提是先能够识别交付过程中的质效瓶颈。

因此，集团在基于部分 BU 的优秀实践下提出了 2-1-1 的愿景。



- 1 小时的发布前置时间是对于基础设施能力的要求，需要保证当达到交付标准后，通过交付流水线能够达到 1 小时内的打包、部署和验证的能力；
- 1 周的开发周期涉及产品需求拆分、研发 QA 协作能力、持续测试以及快速反馈能力方面提出了挑战；
- 2 周的需求交付周期是以前两项为基础，不仅是涉及到产研测三方，还包括其他协同部门的通力合作才能保证业务价值的快速交付。

3. 持续交付在高德

在基于集团愿景的指导下，反观现有高德服务端的交付流程，我们发现在整个流程中，存在很多效率上的竖井，这些效率问题汇总起来，便会成为整个交付流程上的效能瓶颈，进而影响业务价值的尽早交付。

我们先从一个整体的 Milestone 来回顾一下整个持续交付所经过的一些重要时间节点：

- **2018/08 构思与工程能力建设**：项目启动阶段，工程效率团队与业务线明确了持续交付的目标，并启动了工程能力建设。

- **2018/12 初步落地与试点**：项目试点阶段，完成了初步的持续交付流程搭建，并在一个项目中验证流程卡点以及质量标准的基础能力验证。最终建立了基础的质量标准以及降低流程中的耗时。

- **2019/04 推进接入与平台优化**：项目推进阶段，持续交付项目质量项优化并在高德的服务端的 6 条业务线中进行推广，在 9 月份完成 6 条业务线以及 11 个应用的持续交付落地。

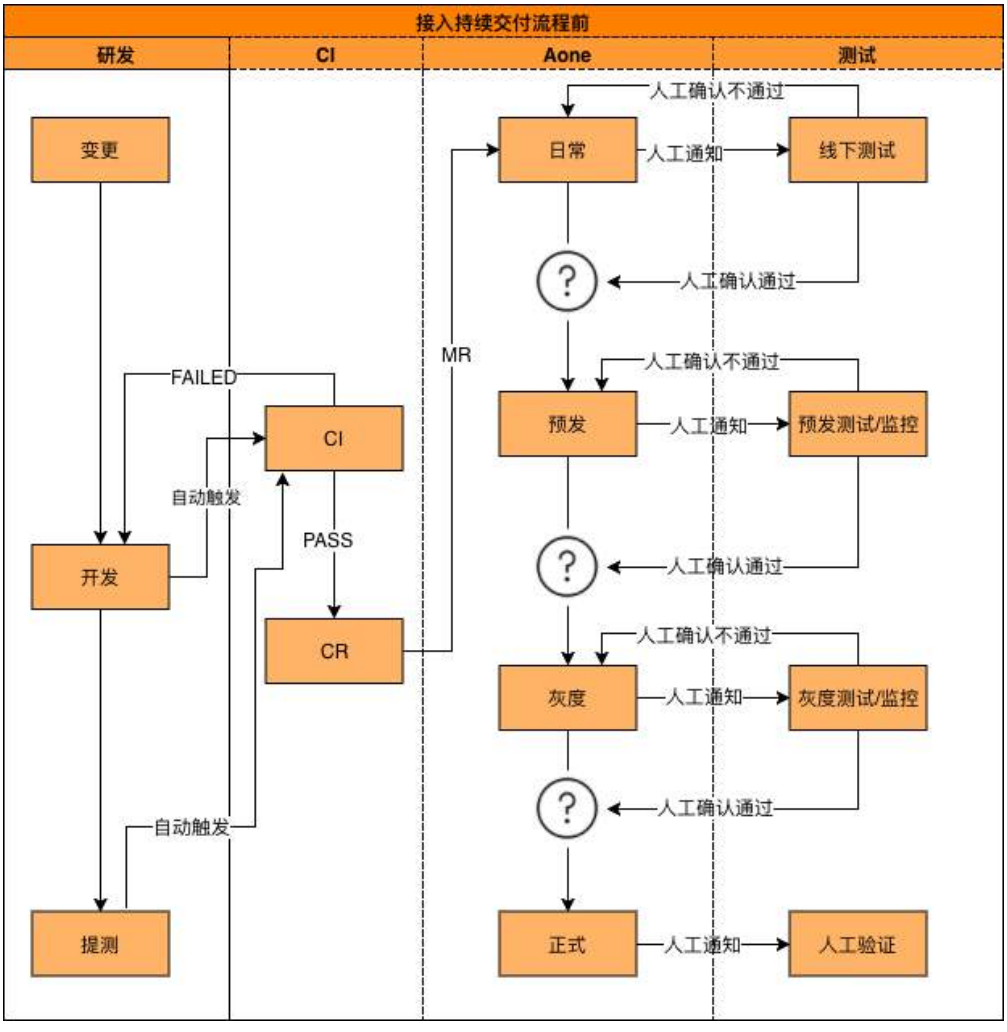
- **2019/09 复盘与展望**：项目推进总结，对整个推进过程进行复盘与后续持续交付如何落地进行复盘与展望，整体产出业务推进中出现的问题以及改进方法。

- **未来**：交付流程上做贴合业务线的微创新，并对效能瓶颈点进行纵深挖掘。结合各纵向平台进行纵深挖掘，例如：覆盖率与精准回归、云歌 Case 平台、代码扫描平台等。

通过 milestone 的展示，对于高德持续交付体系的演进有了大致的了解后，下面对于落地的过程以及改进的内容进行一下详细的梳理。

3.1 接入持续交付前的交付流程

首先先介绍一下在接入持续交付体系之前，高德的服务端如何进行迭代开发与上线。



与大部分互联网公司一样，我们将软件的交付拆分为多个周期，进行迭代式的交付，以便增量式的进行用户价值的交付。上图描述了一个正常迭代周期内的研发、测试以及发布的流程，我们可以拆分为以下几个方面：

- 迭代周期起始于代码库的变更。
- 在功能开发完成后，研发通过 CI 系统进行冒烟测试验证，保证服务可以正常启动以及基础功能可用。
- 在规定的提测时间前，研发将 Feature 分支通过 CR 和 MR 合并到迭代分支，部署到日常环境进行提测。
- QA 在收到提测邮件后，参与到日常环境的测试中。
- 当日常环境测试完成后，QA 会进行测试报告的产出，并确认日常环境测试通过，可以发布到预发环境。
- 部署到预发环境后，会进行流量回放等测试，并最终通过线上的灰度验证，最终发布到正式环境。

通过上述的图片和描述，我们可以看到在看似完善的软件交付过程中，却仍然存在如下一些质量、效率问题：

- 需求堆积提测、发布：

目前高德服务端大部分服务采用的是固定迭代周期进行需求发布，规划到迭代周期内的需求，无论需求大小，均需要等到迭代提测时间点进行提测，在迭代的发布窗口进行发布上线。在这种模式下，好的一点是有固定的版本节奏，整体迭代规划性比较强。但是由于提测、发布窗口固定，从而也带来了整体业务价值交付上的等待。因此，需要通过需求拆分来降低需求内部的耦合性，通过改变研发、QA 的开发测试模式来降低需求提测中间的竖井等待，从而提升业务价值交付的效率。

- 质量标准不透明，无法及时反馈：

从代码提交一直到最终产品发布，一般情况下，会经历日常、预发、灰度、正式发布几个阶段，每个阶段均有每个阶段需要重点解决的问题以及对质量上的要求也不尽然相同。目前结果的收集汇总和通知都是通过跟版人进行人工收集和统计，并邮件通知项目成员。这样所有的标准控制都是有每个版本的跟版人进行把控，存在信息不透明，反馈不及时的问题。通过质量项标准的建立，以及大盘结果透明和及时的通知，能够解决沟通层面的低效以及在传递过程中信息损耗，从而提升沟通效率，并且避免沟通中的误解。在解决了当前透明化和及时通知的问题后，我们需要进一步从以下两方面进行优化：

1）将通知进行分类以及优先级处理，降低通知带来的负面影响

2）通过信息内容优化，辅助业务进行问题的快速定位与排查

- 部署与流程流转过程需要人工参与：

对于持续发布流程来说，有人工参与的地方势必会影响到其中的效率。所以我们将部署和阶段流转拆分为两个方面看：

阶段流转：结合上述的阶段标准，通过程序来计算是否能够满足当前的质量情况是否可以进行阶段的流转，从而排除人为因素以及在阶段流转中的耗时，做到准确

部署：提取相应环境的配置信息，结合 Docker 化，将打包、部署、健康检查等一些列活动转换为机器的标准化执行，通过标准化来避免人为参与所造成的误差或部署失败的问题

- 多机房正式发布验证人工监督：

目前在应用的正式发布流程中，由于涉及的机房和机器数量较多，业务上会进行分批验证，每发布完成一批机器，研发会通知 QA 进行这批机器中部分机器的抽检（部分自动化测试），在这其中也存在着效率上的问题。所以如何节约每次上线过程中的人力损耗，也是在追求效能极致上需要解决的问题。

上述的每个细节的问题，都在我们通往快速业务价值交付的道路上设置了障碍。因此，为了达成更早（快）的交付业务价值的目标下，我们必须要在交付效率、质量标准以及结果快速反馈这几方面的进行优化。

3.2 持续交付在高德的落地

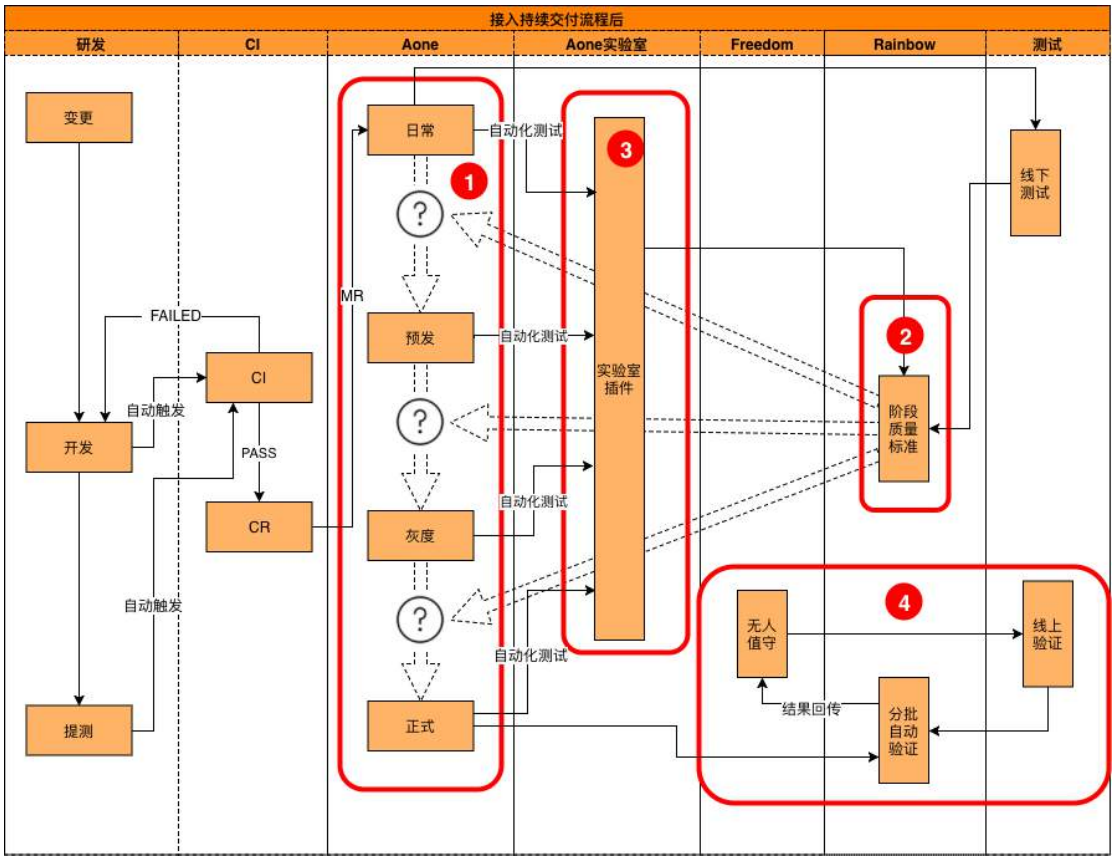
基于上节拆分出来的 4 方面的问题，从工程角度来说，由于迭代的排期，需求的分解与拆分需要进行长期的实践与规划，并且依赖于产、研、测、项乃至于其他部门的支撑，是一个需要进行逐步探索和调整的过程。所以我们将着眼点放到后 3 方面的建设上，期望在短期内先建立起快速发布的能力，清除在交付过程中效率低下的点。

那么在解决效率问题的建设上，借助于集团提供的发布流程以及较好的部署能力，我们将目前拆解为如下几个维度的抓手：

- 依托于集团的发布流程，在持续交付体系中建立与集团发布流程对应的标准化流程流转机制。
- 建立服务端质量标准体系，拉通质量标准，去人工化。
- 打通各环节的快速反馈机制，并对发布流程进行管控，让变更结果随时可见。
- 降低发布过程中的人为参与，让整个发布流程做到全程无人值守。

通过下面持续交付流程图，我们通过接入后的流程图中看一下以上 4 个抓手是如何串联起

整体高德持续交付流程，并且这几项是如何在高德服务端交付流程中进行落地的。



建立标准化的流程流转机制

FY19 高德服务端发生的线上问题中，其中由于变更或发布引发的问题占比约 12%。通过这组数据，我们期望能够通过建立一套完整的交付流转流程，实现对于变更的控制和管理，降低或避免此类问题的发生。

基于以上立论，我们结合当前服务端交付特点，首先先确立以集团标准发布流程为试点，打通整体持续交付流程；其次，针对各应用中不同的需求，例如：需要性能环境、覆盖率环境等，结合流水线配置，将整个持续交付的流程流转进行优化；最终沉淀为各服务的标准化流程流转机制。通过这种先僵化，后优化，再固化的方式，最终在服务端落地了多套标准的交付流程，避免了在交付环节上的遗漏，以及不规范的操作。

拉通并落地服务端质量体系标准

在高德现有的交付流程中，整体的质量保障手段大部分是在日常阶段进行的，在迭代交付的过程中，各项质量保障手段执行了哪些，执行结果是什么，目前还是通过 QA 人员进行人工问题收集与汇总，并判定阶段结果的通过与否。在这种情况下，会出现由于跟版人交替导致的质量项遗漏，以及质量标准难以把控的情况。

所以基于这几方面的问题，我们希望通过用机器把控替代原有的人工把控的方式，通过建立标准化的质量模板，来避免整体执行标准不透明，执行结果无沉淀的情况。并且，通过拉通标准，也进一步的规避掉了非重点服务质量检查点遗漏的情况。

通过与业务团队的沟通，我们在第一阶段将现有服务端的质量保证手段进行拆分，提取了在不同阶段中相对重要的 12 项质量项，通过机器监督替代原有的人为统计的方式。具体覆盖了如下几个维度：

- 代码维度

静态扫描
安全扫描

- 测试维度

冒烟结果
新功能结果
回归结果
Fuzz 测试结果
流量回放结果
兼容性测试结果
压测结果

- 监控

预发、线上监控

- 需求/Bug 管理平台关联

当前版本需求进展
当前版本 Bug 情况

打通各环节的快速反馈机制，并对发布流程进行管控，让变更结果随时可见

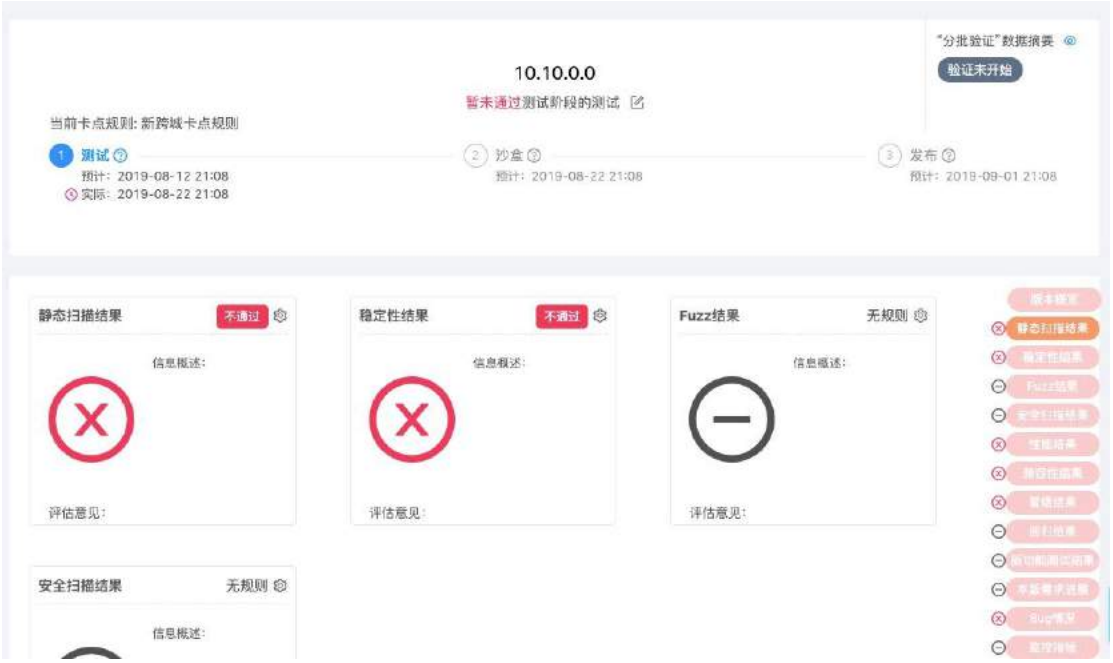
当建立起有效的质量体系后，在各阶段有了质量要求以及准入准出标准，解决了信息收集方面的问题，那么接下来我们要思考的就是如何将收集上来的各种信息，有效的反馈到项目中的各个干系人，以便进行后续的决策支撑，并且当未达到阶段准出标准时，有效的控制项目的阶段流转。

我们将问题拆解为两方面看，一是有效反馈、决策支撑，二是流程流转的管控。

从有效反馈、决策支撑方面看：

- 在接入持续交付之前，各业务线的针对不同类型的自动化测试任务，大部分都有通过 Jenkins 或测试用例工程反馈结果的通知。但是此类反馈有一个致命的问题，就是通过单项反馈无法纵观全局，不足以支撑后续的决策。

- 在接入持续交付后，除了原有业务上的反馈机制，平台提供能针对当期版本的整体状态全览，可以通过平台随时观测到当前版本是否达到可发布的状态或者仍然存在哪些不足。将两者结合起来后，针对项目执行人仍然可以通过原有反馈机制了解到单点的质量结果；对于跟版人、一线、二线管理者这类需要纵观全局的角色来说，通过质量大盘，可以有效且明确的知道当前版本与待发布状态的差距，并支撑后续决策以及调整关注的重点



从流程管控方面看：

- 在接入持续交付之前，可部署的产物无论是否经过阶段验证，都可人为的部署到任意环境下，虽然灵活性比较高，但是也存在一定的质量风险。

- 在设计持续交付流程时，对于灵活性以及规范性的取舍方面，我们也与业务同学进行了讨论。从全局看，为了避免流程不规范引起漏测或其它线上事故，最终确定在初版时先保证流程流转的规范性，从而降低灵活部署上所带来质量上的风险。平台通过集团实验室插件与集团的部署发布系统打通，当阶段中存在质量项尚未达标的情况下，阻止发布流程进入到下一阶段（环节）。

- 当基础的持续交付流程落地后，为了满足业务上对灵活性的要求，目前我们也在尝试通过自定义流水线来进行多环境的分发与部署，从而在保证主要阶段流转有管控的同时，增加部署的灵活性，以适应不同的业务形态。

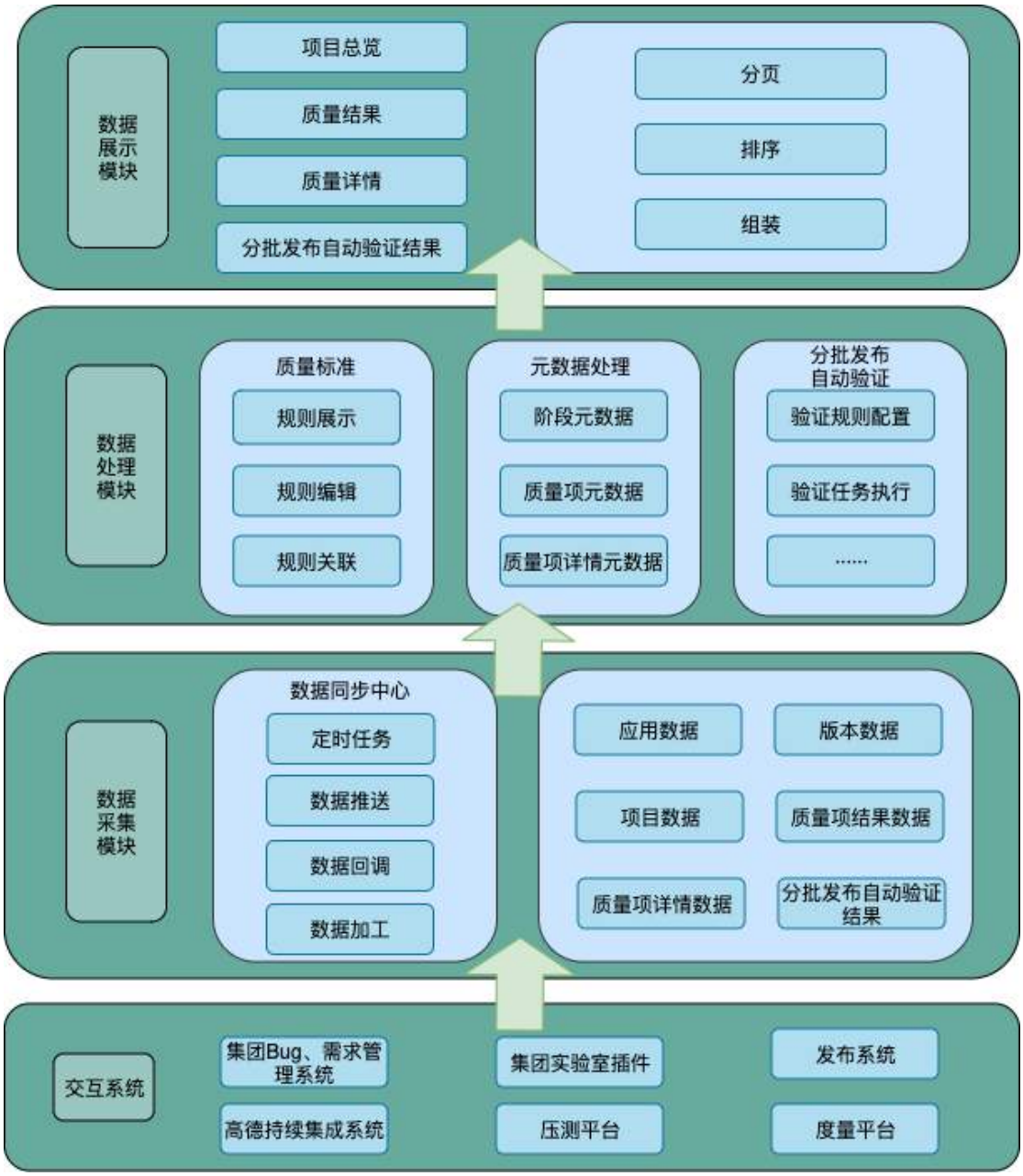
降低流程发布过程中的人为参与，让整个流程做到全程无人值守

我们知道，线上环境部署的复杂程度要远高于在日常和预发环境的部署。由于部分业务线，线上的机器数量众多，且分布在不同机房，为了保证部署时的服务可用性，线上部署时会将上千台机器拆分为多批次进行部署。

在接入持续交付前，为了保证部署后服务的可用性以及对质量上的高标准要求，在每批次部署完成后，QA 都需要针对当前批次进行全批次验证或抽测验证，当验证通过后，再进行下一批次的发布以及后续验证。虽然验证本身是通过自动化脚本进行验证，但由于机器和批次比较多，整个发布和验证流程会持续数小时，存在较大的效率问题。

在了解到业务上此效率瓶颈后，通过打通上下游系统，集团标准流程、集团发布系统以及原有业务的线上验证工程，针对不同业务的发布场景，进行发布验证策略的配置化。通过感知部署时的消息，获取当批次部署的机器列表，依据各业务的验证策略配置进行自动化的验证。并且结合线上阶段的报警监控，当某批次发布验证出现问题后，系统可以第一时间定位到具体是哪一批次中的哪台机器发布出现问题，帮助业务进行部署问题的快速定位。

持续交付体系的业务架构



4.落地效果

整个持续交付体系建设，目前在高德服务端落地已经有一段时间了，截止到目前为止：

- 业务线覆盖：整个持续交付体系已经覆盖了高德服务端大部分重点业务
- 各阶段质量项建设：12 项
- 正式发布阶提效：50%~90%

在获得以上成果的同时，除了上述量化指标外，更有价值的是隐含在背后的研发、测试习惯上的变化。从研发、QA 和项目主动发起的缩短项目周期，到 QA 对于质量项上提出更多

的诉求等等，无一不感知到大家对于尽早且高质量的交付业务价值这件事情的重视。当然对于更早（快）的交付业务价值这个目标还有一定的差距，这个也是后续我们与业务线需要共同解决的问题。

5.持续交付的未来

有人将持续交付形容为在价值交付上的高速公路，持续交付的落地，标志着价值交付到用户的快速路已经建立完成。但是最终是否能做到更早（快）的交付业务价值，还取决于在这条快速路上行驶的车辆。

根据这个理论，我们除了要保证这条高速公路上不出现坑洼的同时，还要兼顾车辆本身的能力，以及车辆的性能。因此，在车辆出发前，我们更需要通过对车辆的车况进行检查，保证在高速路上行驶的车辆不会因为自身的原因提不起速度。

5.1 车况检查

目前，已有的持续集成系统，仅能够保证车辆在这条路上是能开起来的，车况的检查都是在上了高速后才开始的（大部分的质量保证手段是部署到日常环境后才开始）。所以基于上面描述的指导方针，我们需要尽早的做检查，并且需要做更全面的检查（质量保障手段左移）。

基于这个目标以及结合集团内其他 BU 的优秀实践，后续我们希望能通过代码门禁的手段，尽早落地这类全面的检查。若要将代码门禁落地，无论是对于工程效率团队亦或是业务研发与 QA 团队，都有着不小的挑战，我们需要做到以下的转变：

- QA

质量保证的同期化能力建设
质量保证的稳定性与耗时优化

- RD

研发提交代码流程的改变
单元测试能力的建设
Code Review 的常态化落地以及规范总结

- 能力支撑

代码覆盖率，业务场景覆盖率的支撑
代码合并的门禁管控能力
代码扫描结合 CodeReview 的总结的落地

当逐步完成以上任务的落地后，能够消除批量交付业务价值交付中相互等待的时间，并且也能够保证车辆在持续交付这条高速路上行驶得更快更稳定。

5.2 车辆性能提升

前面车辆检查可以说是在车辆上路之前的检查与保障，将质量保证手段左移到研发阶段。相对的，我们希望通过车辆性能提升的方法，在车辆上路后，能够让车辆行驶提速更快，拉高速度的上限。

- 纵向测试能力提升

精准回归：通过感知代码的变化，推导出代码变动所影响的 Case，让质量保障更为精准且耗时更少。

场景覆盖：结合线上流量回放，通过代码覆盖、场景覆盖进行查缺补漏，让质量保障更完整。

问题定位：结合失败用例，快速的进行问题定位与反馈。

同期化能力：结合云歌 Case 平台，通过接口定义进行测试代码与研发代码同期化编写能力的加强，以及降低 Case 编写和维护成本方面的探索。

降低数据干扰：基于高频、隔离和用完即抛的理论实践，降低日常环境的数据干扰，让质量保证更有效。

- 与线上数据挖掘结合

大数据分析

利用线上日志分析，产出线上真实场景模型，降低压测平台语料准备耗时，场景筛选上做到精确、高效。

大数据运用

结合线上真实场景以及场景覆盖率，构造线下回归 Case 集，降低业务回归 Case 维护成本，提升 Case 有效率，并且能够快速定位问题。

利用场景回放，以及记录回放中间产物，解决在单测时场景构造问题。

随着持续交付快速通道的搭建完成，期望通过以持续交付体系为契机，在多个纵向维度进行深入挖掘，并完善整个持续交付体系，最终在更早（快）的交付业务价值的前提下，能够有更高的质量以及更低的人工成本，保证市场竞争的先机，让高德在激烈的竞争中优势更为明显。

招聘

高德质量部热招测试开发高级工程师/专家，Java、C++高级工程师/专家，算法岗位。职位地点：北京、厦门，欢迎有兴趣的同学投递简历到 xi.yang@alibaba-inc.com