<打车软件系统>

<体系结构设计文档V 2.0>

<2019.12.18>

<周子格>

<软件学院>

<学号：2016211978 班级：2017211502>

软件工程导论

2019 秋

**修订记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **描述** | **作者** | **内容** |
| 2019.12.05 | 版本 1 | 周子格 | 第一次修订 |
| 2019.12.17 | 版本 2 | 周子格 | 第二次修订 |
|  |  |  |  |

**文件许可**

以下的软件需求规格已被接受并认可：

|  |  |  |  |
| --- | --- | --- | --- |
| **签名** | **名称** | **标题** | **日期** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**目录**

[1. 项目及文档相关信息 - 4 -](#_Toc1210115845)

[1.1 项目背景 - 4 -](#_Toc662384033)

[1.2 文档范围 - 4 -](#_Toc279948133)

[1.2.1 待开发软件系统名称 - 4 -](#_Toc2072835230)

[1.2.2 项目简介 - 4 -](#_Toc492578504)

[1.2.3 项目价值 - 5 -](#_Toc613664744)

[1.2.4 项目特色 - 5 -](#_Toc33718103)

[1.3 定语、术语、和缩写语 - 5 -](#_Toc634264969)

[1.4 参考文献 - 6 -](#_Toc993749641)

[1.5 文档概述 - 6 -](#_Toc780325706)

[2. 体系结构需求 - 7 -](#_Toc664026319)

[2.1 关键指标 - 7 -](#_Toc1536890293)

[2.2 体系结构用例 - 7 -](#_Toc428528795)

[2.3 各相关方对体系结构的要求 - 8 -](#_Toc1099655750)

[2.4 约束条件 - 9 -](#_Toc2061548300)

[2.5 非功能需求 - 9 -](#_Toc258931446)

[2.6 风险 - 10 -](#_Toc196660399)

[3. 解决方案 - 11 -](#_Toc1235590894)

[3.1 体系结构概述 - 11 -](#_Toc289282789)

[3.2 技术中台 - 12 -](#_Toc1086386461)

[3.2.1 整体架构 - 12 -](#_Toc791638903)

[3.2.2 框架实现具体细节 - 13 -](#_Toc2128200013)

[3.3 数据中台 - 15 -](#_Toc566659242)

[3.3.1 整体架构 - 15 -](#_Toc414884086)

[3.4 业务中台 - 16 -](#_Toc136205433)

[3.4.1 服务化 - 17 -](#_Toc401368699)

[3.4.2 异步化 - 17 -](#_Toc1126800373)

[3.4.3 配置化 - 18 -](#_Toc399636885)

[3.4.4 插件化 - 19 -](#_Toc1426091875)

[3.4.5 数据化 - 20 -](#_Toc1237645929)

[4. 系统的质量分析和评价 - 20 -](#_Toc1303514196)

[4.1 场景分析 - 21 -](#_Toc488724072)

[4.1.1 用例场景 - 21 -](#_Toc1900029962)

[4.1.2 用户用车业务流程图 - 22 -](#_Toc1583462329)

[4.1.3 任务流程图 - 24 -](#_Toc414075654)

[4.1.4 增长性场景 - 24 -](#_Toc245124819)

[4.1.5 探索性场景 - 25 -](#_Toc49643425)

[4.2 原型分析 - 25 -](#_Toc447793758)

[4.2.1 原型评价法 - 25 -](#_Toc879389788)

[4.2.2 效用树法 - 25 -](#_Toc1043393067)

[4.3 风险 - 25 -](#_Toc1228119464)

[A. 附录 - 26 -](#_Toc1543416108)

[A.1 附录一：表对应页码 - 26 -](#_Toc432799712)

[A.2 附录二：图对应页码 - 26 -](#_Toc1656648259)

# 1. 项目及文档相关信息

## 1.1 项目背景

　　经济的发展，导致了我国机动车数量的逐年增长。截止到今年今年第一个季度，我国机动车保有量达到2.83亿辆，给交通和环境带来了压力。

　　出租车行业作为基础服务业，在一定程度上缓解了上述压力，却因服务态度差，车辆老旧，不安全等诟病，使得选择出租车出行的乘客日益减少。约自2010年始，互联网的发展使各行各业都受到了冲击，传统出租车行业也不是例外。

　　滴滴出行将出行进行“互联网化”改造，打造一站式移动出行平台；为5.5亿用户提供出租车、快车、专车、豪华车、公交、代驾、企业级、共享单车、共享电单车、共享汽车、外卖等多元化的出行和运输服务。在全球范围内，滴滴与Grab、Lyft、Ola、99、Taxify、Careem构建的移动出行网络触达全球超过80%的人口、覆盖1000多座城市。目前，滴滴通过旗下99平台服务巴西，在墨西哥和澳洲提供滴滴品牌的出行业务；并在日本通过合资公司提供网约出租车服务。

## 1.2 文档范围

### 1.2.1 待开发软件系统名称

滴滴出行

### 1.2.2 项目简介

　　滴滴出行是以出行为核心、辐射单车、代驾、车服、金融、国际化等领域。在各条业务线飞速发展的过程中会存在着很多相同或者类似的业务需求，需要通过技术的手段抽象、沉淀这些业务为通用、稳定基础能力，让各业务线专注于其个性化的部分，快速的推出适合市场的新产品。

　　业务中台构建订单中心、计价中心、支付中心、passport、用户中心、触达平台六大能力，高效率、低成本的支持了各条业务线的快速发展。在业务多元化发展的组织中，去构建一套工程架构，构建一套组织结构及对应的管理机制，以保证业务可持续的又快又好的发展。

### 1.2.3 项目价值

1. 在滴滴平台，超过3100万车主及司机获得灵活的工作和收入机会。
2. 增加司机的接单率与收入，乘客出行打车等待时间也大大减少。
3. 滴滴出行与监管部门、出租车行业及社群积极协作，致力于以智慧交通创新解决全球交通、环保和就业挑战。
4. 滴滴也和汽车产业链企业建立广泛的合作，携手打造面向未来的汽车运营服务平台。
5. 滴滴始终致力于提升用户体验，创造社会价值，建设安全、开放、可持续的移动出行新生态。

### 1.2.4 项目特色

1. 乘客可以在室内打车，并且在短时间内就能打到车。

过去如果打一辆出租车，大多数城市的路边大概有50%的机会3分钟能等到一辆车，乘客会付出很长的时间。滴滴出行让乘客可以在室内下单，又因为平台派单（司机不可挑单只能被指派），全局会有AI的算法和引擎，基于过去的经验和需求供应的情况去做全局的最优算法。所以，使得在绝大多数城市的打车下单的应答率到了90%。

1. 拼车或顺风车服务给乘客提供便宜的打车价格。

平台利用闲置资源，将车辆空闲的一个座位再卖出去可以打七折，将两个空的座位卖出去可以打四折。

1. 专车、豪华车满足乘客对服务的要求。

保证车辆的干净程度，车辆带水，并且严格要求司机的服务。

## 1.3 定语、术语、和缩写语

表 1 列出了本文档使用到的全部定义、术语；

|  |  |  |
| --- | --- | --- |
|  | 名称 | 名词解释 |
| 滴滴基础平台 | Odin | 运维平台，提供 metrics 上报、多维度监控、报警、服务树等功能 |
| 把脉 | 日志平台、提供日志采集通道、基于 traceid 的全链路日志查询能力 |
| DISF | 服务注册平台，提供高可用的服务名字服务、管理服务分组 |
| RDS | 提供高可用、透明水平扩展的MySQL集群，支持数据总线、分身等能力 |
| DDMQ | 低延迟高可用的消息队列服务，单机TPS吞吐超过百万，支持延时消息 |
| Fusion | 基于rocksdb的高性能高可用分布式持久化存储方案，完全兼容Redis协议 |
| 弹性云 | 基于k8s，高效、可伸缩的集群管理平台，服务自动容错，基础设施免运维 |

## 1.4 参考文献

【１】王安生. 软件工程化[M]. 北京：清华大学出版社,2015.

## 1.5 文档概述

　　本文主要针对滴滴出行系统进行软件结构的设计，得出了滴滴出行系统的体系结构说明书，主要内容包括四部分，具体如下:

第一部分主要介绍了本系统的应用背景、使用范围以及本文使用到的定义、术语与缩写词，同时也列出了本文的参考文献和本文档的组织结构。

第二部分主要对本系统的体系结构进行一个总体的概述，简述了体系结构的需求以及用例，还对各相关方对体系结构的要求进行了说明，并对本系统的约束与假设和依赖做了简单陈述。

　　第三部分主要介绍了对本系统的体系结构进行了分级设计。对相关的体系结构进行了概述。

第四部分对系统进行了质量分析与评价。主要使用了场景分析的方式，描绘了用例场景，增长性场景和探索性场景；并使用该场景进行手工测试。同时对该项目的原型分析和风险进行了说明。

# 2. 体系结构需求

## 2.1 关键指标

　　本系统要求稳定性和安全性达到一定的高度，出错率降低到一定的水平。才能保证软件的正常、快速、高效的使用。

滴滴出行数据量、用户量、访问量

1. 106+ TB　日轨迹数据
2. 4875 PB　日处理数据
3. 400亿+　日路径规划次数
4. 150亿+　日定位数
5. 百万级日订单规模，万级并发长连接
6. 滴滴注册用户已超过5.5亿，每年完成运送110亿次。

本系统下在应对上述数据量、用户量、访问量时应当能够：

1）用户进行查询响应时间不得大于0.2s

2）订单：写入满足每秒１０ｋ事件，读取满足每秒１ｋ事件，时效性５ｓ内数据可用

3）其它所有交互功能反应速度：响应时间不超过0.1s；

4）对处理数据量的要求：系统的响应应当能够在0.6s之内；

软件的可靠性：平均故障时间间隔不超过200小时

所以本系统在可靠性上应当能够：

1）出错率

在累计1亿次服务请求的情况下，其出错次数不得高于3次。

在累计100亿次服务请求的情况下，其出错次数不得高于60次。

2）稳定性

连续12个月的工作情况下，系统不会出现停止服务的情况。

3）易恢复性

在停机后进行恢复之后，损失的数据不得大于10min。

所以本系统在可维护性上应当能够：

1）要求系统1年的停机维护时间不得超过３个小时。

## 2.2 体系结构用例

本说明书的用例是从使用者的角度出发，描述用户所期望的与整个运行系统的交互  [1]。根据《软件需求分析说明书》对本系统用户需求的分析以及对用例角色的陈述，本 体系结构说明书将用户的需求与系统进行交互，设计体系结构用例如表 3

**表2.体系结构用例**

|  |  |
| --- | --- |
| 角色 | *用途* |
| *软件开发人员* | 1.  清楚知道软件的整体结构。  2. 清楚知道各模块间的接口，方便编写。  3. 清楚知道自己所要完成的功能。 |
| *产品经理* | 1. 知道软件系统的规模。  2. 根据体系结构安排工作给各开发小组。  3. 决定开发策略。  4. 规划工期进度安排 |
| *测试人员* | 1. 明确系统的性能指标。  2. 明确系统的组织结构。  3. 编写测试用例的依据。 |
| *甲方人员* | 1.对该体系结构文档给出评价。  2.验证其是否能满足自己的需求。  3.*项目验收的标准。* |
| *老师* | 1.对该体系结构文档提出修改建议。 |

## 2.3 各相关方对体系结构的要求

1. 滴滴出行的复杂业务

随着**网络带宽和浏览器技术升级**，更多的网站开始使用前端渲染，服务端则更多的退化成 API Gateway，前后端有了明显的分层。

同时，由于**互联网业务越来越复杂**，**存储变得越来越多**，不同业务模块之间的**存储需要隔离**。

并且滴滴出行的组织结构为**多事业群架构**，每个事业群里面又有很多事业部，下面还有各种独立的部门，一个部门往往会需要提供一个 API 或 broker 服务来屏蔽公司内其他服务对这个部门服务的调用，在逻辑上就形成了由多个独立微服务构成的“**大型微服务**”。

如果没有业务中台，各个业务模块的支付系统都需要从头开发一套支，非常麻烦。有了业务中台可以一键接入支付，至于下单后，微信支付宝怎么调用，怎么支付，失败后怎么处理，就不需要业务方再去操心了，依赖业务中台的统一接口就可以搞定，在整个业务的支持上来讲，让开发更快。

1. 开发人员

能让来自不同背景、具有基本业务研发能力的开发者专注于业务开发本身，无需关注滴滴各种基础设施底层细节。设计需要直观、简洁、智能、个性化。

1. 架构师

让架构师能够非常轻易的调整微服务整体架构，方便像重构代码一样重构微服务整体架构，从而提升架构的可维护性。

1. 公司

提升人效，降低维护成本；提升整体架构稳定性和可伸缩性；简化技术升级难度。提高专业深度，全局打通公司业务。

* 专业深度：由于是多业务垂直化的架构，会有多个团队开发同样的架构，这就需要很多的工程师。每个团队都是用最快速的方式构建流程，所以技术很难做深。这样一来，导致客户端的流畅度不高，后端不稳定，影响可扩展性。
* 全局打通：所有业务本质都是出行，出行本质具有协同效应。如果各自独立发展，业务间就会完全没有协同性。而在构建中台过程中，可以逐步把协同性建立起来。

1. 用户

流畅度、稳定性、扩展性、界面、交易流程等都是影响用户体验的重要因素。

## 2.4 约束条件

１）五个月内完成最初版本

２）投入人员：２００人内

３）使用GO语言实现

４）因为复杂的业务线以及多事业群架构的组织架构，要求系统选择构建的体系结构为：技术中台、数据中台、业务中台。并且技术中台使用大型微服务架构。

## 2.5 非功能需求

除系统功能性需求外，系统的非功能性需求也越来越重要，一个软件产品的性能与非功能性需求有很大关系。非功能性需求往往是针对一个子系统或模块进行的，而不能与每一个功能需求条款一一对应。[1]主要包括质量要求、工程要求和其他要求。质量要求又包括性能，可靠性，可用性，安全性，可维护性，可移植性等；工程要求包括反要求，设计约束和逻辑数据库要求。表 4 给出了质量和可信赖特征对体系结构的约束。

**表3.质量或可信赖属性**

|  |  |
| --- | --- |
| 质量或可信赖性属性 | 体系结构需求 |
| 性能 | 高性能、高并发。对 90%的请求，系统必须在0.5秒以内做出响应。 |
| 资源管理 | 服务器部件的内存必须有 40%以上的预留量。 |
| 易用性 | 客户端支持安卓及ios平台，兼容旧版本。 |
| 可伸缩性 | 系统在高峰时期能处理一千万个并发用户。 |
| 密安性 | 所有通信必须被授权，并使用认证过程进行加密。 |
| 可使用性 | 系统必须 24×364 运行，可使用性达到 99%以上。 |
| 可靠性 | 不允许丢失信息，所有信息必须在 30s 内完成。 |
| 高伸缩性 | 高可用架构首要应对的是大流量且变动复杂场景下的可用性问题。故在建设过程中，架构需要具备高伸缩性，能实现快速扩展。读Cache也是解决大流量带来麻烦的手段。 |
| 高可用 | **故障监测**与**排除、消除达点故障**，**互备和容灾**。 |
| 一致性 | 支付对一致性的要求远远高于其他逻辑，需要进行事务隔离。 |
| 扩展性 | 1. 日常代码的实现需要能够捕捉异常、处理事务。  2. 互联网生下来就是为了服务海量用户，在这个时期，几近没有哪一个利用再为单机而生。每一个公司的每一个产品将要面临的都是不可预知的用户海量要求。明显这个靠散布式程序来解决，比依托单机靠谱很多。如果一开始你的架构设计不可扩大，有再多的机器，有再多的云解决方案，也只不过是将单机程序跑在了1个虚拟的单机上。 |
| 系统性 | 去单库事务，不再依赖于事务。 |
| **容错性** | 机器会故障；当几百人投入研发工作的时候，一定会有人出错。需要保证在小毛病下，不影响业务，保持服务健康运营。 |
| 易维护 | 透明升级，简化代码依赖管理过程。 |
| 高质量 | 快速实现全链路压测常态化，透明实现多环境部署、单测、集成测试等 |

## 2.6 风险

1）对需求变化的适应性：在软件的开发过程中，用户很可能会中途更换需求，或者要求增加新的需求，体系结构文档能否适应新的需求是一个大问题。

2）工期的综合要求：用户通常会对工期提出要求，比如第一个版本在5个月内完工，第二个版本在9个月内完工，第三个版本要在12个月内完工等等。体系结构的设计能否很好的支持上述要求。

3）软件易用性：软件的易用性是影响软件是否被用户接受的关键因素。在软件产品中，设计复杂，功能强大而完备，但因为操作繁复而被搁置者屡见不鲜。造成的主要原因在于缺乏软件开发中软件体系结构的宏观把握能力。

4）软件的可伸缩性：是指软件在不进行修改的情况下适应不同的工作环境的能力。由于硬件的飞速发展和软件开发周期较长的矛盾，软件升级的需要显得非常迫切。如果软件的升级和移植非常困难，软件的生命期必定很短，使得化费巨大人力物力开发出的软件系统只能在低性能的硬件或网络上运行，甚至被废弃不用，造成巨大的浪费。

5）软件的可维护性：软件的维护也是必然的事情，为了保证软件的较长使用寿命，软件就必须适应不断的业务需求变化，根据业务需求的变化对软件进行修改。修改的成本和周期都直接和软件的体系结构相关。一个好的软件体系结构可以尽可能地将系统的变化放在系统的配置上，即软件代码无需修改，仅仅是在系统提供的配置文件中进行适当的修改，然后软件重新加载进入运行状态，就完成了系统部分功能和性能要求的变化。对于重大改动，需要打开源代码进行修改的，也仅仅是先继承原先的代码，然后用新的功能接替原先的调用接口，这样将把软件改动量减小到最低。

# 3. 解决方案

在业务多元化发展的组织中，去构建一套工程架构，构建一套组织结构及对应的管理机制，以保证业务可持续的又快又好的发展。建设过程中，**一方面**架构上要做到分布式化，业务功能域要做到服务化，这样可以保证架构的高可用、高伸缩性。**另一方面**业务架构与组织架构要相匹配，网站流量逐渐增大的同时组织架构与业务架构要随之变化，相互匹配。

选择构建的体系结构为：**技术中台、数据中台、业务中台。**

## 3.1 体系结构概述

滴滴出行将项目分为技术中台、数据中台和业务中台。业务中台背后有提供大量技术支撑的技术中台。

技术中台：

因为**业务线众多**，组织结构成长为**多事业群架构**，每个事业群里面又有很多事业部，下面还有各种独立的部门，一个部门往往会需要提供一个 API 或 broker 服务来屏蔽公司内其他服务对这个部门服务的调用，在逻辑上就形成了由多个独立微服务构成的“**大型微服务**”。

使用大型微服务架构，如果要让项目容易扩展和维护，就必须得解决大量微服务互相调用的问题。如果把一个微服务类比成一个 class，为了能够让这么复杂的体系可以正常运转，就必须给 class 进行更进一步的分类，形成各种 class 之上的设计模式，比如 **MVC**。

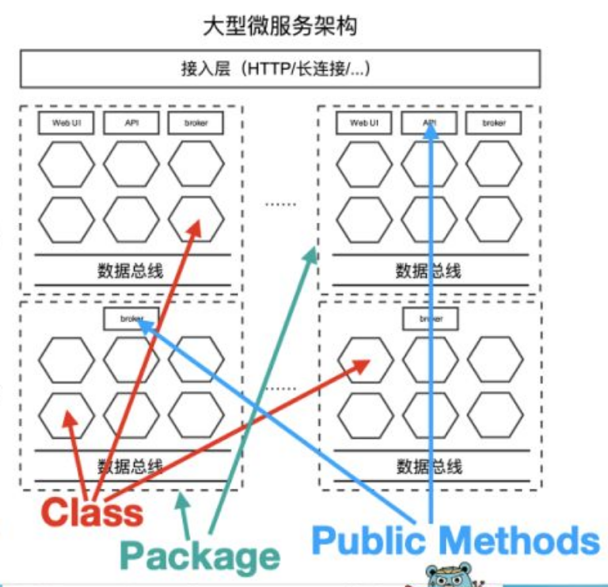
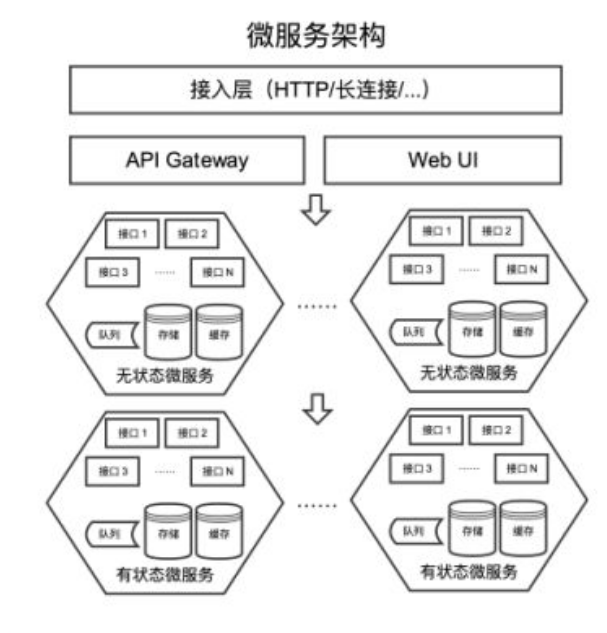


　　　　　图3-1 微服务架构　　　　　　　　　　　　图3-2 大型微服务架构

## 3.2 技术中台

　　提供一些基础的设施，像云基础设施、MQ、NOSQL存储、监控平台等。采用**大型微服务架构**，实现各种微服务框架必有的功能，例如服务治理、水平扩容等。不重复造轮子，而是大量使用已有的技术积累，框架所做的事情是统一并抽象相关接口，让业务代码与具体实现解耦。

设计要点：

1. 从工具链层面来说，让业务无需操心开发调试之外的事情，无缝集成，降低使用难度。
2. 从设计风格上来说，提供非常有限度的扩展度，仅在必要的地方提供 interceptor 模式的扩展接口，所有框架组件都是以“组合”（composite）而不是“继承”（inherit）方式提供给开发者。框架会提供依赖注入的能力，但这种依赖注入与传统意义上 IoC 有一点区别，并不追求框架所有东西都可以 IoC，只在必要的地方有限度的开放这种能力，用来方便框架兼容一些开源的框架或者库，而不是让业务代码轻易的改变框架行为。
3. 大型微服务框架最有特色的部分是提供了非常多的“可靠性”设计。我们刻意让 RPC 调用的使用体验跟普通的函数调用保持一致，使用者只用关系返回值，永远不需要思考崩溃处理、重试、服务异常处理等细节。访问基础服务时，开发者可以像访问本地文件一样的访问分布式存储，也是不需要关心任何可用性问题，正常的处理各种返回值即可。在服务拆分和合并过程中，我们的框架可以让拆分变得非常简单，真的就跟类重构类似，只需要将一个普通的 struct methods 进行拆分即可，剩下的所有事情自然而然会由框架做好。

### 3.2.1 整体架构

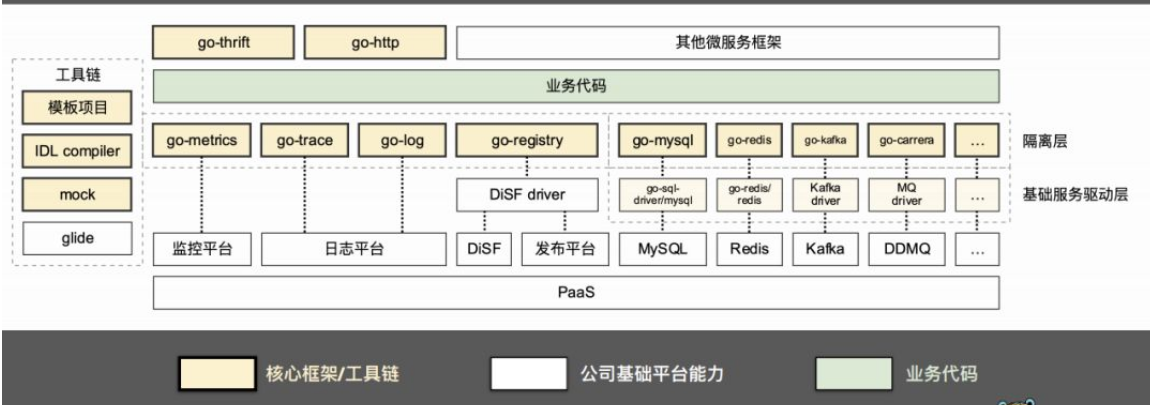


图3-3 技术中台整体架构

上图是框架的整体架构。绿色部分是业务代码，黄色部分是框架，其他部分是各种基础设施和第三方框架。

可以看到，绿色的业务代码被框架整个包起来，屏蔽了业务代码与底层的所有联系。其实框架只做了一点微小的工作：将业务与所有的 I/O 隔离。未来底层发生任何变化，即使换了下面的服务，都能够通过黄色的兼容层解决掉，业务一行代码不用，底层 driver 做了任何升级业务也完全不受影响。

结合微服务开发的经验：微服务开发与传统软件开发唯一的区别就是在于 I/O 的可靠程度不同，各种不同的业务中处理“稳定性”问题，其实归根结底都是类似的问题，本质上就是 I/O 不够可靠。我们的目的并不是要真的让 I/O 变得跟读取本地文件一样可靠，而是由框架统一所有的 I/O 操作并针对各种不可靠场景进行各种兜底，包括重试、节点摘除、链路超时控制等，让业务得到一个确定的返回值——要么成功，要么就彻底失败，无需再挣扎。

实际业务中，使用 I/O 的种类其实很少，也就不过十几种。该框架封装了所有可能用到的 I/O 接口，把它们全部变成 Go interface 提供给业务。

### 3.2.2 框架实现具体细节

1. **框架与业务正交**

* 实现思路：
  + 这个框架没有最基本的框架特征都没有，MVC、middleware、AOP 等各种耳熟能详的框架要素在这里都不存在也不需要。
  + 框架只是设计了一个执行环境，由一堆不关联的基础库组成高度可拓展，它实现了所有业务需要对外暴露的公开方法。业务可独立于框架运行：只需要提供一个入口 type，框架就会自动让业务运转起来。
* 如何实现

同时使用两种技术来实现这一点。

* + 一方面，提供工具链。用于生成最初的项目模板并通过代码生成器实现类似AOP的效果。对于 IDL-based 的服务框架，我们可以直接分析 IDL 和生成的 Go interface 代码的 AST，根据这些信息透明的生成框架代码，在每个接口调用前后插入必要的 stub 方便框架扩展各种能力。
  + 另一方面，在程序启动的时候，基于Go interface的duck-typing特性和通过反射拿到业务 type 的信息，动态生成业务路由。
* 收益
  + 做到了这些事情之后业务开发就完全无需关注框架细节了，甚至可以做到业务像调试本地程序一样调试微服务。
  + 框架本身的升级可以做到完全透明，方便所有服务统一框架版本。

同使用这种方式避免业务思考“版本”这个问题，很多服务框架都因为版本分裂造成了很大的维护成本，当这个框架成为一个开发环境之后，框架升级就变得完全透明。同时，要求业务始终使用最新的框架代码，从来不会使用 semver 标记版本号或者兼容性，这样让框架的维护成本也大大降低。“更大的权力意味着更大的责任”，我们也为框架写了大量的单元测试用例保证框架质量，并且规定框架无限向前兼容。谨慎的开发上线功能，非常收敛的提供接口，从而保持业务对框架的信任。

* 框架的启动逻辑

　　首先创建一个 Server 实例 s，传入必要的配置参数；然后新建一个业务类型实例 handler，这个业务类型只是个简单的 type，并没有任何约束；最后将接口 IDL interface 和 handler 传入 s，启动服务即可。

**在 handler 和 IDL interface 之间加一个夹层**并做了很多事情，这相当于在业务代码的执行开始和结束前后插入了代码，做了参数预处理、日志、崩溃恢复和清理工作。



图3-4 框架的启动逻辑

　　还需要设计一个接口层来**隔绝业务和底层**之间的联系。接口层本身没什么特别技术含量，只是需要认真思考如何保证底层接口非常非常稳定，并且如何避免穿透接口直接调用底层能力。

　　这个接口层的收益是比较容易理解的，可以很好的帮助业务减少无谓的代码修改。开源框架就不能保证这一点，说不定什么时候作者心情好了改了一个框架细节，无法向前兼容，那么业务就必须跟着做修改，而内部框架则一般不太改接口。

1. **隔离层屏蔽业务与底层的联系**

* 如何实现
  + 为所有基础服务（mysql/redis/mq/es/...）定义 interface，业务只允许调用 interface 的方法
  + 基于 SPI 设计思路，提供基础服务的工厂，动态实例化对应 interface
* 收益
  + 可透明的升级服务驱动，快速在大量服务中实现共性逻辑或者修复共性问题
  + 透明的管理基础服务的资源（长连接、mysql cursor 等），避免出现资源泄露
  + 统一控制重试、超时、服务发现、故障摘除逻辑，业务无感知且不易出错，提升整体稳定性
  + 对所有基础服务提供了 mock 能力，可以实现 AOP 能力
* 案例：Redis 接口



图3-5 隔离层屏蔽业务与底层的联系实例

左边是 github.com/go-redis/redis 代码（简称 go-redis），这是一个非常著名的 Redis driver；右边是我们的 Redis 接口设计。

我们的接口设计相比 go-redis 则更加贴近本质－－ Redis 协议本身。Redis 在设计各种命令时非常严谨，做到了极为严格的向前兼容，无论 Redis 从 1.0 到 3.x 如何变化，各个命令字的协议从未发生过不兼容的变化。因此，严格参照 Redis 命令字协议设计了 Redis 接口，连接口的参数名都尽量与 Redis 官方保持一致，并严格规定各种参数的类型。

## 3.3 数据中台

### 3.3.1 整体架构



图3-6 数据中台架构基础版

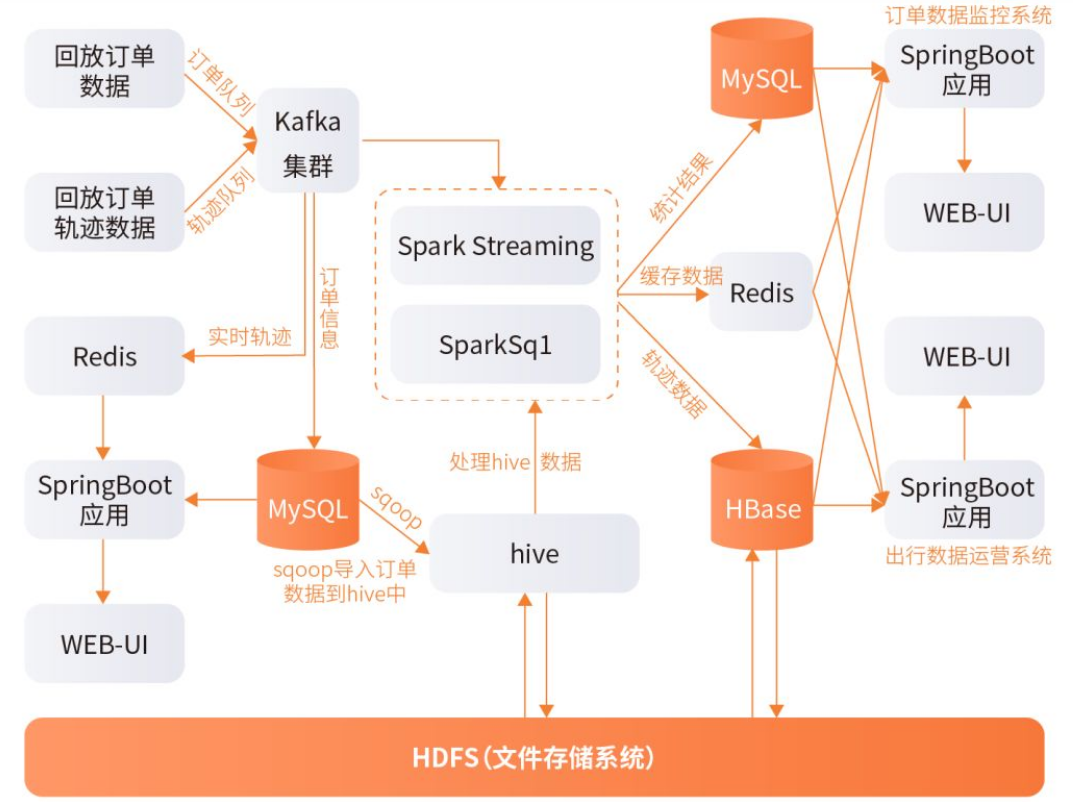


图3-7 数据中台架构详细版

## 3.4 业务中台

业务中台是依托于技术、数据中台，更贴近于业务，主要是对业务负责。滴滴的业务中台是直接是从最大的业务中孵化出来的。订单中心、计价中心、支付中心、passport、用户中心、触达平台等都是配置孵化而来，对于目前不需要中台的业务线，通过沟通机制划定边界。



图3-8 业务中台架构

　　可以看到，整个的架构设计分几个边界的上下文，好处在于把相关性不强的逻辑拆开，同时在一个相关性下面，通过分层对业务进行更好的建模。**调度层**作为入口去牵引多个业务线，**业务流程层**为调度层做服务，**状态智能层**用来支持上面的两层。

在对业务和产品进行更好建模的基础上，进行了“五化”：服务化、异步化、配置化、插件化、数据化。解决了传统的系统架构问题，做到高耦合和内聚，提高迭代。配置化和插件化解决灵活性问题，把灵活性开放给不同团队。数据化实际上是中台赋能业务，有中台的赋能才能变得更好。

### 3.4.1 服务化

服务之间的**协议和规范**要建立好。注意控制力度，力度太小、太大都会有问题。随着时间的发展，服务化本身要不断的演进。

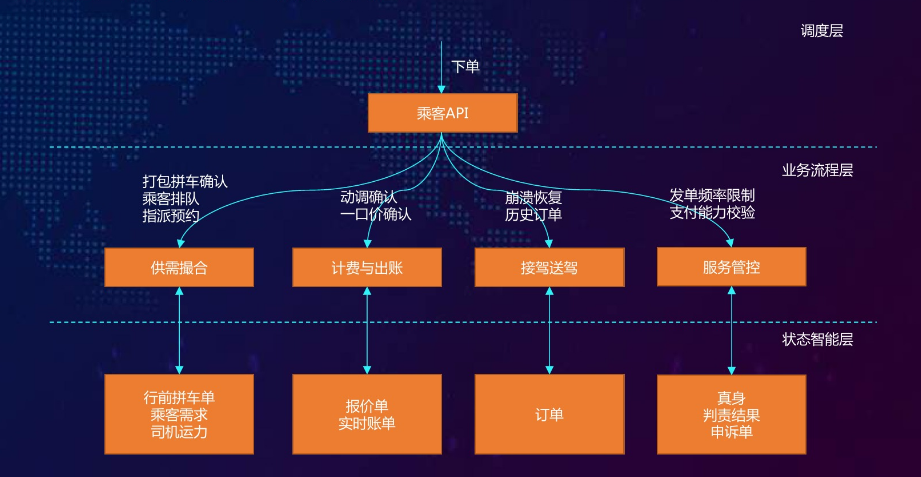


图3.9 下单

**下单**流程能够**调用**很多**服务**，在多个层次，以**接口**层次结果进行拆解。

### 3.4.2 异步化

对每个事件的非核心或不需要实时反馈给客户端的逻辑进行拆解，核心的主流程会变简洁。对非核心的逻辑在事件上做订阅之后，进行二级处理。



图3.10 结束订单

结束订单的时候有很多逻辑要做，但是都是通过 **MysqlBinlog** 处理或 **MQ** 处理。

mysql-binlog是MySQL数据库的二进制日志，用于记录用户对数据库操作的SQL语句（除了数据查询语句）信息。

MQ：生产者者往消息队列中写消息，消费可以读取队列中的消息。

### 3.4.3 配置化

**服务化**和**异步化**能解决很多迭代效率的问题，但由于系统、业务的复杂性，各个业务都有些差异，体现在不同的产品线、城市、区域、时间等等。

**配置化**的核心是对这些进行**建模**，把每个对象模型化，抽象成 ID，在不同的服务化里把这些可配置的能力进行**抽象**。

具体抽象过程，如下图：



图3-11 出行业务抽象过程

第一级抽象采用的是类 iptables 的规则引擎判定产品分类，第二级的规则引擎由模块自定义。所有配置化都是用的自生成平台，要配置什么，自定义配置即可，这个过程是动态进行的。

当前业务中台可以支持上千个配置点，比如不同层次的计价规则不一样、不同产品线的车样子不同、不同的场景，如拼车和接送机，管控规则也不一样等等。

### 3.4.4 插件化

**配置化**解决的是**业务线差异**问题，但如遇到**逻辑差异**较大的情况，就要做插件，统称为 **FPI**。在 FPI 的能力上，不同的团队可以开发很多插件，在特定的**配置点**下，对它的**逻辑**进行加载。

真正业务流程到这儿，可以调起它对应的插件做出来。对于一些没有差异化需求的业务，可以用开发的 **default 逻辑**，这是更极端的灵活性的体现。

有灵活性的体现后，团队还可以做一些组织上的调整。原来每个服务或者平台是一个垂直化的架构，有些团队是横向，是 FT，有些 FT 是接送机 FT，专门做接送机的事情。

**通过插件的形式在每个系统加载它的插件**，它就可以跟着业务思考、跟着产品思考这个业务该怎么走、这个产品怎么演化。

相对的逻辑是更加专注的，这也带来很好的组织结构对中台的适应性。

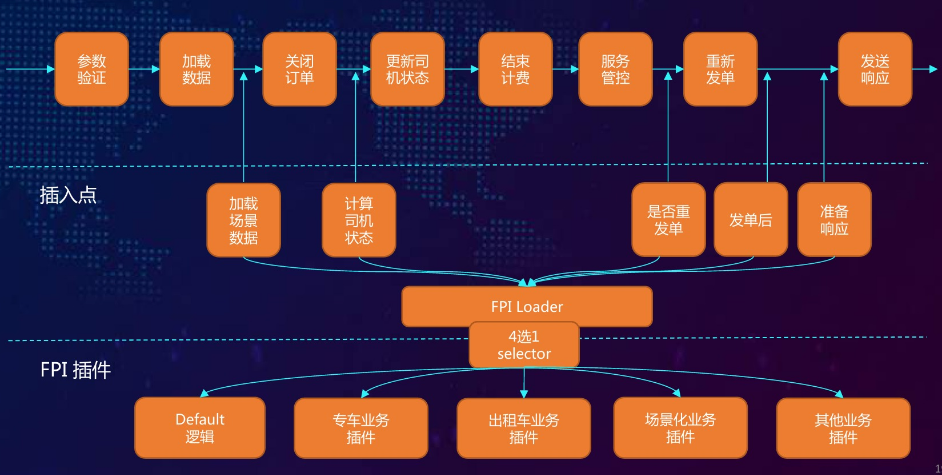


图3-12 系统加载插件过程图

### 3.4.5 数据化

在大数据时代，数据是不得不考虑的问题，所以在业务中台，要实现全局打通，本质是要把数据打通。

数据化方面，需要注意以下三方面：

１）让数据更加规范和标准化。

２）构建完整的数据流，从**在线**到**离线**，从**日志**到**模型**的在线使用。

３）引入机器学习的算法、人工智能的算法去构建在线数据智能的决策。

所以需要制定离线分析与在线决策的方案，如下图：

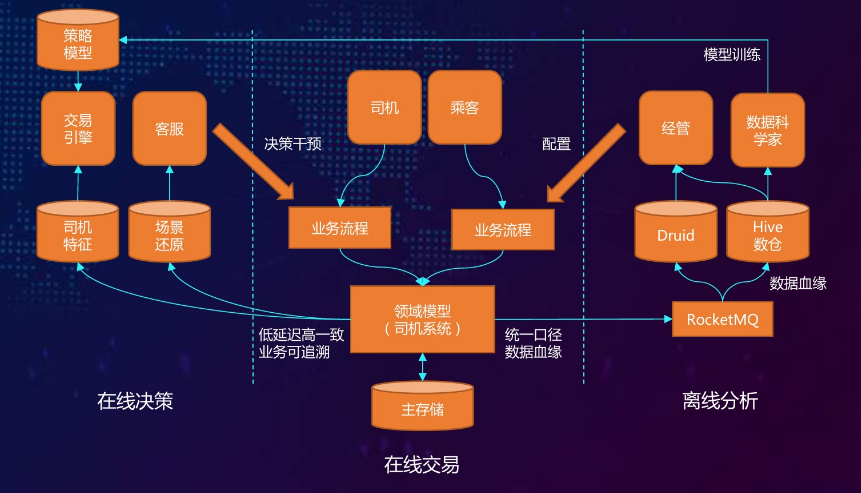


图3-13 离线分析与在线决策方案

第一个是离线做分析，可以做数据血缘、模型训练，同时可以把它放到在线决策层面，构建很好的智能客户引擎和交易引擎，这个可以干预，因为干预可以让升舱或者多业务线的清单成为可能。因为有这样的决策，使在线服务的管控和判断做得更加智能。

# 4. 系统的质量分析和评价

除系统功能性需求外，系统的非功能性需求也越来越重要，一个软件产品的性能与非功能性需求有很大关系。非功能性需求往往是针对一个子系统或模块进行的，而不能与每一个功能需求条款一一对应。[1]主要包括质量要求、工程要求和其他要求。质量要求又包括性能，可靠性，可用性，安全性，可维护性，可移植性等；工程要求包括反要求，设计约束和逻辑数据库要求。

## 4.1 场景分析

系统的使用场景与质量属性是密切相关的，也是决定体系结构的重要依据。通过场景  可以很好的评估体系结构，本文使用 SEI 开发的体系结构折中分析方法(Architecture   Tradeoff Analysis Method).

### 4.1.1 用例场景

用例场景是从使用者角度出发，描述用户所期望的与这个系统的交互。下面是用例场景的几个例子。

**场景１：乘客叫车**

1. 乘客在打开滴滴之后第一屏页面就是打车页面（不需要进行额外跳转）。
2. 点击屏幕上方选择出行方式：出租车、快车、专车、豪华车。
3. 选择打车时间：预约　或　现在。
4. 在下屏选择目的地。
5. 选乘车人：人数（次步骤可跳过）
6. 确认呼叫

整个过程，从地图上会显示：

1. 乘客周围的车辆
2. 车辆和你的距离
3. 到达时间的预估
4. 车费的预估
5. 在线同步最短的行驶路线
6. 车辆行驶过程中每分每秒是如何扣费



图4-1 界面示意图

该场景表明了用户期望系统易于使用，即易用性。

**场景２：司机结束订单后系统重新发单**

1. 系统派单
2. 司机确认订单
3. 到达目的地后，司机关闭订单
4. 系统计算司机状态后，更新司机状态，结束计费
5. 系统重新发单

整个过程，从地图上会显示：

1. 车辆周围的乘客
2. 乘客和你的距离
3. 到达时间的预估
4. 车费的预估
5. 在线同步最短的行驶路线
6. 车辆行驶过程中每分每秒是如何扣费

**司机**希望可以由系统派单，并且可以看到这一趟行程的费用情况。该场景表明了用户期望系统易于使用，即易用性。

**场景３：**用户在下单时，期望可以在0.5s内获得结果。该场景表明了系统的性能需求。

**场景４**：当用户改变图形布局后，屏幕要在0.5s内画出来。该场景表明了系统的性能需求。

**场景５：**当处理器发生故障后，缓存系统要能在1s中内从一个处理器切换到另一个处理器，该场景代表了系统可靠性要求。

### 4.1.2 用户用车业务流程图

1. 涉及系统:后台系统、前端页面；角色类型：用户和商家。
2. 从用户开始叫车到到达目的地，可以将这个过程分为三个阶段，分别为：用户下单阶段、派单阶段、行程开始——结束这三个阶段。

每个阶段具体的流程和交互环节如下图所示：



图4-2 用户叫车流程和交互环节

### 4.1.3 任务流程图



图4-3 任务流程图

### 4.1.4 增长性场景

**增长性场景1：**未来系统所使用的开源工具代码更改或版本升级，只需在框架方面调整，业务代码无需更改。希望系统仅需增加2人周的工作量就能完成修改。

**增长性场景2：**未来可能会增加新的数据服务器，这将会降低用户的访问时间，希望系统仅需增加1人周的工作量就能完成修改。

**增长性场景3：**通过扩充现有数据库表的规模和在特定的列上创建索引，可以降低平均检索时间到0.1s以内。

### 4.1.5 探索性场景

**探索性场景1**：核心支付效果有损，收银熔断、乘客未支付可以发单

**探索性场景2**：核心主流程效果有损，发单限流、内部丢单

**探索性场景3：**正常情况下，当服务器有一半宕机时，不影响整个系统的可用性。

**探索性场景4：**当系统的数据库出现问题时，可以通过日志文件进行恢复，损失的数据不超过5分钟。

## 4.2 原型分析

### 4.2.1 原型评价法

1）写一个测试程序，作为司机用户，在不断地调用查询条件1000万次，看其需要多长时间。

2）写一个测试程序，作为乘客用户（不同类别的管理员均包含在内），在不断地调用查询插入语句10亿次，看其需要多长时间。

### 4.2.2 效用树法

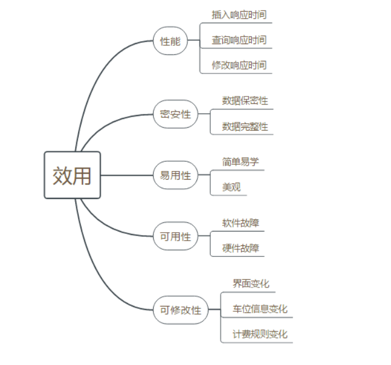


图4-4效用树图

## 4.3 风险

1）项目进度要求：产品的第一个版本必须在6个月内完成，而且第二个版本要在九个月内完成，第三个版本要在12个月内完成。本体系结构是否可以很好的支持进度要求。

2）项目需求变化：在开发过程中用户经常会要求改变需求或者增加需求，此时，体系结构是否还能适用。

3）软件易用性：软件的易用性是影响软件是否被用户接受的关键因素。在软件产品中，设计复杂，功能强大而完备，但因为操作繁复而被搁置者屡见不鲜。造成的主要原因在于缺乏软件开发中软件体系结构的宏观把握能力。

4）软件的可伸缩性：是指软件在不进行修改的情况下适应不同的工作环境的能力。由于硬件的飞速发展和软件开发周期较长的矛盾，软件升级的需要显得非常迫切。如果软件的升级和移植非常困难，软件的生命期必定很短，使得化费巨大人力物力开发出的软件系统只能在低性能的硬件或网络上运行，甚至被废弃不用，造成巨大的浪费。

5）软件的可维护性：软件的维护也是必然的事情，为了保证软件的较长使用寿命，软件就必须适应不断的业务需求变化，根据业务需求的变化对软件进行修改。修改的成本和周期都直接和软件的体系结构相关。一个好的软件体系结构可以尽可能地将系统的变化放在系统的配置上，即软件代码无需修改，仅仅是在系统提供的配置文件中进行适当的修改，然后软件重新加载进入运行状态，就完成了系统部分功能和性能要求的变化。对于重大改动，需要打开源代码进行修改的，也仅仅是先继承原先的代码，然后用新的功能接替原先的调用接口，这样将把软件改动量减小到最低。

# A. 附录

## A.1 附录一：表对应页码

表1 定义、术语------------------------------- 5

表2.体系结构用例---------------------------- 8

表3.质量或可信赖属性---------------------- 9/10

## A.2 附录二：图对应页码

图3-1 微服务架构------------------------------ 11

图3-2 大型微服务架构------------------------ 11

图3-3 技术中台整体架构--------------------- 12

图3-4 框架的启动逻辑------------------------ 14

图3-5 隔离层屏蔽业务与底层的联系实例-- 14

图3-6 数据中台架构基础版-------------------- 15

图3-7 数据中台架构详细版-------------------- 15

图3-8 业务中台架构----------------------------- 16

图3.9 下单----------------------------------------- 17

图3.10 结束订单---------------------------------- 17

图3-11 出行业务抽象过程----------------------- 18

图3-12 系统加载插件过程图-------------------- 19

图3-13 离线分析与在线决策方案---------- 19

图4-1 界面示意图--------------------------------- 21

图4-2 用户叫车流程和交互环节---------------- 22

图4-3 任务流程图---------------------------------- 23

图4-4效用树图------------------------------------- 25