|  |  |
| --- | --- |
|  | 学校代码：10246 |
|  | 学 号： |



|  |
| --- |
| 硕 士 学 位 论 文 |

（专 业 学 位）

|  |
| --- |
| **境外度假系统设计与实践**  **Study on Travel System** |

|  |  |
| --- | --- |
| 院 系： | 软件学院 |
| 专 业： | 软件工程 |
| 姓 名： | 佘山 |
| 指 导 教 师： | 朱东来 |
| 完 成 日 期： | 2018年1月15日 |

# 目 录

[目 录 I](#_Toc492487148)

[摘 要 II](#_Toc492487149)

[ABSTRACT III](#_Toc492487150)

[第一章 绪 论 1](#_Toc492487151)

[1.1 课题研究宏观背景 1](#_Toc492487152)

[1.2 课题研究内容和意义 2](#_Toc492487153)

[1.3 论文框架结构 3](#_Toc492487154)

[第二章 相关工作与关键技术研究 5](#_Toc492487155)

[2.1 相关工作研究 5](#_Toc492487156)

[2.2 业务背景简介 6](#_Toc492487157)

[2.3 关键技术研究 7](#_Toc492487158)

[第三章 系统需求分析与总体设计 9](#_Toc492487159)

[3.1 项目背景介绍 9](#_Toc492487160)

[3.2 系统需求分析 10](#_Toc492487161)

[3.3 系统总体设计 12](#_Toc492487162)

[第四章 系统详细设计与实现 14](#_Toc492487163)

[4.1 产品子系统的设计 14](#_Toc492487164)

[4.2 订单子系统的设计 16](#_Toc492487165)

[4.3 交易子系统的实现 18](#_Toc492487166)

[4.4 管理子系统的实现 20](#_Toc492487167)

[4.5 数据库设计 22](#_Toc492487168)

[4.6 系统用例介绍 25](#_Toc492487169)

[第五章 总结与展望 27](#_Toc492487170)

[5.1 本文内容总结 27](#_Toc492487171)

[5.2 未来工作展望 28](#_Toc492487172)

[参考文献 29](#_Toc492487173)

[致 谢 31](#_Toc492487174)

# 摘 要

首先分析预付费卡行业的现状，明确了系统系统扩展性的不足（或问题）。在此基础上，应用Spring Cloud微服务技术，先做…，然后…。最后…。其中重点做。

在互联网场景中，企业业务增长和变化的速度都非常的快，其所依赖的业务系统规模也飞速增加，传统的垂直单体架构已经很难支撑未来业务的不断发展，其扩展性弱，可维护性差的问题非常明显。于此同时，软件系统架构服务化的思想发展越来越深入，整个软件交付链路上各环节的基础设施越来越成熟，诞生了微服务这一服务化思想的最佳实践，其通过分治的思想可以很好降低业务系统的复杂性，通过合理的水平业务拆分，提供高内聚低耦合的服务。于此同时，我司根据营销和支付等业务的需求，要求设计与实现一个预付卡系统，用以完善整个我司OTA系统的生态环境，保证了用户在一个闭环的场景下，完成旅行商品的购买，支付等流程，增加客户对于整个系统的使用粘度，提高公司整体的绩效水平。因此如何有效的将微服务的思想应用到系统的设计与实践中将变得非常有价值，力求提供高水平的服务质量和10X的交付水平，达到通过技术反哺业务的目的。

本文将对微服务的思想进行简要的介绍，并深入剖析微服务落地过程中所带来的机遇与挑战。然后将根据我司自身的整体IT环境和预付卡系统的使用场景，结合对当前市面上为数不多的微服务框架的分析与比较，并吸取其他企业实践过程中的经验和教训，做出合适的选型决策。之后遵循合理的业务边界划分，提供例如生产、订单、交易等职责单一的服务，并通过相应的协调机制实现服务间的通信。于此同时，也将对以上各部分服务内部的设计和实践难点进行剖析与介绍，最终提供一个完整高效的预付卡系统。

关键词

# ABSTRACT

[Prepaid card](http://dict.youdao.com/w/prepaid%20card/#keyfrom=E2Ctranslation) information system.

test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test

test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test test

Keywords [prepaid card](http://dict.youdao.com/w/prepaid%20card/#keyfrom=E2Ctranslation) information system, Microservice

# 绪 论

# 1.1 课题研究宏观背景

测试测试测试测试。

# 1.2 课题研究内容和意义

第一段应突出虽然本文研究内容是趋势，但目前仍有许多问题待解决，在互联网场景中，企业业务增长和变化的速度都非常的快，其所依赖的业务系统规模也飞速增加，传统的垂直单体架构已经很难支撑未来业务的不断发展，其扩展性弱，可维护性差的问题非常明显。于此同时，软件系统架构服务化的思想发展越来越深入，整个软件交付链路上各环节的基础设施越来越成熟，诞生了微服务这一服务化思想的最佳实践，其通过分治的思想可以很好降低业务系统的复杂性，通过合理的水平业务拆分，提供高内聚低耦合的服务。于此同时，我司根据营销和支付等业务的需求，要求设计与实现一个预付卡系统，用以完善整个我司OTA系统的生态环境，保证了用户在一个闭环的场景下，完成旅行商品的购买，支付等流程，增加客户对于整个系统的使用粘度，提高公司整体的绩效水平。因此如何有效的将微服务的思想应用到系统的设计与实践中将变得非常有价值，力求提供高水平的服务质量和10X的交付水平，达到通过技术反哺业务的目的。

预付卡的个人化和互联网化将更加深化

2015年预付卡市场在普遍不景气中孕育新的变化，预付卡的个人化和互联网化更加深化。在此过程中，通过实现预付卡的互联网化，结合社交、虚拟化属性，预付卡的互联网销售潜力被充分挖掘，出现售卡量超过5亿级的垂直类专业售卡平台（如图表46所示），这是在以往单一以团购为主的售卡点平台难以达到的目标。随着互联网对预付卡产业的改进，预付卡的个人化和互联网化将更加深化。

二、预付卡的虚拟化应用进一步发展

  传统实体零售企业的会员政策在执行过程中会遇到各种问题，从而影响会员的到店和消费体验，而预付卡的虚拟化或可以丰富发卡企业对会员的营销，这种具有预付式现金价值的营销工具属性已经在部分发卡企业的营销工作中取得进一步的发展。

在预付卡业务发展过程中，有部分企业结合预付卡的虚拟化应用，开发基于微信（移动端）的卡券应用可以实现电子礼品卡的售卖、充值、转赠与消费。这一功能的实现，有助于扩大发卡企业营销客户的基础，实现对新客户的引流，进而为结合消费者行为制定分层营销策略提供帮助。

基于预付卡虚拟化应用服务是发卡企业下一个重点关注的领域。预付卡的虚拟化应用也为发卡企业对电信、银行等合作渠道提供帮助，有利于电信、银行积分在实体发卡企业的兑换和应用，这一服务领域公司的商业价值在资本市场上得以体现。6月23日通鼎互联（002491）[20]公告显示，公司正在推动对浙江微能科技有限公司100%股权的收购，该标的公司是一家典型运营商积分兑换的运营商，在此次收购中通鼎互联给与4.8亿元的估值，是其2015年营业收入的15.4倍，是其2015年净利润的131倍。

三、预付卡的营销大战愈加剧烈

随着商业竞争程度的进一步加剧，越来越多的发卡企业考虑加强预付卡营销力度，并从组织架构层面进行规划和梳理，或营销策略属地化，或缩减审批流程为预付卡的营销提供保障。

从营销本身来说，发卡企业结合各种营销手段（如摇一摇、红包、分享、会员返利）层面营销和推广预付卡（如图表47所示）；从渠道合作方面实体店与电商的合作方式拓展了预付卡的销售渠道和范围（如图表48所示）。

四、发卡企业构建基于预付卡为基础的支付体系

正如我们在本报告第一部分中实体零售企业所采取的应对策略中提到的那样，发卡企业除了在重新构建其商品的经营能力、客户的经营能力以及供应链的经营能力，在此过程中基于预付卡支付体系成为打造核心竞争力的重要一环，预付卡成为其经营的引流入口，结合流量导入平台、移动端、促销平台的各种资源投入（如图表49所示），将客户引进来，留下来，进而通过各种服务手段将顾客变为自己的忠实客户。

五、预付卡的账户安全问题凸显

随着电子支付产业规模的进一步发展，预付卡因不记名或客户关注度不够，还有部分发卡企业在预付卡账户安全管理方面有漏洞而被不法分子所利用。不法分子非法入侵账户转移资金的案件时有发生，预付卡的账户安全问题凸显。

据不完全统计，仅2015年国内几个知名电商平台的账户支付体系就因账户安全问题被媒体爆出。预付卡的虚拟化离不开对账户安全的保障，建立预付卡互联网交易方式的安全机制需要广泛关注和重视，并通过切实可行的举措进行杜绝和防范。

随着移动互联网的发展和电子支付的渗透，特别是在电商企业通过发行电子礼品卡的示范效应下，预付卡的电子化已经如火如荼发展起来，国内部分实体零售企业也开始尝试预付卡的电子化，并取得了一定的成效。

预付费卡电子化是消费者主权至上的必由之路

目前，以支付宝、微信支付为代表的移动支付方式，越来越受到消费者的青睐。21至29岁用户占移动支付用户数量的43%[21]，这部分顾客是未来消费的主力。各家零售商都纷纷在自己的实体卖场布局移动支付，希望能够为顾客提供更多便捷的支付体验，通过预付卡的转赠和营销功能丰富营销手段，进而抢占市场，基于此预付卡的电子化可以使消费者的消费更加随性。另一方面，移动支付也在更加深入的影响我们的生活，两者之间的融合使得顾客越来越离不开移动支付。目前，越来越多的商户会员卡已经实现了电子化，使得顾客对移动支付有更大的黏度，所以预付卡的电子化是体现消费者主权至上的必由之路。

重难点，试测试测试测试测试。

# 1.3 论文框架结构

论文根据携程预付卡业务的市场发展需求，以微服务为基础。

11111

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

# 第二章 相关工作与关键技术研究

# 2.1 相关工作研究

微服务的应用，预付卡系统的发展

测试。

# 2.2 业务背景简介

预付卡是什么样的业务？如何赚钱

预付卡能够高效完成市场资源配置，实现商家和消费者的共赢，其对于商户方面有如下好处：

预付卡通过先收款，再消费的方式，可以提前回笼资金，钱款拖欠大幅减少，降低流动性风险。

1．预付卡的相关概念

a).预付卡，全称预付费卡：是指以营利为目的发行的、在发行机构指定范围内购买商品或服务的预付价值，包括采取磁条、芯片等技术以卡片、密码等形式发行的电子支付卡片。

按字面意思理解，就是先付费，后消费的支付卡片。

b).备付金：本文所指备付金，是指客户备付金，和指商业银行为应付日常客户提取现金的业务而保留的一定额度的现金资产所指的备付金不同。他是指客户预存或留存在支付机构的货币资金，以及由支付机构为客户代收或代付的货币资金。

他主要包括：

（一）收款人或付款人委托支付机构保管的货币资金；

（二）收款人委托支付机构收取、且支付机构实际收到但尚未付出的货币资金；

（三）付款人委托支付机构支付、但支付机构尚未付出的货币资金；

（四）预付卡中未使用的预付价值对应的货币资金。

c).预付卡按发卡人不同可划分为两类：

多用途预付卡：指由专营发卡机构发行，可跨法人使用的预付卡种类。如商通卡、联华ok卡等。都可以在不同的签约商户处消费。

单用途预付卡：指由商业企业发行，只能在本企业或同一品牌连锁商业企业购买商品、服务，不得跨法人使用的预付卡种类。如食堂饭卡，游戏充值卡，电话卡等。

d).两种预付卡的不同

多用途预付卡的资金结算是通过具有支付业务许可证的第三方来结算，而单用途预付卡的资金是自行结算，不通过第三方支付机构。

多用途预付卡受央行监管，而单用途预付卡受是商务部监管的。

2011 年 5 月，国务院办公厅转发了七部委《关于规范商业预付卡管理的意见》，首次明确了商业预付卡的地位、作用和分类，明确了分类监管的思路。即：多用途卡由人民银行进 行监管、单用途卡由商务部进行监管；明确了发卡和购买、发票和财务管理、资金管理与业务管理等制度框架。

此外，《关于规范商业预付卡管理的意见》将预付卡分为单用途卡和多用途卡两大类，并实行由人民银行和商务部分类监管的政策。并且依据目前的监管政策，商业银行尚不具备发行预付卡的资格，这在一定程度上避免了来自传统金融体系的巨大冲击，为第三方支付企业的发展营造了良好的条件。在这种情况下， 预付卡企业与以银行为代表的传统金融体系的合作将逐渐增多，旨在营造一个合作共赢的行业环境。

目前购买预付卡的客户多数以企业为主，主要用于解决员工福利、公务消费、商务礼品等方面的需求。这些企业集中在政府部门、企事业单位、金融企业、公关公司以及私营企业等。其中政府部门、事业单位等已经成为各发卡机构关注的重点。而个人消费者用户在发卡机构的客户比例中占据较小的份额。

艾瑞咨询的线上调研数据显示，大型超市、商场百货是中国网民用户使用预付卡的主要 场景，分别占比 59.5%和 15.7%，而除了两大线下场景之外，网络购物成为第三大消费场景， 占比达到 12.8%。另外，便利店、餐饮娱乐场所、酒店旅行、健身也是用户频繁使用预付卡的重要场景。

艾瑞咨询整理研究数据发现，2010 年中国商业预付卡市场规模达到 15936.5 亿元（包括校园卡、游戏点卡、会员卡、加油卡和通讯充值卡在内）。商务部于 2011 年初对部分地区 300 多家商业企业的调研数据测算，中国商业预付卡销售规模为 14203.33 亿元（不含校园卡、游戏点卡、会员卡、加油卡和通讯充值卡），预付卡消费规模达到 10399.58 亿元。

三、预付卡盈利模式分析

1. 预付卡的优势

1.1.预付卡能够高效完成市场资源配臵，实现商家与消费者的共赢。其中对于商户方面（包括单用途卡发卡企业）有如下好处：

a).降低财务风险——预付卡采取先收款，再消费的方式，提前回笼资金，钱款拖欠大

减少，降低流动性风险；

b).提高结算效率——使用预付卡，现金收付、账目收支大量减少，既可提高效率，又可减少人工和设备的费用；

c).绑定客户群体，稳定销售——由于预付卡的使用范围的限定性，所以从持卡人买卡开始实际上就已经跟商家签订了一个一对一或一对多的买卖协议；

d).营销渠道的有效补充——由于预付卡使用范围的限定性，商家可以通过预付卡的销售间接促成对产品的销售，因此构成了对产品营销渠道的有效补充；

e).提升品牌价值，统一品牌管理——一方面商家可以通过品牌影响力较大的第三方企业发售预付卡，借以提升自身的品牌价值，另一方面商家也可以通过对预付卡的统一形象管理形成自己的品牌形象。

1.2.对于预付卡用户：

a).方便安全——预付卡方便办理又易于携带，可减少现金的携带量，购物后可显示余额，不用找零，极大的提高了消费者购物的便利性；

b).经济实惠——部分预付卡采取了会员卡的管理机制，可以进行积分或者打折，给消费者购物提供了一定的实惠；

c).避税——监管措施尚未完善的情况下，企业通过预付卡发放员工福利，可规避企业所得税和员工个人所得税；

2. 监管空白时期的预付卡盈利模式分析

由上面的预付卡市场发展史可以看出，在2006年到2011年之间，预付卡一直处于高速发展的态势。分析认为，2006 年至今之所以呈现高速发展的态势主要出于如下几点原因：

市场监管的宽松

2006 年商务部和国务院先后认定预付卡“债权凭证”的市场地位，使其身份合法化。而在此之后对于第三方机构发行的多用途卡，除了零星的登记之外，基本处于监管真空状态。这为预付卡行业的迅猛发展提供了条件。

市场需求的催生

预付卡使用领域的不断增加、功能的特殊性以及给消费者带来的实惠，使预付卡不断被个人和企业用户所接受。与此同时，因为监管的空白，预付卡开始被用于行贿受贿，偷税漏税等违法行为。

巨额利益的驱使

企业开展预付卡业务，首先可以获得巨额的备付金，部分企业通过备付金的资本化运作带来了巨大的收益；其次，由于预付卡的标的物不是某种固定类别的商品，可按税率较低的商品开具商业发票，承担较低的纳税义务；第三，卡内残余资金也是企业的另一变相收入。

现今预付卡市场盈利模式分析

目前预付卡发卡企业的主流盈利模式分为如下几个方面

a).单用途卡发卡企业发行预付卡的直接目的是促进销售、提前回收资金防范财务风险。因此，单用途卡发卡企业的直接盈利来源以备付金收入和过期沉淀资金为主。

而多用途卡发卡企业的主营业务即为预付卡的发行和受理，其盈利模也更为丰富。主要的可以分为以下四大块。

a).备付金收入

《关于规范商业预付卡管理的意见》颁布前，备用金事实上可以由发卡企业自由支配，一部分资金成为企业的“零息贷款”用于企业经营；部分资金流入资本市场以获得高额收益，因此形成了较高的企业信用风险。

新政颁布后，发卡企业需将备付金存于第三方存管银行，银行账户上的沉淀资金将为企业带来一定的利息收益。因此，此项收益的高低往往取决于发卡规模和沉淀资金规模

b).商户返佣

即商户支付的交易佣金。商户返佣的分配机制平均为发卡方得 50%，收单方得 15%，机具方得 15%，拓展方得 20%。而作为发卡企业方，通常涵盖发卡、收单、机具、业务拓展中的三到四c).卡内残值

根据《关于规范商业预付卡管理的意见》的相关规定，记名商业预付卡将不设有效期，不记名商业预付卡有效期不得少于 3 年，对于超过有效期尚有资金余额的，发卡人应提供激活、换卡等配套服务。

艾瑞咨询认为，监管文件发布前，消费者放弃的卡押金、卡内剩余金额，都沉淀成为发卡公司的纯利润，而伴随着相关文件的出台，这项收入在未来将无法实现。对于不记名预付卡，消费者在预付卡过期之后向商家缴纳的延期手续费将成为发卡企业的又一收入来源。目前行业的平均收费标准为 10 元/次。售卡手续费消费者在购买预付卡的同时，需要缴纳卡片金额 1%-3%（最高不超过 15 元/张）的购卡手续费,这一部分收入在扣除 0.7 元左右的卡片成本后成为企业的净利润。另外，也可以类似银行那样，收取小额账户管理费。

d).创新盈利模式：

 通过折扣、通用积分等服务获取的中间利润发卡企业以其承诺的购买力向商家争取较大的折扣空间，然后在此基础上降低一定的折扣空间让渡给消费者，从而赚取两个折扣的差额利润。

 其他增值服务如发卡企业为企业提供福利解决方案、向其他发卡企业提供发卡系统和机具以及预付卡

e).行业咨询等业务的收入。

项业务。

1.预付卡业务行业市场预测

通过对 2010 年预付卡支付市场的观察认为，未来几年我国预付卡市场将呈现高速增长的态势，多用途卡的增长速度将明显超过单用途卡，监管政策的出台不会对行业造成普遍预期的巨大影响。因为只有在所有企业都能够在良好的监管下开展业务，才能营造一个公平合理的市场环境。已获牌照的发卡企业将会因为竞争对手的牌照和地域限制而加速扩张，同时，由于盈利模式的限制，发卡企业将加大发卡力度以提升整体盈利水平。另外，二三线城市将成为发卡企业未来市场拓展的主要方向。

2.预付卡业务行业发展趋势

a).商业预付卡与电子商务融合趋势明显。

随着互联网、手机支付、固话支付、电视支付应用场景的多样化，预付卡消费将会为消费者提供更为丰富的服务产品。团购、换购等新兴电子商务消费模式的兴起为预付卡的发展提供了应用条件，并进一步助力新兴消费模式的发展。预付卡与电子商务的融合促使预付卡支付渠道进一步拓宽，预付卡功能得到拓展，公共事业费缴费、网络商城等电子商务模式与预付卡的结合为持卡人提供更丰富的应用体验.

b).此外，行业交流合作将不断加强。

随着国内支付峰会、预付卡行业峰会等交流平台的发展和完善，行业自律组织及协会力量的形成，在一些中介力量的积极参与下，国外的研究机构和投资者将会更加关注中国预付卡市场的发展，更多的国内及跨国合作有望展开。

测试。

# 2.3 关键技术研究

微服务概念，

现在，微服务是继SOA(面向服务的软件架构)之后的越来越流行的架构模式。

现在我们来看下一个微服务应该具有的原则

1.单一职责原则

单一职责原则是5个设计原则中（[SOLID design pattern](http://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/#SRP).）的一个，它表明：一个单元（一个类、函数或者微服务）应该有且只有一个职责。

无论如何，一个微服务不应该包含多于一个的职责。

2.围绕业务构建微服务

微服务应当聚焦于某一特定的业务功能，并且确保完成它。

从技术上，微服务不应该局限于某个技术栈或者后端存储，可以非常灵活，以便于解决业务问题。

这一点在非微服务的系统设计时，可能导致我们做一些妥协。而微服务可以让你用你认为最合适的方式解决。

3.谁创建，谁负责

这一点是关于软件开发前后责任归属的观点。在一个大型公司，通常开发团队开发完成之后，通过一些交接会议将应用交接给维护团队，

在微服务里，创建微服务的团队有责任继续维护它。

[You build it, you own it !!](http://aronatkins.github.io/2014/12/23/you-build-it-you-own-it.html)

这种方式让开发团队可以每天操作软件，并且更好的理解，客户是如何使用软件的。

4.基础设施自动化

盛载微服务准备和构建的基础设施也是一个非常重要的需求。

一个服务应当被独立部署，并且包含所有的依赖，环境等的物理资源。

SOA和微服务的一个主要不同点就是自动化程度上的不同。大部分的SOA实现只达到服务级别的抽象，而微服务走的更远，它达到了对实现和运行环境的抽象级别。

也就是说，在传统的开发中，我们构建一个WAR包或EAR包，然后把他们部署在容器上。

而在一个规范的微服务中，每个微服务应该被构建成胖jar（[fat Jar](http://howtodoinjava.com/maven/maven-shade-plugin-create-uberfat-jar-example/)）其中内置了所有的依赖，然后作为一个单独的java进程存在。

“微服务架构”概念的提出已经有很长一段时间了，但在最近几年却开始频繁地出现。微服务架构是一种特定的软件应用程序设计方式——将大型软件拆分为多个独立可部署服务组合而成的套件方案。虽然这种架构风格的确切定义还存在争议，但并不妨碍其在众多企业的实际应用中被实践，并体现出了具备通用特征的业务功能、自动化部署、端点智能化以及对语言与数据的离散化控制能力。Docker 作为一种开源的应用容器引擎，帮助开发者将他们的应用以及依赖打包到一个可移植的容器中，便于应用的部署和扩展。而随之产生的微容器概念和微服务正好相辅相成，通过 Docker 封装的应用可以轻松运行在以扩容能力见长的云计算平台上。数人云作为专业的数据中心管理系统，提供了基于 Mesos 和 Docker 技术的企业级容器云生产环境，通过一键部署、横向扩展、持续集成等特性，助力微服务架构在企业应用环境的实践。

由于篇幅问题，本文将分上下两篇，下篇将于明天放送给大家。

“微服务”——目前可谓早已人满为患的软件架构领域的新兴名词。虽然我们对于这种新生事物往往带着一种先入为主的蔑视与忽略态度，但经过几年的历练，我们发现这种软件构建风格正变得越来越具有吸引力。过去几年中已经有诸多企业将其引入实际项目，而至今其结果仍然相当积极，这甚至促使很多同业人士开始将微服务架构作为企业级应用程序的默认开发途径。但遗憾的是，目前仍然缺乏一套系统的概念定义，告诉我们微服务到底是如何实现这些成效的。

简而言之，微服务架构风格[1]是一类将单一应用程序作为由众多小型服务构成之套件加以开发的方式，其中各项服务都拥有自己的进程并利用轻量化机制（通常为HTTP源API）实现通信。这些服务围绕业务功能建立而成，且凭借自动化部署机制实现独立部署。这些服务匹配一套最低限度的中央式管理机制，且各服务可通过不同编程语言编写而成并使用不同的数据存储技术。

要解释微服务风格，那么首先应当将其与整体风格进行比较：整体应用程序作为单一单元进行构建。企业级应用程序通常包含三个组成部分：一套客户端用户界面（由运行在用户设备上的浏览器中的HTML页面以及Java代码构成）、一套后端数据库（将大量插入至数据库管理系统的大量表构成，通常采用关系数据库）以及一款服务器端应用程序。该服务器端应用程序将负责处理HTTP请求、执行域逻辑、对来自数据库的数据进行检索与更新，同时选定HTML视图并将其发送至浏览器端。此服务器端应用程序通常为单一的逻辑可执行文件[2]。任何针对该系统的变更都需要对该服务器端应用程序进行新版本构建与部署。

这样的整体服务器机制在构建此类系统中可谓不可或缺。我们用于处理请求的全部逻辑都运行在单一进程当中，允许大家使用语言中的基本功能以将该应用程序拆分为类、函数以及命名空间。通过这种方式，我们能够在开发人员的笔记本设备上运行并测试应用程序，同时利用一整套部署流程以确保全部变更都经过妥善测试而后被部署在生产环境当中。大家可以将大量实例运行在一套负载均衡方案之后，从而实现横向扩展能力。

这类整体应用程序当然能够切实起效，但人们却逐渐发现其中存在着诸多弊端——特别是在将大量应用程序部署在云环境当中的情况下。由于变更周期被大量集中于一处——即使仅仅指向应用程序中的一小部分，单一变更亦要求我们对应用程序整体进行重构与重新部署。随着时间推移，我们往往很难保证理想的模块化结构，这意味着本应只影响单一模块的变更往往会扩散至该模块之外。规模伸缩亦要求我们对整体应用程序进行规模调整，而非单纯为其中必要的部分进行资源扩容。

图一：整体型应用程序与微服务架构应用程序

正是这些弊端造就了如今的微服务架构风格：即以服务套件的形式构建应用程序。除了各服务能够单独进行部署与规模伸缩之外，每项服务还具备牢固的模块边界，甚至允许我们在不同的服务当中使用不同的编程语言进行代码编写。另外，各服务亦可由不同团队负责管理。

我们认为微服务风格并不算什么新鲜事物或者创新成果，其历史至少可以追溯至Unix设计时代。但我们同时亦坚信，微服务架构一直未能受到足够的重视，而其确实能够帮助大家更好地完成软件开发工作。

**微服务架构之特性**

我们无法给微服务架构风格出具一条确切的定义，但我们却可以根据该架构表现出的各类共同特性对其加以描述。正如各类根据共同特性做出的定义一样，并不是所有微服务架构都符合这些特性，但可以肯定的是具备这些特性的微服务架构占据大部分比例。尽管我们各部分内容的作者仅仅是相关技术社区中的活跃成员，但制作这份文档是为了对采用微服务架构的工作流程及成果做出总结，而且其中仍有相当一部分表述并非严格定义——只应作为常见情况考量。

**通过服务实现组件化**

长久以来，我们一直参与软件行业之内并意识到人们对利用组件整合方式构建系统的渴望——这种思路与我们在物理世界中采取的构建机制非常相似。而在过去几十年当中，我们发现已经有大量公共库渗透到多数语言平台当中并成为其坚实的组成部分。

在谈到我们所使用的组件时，大家可能会发现不同群体对组件的定义也有所区别。我们对组件做出的定义是，其属于软件中的一类单元，且具备可更替性与可升级性。

微服务架构会使用这些库，但其实现组件化的主要手段则是将软件拆分成多个服务。我们将“库”定义为与程序相对接且可通过内存内函数调用发挥作用的组件，而“服务”则为进程之外的组件，其可通过Web服务请求或者远程程序调用等方式实现通信。（这里的服务概念与多数OO程序中的服务对象概念有所区别[3]）。

将服务作为组件加以使用（而非库）的一大原因在于，服务具备独立可部署能力。如果大家的应用程序[4]由单一进程中的多个库构成，那么指向任何单一组件的变更都会致使该应用程序必须进行重新部署。但如果该应用程序被拆分成多项服务，那么单一服务变更将只会致使该服务进行重新部署。虽然这并非绝对，例如某些变更会导致服务接口受到影响，但一套优秀的微服务架构旨在尽可能少地对服务协议中的服务边界及演进机制产生干扰。

将服务作为组件的另一个理由在于实现更为明确的组件接口。大多数编程语言并不具备用于定义明确发布接口的良好机制。一般来讲，其只会提供说明文档及规则以防止用户打破组件封装，但这同时亦会导致不同组件之间的耦合程度过高。利用明确的远程调用机制，服务能够轻松避免此类难题。

但以这种方式使用服务亦存在一定弊端。远程调用在资源需求方面往往远高于进程内调用，因此远程API需要采取粗粒度设计，但这亦会增加API的使用难度。如果大家需要更改不同组件间的职能分配，那么这类需求在跨越进程边界时往往不易实现。

通过粗略观察，我们往往会发现这些服务会与各运行时进程相映射——但这仅仅只是第一印象。一项服务可能由多个进程构成，且各进程始终共同进行开发与部署——这方面实例包括只由单一服务所使用的应用程序进程以及数据库。

**围绕业务功能构建组织**

当着眼于将单一大型应用程序拆分成多个组成部分时，管理人员通常更重视技术层，其中具体包括UI团队、服务器端逻辑团队以及数据库团队。当这些团队据此进行拆分时，即使是最简单的变更也将给项目造成跨团队协作负担，并因此导致时间与预算的双重支出。睿智的团队会对此进行优化，同时采取两害相权取其轻的办法——即强制要求逻辑存在于一切与之相对接的应用程序当中。换言之，也就是实现逻辑的普遍存在性。这正是所谓康威法则[5]的一种实际表现形式。

任何组织在设计一套系统（广义层面的系统）时，其设计成果都会直接体现该组织所使用的沟通结构。

--梅尔文·康威，1967年

图二：康威定律的实际体现

微服务方案对于各部门而言是一种不同于以往，且以业务功能为核心的服务拆分及组织途径。此类服务采用软件方案在业务层面中的广泛实现堆栈，具体包括用户界面、持久性存储以及任何外部协作机制。因此，各团队将拥有跨职能特性，包括开发过程当中要求的全部技能组合：用户体验、数据库以及项目管理等等。

图三：由团队边界决定的服务边界

**微服务架构有多“微”？**

尽管“微服务”早已成为一种极具人气的架构类型，但这一名称却并不能准确反映服务的实际规模——换言之，“微”服务并不一定微。在与众多微服务从业者的交流当中，我们发现服务的具体规模可谓多种多样。其中规模最大的成果源自Amazon公司旗下的“两块披萨”团队（即整个团队只需两块披萨即可填饱肚子），这意味着其总人数在十位左右。而规模较小的团队则由六人组成，负责支持六项服务。

那么这就带来了新的问题：这种十二人对单项服务的机制同一人对单项服务之间存在着怎样的差别？二者也许不可一概而论。就目前而言，我们姑且认为双方属于同类团队结构，但随着对微服务认识的持续深入，也许我们未来将抱持新的观点。

采取此类组织方式的企业实例可参见www.comparethemarket.com，其各职能团队共同负责构建并运营每款产品，而每款产品则被拆分为一系列独立的服务——且各服务间通过一套消息收发总线实现通信。

大型整体应用程序亦可以始终围绕业务功能实际模块化，不过这种状况并不常见。诚然，我们都听说过由大型团队构建的单一整体应用程序根据自身业务线进行设计与划分。然而在这类情况下，最大的问题在于整体应用程序在组织当中需要考虑太多背景信息。如果其整体范畴当中包含太多模块边界，那么团队中的单一成员将很难通过短期记忆对其进行管理。除此之外，我们发现这种模块化业务线的维护工作还要求相关人员具备极高的专业技能水平。相比之下，服务组件能够令拆分方式更为明确，从而大大简化团队边界的设定与认知。

**产品而非项目**

大部分应用程序开发工作都会遵循项目模式：其目标在于交付软件方案中的特定部分，并拥有直观的完成指标。在软件开发工作完成后，其会被传递至运维部门，这时负责构建该软件的团队也将即刻解散。

微服务的支持者们则认为这种模式并不可取——他们的主张是相关团队应该伴随产品走过整个生命周期。这方面最典型的例子应该是Amazon公司提出的“谁构建，谁运行”原则，其中开发团队需要对生产环境下的软件成果承担全部责任。这就要求开发人员在日常工作中全程关注其软件的生产运行情况，同时掌握来自用户的反馈意见，意味着他们需要在一定程度上为用户提供技术支持服务。

产品的定位应始终与业务功能相协调。相较于以往将软件视为一整套已经完成的功能集的心态，微服务架构要求我们全程与之保持关联，并思考该软件能够如何协助用户加强业务功能。

当然，我们完全可以将同样的思路引入整体应用程序当中，不过大量小型服务集合能够显著简化服务开发人员与及用户之间的个人联系。

**智能化端点与傻瓜式流程**

在跨越不同进程构建通信结构时，我们发现很多产品及方案会直接把智能化机制塞进通信机制本体当中。这方面的典型实例就是企业服务总线（简称ESB），ESB产品当中通常包含复杂度极高的消息跌幅、编排、转换以及业务规则应用等机制。

微服务社区则倾向于使用另一种实现方式：智能化端点与傻瓜式流程。采用微服务架构的应用程序旨在尽可能实现解耦化与关联性——它们各自拥有自己的域逻辑，而且在经典Unix场景下的运作方式更像是过滤器机制——接收请求、应用合适的逻辑并生成响应。这一切都通过简单的REST类协议实现编排，而非经由WS-Choreography或者BPEL等复杂协议以及中央编排工具实现。

目前最常用的两类协议为配合源API的HTTP请求-响应与轻量化消息收发协议[6]。对于前者，最简练而准确的说明是：

立足于Web，而非居于Web背后。

-- Ian Robinson

微服务团队采用的正是万维网（在很大程度上亦包括Unix在内）所遵循的原则与协议。一般来讲，其使用的资源能够为开发人员或者运维人员轻松实现缓存处理。

第二类作法则是立足于轻量化消息总线实现消息收发。这类基础设施选项通常具备傻瓜式特性（这种傻瓜特性体现在实现操作上，即只需匹配消息路由机制，再无其它）——以RabbitMQ或者ZeroMQ为代表的简单实现方案仅仅需要提供一套可靠的异步结构，而服务的全部智能化元素仍然存在于端点当中并负责消息的生成与消费。

在整体应用程序当中，各组件在进程内执行并通过方法调用或者函数调用的方式实现彼此通信。将整体应用程序转化为微服务形式的最大难题在于改变这种通信模式。由内存内方法调用指向PC通信机制的简单转换往往无法良好起效。相反，大家需要利用粗粒度方式取代原本的细粒度通信机制。

**脚注**

1: “微服务”一词最早被威尼斯附近的一个软件架构师小组于2011年5月首次提及，当时他们用这个词汇来描述自己近期研究项目当中所涉及的通用性架构机制。2012年5月，该小组作出最终决议，认为“微服务”是最适合的架构名称。2012年3月，James在《微服务-Java以及Unix方式》当中就此发表了一篇案例研究报告，而Fred George也几乎在同一时间进行了相同的工作。Netflix公司的Adrian Cockcroft将微服务架构称为“细化SOA”，并认为这是一套在Web规模下具备开创意义的架构类型。Joe Walnes、Dan North、Evan Botcher以及Graham Tackley也分别在这篇文章中对此作出了评论。

2: 文章中所使用的“整体”一词长久以来一直被Unix业界所使用。其首次出现在《Unix编程艺术》一书中，用于描述那些过于庞大的系统方案。

3: 很多面向对象设计人员，也包括我们自己，都会在域驱动设计当中使用“服务对象”这一表述，专指那些并不具备实质性联系但却拥有重要作用的对象。这与我们在本文中所使用的“服务”一词在表意上完全不同。遗憾的是，服务这个词汇同时具备两种含义，而我们对这种多义词也没有更好的处理办法。

4: 我们将一款应用程序视为一套社会性体系，其中融合了代码库、函数组以及供应主体。

5: 大家可以查看梅尔文 康韦网站上的原文论述。

6: 对于规模极为庞大的应用体系，企业通常会采用二进制协议——例如protobufs。使用二进制协议的系统仍然符合智能化端点与傻瓜式通道的特性——并为了规模化而在透明度方面作出妥协。不过大多数Web方案与绝大多数企业不需要在这方面考虑太多——一般来讲，透明度越高、效果就越好。

在上篇中我们讲到了微服务的几个架构特性，包括通过服务实现组件化、以业务功能为核心进行组织、产品而非项目、智能化端点与傻瓜式流程，在今天的微服务概念解析下篇中，我们将继续讲述微服务的特性，具体分析它的离散化治理、离散化数据管理、基础设施自动化、故障应对设计以及演进设计，并理性思考微服务作为一项新兴的技术成果，是否能够代表未来。  
​

离散化治理

聚合型治理的一大影响在于使得单一技术平台上出现标准化趋势。经验表明这类方案具备收缩特性——意味着各个实际问题并不能够轻松与解决方案对应起来。我们更倾向于使用正确的工具执行正确的任务，而且虽然部分整体应用程序能够发挥不同编程语言的独特优势，但这种情况并不常见。

微服务与SOA

当我们探讨微服务时，经常出现的问题就是其到底是不是我们十年前就听说过的面向服务架构（简称SOA）的另一种表现形式？二者之间确实存在一定联系，因为微服务风格拥有与SOA相似的逻辑主张。然而问题在于，SOA的实际含义太过广泛，而且当我们提到所谓“SOA”时，实际所指的对象往往跟这里提到的微服务概念差之千里——具体来讲，其通常代表那些专注于利用ESB实现的集成化整体应用程序。

值得强调的是，我们也见证了大量表现糟糕的面向服务实现手段——从将复杂性隐藏在ESB当中[7]的作法，到投入多年以及数百万资金却毫无成效的尝试，再到以聚合型治理模式抑制变更，我们几乎看不到面向服务架构能够带来什么显著的积极影响。

诚然，微服务社区当中使用的不少技术成果都源自开发人员在大型企业当中积累到的集成化服务成果。Tolerant Reader模式正是其中的典型代表。对Web的运用确实带来可观回报，而使用简单协议正是经验积累的直接产物——这显然是为了解决标准汇聚所导致的高复杂性难题（无论何时，如果大家需要利用一种实体来管理其它实体，那么就意味着各位已经面临着大麻烦）。

SOA的这些弊端导致一部分微服务布道者很讨厌人们把SOA的标签加在微服务头上——尽管也有一些人认为微服务正是SOA的一种实现方式[8]，或者说我们可以将微服务称为“面向服务的正确实现”。无论如何，事实上SOA含义的宽泛性意味着其更适合作为一种用于定义架构风格的术语，而非具体解决方案。

通过将整体应用程序的各组件拆分成服务，我们能够对各服务进行分别构建。各位可能希望利用Node.js建立一套简单报告页面？照此办理即可。打算利用C++构建特定的近实时组件？没问题。打算利用不同类型的数据库以匹配单一组件的读取行为？目前的技术方案已经能够实现这种独立重构需求。

当然，我们能够实现以上目标，并不代表我们必须这么做——但对系统进行拆分意味着大家能够拥有更多备用选项。

采用微服务架构的团队倾向于以不同的方式实现所谓标准。相较于以往编写一整套定义标准集的作法，他们更乐于开发实用工具并交付给其他开发人员，从而利用其解决自身面临的类似问题。这些工具通常能够在更为广泛的层面得到实现与共享，但同时又不至于转化为排他性内部开源模式。现在git与github都已经成为客观层面的版本控制系统选项，而开源实践也越来越多地成为内部环境中的常见组成部分。

Netflix公司就是个很好的例子，他们遵循的正是这样一种理念。将具备实用性且经过严格考验的代码作为库，并鼓励其他开发人员利用其以类似的方式解决的类似的问题，这就为各团队成员在必要时选择其它工具保留了空间。共享式库专注于数据存储、进程间通信以及我们在后文中将要探讨的基础设施自动化等问题的解决。

对于微服务社区而言，资源成本显然是种不受欢迎的因素。这并不是说该社区不承认服务协议的价值。恰恰相反，这是因为他们希望构建起大量服务协议。他们希望能够采用多种完全不同的方式对这些协议进行管理。像Tolerant Reader以及Consumer-Driven Contacts这样的模式在微服务架构中非常常见。这些服务协议也各自以独立方式不断演进。将消费者驱动型协议作为构建工作组成部分的作法能够显著增强参与者信心，同时快速获取服务功能能否确切实现的反馈意见。事实上，澳大利亚的某个团队就在积极利用消费者驱动型协议进行新服务构建。他们使用的简单工具确保其能够针对单一服务实现协议定义。其甚至在面向新服务的代码被编写出来之前就已经成为自动化构建流程中的一部分。这意味着服务只有在切实满足该协议要求的前提下才能够实现构建——这就有效解决了构建新软件时经常出现的“YAGNI”[9]难题。这些技术与工具成果围绕协议而生，并通过降低不同服务间的耦合性限制了其对中央协议管理机制的依赖。

多种语言，多种选项

JVM作为平台的快速发展已经成为多种语言混成于单一通用平台内的最新明证。这种作法已经成为一类常见实践，旨在充分发挥高级语言在过去数十年中发展所实现的种种高级抽象优势。其甚至以涓滴效应影响到裸机以及通过低级语言编写的性能敏感型代码。然而，众多整体应用程序并不需要这种级别的性能优化效果，亦非常见的DSL与高级别抽象开发成果。相反，整体应用程序往往使用单一语言，这也严重限制了其能够使用的技术手段。[10]

也许离散化治理的人气正是源自Amazon方面提出的“谁构建，谁运行”原则。各团队需要为其构建的软件的各个方面承担责任，包括为软件提供24/7全天候运维支持。这种程度的责任下放当然还没有成为常态，不过我们已经看到越来越多的企业开始将责任交付至开发团队。Netflix公司亦是另一家采取这种理念[11]的企业。为了不至于在凌晨三点被紧急来电叫醒，开发人员们当然会全力以赴提升所编写代码的质量水平。这些思路与传统的集中化治理模式明显相去甚远。

离散化数据管理

数据管理离散化拥有多种不同的表现形式。从最为抽象的级别来看，这意味着全局概念模型将在不同系统之间有所区别。这种问题常见于解决方案在大型企业当中的部署，毕竟销售团队对于客户概念的理解方式必须不同于技术支持团队的理解方式。被销售人员视为客户的对象也许根本不会出现的技术支持团队的视野当中。不同属性甚至是相同属性的不同理解方式都可能在语义层面产生细微的差异。

实践性规范与执行标准

这种态度实际有点二分法的意味：微服务团队倾向于回避由企业架构部门制定的硬性执行标准，但却乐于使用甚至积极推广HTTP、ATOM以及其它微格式开放标准。

二者之间的本质区别在于标准的开发方式以及执行方式。由IETF等组织管理的标准只会在得到广泛采用之后才能真正成为业界规范，而且其往往脱胎自成功的开源项目。

这些标准拥有与商业世界完全不同的立场与定位——事实上，商业标准的制定工作往往由那些几乎不具备编程经验的团队所负责，或者受到具体厂商的过度影响。

这一问题通常出现在不同应用程序之间甚至是应用程序之内，特别是在将应用程序拆分为多个独立组件的情况下。解决问题的一类可行思路在于基于背景边界化的区域驱动型设计（简称DDD）方案。DDD机制将一个复杂的区域拆分成多个具备边界的背景单元，并对各单元之间的关系加以映射。这种方式同时适用于整体与微服务架构，但服务与背景边界间的自然关联性有助于声明我们曾在业务功能章节中提到过的区分效果。

除了对概念模式进行离散化处理，微服务同时也能够拆分数据存储决策。尽管整体性应用程序倾向于使用单一逻辑数据库保存持久性数据，但企业通常更乐于利用单一数据库涵盖一系列应用程序——而且大多数此类决策立足于具体供应商提供的授权商业模式。微服务机制则选择由每项服务管理其自身数据库的方式，而非不同实例基于同一数据库技术或者完全使用多种不同数据库系统——这种方式亦被称为混合持久化。大家可以利用混合持久化方案打理整体应用程序，但其在微服务架构中的亮相频率明显更高一些。

对微服务架构内数据责任关系的离散化处理也影响到了更新管理工作。常见的更新处理方案是在更新多种资源时，利用事务处理机制来保证其一致性。这种方式通常被用于整体性应用程序汉中。

这种事务处理使用方式确实有助于保障一致性，但却会带来显著的临时性耦合效果，而这在跨越多项服务时会带来新的难题。分布式事务处理非常难以实现，因此微服务架构更强调服务之间的事务处理协调性，同时明确强调只需保障最终一致性并通过补偿运算解决其中的冲突问题。

利用这种方式管理一致性问题已经成为众多开发团队的新困境，但其却能够切实匹配业务实践。一般来讲，企业需要保留一定程度的不一致性以实现某种程度的逆转能力，从而利用快速响应处理错误状况。这种权衡有其必要性，只要确定失误成本要低于高一致性条件下可能造成的业务损失成本即可。

基础设施自动化

基础设施自动化技术在过去几年中得到了长足发展——而云与AWS的演进则显著降低了构建、部署及运维微服务架构所带来的复杂性水平。

大部分利用微服务机制构建的产品或者系统都是由具备丰富的持续交付及其前者——持续集成——经验的团队所完成。通过这种方式构建软件的团队能够充分发挥基础设施自动化技术成果的潜在能力。我们可以将整个流程整理成以下图表：

图五：基本构建流程

让正确决定更易于执行

作为一项连带效应，我们发现实现持续交付与部署能够帮助开发人员及运维人员创造出高实用性工具。这类工具能够创建artifact、管理代码库、建立简单服务或者实现标准监控与记录等常见功能。这方面最典型的实例当数Netflix公司发布的一系列开源工具，险些之外Dropwizard等方案亦得到广泛使用。

整体应用程序的构建、测试与推送流程能够在此类环境下顺利完成。事实证明，一旦大家利用自动化流程进行整体应用开发，那么部署更多应用程序也将成为顺理成章的轻松任务。请记住，持续交付的目标之一就是令部署变得无脑化，这意味着无论是一款应用还是三款，其实际部署流程都不会有什么区别[12]。

我们还发现，不少团队在利用这种广泛的基础设施自动化能力管理生产环境下的微服务架构。相较于前面提到的整体与微服务应用在部署层面并没有太大区别，实际运维环境下的具体条件则存在着巨大差异。

图六：模块部署的具体方式往往差别巨大

故障应对设计

将服务作为组件加以使用的结果之一在于，应用程序需要经过针对性设计以确保其具备服务故障容错能力。任何服务调用都有可能因为供应程序不可用而发生问题。在这种情况下，客户端必须要尽可能做出适当的回应。相较于整体应用程序来说，服务即组件机制会增加额外的处理复杂性，这也是微服务架构的一大弊端。在这种情况下，微服务团队需要不断审视服务故障对用户体验造成的影响。Netflix公司的“猴子军团”项目就专门负责在正常运营期间对服务进行破坏，甚至利用数据中心故障来测试应用程序的弹性及监控能力。

断路器与可交代生产环境之代码

断路器模式出现在Amazon的Release It!当中，其中提到的其它模式还包括隔板模式与超时模式等。在加以结合之后，这些模式将在构建通信应用方面发挥巨大作用。Netflix公司发布的一系列博文就很好地解释了他们对这些模式选项的具体使用方式。  
这类自动化测试机制往往会令正等待周末下班的运维团队们感到不寒而慄。这并不是说整体架构风格就无法使用高复杂性监控机制——只不过这种情况确实不太常见。

由于服务随时可能发生故障，因此最重要的就是保持对故障的快速检测能力，并在可能的情况下对其进行自动恢复。微服务应用程序高度强调对应用程序的实时监控能力，同时不断对架构元素（数据库每秒钟接收到的请求数量）以及业务相关指标（例如每分钟收到的订单数量）进行记录。语义监控能够通过早期预警系统抢先一步做出警示，并引导开发团队对问题加以跟进与调查。

这一点对于微服务架构尤为重要，因为微服务更倾向于采用由编排及事件协作实现的应急处理方式。尽管很多专家都对应急处理方案偶尔带来的收益表示认同，但其实际上往往也是让事情变糟的罪魁祸首。为了及时阻断糟糕的应急处理并确保其拥有可恢复性，监控系统就变得极为重要。

同步调用殊不可取

无论何时时，一旦在不同服务之间进行多次同步调用，那么可能引发宕机的概率也会以乘法形式增长。简单来讲，系统的总体宕机时间为各单个部件宕机时间的乘积。这时我们就面临着具体选择，到底是以异步方式进行调用，还是以计划方式管理由同步调用带来的宕机时间。英国《卫报》网站在其全新平台上执行了一项简单的规则——每个用户请求对应一次同步调用，而Netflix公司所使用的API则经历重新设计，确保其结构内采用异步调用机制。  
整体应用程序的构建方式可与微服务架构同样透明——事实上也本应如此。二者的区别在于，在面对整体应用时我们需要在确切了解其运行在不同进程中的服务何时发生断开。考虑到同一进程当中可能包含多套库，这种透明度水平实际上很难实现。

微服务团队需要利用复杂的监控与记录机制处理各项服务，例如通过仪表板显示上线/下线状态以及一系列运营与业务相关指标。另外，我们还需要面对断路器状态、当前数据吞吐量以及延迟等其它常见的衡量数据。

演进设计

微服务从业者通常都具备演进设计工作背景，并将服务拆分视为一种深入型工具，旨在帮助应用程序开发人员在无需拖慢变更速度的前提下实现面向应用程序的变更控制。变更控制并不一定意味着变更数量削减——配合正确的态度与工具，大家完全可以帮助软件提供快速、频繁且经过良好控制的变更。

当尝试将一套软件系统拆分为多个组件时，我们往往面临着与具体拆分工作相关的决策任务——即我们应该遵循怎样的方针对应用程序进行拆分？而组件中的关键属性则在于其独立替换与可升级特性[13]——这意味着我们要找到确切的平衡点，保证自身能够在不影响其它协作对象的前提下对单一组件进行重写。事实上，很多微服务团队会更进一步，直接清退某些服务而非对其进行长期升级。

英国《卫报》网站就是个很好的例子，其应用程序在设计与构建方面作为整体应用存在，但却在逐步面向微服务架构演进。该网站的核心部分仍然属于整体性项目，但他们更倾向于通过构建微服务利用整体API实现新功能添加。这套方案对于临时性功能的实现非常重要，例如加设专题页面以显示体育赛事报道。网站中的这类组成部分能够通过快速开发语言在短时间内编写完成，并在对应事件结束后立即下线。我们还发现其它一些金融机构亦采取类似的方式公布突发性市场波动，并在数周或者数月之后将其下线。

这也强调了可替换性在模块化设计中的重要地位，其主旨正在于将模块机制贯彻整个变更模式[14]。大家希望只变更其中必须变更的部分，而其它模块则继续保持原样。系统当中那些几乎很少变动的部分应该立足于不同于高变更频率组件的服务。如果大家发现自己经常需要同时对两项服务做出变更，那么明显应该将二者加以合并。

将组件纳入服务也让我们能够以更高的细粒度水平进行规划制定。在整体应用程序当中，任何一项变更都需要对应用整体进行重构与重新部署。但在微服务架构方面，我们只需要重新部署包含对应变更的服务。这能够显著简化并加快发布流程。不过其弊端在于，我们必须考虑针对单一服务的变更是否会影响到其它服务。传统的整体性方案能够通过版本控制解决这类难题，但微服务领域则倾向于将版本控制作为最后一种应急办法。我们可以通过设计保证服务拥有强大的容错能力，从而应对其供应程序中出现的各类代码修改。

微服务是否代表着未来？

我们撰写这篇文章的主要目的在于解释微服务架构的基本思路与原则。而在撰写过程当中，我们明确意识到微服务架构风格确实是一项值得重视的关键成果——企业级应用程序开发人员应当对其加以了解。我们最近利用该架构构建了多套系统，而且了解到亦有其它多家企业将其纳入业务体系。

我们了解到的微服务架构先驱企业包括Amazon、Netflix、英国《卫报》、英国政府数字化服务局、realestate.com.au、Forward以及comparethemarket.com等等。2013年召开的相关会议则公布了更多参与其中的重要厂商。除此之外，另有相当一部分企业一直在使用类似的实现思路——但却并没有使用‘微服务’这样的称谓。（其通常将其冠以SOA标签——不过正如我们之前提到，SOA是一类存在大量矛盾取向的概念组合。[15]）

尽管拥有这些积极的经验，但我们仍然无法完全肯定微服务架构就代表着软件未来的发展方向。虽然我们的实际经历证明微服务架构截至目前仍拥有优于整体性应用程序的积极优势，但必须承认只有充分的时间积累才能帮助我们做出真正完整则准确的判断结论。

我们的同事Sam Newman曾于2014年倾尽心力撰写出这本关于我们如何构建微服务架构类应用的论著。如果大家希望进一步探讨这个议题，请千万不要错过。

通常来说，架构决策的实际影响可能需要几年之后才能逐步显现出来。我们已经看到不少优秀的团队带着巨大的热情与愿景而投入工作，但最终却构建起一套陈旧不堪的整体性架构。很多人认为同样的情况不太可能发生在微服务架构身上，因为其服务边界非常明确因此不太可能发生相互影响。但由于时间尚短且系统程度不足，我们目前还无法真正评估微服务架构的成熟度水平。

人们对微服务成熟度抱持的怀疑态度也有其理由。在任何组件化尝试工作当中，最终结果的成功与否都取决于该软件与拆分后组件的契合效果。我们目前仍然很难说明组件边界的选择原则。演进设计导致边界划分变得非常困难，因此最重要的是保证其重构的简易性。但一旦将组件作为服务处理以实现远程通信，那么其重构难度将远远高于进程内库。在不同服务边界之间进行代码移动难度极大，而任何接口变更都需要在不同相关服务间实现，同时添加层的向下兼容能力，这无疑会令测试工作更加复杂。

另一大问题在于，如果相关组件间的关系不够简洁，那么我们就相当于把组件内部的复杂性转移到了不同组件间的连接当中。这样做不仅会导致复杂性扩散，同时亦会导致其明确性缺失且难以控制。立足于小型、简单组件审视问题总是更为直观，而在不同服务间进行纵览则往往会错失关注点。

最后，团队的技能水平也将起到决定性作用。新型技术成果往往要求高水平技术团队加以实施。不过高水平团队能够顺畅利用的技术方案并不一定能够在低水平人员手中发挥作用。我们已经见证了众多低水平团队构建起的如一团乱麻般的整体架构，但仍需要时间来了解微服务架构是否会在同样的情况下引发同样的状况。诚然，糟糕的团队创建出的始终只能是糟糕的系统——但我们不知道微服务架构到底是会缓解这种状况，还是令状况更中惨不忍睹。

目前有一种较为理性的论调，认为我们不应将微服务架构作为起步方案。相反，大家可以从整体性开发风格出发，保证其结合模块化机制，并在整体性特征引发实际问题后逐步将其拆分为微服务形式。（不过这样的建议并非完全理想，因为良好的进程内接口往往并不能成为良好的服务接口。）

因此我们对此抱持谨慎的乐观态度。到目前为止，我们已经了解到关于微服务架构的方方面面，而且其应该能够成为一种极具价值的开发手段。虽然还不能做出最终判断，但软件开发工作的固有挑战之一，正是我们只能根据目前掌握的远称不上完美的信息做出决策。

脚注

7: 虽然无关紧要，但Jim Webber曾经将ESB解释成“Egregious Spaghetti Box”，也就是“恐怖意面盒”。

8: Netflix公司最近将其架构类型称为“细化SOA”。

9: “YAGNI”的全称是“You Aren’t Going To Need It（你根本不需要它）”，这是一项经典的用户体验原则，即不要自作聪明地添加非必要性功能。

10: 我们所宣称的整体型应用只支持单一语言确实有些不尽不实——在当下的Web系统构建过程中，大家可能需要掌握JavaScript、XHTML以及CSS，而在服务器端的语言选项则包括SQL以及某种ORM（即对象关系映射）衍生语言。没错，单一语言肯定玩不转，但我相信大家明白我想要强调的意思。

11: Adrian Cockcroft在2013年11月的Flowcon大会上作出了精彩演讲，并特别提到了“开发者自助服务”与“开发者应亲自运行所编写代码”的观点。

12: 我们在这里的说法并不准确。很明显，在更为复杂的拓扑结构中部署大量服务肯定要比在单一整体型架构内进行部署困难得多。幸运的是，各类模式能够显著降低这种复杂性——当然，在工具方面的投入仍然不可或缺。

13: 事实上，Dan North将这种类型称为“可替代式组件架构”而非微服务架构。由于其强调内容属于微服务架构的一类子集，所以我们更倾向于使用后一种表达方式。

14: Kent Beck将此作为其《实施模式》一文中的设计原则之一。

15: SOA几乎是此类架构的历史起源。我记得当SOA一词在本世纪初刚刚出现时，很多人表示“我们几年前就已经将其引入日常工作了”。也有意见认为这种架构类型似乎最早出现于早期企业计算当中，COBOL程序通过数据文件实现通信的处理机制。而在另一方面，也有人认为微服务架构与Erlang编程模型其实是同一回事，不过后者只被应用在企业应用程序当中。

http://www.jianshu.com/p/cd8e9ea4afc0

<http://www.sohu.com/a/57910094_332175>

http://dockone.io/article/394

测试。

# 第三章 系统需求分析与总体设计

# 3.1 项目背景介绍

预付卡系统是携程旗下非常重要的一部分业务，尤其其针对企业用户，有很大的价值。更好的销售，为公司带来更大的利润，和其他模块结合的更好。测试试测。

到PRD中找吧，哈哈

测试

试测。

# 3.2 系统需求分析

总体目标，业务目标，技术目标

目的：找出当前礼品卡生产、销售、售后、使用、核算、安全等方面已知的架构重构需求。针对需求背后的问题，评估这些问题对当前业务以及今后业务扩展的影响程度，评估要改正这些问题要耗费的人力，综合考虑得出架构重构需求的优先缓急。我们需要为最优先的那些架构重构需求预先确定方案，以便有开发资源后能尽快开展重构工作。

范围：

礼品卡的主业务范围：

生产：主副项目的申请，依据主副项目进行的实体卡/电子卡造卡。

销售：线上线下的礼品卡销售，礼品卡订单处理与管理

售后：礼品卡销售完成后，对礼品卡执行的作废、延期、退款等变更工作。

使用：礼品卡领用、直充、消费等业务

横跨多个业务的功能：

核算：用户的礼品卡账户明细，销售资金到账，销售与使用环节的明细核对。等。。。

安全：礼品卡充值、领用、使用各个环节的幂等控制措施。对原子业务的事务封装。对卡密的保护。等。。

需求收集手段：

向产品征集

向各位开发Leader征集

主动整理，解析现有流程

影响优先度评估的要素(TODO：和大家讨论，随后给出更精确的度量)

影响类别

影响当前业务，指当前业务不能完全依靠系统正常运行，需要运营人员的人工介入，或者业务的某种妥协才能正常工作

将触及系统能力极限，在可预期的将来，系统将无法提供正常功能

影响业务扩展，当前架构在业务不断调整后，已经僵化，难以添加新的功能

重构需要耗费的人力

需要给出初步方案，评估影响的范围，系统组件数量，乃至代码行数

礼品卡生产(Product)业务域

直充主副项目流程

预赋值主副项目流程

后赋值主副项目流程

造卡

礼品卡销售(Order)业务域

线上销售

[分销](http://conf.ctripcorp.com/pages/viewpage.action?pageId=100668506)

[代销](http://conf.ctripcorp.com/pages/viewpage.action?pageId=101450490)

[商家平台](http://conf.ctripcorp.com/pages/viewpage.action?pageId=100668060)

合作商订单系统

挑卡

积分、配送、消息等处理后续

礼品卡售后(AfterSale)业务域

变更

作废

管理费 & 延期

冻结

专享会员

礼品卡退货

礼品卡使用(Account)业务域

直充

领用

消费

退款

账户明细

商业预付卡是建设于运营方的预付卡系统平台之上，并通过预付卡平台系统提供发卡服务并受理持卡人的查询、消费、积分兑换、预充值等业务。

本方案旨在从业务、技术、安全机制等不同角度来阐述预付卡平台搭建、运营的工作原理和运作流程。

系统的基本功能简单归纳如下：

|  |  |  |
| --- | --- | --- |
| **预付卡系统平台功能项** | **子功能（二级）** | **备注** |
| **平台发卡子系统** | * 实现平台的自发卡 * 平台自己发的卡可以在所有商户中使有 * 支持合作代理方发卡 * 卡档的安全管理 * 随机储值密码管理 * 个性化验密机制 | 支持多个卡类，如福利卡，消费卡等，卡的特性可以自由设置，特性有交易属性，密码属性等 |
| **联机交易子系统** | * 实现实时的联机交易 * 支持消费，消费冲正，消费撤消，消费撤消的冲正，退货，查询 | 支持结算类交易，处理自动清算。  处理卡账户的扣账，换卡，充值等处理 |
| **监控服务与监控机系统** | * 实现多个监控 * 记录和监控交易 * 实时显示当前处理的交易，并显示处理结果 | 显示系统状态，便于工作人员及结决系统问题 |
| **系统管理后台服务** | * 实现卡业务管理 * 各类系统设置 * 处理开卡 * 处理开户,批量 | 如单个开户  处理商户管理  处理安全稽核日志  卡BIN管理等 |
| **软加密处理单元** | * 密钥的分发与管理 * 密码的加解密 * 卡片密码的生成 * 卡账户余额的加解密处理 |  |
| **卡业务管理子系统** | * 系统管理 * 操作员管理 * 路由管理 * 发卡方管理 * 商户管理 * 卡类管理 * 积分规则管理 | 卡段管理  受理卡类管理  终端管理  开户管理  开卡管理  扣率管理  结算处理  各类报表统计查询等  积分规则设定 |
| **网站接口处理模块** | * 负责与管理界面的通讯 * 安全认证 * 交易请求与响应的组包处理 |  |
| **帐务处理模块** | * 转流水处理 * 日终备份 * 数据清算 * 划账单生成 * 手续费计算 * 清算报表和结算报表处理 * 商户入账数据计算统计 * 受理方入账数据计算统计 |  |
| **互联网交易转发前置** | * 互联网交易转加密 * 商场前置收单接口 * 无线POS收单接口 |  |
| **财务清算划账管理** | * 商户开户行信息管理 * 商户结算周期设置 * 商户入账处理 * 自动并生商户划账数据 * 划账核销 | 财务划账辅助功能 |
| **商户查询平台** | * 历史交易查询 * 入账记录查询 * 交易明细查询 * 内部操作员管理 | 为商户提供的在线对账平台 |
| **持卡人查询平台** | * 历史交易查询 * 交易明细查询 * 攺卡密等功能 |  |

测试。

# 3.3 系统总体设计

技术架构图，。

1. **礼品卡生产(Product)业务域**
   1. 主副项目的库存管理缺乏统一标准
   2. 不同用途的主副项目属性缺乏标准规范
   3. 当前的礼品卡序列号生成方式存在已知缺陷
2. **礼品卡销售(Order)业务域**
   1. 商品概念未能进入订单处理流程目前订单处理不依赖订单包含的商品，而依赖订单类别（OrderCategory）
   2. 订单与订单中商品的关系难以扩展
   3. 订单处理流程中未区分主次，全部串行  
      对主流程（如挑卡、落订单）和次要流程（如积分、通知短信发送）全部串行处理
   4. 订单处理流程中所有的业务场景的挑卡只有排队进行。无伸缩性可言  
      目前各业务场景的挑卡能力极限未知。
   5. 线下销售业务未曾收口  
      考虑是否能向合作商订单系统收口
   6. 支付平台接入未曾收口
3. **礼品卡售后(AfterSale)业务域**
4. **礼品卡使用(Account)业务域**
   1. 消费记录、TravelTicket记录持续增长且难以伸缩

数据量在不断增加，一旦触及数据库能力极限系统将无法正常工作。

* 1. 充值记录缺乏清理历史数据的预案

1. **核算**
2. **安全**

需排查出礼品卡生产，销售，售后，使用过程中所有需要幂等控制，而尚未进行控制的场景

业务模块划分，。

技术模块划分，试测试测试测试测试测试测试测试测试测试测试测试测试。

服务器部署情况，测试

交易模块

其他模块

# 第四章 系统详细设计与实现

# 4.1 Product

礼品卡生产(Product)业务域

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

采用同步

后赋值流程（券）：这个流程主要是业务卖实体卡的流程  
PS：此流程审核控制非常严格，造数据请勿用此流程！   
5684加主项目，要填数量（不填总金额）  
5685主项加完要审核  
5688制卡导出，要跑job  
5689制卡导出审核，同时发送邮件（申请人收解压密码），申请人  
5684加副项目，填金额和数量  
5686副项目审核（自动，费用），收入类，费用类，费用类自动审批  
5698副项目复核  
5684到款录入（要从副项目进行到款）  
5687到款审核（自动，费用）  
6070到款复核  
online领用

直充

后赋值

预赋值

库存不足的报警，以及自动化处理流程

实体卡

电子卡

序列号，卡券的设计

# 4.2 Order

111

## 开篇

在OLTP系统领域，我们在很多业务场景下都会面临事务一致性方面的需求，例如最经典的Bob给Smith转账的案例。传统的企业开发，系统往往是以单体应用形式存在的，也没有横跨多个数据库。我们通常只需借助开发平台中特有数据访问技术和框架（例如Spring、JDBC、ADO.NET），结合关系型数据库自带的事务管理机制来实现事务性的需求。关系型数据库通常具有ACID特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。

而大型互联网平台往往是由一系列分布式系统构成的，开发语言平台和技术栈也相对比较杂，尤其是在SOA和微服务架构盛行的今天，一个看起来简单的功能，内部可能需要调用多个“服务”并操作多个数据库或分片来实现，情况往往会复杂很多。单一的技术手段和解决方案，已经无法应对和满足这些复杂的场景了。

对分布式系统有过研究的读者，可能听说过“CAP定律”、“Base理论”等，非常巧的是，化学理论中ACID是酸、Base恰好是碱。这里笔者不对这些概念做过多的解释，有兴趣的读者可以查看相关参考资料。CAP定律如下图：

在分布式系统中，同时满足“CAP定律”中的“一致性”、“可用性”和“分区容错性”三者是不可能的，这比现实中找对象需同时满足“高、富、帅”或“白、富、美”更加困难。在互联网领域的绝大多数的场景，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

## 1.分布式事务

提到分布式系统，必然要提到分布式事务。要想理解分布式事务，不得不先介绍一下两阶段提交协议。先举个简单但不精准的例子来说明：

第一阶段，张老师作为“协调者”，给小强和小明（参与者、节点）发微信，组织他们俩明天8点在学校门口集合，一起去爬山，然后开始等待小强和小明答复。

第二阶段，如果小强和小明都回答没问题，那么大家如约而至。如果小强或者小明其中一人回答说“明天没空，不行”，那么张老师会立即通知小强和小明“爬山活动取消”。

细心的读者会发现，这个过程中可能有很多问题的。如果小强没看手机，那么张老师会一直等着答复，小明可能在家里把爬山装备都准备好了却一直等着张老师确认信息。更严重的是，如果到明天8点小强还没有答复，那么就算“超时”了，那小明到底去还是不去集合爬山呢？

这就是两阶段提交协议的弊病，所以后来业界又引入了三阶段提交协议来解决该类问题。

两阶段提交协议在主流开发语言平台，数据库产品中都有广泛应用和实现的，下面来介绍一下XOpen组织提供的DTP模型图：

在JavaEE平台下，WebLogic、Webshare等主流商用的应用服务器提供了JTA的实现和支持。而在Tomcat下是没有实现的（其实笔者并不认为Tomcat能算是JavaEE应用服务器），这就需要借助第三方的框架Jotm、Automikos等来实现，两者均支持spring事务整合。

而在Windows .NET平台中，则可以借助ado.net中的TransactionScop API来编程实现，还必须配置和借助Windows操作系统中的MSDTC服务。如果你的数据库使用的mysql，并且mysql是部署在Linux平台上的，那么是无法支持分布式事务的。 由于篇幅关系，这里不展开，感兴趣的读者可以自行查阅相关资料并实践。

总结：这种方式实现难度不算太高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况。但分布式事务对性能的影响会比较大，不适合高并发和高性能要求的场景。

## 2.提供回滚接口

在服务化架构中，功能X，需要去协调后端的A、B甚至更多的原子服务。那么问题来了，假如A和B其中一个调用失败了，那可怎么办呢？

在笔者的工作中经常遇到这类问题，往往提供了一个BFF层来协调调用A、B服务。如果有些是需要同步返回结果的，我会尽量按照“串行”的方式去调用。如果调用A失败，则不会盲目去调用B。如果调用A成功，而调用B失败，会尝试去回滚刚刚对A的调用操作。

当然，有些时候我们不必严格提供单独对应的回滚接口，可以通过传递参数巧妙的实现。

这样的情况，我们会尽量把可提供回滚接口的服务放在前面。举个例子说明：

我们的某个论坛网站，每天登录成功后会奖励用户5个积分，但是积分和用户又是两套独立的子系统服务，对应不同的DB，这控制起来就比较麻烦了。解决思路：

1. 把登录和加积分的服务调用放在BFF层一个本地方法中。
2. 当用户请求登录接口时，先执行加积分操作，加分成功后再执行登录操作
3. 如果登录成功，那当然最好了，积分也加成功了。如果登录失败，则调用加积分对应的回滚接口（执行减积分的操作）。

总结：这种方式缺点比较多，通常在复杂场景下是不推荐使用的，除非是非常简单的场景，非常容易提供回滚，而且依赖的服务也非常少的情况。

这种实现方式会造成代码量庞大，耦合性高。而且非常有局限性，因为有很多的业务是无法很简单的实现回滚的，如果串行的服务很多，回滚的成本实在太高。

## 3.本地消息表

这种实现方式的思路，其实是源于ebay，后来通过支付宝等公司的布道，在业内广泛使用。其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。如果不考虑性能及设计优雅，借助关系型数据库中的表即可实现。

举个经典的跨行转账的例子来描述。

第一步伪代码如下，扣款1W，通过本地事务保证了凭证消息插入到消息表中。

第二步，通知对方银行账户上加1W了。那问题来了，如何通知到对方呢？

通常采用两种方式：

1. 采用时效性高的MQ，由对方订阅消息并监听，有消息时自动触发事件
2. 采用定时轮询扫描的方式，去检查消息表的数据。

两种方式其实各有利弊，仅仅依靠MQ，可能会出现通知失败的问题。而过于频繁的定时轮询，效率也不是最佳的（90%是无用功）。所以，我们一般会把两种方式结合起来使用。

解决了通知的问题，又有新的问题了。万一这消息有重复被消费，往用户帐号上多加了钱，那岂不是后果很严重？

仔细思考，其实我们可以消息消费方，也通过一个“消费状态表”来记录消费状态。在执行“加款”操作之前，检测下该消息（提供标识）是否已经消费过，消费完成后，通过本地事务控制来更新这个“消费状态表”。这样子就避免重复消费的问题。

总结：上诉的方式是一种非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。但是，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。所以，在真正的高并发场景下，该方案也会有瓶颈和限制的。

第二步，通知对方银行账户上加1W了。那问题来了，如何通知到对方呢？

通常采用两种方式：

1. 采用时效性高的MQ，由对方订阅消息并监听，有消息时自动触发事件
2. 采用定时轮询扫描的方式，去检查消息表的数据。

两种方式其实各有利弊，仅仅依靠MQ，可能会出现通知失败的问题。而过于频繁的定时轮询，效率也不是最佳的（90%是无用功）。所以，我们一般会把两种方式结合起来使用。

解决了通知的问题，又有新的问题了。万一这消息有重复被消费，往用户帐号上多加了钱，那岂不是后果很严重？

仔细思考，其实我们可以消息消费方，也通过一个“消费状态表”来记录消费状态。在执行“加款”操作之前，检测下该消息（提供标识）是否已经消费过，消费完成后，通过本地事务控制来更新这个“消费状态表”。这样子就避免重复消费的问题。

总结：上诉的方式是一种非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。但是，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。所以，在真正的高并发场景下，该方案也会有瓶颈和限制的。

本地消息表

## 4.MQ（非事务消息）

通常情况下，在使用非事务消息支持的MQ产品时，我们很难将业务操作与对MQ的操作放在一个本地事务域中管理。通俗点描述，还是以上述提到的“跨行转账”为例，我们很难保证在扣款完成之后对MQ投递消息的操作就一定能成功。这样一致性似乎很难保证。

先从消息生产者这端来分析，请看伪代码：

根据上述代码及注释，我们来分析下可能的情况：

1. 操作数据库成功，向MQ中投递消息也成功，皆大欢喜
2. 操作数据库失败，不会向MQ中投递消息了
3. 操作数据库成功，但是向MQ中投递消息时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚

从上面分析的几种情况来看，貌似问题都不大的。那么我们来分析下消费者端面临的问题：

1. 消息出列后，消费者对应的业务操作要执行成功。如果业务执行失败，消息不能失效或者丢失。需要保证消息与业务操作一致
2. 尽量避免消息重复消费。如果重复消费，也不能因此影响业务结果

如何保证消息与业务操作一致，不丢失？

主流的MQ产品都具有持久化消息的功能。如果消费者宕机或者消费失败，都可以执行重试机制的（有些MQ可以自定义重试次数）。

如何避免消息被重复消费造成的问题？

1. 保证消费者调用业务的服务接口的幂等性
2. 通过消费日志或者类似状态表来记录消费状态，便于判断（建议在业务上自行实现，而不依赖MQ产品提供该特性）

## 5.MQ（事务消息）

举个例子，Bob向Smith转账，那我们到底是先发送消息，还是先执行扣款操作？

好像都可能会出问题。如果先发消息，扣款操作失败，那么Smith的账户里面会多出一笔钱。反过来，如果先执行扣款操作，后发送消息，那有可能扣款成功了但是消息没发出去，Smith收不到钱。除了上面介绍的通过异常捕获和回滚的方式外，还有没有其他的思路呢？

下面以阿里巴巴的RocketMQ中间件为例，分析下其设计和实现思路。

RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事物，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。细心的读者可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描消息集群中的事物消息，这时候发现了Prepared消息，它会向消息发送者确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。如下图：

总结：据笔者的了解，各大知名的电商平台和互联网公司，几乎都是采用类似的设计思路来实现“最终一致性”的。这种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大。目前主流的开源MQ（ActiveMQ、RabbitMQ、Kafka）均未实现对事务消息的支持，所以需二次开发或者新造轮子。比较遗憾的是，RocketMQ事务消息部分的代码也并未开源，需要自己去实现。

## 其他补偿方式

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了success字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。

其实这就是一个很典型的补偿例子，跟一些MQ重试补偿机制很类似。

一般成熟的系统中，对于级别较高的服务和接口，整体的可用性通常都会很高。如果有些业务由于瞬时的网络故障或调用超时等问题，那么这种重试机制其实是非常有效的。

当然，考虑个比较极端的场景，假如系统自身有bug或者程序逻辑有问题，那么重试1W次那也是无济于事的。那岂不是就发生了“明明已经付款，却显示未付款不发货”类似的悲剧？

其实为了交易系统更可靠，我们一般会在类似交易这种高级别的服务代码中，加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。

在某些特殊的情况下，还会有“人工补偿”的，这也是最后一道屏障。

试测试。

# 4.3 AfterSale

分布式锁的应用

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

幂等校验

订单支付回调，实时调用支付，订单页的处理

库存的处理（超买，超卖）

支付部分的处理

对账的处理

111

lipinpay，

lipincheckbalance

订单的对账，出错后的报错等问题

# 4.4 Account

幂等校验

1111

扣款，退款

日志

收支明细（账户部分，客户信息密文保存）

前端模块

111

Lizard，兼容三种不同的页面

话术，有效增加PV和UV，转化率等信息。

前端的打包，sdk等等信息

api的调用，H5网关

Online(PC, H5，自适应框架)，Offline两部分

风控的处理，黑名单

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

# 4.5 数据库设计

11111

售前阶段

主副项目，多套主副项目换为一套主副项目。

制卡模块

RechargeCard,

LaterAssignRechargeCard

select rechargecode,count(1),sum(amount) from rechargerecord where createtime >= '20170825' and createtime <= '20170826' and (rechargecode != '' or referrertype = 1)

group by rechargecode

SEQ rechargecode Column Column1

1 100004 654 77529.50

2 100007 4 8420.00

3 100012 15 993.00

4 100018 23 760.00

5 100005 669 581766.50

6 NULL 7522 8015922.00

7 100002 25 27500.00

8 100011 646 18550.00

9 100019 136 24685.00

10 100014 6 749998.00

11 100000 1234 871260.00

12 100003 1733 206299.00

13 100001 434 470470.00

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

订单模块

TMSaleOrder

交易模块

这部分接口并发数比较多

其他模块：变更，结算，合作

内部：火车票，飞机票，报销等问题

外部：微信，分期乐等

管理费的收取

对这么多年工作经验的提炼和升华

尽量依托某个具体系统选题

最好是近一年内基本完成实现的系统

可以是“虚拟系统”

难点的选择

经验分享的思路（正向）

问题解决导向的思路（反向）

材料的拥有情况

另见核心节部分

# 4.6 系统用例介绍

预冻结，冻结，扣款，退款

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

# 第五章 总结与展望

# 5.1 本文内容总结

111

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

1111

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

# 5.2 未来工作展望

111测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试测试。

# 参考文献

[1] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[22] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[3] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[4] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[5] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[1] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[22] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[3] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[4] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[5] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[1] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[22] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[3] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[4] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[5] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[1] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[22] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[3] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[4] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[5] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[1] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[22] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[3] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[4] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[5] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

[21] 新浪微博.OAuth基本流程简介[EB/OL].http://open.weibo.com/wiki/OAuth#.E6.A6.82.E8.BF.B0, 30th Mar 2012

# 致 谢

本论文的写作从开题、搜集资料、起草到最后成文、修订。诸多环节无一不得到导师、同学、同事和亲友的帮助。在此表示衷心地感谢！

首先，也是最重要的，要感谢我的指导老师，XXX老师。在整个论文的写作过程中，XXX老师对我进行了无私的指导和帮助，不厌其烦、逐字逐句地帮助我进行论文的修正和改进。从每一个细节中都体现出XXX老师严谨的治学态度、渊博的学术知识、诲人不倦的敬业精神以及宽容的待人风范。

其次，感谢我的同事在我搜集整理资料中给予的无私的支持。

同时，感谢这篇文论所涉及的各位学者。本文引用数位学者的研究文献，各位学者的研究成果给了我很大的帮助和启发。

最后，感谢复旦大学所提供的一流的学习环境和氛围，使我得以顺利地完成这次论文的写作。

**论文独创性声明**

本论文是我个人在导师指导下进行的研究工作及取得的研究成果。论文中除了特别加以标注和致谢的地方外，不包含其他人或其它机构已经发表或撰写过的研究成果。其他同志对本研究的启发和所做的贡献均已在论文中作了明确的声明并表示了谢意。

作者签名： 日期： 2018.1

**论文使用授权声明**

本人完全了解复旦大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。保密的论文在解密后遵守此规定。

作者签名： 导师签名： 日期： 2018.1