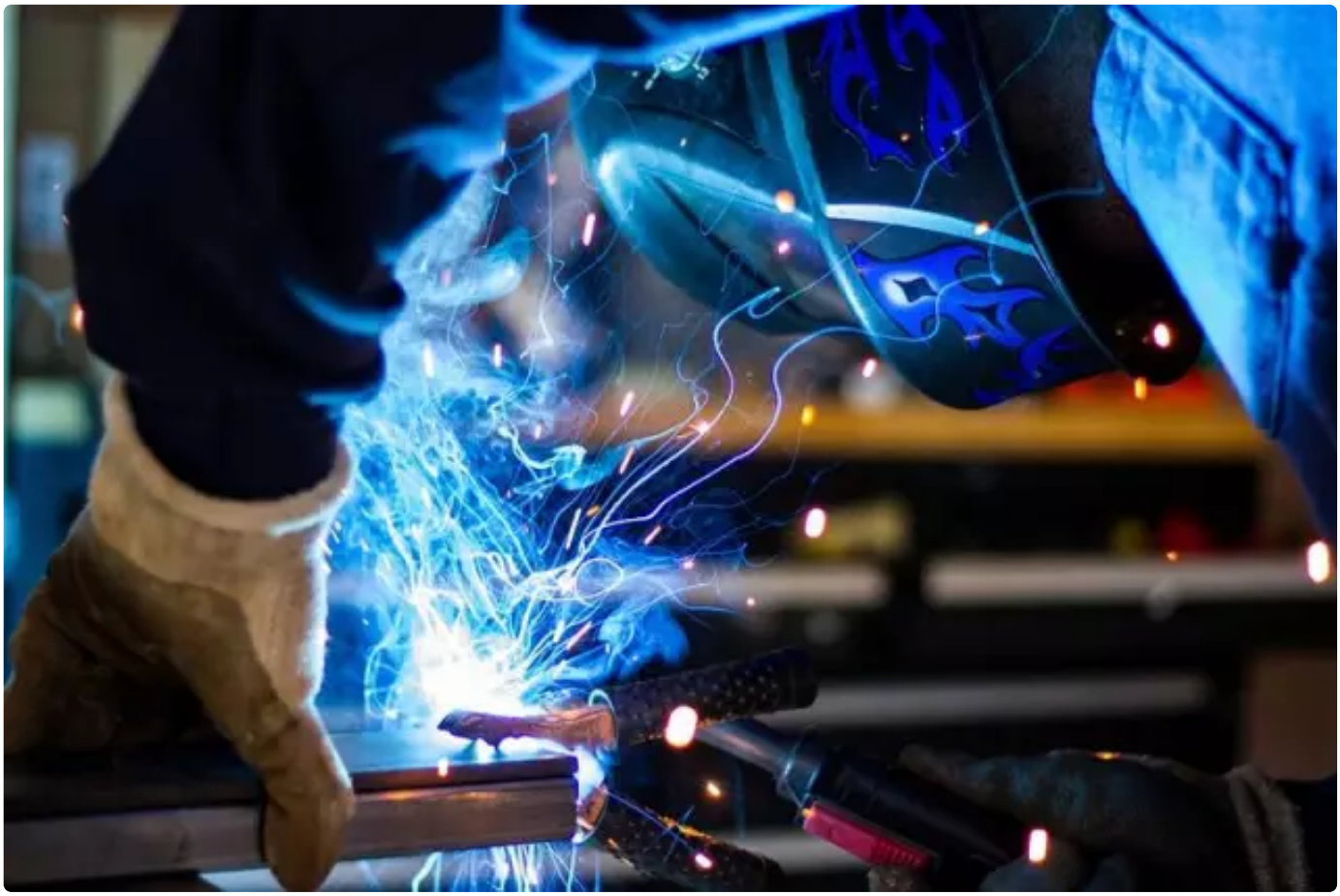
实战案例解析：去哪儿网支付系统架构演进全历程

吕博 InfoQ 2017-01-19



作者 | 吕博

编辑 | 小智

本文将为你详细讲述去哪儿网的支付系统架构演进的全历程，这中间的设计思路是怎样的，踩过怎样的坑，做过哪些可供参考的改进，各种经验分享都在这个实战案例里！

去哪儿支付系统自2011年搭建以来，在五年的时间里逐渐从一个高耦合的单一系统发展为众多子系统组成的高并发、高可用、支持多种交易支付业务的分布式系统。业务从最初的非代收到现在多种非代收、代收场景的支持，B2B业务的从无到有，支付方式从单一网银支付到现在银行卡、拿去花、代金券、红包、立减、积分、趣游宝等多种的组合，订单从单笔支付到多个订单同时支付和多次付款。下面对整体的演变过程进行简单的介绍。

支付系统1.0

新的业务系统初建时，业务逻辑相对简单，业务量也比较小，为了能够快速实现功能，发布上线，大多数团队都会把所有的逻辑都耦合在一个系统。这对于初期业务的快速迭代是有一定好处的。毫不例外，支付交易系统也采用了这样的方式。如下图所示。

一个支付系统不例外包括几个重要组成部分：收银台、交易、支付、网关、账务。

- 收银台：用于展示支付详情、提供各种多样支付方式的选择
- 交易：收单规则和交易规则处理
- 支付：处理各种组合的支付方式，如银行卡、用户余额、信用付、拿去花、红包、代金券、立减、积分等

- 账务：用来记录所有交易、资金往来的明细，财务会计记账
- 网关：用于对接银行通道、第三方支付通道（微信、支付宝）

在业务量不大的情况下，这样的系统结构没有问题。随着更多业务的接入，各种复杂的功能逻辑加入，系统处理起来有点吃力，主要表现以下几个方面：

1. 系统容灾能力：所有的功能都集中在一起，一但某个功能出问题，直接影响全局
2. 系统扩容：在一个分布式系统中，决定系统性能的取决于最差的部分，整体扩容效果差
3. 开发成本高：团队成员的增加，功能的复杂，多个项目并行时，开发效率极低
4. 更多更复杂业务：结构不合理，不能满足业务发展需要
5. 系统职责混乱：如收银台只是简单维护银行列表

在这样的一些背景下，2.0系统应运而生。

支付系统2.0

2.0时代是支付交易系统快速发展的一个重要时段。在此过程中，不仅要从系统架构上进行服务化的拆分，而且需要支持更复杂的业务。

服务化拆分

网关拆分

首先对相对比较独立的网关进行拆分，网关在整个支付系统中属于底层基础服务，是比较重要的基础设施。对外能够提供怎么样的支付交易服务，很多都取决于网关能力的建设。

网关有一些显著特征，它是一个可高度抽象的业务。对外可以抽象到支付、退款、查询这些标准的服务。因此优先将这部分拆分，一是为了能够更好的打好基础，二是其能够独立的发展，三是这部分也相对好实施。

网关的拆分路由系统起到至关重要的作用，对于多通道支付的支持和智能化选择发挥着巨大作用。

账务系统的拆分

做交易支付业务，重要的一件事要记清楚账。记账可以很简单的记录来往流水，也可以更加专业的记财务会计账。在拆分前系统只是记录了交易流水，拆分后实现了更加专业和复杂的

复式记账。

新账务系统的一个简单流程图：

会员系统的独立

会员系统与交易系统本身只是一个依赖关系，在交易支付系统看来只是一个业务系统。比如会员充值业务可以看做是一笔支付交易。为了摆正各自角色，对于会员部分从原有系统中独立出来。这样一来各自定位更加清晰明了，也方便了各自独立发展。现在的会员系统不仅仅只有一个余额，而且引入实名服务、各种资产管理、交易管理等。

基础服务的拆分

更多的系统拆分独立后，原有公用的某些功能会多次复制重复。为方便集中管理维护，通过对各系统公用逻辑更能的统一，提供集中的基础服务，如安全服务、加验签服务、通知服务、基础信息查询等，如下图中talos系统。

上述几个服务的拆分更多是为从业务方面或者技术驱动来考虑。而典型的交易支付过程是有一个时序过程的。比如下单->交易->收银台->支付->网关->银行。这样一个先后时序也是一个比较好的系统拆分方案。根据这样的一个时序，我们针对性的对每个阶段做了拆分（排除网关和银行部分），如下过程：

1、交易核心(Apollo)

关注于收单方式和交易类型。

收单方面系统已经支持单笔订单支付、批量订单支付。交易类型目前支持直接交易、担保交易、直接分账交易、担保分账交易、预授权交易等。在批量订单支付时各种交易类型可以进行混合。且分账交易同时支持多个账户。交易类型除了上面正向交易外，系统还支持很多后续流程交易、如预授权确认、预授权撤销、退款、担保撤销、二次分账交易等。

多种多样的交易源于各事业部业务的复杂性，比起标准化的支付系统，我们提提供了更多灵活方便的业务来支持。

2、支付核心(minos)

关注于支付方案的组合和执行。

支付方式：银行卡、支付宝、微信、拿去花、趣游宝、余额、积分、红包、代金券、会员红包、立减等多种方式支付。

支付组合：可以单一使用，也可以进行组合使用。组合场景区分资金类型，如银行卡、支付宝、微信每次只能选择一个，其它类资金可多个同时使用。

在有上面基础的支持下，对于同一批次交易订单可也进行多次的组合支付扣款，如酒店信用住付款、拿去花还款等业务场景。下图是支付核心（minos）在系统中的位置：

3、收银台

收银台直接面向用户，因此支付体验至关重要。据统计在支付环节放弃的订单占比还比较大。因此一个方便、简洁易用的收银台对于订单转换是有很大帮助的。目前系统支持的收银台主要有app(native)、app前置收银台、touch、PC预授权收银台、PC多单收银台、PC英文版收银台、PC标准收银台等。收银台在系统中的位置如下图所示。

无线端收银台：

PC端收银台：

4、API接入层

交易系统更多的服务是通过后台接口来完成的，这部分占到整体系统很大的业务比重。如支付后期的资金流转、逆向操作退款等。但也有一些是用来查询一些交易订单相关性的信息。在此背景下，对于api接入层采用读写分离方式处理。如下图ares系统，将底层的各dubbo服务包装提供各种查询类服务。Odin系统是可读写，更多的关注跟核心业务相关的写，如解冻、退款、撤销等。

截止目前，整体系统的一个大体结构如下图所示：

服务化拆分带来的挑战

服务化拆分后，在系统结构上更加清晰了，但对于整体系统的开发管理和日常运营带来更大的挑战。比如下几个方面：

如何提高开发效率

系统拆分后主要提供dubbo服务和对外http(https)服务

1.针对Dubbo服务的约定

1. 接口定义：粒度控制、边界控制。一个接口不能存在模棱两可的情况，只做其一
2. 参数标准：复杂接口使用对象做参数(避免map)、统一父类、支持扩展属性透传、提供create/builder构造合法参数、使用枚举限制参数范围。有效避免调用端参数错传
3. 返回值：统一QResponse封装、错误码管理(非数字形式含义明确、按业务区分避免重复等)
4. 业务模板：定义标准业务处理流程、标准化异常处理
5. 接口文档化：定义好接口后，通过注解动态生成接口文档

2.针对http服务的约定

a）接口参数：command、校验器、参数类型配置化。

command中定义接口信息，包括请求返回参数、每个参数的参数类型、参数的校验器、参数类型的校验器。校验器可以组合使用，也可以自定义实现扩展。如下示例：

```
Command定义：
<commands>

<command name="forex_queryExchangeRate">
    <cnName>汇率查询接口</cnName>
    <version>20150808</version>
    <desc>查询本币和目标币种汇率</desc>
    <request>
        <param name="localCurrType" required="true">
            <validator id="CURID"/>
        </param>
        <param name="targetCurrType" required="true">
            <validator id="CURID"/>
        </param>
    </request>

    <!-- 返回参数部分 -->
    <response>
        <param name="localCurrType">
            <cnName>本币</cnName>
            <required>true</required>
        </param>
        <param name="targetCurrType">
            <cnName>目标币种</cnName>
            <required>true</required>
        </param>
        <param name="sellingPrice">
            <cnName>卖出价</cnName>
            <required>true</required>
        </param>
        <param name="buyingPrice">
            <cnName>购买价</cnName>
            <required>true</required>
        </param>
        <param name="rateTime">
            <cnName>汇率时间</cnName>
            <required>true</required>
        </param>
    </response>
</command>
```

```
</commands>
校验器：
<validators>
<validator id="CURID" type="Regex">
    <pattern>^[A-Z]{3}$</pattern>
    </validator>
</validators>
参数类型：
<paramTypes>
<paramType name="merchantCode">
    <cnName>商户号</cnName>
    <desc>用来区分不同商户</desc>
    <type>java.lang.String</type>
    <example>testbgd</example>
    <validator type="Regex">
        <pattern>^[A-Za-z0-9]{1,20}$</pattern>
      </validator>
    </paramType>
</paramTypes>
```

b）并发控制

在某些操作场景下，对于并发写会有一些问题，此时可以通过依赖cache加锁来控制。比如通过在接口增加注解来启用。可以指定接口参数来作为锁的lockKey ，指定锁失效时间和重试次数，并指定异常时(lockGotExIgnore)的处理方案。

```
@RequestLock(lockKeyPrefix = "combdaikoupay:",

        lockKey = "${parentMerchantCode}_${parentTradeNo}",
        lockKeyParamMustExists = true,
        lockKeyExpireSecs = 5,
        lockUsedRetryTimes = 0,
        lockUsedRetryLockIntervalMills = 500,
        lockGotExIgnore = false)
```

c）流量控制

流控目前分两种：qps、并行数。

qps分为节点、集群、接口节点、接口集群。通过对每秒中的请求计数进行控制，大于预设阀值(可动态调整)则拒绝访问同时减少计数，否则通过不减少计数。

行数主要是为了解决请求横跨多秒的情况。此时qps满足条件但整体的访问量在递增，对系统的吞吐量造成影响。大于预设阀值(可动态调整)则拒绝访问。每次请求结束减少计数。

d）安全校验

接口权限：对接口的访问权限进行统一管理和验证，粒度控制到访问者、被访问系统、接口、版本号

接口签名：避免接口参数在传递过程中发生串改

e）统一监控

包括接口计数、响应时长和错误码统计三个维度

f）接口文档化

依赖前面command、校验器、参数类型配置进行解析生成

如何管理多个系统?

1. 接口监控模板化：http、Dubbo多系统统一模板，集中展示管理。
2. 组件可监控化：Redis/Memcache、Mybatis 、Lock 、QMQ 、 EventBus 、DataSource 、JobScheduler
3. 监控面板自动化生成：Python自动化生成脚本，新创建系统只需要提供系统名称和面板配置节点即可生成标准监控面板
4. 系统硬件资源、tomcat、业务关键指标可视化监控

如何高效日常运营?

对于各个场景的关键流程进行格式化日志输出,集中收集处理。如orderLog、userLog、cardLog、binlog、busilog、tracelog、pagelog...

服务化拆分过程中DB处理

分表

随着业务量增加，单表数据量过大，操作压力大。因此分表势在必行。常用的分表策略如按照时间来分表，如月表，季表，按照某个key来hash分表，也可以将两种结合起来使用。分

表的好处是可方便将历史数据进行迁移，减少在线数据量，分散单表压力。

分库、多实例

多库单实例，多业务单库。部分业务存在问题会影响全局，从而会拖垮整个集群。因此在业务系统拆分后，db的拆分也是重要的一个环节。举一个例支付库拆分的例子。支付交易的表都在同一个库中，由于磁盘容量问题和业务已经拆分，因此决定进行拆库。稳妥起见，我们采用保守方案，先对目前实例做一个从库，然后给需要拆分出来的库创建一个新的用户U，切换时先收回U的写权限，然后等待主从同步完成，，确定相关表没有写入后将U切到新的实例上。然后删除各自库中无关的表。

读写分离、读负载均衡

很多业务读多写少，使用MMM结构，基本上只有一台在工作，不仅资源闲置且不利于整体集群的稳定性。引入读写分离、读负责均衡策略。有效使用硬件资源，且降低每台服务器压力。

a）读写负载均衡

b）多动态源

c）多库动态源读负载均衡

异步化使用

1. servlet3异步：释放出http线程提高系统整体吞吐量，可隔离开不同业务的工作线程
2. qmq：使用最广泛也更灵活的异步
3. dubbo：对于服务提供者响应比较慢的情况

servlet异步和qmq结合的场景如下图所示。流程为http服务接到组合扣款请求，然后向后端交易系统下单并发起扣款，此时http服务进入轮询等待，根据轮询间隔定时发起对放在cache中的扣款结果查询。交易系统则根据扣款规则以qmq的方式驱动扣款，直至走完所有流程为止(成功，失败，部分支付)。每次扣款结束将结果放入cache中供http服务查询。

轮询式场景如上图中使用，关键在于确定轮询间隔

监控&报警

Java监控模块

嵌入在应用中，指标数据可灵活配置发送方式到多个地方。也支持api接口直接拉取数据

离线监控框架

1. python监控脚本框架,从db、java模块api、redis等获取数据，计算指标并发送
2. 整体架构可插件化、有通用标准功能、也可定制化开发
3. 指标可直接推送至watcher（dashboard）系统添加监控页
4. 报警方式有mail、sms、qtalk

python监控脚本框架主要包含四个重要组件：

1. metric_manager：指标管理器
2. graphite_sender：指标推送
3. Dbpool：数据库链接池管理
4. Scheduler：调度器，定时执行指标数据获取

数据流系统

采用xflume、kafka、storm、hdfs、hbase、redis、hive对业务日志、binlog等实时收集并处理。提供业务日志、订单生命周期日志、各种格式化日志的查询和一些监控指标的计算存储和报警。整体大致流程如下图所示：

报警

业务和系统结构复杂后报警尤为重要。甄别哪些指标是必须报警的和报警阀值的确定是个很复杂的问题。一般有两种情况：一种是明确认为不能出现的，另一种是需要一定计算来决定是否要报警。当然有些基础层的服务出现问题，可能会导致连锁反应，那么如何甄别最直接的问题来报警，避免乱报影响判断是比较难的事情。目前针对这种情况系统会全报出来，然后人工基本判断下，比如接口响应慢报警，此时又出现了DB慢查询报警，那基本可以确认是DB的问题。

A、明确失败报警

日志NPE、业务FAIL、系统ERROR、Access (4xx\5xx)、接口异常、dubbo超时、fullgc、DB慢查询等

B、计算类报警

调用量特别小，波动明细，没有连续性，不具有对比性

期望值：如下图所示，当前值与期望值偏差加大

写在最后

截止目前交易支付系统从收银台、交易、支付、网关、账务、基础服务、监控等各个模块的拆分并独立完善发展，针对高复杂业务和高并发访问的支撑相比以前强大很多。但还有很多不足的地方有待提高和完善。

继续期待交易支付3.0……

本文系Qunar技术沙龙原创文章，已经授权InfoQ公众号转发传播。

作者介绍

吕博，去哪儿网金融事业部研发工程师，毕业于吉林大学，2012年加入去哪儿网。致力于支付平台研发和支付环节的基础服务建设。

今日荐文

点击下方图片即可阅读

腾讯集团副总裁姚星谈AI：真实的希望与隐忧