我的项目

发布 *v1.0*

yuanjh

2021 年 01 月 08 日

Contents:

项目琐碎整理

CHAPTER 1

项目实战 01 网上交易和网站

网站相关技术点简单说明

1.1 动静分离

基本属于网站必备配置了。web 服务器第一层一般都是 nginx，nginx 三大核心功能（动静分离，负载均衡，多站点反代）之一就是动静分离。典型配置是

```
listen       443 ssl;
server_name  nav.xxxx.cn;

root         /home/xxx/WebStackPage/;# 存放静态文件的文件夹
```

长这样的基本就是文件服务器对应的文件夹了。

1.2 http 长连接

由于 http 请求资源（html，图片，js，css）较多，可开启 http 长连接，这个 nginx 中也有了，开启即可。

1.3 负载均衡

一般服务器都是多组（主要是应用服务器，中小型公司静态文件服务器使用一组 nginx 就足够了，毕竟前面还有一层 cdn 挡掉了大部分请求）。这也是 nginx 的三大核心功能之一，典型配置

```
upstream nav{
        server 127.0.0.1:8081;
}
```

这里还可以指定权重和分发策略（我只用过基础等权分发，复杂的没试过，理论上不难，毕竟 nginx 都已经烂大街的成熟了）。

1.4 网页静态化

我们主要是通过建立数据表，保存 jsp 和 html 映射关系，**然后 jsp 中写函数，进行转化**，将 jsp 转为相应的 html，用户看到的连接已经变成了 html 页面。另外还有个**扫表程序，将 jsp 固化为 html 页面**，这是相对落后的技术了（虽然落后，但却简单，稳定，好用，伴随着网站十多年了）。

最新的版本已经改为 spring-mvc 架构了，没有 jsp 页面了。只留下个别页面（基金净值等高频访问且复杂查询的页面）做了静态化。

需要特殊留意的是目前大多数网站都使用了 cdn，**html 会被 cdn 缓存导致用户取不到最新页面**，可以通过 cdn 中配置刷新策略进行规避。

1.5 cdn

这个不在赘述了，简单，好用，尤其是目前大多服务器都部署在云主机上。阿里云腾讯云等都集成 cdn 服务，只要使用其主机就可以很方便的部署上去，不需要自行购置机房建立 cdn 服务器。

唯一需要留意的是 cdn 底层使用 dns 解析，**不同地域解析到不同 ip 的主机**，所以配置的**子域名指向必须是 dns 服务提供方的**。如果子域名还有子子域名，则无法实现（因为子域名解析到 dns 服务器后，子子域名隶属于子域名，也就同时超出我们的配置范畴了）

比如：helo.wangerxiao.cn 的个人博客，启用了 cdn，那么其域名解析列表必然有一项

```
helo.wangerxiao.cn(cname)=>xxx.dns.tencent.com
```

如果我们想使用域名 nihao.helo.wangerxiao.cn 做其他用途，则是不可行的。

1.6 净值监控

每天开市前都要保证次日净值更新完成（一般当晚 12 点前就应该完成了），如果超过这个时间，会发短信，微信，或 email 进行通知。本身就是一个简单爬虫，后台对接了公司内部监控平台，返回的 code-msg 会被监控平台做简答判定后，按照配置的提现方式进行消息推送。

1.7 informatic 同步

跨数据库数据同步方式，比如净值信息是 IA 核算的，同步给 TA 和我们，就是通过 informatic 的同步功能进行的（视图 =》表）。细节不清楚，并非我们工作内容，只是知道。

1.8 乐观锁

主要是直销会涉及用户份额和金额的一些计算，有些使用了悲观锁（for update），有些是乐观锁。

悲观锁还算常见，乐观锁我还是第一次见到。其实乐观锁并不是真实的锁，而是达到锁效果的操作方式。

比如：

```
a=select amout from table01 where name='john'
old_a=a
a++
a--
a*=
a/=
new_a=a
一系列计算后，执行
update  table01 amout=new_a where name='john' and a= old_a
```

这个就称为乐观锁，如果 update 失败呢？（返回 0 条）。

第一种可能：如果 old_a=new_a, 这时说明业务正常，继续执行就是了。

第二种可能：如果 old_a！=new_a（所以此时应该 update 返回 1 条结果）但是未返回更新，说明条件中的 where a= old_a 条件以及不再成立，也就是说期间 a 被别人改过，所以此时应该报错，将整个事务回滚。

这波操作真的溜！！

1.9 数据库结构

主备结构。

目前互联网领域大多**主从模式 + 读写分离**。但是对于基金公司这种并不面临高并发问题，且对**安全性尤为重视的行业**来讲。主从模式一般实在 cap 中选择 ap（保持可用 + 分区容错），牺牲 c（一致性）。但对金融来讲，**C 基本是必选项目**，基于 C 上，如果采用多库，写入只可能更慢（多库同时确认写入成功才会认为真正成功），所以倒不如直接使用单点库。而且金融公司很多都是使用**存过来保证业务的严谨性和正确性**，使用多数据库这会引发其他一系列复杂问题，依赖金融公司的技术实力解决这种较偏的问题显然不大合适（和互联网不同，对金融公司来讲，IT 只是辅助，并不指望去领导 IT 技术革新）。

采用主备结构也是基于基金行业协会的要求（最少得有一个异地备份，多了人家也不限制），为了避免一些地震或者战争等造成数据丢失所做的严谨要求，一般都是热同步 + 异地备份（北京 + 深圳 + 上海等）。

1.10 session 共享

使用 redis+spring 集成套件即可，配置下就能直接使用。

主要是避免用户被转发到其他主机上后，丢失登录信息，还需要重新登录的问题。也算是主流操作吧（如无特殊要求，随大流就对了，方便，坑少，效率高）。

1.11 日志留存

100M 滚动，长期留存的。

交易历史数据好像是 10 年，国家法律规定的，据说是查反洗钱的。

1.12 安全

花钱请专业公司扫漏洞，扫完按照漏洞清单后我们修补，大多是软件版本，旧版有漏洞，升级新版。还有些是弱口令问题或者 admin 页面路径问题。

由于公司业务简单，况且也不大可能有人敢攻击基金公司站点（搞不定白费劲，搞定了蹲牢房，^_^），所以还没遇到过被恶意攻击的情况。

1.13 抢红包抽奖

用过 redis 第一层限制 + 存储过程最终控制，即使抢红包抽奖，整体并发量也不大，不需要复杂技术也能搞定。

CHAPTER 2

项目实战 02 微信支付宝服务号对接

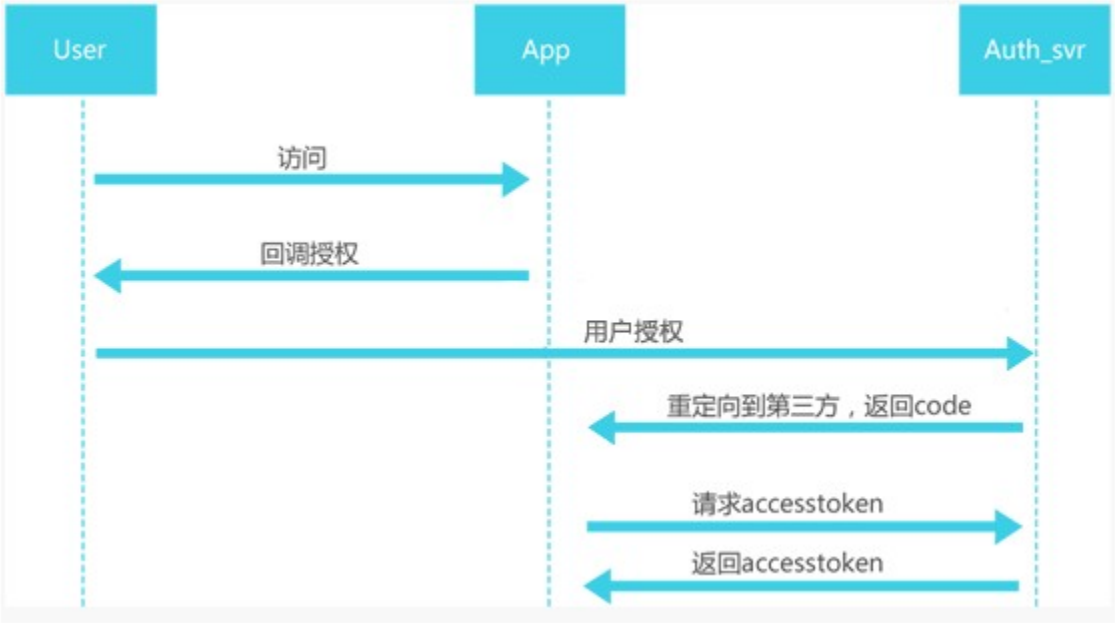
微信公众号，支付宝服务号对接资料现在已经很多了。但当时资料很少的。

微信公众号是前辈开发的，我是进行二次开发。第一代开发主要集中在普通基金净值查询，不需要登录。后来微信提供了 oauth2 的认证，支持公众号提取到用户的更详细的信息，就可以通过公众号和用户账号建立绑定关系，从而实现用户查询资产，以及通过菜单中的网页链接登录 h5 形式的 app，进行基金交易。

2.1 OAUTH2

网页授权方式，需要用户点击确认下，但可以获取到用户更完整的信息。其实就是典型的 **oauth2 认证**。

只需要在菜单中配置网页地址就行了，跳转是微信自动进行的，进行过一次授权后，后台就可以获取到用户的个人信息，获取到用户的唯一 id，后续用户进行绑定操作时就可通过唯一 id 和登录的公司的账号建立映射。



2.2 免登陆

用户第一次会通过 h5 页面登录到公司的简易版交易系统（同时后台建立用户 id 和内部账号的绑定关系）。后续进入 h5 页面时是不需要再次输入用户名密码的。但后台 ctrl 层执行业务逻辑时必然是需要用户登录信息的，这里就需要为用户进行一次“模拟登录”的操作，就是所谓的免登录。

其本质就是通过用户绑定关系，查询到用户个人信息，把**登录时填充的字段，塞到 session 中**，这样后续其他 ctrl 层操作，都和普通网页是一致的（主要是保持业务逻辑一致，否则就需要独立开发接口，进行各种数据校验等，service 可能也需要为其定制，成本会变高）

2.3 如何更新菜单

通过 flidder 发送更新报文的，由于需要先获取授权码，所以需要操作 2 次，才能更新一次菜单，之所以没有开发独立功能，主要是这个功能的使用频率的确很低。由于 post 的报文是 json 格式的，所以更新前最好 format 下，注意层次关系，别更新错了。

目前关于微信公众号的开源工具已经烂大街了，借助开源工具可以很方便的实现这一点。

2.4 如何发布文章

微信公众号有个后台（微信官方的），可以发布文章，自动推送给关注的用户。

2.5 对接内部客服 cms 系统

微信消息直接对接公司的智能机器人（外包购入），如果输入“转人工”，则会将消息转发到公司内部的网页形式的客户服务系统的后台。这也是我在实习期的第一个项目。

主要是通过 flidder 抓包，分析后台客服系统的报文格式，进行 socket 模拟对接的。（主要是客服系统是买的，能用，但没源码，好在报文等都是明文）。客户输入转人工后，就模拟出一个”假人”，通过 socket 连接后台网页客服，由于 socket 可以持续保持，后续连接保证二者一一对应即可。（也就是微信 id-socket 连接的映射，类似一个中转站）。

最初采用轮询方式，挨个扫描 socket，推送相关 mesg，会有卡顿现象。后来改用多线程方式，每各 socket 对应一个线程，负责推送和转发消息，卡顿现象基本消失了。

但也面临一些问题，一个是微信的表情和网页客服表情无法一一对应，微信表情符其实是用特殊文本标识的，当时建立了一个 map，保存这个映射（当然，只是一部分常用的），然后文本替换方式处理。

还有一个就是图片的推送。

2.6 支付宝服务号

这个当时支付宝有专人负责和我司对接的，这种大厂推出新服务时，为了招徕客户以及进行技术验证，都会拉一些各行业的 top 公司，进行前期合作。这样推出服务时，可以同时打出标杆客户，本身是双赢操作。

这个是我负责的，基本把微信做过的流程在走一遍，没有很大差别。不过支付宝服务号用户量的确不如微信。刚开始支付宝强推时还有些用户，后来用户就很少了。

2.7 遇到的坑

微信公众号做活动时，出过一次小事故。最后调查出是微信第一代代码中，有个 wechatid 的字段写成 static 了。导致用户的 id 串了。为了解决这个问题，我们熬了 3 个晚上，能找到的历史日志都挖出来，进行数据修复。

这个问题为啥之前没出现，主要是之前使用人少，所以出问题概率低，而且前期公众号基本没什么人用。后来用了 h5 的 app，可以做抽奖等，用户量激增，并发量变大，并且用户使用公众号的也变多了，这个问题才暴露出来。

CHAPTER 3

项目实战 03 第三方接口对接之注意事项

近期和 p2p 公司合作进行闲置资金的货币基金购买。

核心功能自然是原有的网上交易核心 service 层，之前的 service 层服务的是 ctrl 层，现在 ctrl 层变成了第三方接口，而非常规的网页的 ctrl 层，简单来说就是重写 ctrl 层，返回 json 而非网页。

3.1 何种交互方式

解决问题：客户端如何调用服务器接口 or 服务。

主要有 3 种方案，socket，webservice，json+http。

3.1.1 socket

socket 偏底层，效率高。但是不考虑，因为**开发成本大**。

3.1.2 webservice

优点：支持跨平台，跨语言，远程调用

缺点：wsdl 文件不大容易看懂，并且调试抓包来的内容也难以理解。

从使用角度，好处在于**只需定义好 server 的 interface，根据 interface 生成 wsdl 文件**，对方 (client) 根据 wsdl 文件生成可以调用的 client 的方的 interface，然后直接调用即可。一般需要 server 提供 interface 使用 demo 或者文档，否则裸接口也不大看得明白。

3.1.3 定制型 json

直接使用 http+json

优点：灵活，简单高效。json 没有什么门槛。

最终选择 json+http，主要原因是 json 报文**对人更友好**，容易抓包调试和记录到全局日志中，且报文紧凑，节约网络。

3.2 连接加密和报文加密

解决问题：**报文泄露和篡改**，还有就是**避免报文抵赖**（对方发出的报文，不承认自己发出过，产生纠纷）

金融公司业务肯定是需要加密传输的，这里的加密包含 2 层。

第一层：https 层的加密，也就是非对称加密获取秘钥和对称加密发送报文。

第二层：对 https 传输内容的加密，主要是报文加密 + 加签（己方私钥），然后对整体加密（对方公钥）

整个流程是双加密的过程（https 一次，内容二次）

举例：json content（post 的 body）

```
{
    req:{
        reqId:001,# 唯一 id
        reqType:022,# 操作类型，申购
        data:{
            name:xxx,# 业务参数
            age:xxx,
            amount:100.0,
            vol:100.0
        },
    }
    sign:qrwefafdaf# 己方私钥对 req 的加签
}
```

再对 json content 整体用对方公钥加密，发送给对方。

由于使用对方的公钥加密，所以只要对方私钥不泄露，则报文就是**防偷窥**的。(即使泄露了，责任也在对方一侧)。

由于 sign 有己方私钥的加签，所以报文也做到了防“己方耍赖”的要求。

这种方案**优点就是安全，最重要的是双方都无法抵赖**。

缺点就是慢，一次交互，需要做加密和解密，加签验签，四个流程（如果再考虑 https 自己还有一层非对称 + 对称加密就更复杂了）。好在基金公司并发量都不会太高，即使高了也可以通过升级机器解决（在不行做分布式呗，反正基金公司钱多，能用钱解决的问题都好办）

3.3 免登录

公司自己站点网页上通过登录来校验用户，和第三方对接时，用户在对方网站登录就认为登录过。凡对方接口发过来请求，都认为是用户授权过的操作。

所以对内部接口做免登处理，这个可能需要改动部分 service 层。

3.4 幂等和重试

解决问题：报文已处理和报文未处理情况下的响应丢失问题。

分 2 种情况：

1，报文丢失，真的丢了，server 没收到，所以业务实际没执行

2，报文响应丢失，报文到达 server，server 也处理了。但是 response 后，client 未收到

以上 2 种情况，应该处理的方式是不一样的

第一种，client 重发

第二种，client 发出查询请求，传入刚才未 response 的消息 reqid，查询到执行结果。

但可惜的是，从 client 的角度看，**这 2 种错误，都是同一种现象**，就是自己没有收到 response。

也就是说 client **无法区分到底发生了第一种错误还是第二种错误**。自然也就无法判断目前是那种情况，自己应该采用何中应对之策。而且第二种处理，对不同类型消息，查询到的响应信息是不同的，也就是说各接口需要独立实现额外的查询功能，有**较大开发量**。

解决方案：对所有请求的唯一标志，reqId 建立独立日志表，记录 reqId 和 response。

如果新报文的 reqId

01，不在表中，说明全新请求，正常执行业务逻辑即可。

02，在表中，但是 response 为空，说明这个请求正在处理中，返回约定错误码，让其稍后再试。

03，在表中，且 response 为空，说明这个请求已经处理过，且响应过报文，直接将上次的 response 再返回一次即可。

在这种方案下，如果对方未收到 response，重试即可。

3.5 日志体系

解密前日志，解密后日志，

3.6 研发对接形式

3.6.1 demo 形式 (testcase)

提供我方各接口的自测 test case，加上各函数的字段注释，以及产品部门提供给对方的业务背景知识指导和教育信息。

优点：简单，便于快速启动对接

缺点：沟通成本太高，尤其是缺乏行业背景知识。从我们视角看，各种诡异报文都有，没有申购就赎回（类似于没有买就卖），没有确认就给用户加份额等。

除此之外，字符类型和数字类型有的用 string 表示，有的用 int，有的用 float，非常随意，虽能跑，但感觉在溜冰。一旦我方未校验到，有可能导致非法数据入库。

这种对接模式下，从研发介入到最终上线，大约持续 2 个月左右。6 周对接，2 周联调加灰度。

3.6.2 完善文档 +demo 代码

提供完整的接口文档，包含功能说明，字段名称，含义，格式（int，float，str），取值范围，业务性关联（字段同现，或者二选一等内部逻辑性关联）

样例 resquest，样例 response 等。

优点：减少合作方的一些低级错误。至少数字，字符串保持稳定一致性。参数保证合法区间，字段无逻辑矛盾。

缺点：己方维护文档成本较高，而且是一个持续维护的过程，实际对于部分简单接口，开发自测 30 分钟，维护文档可能需要 2 个小时（编造数据，模拟报文等）

但整体来说可以**较好的提高对接效率**，可以将对接时间缩短到 1 个月左右。文档维护虽然麻烦，但是由于是**一份文档对多个合作方都是通用的**，所以整体还是很划算的。而且也**利于公司内部新人的学习和接手**。

3.6.3 jar 包

最终大招就是为对方提供己方的 jar 包。后期和招行信用卡以及金蝶，用友的对接都是采用这总方式（当然，大客户都做了一些微定制）。

这种方式只需要稍有背景知识即可。至于参数校验，取值合法性，逻辑合法性等都在我方 jar 包内部做了检查。对方只要能梳理好业务逻辑即可。这个门槛已经很低了，基本**类似于淘宝购物的门槛了**。

和招行信用卡对接也不过用了 2 周左右（大约 1 周驻场了），可见，这种方式的效果还是很给力的。

3.7 上线流程

3.7.1 双方自己开发

按照约定提供 demo 或者文档，自己开发自己的功能。

3.7.2 双方联调

双方开发完毕，进行接口对接联调，发现大家接口的一些基础错误（字段缺失，格式错误等）。

同时，也需要做全功能，全流程业务的逻辑测试验证。

3.7.3 双方上测试

双方测试组介入，研发组退出。

测试组有独立的测试环境，可以验证研发的上线步骤/计划是否有问题。

同时也能暴露出研发测试的一些测试遗漏点。

3.7.4 双方上灰度

所谓**灰度就是准生产**了，不过这个生成是对双方公司内部的生产，也就是只有双方公司内部人员才可以操作的流程。

确保**出了问题的话，也都是公司自己人**，不会给陌生用户带来影响。

3.8 其他坑和特殊点总结

3.8.1 1，知识背景

跨行业背景知识很重要，对方如果缺乏对己方的了解，会导致一些非常低级的错误出现，这种错误还是我们不大可能考虑到的（比如没有买就卖，如果没有考虑到，虽然造不成实际损害，但会导致报错信息难以理解）。最好的解决方法是**让对方体验下自己的产品**，比如就给对方 100 块，买 10 多笔基金，在卖 10 多笔，对方自然就明白了。

比产品给他们上半天课效果好多了。

3.8.2 2，永远不要信任对方接口（全部强校验）

永远不要相信对方接口，永远不要相信对方接口，永远不要相信对方接口！

金融行业用 decimal 保存数值是默认行规，人人都知道，但**互联网大多 float** 就够了，会导致 0.999999 这样的数据进来，如果不做校验，会导致一些金额或份额的视觉偏差（本来是整数份，结果显示成 x.99999，虽然实际上也不会给用户带来损失，但会造成**用户恐慌和对公司的不信任**）

3.8.3 3，语言坑 (php json 引号)

php 在处理 json 时会自动对数值添加单引号，这个可能是 php 特定版本的包的问题。

3.8.4 4，float 不精准

上面提到过，不在赘述

3.8.5 5，限制支付方式 (只能走纯接口)

只支持快捷支付，不支持招行直付通 (需要跳转到对方银行鉴权，再跳转回来的,Oauth2 鉴权)。纯粹接口形式无法支持网页跳转，且对接代价高昂。(快捷支付包含最初的通联和后来的快捷支付)

3.8.6 6，RSA 加密（略）

大质树 + 欧拉定理，脑子不够用，看不懂，只要知道公钥私钥不相同，以及公钥加密 + 验签，私钥解密 + 加签就够了。

3.8.7 7，提单时间依据

大多根据公司内部数据库时间，部分渠道经协商可以采用对方接口时间

3.8.8 8，沟通相关

邮箱沟通，万一后续涉及扯皮，可以避免甩锅

3.8.9 9，字段名转换

有些公司是一对多（一个公司对接多个基金公司），未必完全按照我们制定的接口标准，还有些是自己原本就有一套了。当然，同一个业务基本字段都是一致的（感谢基金业协会，一般各公司技术的 boss 都会进行协商出统一交易码和交易核心字段），只是一些非核心，或是针对特定定制业务的字段命名上可能不同，可能需要做字段映射。这个一般是写死到代码里的。一方面使用场景有限，另一方面和定制的特定渠道高度相关，不大好做成通用模块。

3.8.10 10，文件确认对账机制

行规，最终以确认文件为准。

CHAPTER 4

项目实战 04 请求幂等性

4.1 接口调用存在的问题

调用方收不到响应，无法区分请求没过来，还是已经处理过。但是无论哪种情况，都需要“找回对方响应的报文”（可能要根据响应，修改自身状态）

4.2 什么是接口幂等性

任意多次执行所产生的影响均与一次执行的影响相同，

接口幂等性就是用户对于同一操作发起的**一次请求或者多次请求的结果是一致**的，不会因为多次点击而产生了副作用。

4.3 什么情况下需要保证接口的幂等性

必要性: 重试是降低系统失败率的重要手段。

互联网应用一般都是提供 7*24 服务的，而互联网应用本身又是快速迭代，后端系统是随时有可能需要进行发布的。**发布等同于一次宕机（进程被 kill）**，这意味着对于互联网应用的后端系统，宕机是常态而非特例。这也是幂等性和重试的必要性来源之一。

对于业务中需要考虑幂等性的地方一般都是接口的重复请求，重复请求是指同一个请求因为某些原因被多次提交。导致这个情况会有几种场景：

前端重复提交：提交订单，用户快速重复点击多次，造成后端生成多个内容重复的订单。

接口超时重试：对于给第三方调用的接口，为了防止网络抖动或其他原因造成请求丢失，这样的接口一般都会设计成超时重试多次。

消息重复消费：MQ 消息中间件，消息重复消费。

在增删改查 4 个操作中，尤为注意就是增加或者修改，

A: 查询操作

查询对于结果是不会有改变的，查询一次和查询多次，在数据不变的情况下，查询结果是一样的。select 是天然的幂等操作

B: 删除操作

删除一次和多次删除都是把数据删除。(注意可能返回结果不一样，删除的数据不存在，返回 0，删除的数据多条，返回结果多个, 在不考虑返回结果的情况下, **删除操作也是具有幂等性的**)

C: 更新操作

修改在大多场景下结果一样, 但是如果是增量修改是需要保证幂等性的, 如下例子:

把表中 id 为 XXX 的记录的 A 字段值设置为 1, 这种操作不管执行多少次都是幂等的

把表中 id 为 XXX 的记录的 A **字段值增加 1**, 这种操作就不是幂等的

D: 新增操作

增加在重复提交的场景下会出现幂等性问题, 如以上的支付问题

4.4 幂等性实现方式

4.4.1 Token 机制

使用 token 机制实现: 使用 token 机制实现接口幂等性, 通用性强的实现方法

token 特点: 要申请，一次有效性，可以限流

4.4.2 数据库去重表 (唯一索引)

往去重表里插入数据的时候，利用数据库的唯

一索引特性，保证唯一的逻辑。唯一序列号可以是一个字段，例如订单的订单号，也可以是多字段的唯一性组合。

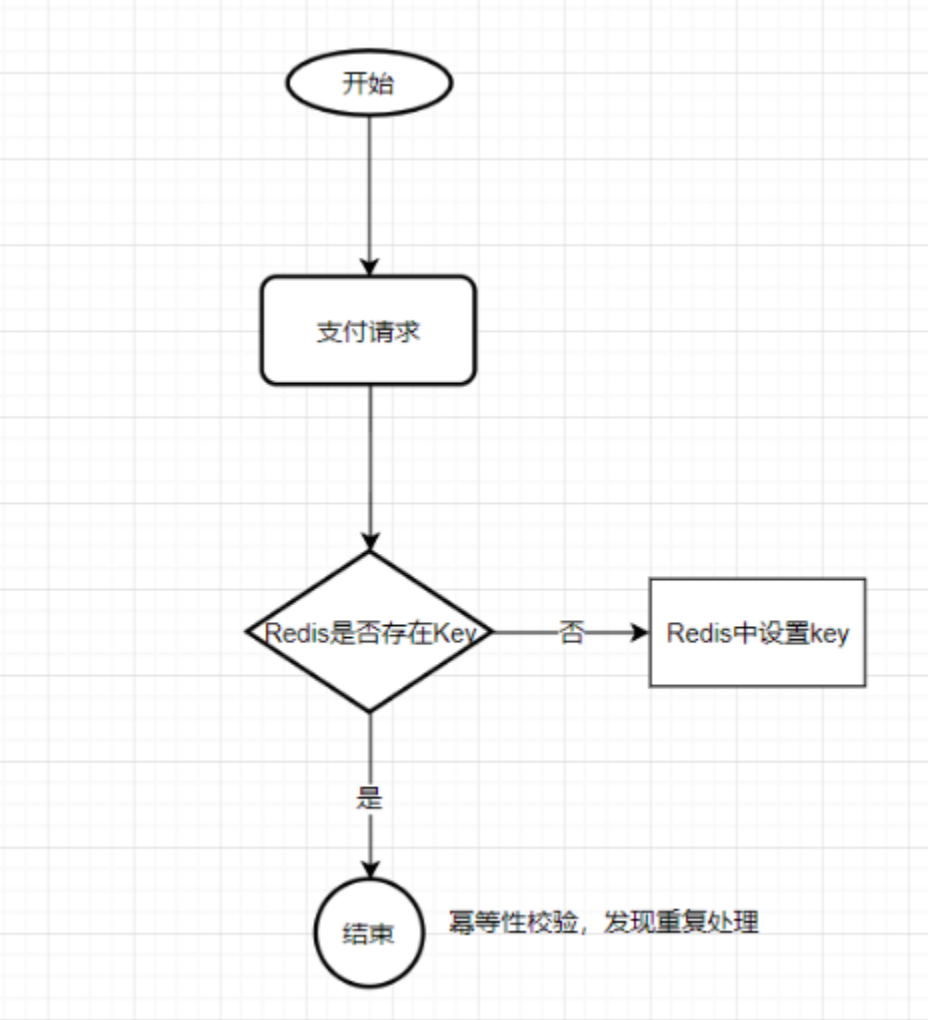
代码方法：提供幂等性校验的接口方法

这个接口的方法具体在项目中合理的使用就看项目要求了，可以通过 @Autowire 注解注入到需要使用的地方，但是缺点就是每个地方都需要调用。我个人推荐的是**自定义一个注解，在需要幂等性保证的接口上加上该注解，然后通过拦截器方法拦截使用**。这样简单便不会造成代码侵入和污染。

另外，使用数据库防重表的方式它有个严重的缺点，那就是系统**容错性不高，如果幂等表所在的数据库连接异常或所在的服务器异常**，则会导致整个系统幂等性校验出问题。如果做数据库备份来防止这种情况，又需要额外忙碌一通了啊。

4.4.3 Redis 实现

Redis 实现的方式就是将唯一序列号作为 Key，唯一序列号的生成方式和上面介绍的防重表的一样，value 可以是你想填的任何信息。唯一序列号也可以是一个字段，例如订单的订单号，也可以是多字段的唯一性组合。当然这里需要设置一个 key 的过期时间，否则 Redis 中会存在过多的 key。



4.4.4 状态机

对于很多业务是有一个业务流转状态的，每个状态都有前置状态和后置状态，以及最后的结束状态。例如流程的待审批，审批中，驳回，重新发起，审批通过，审批拒绝。订单的待提交，待支付，已支付，取消。

以订单为例，已支付的状态的前置状态只能是待支付，而取消状态的前置状态只能是待支付，通过这种状态

机的流转我们就可以控制请求的幂等。

4.4.5 乐观锁和悲观锁也是幂等的解决方案么？

还有些文章把乐观锁和悲观锁也放到幂等的解决方案中了，个人**并不认同这种解释**。

乐观锁和悲观锁是为了解决高并发下的**数据混乱问题**的，和接口幂等和高并发并没有必然联系。**即使一个接口一天只调用一次，也会面临幂等问题**。把昨天一模一样的报文发送给你（唯一序列号相同-对应同一个已支付订单的付款操作），你怎么响应的问题（这个是和业务关联的，个人工作经历，把昨天响应的报文再次 resp 一次，告诉支付成功，支付时间-昨天。有人就会抬杠了，应该响应支付失败吧，毕竟成功的是昨天的，而不是刚刚的）。

4.5 参考

接口的幂等性:https://www.cnblogs.com/huaixiaonian/p/9577567.html

分布式系统中接口的幂等性:https://www.cnblogs.com/jajian/p/10926681.html

使用数据库唯一键实现事务幂等性:https://www.caosh.me/be-tech/idempotence-using-unique-key/

高并发下接口幂等性解决方案:https://blog.csdn.net/u011635492/article/details/81058153

分布式系统接口幂等性：https://nicky-chen.github.io/2018/03/26/interface-idempotency/

CHAPTER 5

项目实战 05 大客户定制相关技术

基于公版，对我司大客户进行定制版本的相关开发。

5.1 公版

5.1.1 整体架构

微服务 spring cloud，抄来的图如下，大多数采用 spring cloud 的都是这个套路。

![ ](_v_images/20201214234001836_723044133.png =641x)

5.1.2 事务机制

二级事务处理，成功 or 回滚。

5.1.3 数据隔离，tenantid 列

优点：所有租户使用同一套数据库，成本低。开发也容易，各查询添加一个条件字段即可。

缺点：隔离级别低，安全性差，数据备份和恢复最困难。

采用 tenant 租户 id 字段进行数据隔离，所有租户数据混合一起，会导致个别租户数据过大会导致所有租户查询速度有受到影响。数据批量更新时也需要格外注意，避免误操作其他 tenant 的数据。

其实 mybatis 有现成的租户隔离插件，MyBatis-Plus，由于历史原因，当时已经积重难返了，只能拿不完美的先用着了。

5.1.4 用户的建立

通过付费，后台建立超级管理员账户，然后密码交给租户（具体到人，比如老板或负责对接实施的管理人员）修改重置，重置后租户可以通过超级管理员，建立其他用户（普通工人，仓库管理员，采购人员等具体角色），设置用户名，初始密码等。建立后，用户可以通过用户名密码登录，查看到自己公司的相关订单，生产单等信息（和用户全新和角色有关）。

由于我们是按照端口付费的，所以对新建用户数量是有限制的。

5.1.5 催收续费

按月定时任务扫数据库，生成邮件通知市场负责人，市场负责人通知具体公司对接人进行续费。

5.1.6 AB 环境升级

由于是微服务架构，所以升级时采用分组升级，一组看做 A 组，另一组看做 B 组。对 A 组做 update 时，nginx 打到 B 组上，对 B 组升级时，nginx 再打回到 A 组上。当然这是不涉及数据库变更的情况下。

如果牵涉数据库变更，由于数据库是共享的（同一个功能的 AB 组都使用了 1 个数据库），所以只能做短暂停机，一般情况数据库的变更都是提前导出部分数据做升级验证，确保所有数据都是可更新的，无遗漏的情况。

如果遇到意外（常在河边走，必然会湿鞋），可以采用提前的备份库顶上，回滚代码的升级。此次升级先暂停，后续再排查原因。不可能让用户一直等着研发调 bug，除非小问题。

5.1.7 解耦和异步 rabbitMq

场景: 当库存变动时或库存配发到车间仓库时，会触发一**系列的通知和生产单流转过程变量**的更新操作。

这部分操作，从变更内容来讲，是必须要完成的（产品业务逻辑定义的）。但是从任务优先级和紧急度来看，既不优先也不紧急 (意味着，即使未通知到，或未重新计算相关变量，也不会导致业务流转出错，只是降低了用户体验而已）。

如果一个接口中这种操作，**占用了较多的处理时间**，则需要做拆分。将任务 push 到消息队列中，然后交给另一个 server 从消息队列取得任务，然后更新表以及缓存等。

5.1.8 低频更新缓存 redis

redis 更多用来当做缓存使用，保存低频的变更信息，比如各租户的底层功能开关。访问频繁但变更频率低，可以看做多 db 层的加速。

这种数据更新后会自动触发 redis 的刷新，没有更新不会主动刷新。

用到才会查询 =》入缓存，也不需要预热

5.1.9 看板优化

看板数据的慢查询部分从 redis 中加载，redis 中定时刷新，等价于将慢查询或复杂计算和页面展示异步化。

缺点：降低实时性

优点：提高页面刷新速度，提升用户体验。

5.1.10 面临问题

大租户拖慢小租户：本质原因是共享了同一个数据表，表数据过大不可避免导致查询速度下降

解决方案：微服务可以将数据打散，降低单个数据库负载，但是单表则无法降低，可以采用主从 + 读写分离进行优化，从服务器可采用多个服务器，这样可降低查询负载。

更新矛盾：有些工厂坚持旧版好用，坚持用旧版，不接受新版

大多数坚持旧版客户，其实只是用不习惯新版而已。昨天能点的按钮今天找不到了。所以这更多是沟通问题，需要产品部门出升级文档，阐述这次迭代升级内容以及原有旧版的功能映射等关系，方便用户理解。

这方面也反向产品部门需求的合理性，我们就开发了很多功能，客户升级后感觉反而难用了，就是产品部门没有把握好用户需求，凭感觉提需求。

功能开关：有些简单的功能开关会导致后台复杂度急剧提升，甚至需要填充额外数据信息。

解决方式：功能开关更多的负责控制展示逻辑（展示不展示，展示哪些，展示顺序），不负责对内部执行逻辑的过多切入。这种变更长期会导致系统数据混乱，不论是对客户还是服务提供商都是非常危险的。

5.2 定制版

5.2.1 特性

独立代码分支，独立机器，独立数据库，独立域名等

5.2.2 可解决哪些问题

个性需求，比如计价工资核算等严格来说不属于 MES 或 ERP 范畴，但是由于系统内部有报工信息，所以可以实现对计价工资的支持，一些复杂的工种区分，计件的阶梯计工资等。

公司的审计需求，部分公司是科创板上市公司，对公司经营数据有一定的安全性要求。像订单，采购单，生产单，生产计划等，对制造业来讲还是比较重要和私密的数据。如果放到公版，会有一定安全风险。毕竟对小公司而言，是没有专业的信息安全部门的，对普通软件研发而言，安全领域还是比较陌生的。

公司独立域名，独立部署可以实现挂到公司独立域名下，不使用 saas 公共域名，看起来可能更正规些。

5.2.3 引入新问题

共版的这个功能我们也要，反正你们有现成的，不用付钱吧

长期分离导致代码差异大，并且新功能需要补充新数据进行支撑。简单来说 (产品) 妥协一时爽，研发火葬场。

5.2.4 无法解决问题

定制化最大问题在于高成本，这个对客户存在，对于研发方依然存在，由于无法公用，导致客户感觉单价高，而对于研发而言，研发成本由于无法多家摊平，而本身研发人员费用相对较高，所以利润空间有限。

5.3 未来前景

saas 整体前景还是比较悲观的，这方面可参考前辈姜孝鹏的经历，如果这种 saas 资深前辈都无法获得成功，那么其他 saas 前景可想而知。巨头扶持的比如钉钉目前尚且处于亏损状态，这可是顶级流量支持 + 大公司背书的 saas。所以，难度可想而知。

个人感觉，国内的浅层客户一般不愿意付费（毕竟人手一份的 windows 都大多盗版）。深层的客户大多都需要定制，一旦定制 saas 本身的成本优势就没了，如果没有很大价格优势，人家当然更愿意自研。

至少就个人经验，续费率很惨，而新客户销售提成占比较高，获客成本太高。

CHAPTER 6

项目实战 07 线上问题定位之自上而下

6.1 查看进程下线程和资源占用

```
使用 ls /proc/pid/task/ 查看线程
使用 ps -eLf 命令/ps aux -L/ps aux -el
使用 pstree
```

查看线程数量

```
cat /proc/19135/status | grep Threads
pstree -p 19135|wc -l
```

6.2 程序状态字段解释

D Uninterruptible sleep (usually IO) 不可中断睡眠

R Running or runnable (on run queue) 正在执行或可执行，表示目前在运行队列里面

S Interruptible sleep (waiting for an event to complete) 可中断睡眠

T Stopped, either by a job control signal or because it is being traced. 停止

W paging (not valid since the 2.6.xx kernel)

X dead (should never be seen)

Z Defunct (“zombie”) process, terminated but not reaped by its parent. 僵尸进程

6.3 查看各进程下的线程数情况

比如某台服务器的 CPU 使用率飙升，通过 top 命令查看是 gitlab 程序占用的 cpu 比较大，”ps -ef|grep gitlab” 发现有很多个 gitlab 程序，现在需要查询 gitlab 各个进程下的线程数情况。批量查询命令如下：

```
# for pid in $(ps -ef|grep -v grep|grep gitlab|awk '{print $2}');do echo ${pid} > /
↪root/a.txt ;cat /proc/${pid}/status|grep Threads > /root/b.txt;paste /root/a.txt /
↪root/b.txt;done|sort -k3 -rn
```

6.4 查看 cpu 使用率告警问题处理案例

查出这个 pid 进程的 cpu 资源各自被哪个线程所占。通过”top -Hp pid” 可以查看该进程下各个线程的 cpu 使用情况；如下：

```
[root@kevin ~]# top -Hp 31969
# 另方式查看子线程: ps -Lf 31969
```

通过 top 命令定位到 cpu 占用率较高的线程之后，继续使用 jstack pid 命令查看当前 java 进程的堆栈状态，这就用到 jstack 工具！jstack 是 java 虚拟机自带的一种堆栈跟踪工具。jstack 用于打印出给定的 java 进程 ID 或 core file 或远程调试服务的 Java 堆栈信息。jstack 可以定位到线程堆栈，根据堆栈信息我们可以定位到具体代码，所以它在 JVM 性能调优中使用得非常多。

6.5 程序卡主跟踪

继续用 strace -p 27678 跟踪，发现卡在 read，文件描述符是 14

```
sangfor ~ # ps -ef | grep main.pyc
root      4795  4794  0 03:11 ?        00:00:17 python /wns/cloud/app/com_host/main.pyc
root      9138 30380  0 06:47 pts/0    00:00:00 grep --colour=auto main.pyc
sangfor ~ # strace -p 4795
Process 4795 attached - interrupt to quit
read(19,
```

现在我们查看一下进程打开的文件描述符 14 代表什么，pipe 文件

```
sangfor ~ # cd /proc/4795/fd
sangfor /proc/4795/fd # ls -l
total 0
lr-x------ 1 root root 64 Jan 15 07:04 0 -> pipe:[16125]
l-wx------ 1 root root 64 Jan 15 07:04 1 -> pipe:[16126]
lrwx------ 1 root root 64 Jan 15 07:04 10 -> socket:[16753]
lr-x------ 1 root root 64 Jan 15 07:04 11 -> /dev/urandom
lrwx------ 1 root root 64 Jan 15 07:04 12 -> socket:[16249]
lrwx------ 1 root root 64 Jan 15 07:04 13 -> socket:[16251]
lrwx------ 1 root root 64 Jan 15 07:04 14 -> socket:[16252]
lrwx------ 1 root root 64 Jan 15 07:04 15 -> socket:[16254]
lrwx------ 1 root root 64 Jan 15 07:04 16 -> socket:[16313]
lrwx------ 1 root root 64 Jan 15 07:04 17 -> socket:[16314]
lrwx------ 1 root root 64 Jan 15 07:04 18 -> socket:[109028]
lr-x------ 1 root root 64 Jan 15 07:04 19 -> pipe:[117025]
```

其实在这里我们也可以使用 lsof 来定位，可以看到进程 27678 打开的 FD 14 是 pipe，这里 u 代表可读可写，r 代表可读

```
sangfor ~ # lsof -d 14
COMMAND     PID USER   FD   TYPE             DEVICE SIZE/OFF     NODE NAME
mongod     1907 root   14u   REG              251,0    36864   130683 /wns/data/
↪mongodb/db/collection-7--588642557116981989.wt
syslog-ng  3446 root   14u  unix 0xffff88012227d800      0t0 40557736 /dev/log
dockerd    4025 root   14u  unix 0xffff8800b8d5d800      0t0    13941 /run/docker/
↪libnetwork/a73bd949b5fbb89c2b8bec3b4ac6af0a948a944958c8b037d9e6c9b324b44331.sock
docker-co  9382 root   14u  0000                0,9        0     9553 anon_inode
docker-co 21204 root   14u  0000                0,9        0     9553 anon_inode
python    27678 root   14r  FIFO                0,8      0t0 38483750 pipe
```

也可以直接查看进程 27678 打开的，可以看到 14 是 pipe

```
sangfor ~ # lsof -p 27678
COMMAND   PID USER   FD   TYPE             DEVICE SIZE/OFF     NODE NAME
python  27678 root    0r  FIFO                0,8      0t0 30690124 pipe
python  27678 root    1w  FIFO                0,8      0t0 30690125 pipe
python  27678 root    2w  FIFO                0,8      0t0 30690126 pipe
python  27678 root    3u  0000                0,9        0     9553 anon_inode
python  27678 root    4u  0000                0,9        0     9553 anon_inode
python  27678 root    5u  pack           30691718      0t0  unknown type=SOCK_RAW
python  27678 root    6w   REG              251,0 76106652   130565 /wns/data/com_
↪host/etc/config/err.log
python  27678 root    7u  IPv4           30691716      0t0      TCP Sangfor:53102->
↪Sangfor:42457 (ESTABLISHED)
```

(continues on next page)

(续上页)

```
python  27678 root    8u  IPv4          30691717      0t0      TCP Sangfor:42457->
↪Sangfor:53102 (ESTABLISHED)
python  27678 root    9u  IPv4          30691731      0t0      TCP db.sdwan:54072->
↪sdwan.io:27017 (ESTABLISHED)
python  27678 root   10u  IPv4          30691732      0t0      TCP db.sdwan:54074->
↪sdwan.io:27017 (ESTABLISHED)
python  27678 root   11r   CHR               1,9      0t0 30690329 /dev/urandom
python  27678 root   12u  IPv4          30719611      0t0      TCP db.sdwan:51404->
↪db.sdwan:37017 (ESTABLISHED)
python  27678 root   13u  IPv4          30719610      0t0      TCP db.sdwan:47124->
↪db.sdwan:27017 (ESTABLISHED)
python  27678 root   14r  FIFO               0,8      0t0 38483750 pipe
```

6.6 参考

如何区分进程和线程 ps -eLf：https://www.cnblogs.com/shengulong/p/11498437.html

如何查询一个进程下面的线程数（进程和线程区别）：https://www.cnblogs.com/kevingrace/p/5252919.html

用 strace 查找进程卡死原因：https://blog.csdn.net/peng314899581/article/details/79064616

CHAPTER 7

Indices and tables

- genindex
- modindex
- search