大话后端开发的奇淫技巧大集合

1. **模块化设计**

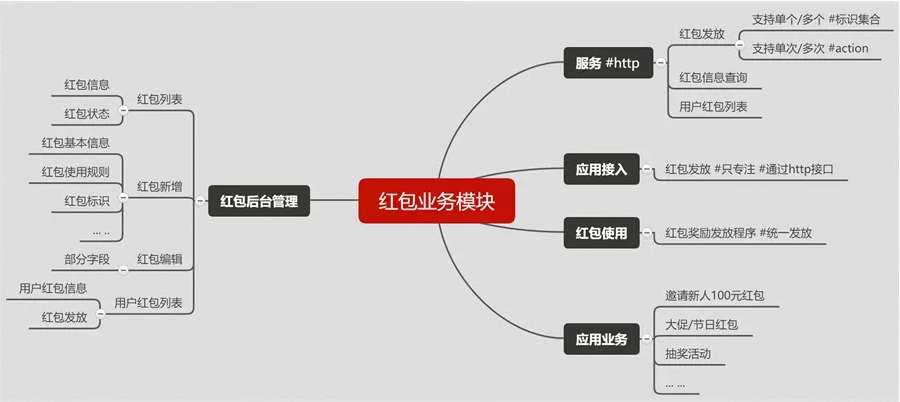
根据业务场景，将业务抽离成独立模块，对外通过接口提供服务，减少系统复杂度和耦合度，实现可复用，易维护，易拓展

* 1. **项目中实践例子**
     1. **Before**

在返还购APP里有个【我的红包】的功能，用户的红包数据来自多个业务，如：邀请新用户注册领取100元红包，大促活动双倍红包，等各种活动红包，多个活动业务都实现了一套不同规则的红包领取和红包奖励发放的机制，导致红包不可管理，不能复用，难维护难拓展。

* + 1. **After**
* 重构红包业务
* 红包可后台管理
* 红包信息管理，可添加，可编辑，可配置红包使用的规则，可管理用户红包
* 红包奖励发放统一处理
* 应用业务的接入只需要专注给用户进行红包发放即可

设计概要如下：



Before VS After ：



产品有时提出的业务需求没有往这方面去考虑，结合场景和未来拓展需要，在需求讨论的时候提出模块化设计方案，并可以协助产品进行设计。

1. **通用服务抽离**

在项目开发中经常会遇到些类似的功能，但是不同的开发人员都各自实现，或者因为不能复用又重新开发一个，导致了类似功能的重复开发，所以我们需要对能够抽离独立服务的功能进行抽离，达到复用的效果，并且可以不断拓展完善，节约了后续开发成本，提高开发效率，易于维护和拓展。

* 1. **项目中实践例子**
     1. **Before**

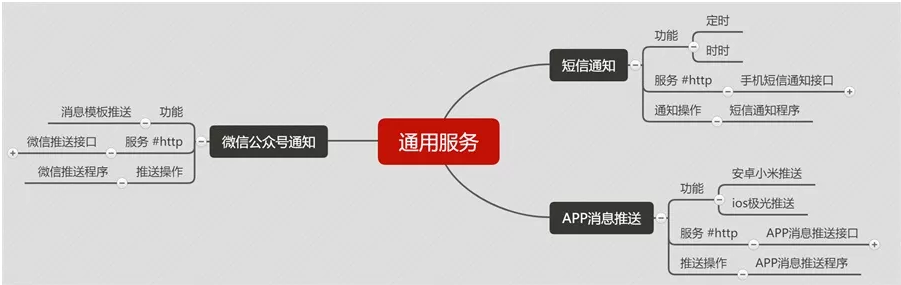
在业务中经常需要对用户进行信息通知，如：短信定时通知，APP消息推送，微信通知等。

开发人员在接到需求中有通知功能的时候没有考虑后续拓展，就接入第三方信息通知平台，然后简单封装个信息通知方法，后续也有类似信息通知需求的时候，另一个开发人员发现当前这个通知方法无法满足自己的需求，然后又自己去了解第三方平台重新封装了通知方法，或者后续需求加了定时通知的功能，开发人员针对业务去实现了个定时通知功能，但是只能自己业务上使用，其他业务无法接入，没有人去做这块功能的抽离，久而久之就演变成功能重复开发，且不易于维护和拓展。

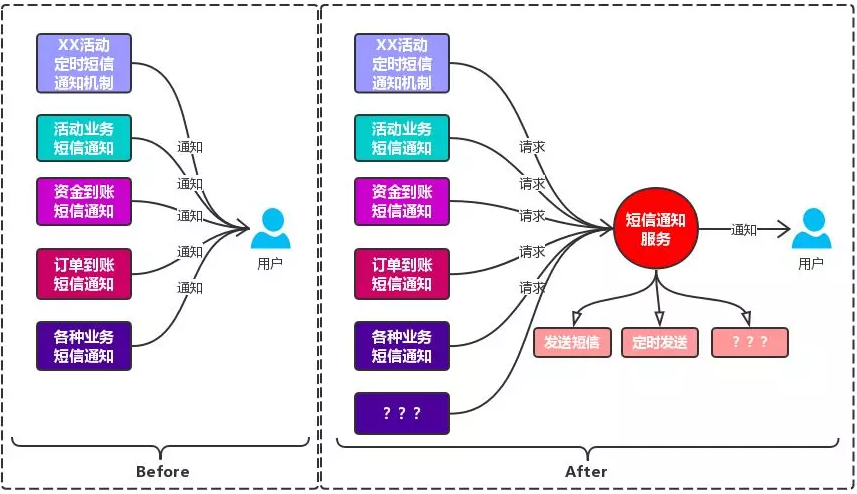
* + 1. **After**

接触到这种可以抽离通用服务需求的时候，就会与产品确认这种需求是否后续会存在类似的需要，然后建议这把块需求抽离成通用服务，方便后续维护和拓展。

设计概要：



Before VS After ：



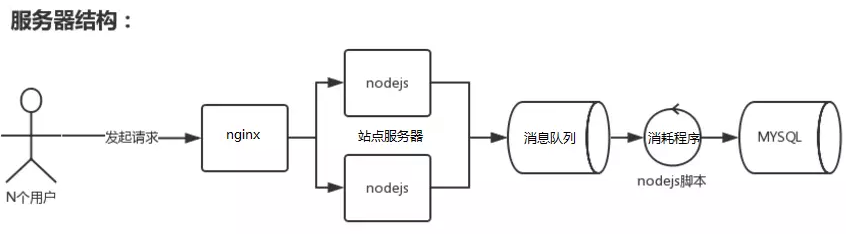
1. **架构独立服务**

项目开发过程中有些需求是与所在项目业务无关，如：收集用户行为习惯，收集商品曝光点击，数据收集提供给BI进行统计报表输出，公用拉新促活业务（柚子街和返还公用），类似这种需求，我们结合应用场景，考虑服务的独立性，以及未来的拓展需要，架构独立项目进行维护，在服务器上独立分布式部署不影响现有主业务服务器资源。

* 1. **项目中实践例子**

架构用户行为跟踪独立服务，在开发前预估了下这个服务的请求量，并会有相对大量的并发请求。

* + 1. **架构方案**
* 项目搭建选择用nodejs来做服务端
* 单进程，基于事件驱动和无阻塞I/O，所以非常适合处理并发请求
* 负载均衡：cluster模块/PM2
* 架构nodejs独立服务
* 提供服务接口给客户端
* 接口不直接DB操作，保证并发下的稳定性
* 数据异步入库
* 通过程序把数据从：消息队列=>mysql
* nodejs+express+redis(list)/mq+mysql
  1. **用户行为跟踪服务的服务架构图**



1. **高并发优化**

高并发除了需要对服务器进行垂直扩展和水平扩展之外，作为后端开发可以通过高并发优化，保证业务在高并发的时候能够稳定的运行，避免业务停滞带来的损失，给用户带来不好的体验。

* 1. **服务端缓存**
     1. **内存数据库**

Redis、memcache

* + 1. **方式**

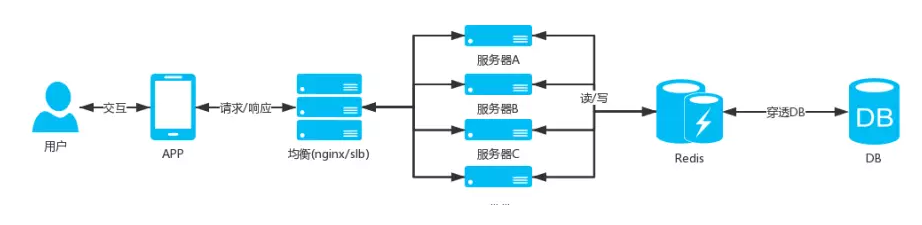
优先缓存->穿透DB问题

只读缓存->更新/失效删除

* + 1. **注意**
* 内存数据库的分配的内存容量有限，合理规划使用，滥用最终会导致内存空间不足
* 缓存数据需要设置过期时间，无效/不使用的数据自动过期
* 压缩数据缓存数据，不使用字段不添加到缓存中
* 根据业务拆分布式部署缓存服务器
  1. **客户端缓存**
     1. **方式**
* 客户端请求数据接口，缓存数据和数据版本号，并且每次请求带上缓存的数据版本号
* 服务端根据上报的数据版本号与数据当前版本号对比
* 版本号一样不返回数据列表，版本号不一样返回最新数据和最新版本号
  + 1. **场景**

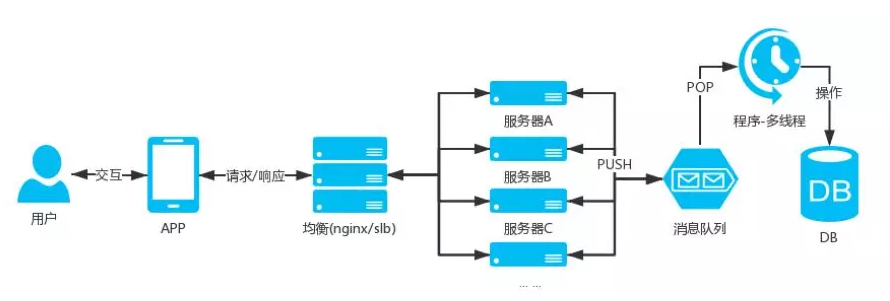
更新频率不高的数据

* 1. **服务端缓存架构图**



1. **异步**
   1. **异步编程**

* 方式：
* 多线程编程
* nodejs异步编程
* 场景：
* 参与活动成功后进行短信通知
* 非主业务逻辑流程需要的操作，允许异步处理其他辅助业务等
  1. **业务异步处理**
* 方式
* 业务接口将客户端上报的数据PUSH到消息队列（MQ中间件），然后就响应结果给用户
* 编写独立程序去订阅消息队列，异步处理业务
* 场景：
* 大促活动整点抢限量红包
* 参与成功后委婉提示：预计X天后进行红包发放
* 并发量比较大的业务，且没有其他更好的优化方案，业务允许异步处理
* 注意：
* 把控队列消耗的进度
* 保证幂等性和数据最终一致性
* 缺陷：
* 牺牲用户体验
  1. **架构图**



【业务异步处理】除了可以在高并发业务中使用，在上面通用服务的设计里也是用这种架构方式。

1. **限流**

在类秒杀的活动中通过限制请求量，可以避免超卖，超领等问题。

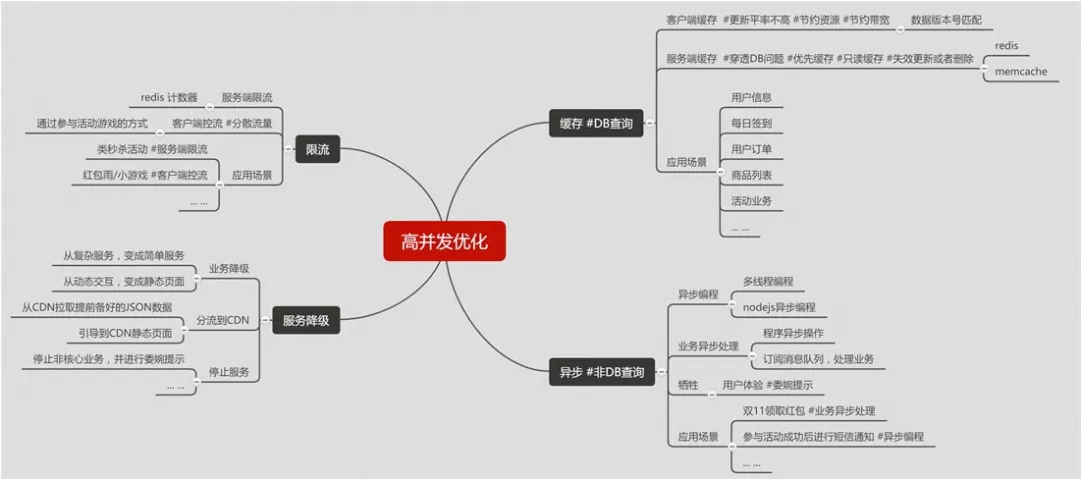
高并发的活动业务，通过前端控流，分散请求，减少并发量。

* 服务端限流
* redis 计数器
* 如：类秒杀活动
* 客户端控流
* 通过参与活动游戏的方式
* 红包雨/小游戏等方式

1. **服务降级**

当服务器资源消耗已经达到一定的级别的时候，为了保证核心业务正常运行，需要丢卒保车，弃车保帅，服务降级是最后的手段，避免服务器宕机导致业务停滞带来的损失，以及给用户带来不好的体验。

* 业务降级
* 从复杂服务，变成简单服务
* 从动态交互，变成静态页面
* 分流到CDN
* 从CDN拉取提前备好的JSON数据
* 引导到CDN静态页面
* 停止服务
* 停止非核心业务，并进行委婉提示
  1. **高并发优化概要图**



1. **防刷/防羊毛党**

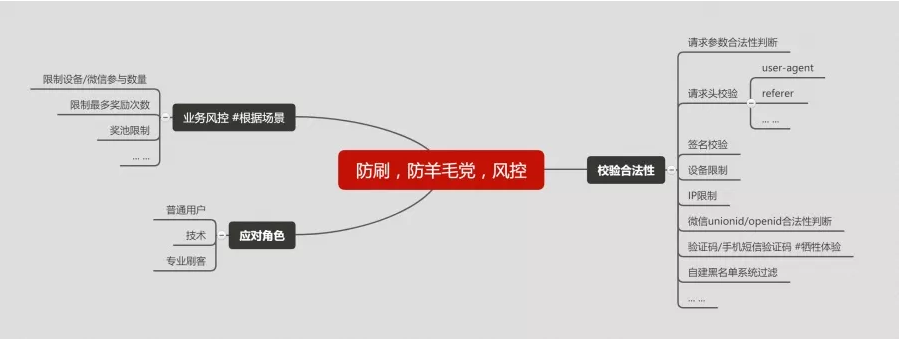
大多数公司的产品设计和程序猿对于推广活动业务的防刷意识不强，在活动业务设计和开发的过程中没有把防刷的功能加入业务中，给那些喜欢刷活动的人创造了很多的空子 等到你发现自己被刷的时候，已经产生了不小的损失，少则几百几千，多则几万。

随着利益的诱惑，现在已经浮现了一个新的职业“刷客”，专业刷互联网活动为生，养了N台手机+N个手机号码+N个微信账号，刷到的奖励金进行提现，刷到活动商品进行低价转手处理，开辟了一条新的灰色产业链。

我们要拿起武器(代码)进行自我的防御，风控，加高门槛，通过校验和限制减少风险发生的各种可能性，减少风险发生时造成的损失。

这里列出常用套路（具体应用结合业务场景）：

* 1. **校验请求合法性**
* 请求参数合法性判断
* 请求头校验
* user-agent
* referer
* … …
* 签名校验
* 对请求参数进行签名
* 设备限制
* IP限制
* 微信unionid/openid合法性判断
* 验证码/手机短信验证码
* 牺牲体验
* 自建黑名单系统过滤
  1. **业务风控**
* 限制设备/微信参与次数
* 限制最多奖励次数
* 奖池限制
* 根据具体业务场景设计… …
  1. **应对角色**
* 普通用户
* 技术用户
* 专业刷客
* 目前还没有很好的限制方式
  1. **防刷/防羊毛党套路概要图**



* 1. **附加**

APP/H5中签名规则应该由客户端童鞋开发，然后拓展API给前端JS调用，在H5发起接口请求的时候调用客户端拓展的签名，这样可以避免前端JS里构造签名规则而被发现破解。

1. **并发问题**
   1. **多操作**
      1. **场景**

当同用户多次触发点击，或者通过模拟并发请求，就会出现多操作的问题，比如：签到功能，一天只能签到一次，可以获得1积分，但是并发的情况下会出现用户可以获得多积分的问题。

* + 1. **剖析**

简化签到逻辑一般是这样的：

查询是否有签到记录 –> 否 –> 添加今日签到记录 –> 累加用户积分 –> 签到成功。

查询是否有签到记录 –> 是 –> 今日已经签到过。

假设这个时候用户A并发两个签到请求，这时会同时进入到 【查询是否有签到记录】，然后同时返回否，就会添加两条的签到记录，并且多累加积分。

* + 1. **解决方案**

最理想简单的方案，只需要在签到记录表添加【签到日期】+【用户ID】的组合唯一索引，当并发的时候只有会一条可以添加成功，其他添加操作会因为唯一约束而失败。

* 1. **库存负数**
     1. **场景**

当多用户并发点击参与活动，如：抽奖活动，这个时候奖品只有一个库存了，理论上只有一个用户可以获得，但是并发的时候往往会出现他们都成功获得奖品，导致奖品多支出，加大了活动成本。

* + 1. **剖析**

有问题的逻辑流程一般是这样的：

中奖 –> 查询奖品库存 –> 有 –> 更新奖品库存 –> 添加中奖纪录 –> 告知中奖

中奖 –> 查询奖品库存 –> 无 –> 告知无中奖

假设抽奖活动，当前奖品A只有最后一个库存，然后用户A、B、C，同时参与活动同时中奖奖品都是A，这个时候查询商品库存是存在1个，就会进行更新库存，添加中奖纪录，然后就同时中奖了

* + 1. **解决方案**

最理想根本就不需要用多做一个库存的SELECT奖品库存操作，只需要UPDATE 奖品库存-1 WHERE 奖品库存>=1，UPDATE成功后就说明是有库存的，然后再做后续操作，并发的时候只会有一个用户UPDATE成功。

* 1. **总结**

在开发业务接口的时候需要把同用户和多用户并发的场景考虑进去，这样就可以避免在并发的时候产生数据异常问题，导致成本多支出。

1. **数据采集技巧**
   1. **普遍方案**

* 获取平台数据接口
* 模拟接口请求
* 数据解析过滤
* 数据构造入库
  1. **使用selenium+Headless自动化测试框架**
* 开发
* 推荐用python开发
* python+selenium+headless
* 控制请求频率，避免被平台限制请求
* 使用代理IP，绕过请求IP限制
* 优点
* 无需模拟接口请求
* 无法攻克数据接口模拟请求(加密签名等)
* 接口版本频繁变化(需要重新调研)
* 平台接口/页面版本变化，可以快速调整
* 只需要调整采集数据所在的HTML元素的位置(class/id)
* 可以用户操作/选中/点击/模拟登陆，等
* 登陆失效后可以模拟登陆
* 可以发送登陆二维码到钉钉进行扫码登录
* 应用场景：
* 竞品数据采集
* 淘宝商品价格和自建商品库后台价格监控
* 淘宝领券金额和自建商品库后台券金额监控
* … …
  1. **反反爬虫**

在做数据采集的过程中，有些平台会对重要数据的请求设置反爬虫策略，避免数据被竞品挖掘和利用，以及消耗大量资源拖垮服务器， 反爬虫和反反爬虫是技术之间的较量，这场没有硝烟的战争永不停息。（程序员何必为难程序员）。

* 反爬虫可以分为以下两种
* 服务端限制
* 服务器端行请求限制，防止爬虫进行数据请求
* 前端限制
* 前端通过CSS和HTML标签进行干扰混淆关键数据，防止爬虫轻易获取数据
* 破解服务端限制：
* 模拟设置请求头
* Referer
* User-Agent
* Authorization
* …..
* 破解签名
* 签名规则
* 在JS中找到签名规则
* 控制请求平率
* 调整请求时间，延迟请求
* 代理IP
* 切换请求的代理IP，自建/第三方
* 登录限制
* 带上登录成功后的Cookie/Authorization
* 验证码限制
* 识图，基于库/第三方
* 投毒破解
* 为了防止被投毒，需要对数据进行抽样校验
* 破解前端限制：
* font-face，自定义字体干扰
* 找到ttf字体文件地址，然后下载下来，使用font解析模块包对ttf文件进行解析，与文字编码进行映射出中文
* 伪元素隐藏式
* 在CSS里找到xxxx::before {content: “中文”;}对应的中文
* backgroud-image移量
* 通过背景图片的position位置偏移量和图片中的内容进行映射
* html标签干扰
* 过滤掉干扰混淆的HTML标签，或者只读取有效数据的HTML标签的内容

1. **总结**

作为后端开发者，不仅是完成需求功能开发，要结合业务场景进行合理设计，架构未来，对核心业务进行压测优化，以保证业务在并发下能够正常运行，同时要考虑安全问题以及防刷，防羊毛党，在编码上避免坏代码味道，面相抽象开发，适当使用设计模式，避免技术债。

* 1. **开发应该铭记于心的精句**
* 技术的存在价值，是让技术推动业务增长，实现公司盈利增长
* 没有最好的架构只有最适合的架构
* 开发语言只是工具，在适合的场景中使用适合的工具
* 抽象思维是从具体存在的各不相同的问题当中洞察问题的本质，理解产品需求的深层次模型，治本而不是治标
* 知识很重要，她虽然不能直接给你财富，但是可以给你很多机会，活到老学到老

1. **来源**

http://url.cn/5bl43lf