朱晔的互联网架构实践心得 S1E6：给飞机换引擎和安全意识十原则

【下载本文 PDF 进行阅读】

本文有两个部分，先介绍一下给飞机换引擎这个事情我的一些经验，因为篇幅较短然后介绍一下安全意识方面的一些心得。

给飞行中的飞机换引擎

所谓给飞行中的飞机（或飞驰的汽车）换引擎说的是我们需要对一个正在飞速发展的系统进行大幅度的架构改造，比如把 All-in-one 的架构改造成微服务架构，尽可能减少或消除停服的时间。一般而言，我们可以这么来考虑方案，从重构的彻底性来说，分为这么几种：

- 彻底重新做，直接从前到后抛弃老系统
- 大规模重构，保留对用户的这层皮，后面从服务到数据全部替换
- 小规模重构，保留对用户的这层皮以及数据结构，逐一替换核心逻辑到微服务

在做换引擎方案选择和设计的时候需要考虑到这么几个现实的情况：

- 业务需要发展。意味着会不断有新需求需要开发，如果重构的时间拖的很长的话，我们需要在这段很长的时间内为两套系统同时做新需求，新的系统还要不断开发新需求，会增加更多的时间，如果新系统的开发不够快的话，甚至一直上不了线。我们最好在重构之前对新增业务有一条界限，新系统只是覆盖到这个版本做需求封板，然后和产品商量出一个妥协是否对于之后我们留 1 周作为系统切换期，这段时间不上线新需求。这 1 周分成几个环节，需要 2 天的时间来做系统切换，然后需要 5 天的时间来观察新系统的稳定性，修复新系统的 Bug，随后我们才能认为系统正式切换成功，把时间精力放到新业务的开发上。
- 数据需要迁移。我们是为一个旧飞机在换引擎，无法抛弃飞机上的乘客。如果我们更改数据结构的话，需要对数据进行迁移。我们需要做下面的工作：
    - 准备迁移脚本
    - 准备缓存预热脚本
    - 使用既有的数据来测试这 2 个脚本，然后观察新系统的运行情况
    - 做反向数据迁移的脚本，我们要考虑到切换到新系统后运行不流畅需要整体回滚的情况，这个时候我们需要把数据从新的数据库迁移回老的数据库

  这种方式的迁移是需要有短暂的停机的，两个脚本的执行和验证需要一段时间（数据量大的话导数据费时）。如果希望尽可能减少停机时间的话可以采用两段走的方式，先同步今天前的 99.9%的数据到新数据库，在停机后再同步今天发生的那 0.1%的数据。极端点希望彻底不停机的话，需要让业务同时走两套系统进行双写，这种方案大大增加复杂度，但是可以换来几乎 0 停机的时间，除非是需要 24 小时在线的金融系统，一般不会考虑新老系统双活的方案。

  你可能会觉得对于重构，我们应该考虑不要改底层数据库，这会大大增加复杂性。这还是取决于我们希望多彻底进行重构，很多时候底层的数据库设计不够灵活这是最致命的，如果不切换到新的数据库，即使搞成了微服务架构数据还是揉在一起，只是形态上是微服务，底层还是乱七八糟，这堆业务代码最终还是要进行一次重构。
- 最好没有停机让用户没有感知。没有停机意味着老的系统一直在线上稳定运行，新的系统可以以独立的形态重新部署一套，用户对我们新系统的开发没有感知。新老系统的面子可以长得一模一样，但是底层的调用面目全非。由于两套系统都在线上运行，我们可以在内部对新系统进行充分测试，还可以比较同样的业务流程在数据呈现上是否有差异。可惜现实往往没有想象的这么简单，很多时候我们会依赖各种三方数据，对于金融系统来说往往对接了银行的系统，这不是依赖这么简单了，而是完完全全的依赖。如果我们依赖的第三方会受到我们重构的影响，或是我们就是在切换依赖方的话，是否停机就不是自己可以掌控的了，第三方需要有停机我们就只能停机。这个时

候我们能做的是，让网站进行部分功能的停机而不是整体停机，用户可以以只读形式使用网站的所有功能，但是不能做写入操作，待三方系统切换完成后切换到新系统了才能使用所有功能，这样用户也会有安全感。

- 万一迁移失败怎么办。之前说了我们迁移数据需要考虑回滚方案，这里还会有一些比较恶心的点是如果新老系统有一些内部外部依赖是公用的话，尽量要隔离清楚，让新老系统彻底独立，这样才好回滚，如果迁移后新系统在跑了污染了老系统的数据，这个时候再要回滚会出现回不去的情况，因为老系统无法适应新的数据。内部要进行彻底的隔离是比较简单，对于三方的数据（比如 CDN）我们也要考虑到数据在边缘节点和用户端缓存的情况，在设计方案的时候就要考虑到切换后回退的可能性，尽可能让两套系统使用独立的数据不要产生数据覆盖的情况造成污染。

还是要具体的事情具体来分析，要给飞行中的飞机换引擎，最重要的就是 1、得到产品方面的配合对需求做好锁定；2、在开发过程中对新系统做好充分测试减少上线后修 Bug 的工作量；3、对迁移方案以及回滚方案做好自动化的脚本以及充分的验证。

安全意识十原则

对于安全我不是专家，但是我发现有这么一个问题，那就是做业务开发的同学往往一点安全意识都没有，如果有的公司没有安全方面的部门或专家的话，那么安全问题真的会很严重，外面所谓的一些安全公司的外包渗透服务往往是浅层次的机器做一下扫描和渗透，很少在代码和逻辑层面做深入的分析，安全要做好还是要靠一线程序员和产品经理的点点滴滴的意识。

在这里，我介绍一下我总结出来的偏产品技术（网络层面还有很多工作需要做，这里没有涉及）方面的安全意识十原则（黑客白帽子肯定会有自己的一些成体系的方法论，我这里更多的是根据之前踩过的坑自己总结的一些经验）：

1. 安全问题是木桶效应。整个系统的安全程度取决于木桶最短的那块板。很多时候我们会召集安全专家和架构师和主站主流程主域名的系统进行安全分析和渗透测试，而黑客知道这点也往往喜欢找边缘化的子站点或非核心逻辑进行攻破，这些模块或站点往往是由初级程序员打造，有的甚至还不是主站统一的技术架构，总体上会防备薄弱。黑客能够攻破任意站点进去到内网，就有种种可能。针对这点，我们需要做的是对于安全的排查，需要全面覆盖，除非子站在部署上用户体系上彻底隔离。
2. 开发层面：不要信任客户端的任何东西。对于 HTTP 协议，不管是头里面的东西（来源、客户端类型、Cookie）还是正文里面的东西，任何数据都是可以伪造的。我们往往会觉得

Get 的东西暴露在浏览器地址上，里面的参数不安全，Post 过来的数据因为不暴露就安全，然后会信任 Cookie 中的数据做一些权限控制，会用头里面的一些数据做一些安全性控制。这些不是说不能做，而是要从根子里面有这个意识所有客户端的东西可以用但不能不经过判断直接相信。有一个容易犯的错误是，在设计 Controller 的时候我们可能会在参数里让客户端传过来 UserID、Price 等信息。这里的问题在于，登录后的 UserID 应该是保存在服务端的，客户是谁不是客户自己说了算的，应该是我们根据 SessionID 在服务端获得的，我们需要全面排查代码，不允许在 Controller 的方法里存在类似于用户 ID 这样的字段。对于 Price 也是一样的道理，如果这是一个购物的过程，那么订单的价格一定是在服务端计算的，我们只能依靠客户端传过来的 ItemID 来计算价格，不能相信和直接使用客户端传过来的价格入库（仅仅作为呈现是可以的，客户端来的还是呈现在客户端，但是不能用于计算）。其实更麻烦的是，很多时候我们会用框架生成的 Controller 的 CRUD 接口的代码，框架会在参数内直接放上完整的 Entity，然后代码里会使用这个 Entity 直接对接数据访问层入库，因为客户端和服务端用 JSON 在通讯，虽然我们看到客户端传给服务端的只是 ItemID，但是你再传一个 Price 确实也是可以生效进入 Update 语句入库的（Entity 某些字段为空那么就不会进入 Update 语句，不为空就会更新）。下面会再提到这个问题，很多时候用户看不到的不等于程序逻辑上不可行，服务端的接口层面往往会有比看到的更多的权限（甚至可以由贯穿数据库的完整 CRUD Restful API）。

3. 开发层面：数据就是数据代码就是代码。不管是 SQL 注入也好 XSS 也好都是这个问题，把数据和代码混在了一起。对于客户端过来的任何信息应该都只是数据，应该很少会让客户端来告诉服务端执行的代码。所以这个事情我们要从两方面来防范，第一客户端过来的数据需要让它当成数据来处理，不管是 Encoding 一下也好，SQL 参数化（Mybatis 的$和#问题）也好都是这个方面的措施（从前到后一路数据都以数据的身份在程序中流转），第二从数据库里出来的东西也只能是数据不能让它变为 HTML 或 JS 代码，也需要 Encoding 一次再在客户端上呈现。SQL 注入的防范很多人喜欢用敏感字符替换的方式来做，这种做法其实是会有遗留的，我们不应该去考虑排除数据中有代码的可能性，而是应该从根源上去让数据只可能成为数据。

4. 开发层面：用户看不到不等于黑客看不到。第一个方面想说的是，随着前后端的分离，现在很多请求都是 AJAX 请求，AJAX 请求会有几方面的安全疏漏：

    - 逻辑分散而明确。对于服务端渲染，我们会把各种逻辑整合在一起，用户看到的是一个完整的渲染后的页面不清楚里面有多少逻辑，对于 AJAX 请求，我们会以良好的方法命名来命名各种 API。这个时候会给黑客可乘之机，会更容易分析理解程序运行的流程，毕竟在寻找突破之前需要先理解程序的流程。

- 容易觉得 AJAX 请求是前端程序发起的而忽略权限问题。没错，AJAX 的发起人也是我们的代码，但是 AJAX 请求也是 HTTP 请求谁都可以发起。如果我们对 AJAX 请求的参数设计有所松懈，犯了之前说的两个错的话那就很危险了。特别是前后端都是一个人来写的话，程序员在实现代码逻辑的时候往往只会考虑数据的传输简单通畅，后端给前端，前端再给后端放松安全意识。

第二方面说的是，后端往往会返回更多的数据给前端，前端选择需要的数据进行呈现，有的时候甚至直接返回的是代码生成器生成的 DbEntity（在三层架构中，DbEntity 对应表结构，ServiceEntity 是充血的领域实体，ResponseEntity 是面向呈现的实体，三者不应该是一套）。这里的问题在于：

- 服务端返回了过多的数据，这是比较危险的事情，因为有一些内部的字段会返回出去。
- 服务端返回的数据列对应了数据库实际的列，相当于暴露了表结构，对于以后各种 API 的尝试和注入极端危险。
- 一些敏感的数据也直接返回给客户端了，虽然客户端不会展现出来，但是对于黑客来说根本不在于用户看得到看不到这些数据。

这就要求我们在做设计的时候尽可能仔细审视 AJAX 的接口的权限、数据开放性和脱敏等问题。一些框架提供的脚手架生成工具以及 Restful API 自动生成工具，即使要用也要做好权限设置。

5. 开发层面：最小化接口权限设计复用性矛盾。在做设计的时候我们会考虑到复用性的问题让方法尽可能通用，但是对于对外的接口我们要尽量收缩功能。举一个之前看到的例子，我们需要验证游戏的密保卡，需要用户告知三个坐标的密保数字，比如 A1B2C3 三个坐标，每一个数字是 0 到 99，三个数字同时猜对的可能性是百万分之一，但是接口的设计居然是直接让用户传三个坐标，这里有两个严重的设计错误：
    - 第一，为什么是允许客户端传过来三个密保的位置，合理的做法是服务端告知客户端此次校验的 ID，然后客户端传过来 ID，服务端在 Session 或缓存中去获取坐标位置。
    - 第二，接口允许传三个坐标也算了，还允许是相同的坐标，我们完全可以改造接口传 A1A1A1 然后依次暴力破解 0 到 99，马上就可以得出 A1 坐标的值，几秒的时间就可以把整个密保卡完全暴破出来。

在设计服务端接口的时候，我们最好针对某个功能设计最小的接口，而不是开放的通用的接口，对外的接口设计安全性需要大于重用性。

6. 开发层面：一开始就要考虑安全，放出去了就没后悔药。比较无奈的是，很多项目我们处于赶进度，一开始第一版的时候并没有对接口做加密和签名验证。如果做的是一个 APP，那么我们要考虑到 APP 用户不升级的情况，即使 v2 的版本的数据都是加密的，参数都是验签的，但是我们还不能下架 v1 版本（强更会损失多少用户难以估计）。这个时候就比较无奈了，明知道 v1 版本的安全性有着很大的风险却不能升级。不仅仅是接口的安全性，安卓客户端的代码也要考虑到反编译的可能性需要混淆。之前遇到过有一个 App 客户端里网络层直接使用了服务端的接口定义，导致客户端代码里可以对服务端所有接口一清二楚，还不乏一些不能对外使用的内部接口以及已经淘汰的老接口，加上接口的调用又没有签名数据传输又不加密，拿着这份网络协议什么都可以干。

7. 产品层面：做好防刷和暴力破解控制。显性的功能固然重要，但是产品经理也需要在隐性功能和风控策略上下一些功夫，产品经理没有这方面意识的话开发往往更不太会做深入考虑。包括：

    - 公开出去的短信验证码服务防刷控制（防止被刷子利用做短信轰炸），在用户体验和防刷上做平衡，比如到达一定的频次后出验证码，服务端对客户端的一些头做校验（刷子一般不知道这样的逻辑）等等。不仅仅是短信验证码，公开出去的不需要授权就可以用的服务都需要防刷。
    - 登录是用户从匿名进入授权的重要环节，对这一环节做一些策略。比如异地登录提醒、错误登录后出验证码、太多次错误后禁用等等。
    - 涉及到积分、兑换、返现、抽奖这种和钱打交道的业务，要多考虑是否会有刷的可能，不仅仅是程序逻辑方面的漏洞，唯一性的判断，还要考虑积分流通起来后是否会有撸羊毛小换大的可能。有利益的地方就有羊毛，如果一个业务都是羊毛在参与的话那么账面数据可能挺好看，但实际上几乎带来不了忠实的有留存的用户。

重要的业务需要有风控方面的产品经理和业务产品经理一起来参与，共同讨论流程，防羊毛，防黑客， 防刷子。

8. 产品层面：注意产品逻辑一体性。很多业务流程是授权-执行这样的两步，但有些时候这两步在产品设计的时候并不一定是同一个产品经理在设计，会出现授权和执行不一致的情况。见过一个修改绑定邮箱的产品逻辑，用户在页面 X 先进行短信验证码验证，然后就可以跳转到 Y 页面进行修改绑定邮箱，X 页面是通用的短信验证页面，Y 是新的修改绑定邮箱的需求，这里出现一个逻辑漏洞，就是在 X 页面验证完成后进入 Y 页面，如果这个时候利用另外一个 TAB 退登登录切换用户后，在 Y 页面还是可以修改绑定邮箱的，这个时候程序从 Session 中读取的到用户并不是之前那个通过短信验证的用户，而是后面登录的新用

户，相当于我们就可以为任意用户修改绑定邮箱了。对于多步走的逻辑，我们一定要作为一个整体来思考。黑客在寻找突破的时候特别喜欢寻找 ABC 几步走的逻辑，然后尝试最后一步 C 是否可以单独来做寻找逻辑方面的漏洞。

9. 运维层面：做好异常数据监控报警。在运营层面，我们需要对方方面面的数据有报表或监控，对于异常的数据徒增及时进行调查，业务量的增加如果不伴随活动或推广的话不一定是好事情。在运维层面，我们同样也需要对网络磁盘的使用徒增以及服务的调用量上升查明原因，看看这是业务导致的还是可能是安全性问题。对于各种三方服务（比如短信通道、CDN、文件存储等）的使用，最好也有日报级别的监控，以免看到月度的账单后再发现被刷的问题哭都来不及。

10. 运维层面：对内的数据和权限问题。我们说内部人是最难防的，内部系统繁杂，授权和审计做的可能不那么完善，内部人员如果有心的话获取一些数据做一些坏事造成的影响会比较大，我们需要从几方面来做一些措施：

    - 线上的配置尽量加密，配置信息和代码分离
    - 敏感数据（用户信息）在数据库中加密保存
    - 线上数据访问需要使用类似于 phpmyadmin 的 Web 网站，不允许使用客户端工具，这样可以做比较健全的授权，防止数据导出
    - 内部系统涉及到用户信息部分脱敏显示，显示完整的数据需要上级授权
    - 所有内部系统的登录和使用需要有完整的日志，对日志进行分析和定期的审计