互联网金融产品移动前端发展简史

标题虽然是互联网金融前端，当然这里还是拿陆金所为基础原型来讲述前端的历史。

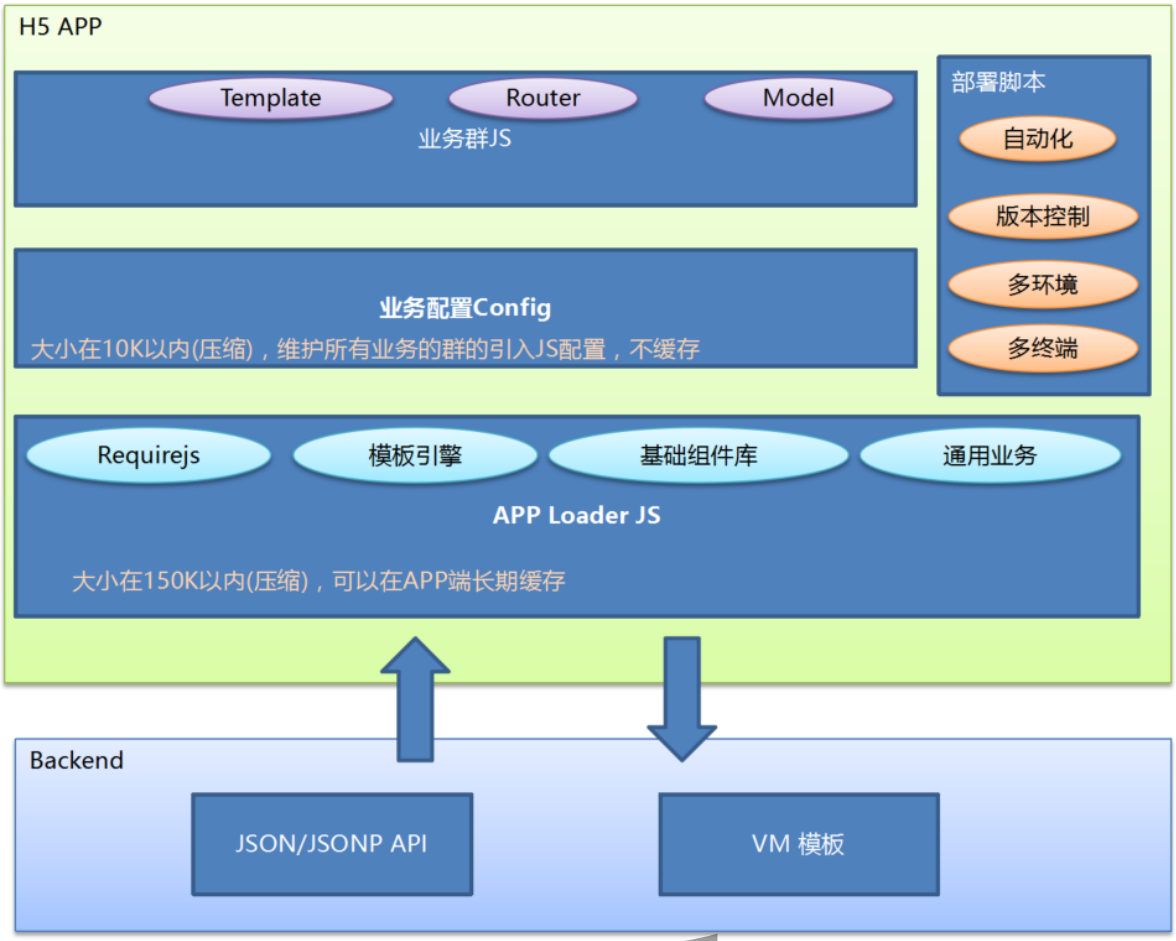
早期的陆金所前端

2015年之前，陆金所的业务复杂度和开发人员规模都不算大，当时的业务主要还集中在PC端，移动业务相对算是残缺的，几乎没有业务是mobile first，所以对移动端的要求也并不是很高，移动端包括陆金所APP的主业务端和和合作伙伴相关的M站点，当时的APP端业务和M站业务共性也并不强，所以采取了两种截然不同的开发技术栈。

Hybrid早期框架

陆金所APP端的早期技术栈，主要是使用underscore.js的模板引擎，使用其强大的函数式能力（可惜当时FP还没有现在这么火），使用AMD异步加载，通过gulp单向打包到hybrid，必须随APP一起发布，并具有一定的静态ZIP包增量更新能力，选择器使用zepto，入门简单，开发成本低，非常适合当时的业务场景。

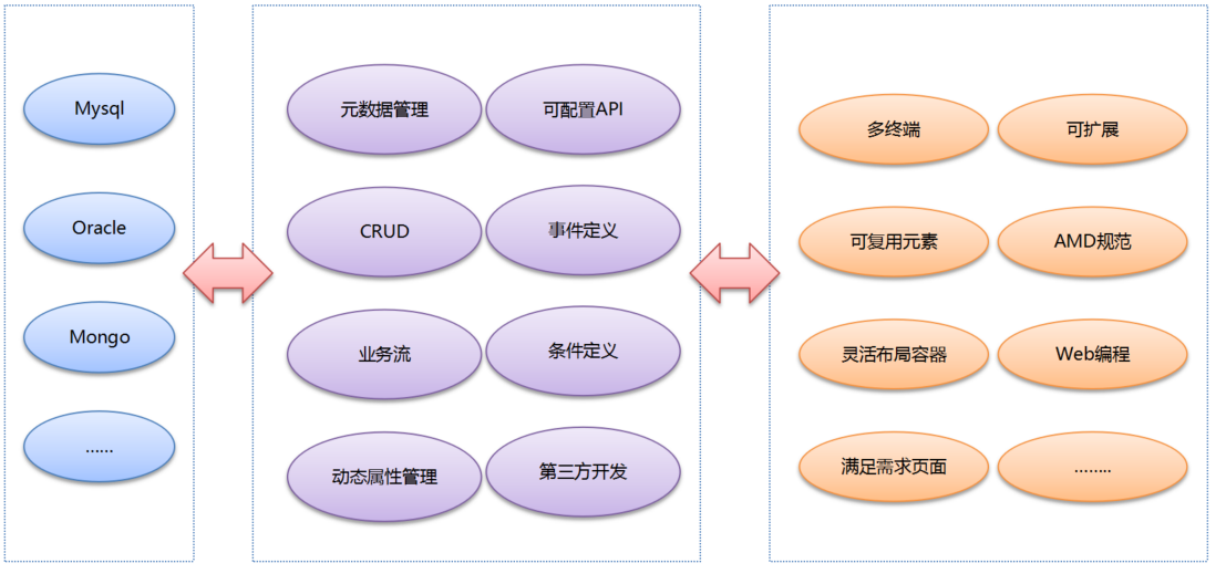
此技术栈最大的弱点是debug阶段无法借助hotload机制，每一次改动都需要一个漫长的webview重新刷新加载过程，在复用性上还没有设计组件化机制，几乎每一个业务实现一套前端页面，没有模块拆分，某些页面的业务代码达到数千行，维护性差，没有抽象可复用部分，但开发效率和运行效率已经能很好的满足当时的需求。



M站的早期框架

陆金所M站的早期技术栈，主要是通过自己研发的NVWA框架，NVWA 框架的核心思想是把开发变成配置，后端接口通过元数据，事件，条件，动态属性配置出满足业务要求的API，前端通过ECP(元素，容器，页面)能够通过拖拽组装出满足产品需求的UI，整个NVWA框架具有很好扩展性，并且提供给第三方系统进行集成。NVWA的Server端可以支持元数据管理，自定义接口、事件、条件和事件流。Client端可以支持开放编程，支持Element、Container、Page概念。不细讲会比较抽象，大家可以看看易企秀和麦客CRM也就部分理解其中的概念了。基于backbonejs和requirejs。

缺点是通用代码抽象不完善，无法支持BU拆分，打包完最大的文件有时候达到3M左右、紧耦合、复用性不高、类似需求重复开发，仅仅支持m站，无法移植到Hybrid。



新的业务要求

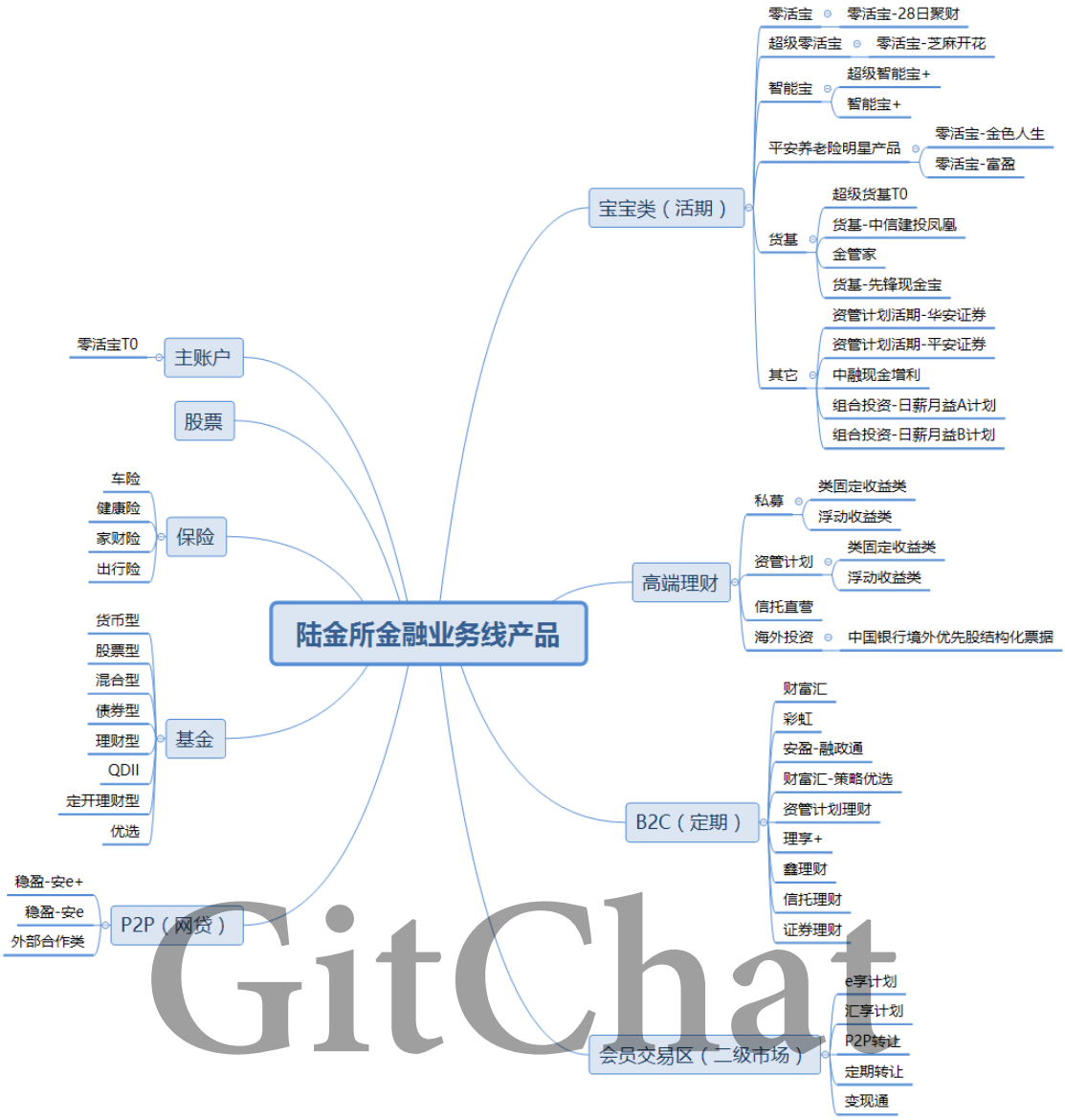
随着陆金所的业务爆发式增长，特别是2015-2016。

- 用户数从几十万增长到上千万。
- 交易量从以万为单位发展到以万亿单位。
- 需求数量（平均值）从5个/月发展到近100个需求/月。
- Native发版频次从1个/季到最高一周一版，另外还有大量Reg常规、Online Quick、紧急EMP、线上热修复hotfix等发版。
- 移动占比也从很小的占比到当前的占比90%以上。

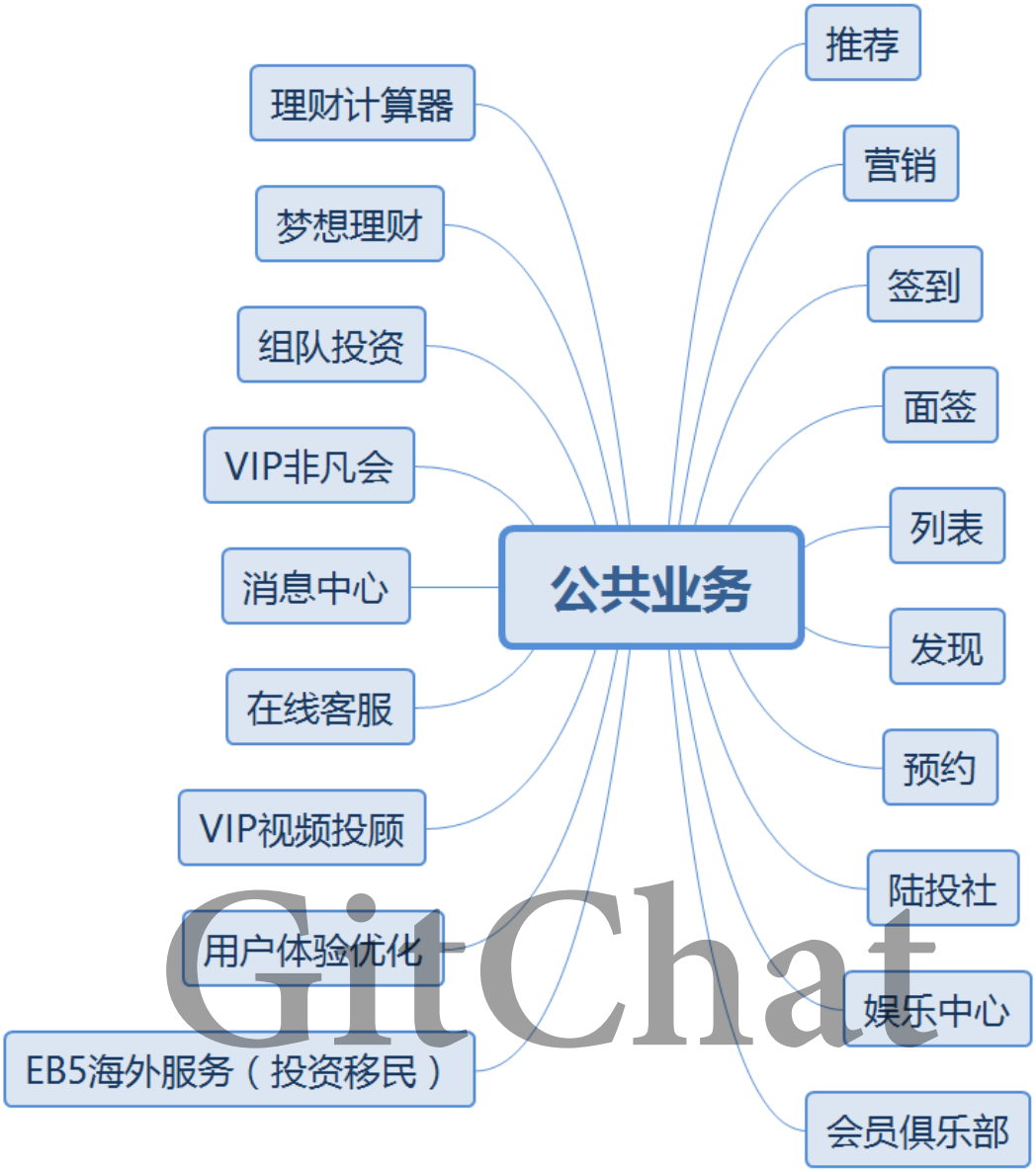
而开发测试人员的增加却远远跟不上业务的增长。

过去的两套分离式开发模式和架构已经完全无法满足业务的需要。陆金所虽然以P2P起家，但是后来的陆金所已经远不是简单的P2P平台，而已是复杂的综合性理财平台。大家可以通过这些图来了解后来庞大的业务体系。

陆金所业务线产品目录：



陆金所公共业务分布：



账户功能模块：

新老更迭

业务挑战

业务爆发式增长，随着越来越多的需求需要同时在多端上线，但是同一个需求却需要使用两个不同的技术栈开发多次，开发效率低，资源大面积浪费，很难保证业务的及时上线。产品经理和业务方对开发效率的吐槽日益增多。

组织挑战

过去由于业务和开发人员的量级都非常小，所以组织结构主要是按照功能拆分，前端团队和后端团队分属于不同的部门，随着开发团队规模的成倍增长，沟通成本和管理成本不断增加。BU化成为必然的趋势，发展到这个阶段，BU化确实可以部分有效提高沟通效率，提升团队协作能力。如何避免最初架构单点系统设计问题，并良好支持BU化后代码的隔离与独立，也是需要面对的。

发版挑战

过去是单点系统，所以也属于单点发布，如果一次发版包含多个需求，只要其中有一个需求出现问题，就需要全部回滚，一个业务的需求会影响其它业务。需要架构上做拆分，做到隔离发布。

开发挑战

开发需要同时学习两个技术体系，来回切换伤身伤脑，技术体系也较老旧，在开发人员配比不足的情况下，常常一个需求需要针对M站和Hybrid开发两次。如何避免这种两套不同架构体系导致的学习成本高，开发效率低的问题。

规范挑战

整个大陆金所的技术体系正在如火如荼的做一次全部改造。看看陆金所做了哪些技术体系改造：

- 基础架构上两地三中心、异地灾备/多活、同城双活、网络区域、专线。
- 系统层面上确立系统层次及依赖原则。
- 运维平台确立域名规划、泳道部署、运维监控能力。
- 数据库按域划分、自动化运维、秒级监控、数据总线。
- 平台中间件将会提供配置中心、消息、JOB、缓存、SOA、DAL、CAT、NAS/TFS、系统监控等。
- 平台线打造统一的安全、用户、产品、交易、支付、账户、资产、我的账户、合同、营销、推荐、广告等。
- 持续交付发力制定需求管理模式、CodeReview流程、UT框架、自动化测试框架、自动部署/发布工具、蓝绿发布。
- 规范落实文档管理、代码管理、域名管理、域/应用管理、代码规范等。

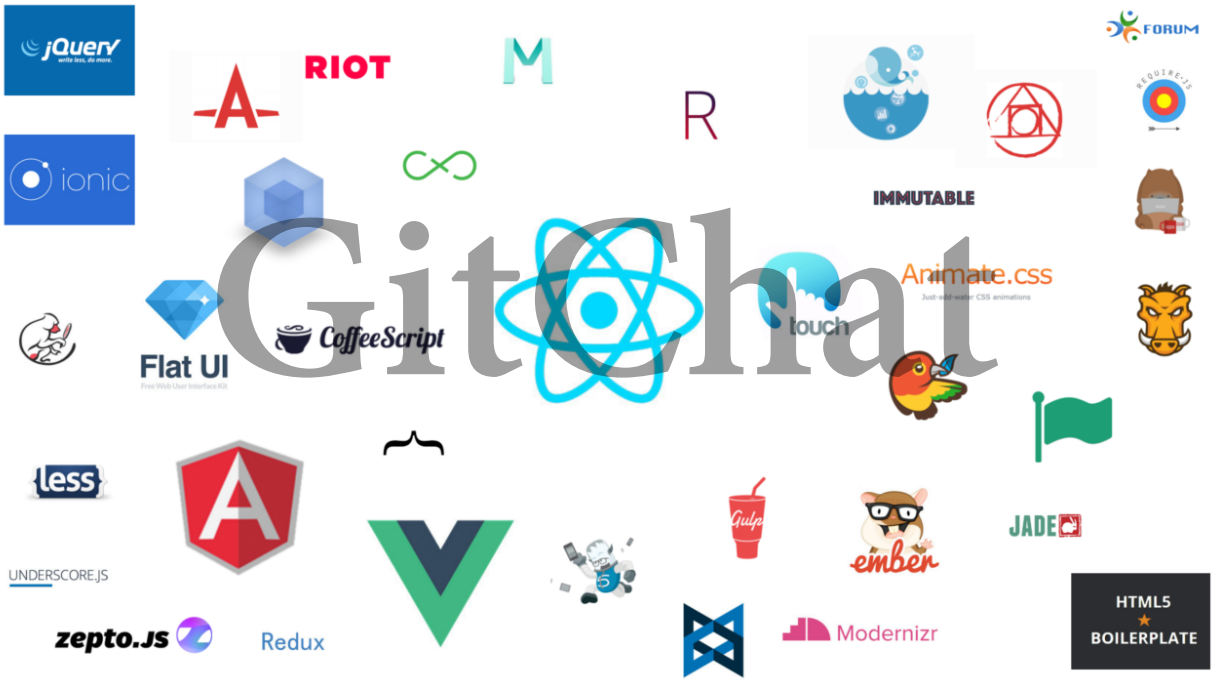
如何解决前端和其它前后左右系统的关系，让前端具有更友好对接其它系统的能力，并接受其它系统的服务，反向来说也是在服务于其它系统。

技术体系选择

在面临如此多挑战的情况下，老的技术体系和架构已经无法满足需求，如何借鉴老系统的优势和经验，并吸取业界的最新解决方案，互相取长补短，前端的技术栈寻求统一和融合，已经迫在眉睫。

正当react、angular、vue等技术迅猛发展，热火朝天之时，当时我们也第一时间对相关技术做了技术预研，并做了原型试点。

回顾前端圈这些年的发展，确实处于百花齐放的状态，也可以说是非常浮躁的状态，大量的框架和库，不间断的革新和迭代，这是一种充满痛苦的幸福，看看这张图，就可以理解前端工程师做一个选择有多难。



选什么架构？

Architecture

为业务服务，制定的合理且可以落地的技术规范准则

新的选择和新的架构设计，除了要解决老系统的问题，我们还需要做到安全、解耦、可监控、可度量、可以快速发版、组件化、高性能、模块化、可扩展、热修复、一致性、隔离发版、BU化支持……

简单对比一些框架：


```
Ember: {{# each}}
Angular 1: ng-repeat
Angular 2: ngFor
Knockout: data-bind="foreach"
React: 直接用 JS 就好啦 :)
Angular 2: 566k (766k with RxJS)
Ember: 435k
Angular 1: 143k
React + Redux: 139k
```

可以看到React作为lib，以 JavaScript 为中心，简约，我们无需学习额外的标签，非常纯粹的javascript和html5。而Angular是框架，以HTML为中心，提供了非常全面的功能。偏见的说，React学习成本更低，会js就搞定一切。在人手缺乏的情况下，我其实比较反感再投入成本去学习新的语法糖和标签，另外强调组件化和关注数据流也是我们非常喜欢的特质，加上virtual DOM让人耳目一新，Unix Philosophy曾推崇KISS（ keep it simple, stupid ），我们认为react可能满足我们要的simple和stupid，或许相对来说是我们的最佳选择。

任何新技术的尝试，一个团队一定会遇到激进派和保守派，在创新阶段，一定要鼓励激进派去试错去冒险，也要不断的听取保守派的建议和监督。在要求业务稳定性的阶段一定要让保守派来主导发展，并参考激进派的建议。

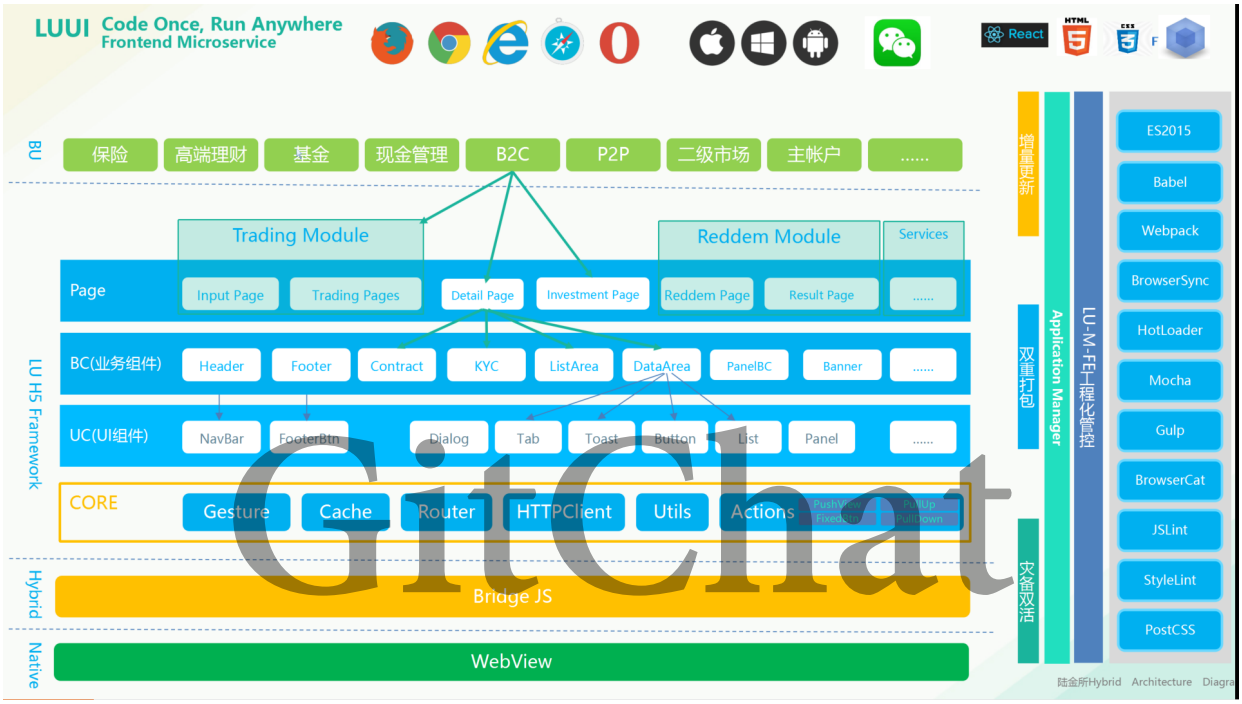
后来在整个团队的努力下，经历了一系列的试错和选型设计后，整个技术栈切换到了react、ES2016、webpack、BrowserSync、Babel、Postcss…的技术体系。

新架构设计

使用选型后的技术体系，如何基于这样的技术选型设计一套能支撑3年（前端的平均寿命周期可能比这个还要低）的框架呢？首先这个框架必须进行合理的粒度拆解，必须能轻松的支持纵向功能级分层，横向按照BU分业务线，我们也叫横向分域。还要考虑前端与其他系统的关联关系。新系统还必须具有严格的编码规范和UT覆盖率等等……

下图就是为了解决以上问题所设计的陆金所移动前端架构总览。

landscape：



这套架构的特征如下：

- 为了提高复用性，按照UI组件、BC组件、页面、模块、前端服务按照不同粒度层次进行组合。
- 提供完备的Bridge层，解决Native和HTML5的双向通讯（这个有机会可以单独主题讲解）。
- 核心系统提供工具、手势、cache控制、HTTP请求控制、页面基类等，封装最核心和最通用的代码。
- 可以支持BU线的独立开发，与独立发版能力。
- 基础LUUI组件库和LUBASE核心库可以支持版本化，并支持业务线灵活依赖。
- 规范的文档体系和code style。
- 具有一定的Run Anywhere能力，支持跨端运行。
- 打包接入持续集成系统，并提供完整的UT测试。

- 提供增量更新机制，基于增量提供offline&online的双活能力。
- 大前端在web层之上架设gateway，解决大前端web层的统一接口接入，适配。统一处理安全校验、流控、熔断、权限控制。
- 底层框架支持性能埋点和错误诊断能力。
- 打包结果同时支持online化war包部署、增量zip、hybrid随版预发。

新架构特性演化

曾经的前端开发方式仅仅停留在页面级，由一部分后端开发同事兼任一定的前端开发，而现在的独立前端已经具有工程级开发协作方式，已经融合了很多后端的思想，比如MVC、MVVM、存储、多线程...，不仅仅在技术模式上革新，还是一个完备的工程化体系，包括团队协同方式、性能优化、流程规范、可视化开发、安全渗透、工具选型、构建方式、自动化测试、持续集成、性能业务服务错误的监控等多个领域。

软件工程是一门研究用工程化方法构建和维护有效的、实用的和高质量的软件的学科。而前端工程已经作为一个探索中的新子集。步入了软件工程的领域。我们认为一个好的前端工程体系应该具备如下功能：

- 开发流程及开发规范，包括代码规范、模块化规范、组件化规范等。通过ESLint针对代码强制按照规范做检测
- 符合业务的UI和BC组件库。
- 合理的分支管理策略，约束commit msg行文规范，方便在提交历史中清楚的区分本次提交是架构需求还是业务需求，如果是业务需求。
- 核心功能和组件库的UT覆盖。
- 发布 / 部署系统，一套自动化及高度适应性的项目。
- 完备的错误与性能监控系统。
- 性能优化标准，剔除冗余的接口请求及资源请求、离线更新机制提高缓存命中率、common代码的有效抽离等，速度就是体验。
- 灰度发布、清晰的发布依赖关系。
- 支持版本化。
- 脚手架 lu-cli。
- 前端安全渗透能力。

那我们在追求以上标准的同时，重点阐述如下特性。

粒度化（组件化、模块化）

一个大的系统，无论从功能层级角度、还是从业务角度、都应该能灵活的进行纵横拆解。一个系统应该可以分为独立的系统、系统中的标准流程、流程中的模块、模块中的组件、组件中的元素…，这些不同大小粒度代表了不同层级的复用性。而如何拆解才是正确的选择呢？

先谈组件，组件能有效提升复用性，能隔离一些复杂细节，让业务开发能更关注业务开发。当前的前端组件库已经多如牛毛，估计没有其他领域能像前端组件库如此欣欣向荣了吧。但陆金所为了拥有足够的定制化能力，还是自己创造了一套自定义的前端组件库：**LUUI**。

LUUI组件库分为UI组件和业务组件，UI组件按照UED的视觉标准规范提供了一整套细粒度的纯展示层解决方案。BC（Business Components）业务组件抽象了大多数的通用业务场景，这些组件会内聚部分的前端业务交互逻辑，并负责接收标准化的业务数据。

模块化则是在组件的基础上，封装的更高层级抽象，作为可以轻松组合、分解和更换的单元，模块的功能应该是独立的，对外暴露标准的功能、状态与接口，并隐藏内部逻辑和特性。

如何定义一个组件和模块的规范：

- 页面上的每一个独立的可视/可交互区域都可以视为一个组件。
- 按钮、label、panel等无业务逻辑的可视区域抽象为UI组件。
- 包含业务逻辑的信息区域和交互区域，根据业务场景，抽象为可大可小的业务组件，业务组件可以由多个UI组件组合而成。
- 每个组件都包括独立的packpage.json描述，一个组件具有一个独立工程目录，组件对应的资源（包括css、图片、 ）都在这个目录下维护,这个目录是一个自包含的结构。
- 组件与组件之间可以自由组合；一般情况业务组件会包含UI组件，但是不建议太复杂的业务组件嵌套业务组件。依赖太深会降低独立性。
- 页面作为组件的容器，页面的数据适配器负责提供组件化数据，并负责组合组件为功能完整的界面。
- 多个页面组合为Micro SPA，此SPA就是一个独立的模块。
- 模块也是自包含的，可以独立打包，和其他模块完全独立，不允许嵌套和功能依赖。
- 模块可以和其他模块按照协议组合为一个标准业务流程。

规范化

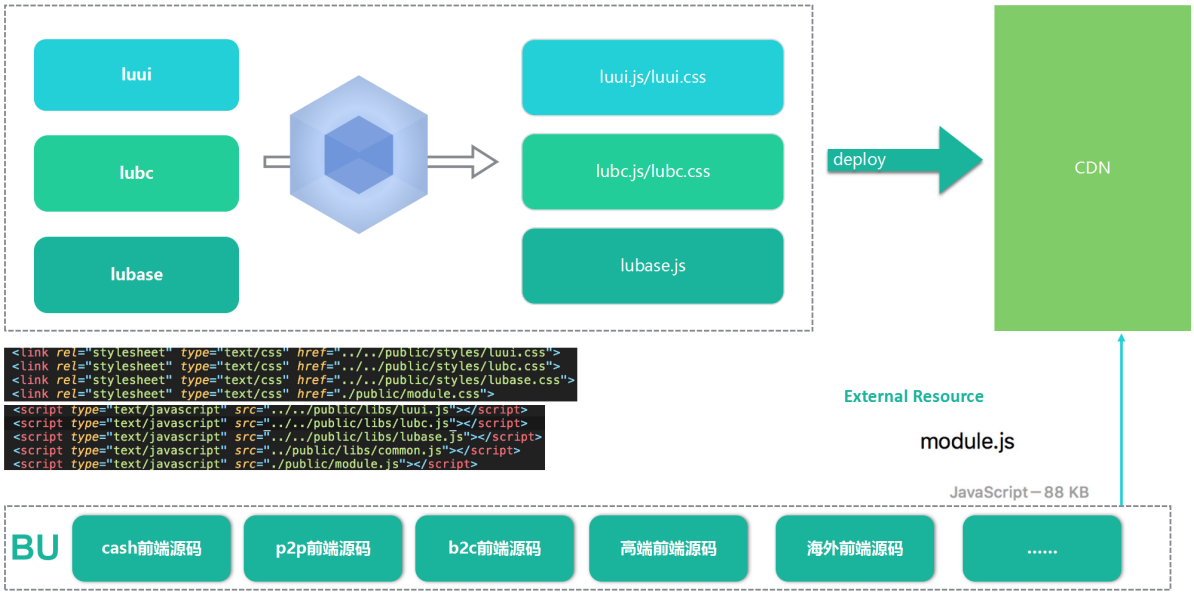
所有的开发技术都需要规范化，规范方能促进团队协作，并提高开发质量。规范一部分除了靠人的监督与执行，另外还是需要靠工具去强制约束，我们主要使用ESLint和StyleLint做强制代码check，git commit hook强制描述规则，文档描述使用markdown，核心功能codereview使用gerrit。

规范的内容大概整理如下：

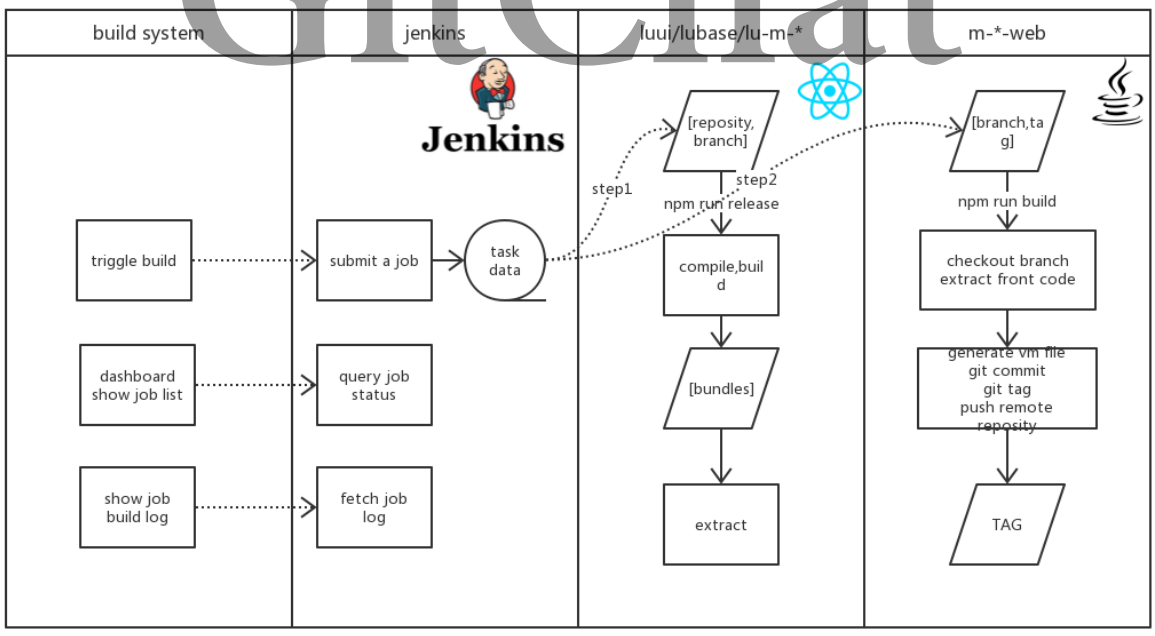
- 目录结构的制定。
- 编码规范。
- 视觉规范。
- 前后端接口规范。
- 文档规范（markdown，自描述）。
- 组件规范。
- Git分支规范（gitflow）。
- Commit描述。
- CodeReview（gerrit）。
- 发版流程规范（灰度、蓝绿）。
- 放量规范。
- 自测规范。

自动化（CI）

自动化构建



在前端源码项目里面采用分布式build方案，自动探测没有修改的代码，并在打包的时候自动过滤没有修改的代码，做到增量更新打包，使用happypack的多线程特性提升性能，由于陆金所内部的网络管控非常严格（金融公司），npm、cnpm就算开墙，有时候还是会不稳定，在网络不稳定的情况下，特别是cnpm（我经常开玩笑说吹牛拍马）会出现提示安装完成，但是却缺包的情况。所以我们采取了复制本地node_modules的形式来保证完整性和性能。打包性能提升了接近10倍。



BU化和系统拆分后，前端源码库和对应的web层项目关联变的非常复杂，如何保证几十个库之间的关联准确性呢？通过mapping配置吗？我们后来发现，最好的办法是前端把打包的结果放在一个共享的地方，web层自动按需索取是最好的方案。

源码是通过webpack打包为前端的build结果，然后在java web层则使用gulp根据各web的需求配置，自动各取所需，自动拉取由webpack生产的结果，并由gulp自动生成java velocity模板，并按照规划存放和引用build后的js。

在HTTP2.0没有大面积运用之前，打包的时候common code的合理抽离可以充分利用浏览器cache和并发请求能力。我们把ui组件库和lubase核心框架库独立作为版本发布，各业务线可以按需依赖。

在这里多提一点，package.json的依赖声明尽可能锁定版本，javascript的很多libs升级很快，很多底层的libs兼容性做的不好，可能会导致意想不到的问题。稳定是我们的职责。

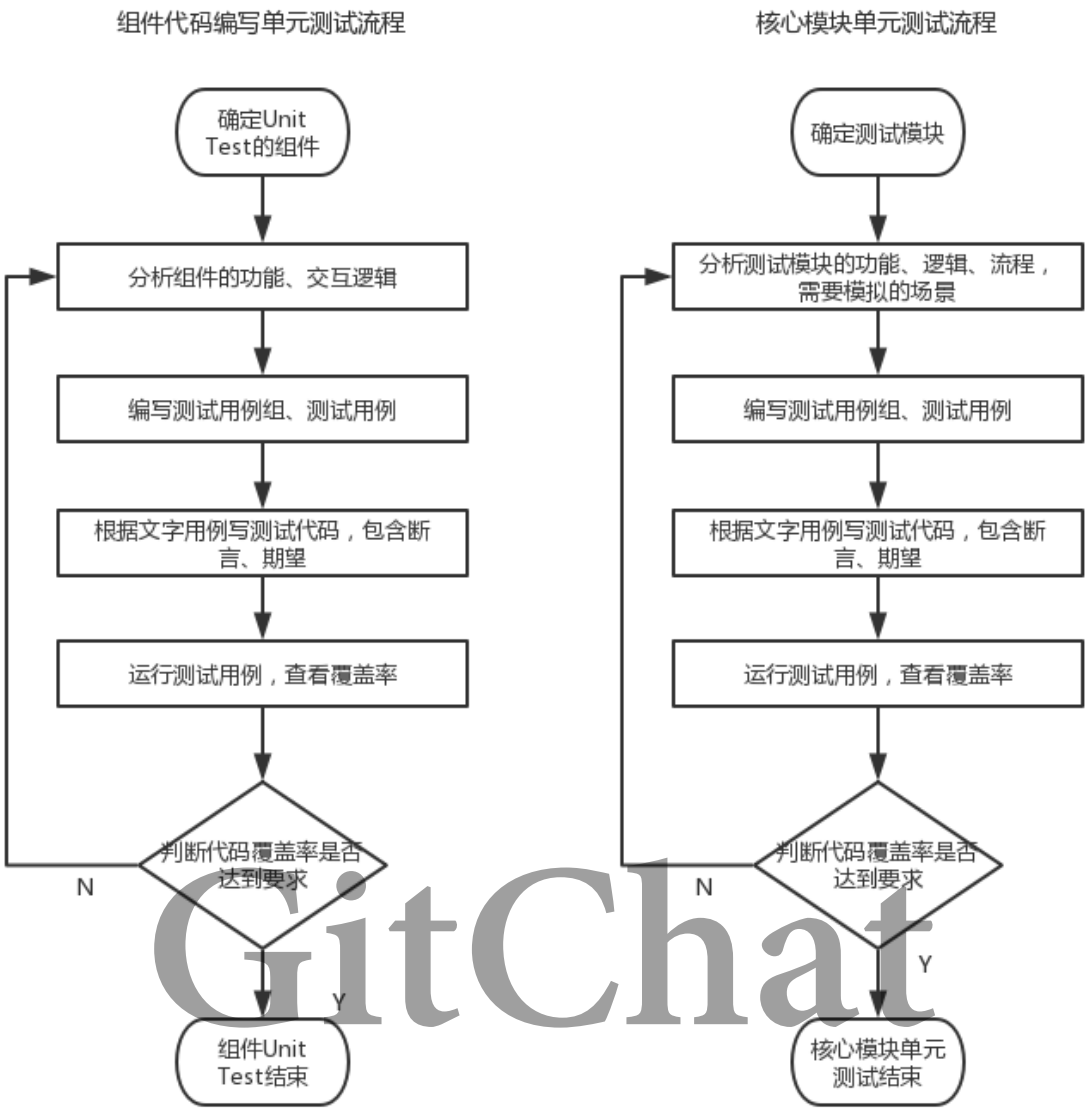
自动化测试

前端UT我们选择用Karma作为测试过程管理工具、Mocha作为测试框架、Chai作为测试断言库、Sinon作为测试虚拟方法库、Enzyme作为测试React代码的工具库，给所覆盖的代码包括前端UI组件库、核心架构代码，业务库按照需求定级，并根据级别定义覆盖率指标。

比如luui库提供了基础的UI组件，因此，针对基础组件，必须要对基本的函数逻辑、组件DOM元素、组件基本交互流程进行单元测试包括组件边界值的检测。

而lubase库提供的是核心的基础框架功能，包含与页面管理、与Native功能的交互、与容器的交互。因此，必须要对核心工具方法、基本的函数逻辑、API接口进行单元测试，也要对与Native交互的Task进行单元测试，还需要对底层的页面管理、内存管理功能进行单元测试。

为了方便UT的自动化，结合Jenkins是最佳手段。通过在Jenkins上配置测试脚本Job, 可以在Jenkins平台上完成测试用例的执行。配置Jenkins的Job执行后的报告输出为xml文件，可以将测试结果对接到Jenkins平台。同时，我们也需要将具体的在coverage目录下的测试报告通过静态文件形式维护在服务器端，通过链接展示详细的单元测试报告。测试报告介入到自动化报告平台。



前后端分离与SPA+MPA的思考

老生常谈了，我觉得很多人完全是因为前后端分离流行，大家也不管三七二十一，不理清楚做不做分离的好坏，一窝蜂的追潮流，就去做了分离，我觉得这是不可取的。

前后端分离我认为分三种，第一种是前端开发和后端开发分离，第二种是狭义前端代码和狭义后端代码分离，比如大家认为的浏览器端javascript和后端API的分离，第三种是广义前端和广义后端的分离，比如node层或者java web层作为大前端，后端服务层和数据层为后端的分离，web层仅仅做代码适配和透传。

分离与否？如何分离？这个要看你的项目性质，人员规模，公司发展阶段，你这个项目也就5个人开发，估计半年就结束，公司还有一堆其他更重要的事情有待处理，你说说看价值何在？

而陆金所的业务发展规模已经是涵盖十几条产品线，每一个产品线又可以继续细分出几十上百种类别，加上子系统已经在200个以上，系统的复杂度也已经到了不得不治理的阶

段。人员规模也发展到近千人的开发团队，如何更流畅的协作。另外hybrid还需要预埋和离线更新能力，前端代码的独立就更迫切了。

狭义上由于前端代码是需要同时打包到三个位置，第一个位置为离线增量更新zip包并放在独立的离线资源服务器，第二个位置还需要部分内置在Native APP中，变化少的通用资源预埋，好处是可以提高加载性能。第三个位置是打包的结果注入到各大web层，并随着web的java spring mvc controller层一起进一步打包为war包，并deploy到web server。所以前后端分离后可以保证同一份源码可以把结果同时吐到三个不同的位置。

SPA（ Single Page Application ）顾名思义是一个只有一个页面的应用程式，也就是说网页间并不会通过独立页面的跳转就可以达到一个复杂web应用的功能。

MPA (Multi Page Application)为了和SPA区分，在SPA为诞生之前的传统多页面应用，可以是后端根据每一个URL独立渲染的页面，也可以是静态化的独立页面。传统的网页主要采用「Multi-page」的设计模式，一个功能会有一个动作及一个页面。主要是因为过去是以 Client Request - Server Response的沟通方式，但随着网页技术的进步及使用者体验的考量，通过AJAX，能够局部刷新，而不用整个画面重新载入。让使用者更轻易地感受到与 Desktop application 的使用感。

SPA是基于狭义的前后端分离的规则，后端负责产生计算资料，前端负责页面的呈现。透过 Client 及 Server 端的区分，让前后端有更加的职责区分。主流的React、 VUE、Angular, Ember, Meteor, ExtJS …均支持SPA。

SPA带来的好处

- 因为第一次就加载了整个SPA应用逻辑到浏览器，所以此应用逻辑中单页内的子页跳转是不用重新发请求的，避免了页面闪烁。
- 前端产生直接render界面，无需请求后端。
- 更好的使用者体验（ UX ）及更快的互动性。
- 减少了服务器压力，因为在HTML资源加载上减少了后端的请求，而后续的交互只需要请求轻量级的数据API。

SPA带来的坏处

- 复杂，如果单页应用越大，所有的路由机制和状态控制都需要在前端js做，复杂度增加。
- 内存消耗高，内存控制比较难，如果是移动端开发，低配置机型，由于单页加载了大量的前端逻辑，GC很难控制，对性能和体验有负面影响。
- 不利于协作开发，如果一个开发介入一个SPA，其他开发很难参与协助，只能找关联度较小的页面参与。
- 初次加载非常耗时，因为需要加载大量的子页面逻辑，如果用户多数情况停留在此SPA的首页，当然不划算。

- 浏览器的原生导航不可用，如果一定要导航必须自行在前端代码自我管理前进、后退。

MPA的优点（所有SPA的缺点都是MPA的有点）

- 简单，对团队的技能要求不高。
- 无需担心内存问题，页面跳转后浏览器会自动释放所有内存，不用担心手机上面的内存溢出。
- 所有开发都可以并行协助开发，一个人一个页面，代码完全隔离，不受其他同事的逻辑影响。
- 首屏加载速度更快，更轻。
- 轻松使用浏览器的导航和书签功能等。

MPA的缺点刚好是SPA的优点。

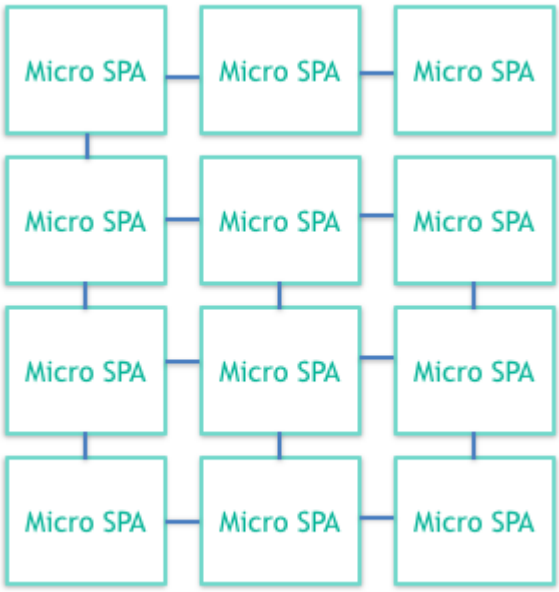
那好了，是不是如果你使用SPA就一定不能MPA呢？记得当时在SAP做基于移动的ERP解决方案，业务非常复杂，整个系统就基于一个超级SPA，虽然做了非常多精巧的GC和内存控制，但是性能问题始终是最头痛的问题，而且还完全牺牲了SEO。那我们是不是可以适当的根据自己的业务场景结合MPA和SPA的优势呢？

陆金所的业务是相当独立的，所以我们尝试把SPA微型化，大量的微型SPA也可以看成是具有一定MPA的页面，首屏和活动页可以采用server render，加载速度更快，SEO友好，然后预加载SPA的公共js，这样可以提高加载速度，然后client端渲染Micro SPA，由于是微型的，所以加载效率和渲染效率都非常给力，而且不需要去解决头疼的内存问题。

我们在团队内部曾经讨论过是否可以让这些独立的服务单元，灵活的组合，就像Martin Fowler提出的微服务概念：

- 分布式服务系统。
- 按照业务而不是技术来划分。
- DevOps。
- 智能的服务个体和弱通信低耦合。
- 容错。
- 快速进化。

每一个micro SPA是一个功能独立的模块，作为一个个独立的服务单元，那么我们是否可以把这些独立的服务单元通过协议组合在一起，形成一个大的服务流程？就像一堆前端的微型服务，服务之间可以随意按照协议组合。而每一次模块切换，内存会自动释放并重新加载，因为做的非常轻巧，所以不用担心重SPA的一堆问题。是不是可以发展FAAS（Frontend As a Service）的概念呢？



Code Once Run Anywhere尝试

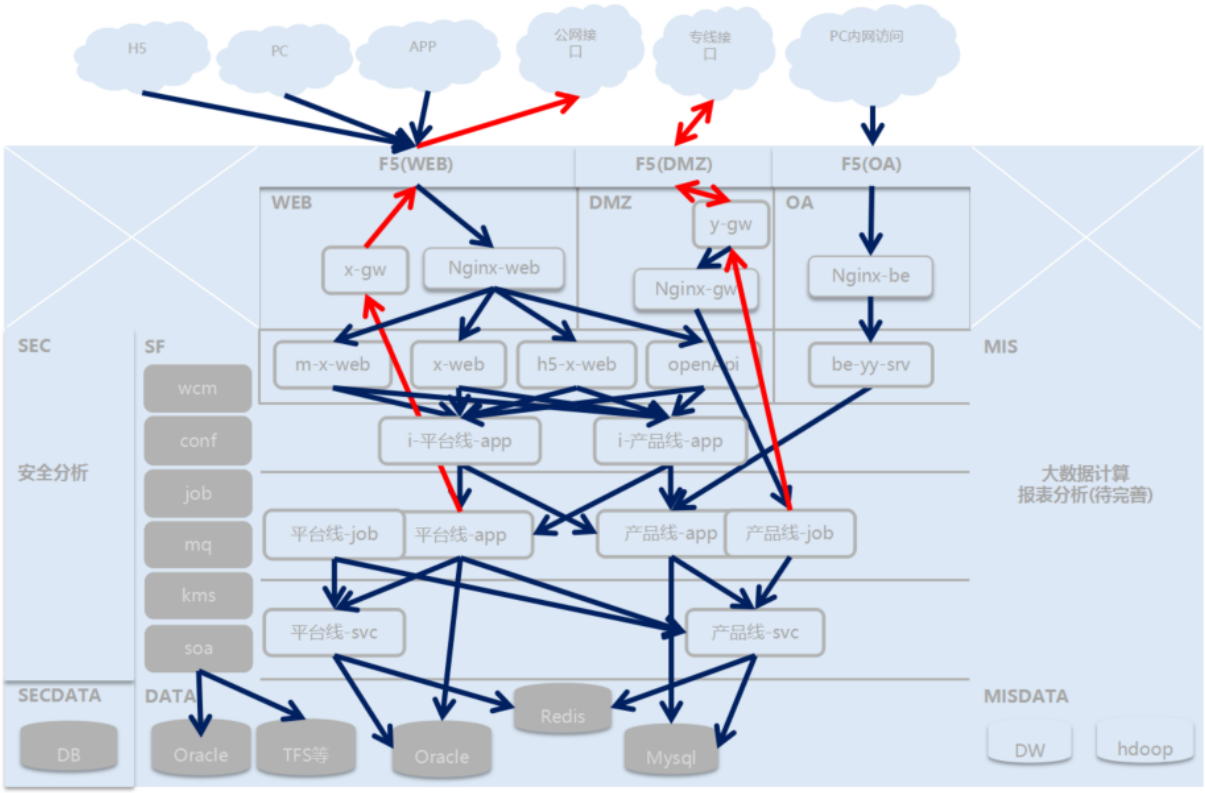
陆金所的一个业务上线，可能不仅仅是在陆金所APP一端使用，还需要同时运行在中国平安的一账通APP、金管家APP....等多个其它合作伙伴的APP，还有若干移动浏览器、微信等多达几十个完全不同的场景。

几十年前，无论java、.net、还有多少英雄豪杰，都提出了理想的Anywhere，回过头看历史，很多人都清楚了现实的残酷，能完美的实现Anywehre梦想的又有几何？但是梦想不灭，好处大家都非常清楚，但是要真做到，要克服的困难是难以想象的。不仅仅是架构和技术的单方面问题了，需要整个陆金所上上下下很多部门的参与和支持，这其中的挑战可想而知：

- 如何统一后端和前端的异常错误处理机制，并制定标准。
- 产品部门如何针对多端提供统一的需求。
- UED部门在交互和视觉的设计上面需要让步，并甄别交互的异同。
- 架构上如何识别并处理不同APP Webview和不同的移动浏览器，并在组件设计时做兼容。
- 如何支持多端的复杂页面导航跳转与返回。

系统拆分与依赖治理

在陆金所的早期，业务线的数量非常少，产品类别和开发人员也非常少，针对移动端仅仅由一个独立的接入层web提供服务。这样就存在一个严重的问题：单点故障，如果出问题，整个陆金所的移动服务将全部不可用。由于不能隔离发版，可控性差，发版内容互相影响非常严重。



看了这张依赖关系图，那一堆凌乱的箭头是不是很头晕？所有处于高速业务发展阶段的公司，为了把业务开发上线作为第一优先级，都会造成架构上无法跟上脚步，导致凌乱的系统关系，和后期的高维护成本。为了不让这种不合理的乱头发越来越糟糕，治理宜越早越好。陆金所的架构大牛们提出了一个非常好的治理规范：

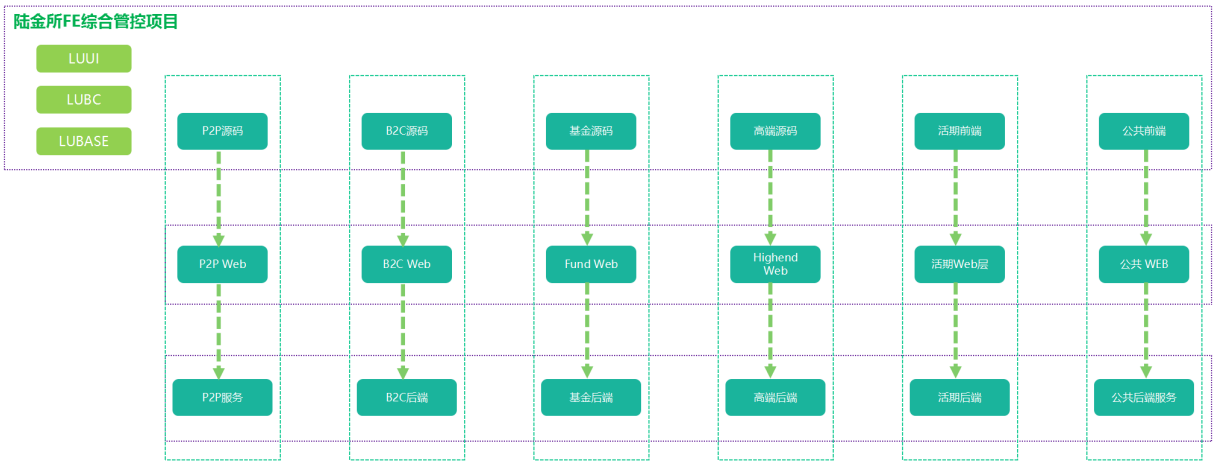
- 上依赖下，下不可依赖上，允许跨层依赖（除gw层）。
- 单向依赖，不允许循环依赖（除dmz区gw允许双向）。
- 同层不允许横向依赖。
- 线上系统不允许依赖线下系统。
- 核心系统不允许实时依赖非核心系统。
- Svc层不允许依赖其它系统，只允许依赖数据层。
- Web层、gw层、整合层不允许依赖数据库。
- 非本域系统不允许直接访问本域相关数据库。

各个域拆分，先按照一个规范的标准进行各个域的治理成为了解决这个问题的有效办法，按照业务线进行拆分，比如网贷会有一个独立的网贷web，网贷web只允许调用网贷app的接入层，由接入层屏蔽底层的复杂网状逻辑，让web层干净清晰，但是域治理的过程一定是一个艰难且长时间的过程。

要配合拆分和治理，前端必须具有非常高的柔性，模块的迁移和系统的拆分必须非常容易，接口的调用必须合理。迁移能力就必须确保模块的独立和内聚，接口的迁移尽可能避免网状调用。前端代码应该按照业务线分离，形成独立的代码库，并遵循统一的前端

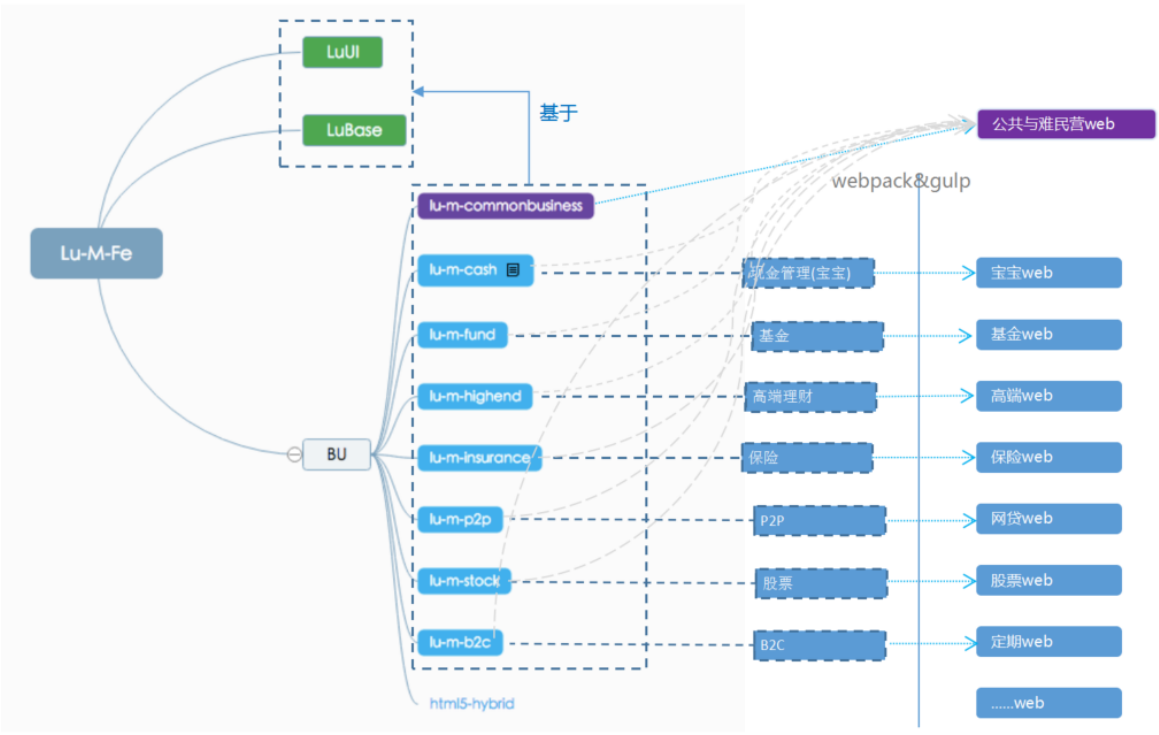
配置和规范。公共库由架构负责维护升级，并支持版本化，发布到公司的私有npm，各BU自由依赖公共库。BU线纵向拆分，拆分为独立前端的git库和与之一一对应的web层git库。完全分离web和后端application，清理混乱的交叉调用，比如A-WEB只能调用A-APP系统，不能交叉调用B-APP的API，这样做的好处是发版的影响可以尽可能的控制在A这条前后链路上，而不受B的影响，当然也不去影响B。

理顺后应该是这样的：



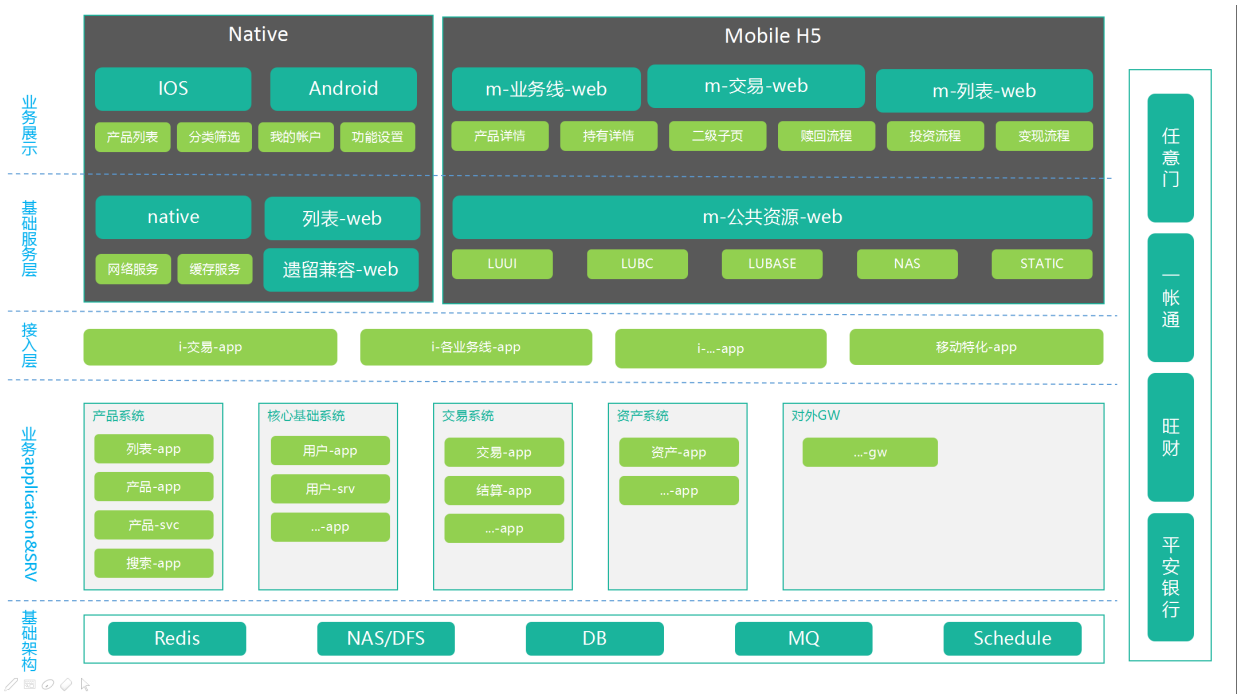
现在看起来调用链清晰多了。每一个前端代码和后端的域一样保持完全独立，拥有独立的git库，可以独立打包，输出到独立的web，并走独立的发版窗口。

拆库和治理一定是一个充满风险而且漫长的过程，这个过程中，会导致产生很多中间临时态系统。比如下图中的公共与难民营web，就必须承接没有完全拆分的代码，这样造成了前端源码和前端web的关系变得错综复杂，给打包工作增加了很大的复杂性。你出来混留下的技术债务迟早是要还的，如果能在前期睡服产品经理和测试当然最好不过。



上图中间哪一堆像乱头发丝的就是我们希望梳理清楚的点。

这张图是一个简单的前端与后端的概要图，几百个子系统画不下。



兼容性

切换到新框架并不是一蹴而就的事情，到今天我们还在同时维护5套框架体系，在打包的时候必须考虑如何兼容老框架系统。我们是在打包的shell脚本中，简单的包含老框架的打包程序，老框架的更新是一个持久的过程，如果测试愿意，一次性的代码清理是必要的。

Online化与Offline Online双活机制

如何提升性能都已经迫在眉睫。移动端承载了92%以上的流量，移动端的特殊性，如何在保证稳定性的同时支持动态更新能力。

陆金所的业务体系决定了他需要比其他互联网公司更快的交付能力，互联网金融行业，承载的业务与功能政策变化很大，需要具备很强的动态更新能力，第一时间响应需求的变化，并保证最快速的上线和最高的用户可达率，比如合规问题需要改造等。

在这样的背景下，业务直接走online化是最直接的做法，虽然业界有各种离线更新，增量更新方案，但是成功率和可达率都无法和online化匹敌。online化最大的问题就是性能问题。

Offline的问题：

- android机型错综复杂，如何保障离线包的下载成功率。
- 解压的过程如果突然断电，或者kill app，就算有恢复机制，如何监控所有机型都能成功恢复。
- 由于webpack打包结果很难做diff，并不是所有的增量更新都是安全的，所以必须通过全量更新来弥补安全问题，如果不进行系统拆分和模块拆分，全量包就会非常

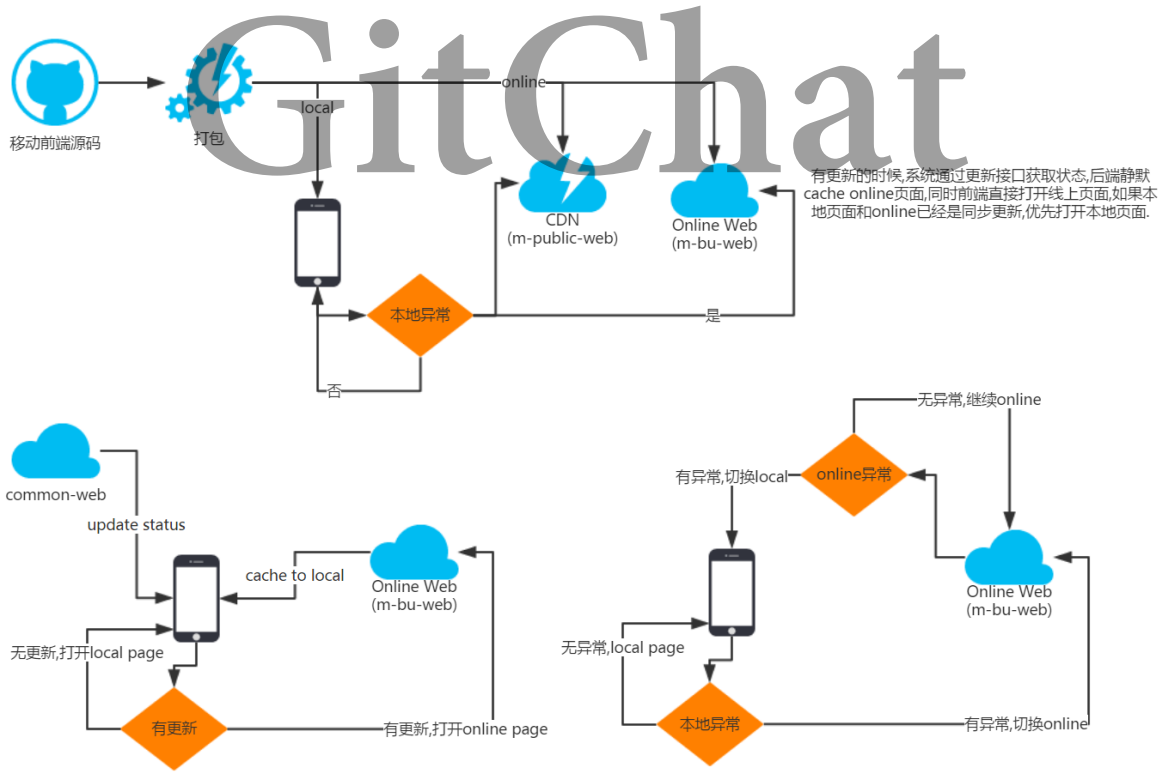
大。所以前端打包必须支持到针对一个git库的每一个module的独立打包。这样可以确保离线包的粒度更细小。

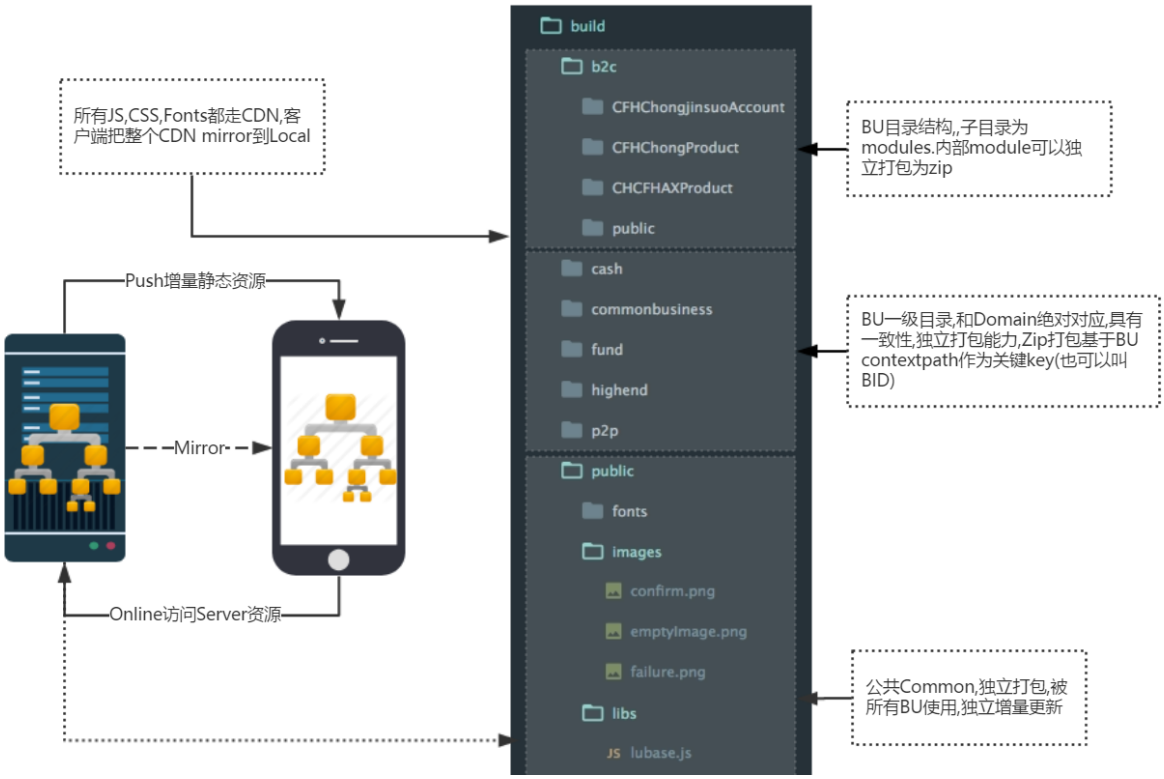
- 很难做到按需推包，如果过于频繁的推包会导致用户的流量浪费，所以我们把打包和发包的粒度控制在尽可能小的粒度。
- 提供一个推包平台，可以让各业务团队的开发自己推送各自的离线包。

但是online化带来的最大问题就是性能问题，而且online也不是完全稳定的，比如2016年曾经出现CDN全线瘫痪，半天的瘫痪造成了几十亿的交易量损失，当时数据接口调用一切正常，如果我们有offline，就可以完全避免这样的损失。offline和online都有自己的优缺点，那如果我们探测他们的异常情况，并自动或手动的切换这两种模式，就可以取长补短，做到双活的能力。

我们通过一套第三方配置和开关系统，提供配置策略，根据某一个业务或者模块切换offline或online，这是手动模式，下一步我们希望做的是根据探测线上和APP的异常，提供自动切换的机制。每一次这样的基础架构改动和上线，我们都是很紧张的，所以我们采取放量策略，针对任何一次基础架构的更新，我们都采用10%、50%、100%的放量策略。

双活机制





性能分析

这是最近陆金所的重点，由于篇幅所限，我在这里做一些介绍，可以另开篇幅讨论。

为了做到全链路，需要同时在native和javascript两端基于同一套标准进行关键点位的埋点，并把埋点类型分为如下两种：

-业务打点（业务方和产品经理比较关系业务数据，并通过业务点位数据进行用户行为分析）。

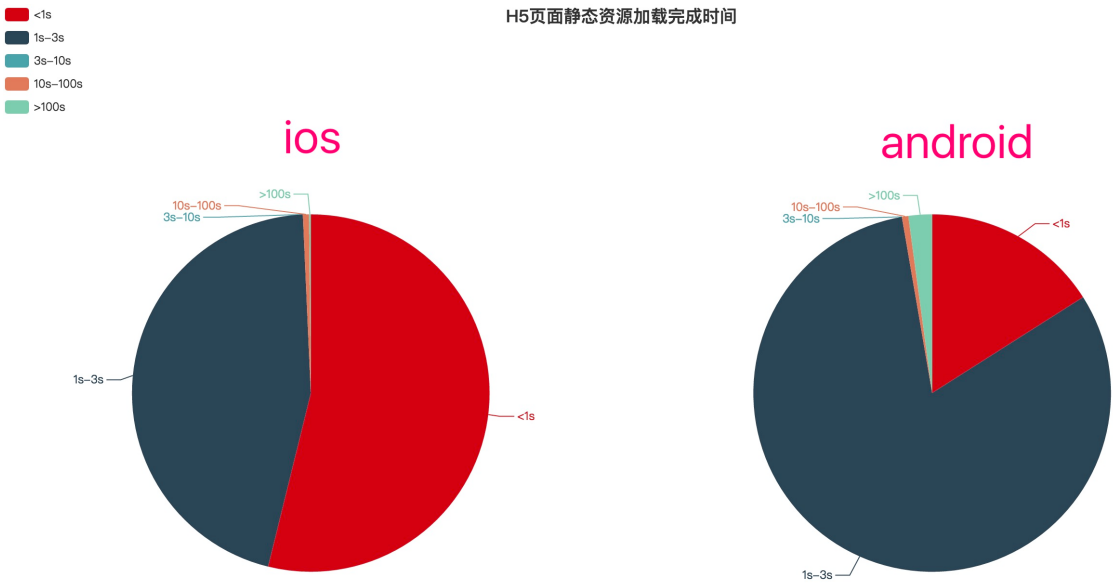
-开发打点（开发更关心运行期的异常和性能问题）。

性能指标综合归纳如下：

- Native端相关
    - APP启动速度优化
    - 关键（底层）方法性能优化
    - 首屏加载速度优化
    - H5/webview加载速度优化
    - 网络整体性能优化（http/2.0、protobuf、连接补偿）
    - 网络服务成功率
    - 网络服务耗时分布

    - 网络服务失败原因统计
    - 网络连接成功率
    - 网络连接耗时分布
- 页面和启动相关
    - 平均页面的渲染时间
    - 首屏出现内容及全部完成加载时间
    - 页面显示切换帧数
- Hybrid相关
    - 平均增量包下载个数
    - 离线包及资源平均下载大小
    - 离线包及资源下载耗时分布
- 内存和加载相关
    - 启动时内存使用量
    - 最大内存使用量
    - 启动耗时分布
- 崩溃率
    - App崩溃数与App启动次数的比率

分析的一些简单结果：



分析结果表明：

- Android的平均速度比IOS综合慢1.2秒左右。
- Android的initDomTree平均耗时达到1.9秒，说明Android对HTML5和javascript的综合处理能力还是比较糟糕的，一方面是被低配手机平均了，一方面是系统优化确实落后于IOS。
- 某些业务的白屏率很高，达到8%，一般情况可能是js error导致，也可能是弱网导致，这需要根据你的采集维度去进一步确认。

React Native和Weex

为什么Hybrid会成为现在的主流解决方案，因为我们的业务要求我们有更快的开发和发版效率，同时又要求我们有机制的性能和体验，而这两个在native和h5互相望眼欲穿，Native期望H5的跨端和发版灵活度、H5渴望更快的速度和更高的硬件调用权限。虽然国内已经有很多解决方案，比如RN和Weex，虽然解决了代码编译成二进制代码直接运行，秒开体验，我们也尝试了一些活动页面使用weex，但是开发体验真的还有很长的路要走，做web的看到这一堆阉割版的布局和标签，也是欲哭无泪。无论RN还是Weex在热度过后还需要趋于冷静。2017年以后还会有很多前端技术会程序爆发，让我们继续痛苦的幸福着吧。

最后

以上内容每一个点展开都是一个专题，更多的是让大家了解我们这些年在互联网金融领域的一些前端尝试，虽然已经累计了一些经验，但是摆在面前的难题还有很多，也希望大家一起讨论交流，我们也给自己提出了一些需要继续努力的问题：

- 如何实现更高的动态化，降级？以及不同技术方案间的无缝切换？
- 如何更好地利用大数据进行更有效、更实时的风控、推荐、分析？
- 如何有效地提供发布前的通用生产验证方案，以及发布后的灰度方案？
- 如何进行平台级的优化，在整体架构、H5/Native技术、网络等各方面达到业界领先的标准？
- 在采用新的技术和特性时，如何评估与现有平台的关系和利弊？
- 如何搭建更透明、实时的监控体系，加快线上问题的处理速度？
- 如何设计出松耦合，高内聚，更具备可伸缩性，可扩展性的架构，并利用更新的技术应对快速变化及发展的需求以及项目周期？