阿里云 开发者社区 D2 前端技术论坛

2020前端工程师必读手册

阿里巴巴前端委员会推荐

覆盖5大热点前端技术方向

语言框架/智能化/微前端/Serverless/工程化，
5大热点方向10+核心实战，解锁前端新方式，
挖掘前端新思路

关注 Alibaba F2E
了解阿里巴巴前端新动态

访问开发者社区
扫码领取更多免费电子书

D2 前端技术论坛
扫一扫二维码，关注我吧

扫一扫二维码
获取超全 D2 大会 PPT 资料下载

目录

语言框架

JavaScript 语言在引擎级别的执行过程

作者：周爱民

文本中将由南潮首席架构师周爱民为大家介绍 JavaScript 语言在引擎级别的执行过程，其中包括 JavaScript 语言中的环境的准备，作用域及环境的区别，可执行上下文的构建及执行原理，过程中的控制和执行结果的返回。在最后，周爱民展开语法的概念，解释 ...x 如何构成可执行组件。

嘉宾：周爱民，南潮首席架构师，曾担任支付宝业务架构师，盛大网络平台架构师。著有《大道至简——软件工程实践者的思想》、《大道至易——实践者的思想》、《Delphi 源代码分析》、《JavaScript 语言精髓与编程实践》等专著。

本次分享将主要围绕以下五个方面展开：

一、环境的准备

二、执行上下文

三、过程控制

四、结果返回

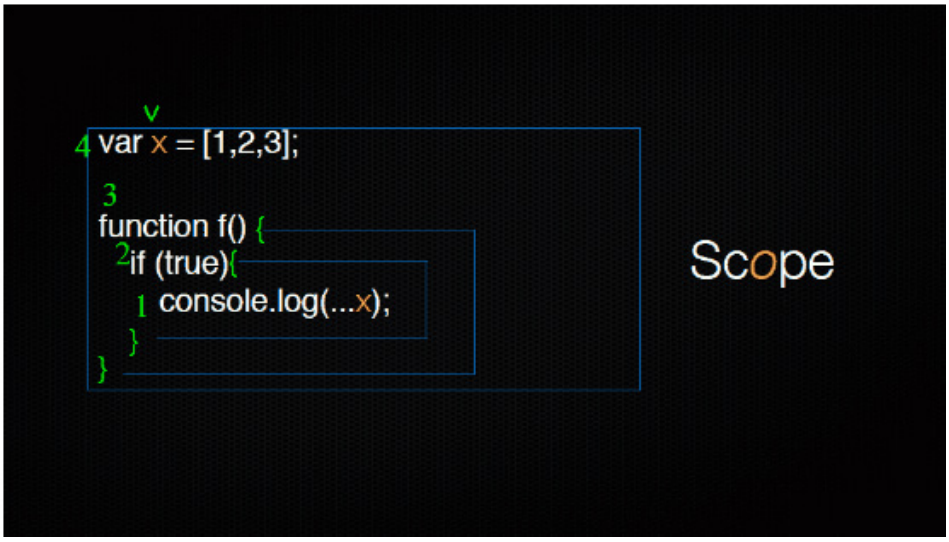
五、展开语法

一、环境的准备

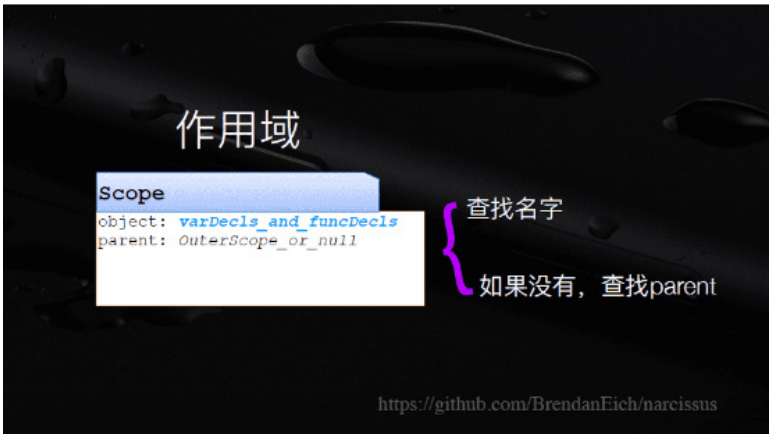
1. 作用域 Scope

代码当中经常出 ...x 的一段代码，表明可迭代对象的展开。而事实上，直接执行这段代码并不正确。必须将其放在一段表达式内才有可能被执行。如下图中的

console.log(...x)。但是如果仔细推敲，此表达式依然无法执行，再往上追溯到第二层，可以将此表达式放到 if 语句中，但此时如果 if 语句没有指明其作用域，则 x 依然无法在 if 语句中进行查找。此时需要为 if 加作用域，使其变成有块级作用域的 if 语句。但是下图中的 if 语句仍然无法找到 x，可以再上到第三层，在 function 函数作用域内，x 仍然没有被找到。直到全局的范围内才可以找到 x 的信息，代码才可以被成功执行。整个过程中会涉及到作用域的概念，下图中蓝色框的即表明作用域。

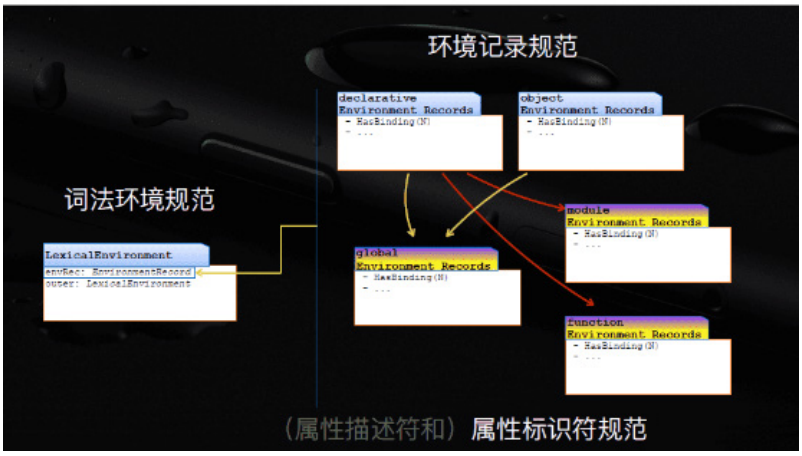


传统的作用域的概念在 JavaScript 之父 BrendanEich 在 github 上的 narcissns 项目中有所介绍。作用域本身有两个成员，object 和 parent，作用域中包含对象及属性。Object 是属性列表，其中做变量定义以及函数定义等声明，变量的名字可以被映射为 JavaScript 中的名称列表。因此作用域主要有两项功能，首先是查找名字，如果没有，查找 parent 上一层。



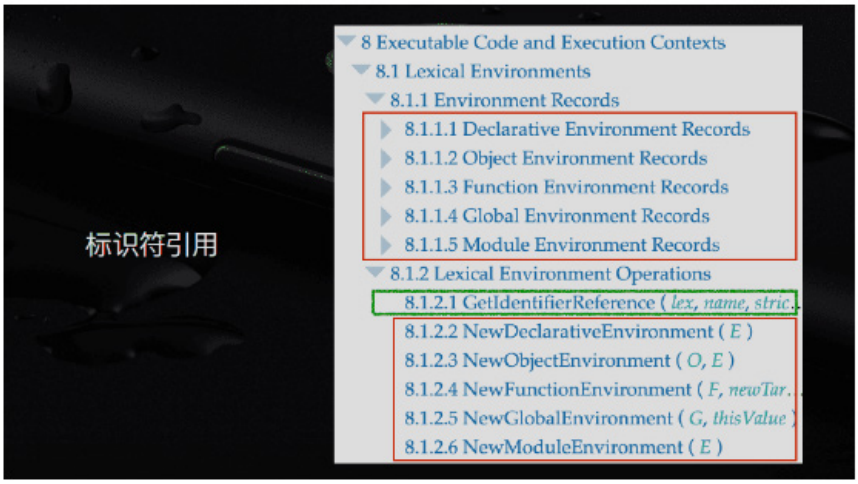
2. 环境 Environment

作用域 scope 的概念在 ECMAScript5（ES5）之后被替代为 Environment 环境，Environment 取代了作用域的价值和作用。下图中展示了环境在 ES5 以后的规范。首先是词法环境规范，依然包含两个成员，环境记录和 outer。环境记录可以映射为作用域中的 object，outer 映射为作用域中的 parent。此时，词法环境规范与作用域的内容完全一致，但不同点在于环境记录成员是由下图中右侧的五种环境记录规范所实现。可以发现，五种环境记录规范中都有一个共同的方法 HasBinding(N)，这个方法本身只是细化了查找名字功能。

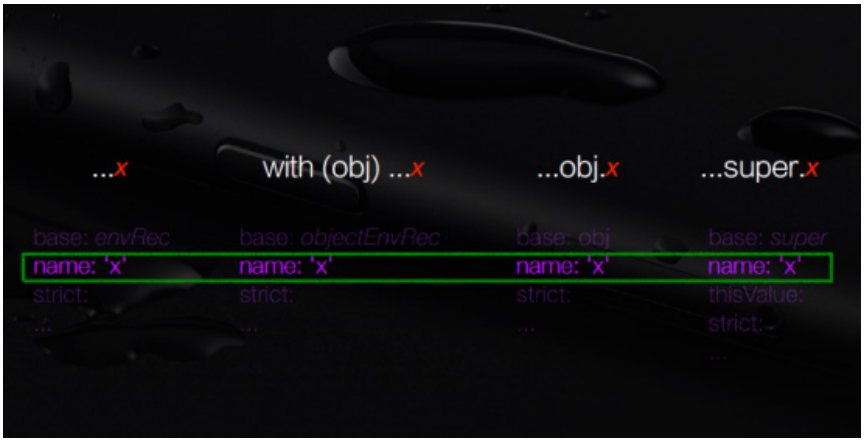


3. 属性标识符

ES5 中较为重要的规范是属性描述符和属性标识符规范。所有的环境记录通过环境对外只有一个有意义的 Interface，即标识符引用 GetIdentifierReference。



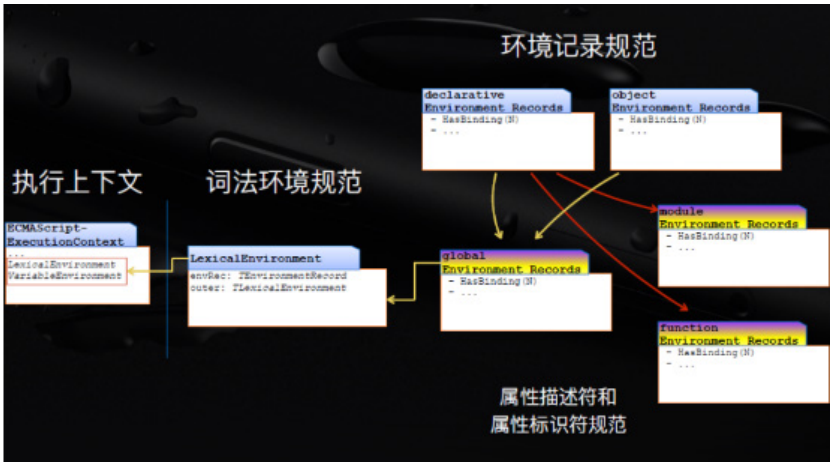
无论哪种环境记录都通过标识符引用取到 ...x，都会将其转化为同一种格式，如下图。其中有 base，name，strict 等信息，其中 name 都是一致的。标识符引用的作用代替了作用域中的查找名字功能。而 ES5 中引入标识符引用的方式的目的是统一和规范下一步的操作，即执行上下文。



当代码功能简化到查找名字功能时，才开始涉及到执行。下图中最外层是全局环境，里面一层是函数环境，再里面一层是词法作用域。将代码分为这几种环境之后，每个环境对外 public 的功能就是查找名字。

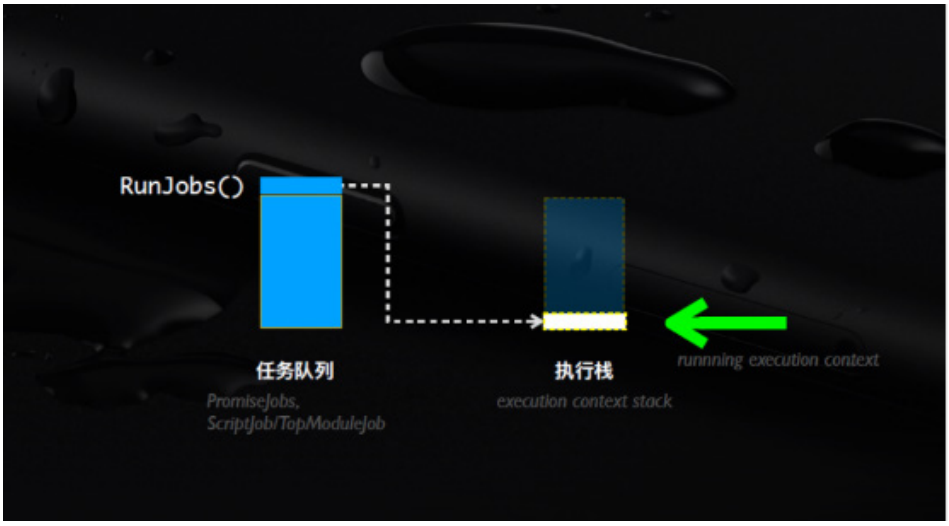
二、可执行上下文 Executive Context

在此基础上，执行上下文添加了两个成员，词法环境和变量环境。理论上词法环境和变量环境只需要有一个就可以查找名字。但 JavaScript 中变量环境解决 var 声明，词法环境解决一般变量声明，两种声明在 JavaScript 中不兼容。



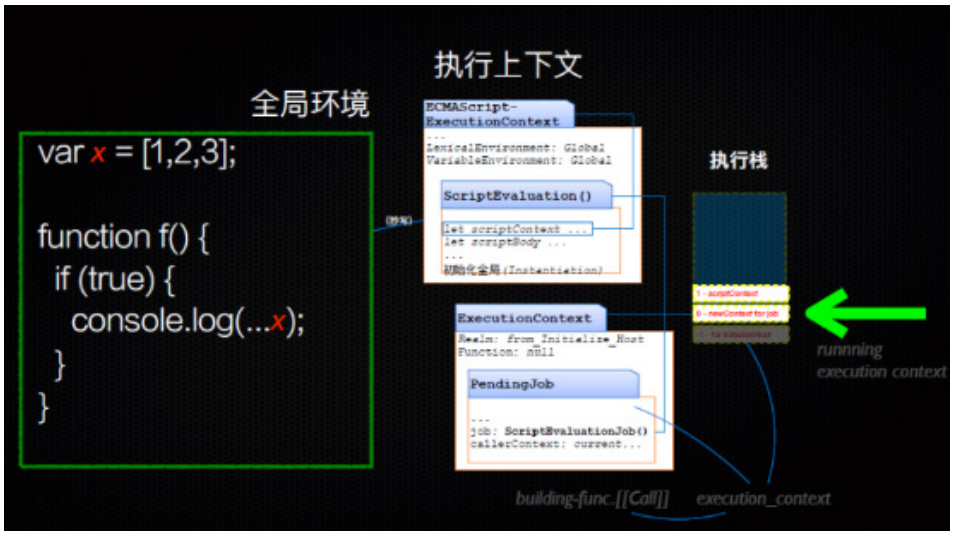
任务队列 RunJobs：任务队列以先前先出的规则处理任务。最早放到任务队列中的 job 是脚本执行 job（ScriptsEvaluationJob）以及顶层模块 job（TopModule-Job），之后开始 run。

执行栈 Execution context stack：在此基础上，又加了可执行组件执行栈。当执行栈为空时，自动取 RunJobs 中的最顶上的 job，开始执行。

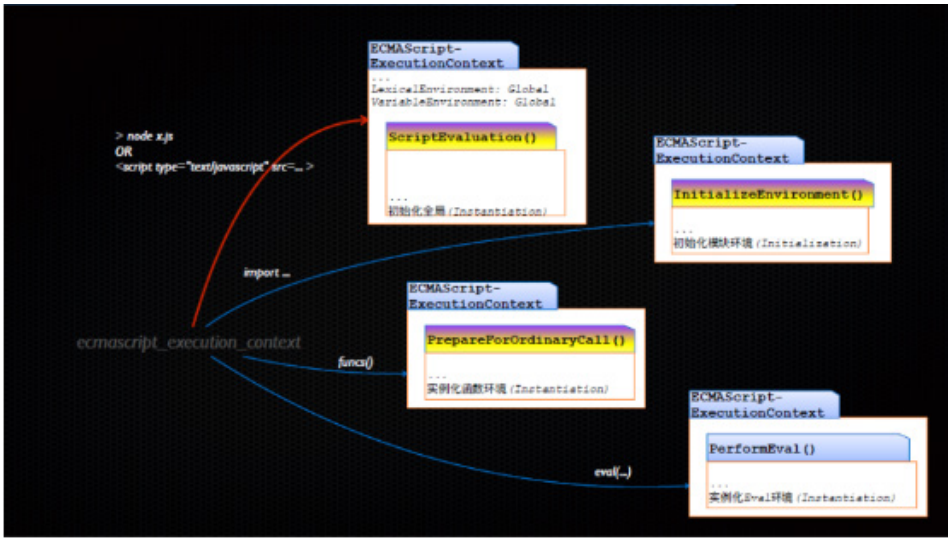


1. 代码层面如何 run

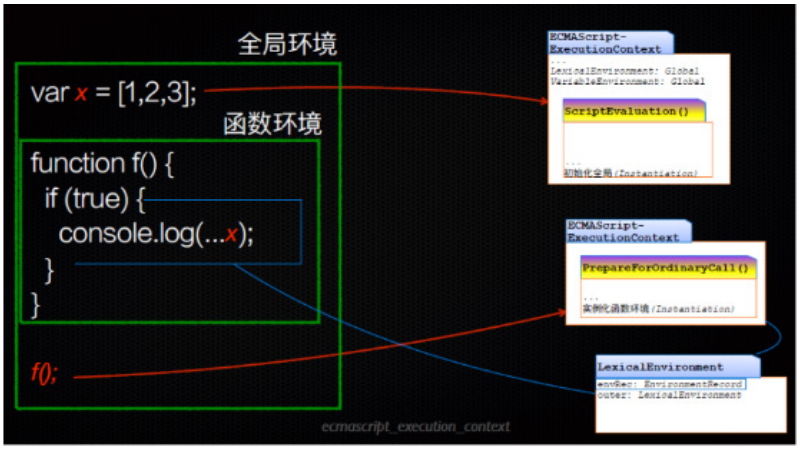
下图中左侧的代码块放到 JavaScript 执行引擎，此时处于代码还未正式执行，但引擎以准备好了前期工作。执行栈中有三个任务，最底层任务虚化的是初始化操作，第 0 个任务是 newContext for job，是为任务队列中的脚本执行 job 或者顶层模块 job 执行的新的上下文。此时可以执行这个 job，然后再创建一个 scriptContext 执行上下文。这里需要注意 scriptContext 执行上下文和第 0 个任务 newContext for job 的执行上下文稍微不同。第 0 个任务的上下文是内核引擎所需要的执行上下文，而 scriptContext 执行上下文是 JavaScript 代码可执行的上下文。此外，scriptContext 执行上下文中有变量环境和词法环境，可以访问代码。而第 0 个任务 newContext for job 的执行上下文没有这两个环境。



ScriptContext 执行上下文：ScriptContext 执行上下文具体还可分为四种可执行的上下文，全局初始化、模块初始化环境、实例化函数环境、实例化 Eval 环境等。



只有全局上下文的准备，代码中的 console.log(...x) 依然无法执行，还需准备函数环境上下文 call f()，代码才可以被执行。

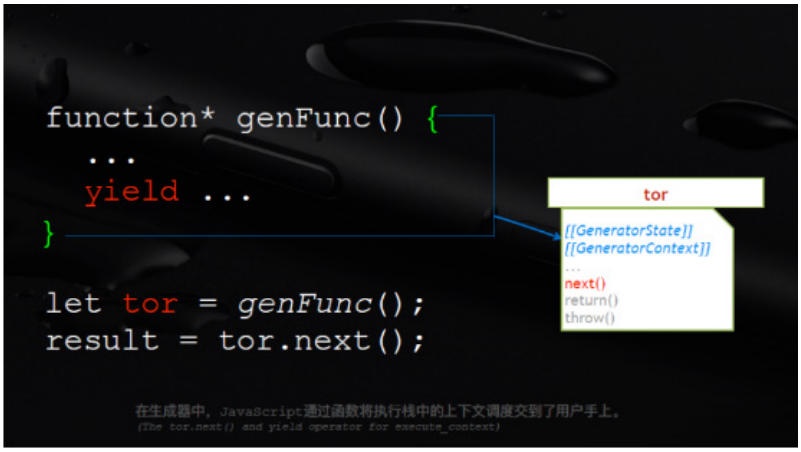


三、过程的控制

函数环境在 ES6 之后变得非常重要，几乎所有 job 都变成函数的调用。

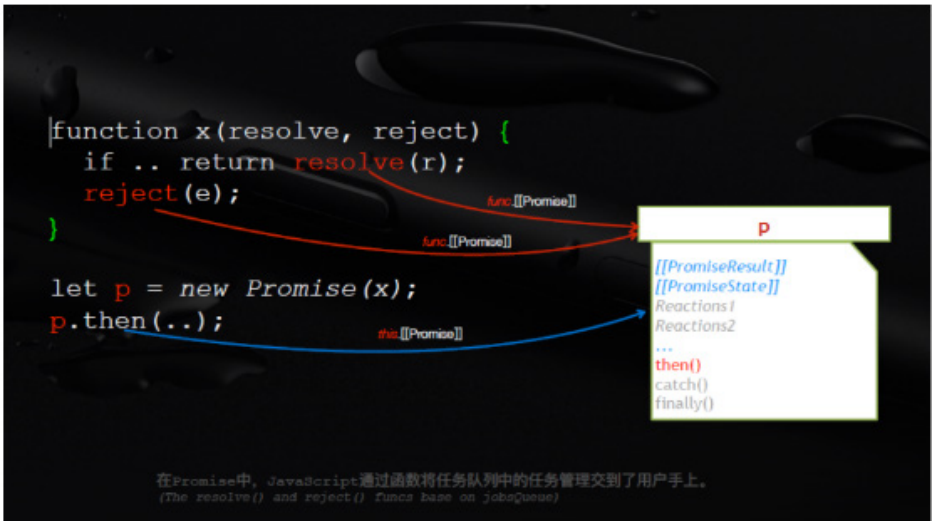
1. 生成器

下图中从 tor 中获取到对象，其中包含 GeneratorContext 和 GeneratorState，即生成器的上下文。在生成器中，JavaScript 通过函数将执行中的上下文调度交到了用户手上。在 tor.next() 中即直接将生成器上下文放到执行栈中，yield 可以将执行上下文从执行栈中弹出。



2. Promise

在 Promise 中，JavaScript 通过函数将任务队列中的任务管理交到了用户手中。下图中 Promise 需要先 new，再 p.then(..)。而 p.then(..) 的功能只是将所接到的函数放到对象 p 中的 Reaction 列表中。此时任务队列有两种，resolve 时的任务队列和 reject 时的任务队列。如果执行 resolve，对应的任务队列会被执行，即总会有一个任务队列不会被执行。但此时发现下图中的函数没有执行语句。原因是 JavaScript 创建 Promise 时一共创建了三个对象，首先是 Promise 对象自己，第二个是函数 resolve，第三个是函数 reject，三个同时被创建。后两个函数有同样的内部属性，promise 内部槽，指向对象 p。用户可指定执行 resolve 还是 reject。但如果执行栈中的全局初始化没有完成，任务队列仍然不会被执行。



四、结果的返回

函数调用后会有结果的返回，意味着 ES6 之后执行相关的特性都需要有值的返回。下图中第一行代码会返回 true，第二行代码返回 false。这两行代码代表了 JavaScript 两种代码的核心执行逻辑，执行表达式和执行语句。两种执行逻辑返回的结果值是不一样的。

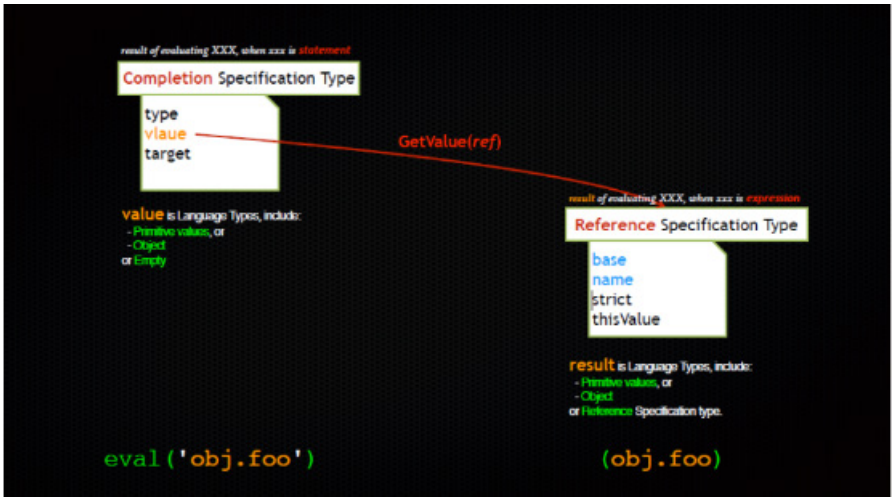
```
> obj = { foo() { return this } }

> (obj.foo)() === obj
true

> eval('obj.foo')() === obj
false
```

1. 执行表达式

如下图右侧，执行表达式返回的结果包括原始值，对象，引用规范类型。

2. 执行语句

执行语句返回的结果是完整规范类型，表示语句是否被完整执行，是否中断，返回值不包含引用。



五、展开语法

执行函数是执行表达式的一种，而执行表达式只能返回一个值，语句不能返回引用。可以发现 ...x 并不满足上述任何的返回值。当 ...x 放在“[ ]”中，则变成数值的展开，表示一堆 element 的填充，放到“( )”时，则变成参数的展开。只有在这两种场景中才可以使用 ...x，[...x]/(...x) 既不是语句也不是表达式，而是展开语法，是目前为数不多的可称为语法的可执行组件。展开语法不是以表达式或语句的方式执行的，而是直接在执行位置插入代码，即当解析数据声明时，遇到 ...x，于是将其放到数组列表中。

基于 WebAssembly 的 H.265 播放器

作者：陈映平

本次演讲由腾讯高级工程师陈映平为大家介绍基于 WebAssembly 的 H.265 播放器，以及在 NOW 直播中的应用。本次主要介绍采用 WebAssembly 制作音视频 H.265 播放器的整体思路、播放器架构设计以及线上实践。

演讲嘉宾简介：陈映平（程序猿小卡），腾讯高级工程师，IVWEB 团队负责人之一。

本次分享主要围绕以下五个方面：

一、方案背景介绍

二、方案选型

三、播放器架构设计

四、线上实践

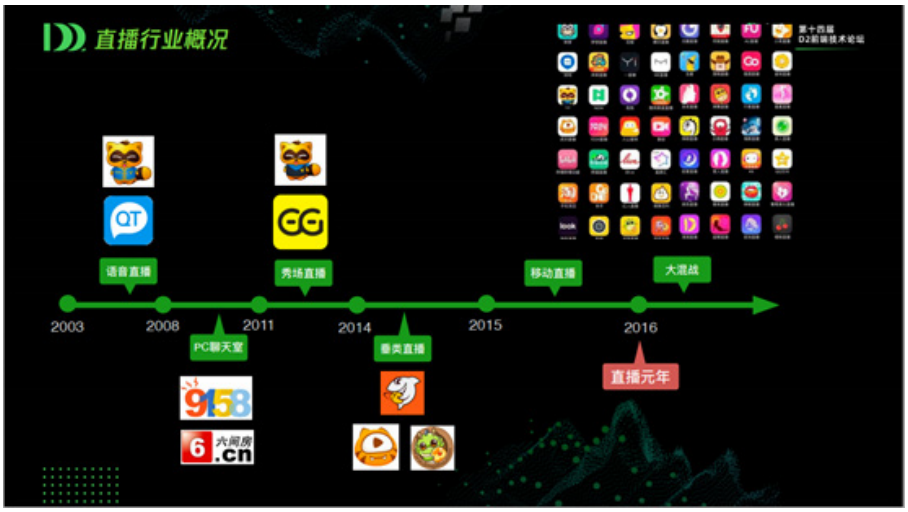
五、未来展望

一、方案背景介绍

1. 直播行业概况

下图为互动直播行业从 2003 年至 2016 年的发展变化历程。2003 年至 2008 年，语音直播行业比较受欢迎，平台有 YY 语音直播、QT 等。2008 年至 2011 年 PC 聊天室场景逐渐火爆，广受欢迎的有 9158、六间房。2011 年至 2014 年，秀场直播异军突起。2014 至 2015 年，垂类直播开始火爆。垂类直播主要指集中在某些特定领域和特定需求，以直播形式提供相关信息与服务的直播，例如游戏直播等。主要平台包括虎牙、龙珠等。2015 年移动直播出现，2016 年被称为直播元年。2015

年开始我国提倡大众创业、万众创新，市场上涌现众多资本。由于直播行业盈利强劲，资本纷纷涌入，出现千团混战局面。代表厂家包括虎牙直播、企鹅电竞、花椒直播、NOW 直播等。



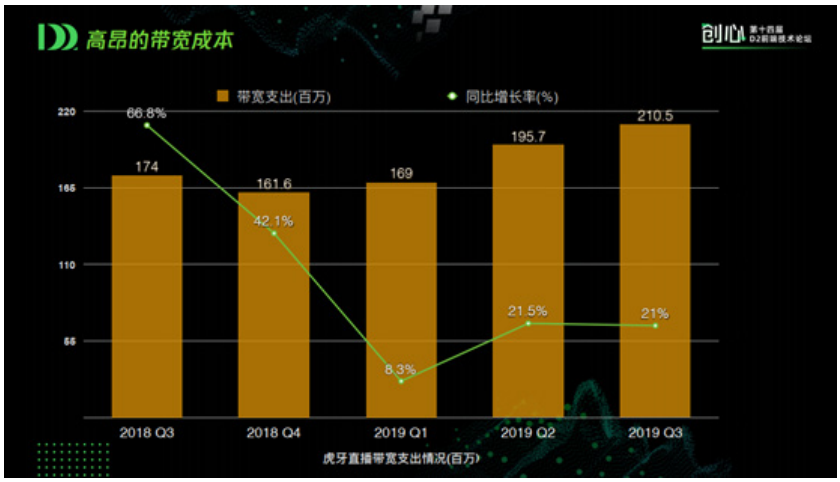
2. 直播行业成本支持

内容：直播内容成本包括聘请优质主播等花费。

薪酬：企业员工薪酬成本。

带宽：根据部分上市公司（例如虎牙）的统计材料，带宽成本比重大约为 10%~30%。对于部分小企业而言，内容成本比大企业低，带宽成本可能占总成本 40% 甚至 50% 以上。目前国内带宽计费方式通常有两种。一是根据使用量计费。例如观看一天视频，则根据当天消耗流量的总量计费。二是峰值带宽收费，即根据流量速度峰值计费，直播行业一般采用峰值带宽收费。下图所示为国内某知名云服务厂商的带宽流量计费方式，峰值带宽计费。例如某天游戏总决赛直播峰值带宽为 3.5T，按照下表收费即为 3.5*1024*1024*0.58。即一场比赛当天带宽成本就达到 212 万元，带宽成本高昂。

一些大公司会与云服务厂商签订战略合作协议，带宽收费可能会比对外公开价格低。下图为虎牙直播 2018 年 Q3 至 2019 年 Q3 的带宽支出情况统计。2018 年 Q3 虎牙直播带宽支出为 1.74 亿，2019 年 Q3 支出为 2.1 亿。同比增长率由 66.8% 降低至 21%，这是虎牙直播带宽成本优化的结果。



3. 直播方案需求

直播行业在技术选型时需要考虑如下因素。

延迟：互动延迟会影响用户体验，解决直播互动延迟问题十分重要。

成本：内容、薪酬、带宽等成本。

性能：关键指标包括音视频播放时的 CPU 占用、内存占用、卡顿、帧率等。

扩展性：音视频直播行业出现至今，音视频编解码领域飞速发展，经常发布更新编解码协议。如果播放器架构扩展性设计不好，一旦编解码协议更新或替换，就可能需要重构播放器架构，于业务方而言是难以承受的。

二、方案选型

1. 常见直播协议

RTMP、HLS、HTTP-FLV 都有各自的应用场景。

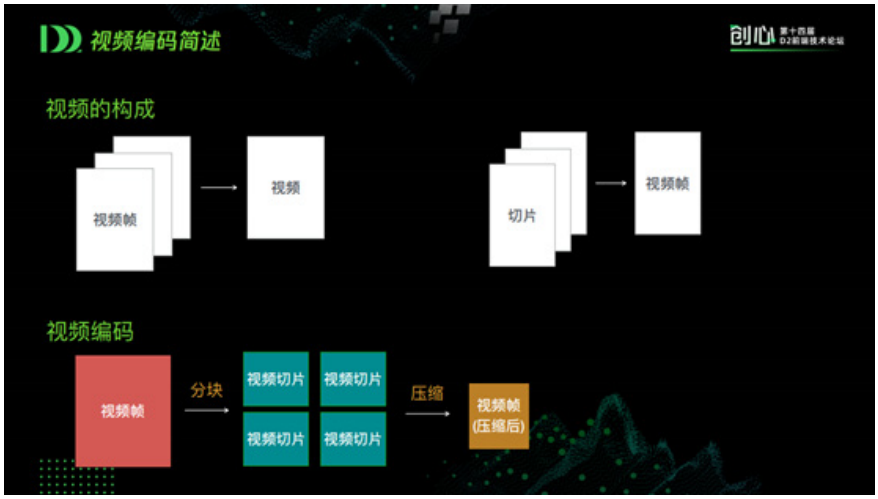
HLS：HLS 在移动端使用非常多，是苹果主导的一个规范。HLS 的特点是可以自适应码率以及带宽状况。例如可以根据手机的网络情况自动切换到不同的码率，可以使用户在 4G、3G 等不同网络条件下正常观看直播。

RTMP：Adobe 公司的私有协议，特点是延迟比较低。RTMP 的缺点，首先作为私有协议，协议细节部分为黑盒，出现问题时难以排查。第二是 RTMP 使用的不是标准 80 端口，会导致一些问题。例如一位直播观众需要在校园网环境观看直播。众所周知校园网络有各种保护措施，如防火墙等会屏蔽一些非标准端口。此时不适合使用 RTMP 观看。RTMP 协议较适合做推流。例如主播需要在直播间里进行直播，那么可以采用 RTMP 进行推流。即将直播流采集后上传到服务器，然后给用户观看。

HTTP-FLV：协议栈与 RTMP 相似。HTTP-FLV 与 RTMP 内部数据封装都基于 FLV Tag。HTTP-FLV 的延迟介于 RTMP 与 HLS 之间。同时 HTTP-FLV 使用的是标准的 HTTP 协议，端口是 80 标准端口，网络穿透性较好，因此一般在观看端会采用 HTTP-FLV。过去几年，由于浏览器不支持 FLV 的原生播放，许多时候需要借助 Flash 播放器进行播放。bilibili 出品的 flv.js，采用 MSE 解码播放的方式达到对 HTTP-FLV 的支持。

2. 视频编码简述

由于流量的重要性，需要降低视频对带宽的占用。如下图，视频是由一个一个的视频帧构成的。例如每秒播放 20 视频帧，连续播放就形成视频。视频帧在编码层面又是由一个一个的切片组成的。视频编码首先将视频帧分块为多个视频切片，然后采取帧内预测等手段对视频切片进行压缩。实际压缩效果较下图图示更加优秀。

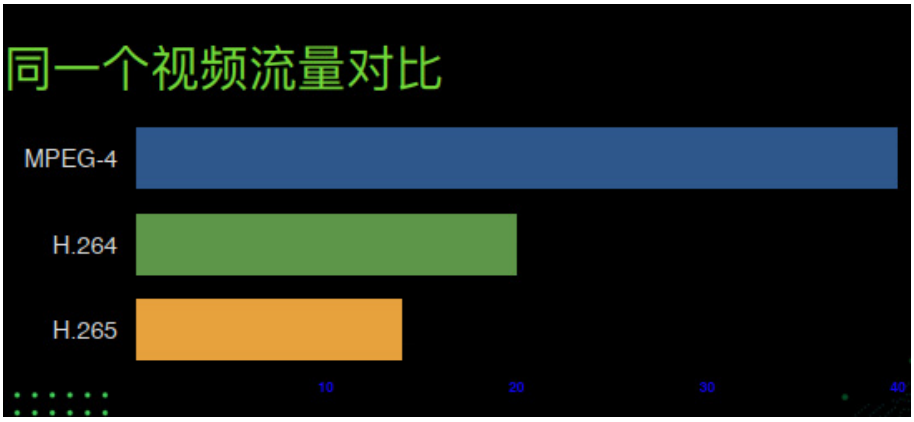


视频编码技术的重要性：举例说明直播视频的数据流量消耗。下图所示直播页面宽高比为 540*960，每秒 15 帧。每一个像素由三个通道组成，每一个通道占 8bit。则一分钟数据量为 540*960*8*3*15*60bit。转化为 Mb 为 1334Mb，即一分钟直播流量约 1.33G。如果视频的带宽消耗的确为上述程度，多数人应该不会在数据流量环境下观看视频。因此通过视频编码来压缩视频尺寸十分重要。对视频采用 H264 进行编码，上述一分钟时长 1334Mb 尺寸的视频压缩后尺寸仅为 11Mb。视频编码压缩的效果比前文图示效果更为显著，带宽费用降低至不到原来的 1%。

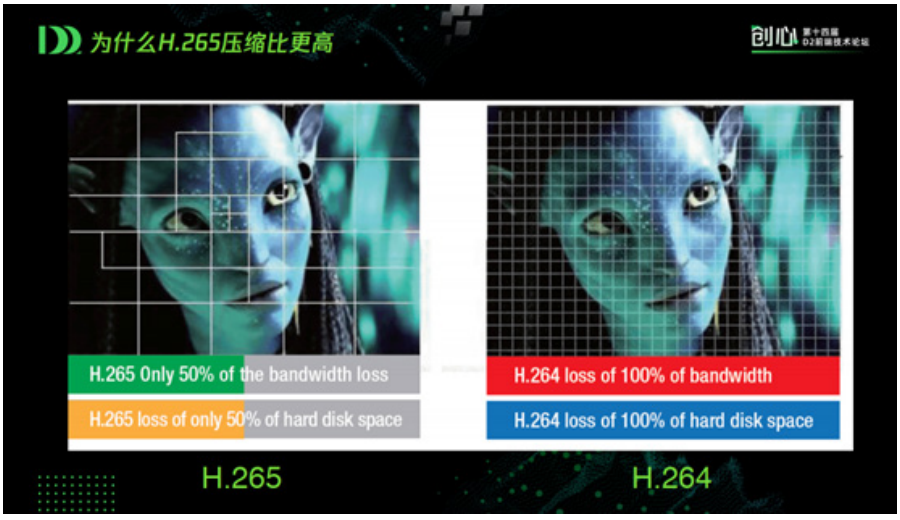
视频编码技术的发展：下图为音视频编码技术发展历程。1992 年 ISO/IEC（专家工作组）发布 MPEG-1 标准。MPEG-1 主要为 VCD 服务。1994 年 MPEG-2 发布，主要服务于 DVD。MPEG-3 发布目的是为 HDTV 服务，后发现 MPEG-2 已经满足该需求，因此废弃 MPEG-3，并将其优化部分并入了 MPEG-2。1998 年发布 MPEG-4 标准。2003 年 ISO 与 ITU 联合发布了一个非常重要的标准 H.264。H.264 与 AVC 是同一个东西，是在 MPEG-4 Part 10 中定义的。2012 年由谷歌主导的 VP9 发布，其被定位为下一代的视频编码解决方案。VP9 在 H.264 基础上进一步压缩视频带宽的占用，目标是提升 50% 左右的带宽。目前国内 VP9 的应用相对较少。2013 年 ISO 与 ITU 联合发布 H.265，相较于 H.264 进一步提升了带宽。

3. 选择 H.265

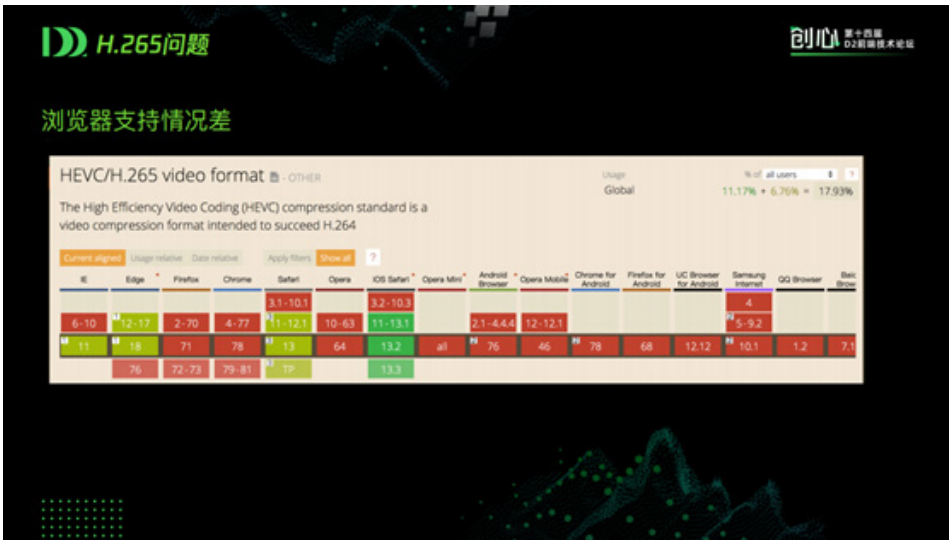
特点：首先在相同码率条件下 H.265 视频画质更高。第二，压缩效率更高：例如同时观看同一份分别采用 H.264 和 H.265 标准编码的视频，H.265 节省更多带宽，平均能达到 30%~ 50%。第三，由于 H.265 压缩效率高，因此码率要求更低。下图为同一个视频的流量对比，由于根据不同视频优化带宽能力不同，因此下图没有具体比重或数值。整体上 H.265 较 H.264 节省 30% 以上的带宽。



原因：H.265 压缩比较 H.264 高很多是因为 H.265 优化了非常多的技术细节。如下图，以对视频分块压缩过程为例。H.264 会将下图分为若干 16*16 的宏块，然后对其进行编码。采用例如计算残差等一系列压缩手段。H.265 整体播放设计架构与 H.264 没有太大差异，同样采用混合编码架构，同时采用的编码压缩的优化手段也差不多，可能会支持更多帧间预测的方式等。H.265 支持了更大的宏块，以及可变的宏块。例如下图左上角黑色区域，视觉效果乌黑一片。如果采用 H.264 编码方式，虽然其像素 RGB 值完全相同，但是编码为许多像素款。而使用 H.265 编码方式，当该区域 RGB 值接近，可以直接采用 64*64 的宏块提高压缩效率。



浏览器支持情况差问题：如下图，诸多浏览器不支持 H.265。H.265 是由多个厂商联合发布的标准，同时发布厂商对于 H.265 涉及的专利如何进行收费没有统一的标准。例如起初用户使用 H.265 标准进行视频编码并不收费，产品上线一年规模扩大之后，可能突然收到律师函通知用户专利费欠款上千万。据不完全统计，H.265 一年的专利授权费在 1 亿美元以上，价格非常高昂。因此 H.265 没有推广开来并不主要是技术方面的原因，而是专利授权方面的原因。相比之下 H.264 收费公开透明，也较 H.265 低。



4. 方案选型

如果需要在浏览器中支持 H.265 编码标准，应该做哪些准备。

业界方案参考：下表所示为业界支持 H.264 等标准的方案参考。flv.js 基于 HTTP-FLV 协议，解码方式为 MSE 软解。支持音频播放，支持 H.264 标准。不支持并且不打算支持 H.265。libe265.js 基于 WASM H.265 解码方案，H.265 C 库基于 libe265。现已支持视频播放，但不打算支持音频播放、流媒体等。

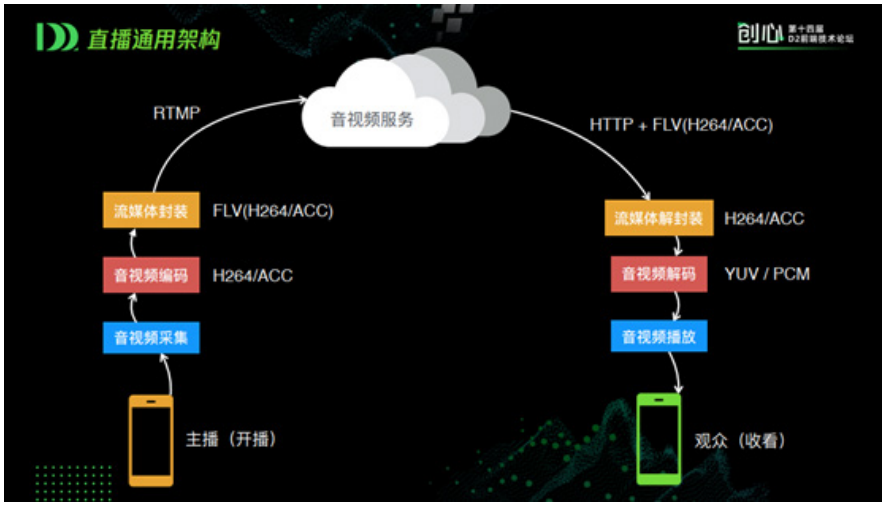
解码方案对比：MSE 复用现有能力方面，如果要支持 H.264，需要自行实现解码算法，获取 H.264 的视频数据，再重新编码为浏览器可支持的格式。如果要支持 H.265 也一样。WASM 复用现有能力比 MSE 好很多。WASM 可以将采用 C/C++ 编写的一些库转化为浏览器可支持的 WASM 模块。因此只需要做非常少量改动甚至无需改动就能在浏览器中复用现有能力，例如 FFMpeg。性能方面，MSE 采用 js 对视频进行软解，因此性能一般。WASM 性能中等，仍在优化。MSE 可维护性非常差，每当新加一种编码就需要重新实现解码方式。WASM 可维护性好。

最终方案：在业务场景中采用 HTTP-FLV+WASM+FFMpeg+H.265 解码方案。WASM 是可在浏览器运行的高性能模块，可以采用传统强类型语言如 C、C++ 等进行编写，再编译为浏览器可识别的高性能模块。FFMpeg 为跨平台的音视频录制、转码解决方案。FFMpeg 功能复杂，方案中主要利用其编解码功能。WASM 浏览器支持情况如下图。用户可根据产品支持平台的需要等判断 WASM 是否适用。

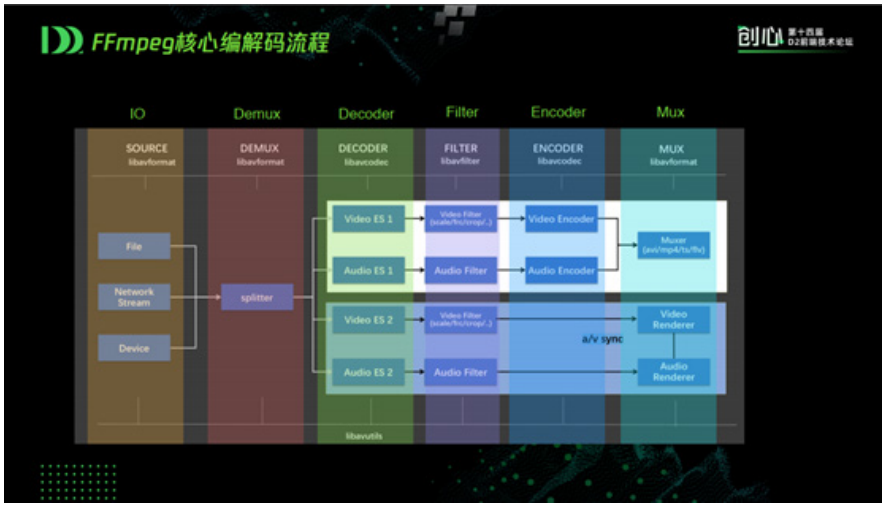
三、播放器架构设计

1. 直播通用架构

传统直播由主播到用户的链路架构如下图。主播开播、打开设备进行音视频采集、音视频编码、采用流媒体协议进行封装以在网络上传输、推向音视频服务。

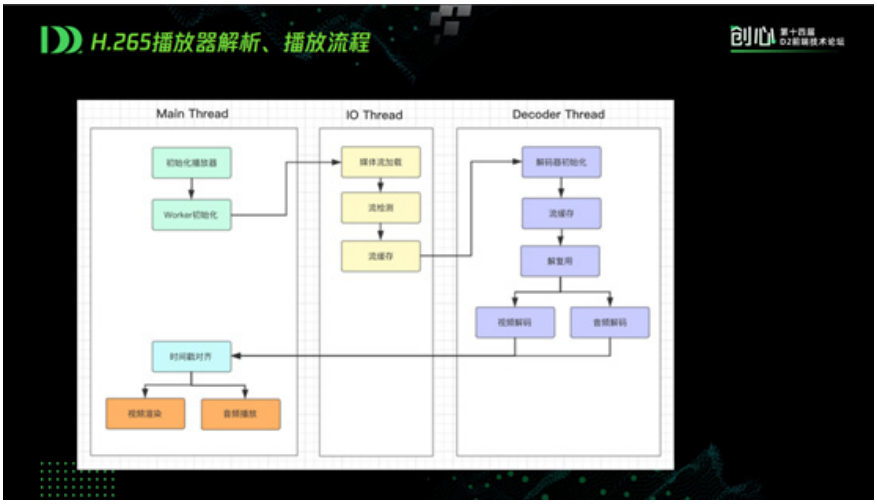


2. FFMpeg 核心编解码流程



3. H.265 播放器解析、播放流程

解码器初始化后，会在解码线程中进行一定的流缓存，存储到一定量的数据后再将数据发送给解码器。否则解析过程可能出现问题。例如数据发送给解码器后，解码器首先需要确认数据的编码，第二确认音视频帧率，第三确认视频的宽、高，若缺少以上初始化信息，可能导致解码出错。接下来进行解复用。然后进行视频解码以及音频解码获取视频帧与音频帧数据。将音视频数据发送至浏览器播放之前，需要进行时间戳对齐，否则会导致音画不同步等问题。最后进行播放。



4. 播放器整体架构

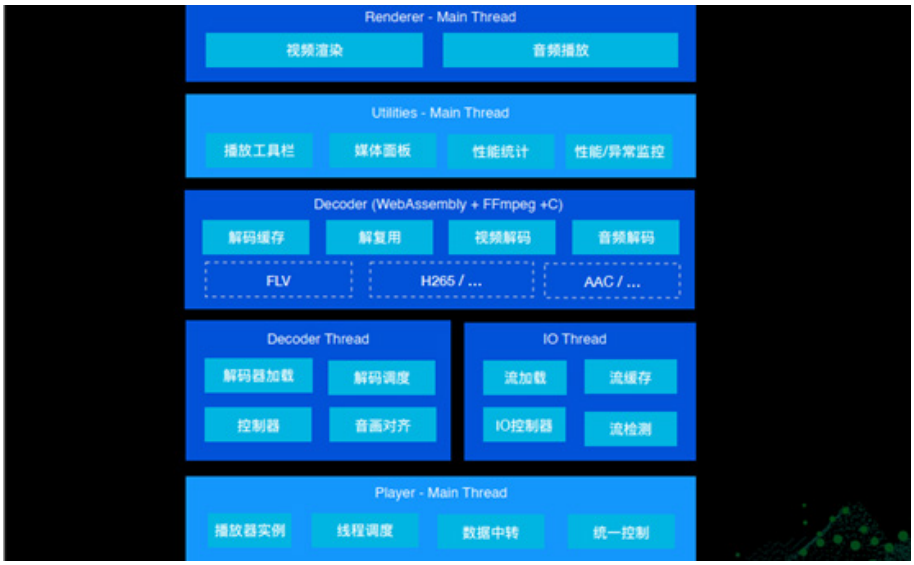
最底层：主线程，包括播放器实例、线程调度、数据中转以及统一的控制逻辑。

上一层：两个核心线程，包括 IO 线程以及解码线程。

中间层：播放器的核心环节，借助 WASM 以及 FFMpeg 对音视频数据进行解封装以及解码。

中上层：基于播放器的工具。例如在播放器中可能包括播放工具栏、用于展示流媒体信息的媒体面板。比如展示视频地区、宽高、帧率、码率等。

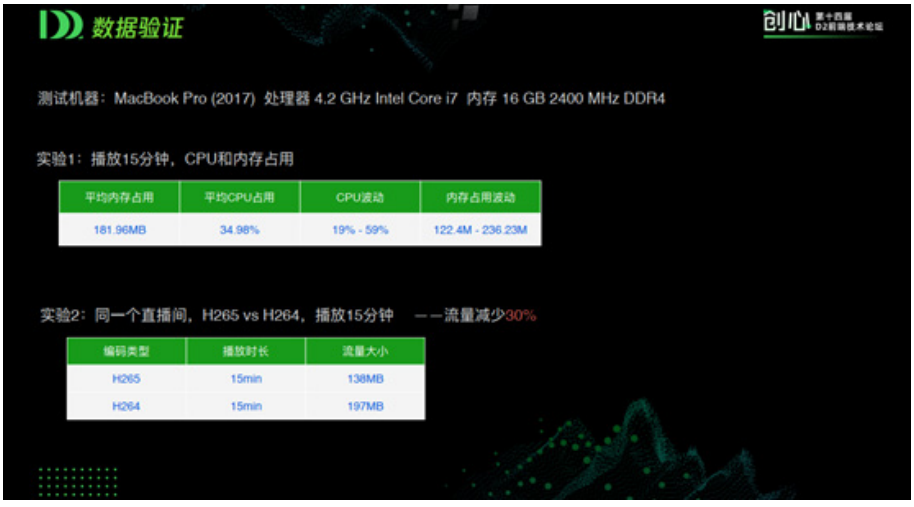
顶层：播放及渲染。



四、线上实践

1. 数据验证

如下图，实验 1 为本地测试播放 15 分钟视频的平均 CPU 及内存占用分别为 181.96MB 以及 34.98%。实验二为 H.265 与 H.264 观看同一直播间 15 分钟的对比情况。H.265 占用流量减少 30% 左右。

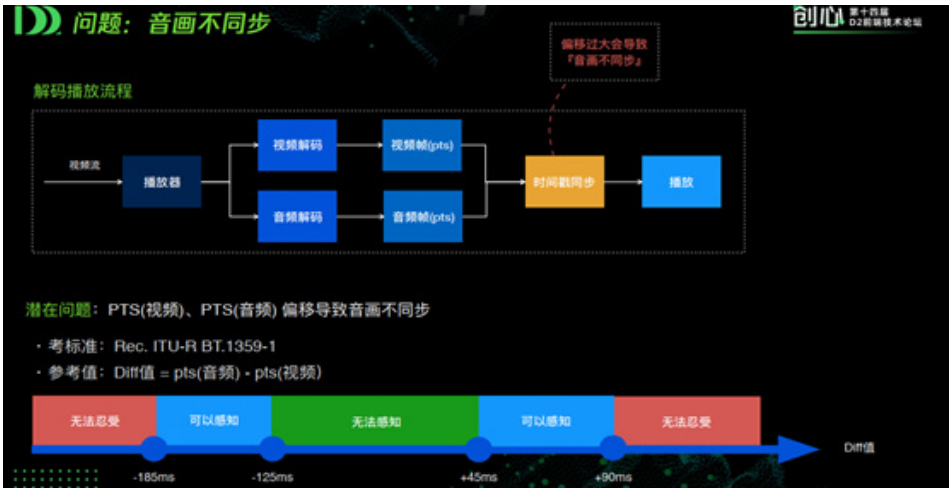


2. 典型问题

FLV 规范如何支持 H.265

在 FLV Tag 中存在一个关键的 CodeID 字段，标识了 FLV Tag 中视频数据的编码。然而 FLV 规范中还未明确指出 H.265 应该采用哪个 CodeID。目前腾讯云等用于标识 H.265 的 CodeID 为 0x12。

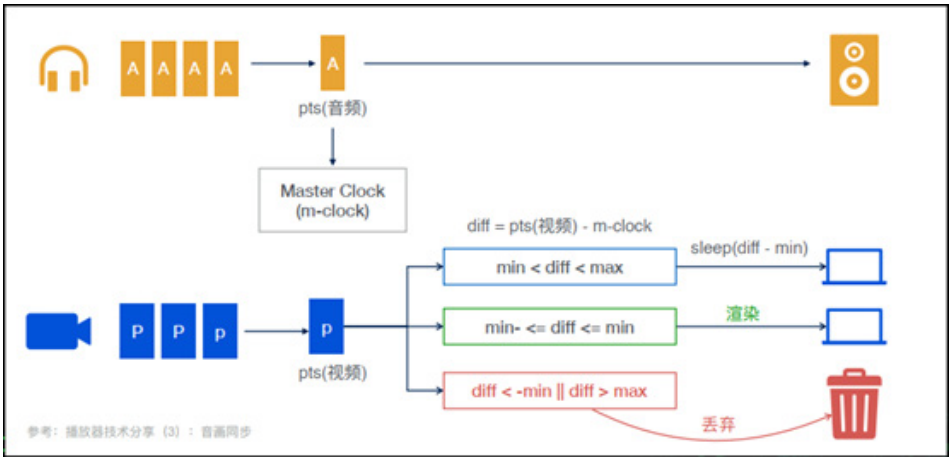
音画不同步：每一个音频帧与视频帧都包含 PTS 信息，用于标识音频帧与视频帧应在哪一个时间进行播放。由于视频与音频分开播放，如果二者 PTS 不同、出现明显偏移，就会导致音画不同步。根据 ITU 的规范，设定 Diff 值为音频与视频 PTS 值差异。如下图，当 Diff 值在不同范围内，的音画差异可分为无法感知、可以感知、无法忍受等程度。当 Diff 值在 −125ms 至 45ms 范围内，可视为音画同步。故解决音画不同步问题由此着手。



第一种方案：音频帧与视频帧进行对齐，音频帧根据视频帧播放进度对齐播放。

第二种方案：视频帧与音频帧进行对齐，视频帧根据音频帧播放进度对齐播放。

人耳对流式音频更敏感，容易察觉到不连贯。人眼对视频帧刷新相对不敏感，不易察觉。故采用方案二。设置 diff 值为 PTS（视频）–PTS（音频），当 diff 值在正负 min 值内，允许播放。当 diff 值大于 max 值或小于 min 值，丢弃音视频。当 diff 值处于 min 值与 max 值之间，说明视频帧解码速度较快，音频帧还未跟上，暂时缓存在本地，当解码完成后再进行播放。



五、未来展望

1. 完善的 WEB 音视频播放能力

支持直播、录播；支持多种流媒体协议，如 FLV、HLS、WEBRTC 等；支持多种编码格式，如 H.264、H.265、VP9、AV1 等（AV1 是 VP9 的升级版本，由包括谷歌、微软、腾讯等大厂商主导的规范）；支持扩展选项，如截图、滤镜等。

2. 定制化能力

可根据使用场景选择、组合播放能力。抹平不同流媒体协议 API 之间的差异，开发者实现无缝切换播放方案。

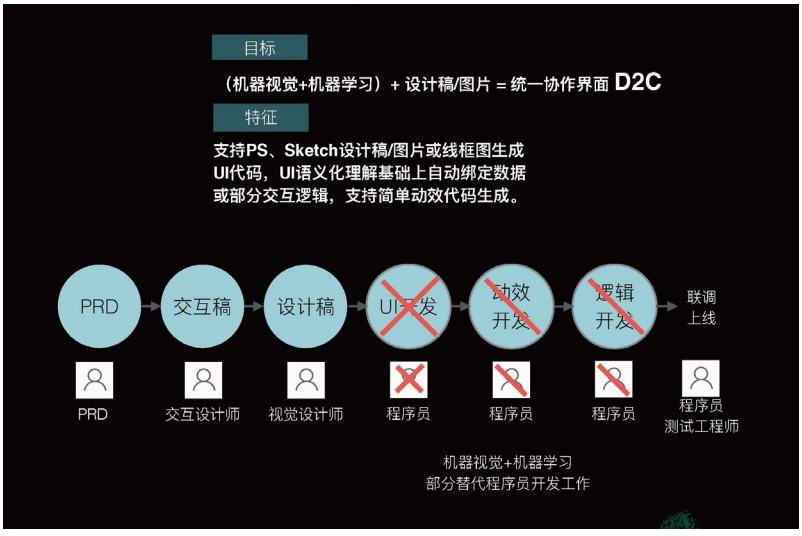
前端智能化

《前端智能化实践》——逻辑代码生成

作者：甄子

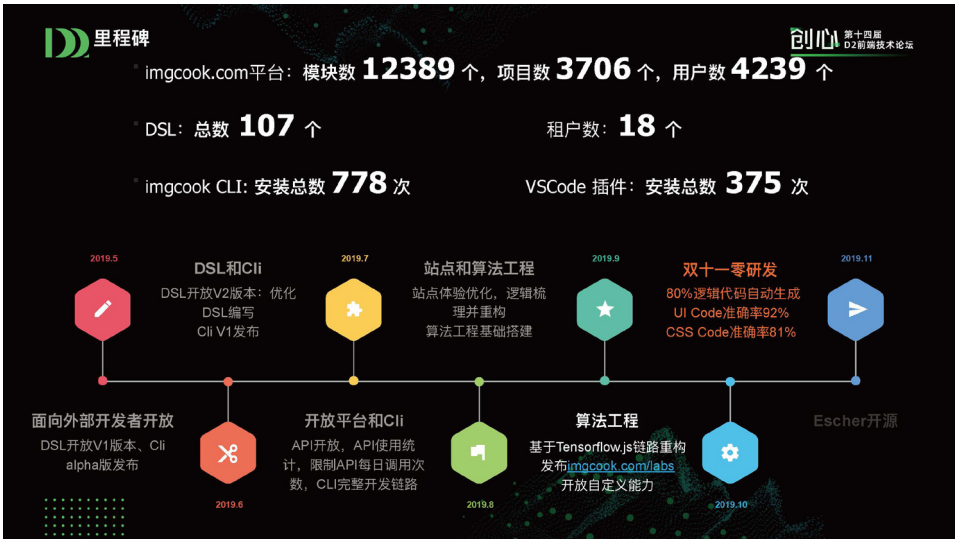
阿里巴巴高级前端技术专家甄子带来“《前端智能化实践》——逻辑代码生成”为题的演讲。前端智能领域到底该如何落地人工智能技术呢？本文从 UICode 到 LogicCode 有多远开始谈起，接着从页面结构和数据结构的视角分析，进而讲述了开发者赋予的自定义能力，最后对前端智能化的未来进行了展望。

到了机器学习时代，我们能做的事情非常多，业务压力也非常大，那么，如何提效把我们从日常重复性劳动中释放出来，让我们有机会尝试更多新技术，并用新技术改变业务？我们的初衷是想要通过机器学习提效，通过机器视觉 + 机器学习来理解设计稿，将设计稿转成代码，避免做一些人工的事情。我们希望能够替代 UI 开发，能够替代部分动效和逻辑开发。

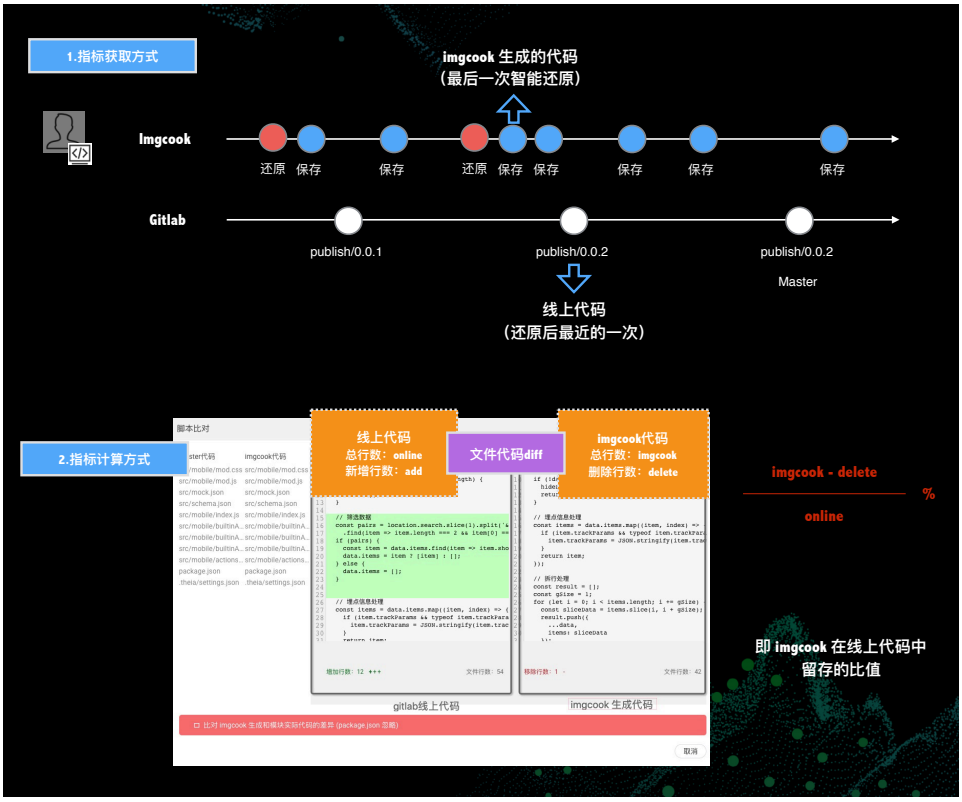


我们到底做的怎么样呢?

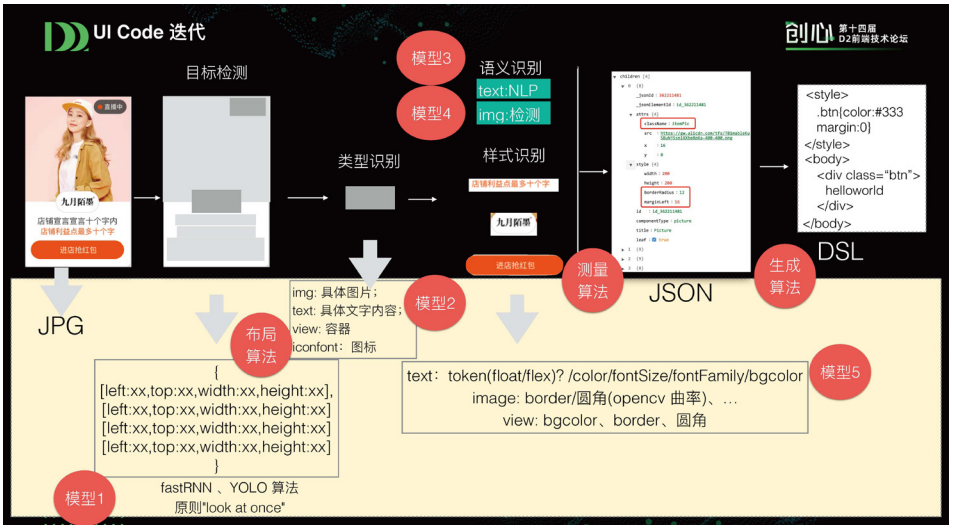
当我们真正涉及业务中如何识别设计稿，正确识别设计稿中 UI 元素并保证很高的还原度，尤其是后期如何在日常工作中做到工业级可用的技术水平。我们主要将模型精度及工程链路上工程工具的成熟度做了大幅度的提升，借助在阿里的天然优势，每年在 618、99 大促、双十一、双十二等节日都有技术大考，如何在大规模的业务诉求上，使用我们的技术真正做到提效?



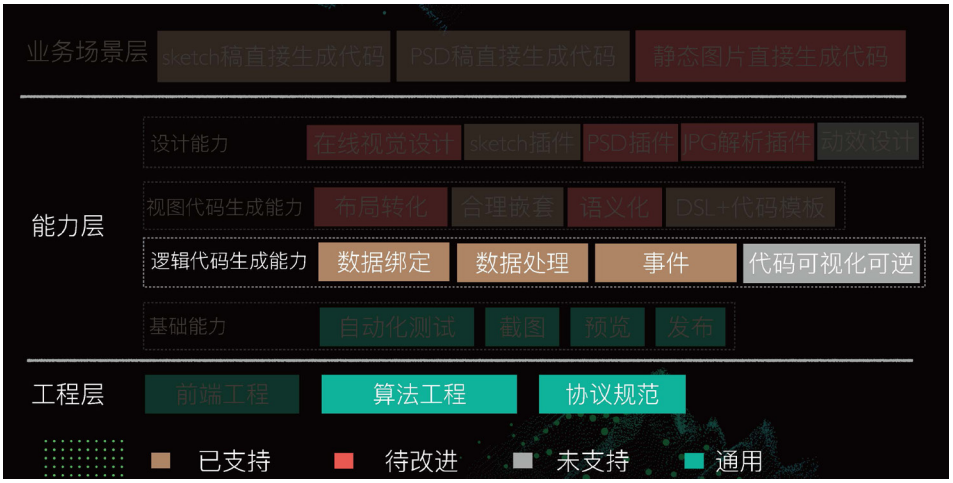
如图所示，图中数字每天都在不断的变化，我们的用户数已经突破 5000，模块数也已经突破 14000 个，其中的数字不只在阿里内部使用，我们也开放到行业内，大家利用这些工具确实能够带来生产力方面的变化。到 2019 年 9 月，我们也与 tensorflow.js 团队做了比较深入的交流，他们也非常兴奋 tensorflow.js 可以帮助前端工程师带来实实在在的价值。



我们定义了一个衡量标准，通过 imgcook 生成的代码最后有多少被开发人员删掉了，留下的代码有多少，用这样的方式衡量 imgcook 在整个工程链路中的提效程度到底有多少。该事件源于我们内部有一个零研发的战役，真正用一个新技术做到零研发，这对我们来说挑战很大，质疑声也非常多。



所谓千里之行始于足下，我们通过多模型链路不断提升每一块效果，包括计算机视觉、图像识别到算法识别，包括组件和控件的语义化、属性的测量，保证我们对未来生成的 CSS 及其它属性的准确性，再到最后 DSL 和 UIcode 生成的过程。



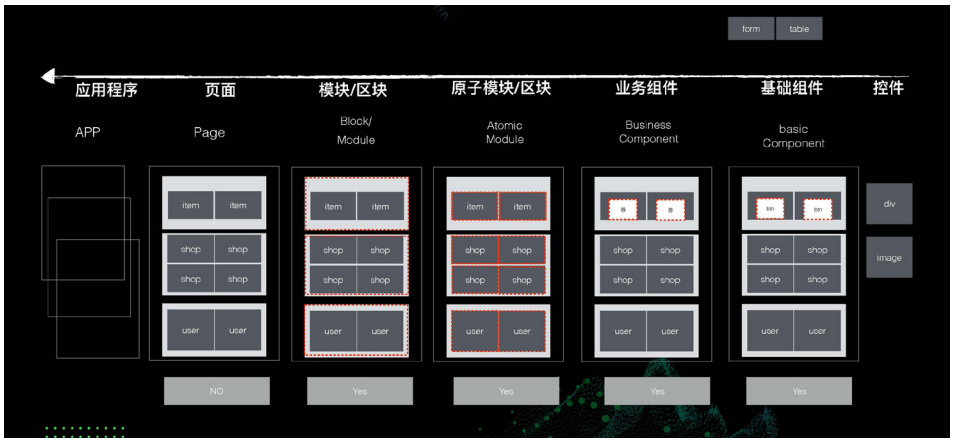
从设计稿到 UI 生成的技术栈如图所示，上层业务层接入的 DSL 应用到不同的技术框架，包括 react、vue、reactnative 甚至近期同学贡献的基于 flutter 的 DSL 应

用，中间层主要是 designtocode 的能力层，之前我们怎样把设计稿生成 UIcode，到今天我们怎样生成更多的逻辑代码，真正做到零研发，主要表现在几方面：一是相对来说比较简单的展示性业务都是数据驱动的 UI，怎样把数据绑定做好是很重要的一点，还有怎样生成更多常规的 UI 交互代码，而不需要我们每次生成一个模块或开发新模块时都要重复开发，我们希望整个代码生成都是可逆的，假如有一天对生成的代码进行二次编辑，我们可以将编辑结果和还原链路串在一起，如果设计稿进行了改动，也可以正确自动复用到代码上。在工程层，我们希望未来 designtocode 能力在算法和数据层面可以形成标准化，可以在特定领域彼此复用生成的算法模型和处理好的数据，使整个技术体系更快成熟起来。

UICode 到 LogicCode 有多远?

如何做逻辑代码生成? 这源于 BERT 模型，它特别像翻译的过程，比如我们基于技术对各种各样的语言进行语义化的分析和理解，最后通过语义关系的抽象形成的模式可以把任何一种语言翻译成目标语言，那么，这个过程是如何做到的?

语言都是由词、句、段、节、章、文、书组成，整个过程相当于把复杂的事情变得简单化，把里面存在的元信息抽取出来，然后对这些元数据信息，如果机器学习是一个皇冠，NLP 就是皇冠上的宝石，NLP 从搜索开始就对互联网产生非常大的价值，但是用户真正在 keyword 中输入的词并不一定是他们真实想要表达的诉求，后来的搜索引擎、问答系统、个人助理都会用到自然语言技术，比如空间上比较典型的 W2V，我们会把抽出来的元信息看作一个个 vector，vector 之间有很多的算法（如余弦相似度），可以探索用户真实诉求；还有语义化，我们定义的控件是一个 link，这个 link 可以代表一个点击跳转的链接地址，但真正在日常工程中实现的是基于 URL 路由的 trigger 作用，这时我们就要对 link 标签在上下文中的语义做理解，才能对 link 对当前 DSL 语义做准确判断；以及时间维度，比如当下语境说的事情和另外一个语境说的事情在语言表达上可能没有区别，但是因为所处语境不同，表达含义也可能不同。



我们首先要对控件进行语义化，控件到底是怎样组成基础组件的？在基础组件里，控件本身的语义以及控件所处的组件中 context 上下文语境是什么，紧接着是通过基础组件之间的关系怎么满足业务诉求将它们组合在一起，然后是将业务组件以模块化页面形式组合在一起，其中包括将元信息根据上下文信息以它本身语义做理解，在理解的前提下，我们才能知道应该如何去表达。

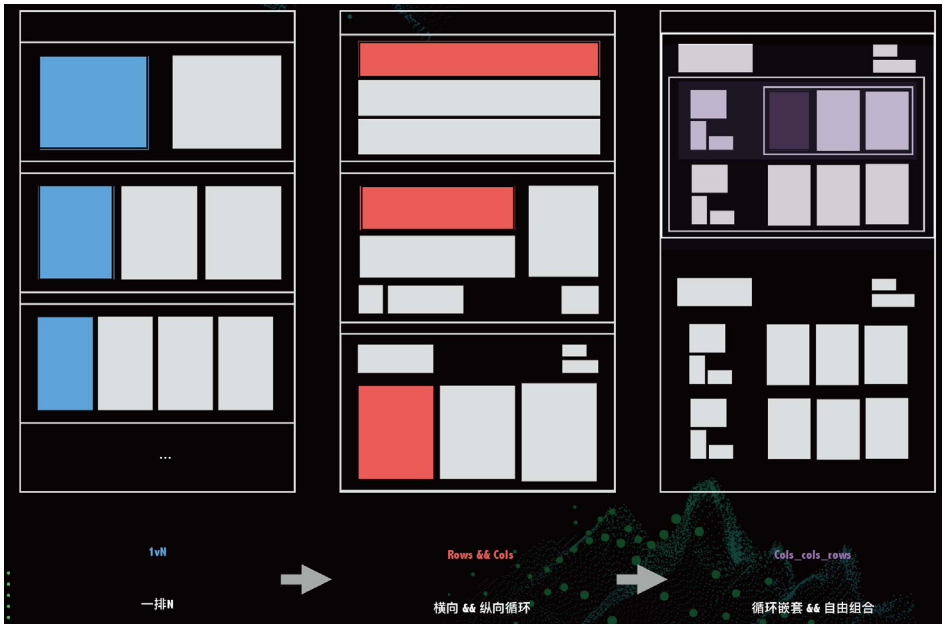
那么，这个过程就会变成通过设计稿、交互稿、线框图、需求文档的描述进行理解，再到生成逻辑代码的过程，它与以往的搭建和低代码开发是两个概念，以往的搭建和低代码开发是预先设定好能够实现的能力或功能是什么，然后以组件或控件模块形式组装好，让业务人员使用这些组件或者模块，但其中会有很多问题，比如业务人员是否具备程序员的软件工程思想，如果不具备，那么理解的成本会很高。我们希望未来可以变成相互之间解耦的事情，在需求理解方面可以独立做需求理解，当输入设计稿、需求文档、线框图、交互稿后，只需要知道产品或业务真实诉求是什么，在生成代码时，参考功能，从软件工程角度做过往的前端代码语义分析，最后知道怎么组织 API 和数据，用 javascript 等语言特性将它们组织在一起，然后生成这些功能和逻辑代码。

我们现在做的从识别到表达的解决方案整个过程如图所示，在识别部分，UI 及 UI 上的文本、后续对需求的结构化收集和线框图、交互稿的信息收集进行识别，识别之后需要有一个理解的过程，这个理解就像将一个语言翻译成另外一个语言一样，我们识别出背后的模式，模式可以在任何语言中进行复用，未来的表达就是解耦生成的过程，只要用语言的特性去组织代码即可。

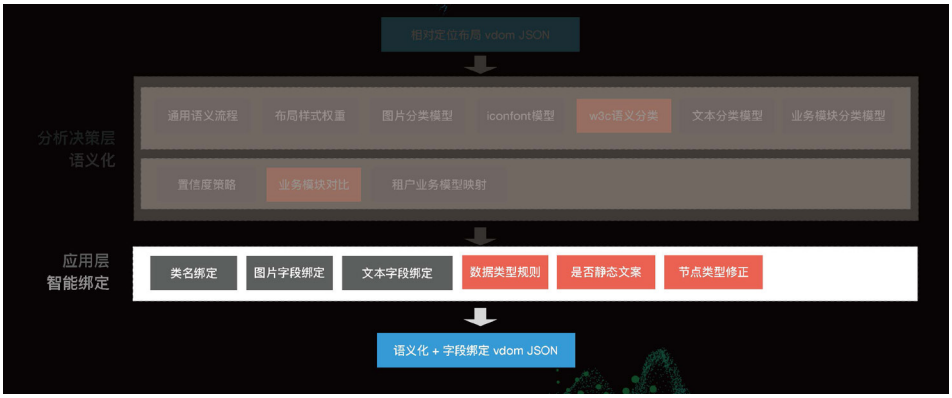
如图即是我们权益表达的一个模块，以软件工程视角看待此模块时，最开始为最上面左边第一个模块的样子，repeat 两次变成视觉稿展示的形式，最后实现模块代码时，就不会把该模块写三遍，而是写一遍并在布局时 repeat 两次。紧接着可以看到，在一个模块中包含权益，即到底可以从优惠券的交互操作上得到什么，背后代表一些限制条件和发放规则，加购的文案背后代表的是要实现一个加购的逻辑，我们希望有一些品牌的透出，所以要展示一些品牌的 logo，我们不会把这个 logo 写死，但是当一个产品或业务去思考这个问题时，他们一般不会考虑将 logo 写成动态去取的过程。还有进店，店名不仅仅是一个文案显示一个店名，更多的业务和产品的诉求希望能够通过优惠券产生一个进店的引导。将模块内部的元素全部都梳理出来之后，我们就能抽象出来整个逻辑，包括店名 / 进店、品牌 / 进品牌号、权益 / 条件、文案 / 加购、主视觉 / 主色调、背景 / 氛围，这些都会映射到表达的过程中，可能变成子业务

数据、业务数据、权益组件、基础样式、业务样式等。

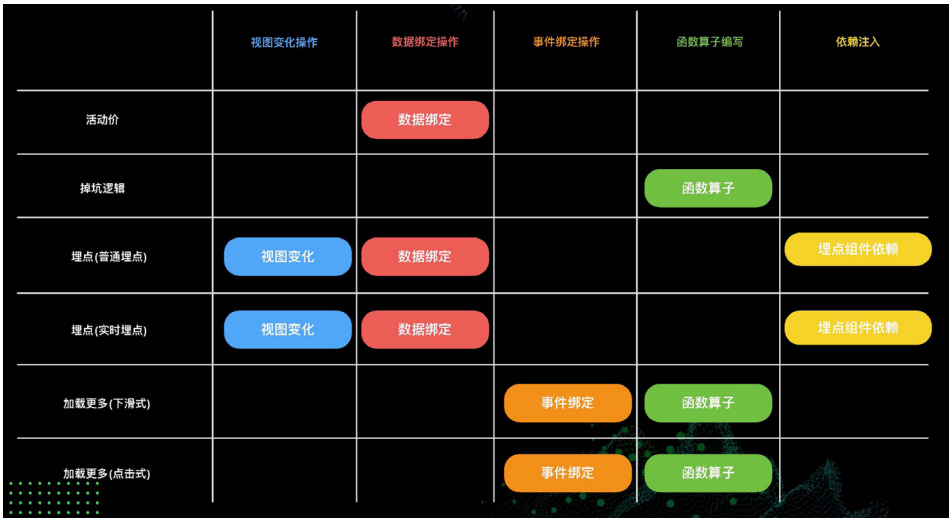
页面结构和数据结构的视角



大家可以看到，左边是一排 N 的模块，中间是横向 & 纵向循环，我们怎样识别出哪些是 row 哪些是 column，通过对设计稿和需求的语义化理解后就会知道哪些控件适配于哪些组件的，比如几个人在场景中转，到底哪些像素属于头，哪些像素属于身体，哪些像素属于胳膊，这就是语义化的分割，怎么能够将相同语义的元素聚合到一起，通过聚合到一起的元素找出哪些是非重复的元数据。



通过元数据再到以往生成的 scheme，找出到底组件上的数据 scheme 是什么，再把 scheme 上的数据绑定到类名、图片字段、文本字段上去，这其中涉及到如何做数据节点的识别、数据类型规则、文案检验和节点类型修正，通过准确的语义化，加上过往的 scheme 和 UI 控件属性之间的关系，能够做到自动进行数据字段的绑定。

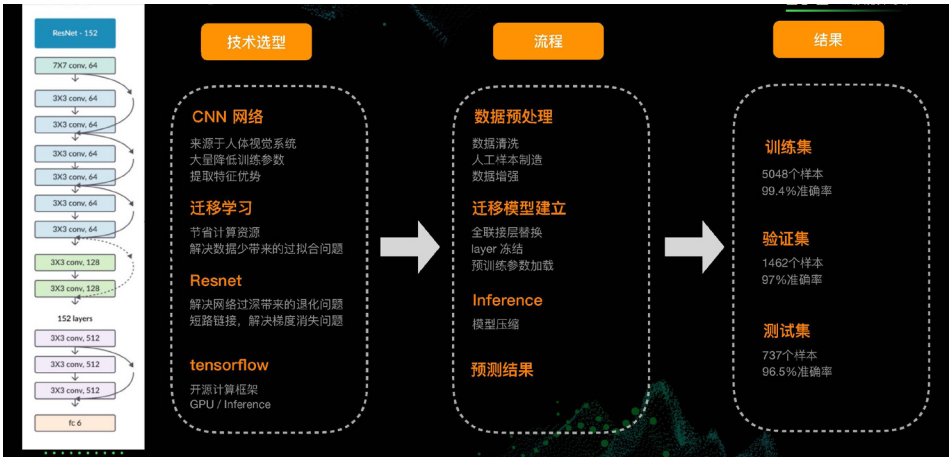


如图是我们抽象出来的一些逻辑点，当我们做到数据字段的绑定后，会发现日常整个的开发工作里还有大量的重复代码工作，比如上报一些数据、处理页面初始化时配置的读取、配置的下发以及当节点销毁后，页面元素发生变化后需要做的一些事

情，这些事情确实会有一些个性化的地方，但是个性化往往来源于对于输入参数的生成及对于输出数据如何去应用，这部分代码一般来说都是抽象的，都可以以一种模式提取出来，我们给它取名为逻辑点，对页面上的很多代码能够抽象成同一种模式或统一代码块的称为逻辑点，在这些逻辑点上把输入输出做语义化识别后得到字段属性，通过测量知道输入参数是什么，这样，我们就可以做到把平时常用且需要大量人工参与的开发变成通过识别到表达的方式，通过逻辑布局和属性的识别等，调用相应的逻辑点进行表达，这样的代码都是通过 imgcook 平台的语义识别和自然语言理解自动生成的。

因为我们背后的表达能力，我们生成依据的必要输入包括通用的语义流程、布局样式权重、图片分类、iconfont 模型、w3c 语义分类、文本分类、业务模型分类、配置策略等，我们发现，当我们去实现后续的逻辑字段绑定、代码生成的过程时，我们能够梳理出到底语义化给我们理解到什么程度，需要从设计稿、交互稿、需求文档中收集的信息和缺失的信息到底是什么，收集的信息和缺失的信息尽量用机器学习和自然语言理解的方式把它做自动化补全，但这个过程一定是有精度的损耗，这也是为什么到现在为止零研发只能覆盖到 80% 的日常模块开发中。

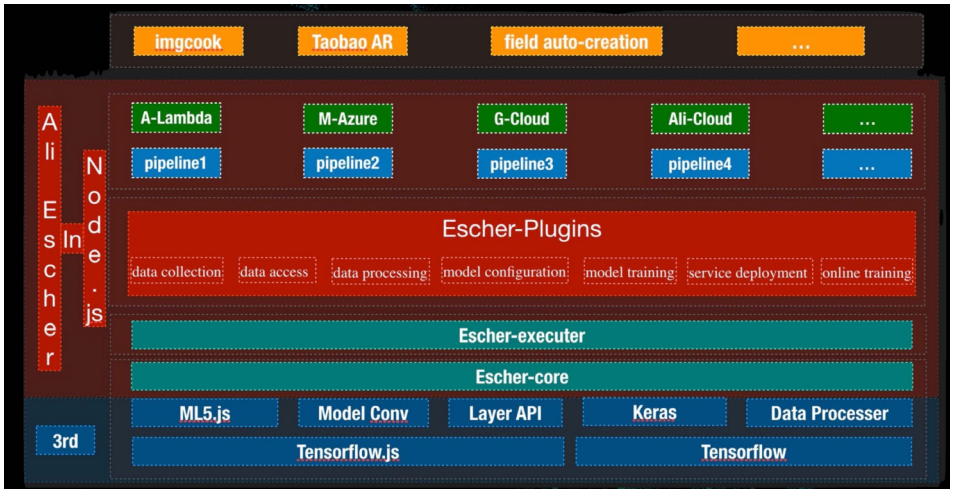
最后，我们回过头看前面这张图，左边是逻辑识别的协议，这个协议是将理解生成代码和后面流程串联在一起的交互界面，右边是逻辑表达协议，通过这个过程生成代码未来怎样才能复用更多的不同架构中去。



在整个过程中，我们使用到的机器学习能力包括迁移学习、卷积神经网络、Resnet、tensorflow 等，现在机器学习在图像识别和自然语言理解领域已经非常成熟，都有非常 general 的模型可以解决业务中常见的 80% ~ 90% 的问题，但是大家对这些能力的理解是欠缺的。因此，我们与 tensorflow.js 合作，未来希望能够降低大家使用技术的成本，便于大家快速进入到其中，快速了解技术能力的优缺点，以给自己做技术选型时候做支撑，还有使用技术时的流程是怎样的，和现有前端链路怎样做深度结合，来保证未来使用技术时成本更低、体验更好。

赋予开发者自定义的能力

如何能够更好的使用 tensorflow 和机器学习的能力？我们有很多环节已经被过往的先驱屏蔽了，比如对于创建模型来说，将这个模型实例化就可以调用了；添加分类和样本，前端代码从源码分析到生成，整个技术生态非常完备，因此我们可以自动化产生标注数据，而不像传统机器学习很痛苦，需要请一堆人员做样本数据，我们现在的样本大部分可以通过调用框架、读取 gitlab 上业务代码，通过 headless 技术用 poptear 渲染出来，渲染之前就会写清楚是什么模块什么组件，将来绑定数据是什么，整个技术体系非常完备。未来我们想训练自己的模型时，我们的优势是可以生成大量优质的标注数据来帮助我们描述定义的问题的重要特征。在训练模型部分，我们有 auto-machine-learning 技术，部署模型也有各种各样的工具，尤其是有了 tensorflowdistribution 的一些工具后，再加上现在可以在云端很多容器里做部署，都是很简单的。



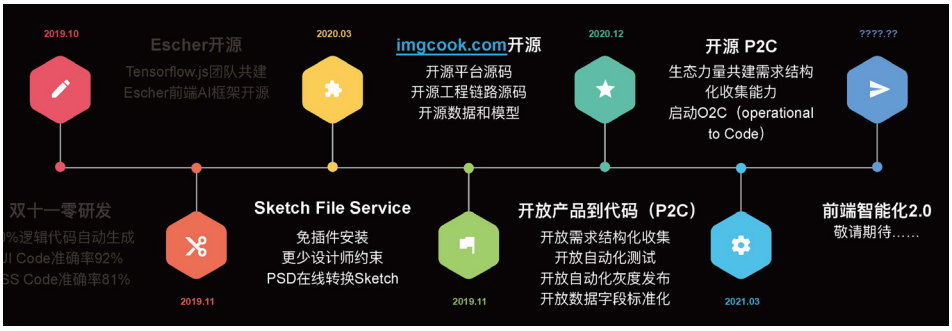
所以，我们与 tensorflow.js 技术团队一起合作，希望能够降低前端学习机器学习的成本，我们发明了 Escher 框架，Escher 想让前端开发人员零成本进入前端智能化领域去用机器学习能力解决业务上的问题，我们要保证未来大家在使用技术时能够跟现有的前端技术生态可以无缝连接在一起，比如现有技术生态已经有日志收集工具，这个日志收集工具收集的日志可以直接被 Escher 使用做数据清洗、数据增强到

模型中进行训练等。

在此非常激动的告诉大家，我们已经走完了 pipcook 的开源流程，欢迎大家加入到我们的开源项目中来，初期，我们只提供了为数不多的几条 pipeline，在 pipeline 中大家可以很低成本实现数据收集、模型训练到发布的全流程，未来我们希望能够通过数据和模型统一的标准，让大家可以在统一的技术生态里做技术促进。开源网址如下：

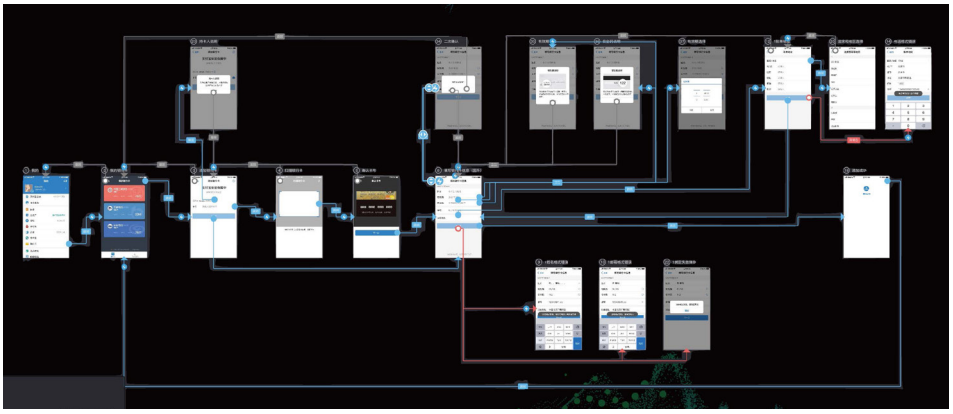
GitHub：https://github.com/alibaba/pipcook

前端智能化的未来

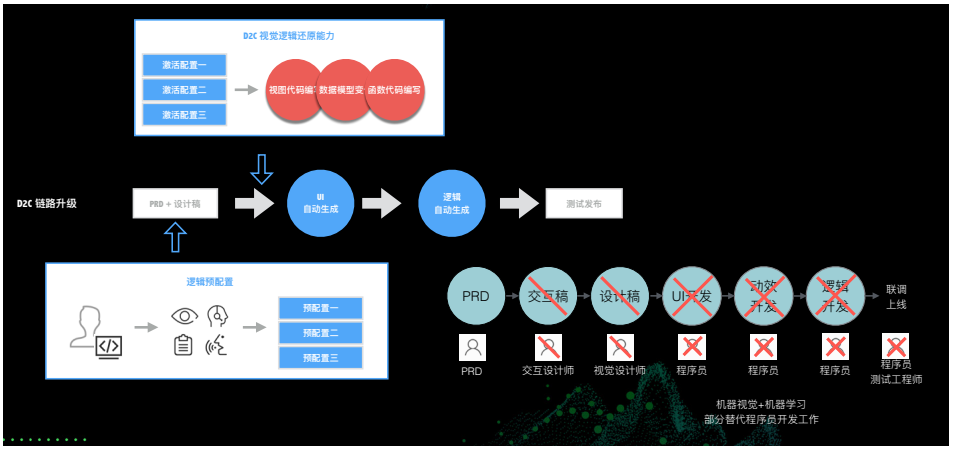


最后，我们对 designtocode 的规划如图所示，需求的结构化收集是我们后面比较重要的一点，在本身的 imgcook 链路里，体验相对来说不是特别好，要求大家安装 sketch 或者 Photoshop，还要再装插件，这些问题我们都在一点点解决，我们在下个月会 release 一个版本，用户不再需要安装 sketch 和插件了，只需要上传设计师给你的源文件，或提供一个 URL，我们就可以自动分析源文件，帮你生成代码，也不存在复杂粘贴的过程。只有当你认为代码在你的工程链路中无法满足你的需求时，需要自己二次编辑时，才有必要打开 web 编辑器。未来我们希望通过需求结构化收集进一步提升生成逻辑代码的准确度以及我们可以覆盖的业务场景，最后我们希望通过开源的方式把 imgcook 内部的模型能力和生成代码的质量进一步打磨好，争

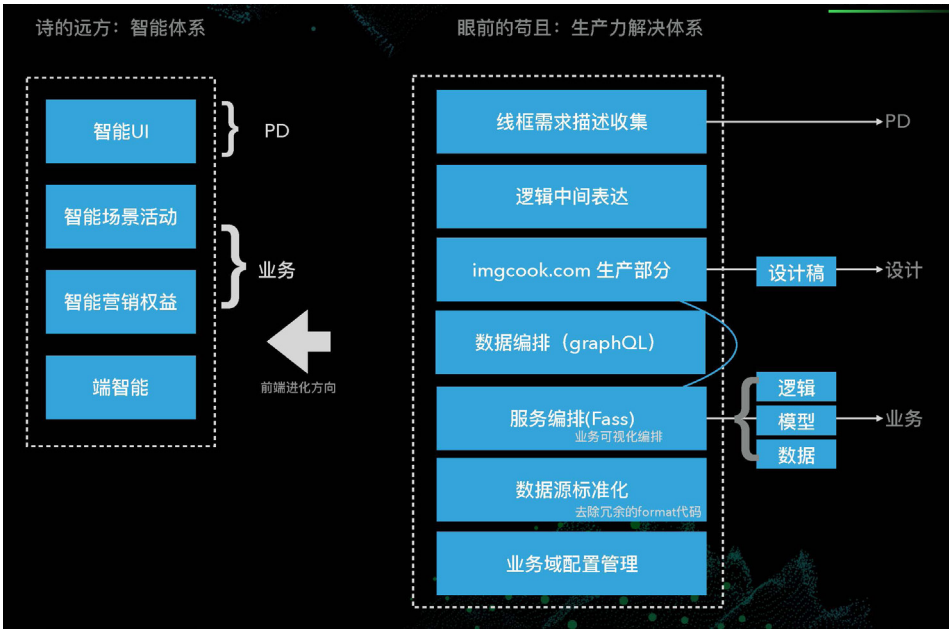
取在明年年底开源出来。



我们希望从以前单设计稿页面，变成更详细的收集需求，把我们生成逻辑代码能力覆盖到多页面的复杂交互场景中。



最终，我们希望可以完全替代动效的开发和逻辑开发，把这部分精力释放出来做更多有意义的事情。



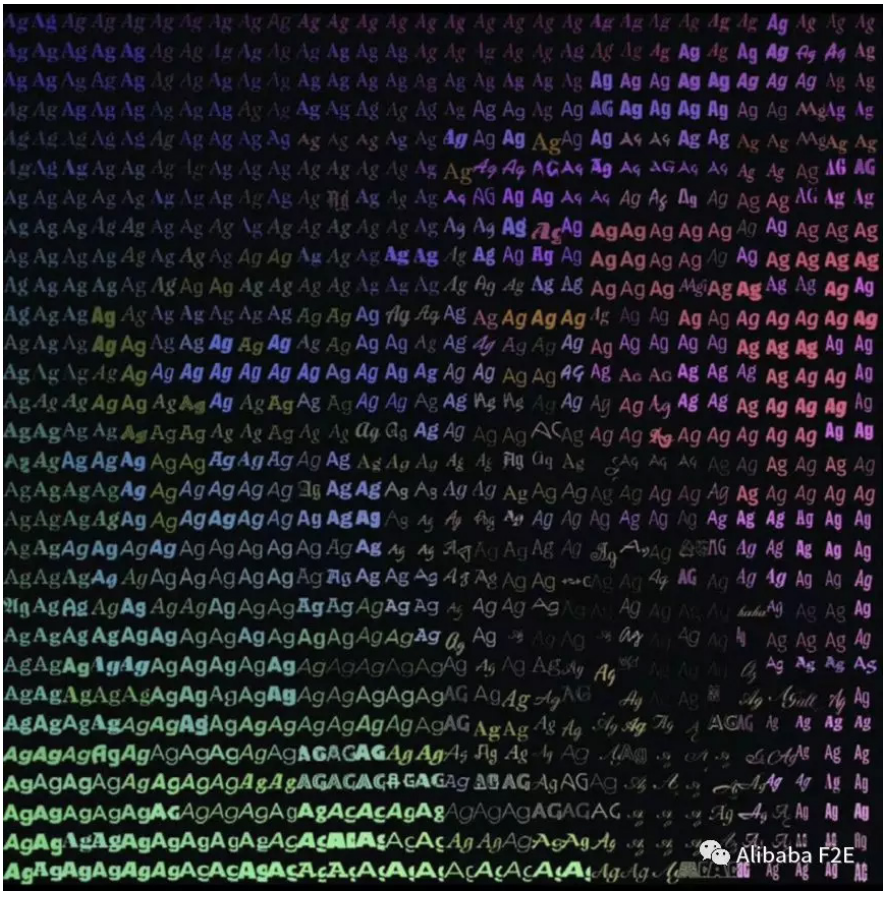
我们希望当下做的很多事情，未来可以找到更多的业务场景。

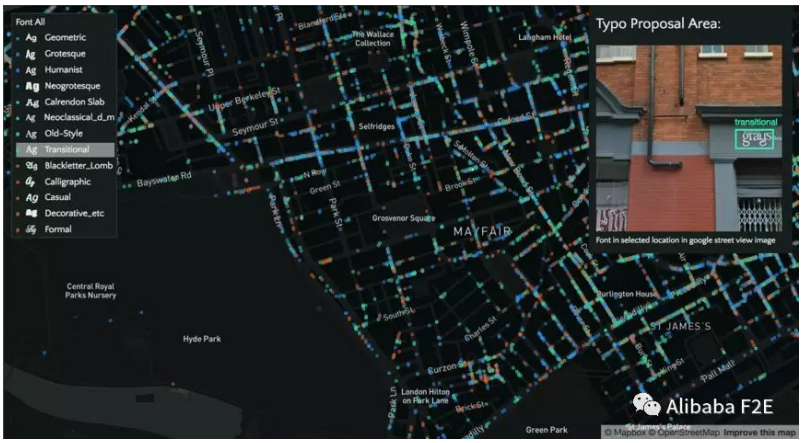
数据分析的人工智能画板—马良

作者：言顾

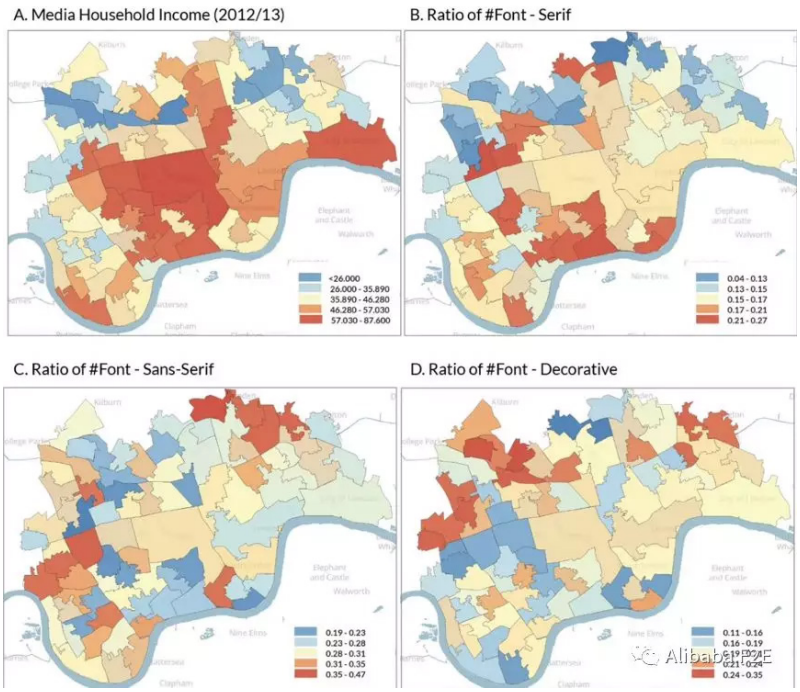
“马良”的诞生

DataV 一直致力于解决云上中小企业数据可视化的难题，也包括城市大脑的数据可视化业务。今天向大家介绍我们的新成果马良，我们希望将来做数据分析和调研时，仅仅通过手绘方式就能达到我们想要的数据分析规律。





如图为我们与麻省理工感知城市实验室合作的关于字体在城市中分布的研究，我们将城市中的常用字体进行分类，用谷歌的街景方式提取到街景中的字体和文字信息，我们也是用的物体识别模型来做这样一个提取方案，在我们提取到整个城市近百万张的街景之后，我们发现字体分布与街区的经济和商业都是有很强相关性的。



每一个不同的字体在区域内都和经济呈现正相关或者负相关，比如设计师经常用的无衬线字体与伦敦中心经济呈负相关，而衬线字体是正相关的，所以，高收入人群可能更倾向于选择带衬线字体，人们可能因为其它趋势选择无衬线字体。

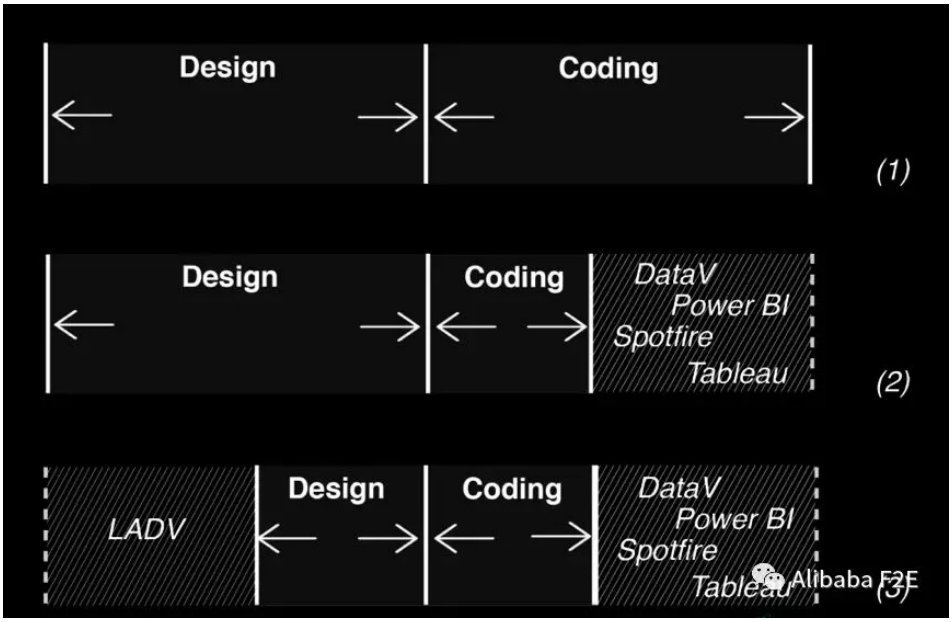
通过这个研究，希望让大家了解我们对可视化的看法，我们希望从设计的角度和思维去解决工程化的问题。

马良仅仅通过手绘稿就可以生成完整的数据可视化大屏，同时，我们支持设计稿的上传，取得的设计稿通过马良可以在秒级内生成供大家继续编辑的数据可视化大屏。

可视化界面搭建难题

在整个马良研发过程中遇到的最大问题是可视化界面的搭建，我们在搭建过程中确实遇到了许多问题，其中也涉及到了业务方需求，我们发现很多时候做数据可视化大屏搭建时，首先要考虑设计中原型的搭建和开发，但近年来，很多可视化产品都极大降低了开发门槛，在设计方面并没有很好的解决。

我们现在在解决数据可视化设计问题时，在大屏领域包括 DataV 或其它可视化产品，都是通过选择模板来降低可视化设计的高开发门槛。在提供模板时，会遇到一些问题，比如自己做数据可视化大屏时，没有办法完全匹配自己的数据维度，数据可视化模板数量的限制导致没有办法选取更多样的风格。为了解决这样的问题，我们研发了马良这样的产品。

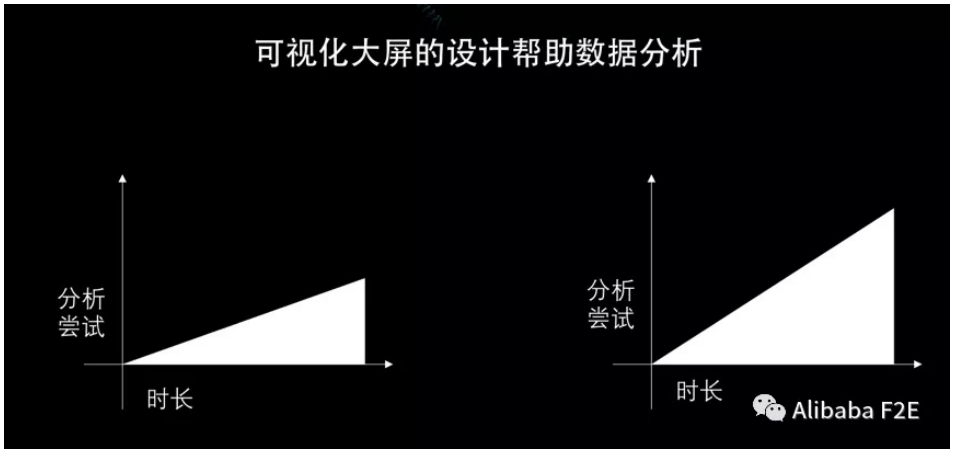


我们对近年来的数据可视化产品进行了相关分析，我们发现，可视化大屏创建包括了设计和研发，而很多可视化产品都在 coding 方面极大降低了开发成本，我们希望马良在开发和设计方面能够做好。

可视化大屏模板的设计流程

首先，可视化大屏模板需要有产品经理帮助我们梳理数据，同时设计师需要对梳理好的数据进行模板布局，包括颜色选取，最终交由工程师完成产品落地，可以看到整个流程非常冗长，需要不同角色配合。有了马良之后，任何单一角色不需要依赖于

其他角色，都能够完成整个数据可视化大屏的搭建，比如产品经理不需要有设计知识也可完成搭建，设计师也可用草稿搭建数据可视化大屏。而且，马良对数据规律可以进行真正意义上的探索。



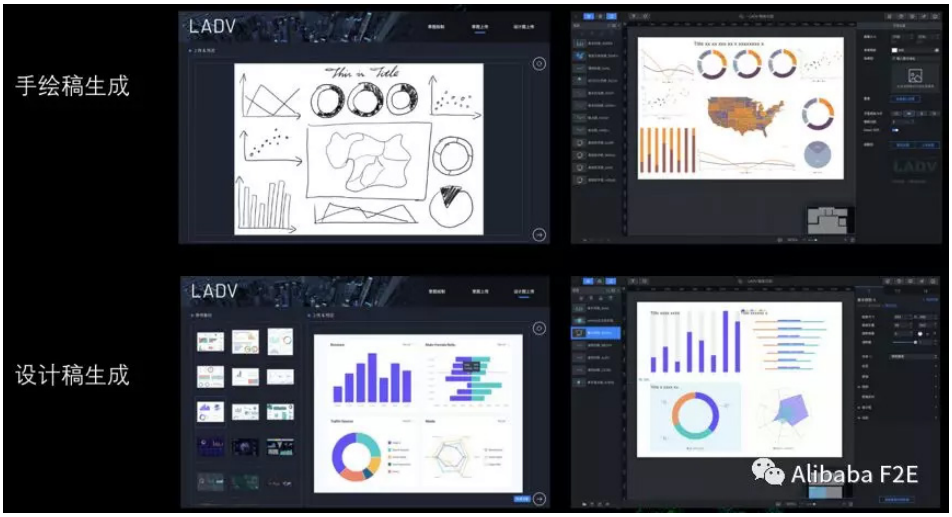
如图，右边为马良的时长，同样时长下我们相信有更多分析尝试，以前做一块大屏需要几天，如果大家用过 DataV 就会知道，DataV 是支持拖拽的，可以选择图表组件进行拖拽，大概几小时完成。

为什么要推出马良？因为很多用户没有很好的设计背景和专业知识，同样只是简单的拖拽图表组件和布局，普通用户和设计师做的是完全不一样的，我们希望当发现很好的案例时，可以直接拖下来用马良生成，完成数据风格的迁移，而不是再拖拽，之前做过相应的测试，发现拖拽的创建过程还是要以小时计算的。而马良是以秒级来计算流程的。

基于深度学习辅助大屏可视化设计的方法

随着深度学习的发展，尽管深度学习与可视化有很强的沟通，但是一直没有很完善的产品让我们理解深度学习和可视化二者如何互相帮助的，我们知道一些比较知名的可视化与深度学习结合的案例如 google 用可视化来可视整个深度学习的神

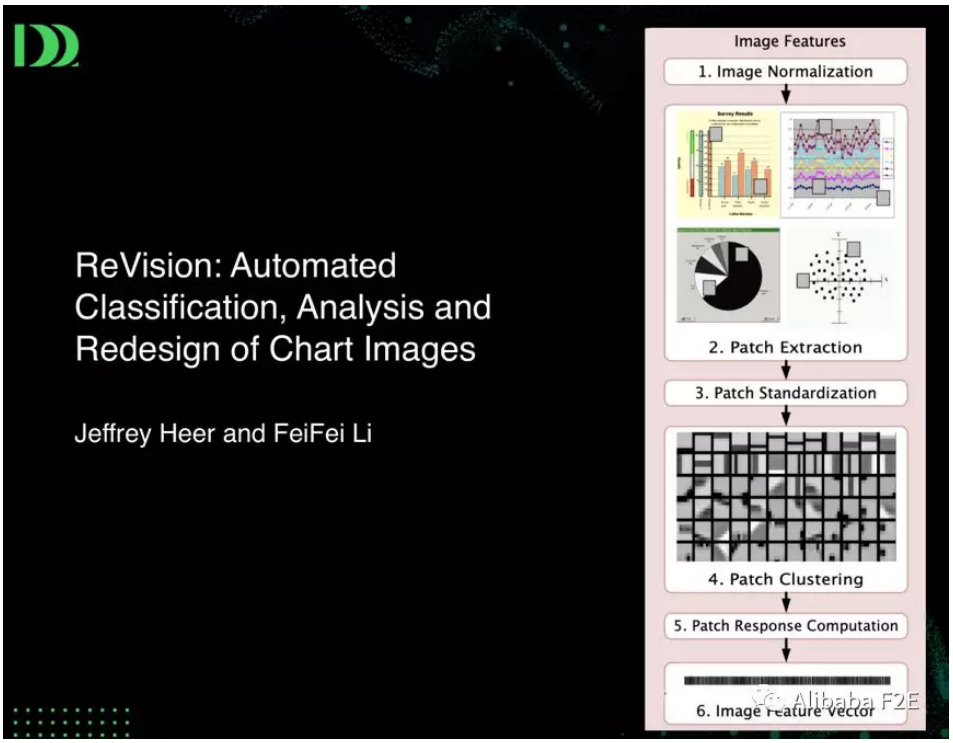
经网络，同样，我们相信可以通过深度学习及物体识别技术等能够帮助可视化进行搭建和还原。



如图为马良两个主要案例，上面为手绘稿生成，下面为设计稿生成，以应对生产环境中的两种需求。生成手绘稿是因为很多产品经理做数据可视化大屏时，可能更倾向于把想法概念绘制下来，这样当有新想法时更容易移除或擦除，有了马良之后，可以依据手绘稿生成继续支持开发的可视化大屏模板；同时我们也支持设计稿，很多时候产品经理拿着网上找到的设计稿可视化大屏与工程师沟通，希望工程师按照某种风格开发，有了马良之后，自己找到的设计稿可以很快实现数据可视化大屏。这样，单一角色可以完成搭建数据可视化大屏工作，同时，如果后续有更多个性化需求，更多人员接入，整个效果会有更好的提升。

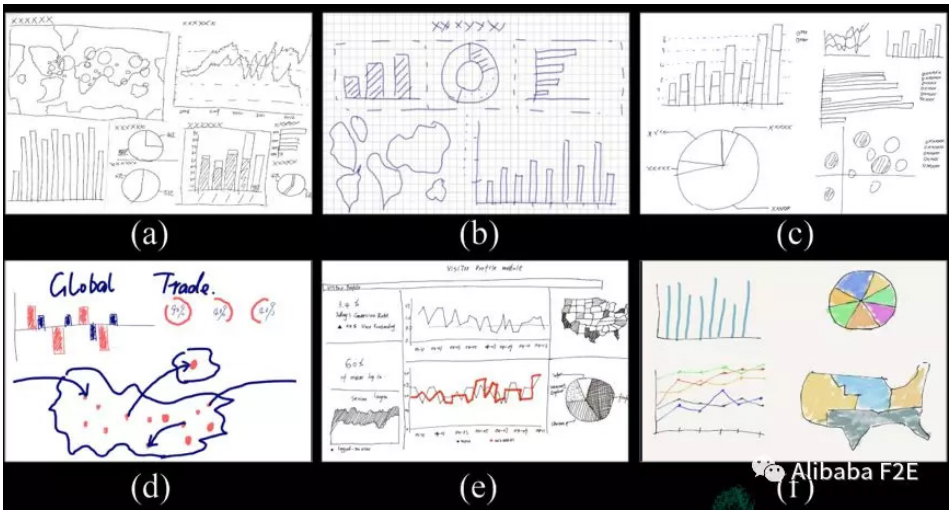
图表识别与可视化界面识别的区别

整个技术内部最重要的是图表位置提取和生成可视化界面布局优化。在可视化领域，图表识别与可视化界面识别的区别在于，可视化界面识别不仅要识别图表是什么，同时要知道你的位置在哪里。

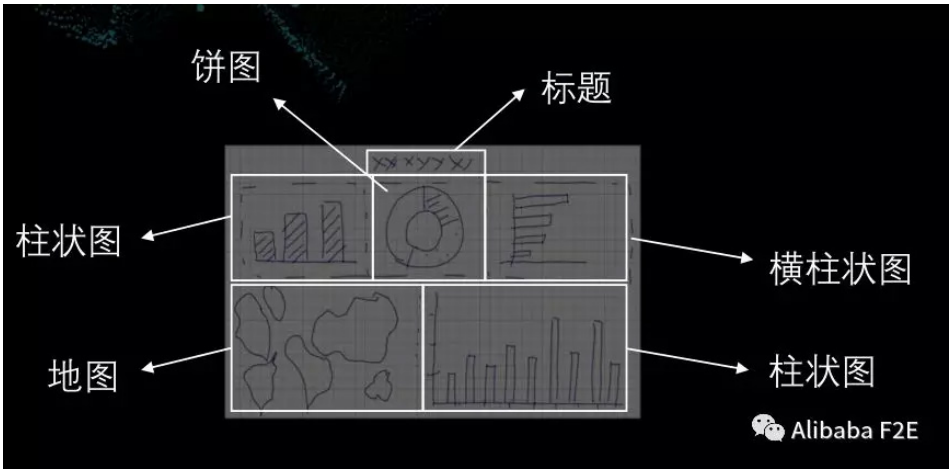


图表领域最原始的深度学习研究是斯坦福教授 FeiFei Li 和 Jeffrey Heer 发表的这篇文章，ReVision 是把对应图表的特征提取出来，提取出来的便于机器理解的特征进行一个全链接层的映射，得到结果图形，但是并没有解决物体识别和位置识别。

因此，我们引入了物体识别模型如 Faster R-CNN 等，物体识别不仅知道你的物体在哪里，这样的模型应用于交通领域比较多，比如行人、车辆等，去年开始，我们与浙大在图表识别领域进行合作，得到了不错的效果。



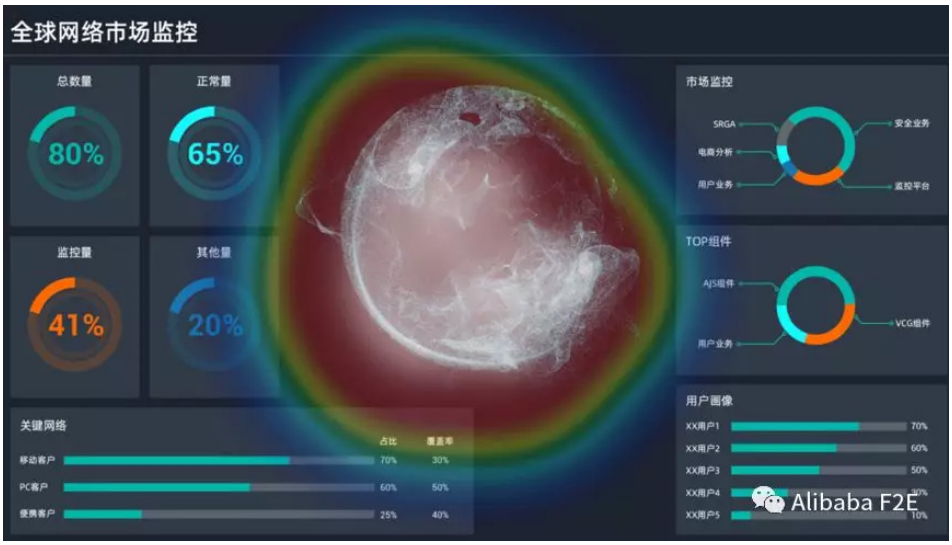
深度学习和机器学习需要非常大的训练样本，我们在做相关的训练样本收集时，为了让我们的模型识别不同的风格，尽量让更多的参与者去参与，能够识别不同的风格。



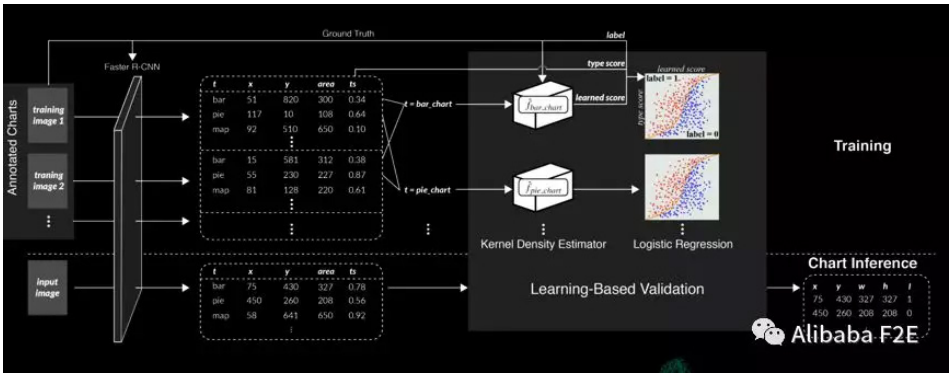
如图就是一个训练样本，一个参与者绘制完不同的图表之后，对图表进行打标。为什么没有用随机的生成方式呢？因为让许多参与者参与需要大量的时间，能不能用随机的方式，比如有了图表之后，更换布局在一个空白的界面上随机放置，只要不重叠情况下就可以作为训练数据。我们最终没有使用这个方法，而是选择了 DataV 自己的数据，是因为我们相信可视化图表在界面中是有一定规律的，这也引入了我们对物体识别模型更多的算法和函数的思考和提高。

可视化界面的概率密度函数

可视化界面内部也有自己的概率密度，比如如图所示的大屏，title 所在的位置一般都在图表的上半部分，地球 map 一般都会在屏幕中间，我们也是通过自己已有的数据进行概率密度的匹配和测试。



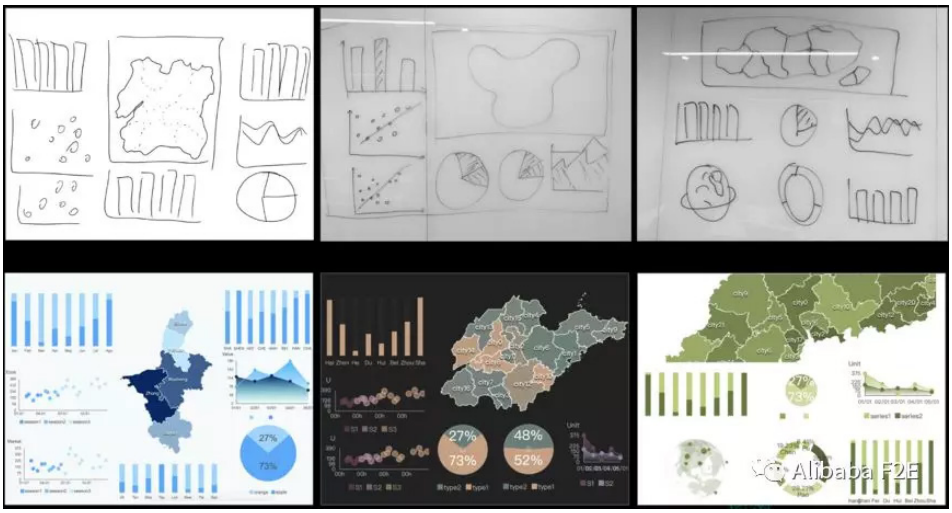
我们发现大部分的图表确实是有概率存在的，比如一些柱状图如果是横向柱状图，因为 Y 轴在图表左侧，所以横向柱状图更倾向于在整张大屏的左侧。



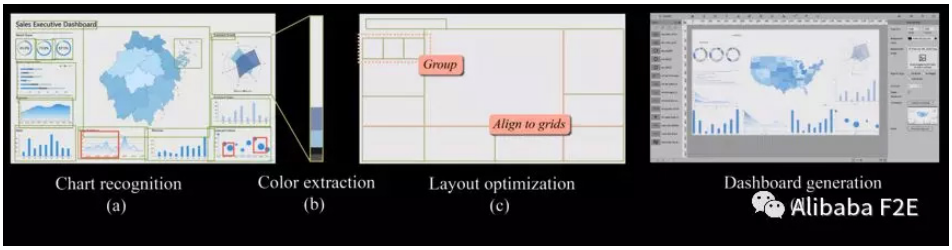
我们可以对深度学习模型做更高的优化。如图左侧是正常的深度学习模型，包括扔进图片之后经过训练得到一个结果，扔进一个数据可视化大屏进去得到具体信息，比如 bar chart 在屏幕什么位置，大小是什么，同时在这时进行概率密度的计算，我们会计算 bar chart 在某个位置的概率大概是多少，这就引入了后半部分 Learning-Based Validation 概率密度函数，引入此函数对整个模型准确率提升

6% ~ 7% 左右。

返回到图中，当我知道在某处有一个 bar chart 或 pie chart 时，对 bar chart 或 pie chart 在这个位置的概率再进行一次计算，如果概率非常低，我就认为识别到的是错误的，相当于先有识别结果，但同时对识别结果再进行一次验证，这样对整个模型优化是非常大的。

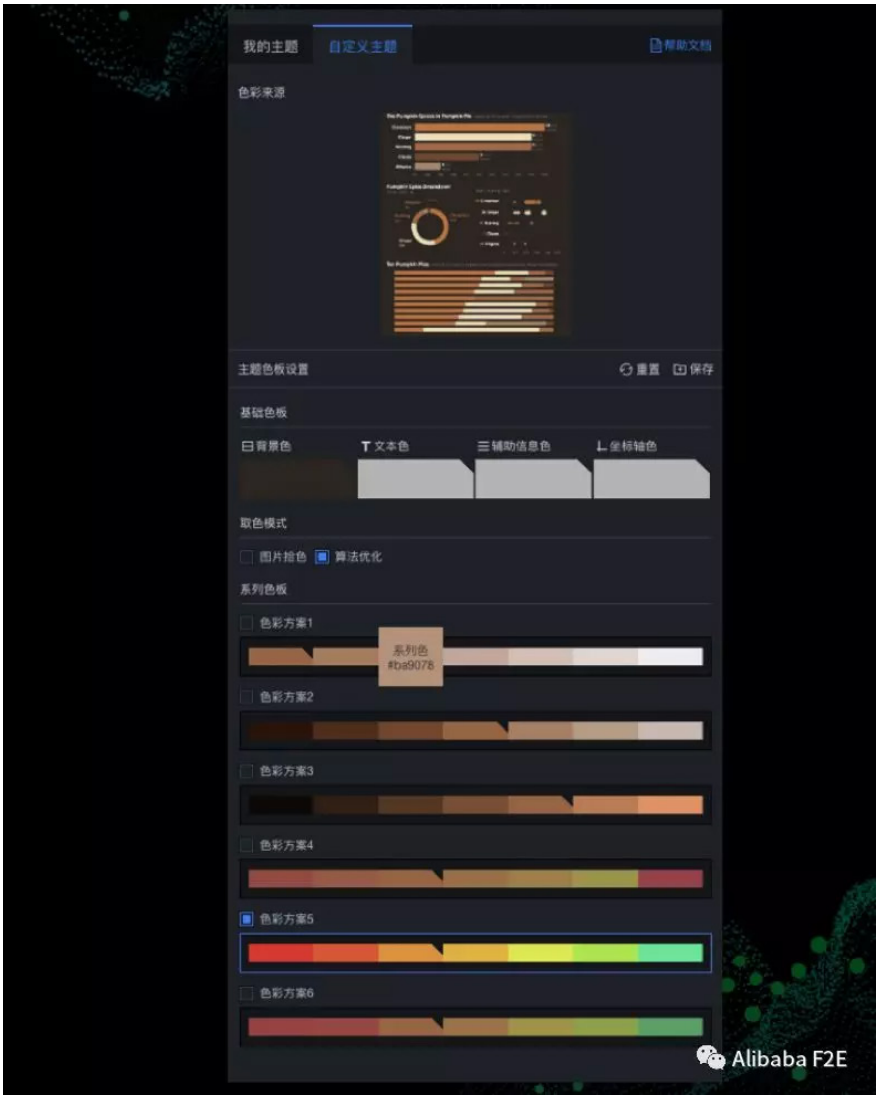


同时我们也希望我们的产品不止能够识别电子版上的业务场景，同时也能够识别包括会议室内的数据分析，草图绘制也上传到马良，马良会在秒级内回复一个编辑好的数据可视化大屏。如果之前你有数据已经导入的情况下，可以很快地得到一个非常完善的可视化大屏。



无论是手绘稿识别还是设计稿识别，马良都遵循着如图几个步骤。首先会有一个图表识别，接下来会有一个颜色提取部分，第三步会被之前识别到的位置结果进行优化，最后基于之前收集到的数据可视化信息，生成一幅已选择的设计风格的数据可视化大屏。

主题色提取及赋予生成可视化界面的颜色

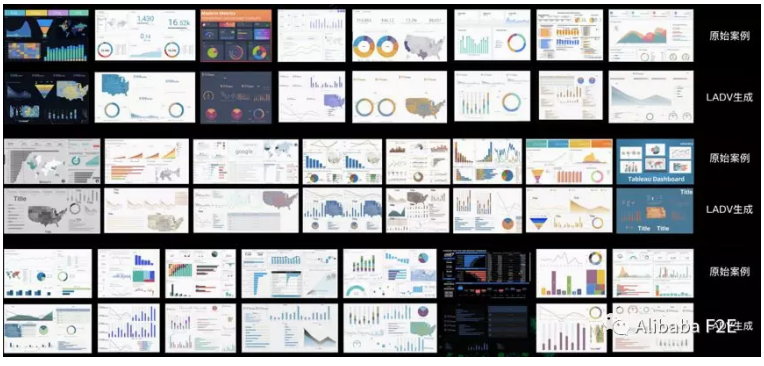


可视化界面颜色生成部分，我们做了主题色提取和色板生成。首先要提取背景色和主题色，同时依据背景色也会推荐字体颜色和其它辅助颜色，我们也会推荐不同的色板方案供大家参考。

有时，我们喜欢的图片不一定是数据可视化作品，可能只是一个音乐会或自然风光等不同场景，用这样的图片来生成数据可视化颜色，对此，我们也做了相应的工作，只要你上传任何图片，我们能够把图片风格的颜色设置到大屏中。

以上给大家介绍的更多的是依图生图的功能，我们现在也在做数据生图和风格迁移，底层使用的模型算法包括深度学习算法、机器学习算法和基础算法等，通过识别、回归和概率计算帮助上层的信息搭建。

图中彩色部分是马良现在做的部分，用户可以上传自己的可视化屏幕进行识别，整个识别是依据马良现有的模型，我们也在做马良的自动机器学习，使用户可以上传自己的图表，数据打标非常痛苦，可能上万张图片，马良作为迁移学习的模型后，相信大家可以只上传几十张上百张的图表就可以更针对的识别属于自己的数据可视化图表库。



我们从 google 上搜索不同的 dashboard，随机的选取结果后拖下来由马良来生成，上面为原始数据可视化大屏，下面是由马良生成的数据可视化大屏。通过用户的反馈，我们了解到有一些生成的颜色和布局是优于原始可视化大屏的，这样的结果是激励我们前进。

现有场景中，很多厂商因为没有自己的能力，没有自己的专业的数据可视化设计团队，导致很多厂商的数据可视化从来不换。有了马良之后，我们希望至少快速帮助用户测试是否能够生成一个更好的更美的更适合的数据可视化模板。

微前端

云前端新物种 – 微前端体系

作者：克军

随着云时代拉开帷幕，前端迎来机遇和挑战。在 2016 年，微前端的概念被提出，前后端架构形成了微前端 + 微服务的模式。微前端是云时代的前端架构体系，它不止是一个框架或者组件。阿里云智能安全前端团队负责人克军将从前端架构的演变、微前端的价值、微前端与云生态的关系、以及微前端的工作原理等几大方面为大家详细介绍云前端新物种 – 微前端体系。

嘉宾：克军，阿里云智能安全前端团队负责人。长期专注于前端体系的探索，目前带领团队探索云上安全能力的交互创新。

本次分享将主要围绕以下五个方面展开：

一、前端架构背景介绍

二、微前端价值

三、微前端与云生态

四、微前端工作原理

五、总结

一、前端架构背景介绍

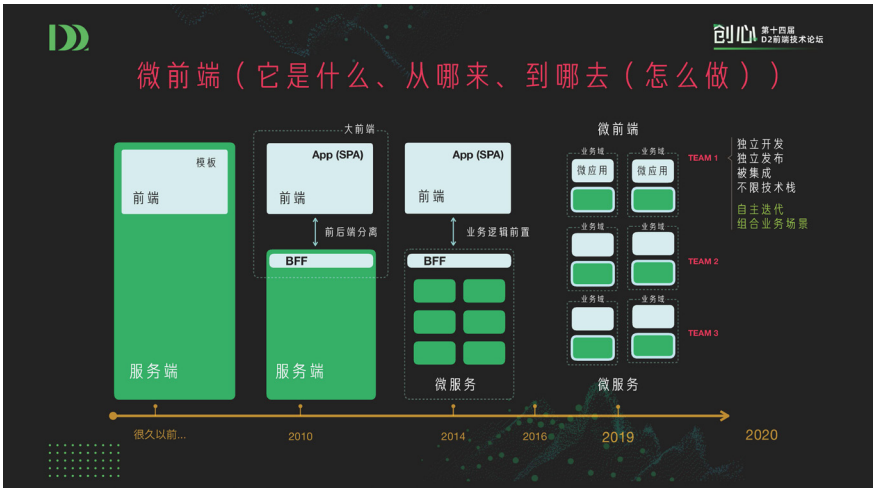
1. 前端架构体系

云原生应用的三个特征包括容器化、动态管理和面向微服务。其中缺少对前端应用的特征描述，而本文将介绍的微前端正是代表了这个特征。前端架构需要建立一个

可以满足业务当前及未来的前端技术体系，其中包含了软件库、框架、开发流程、技术栈设计、以及管理工具的建设。前端体系的建设是一个可持续的事情。在目前的云时代，微前端是目前的前端架构体系的方向。

2. 架构的演变

在很久以前，前端和后端加在一起构成 MVC 架构，前端只是后端渲染的模版。在 2010 年，提出前后端分离的概念，服务端 BFF 作为数据的网段。到 2014 年，微服务概念被提出，后端的架构开始转型，面向微服务，而前端架构自 2010 年到目前依然没有变化。随着云时代拉开帷幕，前端或许会迎来机遇和挑战。在 2016 年，微前端的概念被提出，到目前完整的前后端架构是微前端 + 微服务。微前端和微服务的拆分是按照业务域进行拆分，每个业务域持续化的组织架构，每一个 team 维护单独的业务域。每一个业务域可独立开发、独立发布、可以被集成到环境当中、不限技术栈、被拆开的业务域可以自由组合。如果可以做到这些，基本上符合云时代分布式开发前后端保持一致的理念。



二、微前端体系价值

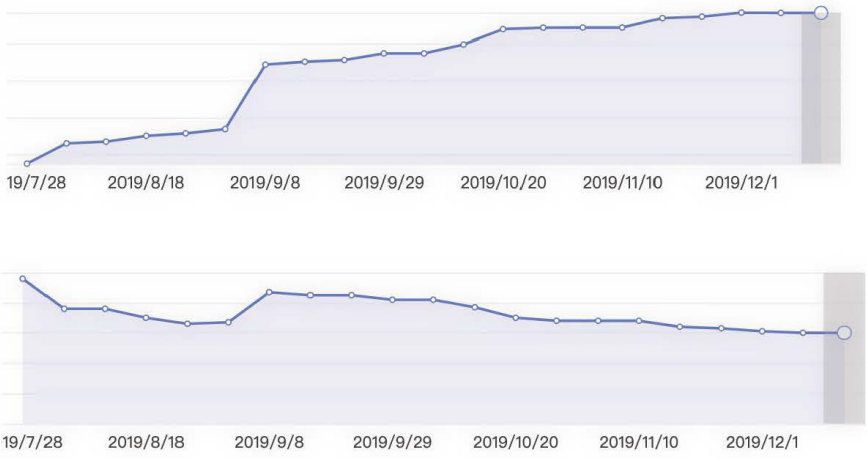
当然，大家对于新事物需要保持不吹捧的态度。在构建微前端体系过程中，需要

克服微前端可能带来的问题。那么微前端与 iFrame、WebComponents、NPM 包、BigPepe、插件有什么区别?

1. 微前端与 Widget/ 业务组件的区别

首先，微前端是一个架构体系，用于实现大型 Web 应用。Widget 组件是以库、外联 npm 包等方式实现复用。微前端带来的是生产方式的不同，给开发方式带来了本质的变化，而 Widget 组件是生产工具，目的是提高复用。此外，微前端通过隔离机制实现技术栈无关，而 Widget 组件没有考虑这个问题，还是需要人工解决依赖和冲突问题。微前端可以单独构建、单独发布、热升级，这是微前端最大的优点。如页面中的某一个区域无需构建整个大应用，只需构建局部，这与日程业务中的场景非常契合，往往只是某个子域需要不断的迭代。而传统的 Widget 组件方法中，每次改动一点都需要整体发布，不够灵活。同时，微前端体制化治理性非常强，与云应用中动态化管理相似，本质上是对微服务的编排和调度机制。任何分布式特点的系统中，体制化治理都非常重要。因为当业务被拆成多个碎片化的部分后，管理的难度大大提高，体制化治理环节正是用于解决管理难度的问题。此外，微前端采用路由映射，消息机制符合主从关系，而 Widget 则相互之间并不有关系。微前端体系还衍生了微应用的概念，即属于整个产品的子集，变化较快，整个应用是若干个微应用的组合。Widget 组件具备“外挂”属性，变化小，抽象出了整个应用的通用功能。

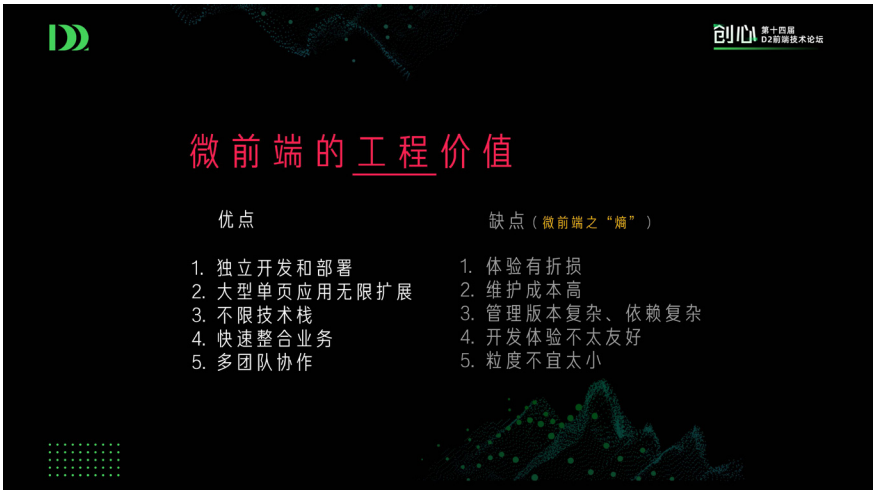
从 2019 年 7 月份到 12 月，阿里云智能安全团队代码变化如下图，代码增长的幅度达 10 万行。很明显，随着代码行的增加，其可维护性呈下降趋势。架构的思路是分层、分治和抽象，而微前端正是借助了分治的思路。



2. 微前端的工程价值

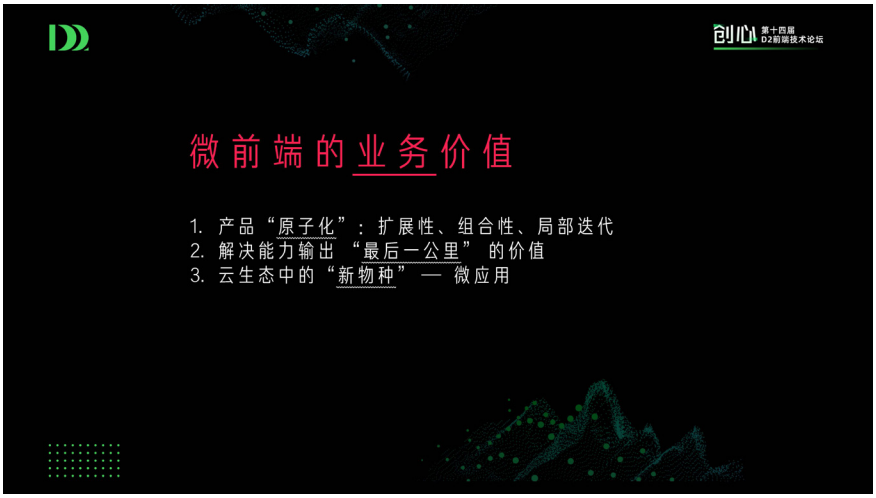
微前端的优点：微前端的优点包括独立开发、独立部署。对于大型的单页应用，采用微前端架构方式可无限扩展，其复杂度不会很明显的增长。此外，微前端不限技术栈，借助隔离机制保证技术栈的隔离性。第四点，可快速整合业务，如 toB 的产品需要符合客户的定制化需求，每个客户的业务场景都互不相同，此时需要产品具备更灵活的特点符合客户的业务定制需求。最后，微前端可以多人协作。

微前端的缺点：当然，微前端的缺点也很明显。首先，微前端会导致体验折损，微前端每一步都是异步加载，中间会出现不流畅的问题。此外，当一个单体应用被拆成若干个，其维护成本也相应增加，如如何管理多个版本，如何复用公共组件等，导致管理版本变得复杂，依赖关系也极其复杂。还有，如果应用拆分的粒度过小，对于工程师的开发体验也会不太友好，如果工程师负责的需求跨多个业务域，此时他 / 她需要与多个团队合作，沟通成本大大增加。



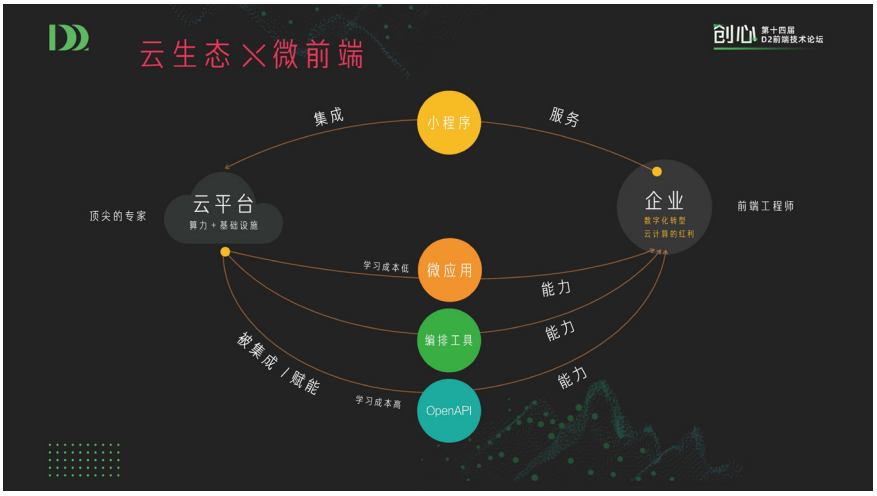
3. 微前端的业务价值

如果微前端只具备工程价值，如提高效率，则对团队内部没有多少帮助。因此，微前端还需具备三个业务价值。首先是产品“原子化”，可以根据客户的使用场景，可自由的编排组合，即扩展性、组合性和局部迭代的特点。其次是解决能力输出“最后一公里”的价值，云时代的平台与传统的平台不同，云平台更注重赋能，即能力的输出。最后，在能力输出过程中，需要将能力变得组件化，就是微前端中微应用的概念，降低用户使用的难度。



三、微前端与云生态

云平台提供基础的设施和算力，云平台可以以 OpenAPI 的方式被消费，此时要求企业具备研发能力较强的团队。在未来，云平台还需要提供编排工具，提供更简单组合的使用方式。再进一步，通过微应用的方式，使得企业的使用成本降低。



Idea1: 云时代需要一个企业级的开发套件，包含了 OpenAPI、编排工具、UI 组件库以及微应用。

Idea2: 在 2014 年，已经出现了微应用市场的概念，但当时还没有很流行。随着云时代的演进，开放的微应用市场应该是目前较为适合的方向。阿里巴巴内部很多团队已经开始探索这个方向，如 single-spa 是最早是微前端框架，MicroX 是阿里巴巴云智能安全团队的微前端框架。

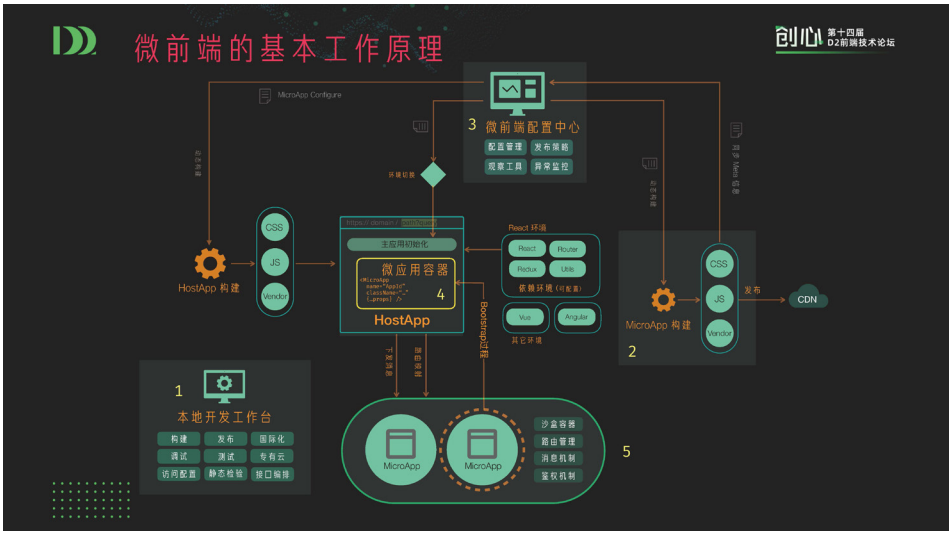
四、微前端基本工作原理

1. 微前端体系构成

微前端不是一个工具或者框架，而是一套架构体系。微前端有三部分构成，首先是基础设施，no Framework 特点。其次是配置中心，用于管理版本、发布策略、以及动态构建。微前端中的若干个微应用需要通过观察工具充当运维职能，可观察工具具备可见性和可控性。如下图，微前端包括配置中心、观察工具、微应用市场、微应用容器。但这远远不够，还需要配套设施即本地开发工作台，因为微前端是在传统的工程化体系之上发展而来。最下层是数据网关的对接口。

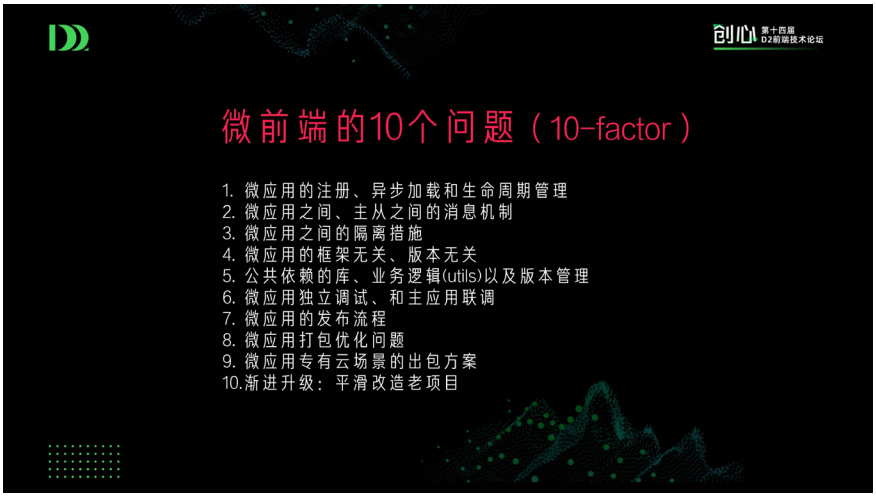
2. 微前端工作原理

微前端的结构是主从结构，主应用中通过微应用容器组件将微应用异步加载进来。其次本地开发工作台具备测试、接口编排、构建、发布等等能力。之后在微前端配置中心做统一的版本管理、发布策略的制定。

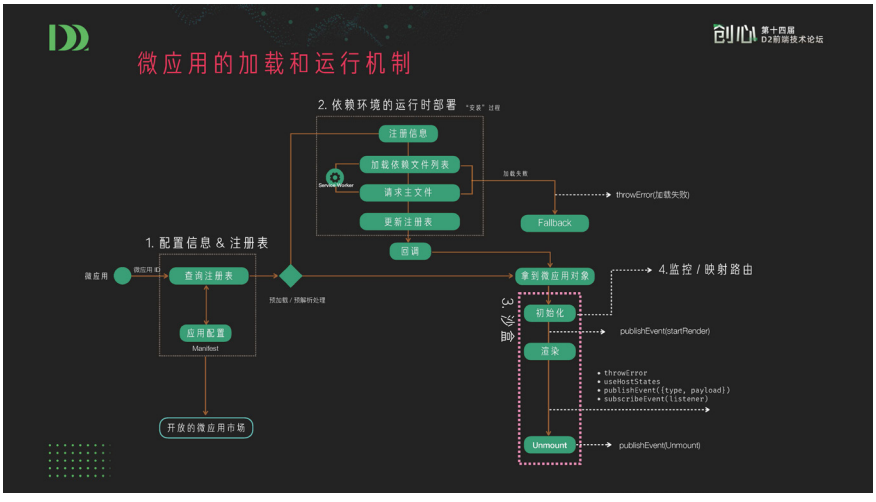


3. 微前端 10 大问题

微服务中有 10 要素的概念，那么微前端大体上也有 10 个主要的问题需要得到重视。

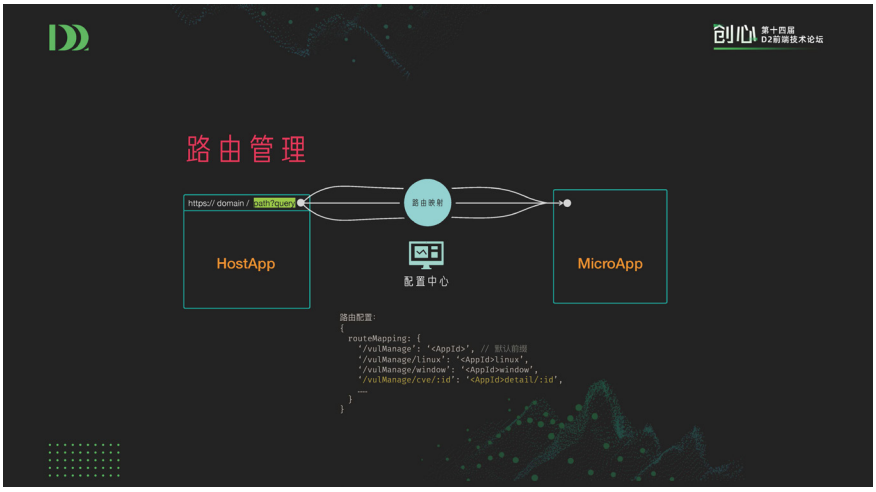


具体而言，微应用的加载和运行机制如下图，首先是配置好应用信息，其次是依赖环境的运行时配置“安装”过程。拿到应用对象后，开始沙盒部分的操作，初始化、渲染。然后进行监控和映射路由。其中沙盒部分的能力提升是阿里云智能团队目前的难点。如果有较好的沙盒工具，也希望可以集成进来。

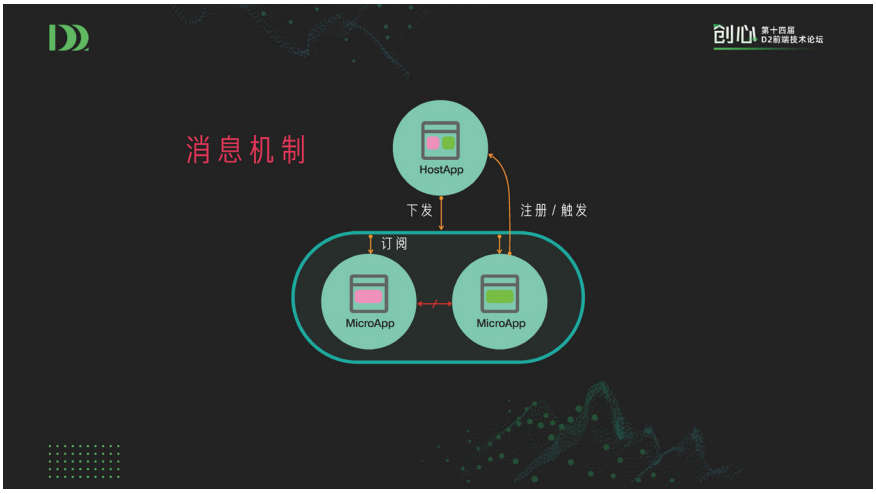


微应用容器需要不断提高几个方面的能力，如安全性、隔离性、回弹性、可见性和通讯能力。其中，安全能力可以分级处理，对于一 / 二级的可信平台，安全级别要求相应降低。对于不可信的第三方应用或者 ISV 开发的应用需要制定安全等级较高的方案。而在目前的安全方案中，iFrame 方案是相对最安全的。再在此基础上，做静态检查，以及安全的 DSL。隔离样式可以用 ShadowDOM，但隔离 JavaScript 和 DOM 的侵入性还需要自己实现。回弹性指的是当加载失败时内部需要有 FallBack 的机制，在容错上上报错误信息。可见性指买点监控，对每个子应用做性能和健康度调试，通过 Inspector 在浏览器中查看出现问题的应用。

目前，阿里云智能团队的微前端框架使用路由管理的方式，在主应用中配置路由，映射子应用。



消费机制中微应用与微应用之间不允许用通讯的方式。而且微应用注册，主应用下发，将触发的事件下发下来，再订阅。



4. 微前端的望楼系统

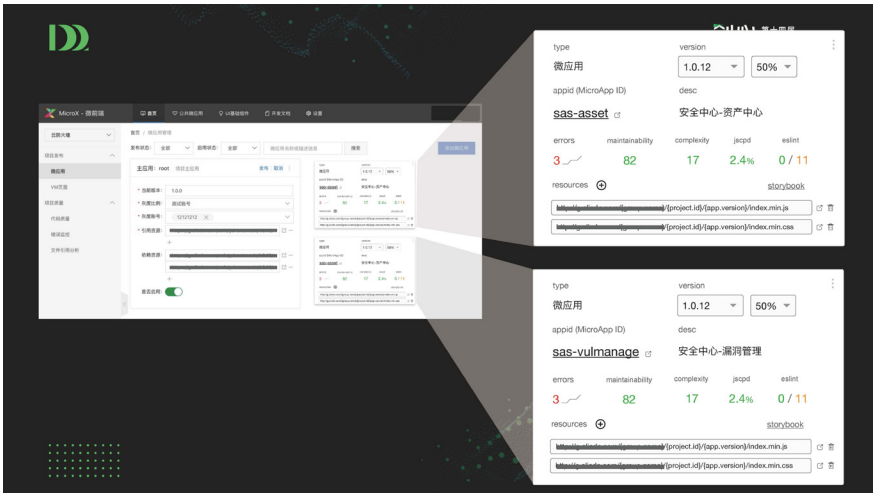
微前端的望楼系统是一个可观察性工具。微前端的望楼系统除了监控系统健康度指标、告警和阀值等，还需要观察系统运行状态和可控性，如观察工具、跟踪、调试

等。可观察性工具包括配置、代码质量、观察、数据服务、度量等等功能。

在未来，前端不只是开发，而是充当开发运维一体化的角色。当一个微应用发布后，如果出错，需要具备运维的能力。运维系统的设计原则中包括可控性、可观察、分析性和自动化。任何具备运维能力的系统都需要遵循这个设计原则，不断的完善系统。

5. 微前端配置中心方案

微前端的配置中心需要具备可灰度、可回滚的中心化管理的能力。下图是阿里云智能团队微前端框架的配置台界面，右侧是每个子应用的子应用的信息。



每个子应用包含基本信息、异常监控情况、代码的健康状态、代码可维护性、代码复杂度变化、版本控制、发布的策略、依赖的资源、以及演示部分。通过视频演示的方式看到子应用的使用方法。

五、总结

微前端是云时代的前端架构体系，不止是一个框架或者组件。微前端的一个使命是构建体系，其中包括基础设施、全链路机制、以及每个环节核心能力的定位。微前端的治理包括配置中心的治理，和运维能力的治理。希望在未来，能够对微前端的涉及的一些概念，如微应用、公共依赖、微应用容器、引导（Bootstrap）、配置中心、微前端治理等等概念构建标准。

标准微前端架构在蚂蚁的落地实践

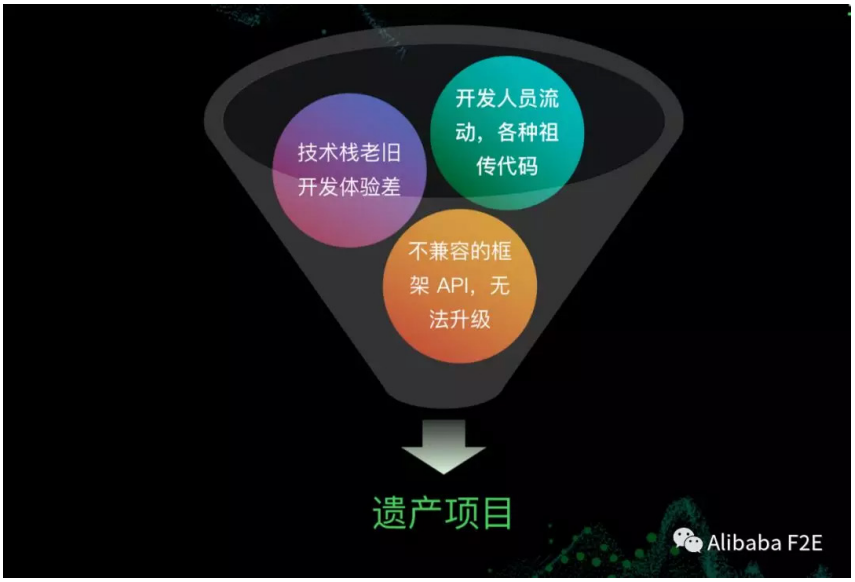
作者：有知

摘要：蚂蚁金服前端技术专家有知在 D2 带来以“标准微前端架构在蚂蚁的落地实践”为题的演讲。首先提出了微前端的场景域在蚂蚁落地时常遇到的问题，然后详细介绍了微前端的定义，最后通过实施一个标准的微前端架构，提出面临的技术决策以及需要处理的技术细节，经过在蚂蚁的实践证明，微前端是一个具有优势的方案。

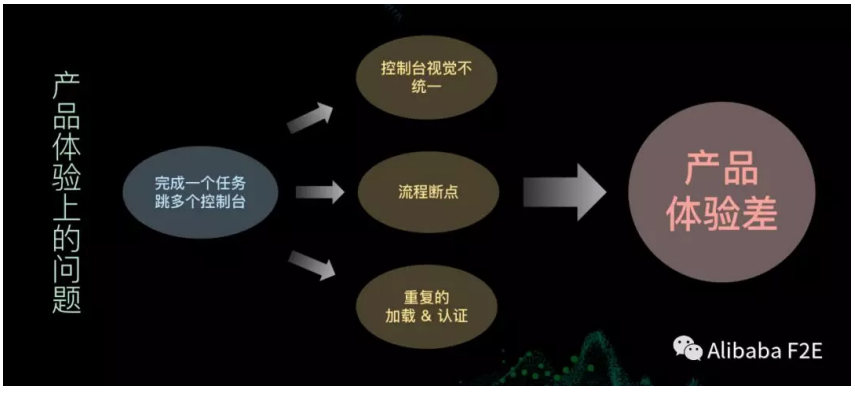
微前端的场景域

在选择一个微前端方案之前，常常需要思考这样一个问题，我们为什么需要微前端。通常对微前端的诉求有两个方面，一是工程上的价值，二是产品上的价值。

对于工程上的价值，可以从一个三年陈的项目来看，如上图所示，左上角 commit 的记录显示，第一次提交是 2016 年 8 月。它的下面是一个 codebase 代码，中间是基本的依赖树 dependencies，右侧为打包的体积。



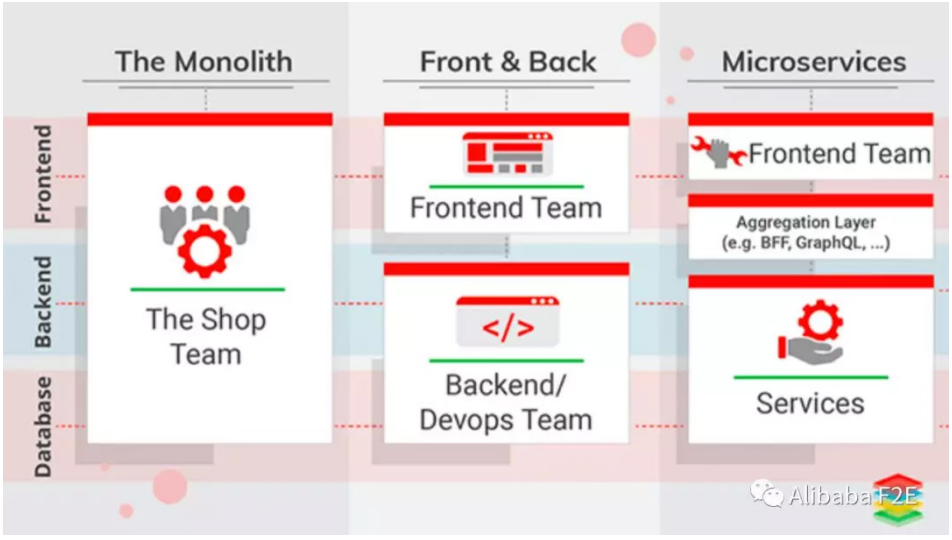
虽然这个三年陈的项目看上去版本比较低，但仍然是相对主流的全家桶方案。这样一个乐观的项目，在真实的场景中经过三年的时间，也不实用了。因为开发的时间比较长，并且人员流动也比较大，会导致一些祖传的代码出现，其次，在技术上不能及时的升级，导致开发体验变得很差，例如打包的时间就超过三分钟。也有可能在不经意间依赖一些不兼容的框架，导致项目无法升级。种种原因，最后很有可能变成一个遗产项目。



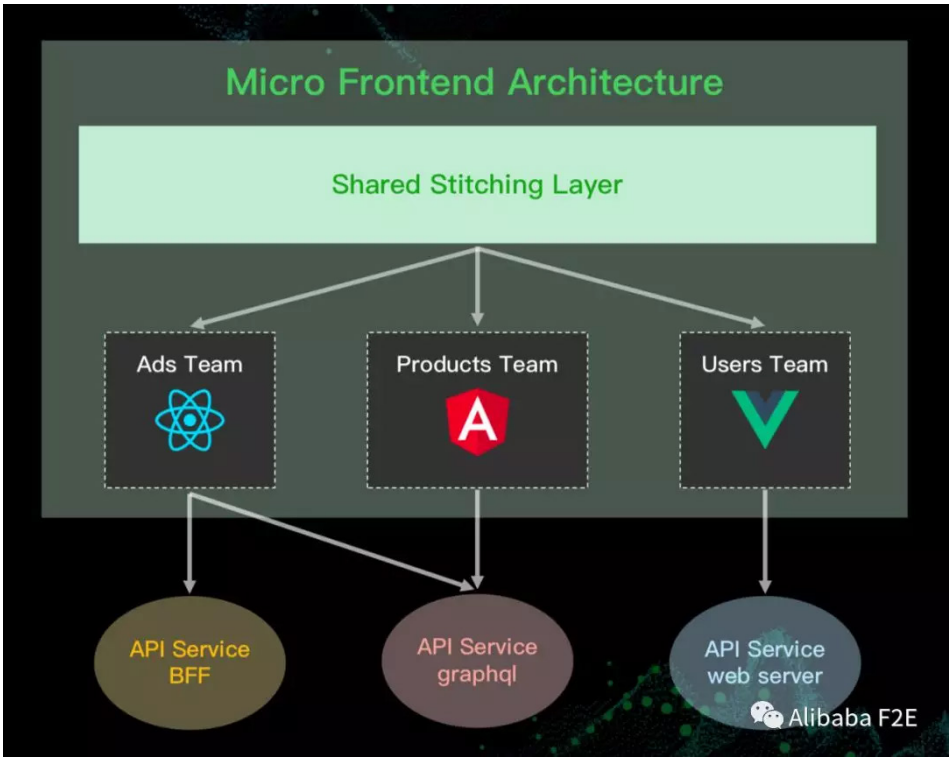
对于产品体验上的问题，例如上图所示，要完成一个跳多个控制台任务，在过程中发现每个控制台视觉不统一、流程出现断点以及重复加载或认证的问题导致整个产品的体验较差。

微前端的定义

上图是来自 Micro Frontends 网站对微前端的定义。意思是所谓微前端就是一种多个团队之间可以使用不同技术构建一个现代化 web 的技术手段以及方法策略。其中的关键字是多团队、采用不同的技术栈以及现代化的 web。



微前端的思路是继承自微服务的思想，如上图所示。



上图所示为微前端的架构图，其中上层为统一共享的拼接层，主要做一些基础信息的加载，和对来自不同团队不同技术栈的客户端在运行时动态组成一个完整的 SPA 应用，以及生命周期的调度和事件的管理。总之，微前端是将微服务概念做了一个很好的延伸和实现。

在具体实践中，衡量一个微前端方案是否是可利用的，需要满足以下几个条件，一是技术栈无关性，不仅指子应用之间使用多个不同的框架，也指在使用同一个框架时，有可能在一个长的时间跨度下，由于框架的不兼容的升级，导致应用被锁死的情况。二是开发、发布及部署独立，要求子应用和主应用做到工程上的解耦和独立。三是应用隔离的能力，是指需要考虑如何不干扰到原来子应用的开发模式和部署模式的情况下，做好运行时的样式隔离、JS 隔离以及异常隔离等。以上几点是基于工程价值方面考虑的。此外，也需要动态组合的能力，是基于产品价值方面考虑的。

落地的关键问题

微前端架构中的技术选择

产品	架构(截止 2017-12)	实现技术
google cloud	纯 SPA	主 portal angularjs，部分页面 angular(ng2)。
aws	纯 MPA 架构	首页基于 angularjs。各系统独立域名。
阿里云	纯 MPA 架构	首页基于 angularjs，各系统独立域名。
七牛	SPA & MPA 混合架构	入口 dashboard 及 个人中心模块为 spa，使用同一 portal 模块（AngularJs(1.5.10) + webpack）。其他模块自治，或使用不同版本 portal，或使用其他技术栈。
又拍云	纯 SPA 架构	基于 angularjs 1.6.6 + ui-bootstrap。控制台内容较简单。
ucloud	纯 SPA 架构	angularjs 1.3.12

按架构类型区分，常规 web 应用的架构类型分为两种，一种是 MPA，另一种是 SPA。如上图所示为 2017 年各云产品控制台架构调研，除了 google cloud 之外，大部分的云厂商都使用 MPA 架构。MPA 的优点在于部署简单，具备独立开发和独立部署的特性。但是，它的缺点是完成一个任务要跳到多个控制台，并且每个控制台又是重复刷新的。而 SPA 能极大保证多个任务之间串联的流畅性，但问题是通常一个 SPA 是一个技术栈的应用，很难共存多个技术栈方案的选型。SPA 和 MPA 都是微前端方案的基础选型，但是也都存在各自的问题。

按运行时特性区分，微前端包含两个类别，一类是单实例，另一类是多实例。单实例场景如上图中左侧，通常是一个页面级别的组合，例如一个运行时只有一个 App 被激活。多实例场景如上图右侧，像一个组件或者是容器级别的应用，运行时可以做到多个应用被同时激活。这两种模式都有自己适应的场景和优势。微前端架构的核心诉求是实现能支持自由组合的微前端架构，将其他的 SPA 应用以及其他组件级别的应用自由的组合到平台中。那么，如何选择 SPA 和 MPA 以及单实例和多实例是一个问题，我们是否能探索出一种方案，将 SPA 和 MPA 工程上的特点结合起来，同时兼顾多实例和单实例运行时的场景来实现。

技术细节上的决策

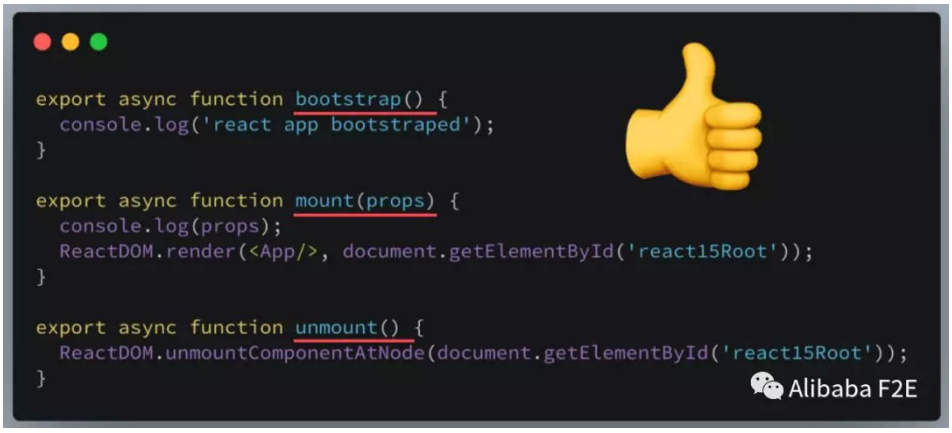
为了实现上述的方案，在技术细节上的决策需要注意以下问题：

一是如何做到子应用之间的技术无关；

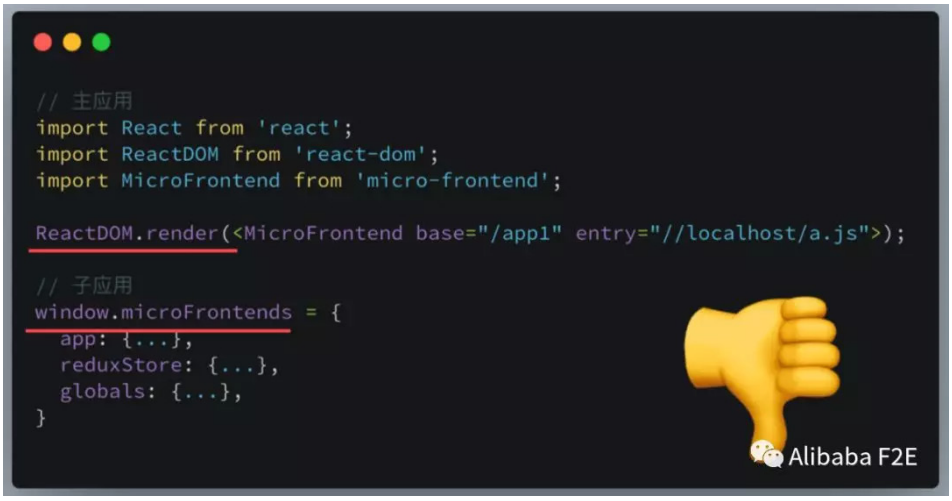
二是如何设计路由和应用导入；

三是如何做到应用隔离；

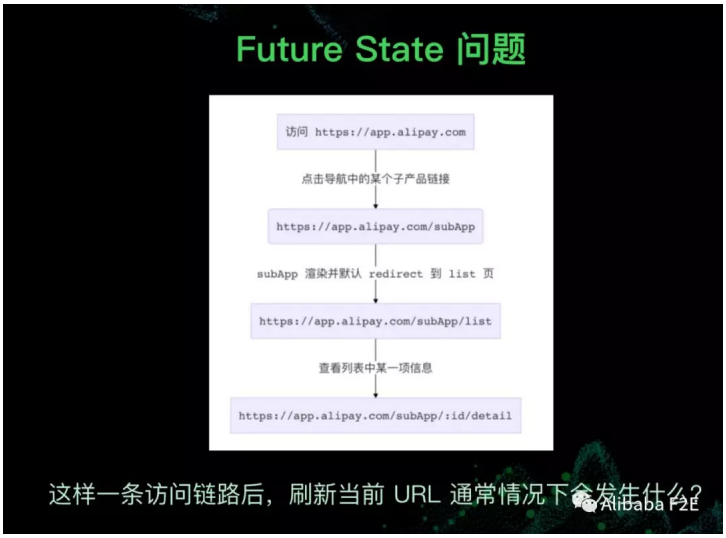
四是基础应用之间资源的处理以及跨应用间通信的选择。

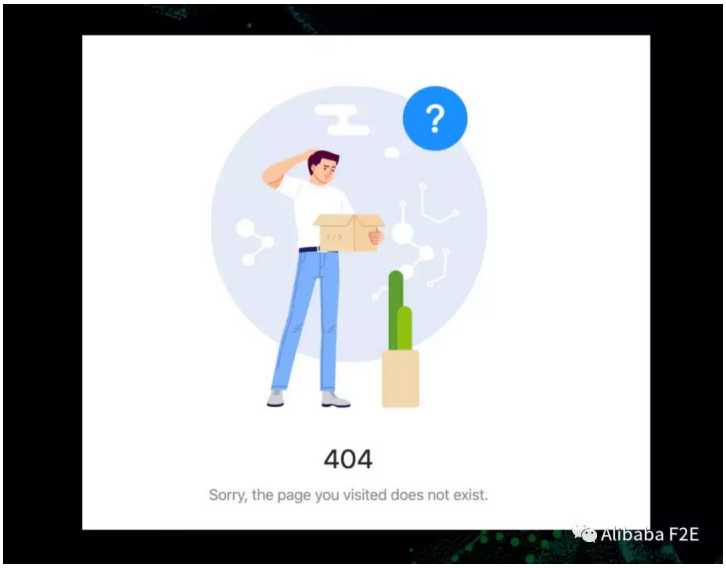


对于如何做到子应用之间的技术无关问题，我们是通过协议来解决的。如上图所示的方式，就可以完成子应用的导入。如果子应用接入时做了一些框架上的耦合或者依赖一个具体实现库的机制，就一定会存在与实现库版本耦合的可能，不利于整个微前端生态的统一和融合。



如上图所示是一个不与某个具体框架实现耦合的例子。





对于路由的问题，如上图所示。这样一条访问链路后，刷新当前 URL 通常情况下会发生什么？正常访问一个站点，经过一番操作之后，进入到站点的列表页，路由会变大很复杂，但如果是一个微前端用户，刷新一下页面会出现 404 的情况。解决思路是将 404 路由 fallback 到一个异步注册的子应用路由机制上。

```
<html>
  <head>
    <title>sub app</title>
    <link rel="stylesheet" href="//localhost/app.css">
  </head>
  <body>
    <main id="root"></main>
    <script src="//locahost/base.js">
  </body>
</html>

<script>
  import React from 'react'
  import ReactDOM from 'react-dom'

  ReactDOM.render(<App/>, document.getElementById('root'))
</script>
```

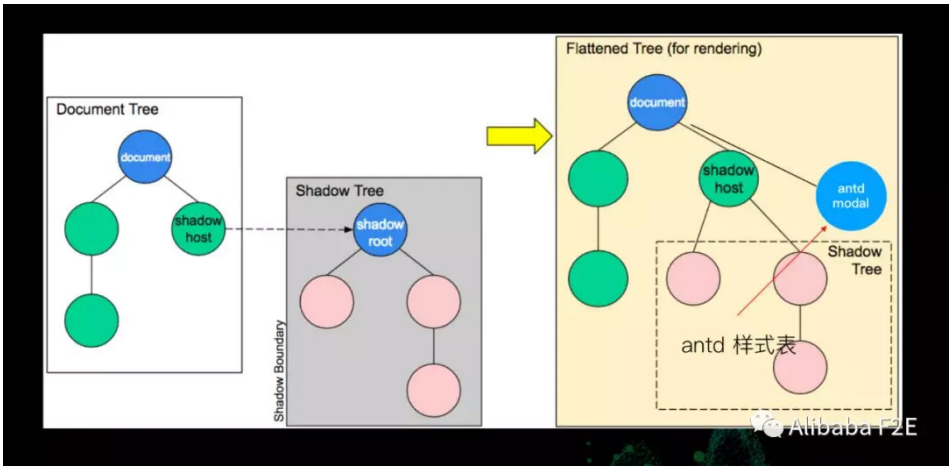
Alibaba F2E

对于应用导入方式的选择，比较常见的方案是 Config Entry，如上图所示。通过在主应用中注册子应用依赖哪些 JS。这种方案一目了然，但是最大的问题是 ConfigEntry 的方式很难描述出一个子应用真实的应用数据信息。真实的子应用会有一些 title 信息，依赖容器 ID 节点信息，渲染时会依赖节点做渲染，如果只配 JS 和 CSS，那么很多信息是会丢失的，有可能会导致间接上的依赖。

另外一种方案是 HTML Entry，如上图所示，直接配 html，因为 html 本身就是一个完整的应用的 manifest，包含依赖的信息。HTML Entry 的优点是接入应用的信息可以得到完整的保留，接入应用地址只需配一次，子应用的原始开发模式得到完整保留，因为子应用接入只需要告知主应用 html 在哪，包括在不接入主应用时独立的打开。它的缺点是将解析的消耗留给了运行时。而 Config Entry 相较于 HTML Entry 减少了运行时的解析消耗。Config Entry 的缺点是主应用需配置完整的子应用信息，包含初始 DOM 信息、js/css 资源地址等。

对于样式隔离问题，例如 BEM，每个子应用在写样式之前要加一些前缀，做一些隔离，但是这个做法并不推荐。相对而言，CSS Module 更简单高效，也更智能化，是比较推荐的方式，但是也存在着问题。而 Web Components 看上去很不错，

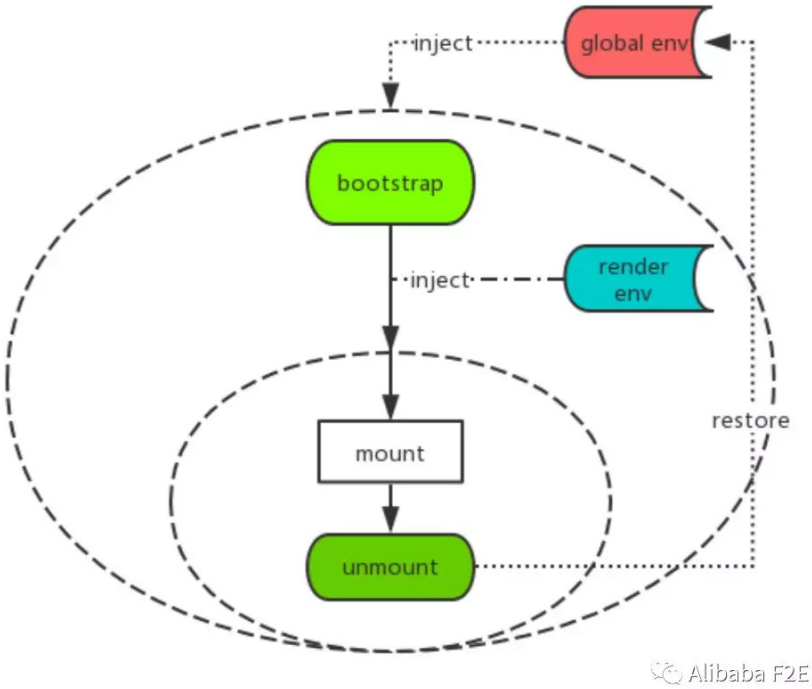
但在实践过程中也会发生一些问题。



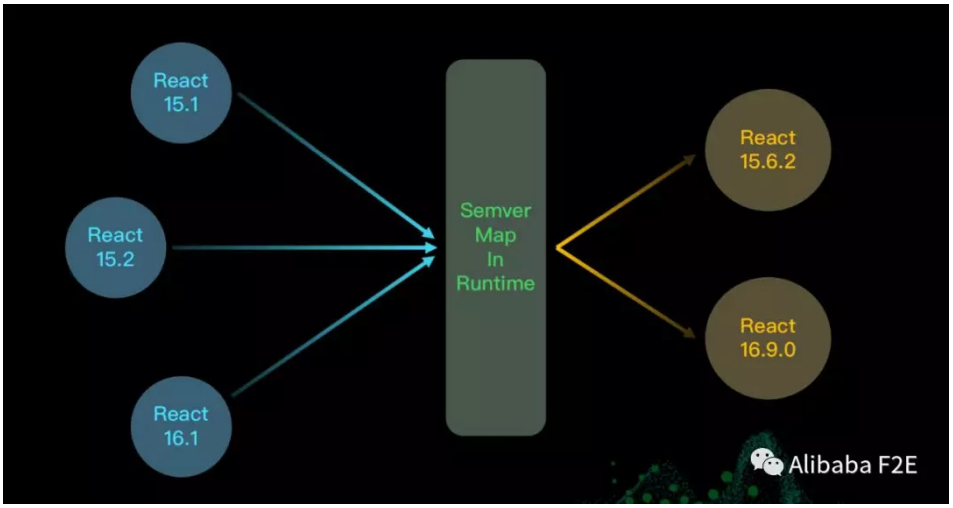
例如在 Web Components 渲染的流程中出现了问题，如上图所示。

解决方案为上图所示。在 antd 中提供了全局的 API，可以提前设置好所有的弹框的 container，但是也不是每个组件库都能像 antd 一样完成度那么高。

蚂蚁所采用的解决方案是做动态的加载和卸载样式表，如上图所示，这种方案是很有效的。



对于 JS 隔离，蚂蚁提出了 JS Sandbox 机制，如上图所示，其中 bootstrap、mount 及 unmount 生命周期是子应用需要暴露出来的，因为子应用的整个生命周期都是被主应用所管理的，所以可以在主应用中给子应用插入各种拦截的机制，也可以捕获到子应用在加载期间做了哪些全局上的修改。在 unmount 时，可以将全局上的副作用全部手动移除掉，同时也可以实现在重新进来时，将上次忘记卸载的副作用重建一遍，因为需要保证下次进来时能完整回复到与上次一致的上下文。



对于资源加载问题，在微前端方案中存在一个典型的问题，如果子应用比较多，就会存在之间重复依赖的场景。解决方案是在主应用中主动的依赖基础框架，然后子应用保守的将基础的依赖处理掉，但是，这个机制里存在一个问题，如果子应用中既有 react15 又有 react16，这时主应用该如何做？蚂蚁的方案是在主应用中维护一个语义化版本的映射表，在运行时分析当前的子应用，最后可以决定真实运行时真正的消费到哪一个基础框架的版本，可以实现真正运行时的依赖系统，也能解决子应用多版本共存时依赖去从的问题，能确保最大程度的依赖复用。

对于应用之间数据共享及通信的问题，蚂蚁提出了两个原则，第一个原则是基于 props 以单向数据流的方式传递给子应用。第二个原则是基于浏览器原生事件做跨业务之间的通信。

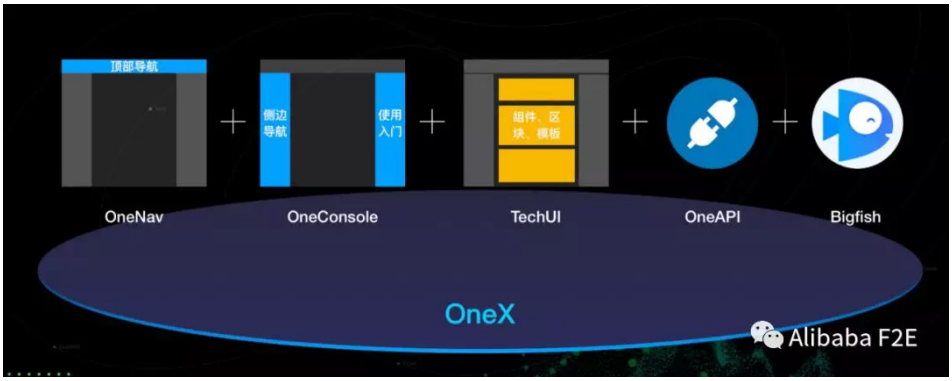
在真实的生产实践中，蚂蚁总结出了几点经验及建议：兄弟节点间通信以主应用作为消息总线，不建议自己封装的 Pub/Sub 机制，也不推荐直接基于某一状态管理库做数据通信。

上图所示为蚂蚁在实践中做的性能优化，包括异步样式导致闪烁问题的解决以及预加载问题的解决。

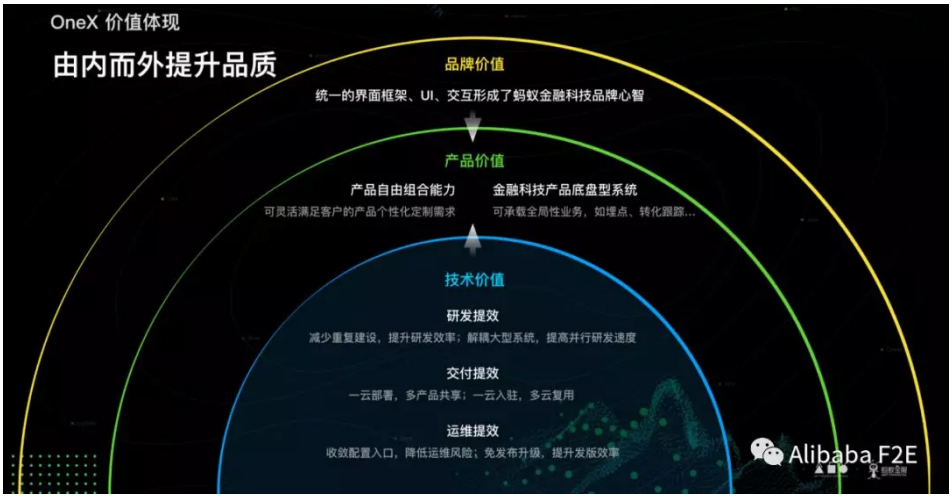
上图所示为微前端方案涉及到的技术点，本文分享了图中三分之二的内容。

在蚂蚁金服做了大量关于微前端方案之后，总结了衡量一个微前端方案是否友好的两个标准，第一个标准是技术无关，也是微前端最核心的特性，不论是子应用还是主应用都应该做到框架不感知。第二个标准是接入友好，子应用接入应该像接入一个 iframe 一样轻松自然。

蚂蚁的微前端落地的实践成果

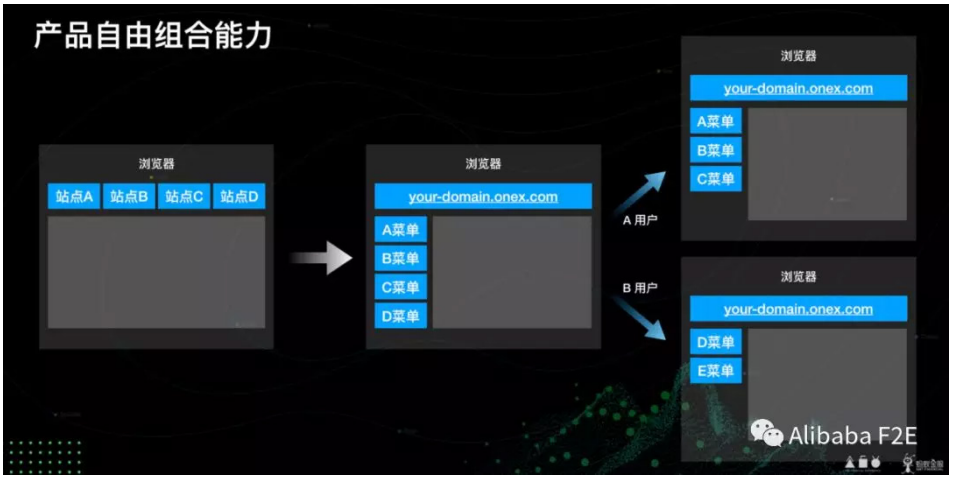


蚂蚁内部基于微前端基础架构提出了一体化上云解决方案，称为 OneX，是一个基础的平台，它可以将各种流程和工具串联，其价值体现在品牌、产品和技术方面。



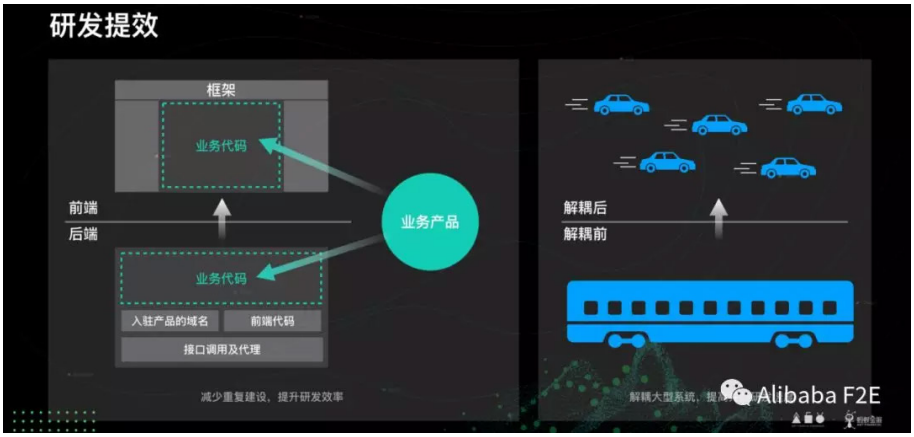
品牌价值指的是统一的界面框架、UI、交互形成了蚂蚁金服科技品牌心智。

上图所示为蚂蚁的一个真实应用的例子，除了中间接入的产品是自己控制之外，其他内容都是由平台提供，这样，如论是一个三年陈项目还是新做的项目，在基本的视觉上可以做到统一。

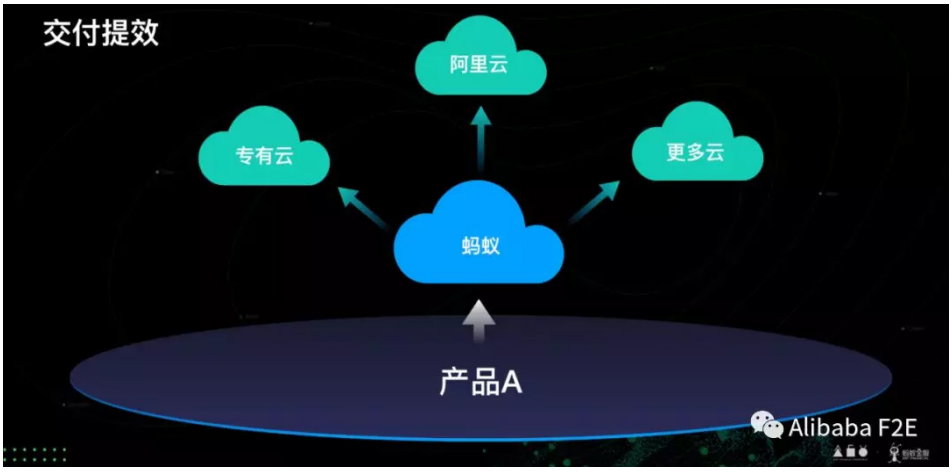


产品价值指的是产品具有自由组合能力。之前的产品是多个产品、多个站点的控制台，而现在只需要一个控制台，将多个产品自由的组合，这样，可以在商业上有更

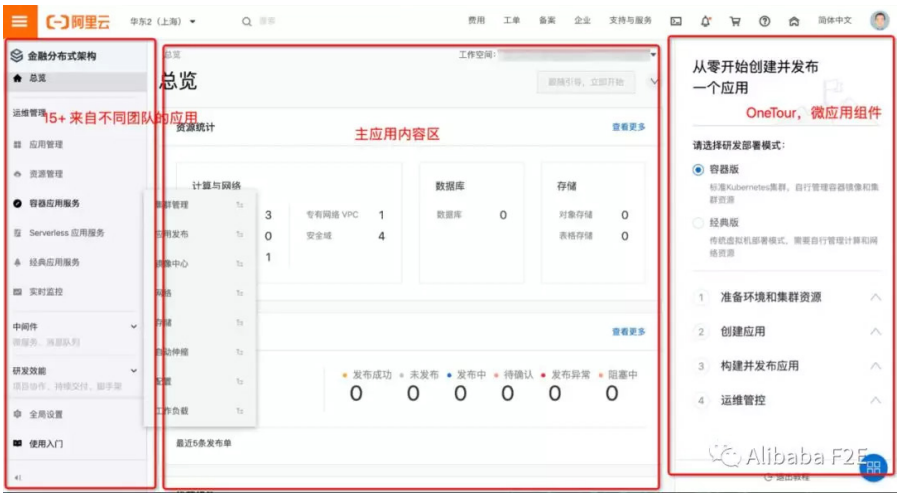
多的相应空间以及更多自由的搭配。基于这样的系统也可以做一些全局性的事情，例如埋点、用户的转化跟踪业务。



技术价值指的是研发上的提效。经过微前端的改造后，蚂蚁可以将大型的系统解耦成可以独立开发的并行的小型的系统，这些小型系统可以交给别的团队或者使用可视化的系统去实现，最后在运行时只需要将他们集成起来。

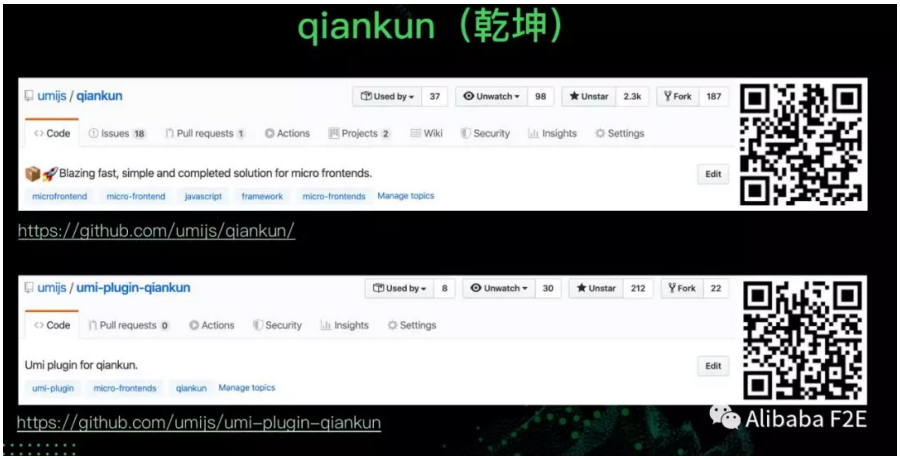


在技术价值方面也可以实现交付上的提效，只需要在某一个环境的任意一个环境中做平台上的接入，应用就可以做到在多余的环境中不改代码，直接运行。



上图所示为阿里云刚上市的一个产品例子，其中包括 15 个来自不同团队的应用进行维护，它的特点是并不是单独为阿里云而设计的，之前在蚂蚁也有运行，只不过在阿里云中做了动态的组合。OneTour 微应用组件主要解决的是在多个产品控制台之间自由切换导致流程割裂的问题。

蚂蚁微前端的落地成果包括：有 70+ 线上应用接入（阿里云 + 蚂蚁云 + 专有云），最复杂一个控制台同时集成 15 个应用，并且有 4+ 不同技术栈，以及开发到发布上线全链路的自动化支持，一云入驻多云运行。



蚂蚁也热衷于分享技术上的成果，包括微前端内核的开源、umi 插件。上图所示为 qiankun 方案，是与框架无关的微前端内核，umi-plugin-qiankun 是基于 umi 应用提供的一个插件，如果是 umi 应用，就可以通过集成插件和更改配置成为微前端应用。基于上述实践的检验和内部的实践结果来看，在大规模中后台应用场景下，微前端架构是一个值得尝试的方案。

Serverless

前端新思路：组件即函数和 Serverless SSR 实践

作者：狼叔

在今天，对于 Node.js 运维和高并发依然是很有挑战的，为了提效，将架构演进为页面即服务，可是粒度还不够，借着云原生和 Serverless 大潮，无运维，轻松扩展，对前端是极大的诱惑。那么，基于 FaaS 之上，前端有哪些可能性呢?

2019 年上半年，我在阿里巴巴经济体前端委员会推进的 Serverless 研发体系共建项目中负责 Serverless SSR 的研究，将 CSR，SSR，边缘渲染进行整合和尝试，提出组件即服务的概念（Component as Service），试图结合 FaaS，做出更简单的开发方式。本次分享主要围绕 Serverless SSR 和它的演进过程、背后思考为主。

本文是狼叔在 D2 大会的分享《前端新思路：组件即服务和 Serverless SSR 实践》的内容，阅读需要 10 分钟，你讲了解如下内容。

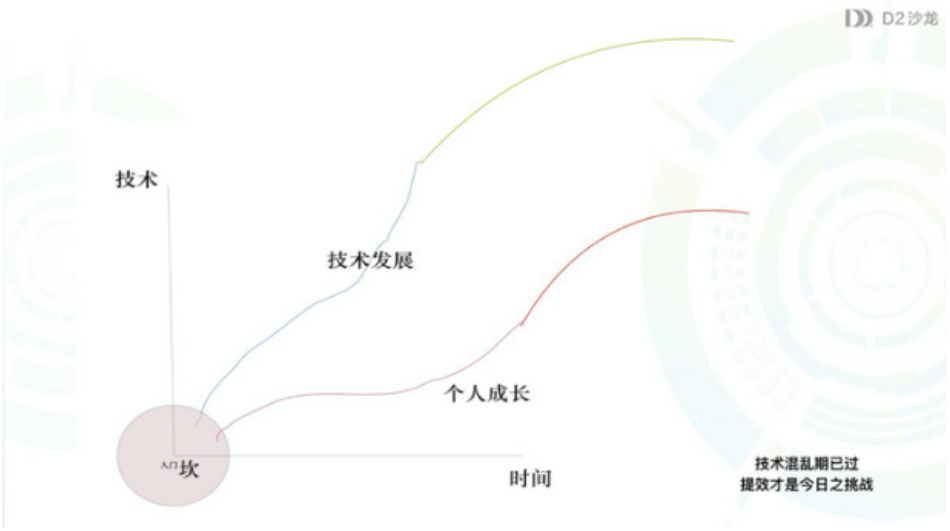
1. 可以了解 Serverless 时代端侧渲染面临的具体问题
2. 可以了解 Serverless SSR 规范以及渲染体系的完整工作链路和原理
3. 为业内提供解决 Serverless SSR 渲染问题的新思路

狼叔（网名 i5ting）现为阿里巴巴前端技术专家，Node.js 技术布道者，Node 全栈公众号运营者，曾就职于去哪儿、新浪、网秦，做过前端、后端、数据分析，是一名全栈技术的实践者。目前负责 BU 的 Node.js 和基础框架开发，已出版《狼书（卷 1）更了不起的 Node.js》。

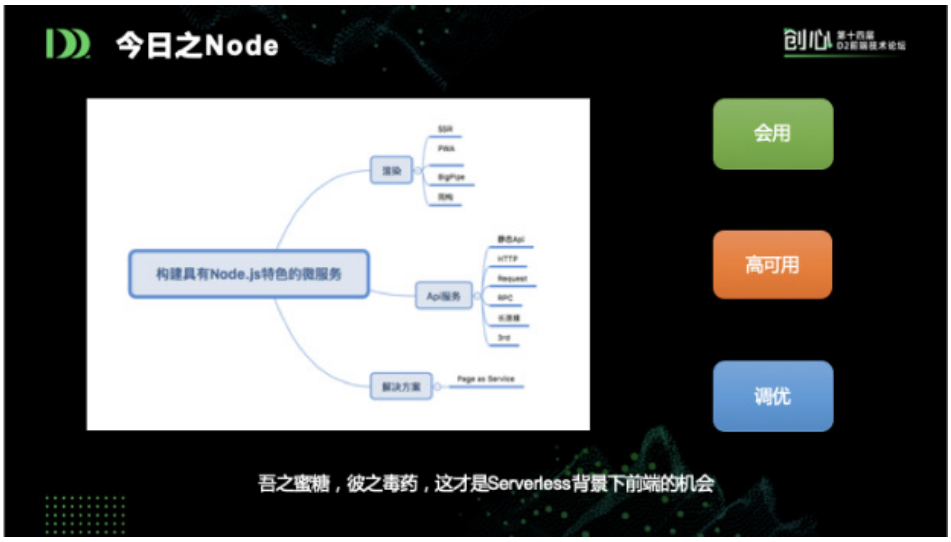
Wolfred Sang (a.k.a. i5ting) is a full-stack developer and Node.js evangelist. He works for Alibaba Group as a Principal Front-End Developer and runs a self-media on the topic of Full-stack Node.js. He worked for well-known dot-com companies in the past, such as Qunar, Sina and NQ Mobile. His expertise and experiences cover not only front-end development but also back-end engineering and data insight extracting. He is currently leading the

Node.js development and maintaining the core codebase in his business unit. His book “The Marvelous Node.js” (Part I) was published in July 2019.

技术趋势分析



今年的技术趋势，我的判断是技术混乱期已过，提效才是今日的挑战。



在 Node.js 领域，今年新东西也不多，最新已经发布到 13，lts 是 12，Egg.js 的生态持续完善，进度也不如前 2 年，成熟之后创新就少了。在很多框架上加入 ts 似乎已经政治正确了。比如自身是基于 ts 的 nest 框架，比如阿里也开源了基于 Egg 生态的 midway 框架，整体加入 ts，类型系统和 oop，对大规模编程来说是非常好的。另外 GraphQL 也有很强的应用落地场景，尤其是 Apollo 项目带来的改变最大，极大的降低了落地成本。已经用 rust 重写的 deno 稳步进展中，没有火起来，但也有很高的关注度，它不会替代 Node.js，而是基于 Node 之上更好的尝试。

今日的 Node.js 存在的问题是会用很容易，做到高可用不容易，毕竟高可用对架构运维要求更好一些，这点对前端同学要求会更难一些。做到高可用之后，做性能调优更难。其实，所有这些难点的背后是基于前端工程师的角度来考虑的，这也是非常现实的问题。

对于很多团队，上 Node 是找死，不上 Node 是等死。在今天 web 框架已经相当成熟，所以如何破局，是当下最破解需要解决的问题。

你可能会感觉 Node.js 热度不够，但事实很多做 Node.js 的人已经投身到研发模式升级上了。对于今天的 Node.js 来说，会用很容易，但用好很难，比如高可用，性能调优，还是非常有挑战的。我们可以假想一下，流量打网关，网关根据流量来实例化容器，加载 FaaS 运行时环境，然后执行对应函数提供服务。在整个过程中，不许关心服务器和运维工作，不用担心高可用问题，是不是前端可以更加轻松的接入 Node.js。这其实就是当前大厂前端在做的前端基于 Serverless 的实践，比如基于 FaaS 如何做服务编排、页面渲染、网关等。接入 Serverless 不是目的，目的是让前端能够借助 Serverless 创造更多业务价值。

为何如此钟爱 SSR？

在 2017 年底，优酷只有 passport 和土豆的部分页面用 Node.js，QPS 不高，大多是一些尝试性业务，优酷 PC 和 H5 核心页面还都是 PHP 模板渲染。最近 2 年基于阿里强大的技术体系，我们也对 PC、H5 多端进行了技术改造。今年双十一是

React SSR 第一次扛双十一，具有一定意义，这里简单总结回顾和展望。

背景就不赘述了，参见《这！就是优酷 Node.js 重构之路》

- 将优酷 C 端核心页面全部用 Node 重写，完成了 PHP 到 Node.js 的迁移。在没有 PHP 同学的情况下，前端可以支撑业务。
- 性能提升明显，从 v1（Bigpipe+jQuery）到 v2（React SSR），性能逐步提升。PC 页面首屏渲染降到 150ms、播放器起播时间从 4.6 秒优化到 2 秒。H5 站上了 React SSR 后，性能提升 3 倍，H5 唤端率提升也极其明显，头条短视频唤端率由 5.68% 提升到 9.4%，环比提升 65%。单机性能 qps 从 80 提升 150+（压测最高可以到 300 左右）。
- QPS 过万，2 年没有 p4 以上故障，相对来说是比较稳定的。扛过双十一、世界杯，最高三倍以上的流量。
- 在集团前端委员会承担 Serverless SSR 专项。

裕波曾经问我，为何如此钟爱 SSR？

从前端的角度看，它是一个相对小的领域。PC 已经非主流，H5 想争王者，却

不想被 rn、weex 中间截胡。怎么看，SSR 能做的都有限。但是，用户体验提升是永远的追求，另外 web 标准化是正统，在二者之间，和 Node 做结合，除了 SSR，目前想不到更好的解法。

贴着 C 端业务，从后端手里接过来 PC、H5，通过 Node 构建自己的生存之地是必然的选择。

活下来之后就开始有演进，沉淀，通过 C 端业务和 egg-react-ssr 开源项目的沉淀，我们成功的打通 2 点。

1. 写法上的统一：CSR 和 SSR 可以共存，继而实现二种模式的无缝切换
2. 容灾降级方案：从 Node SSR 无缝切换到 Node 的 CSR，做到第一层降级，从 Node CSR 还可以继续降到 CDN 的 CSR

2019 年，另外一个风口是 Serverless，前端把 Serverless 看成是生死之地，下一代研发模式的前端价值证明。那么，在这个背景下，SSR 能做什么呢？基于 FaaS 的 SSR 如何呢？继续推演，支持 SSR，也可以支持 CSR，也就是说基于 FaaS 的渲染都可以支持的。于是和风驰商量，做了 Serverless 端侧渲染方向的规划。

本来 SSR 是 Server-side render，演进为 Serverless-side render。元彦给了一个非常好概念命名，Caaf，即 Component as a fuction。渲染层围绕在以组件为核心，最终统统简化到函数层面。

在今天看，SSR 是成功的，一个曾经比较偏冷的点已经慢慢变得主流。集团中，基于 React/Rax 的一体化开发，可以满足前端所有开发场景。优酷侧的活动搭建已经升级到 Rax1.0，对外提供 SSR 服务。在 uc 里，已经开始要将 egg-react-ssr 迁移到 FaaS 上，代码已经完成迁移。

- PC/ 中后台，React 的 CSR 和 SSR
- 移动端 /H5，Rax 的 CSR 和 SSR。尤其是 Rax SSR 给站外 H5 提供了非常好的首屏渲染时间优化，对 C 端或活动支持是尤其有用的。

在 2020 年，基于 FaaS 之上的渲染已经获得大家的认可。另外大量的 Node.js 的 BFF 应用已经到了需要治理的时候，BFF 感觉和当年的微服务一样，太多了就会牵扯到管理成本，这种情况下 Serverless 是个中台内敛的极好解决方案。对前端来说，SSR 让开发变得简单，基于 FaaS 又能很好的收敛和治理 BFF 应用，结合 WebIDE，一种极其轻量级基于 Serverless 的前端研发时代已经来临了。

Serverless-side render 概念升级

从 BFF 到 SFF

了解 SSR 之前，我们先看一下架构升级，从 BFF 到 SFF 的演进过程。

BFF 即 Backend For Frontend（服务于前端的后端），也就是服务器设计 API 时会考虑前端的使用，并在服务端直接进行业务逻辑的处理，又称为用户体验适配器。BFF 只是一种逻辑分层，而非一种技术，虽然 BFF 是一个新名词，但它的理念由来已久。

在 Node.js 世界里，BFF 是最合适的应用场景。常见的 API、API proxy、渲染、SSR+API 聚合，当然也有人用来做网关。

从 Backend For Frontend 升级 Serverless For Frontend，本质上就是利用 Serverless 基建，完成之前 BFF 的工作。那么，差异在哪里呢?

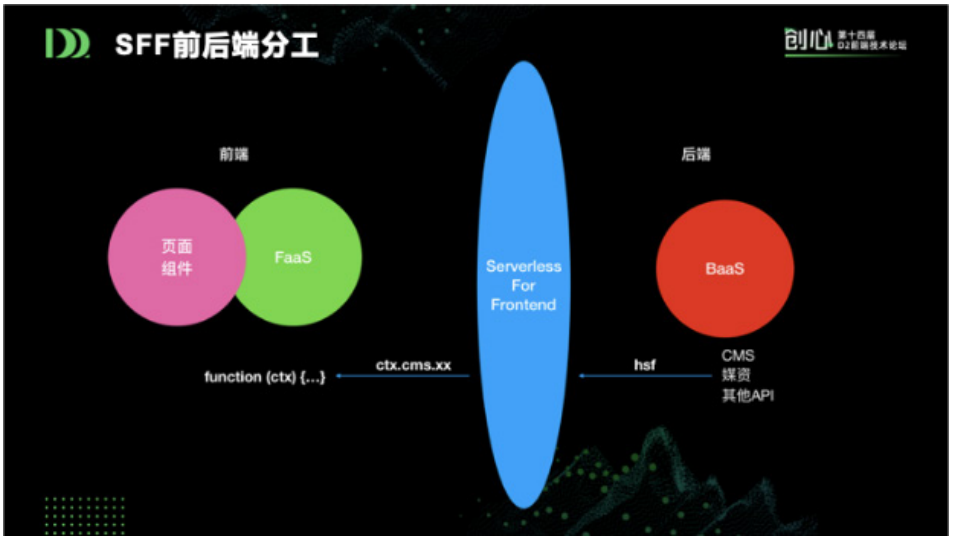
核心是从 Node 到 FaaS，本质上还是 Serverless，省的其实只是运维和自动扩缩容的工作，一切看起来都是基建的功劳，但对于前端来说，却是极为重大的痛点解决方案，能够满足所有应用场景，基于函数粒度可以简化开发，乃生死必争之地。

SFF 前后端分工

Serverless 简单理解是 FaaS+BaaS。FaaS 是函数即服务，应用层面的对外接口，而 BaaS 则是后端即服务，更多的是业务系统相关的服务。当下的 FaaS 还是围绕 API 相关的工作为主，那么，前端如何和 Serverless 绑定呢?

Serverless For Frontend（简称 SFF）便是这样的概念，基于 Serverless 架构提供对前端开发提效的方案。

下面看一下 SFF 分工，这张图我自认为还是非常经典的。首先将 Serverless 劈成 2 半，前端和后端，后端的 FaaS 大家都比较熟悉了，但前端页面和 FaaS 如何集成还是一片待开发的新领域。



举个例子，常见 BFF 的例子，hsf 调用获得服务端数据，前端通过 ctx 完成对前端的输出。这时有 2 种常见应用场景

1. API，同后端 FaaS（RPC 居多）
2. 页面渲染（http 居多）

基于 FaaS 的页面渲染对前端来说是必须的。从 beidou、Next.js、egg-react-ssr 到 Umi SSR，可以看出服务端渲染是很重要的端侧渲染组成部分。无论如何，React SSR 都是依赖 Node.js Web 应用的。那么，在 Serverless 时代，基于函数即服务（Functions as a Service，简写为 FaaS）做 API 开发相关是非常简单的。

1）无服务，不需要管运维工作

2）代码只关系函数粒度，面向 API 变成，降低构建复杂度

3）可扩展

目前还是 Serverless 初期，大家还是围绕 API 来做，那么，在 FaaS 下，如何做好渲染层呢？直出 HTML，做 CSR 很明显是太简单了，对于 React 这种高级玩法如何集成呢？

其实我们可以做的更多，笔者目前能想到的 Serverless 时代的渲染层具有如下特点。

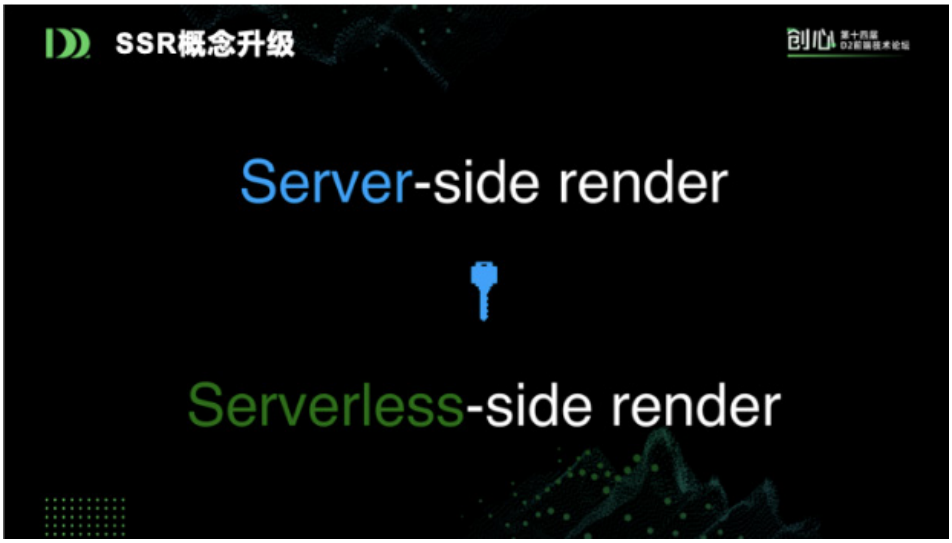
- 采用 Next.js/egg-react-ssr 写法，实现客户端渲染和服务端渲染统一
- 采用 Umi SSR 构建，生成独立 umi.server.js 做法，做到渲染
- 采用 Umi 做法，内置 Webpack 和 React，简化开发，只有在构建时区分客户端渲染和服务端渲染，做好和 CDN 如何搭档，做好优雅降级，保证稳定性
- 结合 FaaS API，做好渲染集成。 为了演示 Serverless 下渲染层实现原理，下面会进行简要说明。在 Serverless 云函数里，一般会有 server.yml 作为配置文件，这里以 lamda 为例子。

SSR 概念升级

现状

1. 现有 FaaS 主要是针对 API 层做扩展，视图渲染是一个新的命题。
2. 竞品 Serverless.com 提供了 Components 类似的视图渲染层方案
3. 如何打造一个基于阿里技术栈又有业界领先的端侧渲染解决方案

业界还缺少最佳实践，这是极好的机会。因此，我们对 SSR 做了概念上的升级（感谢 justjavac 大佬的提示）



Serverless 端渲染层，是针对 SSR 做概念和能力升级

- SSR 从 Server side render 升级为 Serverless side render，基于 FaaS 环境，提供端侧页面渲染能力。
- Serverless 渲染层涵盖的范围扩展，从服务器端渲染升级到同时支持 CSR 和 SSR 2 种渲染模式。

在 Serverless 背景下，页面渲染层包含 2 种情况。

- 基于 FaaS 的客户端渲染
- 基于 FaaS 的服务器端渲染

目标是提供基于 FaaS 的页面渲染描述规范，提供标准化组件描述，统一组件写法，用法简单，易实现，可扩展。因此，我们制定了 SSR-spec 规范，下面会详细讲解。

Serverless-side render 规范和实现原理

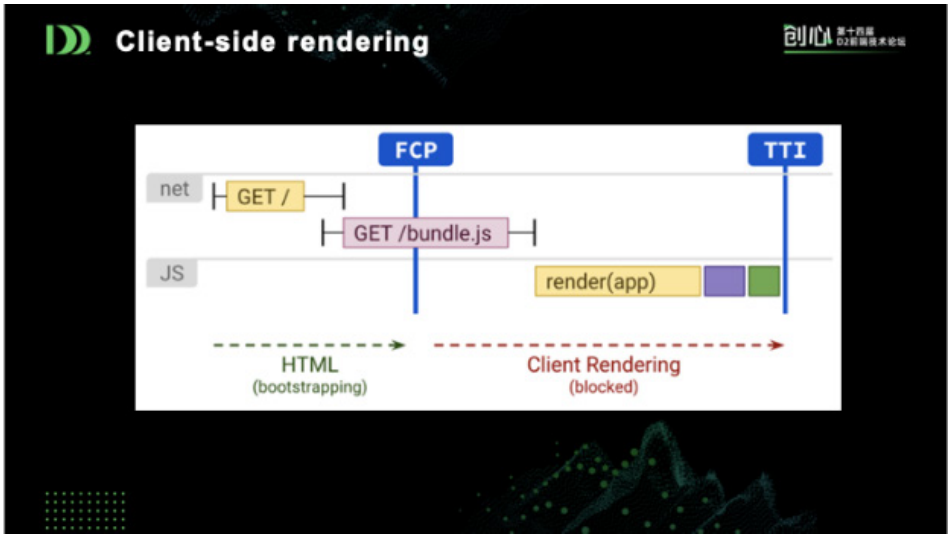
在讲规范之前，我们先简单了解 3 个术语。

名称	英文	简写	描述
客户端渲染	client-side render	CSR	通过React、Rax编写的组件，打包构建后，以html文件形式分发到CDN上，不需要Node.js支持
服务器端渲染	Server-side render	SSR	通过React、Rax编写的组件，打包构建后，分别生成server bundle和client bundle，其中server bundle由Node.js服务端写入到浏览器，client bundle分发到CDN上，采用混搭的写入渲染方式，来提高首屏渲染效率。
组件即函数	Component as a function	Caaf	组件即函数，在Serverless领域以函数为核心，在渲染也是已函数为核心，做到资源与函数分类，实现渲染层独立，为开发提供更大家便利。

CSR 和 SSR

先科普一下 CSR 和 SSR 的概念

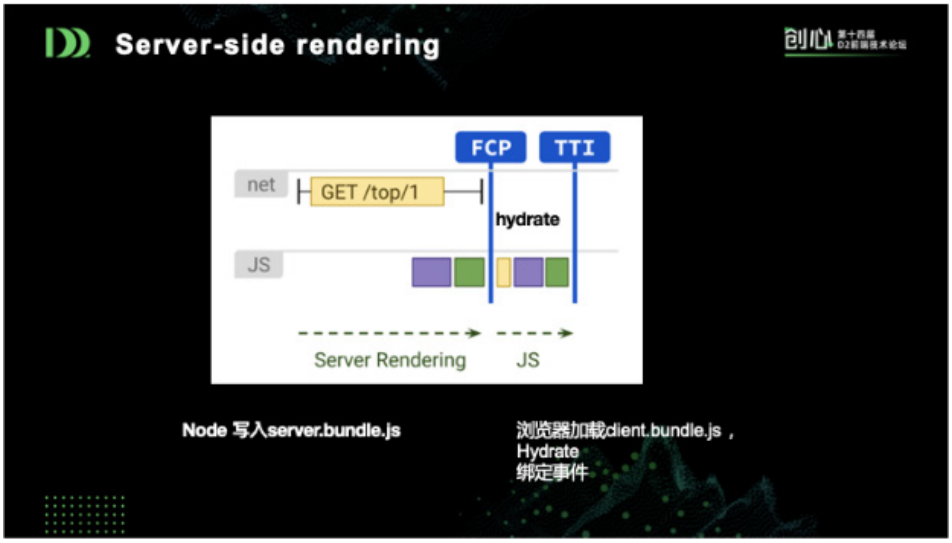
客户端渲染（简称 CSR），简单理解就是 html 是没有被动态数据灌入的，即所谓的静态页面。比如通过 React、Rax 编写的组件，打包构建后，以 html 文件形式分发到 CDN 上，不需要 Node.js 支持，就是非常典型的 CSR。



资源加载完成后（注意：只有一个 bundle，一次性吐出），通过 React 中的 render API 进行页面渲染。优点是不需要服务端接入，简单，对于性能要求不高的

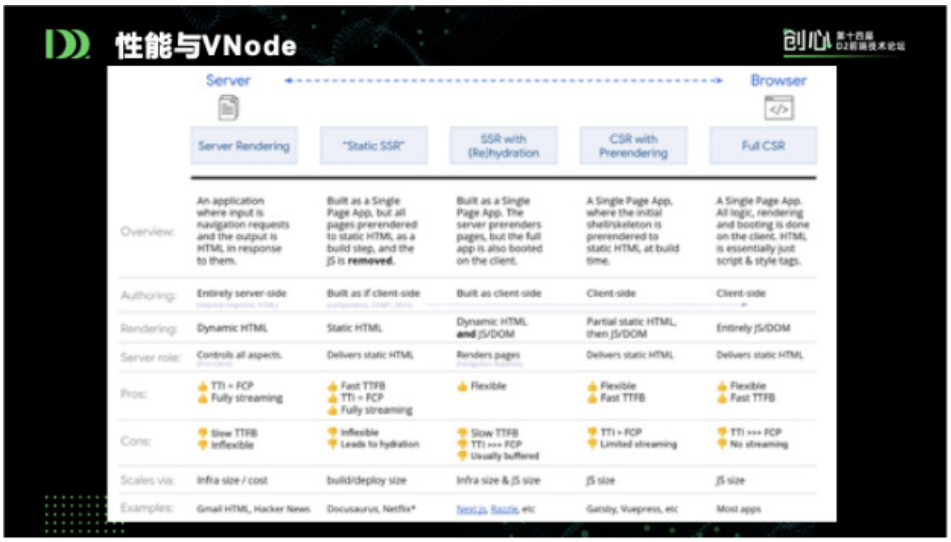
页面是非常合适的。中后台应用大多是 CSR，优化也都是打包环节玩。

服务器端渲染（SSR），简单理解就是 html 是由服务端写出，可以动态改变页面内容，即所谓的动态页面。早年的 php、asp、jsp 这些 Server page 都是 SSR 的。但基于 React 技术栈，又有些许不同，server bundle 构建的 时候，要吐多少模块，是 server 端决定的。client bundle 和之前一样，差别在于这次是 hydrate，而非 render。



hydrate 是 React 中提供在初次渲染的时候，去复用原本已经存在的 DOM 节点，减少重新生成节点以及删除原本 DOM 节点的开销，来加速初次渲染的功能。主要使用场景是服务端渲染或者像 prerender 等情况，所以在图中 hydrate 之后才是 tti 时间。

如果想全局了解 CSR 和 SSR，共分 5 个阶段，参考下图。

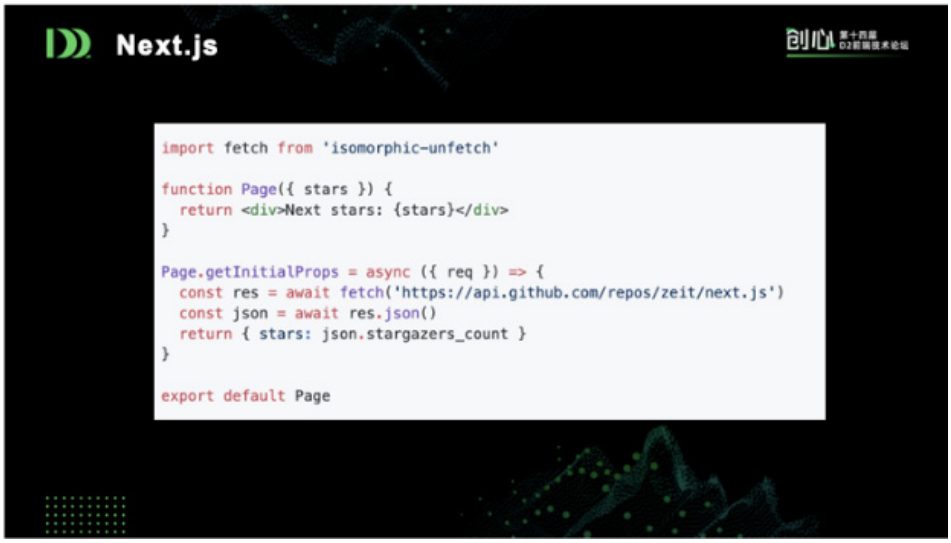


纯服务端渲染和纯客户端渲染是 2 个极端，React SSR 是属于中间的，这种服务端吐出的粒度是可以根据业务来控制的，可以服务端多一点，性能会差，也可以服务端吐出刚好够首屏的数据，其他由客户端来处理，这种性能会好很多。Static SSR 和预渲染的 CSR 也是特定场景优化的神器。

最佳写法

了解了 CSR 和 SSR 的区别，下面我们看一下最佳写法是如何演进的。业内最好的实现大概是 next.js 了，抛开负责度不谈，单就写法来说，它确实是最合理的。

既然是基于 React 做法，核心肯定以 Component 为主，在 Component 上扩展静态方法用于接口请求是很好的实践。写法如下。



早年写过 bigpipe 相关事项，其中模块成为 biglet，它的作用是获取接口，结合 tpl 生成 html。

```
module.exports = class MyPagelet extends Pagelet {
  constructor () {
      super()
      this.root = __dirname
      this.name = 'pagelet1'
      this.data = {
          is: "pagelet1测试",
          po: {
              name: this.name
          }
      }
      this.domid = 'pagelet1'
      this.location = 'pagelet1'
      // this.tpl = 'tpl/p1.html'
      this.delay = 4000
  }

  // 获取数据
  fetch() {
    return this.sleep(this.delay)
  }

  // 模拟sleep
  sleep(time) {
    return new Promise((resolve)=> setTimeout(resolve, time))
  }
}
```

biglet 的生命周期如下。

```
before
.then(self.fetch.bind(self))
.then(self.parse.bind(self))
.then(self.render.bind(self))
end
```

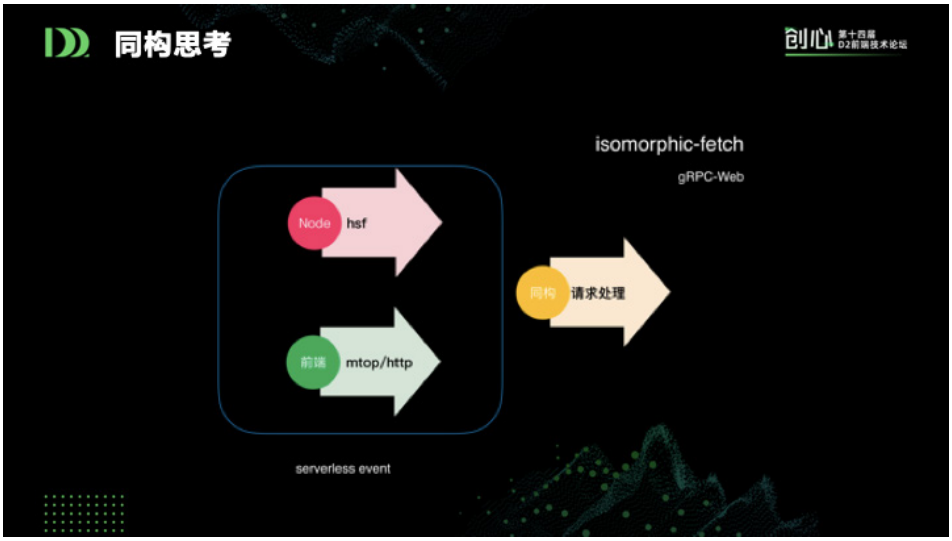
fetch 是获取接口数据，parse 解析数据，最终赋值给 data。render 是模板引擎编译的函数。每个函数的返回值都约定是 promise，便于做流程控制。

很明显，biglet 和 Component 是异曲同工的，而 fetch 是对象上的方法，必须实例化 biglet 才能调用，而 next 的做法 getInitialProps 是静态方法，不必实例化，内存和复用性上都是非常好的。

这点在 egg-react-ssr 项目技术选型调研期，我们就已经达成一致了。问题是，next 只支持 SSR，如何能够更好的支持 CSR？做到真正的组件级别的同构。于是，基于这种写法，通过高阶组件进行包装，轻松实现了 CSR，核心代码如下。

既然写法上统一了，那么，我们还能进一步进行优化么？核心点在网络获取部分。我们能看到的 gRPC-web 或 isomorphic-fetch，分别实现了：1）RPC 和 http

的约定，2）CSR 和 SSR 中 fetch 统一。给我们带来的启示是在 getInitialProps 里，我们还可以做更多同构的玩法。



在 webpack 打包构建 server bundle 了的时候会注入 isBrowser 变量。

```
const plugins = [
  new webpack.DefinePlugin({
    '__isBrowser__': false //eslint-disable-line
  })
]
```

以此来区分 CSR 和 SSR 做同构兼容就更简单了。

```
Page.getInitialProps = async (ctx) => {
    if (__isBrowser__) {
        // for CSR
    } else {
        // for SSR
    }
}
```

以上做法，都是我们基于 https://github.com/ykfe/egg-react-ssr 提炼出来的实践，这些都是 SSR-spec 的基础。

FaaS 和 SSR 如何结合

有了 egg-react-ssr，确立了最佳写法，接下来就是结合 FaaS 做好集成。

对比一下，getInitialProps 的参数和 FaaS 函数的参数都有 context，这个是非常好的

- egg-react-ssr 中 getInitialProps 参数 ctx 是 egg/koa 的 ctx
- FaaS 函数的参数 context 挂了各种函数、内存、日志、client 相关的信息，把 response 信息挂上理论上也是可行的，只是处理位置问题，参见 https://docs.aws.amazon.com/lambda/latest/dg/Nodejs-context.html

解法：给 context 扩展 SSRRender 方法。

为了演示 Serverless 下渲染层实现原理，下面会进行简要说明。在 Serverless 云函数里，一般会有 server.yml 作为配置文件，这里以 lamda 为例子。

通过这个配置，我们可以看出函数 app.server 对应的 http 请求路径是 '/'，这个配置其实描述的就是路由信息。对应的 app.server 函数实现如下图。

```
module.exports = async function (ctx) {
  ctx.body = await ctx.ssrRender(ctx, 'Page', {
    serverJs: path.join(__dirname, './dist/Page.server.js')
  })
}
```

通过提供 ctx.SSRRender 方法，读取 dist 目录下的 Page.server.js 完成服务端渲染。

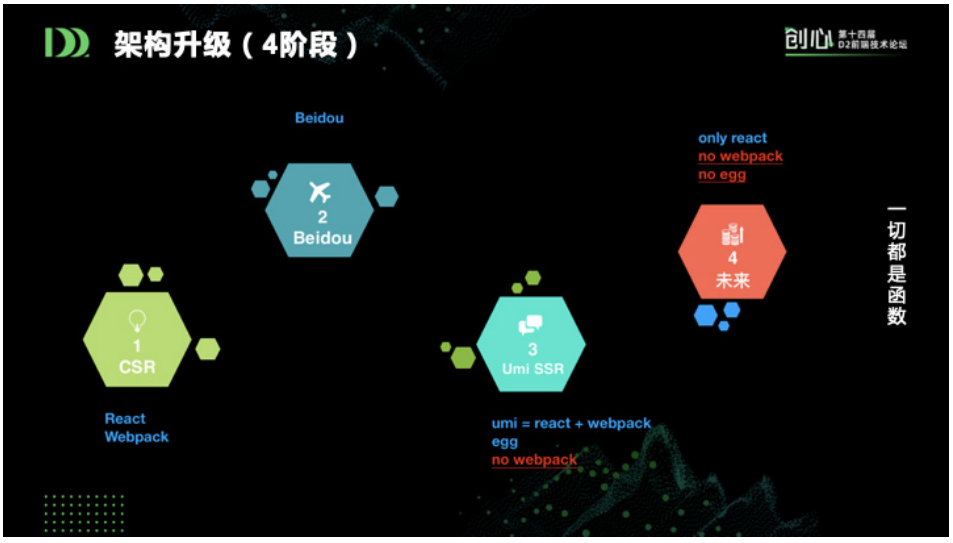
核心要点：

- SSRRender 方法比较容易实现
- 采用类似 Umi SSR 的方式，将源码打包到 Page.server.js 文件中
- 在发布的时候，将配置，app.server 函数和 Page.server.js 等文件上传到 Serverless 运行环境即可

架构升级 4 阶段

纵观 SSR 相关技术栈的演进过程，我们大致可以推出架构升级的 4 阶段。

- CSR，很多中后台都是这样的开发的，最常见
- 其次是阿里开源的 beidou，基于 egg 做的 React SSR，这是一个集成度很高的项目，很好用，难度也很大
- Umi SSR 是基于 egg-react-ssr 上演进出来的，在 umi 之上，用户不需要关心 webpack，但打包后的代码返回的是 stream，这点抽象，云谦做的非常到位，对于开发者来说，还是需要自建 Node web server 的。
- 在 Serverless 里，具体怎么玩是需要我们来创造的。



对照上图，说明如下

1. 在 CSR 中，开发者需要关心 React 和 Webpack
2. 在 SSR 中，开发者需要关心 React、Webpack 和 Egg.js
3. 在 Umi SSR 同构中，开发者需要关心 React 和 Egg.js，由于 Umi 内置了 Webpack，开发者基本不需要关注 Webpack

4. 在 Serverless 时代，基于 FaaS 的渲染层，开发者需要关心 React，不需要关心 Webpack 和 Egg.js

在这 4 个阶段中，依次出现了 CSR 和 SSR，之后在同构实践中，对开发者要求更高，甚至是全栈。所有这些经验和最佳实践的积累，沉淀出了更简单的开发方式，在 Serverless 环境下，可以让前端更加简单、高效。

组件即函数

对于组件写法，我们继续抽象，将布局也拉出来。

```
function Page(props) {
  return <div> {props.name} </div>
}

Page.fetch = async (ctx) => {
  return Promise.resolve({
    name: 'Serverless side render'
  })
}

Page.layout = (props) => {
    const { serverData } = props.ctx
    const { injectCss, injectScript } = props.ctx.app.config
    return (
      <html lang='en'>
        <head>
          <meta charSet='utf-8' />
          <meta name='viewport' content='width=device-width, initial-scale=1,
shrink-to-fit=no' />
          <meta name='theme-color' content='#000000' />
          <title>React App</title>
          {
            injectCss && injectCss.map(item => <link rel='stylesheet'
href={item} key={item} />)
          }
        </head>
        <body>
          <div id='app'>{ commonNode(props) }</div>
          {
            serverData && <script dangerouslySetInnerHTML={{
              __html: `window.__USE_SSR__=true; window.__INITIAL_DATA__
=${serialize(serverData)}`
```

```
            }} />
          }
          <div dangerouslySetInnerHTML={{
            __html: injectScript && injectScript.join('')
          }} />
        </body>
      </html>
    )
}

export default Page
```

layout 目前看只有第一次渲染有用，但做 Component 抽象是可以考虑的。现在是首次渲染模式，以后不排除递归组件树的方式（结合 bigpipe 可以更嗨），那时 layout 还是有用的。

这样看来，render、fetch、layout 是函数，结合 Serverless.yml（f.yml）配置，能否将他们放到配置里呢？

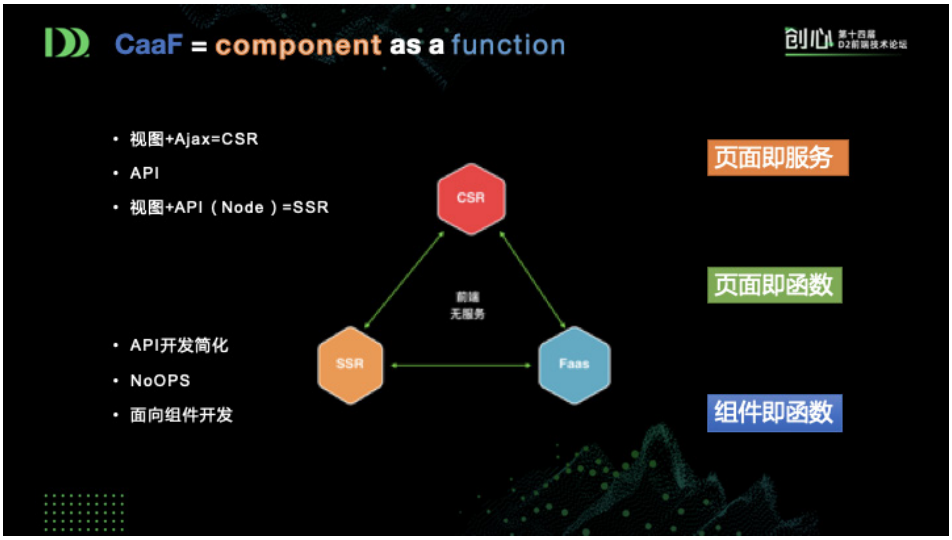
路由由 f.yml 的配置文件中，所以在 f.yml 增加 render 配置扩展，具体如下。

```
functions:
  home:
    handler: index.handler
    render:
      - Component: src.home.index
      - layout: src.home.layout
      - fetch: src.home.fetch
      - mode: SSR | CSR(默认 SSR)
      - injectScript(自己改 loyout 更好)
          - runtime~Page.js
          - vendor.chunk.js
          - Page.chunk.js
      - injectCSS
          - Page.chunk.css
      - serverBundle: Page.server.js
    events:
      - http:
          path: /
          method:
            - GET
  news:
    handler: index.handler
    render:
```

```
    - Component: src.news.index
    - layout: src.news.layout
    - fetch: src.news.fetch
    - mode: SSR | CSR(默认 SSR)
  events:
    - http:
        path: /
        method:
          - GET
```

元彦提了一个非常好的命名：组件即函数。最贴切不过。

将渲染、接口请求、布局分别拆成独立函数，放到配置文件里。如此做法，可以保证函数粒度的职责单一，对于可选项默认值也更友好。比如没有接口请求就不调用 fetch，或者没有布局使用默认布局。



这里再拔高一下，CSR 和 SSR 写法一致了，这种写法可以和 FaaS 结合，也就是说写法上拆成函数后，前端关心的只有函数写法了，使得面向组件开发更容易。

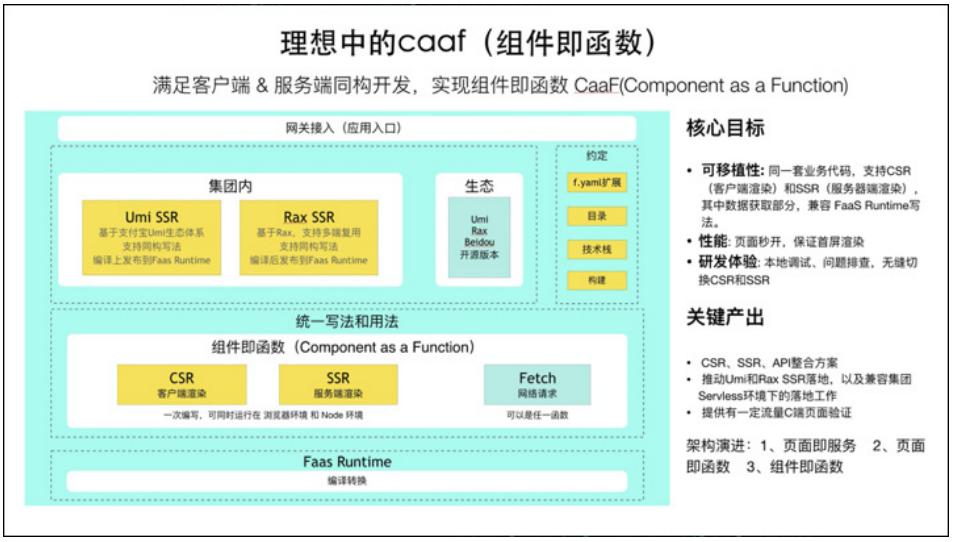
我们在抽象一下，提炼 3 重境界。

- 组件即函数，就是上面将的内容。

- 页面即函数，对开发者而言，其实页面的概念不大，第一次渲染布局，然后 entry 执行而已，如果抛开构建和配置细节，页面就是第一个组件，即函数
- 页面即服务，是我之前提的概念，每个页面对应一个 Node 服务，这样的好处是避免服务器雪崩，同时可以降低页面开发复杂度。

对于页面即服务来说，一个 FaaS 函数提供 http，天然就是独立服务，在基建侧，网关根据流量决定该服务的容器个数，完美的解决了页面渲染过程的所有问题，也就是我们只需要关注组件写法，组件写法又都是函数。所以，组件即函数，是 Serverless 渲染层最好的概括。

下面这张图很好的表达了组件即函数的核心：统一写法和用法。



制定规范之前，定位还是先要想明白的。

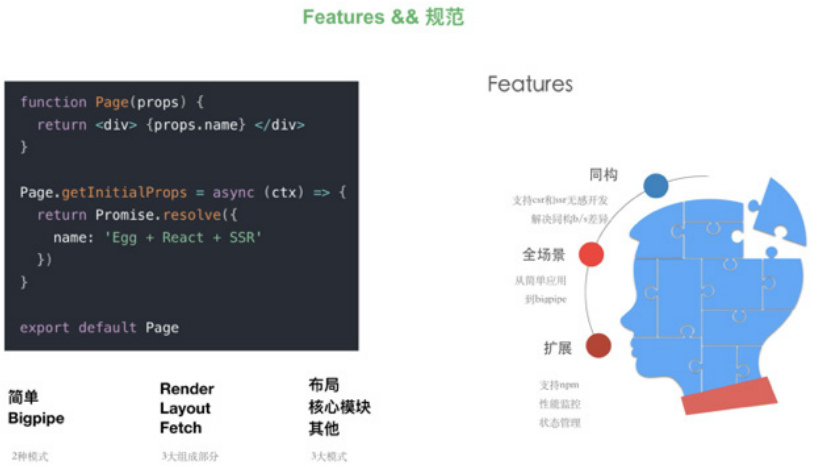
- 在 FaaS rutime 之上，保证可移植性
- 通过 CSR 和 SSR 无缝切换，可以保证首屏渲染效率
- 由于面向组件和配置做到轻量级开发，可以很好的结合

统一写法和用法是件约定大于配置的事儿，约定才是最难的，既要保证功能强

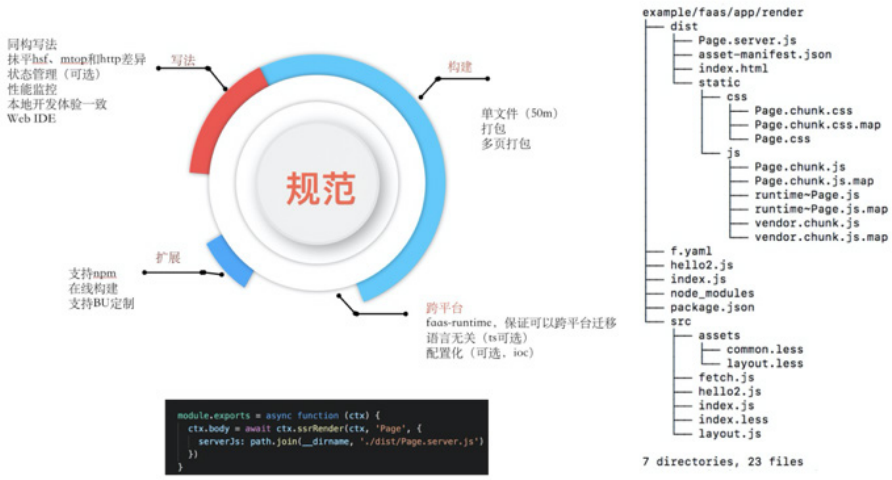
大，还要写法简单，又要有扩展性。显然，这是比写代码更有挑战的事儿。

SSR 规范

统一写法，上一小结已经讲过了。



接下来就是规范相关的周边，如下图。



构建，扩展，跨平台，以及目录都需要约定。

SSR-spec 规范主要定义 SSR 特性，组件写法、目录结构以及 f.yml 文件扩展的编写规范。目录结构待讨论，例如新增功能的 API 与删除功能的 API 理论上应该放在一个 project 当中，此时应该在 src 目录下建立不同的文件夹来隔离不同函数的模块 .

```
├── dist // 构建产物
│   ├── Page.server.js // 服务端页面 bundle
│   ├── asset-manifest.json // 打包资源清单
│   ├── index.html // 页面承载模版文件，除非想换成传统的直接扔一个 html 文件部署的方式
│   └── static // 前端静态资源目录
│       ├── css
│       └── js
├── config // 配置
│   ├── webpack.js // webpack 配置文件，使用 chainWebpackConfig 方式导出，非必选
│   └── other //
├── index.js // 函数入口文件
├── f.yml // FaaS 函数规范文件
├── package.json
├── src // 存放前端页面组件
│   ├── detail // 详情页
│   │   ├── fetch.js // 数据预取，非必选
│   │   ├── index.js // React 组件，必选
│   │   └── layout.js // 页面布局，非必选，没有默认使用 layout/index.js
│   ├── home // 首页
│   │   ├── fetch.js
│   │   ├── index.js
│   │   └── layout.js
│   └── layout
│       └── index.js // 默认的布局文件，必选，脚手架默认生成
└── README.md //
```

命令用法

```
$ ssr build
$ ssr deploy
```

生成的 dist 目录结构如下。

- dist
  - funcName
    - static
      - clientBundle.js
      - js
      - css
      - images
    - serverBundle.js

构建命令

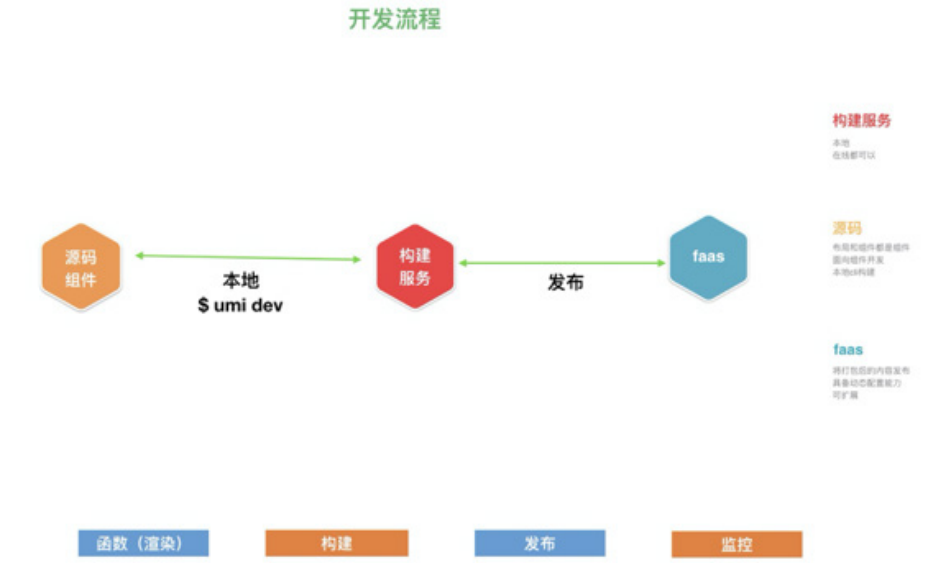
```
$ ssr build
$ ssr build --spa
$ ssr build hello
$ ssr build hello2
```

Serverless 集成步骤，通用方案集成

```
$ ssr xxx
$ Serverless deploy // f deploy
```

这里以 Umi 为例，开发过程分 3 个步骤。

1. 本地源码组件开发，然后通过 umi dev 完成构建
2. 如果是线上，可以走线上构建服务（webpack 打包），如果需要修改，可以走
3. 构建后的产物结合 FaaS 函数，直接发布到 Serverless 平台上



规范里还有很多点也是有思考的，比如多组件支持是基于 bigpipe 的方式，首先写入 layout 布局，然后处理多个组件的组合逻辑，最终 res.end 即可。另外，组件上如果只有 fetch 方法，没有 render 方法也是没有问题的。写法有 2 种，Component 的值是数组，即串行方式。Component 的值是对象，即并行方式。限于篇幅，这里就不一一赘述了。参见 https://github.com/ykfe/ssr#specification。

落地实践、性能和未来思考

打包与构建

Umi 的实现是非常巧妙的，核心在于构建后的 server bundle 返回值是 stream，解耦了对 web 框架的依赖。在当时还没有实现更好的，所以以 Umi 为例。

构建产物，各个文件大小的说明，有 2 种方式。

- Node_modules 打包到 bundle，对 FaaS runtime 无依赖。
- Node_modules 不打包到 bundle，放到 FaaS runtime 里。

这 2 种方式打包大小可以接受，性能上第二种会更好一点，但没有差很多。

快速切换 CSR 还是 SSR

前面说了写法上的统一，在工程实践中，可以在配置里，直接设置 type 快速切换 CSR 还是 SSR。在公司内部，还可以通过 diamand 配置下发的方式进行动态控制。

其实 SSR-spec 规范里，还做了更多扩展 .

```
// 检查 query 的信息或者 url 查询参数或者头信息
conf.mode = req.query.SSR || req.headers[‘x-mode-SSR’];
```

容灾打底方案

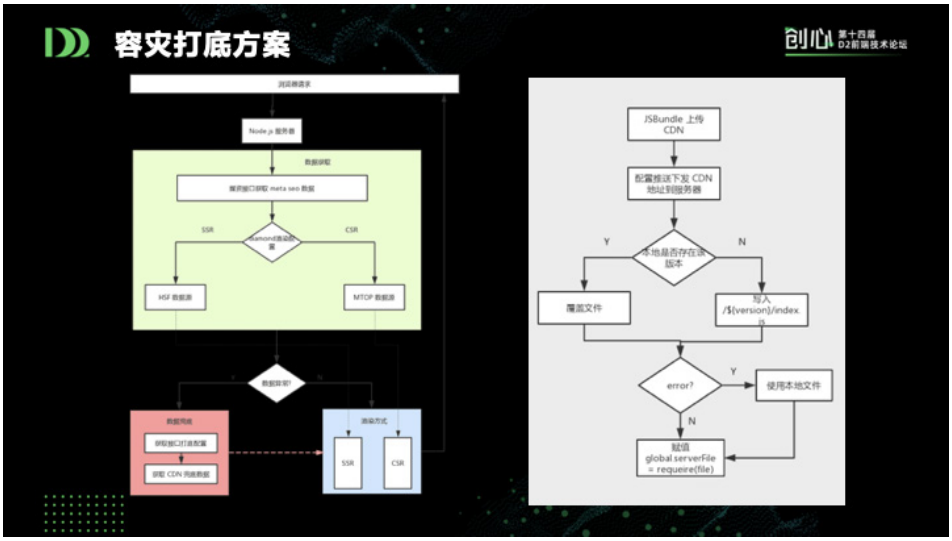
简单说，3 层容灾

- Node SSR 优先，Node 调用 hsf。
- Node CSR 通过 diamind 可以快速切换，html 是 Node 吐出的，前端走 MTop 请求。
- 当 Node 服务挂掉，走 CDN 上的纯 CSR，前端 MTop 请求。

该流程有以下优点 :

- 构建方式一致
  - 服务端 / 客户端文件构建方式一致
- 发布方式一致
  - 发布方式一致，统一发布到 CDN，前端资源可以使用 CDN 加速
- 无需服务端发布
  - 组件代码变动统一使用 diamond 下发版本号，无需服务端发布
- 及时生效
  - diamond 配置下发后可及时生效

如图。



性能优化

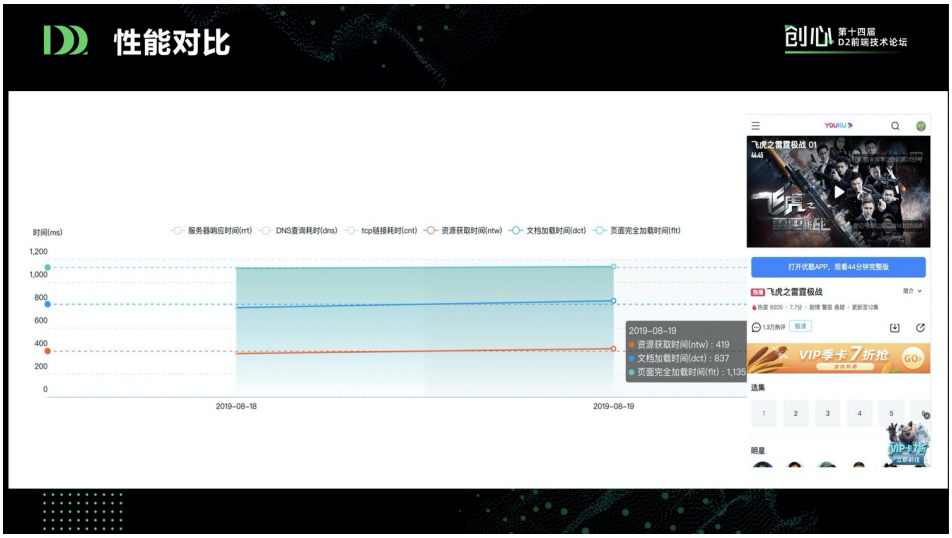
性能优化的要点

- 控制 SSR 中 server 端占的比例，性能和体验是鱼和熊掌不可兼得。看业务诉求。
- 能缓存的尽量缓存，和 BFF 应用一样。

这里举个例子，对接口字段瘦身，就可以渠道意想不到的效果。

性能对比

优酷 PC 的 React SSR 性能很好，从 v1（Bigpipe+jQuery）到 v2（React SSR），性能逐步提升。PC 页面首屏渲染降到 150ms、播放器起播时间从 4.6 秒优化到 2 秒。H5 站上了 React SSR 后，性能提升 3 倍，H5 唤端率提升也极其明显，头条短视频唤端率由 5.68% 提升到 9.4%，环比提升 65%。单机性能 qps 从 80 提升 150+（压测最高可以到 300 左右）。



淘宝的 Rax SSR 也性能优异，以一个带数据请求的真实 Rax SSR 应用为例，性能对比数据显示：WIFI 下，SSR 的首屏呈现时间相比 CSR 提升 1 倍；弱网环境下，SSR 相比 CSR 提升约 3.5 倍。

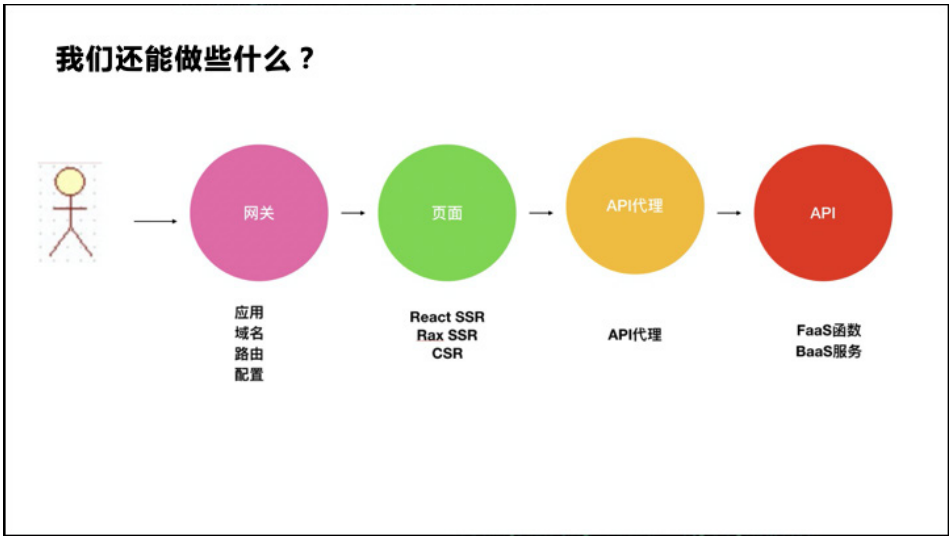
参见水澜 Rax SSR：重塑 SSR 应用的开发体验

SSR Demo 地址：https://Rax-demo.now.sh/SSR/home

CSR Demo 地址：https://Rax-demo.now.sh/CSR/home

未来思考

从用户访问页面流程，具体如下。



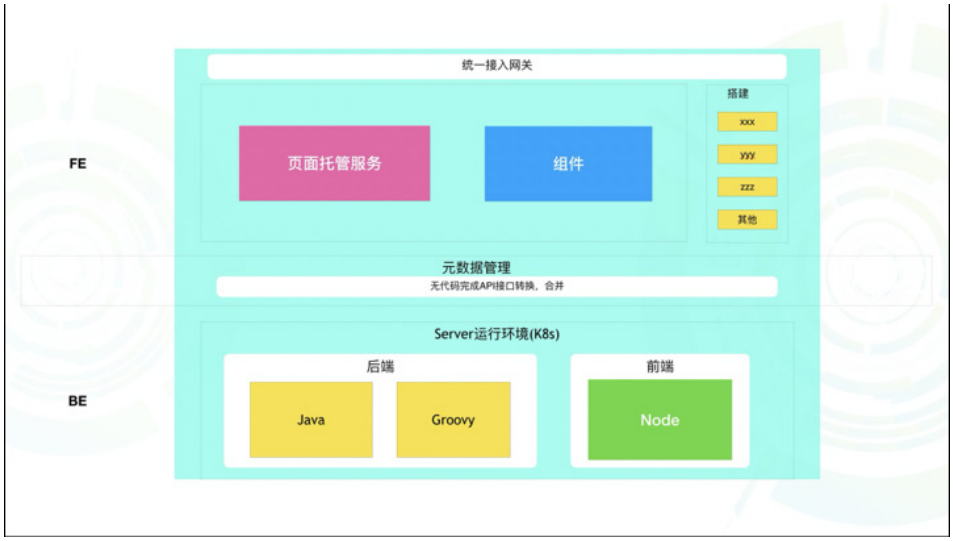
要点

- 应用网关，肯定是要有的，毕竟要挂域名，反向代理等
- 页面，已经可以放到 FaaS 上，FaaS 之上的 http 网关可以满足基本需求，离应用级别的还差很多。
- API 代理，这个基于 Node FaaS 函数非常容易实现
- API，调用后端服务提供 API，比如访问 db 等，基于 Node FaaS 函数也是非常容易实现

FaaS 细化到函数粒度，在管理上会有巨大挑战，在架构上也需要重新设计。

- 具有前端特色的函数管理：比如 API、Page 等类型，path，域名，甚至是具体的应用设置
- 页面、组件和网关联动，让开发更简单快速
- 周边，比如监控，统计，数据等等

这里尝试一张图来表示一下前后端的 Serverless 时代的分工。

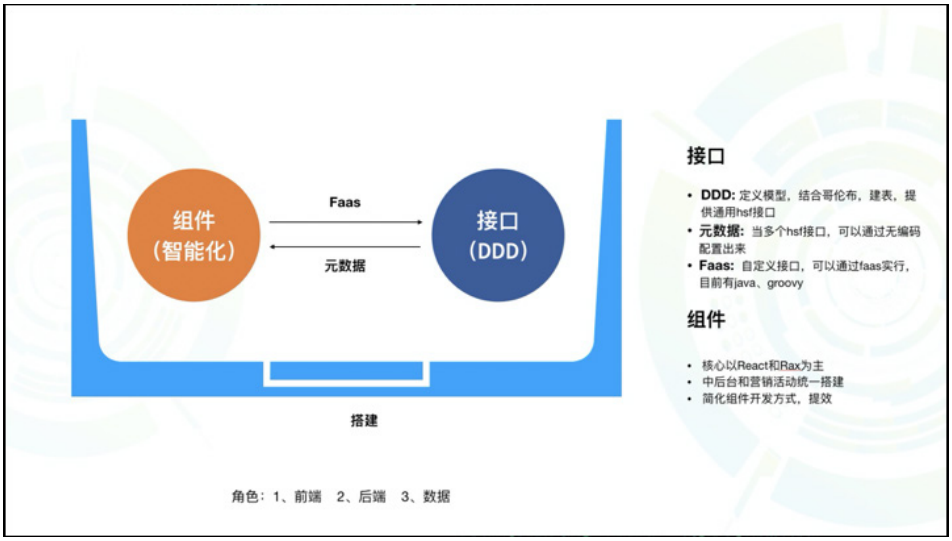


- 统一接入网关是必须的，主要是处理页面和域名的接入。
- 对于页面进行抽象，围绕组件和搭建来展开，通过在线 | 本地构建，最终放到页面托管服务中。有了页面托管，才是万里长城的一小步。
- 接下对 API 进行拆分，这也是组件组成里重要的部分。

围绕搭建，可以想象到的是组件和接口的抽象。组件除了智能化我能想到的很少，智能化在计算上也能弹性玩就更有想象力了。对于接口可以再细分

- 直接操作表，虽不推荐前端做，但确实是必备能力。
- 通过配置来生成，即元数据管理，对于已有 API 进行包装，逻辑编排是非常好的。

- 如果都不满足，自己基于 FaaS 函数定制就好了。



搭建本身是提效的，组件和接口都能提效，对于前端的价值是尤其大的。前端 Serverless 专项里，也是有逻辑编排组的，原因大抵如此。

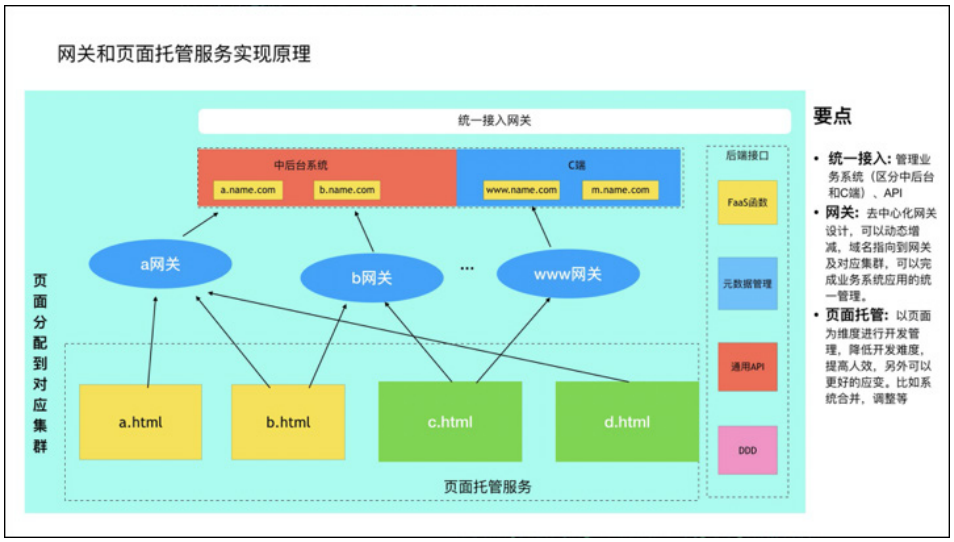
下面再解释一下页面托管服务实现原理。其实这是页面即服务的升级版，以前每个页面对应一个 Node 服务，这就导致很多 Node 服务的运维成本非常高，有了 Serverless，依然还是一个页面对应一个 FaaS 函数，但函数的扩容是 Serverless 基建做的事儿。也就是说页面托管，其实是基于页面的 FaaS 函数的管理。垂直到页面管理，一切都自动化，会让开发更简单。

目前每个 FaaS 函数是可以提供 http 透出地址的，但这个地址不具备定制能力，所以在页面托管服务之上有一层应用网关是必须的。那么，应用网关和页面页面托管服务之间如何联动呢?

- 最外层，统一接入网关里做应用管理，每个应用都有对应的域名和子应用网关，二者进行绑定 根据流浪，子应用网关也可以自动扩缩容。
- 在应用设置里，管理子应用网关包含的 path 和页面，提供反向代理相关的基础功能即可。

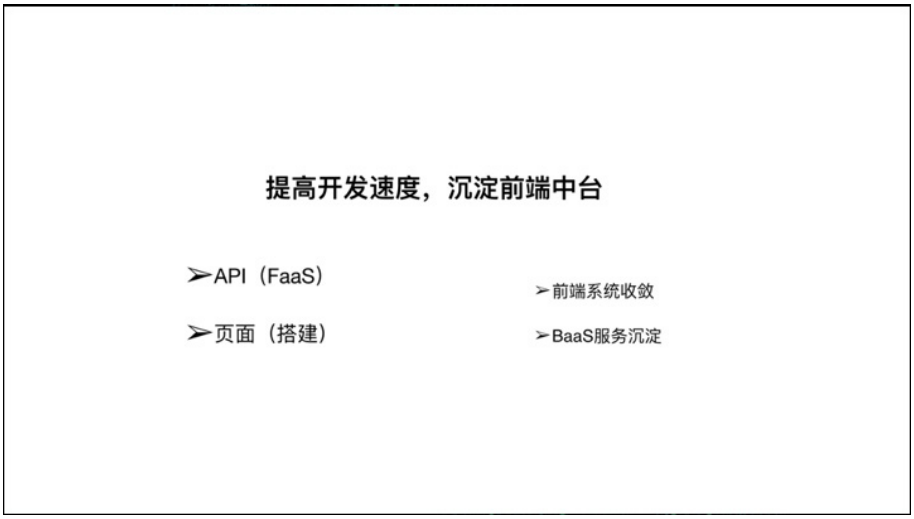
- 设置完子应用包含的页面之后，系统具备将对应页面同步到子应用网关的能力，并且当页面更新的时候能够自动同步，类似于 etcd/consul 等服务发现同步功能。

有了这部分设计，开发者只需要关注页面的编写就好了。比较上面的 3 点配置在系统中并不经常做。



基于上面的设计，目的是提高开发速度，沉淀前端中台，具体好处如下。

- 中后台能够一定程度的收敛到一起，配置化
- 页面和系统分离，应变能力更强，结合微前端可以有更多想象力
- 所有开发聚焦到页面维度，能够更好的提效，比如组件沉淀，智能化，搭建等都可以更专注。
- 很多后端服务也能够很好的沉淀和复用，这和后端常说的能力地图类似，将 BFF 聚合管理，简化开发



关于未来，我能想到的是

- 组件：写代码或智能化生成
- 页面：配置出来
- 系统：配置出来

只需要组件级别的函数的轻量级开发模式里，必然会简化前端开发方式，提供人效，最终实现技术赋能业务的目的。在“大中台，小前台”的背景下，贡献前端应变能力。

总结

在《2019，如何放大大前端的业务价值?》一文中，我曾提过："前端技术趋于成熟，不可否认，这依然是个大前端最好的时代，但对前端来说更重要的是证明自己，不是资源，而是可以创造更多的业务价值。在垂直领域深耕可以让大家有更多生存空间，但我更愿意认为 Serverless 可以带来前端研发模式上的颠覆，只有简化前后端开发难度，才能更好的放大前端的业务价值。"

致敬所有为 Serverless 付出的同仁们!

Serverless 函数应用架构升级

作者：张挺

本次 D2 分享的话题叫《Serverless 函数应用架构升级》，主要讲述的是阿里集团内部从传统一步步抽离出 midway-faas 框架，并赋予其核心能力，解决用户诉求的一些思考和实践。

社区的 Serverless 在不断升温，今年成了阿里经济体前端委员会的四大方向之一，给了前端非常大的机遇和挑战，业务在不断尝试之余，也逐步沉淀出了一套可迭代，可维护，可扩展，可复用的开发模型。如今云厂商不断发力，无声的战争正在打响。

现在，随着 Serverless 的深入人心，云厂商都在说，“我们在定义 Serverless”，而开发者都说”我们在做 Serverless“，用户都是“在用 Serverless", 人人都在往 Serverless 体系上靠，或多或少的沾点边，使得整个社区欣欣向荣。

而对于我们前端本身来说，引进 Serverless，也能在各个场景下带来帮助。目前 Serverless 体系，首选的是 FaaS + BaaS 模型，使用了 FaaS。

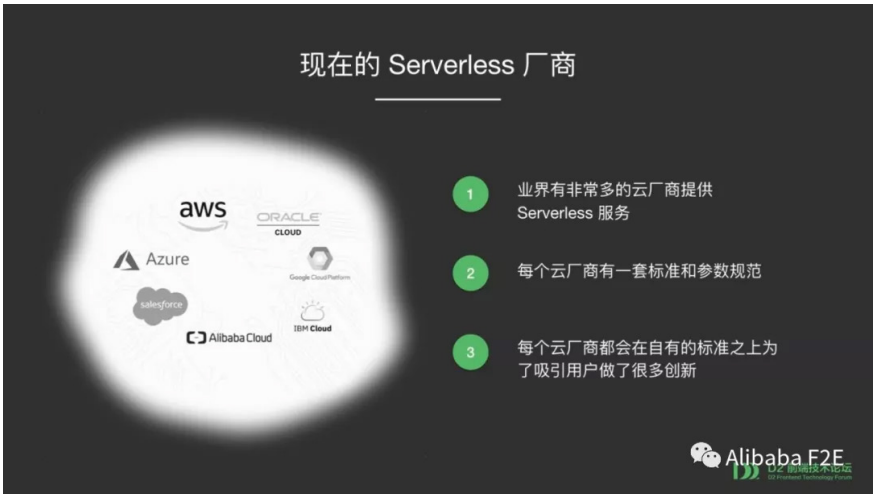
首先是更快的开发速度，基于时间驱动模型，能够更方便的进行开发和迭代，让

用户快速上线；其次是更加安全的隔离环境，用户再也不能，也不希望登录服务去排查问题；第三是按量付费，由于计费模型的变化，比传统的按实例付费，在访问率低的时候，更能降低成本；最后是弹性实例，在部署服务时，不再考虑峰值等情况，不需要提前预估流量，预先准备实例。

这些优势，看到了一个新的契机，让前端能够减少开发成本，弱化运维，同时又能更多的关注数据，真正的向着后端、一体化发展，也为跨技术栈研发效能提升提供了可能。

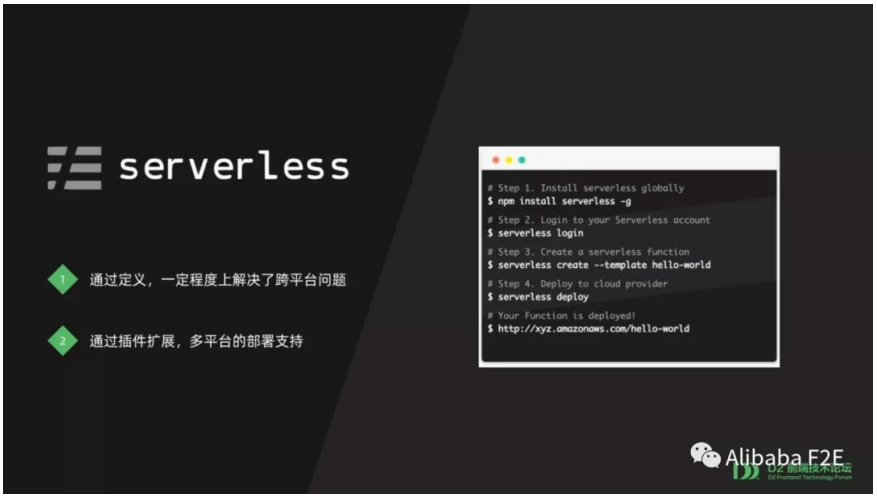
社区背景

现有的社区，有非常多的云厂商提供函数服务。不管是 Amazon，微软，还是谷歌，以及国内的阿里云和腾讯云，都推出了自己的云函数，同时，每个云厂商都有着一套自己的标准和规范，同时，在这些标准之上，云厂商为了吸引用户，更快的占领市场，做了很多创新，阿里云的 CustomRuntime，以及前段时间腾讯云推出的 Service 2.0 概念，就是比较好的例子。



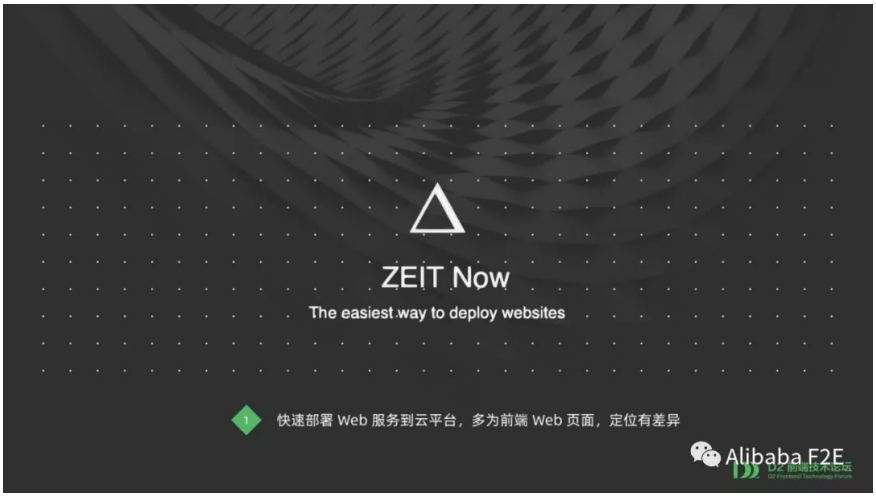
作为普通用户，如果想快速的使用 Serverless，在这么多平台面前，光是选择就得斟酌一番，在各个平台之间反复对比，同时，如果想多尝试几个平台，就会发现每个平台的规范，代码行为，启动方式，触发器都不尽相同，对于用户来说，能有跨平台的方案，可以在不同的厂商之间做一些迁移，互调等有意思的尝试。

在社区中已经有一些能够帮助我们更好的部署到多云的工具链，Serverless Framework 就是其中做的不错的代表，已经有不少云厂商通过开发插件接入了 serverless 工具链。它通过 serverless.yml 文件，定义了多云的能力，使用云厂商开发的插件，一定程度上解决了跨平台的问题。



这其中没有很好的解决云厂商之前代码层面的不一致（因为是云厂商自己做的），第二，这些插件是云厂商自己提供，如果某些厂商不做支持，那么能力就会缺乏。

而另一家 ZEIT，推出了 now 系列工具，以轻量简单快速闻名，他们的体验非常棒，目前支持部署到他们接入的平台，示例基本以前端项目为主，和我们 FaaS 本身的实践有所区别，定位也有一些差异。



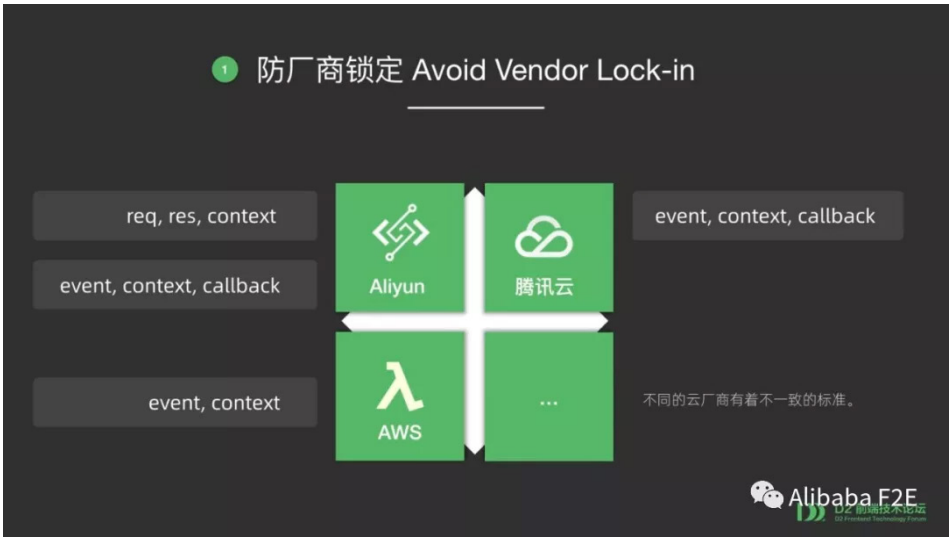
在调研完社区的生态之后，我们发现，没有非常契合我们的诉求的产品，我们觉得自己设计一套符合我们定位，且能够满足场景，又面向未来的框架。经过讨论，我们将框架的特性归纳、总结出以下四点：

- 第一，需要能够防厂商锁定，由于 FaaS 目前没有成熟的标准，导致社区平台割裂，同时因为集团内部也有多套环境的需求，所以我们决定把这个特定放到第一位。
- 第二，是灵活性，我们称之为 Flexibility，传统的 FaaS 部署模型和计费模型，在某些场景下，成本和性能都有所欠缺，同时，在扩展性、复用性上也不够灵活。
- 第三，在团队规模越来越大的今天，标准，复用和扩展性已经变的越来越重要，如何解决这些问题，我们也进行了深入思考。
- 最后，是一些实际的诉求，比如在多个函数之前，复用一些逻辑，统一埋点，监控等，我们需要在这一方面做一些扩展，目前的社区平台无法满足，我们需要设计出一套生命周期，来完善整个体系，这样内外的逻辑保持一致，也可以把我们的实践输出出去。

防厂商锁定

前面我们提到，不同的云厂商提供的云函数服务，标准不同，写法不同，特别是出入参，会有一些差异，给我们的同学带来了不少困扰。参数这一块这是由不同的运

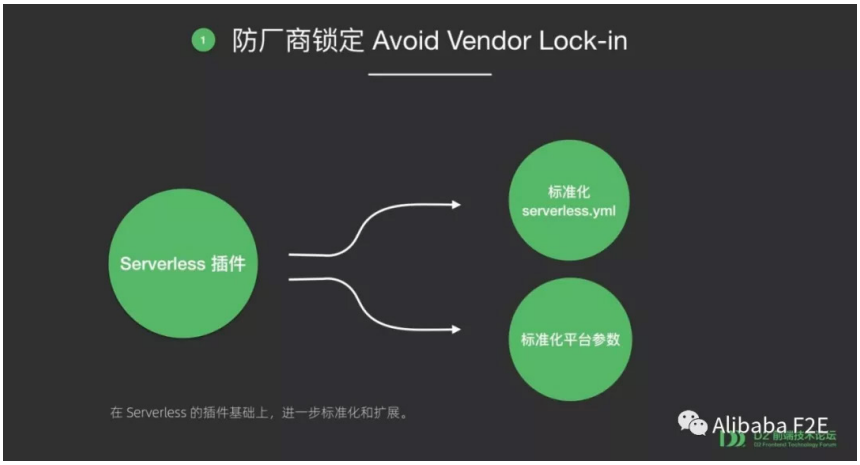
行时决定，像阿里云，就会有 req, res, context 以及 event, context, callback 两种写法，腾讯云和 aws 也有一些异步处理以及参数个数上的差异。



社区的 Serverless 由于起步比较早，各家云厂商都对它有一定程度的支持，通过 serverless.yml 来定义每个平台的配置、资源、能力，前面也提到，由于是社区化，各家云厂商对插件的支持力度不一，更多的云厂商还是会着重打造自己的 cli 工具链，以及结合自家平台或者 IDE。

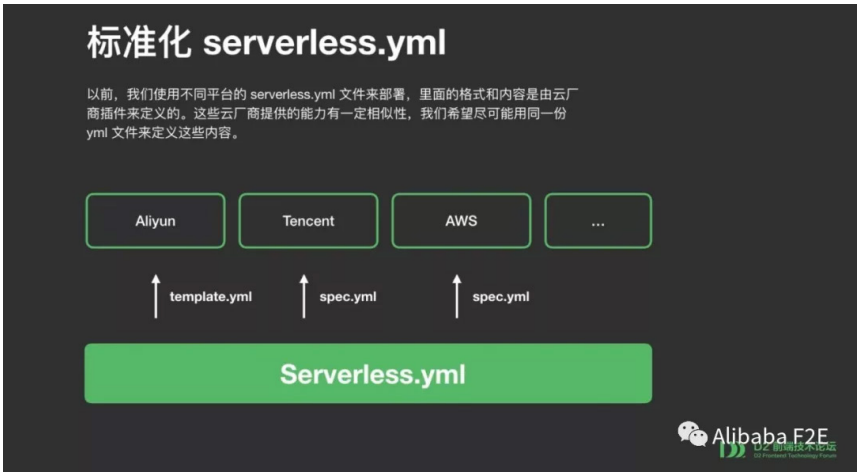
我们基于 Serverless Framework 的想法，希望能进一步做标准化和扩展，这样既可以复用社区生态，也可以进一步扩展我们自己的能力。由此，我们的方向分为两部分：

1. 进一步标准化 serverless.yml
2. 针对不同的云厂商，标准化代码层面的出入参



severless.yml 的标准化其实比较困难，这一块会参考现有的 serverless 体系，在他们没有做好的地方继续做规范和定义，我们的优势在于，会结合实际场景去考虑，也没有云厂商的包袱，相对来说可以自由一些。

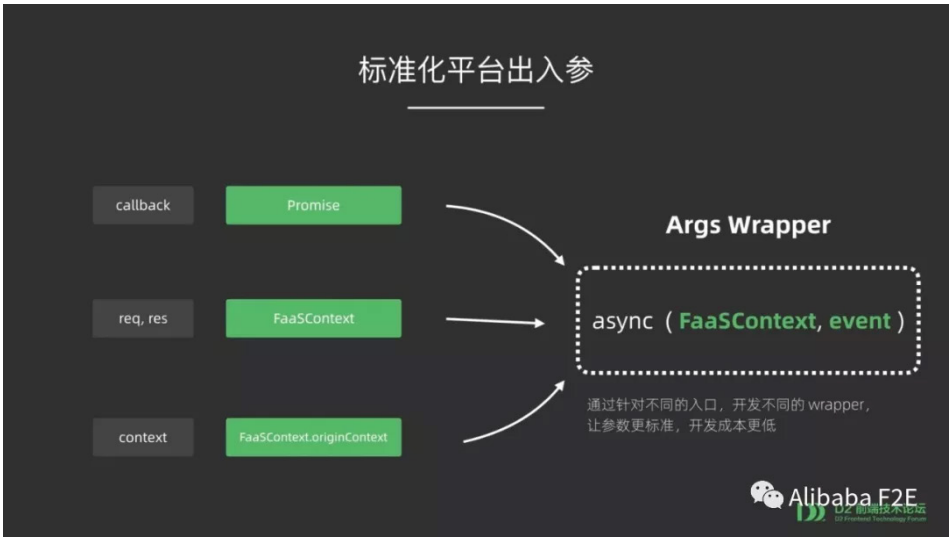
我们设想不同的云平台，都会有类似的能力，比如 http，api gateway，以及 timer，对象存储，消息队列等等，把这些常用的都定义为相同的字段，通过构建来生成出各个平台自己的字段，可以帮用户省下了解的时间。通过这样的变化，我们直接将一份 yml 文件，转变为不同平台自己的 yml 文件，这样缓冲层的设计，也屏蔽了后续因为平台的变化导致用户使用层面的差异性。



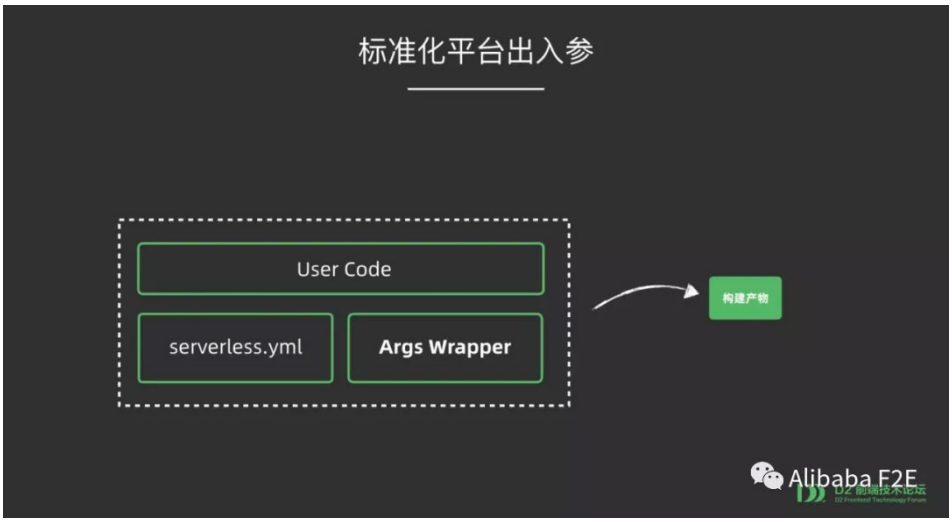
标准化出入参，只是代码上的包裹，相对来说就容易许多。

1. 把 callback 变为 Promise(async)
2. 把原有的入参做一些抽象，变为一个叫 FaaSContext 的请求上下文（包含了原有 event，context 的部分内容，更像 koa 模型）
3. 把原有的 context 变为 FaaSContext 上的属性（保留，后续可能有用）

这样做完之后，针对不同的平台，统一将上下文作为第一个参数传入（必选），event 第二个参数（可选），同时整个方法直接是 async 函数。

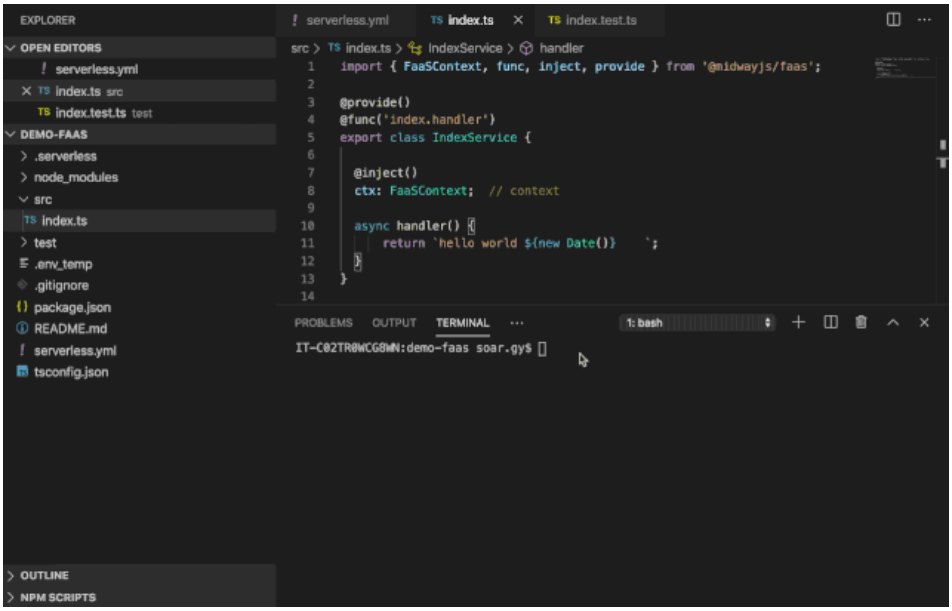


实现层面相对简单，我们把这些内容做成了参数包裹器 (parameter wrapper)，和之前的 yml 文件一起，结合用户代码的构建产物，分发部署到各个云平台，目前社区我们还只完成了阿里云和腾讯云，这种模型相当轻量，和原有的平台能力也不冲突，后续我们将会继续支持其他平台。



同时，在这些过程之中，我们采用了 interface 的定义方式，产出了几份针对不同平台的 yml 标准化定义，后续可以用它们快速的格式化，校验，减少用户开发成本。

下面是我们发布到多平台的演示，展示的是同一份代码文件，在不修改代码本身的情况下，发布到多云。



灵活性

我们也把灵活性是灵活性（Flexibility）作为第二个设想的能力，理想情况下，是部署模型的延伸，可以让函数在水平和垂直两个层面自由扩展。

我们以一个传统 Web 栈作为示例。

传统 Web 应用，路由会写在一个固定的文件中，每个路由一般会关联到一个 Controller 上，在函数的体系下，每个路由都将是一个 http 触发器，都将拆为一个独立的的函数，有着独立的代码、配置等。由于函数的热度不同，调用的频次也会不同。同时，如果原来的路由和 Controller，每个函数可以绑定多个路由。

所以说，FaaS 栈和传统 Web 栈有着很高的相似度，但是不同的是，这些函数都将部署到不同的容器中（隔离性），再加上平台本身宣传的那样，按调用量付费，看起来十分完美。

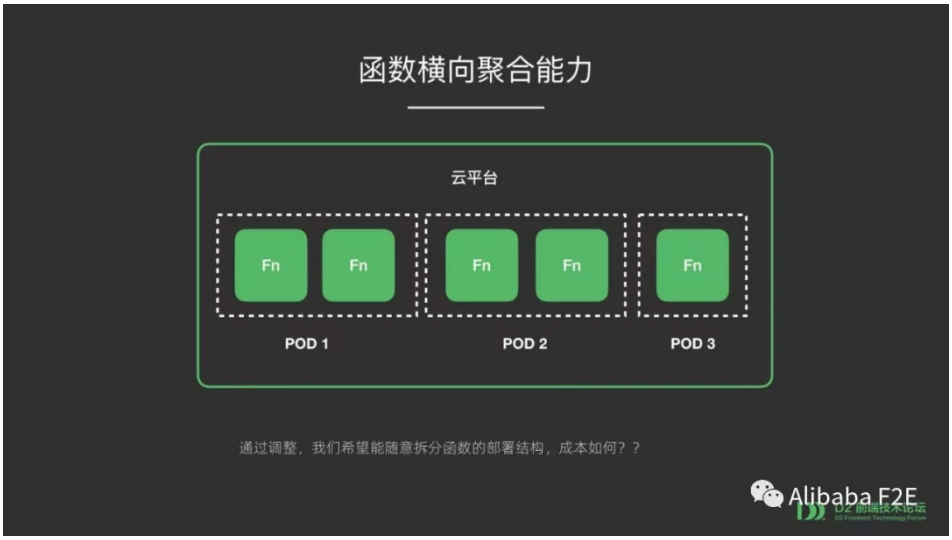
殊不知，函数整体的计费模型，除了传统的调用次数，还有一个 CU 的概念，一般来说，CU 会和容器本身的 CPU、内存消耗相关，调用次数容易计算，CU 要优化就比较困难。如果按请求调用量付费，每次调用都启动一个新的实例的话，创建实例本身的开销加上函数调用其实是比较浪费的。

为此，我们希望在某些场景下，能够尽可能的节省成本。这就产生了一些可能性，如果我们能复用函数实例，并发处理业务，或者共享一些资源，在一定程度上就能降低实例重复创建本身的开销。

我们设想，如果把多个请求在一个实例内处理，创建新实例的次数会变少，如果多个函数的逻辑在一个函数里完成，这样函数冷启动概率是不是就会降低，从而在另一个层面降低成本呢?

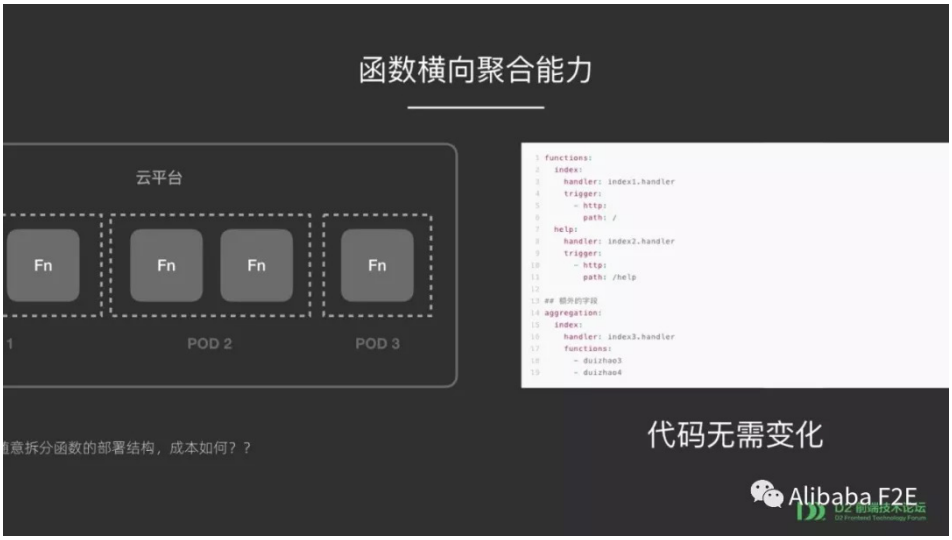
如果多个函数在一起，是否可以选择性聚合，假如某个函数请求量变高（热点函数），是否能够随时拆分出来呢?

这个答案是肯定的，经过实践，我们已经可以做到函数本身的随意组合部署。这就是我们设想的 Flexibility 的一部分，让函数在部署模型上变的更加自由，更加灵活。每个函数可以按传统的部署模型，部署到多个容器中，也可以按照自己的想法（流量情况）聚合到一起部署，随意的组合。



但是有些同学觉得这样的组合很不错，就会担心成本的问题，毕竟函数的部署模型调整的话，代码也会跟着调整，如果改动复杂的话，这个成本不会接受。

经过我们的设计，我们已经做到了只在 yml（配置）层面做调整，代码层面不需要改动，同时聚合的配置部分不影响原有的函数配置（随意增减），这就是我们 Flexibility 水平聚合能力，我们也叫它“高密度部署”。

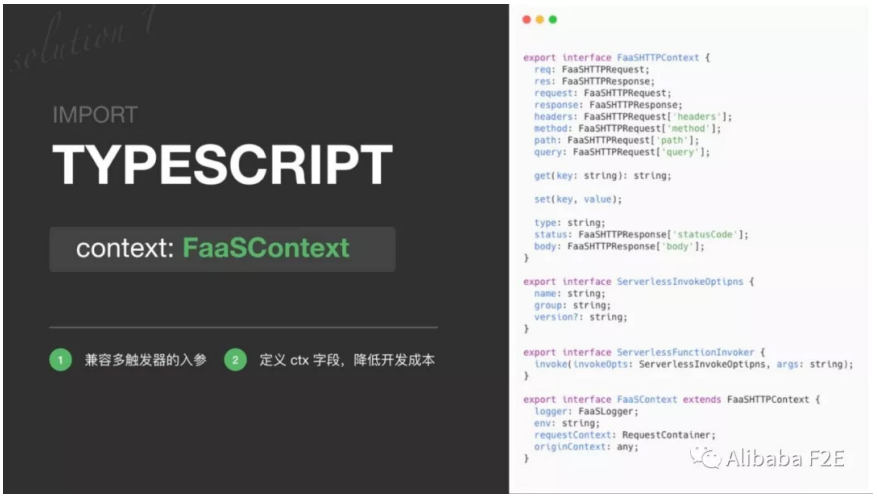


开发效率

第三点，我们考虑的是开发效率，在团队越来越大之后，开发的标注化，扩展性和可维护性一直是关注的重点，而随着 TypeScript 的推出，越来越多人将应用本身迁移到了这个体系中，我们也在不断思考，但是目前为止，没有在 FaaS 上看到这些方案。

前面提到过，我们将传统的 Event 和 Context 字段做了转换，同时，因为对代码做了变化，也需要给用户足够多的可用性提示，为此，我们定义一些 interface（比如 FaaSContext）辅助开发。

将 TypeScript 优秀的特性引入 FaaS 体系一直是我们的目标，从 17 年的 pandora 开始，包括去年我们开源的 midway，都全部使用 TypeScript 进行研发，内部的所有库已经全部迁移到了 TypeScript 体系上，让 FaaS 用上 TypeScript 也并不困难。



为此，我们将 midway 逐步改成了多场景方案，迁移出了名为 midway-faas 的函数框架，在标准的 Class 模型基础上，提供了基于 IoC，装饰器等新的特性，既和原来的体系一脉相承，能力共享，也在 FaaS 场景提供了更多的 Feature。

运行时扩展

在上面这些特性定义完之后，我们觉得函数的开发本身能力差不多了，但是还有些问题没有解决。

在使用一段时间之后，我们发现了一些用户诉求。

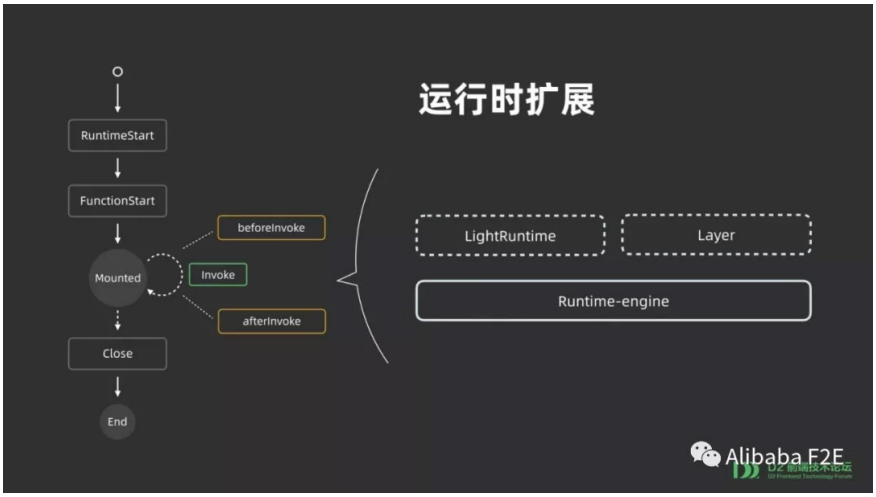
- 现有的函数，用户面对的直接是入口参数，我们如果要在跨函数前做一些参数校验，转换的的事情，目前需要做基类继承，或者直接写多次方法调用，这种类 middleware，或者 AOP 的做法，是否能够有方案支持。
- 内部运行时还好，运行时是由平台掌控，有些有初始化方法，有些没有，那么能不能在一定程度上抹平这些差异，让用户使用无感。
- 还有一些统一监控，埋点的需求，以及集团内部一直提的治理的需求

这些能力不一定需要在框架层本身实现，由于我们的构建特性以及隐藏了真实的入口，我们完全可以在整个调用之前加入这些能力。

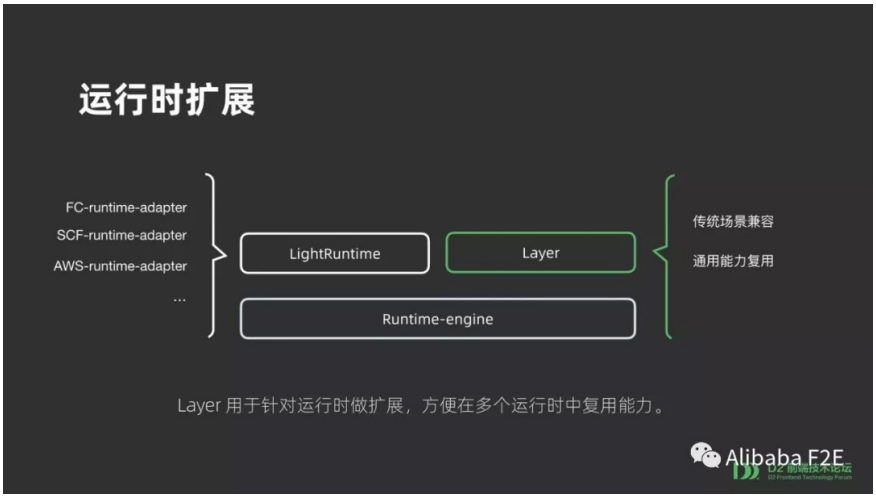
为此我们设计了一套针对运行时的生命周期，包括 RuntimeStart、Function-Start、Invoke、Close 四个阶段，以及他们的 before 和 after 的 hook 能力。同时将内部的运行时都基于这套编写，目前已经实现了多个平台的完整运行时。

不过这一套在社区平台稍有不一致，社区的平台基本无法自定义运行时，也就无法执行完整的生命周期，所以我们针对社区平台（例如阿里云 FC），就进行了简化处理。

整个运行时扩展简单来说分为几个部分，最底层的是 runtime-engine，用于整个生命周期的管理和执行。之上的是 LightRuntime，一个轻量的运行时，用于在不同的平台之中完成我们的运行时生命周期，同时适配各个平台的出入参。

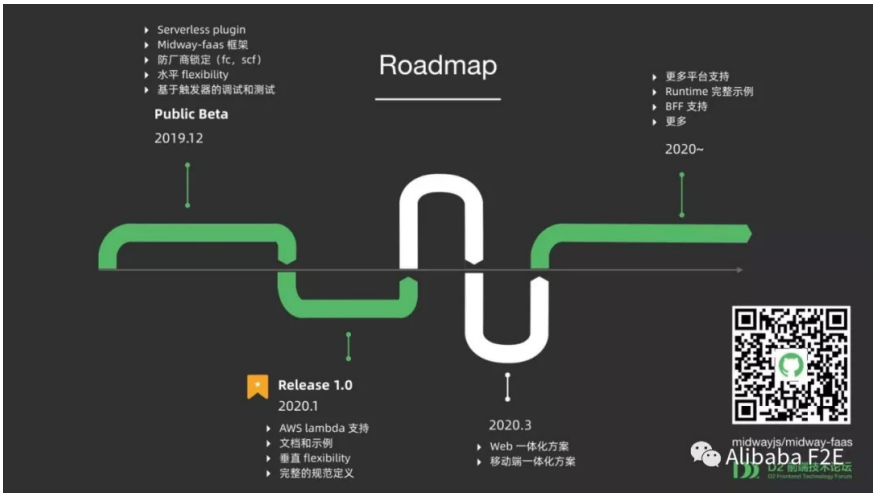


而另一块是每个运行时复用的能力，我们从 Lambda 中借鉴了概念，称之为 Layer。Layer 在设计上可以和任意一个运行时合并，赋予其额外的能力，我们一般用在多场景兼容（传统应用迁移），统一监控、metrics 等能力上。



经过快一年的迭代更新，我们将淘宝的导购业务迁移到了 Serverless 体系，同时集团其他 BU 也逐步的复制了这套方案，都顺利通过了双促的大考。

我们计划将这套 midway-faas 的能力开放给社区，如今，代码已经提交到了 github，还在飞速迭代中，算是 public beta 阶段，我们希望在明年的一月发布 v1.0，提供更多的 Feature 支持。



Serverless 的路依然还处在起步阶段，不断尝试，降低成本，为前端赋能是我们的目标，也欢迎更多的同学加入到我们的队伍中。

基于 FAAS 构建 NPM 同步 CDN

作者：张立理

本次演讲由百度资深前端工程师张立理为大家分享百度基于 FAAS 服务如何构建从 NPM 到 CDN 的全自动同步机制。本文主要介绍了云服务存在的问题及其解决方案，在包占用内存、时间较大时进行同步操作的细节问题及其处理方法。

演讲嘉宾简介：张立理，百度资深前端工程师。

本次分享主要围绕以下四个方面：

一、背景

二、目的——从 NPM 到 CDN 全自动同步

三、为何基于 FAAS?

四、云服务存在的问题

一、背景

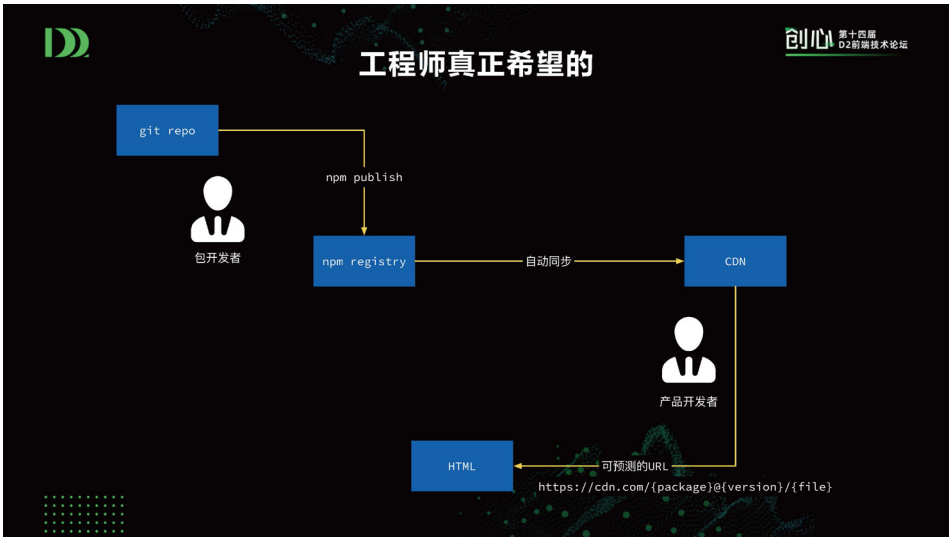
1. 为何需要 CDN？

百度有很多形态的产品和项目，其中一部分产品不使用 NPM，如一些简单产品、简单内部系统甚至没有 FE 的团队研发系统。但这些产品需要依赖一些通用包。还有一部分产品会使用 webpack 构建，但是如果将所有依赖都放入 webpack 构建，bundle 会比较大。如果能在 HTTP/2 的加持下，拆分大 bundle，分散引入多个体积合理的资源，可带来性能收益。同一个 URL 资源，共享使用者越多，缓存效应更强。

2. CDN 服务流程

基于以上原因，可以使用一个 CDN 来服务。但是当需要直接访问 NPM 上的某

个文件，需要先开通 CDN、下载 NPM 包、解压、上传至 CDN、获取 URL、最后修改 HTML。上述流程非常繁琐，对很多开发者而言非常麻烦，不如将代码直接放进代码库操作上线。而将代码直接放进代码库会带来几点问题。首先，代码存储中会有许多重复代码文件。其次，失去 CDN 的性能优势。第三，不同业务不能使用同一个 CDN 的 URL 资源，无法共享 CDN 缓存。而工程师的需求如下图所示，当任何一个开发者发送一个包到 NPM，该包最好能够直接在 CDN 上使用，即自动同步到 CDN。并且产品开发者使用时不需要主动到 CDN 上寻找，而是有可预测的、有规则的 URL 资源。



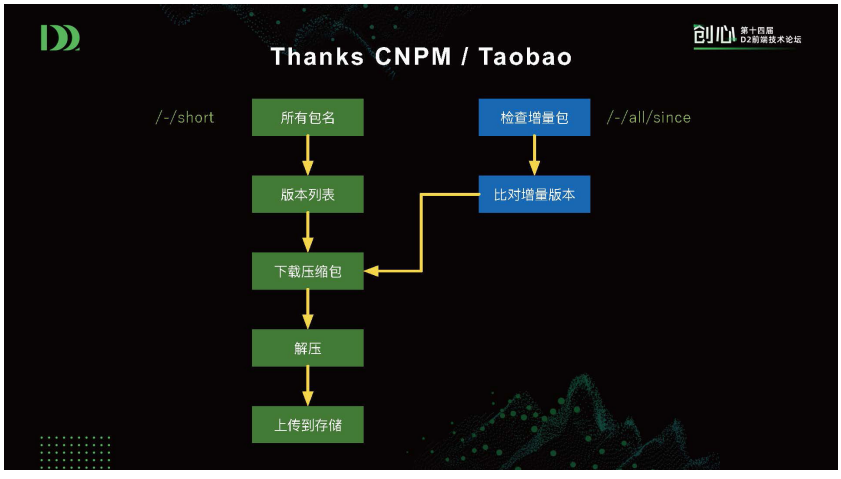
3. 社区现有方案

目前，最典型的产品是 UNPKG，其背后为 Cloudflare CDN，较为常见的有 jsDelivr，其背后为 Cloudflare、Fastly、NS1、StackPatch 等基础设施。以上方案被应用于国内产品时仍存在一些问题。首先，国外网络存在延迟、连通性问题。第二，这些产品按需同步，首次访问速度较慢，耗时可能达分钟级。另外，内部除了访问需求以外，还存在扩展需求。例如需要分析依赖的代码，对其进行自主存储，指纹库建设，漏洞分析等。

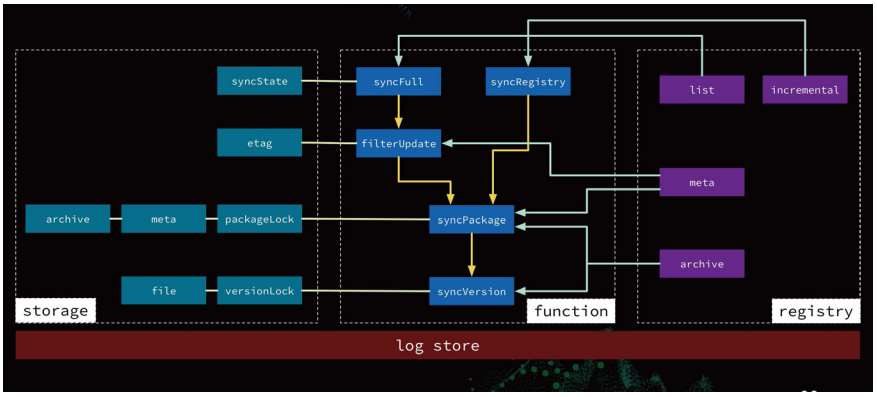
二、目的——从 NPM 到 CDN 全自动同步

1. 逻辑

在国内，CNPM 和 Taobao NPM 在官方 NPM 基础上提供了两个接口。一个接口可以拉到所有的包，另一个接口可以拉到一定时间内检查到的增量包。如果用户需要进行全量操作，需要拉到所有包名，请求每一个包的版本列表，下载每个版本的压缩包并进行解压后上传到存储。如果需要进行增量更新，首先记录上一次更新的时间，获取增量包，比对增量版本。然后同样下载每个版本的压缩包，进行解压后上传到存储。



如下图所示，上述逻辑实现有三大核心部分。中间部分为所有需要的函数，即逻辑。包括全量同步，增量同步，从版本列表中过滤出需要更新的东西，然后同步一个具体的包，再同步一个包下面的版本。相对应地，NPM 镜像提供全量列表、增量列表，获取一个包的元数据，获取一个版本的压缩包等。左侧是存储层面，需要已有同步的状态、每一个包的 etag，用于确认是否更新，存储压缩包，存储包的元数据，存储解压包后的每一个文件，可能还需要处理一些加锁的机制。



三、为何基于 FAAS？

1. Why FAAS？

NPM 同步 CDN 过程中为何要使用 Serverless 方案，而非直接使用一台服务器操作部署?

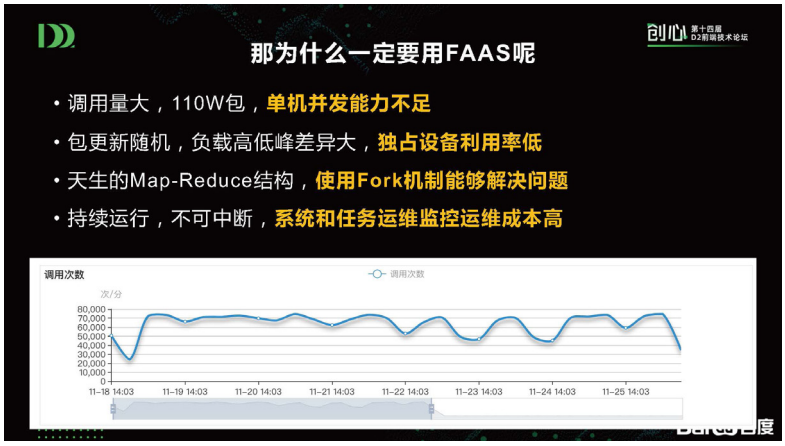
调用量大：NPM 现有约 110W 包，每一个包分不同版本，共有千万级别的版本需要同步。单机并发能力不足，需要 Serverless 的弹性能力。

包更新随机，负载高低峰差异大：全球开发者中欧美国家开发者较多，在欧美地区晚上八点钟左右包的更新量较多，其它时间更新量较少。如果开发者不按需使用函数，而始终使用一台服务器进行处理，独占设备利用率就会比较低。

包更新是天生 MapReduce 结构：每个包分为多个版本，版本下面分为多个文件。

使用 Fork 机制可以将一个任务变成几十、几百、几千个小任务，使用函数解决问题。

持续运行不可中断：在增量更新的情况下，如每五分钟做一次增量更新，可能只需要处理 20 个包，负载正常。如果更新同步挂掉了，而五天后才被发现，则此时需要重新同步五天的量，负载将会非常大，因此需要保证其持续运行不可中断。如果使用单机，需要做监控、重启等能力，系统和任务运维、监控运维成本高。下图曲线为典型函数的整体弹性情况，每分钟调用量约 5 万到 8 万次，与普通请求不同，每一个函数少则二三十秒，长则可能三五分钟。



2. 若只使用 Serverless

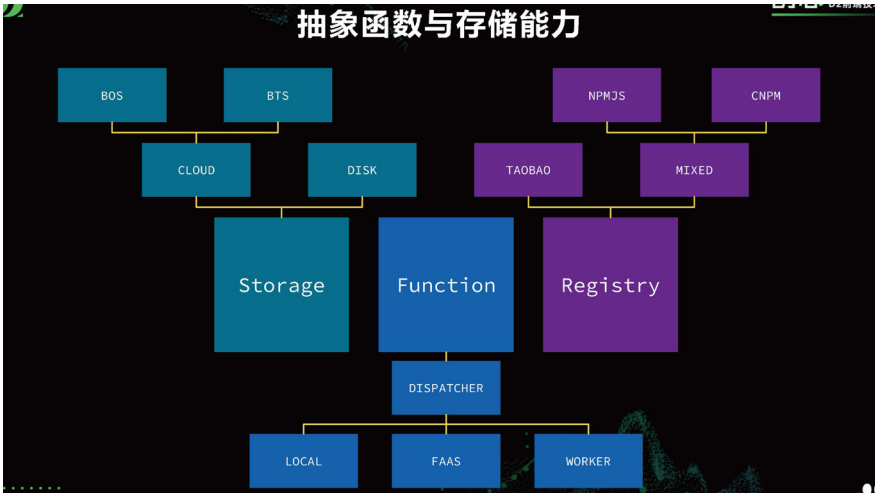
能否放弃其它方案，只基于 Serverless 进行操作？目前，只基于云服务构建有一系列优势。如完善的执行环境。Serverless 的任何厂商都会直接提供完整的 Node 环境。第二，可以自动化部署与更新。第三，无限存储与计算能力，因为云服务本身具备弹性。第四，优秀的云带宽和延迟。此外，只基于云服务进行构建还是存在一些问题，主要存在于开发环节。首先，测试需要本地构建函数发布到云上，再调用云的函数的测试，测试流程冗长，并且不便于本地调度。同时，开发的所有东西放在云上，测试也运行在云上，虽然前 100 万次调度不收费，但是已经在云上大量使用函数的情况下，显然测试的时间、内存、CPU 的消耗、调用次数等都会消耗云服务费用。

3. 抽象函数与存储能力

基于以上原因，开发者希望改良本地调试和测试的过程。

抽象函数能力：在函数基础上进行抽象。抽象函数能力核心就是存在一个函数，无论是在云上、本地还是其它地方都能够调用。抽象能力的基础是 DISPATCHER 调用器。DISPATCHER 能够防止开发者被厂商锁定，如调用阿里云、百度云、亚马逊云的接口是不同的，可以使用 DISPATCHER 进行归一化。另外也包括云、本地和不同本地实现之间的统一。作为调试使用的基本上有两种，一种是 LOCAL，即直接在本地调用函数，一种是 WORKER。使用 WORKER 调用函数的优势是如果卡住了，WORKER 可以杀掉，而如果 LOCAL 调用卡住，上下游可能都会卡住。

抽象存储能力：云上有完整的存储方案，如同步文件存储在对象存储中，其它元数据信息存储在表格存储中。将二者结合在一起就是一套基于云的存储接口。在本地测试时会基于本地硬盘做一套实现。可与云的存储接口统一为一套存储接口。同时还需要对远程 NPM 镜像进行处理。在国内可以使用淘宝镜像，但是存在 ip 被加入黑名单的风险。因为 Serverless 弹性较强，当一个高峰过去易被自动化屏蔽 ip。因此百度考虑分流压力，例如一部分可以到官方镜像，一部分可以到 CNPM，一部分可以到淘宝镜像。对于镜像做抽象，淘宝镜像和官方镜像返回的结构会有所不同，中间需要做屏蔽。

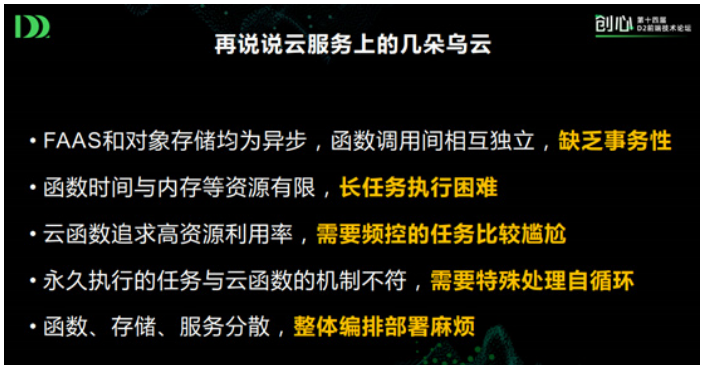


基于以上抽象能力，操作既可以运行在函数云上，也可以运行在本地或者运行在任何一台虚拟机上。对应配置如下图所示。在不同的函数中，用户可以使用环境变量说明所使用的实现，也可以用默认实现。只要在本地覆盖了环境变量，就可以有一套完整的、可直接本地调用的版本。若使用默认的环境变量，则可以完整使用一套 Serverless 函数的版本。

```
dispatcher: {
    mapping: {
        syncFull: env('DISPATCHER_TYPE_SYNC_FULL') || env('DISPATCHER_TYPE') || 'cfc',
        syncRegistry: env('DISPATCHER_TYPE_SYNC_REGISTRY') || env('DISPATCHER_TYPE') || 'cfc',
        syncPackage: env('DISPATCHER_TYPE_SYNC_PACKAGE') || env('DISPATCHER_TYPE') || 'cfc',
        syncVersion: env('DISPATCHER_TYPE_SYNC_VERSION') || env('DISPATCHER_TYPE') || 'cfc',
        filterUpdate: env('DISPATCHER_TYPE_FILTER_UPDATE') || env('DISPATCHER_TYPE') || 'cfc',
        default: env('DISPATCHER_TYPE', 'cfc'),
    },
    cfc: {
        region: env('BCE_REGION'),
        ak: env('BCE_AK'),
        sk: env('BCE_SK'),
        functionNames: {
            syncFull: env('CFC_FUNCTION_SYNC_FULL', 'npmSyncFull'),
            syncRegistry: env('CFC_FUNCTION_SYNC_REGISTRY', 'npmSyncRegistry'),
            syncPackage: env('CFC_FUNCTION_SYNC_PACKAGE', 'npmSyncPackage'),
            syncVersion: env('CFC_FUNCTION_SYNC_VERSION', 'npmSyncVersion'),
            filterUpdate: env('CFC_FUNCTION_FILTER_UPDATE', 'npmFilterUpdate'),
        },
        invocationType: env('CFC_INVOCATION_TYPE', 'Event'),
    },
    worker: {
        timeout: 1000 * 60 * 10,
    },
    local: {},
},
```

四、云服务存在的问题

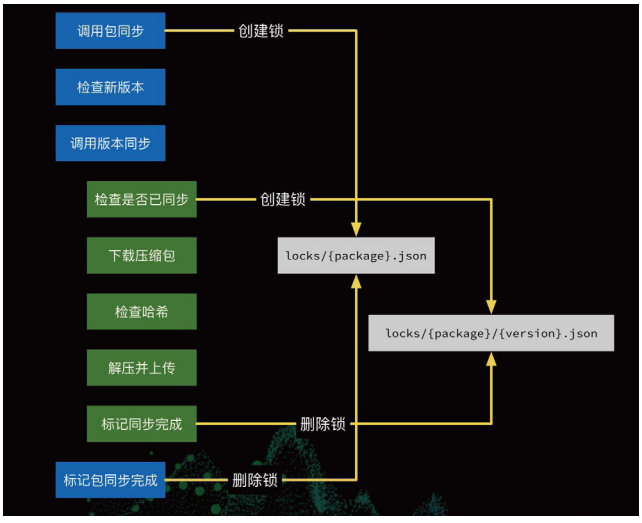
Serverless 服务整体非常完善，然而云服务上的用法并不经典，存在如下问题。



1. 事务

缺乏事务性：同步过程的上下游有串联关系。从包到版本，版本未结束同步，则包不能完成同步。从版本到文件，文件没有同步上传结束，则版本不能完成同步。但是文件、版本、包的同步等过程是在不同函数中进行。FAAS 和对象存储异步，函数调用相互独立，缺乏事务性。例如当版本同步失败时，并不能知道包同步是否成功，因为包同步操作不能等待版本同步完成再进行，只能将其发出去，假设会自行同步。

解决方案：异步环境下事务性问题不易解决，只能在一定程度上进行处理，如下图所示。使用单个文件作为锁，锁文件内写入启动时间、重试次数等同步相关信息。例如，要求当一个包的所有版本同步完成才能说明包同步完成。因此在开始同步包时，在每一个版本中加入一个 lock 文件作为锁，当一个版本同步完成后将删除 lock 文件。若 lock 文件最终未被删除，说明该文件所在版本未完成同步。并且由于难以判断该版本下已完成同步文件数，因此再次执行时需要重新同步该版本下所有文件。每一个锁有超时机制，设定时间为 10 分钟。若某版本启动同步 3 分钟时 lock 文件依然存在，可能是相关函数仍在执行。若启动同步 10 分钟后锁文件依然存在，则说明锁无效，重新触发。该方案仅支持简单不重复同步，不妨并发。要求同步过程幂等，少量重复执行并不会存在并发冲突的数据不一致性。若同步过程不是幂等的，需要更好的解决方案。



2. 长任务

长任务执行困难：与访问数据库、请求 web 接口等短时间操作不同，同步可能是长时间操作。例如所需同步的包很大、文件很多时，因此同步过程占用较多内存和时间。多数任务所需内存超过 1G 级别。而云函数时间与内存等资源有限，例如百度云最大占用内存为 512MB，最长执行时间为 5 分钟。因此长任务执行存在困难。

解决方案：首先尝试复用并发 Map 切割任务至多个函数。将大型任务切割为多个小任务送往下游函数。以同步包为例，需要获取包的版本列表。如果在一个函数中依次 for 循环处理所有版本，函数时间不足。因此可尝试将每一个版本作为一个任务送往下游函数，同时关注函数利用率。假设每一个函数时间为 5 分钟，而处理一个版本只需要 3 秒或者 10 秒，函数使用效率低且过多函数调用将导致收费高。因此采取 chunk 模式，根据实际运行时间等统计情况给下游每一个函数分发多个版本。例如一个函数 5 分钟可以处理 200 个版本，则以 200 个 版本为一个任务单位送往下游函数。

```
const pLimit = require('p-limit');
const {chunk} = require('lodash');

const main = async ({packageName}) => {
    const packages = registry.fetchVersionsOfPackage(packageName);

假设下游单个输入的运行时间为timeOfTask
一次函数的最长时间为timeTotal
预计多个输入间切换消耗为extra（比例）
则可以预测CHUNK_SIZE为
   Math.floor(timeTotal / (timeOfTask * (1 + extra)))

    const chunks = chunk(packages, process.env.CHUNK_SIZE);
    const limit = pLimit(process.env.CONCURRENCY);
    await Promise.all(chunks.map(c => limit(dispatchFilterUpdate, c)));
};
```

第二，同步过程涉及从镜像源下载压缩包、计算并校验 SHA 值、统计文件个数、解压文件、全部上传至 BOS（百度对象存储）等多个步骤。上述步骤如果依次进行，进行其中一步时其它资源均处于闲置状态，总耗时长。为解决该问题，不再基于任务步骤进行编程，而采取经典流式编程。即开始下载镜像源或者下载一部分后，开始同时也进行后续计算校验、统计、解压、上传步骤。流失编程可以最大程度上节省

时间与资源，提高效率，使更多、更大的压缩包能在指定时间内完成处理。但是流式编程较函数任务等待编程更为复杂，处理更加困难。好在由于同步任务是幂等过程，仍适用流式编程。对应代码如下图所示。获取 response 后将其写入文件系统。若压缩包解压后文件很多，一边解压一边上传，会因为上传时间较长导致网络流超时，以文件系统作为中转，一边写进文件系统，一边从另一端读取文件流，同时将其分出 3 个子流进行计算校验、统计、解压、上传工作，则可以有效避免网络流超时问题。

3. 频控

云函数追求高资源利用率：云函数追求高弹性、高资源利用率，可以瞬间达到非常高的执行 QPS。但是远端外部服务不能承受过高的执行频率，因此需要进行任务频控。例如要求 2W 个任务需要在 30 分钟内执行完成，而不能在 5 分钟内就执行完成。

解决方案：频控需要通过 sleep 函数等函数等待执行。而函数在云上运行时即使在等待时间也需要根据其配置的内存按秒收费，因此希望能在高弹性环境下进行频控的同时尽可能减少函数无意义等待时间，减少浪费，节省费用。下图所示为经典频控示例。处理某项任务时设置 timeout 时间，timeout 时间结束再处理下一个任务。该机制存在一些问题。第一，如果处理 10 个任务，前 9 项任务各等待 30 秒是正常的，

但是最后一项任务处理后无意义等待 30 秒是浪费的。第二，如果 5 分钟函数时间内只能处理 5 项任务，如果处理完第 5 项任务再等待 30 秒也是无意义的。因此需要优化频控方案。

如下图所示，在函数开始执行时记录时间，每一项任务完成后判断剩余时间是否足够进行等待。若剩余时间不够尽兴 sleep 等待，直接退出函数执行。另外，最后一项任务单独执行，不需要进行等待，处理完成即退出函数执行。

4. 自循环

永久执行任务需特殊处理自循环：同步函数过程需要不断检查是否有新增，是永

久执行的任务。而云函数机制是仅在需要时进行调用。因此需要特殊处理自循环机制，使同步操作持续不断地进行。

解决方案：第一，尝试让函数复活自己以保持永久持续执行。在一次完整的函数过程中，首先要预留足够时间供存储当前状态供下一次执行使用。例如需要处理 1W 项任务，在一个函数时间内处理 270 个后，要有预留时间记录并存储当前已完成任务状态，在下一次函数执行开始时直接从第 271 项任务开始处理。假设一个函数时间最长 5 分钟，预留 30 秒时间供存储状态，其余 270 秒可用于执行。每一项任务处理完成时计算剩余安全时间是否足够 30 秒，安全时间剩余 30 秒时停止任务执行，存储执行状态。然后函数递归调用自己，重新开始新的 5 分钟，取出上一次的任务状态继续执行。函数调用自己也可能因超时、内存不足、网络等原因调用失败。一旦函数调用失败同步任务就会停止，并且难以监控，因此风险较大。

```
const packages = await fetchRemainingPackages();
const start = Date.now();
假设一个函数最多5分钟，预留30秒供存储状态，则一共有270秒可用于执行
const FUNCTION_TIME_ALLOWED = FUNCTION_TIME_TOTAL - SAFE_EDGE;

while (packages.length) {
    const current = packages.shift();
    await Promise.all([dispatchNext(current), timeout(5 * 1000)]);
如果已经超时，则需要将剩余的内容持久化，供下一次执行取用，
数据量大的时候存储内容也比较耗时，要留出足够的安全时间
    if (Date.now() - start >= FUNCTION_TIME_ALLOWED) {
        await saveRestOfPackagesToState(packages);
        break;
    }
}
自己调用一下自己，重新开始一次5分钟的执行，俗称续命
dispatchSelf();
```

第二，使用定时触发器定时触发函数，假设函数时间为 5 分钟，则每隔 5 分钟触发一次函数执行。定时触发器由 Serverless 服务商进行保障，不会触发失败，保持同步过程稳定运行，比函数自调用更加安全可靠。

5. 编排

函数、存储、服务分散：同步服务涉及 Serverless 函数、表格存储、对象存储、缓存机制、CDN 服务、虚拟机等多种资源，以上资源分散并具有复杂的依赖关系，整体部署编排复杂。

解决方案：最初以手工方式处理。首先在云 console 界面手动申请一个对象存储空间、一个存放日志的存储空间、两个表格存储的表格、新建 5~6 个函数，本地打包函数、将函数上传到云上等多种操作。然后尝试调用，测试其能否跑起来，操作处理复杂麻烦。

后来尝试用 SDK 自动化，通过写脚本部署任务操作。如下图所示，部署函数、配置，然后分别调用云 SDK 进行创建或更新操作。然而使用 SDK 自动化仍然存在许多问题。存在先有存储才能有函数的依赖关系。函数版本和别名需要保持一致。环境变量不能写错。函数间有调用关系、调用方和被调用方发布有先后顺序。触发器、对象存储生命时长、配置复杂。不能每做一项工作每一次调整部署都写一堆脚本，每一次脚本测试也较为复杂。

```
const main = async () => {
    const deploys = {
        'filterUpdate': ['npmFilterUpdate', 'cfc.filterUpdate', env.filterUpdate, 'NPM同步包更新判断'],
        'syncPackage': ['npmSyncPackage', 'cfc.syncPackage', env.syncPackage, 'NPM同步包'],
        'syncVersion': ['npmSyncVersion', 'cfc.syncVersion', env.syncVersion, 'NPM同步版本'],
        'syncRegistry': ['npmSyncRegistry', 'cfc.syncRegistry', env.syncRegistry, 'NPM增量同步入口'],
        'syncFull': ['npmSyncFull', 'cfc.syncFull', env.syncFull, 'NPM全量同步入口'],
    };

    try {
        const functions = Object.values(deploys);
        for (const args of functions) {
            await deployFunction(...args);
        }
    }
    catch (ex) {
        console.error(ex); // eslint-disable-line no-console
    }
};
```

为解决以上问题，在社区中找到了 Terraform 自动编排方案。Terraform 是当前较为流行的希望统一云的资源管理的方案。Terraform 可以让用户声明各个资源，相互引用。比如某个函数要向对象存储写入内容，此时有环境变量统一管理调度，告知该函数应该向哪一空间进行写入。环境变量的值可以选择使用另外一个存储着资源的 name 字段。建立了依赖关系后 Terraform 将自动解析资源间的依赖关系，明确先后执行的顺序。同时 Terraform 可以基于配置文件制定计划，并且预测、可视化资源的变更影响，例如可能新建存储或更新函数等。当用户确认后再进行连接更新。Terraform 变量、环境可以使用专门文件进行统一管理，避免出现写入错误、和系统环境变量做耦合、被其它工具冲掉环境变量等问题。同时 Terraform 支持不同云服务商。Terraform 并不能解决调用一个函数时接口不同的问题，而能够解决将函数更新到云服务时不同云服务的接口不同的问题。Terraform 可以做到优秀的可视化部署。

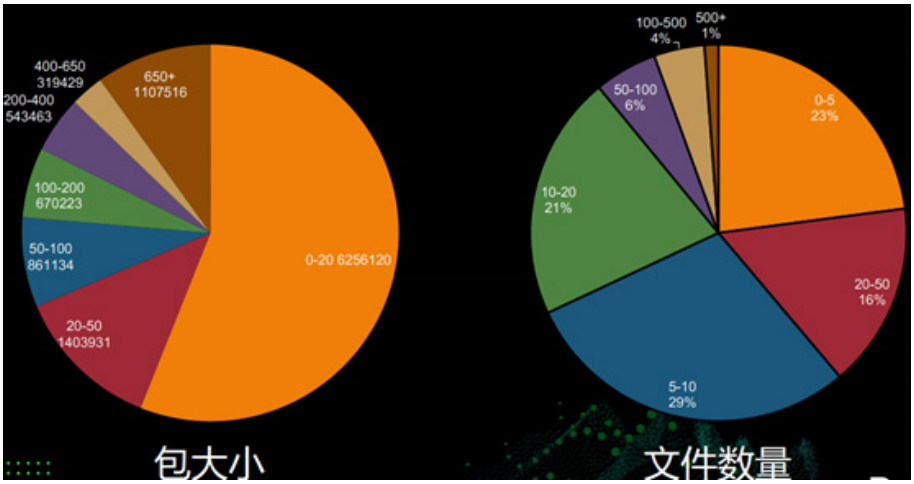
6. 完善服务

以上问题解决后，尝试在云上运行测试，仍然发现许多错误，以及许多方面与预期不符。完善服务有复杂之处。

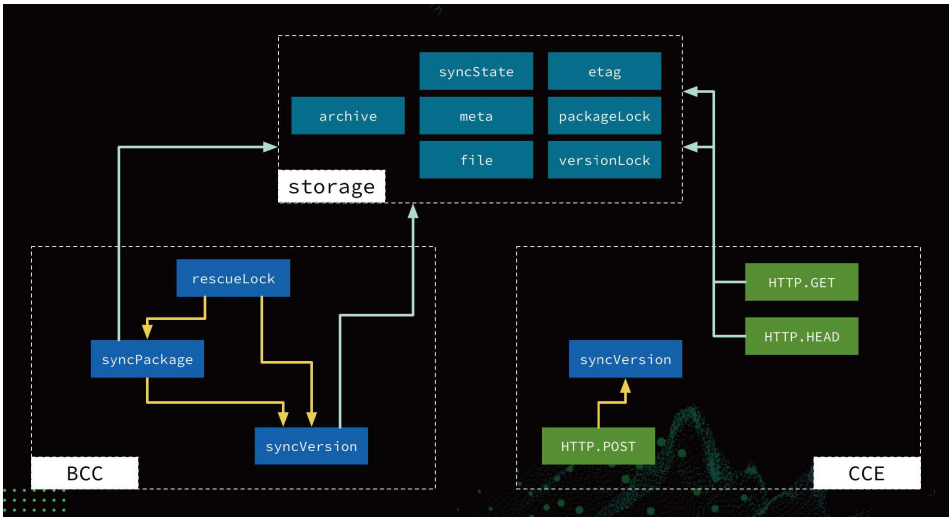
NPM 的包形形色色：下图分析了 NPM 上所有包的大小和文件数量。大部分的包大小在 20KB 以下，同时存在一些比较大的包，比如存在一部分大于 650KB 的包。大的包会影响执行时间与效率。从文件数量方面而言，大部分包的文件数量在 20 以下，但是存在一些文件数量非常大的包，1% 的包文件数量大于 500。统计数据如下：

- 平均尺寸 635.4KB，最小 10KB，最大 267MB；
- 平均文件数 45.2，最少 0 字节，最多 140442 个字节；
- 大体积、多文件的包，往往版本多，大部分采用定时发布策略。

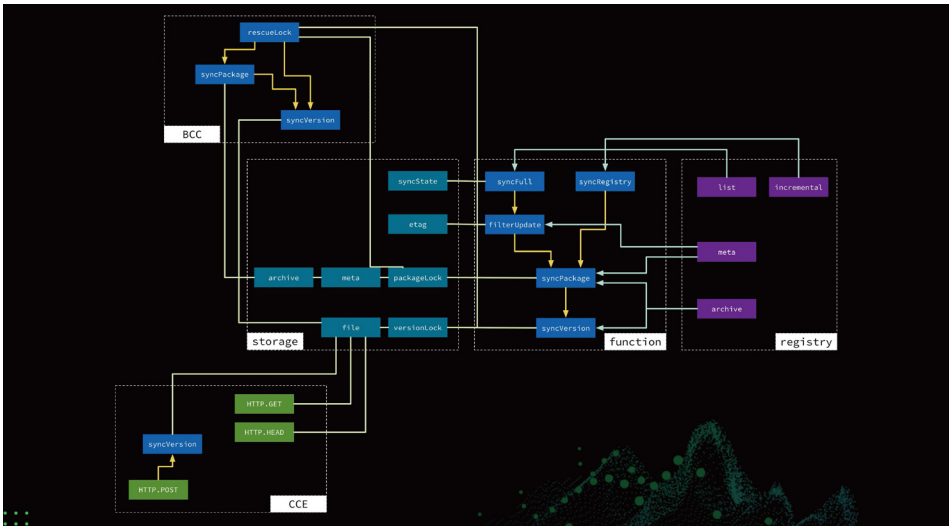
由于内存、时间有限，NPM 上部分包不适合使用 FAAS 进行同步。少部分大尺寸、多文件的包拉高了平均水平。数量 500+ 文件、体积 600KB+ 的包几乎无法在有限时间、内存下完成同步。



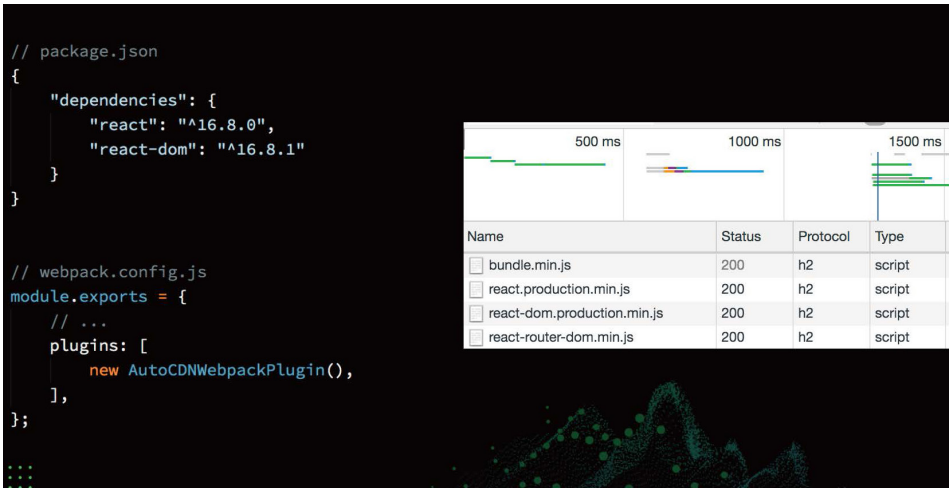
解决方案：在 Serverless 基础上备用本地同步环境 +CDN 服务。上文提到为了能够在本地进行测试，对镜像、存储、函数调用等进行了抽象。再将本地测试的这一套东西做成 Docker 镜像，放入 Docker K8S 容器中，就可以运行。如果不做成镜像，直接将源码放入虚拟机，也可以运行。本地同步本身就可以运行在多种不同运行环境中。



将本地同步与 CDN 服务结合在一起，如下图所示。除了中间云函数部分，将通过 K8S 集群做一套备份，再通过虚拟机集群做一套备份。备份专门用于检查长时间同步失败的包，并通过本地无限时长、大容量的环境进行恢复。在上述完整环境下，基本所有包都可以同步成功。



辅以开发插件自动使用 CDN：上述服务完善基础上，同步服务可以较好地运行，接下来考虑如何令开发者更好地使用 CDN，对研发环境与设施进行了处理。如下图所示，百度制作了一套 Webpack 插件。当开发者在前端项目中，可以随意编写 dependencies，例如依赖”react”以及”react-don”。编写完成后不需要开发者自己进行繁杂的配置工作，只需要引入 plugins。plugins 可以自动识别出其中 CDN 上有的常用依赖，将其提取出来从开发者的 module 中删除，并将其指向 CDN 文件。相对应地，开发者的系统构建出来就会从 CDN 上加载”react”以及”react-don”，而开发者的代码中不需要感知这些操作。百度已经使用该 Webpack 插件支撑了内部五、六个产品。有效见证了该插件的启用缩减了百度一部分内部产品的启动响应时间约 500~600 毫秒。



工程化

前端工程化下一站：IDE

作者：上坡、坑头

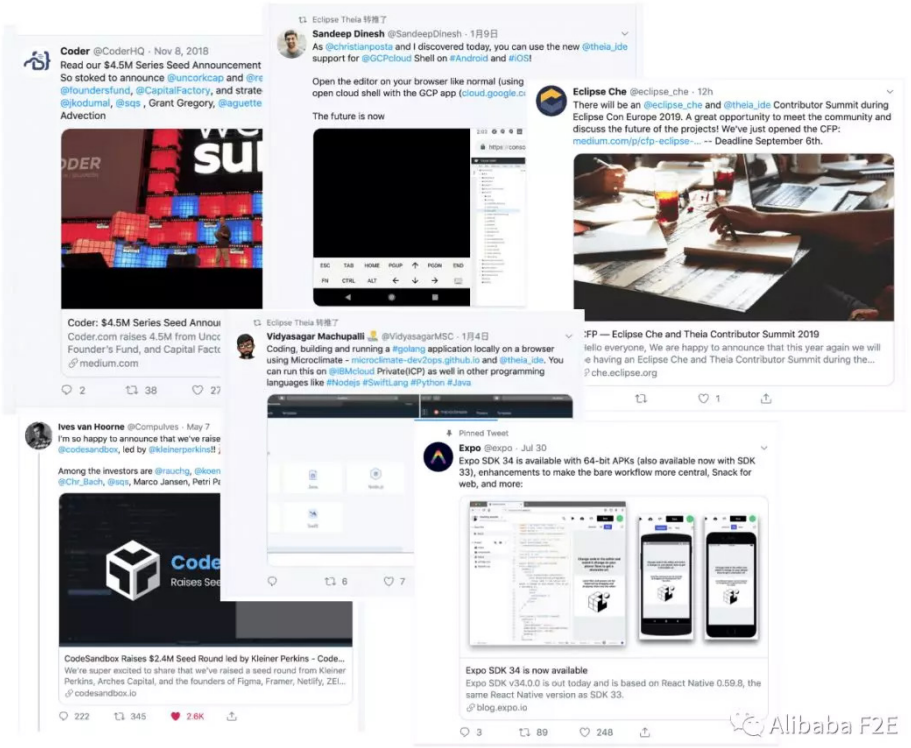
随着前端开发的发展更迭，前端日常开发工作变得愈发复杂愈发深入，同时前端工程中从项目初始化、编译、构建到发布、运维也变得细化而成熟。日常前端工作的每个环节都涌现出丰富的工具、服务和解决方案来解决工程效率的问题。那么，前端工程化的下一阶段的突破口是什么，我们期望通过 IDE 的方式和视角来找寻答案。

行业分析

行业趋势

其实 IDE 本身不是一个新兴的概念，在维基百科上能找到第一款的 IDE 是来自 55 年前为 BASIC 语言提供研发能力的一款软件工具，而最近的一到两年时间内，IDE 的业内动态突然呈上涨趋势。

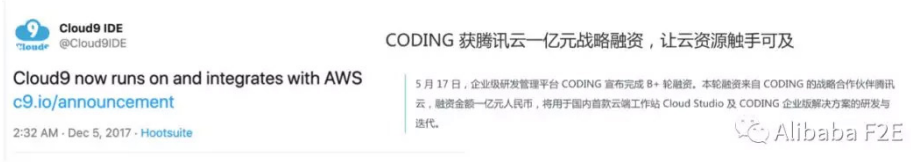
从外部视角来看，可以看到有两个趋势浮出水面，一个是 IDE 领域相关的创业公司逐渐浮现出来，出现了许多相关的创业明星公司：



1. 起初由前端脚本编译项目到现阶段容器化能力支撑编译服务能力、不断突破倍受好评的 codesandbox 平台；
2. 出自 Eclipse 研发老牌团队研发，目前被各大公司、厂商，包括谷歌计算服务 (GCP) 采用 theia-ide 解决方案；
3. 以及号称兼容 VSCode 程度最高的 coder 项目；

除此之外还有很多小而美的 IDE 落地场景，如 RN 研发明星产品 expo 的配套云端工具，发展成 npm 官方配套的 runkit 平台。

另一个趋势是云厂商市场上巨头的投入也开始增加：



随着 AWS 完成对老牌云端 IDE 工具 Cloud9 的收购之后，国内腾讯云也完成对 Coding 的战略投资，而近期也完成对 Coding 公司收购。而近期微软也拿出了自己的解决方案，推出了云端版本的 VSCode。

从阿里体系视角来看，与集成研发环境相关的工具平台也在这段时间周期内如雨后春笋涌现出来：

随着近年不断发展的**支付宝小程序、函数服务、中后台、智能化**等前端技术的发展，在不同前端技术方向，都涌现出了通过集成研发环境的方式来整体研发过程中的工程工具服务，来提升整体研发的效率和体验。

起因分析

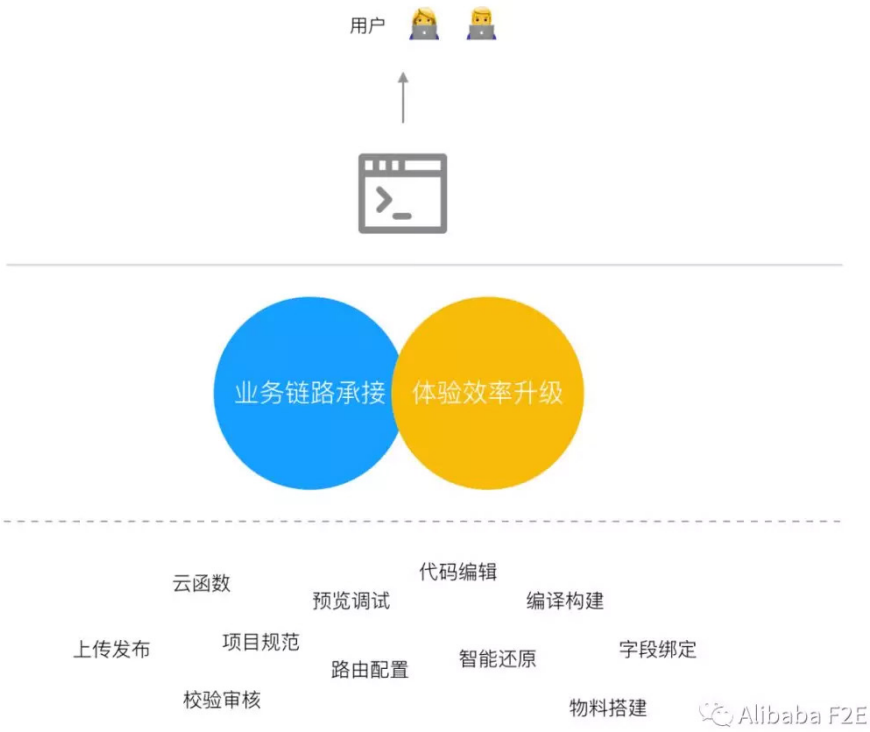
面对当前的趋势，为了更好地到 IDE 共建的研发落地思路，我们从**业务侧、技术侧**两个角度来分析内在的起因。

业务侧

业务链路承接：针对许多与研发强相关相关的如小程序、函数计算等场景，技术产品链路背后都有一个完整技术、产品体系需要一个承接主体，来完成对产品功能、信息、能力的传达。通过日常研发人员熟悉的 IDE 形式，将各个研发体系中的关联功能进行集成，向用户透出。通过集成方案降低开始成本与门槛，提高品牌、产品的

触达效率。

体验效率升级：随着工程化的发展，代码研发模式也从较单一的编辑工具编辑延展到 终端命令、本地工具、线上研发平台 相互穿插交织的方式。而 IDE 刚好能作为研发编辑工具与研发工具服务的结合平台，提高工具、服务之间的串联效率，提升链路的完成性和交互体验，最终完成研发链路的效率升级。

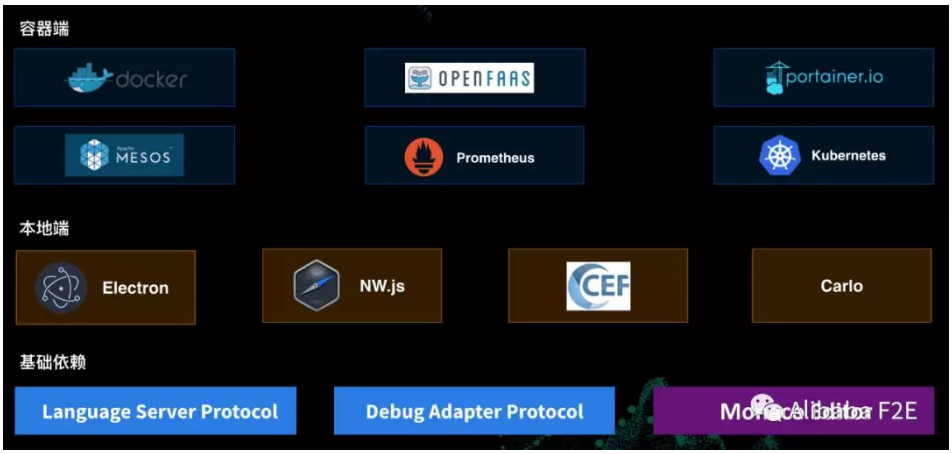


业务侧来看，可以看出两个观点其实相辅相成，正式由于通过 IDE 这种形式，能更好地完成业务链路和产品细节的整合，将复杂度和衔接成本降低，同时能提供良好的流程体验，更有利于业务场景的落地和推广。

技术侧

一项技术产品站上舞台往往是技术的底层设施的完善程度有一个里程碑式的提

升。随着当前线上线下领域中底层技术的成熟与开放，为搭建 IDE 相关的技术工具、平台降低了门槛，这里举两个代表性的例子：



随着在容器、本地、基础依赖技术体系的不断成熟和得到开发者的认可，搭建 IDE 集成环境的效率和门槛进一步提升和降低，促进 IDE 在不同研发场景中的落地和发展。

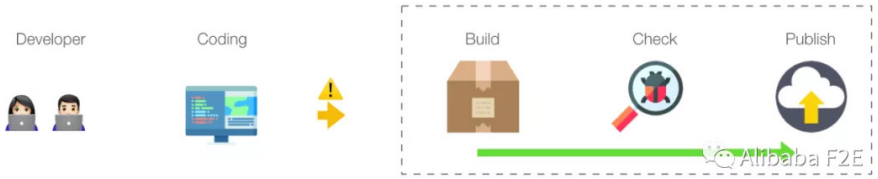
分析小结

由整体行业的发展我们可以窥见，当下一体化的研发模式会逐渐随着**业务上的诉求**以及**技术上的成熟**的趋势 ，通过 IDE 集成研发环境整合的方式在各个业务场景上铺开。

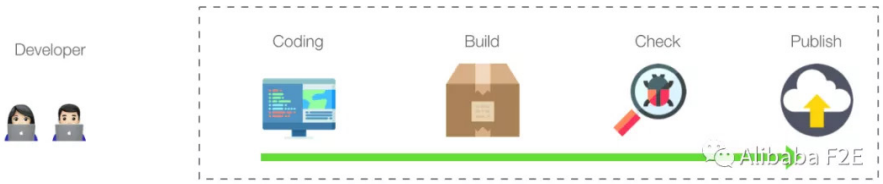
而回过头来，当下研发模式面临的问题和痛点是什么，是我们 IDE 共建项目的目标和出发点。

起因与目标

起因



一方面当下的工程体系、平台很多更关注于线上资源的**编译构建**、**审核检查**以及**上线发布**流程，而对于研发态的干预和定制能力有限。



代码研发过程往往与上线过程存在一定的隔阂。面向未来的理想全链路工程体系，可以将研发态的能力与服务也进行链路整合，完成从仓库创建到仓库资源发布的整合，最大化的集成研发过程中的能力与服务，提升研发效率。

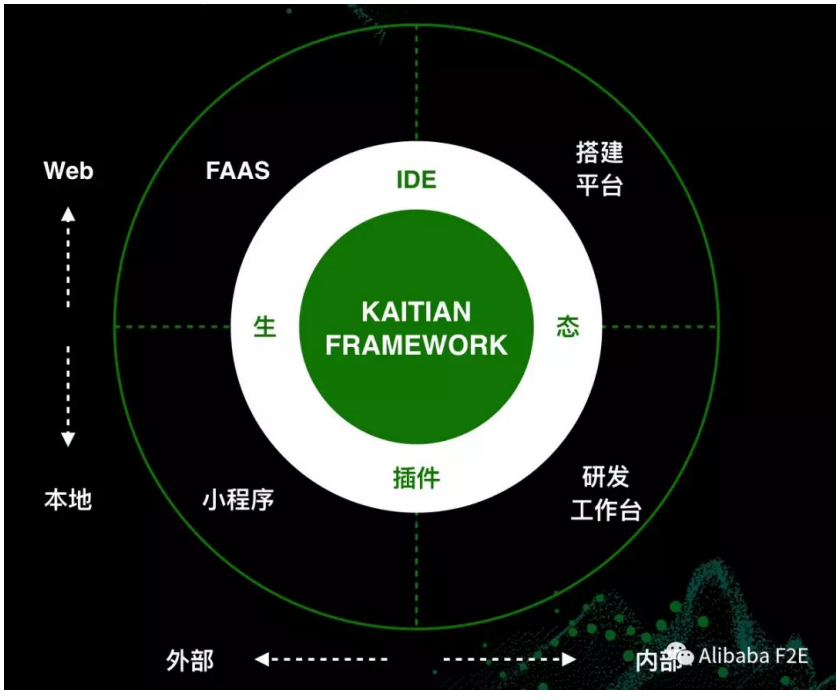
另一方面，由前文提到，当下其实在公司内外部的很多产品中，都有各自自研或者基于开源项目进行 IDE 能力研发与定制，同一个功能点，例如基于 monaco-editor 代码语言提示等功能在现有的产品中的实现都大同小异，而这块相同的功能的投入每个平台都需要投入一定的精力，存在 重复建设的问题，而向更深层的探索理解和研发投入也受到限制。

目标

从当前的现状出发，引申出共建 IDE 项目的目标：

> 面向经济体业务研发场景，打造云端与本地端一致，内外统一的 IDE 通用底层能力，支撑研发场景的 IDE 能力的快速搭建与集成

基于 IDE 底层能力，业务场景研发平台能快速地完成 IDE 能力的搭建或集成，同时通过底层的**插件体系能力**完成业务研发链路的能力的定制集成。



通过通用统一体系的建设，将重复投入的精力进行收敛，让更多的精力基于统一的研发底层投入到更深层次能力的建设和创新中去，形成技术生态的良性循环，在内外部各个系统之间形成扭转，让技术产出发挥最大化的价值。

方案与策略

项目定位

IDE 的本意是**集成开发环境**。从狭义上理解就是我们日常研发的代码开发工具，例如以 VSCode 举例，集成研发环境中包含的代码编辑器 Editor 、资源查看 Explorer、终端 Terminal` 等相关的功能模块。

当前实际业务场景的诉求上所期望的是作为**集成开发环境**角色，研发平台需要支撑的除基本编辑研发能力相关之外，同样需要将用户在研发流程中所用到工具、服务都进行串联和集成，才能让研发流程在一个环境或者体系中运转，发挥工具、服务的最大效应和价值。

所以 IDE 共建底层项目需要做的，一方面往技术底层走，需要在常规的 IDE 基础模块能力方面提供完整的通用体系能力。另一方面往上层走，面向业务场景提供 IDE 服务集成解决方案，方便快速完成业务场景中研发工具、服务的整合与落地。

基础能力

在开展上层建筑业务价值之前，需要将最底层基础的基本能力进行夯实，我们将现有传统 IDE 领域中所关联的一些模块能力进行分成划分：

目前大致化为三个部分。

基础能力层：有构建 IDE 功能的底层依赖模块组成

1. 例如基于布局能力来完成整个 IDE 应用界面的搭建；
2. 基于最底层的文件服务模块完成目录浏览的数据读取支持和进行目录和文件的浏览查看，同时也向文档 Model 文档定制模块提供相应的文件监听能力，用于前台编辑文本的状态同步；

封装能力层：这一层与基础能力层一样，也是属于共建底层能力的一部分。而这层则是着重通过基础能力层提供的能力，来实现在 IDE 领域相关的一些核心模块。

1. 例如核心编辑器模块的运行需要依赖底层的文件服务、通信能力、命令机制、快捷键绑定等一系列的模块；
2. 而在终端模块的协议适配过程中则需要通信能力提供底层的协议适配能力；

支撑服务层：除了 IDE 集成需要的底层模块能力，这层主要是针对 IDE 的运行态所需要具备的一些线上服务。

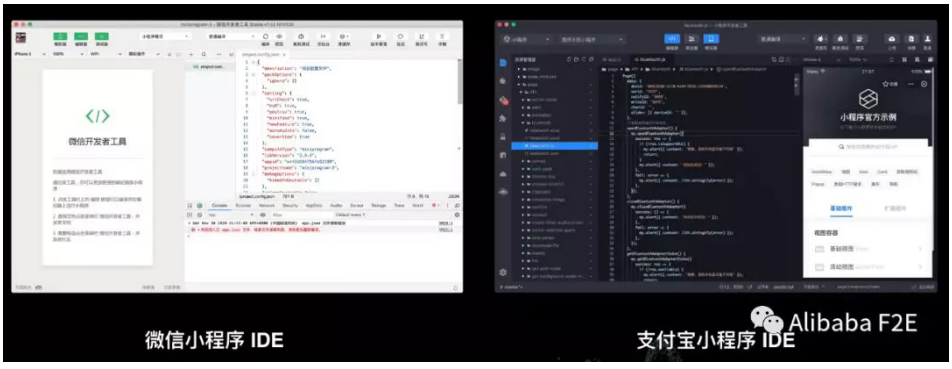
1. 例如通过插件市场进行插件的下载安装；
2. 通过容器服务来提供云端版本 IDE 的运行时环境等；

目前共建项目现阶段主要的精力就是投入到基础 IDE 底层模块及支撑服务的研发中，保证功能在符合现有落地的场景下，达到现有 IDE 体系一线水平能力。

业务能力

在业务能力话题中，前文提到 IDE 的核心价值其实对业务场景研发链路的整合，完成研发效率的提升，在现有的环境中确实有非常多这样的场景：

在业务集成能力的采取的策略方案上，因为我们期望想实现内部与外部、本地端与云端的同步一致，我们期望通过插件的方式来解耦打包集成能力，形成技术生态服务业务。

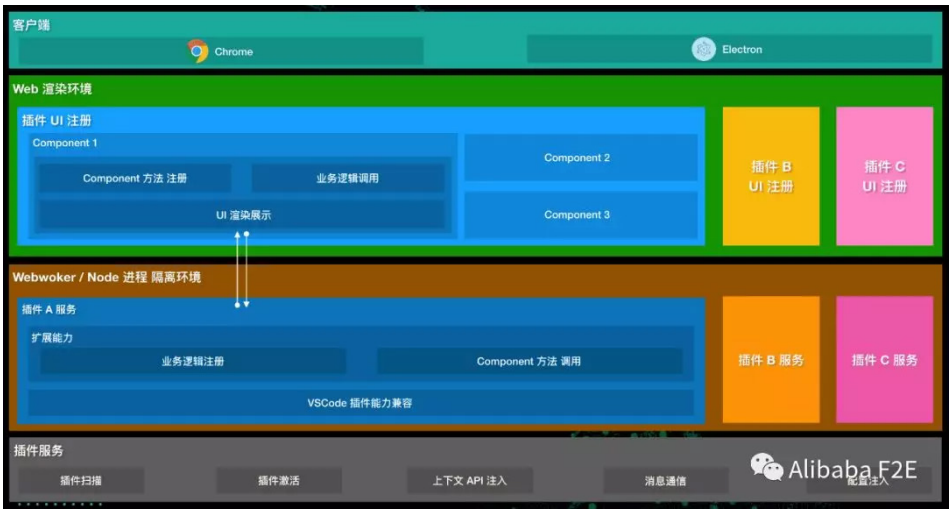


以**微信小程序**、**支付宝小程序**举例，相比现有 VSCode 体系中的插件主要是针

对研发流程中的**编辑体验增强**，业务上的场景往往是**研发链路界面**、**入口的集成**来实现链路的串联。

所以在目前设计的插件体系中，在兼容 VSCode 插件生态 的基础上，扩展出插件对 UI 组件定制的逻辑，针对插件中的业务逻辑，底层提供提供可选的浏览器 WebWorker 环境与 Node 环境 运行时环境，并在运行环境时中注入底层能力中一些响应 API ，提供业务逻辑运行时与 UI 组件的双向调用链路。完成 UI 组件定制与业务逻辑的响应。

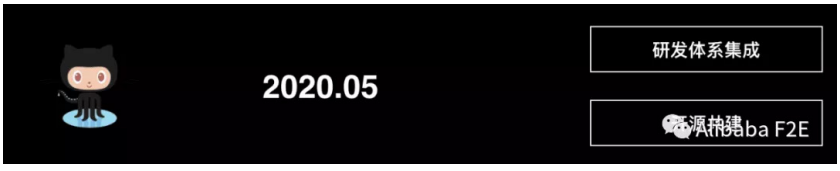
在插件中可以在逻辑服务侧进行对界面中注册的通过不同组件注册进行 API 调用，反馈到组件的展示上，同时在浏览器中不同的 UI 组件也能对插件服务逻辑提供出来的服务接口逻辑进行调用，分离逻辑执行，提高整体的业务逻辑执行性能和稳定性。



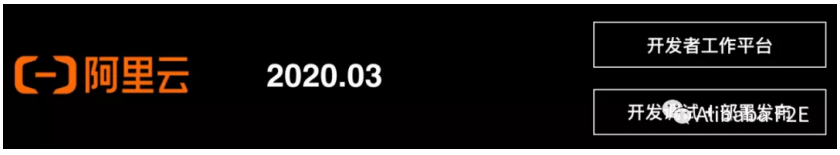
在从业务场景中去碰撞汲取经验、沉淀技术的同时，我们也对接下来的发展有既定的计划。

下一步

我们期望在随着底层的成熟和稳定，在明年的时间点能在下一步进行更多场景的尝试。



一方面进行外部开源开放，用户也可以借助底层能力来做符合自身日常工作的集成研发解决方案，并也亲身参与到底层能力的改造优化中。



另一方面在阿里云上基于 KAITIAN-FRAMEWORK 云上也会提供开发者工作台串联云上研发流程的产品，来集成掉开发调试与部署发布全链路流程，提升整体云上服务的研发体验，提供未来研发模式的探索体验。

我们期望能通过 IDE 项目，凝聚合力，形成生态环境，借助集体的力量，一步一个脚印迭代完成从轻量级研发到主要研发模式的逐步替换，形成未来日常工作的基础设施。

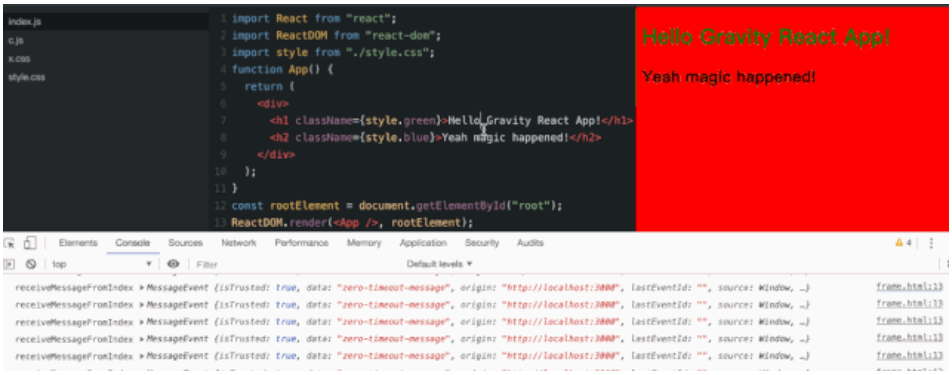
基于浏览器的实时构建探索之路

作者：玄寂

自我介绍

我是来自 RichLab 花呗借呗前端团队的同学。在公司大家喊我玄寂，生活中大家称呼我 pigcan 或者猪罐头。除了是一个程序员，我现在也在尝试做一名 YouTuber 和 up 主，也在微信公众号中分享我的生活，我自己的方式践行快乐工作，认真生活。

体感案例



首先为了让大家有更好的体感，我们先来看一个案例。这个案例是使用 code mirror 加 Antd tab 组件加 Gravity 做的一个实时预览。大家可以通过这个 gif 能看到，我变更 js 文件或者样式文件的时候，在右侧这个预览区域可以进行实时的更新，这部分的能力是完全由浏览器作为支撑提供出来的，并不依赖任何本地 server 或者远程 server 的能力。

在有了这个体感之后，大家可能会更容易理解我之后讲的内容。

文章提纲

接下来我会从 5 个方面切入，来谈一谈基于浏览器的实时构建探索之路。

- 首先是背景，从历史来看构建工具每次发生大变更时，都和前端的技术风潮息息相关。而 2019 年前端界发生的变化，也可以说是促使我做这个技术探索的原因。
- 有了这些变化，通常情况下现有的技术架构就可能会出现不满足现状的情形，这就是机遇了，这也就是我想要说的第二部分，这些变化会给我们的构建带来哪些机遇，而面对这些机遇，我们在技术上又会有哪些挑战。
- 第三点我会来谈一下在面对这些机遇和挑战时，我们在技术上所做的选择，也就是我们如何来架构整个技术方案。
- 第四点我会基于前面所说的技术架构，谈一谈需要克服的技术难点，主要是要抛一些我的解决思路。
- 第五点也是最后一点，我会畅想一下这个技术方案可能的未来，其实更多的是我对它的期待。

背景

时间回到 2011 年，那会儿我们前端一直在强调复用性，基于复用性的考虑，我们会把所有的文件尽可能的按照功能维度进行拆分，拆的越小越好，这种追求我称它为粒子化。粒子化的结果是工程的文件会非常非常碎，所以那个时候的构建工具，更多的思路是化零为整，典型工具有 Grunt 和 Gulp。

随着粒子化时代的到来，到 13 年左右很快新的问题出现了，这时的问题在我看来主要集中在了两个部分：第一个是，传统的拼接脚本的方式开始不能满足模块化的需求，因为模块之间存在依赖关系，再者还有动态化载入的需求；第二个是那么多功能模块被划分出来了，把划分后的模块放哪里是一个问题，最初 NPM 是并没有向前

端模块开放的。所以接下来便出现了模块加载器，和包管理之战。这场战役让我们的前端模块规范变得五花八门，最后好在所有的包落在了 NPM 了。所以这个时候的构建工具更多的是抹平模块规范，典型工具 Webpack 的出现意义很大一部分就在于此（当然在这个过程中，其实还出现了各种基于加载器的定向构建工具和包管理，这里就先不谈了）。

那时间再次回到 2019 年，我们听到了不一样的声音，这些声音都在对抗 bundler 的理念。

比较典型的有两篇文章：

- Luke Jackson – Don't Build That App!
- Fred K. Schott – A Future Without Webpack

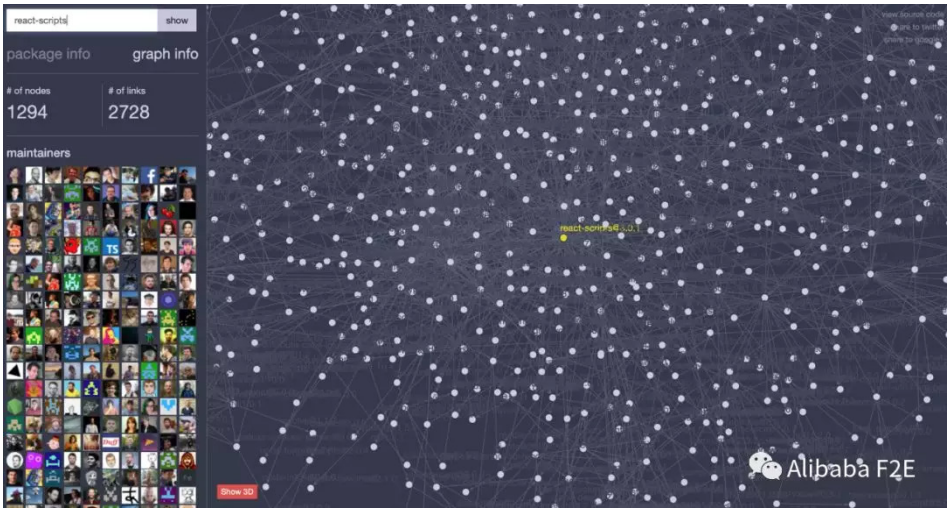
为什么会有这些声音，这些声音背后的原因是什么？一方面是因为新的技术标准的出现，另外一方面也来源于日益陡峭的学习曲线。

现在，要运行一个前端项目，我们通常需要知道：

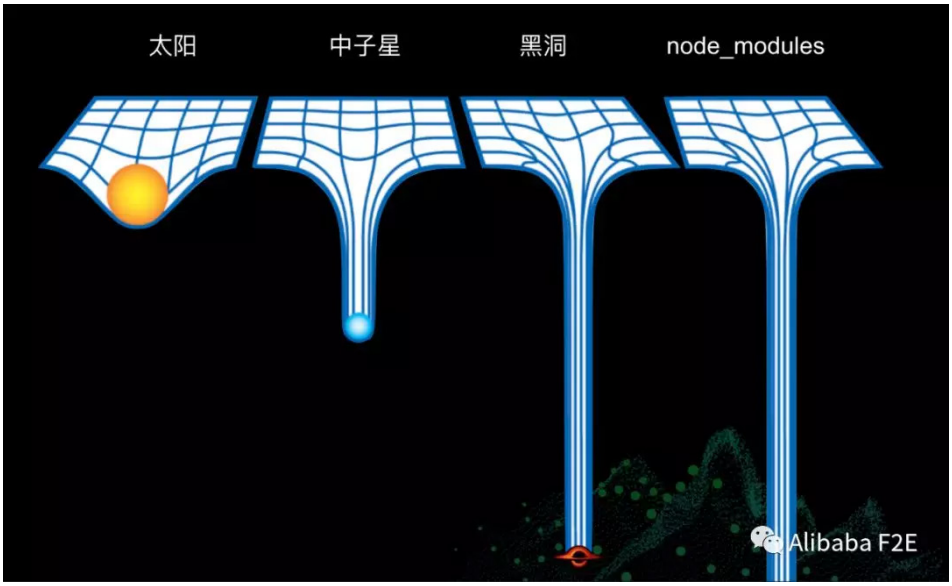
- 前端构建的概念
- 要知道在琳琅满目的打包工具中做合理选择
- 要知道如何安装开发环境，如何执行构建，如何执行调试
- 要知道如何配置 – Webpack、Webpack Loaders and plugins etc.
- 要知道如何写插件 – Babel APIs、Webpack APIs etc.
- 如何调试插件
- 如何解决依赖升级 – Babel 5 –> 6 –> 7, Webpack 1 –> 2 –> 3 –> 4 –> 5

反正就是一个字——“南”！

再来看看我们的包管理。以 CRA 为例，只是为了运行一个 React 应用，我们居然还需要附加如此复杂的依赖。



在网上也有一些调侃，前端的依赖比黑洞造成的时间扭曲还要大。



回过头再来看，2019 的趋势是什么，相信大家都感觉到了「云」这个词，我们很多的流程都在上云。

那面向上云的这种场景，我们如此复杂的 bundler 和包管理是否符合这种趋势呢?

归根结底，其实是要探讨一个问题：

前端资源的加载和分发是不是还会有更好的形式?

而对这个问题的回答，我觉得是有空间的——正是这种笃定，才有了接下来的内容。

机遇和挑战

现状

在上一小节中我们已经谈到了 2019 年不管是 pro / low code 都在朝着上云的趋势在变化，那应对这些变化，我们先来看看现有的一些平台，他们对于构建的态度是什么。

类型	代表
专业	Codesandbox、Stackbiltz、Gitlab Web IDE、Ali Cloud IDE
辅助	Outsystems、Mendix、云凤蝶

从这些平台中我们可以总结出三种态度：

- 只做编辑器或者画板
- 做编辑器或者画板并且提供了一个限制性的研发环境
- 做编辑器或者画板并且提供了一个完全开放的研发环境

总结下这三种态度，本质上是使用了两种技术方案：

- 容器技术

- 基于浏览器的加载策略

最终其实可以总结为：

- 把服务端的能力进行输出。这种方案的优势是服务端拥有和本地研发环境一致化的环境；缺点是即时性较差、效率较差、无法离线、成本高昂。
- 把客户端的能力释放出来。这种方案的优势是无服务端依赖、即时性、高效率、可离线运行；但缺点也比较明显，所有能力建设都必须围绕着浏览器技术

云时代的来临，我认为配套的构建也来到了十字路口，到底是继续维持现有的技术架构走下去，还是说另辟蹊径，寻找一条更加轻薄的方式来配合上云。

Bundless

我们再回过来看看，2019 年为什么在社区能释放出这些声音来（Luke Jackson - Don't Build That App!、Fred K. Schott - A Future Without Webpack），为什么会有人敢说，我们可以有一个没有 webpack 的未来，为什么 Bundless 的想法能够成立，支撑他们这些说法的技术依据到底是什么。

归纳总结下：

- 使用模块加载器，在运行时进行文件分析，从而获取依赖，完成树结构的梳理，然后对树结构开始编译

比较典型的产品有：systemjs 0.21.x & JSPM 1.x 、stackblitz 、codesandbox

- 使用 Native-Module，即在浏览器中直接加载 ES-Module 的代码

比较典型的产品有：systemjs >= 3.x & JSPM 2.x 、@pika/web

再看了这些产品和技术实现后，我内心其实非常笃定，我觉得机会来了，未来肯定会是轻薄的方式来配合上云，只是这一块目前还没有人来专心突破这些点。

所以我觉得未来肯定是 云 + Browser Based Bundless + Web NPM，这就是

Gravity 这套技术方案出现的背景了。

Gravity 的挑战

所有的挑战其实来源于我们从 nodejs 抽出来之后，在浏览器内的适配问题。

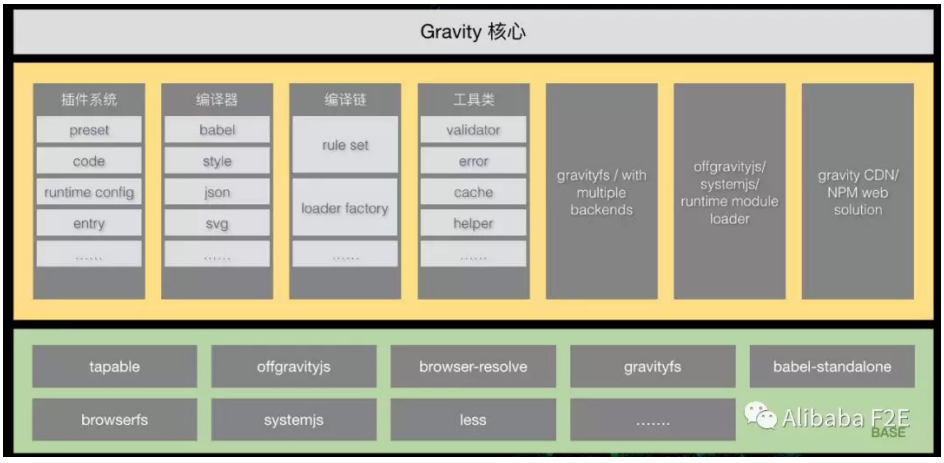
可以罗列下我们会碰到的问题：

- nodejs 文件系统
- nodejs 文件 resolve 算法
- nodejs 内置模块
- 任意模块格式的加载
- 多媒体文件
- 单一文件多种编译方式
- 缓存策略
- 包管理
- ……

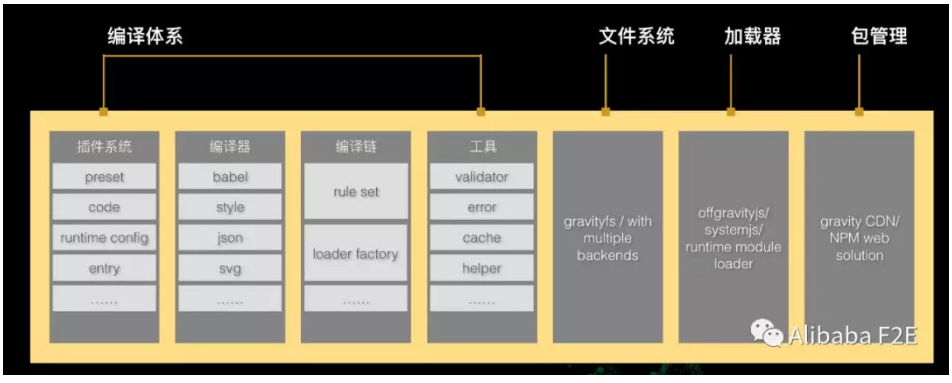
总结下其实是四个方面的问题：

- 如何设计资源文件的加载器
- 如何设计资源文件的编译体系
- 如何设计浏览器端的文件系统
- 如何设计浏览器端的包管理

Gravity 架构大图



架构图



从这个图中其实可以归纳出，我们就是在解决上面提到四个问题，即：

- 如何设计资源文件的加载器
- 如何设计资源文件的编译体系
- 如何设计浏览器端的文件系统
- 如何设计浏览器端的包管理

名词解释

这里会提几个名词，方便之后大家理解。

Transpiler: 代码 A 转换为代码 B 转换器

Preset: 是一份构建描述集合，该集合包含了模块加载器文件加载的描述，转换器的描述，插件的描述等。

Ruleset: 具体一个文件应该被怎么样的 transpilers 来转换。

这里可以衍生出来说一说为什么要设计 Preset 的概念。在文章的最前面我提到了现在要构建一个前端的项目学习曲线非常陡峭。在社区我们能看到两种解法：

- create-react-app: 它把 react 应用开发所需要的所有细节都封装在了这个库里面，对用户只是暴露了一些基本的入口，比如启动应用，那它的好处是为着这一类 react 应用开发者提供了极致的体验，降低了整个学习曲线。但缺点也比较明显就是 CRA 并不支持自定义配置，如果你需要个性化，那不好意思，你只能 eject，一旦 eject 之后后续所有的配置就交给应用开发者，后续便不能再融入回 CRA 的闭环了。
- @vue/cli: 它和 CRA 一样做了配置封装，但是和 CRA 不一样的地方是，它自身提供了一些个性化的能力，允许用户修改一些参数。

通过以上两者不难发现，他们都在做一件事情：解耦应用开发者和工具开发者。

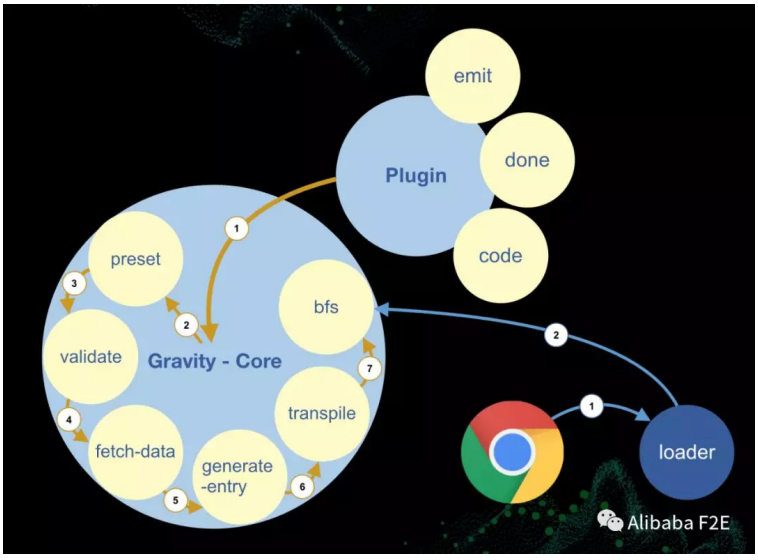
再回到 Preset，我的角色是工具专家，提供一系列的底层能力，而 Preset 则是垂直业务专家，他们基于我的底层能力去做的业务抽象，然后把业务输出为一个 preset。而真正的应用用户其实无需感知这部分的内容，对他们而言或许只需要知道一些扩展配置。

Gravity 的消费链

在 Gravity 的设计中，Core 层其实没有耦合任何的具体业务逻辑（这个逻辑指的是，比如 react 应用要怎么执行，vue 应用要怎么执行等），Core 层简单来讲，它是实现浏览器实时构建的事件流注册、分发、执行的集合。而具体的业务场景，比如 React，Vue，小程序等则是通过具体的 Preset 来实现整合。而我们的 Preset 会再交给对应的垂直场景的载体，比如 WebIDE 等。

专题深入

专题一：插件机制



事先我们来看一看 Gravity 是如何运作的，上图只是一个流程示意，但也能说明一下流程上的设计。注意看我们在 Plugin 类上定义了一些事件，而这些事件是允许被用户订阅的，那 Gravity 在执行时，会对这些事件先尝试绑定。在进入到相关的流程时，会分发这些事件，订阅了该事件的订阅者，就会在第一时间收到信息。举例来说，Plugin 中的 Code 描述了如何来获取代码的方式，而在 Gravity Core 的整个生命周期中，会调用 fetch-data 去分发 Code 事件，如果说用户订阅了该事件，那么就会马上响应去执行用户定义的获取代码的方式，并得到代码进而告诉内核。

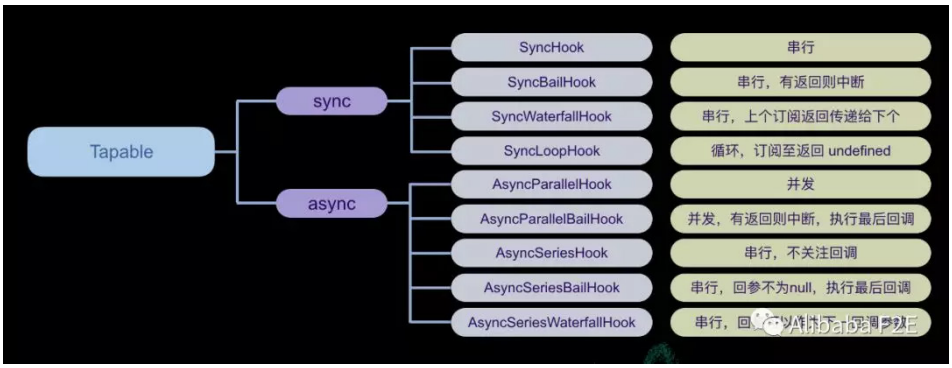
所以不难看出，Gravity 本质上是事件流机制，它的核心流程就是将插件连接起来。

既然如此，其实我们要解决的重点就是：

- 如何进行事件编排
- 如何保证事件执行的有序性
- 如何进行事件的订阅和消息的分发

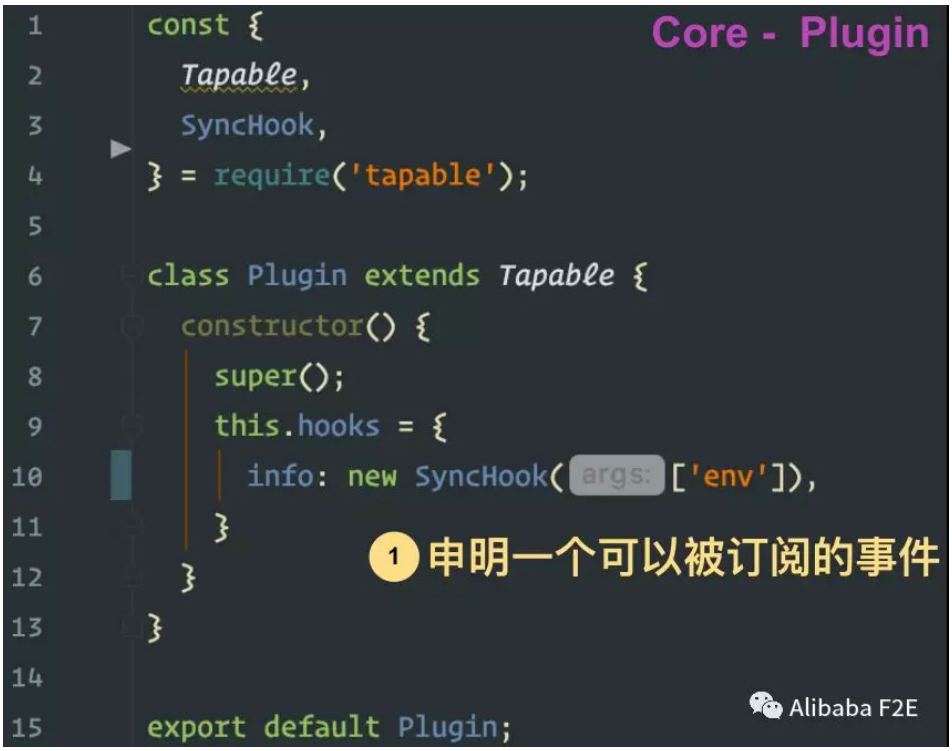
说到这里不知道大家是不是有一种似曾听闻的感觉，没错，其实这些思路都是来自于 webpack 的设计理念，webpack 是由一堆插件来驱动的，而背后的驱动这些插件的底层能力，来源于一个名叫 Tapable 的库。

Tapable 这个库我个人非常非常非常喜欢。原因在于它解决了很多我们在处理事件时会碰到的问题，比如有序性。另外要做一个插件系统的设计其实很简单，但后果是对用户会有额外的负担来学习如何书写，所以我选择 Tapable 来做还有另外很重要的一个原因，用户可以继续延续 webpack 插件写法到 Gravity 中来。



这里我罗列一下 Tapable 所拥有的能力。并用伪代码的方式为例来讲一讲我们在核心层如何定义一个插件（定义可被订阅的事件），业务专家如何来使用这个自定义插件（订阅该事件），以及我们在核心层如何来执行这个插件（绑定，分发）。

定义插件：



自定义插件：

核心层绑定和分发：

所以 Gravity-Core 重在事件的编排和分发，Plugin 则重在事件的申明，而 Custom plugins 则是订阅这些事件来达到个性化的目的。

专题二：如何实现编译链

在讲如何实现前，我们再回过来看下 Ruleset，在架构大图小节中我说明了下，Ruleset 是用来描述一个文件应该被怎么样的 transpilers 来转换。而 Ruleset 的生

成其实是依赖于 preset 中 rule 的配置，这一点，其实 Gravity 和 webpack 是一致的，这种设计原因有两点：1. 用户可以沿用 webpack 的 rule 配置习惯到 Gravity 中来；2. 我们甚至可以复用一些现有的 webpack loader，或者说让改造量变得更小。

在这里我们以小程序中的 axml 文件为例，假设现在有一个 index.axml 需要被被编译，此时会通过 Preset 中 rule 描述，最终被拆解为一个 ruleset，在这个 set 信息中我们可以获取到 index.axml 文件需要经过怎么样的转换流程（也可以理解为该 index.axml 文件需要什么 transpiler 来进行编译）。该示例中我们可以看到，index.axml 需要经过一层 appx 小程序编译后再把对应的结果交给 babel 进行编译，而 babel 编译的结果再交给下级的消费链路。

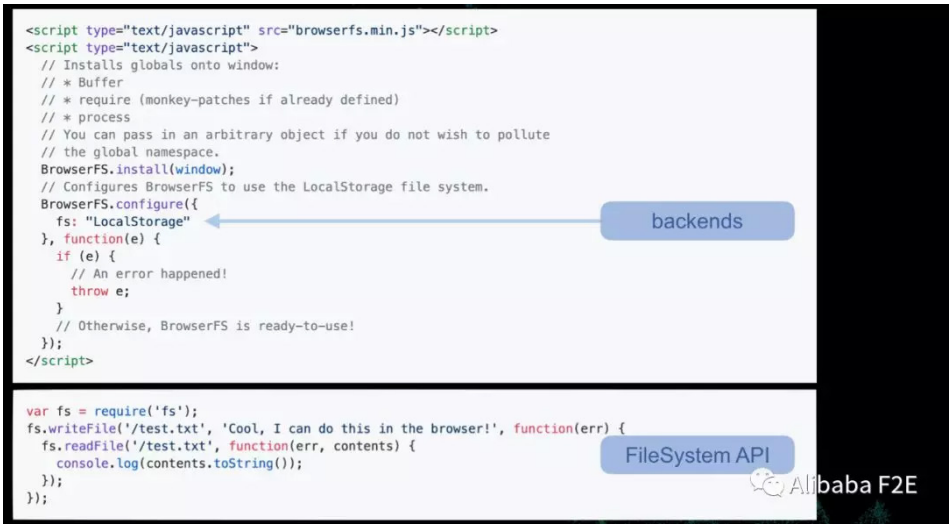
暂时抛开复杂的业务层实现，我们想一想要实现这条串行的编译链路的本质是什么。相信大家都能找到这个答案，答案就是如何保证事件的有序性。既然又是事件，是不是我们又可以回过来看一看 Tapable，没错，在 Tapable 中就有这样一个 hook - AsyncSeriesWaterfallHook，异步串行，上一个回调函数的返回的内容可以作为下一回调函数的参数。说到这是不是很多问题就迎刃而解了。没错，那么在 Tapable 中实现编译链是不是就被简化为如何基于 ruleset 动态创建 AsyncSeriesWaterfall-Hook 事件，以及如何分发的问题。

文件系统和包管理

BrowserFS

如果我们在浏览器中没有文件系统的支撑，其实可以想象本地的文件的依赖将无法被解析出来（即无法完成 resolve 过程），所以实现浏览器内的文件系统是实现浏览器编译的前提条件。这里幸运的是 John Vilk 前辈有一个项目叫做 BrowserFS，这个库在浏览器内实现了一个文件系统，同时这个文件系统模拟了 Nodejs 文件系统的 API，这样的好处就是，我们所有的 resolve 算法就可以在浏览器内实现了。同时这个库最棒的一点是提出了 backends 的概念。这个概念的背后是，我们可以自定义文件的存储和读取过程，这样文件系统的概念和思路一下子就被打开了，因为这个文件系统其实本质上并不局限于本地。

在这里我们可以大概看下如何使用 BFS。



包管理

有了文件系统我们再来想一想前端不可分割的一个部分，包管理。

思路一：浏览器内实现 NPM

这个思路是最容易想到的，通常做法是我们会拉取包信息，然后对包进行依赖分析，然后安装对应的包，最后把安装的包内容存储到对应的文件系统，编译器会对这些文件进行具体的编译，最后把编译结果存在文件系统里面。浏览器加执行文件时，模块加载器会加载这些编译后的文件。思路很通畅。但是这种方式的问题是原模原样照搬了 npm 到浏览器中，复杂度还是很高。

缺点：

- 首次很慢
- 存储量大
- 依赖 NPM Scripts 的包得不到解决

思路二：服务化 NPM

这一块的思路其实来自于对我影响最大的两篇文章

- stackblitz 的 turbo CDN 思路
- codesandbox 的 dependency-packer 思路

非常精彩，我也写过一些文章来分析他们。但是 stackblitz 和 codesandbox 在 npm 思路上各自都有一些缺陷，比如 stackblitz 的资源分发形式，codesandbox 的服务端缓存策略。

服务化的 NPM 本质是基于网络的本地文件系统。怎么来理解这句话呢？我们来举个例子，一起来构想一下如何基于 unpkg / jsdelivr 做一个的文件系统。

假设我们现在依赖 lodash 这个库，那么在我们对接的文件系统里面会发一个请求给远程的 unpkg，该请求可以获取到完整的目录结构（数据结构），那么在得到这份数据后，我们便可以初始化一个文件系统了，因为我们可以通过接口返回的数据完整的知道目录内会有什么，以及这个文件的尺寸，虽然没有内容。所以此时文件系统

内包含了一整个完整的树结构。假设此时我们通过 resolve 发现，我们的文件中确切依赖了一个文件是 lodash/upperCase.js，这个文件系统事先需要做的事情是先在本地文件数里面找下是否存在 upperCase.js，这里毫无疑问是存在的，因为我们在这个接口中能找到对应的 upperCase.js 这个文件，能确定肯定是在文件系统里面是有标记的但是如之前所说 meta 信息只是一种标记，他是没有内容的，那么接下来我们就会去往 unpkg 服务器上那固定的文件，发送请求获取该文件内容，至此我们的基于 unpkg / jsdelivr 的文件系统就设计好了。

所以服务化 NPM 的关键是：

需要我们抽象

- 如何设计包管理依赖的下发逻辑

需要我们包装

- 如何把这个下发逻辑桥接到对应的文件系统

注明：下发逻辑指的是我们按什么规则去下发用户的 dependencies。

服务化 NPM 的要点是：

- 建立一个下发策略，比如基于项目维度的 deps，依赖的下发是基于依赖包的入口文件分析所产生的依赖文件链
- 补充在默认下发策略不满足需求时，如何建立动态下发的过程
- 依赖下发的数据结构，如何体现依赖关系，父子关系等
- 如何快速分析依赖关系
- 如何缓存依赖关系
- 如何更新缓存的依赖关系
- 如何把以上这些信息桥接到我们的文件系统

未来

提到 Gravity 的未来，其实更多的是我对他的憧憬，总结一下可以是三个要点。

PVC

- Pipelined 流水线化

垂直业务场景所对应的 Preset 的产出，可以按着某个流程，用极少的成本自由组合一下就可以使用。

- Visualized 可视化

所有搭建 Preset 、以及 Preset 内配置都可以通过可视化方式露出。

- Clouds 云化

Gravity 服务化。

关注 Alibaba F2E
了解阿里巴巴前端新动态

访问开发者社区
扫码领取更多免费电子书

D2 前端技术论坛
扫一扫二维码，关注我吧

扫一扫二维码
获取超全 D2 大会 PPT 资料下载