从前端架构的出现到微前端架构设计

写在前面的话

对于大前端开发岗位，在技术实现上各行业以及应用体系区域完善，也建立了统一的技术栈和规范，这就意味着如果想要从编码为主的开发岗位进一步越迁到架构设计岗位，开发者需要具备完整的技术视野和架构设计思想，完全掌控从抽象的设计层面到具体的落地层面，能帮助前端开发者在行业内走向一个新的高度。

1. 大前端的架构变迁

随着互联网技术的演进，大前端岗位逐渐成为IT行业的一大不可或缺的岗位，大前端从酝酿到出现经历了几代技术的演进。

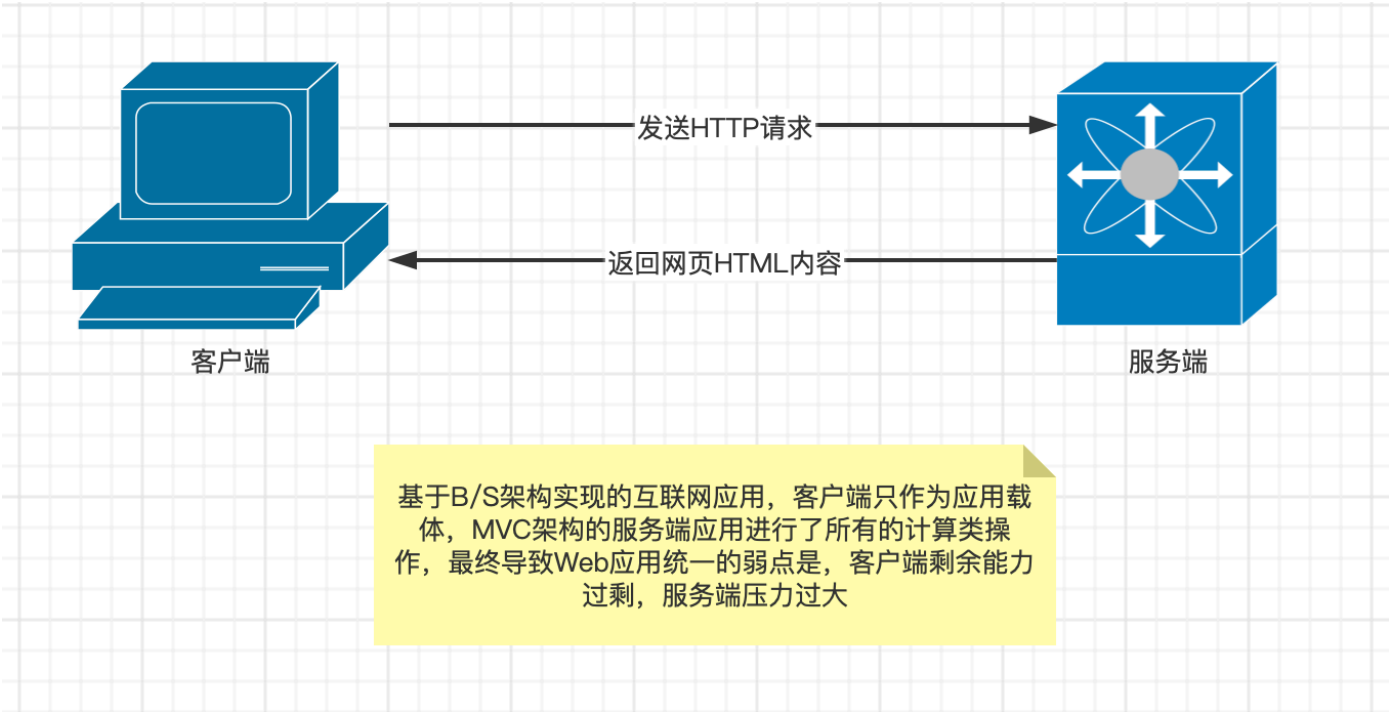
1.1 切图仔时代

早期的前端并不是单独的编程岗位，它更贴近于设计岗位。国内早期的互联网或软件项目大多以MVC架构为主，通过服务端技术实现的MVC架构几乎都是基于个平台的动态网页技术实现的。动态网页技术是通过静态网页结合服务端渲染模版进行网页绘制，该时代的前端不需要使用模版引擎或大量的JavaScript编程，而通常的解决方案便是UI设计将设计稿完成后，通过HTML+CSS+少量成熟的JavaScript库变成网页，而数据采用服务端模版引擎，如JSP、ASP、Smarty等进行渲染，所以当时的前端以设计岗位为主。

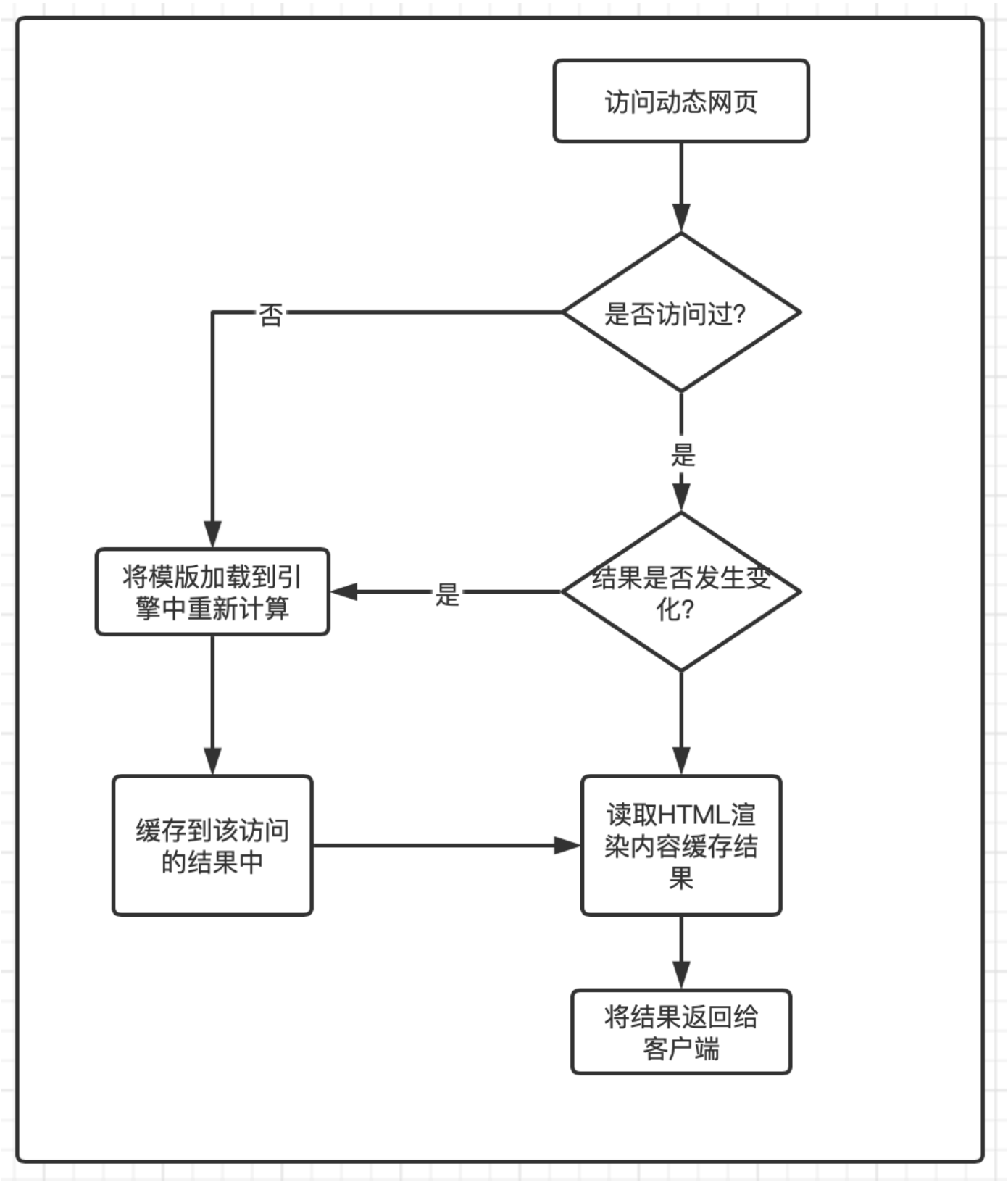
1.2 基于AJAX技术的前后交互时代

1.2.1 MVC架构的疲软期

随着互联网网民的不断增加，推动互联网应用架构提升规模和复杂度。这个时代的Web应用不在满足于提供基础服务，更注重交互、性能和用户体验，此时基于WebMVC架构实现的应用在大量用户群体上逐渐显得心有余而力不足，其具体原因，如图。



MVC架构应用之所以难以支撑发展的较快的用户基数，其原因是MVC架构的视图解析引擎在服务端，所有的网络请求压力均由服务器承受，这就导致了访问频繁的Web应用会经常出现访问阻塞的情况导致应用界面完全打不开。其具体原因，如图。

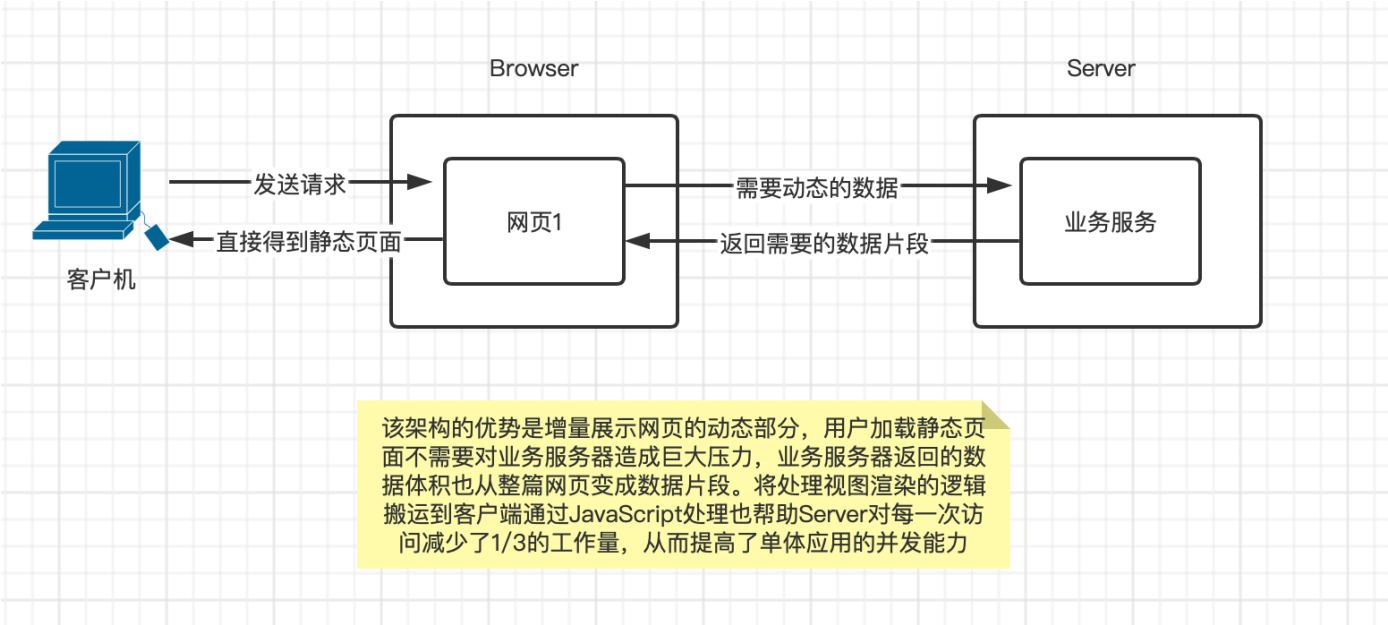


观察图片可以得知，该架构中所有的计算步骤都由Server处理，这意味着，若10000人访问该Web应用，会导致Server必须为这10000人分别计算其所请求的视图结果，否则用户无法获取正确的页面展示，而这个过程中还会包含Server->DB的访问时间，并且这个过程是一个同步操作，虽然Server是多线程处理客户端请求的，但整个过程中只要渲染逻辑没有跑完，则用户在浏览器中看到的只有空白的网页，因为自始至终MVC架构的动态网页项目中，浏

览器只作为结果的展示容器存在。

1.2.2 前后分离的雏形

当部分应用架构师意识到该问题的严重性后，发现了客户端性能过剩的问题，便有了前后分离架构的雏形。该思想是将Server端处理的视图渲染逻辑搬运到Browser中处理，应用中将视图部分转移到了单纯的静态资源服务中，这样会使得访问应用的用户无需等待服务端处理业务，便可以直接加载大部分静态视图，视图中所包含的动态数据通过AJAX技术与Server交互，最终通过JavaScript执行模版的渲染，如图。



此架构更加符合当时的互联网基础建设情况以及大多数的用户需求，举个简单的例子，当参加完高考的考生迫不及待的查询高考成绩时，若该场景使用MVC架构处理则其流程为：

1. 考生访问成绩查询页面（触发一次服务端渲染）
2. 考生在查询成绩页面中输入个人信息准考证号并提交一次表单
3. 服务器根据考生数据连接DB进行考生成绩查询
4. 服务器将DB返回的数据保存到合适的变量中如List、Map等
5. 服务器将数据通过JSP或ASP等模版引擎进行计算生成考生成绩视图
6. 服务器将完整的成绩单页面代码返回给Browser

此架构场景下，必须进行到第6步，考生才能在浏览器中查看到自己的考试结果，若当前访问服务器的人数较多，服务器压力巨大，此时计算网页结果或连接DB部分就会消耗更大量的时间，导致查询成绩时出现网页超时的情况。而当使用前后分离模式后其步骤为：

1. 考生访问成绩查询页面（从静态资源中直接加载HTML）
2. 考生在查询成绩页面中输入个人信息准考证号并通过AJAX提交数据（此时网页不需要重新加载）
3. 服务器根据考生数据连接DB进行考生成绩查询
4. 服务器将DB返回的数据封装成JSON或XML对象返回
5. 浏览器将收到的成绩数据通过JavaScript加载到网页需要展示的位置

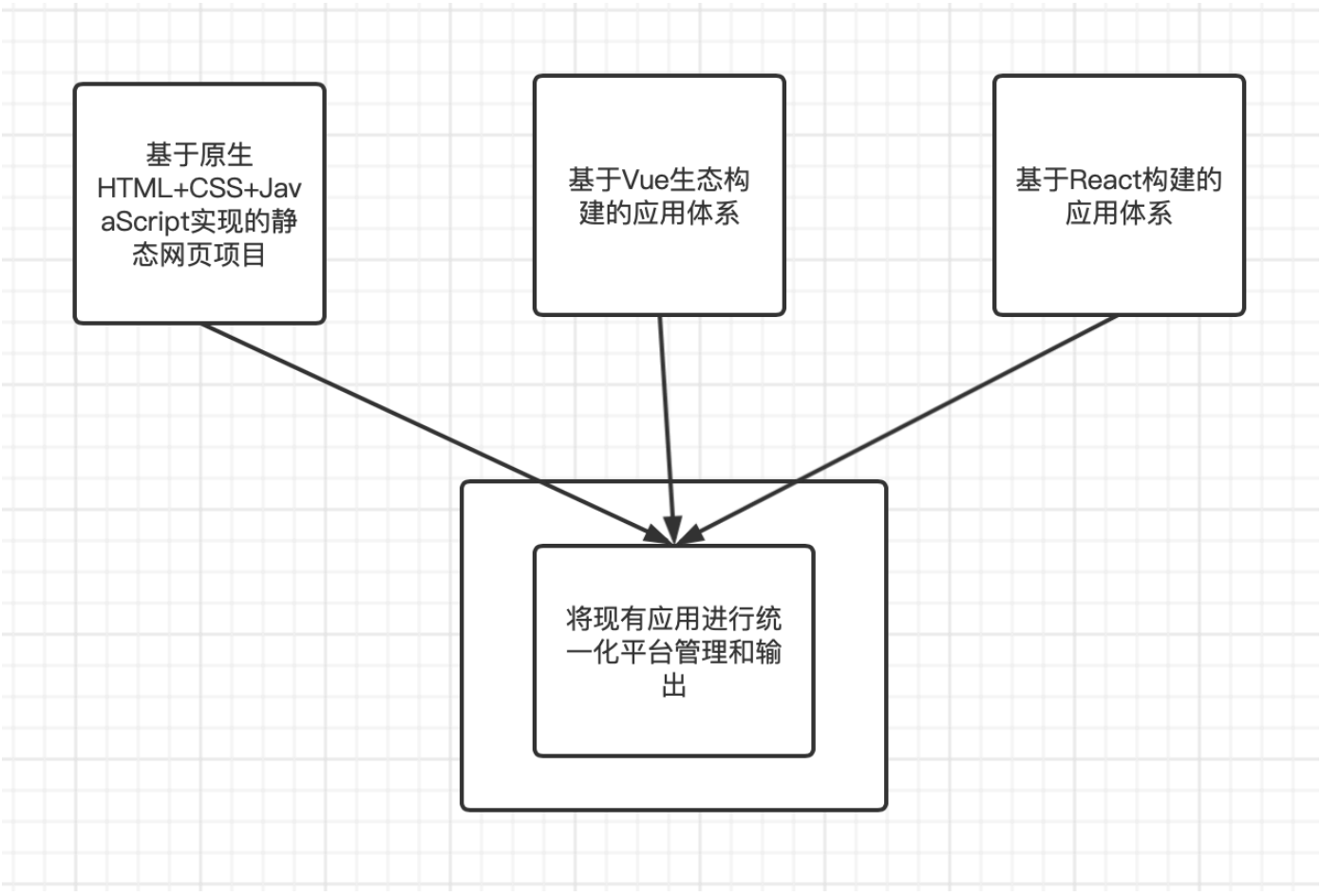
使用此步骤会发现，服务器无需处理视图解析，并且也不需要返回大量的数据，从处理过程到整个交互产生的流量上都实现了较好的优化。

1.2.3 基于前端MVVM架构的前后分离架构

此架构典型存在于现在的Web应用开发场景中，遂本文并不多做介绍，通过1.2.1和1.2.2的介绍便可以明确的解释，为什么当今大量的Web应用采用JavaScript的模版引擎（Vue、React或Angular等）来进行视图数据的渲染，以及为何大量的Web应用都是钱后分离架构的。当然，前后分离架构也对服务端在另外一个层面上造成了巨大的压力。虽然前后分离技术使得服务端在处理每一个请求的工作量大大的减小，增量获取数据的客户端会在同一个网页中瞬间产生大量的AJAX请求，会在数量上对Server产生另一种冲击。所以现今的前后分离架构的Server端都是分布式+集群模式（此部分由于非前端主题本节进行掠过，对服务端架构演进感兴趣的同学可以期待后续的内容，提前种草～）。

1.2.4 基于服务端分布式概念的微前端

大前端架构形态演进至今已经趋于一个阶段的成熟，但它并没有停止演进，目前的分布式微前端技术就是下一代前端架构的一个基础雏形。该技术思路源于服务端的分布式思想，可以实现应用的纵向拆分、微应用化、云组件化等等好处。微前端架构思想之所以出现，是因为在前端架构发展历程中，历代架构体系泾渭分明，同期架构中不同技术栈的生态又造成了应用间的生殖隔离，这就导致若想将上一代或近代遗留的成熟技术产品做统一入口形成大平台就不得不将其重新开发，这个发展路线显然是不对的，上面描述的现状，如图。



当面对以上需求时，由于各技术栈存在生殖层面的隔离，无法统一到同一个页面或路由系统中，这就导致了想要将现有资源整合，通过传统的工程化开发模式也难以对其实现良好的解决方案。若重新开发必将面临着巨大的成本和试错挑战，若真的重新开发也势必要统一技术栈，并针对该技术栈制定长远的技术方针，这种方式显然是风险极大的。综合以上问题，微前端思想便是未来混合技术栈实现前端微应用化的出路。微前端可以实现解决应用隔离，同一个大型应用下可以采用多种技术栈和生态进行开发，其沙箱运行模式的思想可以很好的实现技术栈隔离，并提供了完善的跨技术栈通信系统。当然微前端架构也存在多种设计思路和呈现模式并不是一种，后文会对其做完整的介绍。

2.架构设计思想篇

学架构先学思想，好的设计思想决定架构的成败。经过架构演进的学习，相信大家对应用架构的设计思想已经有了一些基本的认识，本节的目的是介绍前端的软件架构设计思想以帮助大家进一步理解微前端架构的思想和实现。

2.1什么是应用架构

应用架构最简单的理解就是盖房子，比如生活中想要盖一栋大房子，需要从房子的设计图开始，进入房屋的设计期，设计的过程中还要考虑每个环节需要的材料，物理承重计算等等问题。然后才会进入选材料，实际打地基，根据不同结构请不同的人进行落地实施。后面还要对盖好的房屋进行验收和实际售卖等等操作，还需要设计好房屋的维修周期等等。

应用架构指的是从设计应用到构建应用的整个过程以及其包含的所有环节。对于前端开发岗位的技术人员来说，很多人对架构设计的理解仅仅是片面的，比如开发Vue项目或React项目时，使用路由和状态管理系统，配合UI框架，最后加入网络请求框架和一些优化手段。这个过程是架构的一个具体体现之一并不代表前端的整个架构就是仅仅而已。应用架构包括的内容繁多流程复杂，比如实际开发过程中，脚手架的设计和实现就包括了脚手架工具的内部架构从设计到落地到可伸缩概念，比如应用开发中除技术组成外，还需要设计业务分别和业务组成，所以业务组成也是架构设计中的一部分。再比如应用架构构建过程中，项目从开发环境到发布上线的整体流程，测试步骤和持续集成方案，这些都是应用架构中必不可少的部分。

不以实现为目的的架构都是耍流氓，所以任何应用架构，软件开发和互联网行业中的目的都是落地和实现。所以有如下总结：

1. 一个无法上线的应用架构，算不上好的软件架构。
2. 一个没有人能完成开发的软件架构，算不上具有可行性的软件架构。
3. 一个在现有技术上不可行的架构，算不上合理的软件架构。

所以一旦我们谈及软件架构，需要讨论的第一个重点就是因地制宜。比如在一线互联网公司的软件架构，都属于行业内的顶级架构设计方案，但是该架构在中小型企业并不是适合的架构，结合以上三点考虑的话，一线顶级架构在中小型企业就很容易成为2和3点所覆盖的内容。若使用优秀的软件架构，必须结合优秀的软件开发人才，若公司体量和技术能力有限，则选择眼下最合适公司情况的架构才是最好的选择。

2.2 架构设计的几种原则

不同的人在设计架构时会产生不同的风格，在细节把控上也会出现特有的风格，虽然最终呈现的结果会产生很大的差异，但是在设计过程中大部分人会秉承着几点相近的思想，如下：

1. 刚刚好
2. 演进式
3. 可持续

2.2.1 刚刚好架构

刚刚好的意思是指架构设计者设计的架构恰好符合公司内部的使用，不做过多的无用设计也不会没有任何规则让开发者各自发挥，会秉承着有一个大方向的统一化，和细节的自由度最终目的是完成该应用。

在架构设计的路上，大多数设计师会在架构设计过程中埋下很多伏笔并为其做落地实现，以确保在未来的架构演进上可以预留通道，这个思路显然是好的。但面对实际场景时，针对未来预留过多的设计更多的会变成过度设计。因为在设计初期，为未来预留的设计都是假象的现在并未发生的，该设计的实现对目前的架构时期并不是必要的。所以这种预留和可能会在需要使用其的将来，由于实现方案发生变化而导致之前的预留都是无用的。软件架构设计阶段更多的需要解决现有架构基础上可能会发生的问题，并作出针对性的完善设计，这样可以提高架构本身的健壮

性。

设计过少会产生架构自身能力不足的问题，由于架构设计内容过少，导致架构可扩展性下降，无法适应更多的变化。无论是架构设计过度还是架构设计不足都是架构设计场景中很容易出现的问题，这个问题在实际开发场景中很难找到一个平衡点，为什么？设计者通常会存在两种思想：一种是认为未来的人一定比现在的人聪明，所以少做一些设计，让未来的人来弥补；另一种是认为下一个接手项目的人一定技术不如自己，所以会多做一些设计。

2.2.2 演进式架构

没有任何一个公司从设计时就将架构直接定义为现在的淘宝或阿里云一样复杂，任何架构都是随着时间和各方面变化逐渐过渡到如今的复杂度的，所以在架构设计过程中一定要考虑的就是架构演进，要将架构设计为当前刚好符合，未来还可适应。

此外架构设计也要参考公司项目的开发模式，比如针对敏捷开发场景，并不适合做过多的细节设计，因为敏捷开发场景在实际落地时会让开发流程变得异常混乱，项目频繁迭代，并不适合做过多的提前设计，这会导致提前设计往往都打水漂了。若公司采用的是传统大型应用开发流程，从需求分析，概要设计开始进行软件开发，那么细节设计将觉得未来软件是否稳定的成败。

2.2.3 可持续架构

可持续是指应用架构在发展过程中，可以逐渐的通过改造和迭代慢慢适应时代发展，而不会发生突然到了某个节点时该架构走向死路的情况，这就要求架构师具备前瞻性思想。

从某种意义上来说，持续性和演进式架构很像，他们的区别是：演进式架构是指架构上的一些变化，而持续性针对的是开发人员的变化。架构的持续性原则的意图是，敢于修正架构中的错误部分，让架构变得尽善尽美，在修正的过程中可能会带来一些新的问题，但是很快也会被纠正过来。

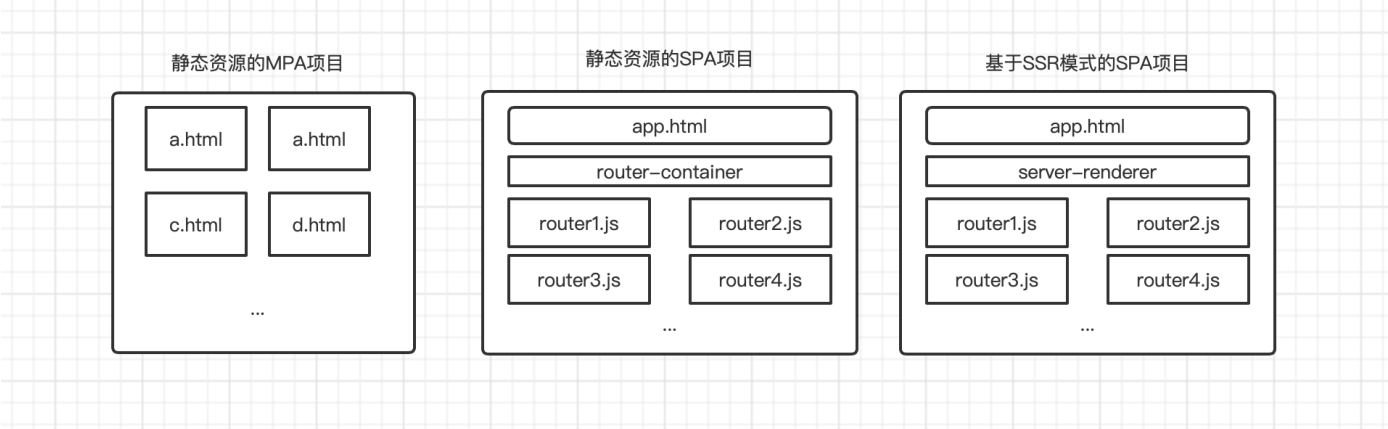
在架构演进的过程中，若未来可选方案有多种时，不要纠结各种演进方案本身，这种情况可以延迟架构的修正，随着应用发展获得更多参考数据时再确定未来的架构走向，否则会花费大量的时间在研究架构路线上，得不偿失。

3.微前端架构篇

3.1 单体应用架构和微前端

3.1.1 单体应用架构

单体应用架构指的是大前端项目部署阶段，一个工程化项目对应一个应用，该应用内部可以有很多的业务模块和角色，随着应用的膨胀项目规模也会跟随一起膨胀。单体应用架构中包含了常见的MPA项目和SPA项目，是目前为止市场现存最多的前端架构之一，如图。



单体应用架构主要包含几大类型：

1. 基于静态资源的多页面应用（MPA）：该应用为传统的HTML+CSS+JavaScript项目，该项目的好处是快速建站，无需架构和复杂的技术栈。
2. 基于静态资源的单页面应用（SPA）：该应用目前大多数为基于MVVM思想实现的单页面应用，其特点是通过独有的路由系统管理视图组件，将前端项目实现组件化和工程化，让大前端Web应用可以实现大型应用的构建。这种单体应用的好处是用户体验非常好，资源加载快，但是需要一定的技术架构经验和对SPA生态具备较高的理解，否则构建的项目会在应用体积和合理化方向出现问题。
3. 基于SSR模式的SPA项目：由于SPA的特点是通过JavaScript模版引擎渲染DOM，导致了服务器实际返回的网页代码中并不包含运行后的应用节点，这种架构对SEO造成了负担。SSR模式是将Browser中的JavaScript模版渲染部分再次交给服务器处理，并将其结果返回给客户端，这种方式相当于将过去的动态网页模版渲染的部分提升到一个前置Server中，这种方式可以提升网页对SEO的友好度，也可以将过去的异步渲染的SPA转化成直接渲染处理结果，SSR由于与过去的动态网页思路相同，也需要在Server中通过模版引擎进行计算，所以更加适合数据不频繁变化的应用开发，使用场景有针对性，并不代表SSR是优于传统SPA的。

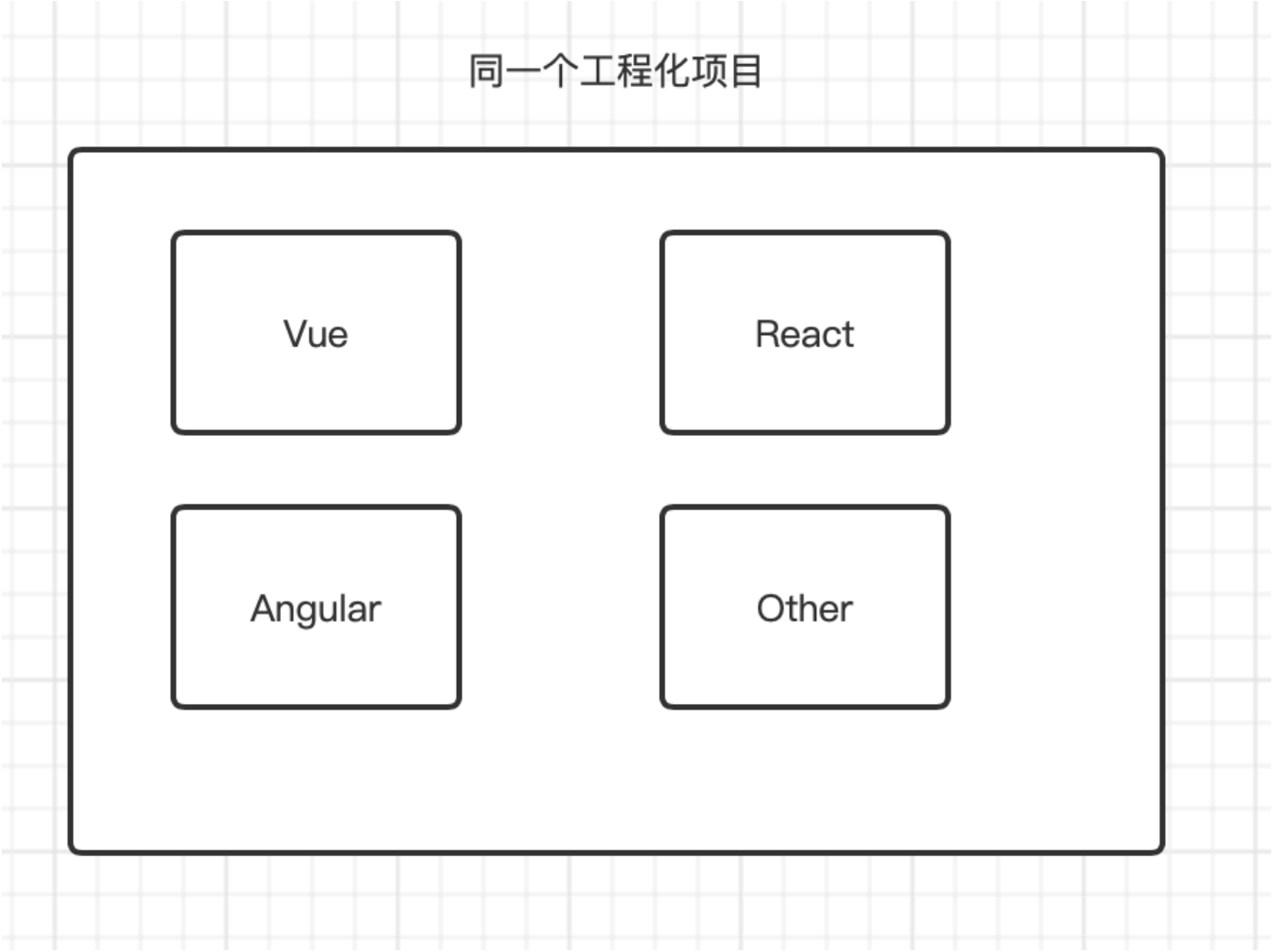
3.1.2 微前端架构

单体应用架构和微前端架构有本质上的区别，正如分布式和集群是两种概念一样。微前端架构的思路并不是将传统的项目中融入多个技术栈，来实现对现有单一技术栈限制的项目实现多元化。

一个合适的场景：

当公司存在三个前端程序员并且短时间内无法找到可替代人群时，若该3人分别使用的技术栈微Vue、React和Angular，并且三人技术栈完全隔离互相不会对方的技术。在这种情况下，单体应用架构无法满足现有团队的开发需求。由于3人技术栈不同，若规定技术栈以React为主，则其他2人需要耗费大量的学习时间才能融入到团队协作开发中，其他风险另算。所以此时，如果存在一种软件架构，支持3人可以使用自己最擅长的技术栈进行业务开发，并且最终开发的还是同一个完整的项目，这样在技术团队的技术隔离上便有了里程碑一般的突破。

所以微前端架构在实际生产中解决的问题，如图所示。



微前端之所以能出现，来源于：沙箱运行环境思想。

什么是沙箱环境?

沙箱是一个天然封闭的隔离环境，他可以做到真正的环境隔离，这里最具代表性的体现就是虚拟机和Docker容器等，它们都是利用沙箱隔离环境来实现的。比如在同一个操作系统内使用N个操作系统，每个操作系统又有独立的CPU架构和注册表系统等，这样可以实现完整的环境隔离，MacOS上的文件和应用以及后台进程并不会影响Windows中的任何部分，本质上两个操作系统依赖相同电脑的硬件资源，这个就是沙箱环境的优势。

对于前端来说，由于任何模块解析的模式最终都要由HTML来进行网页的解析，所以实际编写的应用逻辑代码最终都是全局加载到同一张网页中的。这种情况想要实现跨技术栈开发，并在应用间实现技术隔离是很麻烦的，所以网页中也需要存在一套沙箱系统来针对不同的技术栈和应用拆分思想做相应的隔离，这里最典型的沙箱环境就是iframe。虽然iframe在浏览器标签中存在历史悠久，但它对微前端架构的新技术起到了很多作用。

JavaScript沙箱认知

关于JavaScript中的沙箱环境，是一种抽象的代码环境。大家熟知的闭包或模块管理方式，只能将作用域降级，可以包装全局变量，但此方式无法阻止包内变量访问外部所以这并不是一个完整的沙箱。

举个例子：

```
function test(){
  var a = 123
  console.log(a)
}
test()
console.log(a)
```

此函数会将JavaScript中的函数作用域封闭起来，保证函数中的a并不对外泄漏，这样便外部的a变量在运行时会触发not defined错误。

再举个例子：

```
var a = 123
function test(){
  console.log(a)
}
test()
console.log(a)
```

此案例中的a变量是全局变量，所以在test和外部都可以直接访问，这样就可以完整的展示为何闭包等结构不能单独作为完整沙箱环境，其原因就是沙箱的封闭环境除将环境做隔离区外，还需要将环境对外访问的通道“掐死”（这个掐死是选择性的，比如限制某些变量可以访问而某些变量不可以）。

再举个例子

```
var a = 123
function test(sandbox){
  //创建沙箱对象的代理
  let s = new Proxy(sandbox,{
    has(target,key){
      // 模拟作用域检测
      if(key!='console'&&!target[key]){
        throw(key+' is not defined')
      }
      if(target[key]){
        return true
      }
      // throw(key+' is not defined')
    }
  })
  // 如此该作用域将无法使用a变量
  with(s){
    console.log(b)
    console.log(a)
  }
}
test({b:111})
```

该案例配合了Proxy对象，浏览器API新增的Proxy对象可以通过代理的方式实时追踪对象内的各种状态变更，当在with作用域内部使用变量时，就会触发代理对象的has函数，通过has函数模拟作用域限制，with作用域中使用的b、a以及console变量都会触发has函数进行检测，这里为了描绘思想仅仅做了最基本的实现，排除console后，若target对象中不包含key则直接报key is not defined阻止程序运行，这样实现后，只要test函数中传递的对象中不包含的属性，即使全局作用域中存在也无法在with内部使用，这样便实现了对外层作用域访问的限制。

微前端架构的架构思想

微前端的主要目的是将大型的单页面或多页面应用进一步的拆分成N个子应用，比如过去一个10万行代码级别的前端应用，在微前端项目中可以通过拆解成10个子应用实现将每个应用的代码量降至1万行。这种拆解方式可以让前端子应用变得更加容易维护。通过实现应用自治可以实现应用单独部署，这样子应用及可以独立访问，又可以整合在主应用中。

微前端架构的思想主要体现在3个方面：

1. 应用自治

   应用自治指的是多个子应用汇总成一个完整的应用，这样一个大型的引用可以拆解成多个子应用交给不同的团队进行开发和维护，总项目组只需要制定完整统一的开发规则，个团队按照主应用的设计规则开发及可以实现开发上互不干扰，部署上互相独立，运行上多个统一。

2. 单一职责

   单一职责指的是与微服务的服务拆分类似，若了解后端微服务架构的服务拆分原则，应该会理解，被拆分的任何一个子应用都应保持单一职责，即每个应用只做一件事情，拆分的应用只做应用内分内的事情，不越权处理其他应用的职责。对于前端页面来说，大多数页面间是存在关联的，若用户访问过程中两个页面A和B存在互相依赖的关系，就没必要将其拆分成多个子应用，这样在即破坏了单一职责，在实际开发时，又回由于大量的数据关联导致拆分的两个团队需要大量沟通，失去了拆分的意义。

3. 脱离技术栈限制

   这也是微前端最直接的优势，可以实现同一个应用中既包含Vue又包含React和Angular，从而大大的降低了团队技术栈强一致性的要求，在人员招聘上可以给公司减轻不晓得压力。

3.2 为什么企业需要微前端

首先微前端并不是解决行业生产效率提升的银弹，他只不过是一个时代变化所需要而诞生的，从目前的技术演进情况看微前端在未来有着很多的应用场景并且能解决大量的问题，但这不代表今后所有的项目都要变成微前端架构。正如分布式微服务的后端架构出现后，还出现了基于容器编排的分布式容器集群架构，但是目前很多正在线上运行的应用程序服务端还在使用单体应用架构。

目前部分企业需要使用微前端主要由于一下几种原因：

1. 遗留系统的迁移

   任何企业在配合互联网和信息化建设发展道路上都会产生大量的技术沉淀和应用沉淀，这些都是对未来发展的宝贵财富。在这个过程中，为保证更好的对社会提供服务，各企业对中台的建设以及统一服务的建设都投入了非常大的精力。站在前端角度考虑，之前遗留的应用中，可能随着互联网发展的不同时期，应用建设所使用的主要技术和技术栈各不相同，这就导致了若想要将现有的应用变成统一化平台进行管理是很难的。所以企业在做前端应用的下一代建设上需要一门新的技术，来将现有的应用体系整合起来实现跨技术栈的应用编排和更加合理的分布式部署策略，来实现前端的一体化服务输出。

可能有些人会认为较老的应用可以通过使用新技术栈重新编写来实现技术栈的统一化，但是这个代价是极大的。比如企业遗留了一些较早的应用是针对动态网页技术实现的HTML单体应用，中间过渡了一批前后分离的应用，后续又积攒了通过Vue或React等不同技术栈实现的应用。若要将其统一化，势必进行大量的破坏性迁移（相当于重写）。首先，在重写上就会存在较大风险，因为不同时期的应用内部势必包含大量的业务体系，已经成熟的业务体系是通过不断迭代形成的最合理最过，若重写在业务稳定性上就会出现较大风险。再次，不同前后端架构所形成的应用在重构时还可能会牵扯服务端架构的重建，这会造成更大的风险。所以最好的方式是将现有稳定的应用想办法编排起来。

2. 配合后端一体化服务

   配合后端一体化服务的思想也是构建微前端的一个原因，由于分布式服务端的架构可以将服务接口全部看作云而形成一体化服务平台，通过统一入口，前端和后端通信时并不需要关心该服务具体运行在什么位置、属于什么项目。所以在前端构建的思想上与后端的解耦改变正好相反，前端部分在此场景希望的是将一切聚合到一个平台来配合服务端的统一入口。这里最典型的体现就是各大移动端应用，如：支付宝、微信等等，这些综合类APP内部包含了大量的服务端业务与复杂的服务端架构，但对于用户来说，用户并不希望每一个业务线安装一个独立的APP，而是主动希望通过一个应用来解决所有问题，这就是前端一体化的需求所在，为了达到这种一体化，将前端项目拆分成子应用是势在必行的（可以结合小程序等思考，理论类似，实际架构却又有些却别）。

3. 综合其优势的尝试

   剩下的微前端架构尝试便是基于其架构本身特点而来的，各个公司针对其架构特点以及公司目前适合的场景都会进行下一代技术架构的尝试。

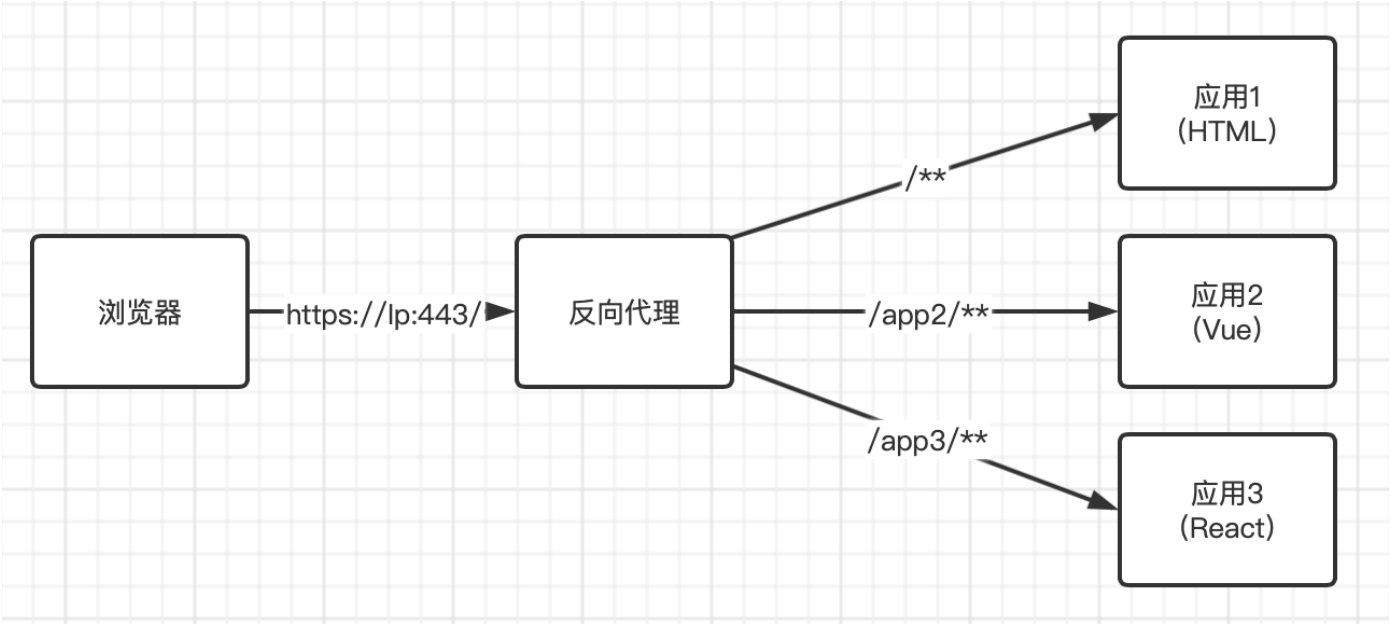
3.3 业内主流的微前端架构和应用案例

微前端架构并不是最近才产生的架构体系，最早在使用iframe进行跨应用管理时便已初步出现了微前端架构的雏形，所以本节主要介绍微前端架构是如何一路走来，并介绍个情况的微前端架构的实际应用案例。

3.3.1 路由分发模式

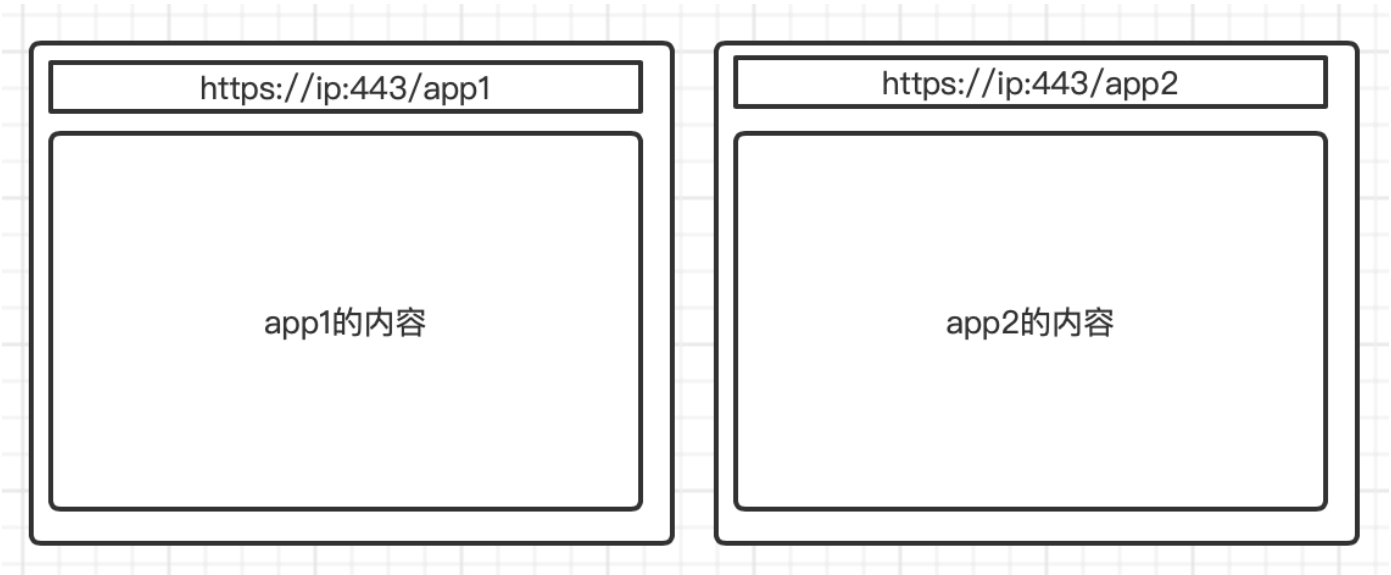
在还没有提出微前端概念时，就有很多公司通过反向代理服务器来编排公司不同业务线的前端应用，该模式支持将不同的前端项目部署在N个服务器上，每个服务器的前端项目可支持单独访问和代理访问。这样部署的好处是最终可以通过反向代理，将不同的前端项目通过代理服务器的路由合并到统一的域名和端口下，最终实现在用户眼里该平台仅仅是一个应用，而子应用间又可以通过单点登录的方式来共享登陆状态。

路由分发模式的落地实现，如图所示。



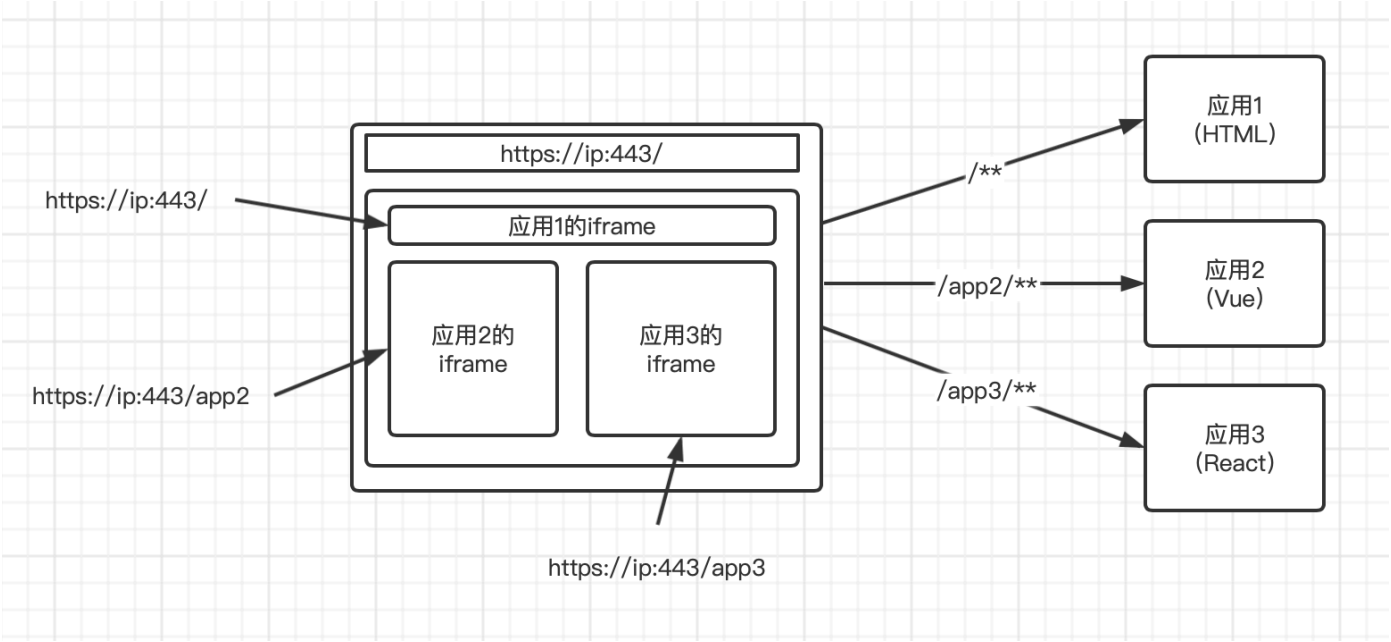
相信该架构是目前互联网体系内使用最多的微前端雏形架构，大量的公司通过这种设计方式将复杂的应用系统拆分成多个应用，在多个应用间通过服务端技术来实现登录状态的共享和通用数据的共享，这种方式实现的应用拆分和编排很好的解决了企业遗留系统的微前端迭代。

当然，这种方式也并不是微前端架构理想的解决方案，因为该路由系统必须借助反向代理服务器（如Nginx），路由编排是通过反向代理实现的。而这种方式看似将应用融入了一个平台，但实际上应用间仍然是独立存在的，所以与服务端的微服务不尽相同。若用户从app2跳转到app3的界面时，仍然需要将视图重新加载。基于反向代理的路由编排实现的微前端项目存在一个本质上的弊端，即子应用无法在同一个窗口中同时存在，切换到新的路由时整个窗口都会发生变化，如图所示。



3.3.2 基于ifame容器的微前端

iframe做为内嵌页组件存在于Web项目的历史相当悠久，他可以实现在一个网页中通过URL路径加载另一个网页，两个页面间的CSS和JavaScript是天然隔离的，iframe容器还提供了postMessage()等API帮助开发者在不同的内嵌页面实现互相通信，iframe就像一个天然的HTML沙箱容器，为微前端提供了入口。所以结合路由编排和iframe容器便可以将目前最早期的前端项目轻易的微前端化，如图所示。



虽然iframe容器解决了单独路由系统无法在相同页面中加载N个项目的问题，但其自身也存在一定的弊端：

1. 网站SEO的优化问题

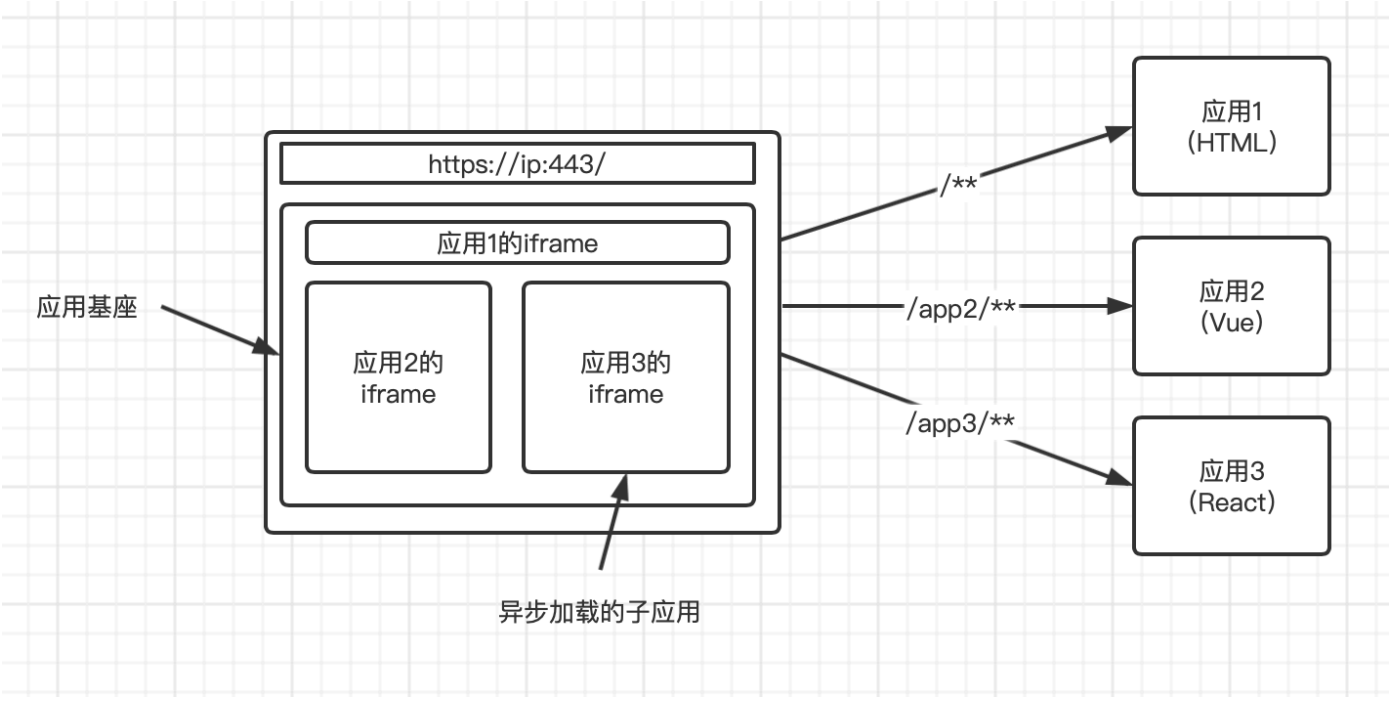
   SEO问题的体现于静态SPA的问题相同，基于iframe容器加载的应用并不能在网页源代码中直接展示子应用的源代码而导致首页的关键字无法被爬虫机器人获取。若为自应用单独定义SEO并对外暴露URL地址，则会导致爬虫机器人获取的快照信息直接指向子应用，这种情况在搜索引擎中打开的视图就是https://ip:443/app2内部的视图，并没有外部容器了。

2. 保存子应用状态的问题

   iframe除存在SEO问题外，在状态保存上也存在比较重大的难题，iframe与SPA的Router架构不同，由于iframe采用HTTP请求加载数据，一旦页面加载也会触发iframe的加载，这就导致了当视图切换到新页面再跳转回原页面时，iframe默认无法记录其内部上一次访问的应用地址，导致重新回到默认的src位置。这种情况虽然可以通过使用JavaScript制造缓存的方式解决，但依然避免不了iframe内部每次重新加载的问题，若子应用内部存在跳转参数，做状态保存更加困难。

3.3.3 前端微服务化架构

前端微服务化架构是下一代微前端解决方案的思想核心，该思想来自于后端的微服务思想，基于“注册中心”的应用自治环境。前端微服务化指的是前端应用中存在一个类似“注册中心”的服务，该服务可以管理应用内的所有子应用的完整生命周期，并与子应用建立通信。通过模块化的加载方式将子应用与主应用组合，最终形成一个整体的应用，如图所示。

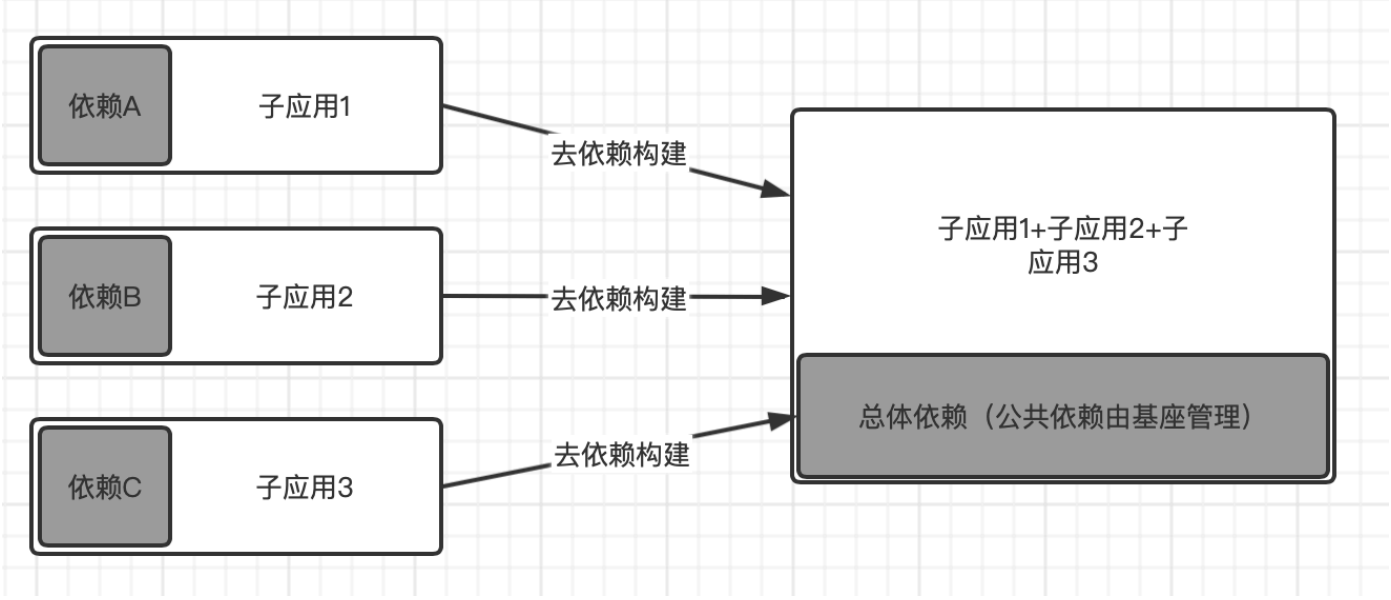


微前端化的思想模式类似于将Nginx反向代理和iframe整合后的设计思路，其应用加载并不通过反向代理服务器而是通过基座应用进行应用和应用对应的路由进行注册，在基座应用上注册路由的应用可以看作是组件，所以应用基座可以看作是反向代理服务器。通过之前的学习会了解到通过路由分发的模式，一个页面中只能加载一个自应用，而前端微服务化之所以可以将多个子应用，是由于其路由策略类似于MVVM框架的路由系统，注册路由的子应用可以在基座应用的指定路由容器组件中加载，其加载方式并不是通过iframe的方式，而是通过类似AJAX访问数据的形式，将子应用的整体代码加载到网页中，这样可以完美的规避iframe无法保存视图状态的问题。

当访问指定子应用所对应的路由时，基座应用会加载、运行对应的应用。而原有的一个或多个应用，仍然可以在页面上保持运行的状态。这些子应用的状态完全交给基座应用管理，子应用间可以通过基座应用进行通信，子应用可以使用不同的技术栈来进行实现，也可以单独部署到任何服务器。

3.3.4 微前端的微应用化

微应用化指的是微前端的另一种落地实现，该实现方式需要结合业务拆分，将子应用进行和里的分配，微应用化的子应用依然是独立开发和维护的，不同的是每个子应用即服务于主应用，子应用间的依赖可以在部署时统一由主应用管理，其架构组成，如图所示。



微应用化的的部署方案最优先适合统一化技术栈的子应用做综合协作，当然也适合跨技术栈的应用部署，可以将技术栈相同的子应用技术栈抽离交给基座进行统一管理，这种加载的子应用由于去除了通用核心依赖会导致无法单独运行，但对于主应用加载时可以大大提高子应用的加载速度。

3.3.5 微前端的微件化

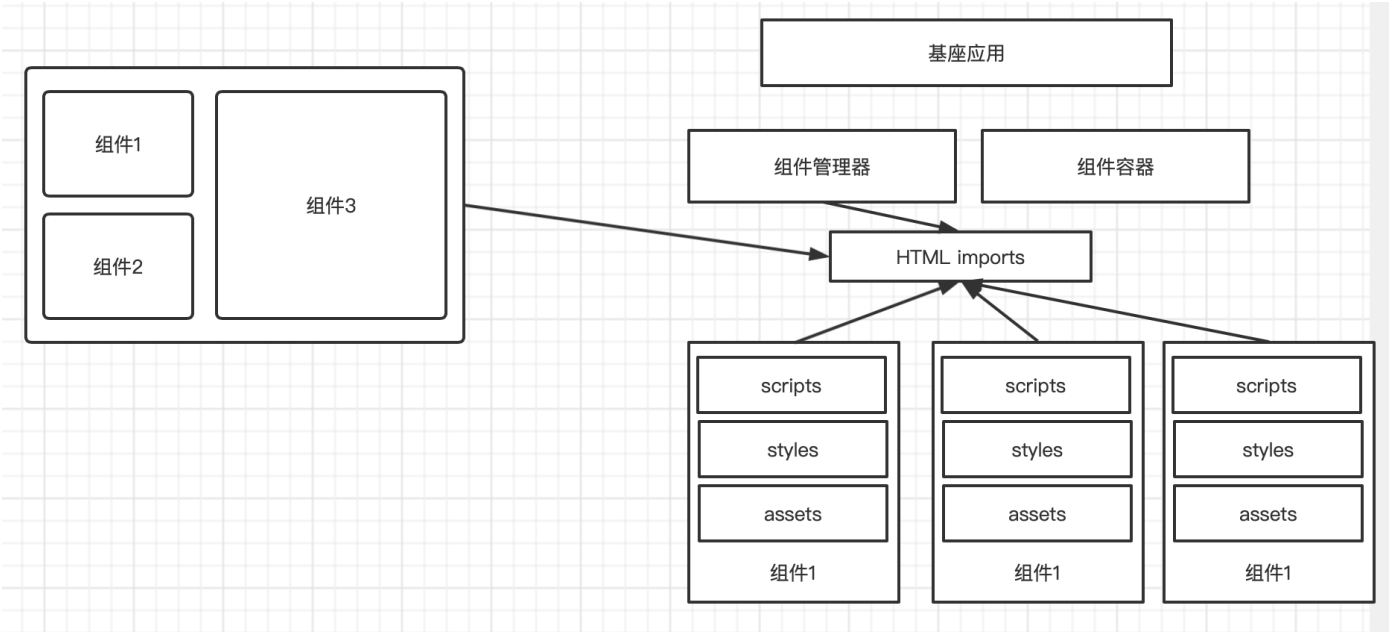
微件的概念很好理解，他与在HTML网页中引用第三方JavaScript依赖有这相似的思路。他是由开发者预先将功能代码打包构建成依赖，由主应用的HTML容器加载，这样任何子应用都可以直接按照组件加载的方式使用该应用。微件化在过去的MPA项目中很好实现，与加载Bootstrap、JQuery等框架的方式完全一样，对于SPA项目实现微件化就不是那么容易了。为了支持微件化，需要处理如下内容：

1. 持有一个完整的框架运行时和编译环境，这用来保证微件可以正常使用。
2. 由于微件需要提前加载编译好的文件，微件化可能会出现一些性能问题，必须等待微件系统加载完毕才能在应用中使用。
3. 在编排和构建上也会造成很多不方便的地方（比如这种模式更加一个框架开发团队和一个应用开发团队对接）。

WebComponents的介绍

结合微件系统最好的结果就是WebComponents。官方文档：https://developer.mozilla.org/zh-CN/docs/Web/Web_Components。WebComponents可以简单的理解为通过JavaScript定义一个可以在HTML网页环境中直接工作的新标签，该标签可以表示为任何内容，这个功能开放后，网页中的任何复杂并通用的部分都可以标签化，若通过WebComponents结合微件系统，可以在子应用中无缝使用构建好的WebComponents。目前情况来看WebComponents的兼容问题还没有得到改善，所以无法大量的应用到生产环境中，不过目前的Angular和Vue等框架都支持将应用构建成WebComponents，并在在其他任何支持WebComponents的框架中引用，这个方案无疑是未来微前端中微件场景的映射。

微件化实际工作案例，如图所示。



3.4 微前端架构模式探讨

从目前微前端的概念到落地实现来看，微前端架构主流的模式可以分为两种思想：

1. 基座模式

   基座模式就是上一节核心讨论的架构模式，通过一个主应用来管理其他的子应用，实现方式多种多样，设计难度小实现简单，可以满足各种主流的微前端需求，但自适应性差。目前主流的基座模式，并不像分布式服务端架构的注册中心一样，具备自动发现和注册应用的能力，需要在主应用中指定子应用的名称与URL路径。所以基座模式所实现的应用编排是建立在明确URL路径基础上实现的，并不能通过应用名称动态分析子应用，也不具备集群编排能力等等。

2. 自组织模式

   自组织模式讲求应用间是平等的，没有相互管理的说法，在分布式环境内可以自动发现并注册子应用。这种设计思路与分布式服务端架构的服务治理概念相同，但对于目前前端领域来说实现难度较大，架构维护成本太高。

3.5 浅析微前端架构带来的问题

微前端架构是一把技术架构的双刃剑，它在提供了更大能力的同时也将技术架构的复杂度推向了一个新的高度，并不是所有的应用场景和公司都适合使用微前端架构它可能会带来以下问题：

1. 技术拆分问题

   在公司中使用微前端架构最直面的问题就是技术拆分问题，这里就包括设计不足和设计过度问题。在定义微前端基座时，如何拆分子应用，如何管理公共模块，如何共享状态，如何做数据通信，很多问题都需要架构师做深思熟虑，在这个架构体系中并没有保准答案，只有针对公司现有的情况做合理的设计才是最优解，所以在做技术拆分时很容易由于思考不全面而造成为后续留坑，这也会为应用构建增加隐形成本。

2. 业务拆分问题

   回忆一下单一职责化理论，这个问题无论是前端还是后端都会面临。任何项目一旦分布式化，就会出现业务拆分的场景，拆分时一定要把业务解耦合，一旦出现拆分的两个业务在实际运作时仍然具备紧密的业务间联系，这个业务拆分就是失败的。此时再做改进的话相当于其中一个业务线都要重做，所以在拆分业务问题上一定要保证将项目构建成相互独立业务线，若业务间存在少量交集，也是通过业务总线进行其他业务调用，千万不要出现在不同的子应用中出现业务互相耦合的情况。

3. 团队管理问题

   虽然微前端的概念是将应用拆分相互独立开发，但基于微前端技术栈进行子应用开发时，对团队成员必须要做微前端概念的普及，否则在独立开发过程中会出现重复业务进入不同子应用，不同应用间的通信问题，代码质量问题，部署问题等等问题。

4. 其他问题

   除以上问题外，还会出现分布式部署的问题、持续集成问题、架构演进等一系列问题，以及技术债的填坑。所以微前端作为目前前端架构体系中的新架构体系，想要尝试需要投入试错成本。

3.5 微前端在目前的落地实现

目前常见的微前端架构有以下实际落地案例（排除传统的代理服务器和iframe）：

1. single-spa（文档地址：https://zh-hans.single-spa.js.org/）

Single-spa 是一个将多个单页面应用聚合为一个整体应用的 JavaScript 微前端框架。 使用 single-spa 进行前端架构设计可以带来很多好处，例如:

- 在同一页面上使用多个前端框架 而不用刷新页面 (React, AngularJS, Angular, Ember, 你正在使用的框架)
- 独立部署每一个单页面应用
- 新功能使用新框架，旧的单页应用不用重写可以共存
- 改善初始加载时间，延迟加载代码

2. QianKun.js（文档地址：https://qiankun.umijs.org/zh/）

   qiankun 是一个基于 single-spa 的微前端实现库，旨在帮助大家能更简单、无痛的构建一个生产可用微前端架构系统。

   qiankun 孵化自蚂蚁金融科技基于微前端架构的云产品统一接入平台，在经过一批线上应用的充分检验及打磨后，我们将其微前端内核抽取出来并开源，希望能同时帮助社区有类似需求的系统更方便的构建自己的微前端系统，同时也希望通过社区的帮助将 qiankun 打磨的更加成熟完善。

   目前 qiankun 已在蚂蚁内部服务了超过 200+ 线上应用，在易用性及完备性上，绝对是值得信赖的。

3. Mooa（文档地址：https://www.npmjs.com/package/mooa）

   1. 构建插件化的 Web 开发平台，满足业务快速变化及分布式多团队并行开发的需求
   2. 构建服务化的中间件，搭建高可用及高复用的前端微服务平台
   3. 支持前端的独立交付及部署

4 总结与展望

微前端架构可以说很新颖，也可以说由来已久，就目前的前端发展趋势来看，它的存在是有很大意义的，这个架构思想起到了承上启下的作用，它可以算Web2.0时代的尾声，面向Web3.0时代Web前端可能会往去中心化发展，从而实现真正意义的自组织模式的分布式微前端架构（作者的一些展望）。

所以在前端开发的领域上，若想要在技术层面上走的更高更远，第一步是思想的跃迁。首先要从一成不变的业务开发思想中跳脱出来，参与架构设计和架构演进的分析，对未来的技术发展风向有自己独特的见解和思考，培养设计思维而不是复制思维，一个好的架构师一定不是从A架构中复制出B架构的解决方案，而是从现有的技术架构及组成上分析出未来的风向以及发展趋势，根据设计思路提前做好技术储备或可以实现技术输出。最后希望本篇教程可以对所有学习路上的同学产生一些正面影响。