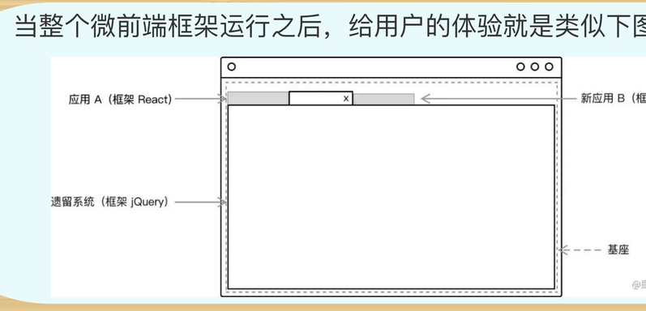
技术广度

PWA

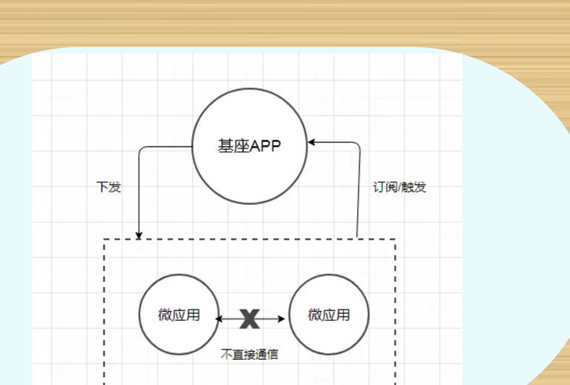
- PWA的中文名叫做渐进式网页应用
  - PWA的核心目标就是提升 Web App 的性能，改善 Web App 的用户体验。媲美native的流畅体验，将网络之长与应用之长相结合。
- 为什么W3C和谷歌在推广这项技术
  - Native APP 用起来很流畅，但是也有其天然的基因缺陷：
    - 由于其天生封闭的基因，内容无法被索引（无法SEO）
    - 用户 80% 的时间被 Top3 的超级 App 占据，对于站点来说，应用分发的性价比也越来越不划算
    - 要使用它，首先还需要下载几十兆上百着兆的安装包
  - WEB前端虽然天生具有开放的基因，但是很多时候页面会卡顿，用户体验不佳。
    - 虽然社区之前也做过很多努力，例如virtual dom、spa、混合编程、用canvas将整个页面画出来，用户体验也有了很大的改善，但是仍然无法解决几个重要的问题：
      - 离线时用户无法使用
      - 无法接收消息推送
      - 移动端没有一级入口
- 特点
  - 渐进式 - 适用于选用任何浏览器的所有用户，因为它是以渐进式增强作为核心宗旨来开发的。
  - 自适应 - 适合任何机型：桌面设备、移动设备、平板电脑或任何未来设备。
  - 连接无关性 - 能够借助于服务工作线程在离线或低质量网络状况下工作。
  - 类似应用 - 由于是在 App Shell 模型基础上开发，因此具有应用风格的交互和导航，给用户以应用般的熟悉感。
  - 持续更新 - 在服务工作线程更新进程的作用下时刻保持最新状态。
  - 安全 - 通过 HTTPS 提供，以防止窥探和确保内容不被篡改。
  - 可发现 - W3C 清单和服务工作线程注册作用域能够让搜索引擎找到它们，从而将其识别为"应用"。
  - 可再互动 - 通过推送通知之类的功能简化了再互动。
  - 可安装 - 用户可免去使用应用商店的麻烦，直接将对其最有用的应用"保留"在主屏幕上。
  - 可链接 - 可通过网址轻松分享，无需复杂的安装。
  - 跨平台
  - 离线工作
    - 借助服务工作线程（Service worker）实现
      - 可以把Service worker当做是一种客户端代理
  - 无需安装
    - 但可以像原生APP在主屏幕上留有图标。
  - 持续更新
- 目前的应用
  - 比如支付宝就有把某些应用添加到主屏幕的功能
- https://juejin.cn/post/6844903556470816781

微前端

- 微前端（Micro-Frontends）是一种类似于微服务的架构，它将微服务的理念应用于浏览器端，即将 Web 应用由单一的单体应用转变为多个小型前端应用聚合为一的应用
  - 各个前端应用还可以独立运行、独立开发、独立部署。微前端不是单纯的前端框架或者工具，而是一套架构体系
- https://juejin.cn/post/6844904162509979662
- 作用
  - 将这些庞大应用进行拆分，并随之解耦，每个部分可以单独进行维护和部署，提升效率。
  - 整合历史系统
- 方案
  - Nginx路由转发
    - 通过Nginx配置反向代理来实现不同路径映射到不同应用，例如www.abc.com/app1对应app1，www.abc.com/app2对应app2，这种方案本身并不属于前端层面的改造，更多的是运维的配置
    - 优点：简单，快速，易配置
    - 缺点：在切换应用时会触发浏览器刷新，影响体验
  - iframe嵌套
    - 父应用单独是一个页面，每个子应用嵌套一个iframe，父子通信可采用postMessage或者contentWindow方式
    - 优点：实现简单，子应用之间自带沙箱，天然隔离，互不影响
    - 缺点：iframe的样式显示、兼容性等都具有局限性；太过简单而显得low
  - Web Components
    - 每个子应用需要采用纯Web Components技术编写组件，是一套全新的开发模式
    - 优点：每个子应用拥有独立的script和css，也可单独部署
    - 缺点：对于历史系统改造成本高，子应用通信较为复杂易踩坑
  - 组合式应用路由分发（当前流行方案）
    - 每个子应用独立构建和部署，运行时由父应用来进行路由管理，应用加载，启动，卸载，以及通信机制
    - 优点：纯前端改造，体验良好，可无感知切换，子应用相互隔离
    - 缺点：需要设计和开发，由于父子应用处于同一页面运行，需要解决子应用的样式冲突，变量对象污染，通信机制等技术点
- 组合式应用路由分发方案
  - 该方案的核心是"主从"思想，即包括一个基座（MainApp）应用和若干个微（MicroApp）应用
    - 基座应用大多数是一个前端SPA项目，主要负责应用注册，路由映射，消息下发等
    - 微应用是独立前端项目，这些项目不限于采用React，Vue，Angular或者JQuery开发，每个微应用注册到基座应用中，由基座进行管理，但是如果脱离基座也是可以单独访问
  - 当整个微前端框架运行之后，给用户的体验就是类似下图所示：



    - 简单描述下就是基座应用中有一些菜单项，点击每个菜单项可以展示对应的微应用，这些应用的切换是纯前端无感知的，
  - 需要解决的问题
    - 路由切换的分发问题。
      - 基座要能拉取到微应用的页面内容，需要远程拉取机制
        - 远程拉取机制通常会采用fetch API来首先获取到微应用的HTML内容，然后通过解析将微应用的JavaScript和CSS进行抽离，采用eval方法来运行JavaScript，并将CSS和HTML内容append到基座应用中留给微应用的展示区域，当微应用切换走时，同步卸载这些内容，这就构成的当前应用的展示流程。
          - 当然这个流程里会涉及到CSS样式污染以及JavaScript对全局对象污染
        - 目前针对远程拉取机制这套流程，已有现成的库来实现，可以参考import-html-entry和system.js。
      - 路由切换
        - vue-router开发的基座SPA应用来举例，
          1. 当浏览器的路径变化后，vue-router会监听hashchange或者popstate事件，从而获取到路由切换的时机。
          - 最先接收到这个变化的是基座的router，通过查询注册信息可以获取到转发到那个微应用，经过一些逻辑处理后，采用修改hash方法或者pushState方法来将路由信息推送给微应用的路由，微应用可以是手动监听hashchange或者popstate事件接收，或者采用React-router，vue-router接管路由，后面的逻辑就由微应用自己控制。
    - 主微应用的隔离问题。
      - css隔离：可以采用webpack的postcss插件，在打包时添加特定的前缀。
      - JavaScript隔离：在window上挂载特定对象
    - 通信问题
      - 在基座应用中会定义事件中心Event，每个微应用分别来注册事件，当被触发事件时再由事件中心统一分发，这就构成了基本的通信机制



      - 如果基座和微应用采用的是React或者是Vue，是可以结合Redux和Vuex来一起使用，实现应用之间的通信。
- 目前的微前端框架
  - Qiankun：基于Single-Spa，阿里系开源微前端框架。
  - Icestark：阿里飞冰微前端框架，兼容多种前端技术栈
- 使用场景
  1. 微前端最佳的使用场景是一些B端的管理系统，既能兼容集成历史系统，也可以将新的系统集成进来，并且不影响原先的交互体验。

函数柯里化

- 柯里化是一种函数的转换，它是指将一个函数从可调用的 f(a, b, c) 转换为可调用的 f(a)(b)(c)。柯里化不会调用函数。它只是对函数进行转换。

```
function curry(f) { // curry(f) 执行柯里化转换
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}

// 用法
function sum(a, b) {
  return a + b;
}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3
```

- **柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。**
- 作用
  - 参数复用
  - **提前返回**：返回接受余下的参数且返回结果的新函数
  - **延迟执行**：返回新函数，等待执行
  - 当一个函数需要提前处理并需要等待执行或者接受多个不同作用的参数时候，我们便可应用柯里化。
- 应用场景
  - 防抖和节流
  - **兼容浏览器事件监听方法**

```
var addEvent = function(ele, type, fn, isCapture) {
    if(window.addEventListener) {

        ele.addEventListener(type, fn, isCapture)

    } else if(window.attachEvent) {

        ele.attachEvent("on" + type, fn)
    }
}
```

addEvent('div1',click,change,false)
addEvent('div2',click,change,false)
每调用一次 addEvent,就会判断一次 if/else

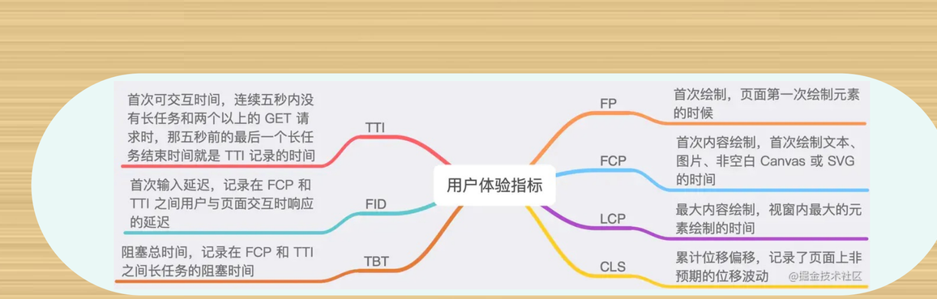
    - 柯里化后

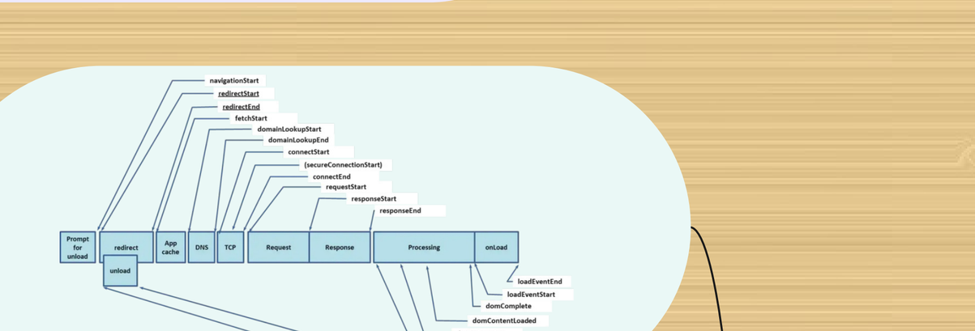
```
var addEvent = (function() {
    if(window.addEventListener) {
        return function(ele, type, fn, isCapture) {
A           ele.addEventListener(type, fn, isCapture)
        }
    } else if(window.attachEvent) {
        return function(ele, type, fn) {
B           ele.attachEvent("on" + type, fn)
        }
    }
})()
```

柯里化后：只判断一次 if/else
addEvent = A
或者
addEvent = B
addEvent('div',click,change,false)

前端性能监控

- window.preformance.timing





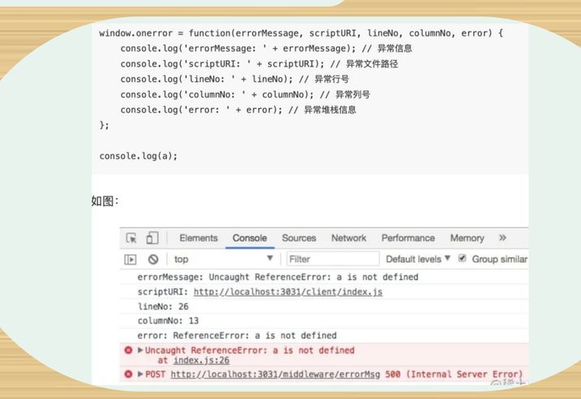
  - 由于window.preformance.timing是一个在不同阶段，被不停修正的一个参数对象，所以，建议在window.onload中进行性能数据读取和上报。
  - navigationStart，开始导航
    unloadEventStart，开始卸载上一个页面
    unloadEventEnd，完成卸载上一个页面
    redirectStart，开始重定向
    redirectEnd，结束重定向
    fetchStart，请求开始
    domainLookupStart，域名解析开始
    domainLookupEnd，域名解析结束
    connectStart，开始建立连接
    connectEnd，完成建立连接
    secureConnectionStart，开始建立安全连接
    requestStart，请求发出
    responseStart，接收到响应
    responseEnd，接收全部响应
    domLoading，当前网页DOM结构开始解析时，Document.readyState属性变为"loading"
    domInteractive，HTML加载和解析完成，DOM树构建完成（可交互），开始加载内嵌资源，Document.readyState属性变为"interactive"
    domContentLoadedEventStart，注册的（DOMContentLoaded）事件处理器开始执行，此时DOM树和CSSOM树构建完成（渲染树构建完成）
    domContentLoadedEventEnd，所有事件处理器执行完毕
    domComplete，当前文档解析完成并加载完所有的子资源，即Document.readyState 变为 'complete'
    loadEventStart，页面资源加载完毕，触发onload事件
    loadEventEnd，onload事件执行完毕
- 工具
  - Google PageSpeed Insights
    - 可分析网页内容并生成建议，还可以分别对移动端和PC端进行分析
    - https://pagespeed.web.dev/?utm_source=psi&utm_medium=redirect
  - LightHouse
    - Lighthouse 是一个开源的自动化工具，用于改进网络应用的质量。 您可以将其作为一个 Chrome 扩展程序运行，或从命令行运行。 您为 Lighthouse 提供一个您要审查的网址，它将针对此页面运行一连串的测试，然后生成一个有关页面性能的报告。
    - https://developers.google.com/web/tools/lighthouse

跨平台如何实现

- 跨平台框架
  - Flutter

sentry

- 不用sentry怎么捕获和上报异常?
  - 捕获异常的方式：
    - try/catch
    - **window.onerror**
      - 监控JavaScript运行时错误（包括语法错误）和 资源加载错误
      - 提供了全局监听异常的功能



      - 可以获取到报错的位置
      - 缺点：window.onerror这样的异常捕获不能捕获promise的异常错误
    - unhandledrejection
    - 框架自身的捕获机制（优先级高于window.onerror）
      - Vue.config.errorHandler

```
Vue.config.errorHandler = function (err, vm, info) {
    let {
        message, // 异常信息
        name, // 异常名称
        script,  // 异常脚本url
        line,  // 异常行号
        column,  // 异常列号
        stack  // 异常堆栈信息
    } = err;

    // vm为抛出异常的 Vue 实例
    // info为 Vue 特定的错误信息，比如错误所在的生命周期钩子
}
```

  - 上报机制
    - 设置node中间层
      - node安装'source-map'插件
    - 我们通过前端传过来的异常文件路径获取服务器端map文件地址，然后将压缩后的行列号传递给sourceMap返回的promise对象进行解析，通过originalPositionFor方法我们能获取到原始的报错行列号和文件地址，最后通过ajax将需要的异常信息统一传递给后台存储，完成异常上报。
  - https://juejin.cn/post/6844903582672650253
- sentry原理
  - 异常捕获
    - 全局捕获
      - 通过全局的接口，将捕获代码集中写在一个地方，可以利用的接口有
        - window.addEventListener('error') / window.addEventListener("unhandledrejection") / document.addEventListener('click')
      - 框架级别的全局监听
        - vue、react都有自己的错误采集接口
        - 例如aixos中使用interceptor进行拦截，
      - 对全局函数进行封装包裹，实现在在调用该函数时自动捕获异常
        - 比如重写setTimeout：

```
const prevSetTimeout = window.setTimeout;

window.setTimeout = function(callback, timeout) {
  const self = this;
  return prevSetTimeout(function() {
    try {
      callback.call(this);
    } catch (e) {
      // 捕获到详细的错误，在这里处理日志上报等了逻辑
      // ...
      throw e;
    }
  }, timeout);
}
```

      - 专门写一个函数来收集异常信息，在异常发生时，调用该函数
    - 单点捕获
  - 上报方式
    - 优先fetch,其次XHR，最后new Image
    - 使用new Image上报

```
var REPORT_URL = 'xxx'  //数据上报接口
var img = new Image; //创建img标签
img.onload = img.onerror = function(){  //img加载完成或加载src失败时的处理
    img = null; //img置空，不会循环触发onload/onerror
};
img.src = REPORT_URL + Build._format(params); //数据上报接口地址拼接上报参数作为img的src
// 或者
<img src="images/bg.png" onerror="javascript:this.src='images/bgError.png';">
```

      - 为什么不能用请求其他的文件资源（js/css/ttf）的方式进行上报?
        - 创建资源节点后只有将对象注入到浏览器DOM树后，浏览器才会实际发送资源请求。图片不仅不用插入DOM，而且只要在js中new出Image对象就能发起请求。
        - 载入js/css资源还会阻塞页面渲染，没有阻塞问题
        - 在没有js的浏览器环境中也能通过img标签正常打点

webwoker

- Web Worker 并不属于 JavaScript 语言的一部分，它仅仅是浏览器提供的一项特性，只是它可以被 JavaScript 访问、调用罢了
- 包含了 3 种类型的 Worker：
  - Dedicated Worker（专用 Worker）
    - 由主线程实例化且只能与它通信
    - 兼容性良好
  - Shared Worker（共享 Worker）
    - 可以被同一域（浏览器中不同的 tab、iframe 或其他 Shared Worker）下的所有线程访问。
    - 兼容不行
  - Service worker（服务 Worker）
- **原理**
  - 最终实现 Web Worker 的是一堆 .js 文件，网页会通过异步 HTTP 请求来加载它们。当然 Web Worker API 已经包办了这一切，上述加载对使用者完全无感。Worker 利用类似线程的消息机制保持了与主线程的平行
- 使用
  - Web Worker 是运行在浏览器内部的一条独立线程，因此需要使用 Web Worker 运行的代码块也必须存放在一个 **独立文件** 中
  - var worker = new Worker('task.js');
    - 如果此处的 "task.js" 存在且能被访问，那么浏览器会创建一个新的线程去异步地下载源代码文件。一旦下载完成，代码将立刻执行，此时 Worker 也就开始了它的工作。如果提供的代码文件不存在返回 404，那么 Worker 会静默失败并不抛出异常
  - worker.postMessage();
    - 为了启动创建好的 Worker，你需要显式地调用 postMessage 方法：

    - 有两种停止 Worker 的方法：1、在主页面中显示地调用 worker.terminate()；2、在脚本中调用 self.close() 让 Worker 自行了断。
- **局限性**
  - 无法访问一些非常重要的 JavaScript 特性：
    - DOM 元素（访问不是线程安全的）
    - window 对象
    - document 对象
    - parent 对象
  - 同源限制：主线程代码与Worker线程代码必须同源才能一起正常工作。
  - 任务顺序
    - Worker线程任务需要等待主线程任务结束才能进行。
- 应用场景：
  - 用于处理一些耗时、耗费性能的任务（异步的除外）。
  - 复杂的加密解密过程
  - **数据预获取**
    - 为优化你的网站或 web 应用的数据加载时长，你可以使用 Web Worker 预先获取一些数据，存储起来以备后续使用。Web Worker 在这里发挥着重要作用

虚拟列表

- 解决长列表加载和渲染问题
  - 虚拟列表其实是按需显示的一种实现，即只对 可见区域进行渲染，对 非可见区域中的数据不渲染或部分渲染的技术，从而达到极高的渲染性能。
- 实现
  - 虚拟列表的实现，实际上就是在首屏加载的时候，只加载 可视区域内需要的列表项，当滚动发生时，动态通过计算获得 可视区域内的列表项，并将 非可视区域内存在的列表项删除。
    - 假设有1万条记录需要同时渲染，我们屏幕的 可见区域的高度为 500px,而列表项的高度为 50px，则此时我们在屏幕中最多只能看到10个列表项，那么在首次渲染的时候，我们只需加载10条即可。
    - 假设滚动发生，滚动条距顶部的位置为 150px,则我们可得知在 可见区域内的列表项为 第4项至`第13项。

单点登录

- 是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统
  - 淘宝、天猫都属于阿里旗下，当用户登录淘宝后，再打开天猫，系统便自动帮用户登录了天猫，这种现象就属于单点登录
- 如何实现
  - 同域名共享cookie
    - 我们只需要将Cookie的 domain属性设置为父域的域名（主域名），同时将 Cookie 的path 属性设置为根路径，将 Session ID（或 Token）保存到父域中。这样所有的子域应用就都可以访问到这个Cookie
  - 不同域名
    - 后端控制
      - 部署一个认证中心，用于专门处理登录请求的独立的 Web 服务
      - 用户统一在认证中心进行登录，登录成功后，认证中心记录用户的登录状态，并将 token 写入 Cookie（注意这个 Cookie 是认证中心的，应用系统是访问不到的）
      - 应用系统检查当前请求有没有 Token，如果没有，说明用户在当前系统中尚未登录，那么就将页面跳转至认证中心的登录页。如果发现用户已经登录过了，就不会让用户再次登录了，而是会跳转回目标 URL ，并在跳转前生成一个 Token，拼接在目标 URL 的后面，回传给目标应用系统
      - 应用系统拿到 Token 之后，还需要向认证中心确认下 Token 的合法性，防止用户伪造。确认无误后，应用系统记录用户的登录状态，并将 Token 写入 Cookie，然后给本次访问放行。（注意这个 Cookie 是当前应用系统的）当用户再次访问当前应用系统时，就会自动带上这个 Token，应用系统验证 Token 发现用户已登录，于是就不会有认证中心什么事了
    - 前端控制
      - 后端需要做的仅仅是在用户登录成功后，将 Session ID （或 Token ）放在响应体中传递给前端
      - 前端将 Session ID （或 Token ）保存到浏览器的 LocalStorage 中，让前端在每次向后端发送请求时，主动将LocalStorage的数据传递给服务端
      - 前端通过 iframe+postMessage() 方式，将同一份 Token 写入到了多个域下的 LocalStorage 中，前端每次在向后端发送请求之前，都会主动从 LocalStorage 中读取Token并在请求中携带，这样就实现了同一份 Token 被多个域所共享

名词解释

- 域名收敛
  - 就是将静态资源放在一个域名下。减少DNS解析的开销。