浏览器的运行机制：
1.构建 DOM 树（parse）：渲染引擎解析 HTML 文档，首先将标签转换成 DOM 树中的 DOM node（包括 js 生成的标签）生成内容树（Content Tree/DOM Tree）；
2.构建渲染树（construct）：解析对应的 CSS 样式文件信息（包括 js 生成的样式和外部 css 文件），而这些文件信息以及 HTML 中可见的指令（如），构建渲染树（Rendering Tree/Frame Tree）；render tree 中每个 NODE 都有自己的 style，而且 render tree 不包含隐藏的节点(比如 display:none 的节点，还有 head 节点)，因为这些节点不会用于呈现；
3.布局渲染树（reflow/layout）：从根节点递归调用，计算每一个元素的大小、位置等，给出每个节点所应该在屏幕上出现的精确坐标；
4.绘制渲染树（paint/repaint）：遍历渲染树，使用 UI 层来绘制每个节点。

操作系统底层API

重绘（repaint 或 redraw）：是指一个元素外观的改变所触发的浏览器行为，浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。触发重绘的条件：改变元素外观属性。如：color，background-color 等。
注意：table 及其内部元素可能需要多次计算才能确定好其在渲染树中节点的属性值，比同等元素要多花两倍时间，这就是我们尽量避免使用 table 布局页面的原因之一。
重排（重构/回流/reflow）：当渲染树中的一部分(或全部)因为元素的规模尺寸、布局、隐藏等改变而需要重新构建，这就称为回流。每个页面至少需要一次回流，就是在页面第一次加载的时候。
重绘和重排的关系：在回流的时候，浏览器会使渲染树中受到影响的部分失效，并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕中，该过程称为重绘。所以，重排必定会引发重绘，但重绘不一定会引发重排。重绘重排的代价：耗时，导致浏览器卡慢。
触发重排的条件：任何页面布局和几何属性的改变都会触发重排，比如：
1、页面渲染初始化(无法避免)；　　2、添加或删除可见的 DOM 元素；　　3、元素位置的改变或使用动画；
4、元素尺寸的改变，如大小、外边距、边框；　　5、浏览器窗口尺寸的变化（resize 事件发生时）；
6、填充内容的改变，如文本的改变或图片大小改变而引起的计算值宽度和高度的改变；
7、读取某些元素属性：（offsetLeft/Top/Height/Width，clientTop/Left/Width/Height，scrollTop/Left/Width/Height，width/height，getComputedStyle()，currentStyle(IE)　）

有关重绘回流的性能优化：
1、浏览器自身的优化：浏览器会维护 1 个队列，把所有会引起回流、重绘的操作放入这个队列，等队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会 flush 队列，进行一个批处理。这样就会让多次的回流、重绘变成一次回流重绘。

重新处理更新

2、开发者要注意的优化：减少重绘和重排就是要减少对渲染树的操作，我们可以合并多次的 DOM 和样式的修改，并减少对 style 样式的请求。比如：
（1）直接改变元素的 className
（2）先设置元素为 display：none；然后进行页面布局等操作，完成后将元素设置为 display：block；这样的话就只引发两次重绘和重排；
（3）使用 cloneNode(true or false) 和 replaceChild 技术，引发一次回流和重绘；
（4）将需要多次重排的元素，position 属性设为 absolute 或 fixed，元素脱离文档流，其变化不会影响到其它元素；
（5）如果需要创建多个 DOM 节点，可以使用 DocumentFragment 创建完后一次性的加入 document；

网页验证码的作用：
1.验证码是目前大多网站所支持并使用于注册登录的，其能有效防止恶意登录注册，验证码每次都不同，这就可以排除用其他病毒或者软件自动申请用户及自动登陆，有效防止这种问题。 2.短信验证码等可以验证用户的合法性。
验证码和后端关系很大，基本对于前端来说，就是发送 ajax 就行。
验证码如：1.智能选图、文字点选、短信、滑动等，一般都是购买的服务；
2.图片文字验证码，这个后台可以做，比如 php java 等，当然也可以去购买。
例如：短信验证码 可以在聚合（https://www.juhe.cn/）购买后 前端只需要引入 js 文件 然后 按照文档 写上就行

async 函数返回一个 Promise 对象，可以使用 then 方法添加回调函数。当函数执行的时候，一旦遇到 await 就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

阮老师文档 http://es6.ruanyifeng.com/#docs/async

async +await 原理： generate+yield（的语法糖）。

async 函数就是 Generator 函数的语法糖。Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。Generator 函数有多种理解角度。语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。执行 Generator 函数会返回一个遍历器对象，也就是说，Generator 函数除了状态机，还是一个遍历器对象生成函数。返回的遍历器对象，可以依次遍历 Generator 函数内部的每一个状态。

形式上，Generator 函数是一个普通函数，但是有两个特征。一是，function 关键字与函数名之间有一个星号；二是，函数体内部使用 yield 表达式，定义不同的内部状态（yield 在英语里的意思就是“产出”）。

```
// Generator 方式的函数 里面的代码是 分段执行 看到 yield 就给你分一段
    function* helloWorldGenerator() {
      yield 'hello'; // yield 类似 暂停标记
      yield 'world';
      return 'ending';
    }
    var hw = helloWorldGenerator();//hw 返回一个
    // console.log(hw);// 这个函数的结果 不是 ending，因为代码是暂停的 是一个暂停标记 指向 hello
    console.log(hw.next()) ;//next 表示 拿出这个暂停的值
    console.log(hw.next())
    console.log(hw.next())
    console.log(hw.next())
   // 第一个 console.log(hw.next())     // { value: 'hello', done: false }
    // hw.next()     // { value: 'world', done: false }
    // hw.next()     // { value: 'ending', done: true }
    // hw.next()     // { value: undefined, done: true }
```

上面代码定义了一个 Generator 函数 helloWorldGenerator，它内部有两个 yield 表达式（hello 和 world），即该函数有三个状态：hello，world 和 return 语句（结束执行）。然后，Generator 函数的调用方法与普通函数一样，也是在函数名后面加上一对圆括号。不同的是，调用 Generator 函数后，该函数并不执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象，也就是遍历器对象（Iterator Object）。下一步，必须调用遍历器对象的 next 方法，使得指针移向下一个状态。也就是说，每次调用 next 方法，内部指针就从函数头部或上一次停下来的地方开始执行，直到遇到下一个 yield 表达式（或 return 语句）为止。

换言之，Generator 函数是分段执行的，yield 表达式是暂停执行的标记，而 next 方法可以恢复执行。由于 Generator 函数返回的遍历器对象，只有调用 next 方法才会遍历下一个内部状态，所以其实提供了一种可以暂停执行的函数。yield 表达式就是暂停标志。Iterator 遍历器对象的 next 方法的运行逻辑如下：Generator 函数可以返回一系列的值，因为可以有任意多个 yield。从另一个角度看，也可以说 Generator 生成了一系列的值，这也就是它的名称的来历（英语中，generator 这个词是“生成器”的意思）。

```
    const fs = require('fs');
    const readFile = function (fileName) {
      return new Promise(function (resolve, reject) {
        fs.readFile(fileName, function(error, data) {
          if (error) return reject(error);
          resolve(data);
        });
      });
    };
    const gen = function* () {
      const f1 = yield readFile('/etc/fstab');
      const f2 = yield readFile('/etc/shells');
```

```
        console.log(f1.toString());
        console.log(f2.toString());
    };
    改造
    const asyncReadFile = async function () {
    const f1 = await readFile('/etc/fstab');
    const f2 = await readFile('/etc/shells');
    console.log(f1.toString());
    console.log(f2.toString());
  };
// Primose.all([p1,p2,p3...],function)
// Primose.all 必须数组里面的所有的 promise 执行完毕 才成功 用在 要同时 有很多 结果一起成功的 情况
// Primose.race([p1,p2,p3...],function)
// Primose.race 只要 数组里面的 任何一个 成功 整个 race 就 执行了
```

TypeScript 是 JavaScript 的一个超集，支持 ECMAScript 6 标准。
TypeScript 由微软开发的自由和开源的编程语言。
TypeScript 设计目标是开发大型应用，它可以编译成纯 JavaScript，编译出来的 JavaScript 可以运行在任何浏览器上。
typescript 比 javascript 有更严格的类型要求，这样有类型的约束 就不会乱写 不同的值 大型项目中 bug 就少。
注：msg!: string; 确定 msg 非空 msg?: string; msg 可有可没有

```
import { Component, Prop, Vue } from 'vue-property-decorator';
import Add from '@/components/Add'
@Component({
  components:{Add}
})
export default class HelloWorld extends Vue {
  @Prop() private msg!: string;
}
```

ES6 的类 Class：

```
  class Person {
        constructor(){
            this.name='建林'
            this.age=18
        }
        say() {
            console.log('say 方法')
        }
  }
  class Teacher extends Person {
        constructor(){
            super();// 继承必须写 super 他就是父类 上面的那个 constructor

            this.name='思聪'// 覆盖上面那个
        }
        // 简单写法 name='思聪'
        eat(){
            console.log('eat')
```

```
        }
    }
    let t1=new Teacher()
    console.log(t1)
```

装饰器（Decorator）是一种与类（class）相关的语法，用来注释或修改类和类方法与属性。许多面向对象的语言都有这项功能。一般和类 class 相关 普通函数 进入代码就会执行完成
装饰器是一种函数，写成@ + 函数名。它可以放在类和类方法的定义前面。
1.修饰类 基本形式：

```
@testable
class MyTestableClass {
  // ...
}
function testable(target) {
  target.isTestable = true;
}
MyTestableClass.isTestable // true
//为它加上了静态属性 isTestable。testable 函数的参数 target 是 MyTestableClass 类本身。
```

2.修饰的 复杂形式 多套一个函数 返回一个（匿名）函数

```
//testable 是一个 Factory
function testable(isTestable) {
  return function(target) {
    target.isTestable = isTestable;
  }
}
@testable(true)
class MyTestableClass {}
MyTestableClass.isTestable // true
@testable(false)
class MyClass {}
MyClass.isTestable // false
```

3.修饰 类方法。修饰器：第一个参数是类的原型对象，是 Person.prototype，修饰器的本意是要“修饰”类的实例，但是这个时候实例还没生成，所以只能去修饰原型（这不同于类的修饰，那种情况时 target 参数指的是类本身）；第二个参数是所要修饰的属性名；第三个参数是该属性的描述对象。

```
function readonly(target, name, descriptor){
  // descriptor 对象原来的值如下
  // {
//    configurable:false,//能否使用 delete、能否需改属性特性、或能否修改访问器属性。false 为不可重新定义，默认值为 true
//    enumerable:false,//对象属性是否可通过 for-in 循环，flase 为不可循环，默认值为 true
//    writable:false,//对象属性是否可修改,flase 为不可修改，默认值为 true
//    value:'xiaoming' //对象属性的默认值
  // };
  descriptor.writable = false;
  return descriptor;
}
class Person {
  @readonly
  abc() { console.log('我是 person 的 abc 函数') }
```

```
}
```
4.多个装饰器一起，同一处的多个装饰器是按照洋葱模型，由外到内进入，再由内到外执行。
```
function dec(id){
    console.log('进入', id);
    return (target, property, descriptor) => {
        console.log('执行', id)
    };
}
class Example {
    @dec(1)
    @dec(2)
    xxx(){
        console.log('xxx')
    }
}
// 进入 1// 进入 2// 执行 2// 执行 1
```

Js 代码执行机制：所有同步任务都在主线程上的栈中执行。主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。一旦"栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，选择出需要首先执行的任务（由浏览器决定，并不按序）。

哪些是异步 1 定时器 2 ajax 3 事件处理(绑定的 onclick 等) 4 nodejs 读取文件也有异步。

宏任务：MacroTask（宏观 Task）：setTimeout, setInterval, , requestAnimationFrame, I/O。

微任务：MicroTask（微观任务）：process.nextTick, Promise（的 then）, Object.observe, MutationObserver。

先执行完同步任务，再取出第一个宏任务执行，所有的相关微任务总会在下一个宏任务之前全部执行完毕，如果遇见就先微后宏。

```
  console.log('1');
  setTimeout(function () {
    console.log('2');
    new Promise(function (resolve) {
      console.log('3');
      resolve();
    }).then(function () {
      console.log('4')
    })
  },0)
  new Promise(function (resolve) {
    console.log('5');
    resolve();
  }).then(function () {
    console.log('6')
  })
  setTimeout(function () {
    console.log('7');
    new Promise(function (resolve) {
      console.log('8');
      resolve();
    }).then(function () {
      console.log('9')
    })
```

```
    console.log('10')
  },0)
  console.log('11')
  // 1  5 11 6 2 3  4 7 8  10 9
```

flex 属性是 flex-grow, flex-shrink 和 flex-basis 的简写，默认值为 0 1 auto。后两个属性可选。
flex-grow ：flex-grow 属性定义盒子的放大比例，默认为 0 不放大 其他数字按比例放大
flex-shrink：如果所有项目的 flex-shrink 属性都为 1，当空间不足时，都将等比例缩小。如果一个项目的 flex-shrink 属性为 0，其他项目都为 1，则空间不足时，前者 0 的不缩小，其它缩小。
flex-basis 属性定义了在分配多余空间之前，项目占据的主轴空间（main size）提前写的宽高大小。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为 auto，即项目的本来大小。可以写 px。
如：flex:none|[<'flex-grow'><'flex-shrink'>?||<'flex-basis'>] 该属性有两个快捷值：auto (1 1 auto) 和 none (0 0 auto)。
建议优先使用这个属性，而不是单独写三个分离的属性，因为浏览器会推算相关值。

网站部署：一般是把自己 build（npm run build 等）的代码复制上传到 买的服务器的特定文件夹，如 www 里面。
不懂就问一下 上线没那么复杂 是大家想多了 去公司两分钟就会。
打包后的文件放到到服务器要注意：
如果是 linux 服务器 可以使用 ssh 或者 ftp 去上传 html css js 到服务器；
如果是 windows 服务器 可以使用 远程桌面连接 或者 ftp 上传到 服务器。
在公司里面一般只需要
1 把你的代码 git 提交给 老大 老大自己放到服务器
2 或者 把 build 的代码 发给 后台 后台给你把网址放到服务器
3 或者 后台会告诉你一个特定的文件夹 你只需要把代码放到服务器上的对应文件夹就行 比如我的叫 www

用 vue react 如果 history 模式下面 刷新网站 404 了 就需要服务器配置路由重写 指向 index.html 才可以 。
apache 服务器 要配置 .htaccess 文件 设置重写；nginx 也需要配置 服务器。

cookie：一般用来存储数据，如用户的登录状态（1 存的数据量小 2 默认浏览器关掉就过期了，但是可以自己设置过期时间 3 不太安全，因为每次请求头会带上 4 cookie 跨域有问题）。
现在经常用 token 和 localStorage（1 存的数据量大 2 不过期，除非删掉）。
cookie 机制：客户端浏览器会把 Cookie 保存起来，当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器。
HTTP 协议本身是无状态的，即服务器无法判断用户身份。Cookie 实际上是一小段的文本信息（key-value 格式）。客户端向服务器发起请求，如果服务器需要记录该用户状态，就使用 response 向客户端浏览器颁发一个 Cookie。客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器。服务器检查该 Cookie，以此来辨认用户等状态。
可以直接操作 cookie：设置 document.cookie='key=value'；获取 document.cookie，取出来是一个字符串形式，或许需要 split 拆分处理字符串 获取单独的 key

```
  // cookie
  //      // locastorage 1 存的数据量大 2 不过期 除非你删掉

  // cookie 是很久以前的技术 那时候用来存储用户登录 现在 locastorage 存 token 来操作
  // 注意：cookie 跨域有问题 现在都是 用 locastorage 存 token 统一

function Setcookie (name, value) {
    //设置名称为 name,值为 value 的 Cookie
    var expdate = new Date();     //初始化时间
    expdate.setTime(expdate.getTime() + 30 * 60 * 1000);     //时间
```

document.cookie = name+"="+value+";expires="+expdate.toGMTString()+";path=/";

//document.cookie= name+"="+value+";path=/"; 过期时间可以不要，但路径(path)必须要填写，因为 JS 的默认路径是当前页，如果不填，此 cookie 只在当前页面生效！~

网站优化？1 精灵图 2 懒加载 3 减少 http 请求.... 4 一定要答缓存 浏览器有缓存 h5 的 manifest 也可以存一下。

缓存：浏览器可能会把你上一次的代码存起来 你再次访问就没有去拿新代码 而是直接拿的缓存。

强缓存不发请求到服务器 直接拿缓存，协商缓存会发请求到服务器 服务器告诉你 去拿缓存 就拿 不拿缓存就拿新的代码。

浏览器缓存：分别有强制缓存和协商缓存 都可以通过后台设置响应头控制。强制缓存优先于协商缓存进行，若强制缓存(Expires 和 Cache-Control)生效则直接使用缓存，若不生效则进行后台设置头 协商缓存。

1.强缓存：不会向服务器发送请求，直接从缓存中读取资源 每次访问本地缓存直接验证看是否过期。强缓存可以通过设置两种 HTTP Header 实现：Expires 过期时间 和 Cache-Control 缓存控制。如：Cache-Control:max-age=300 缓存 300 秒。 服务器返回 服务器返回

2.协商缓存(Last-Modified / If-Modified-Since 和 Etag / If-None-Match)：协商缓存命中，服务器会将这个请求返回，且不会返回这个资源的数据 而是告诉客户端可以直接从缓存拿，于是浏览器就又会从自己的缓存中去加载这个资源。 当协商缓存也没有命中的时候，浏览器直接从服务器加载资源数据，此时代表该请求的之前的缓存失效，返回 200，重新返回资源和缓存标识，再存入浏览器缓存中，下次再请求的话若生效则返回 304，继续使用该缓存。

服务器缓存：服务器可以主动把需要缓存的数据或者页面内容存到 redis(类似数据库 比数据库快)中，后面可以从里面取，就不去数据库拿了。

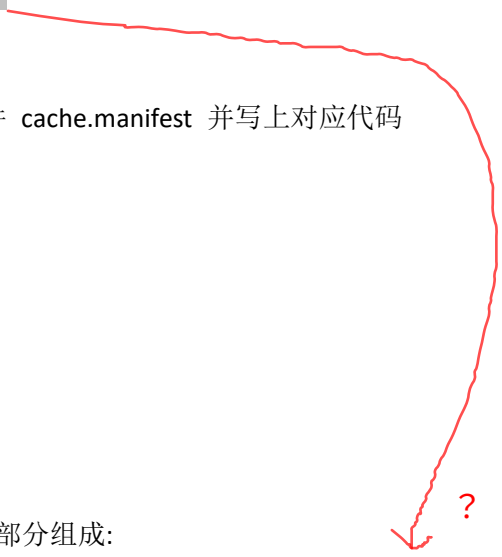
离线存储 manifest：html5 有个 manifest 也可以缓存 但是不常用。查看：在 application cache 里面可以看见。

使用方法有： 1 在需要离线缓存存储的页面 加上 manifest = "cache.manifest"

```
<!DOCTYPE HTML>
<html manifest = "cache.manifest">
...
</html>
```

2 在（服务器）根目录 新建文件 cache.manifest 并写上对应代码

```
CACHE MANIFEST
#v0.11
CACHE:
js/app.js
css/style.css
NETWORK:
resourse/logo.png
FALLBACK:
/ /offline.html
```



离线存储的 manifest 一般由三个部分组成:

CACHE:表示需要离线存储的资源列表，由于包含 manifest 文件的页面将被自动离线存储，所以不需要把页面自身也列出来。 会在当前浏览器存上 联网？

NETWORK:表示在它下面列出来的资源只有在在线的情况下才能访问，他们不会被离线存储，所以在离线情况下无法使用这些资源。不过，如果在 CACHE 和 NETWORK 中有一个相同的资源，那么这个资源还是会被离线存储，也就是说 CACHE 的优先级更高。

FALLBACK:表示如果访问第一个资源失败，那么就使用第二个资源来替换他，比如上面这个文件表示的就是如果访问根目录下任何一个资源失败了，那么就去访问 offline.html。

即时通讯 实现原理有两种

1 ajax 轮询 使用定时器 每隔 1s 时间 发送 ajax 到后台

2 websocket(常用) 有一个好用的封装的 socket.io 一般 在公司就是直接买一个就行 一般 只需要引入 js 就可以

如：https://www.7moor.com/successcasenew 七陌 客服 可以试用

https://www.easemob.com/product/cs/online 环信客服 可以试用

HTTPS中的S指SSL/TLS协议本身

Websocket 是应用层第七层上的一个应用层协议，WebSocket 的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话。

HTTP 有 1.1 和 1.0 之说，也就是所谓的 keep-alive ，把多个 HTTP 请求合并为一个，但是 Websocket 其实是一个新协议，跟 HTTP 协议基本没有关系，只是为了兼容现有浏览器，所以使用了 HTTP 。

客户端首先会向服务端发送一个 HTTP 请求，包含一个 Upgrade 请求头来告知服务端客户端想要建立一个 WebSocket 连接。

WebSocket 的其他特点：

1.建立在 TCP 协议之上，服务器端的实现比较容易。

2.与 HTTP 协议有着良好的兼容性。默认端口也是 80 和 443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。

3.数据格式比较轻量，性能开销小，通信高效。

4.可以发送文本，也可以发送二进制数据。

5.没有同源限制，客户端可以与任意服务器通信。

6.协议标识符是 ws（如果加密，则为 wss），服务器网址就是 URL。

```
//原生的 websocket
var websocket;
if (window.WebSocket) {
    websocket = new WebSocket('ws://127.0.0.1:3000');
    websocket.onopen = function (evevt) {
        console.log("WebSocket 链接上了");
    }
    websocket.onmessage = function (event) {
        // 在这里就 解析判断 后台返回什么数据 对应操作
        var msg = JSON.parse(event.data); //解析收到的 json 消息数据
    }
    websocket.onerror = function (event) {
        //发生错误 链接出错
    }
    websocket.onclose = function (event) {
        //连接关闭 关闭链接
    }

    function send() {
        try {
            websocket.send(JSON.stringify(message));
        } catch (ex) {
            console.log(ex);
        }
    }
}
else {
    alert('该浏览器不支持 web socket');
}
```

封装的 ws：

index.html：

```
<script src="/socket.io/socket.io.js"></script>
<script>
  // 前端 html js 代码
  // 1 引入 socket.io.js
  // 2 调用 io 链接  var socket = io() // 相当于 new websocket io 给我们封装了
  // 3 发消息到后台 socket.emit("自己定义的", 参数)
  // 4 接受消息 socket.on("自己定义的", function(msg) {})
  var name   = prompt("请输入你的昵称：");
  var socket = io()
  //发送昵称给后端，并更改网页 title
  socket.emit("join", name)
  document.title = name + "的群聊"
  socket.on("join", function (user) {
    addLine(user + " 加入了群聊")
  })
  //接收到服务器发来的 message 事件
  socket.on("message", function(msg) {
    addLine(msg)
  })
  //当发送按钮被点击时
  $('form').submit(function () {
    var msg = $("#m").val() //获取用户输入的信息
    socket.emit("message", msg) //将消息发送给服务器
    $("#m").val("") //置空消息框
    return false //阻止 form 提交
  })
  function addLine(msg) {
    $('#messages').append($('<li>').text(msg));
  }
</script>
```

server.js：

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);
var usocket = [];
// 后台 nodejs 代码 php java 都类似
app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});
// 后端链接
io.on('connection', function(socket){
  console.log('a user connected')
  // 后台监听 emit 过来
  socket.on("join", function (name) {
    usocket[name] = socket
    io.emit("join", name)
  })
  socket.on("message", function (msg) {
```

```
    io.emit("message", msg) //将新消息广播出去
  })

});
http.listen(3000, function() {
  console.log('listening on *:3000');
});
```

移动端的兼容（安卓和 iOS 手机）：

1.怎么判断是安卓还是 ios

```
//获取浏览器的 userAgent,并转化为小写
var ua = navigator.userAgent.toLowerCase();
//判断是否是苹果手机，是则是 true
var isIos = (ua.indexOf('iphone') != -1) || (ua.indexOf('ipad') != -1);
if(isIos){
    做苹果手机兼容
}else{
   做安卓
}
```

2.禁止图片点击放大：部分安卓手机点击图片会放大，如需要禁止放大，只需要设置 css 属性

```
img{
    pointer-events: none;
} //但是这个会让 img 标签的点击事件失效，如果想要给图片添加点击事件就要给上面再写一层
```

3.禁止 iOS 识别长串数字为电话

```
<meta name="format-detection" content="telephone=no">
```

4.禁止复制、选中文本：设置 CSS 属性 -webkit-user-select:none

5.一些情况下对非可点击元素如(label,span）监听点击事件，不会在 IOS 下触发，css 增加 cursor:pointer 就搞定了。

6.上下拉动滚动条时卡顿、慢：

```
body {
-webkit-overflow-scrolling: touch;
overflow-scrolling: touch;
} // Android3+和 iOS5+支持 CSS3 的新属性为 overflow-scrolling
```

7.安卓不会自动播放视频：安卓 autoplay 没效果 需要手动触发一下

```
window.addEventListener('touchstart', function(){
    audio.play(); // 需要主动调用一下 js 让视频播放
}, false);
```

8.半透明的遮罩层改为全透明，即解决在 ios 上，当点击一个链接或者通过 js 绑定了点击事件的元素时，会出现一个半透明的背景，当手指离开屏幕，该灰色背景消失，出现“闪屏”。

```
html, body {
    -webkit-tap-highlight-color: rgba(0,0,0,0);
}
```

混合开发：Hybrid App 主要以 JS+Native 两者相互调用为主，从开发层面实现“一次开发，多处运行”的机制，成为真正适合跨平台的开发。Hybrid App 兼具了 Native App 良好用户体验的优势，也兼具了 Web App 使用 HTML5 跨平台开发低成本的优势。目前已经有众多 Hybrid App 开发成功应用，比如美团、爱奇艺、微信等知名移动应用，都是采用 Hybrid App 开发模式。

移动应用开发的方式，目前主要有三种：Native App： 本地应用程序（原生 App）：

Web App：网页应用程序（移动 web）；Hybrid App：混合应用程序（混合 App）

特性	Native App	Hybrid App	Web App
开发语言	只用Native开发语言	Native和Web开发语言或只用Web开发语言	只用Web开发语言
代码移植性和优化	无	高	高
访问针对特定设备的特性	高	中	低
充分利用现有知识	低	高	高
高级图形	高	中	中
升级灵活性	低，总通过应用商店升级	中，部分更新可不通过应用商店升级	高
安装体验	高，从应用商店安装	高，从应用商店安装	中，通过移动浏览器安装

注意：

1 原生 ios 或安卓开发的 app 他基本可以操作 任何手机系统(视频 扫码 读取通讯录...)

2 混合开发：一部分安卓或 ios 一部分 html 如果要操作手机 就需要 安卓 ios 配合 前端 一起才行。前端也可以做混合开发 但是需要 借助框架或者 uniapp 编辑器等等帮你 打包嵌套 壳

3 普通手机移动端网页 对于手机操作 是比较困难 这些权限基本没有。

折中考虑——如果企业使用 Hybrid 开发方法，就能集 Native 和 web 两者之所长。一方面，Native 让开发者可以充分利用现代移动设备所提供的全部不同的特性和功能。另一方面，使用 Web 语言编写的所有代码都可以在不同的移动平台之间共享，使得开发和日常维护过程变得集中式、更简短、更经济高效。

混合开发框架和层次结构图（混合开发结构图）：

1）移动终端 web 壳（以下简称“壳”）：壳是使用操作系统的 API 来创建嵌入式 HTML 的渲染引擎。壳主要功能是定义 Android 应用程序与网页之间的接口，允许网页中的 JavaScript 调用 Android 应用程序，提供基于 web 的应用程序的 Android API，将 Web 嵌入到 Android 应用程序中。

2）前端交互 js：包括基础功能 js 和业务功能 js。

3）前端适配器：适配不同的终端：Pad、android、ios、wap。

vue 语法的 框架 weex（脚手架） 他可以打包生成 app。（不是 vue/cli、vite）

react 语法的框架 react-native（脚手架） 他也可以打包生成 app。（不是 react-app）

uni-app 这个框架 也可以打包生成 app。uniapp 的语法类似 vue。编辑器 hbuilderx 支持。写完然后用编辑器打包，直接就用它的语法开发。

app 现在做的很少 小公司做了也没人下载 成本也高 必要性不大 一般大公司才做 如果要做 基本现在使用 uniapp 就行。vue 项目 build 打包之后 把打包的项目 直接使用 编辑器 hbuilderx 打包成 app，其实就是 把你的写的 html 使用 编辑器 hbuilderx 给你套了一个 app 的壳，对于我们来说 必须使用编辑器 hbuilderx 才能打包的。

h5 与原生 app 的交互，本质上说，就是两种调用：app 调用 h5 的代码，h5 调用 app 的代码。

app 调用 h5 的代码：因为 app 是宿主，可以直接访问 h5，所以这种调用比较简单，就是在 h5 中曝露一些全局对象（包括方法），然后在原生 app 中调用这些 sdk（就是 类似封装的代码的意思） 。

h5 调用 app 的代码：因为 h5 不能直接访问宿主 app，所以这种调用就相对复杂一点。这种调用常用有两种方式：

1.由 app 向 h5 注入一个全局 js 对象，然后在 h5 直接访问这个对象，但这种方式可能存在安全隐患；

2.由 h5 发起一个自定义协议请求，app 拦截这个请求后，再由 app 调用 h5 中的回调函数：

app会进行相应处理，然后返回参数

由 app 自定义协议，如 sdk://action?params
在 h5 定义好回调函数，如 window.bridge={getDouble:value=>{},getTriple:value=>{}}
由 h5 发起一个自定义协议请求，如 location.href=‘sdk://double?value=10’
app 拦截这个请求后，进行相应的操作，获取返回值由 app 调用 h5 中的回调函数，比如 window.bridge.getDouble(20);。
uni-app 是一个使用 Vue.js 开发所有前端应用的框架，开发者编写一套代码，可发布到 iOS、Android、H5、以及各种小程序（微信/支付宝/百度/头条/QQ/钉钉）等多个平台。即使不跨端，uni-app 同时也是更好的小程序开发框架。
HBuilderX 是通用的前端开发工具，但为 uni-app 做了特别强化。下载 App 开发版，可开箱即用。
利用 HbuilderX 初始化项目：点击 HbuilderX 菜单栏文件>项目>新建；选择 uni-app,填写项目名称，项目创建的目录建议直接选 uniapp 里面的看图模板 然后直接按着他的写代码 官网做参考。
写完代码，点击编辑器 上面的 发行 就可以 但是先看看 manifest.json，这个是配置 app，安卓和 ios 的，发行 云打包 之后 可以在 发行 查看 打包状态，如果打包成功 就去打开它的下载地址 下载 apk 文件就行 安卓的安装包叫 apk。如果要到商店去 就打包的时候发行 这个需要公司的资质的。
如果是 vue 项目 也可以 打包成 app，用 hbuilder x 编辑器 ：
1 开发 vue 然后 npm run build 生成 dist 文件夹 里面有 html css js
2 我们先用 hbuilderx 新建 项目 选择 5+app 创建一个空模板
3 把我们 build 的 html css js 复制到 创建的空模板，即把对应的 js css 图片 放到文件夹 或者直接复制粘贴就行
4 打包 --先打开 manife.json 配置 然后 在发行 云打包。

Vuex：
根目录》src》store》index.js：

```
import Vue from 'vue'
import Vuex from 'vuex'
Vue.use(Vuex)
export default new Vuex.Store({
  state: {
    count:1//count 相当于 data 里面的数据 但是这是全局的 就是 全部组件都可以拿到了 用$store
  },
  mutations: {
    increment (state,num) {
      console.log('mutation 触发')
      // 变更状态 // state.count++
      state.count=state.count+num
    }
  },
  actions: {
    actionAdd ({commit},num) {
      //这里可以异步 发送 ajax 拿到数据 再触发 mutation 修改
       // context 是个对象 由里面的 commit 用来触发 mutation 必须 在 mutation 里面才可以修改 state
      // dispatch 触发 action ---> commit 触发 mutation --》在 mutation 里面才可以修改 state
       console.log('action 的函数 actionAdd')
      //  console.log('context',context)
      //  context.commit(mutation 函数名,参数..)
      commit('increment',num)
    }
  },
  modules: {
  }
})
```

在main文件里：

```
import store from './store'
new Vue({
    el: '#app',
    store,
    render: h => h(App)
})
```

根目录》src》views》Home.vue：

```
methods:{
    add(){
      //  修改全局数据  正常情况：
      // 1  先 dispatch action 函数
      // 2  在 action 里面  commit 触发 mutation
      // 3  在 mutation 里面  才能修改全局数据
      // this.$store.dispatch(action 函数名,参数..)
      // this.$store.dispatch('actionAdd',10)
        //注意：  1  如果有异步  需要在 action 写异步操作
        // 2 mutation 必须写同步操作  一般只用来直接修改 state
        // 3  如果没有涉及到异步操作  其实你可以跳过 action  直接触发 mutation
        // 4  但是不能直接修改 state  必须最少要由 mutation 修改
       this.$store.commit("increment",10)
    }
  }
```