开课吧中高级前端面试题

ES6语法

1. 原型如何实现继承？Class 如何实现继承？Class 本质是什么？

组合继承

```
function Parent(value) {
  this.val = value
}
```

```
Parent.prototype.getValue = function() {
  console.log(this.val)
}
function Child(value) {
  Parent.call(this, value)
}
Child.prototype = new Parent()

const child = new Child(1)

child.getValue() // 1
child instanceof Parent // true
```

class继承

```
class Parent {
  constructor(value) {
    this.val = value
  }
  getValue() {
    console.log(this.val)
  }
}
class Child extends Parent {
  constructor(value) {
    super(value)
    this.val = value
  }
}
let child = new Child(1)
child.getValue() // 1
child instanceof Parent // true
```

JS异步

并发（concurrency）和并行（parallelism）区别

异步和这小节的知识点其实并不是一个概念，但是这两个名词确实是很多人都常会混淆的知识点。其实混淆的原因可能只是两个名词在中文上的相似，在英文上来说完全是不同的单词。

并发是宏观概念，我分别有任务 A 和任务 B，在一段时间内通过任务间的切换完成了这两个任务，这种情况就可以称之为并发。

并行是微观概念，假设 CPU 中存在两个核心，那么我就可以同时完成任务 A、B。同时完成多个任务的情况就可以称之为并行。

你理解的 Generator 是什么?

```
function *foo(x) {
  let y = 2 * (yield (x + 1))
  let z = yield (y / 3)
  return (x + y + z)
}
let it = foo(5)
console.log(it.next())   // => {value: 6, done: false}
console.log(it.next(12)) // => {value: 8, done: false}
console.log(it.next(13)) // => {value: 42, done: true}
```

首先 Generator 函数调用和普通函数不同，它会返回一个迭代器
当执行第一次 next 时，传参会被忽略，并且函数暂停在 yield (x + 1) 处，所以返回 5 + 1 = 6
当执行第二次 next 时，传入的参数等于上一个 yield 的返回值，如果你不传参，yield 永远返回 undefined。此时 let y = 2 * 12，所以第二个 yield 等于 2 * 12 / 3 = 8
当执行第三次 next 时，传入的参数会传递给 z，所以 z = 13, x = 5, y = 24，相加等于 42

手写Promise

在完成符合 Promise/A+ 规范的代码之前，我们可以先来实现一个简易版 Promise，因为在面试中，如果你能实现出一个简易版的 Promise 基本可以过关了。

```
const PENDING = 'pending'
const RESOLVED = 'resolved'
const REJECTED = 'rejected'

function MyPromise(fn) {
  const that = this
  that.state = PENDING
  that.value = null
  that.resolvedCallbacks = []
  that.rejectedCallbacks = []
  // 待完善 resolve 和 reject 函数
  // 待完善执行 fn 函数
}
```

1. 首先我们创建了三个常量用于表示状态，对于经常使用的一些值都应该通过常量来管理，便于开发及后期维护
2. 在函数体内部首先创建了常量 that，因为代码可能会异步执行，用于获取正确的 this 对象
3. 一开始 Promise 的状态应该是 pending
4. value 变量用于保存 resolve 或者 reject 中传入的值
5. resolvedCallbacks 和 rejectedCallbacks 用于保存 then 中的回调，因为当执行完 Promise 时状态可能还是等待中，这时候应该把 then 中的回调保存起来用于状态改变时使用
6. 接下来我们来完善 resolve 和 reject 函数，添加在 MyPromise 函数体内部

```
function resolve(value) {
  if (that.state === PENDING) {
    that.state = RESOLVED
```

```
    that.value = value
    that.resolvedCallbacks.map(cb => cb(that.value))
  }
}

function reject(value) {
  if (that.state === PENDING) {
    that.state = REJECTED
    that.value = value
    that.rejectedCallbacks.map(cb => cb(that.value))
  }
}
```

再实现then

```
MyPromise.prototype.then = function(onFulfilled, onRejected) {
  const that = this
  onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v => v
  onRejected =
    typeof onRejected === 'function'
      ? onRejected
      : r => {
          throw r
        }
  if (that.state === PENDING) {
    that.resolvedCallbacks.push(onFulfilled)
    that.rejectedCallbacks.push(onRejected)
  }
  if (that.state === RESOLVED) {
    onFulfilled(that.value)
  }
  if (that.state === REJECTED) {
    onRejected(that.value)
  }
}
```

参考代码

```
// 1. 术语
// promise是一个包含了兼容promise规范then方法的对象或函数,
// thenable 是一个包含了then方法的对象或函数。
// value 是任何Javascript值。 (包括 undefined, thenable, promise等).
// exception 是由throw表达式抛出来的值。
// reason 是一个用于描述Promise被拒绝原因的值。

// 要求
// 2 Promise状态
// 一个Promise必须处在其中之一的状态: pending, fulfilled 或 rejected.
```

```
// 如果是pending状态,则promise:

// 可以转换到fulfilled或rejected状态。
// 如果是fulfilled状态,则promise:

// 不能转换成任何其它状态。
// 必须有一个值，且这个值不能被改变。
// 如果是rejected状态,则promise可以:

// 不能转换成任何其它状态。
// 必须有一个原因，且这个值不能被改变。
// ”值不能被改变”指的是其identity不能被改变，而不是指其成员内容不能被改变。
// x 是then中的回调函数的返回值。可能的值为Promise，value， 和undefined。
function resolvePromise(promise, x, resolve, reject) {
  let then,thenCalledOrThrow = false
  //如果promise 和 x 指向相同的值，使用 TypeError做为原因将promise拒绝。
  if (promise === x) {
    return reject(new TypeError('Chaining cycle detected for promise!'))
  }

  //判断x是否是一个Promise，如果是，那么就直接把MyPromise中的resolve和reject传给then;
  //返回值是一个Promise对象，直接取它的结果做为promise2的结果
  if ((x !== null) && ((typeof x === 'object') || (typeof x === 'function'))) {
    try {
      then = x.then
      if (typeof then === 'function') { // typeof

        //x.then(resolve, reject);
        then.call(x, function rs(y) {

          if (thenCalledOrThrow) return

          thenCalledOrThrow = true

          return resolvePromise(promise, y, resolve, reject)

        }, function rj(r) {

          if (thenCalledOrThrow) return

          thenCalledOrThrow = true

          return reject(r)

        })
      } else {

        return resolve(x)
      }
    } catch(e) {
      if (thenCalledOrThrow) return

      thenCalledOrThrow = true
```

```
      return reject(e)
    }
  } else {

    return resolve(x)
  }

}

function MyPromise(callback) {

  let that = this;
  //定义初始状态
  //Promise状态
  that.status = 'pending';
  //value
  that.value = '1';
  //reason 是一个用于描述Promise被拒绝原因的值。
  that.reason = 'undefined';
  //用来解决异步问题的数组
  that.onFullfilledArray = [];
  that.onRejectedArray = [];

  //定义resolve
  function resolve(value) {
    //当status为pending时，定义Javascript值，定义其状态为fulfilled
    if(that.status === 'pending') {
      that.value = value;
      that.status = 'fulfilled';
      that.onFullfilledArray.forEach((func) => {
        func(that.value);
      });
    }
  }

  //定义reject
  function reject(reason) {
    //当status为pending时，定义reason值，定义其状态为rejected
    if(that.status === 'pending') {
      that.reason = reason;
      that.status = 'rejected';
      that.onRejectedArray.forEach((func) => {
        func(that.reason);
      });
    }
  }

  //捕获callback是否报错
  try {
    callback(resolve, reject);
  } catch (error) {

    reject(error);
```

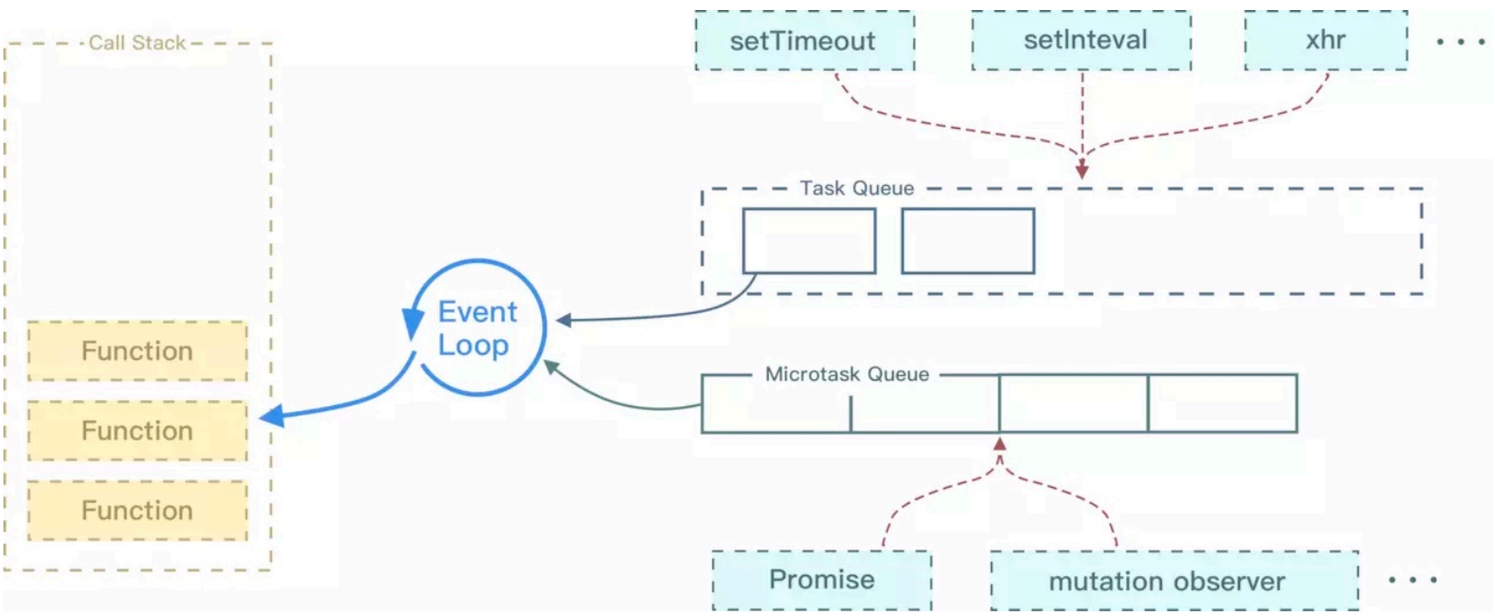
```
  }
}

// 定义then
// 一个promise必须有一个then方法，then方法接受两个参数：
// promise.then(onFulfilled,onRejected)
//观察者模式解决异步调用问题

// 如果onFulfilled是一个函数：
// 它必须在promise fulfilled后调用， 且promise的value为其第一个参数。
// 它不能在promise fulfilled前调用。
// 不能被多次调用。

// 如果onRejected是一个函数，
// 它必须在promise rejected后调用， 且promise的reason为其第一个参数。
// 它不能在promise rejected前调用。
// 不能被多次调用。
MyPromise.prototype.then = function(onFulfilled, onRejected) {
  let that = this;
  let promise2;

  // 根据标准，如果then的参数不是function，则我们需要忽略它
  onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : function(f) { ret
urn f};
  onRejected = typeof onRejected === 'function' ? onRejected : function(r) {throw r
};
  //需要修改下，解决异步问题，即当Promise调用resolve之后再调用then执行onFulfilled(that.valu
e)。
  //用两个数组保存下onFulfilledArray
  if(that.status === 'pending') {

    return promise2 = new Promise(function(resolve, reject) {
      that.onFullfilledArray.push((value) => {
        try {
          let x = onFulfilled(value);
          resolvePromise(promise2, x, resolve, reject)
        } catch(e) {
          return reject(e)
        }
      });

      // that.onFullfilledArray.push((reason) => {
      //   onFulfilled(reason);
      // });

      // that.onRejectedArray.push((reason) => {
      //   onRejected(reason);
      // });

      that.onRejectedArray.push((value) => {
        try {
          let x = onRejected(value);
          resolvePromise(promise2, x, resolve, reject)
        } catch(e) {
```

```
          return reject(e)
        }
      });

    })


  }

  if(that.status === 'fulfilled') {
    return promise2 = new MyPromise(function(resolve, reject) {
      try {
        let x = onFulfilled(that.value);
        //处理then的多种情况
        resolvePromise(promise2, x, resolve, reject)
      } catch (error) {
        reject(error);
      }

      // try {
      //   let x = onFulfilled(that.value);
      //   //判断onFulfilled是否是一个Promise，如果是，那么就直接把MyPromise中的resolve和reject传给then;
      //   //返回值是一个Promise对象，直接取它的结果做为promise2的结果
      //   if(x instanceof MyPromise) {
      //     x.then(resolve, reject);
      //   }
      //   //否则，以它的返回值做为promise2的结果
      //   resolve(x);
      // } catch (error) {
      //   reject(error);
      // }

    })
  }

  if(that.status === 'rejected') {
    return new MyPromise(function(resolve, reject) {
      try {
        let x = onRejected(that.value);
        //处理then的多种情况
        resolvePromise(promise2, x, resolve, reject);
      } catch (error) {
        reject(error)
      }

    })

  }
}

module.exports = MyPromise;

// 如果没有异步，此时status状态肯定为三种状态之一，一般为resolved，反之，为pending。
```

```
// 测试
// new MyPromise((resolve, reject) => {
//   setTimeout(() => {
//     resolve(1);
//   }, 1000)
// }).then((data) => {
//   console.log(data);
//   return new MyPromise((res) => {
//     setTimeout(() => {
//       res(2);
//     },1000)
//   })
// }).then((res) => {
//   console.log(res);
// })

new MyPromise(resolve=>resolve(8))
  .then()
  .then()
  .then(function foo(value) {
    console.log(value)
  })
```

Event Loop

执行栈



JS进阶

手写call apply和bind

```
Function.prototype.myCall = function(context) {
  if (typeof this !== 'function') {
```

```
    throw new TypeError('Error')
  }
  context = context || window
  context.fn = this
  const args = [...arguments].slice(1)
  const result = context.fn(...args)
  delete context.fn
  return result
}
```

apply

```
Function.prototype.myApply = function(context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  context = context || window
  context.fn = this
  let result
  // 处理参数和 call 有区别
  if (arguments[1]) {
    result = context.fn(...arguments[1])
  } else {
    result = context.fn()
  }
  delete context.fn
  return result
}
```

bind

```
Function.prototype.myBind = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  const _this = this
  const args = [...arguments].slice(1)
  // 返回一个函数
  return function F() {
    // 因为返回了一个函数，我们可以 new F()，所以需要判断
    if (this instanceof F) {
      return new _this(...args, ...arguments)
    }
    return _this.apply(context, args.concat(...arguments))
  }
}
```

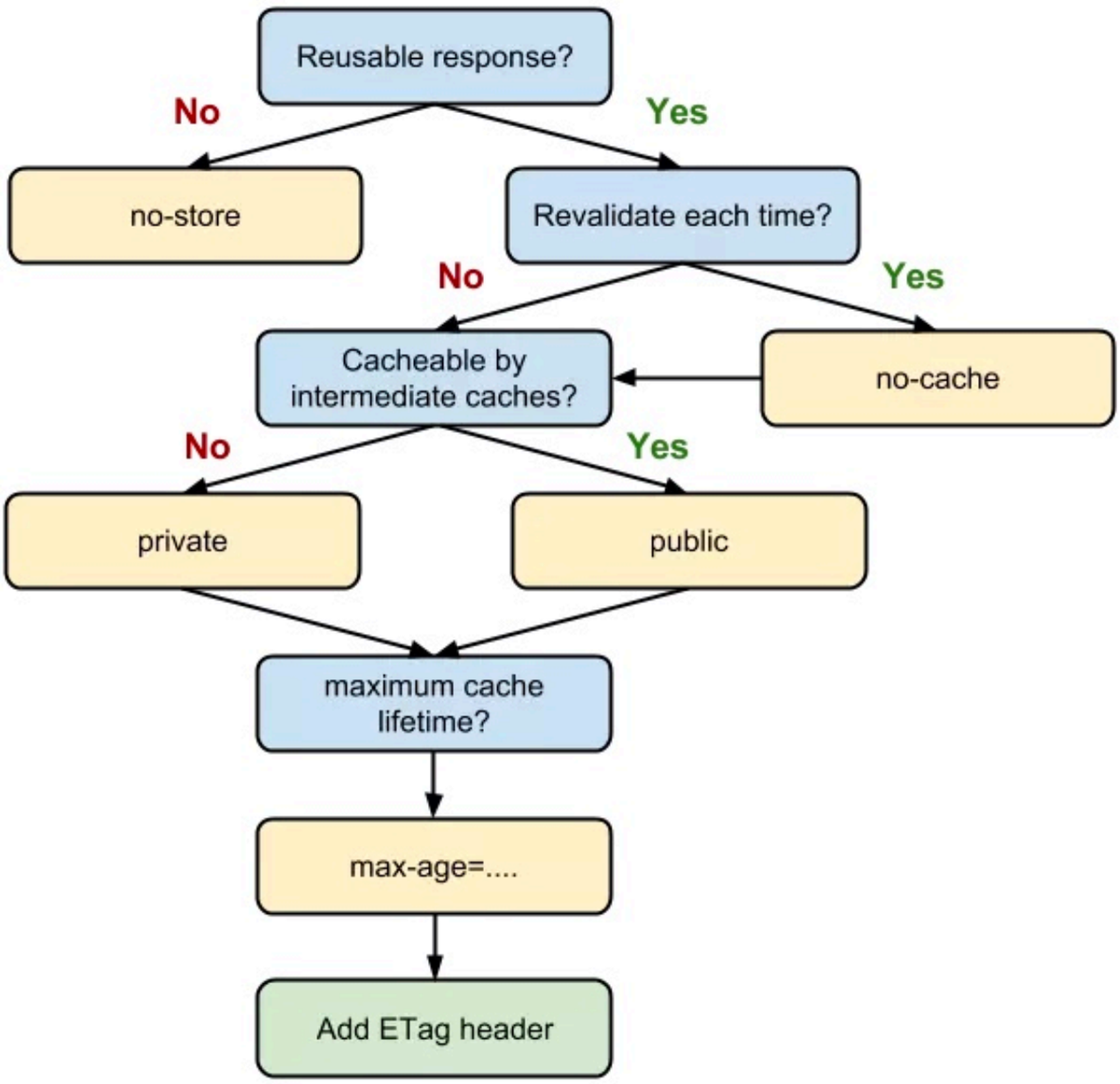
浏览器缓存机制

从缓存位置上来说分为四种，并且各自有优先级，当依次查找缓存且都没有命中的时候，才会去请求网络

Service Worker
Memory Cache
Disk Cache
Push Cache
网络请求

1. 强缓存

强缓存可以通过设置两种 HTTP Header 实现：Expires 和 Cache-Control

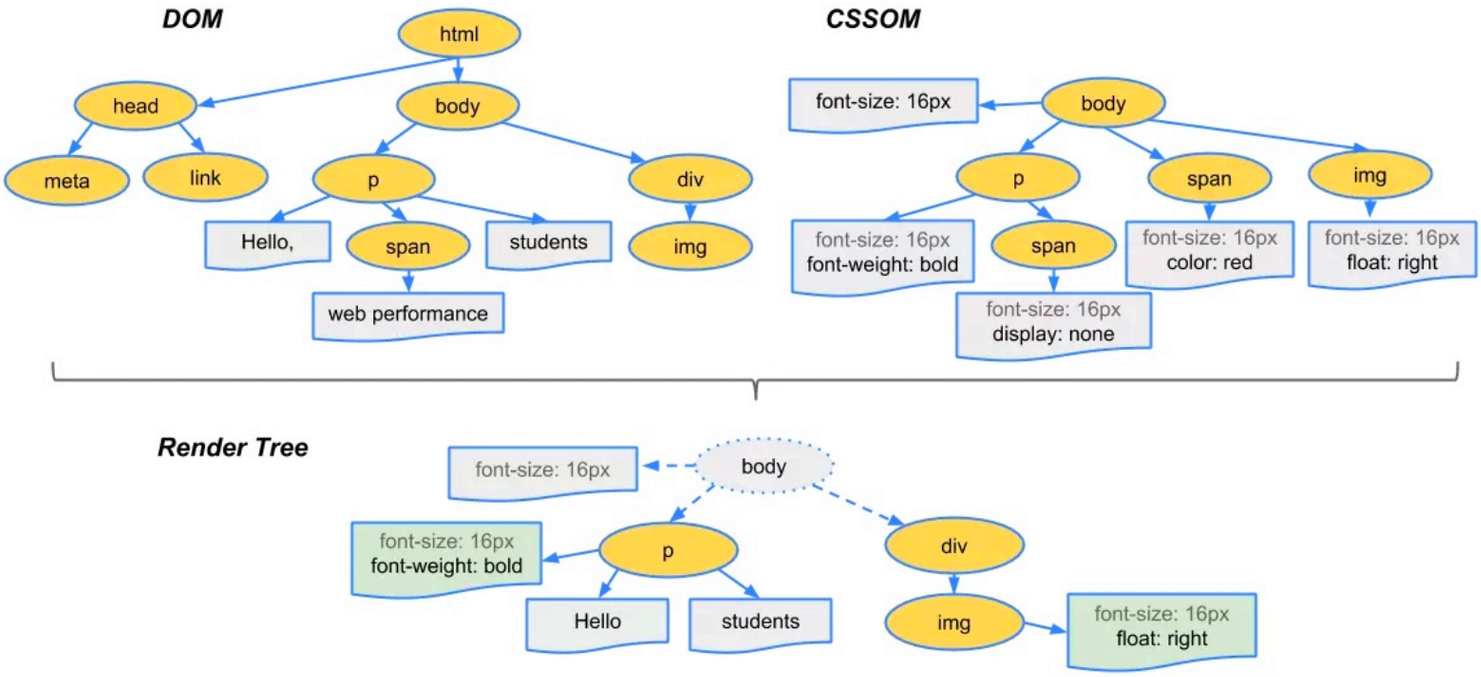


1. 如果缓存过期了，就需要发起请求验证资源是否有更新。协商缓存可以通过设置两种 HTTP Header 实现：Last-Modified 和 ETag 。

浏览器渲染原理

1. 浏览器接收到 HTML 文件并转换为 DOM 树

2. 将 CSS 文件转换为 CSSOM 树
3. 生成渲染树



前端常见安全问题

1. XSS
2. CSRF
3. 点击劫持
4. 中间人攻击

性能优化

1. 图片优化
2. 懒加载
3. DNS预解析
4. 函数节流
5. 函数防抖
6. 预渲染
7. webpack性能优化
    1. 减少webpack打包事件
    2. loader优化文件搜索
    3. DllPlugin
    4. 按需加载

Vue

组件通信

1. 父子组件通信
    1. 通过props

2. 兄弟组件通信
    1. 通过父元素中转
3. 跨多层级组件通信
    1. provide & inject
    2. eventBus模式
4. 任意
    1. vuex

响应式原理

Vue 内部使用了 Object.defineProperty() 来实现数据响应式，通过这个函 数可以监听到 set 和 get 的事件。

```
    function observe(obj) {
  // 判断类型
  if (!obj || typeof obj !== 'object') {
    return
  }
  Object.keys(obj).forEach(key => {
    defineReactive(obj, key, obj[key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    // 可枚举
    enumerable: true,
    // 可配置
    configurable: true,
    // 自定义函数
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}
```

React

组件通信

1. 父子
    1. props

2. 多层嵌套
    1. context
3. 任意
    1. redux
    2. mobx
    3.

前端监控

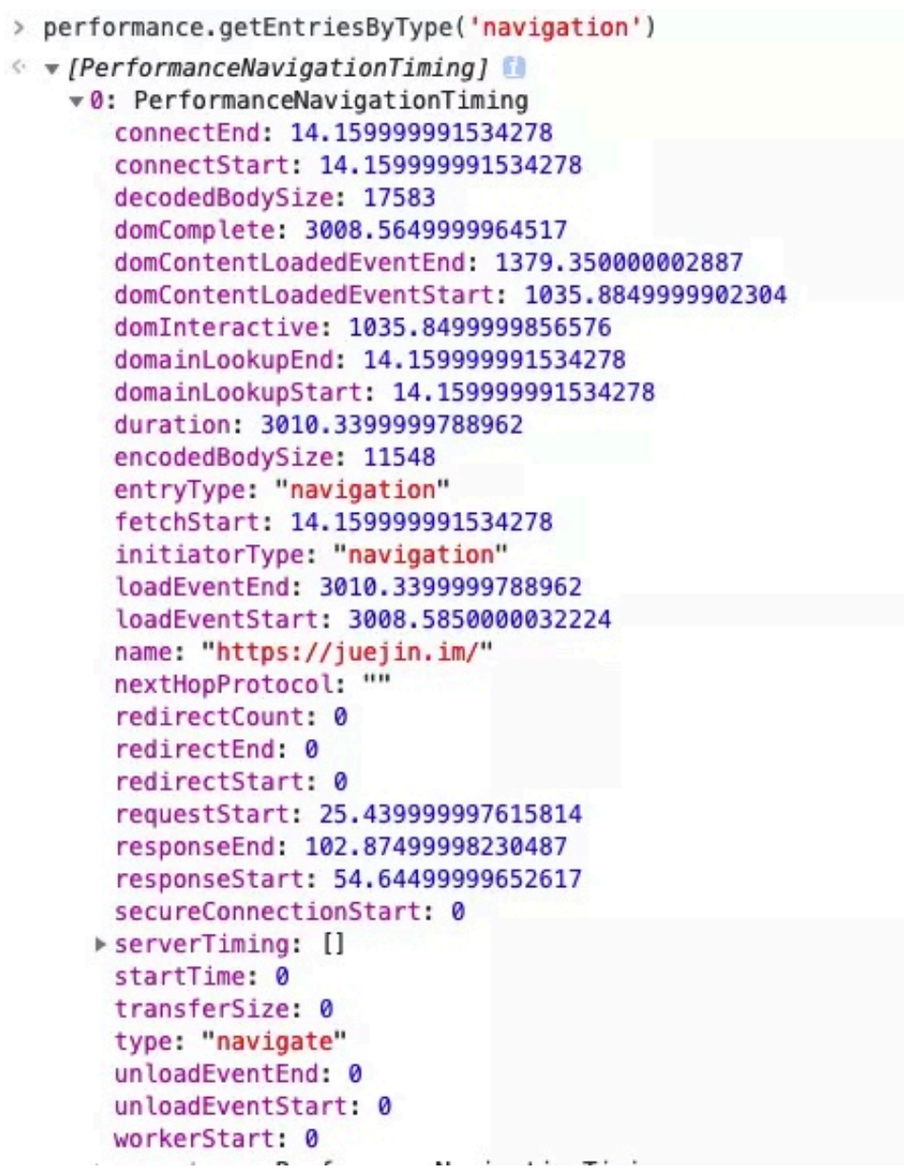
页面埋点

页面埋点应该是大家最常写的监控了，一般起码会监控以下几个数据：

PV / UV
停留时长
流量来源
用户交互

性能监控

我们可以直接使用浏览器自带的 Performance API 来实现这个功能。



异常监控

对于代码运行错误，通常的办法是使用 window.onerror 拦截报错。该方法能拦截到大部分的详细报错信息，但是也有例外

对于跨域的代码运行错误会显示 Script error. 对于这种情况我们需要给 script 标签添加 crossorigin 属性
对于某些浏览器可能不会显示调用栈信息，这种情况可以通过 arguments.callee.caller 来做栈递归
对于异步代码来说，可以使用 catch 的方式捕获错误。比如 Promise 可以直接使用 catch 函数，async await 可以使用 try catch。

对于捕获的错误需要上传给服务器，通常可以通过 img 标签的 src 发起一个请求。

设计模式

工厂模式

```
class Man {
  constructor(name) {
    this.name = name
  }
  alertName() {
    alert(this.name)
  }
}

class Factory {
  static create(name) {
    return new Man(name)
  }
}

Factory.create('kaikeba').alertName()
```

单例模式

其实全局状态管理就是单例模式，全局唯一

装饰器模式

@就是装饰器，对函数进行增强

```
function readonly(target, key, descriptor) {
  descriptor.writable = false
  return descriptor
}
```

```
class Test {
  @readonly
  name = 'kkb'
}

let t = new Test()

t.name = '111' // 不可修改
```

发布-订阅模式

vue里的*on*和emit就是发布订阅的实现

算法

冒泡排序

```
function bubble(array) {
  checkArray(array);
  for (let i = array.length - 1; i > 0; i--) {
    // 从 0 到 `length - 1` 遍历
    for (let j = 0; j < i; j++) {
      if (array[j] > array[j + 1]) swap(array, j, j + 1)
    }
  }
  return array;
}
```

插入排序

```
function insertion(array) {
  checkArray(array);
  for (let i = 1; i < array.length; i++) {
    for (let j = i - 1; j >= 0 && array[j] > array[j + 1]; j--)
      swap(array, j, j + 1);
  }
  return array;
}
```

快速排序

```
function sort(array) {
  checkArray(array);
```

```
  quickSort(array, 0, array.length - 1);
  return array;
}

function quickSort(array, left, right) {
  if (left < right) {
    swap(array, , right)
    // 随机取值，然后和末尾交换，这样做比固定取一个位置的复杂度略低
    let indexs = part(array, parseInt(Math.random() * (right - left + 1)) + left, r
ight);
    quickSort(array, left, indexs[0]);
    quickSort(array, indexs[1] + 1, right);
  }
}
function part(array, left, right) {
  let less = left - 1;
  let more = right;
  while (left < more) {
    if (array[left] < array[right]) {
      // 当前值比基准值小，`less` 和 `left` 都加一
       ++less;
       ++left;
    } else if (array[left] > array[right]) {
      // 当前值比基准值大，将当前值和右边的值交换
      // 并且不改变 `left`，因为当前换过来的值还没有判断过大小
      swap(array, --more, left);
    } else {
      // 和基准值相同，只移动下标
      left++;
    }
  }
  // 将基准值和比基准值大的第一个值交换位置
  // 这样数组就变成 `[比基准值小，基准值，比基准值大]`
  swap(array, right, more);
  return [less, more];
}
```