前端面试指南

INTERVIEW-MAP

2018年08月刊

Geekbang. 极客邦科技 InfoQ

前端面试指南

作者 YuChengKai luoguangcong
Sean Bill yygmind 穆尔

作者GitHub：https://github.com/KieSun
作者邮箱：zx597813039@gmail.com

=>>前端面试图谱GitHub

前言

当你老了，回顾一生，就会发觉：什么时候出国读书，什么时候决定做第一份职业，何时选定对象而恋爱，什么时候结婚，其实都是命运的巨变。只是当时站在三岔路口，眼见风云千樯，你做出选择的那一日，在日记上，相当沉闷和平凡，当时还以为是生命中普通的一天。

一个改变面试的项目 -- 面试图谱。

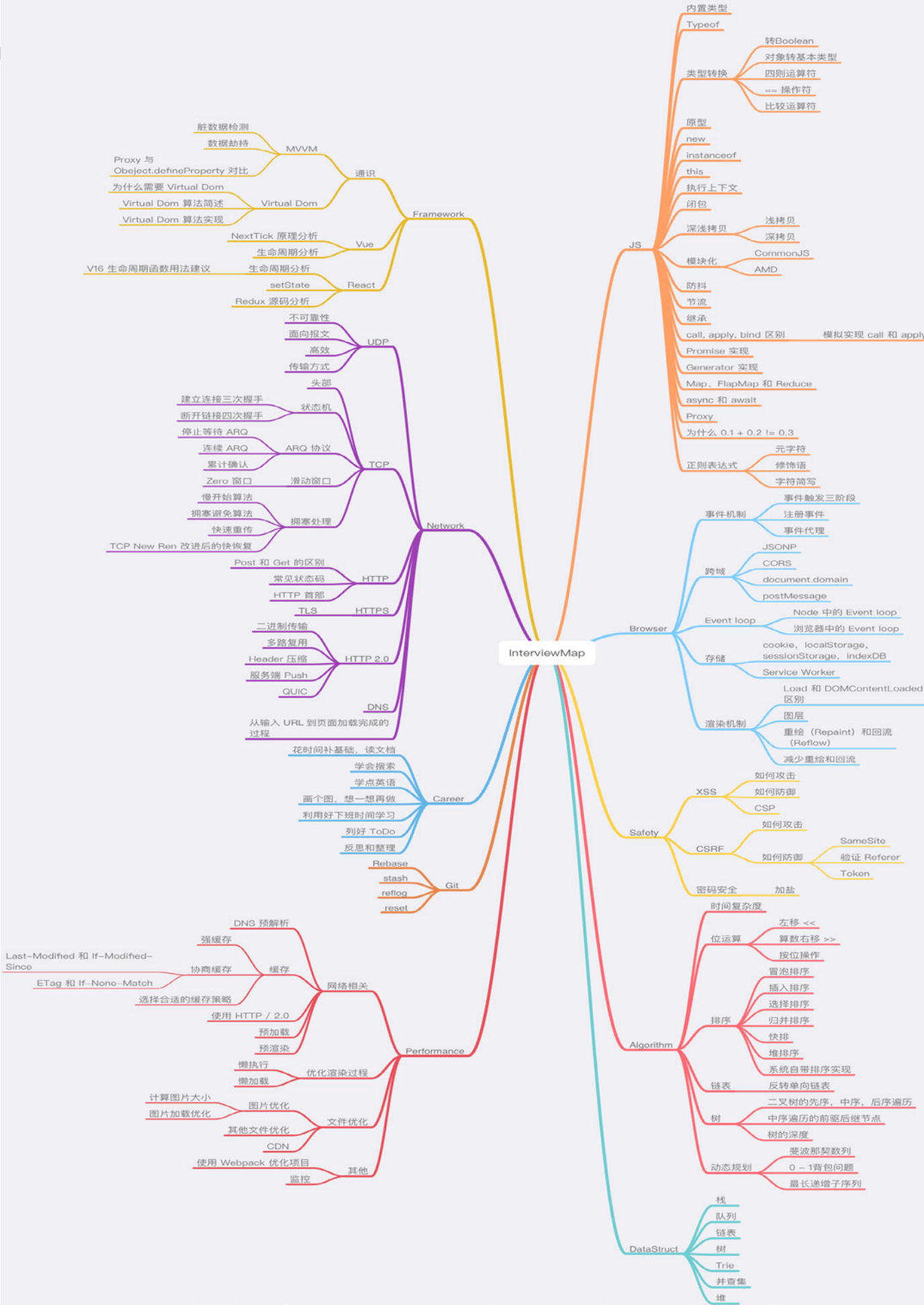
金九银十的秋招季近在眼前，想必大家也都心痒难耐，准备挑战更好的工作机会。那么，面试肯定是最大的挑战。

对于面试来说，平时的积累肯定是必须的，但是在面试前的准备也是至关重要的。

在几月前我个人组建了一个小团队，花了将近半年的时间寻找大厂的面试题，筛选出了近百个知识点然后成文，并全部翻译为英文。今天，终于开源出了第一个版本，目前总字数已高达 10 余万字。

我们认为，一味的背面试题是没多大作用的。只有熟悉了各个知识点并融会贯通，才能在面试中披荆斩棘。本图谱目前包含了近百个高频知识点，无论是面试前的准备还是平时学习中的查漏补缺，我们相信肯定能帮助到大家。目前内容包含了 JS、网络、浏览器相关、性能优化、安全、框架、Git、数据结构、算法等内容，无论是基础还是进阶，亦或是源码解读，你都能在本图谱中得到满意的答案，希望这个面试图谱能够帮助到大家更好的准备面试。

该仓库内容会持续更新，后期将会包含更多的内容，比如：系统设计、区块链、运维、后端等等，当然这些不是我的强项，我会邀请这方面有不错经验的朋友来书写内容。


InterviewMap
JS
内置类型
Typeof
类型转换
转Boolean
对象转基本类型
四则运算符
== 操作符
比较运算符
原型
new
instanceof
this
执行上下文
闭包
深浅拷贝
浅拷贝
深拷贝
模块化
CommonJS
AMD
防抖
节流
继承
call, apply, bind 区别
模拟实现 call 和 apply
Promise 实现
Generator 实现
Map、FlapMap 和 Reduce
async 和 await
Proxy
为什么 0.1 + 0.2 != 0.3
正则表达式
元字符
修饰语
字符简写
Browser
事件机制
事件触发三阶段
注册事件
事件代理
跨域
JSONP
CORS
document.domain
postMessage
Event loop
Node 中的 Event loop
浏览器中的 Event loop
存储
cookie，localStorage，sessionStorage，indexDB
Service Worker
渲染机制
Load 和 DOMContentLoaded 区别
图层
重绘（Repaint）和回流（Reflow）
减少重绘和回流
Safety
XSS
如何攻击
如何防御
CSP
CSRF
如何攻击
如何防御
SameSite
验证 Referer
Token
密码安全
加盐
Algorithm
时间复杂度
位运算
左移 <<
算数右移 >>
按位操作
排序
冒泡排序
插入排序
选择排序
归并排序
快排
堆排序
系统自带排序实现
链表
反转单向链表
树
二叉树的先序，中序，后序遍历
中序遍历的前驱后继节点
树的深度
动态规划
斐波那契数列
0 － 1背包问题
最长递增子序列
DataStruct
栈
队列
链表
树
Trie
并查集
堆
Framework
通识
MVVM
脏数据检测
数据劫持
Proxy 与 Obeject.defineProperty 对比
Virtual Dom
为什么需要 Virtual Dom
Virtual Dom 算法简述
Virtual Dom 算法实现
Vue
NextTick 原理分析
生命周期分析
React
生命周期分析
V16 生命周期函数用法建议
setState
Redux 源码分析
Network
UDP
不可靠性
面向报文
高效
传输方式
TCP
头部
状态机
建立连接三次握手
断开链接四次握手
ARQ 协议
停止等待 ARQ
连续 ARQ
累计确认
滑动窗口
Zero 窗口
拥塞处理
慢开始算法
拥塞避免算法
快速重传
TCP New Ren 改进后的快恢复
HTTP
Post 和 Get 的区别
常见状态码
HTTP 首部
HTTPS
TLS
HTTP 2.0
二进制传输
多路复用
Header 压缩
服务端 Push
QUIC
DNS
从输入 URL 到页面加载完成的过程
Career
花时间补基础，读文档
学会搜索
学点英语
画个图，想一想再做
利用好下班时间学习
列好 ToDo
反思和整理
Git
Rebase
stash
reflog
reset
Performance
网络相关
缓存
DNS 预解析
强缓存
协商缓存
Last-Modified 和 If-Modified-Since
ETag 和 If-None-Match
选择合适的缓存策略
使用 HTTP / 2.0
预加载
预渲染
优化渲染过程
懒执行
懒加载
文件优化
图片优化
计算图片大小
图片加载优化
其他文件优化
CDN
其他
使用 Webpack 优化项目
监控

InfoQ

在微信上关注我们

InfoQ 简介

InfoQ 面向 5 至 8 年工作经验的研发团队领导者、CTO、架构师、项目经理、工程总监和高级软件开发者等中高端技术人群，提供中立的、由技术实践主导的技术资讯及技术会议，搭建连接中国技术高端社区与国际主流技术社区的桥梁。

InfoQ 是一家全球性在线新闻 / 社区网站，创立于 2006 年，目前在全球拥有英、法、中、葡、日 5 种语言的站点。InfoQ 中国于 2007 年由极客邦科技创始人兼 CEO 霍泰稳先生引入中国，同年 3 月 28 日，InfoQ 中文站InfoQ.com.cn 正式上线。每年独立访问用户超过 2000 万人次。

InfoQ

国内最好的原创技术社区，一线互联网公司核心技术人员提供优质内容。订阅 InfoQ，看全球互联网技术最佳实践。

关注「InfoQ」

回复“架构师”，获取《架构师》电子书2017版合集

AI前线

提供最新最全AI领域技术资讯、一线业界实践案例、业界技术分享干货、最新AI论文解读。

关注「AI前线」

回复“AI”，下载《AI前线》系列迷你书

聊聊架构

以架构之“道”为基础，呈现更多的务实落地的架构内容。

关注「聊聊架构」

和百位架构师共聊架构

前端之巅

InfoQ大前端技术社群：囊括前端、移动、Node全栈一线技术，紧跟业界发展步伐。

关注「前端之巅」

回复“大前端”，下载大前端电子书

区块链前哨

掌握最前沿区块链资讯，深度分析区块链技术。从新手到精通，你只需要这一个专业助手。

关注「区块链前哨」

应对下一个互联网的到来

高效开发运维

常规运维，亦或是崛起的DevOps，探讨如何IT交付实现价值。

关注「高效开发运维」

回复“DevOps”，四篇精品文章领悟DevOps

目录 | Contents

JS

内置类型

JS 中分为七种内置类型，七种内置类型又分为两大类型：基本类型和对象（Object）。

基本类型有六种： null，undefined，boolean，number，string，symbol。

其中 JS 的数字类型是浮点类型的，没有整型。并且浮点类型基于 IEEE 754 标准实现，在使用中会遇到某些 Bug。NaN 也属于 number 类型，并且 NaN 不等于自身。

对于基本类型来说，如果使用字面量的方式，那么这个变量只是个字面量，只有在必要的时候才会转换为对应的类型

```
let a = 111 // 这只是字面量，不是 number 类型
a.toString() // 使用时候才会转换为对象类型
```

对象（Object）是引用类型，在使用过程中会遇到浅拷贝和深拷贝的问题。

```
let a = { name: 'FE' }
let b = a
b.name = 'EF'
console.log(a.name) // EF
```

Typeof

typeof 对于基本类型，除了 null 都可以显示正确的类型

```
typeof 1 // 'number'
typeof '1' // 'string'
typeof undefined // 'undefined'
```

```
typeof true // 'boolean'
typeof Symbol() // 'symbol'
typeof b // b 没有声明，但是还会显示 undefined
```

typeof 对于对象，除了函数都会显示 object

```
typeof [] // 'object'
typeof {} // 'object'
typeof console.log // 'function'
```

对于 null 来说，虽然它是基本类型，但是会显示 object，这是一个存在很久了的 Bug

```
typeof null // 'object'
```

PS：为什么会出现这种情况呢？因为在 JS 的最初版本中，使用的是 32 位系统，为了性能考虑使用低位存储了变量的类型信息，000 开头代表是对象，然而 null 表示为全零，所以将它错误的判断为 object 。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

如果我们想获得一个变量的正确类型，可以通过 Object.prototype.toString.call(xx)。这样我们就可以获得类似 [Object Type] 的字符串。

```
let a
// 我们也可以这样判断 undefined
a === undefined
// 但是 undefined 不是保留字，能够在低版本浏览器被赋值
let undefined = 1
// 这样判断就会出错
// 所以可以用下面的方式来判断，并且代码量更少
// 因为 void 后面随便跟上一个组成表达式
// 返回就是 undefined
a === void 0
```

类型转换

转Boolean

在条件判断时，除了 undefined， null， false， NaN， ''， 0， -0，其他所有值都转为 true，包括所有对象。

对象转基本类型

对象在转换基本类型时，首先会调用 valueOf 然后调用 toString。并且这两个方法你是可以重写的。

```
let a = {
    valueOf() {
        return 0
    }
}
```

当然你也可以重写 Symbol.toPrimitive ，该方法在转基本类型时调用优先级最高。

```
let a = {
```

```
  valueOf() {
      return 0;
    },
    toString() {
      return '1';
    },
    [Symbol.toPrimitive]() {
      return 2;
    }
  }
  1 + a // => 3
  '1' + a // => '12'
```

四则运算符

只有当加法运算时，其中一方是字符串类型，就会把另一个也转为字符串类型。其他运算只要其中一方是数字，那么另一方就转为数字。并且加法运算会触发三种类型转换：将值转换为原始值，转换为数字，转换为字符串。

```
1 + '1' // '11'
2 * '2' // 4
[1, 2] + [2, 1] // '1,22,1'
// [1, 2].toString() -> '1,2'
// [2, 1].toString() -> '2,1'
// '1,2' + '2,1' = '1,22,1'
```

对于加号需要注意这个表达式 'a' + + 'b'

```
'a' + + 'b' // -> "aNaN"
// 因为 + 'b' -> NaN
// 你也许在一些代码中看到过 + '1' -> 1
```

== 操作符

比较运算x==y, 其中x和 y是值，产生true或者false。这样的比较按如下方式进行:

1. 若Type(x)与Type(y)相同， 则
   a. 若Type(x)为Undefined， 返回true。
   b. 若Type(x)为Null， 返回true。
   c. 若Type(x)为Number， 则
      i. 若x为NaN， 返回false。
      ii. 若y为NaN， 返回false。
      iii. 若x与y为相等数值， 返回true。
      iv. 若x 为 +0 且 y为−0， 返回true。
      v. 若x 为 −0 且 y为+0， 返回true。
      vi. 返回false。
   d. 若Type(x)为String, 则当x和y为完全相同的字符序列（长度相等且相同字符在相同位置）时返回true。 否则，返回false。
   e. 若Type(x)为Boolean, 当x和y为同为true或者同为false时返回true。 否则， 返回false。
   f. 当x和y为引用同一对象时返回true。否则，返回false。
2. 若x为null且y为undefined， 返回true。
3. 若x为undefined且y为null， 返回true。
4. 若Type(x) 为 Number 且 Type(y)为String， 返回comparison x == ToNumber(y)的结果。
5. 若Type(x) 为 String 且 Type(y)为Number，
6. 返回比较ToNumber(x) == y的结果。
7. 若Type(x)为Boolean， 返回比较ToNumber(x) == y的结果。
8. 若Type(y)为Boolean， 返回比较x == ToNumber(y)的结果。
9. 若Type(x)为String或Number，且Type(y)为Object，返回比较x == ToPrimitive(y)的结果。
10. 若Type(x)为Object且Type(y)为String或Number， 返回比较ToPrimitive(x) == y的结果。
11. 返回false。

上图中的 toPrimitive 就是对象转基本类型。

这里来解析一道题目 [] == ![] // -> true ，下面是这个表达式为何为 true 的步骤

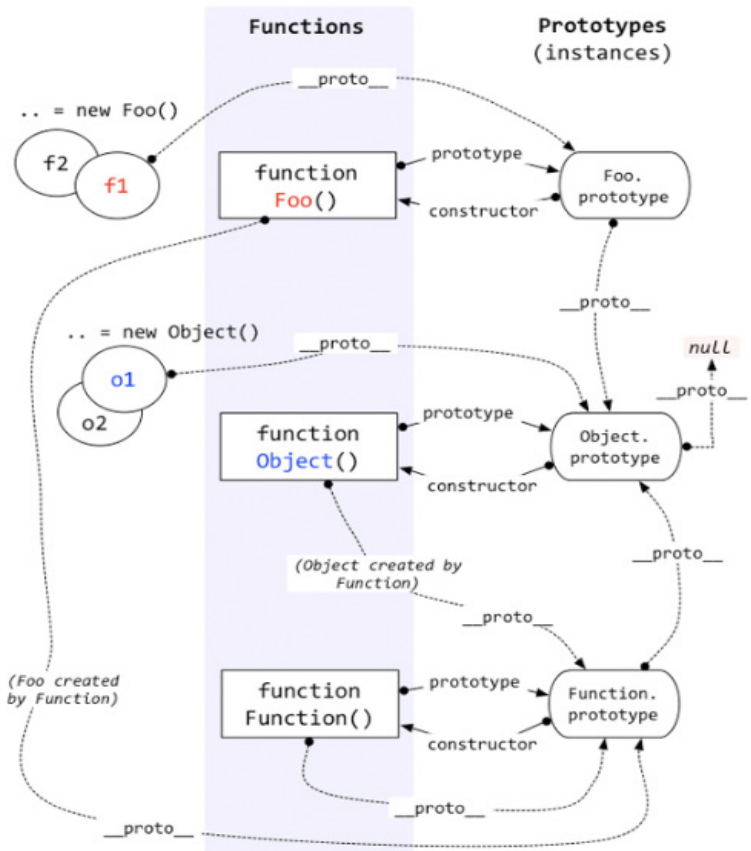
```
// [] 转成 true，然后取反变成 false
[] == false
// 根据第 8 条得出
[] == ToNumber(false)
[] == 0
// 根据第 10 条得出
ToPrimitive([]) == 0
// [].toString() -> ''
'' == 0
// 根据第 6 条得出
0 == 0 // -> true
```

比较运算符

如果是对象，就通过 toPrimitive 转换对象

如果是字符串，就通过 unicode 字符索引来比较

原型



每个函数都有 prototype 属性，除了 Function.prototype.bind()，该属性指向原型。

每个对象都有 __proto__ 属性，指向了创建该对象的构造函数的原型。其实这个属性指向了 [[prototype]]，但是 [[prototype]] 是内部属性，我们并不能访问到，所以使用 _proto_ 来访问。

对象可以通过 __proto__ 来寻找不属于该对象的属性，__proto__ 将对象连接起来组成了原型链。

如果你想更进一步的了解原型，可以仔细阅读 深度解析原型中的各个难点。

new

1. 新生成了一个对象
2. 链接到原型
3. 绑定 this
4. 返回新对象

在调用 new 的过程中会发生以上四件事情，我们也可以试着来自己实现一个 new

```
function create() {
    // 创建一个空的对象
    let obj = new Object()
    // 获得构造函数
    let Con = [].shift.call(arguments)
    // 链接到原型
    obj.__proto__ = Con.prototype
    // 绑定 this，执行构造函数
    let result = Con.apply(obj, arguments)
    // 确保 new 出来的是个对象
    return typeof result === 'object' ? result : obj
}
```

对于实例对象来说，都是通过 new 产生的，无论是 function Foo() 还是 let a = { b : 1 } 。

对于创建一个对象来说，更推荐使用字面量的方式创建对象（无论性能上还是可读性）。因为你使用 new Object() 的方式创建对象需要通过作用域链一层层找到 Object，但是你使用字面量的方式就没这个问题。

```
function Foo() {}
// function 就是个语法糖
// 内部等同于 new Function()
let a = { b: 1 }
// 这个字面量内部也是使用了 new Object()
```

对于 new 来说，还需要注意下运算符优先级。

```
function Foo() {
    return this;
}
```

```
Foo.getName = function () {
    console.log('1');
};
Foo.prototype.getName = function () {
    console.log('2');
};

new Foo.getName();   // -> 1
new Foo().getName(); // -> 2
```

Precedence	Operator type	Associativity	Individual operators
20	`Grouping`	n/a	`( … )`
19	`Member Access`	left-to-right	`… . …`
	`Computed Member Access`	left-to-right	`… [ … ]`
	`new` (with argument list)	n/a	`new … ( … )`
	Function Call	left-to-right	`… ( … )`
18	`new` (without argument list)	right-to-left	`new …`

从上图可以看出，new Foo() 的优先级大于 new Foo ，所以对于上述代码来说可以这样划分执行顺序

```
new (Foo.getName());
(new Foo()).getName();
```

对于第一个函数来说，先执行了 Foo.getName() ，所以结果为 1；对于后者来说，先执行 new Foo() 产生了一个实例，然后通过原型链找到了 Foo 上的 getName 函数，所以结果为 2。

instanceof

instanceof 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 prototype。

我们也可以试着实现一下 instanceof

```
function instanceof(left, right) {
    // 获得类型的原型
    let prototype = right.prototype
    // 获得对象的原型
    left = left.__proto__
    // 判断对象的类型是否等于类型的原型
    while (true) {
          if (left === null)
                return false
          if (prototype === left)
                return true
          left = left.__proto__
    }
}
```

this

this 是很多人会混淆的概念，但是其实他一点都不难，你只需要记住几个规则就可以了。

```
function foo() {
  console.log(this.a)
}
var a = 2
foo()

var obj = {
  a: 2,
  foo: foo
}
obj.foo()

// 以上两者情况 `this` 只依赖于调用函数前的对象，优先级是第二个情况大于第一个情况
// 以下情况是优先级最高的，`this` 只会绑定在 `c` 上，不会被任何方式修改 `this` 指向
var c = new foo()
c.a = 3
console.log(c.a)

// 还有种就是利用 call，apply，bind 改变 this，这个优先级仅次于 new
```

以上几种情况明白了，很多代码中的 this 应该就没什么问题了，下面让我们看看箭头函数中的 this

```
function a() {
    return () => {
        return () => {
          console.log(this)
        }
    }
}
console.log(a()()())
```

箭头函数其实是没有 this 的，这个函数中的 this 只取决于他外面的第一个不是箭头函数的函数的 this。在这个例子中，因为调用 a 符合前面代码中的第一个情况，所以 this 是 window。并且 this 一旦绑定了上下文，就不会被任何代码改变。

执行上下文

当执行 JS 代码时，会产生三种执行上下文

- 全局执行上下文
- 函数执行上下文
- eval 执行上下文

每个执行上下文中都有三个重要的属性

变量对象（VO），包含变量、函数声明和函数的形参，该属性只能在全局上下文中访问

作用域链（JS 采用词法作用域，也就是说变量的作用域是在定义时就决定了）

```
this
var a = 10
function foo(i) {
  var b = 20
}
foo()
```

对于上述代码，执行栈中有两个上下文：全局上下文和函数 foo 上下文。

```
stack = [
    globalContext,
    fooContext
]
```

对于全局上下文来说，VO 大概是这样的

```
globalContext.VO === globe
globalContext.VO = {
    a: undefined,
  foo: <Function>,
}
```

对于函数 foo 来说，VO 不能访问，只能访问到活动对象（AO）

```
fooContext.VO === foo.AO
fooContext.AO {
    i: undefined,
  b: undefined,
    arguments: <>
}
// arguments 是函数独有的对象(箭头函数没有)
// 该对象是一个伪数组，有 `length` 属性且可以通过下标访问元素
// 该对象中的 `callee` 属性代表函数本身
// `caller` 属性代表函数的调用者
```

对于作用域链，可以把它理解成包含自身变量对象和上级变量对象的列表，通过 [[Scope]] 属性查找上级变量

```
fooContext.[[Scope]] = [
    globalContext.VO
]
fooContext.Scope = fooContext.[[Scope]] + fooContext.VO
fooContext.Scope = [
    fooContext.VO,
    globalContext.VO
]
```

接下来让我们看一个老生常谈的例子，var

```
b() // call b
console.log(a) // undefined

var a = 'Hello world'

function b() {
  console.log('call b')
}
```

想必以上的输出大家肯定都已经明白了，这是因为函数和变量提升的原因。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解。但是更准确的解释应该是：在生成执行上下文时，会有两个阶段。第一个阶段是创建的阶段（具体步骤是创建 VO），JS 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 undefined，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用。

在提升的过程中，相同的函数会覆盖上一个函数，并且函数优先于变量提升

```
b() // call b second

function b() {
  console.log('call b fist')
}
function b() {
  console.log('call b second')
}
var b = 'Hello world'
```

var 会产生很多错误，所以在 ES6 中引入了 let。let 不能在声明前使用，但是这并不是常说的 let 不会提升，let 提升了声明但没有赋值，因为临时死区导致了并不能在声明前使用。

对于非匿名的立即执行函数需要注意以下一点

```
var foo = 1
(function foo() {
    foo = 10
    console.log(foo)
}()) // -> ƒ foo() { foo = 10 ; console.log(foo) }
```

因为当 JS 解释器在遇到非匿名的立即执行函数时，会创建一个辅助的特定对象，然后将函数名称作为这个对象的属性，因此函数内部才可以访问到 foo，但是这又个值是只读的，所以对它的赋值并不生效，所以打印的结果还是这个函数，并且外部的值也没有发生更改。

```
specialObject = {};

Scope = specialObject + Scope;
```

```
foo = new FunctionExpression;
foo.[[Scope]] = Scope;
specialObject.foo = foo; // {DontDelete}, {ReadOnly}

delete Scope[0]; // remove specialObject from the front of scope chain
```

闭包

闭包的定义很简单：函数 A 返回了一个函数 B，并且函数 B 中使用了函数 A 的变量，函数 B 就被称为闭包。

```
function A() {
  let a = 1
  function B() {
      console.log(a)
  }
  return B
}
```

你是否会疑惑，为什么函数 A 已经弹出调用栈了，为什么函数 B 还能引用到函数 A 中的变量。因为函数 A 中的变量这时候是存储在堆上的。现在的 JS 引擎可以通过逃逸分析辨别出哪些变量需要存储在堆上，哪些需要存储在栈上。

经典面试题，循环中使用闭包解决 var 定义函数的问题

```
for ( var i=1; i<=5; i++) {
  setTimeout( function timer() {
        console.log( i );
  }, i*1000 );
}˝
```

首先因为 setTimeout 是个异步函数，所有会先把循环全部执行完毕，这时候 i 就是 6 了，所以会输出一堆 6。

解决办法两种，第一种使用闭包

```
for (var i = 1; i <= 5; i++) {
  (function(j) {
    setTimeout(function timer() {
      console.log(j);
    }, j * 1000);
  })(i);
}
```

第二种就是使用 setTimeout 的第三个参数

```
for ( var i=1; i<=5; i++) {
  setTimeout( function timer(j) {
        console.log( j );
  }, i*1000, i);
}
```

第三种就是使用 let 定义 i 了

```
for ( let i=1; i<=5; i++) {
  setTimeout( function timer() {
         console.log( i );
  }, i*1000 );
}
```

因为对于 let 来说，他会创建一个块级作用域，相当于

```
{ // 形成块级作用域
  let i = 0
  {
    let ii = i
    setTimeout( function timer() {
        console.log( i );
    }, i*1000 );
  }
  i++
  {
    let ii = i
  }
  i++
  {
    let ii = i
  }
  ...
}
```

深浅拷贝

```
let a = {
    age: 1
}
let b = a
a.age = 2
console.log(b.age) // 2
```

从上述例子中我们可以发现，如果给一个变量赋值一个对象，那么两者的值会是同一个引用，其中一方改变，另一方也会相应改变。

通常在开发中我们不希望出现这样的问题，我们可以使用浅拷贝来解决这个问题。

浅拷贝

首先可以通过 Object.assign 来解决这个问题。

```
let a = {
    age: 1
}
let b = Object.assign({}, a)
a.age = 2
console.log(b.age) // 1
```

当然我们也可以通过展开运算符（…）来解决

```
let a = {
    age: 1
}
let b = {...a}
a.age = 2
console.log(b.age) // 1
```

通常浅拷贝就能解决大部分问题了，但是当我们遇到如下情况就需要使用到深拷贝了

```
let a = {
    age: 1,
    jobs: {
        first: 'FE'
    }
}
let b = {...a}
a.jobs.first = 'native'
console.log(b.jobs.first) // native
```

浅拷贝只解决了第一层的问题，如果接下去的值中还有对象的话，那么就又回到刚开始的话题了，两者享有相同的引用。要解决这个问题，我们需要引入深拷贝。

深拷贝

这个问题通常可以通过 JSON.parse(JSON.stringify(object)) 来解决。

```
let a = {
    age: 1,
    jobs: {
        first: 'FE'
    }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

但是该方法也是有局限性的：

- 会忽略 undefined
- 不能序列化函数
- 不能解决循环引用的对象

```
let obj = {
  a: 1,
  b: {
    c: 2,
    d: 3,
  },
}
```

```
obj.c = obj.b
obj.e = obj.a
obj.b.c = obj.c
obj.b.d = obj.b
obj.b.e = obj.b.c
let newObj = JSON.parse(JSON.stringify(obj))
console.log(newObj)
```

如果你有这么一个循环引用对象，你会发现你不能通过该方法深拷贝

```
▶Uncaught TypeError: Converting circular structure to JSON
    at JSON.stringify (<anonymous>)
    at <anonymous>:13:30
```

在遇到函数或者 undefined 的时候，该对象也不能正常的序列化

```
let a = {
    age: undefined,
    jobs: function() {},
    name: 'yck'
}
let b = JSON.parse(JSON.stringify(a))
console.log(b) // {name: "yck"}
```

你会发现在上述情况中，该方法会忽略掉函数和 undefined 。

但是在通常情况下，复杂数据都是可以序列化的，所以这个函数可以解决大部分问题，并且该函数是内置函数中处理深拷贝性能最快的。当然如果你的数据中含有以上三种情况下，可以使用 lodash 的深拷贝函数。

如果你所需拷贝的对象含有内置类型并且不包含函数，可以使用 MessageChannel

```
function structuralClone(obj) {
  return new Promise(resolve => {
    const {port1, port2} = new MessageChannel();
    port2.onmessage = ev => resolve(ev.data);
    port1.postMessage(obj);
  });
}

var obj = {a: 1, b: {
    c: b
}}
// 注意该方法是异步的
// 可以处理 undefined 和循环引用对象
const clone = await structuralClone(obj);
```

模块化

在有 Babel 的情况下，我们可以直接使用 ES6 的模块化

```
// file a.js
export function a() {}
export function b() {}
// file b.js
export default function() {}

import {a, b} from './a.js'
import XXX from './b.js'
```

CommonJS

CommonJs 是 Node 独有的规范，浏览器中使用就需要用到 Browserify 解析了。

```
// a.js
module.exports = {
    a: 1
}
// or
exports.a = 1

// b.js
var module = require('./a.js')
module.a // -> log 1
```

在上述代码中，module.exports 和 exports 很容易混淆，让我们来看看大致内部实现

```
var module = require('./a.js')
module.a
// 这里其实就是包装了一层立即执行函数，这样就不会污染全局变量了，
// 重要的是 module 这里，module 是 Node 独有的一个变量
module.exports = {
    a: 1
}
// 基本实现
var module = {
  exports: {} // exports 就是个空对象
}
// 这个是为什么 exports 和 module.exports 用法相似的原因
var exports = module.exports
var load = function (module) {
    // 导出的东西
    var a = 1
    module.exports = a
    return module.exports
};
```

再来说说 module.exports 和 exports，用法其实是相似的，但是不能对 exports 直接赋值，不会有任何效果。

对于 CommonJS 和 ES6 中的模块化的两者区别是：

- 前者支持动态导入，也就是 require(${path}/xx.js)，后者目前不支持，但是

已有提案

- 前者是同步导入，因为用于服务端，文件都在本地，同步导入即使卡住主线程影响也不大。而后者是异步导入，因为用于浏览器，需要下载文件，如果也采用导入会对渲染有很大影响
- 前者在导出时都是值拷贝，就算导出的值变了，导入的值也不会改变，所以如果想更新值，必须重新导入一次。但是后者采用实时绑定的方式，导入导出的值都指向同一个内存地址，所以导入值会跟随导出值变化
- 后者会编译成 require/exports 来执行的

AMD

AMD 是由 RequireJS 提出的

```
// AMD
define(['./a', './b'], function(a, b) {
    a.do()
    b.do()
})
define(function(require, exports, module) {
    var a = require('./a')
    a.doSomething()
    var b = require('./b')
    b.doSomething()
})
```

防抖

你是否在日常开发中遇到一个问题，在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作。

这些需求都可以通过函数防抖动来实现。尤其是第一个需求，如果在频繁的事件回调中做复杂计算，很有可能导致页面卡顿，不如将多次计算合并为一次计算，只在一个精确点做操作。因为防抖动的轮子很多，这里也不重新自己造个轮子了，直接使用 underscore 的源码来解释防抖动。

```
/**
 * underscore 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行
 *
 * @param  {function} func        回调函数
 * @param  {number}   wait        表示时间窗口的间隔
 * @param  {boolean}  immediate   设置为ture时，是否立即调用函数
 * @return {function}             返回客户调用函数
 */
_.debounce = function(func, wait, immediate) {
    var timeout, args, context, timestamp, result;
```

```
    var later = function() {
      // 现在和上一次时间戳比较
      var last = _.now() - timestamp;
      // 如果当前间隔时间少于设定时间且大于0就重新设置定时器
      if (last < wait && last >= 0) {
        timeout = setTimeout(later, wait - last);
      } else {
        // 否则的话就是时间到了执行回调函数
        timeout = null;
        if (!immediate) {
          result = func.apply(context, args);
          if (!timeout) context = args = null;
        }
      }
    };

    return function() {
      context = this;
      args = arguments;
      // 获得时间戳
      timestamp = _.now();
      // 如果定时器不存在且立即执行函数
      var callNow = immediate && !timeout;
      // 如果定时器不存在就创建一个
      if (!timeout) timeout = setTimeout(later, wait);
      if (callNow) {
        // 如果需要立即执行函数的话 通过 apply 执行
        result = func.apply(context, args);
        context = args = null;
      }

      return result;
    };
  };
```

整体函数实现的不难，总结一下。

- 对于按钮防点击来说的实现：一旦我开始一个定时器，只要我定时器还在，不管你怎么点击都不会执行回调函数。一旦定时器结束并设置为 null，就可以再次点击了。
- 对于延时执行函数来说的实现：每次调用防抖动函数都会判断本次调用和之前的时间间隔，如果小于需要的时间间隔，就会重新创建一个定时器，并且定时器的延时为设定时间减去之前的时间间隔。一旦时间到了，就会执行相应的回调函数。

节流

防抖动和节流本质是不一样的。防抖动是将多次执行变为最后一次执行，节流是将多次执行变成每隔一段时间执行。

```
/**
 * underscore 节流函数，返回函数连续调用时，func 执行频率限定为 次 / wait
 *
 * @param  {function}   func      回调函数
 * @param  {number}     wait      表示时间窗口的间隔
 * @param  {object}     options   如果想忽略开始函数的的调用，传入{leading: false}。
 *                                如果想忽略结尾函数的调用，传入{trailing: false}
 *                                两者不能共存，否则函数不能执行
 * @return {function}             返回客户调用函数
 */
_.throttle = function(func, wait, options) {
    var context, args, result;
    var timeout = null;
    // 之前的时间戳
    var previous = 0;
    // 如果 options 没传则设为空对象
    if (!options) options = {};
    // 定时器回调函数
    var later = function() {
      // 如果设置了 leading，就将 previous 设为 0
      // 用于下面函数的第一个 if 判断
      previous = options.leading === false ? 0 : _.now();
      // 置空一是为了防止内存泄漏，二是为了下面的定时器判断
      timeout = null;
      result = func.apply(context, args);
      if (!timeout) context = args = null;
    };
    return function() {
      // 获得当前时间戳
      var now = _.now();
      // 首次进入前者肯定为 true
    // 如果需要第一次不执行函数
    // 就将上次时间戳设为当前的
      // 这样在接下来计算 remaining 的值时会大于0
      if (!previous && options.leading === false) previous = now;
      // 计算剩余时间
      var remaining = wait - (now - previous);
      context = this;
      args = arguments;
```

```
      // 如果当前调用已经大于上次调用时间 + wait
      // 或者用户手动调了时间
    // 如果设置了 trailing，只会进入这个条件
    // 如果没有设置 leading，那么第一次会进入这个条件
    // 还有一点，你可能会觉得开启了定时器那么应该不会进入这个 if 条件了
    // 其实还是会进入的，因为定时器的延时
    // 并不是准确的时间，很可能你设置了2秒
    // 但是他需要2.2秒才触发，这时候就会进入这个条件
      if (remaining <= 0 || remaining > wait) {
        // 如果存在定时器就清理掉否则会调用二次回调
        if (timeout) {
          clearTimeout(timeout);
          timeout = null;
        }
        previous = now;
        result = func.apply(context, args);
        if (!timeout) context = args = null;
      } else if (!timeout && options.trailing !== false) {
        // 判断是否设置了定时器和 trailing
      // 没有的话就开启一个定时器
        // 并且不能不能同时设置 leading 和 trailing
        timeout = setTimeout(later, remaining);
      }
      return result;
    };
  };
```

继承

在 ES5 中，我们可以使用如下方式解决继承的问题

```
function Super() {}
Super.prototype.getNumber = function() {
  return 1
}

function Sub() {}
let s = new Sub()
Sub.prototype = Object.create(Super.prototype, {
  constructor: {
    value: Sub,
    enumerable: false,
    writable: true,
    configurable: true
  }
})
```

以上继承实现思路就是将子类的原型设置为父类的原型

在 ES6 中，我们可以通过 class 语法轻松解决这个问题

```
class MyDate extends Date {
  test() {
    return this.getTime()
  }
}
let myDate = new MyDate()
myDate.test()
```

但是 ES6 不是所有浏览器都兼容，所以我们需要使用 Babel 来编译这段代码。

如果你使用编译过得代码调用 myDate.test() 你会惊奇地发现出现了报错。

```
▶Uncaught TypeError: this is not a Date object.
    at MyDate.getDate (<anonymous>)
    at MyDate.test (<anonymous>:23:19)
    at <anonymous>:1:8
```

因为在 JS 底层有限制，如果不是由 Date 构造出来的实例的话，是不能调用 Date 里的函数的。所以这也侧面的说明了：ES6 中的 class 继承与 ES5 中的一般继承写法是不同的。

既然底层限制了实例必须由 Date 构造出来，那么我们可以改变下思路实现继承

```
function MyData() {
}
MyData.prototype.test = function () {
  return this.getTime()
}
let d = new Date()
Object.setPrototypeOf(d, MyData.prototype)
Object.setPrototypeOf(MyData.prototype, Date.prototype)
```

以上继承实现思路：先创建父类实例 => 改变实例原先的 _proto__ 转而连接到子类的 prototype => 子类的 prototype 的 __proto__ 改为父类的 prototype。

通过以上方法实现的继承就可以完美解决 JS 底层的这个限制。

call, apply, bind 区别

首先说下前两者的区别。

call 和 apply 都是为了解决改变 this 的指向。作用都是相同的，只是传参的方式不同。

除了第一个参数外，call 可以接收一个参数列表，apply 只接受一个参数数组。

```
let a = {
    value: 1
}
function getValue(name, age) {
```

```
    console.log(name)
    console.log(age)
    console.log(this.value)
}
getValue.call(a, 'yck', '24')
getValue.apply(a, ['yck', '24'])
```

模拟实现 call 和 apply

可以从以下几点来考虑如何实现

不传入第一个参数，那么默认为 window

改变了 this 指向，让新的对象可以执行该函数。那么思路是否可以变成给新的对象添加一个函数，然后在执行完以后删除？

```
Function.prototype.myCall = function (context) {
  var context = context || window
  // 给 context 添加一个属性
  // getValue.call(a, 'yck', '24') => a.fn = getValue
  context.fn = this
  // 将 context 后面的参数取出来
  var args = [...arguments].slice(1)
  // getValue.call(a, 'yck', '24') => a.fn('yck', '24')
  var result = context.fn(...args)
  // 删除 fn
  delete context.fn
  return result
}
```

以上就是 call 的思路，apply 的实现也类似

```
Function.prototype.myApply = function (context) {
  var context = context || window
  context.fn = this

  var result
  // 需要判断是否存储第二个参数
  // 如果存在，就将第二个参数展开
  if (arguments[1]) {
    result = context.fn(...arguments[1])
  } else {
    result = context.fn()
  }

  delete context.fn
  return result
}
```

bind 和其他两个方法作用也是一致的，只是该方法会返回一个函数。并且我们可以通过 bind 实现柯里化。

同样的，也来模拟实现下 bind

```
Function.prototype.myBind = function (context) {
  if (typeof this !== 'function') {
    throw new TypeError('Error')
  }
  var _this = this
  var 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))
  }
}
```

Promise 实现

Promise 是 ES6 新增的语法，解决了回调地狱的问题。

可以把 Promise 看成一个状态机。初始是 pending 状态，可以通过函数 resolve 和 reject ，将状态转变为 resolved 或者 rejected 状态，状态一旦改变就不能再次变化。

then 函数会返回一个 Promise 实例，并且该返回值是一个新的实例而不是之前的实例。因为 Promise 规范规定除了 pending 状态，其他状态是不可以改变的，如果返回的是一个相同实例的话，多个 then 调用就失去意义了。

对于 then 来说，本质上可以把它看成是 flatMap

```
// 三种状态
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";
// promise 接收一个函数参数，该函数会立即执行
function MyPromise(fn) {
  let _this = this;
  _this.currentState = PENDING;
  _this.value = undefined;
  // 用于保存 then 中的回调，只有当 promise
  // 状态为 pending 时才会缓存，并且每个实例至多缓存一个
  _this.resolvedCallbacks = [];
  _this.rejectedCallbacks = [];
  _this.resolve = function (value) {
    if (value instanceof MyPromise) {
      // 如果 value 是个 Promise，递归执行
```

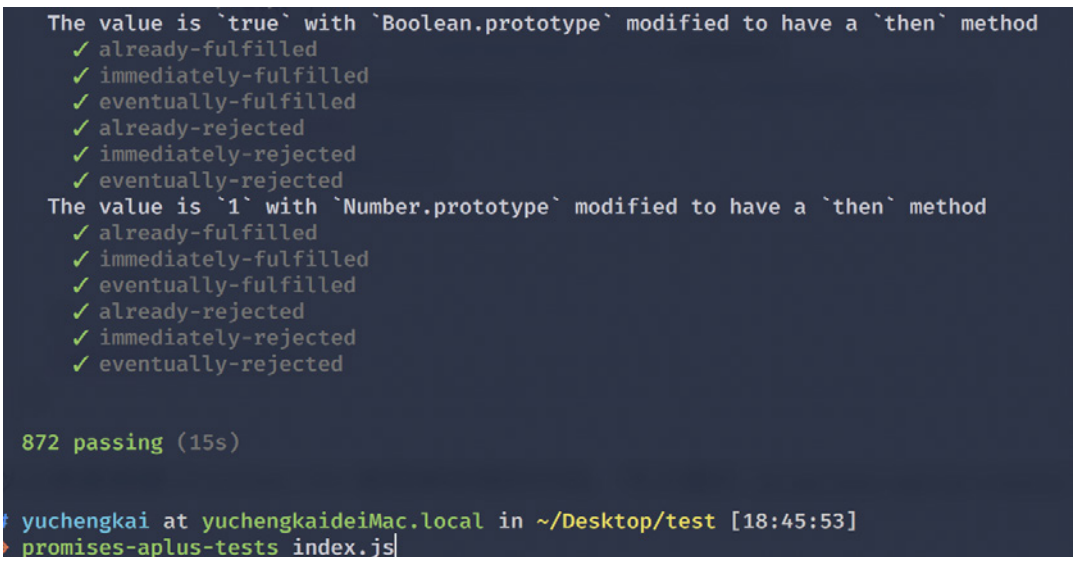
```
      return value.then(_this.resolve, _this.reject)
    }
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = RESOLVED;
        _this.value = value;
        _this.resolvedCallbacks.forEach(cb => cb());
      }
    })
  };

  _this.reject = function (reason) {
    setTimeout(() => { // 异步执行，保证执行顺序
      if (_this.currentState === PENDING) {
        _this.currentState = REJECTED;
        _this.value = reason;
        _this.rejectedCallbacks.forEach(cb => cb());
      }
    })
  }
  // 用于解决以下问题
  // new Promise(() => throw Error('error'))
  try {
    fn(_this.resolve, _this.reject);
  } catch (e) {
    _this.reject(e);
  }
}

MyPromise.prototype.then = function (onResolved, onRejected) {
  var self = this;
  // 规范 2.2.7，then 必须返回一个新的 promise
  var promise2;
  // 规范 2.2.onResolved 和 onRejected 都为可选参数
  // 如果类型不是函数需要忽略，同时也实现了透传
  // Promise.resolve(4).then().then((value) => console.log(value))
  onResolved = typeof onResolved === 'function' ? onResolved : v => v;
  onRejected = typeof onRejected === 'function' ? onRejected : r => throw r;
  if (self.currentState === RESOLVED) {
    return (promise2 = new MyPromise(function (resolve, reject) {
      // 规范 2.2.4，保证 onFulfilled，onRjected 异步执行
      // 所以用了 setTimeout 包裹下
      setTimeout(function () {
        try {
```

```
        var x = onResolved(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
      } catch (reason) {
        reject(reason);
      }
    });
  }));
}

if (self.currentState === REJECTED) {
  return (promise2 = new MyPromise(function (resolve, reject) {
    setTimeout(function () {
      // 异步执行onRejected
      try {
        var x = onRejected(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
      } catch (reason) {
        reject(reason);
      }
    });
  }));
}

if (self.currentState === PENDING) {
  return (promise2 = new MyPromise(function (resolve, reject) {
    self.resolvedCallbacks.push(function () {
      // 考虑到可能会有报错，所以使用 try/catch 包裹
      try {
        var x = onResolved(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
      } catch (r) {
        reject(r);
      }
    });

    self.rejectedCallbacks.push(function () {
      try {
        var x = onRejected(self.value);
        resolutionProcedure(promise2, x, resolve, reject);
      } catch (r) {
        reject(r);
      }
    });
  }));
```

```
  }
};
// 规范 2.3
function resolutionProcedure(promise2, x, resolve, reject) {
  // 规范 2.3.1，x 不能和 promise2 相同，避免循环引用
  if (promise2 === x) {
    return reject(new TypeError("Error"));
  }
  // 规范 2.3.2
  // 如果 x 为 Promise，状态为 pending 需要继续等待否则执行
  if (x instanceof MyPromise) {
    if (x.currentState === PENDING) {
      x.then(function (value) {
        // 再次调用该函数是为了确认 x resolve 的
        // 参数是什么类型，如果是基本类型就再次 resolve
        // 把值传给下个 then
        resolutionProcedure(promise2, value, resolve, reject);
      }, reject);
    } else {
      x.then(resolve, reject);
    }
    return;
  }
  // 规范 2.3.3.3.3
  // reject 或者 resolve 其中一个执行过得话，忽略其他的
  let called = false;
  // 规范 2.3.3，判断 x 是否为对象或者函数
  if (x !== null && (typeof x === "object" || typeof x === "function")) {
    // 规范 2.3.3.2，如果不能取出 then，就 reject
    try {
      // 规范 2.3.3.1
      let then = x.then;
      // 如果 then 是函数，调用 x.then
      if (typeof then === "function") {
        // 规范 2.3.3.3
        then.call(
          x,
          y => {
            if (called) return;
            called = true;
            // 规范 2.3.3.3.1
            resolutionProcedure(promise2, y, resolve, reject);
          },
          e => {
```

```
                if (called) return;
                called = true;
                reject(e);
              }
            );
          } else {
            // 规范 2.3.3.4
            resolve(x);
          }
        } catch (e) {
          if (called) return;
          called = true;
          reject(e);
        }
      } else {
        // 规范 2.3.4，x 为基本类型
        resolve(x);
      }
    }
```

以上就是根据 Promise / A+ 规范来实现的代码，可以通过 promises-aplus-tests 的完整测试



Generator 实现

Generator 是 ES6 中新增的语法，和 Promise 一样，都可以用来异步编程

```
// 使用 * 表示这是一个 Generator 函数
// 内部可以通过 yield 暂停代码
```

```
// 通过调用 next 恢复执行
function* test() {
  let a = 1 + 2;
  yield 2;
  yield 3;
}
let b = test();
console.log(b.next()); // >  { value: 2, done: false }
console.log(b.next()); // >  { value: 3, done: false }
console.log(b.next()); // >  { value: undefined, done: true }
```

从以上代码可以发现，加上 * 的函数执行后拥有了 next 函数，也就是说函数执行后返回了一个对象。每次调用 next 函数可以继续执行被暂停的代码。以下是 Generator 函数的简单实现

```
// cb 也就是编译过的 test 函数
function generator(cb) {
  return (function() {
    var object = {
      next: 0,
      stop: function() {}
    };

    return {
      next: function() {
        var ret = cb(object);
        if (ret === undefined) return { value: undefined, done: true };
        return {
          value: ret,
          done: false
        };
      }
    };
  })();
}
// 如果你使用 babel 编译后可以发现 test 函数变成了这样
function test() {
  var a;
  return generator(function(_context) {
    while (1) {
      switch ((_context.prev = _context.next)) {
        // 可以发现通过 yield 将代码分割成几块
        // 每次执行 next 函数就执行一块代码
        // 并且表明下次需要执行哪块代码
        case 0:
```

```
            a = 1 + 2;
            _context.next = 4;
            return 2;
          case 4:
            _context.next = 6;
            return 3;
            // 执行完毕
          case 6:
          case "end":
            return _context.stop();
        }
      }
    });
}
```

Map、FlatMap 和 Reduce

Map 作用是生成一个新数组，遍历原数组，将每个元素拿出来做一些变换然后 append 到新的数组中。

```
[1, 2, 3].map((v) => v + 1)
// -> [2, 3, 4]
```

Map 有三个参数，分别是当前索引元素，索引，原数组

```
['1','2','3'].map(parseInt)
//  parseInt('1', 0) -> 1
//  parseInt('2', 1) -> NaN
//  parseInt('3', 2) -> NaN
```

FlapMap 和 map 的作用几乎是相同的，但是对于多维数组来说，会将原数组降维。可以将 FlapMap 看成是 map + flatten ，目前该函数在浏览器中还不支持。

```
[1, [2], 3].flatMap((v) => v + 1)
// -> [2, 3, 4]
```

如果想将一个多维数组彻底的降维，可以这样实现

```
const flattenDeep = (arr) => Array.isArray(arr)
  ? arr.reduce( (a, b) => [...flattenDeep(a), ...flattenDeep(b)] , [])
  : [arr]

flattenDeep([1, [[2], [3, [4]], 5]])
```

Reduce 作用是数组中的值组合起来，最终得到一个值

```
function a() {
    console.log(1);
}

function b() {
    console.log(2);
}
```

```
[a, b].reduce((a, b) => a(b()))
// -> 2 1
```

async 和 await

一个函数如果加上 async ，那么该函数就会返回一个 Promise

```
async function test() {
  return "1";
}
console.log(test()); // -> Promise {<resolved>: "1"}
```

可以把 async 看成将函数返回值使用 Promise.resolve() 包裹了下。

await 只能在 async 函数中使用

```
function sleep() {
  return new Promise(resolve => {
    setTimeout(() => {
      console.log('finish')
      resolve("sleep");
    }, 2000);
  });
}
async function test() {
  let value = await sleep();
  console.log("object");
}
test()
```

上面代码会先打印 finish 然后再打印 object 。因为 await 会等待 sleep 函数 resolve ，所以即使后面是同步代码，也不会先去执行同步代码再来执行异步代码。

async 和 await 相比直接使用 Promise 来说，优势在于处理 then 的调用链，能够更清晰准确的写出代码。缺点在于滥用 await 可能会导致性能问题，因为 await 会阻塞代码，也许之后的异步代码并不依赖于前者，但仍然需要等待前者完成，导致代码失去了并发性。

下面来看一个使用 await 的代码。

```
var a = 0
var b = async () => {
  a = a + await 10
  console.log('2', a) // -> '2' 10
  a = (await 10) + a
  console.log('3', a) // -> '3' 20
}
b()
a++
console.log('1', a) // -> '1' 1
```

对于以上代码你可能会有疑惑，这里说明下原理

首先函数 b 先执行，在执行到 await 10 之前变量 a 还是 0，因为在 await 内部实现了 generators ，generators 会保留堆栈中东西，所以这时候 a = 0 被保存了下来

因为 await 是异步操作，所以会先执行 console.log('1', a)

这时候同步代码执行完毕，开始执行异步代码，将保存下来的值拿出来使用，这时候 a = 10

然后后面就是常规执行代码了

Proxy

Proxy 是 ES6 中新增的功能，可以用来自定义对象中的操作

```
let p = new Proxy(target, handler);
// `target` 代表需要添加代理的对象
// `handler` 用来自定义对象中的操作
```

可以很方便的使用 Proxy 来实现一个数据绑定和监听

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver);
    },
    set(target, property, value, receiver) {
      setBind(value);
      return Reflect.set(target, property, value);
    }
  };
  return new Proxy(obj, handler);
};

let obj = { a: 1 }
let value
let p = onWatch(obj, (v) => {
  value = v
}, (target, property) => {
  console.log(`Get '${property}' = ${target[property]}`);
})
p.a = 2 // bind `value` to `2`
p.a // -> Get 'a' = 2
```

为什么 0.1 + 0.2 != 0.3

因为 JS 采用 IEEE 754 双精度版本（64 位），并且只要采用 IEEE 754 的语言都有该问题。

我们都知道计算机表示十进制是采用二进制表示的，所以 0.1 在二进制表示为

```
// (0011) 表示循环
0.1 = 2^-4 * 1.10011(0011)
```


那么如何得到这个二进制的呢，我们可以来演算下

小数算二进制和整数不同。乘法计算时，只计算小数位，整数位用作每一位的二进制，并且得到的第一位为最高位。所以我们得出 0.1 = 2^-4 * 1.10011(0011)，那么 0.2 的演算也基本如上所示，只需要去掉第一步乘法，所以得出 0.2 = 2^-3 * 1.10011(0011)。

回来继续说 IEEE 754 双精度。六十四位中符号位占一位，整数位占十一位，其余五十二位都为小数位。因为 0.1 和 0.2 都是无限循环的二进制了，所以在小数位末尾处需要判断是否进位（就和十进制的四舍五入一样）。

所以 2^-4 * 1.10011...001 进位后就变成了 2^-4 * 1.10011(0011 * 12 次)010 。那么把这两个二进制加起来会得出 2^-2 * 1.0011(0011 * 11 次)0100 ，这个值算成十进制就是 0.30000000000000004

下面说一下原生解决办法，如下代码所示

```
parseFloat((0.1 + 0.2).toFixed(10))
```

正则表达式

元字符

元字符	作用
.	匹配任意字符除了换行符和回车符
[]	匹配方括号内的任意字符。比如 [0-9] 就可以用来匹配任意数字
^	^9，这样使用代表匹配以 9 开头。[^9]，这样使用代表不匹配方括号内除了 9 的字符
{1, 2}	匹配 1 到 2 位字符
(yck)	只匹配和 yck 相同字符串
|	匹配 | 前后任意字符
\	转义
*	只匹配出现 -1 次以上 * 前的字符
+	只匹配出现 0 次以上 + 前的字符
?	? 之前字符可选

修饰语

修饰语	作用
i	忽略大小写
g	全局搜索
m	多行

字符简写

简写	作用
\w	匹配字母数字或下划线
\W	和上面相反
\s	匹配任意的空白符
\S	和上面相反
\d	匹配数字
\D	和上面相反
\b	匹配单词的开始或结束
\B	和上面相反

V8 下的垃圾回收机制

V8 实现了准确式 GC，GC 算法采用了分代式垃圾回收机制。因此，V8 将内存（堆）分为新生代和老生代两部分。

新生代算法

新生代中的对象一般存活时间较短，使用 Scavenge GC 算法。

在新生代空间中，内存空间分为两部分，分别为 From 空间和 To 空间。在这两个空间中，必定有一个空间是使用的，另一个空间是空闲的。新分配的对象会被放入 From 空间中，当 From 空间被占满时，新生代 GC 就会启动了。算法会检查 From 空间中存活的对象并复制到 To 空间中，如果有失活的对象就会销毁。当复制完成后将 From 空间和 To 空间互换，这样 GC 就结束了。

老生代算法

老生代中的对象一般存活时间较长且数量也多，使用了两个算法，分别是标记清除算法和标记压缩算法。

在讲算法前，先来说下什么情况下对象会出现在老生代空间中：

- 新生代中的对象是否已经经历过一次 Scavenge 算法，如果经历过的话，会将对象从新生代空间移到老生代空间中。
- To 空间的对象占比大小超过 25 %。在这种情况下，为了不影响到内存分配，会将对象从新生代空间移到老生代空间中。

老生代中的空间很复杂，有如下几个空间

```
enum AllocationSpace {
  // TODO(v8:7464): Actually map this space's memory as read-only.
  RO_SPACE,    // 不变的对象空间
  NEW_SPACE,   // 新生代用于 GC 复制算法的空间
  OLD_SPACE,   // 老生代常驻对象空间
  CODE_SPACE,  // 老生代代码对象空间
  MAP_SPACE,   // 老生代 map 对象
  LO_SPACE,    // 老生代大空间对象
  NEW_LO_SPACE,  // 新生代大空间对象
```

```
    FIRST_SPACE = RO_SPACE,
    LAST_SPACE = NEW_LO_SPACE,
    FIRST_GROWABLE_PAGED_SPACE = OLD_SPACE,
    LAST_GROWABLE_PAGED_SPACE = MAP_SPACE
  };
```

在老生代中，以下情况会先启动标记清除算法：

- 某一个空间没有分块的时候
- 空间中被对象超过一定限制
- 空间不能保证新生代中的对象移动到老生代中

在这个阶段中，会遍历堆中所有的对象，然后标记活的对象，在标记完成后，销毁所有没有被标记的对象。在标记大型对内存时，可能需要几百毫秒才能完成一次标记。这就会导致一些性能上的问题。为了解决这个问题，2011 年，V8 从 stop-the-world 标记切换到增量标志。在增量标记期间，GC 将标记工作分解为更小的模块，可以让 JS 应用逻辑在模块间隙执行一会，从而不至于让应用出现停顿情况。但在 2018 年，GC 技术又有了一个重大突破，这项技术名为并发标记。该技术可以让 GC 扫描和标记对象时，同时允许 JS 运行，你可以点击 该博客 详细阅读。

清除对象后会造成堆内存出现碎片的情况，当碎片超过一定限制后会启动压缩算法。在压缩过程中，将活的对象像一端移动，直到所有对象都移动完成然后清理掉不需要的内存。

浏览器

事件机制

事件触发三阶段

事件触发有三个阶段

- document 往事件触发处传播，遇到注册的捕获事件会触发
- 传播到事件触发处时触发注册的事件
- 从事件触发处往 document 传播，遇到注册的冒泡事件会触发

事件触发一般来说会按照上面的顺序进行，但是也有特例，如果给一个目标节点同时注册冒泡和捕获事件，事件触发会按照注册的顺序执行。

```
// 以下会先打印冒泡然后是捕获
node.addEventListener('click',(event) =>{
  console.log('冒泡')
},false);
node.addEventListener('click',(event) =>{
  console.log('捕获 ')
},true)
```

注册事件

通常我们使用 addEventListener 注册事件，该函数的第三个参数可以是布尔值，也可以是对象。对于布尔值 useCapture 参数来说，该参数默认值为 false 。useCapture 决定了注册的事件是捕获事件还是冒泡事件。对于对象参数来说，可以使用以下几个属性

- capture，布尔值，和 useCapture 作用一样
- once，布尔值，值为 true 表示该回调只会调用一次，调用后会移除监听
- passive，布尔值，表示永远不会调用 preventDefault

一般来说，我们只希望事件只触发在目标上，这时候可以使用 stopPropagation 来阻止事件的进一步传播。通常我们认为 stopPropagation 是用来阻止事件冒泡的，其实该函数也可以阻止捕获事件。stopImmediatePropagation 同样也能实现阻止事件，但是还能阻止该事件目标执行别的注册事件。

```
node.addEventListener('click',(event) =>{
  event.stopImmediatePropagation()
  console.log('冒泡')
},false);
// 点击 node 只会执行上面的函数，该函数不会执行
node.addEventListener('click',(event) => {
  console.log('捕获 ')
},true)
```

事件代理

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话应该注册在父节点上

```
<ul id="ul">
  <li>1</li>
    <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
<script>
  let ul = document.querySelector('##ul')
  ul.addEventListener('click', (event) => {
         console.log(event.target);
  })
</script>
```

事件代理的方式相对于直接给目标注册事件来说，有以下优点

- 节省内存
- 不需要给子节点注销事件

跨域

因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，Ajax 请求会失败。

我们可以通过以下几种常用方法解决跨域的问题

JSONP

JSONP 的原理很简单，就是利用 〈script〉 标签没有跨域限制的漏洞。通过

<script> 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时。

```
<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
          console.log(data)
  }
</script>
```

JSONP 使用简单且兼容性不错，但是只限于 get 请求。

在开发中可能会遇到多个 JSONP 请求的回调函数名是相同的，这时候就需要自己封装一个 JSONP，以下是简单实现

```
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement("script");
  script.src = url;
  script.async = true;
  script.type = "text/javascript";
  window[jsonpCallback] = function(data) {
    success & success(data);
  };
  document.body.appendChild(script);
}
jsonp(
  "http://xxx",
  "callback",
  function(value) {
    console.log(value);
  }
);
```

CORS

CORS 需要浏览器和后端同时支持。IE 8 和 9 需要通过 XDomainRequest 来实现。

浏览器会自动进行 CORS 通信，实现 CORS 通信的关键是后端。只要后端实现了 CORS，就实现了跨域。

服务端设置 Access-Control-Allow-Origin 就可以开启 CORS。 该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。

document.domain

该方式只能用于二级域名相同的情况下，比如 a.test.com 和 b.test.com 适用于该方式。

只需要给页面添加 document.domain = 'test.com' 表示二级域名都相同就可以实现跨域

postMessage

这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

```
// 发送消息端
window.parent.postMessage('message', 'http://test.com');
// 接收消息端
var mc = new MessageChannel();
mc.addEventListener('message', (event) => {
    var origin = event.origin || event.originalEvent.origin;
    if (origin === 'http://test.com') {
        console.log('验证通过')
    }
});
```

Event loop

众所周知 JS 是门非阻塞单线程语言，因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是门多线程的语言话，我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点），当然可以引入读写锁解决这个问题。

JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task（有多种 task） 队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为。

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

以上代码虽然 setTimeout 延时为 0，其实还是异步。这是因为 HTML5 标准规定这个函数第二个参数不得小于 4 毫秒，不足会自动增加。所以 setTimeout 还是会在 script end 之后打印。

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（microtask） 和 宏任务（macrotask）。在 ES6 规范中，microtask 称为 jobs，macrotask 称为 task。

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
    console.log('Promise')
    resolve()
}).then(function() {
```

```
    console.log('promise1');
  }).then(function() {
    console.log('promise2');
  });

  console.log('script end');
  // script start => Promise => script end => promise1 => promise2 => setTimeout
```

以上代码虽然 setTimeout 写在 Promise 之前，但是因为 Promise 属于微任务而 setTimeout 属于宏任务，所以会有以上的打印。

微任务包括 process.nextTick ，promise ，Object.observe ，MutationObserver

宏任务包括 script ， setTimeout ，setInterval ，setImmediate ，I/O ，UI rendering

很多人有个误区，认为微任务快于宏任务，其实是错误的。因为宏任务中包括了 script ，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务。

所以正确的一次 Event loop 顺序是这样的

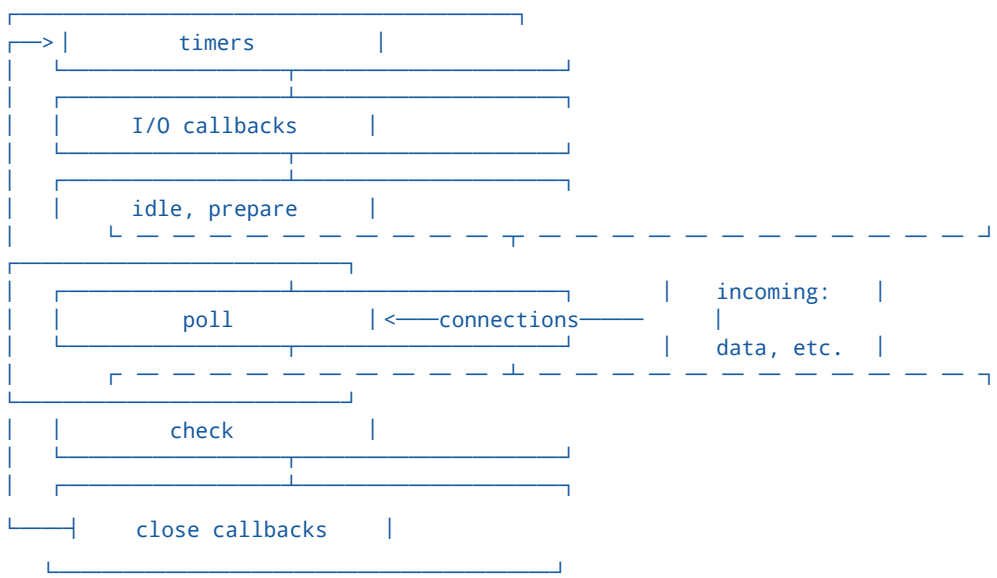
1. 执行同步代码，这属于宏任务
2. 执行栈为空，查询是否有微任务需要执行
3. 执行所有微任务
4. 必要的话渲染 UI
5. 然后开始下一轮 Event loop，执行宏任务中的异步代码

通过上述的 Event loop 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话，为了更快的 界面响应，我们可以把操作 DOM 放入微任务中。

Node 中的 Event loop

Node 中的 Event loop 和浏览器中的不相同。

Node 的 Event loop 分为 6 个阶段，它们会按照顺序反复运行



timer

timers 阶段会执行 setTimeout 和 setInterval

一个 timer 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟。

下限的时间有一个范围：[1, 2147483647] ，如果设定的时间不在这个范围，将被设置为 1。

I/O

I/O 阶段会执行除了 close 事件，定时器和 setImmediate 的回调

idle, prepare

idle, prepare 阶段内部实现

poll

poll 阶段很重要，这一阶段中，系统会做两件事情

1. 执行到点的定时器
2. 执行 poll 队列中的事件

并且当 poll 中没有定时器的情况下，会发现以下两件事情

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
- 如果 poll 队列为空，会有两件事发生
  - 如果有 setImmediate 需要执行，poll 阶段会停止并且进入到 check 阶段执行 setImmediate
  - 如果没有 setImmediate 需要执行，会等待回调被加入到队列中并立即执行回调

如果有别的定时器需要被执行，会回到 timer 阶段执行回调。

check

check 阶段执行 setImmediate

close callbacks

close callbacks 阶段执行 close 事件

并且在 Node 中，有些情况下的定时器执行顺序是随机的

```
setTimeout(() => {
    console.log('setTimeout');
}, 0);
setImmediate(() => {
    console.log('setImmediate');
})
// 这里可能会输出 setTimeout，setImmediate
// 可能也会相反的输出，这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒，这时候会执行 setImmediate
// 否则会执行 setTimeout
```

当然在这种情况下，执行顺序是相同的

```
var fs = require('fs')
```

```
fs.readFile(__filename, () => {
    setTimeout(() => {
        console.log('timeout');
    }, 0);
    setImmediate(() => {
        console.log('immediate');
    });
});
// 因为 readFile 的回调在 poll 中执行
// 发现有 setImmediate ，所以会立即跳到 check 阶段执行回调
// 再去 timer 阶段执行 setTimeout
// 所以以上输出一定是 setImmediate，setTimeout
```

上面介绍的都是 macrotask 的执行情况，microtask 会在以上每个阶段完成后立即执行。

```
setTimeout(()=>{
    console.log('timer1')

    Promise.resolve().then(function() {
        console.log('promise1')
    })
}, 0)

setTimeout(()=>{
    console.log('timer2')

    Promise.resolve().then(function() {
        console.log('promise2')
    })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中打印 timer1, promise1, timer2, promise2
// node 中打印 timer1, timer2, promise1, promise2
```

Node 中的 process.nextTick 会先于其他 microtask 执行。

```
setTimeout(() => {
 console.log("timer1");

 Promise.resolve().then(function() {
   console.log("promise1");
 });
}, 0);

process.nextTick(() => {
```

```
  console.log("nextTick");
});
// nextTick, timer1, promise1
```

存储

cookie，localStorage，sessionStorage，indexDB

特性	cookie	localStorage	sessionStorage	indexDB
数据生命周期	一般由服务器生成，可以设置过期时间	除非被清理，否则一直存在	页面关闭就清理	除非被清理，否则一直存在
数据存储大小	4K	5M	5M	无限
与服务端通信	每次都会携带在 header 中，对于请求性能影响	不参与	不参与	不参与

从上表可以看到，cookie 已经不建议用于存储。如果没有大量数据存储需求的话，可以使用 localStorage 和 sessionStorage 。对于不怎么改变的数据尽量使用 localStorage 存储，否则可以用 sessionStorage 存储。

对于 cookie，我们还需要注意安全性。

属性	作用
value	如果用于保存用户登录态，应该将该值加密，不能使用明文的用户标识
http-only	不能通过 JS 访问 Cookie，减少 XSS 攻击
secure	只能在协议为 HTTPS 的请求中携带
same-site	规定浏览器不能在跨域请求中携带 Cookie，减少 CSRF 攻击

Service Worker

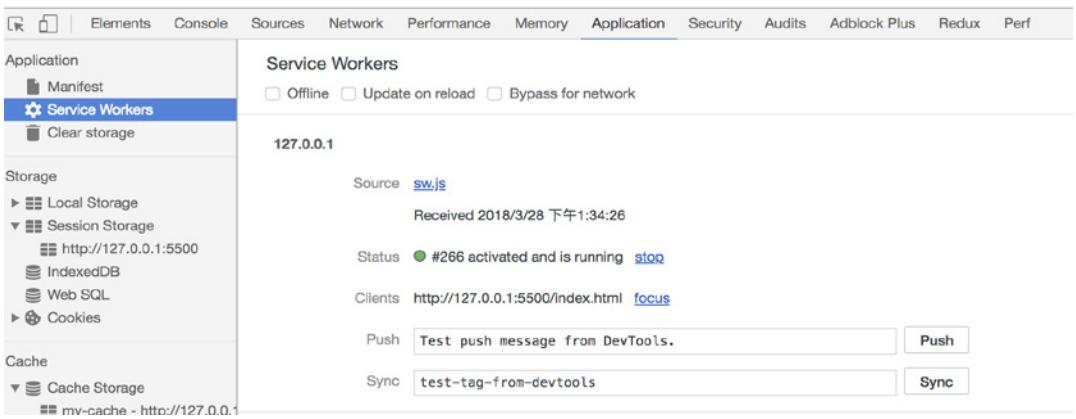
Service workers 本质上充当 Web 应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理。它们旨在（除其他之外）使得能够创建有效的离线体验，拦截网络请求并基于网络是否可用以及更新的资源是否驻留在服务器上来采取适当的动作。他们还允许访问推送通知和后台同步 API。

目前该技术通常用来做缓存文件，提高首屏速度，可以试着来实现这个功能。

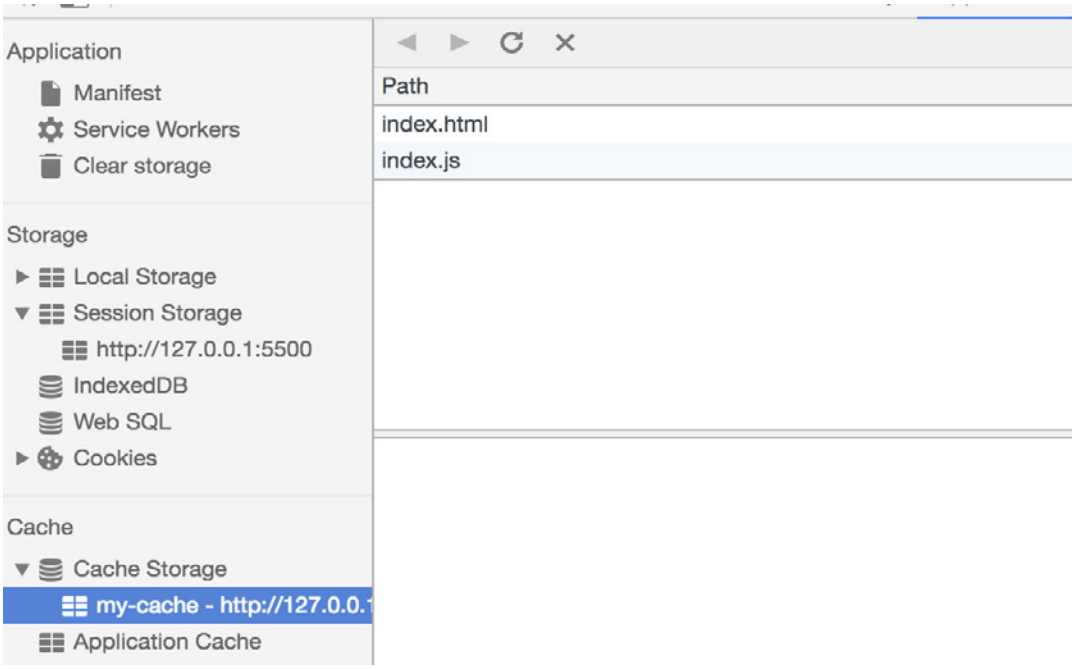
```
// index.js
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register("sw.js")
    .then(function(registration) {
      console.log("service worker 注册成功");
    })
    .catch(function(err) {
      console.log("servcie worker 注册失败");
    });
}
// sw.js
```

```
// 监听 `install` 事件，回调中缓存所需文件
self.addEventListener("install", e => {
  e.waitUntil(
    caches.open("my-cache").then(function(cache) {
      return cache.addAll(["./index.html", "./index.js"]);
    })
  );
});

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener("fetch", e => {
  e.respondWith(
    caches.match(e.request).then(function(response) {
      if (response) {
        return response;
      }
      console.log("fetch source");
    })
  );
});
```

打开页面，可以在开发者工具中的 Application 看到 Service Worker 已经启动了。



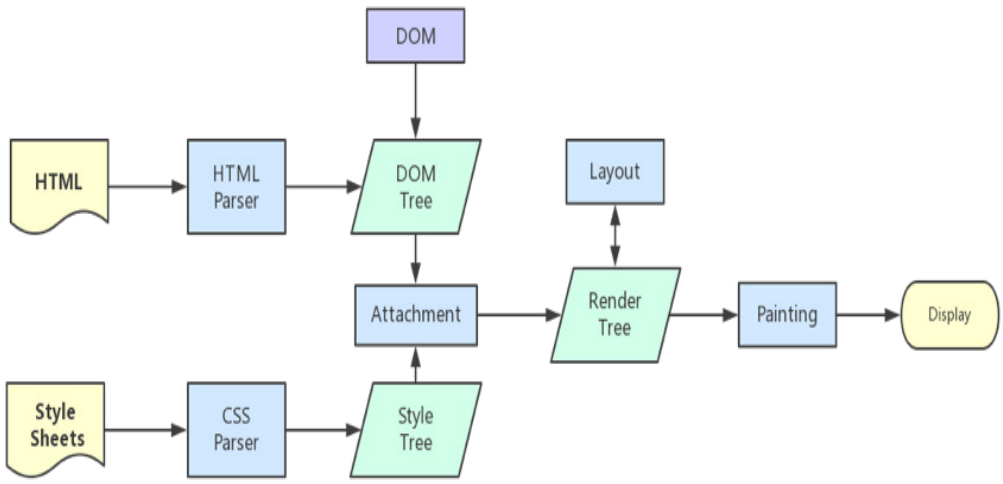
在 Cache 中也可以发现我们所需的文件已被缓存



当我们重新刷新页面可以发现我们缓存的数据是从 Service Worker 中读取的

渲染机制

浏览器的渲染机制一般分为以下几个步骤

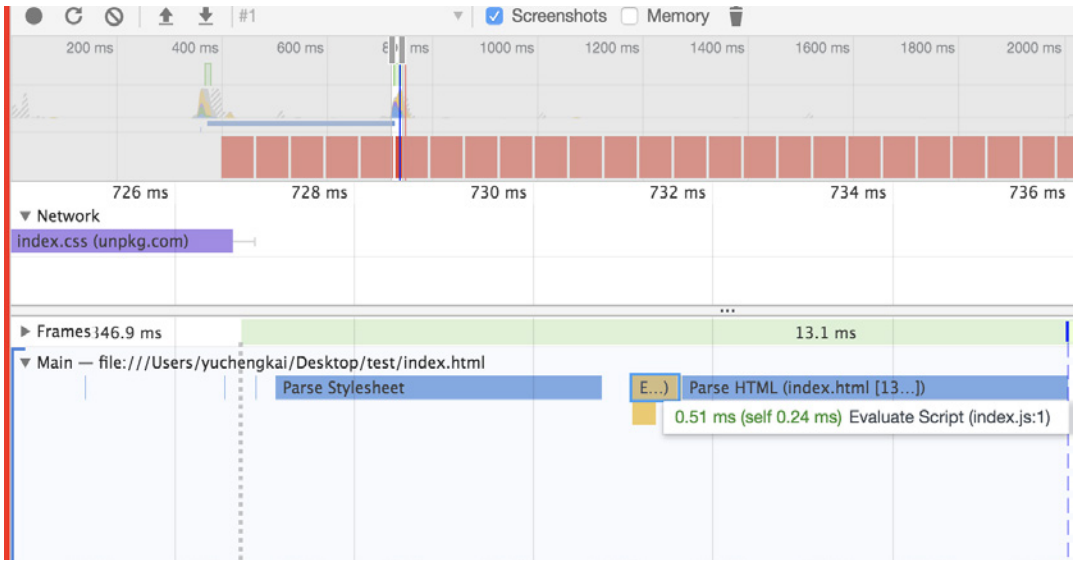


1. 处理 HTML 并构建 DOM 树。
2. 处理 CSS 构建 CSSOM 树。
3. 将 DOM 与 CSSOM 合并成一个渲染树。
4. 根据渲染树来布局，计算每个节点的位置。
5. 调用 GPU 绘制，合成图层，显示在屏幕上。

在构建 CSSOM 树时，会阻塞渲染，直至 CSSOM 树构建完成。并且构建 CSSOM 树是一个十分消耗性能的过程，所以应该尽量保证层级扁平，减少过度层叠，越是具体的 CSS 选择器，执行速度越慢。

当 HTML 解析到 script 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。也就是说，如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件。并且 CSS 也会影响 JS 的执行，只有当解析完样式表才会执行 JS，所以也可以认为这种情况下，CSS 也会暂停构建 DOM。

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <link rel="stylesheet" href="https://unpkg.com/element-ui/lib/theme-chalk/index.css">
</head>

<body>
    <script src="./index.js"></script>
    <div>123</div>
    <a>Button1111</a>
    <a href="">111</a>
```



Load 和 DOMContentLoaded 区别

Load 事件触发代表页面中的 DOM，CSS，JS，图片已经全部加载完毕。

DOMContentLoaded 事件触发代表初始的 HTML 被完全加载和解析，不需要等待 CSS，JS，图片加载。

图层

一般来说，可以把普通文档流看成一个图层。特定的属性可以生成一个新的图层。不同的图层渲染互不影响，所以对于某些频繁需要渲染的建议单独生成一个新图层，提高性能。但也不能生成过多的图层，会引起反作用。

通过以下几个常用属性可以生成新图层

- 3D 变换：translate3d、translateZ
- will-change
- video、iframe 标签
- 通过动画实现的 opacity 动画转换
- position: fixed

重绘（Repaint）和回流（Reflow）

重绘和回流是渲染步骤中的一小节，但是这两个步骤对于性能影响很大。

重绘是当节点需要更改外观而不会影响布局的，比如改变 color 就叫称为重绘

回流是布局或者几何属性需要改变就称为回流。

回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变深层次的节点很可能导致父节点的一系列回流。

所以以下几个动作可能会导致性能问题：

- 改变 window 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型

很多人不知道的是，重绘和回流其实和 Event loop 有关。

1. 当 Event loop 执行完 Microtasks 后，会判断 document 是否需要更新。因为浏览器是 60Hz 的刷新率，每 16ms 才会更新一次。
2. 然后判断是否有 resize 或者 scroll ，有的话会去触发事件，所以 resize 和 scroll 事件也是至少 16ms 才会触发一次，并且自带节流功能。
3. 判断是否触发了 media query
4. 更新动画并且发送事件
5. 判断是否有全屏操作事件
6. 执行 requestAnimationFrame 回调
7. 执行 IntersectionObserver 回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好
8. 更新界面

9. 以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 requestIdleCallback 回调。

以上内容来自于 HTML 文档

减少重绘和回流

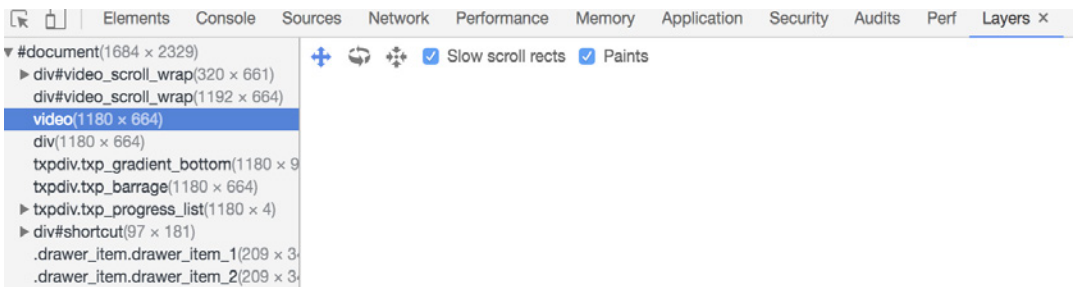
- 使用 translate 替代 top

```
<div class="test"></div>
<style>
  .test {
         position: absolute;
         top: 10px;
         width: 100px;
         height: 100px;
         background: red;
  }
</style>
<script>
  setTimeout(() => {
       // 引起回流
         document.querySelector('.test').style.top = '100px'
  }, 1000)
</script>
```

- 使用 visibility 替换 display: none ，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 把 DOM 离线后修改，比如：先把 DOM 给 display:none (有一次 Reflow)，然后你修改100次，然后再把它显示出来
- 不要把 DOM 结点的属性值放在一个循环里当成循环里的变量

```
for(let i = 0; i < 1000; i++) {
    // 获取 offsetTop 会导致回流，因为需要去获取正确的值
    console.log(document.querySelector('.test').style.offsetTop)
}
```

- 不要使用 table 布局，可能很小的一个小改动会造成整个 table 的重新布局
- 动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 requestAnimationFrame
- CSS 选择符从右往左匹配查找，避免 DOM 深度过深
- 将频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于 video 标签，浏览器会自动将该节点变为图层。



性能

网络相关

DNS 预解析

DNS 解析也是需要时间的，可以通过预解析的方式来预先获得域名所对应的 IP。

```
<link rel="dns-prefetch" href="//yuchengkai.cn">
```

缓存

缓存对于前端性能优化来说是个很重要的点，良好的缓存策略可以降低资源的重复加载提高网页的整体加载速度。

通常浏览器缓存策略分为两种：强缓存和协商缓存。

强缓存

实现强缓存可以通过两种响应头实现：Expires 和 Cache-Control 。强缓存表示在缓存期间不需要请求，state code 为 200

```
Expires: Wed, 22 Oct 2018 08:41:00 GMT
```

Expires 是 HTTP / 1.0 的产物，表示资源会在 Wed, 22 Oct 2018 08:41:00 GMT 后过期，需要再次请求。并且 Expires 受限于本地时间，如果修改了本地时间，可能会造成缓存失效。

```
Cache-control: max-age=30
```

Cache-Control 出现于 HTTP / 1.1，优先级高于 Expires 。该属性表示资源会在 30 秒后过期，需要再次请求。

协商缓存

如果缓存过期了，我们就可以使用协商缓存来解决问题。协商缓存需要请求，如果

缓存有效会返回 304。

协商缓存需要客户端和服务端共同实现，和强缓存一样，也有两种实现方式。

Last-Modified 和 If-Modified-Since

Last-Modified 表示本地文件最后修改日期，If-Modified-Since 会将 Last-Modified 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来。

但是如果在本地打开缓存文件，就会造成 Last-Modified 被修改，所以在 HTTP / 1.1 出现了 ETag 。

ETag 和 If-None-Match

ETag 类似于文件指纹，If-None-Match 会将当前 ETag 发送给服务器，询问该资源 ETag 是否变动，有变动的话就将新的资源发送回来。并且 ETag 优先级比 Last-Modified 高。

选择合适的缓存策略

对于大部分的场景都可以使用强缓存配合协商缓存解决，但是在一些特殊的地方可能需要选择特殊的缓存策略：

- 对于某些不需要缓存的资源，可以使用 Cache-control: no-store ，表示该资源不需要缓存；
- 对于频繁变动的资源，可以使用 Cache-Control: no-cache 并配合 ETag 使用，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新；
- 对于代码文件来说，通常使用 Cache-Control: max-age=31536000 并配合策略缓存使用，然后对文件进行指纹处理，一旦文件名变动就会立刻下载新的文件。

使用 HTTP / 2.0

因为浏览器会有并发请求限制，在 HTTP / 1.1 时代，每个请求都需要建立和断开，消耗了好几个 RTT 时间，并且由于 TCP 慢启动的原因，加载体积大的文件会需要更多的时间。

在 HTTP / 2.0 中引入了多路复用，能够让多个请求使用同一个 TCP 链接，极大的加快了网页的加载速度。并且还支持 Header 压缩，进一步的减少了请求的数据大小。

更详细的内容你可以查看 该小节

预加载

在开发中，可能会遇到这样的情况。有些资源不需要马上用到，但是希望尽早获取，这时候就可以使用预加载。

预加载其实是声明式的 fetch ，强制浏览器请求资源，并且不会阻塞 onload 事件，可以使用以下代码开启预加载

```
<link rel="preload" href="http://example.com">
```

预加载可以一定程度上降低首屏的加载时间，因为可以将一些不影响首屏但重要的文件延后加载，唯一缺点就是兼容性不好。

预渲染

可以通过预渲染将下载的文件预先在后台渲染，可以使用以下代码开启预渲染

```
<link rel="prerender" href="http://example.com">
```

预渲染虽然可以提高页面的加载速度，但是要确保该页面百分百会被用户在之后打开，否则就白白浪费资源去渲染

优化渲染过程

对于代码层面的优化，你可以查阅浏览器系列中的 相关内容。

懒执行

懒执行就是将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒。

懒加载

懒加载就是将不关键的资源延后加载。

懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西。对于图片来说，先设置图片标签的 src 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 src 属性，这样图片就会去下载资源，实现了图片懒加载。

懒加载不仅可以用于图片，也可以使用在别的资源上。比如进入可视区域才开始播放视频等等。

文件优化

图片优化

计算图片大小

对于一张 100 * 100 像素的图片来说，图像上有 10000 个像素点，如果每个像素的值是 RGBA 存储的话，那么也就是说每个像素有 4 个通道，每个通道 1 个字节（8 位 = 1 个字节），所以该图片大小大概为 39KB（10000 * 1 * 4 / 1024）。

但是在实际项目中，一张图片可能并不需要使用那么多颜色去显示，我们可以通过减少每个像素的调色板来相应缩小图片的大小。

了解了如何计算图片大小的知识，那么对于如何优化图片，想必大家已经有 2 个思路了：

- 减少像素点
- 减少每个像素点能够显示的颜色

图片加载优化

1. 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 CSS 去代替。
2. 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 CDN 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。
3. 小图使用 base64 格式

4. 将多个图标文件整合到一张图片中（雪碧图）
5. 选择正确的图片格式：
    - 对于能够显示 WebP 格式的浏览器尽量使用 WebP 格式。因为 WebP 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
    - 小图使用 PNG，其实对于大部分图标这类图片，完全可以使用 SVG 代替
    - 照片使用 JPEG

其他文件优化

- CSS 文件放在 head 中
- 服务端开启文件压缩功能
- 将 script 标签放在 body 底部，因为 JS 文件执行会阻塞渲染。当然也可以把 script 标签放在任意位置然后加上 defer ，表示该文件会并行下载，但是会放到 HTML 解析完成后顺序执行。对于没有任何依赖的 JS 文件可以加上 async ，表示加载和渲染后续文档元素的过程将和 JS 文件的加载与执行并行无序进行。
- 执行 JS 代码过长会卡住渲染，对于需要很多时间计算的代码可以考虑使用 Webworker。Webworker 可以让我们另开一个线程执行脚本而不影响渲染。

CDN

静态资源尽量使用 CDN 加载，由于浏览器对于单个域名有并发请求上限，可以考虑使用多个 CDN 域名。对于 CDN 加载静态资源需要注意 CDN 域名要与主站不同，否则每次请求都会带上主站的 Cookie。

其他

使用 Webpack 优化项目

- 对于 Webpack4，打包项目使用 production 模式，这样会自动开启代码压缩
- 使用 ES6 模块来开启 tree shaking，这个技术可以移除没有使用的代码
- 优化图片，对于小图可以使用 base64 的方式写入文件中
- 按照路由拆分代码，实现按需加载
- 给打包出来的文件名添加哈希，实现浏览器缓存文件

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

但是要注意线上运行的代码都是压缩过的，需要在打包时生成 sourceMap 文件便于 debug。

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。

面试题

如何渲染几万条数据并不卡住界面

这道题考察了如何在不卡住页面的情况下渲染数据，也就是说不能一次性将几万条都渲染出来，而应该一次渲染部分 DOM，那么就可以通过 requestAnimationFrame 来每 16 ms 刷新一次。

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <ul>控件</ul>
  <script>
    setTimeout(() => {
      // 插入十万条数据
      const total = 100000
      // 一次插入 20 条，如果觉得性能不好就减少
      const once = 20
      // 渲染数据总共需要几次
      const loopCount = total / once
      let countOfRender = 0
      let ul = document.querySelector("ul");
      function add() {
        // 优化性能，插入不会造成回流
        const fragment = document.createDocumentFragment();
        for (let i = 0; i < once; i++) {
          const li = document.createElement("li");
          li.innerText = Math.floor(Math.random() * total);
          fragment.appendChild(li);
        }
        ul.appendChild(fragment);
        countOfRender += 1;
        loop();
      }
```

```
      function loop() {
        if (countOfRender < loopCount) {
          window.requestAnimationFrame(add);
        }
      }
      loop();
    }, 0);
  </script>
</body>
</html>
```

安全

XSS

跨网站指令码（英语：Cross-site scripting，通常简称为：XSS）是一种网站应用程式的安全漏洞攻击，是代码注入的一种。它允许恶意使用者将程式码注入到网页上，其他使用者在观看网页时就会受到影响。这类攻击通常包含了 HTML 以及使用者端脚本语言。

XSS 分为三种：反射型，存储型和 DOM-based

如何攻击

XSS 通过修改 HTML 节点或者执行 JS 代码来攻击网站。

例如通过 URL 获取某些参数

```
<!-- http://www.domain.com?name=<script>alert(1)</script> -->
<div>{{name}}</div>
```

上述 URL 输入可能会将 HTML 改为 <script>alert(1)</script> ，这样页面中就凭空多了一段可执行脚本。这种攻击类型是反射型攻击，也可以说是 DOM-based 攻击。

也有另一种场景，比如写了一篇包含攻击代码 <script>alert(1)</script> 的文章，那么可能浏览文章的用户都会被攻击到。这种攻击类型是存储型攻击，也可以说是 DOM-based 攻击，并且这种攻击打击面更广。

如何防御

最普遍的做法是转义输入输出的内容，对于引号，尖括号，斜杠进行转义

```
function escape(str) {
```

```
    str = str.replace(/&/g, "&amp;");
    str = str.replace(/</g, "&lt;");
    str = str.replace(/>/g, "&gt;");
    str = str.replace(/"/g, "&quto;");
    str = str.replace(/'/g, "&##39;");
    str = str.replace(/`/g, "&##96;");
      str = str.replace(/\//g, "&##x2F;");
      return str
  }
```

通过转义可以将攻击代码 <script>alert(1)</script> 变成

```
// -> &lt;script&gt;alert(1)&lt;&##x2F;script&gt;
escape('<script>alert(1)</script>')
```

对于显示富文本来说，不能通过上面的办法来转义所有字符，因为这样会把需要的格式也过滤掉。这种情况通常采用白名单过滤的办法，当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式。

```
var xss = require("xss");
var html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>');
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html);
```

以上示例使用了 js-xss 来实现。可以看到在输出中保留了 h1 标签且过滤了 script 标签

CSP

内容安全策略 (CSP) 是一个额外的安全层，用于检测并削弱某些特定类型的攻击，包括跨站脚本 (XSS) 和数据注入攻击等。无论是数据盗取、网站内容污染还是散发恶意软件，这些攻击都是主要的手段。

我们可以通过 CSP 来尽量减少 XSS 攻击。CSP 本质上也是建立白名单，规定了浏览器只能够执行特定来源的代码。

通常可以通过 HTTP Header 中的 Content-Security-Policy 来开启 CSP

只允许加载本站资源

```
Content-Security-Policy: default-src 'self'
```

只允许加载 HTTPS 协议图片

```
Content-Security-Policy: img-src https://*
```

允许加载任何来源框架

```
Content-Security-Policy: child-src 'none'
```

更多属性可以查看 这里

CSRF

跨站请求伪造（英语：Cross-site request forgery），也被称为 one-click attack 或者 session riding，通常缩写为 CSRF 或者 XSRF， 是一种挟制用户在当前已登录的 Web 应用程序上执行非本意的操作的攻击方法。[1] 跟跨網站指令碼（XSS）相比，XSS 利用的是用户对指定网站的信任，CSRF 利用的是网站对用户网页浏览器的

信任。

简单点说，CSRF 就是利用用户的登录态发起恶意请求。

如何攻击

假设网站中有一个通过 Get 请求提交用户评论的接口，那么攻击者就可以在钓鱼网站中加入一个图片，图片的地址就是评论接口

```
<img src="http://www.domain.com/xxx?comment='attack'"/>
```

如果接口是 Post 提交的，就相对麻烦点，需要用表单来提交接口

```
<form action="http://www.domain.com/xxx" id="CSRF" method="post">
    <input name="comment" value="attack" type="hidden">
</form>
```

如何防御

防范 CSRF 可以遵循以下几种规则：

1. Get 请求不对数据进行修改
2. 不让第三方网站访问到用户 Cookie
3. 阻止第三方网站请求接口
4. 请求时附带验证信息，比如验证码或者 token

SameSite

可以对 Cookie 设置 SameSite 属性。该属性设置 Cookie 不随着跨域请求发送，该属性可以很大程度减少 CSRF 的攻击，但是该属性目前并不是所有浏览器都兼容。

验证 Referer

对于需要防范 CSRF 的请求，我们可以通过验证 Referer 来判断该请求是否为第三方网站发起的。

Token

服务器下发一个随机 Token（算法不能复杂），每次发起请求时将 Token 携带上，服务器验证 Token 是否有效。

密码安全

密码安全虽然大多是后端的事情，但是作为一名优秀的前端程序员也需要熟悉这方面的知识。

加盐

对于密码存储来说，必然是不能明文存储在数据库中的，否则一旦数据库泄露，会对用户造成很大的损失。并且不建议只对密码单纯通过加密算法加密，因为存在彩虹表的关系。

通常需要对密码加盐，然后进行几次不同加密算法的加密。

```
// 加盐也就是给原密码添加字符串，增加原密码长度
sha256(sha1(md5(salt + password + slat)))
```

但是加盐并不能阻止别人盗取账号，只能确保即使数据库泄露，也不会暴露用户的真实密码。一旦攻击者得到了用户的账号，可以通过暴力破解的方式破解密码。对于这种情况，通常使用验证码增加延时或者限制尝试次数的方式。并且一旦用户输入了错误

的密码，也不能直接提示用户输错密码，而应该提示账号或密码错误。

前端加密

虽然前端加密对于安全防护来说意义不大，但是在遇到中间人攻击的情况下，可以避免明文密码被第三方获取。

框架通识

MVVM

MVVM 由以下三个内容组成

- View：界面
- Model：数据模型
- ViewModel：作为桥梁负责沟通 View 和 Model

在 JQuery 时期，如果需要刷新 UI 时，需要先取到对应的 DOM 再更新 UI，这样数据和业务的逻辑就和页面有强耦合。

在 MVVM 中，UI 是通过数据驱动的，数据一旦改变就会相应的刷新对应的 UI，UI 如果改变，也会改变对应的数据。这种方式就可以在业务处理中只关心数据的流转，而无需直接和页面打交道。ViewModel 只关心数据和业务的处理，不关心 View 如何处理数据，在这种情况下，View 和 Model 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 ViewModel 中，让多个 View 复用这个 ViewModel。

在 MVVM 中，最核心的也就是数据双向绑定，例如 Angluar 的脏数据检测，Vue 中的数据劫持。

脏数据检测

当触发了指定事件后会进入脏数据检测，这时会调用 $digest 循环遍历所有的数据观察者，判断当前值是否和先前的值有区别，如果检测到变化的话，会调用 $watch 函数，然后再次调用 $digest 循环直到发现没有变化。循环至少为二次 ，至多为十次。

脏数据检测虽然存在低效的问题，但是不关心数据是通过什么方式改变的，都可以完成任务，但是这在 Vue 中的双向绑定是存在问题的。并且脏数据检测可以实现批量检测出更新的值，再去统一更新 UI，大大减少了操作 DOM 的次数。所以低效也是相对的，这就仁者见仁智者见智了。

数据劫持

Vue 内部使用了 Obeject.defineProperty() 来实现双向绑定，通过这个函数可以监听到 set 和 get 的事件。

```
var data = { name: 'yck' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

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

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

以上代码简单的实现了如何监听数据的 set 和 get 的事件，但是仅仅如此是不够的，还需要在适当的时候给属性添加发布订阅

```
<div>
    {{name}}
</div>
```

在解析如上模板代码时，遇到 {{name}} 就会给属性 name 添加发布订阅。

```
// 通过 Dep 解耦
class Dep {
  constructor() {
    this.subs = []
  }
  addSub(sub) {
    // sub 是 Watcher 实例
    this.subs.push(sub)
  }
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null

function update(value) {
  document.querySelector('div').innerText = value
}

class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
var data = { name: 'yck' }
observe(data)
```

```
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'
接下来,对 defineReactive 函数进行改造

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
      // 执行 watcher 的 update 方法
      dp.notify()
    }
  })
}
```

以上实现了一个简易的双向绑定，核心思路就是手动触发一次属性的 getter 来实现发布订阅的添加。

Proxy 与 Obeject.defineProperty 对比

Obeject.defineProperty 虽然已经能够实现双向绑定了，但是他还是有缺陷的。

只能对属性进行数据劫持，所以需要深度遍历整个对象

对于数组不能监听到数据的变化

虽然 Vue 中确实能检测到数组数据的变化，但是其实是使用了 hack 的办法，并且也是有缺陷的。

```
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// hack 以下几个函数
const methodsToPatch = [
  'push',
  'pop',
```

```
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 获得原生函数
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    // 调用原生函数
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // 触发更新
    ob.dep.notify()
    return result
  })
})
```

反观 Proxy 就没以上的问题，原生支持监听数组变化，并且可以直接对整个对象进行拦截，所以 Vue 也将在下个大版本中使用 Proxy 替换 Obeject.defineProperty

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver);
    },
    set(target, property, value, receiver) {
      setBind(value);
      return Reflect.set(target, property, value);
    }
  };
  return new Proxy(obj, handler);
};

let obj = { a: 1 }
let value
let p = onWatch(obj, (v) => {
  value = v
}, (target, property) => {
  console.log(`Get '${property}' = ${target[property]}`);
})
```

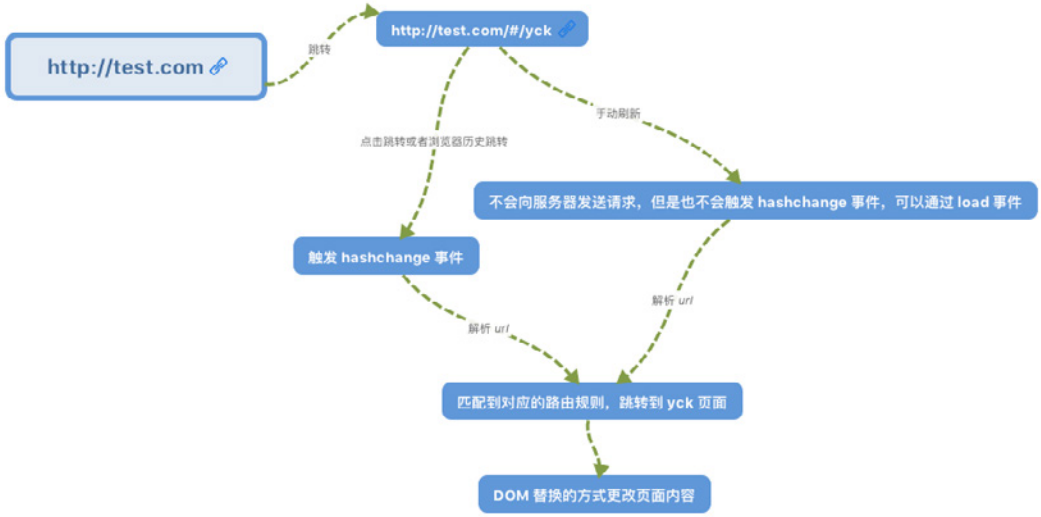
```
p.a = 2 // bind `value` to `2`
p.a // -> Get 'a' = 2
```

路由原理

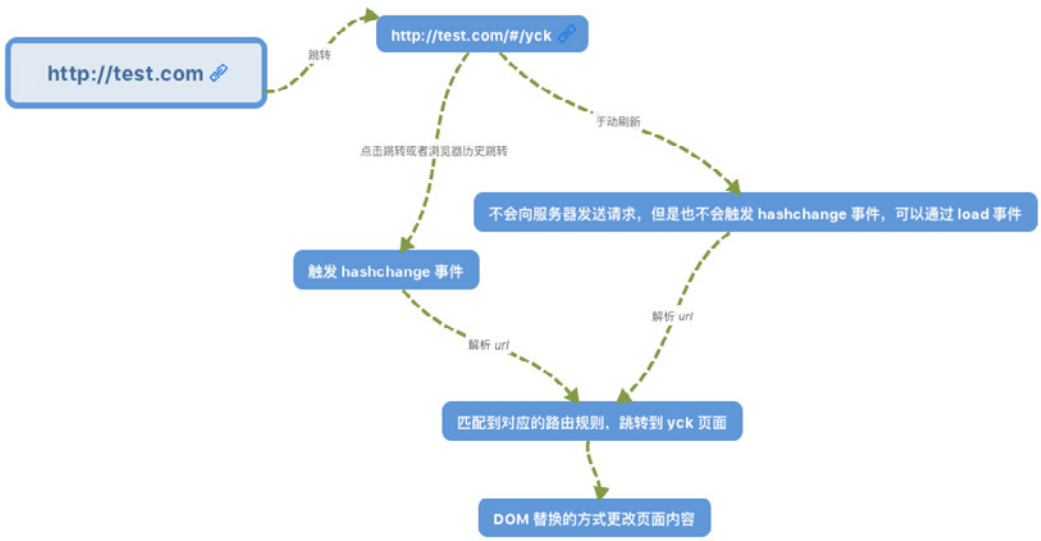
前端路由实现起来其实很简单，本质就是监听 URL 的变化，然后匹配路由规则，显示相应的页面，并且无须刷新。目前单页面使用的路由就只有两种实现方式

- hash 模式
- history 模式

www.test.com/##/ 就是 Hash URL，当 ## 后面的哈希值发生变化时，不会向服务器请求数据，可以通过 hashchange 事件来监听到 URL 的变化，从而进行跳转页面。



History 模式是 HTML5 新推出的功能，比之 Hash URL 更加美观。



Virtual Dom

代码地址

为什么需要 Virtual Dom

众所周知，操作 DOM 是很耗费性能的一件事情，既然如此，我们可以考虑通过 JS 对象来模拟 DOM 对象，毕竟操作 JS 对象比操作 DOM 省时的多。

举个例子

```
// 假设这里模拟一个 ul，其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

从上述例子中，我们一眼就可以看出先前的 ul 中的第三个 li 被移除了，四五替换了位置。

如果以上操作对应到 DOM 中，那么就是以下代码

```
// 删除第三个 li
ul.childNodes[2].remove()
// 将第四个 li 和第五个交换位置
let fromNode = ul.childNodes[4]
let toNode = node.childNodes[3]
let cloneFromNode = fromNode.cloneNode(true)
let cloenToNode = toNode.cloneNode(true)
ul.replaceChild(cloneFromNode, toNode)
ul.replaceChild(cloenToNode, fromNode)
```

当然在实际操作中，我们还需要给每个节点一个标识，作为判断是同一个节点的依据。所以这也是 Vue 和 React 中官方推荐列表里的节点使用唯一的 key 来保证性能。

那么既然 DOM 对象可以通过 JS 对象来模拟，反之也可以通过 JS 对象来渲染出对应的 DOM

以下是一个 JS 对象模拟 DOM 对象的简单实现

```
export default class Element {
  /**
   * @param {String} tag 'div'
   * @param {Object} props { class: 'item' }
   * @param {Array} children [ Element1, 'text']
   * @param {String} key option
   */
  constructor(tag, props, children, key) {
    this.tag = tag
    this.props = props
    if (Array.isArray(children)) {
      this.children = children
    } else if (isString(children)) {
      this.key = children
```

```
    this.children = null
  }
  if (key) this.key = key
}
// 渲染
render() {
  let root = this._createElement(
    this.tag,
    this.props,
    this.children,
    this.key
  )
  document.body.appendChild(root)
  return root
}
create() {
  return this._createElement(this.tag, this.props, this.children, this.key)
}
// 创建节点
_createElement(tag, props, child, key) {
  // 通过 tag 创建节点
  let el = document.createElement(tag)
  // 设置节点属性
  for (const key in props) {
    if (props.hasOwnProperty(key)) {
      const value = props[key]
      el.setAttribute(key, value)
    }
  }
  if (key) {
    el.setAttribute('key', key)
  }
  // 递归添加子节点
  if (child) {
    child.forEach(element => {
      let child
      if (element instanceof Element) {
        child = this._createElement(
          element.tag,
          element.props,
          element.children,
          element.key
        )
      } else {
```

```
            child = document.createTextNode(element)
          }
          el.appendChild(child)
        })
      }
      return el
    }
  }
```

Virtual Dom 算法简述

既然我们已经通过 JS 来模拟实现了 DOM，那么接下来的难点就在于如何判断旧的对象和新的对象之间的差异。

DOM 是多叉树的结构，如果需要完整的对比两颗树的差异，那么需要的时间复杂度会是 O(n ^ 3)，这个复杂度肯定是不能接受的。于是 React 团队优化了算法，实现了 O(n) 的复杂度来对比差异。

实现 O(n) 复杂度的关键就是只对比同层的节点，而不是跨层对比，这也是考虑到在实际业务中很少会去跨层的移动 DOM 元素。

所以判断差异的算法就分为了两步

- 首先从上至下，从左往右遍历对象，也就是树的深度遍历，这一步中会给每个节点添加索引，便于最后渲染差异
- 一旦节点有子元素，就去判断子元素是否有不同

Virtual Dom 算法实现

树的递归

首先我们来实现树的递归算法，在实现该算法前，先来考虑下两个节点对比会有几种情况

新的节点的 tagName 或者 key 和旧的不同，这种情况代表需要替换旧的节点，并且也不再需要遍历新旧节点的子元素了，因为整个旧节点都被删掉了

新的节点的 tagName 和 key（可能都没有）和旧的相同，开始遍历子树

没有新的节点，那么什么都不用做

```
import { StateEnums, isString, move } from './util'
import Element from './element'

export default function diff(oldDomTree, newDomTree) {
  // 用于记录差异
  let pathchs = {}
  // 一开始的索引为 0
  dfs(oldDomTree, newDomTree, 0, pathchs)
  return pathchs
}

function dfs(oldNode, newNode, index, patches) {
  // 用于保存子树的更改
  let curPatches = []
  // 需要判断三种情况
```

```
  // 1.没有新的节点，那么什么都不用做
  // 2.新的节点的 tagName 和 `key` 和旧的不同，就替换
  // 3.新的节点的 tagName 和 key（可能都没有） 和旧的相同，开始遍历子树
  if (!newNode) {
  } else if (newNode.tag === oldNode.tag && newNode.key === oldNode.key) {
    // 判断属性是否变更
    let props = diffProps(oldNode.props, newNode.props)
    if (props.length) curPatches.push({ type: StateEnums.ChangeProps, props })
    // 遍历子树
    diffChildren(oldNode.children, newNode.children, index, patches)
  } else {
    // 节点不同，需要替换
    curPatches.push({ type: StateEnums.Replace, node: newNode })
  }

  if (curPatches.length) {
    if (patches[index]) {
      patches[index] = patches[index].concat(curPatches)
    } else {
      patches[index] = curPatches
    }
  }
}
```

判断属性的更改

判断属性的更改也分三个步骤

1. 遍历旧的属性列表，查看每个属性是否还存在于新的属性列表中
2. 遍历新的属性列表，判断两个列表中都存在的属性的值是否有变化
3. 在第二步中同时查看是否有属性不存在与旧的属性列列表中

```
function diffProps(oldProps, newProps) {
  // 判断 Props 分以下三步骤
  // 先遍历 oldProps 查看是否存在删除的属性
  // 然后遍历 newProps 查看是否有属性值被修改
  // 最后查看是否有属性新增
  let change = []
  for (const key in oldProps) {
    if (oldProps.hasOwnProperty(key) && !newProps[key]) {
      change.push({
        prop: key
      })
    }
  }
  for (const key in newProps) {
    if (newProps.hasOwnProperty(key)) {
      const prop = newProps[key]
      if (oldProps[key] && oldProps[key] !== newProps[key]) {
        change.push({
```

```
            prop: key,
            value: newProps[key]
          })
        } else if (!oldProps[key]) {
          change.push({
            prop: key,
            value: newProps[key]
          })
        }
      }
    }
    return change
  }
```

判断列表差异算法实现

这个算法是整个 Virtual Dom 中最核心的算法，且让我一一为你道来。 这里的主要步骤其实和判断属性差异是类似的，也是分为三步

1. 遍历旧的节点列表，查看每个节点是否还存在于新的节点列表中
2. 遍历新的节点列表，判断是否有新的节点
3. 在第二步中同时判断节点是否有移动

PS：该算法只对有 key 的节点做处理

```
function listDiff(oldList, newList, index, patches) {
  // 为了遍历方便，先取出两个 list 的所有 keys
  let oldKeys = getKeys(oldList)
  let newKeys = getKeys(newList)
  let changes = []

  // 用于保存变更后的节点数据
  // 使用该数组保存有以下好处
  // 1.可以正确获得被删除节点索引
  // 2.交换节点位置只需要操作一遍 DOM
  // 3.用于 `diffChildren` 函数中的判断，只需要遍历
  // 两个树中都存在的节点，而对于新增或者删除的节点来说，完全没必要
  // 再去判断一遍
  let list = []
  oldList &&
    oldList.forEach(item => {
      let key = item.key
      if (isString(item)) {
        key = item
      }
      // 寻找新的 children 中是否含有当前节点
      // 没有的话需要删除
      let index = newKeys.indexOf(key)
```

```
    if (index === -1) {
      list.push(null)
    } else list.push(key)
  })
// 遍历变更后的数组
let length = list.length
// 因为删除数组元素是会更改索引的
// 所有从后往前删可以保证索引不变
for (let i = length - 1; i >= 0; i--) {
  // 判断当前元素是否为空，为空表示需要删除
  if (!list[i]) {
    list.splice(i, 1)
    changes.push({
      type: StateEnums.Remove,
      index: i
    })
  }
}
// 遍历新的 list，判断是否有节点新增或移动
// 同时也对 `list` 做节点新增和移动节点的操作
newList &&
  newList.forEach((item, i) => {
    let key = item.key
    if (isString(item)) {
      key = item
    }
    // 寻找旧的 children 中是否含有当前节点
    let index = list.indexOf(key)
    // 没找到代表新节点，需要插入
    if (index === -1 || key == null) {
      changes.push({
        type: StateEnums.Insert,
        node: item,
        index: i
      })
      list.splice(i, 0, key)
    } else {
      // 找到了，需要判断是否需要移动
      if (index !== i) {
        changes.push({
          type: StateEnums.Move,
          from: index,
          to: i
        })
```

```
            move(list, index, i)
          }
        }
      })
    return { changes, list }
  }

  function getKeys(list) {
    let keys = []
    let text
    list &&
      list.forEach(item => {
        let key
        if (isString(item)) {
          key = [item]
        } else if (item instanceof Element) {
          key = item.key
        }
        keys.push(key)
      })
    return keys
  }
```

遍历子元素打标识

对于这个函数来说，主要功能就两个

1. 判断两个列表差异
2. 给节点打上标记

总体来说，该函数实现的功能很简单

```
function diffChildren(oldChild, newChild, index, patches) {
  let { changes, list } = listDiff(oldChild, newChild, index, patches)
  if (changes.length) {
    if (patches[index]) {
      patches[index] = patches[index].concat(changes)
    } else {
      patches[index] = changes
    }
  }
  // 记录上一个遍历过的节点
  let last = null
  oldChild &&
    oldChild.forEach((item, i) => {
      let child = item && item.children
      if (child) {
        index =
```

```
            last && last.children ? index + last.children.length + 1 : index + 1
          let keyIndex = list.indexOf(item.key)
          let node = newChild[keyIndex]
          // 只遍历新旧中都存在的节点，其他新增或者删除的没必要遍历
          if (node) {
            dfs(item, node, index, patches)
          }
        } else index += 1
        last = item
      })
  }
```

渲染差异

通过之前的算法，我们已经可以得出两个树的差异了。既然知道了差异，就需要局部去更新 DOM 了，下面就让我们来看看 Virtual Dom 算法的最后一步骤

这个函数主要两个功能

1. 深度遍历树，将需要做变更操作的取出来
2. 局部更新 DOM

整体来说这部分代码还是很好理解的

```
let index = 0
export default function patch(node, patchs) {
  let changes = patchs[index]
  let childNodes = node && node.childNodes
  // 这里的深度遍历和 diff 中是一样的
  if (!childNodes) index += 1
  if (changes && changes.length && patchs[index]) {
    changeDom(node, changes)
  }
  let last = null
  if (childNodes && childNodes.length) {
    childNodes.forEach((item, i) => {
      index =
        last && last.children ? index + last.children.length + 1 : index + 1
      patch(item, patchs)
      last = item
    })
  }
}

function changeDom(node, changes, noChild) {
  changes &&
    changes.forEach(change => {
      let { type } = change
      switch (type) {
```

```
        case StateEnums.ChangeProps:
          let { props } = change
          props.forEach(item => {
            if (item.value) {
              node.setAttribute(item.prop, item.value)
            } else {
              node.removeAttribute(item.prop)
            }
          })
          break
        case StateEnums.Remove:
          node.childNodes[change.index].remove()
          break
        case StateEnums.Insert:
          let dom
          if (isString(change.node)) {
            dom = document.createTextNode(change.node)
          } else if (change.node instanceof Element) {
            dom = change.node.create()
          }
          node.insertBefore(dom, node.childNodes[change.index])
          break
        case StateEnums.Replace:
          node.parentNode.replaceChild(change.node.create(), node)
          break
        case StateEnums.Move:
          let fromNode = node.childNodes[change.from]
          let toNode = node.childNodes[change.to]
          let cloneFromNode = fromNode.cloneNode(true)
          let cloenToNode = toNode.cloneNode(true)
          node.replaceChild(cloneFromNode, toNode)
          node.replaceChild(cloenToNode, fromNode)
          break
        default:
          break
      }
    })
}
```

最后

Virtual Dom 算法的实现也就是以下三步

1. 通过 JS 来模拟创建 DOM 对象
2. 判断两个对象的差异
3. 渲染差异

```
let test4 = new Element('div', { class: 'my-div' }, ['test4'])
let test5 = new Element('ul', { class: 'my-div' }, ['test5'])

let test1 = new Element('div', { class: 'my-div' }, [test4])

let test2 = new Element('div', { id: '11' }, [test5, test4])

let root = test1.render()

let pathchs = diff(test1, test2)
console.log(pathchs)

setTimeout(() => {
  console.log('开始更新')
  patch(root, pathchs)
  console.log('结束更新')
}, 1000)
```

当然目前的实现还略显粗糙，但是对于理解 Virtual Dom 算法来说已经是完全足够的了。

Vue

NextTick 原理分析

nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM。

在 Vue 2.4 之前都是使用的 microtasks，但是 microtasks 的优先级过高，在某些情况下可能会出现比事件冒泡更快的情况，但如果都使用 macrotasks 又可能会出现渲染的性能问题。所以在新版本中，会默认使用 microtasks，但在特殊情况下会使用 macrotasks，比如 v-on。

对于实现 macrotasks ，会先判断是否能使用 setImmediate ，不能的话降级为 MessageChannel ，以上都不行的话就使用 setTimeout

```
    if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
      macroTimerFunc = () => {
        setImmediate(flushCallbacks)
      }
    } else if (
      typeof MessageChannel !== 'undefined' &&
      (isNative(MessageChannel) ||
        // PhantomJS
        MessageChannel.toString() === '[object MessageChannelConstructor]')
    ) {
  const channel = new MessageChannel()
      const port = channel.port2
```

```
    channel.port1.onmessage = flushCallbacks
    macroTimerFunc = () => {
      port.postMessage(1)
    }
  } else {
    /* istanbul ignore next */
    macroTimerFunc = () => {
      setTimeout(flushCallbacks, 0)
    }
  }
```

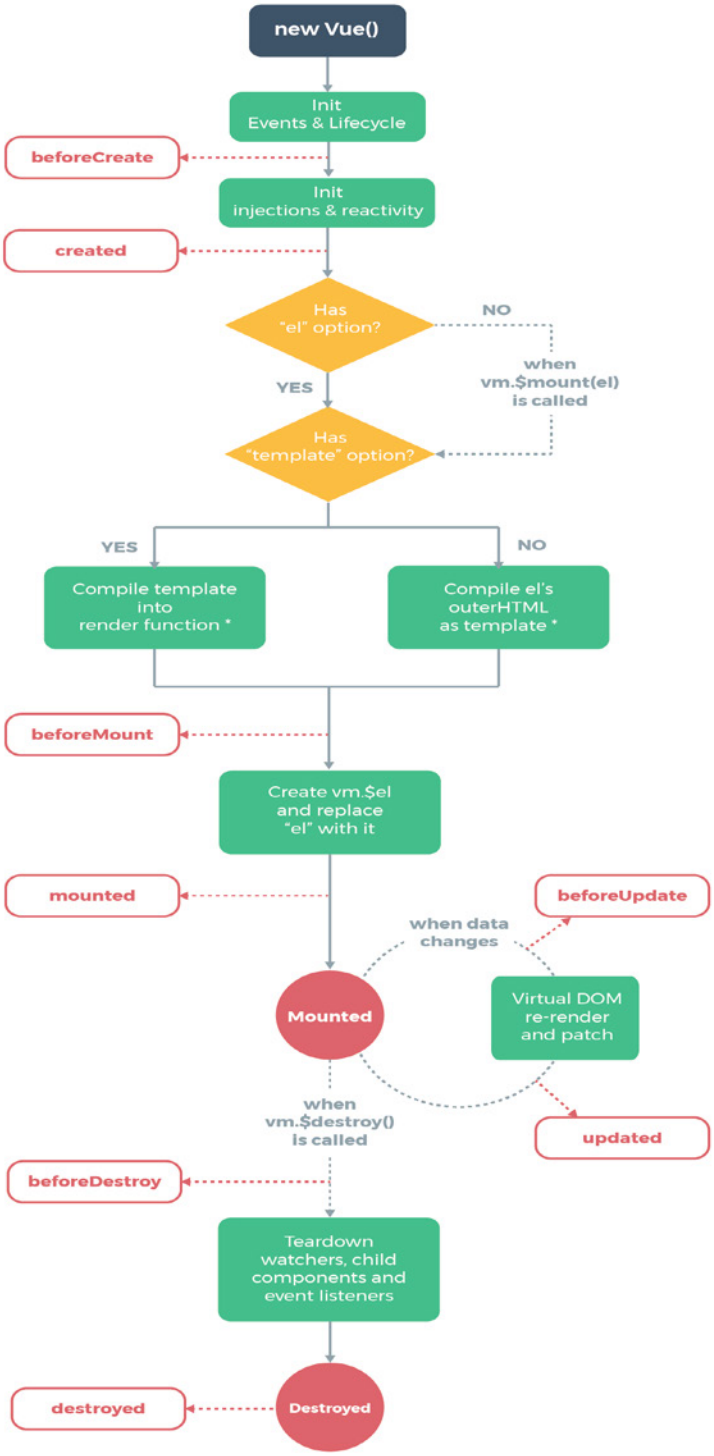
nextTick 同时也支持 Promise 的使用，会判断是否实现了 Promise

```
 export function nextTick(cb?: Function, ctx?: Object) {
    let _resolve
    // 将回调函数整合进一个数组中
    callbacks.push(() => {
      if (cb) {
        try {
          cb.call(ctx)
        } catch (e) {
          handleError(e, ctx, 'nextTick')
        }
      } else if (_resolve) {
        _resolve(ctx)
      }
    })
    if (!pending) {
      pending = true
      if (useMacroTask) {
        macroTimerFunc()
      } else {
        microTimerFunc()
      }
    }
    // 判断是否可以使用 Promise
    // 可以的话给 _resolve 赋值
    // 这样回调函数就能以 promise 的方式调用
    if (!cb && typeof Promise !== 'undefined') {
      return new Promise(resolve => {
        _resolve = resolve
      })
    }
  }
```

生命周期分析

生命周期函数就是组件在初始化或者数据更新时会触发的钩子函数。

在初始化时，会调用以下代码，生命周期就是通过 callHook 调用的



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

```
Vue.prototype._init = function(options) {
    initLifecycle(vm)
    initEvents(vm)
    initRender(vm)
    callHook(vm, 'beforeCreate') // 拿不到 props data
    initInjections(vm)
    initState(vm)
    initProvide(vm)
    callHook(vm, 'created')
}
```

可以发现在以上代码中，beforeCreate 调用的时候，是获取不到 props 或者 data 中的数据的，因为这些数据的初始化都在 initState 中。

接下来会执行挂载函数

```
export function mountComponent {
    callHook(vm, 'beforeMount')
    // ...
    if (vm.$vnode == null) {
        vm._isMounted = true
        callHook(vm, 'mounted')
    }
}
```

beforeMount 就是在挂载前执行的，然后开始创建 VDOM 并替换成真实 DOM，最后执行 mounted 钩子。这里会有个判断逻辑，如果是外部 new Vue({}) 的话，不会存在 $vnode ，所以直接执行 mounted 钩子了。如果有子组件的话，会递归挂载子组件，只有当所有子组件全部挂载完毕，才会执行根组件的挂载钩子。

接下来是数据更新时会调用的钩子函数

```
function flushSchedulerQueue() {
  // ...
  for (index = 0; index < queue.length; index++) {
    watcher = queue[index]
    if (watcher.before) {
      watcher.before() // 调用 beforeUpdate
    }
    id = watcher.id
    has[id] = null
    watcher.run()
    // in dev build, check and stop circular updates.
    if (process.env.NODE_ENV !== 'production' && has[id] != null) {
      circular[id] = (circular[id] || 0) + 1
      if (circular[id] > MAX_UPDATE_COUNT) {
        warn(
          'You may have an infinite update loop ' +
            (watcher.user
              ? `in watcher with expression "${watcher.expression}"`
```

```
                : `in a component render function.`),
              watcher.vm
            )
            break
          }
        }
      }
      callUpdatedHooks(updatedQueue)
    }

    function callUpdatedHooks(queue) {
      let i = queue.length
      while (i--) {
        const watcher = queue[i]
        const vm = watcher.vm
        if (vm._watcher === watcher && vm._isMounted) {
          callHook(vm, 'updated')
        }
      }
    }
```

上图还有两个生命周期没有说，分别为 activated 和 deactivated ，这两个钩子函数是 keep-alive 组件独有的。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 actived 钩子函数。

最后就是销毁组件的钩子函数了

```
Vue.prototype.$destroy = function() {
  // ...
  callHook(vm, 'beforeDestroy')
  vm._isBeingDestroyed = true
  // remove self from parent
  const parent = vm.$parent
  if (parent && !parent._isBeingDestroyed && !vm.$options.abstract) {
    remove(parent.$children, vm)
  }
  // teardown watchers
  if (vm._watcher) {
    vm._watcher.teardown()
  }
  let i = vm._watchers.length
  while (i--) {
    vm._watchers[i].teardown()
  }
  // remove reference from data ob
  // frozen object may not have observer.
  if (vm._data.__ob__) {
    vm._data.__ob__.vmCount--
```

```
    }
    // call the last hook...
    vm._isDestroyed = true
    // invoke destroy hooks on current rendered tree
    vm.__patch__(vm._vnode, null)
    // fire destroyed hook
    callHook(vm, 'destroyed')
    // turn off all instance listeners.
    vm.$off()
    // remove __vue__ reference
    if (vm.$el) {
      vm.$el.__vue__ = null
    }
    // release circular reference (##6759)
    if (vm.$vnode) {
      vm.$vnode.parent = null
    }
  }
```

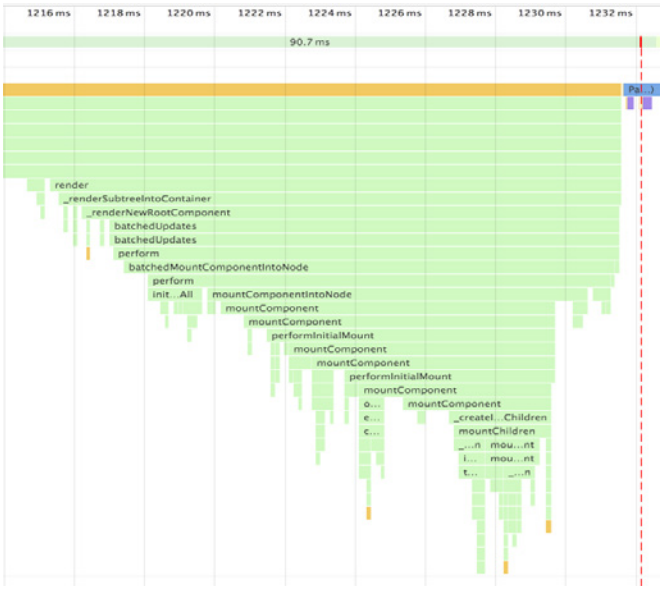
在执行销毁操作前会调用 beforeDestroy 钩子函数，然后进行一系列的销毁操作，如果有子组件的话，也会递归销毁子组件，所有子组件都销毁完毕后才会执行根组件的 destroyed 钩子函数。

React

React 生命周期分析

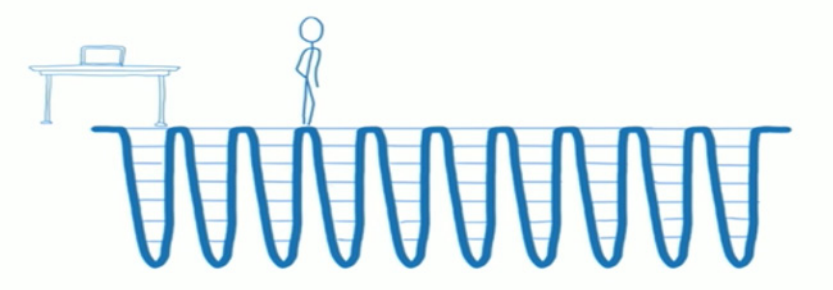
在 V16 版本中引入了 Fiber 机制。这个机制一定程度上的影响了部分生命周期的调用，并且也引入了新的 2 个 API 来解决问题。

在之前的版本中，如果你拥有一个很复杂的复合组件，然后改动了最上层组件的 state，那么调用栈可能会很长



调用栈过长，再加上中间进行了复杂的操作，就可能导致长时间阻塞主线程，带来不好的用户体验。Fiber 就是为了解决该问题而生。

Fiber 本质上是一个虚拟的堆栈帧，新的调度器会按照优先级自由调度这些帧，从而将之前的同步渲染改成了异步渲染，在不影响体验的情况下去分段计算更新。



对于如何区别优先级，React 有自己的一套逻辑。对于动画这种实时性很高的东西，也就是 16 ms 必须渲染一次保证不卡顿的情况下，React 会每 16 ms（以内） 暂停一下更新，返回来继续渲染动画。

对于异步渲染，现在渲染有两个阶段：reconciliation 和 commit 。前者过程是可以打断的，后者不能暂停，会一直更新界面直到完成。

Reconciliation 阶段

- componentWillMount
- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate

Commit 阶段

- componentDidMount
- componentDidUpdate
- componentWillUnmount

因为 reconciliation 阶段是可以被打断的，所以 reconciliation 阶段会执行的生命周期函数就可能会出现调用多次的情况，从而引起 Bug。所以对于 reconciliation 阶段调用的几个函数，除了 shouldComponentUpdate 以外，其他都应该避免去使用，并且 V16 中也引入了新的 API 来解决这个问题。

getDerivedStateFromProps 用于替换 componentWillReceiveProps ，该函数会在初始化和 update 时被调用

```
class ExampleComponent extends React.Component {
  // Initialize state in constructor,
  // Or with a property initializer.
  state = {};

  static getDerivedStateFromProps(nextProps, prevState) {
    if (prevState.someMirroredValue !== nextProps.someValue) {
```

```
        return {
          derivedData: computeDerivedState(nextProps),
          someMirroredValue: nextProps.someValue
        };
      }

      // Return null to indicate no change to state.
      return null;
    }
  }
```

getSnapshotBeforeUpdate 用于替换 componentWillUpdate ，该函数会在 update 后 DOM 更新前被调用，用于读取最新的 DOM 数据。

V16 生命周期函数用法建议

```
class ExampleComponent extends React.Component {
  // 用于初始化 state
  constructor() {}
  // 用于替换 `componentWillReceiveProps` ，该函数会在初始化和 `update` 时被调用
  // 因为该函数是静态函数，所以取不到 `this`
  // 如果需要对比 `prevProps` 需要单独在 `state` 中维护
  static getDerivedStateFromProps(nextProps, prevState) {}
  // 判断是否需要更新组件，多用于组件性能优化
  shouldComponentUpdate(nextProps, nextState) {}
  // 组件挂载后调用
  // 可以在该函数中进行请求或者订阅
  componentDidMount() {}
  // 用于获得最新的 DOM 数据
  getSnapshotBeforeUpdate() {}
  // 组件即将销毁
  // 可以在此处移除订阅，定时器等等
  componentWillUnmount() {}
  // 组件销毁后调用
  componentDidUnMount() {}
  // 组件更新后调用
  componentDidUpdate() {}
  // 渲染组件函数
  render() {}
  // 以下函数不建议使用
  UNSAFE_componentWillMount() {}
  UNSAFE_componentWillUpdate(nextProps, nextState) {}
  UNSAFE_componentWillReceiveProps(nextProps) {}
}
```

setState

setState 在 React 中是经常使用的一个 API，但是它存在一些问题，可能会导致

犯错，核心原因就是因为这个 API 是异步的。

首先 setState 的调用并不会马上引起 state 的改变，并且如果你一次调用了多个 setState ，那么结果可能并不如你期待的一样。

```
handle() {
  // 初始化 `count` 为 0
  console.log(this.state.count) // -> 0
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  console.log(this.state.count) // -> 0
}
```

第一，两次的打印都为 0，因为 setState 是个异步 API，只有同步代码运行完毕才会执行。setState 异步的原因我认为在于，setState 可能会导致 DOM 的重绘，如果调用一次就马上去进行重绘，那么调用多次就会造成不必要的性能损失。设计成异步的话，就可以将多次调用放入一个队列中，在恰当的时候统一进行更新过程。

第二，虽然调用了三次 setState ，但是 count 的值还是为 1。因为多次调用会合并为一次，只有当更新结束后 state 才会改变，三次调用等同于如下代码

```
Object.assign(
  {},
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
)
```

当然你也可以通过以下方式来实现调用三次 setState 使得 count 为 3

```
handle() {
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
}
```

如果你想在每次调用 setState 后获得正确的 state ，可以通过如下代码实现

```
handle() {
    this.setState((prevState) => ({ count: prevState.count + 1 }), () => {
        console.log(this.state)
    })
}
```

Redux 源码分析

首先让我们来看下 combineReducers 函数

```
// 传入一个 object
export default function combineReducers(reducers) {
 // 获取该 Object 的 key 值
  const reducerKeys = Object.keys(reducers)
```

```
  // 过滤后的 reducers
  const finalReducers = {}
  // 获取每一个 key 对应的 value
  // 在开发环境下判断值是否为 undefined
  // 然后将值类型是函数的值放入 finalReducers
  for (let i = 0; i < reducerKeys.length; i++) {
    const key = reducerKeys[i]

    if (process.env.NODE_ENV !== 'production') {
      if (typeof reducers[key] === 'undefined') {
        warning(`No reducer provided for key "${key}"`)
      }
    }

    if (typeof reducers[key] === 'function') {
      finalReducers[key] = reducers[key]
    }
  }
  // 拿到过滤后的 reducers 的 key 值
  const finalReducerKeys = Object.keys(finalReducers)

  // 在开发环境下判断，保存不期望 key 的缓存用以下面做警告
  let unexpectedKeyCache
  if (process.env.NODE_ENV !== 'production') {
    unexpectedKeyCache = {}
  }

  let shapeAssertionError
  try {
  // 该函数解析在下面
    assertReducerShape(finalReducers)
  } catch (e) {
    shapeAssertionError = e
  }
// combineReducers 函数返回一个函数，也就是合并后的 reducer 函数
// 该函数返回总的 state
// 并且你也可以发现这里使用了闭包，函数里面使用到了外面的一些属性
  return function combination(state = {}, action) {
    if (shapeAssertionError) {
      throw shapeAssertionError
    }
    // 该函数解析在下面
    if (process.env.NODE_ENV !== 'production') {
      const warningMessage = getUnexpectedStateShapeWarningMessage(
```

```
            state,
            finalReducers,
            action,
            unexpectedKeyCache
          )
          if (warningMessage) {
            warning(warningMessage)
          }
        }
        // state 是否改变
        let hasChanged = false
        // 改变后的 state
        const nextState = {}
        for (let i = 0; i < finalReducerKeys.length; i++) {
        // 拿到相应的 key
          const key = finalReducerKeys[i]
          // 获得 key 对应的 reducer 函数
          const reducer = finalReducers[key]
          // state 树下的 key 是与 finalReducers 下的 key 相同的
          // 所以你在 combineReducers 中传入的参数的 key 即代表了 各个 reducer 也代表了各
    个 state
          const previousStateForKey = state[key]
          // 然后执行 reducer 函数获得该 key 值对应的 state
          const nextStateForKey = reducer(previousStateForKey, action)
          // 判断 state 的值，undefined 的话就报错
          if (typeof nextStateForKey === 'undefined') {
            const errorMessage = getUndefinedStateErrorMessage(key, action)
            throw new Error(errorMessage)
          }
          // 然后将 value 塞进去
          nextState[key] = nextStateForKey
          // 如果 state 改变
          hasChanged = hasChanged || nextStateForKey !== previousStateForKey
        }
        // state 只要改变过，就返回新的 state
        return hasChanged ? nextState : state
      }
    }
```

combineReducers 函数总的来说很简单，总结来说就是接收一个对象，将参数过滤后返回一个函数。该函数里有一个过滤参数后的对象 finalReducers，遍历该对象，然后执行对象中的每一个 reducer 函数，最后将新的 state 返回。

接下来让我们来看看 combinrReducers 中用到的两个函数

```
// 这是执行的第一个用于抛错的函数
```

```
function assertReducerShape(reducers) {
// 将 combineReducers 中的参数遍历
  Object.keys(reducers).forEach(key => {
    const reducer = reducers[key]
    // 给他传入一个 action
    const initialState = reducer(undefined, { type: ActionTypes.INIT })
    // 如果得到的 state 为 undefined 就抛错
    if (typeof initialState === 'undefined') {
      throw new Error(
        `Reducer "${key}" returned undefined during initialization. ` +
          `If the state passed to the reducer is undefined, you must ` +
          `explicitly return the initial state. The initial state may ` +
         `not be undefined. If you don't want to set a value for this reducer, ` +
          `you can use null instead of undefined.`
      )
    }
    // 再过滤一次，考虑到万一你在 reducer 中给 ActionTypes.INIT 返回了值
    // 传入一个随机的 action 判断值是否为 undefined
    const type =
      '@@redux/PROBE_UNKNOWN_ACTION_' +
      Math.random()
        .toString(36)
        .substring(7)
        .split('')
        .join('.')
    if (typeof reducer(undefined, { type }) === 'undefined') {
      throw new Error(
        `Reducer "${key}" returned undefined when probed with a random type. ` +
          `Don't try to handle ${
            ActionTypes.INIT
          } or other actions in "redux/*" ` +
         `namespace. They are considered private. Instead, you must return the ` +
          `current state for any unknown actions, unless it is undefined, ` +
         `in which case you must return the initial state, regardless of the ` +
         `action type. The initial state may not be undefined, but can be null.`
      )
    }
  })
}

function getUnexpectedStateShapeWarningMessage(
  inputState,
  reducers,
  action,
```

```
  unexpectedKeyCache
) {
  // 这里的 reducers 已经是 finalReducers
  const reducerKeys = Object.keys(reducers)
  const argumentName =
    action && action.type === ActionTypes.INIT
      ? 'preloadedState argument passed to createStore'
      : 'previous state received by the reducer'

  // 如果 finalReducers 为空
  if (reducerKeys.length === 0) {
    return (
      'Store does not have a valid reducer. Make sure the argument passed ' +
      'to combineReducers is an object whose values are reducers.'
    )
  }
    // 如果你传入的 state 不是对象
  if (!isPlainObject(inputState)) {
    return (
      `The ${argumentName} has unexpected type of "` +
      {}.toString.call(inputState).match(/\s([a-z|A-Z]+)/)[1] +
      `". Expected argument to be an object with the following ` +
      `keys: "${reducerKeys.join('", "')}"`
    )
  }
    // 将参入的 state 于 finalReducers 下的 key 做比较，过滤出多余的 key
  const unexpectedKeys = Object.keys(inputState).filter(
    key => !reducers.hasOwnProperty(key) && !unexpectedKeyCache[key]
  )

  unexpectedKeys.forEach(key => {
    unexpectedKeyCache[key] = true
  })

  if (action && action.type === ActionTypes.REPLACE) return

// 如果 unexpectedKeys 有值的话
  if (unexpectedKeys.length > 0) {
    return (
      `Unexpected ${unexpectedKeys.length > 1 ? 'keys' : 'key'} ` +
      `"${unexpectedKeys.join('", "')}" found in ${argumentName}. ` +
      `Expected to find one of the known reducer keys instead: ` +
      `"${reducerKeys.join('", "')}". Unexpected keys will be ignored.`
    )
  }
}
```

接下来让我们先来看看 compose 函数

```
// 这个函数设计的很巧妙，通过传入函数引用的方式让我们完成多个函数的嵌套使用，术语叫做高阶函数
// 通过使用 reduce 函数做到从右至左调用函数
// 对于上面项目中的例子
compose(
    applyMiddleware(thunkMiddleware),
    window.devToolsExtension ? window.devToolsExtension() : f => f
)
// 经过 compose 函数变成了 applyMiddleware(thunkMiddleware)(window.
devToolsExtension()())
// 所以在找不到 window.devToolsExtension 时你应该返回一个函数
export default function compose(...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }

  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
```

然后我们来解析 createStore 函数的部分代码

```
export default function createStore(reducer, preloadedState, enhancer) {
  // 一般 preloadedState 用的少，判断类型，如果第二个参数是函数且没有第三个参数，就调换位置
  if (typeof preloadedState === 'function' && typeof enhancer === 'undefined')
{
    enhancer = preloadedState
    preloadedState = undefined
  }
  // 判断 enhancer 是否是函数
  if (typeof enhancer !== 'undefined') {
    if (typeof enhancer !== 'function') {
      throw new Error('Expected the enhancer to be a function.')
    }
    // 类型没错的话，先执行 enhancer，然后再执行 createStore 函数
    return enhancer(createStore)(reducer, preloadedState)
  }
  // 判断 reducer 是否是函数
  if (typeof reducer !== 'function') {
    throw new Error('Expected the reducer to be a function.')
  }
  // 当前 reducer
```

```
  let currentReducer = reducer
  // 当前状态
  let currentState = preloadedState
  // 当前监听函数数组
  let currentListeners = []
  // 这是一个很重要的设计，为的就是每次在遍历监听器的时候保证 currentListeners 数组不变
   // 可以考虑下只存在 currentListeners 的情况，如果我在某个 subscribe 中再次执行
subscribe
   // 或者 unsubscribe，这样会导致当前的 currentListeners 数组大小发生改变，从而可能导
致
  // 索引出错
  let nextListeners = currentListeners
  // reducer 是否正在执行
  let isDispatching = false
  // 如果 currentListeners 和 nextListeners 相同，就赋值回去
  function ensureCanMutateNextListeners() {
    if (nextListeners === currentListeners) {
      nextListeners = currentListeners.slice()
    }
  }
  // ......
}
```

接下来先来介绍 applyMiddleware 函数

在这之前我需要先来介绍一下函数柯里化，柯里化是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

```
function add(a,b) { return a + b }
add(1, 2) => 3
// 对于以上函数如果使用柯里化可以这样改造
function add(a) {
    return b => {
        return a + b
    }
}
add(1)(2) => 3
// 你可以这样理解函数柯里化，通过闭包保存了外部的一个变量，然后返回一个接收参数的函数，在该函
数中使用了保存的变量，然后再返回值。
// 这个函数应该是整个源码中最难理解的一块了
// 该函数返回一个柯里化的函数
//  所以调用这个函数应该这样写 applyMiddleware(...middlewares)(createStore)
(...args)
export default function applyMiddleware(...middlewares) {
  return createStore => (...args) => {
   // 这里执行 createStore 函数，把 applyMiddleware 函数最后次调用的参数传进来
```

```
    const store = createStore(...args)
    let dispatch = () => {
      throw new Error(
        `Dispatching while constructing your middleware is not allowed. ` +
          `Other middleware would not be applied to this dispatch.`
      )
    }
    let chain = []
    // 每个中间件都应该有这两个函数
    const middlewareAPI = {
      getState: store.getState,
      dispatch: (...args) => dispatch(...args)
    }
    // 把 middlewares 中的每个中间件都传入 middlewareAPI
    chain = middlewares.map(middleware => middleware(middlewareAPI))
    // 和之前一样，从右至左调用每个中间件，然后传入 store.dispatch
    dispatch = compose(...chain)(store.dispatch)
    // 这里只看这部分代码有点抽象，我这里放入 redux-thunk 的代码来结合分析
    // createThunkMiddleware返回了3层函数，第一层函数接收 middlewareAPI 参数
    // 第二次函数接收 store.dispatch
    // 第三层函数接收 dispatch 中的参数
{function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
  // 判断 dispatch 中的参数是否为函数
    if (typeof action === 'function') {
      //  是函数的话再把这些参数传进去，直到 action 不为函数，执行 dispatch({tyep:
'XXX'})
      return action(dispatch, getState, extraArgument);
    }

    return next(action);
  };
}
const thunk = createThunkMiddleware();

export default thunk;}
// 最后把经过中间件加强后的 dispatch 于剩余 store 中的属性返回，这样你的 dispatch
    return {
      ...store,
      dispatch
    }
  }
}
```

好了，我们现在将困难的部分都攻克了，来看一些简单的代码

```
// 这个没啥好说的，就是把当前的 state 返回，但是当正在执行 reducer 时不能执行该方法
function getState() {
    if (isDispatching) {
      throw new Error(
        'You may not call store.getState() while the reducer is executing. ' +
          'The reducer has already received the state as an argument. ' +
           'Pass it down from the top reducer instead of reading it from the
store.'
      )
    }

    return currentState
}
// 接收一个函数参数
function subscribe(listener) {
    if (typeof listener !== 'function') {
      throw new Error('Expected listener to be a function.')
    }
// 这部分最主要的设计 nextListeners 已经讲过，其他基本没什么好说的
    if (isDispatching) {
      throw new Error(
        'You may not call store.subscribe() while the reducer is executing. ' +
           'If you would like to be notified after the store has been updated,
subscribe from a ' +
          'component and invoke store.getState() in the callback to access the
latest state. ' +
             'See http://redux.js.org/docs/api/Store.html##subscribe for more
details.'
      )
    }

    let isSubscribed = true

    ensureCanMutateNextListeners()
    nextListeners.push(listener)

// 返回一个取消订阅函数
    return function unsubscribe() {
      if (!isSubscribed) {
        return
      }

      if (isDispatching) {
        throw new Error(
          'You may not unsubscribe from a store listener while the reducer is
executing. ' +
             'See http://redux.js.org/docs/api/Store.html##subscribe for more
details.'
        )
      }
      isSubscribed = false
```

```
      ensureCanMutateNextListeners()
      const index = nextListeners.indexOf(listener)
      nextListeners.splice(index, 1)
    }
  }

function dispatch(action) {
// 原生的 dispatch 会判断 action 是否为对象
    if (!isPlainObject(action)) {
      throw new Error(
        'Actions must be plain objects. ' +
          'Use custom middleware for async actions.'
      )
    }

    if (typeof action.type === 'undefined') {
      throw new Error(
        'Actions may not have an undefined "type" property. ' +
          'Have you misspelled a constant?'
      )
    }
// 注意在 Reducers 中是不能执行 dispatch 函数的
// 因为你一旦在 reducer 函数中执行 dispatch，会引发死循环
    if (isDispatching) {
      throw new Error('Reducers may not dispatch actions.')
    }
// 执行 combineReducers 组合后的函数
    try {
      isDispatching = true
      currentState = currentReducer(currentState, action)
    } finally {
      isDispatching = false
    }
// 然后遍历 currentListeners，执行数组中保存的函数
    const listeners = (currentListeners = nextListeners)
    for (let i = 0; i < listeners.length; i++) {
      const listener = listeners[i]
      listener()
    }

    return action
  }
 // 然后在 createStore 末尾会发起一个 action dispatch({ type: ActionTypes.INIT });
 // 用以初始化 state
```

版权声明

InfoQ 中文站出品

前端面试指南

©2018 北京极客邦科技有限公司

本书版权为 6 位作者（YuChengKai luoguangcong Sean Bill yygmind 穆尔）和北京极客邦科技有限公司共同所有，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司

北京市朝阳区来广营容和路叶青大厦（北园）五层

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。

网 址：www.infoq.com.cn

QCon
全球软件开发大会

聚焦

·互联网高可用架构
·国际化互联网业务架构
·工程师个人成长与技术领导力
·架构设计
·后移动互联网时代的技术思考与实践
·大规模基础设施DevOps探索
·硅谷人工智能
·人工智能与业务实践
·Java生态与创新
·大数据系统架构
·区块链技术与应用
·前端新趋势
·深度学习技术与应用
·微服务架构 & Serverless
·产品经理必修之用户细分与产品定位

实践

Facebook / 硅谷公司的互联网计算性能优化经验谈
LinkedIn / 推荐系统：提升用户增长与参与的利器
Uber / 核心Trip Flow容量管理
Confluent /Apache Kafka / 从0.8到2.0：那些年我们踩过的坑
快手 / 如何快速打造高稳定千亿级别对象存储平台
微软 / 集成AI开发平台实践

会议：2018年10月18–20日
培训：2018年10月21–22日
地址：上海·宝华万豪酒店

8折报名中，立减1360元
团购享更多优惠，
截至2018年8月19日

主办方 Geekbang. 极客邦科技 InfoQ

大咖助阵

专题：硅谷人工智能

夏磊 / LinkedIn高级工程师，
湾区同学技术沙龙Board Member

专题：Java生态与创新

张建锋 / 永源中间件 共同创始人

专题：互联网高可用架构

吴其敏 / 平安银行
零售网络金融事业部首席架构师

专题：研发效率提升

徐毅 / 华为 技术专家

专题：产品经理必修之用户细分与产品定位

袁店明 / Dell EMC
敏捷与精益创业咨询师

……

分享嘉宾

张翔
三一重工
所长兼项目经理

庄振运
Facebook
计算机性能高级工程师

邹欣
微软亚洲研究院
首席研发经理

李欣（Bruce Li）
Paypal
PayPal Risk Infra
Director level architect

Hien Luu
LinkedIn
工程经理

孙彦
Pinterest
视觉搜索团队高级工程师

施磊
Airbnb
技术经理

俞戴龙
Red Pulse
高级架构师

……

100+技术专家的实践分享

联系我们：

热线：010-84782011 微信：qcon-0410