前端面试总结

- css
  - 水平垂直居中
  - 清除浮动
- Javascript
  - 数据类型
  - 作用域
  - 闭包
  - 单线程和异步
  - event-loop
  - 原型和原型链
  - 继承
- Javascript API 实现
  - 实现 let
  - 实现 const
  - 实现数组 push
  - 实现数组 filter
  - 实现数组 map
  - 实现数组 flat
  - 实现字符串 trim
  - 实现字符串 repeat
  - 实现 Object.create
  - 实现 Object.is
  - 实现 Object.assign
  - 实现 instanceof
  - 实现 JSON.parse
  - 实现 new
  - 实现 call apply bind
  - 实现 Promise

- 实现 Promise.all
- 实现 Promise.race

Javascript 手写题

- 数组去重
- 浅拷贝
- 深拷贝
- 节流
- 防抖
- 发布订阅
- sleep 函数
- ajax 封装
- setTimeout 实现 setInterval
- setInterval 实现 setTimeout
- Jsonp 封装
- 解析 URL Params
- 对象扁平化
- 函数柯里化
- compose
- LazyMan

浏览器

- 前端存储方式
- 跨域
- Html 文档中各种资源的解析规则
- script 标签中 defer 和 async 的区别
- link 标签中 preload, prefetch, preconnect, dns-prefetch 的区别
- 输入地址按下回车的整个流程

网络

- HTTP 协议格式
- HTTP 缓存

性能优化

- 标准
- 谷歌官方学习网站

慢的影响
性能指标
获取指标
性能优化
vue
vue 组件间通信
vue 父子组件生命周期钩子执行顺序
vue 双向绑定实现原理
vue 源码调试技巧
vue 源码分析
工程化
webpack 优化
webpack 原理
手写一个 loader
手写一个 plugin

CSS

水平垂直居中

水平居中：

HTML

```html
<div class="parent">
  <div class="child"> 啦啦啦 </div>
</div>

<style>
/* 方案一：inline-block + text-align */
.parent { text-align: center; }
.child { display: inline-block; }
/* 方案二：block + margin */
.child {
  width: 100px; /* 需要设置宽度 */
  display: block; /* 设置成table可以不设置宽度 */
  margin: 0 auto;
}
/* 方案三：absolute + transform/margin */
.parent { position: relative; }
.child {
  position: absolute;
  left: 50%;
  transform: translateX(-50%);
}
</style>
```

垂直居中：

```HTML
<div class="parent">
  <div class="child"> 啦啦啦 </div>
</div>

<style>
/* 方案一：table-cell + vertical-align */
.parent {
  display: table-cell; /* 单元格的内容是可以设置水平垂直对齐的 */
  vertical-align: middle; /* 用于设置文本内容的垂直方向对齐方式 */
}
/* 方案二：absolute + transform/margin */
.parent { position: relative; }
.child {
  position: absolute;
  top: 50%;
  transform: translateY(-50%);
}
</style>
```

清除浮动

1.父级添加 overflow 属性

通过触发 BFC 方式（就是负责接管自己的宽高），实现清除浮动。

```CSS
.fahter {
  overflow: hidden; /* auto 也可以 */
}
```

2.额外标签 clear: both;

在最后一个浮动标签后，新加一个标签，给其设置 clear: both;

3.使用 after 伪元素清除浮动

```css
.clearfix::after {
  display: block; /* 伪元素默认是 inline 的，inline 元素无法帮我们做清除浮动的事情。 */
  content: " ";
  clear: both;
  height: 0;
  visibility: hidden;
  overflow: hidden;
}
```

4.万能清除法

```css
.clearfix::before, .clearfix::after {
  display: block;
  content: " ";
}
.clearfix::after {
  clear: both;
}
```

Javascript

数据类型

typeof：

值类型：'undefined', 'number', 'string', 'boolean', 'symbol'

引用类型：'function', 'object'(对象，数组，null)

Object.prototype.toString.call(obj)：

'[object Undefined]', '[object Number]', '[object String]', '[object Boolean]', '[object Symbol]', '[object Function]', '[object Object]', '[object Array]', '[object Null]'

获取数据类型：

```javascript
function getType(value) {
  if (value === null) return value + "";
  if (typeof value === "object") {
    let valueClass = Object.prototype.toString.call(value),
    type = valueClass.split(" ")[1].split("");
    type.pop();
    return type.join("").toLowerCase();
  } else {
    return typeof value;
  }
}
```

作用域

作用域：作用域就是变量，函数能够使用的范围。

作用域链：自由变量的查找是在函数定义的地方向上级作用域查找，不是在函数执行的地方。

自由变量：当前作用域中使用了却没有声明的变量。

闭包

闭包是一个绑定了执行环境的函数。

- 函数的词法环境
- 函数中用到的未声明的变量
- 函数体

有两个函数，内部函数定义在外部函数中，并且内部函数使用了外部函数环境中定义的变量，当内部函数执行的时候，外部函数就形成了一个闭包。

```JavaScript
(function () {
  var a = 1;
  function add() {
    var b = 2;
    var sum = b + a;
    console.log(sum) // 3
  }
  add();
})();
```

闭包作用：

1.私有化数据。

函数中的 data 只能通过 get 和 set 访问，外部无法访问，相当于将变量私有化。

```JavaScript
function createPrivate() {
  const data = {};
  return {
    get(key) {
      return data[key];
    },
    set(key, value) {
      data[key] = value;
    }
  }
}
```

2.使局部变量常驻内存

```javascript
// 打印 10个10
for (var i = 0; i < 10; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}
// 闭包
for (let i = 0; i < 10; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
}
for (var i = 0; i< 10; i++){
  (function(i) {
    setTimeout(() => {
      console.log(i);
    }, 1000);
  })(i);
}
```

3.节流/防抖函数里都用到了闭包

单线程和异步

单线程：只有一个线程，同一时间只能做一件事情。

单线程原因：避免 dom 渲染的冲突。

异步：在单线程的环境下，针对耗时很长的任务会阻塞后面代码的执行，造成页面卡死状态，所以要将这些任务变成异步来处理。

异步写法：

- 回调函数
- jquery deferred
- Promise
- async/await
- generator

event-loop

事件循环是浏览器对异步的实现方式。

整体的 js 代码这个宏任务先执行，同步代码执行完后有微任务执行微任务，没有微任务执行下一个宏任务，如此往复循环至结束。

宏任务之间会触发页面渲染。

```JavaScript
// 修改DOM
const $p1 = $('<p>一段文字</p>')
const $p2 = $('<p>一段文字</p>')
const $p3 = $('<p>一段文字</p>')
$('#container')
  .append($p1)
  .append($p2)
  .append($p3);
// 微任务：DOM 渲染之前执行
Promise.resolve().then(() => {
  const length = $('#container').children().length;
  alert(`micro task ${length}`);
});
// 宏任务：DOM 渲染之后执行
setTimeout(() => {
  const length = $('#container').children().length;
  alert(`macro task ${length}`);
});
```

原型和原型链

原型：

- 每个 class 都有显示原型 prototype
- 每个实例都有隐式原型 __proto__
- 实例的隐式原型 __proto__ 指向对应 class 的显示原型 prototype
- prototype 中有一个 constructor 属性，用来引用它的构造函数，Person.prototype.constructor === Person，这是一种循环引用。

原型链：我们把由 __proto__ 串起来的直到 Object.prototype.__proto__ 为 null 的链叫做原型链。

```JavaScript
Object.__proto__ === Function.prototype
Function.__proto__ === Function.prototype
Function.prototype.__proto__ === Object.prototype
Object.prototype.__proto__ === null // 原型链顶端

Object instanceof Function; // true
Function instanceof Object; // true
```

继承

对象冒充继承：

```JavaScript
function Person(name) {
  this.name = name;
}
function Student(name) {
  this.fn = Person;
  this.fn(name);
  delete this.fn;
}
```

call,apply,bind

call 和 apply 可以实现多重继承，一个子类能够继承多个父类，F1 可以同时从 F2, F3 ... 继承。

原型链继承：

对象继承类

```JavaScript
o.__proto__ = F.prototype;
```

类继承类

```JavaScript
Student.prototype.__proto__ = Person.prototype;
```

混合方式继承：

```JavaScript
function Person(name) {
  this.name = name;
}
Person.prototype.sayName = function() {
  console.log(this.name);
}
function Student(name, age) {
  Person.call(this, name);
  this.age = age;
}
Student.prototype.__proto__ = Person.prototype;
```

class 继承：

```JavaScript
class Person {};
class Student extends Person {
  super();
};
```

new 继承：

构造函数创建实例的过程本身就是一种继承，new 的内部其实是做了继承里面的合体工作。

```JavaScript
Student.prototype = new Person();
```

Javascript API 实现

实现 let

```JavaScript
// 1
{
  let a = 1;
  console.log(a); // 1
}
console.log(a);
// 相当于
(function () {
  var a = 1;
  console.log(a); // 1
})();
console.log(a); // a is not defined

// 2
var fns = [];
for (let i = 0; i < 10; i++) {
  fns[i] = function() {
    console.log(i);
  };
}
fns[0](); // 0
// 相当于
var fns = [];
for (var i = 0; i < 10; i++) {
  (function(i) {
    fns[i] = function() {
      console.log(i);
    };
  })(i);
}
fns[0](); // 0
```

实现 const

```javascript
function _const(key, value) {
  window.key = value;
  Object.defineProperty(window, key, {
    enumerable: false,
    configurable: false,
    get() {
      return value;
    },
    set(newVal) {
      throw new TypeError('不能重复定义');
      // if(newVal !== value){
      //     throw new TypeError('不能重复定义');
      // } else {
      //     return value;
      // }
    }
  });
}
```

实现数组 push

```javascript
Array.prototype.push = function(...args) {
  for (let i = 0; i < args.length; i++) {
    this[this.length] = args[i];
  }
  return this.length;
}
```

实现数组 filter

```JavaScript
Array.prototype.filter = function(fn) {
  if (typeof fn !== 'function') throw TypeError('参数必须是一个函数');
  const res = [];
  for (let i = 0; i < this.length; i++) {
    if (fn(this[i], i)) res.push(this[i]);
  }
  return res;
}
```

实现数组 map

```JavaScript
Array.prototype.map = function(fn) {
  if (typeof fn !== 'function') throw TypeError('参数必须是一个函数');
  const res = [];
  for (let i = 0; i < this.length; i++) {
    res.push(fn(this[i], i));
  }
  return res;
}
```

实现数组 flat

```JavaScript
// 1 reduce + 递归
function flat(arr, depth = 1) {
  if(!Array.isArray(arr) || depth <= 0) return arr;
  return arr.reduce((acc, cur) => {
    return Array.isArray(cur) ?  acc.concat(flat(cur, depth - 1)) : acc.concat(cur);
  }, []);
}
// 2 迭代
function flat(arr) {
  const res = [];
  const arrs = [...arr];
  while(arrs.length) {
    const tmp = arrs.shift();
    Array.isArray(tmp) ? arrs.unshift(...tmp) : res.push(tmp);
  }
  return res;
}
// 3 递归
function flat(arr, depth = 1) {
  if(!Array.isArray(arr) || depth <= 0) return arr;
  const res = [];
  arr.forEach(item => {
    Array.isArray(item) ? res.push(...flat(item, depth - 1)) : res.push(item);
  })
  return res;
}
```

实现字符串 trim

```JavaScript
String.prototype.trim = function() {
  return this.replace(/(^\s*)|(\s*$)/g, '');
}
```

实现字符串 repeat

```javascript
// 1
String.prototype.repeat = function(n) {
  return (new Array(n + 1)).join(this);
}
// 2 递归
String.prototype.repeat = function(n) {
  return n > 0 ? this.repeat(n - 1) + this : '';
}
```

实现 Object.create

```javascript
Object.create = function(obj) {
  function F() {};
  F.prototype = obj;
  return new F();
}
```

实现 Object.is

```javascript
Object.is = function (x, y) {
  // 全等情况下，只有 +0 -0 返回 false
  if (x === y) {
    // 1/+0 === Infinity 1/-0 === -Infinity
    return x !== 0 || 1 / x === 1 / y;
  }
  // 不全等情况下，只有 NaN NaN 返回 true
  return x !== x && y !== y;
};
```

实现 Object.assign

```javascript
Object.assign = function(target, ...sources) {
    if (target == null) throw new TypeError('Cannot convert undefined or null to object');
    const res = Object(target);
    sources.forEach(source => {
      for (let key in source) {
        if (source.hasOwnProperty(key)) res[key] = source[key];
      }
    });
    return res;
}
```

实现 instanceof

```javascript
function Instanceof(left, right) {
  while (true) {
    if (left == null) return false;
    if (left.__proto__ === right.prototype) return true;
    left = left.__proto__;
  }
}
```

实现 JSON.parse

```javascript
// 1 eval
JSON.parse = function(jsonStr) {
  return eval(`(${jsonStr})`);
}
// 2 new Function()
JSON.parse = function(jsonStr) {
  return (new Function(`return ${jsonStr}`))();
}
```

实现 new

```JavaScript
function New(fn, ...args) {
  const obj = Object.create(fn.prototype);
  const res = fn.call(obj, ...args);
  if (res && (typeof res === 'object' || typeof res === 'function')) return res;
  return obj;
}
```

实现 call apply bind

call

```JavaScript
Function.prototype.call = function (obj, ...args) {
  obj = obj == null ? window : Object(obj);
  obj.fn = this;
  const res = obj.fn(...args);
  delete obj.fn;
  return res;
}
```

apply

```JavaScript
Function.prototype.apply = function (obj, args) {
  obj = obj == null ? window : Object(obj);
  obj.fn = this;
  const res = args ? obj.fn(...args) : obj.fn();
  delete obj.fn;
  return res;
}
```

bind

JavaScript

```javascript
Function.prototype.bind = function(obj, ...args) {
  obj = obj == null ? window : Object(obj);
  const fn  = this;
  const bound = function(...innerArgs) {
    if (this instanceof bound) {
      return new fn(...args, ...innerArgs);
    } else {
      // return fn.call(obj, ...args, ...innerArgs);
      obj.fn = fn;
      const res = obj.fn(...args, ...innerArgs);
      delete obj.fn;
      return res;
    }
  }
  return bound;
}
```

实现 Promise

JavaScript

```
class Promise {
  constructor(fn) {
    this.status = 'pending';
    this.onFulfilledCallbacks = [];
    this.onRejectedCallbacks = [];
    // 更改成功后的状态
    const resolve = value => {
      if (this.status === 'pending') {
        this.status = 'fulfilled';
        this.onFulfilledCallbacks.forEach(fn => fn(value));
      }
    }
    // 更改失败后的状态
    const reject = err => {
      if (this.status === 'pending') {
        this.status = 'rejected';
        this.onRejectedCallbacks.forEach(fn => fn(err));
      }
    }
    try {
      fn(resolve, reject)
    } catch(err) {
      reject(err)
    }
  }
  then(onFulfilled, onRejected) {
    onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : v => v
;
    onRejected = typeof onRejected === 'function' ? onRejected : err => {t
hrow err};
    // 为了链式调用，这里直接返回一个 Promise
    return new Promise((resolve, reject) => {
      const fulfilledMicrotask = value =>  {
        queueMicrotask(() => {
          try {
            const res = onFulfilled(value);
            res instanceof Promise ? res.then(resolve, reject) : resolve(r
es);
          } catch(err) {
            reject(err);
          }
        })
      }
      const rejectedMicrotask = value => {
        queueMicrotask(() => {
```

```
            try {
              const res = onRejected(value);
              res instanceof Promise ? res.then(resolve, reject) : reject(res);
            } catch(err) {
              reject(err);
            }
          })
        }
        this.onFulfilledCallbacks.push(fulfilledMicrotask);
        this.onRejectedCallbacks.push(rejectedMicrotask);
      });
    }
    static resolve(value) {
      if (value instanceof Promise) {
        return value;
      }
      return new Promise(resolve =>  {
        resolve(value);
      });
    }
    static reject(err) {
      return new Promise((resolve, reject) => {
        reject(err);
      });
    }
}
```

实现 Promise.all

```javascript
Promise.all = function(promises) {
  return new Promise((resolve, reject) => {
    const result = [];
    let count = 0;
    if (promises.length === 0) {
      resolve(result);
    } else {
      for (let i = 0; i < promises.length; i++) {
        Promise.resolve(promises[i]).then(res => {
          result[i] = res;
          count++;
          if (count === promises.length) {
            resolve(result);
          }
        }, err => {
          reject(err);
        });
      }
    }
  });
}
```

实现 Promise.race

```javascript
Promise.race = function(promises) {
  return new Promise((resolve, reject) => {
    for (let i = 0; i < promises.length; i++) {
      Promise.resolve(promises[i]).then(res => {
        resolve(res);
      }, err => {
        reject(err);
      });
    }
  });
}
```

Javascript 手写题

数组去重

```JavaScript
// 1.传统方式，遍历元素挨个比较，去重
function uniqueArr(arr) {
  const res = [];
  const map = new Map();
  for (let item of arr) {
    if (!map.has(item)) {
      res.push(item);
      map.set(item, 1);
    }
  }
  return res;
}
// 2.使用Set（无序，不能重复）
function uniqueArr(arr) {
  return [...new Set(arr)];
}
```

浅拷贝

```JavaScript
// Object.assign
let a = {a: 1, b: 2};
let b = Object.assign({}, a);
// ...
let a = {a: 1, b: 2};
let b = {...a};
// slice
let a = [1, {a: 1}];
let b = a.slice();
// concat
let a = [1, {a: 1}];
let b = [].concat(a);
```

深拷贝

```javascript
// 1
// 1.如果obj里面存在时间对象,JSON.parse(JSON.stringify(obj))之后，时间对象变成了字符串。
// 2.如果obj里有RegExp、Error对象，则序列化的结果将只得到空对象。
// 3.如果obj里有函数，undefined，则序列化的结果会把函数，undefined丢失。
// 4.如果obj里有NaN、Infinity和-Infinity，则序列化的结果会变成null。
// 5.JSON.stringify()只能序列化对象的可枚举的自有属性。如果obj中的对象是有构造函数生成的，则使用JSON.parse(JSON.stringify(obj))深拷贝后，会丢弃对象的构造函数constructor，即无法继续使用构造函数原型上的属性。
// 6.如果对象中存在循环引用的情况也无法正确实现深拷贝。
const o = JSON.parse(JSON.stringify(obj));

// 2
function isObject(val) {
  return typeof val === "object" && val !== null;
}
function deepClone(obj, hash = new WeakMap()) {
  if (!isObject(obj)) return obj;
  if (hash.has(obj)) return hash.get(obj);
  const res = Array.isArray(obj) ? [] : {};
  hash.set(obj, res);
  // Reflect.ownKeys(obj)=Object.getOwnPropertyNames(obj)+Object.getOwnPropertySymbols(obj)
  [
    ...Object.getOwnPropertyNames(obj),
    ...Object.getOwnPropertySymbols(obj),
  ].forEach(key => {
    res[key] = deepClone(obj[key], hash);
  });
  return res;
}
```

节流

节流是频繁操作的时候保持一个频率触发。比如拖拽一个元素时，要随时拿到这个元素被拖拽的位置。

JavaScript

```javascript
// 定时器版
function throttle(fn, delay = 100) {
  let flag = true;
  return function(...args) {
    if (!flag) return;
    flag = false;
    setTimeout(() => {
      fn.call(this, ...args);
      flag = true;
    }, delay);
  }
}
// 时间戳版
function throttle(fn, delay = 100) {
  let prev = 0;
  return function(...args) {
    const now = Date.now();
    if (now - prev > delay) {
      fn.call(this, ...args);
      prev = now;
    }
  }
}
// 使用
div.addEventListener('drag', throttle(function(e) {
  console.log(e.offsetX, e.offsetY);
}, 300));
```

防抖

防抖是频繁操作的最后时刻触发。比如输入停止后一段时间没有再输入才会请求接口。

JavaScript

```
// 定时器版
function debounce (fn, delay = 100) {
  let timer = null;
  return function(...args) {
    if (timer) clearTimeout(timer);
    timer = setTimeout(() => {
      fn.call(this, ...args);
      timer = null;
    }, delay)
  }
}
// 使用
input.addEventListener('keyup', debounce(function() {
  console.log(input.value);
}, 300));
```

发布订阅

JavaScript

```javascript
class EventEmitter {
  constructor() {
    this.events = {};
  }
  on(name, handler){
    this.events[name] = this.events[name] || [];
    this.events[name].push(handler);
  }
  emit(name, ...args) {
    if (!this.events[name]) throw new Error('该事件未注册');
    this.events[name].forEach(fn => fn.call(this, ...args));
  }
  off(name, handler) {
    if (!this.events[name]) throw new Error('该事件未注册');
    if (!handler) {
      delete this.events[name];
    } else {
      this.events[name] = this.events[name].filter(fn => fn !== handler);
    }
  }
  once(name, handler) {
    function fn(...args) {
      handler.call(this, ...args);
      this.off(name, fn);
    }
    this.on(name, fn);
  }
}
```

sleep 函数

```JavaScript
function sleep(time) {
  return new Promise(resolve => setTimeout(resolve, time));
}
// 使用
sleep(1000).then(()=>{
  console.log(1);
})
async function output() {
  let out = await sleep(1000);
  console.log(1);
}
output();
```

ajax 封装

```JavaScript
function ajax (url, method, postData) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open(method, url);
    xhr.setRequestHeader("Content-Type", "application/json");
    xhr.onreadystatechange = function () {
      if (xhr.readyState === 4) {
        if (xhr.status === 200) {
          resolve(JSON.parse(xhr.responseText));
        } else {
          reject(new Error(xhr.responseText));
        }
      }
    }
    method === 'GET' ? xhr.send() : xhr.send(JSON.stringify(postData));
  });
}
```

setTimeout 实现 setInterval

```JavaScript
function setInterval(fn, time) {
  function interval() {
    fn();
    setTimeout(interval, time);
  }
  return setTimeout(interval, time);
}
```

setInterval 实现 setTimeout

```JavaScript
const setTimeout = (fn, time) => {
  const timer = setInterval(() => {
    fn();
    clearInterval(timer);
  }, time);
  return timer;
}
```

Jsonp 封装

```JavaScript
// 发送请求
function createScript(src) {
  const script = document.createElement('script');
  script.src = src;
  script.type = "text/javascript";
  document.body.appendChild(script);
}
createScript("http://xxx.xxx.com/xxx.js?callback=handleResponse");
// 接收数据
function handleResponse(res) {
  console.log(res);
}
// 接口返回一个携带数据的函数调用的js
handleResponse({a: 1, b: 2});
```

解析 URL Params

```JavaScript
// 1.传统方式，查找location.search
function query(name) {
  const search = location.search.substring(1);
  const reg = new RegExp(`(^|&)${name}=([^&]*)(&|$)`, 'i');
  const res = search.match(reg);
  return res ? res[2] : null;
}
// 2.新api，URLSearchParams
function query(name) {
  const p = new URLSearchParams(location.search);
  return p.get(name);
}
```

对象扁平化

JavaScript

```javascript
function isObject(val) {
  return typeof val === "object" && val !== null;
}
function flatten(obj) {
  if (!isObject(obj)) return obj;
  const res = {};
  function dfs(cur, prefix) {
    if (isObject(cur)) {
      if (Array.isArray(cur)) {
        cur.forEach((item, index) => {
          dfs(item, `${prefix}[${index}]`);
        });
      } else {
        for (let key in cur) {
          dfs(cur[key], `${prefix}${prefix ? "." : ""}${key}`);
        }
      }
    } else {
      res[prefix] = cur;
    }
  };
  dfs(obj, '');
  return res;
}
// 测试
const obj = {
  a: {
    b: 1,
    c: 2,
    d: {e: 5}
  },
  b: [1, 3, {a: 2, b: 3}],
  c: 3
};
flatten(obj);
```

函数柯里化

柯里化（currying）是把接受多个参数的函数变换成接受一个参数的函数，并且返回接受剩余参数且返回结果的新函数的技术。

柯里化后的函数接收参数的数量与原函数的形参数量相等时，执行原函数；当接收的参数数量小于原函数形参数量时，返回一个用于接收剩余参数的函数。

JavaScript

```
// 1 只能传被柯里化函数形参的个数，多少都不行
function currying(fn, ...args) {
    const length = fn.length;
    return function(...newArgs) {
        const allArgs = [...args, ...newArgs];
        if (allArgs.length < length) {
            return currying.call(this, fn, ...allArgs);
        } else {
            return fn.apply(this, allArgs);
        }
    }
}
const add = (a, b, c) => a + b + c;
const addCurrying = currying(add);
console.log(addCurrying(1)(2)(3));
console.log(addCurrying(1,2)(3));
console.log(addCurrying(1)(2,3));
console.log(addCurrying(1,2,3));

// 2 参数长度不固定，最后需要手动调用一次
function currying(fn, ...args) {
    let allArgs = [...args];
    return function temp(...newArgs) {
        if (newArgs.length) {
            allArgs = [...allArgs, ...newArgs];
            return temp;
        } else {
            const res = fn.apply(this, allArgs);
            allArgs = [...args];
            return res;
        }
    }
}
const add = (...args) => args.reduce((a, b) => a + b);
const addCurrying = currying(add);
console.log(addCurrying(1)(2)(3)(4,5)());
console.log(addCurrying(1)(2)(3,4,5)());
console.log(addCurrying(1)(2,3,4,5)());
```

柯里化用途

```javascript
// 1 校验规则
// 原函数
function checkByRegExp(regExp, string) {
    return regExp.test(string);
}
// 普通使用
checkByRegExp(/^1\d{10}$/, '18642838455'); // 校验电话号码
checkByRegExp(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/, 'test@163.com'); // 校验邮箱
// 柯里化后使用
let check = currying(checkByRegExp);
let checkCellPhone = check(/^1\d{10}$/);
let checkEmail = check(/^(\w)+(\.\w+)*@(\w)+((\.\w+)+)$/);
checkCellPhone('18642838455'); // 校验电话号码
checkCellPhone('13109840560'); // 校验电话号码
checkCellPhone('13204061212'); // 校验电话号码
checkEmail('test@163.com'); // 校验邮箱
checkEmail('test@qq.com'); // 校验邮箱
checkEmail('test@gmail.com'); // 校验邮箱
// 2 提取对象数组的某一属性
const list = [
    {name:'lucy'},
    {name:'jack'}
];
// 普通使用
const names = list.map(item => item.name);
// 柯里化使用
const prop = currying((key, obj) => obj[key]);
const names = list.map(prop('name'));
```

compose

```javascript
function compose(...fns) {
    if (!fns.length) return v => v;
    if (fns.length === 1) return fns[0];
    return fns.reduce((pre, cur) => (...args) => pre(cur(...args)));
}
function compose(...fns) {
    if (!fns.length) return v => v;
    if (fns.length === 1) return fns[0];
    return fns.reduce((pre, cur) => {
        return (...args) => {
            return pre(cur(...args));
        }
    });
}
// 使用
function fn1(x) {
  return x + 1;
}
function fn2(x) {
  return x + 2;
}
function fn3(x) {
  return x + 3;
}
function fn4(x) {
  return x + 4;
}
const a = compose(fn1, fn2, fn3, fn4); // a = (...args) => fn1(fn2(fn3(fn4(...args))))
console.log(a(1)); // 1+4+3+2+1=11
```

LazyMan

JavaScript

```javascript
class LazyManClass {
  constructor(name) {
    this.tasks = [];
    const task = () => {
      console.log(`Hi! This is ${name}`);
      this.next();
    };
    this.tasks.push(task);
    setTimeout(() => {
      // 所有任务添加完之后开始初始化执行任务队列的任务。
      this.next();
    }, 0);
  }
  next() {
    // 取第一个任务执行
    const task = this.tasks.shift();
    task && task();
  }
  sleep(time) {
    this.sleepWrapper(time, false);
    return this;
  }
  sleepFirst(time) {
    this.sleepWrapper(time, true);
    return this;
  }
  sleepWrapper(time, isFirst) {
    const task = () => {
      setTimeout(() => {
        console.log(`等待${time}秒`);
        this.next();
      }, time * 1000);
    };
    if (isFirst) {
      this.tasks.unshift(task);
    } else {
      this.tasks.push(task);
    }
  }
  eat(name) {
    const task = () => {
      console.log(`eat ${name}`);
      this.next();
    };
    this.tasks.push(task);
```

```
    return this;
  }
}
function LazyMan(name) {
  return new LazyManClass(name);
}
LazyMan('Tony').eat('lunch').eat('dinner').sleepFirst(5).sleep(4).eat('junk food');
```

浏览器

前端存储方式

cookie：大小只有4k，设置后自动加入请求头浪费流量，每个 domain 限制 20 个。api 怪异，使用需要自行封装。

localStorage：大小 5M，操作方便，永久性存储。

sessionStorage：只存在于当前页面，不能在窗口之间共享，页面关闭后就会被清理。

Web SQL：关系型数据库。2010 年被废弃。

IndexedDB：NoSQL 非关系型数据库，用键值对进行存储，读取速度快，javascript 操作方便。

跨域

同源策略：针对 ajax 请求，浏览器要求当前网页和请求的服务必须同源，即协议、域名、端口三者一致。

image，css，js，form 表单提交也不受跨域限制：

1.<img/>

图片可用于统计打点。统计可能是使用第三方统计服务，比如站长之家，百度统计等，这些都是外域的，统计打点无非就是发一个请求，如果用 ajax 发的话就会出现跨域。所以说我们用图片，初始化一个图片，把图片的地址写成第三方统计服务的地址，通过图片去发这个请求就可以了。

2.<link> <script>

<link /> <script> 可以使用 cdn，cdn 一般都是外域。

跨域解决方案：

所有的跨域解决方案都必须经过 server 端允许和配合。

1.Jsonp

<script> 可以绕过跨域限制。

只能用 GET 请求，并且要求返回 JavaScript。

2.cors

服务端支持的一种解决跨域的方式，是纯服务器端的操作。

JavaScript

```
response.setHeader('Access-Control-Allow-Origin', 'http://localhost:8081')
// 允许的域名是什么
response.setHeader('Access-Control-Allow-Headers', 'X-Requested-With')
// 允许的headers是什么
response.setHeader('Access-Control-Allow-Methods', 'PUT,POST,GET,DELETE,OPTIONS') // 允许的methods是什么
response.setHeader('Access-Control-Allow-Credentials', 'true') // 接收跨域的cookie，是否允许传cookie
```

3.设置反向代理

Html 文档中各种资源的解析规则

html 在接收到一部分之后就开始解析。

css 下载和解析不会阻塞 dom 的解析。

js 的下载和解析都会阻塞 dom 的解析。

js 的解析需要等待 cssom 全部解析完。

页面渲染需要 dom 和 cssom 全部解析完。

script 标签中 defer 和 async 的区别

```
<script src="script.js"></script>
```

按照顺序来加载并执行脚本，在脚本加载及执行过程中，会阻塞后续 html 文档的解析。

```
<script defer src="script.js"></script>
```

加载过程不会影响 html 文档解析，并且在 html 文档解析成功后，DOMContentLoaded 事件触发之前执行脚本。

```
<script async src="script.js"></script>
```

加载过程不会影响 html 文档解析，加载成功后会立即执行脚本内容，这个过程会阻塞后续 html 文档的解析。

link 标签中 preload, prefetch, preconnect, dns-prefetch 的区别

```
<link rel="preload" href="/main.js" as="script">
```

浏览器会在遇到如上 link 标签时，立刻开始下载 main.js(异步加载)，并放在内存中，但不会执行。只有当遇到 script 标签加载的也是 main.js 的时候，浏览器才会将预先加载的 JS 执行掉。如果这个时候 JS 仍然没有下载完，浏览器不会重新发请求，而是等待此文件的加载。字体和图片等资源也可以用这个属性，要用 as 属性标明资源类型，否则这个设置会失效。

```
<link rel="prefetch" href="main.js" as="script">
```

与 preload 类似。区别是浏览器会在空闲的时候下载，在还没下载完的时候就用到了该资源，会再次发起请求。所以在当前页面马上就要用的资源用 preload，不是马上用的资源用 prefetch。

```
<link rel="preconnect" href="https://cdn.bootcss.com">
```

提前建立 tcp 链接。

```
<link rel="dns-prefetch" href="https://cdn.bootcss.com">
```

提前查找 dns 解析域名。

输入地址按下回车的整个流程

1. 查找缓存：有缓存，返回缓存副本，并直接结束请求。没有缓存，发起网络请求过程。
2. 准备 IP 地址：先查找浏览器中的 DNS 数据缓存，没有缓存浏览器会请求 DNS 返回域名对应的 IP。
3. 等待 TCP 队列：Chrome 同一个域名同时最多只能建立 6 个 TCP 连接。
4. 建立 TCP 连接：排队等待结束之后，浏览器通过 TCP 与服务器建立连接。
5. 发送 HTTP 请求：一旦建立了 TCP 连接，浏览器就可以和服务器进行通信了。而 HTTP 中的数据正是在这个通信过程中传输的。浏览器会向服务器发送请求行，请求头，请求体信息。
6. 服务器处理请求。
7. 服务器返回请求。
8. 断开 TCP 连接：一旦服务器向客户端返回了请求数据，它就要关闭 TCP 连接。如果浏览器或者服务器在其头信息中加入了 Connection: Keep-Alive，TCP 会一直保持连接。
9. 重定向：返回的状态码是 301，告诉浏览器要重定向到另外一个网址，重定向的网址包含在响应头 Location 字段中，浏览器使用该地址重新导航。
10. 构建 DOM 树：由 HTML 解析器将 html 文件解析成树状结构的 DOM。
11. 样式计算：计算出 DOM 节点中每个元素的具体样式。
12. 布局：根据 DOM 和 ComputedStyle 生成一棵只包含可见元素的布局树，并计算出布局树节点的具体坐标位置。
13. 分层：将页面分成很多图层。
14. 绘制：为每个图层生成绘制命令列表。
15. 栅格化：将图层分成图块，并将图块利用 GPU 转换成位图。
16. 合成和显示：浏览器根据绘制命令将页面内容绘制到内存，将渲染好的页面显示到显示器上。停止标签图标上的加载动画。

重排会走整个渲染流程，重绘会走绘制之后的渲染流程，合成会走栅格化之后的渲染流程（如使用 transform）

网络

HTTP 协议格式

```bash
curl -v http://www.baidu.com
```

请求部分：

- 请求行 request line
  - 请求方法：表示此次 HTTP 请求希望执行的操作类型。只是语义上的约定，并没有强约束。
    GET, POST, HEAD, PUT, DELETE, CONNECT, OPTIONS, TRACE。
    浏览器通过地址栏访问页面都是 GET 方法。表单提交产生 POST 方法。
    HEAD 则是跟 GET 类似，只返回请求头，多数由 JavaScript 发起。
    PUT 和 DELETE 分别表示添加资源和删除资源。
    CONNECT 现在多用于 HTTPS 和 WebSocket。
    OPTIONS 和 TRACE 一般用于调试，多数线上服务都不支持。预检请求的 method 也是 OPTIONS。
  - 请求路径
  - 协议和版本
- 请求头 request header
  HTTP 头也是一种数据，可以自由定义 HTTP 头和值。不过在 HTTP 规范中，规定了一些特殊的 HTTP 头。
  - Accept：告诉服务端想要的数据类型。
  - Accept-Charset：想要接收数据的字符集。
  - Accept-Encoding：数据编码方式，用来限制服务端如何进行数据压缩。
  - Accept-Language：语言。
  - Connection：连接方式，如果是 keep-alive，且服务端支持，则会复用连接。
  - Cookie：客户端存储的 cookie 字符串。
  - User-Agent：浏览器的一些相关的信息。操作系统及版本/cpu/浏览器及版本/浏览器渲染引擎/浏览器语言/浏览器插件。
  - If-Modified-Since：上次访问时服务端返回的 Last-Modified。
  - If-None-Match：上次访问时服务端返回的 ETag。
- 请求体：请求体可能包含文件或者表单数据
  HTTP 请求的 body 主要用于提交表单场景。一些常见的 body 格式是：
  - application/json
  - application/x-www-form-urlencoded
  - multipart/form-data：既有文本数据，又有文件等二进制数据。所有的传输数据类型都会在编码里面去体现。
  - text/xml

响应部分：

- 响应行 response line
  - 协议和版本
  - 状态码
    - 1xx：临时回应，表示客户端请继续。对前端来说，1xx 系列的状态码是非常陌生的，原因是 1xx 的状态被浏览器 HTTP 库直接处理掉了，不会让上层应用知晓。

        - 2xx：请求成功。
            - 200：请求成功。
        - 3xx: 表示请求的目标有变化，希望客户端进一步处理。
            - 301&302：永久性与临时性跳转。表示当前资源已经被转移。
            - 304：跟客户端缓存没有更新。
        - 4xx：客户端请求错误。
            - 400：请求参数有语法错误，不能被服务器理解。
            - 401：没登录，鉴权失败。
            - 403：无权限。禁止访问，服务器收到请求，但是拒绝提供服务。
            - 404：表示请求的资源不存在。
        - 5xx：服务端请求错误。
            - 500：服务端错误。
            - 502：网关错误。
            - 503：由于超载，请求超时或停机维护，服务器目前无法使用，一段时间后可恢复正常，服务端暂时性错误，可以一会再试。
    - 状态文本
- 响应头 response header
    - Content-Type：对应 Accept，Accept 里面可以接收好几种不同的数据格式，那么 Content-Type 可以从里面选择一种然后做为它真正返回的数据格式进行一个返回，客户端根据这个来进行一个怎么样的显示。
    - Content-Encoding：对应的是 Accept-Encoding，服务端具体使用的数据压缩方式。
    - Content-Language：语言。
    - Content-Length：内容长度，有利于浏览器判断内容是否已经结束。
    - Connection：连接方式，keep-alive 表示复用连接。
    - Keep-Alive：保持连接不断时需要的一些信息，如 timeout=5, max=100。
    - Location：告诉客户端重定向的地址。
    - Set-Cookie：设置 cookie，可以存在多个。
    - Cache-Control：缓存控制，用于通知各级缓存保存的时间，例如 max-age=0，表示不要缓存。
    - Expires：过期时间，用于判断下次请求是否需要到服务端取回页面。
    - Last-Modified：页面上次修改的时间。
    - ETag：页面信息摘要，用于判断是否需要重新到服务端取回页面。
    - Access-Control-Allow-Origin：允许的跨域的源，如：'http://localhost:3000'
    - Access-Control-Allow-Headers：允许跨域的请求头，如：'X-Token,Content-Type'
    - Access-Control-Allow-Method：允许跨域的方法，如：'PUT,OPTIONS'
    - Access-Control-Allow-Credentials: true。跨域时默认是不记录 cookie 认证信息的。加上这个让它能够记录，从而能够使用 cookie。
- 响应体：头之后，以一个空行为分隔，响应体则是 HTML 代码。

预检请求：使用了非正常的请求报头或使用非 get/post 的请求会触发预检请求。

HTTP 缓存

命中强缓存后不会发送请求，没有命中强缓存后走协商缓存。

强缓存：

cache-control 优先级高于 expires

- expires：它的值为一个绝对时间的 GMT 格式的时间字符串。发送请求的时间在 expires 之前，本地缓存始终有效，强缓存命中。
- cache-control：max-age=number，它是一个相对值，根据资源第一次的请求时间和这个相对值，计算出一个资源过期时间，之后的请求时间在过期时间之前，就能命中缓存。该头可以存在多个。
  - no-cache：不使用强缓存，需要使用缓存协商。
  - no-store：禁止使用强缓存和协商缓存等任何缓存行为。
  - public：可以被所有的用户缓存，包括终端和 CDN 等中间代理服务器。
  - private：只能被终端的浏览器缓存，不允许 CDN 等中继缓存服务器对其缓存。

协商缓存：

协商缓存由两对 http 头组成。

服务器会优先验证 ETag，一致的情况下，才会继续比对 Last-Modified。

- Last-Modified/If-Modified-Since：
  这两个值是 GMT 格式的时间字符串。
  - 浏览器在第一次请求一个资源，在 respone 的 header 加上 Last-Modified 的 header，表示这个资源在服务器上的最后修改时间。
  - 浏览器再次跟服务器请求这个资源时，在 request 的 header 上加上 If-Modified-Since 的 header，这个 header 的值就是上一次请求时返回的 Last-Modified 的值。服务器根据浏览器传过来 If-Modified-Since 和资源在服务器上的最后修改时间判断资源是否有变化，如果没有变化则返回 304 Not Modified，但是不会返回资源内容和 Last-Modified；如果有变化，就正常返回资源内容和新的 Last-Modified。
- Etag/If-None-Match
  这个值是由服务器生成的资源的唯一标识字符串，只要资源有变化这个值就会改变。
  过程与 Last-Modified/If-Modified-Since 类似。不同的是，当服务器返回 304 Not Modified 的响应时，由于 ETag 重新生成过，response header 中还会把这个 ETag 返回，即使这个 ETag 跟之前的没有变化。
- 优缺点

- Last-Modified
  - 一些文件会周期性的修改时间，但内容并没有改变，这个时候我们并不希望客户端认为这个文件被修改了，而重新 GET；
  - 某些文件修改非常频繁，比如在秒以下的时间内进行修改，If-Modified-Since 能检查到的粒度是 s 级的，这种修改无法判断(或者说UNIX记录MTIME只能精确到秒)；
  - 某些服务器不能精确的得到文件的最后修改时间。
- Etag
  - Etag 能很好的解决上面 Last-Modified 遇到的问题，但由于要生成 hash，会消耗性能。

用户行为对缓存的影响：

性能优化

标准

长任务：https://www.w3.org/TR/2017/WD-longtasks-1-20170907/

性能：https://www.w3.org/TR/navigation-timing-2/

1.使用PerformanceNavigationTiming界面获取与文档导航相关的准确计时数据·

JavaScript

```
<script>
function showNavigationDetails() {
  // Get the first entry
  const [entry] = performance.getEntriesByType("navigation");
  // Show it in a nice table in the developer console
  console.table(entry.toJSON());
}
</script>
<body onload="showNavigationDetails()">
```

谷歌官方学习网站

https://web.dev/

慢的影响

如果网站太慢会影响用户的体验，会造成客诉或资损。

- 57%的用户更在乎网页在3秒内是否完成加载。
- 52%的在线用户认为网页打开速度影响到他们对网站的忠实度。
- 每慢1秒造成页面 PV 降低11%，用户满意度也随之降低降低16%。
- 近半数移动用户因为在10秒内仍未打开页面从而放弃。

性能指标

https://juejin.cn/post/6850037270729359367

- FP（First Paint）：首次绘制。
- FCP（First Contentful Paint）：首次内容绘制。2s 内优秀。
- 白屏时间：输入网址回车后的时间到 FCP 的时间。
- LCP（Largest Contentful Paint）：最大内容绘制（2.5s – 4.0s）
- FMP（First Meaningful Paint）：首次有意义绘制。
- DCL（DOMContentLoaded Event）：dom 渲染完成事件。
- L（Loaded Event）：全部元素渲染完成事件。
- 首屏时间：输入网址回车后的时间到全部页面展示出来的时间。
- TTI（Time to Interactive）：首次可交互时间
  - 从 FCP 指标后开始计算
  - 持续 5 秒内无长任务（执行时间超过 50 ms）且无两个以上正在进行中的 GET 请求
  - 往前回溯至 5 秒前的最后一个长任务结束的时间
- FID（First Input Delay）：首次输入延迟，在 FCP 和 TTI 之间，用户首次与页面交互到 TTI 的时间。用户交互事件触发到页面响应中间耗时多少，如果其中有长任务发生的话那么势必会造成响应时间变长（100ms – 300ms）。
- TBT（Total Blocking Time）：阻塞总时间，记录在 FCP 到 TTI 之间所有长任务的阻塞时间总和。每个长任务的阻塞时间就等于它所执行的总时间减去 50ms。执行时间大于 50ms就是长任务，否则是短任务。（200ms – 600ms）
- CLS（Cumulative Layout Shift）：累计位移偏移。位移距离 / 位移影响的面积（0.1 – 0.25）
- 除了这些指标以外，我们还需要获取网络、文件传输、DOM等信息丰富指标内容。

获取指标

- web-vitals-extension
- web-vitals 库
- Lighthouse

```JavaScript
import {getCLS, getFID, getLCP} from 'web-vitals';

getCLS(console.log);
getFID(console.log);
getLCP(console.log);
```

- Chrome DevTools – Performance
  https://zhuanlan.zhihu.com/p/163474573
- Performance API
  mdn：https://developer.mozilla.org/zh-CN/docs/Web/API/Performance

性能优化

- 优化 FP、FCP、LCP、FMP 指标（白屏、首屏时间）
  - 资源优化
    - 图片优化
      - 使用合适的图片格式
      - 小图标使用字体图标
      - 小图使用 base64
      - 图片懒加载
      - 图片渐进式加载
    - 文件压缩：服务端配置 Gzip 压缩文件体积
    - 代码压缩
    - 异步组件，按需加载
    - Code Splitting
    - 动态 polyfill
    - Tree shaking
    - Scope Hoisting
    - 单页应用改为多页应用
  - 网络优化
    - 缓存文件，对首屏数据做离线缓存
    - 服务端渲染
    - 使用 CDN 加载资源
    - 首屏不需要使用的 CSS 文件不加载

        - 内联关键的 CSS 代码
        - 资源预加载
        - 使用 dns-prefetch 预解析 IP 地址
        - 使用 preconnect，提前建立 TCP 连接
        - 使用 HTTP2.0 协议、TLS 1.3 协议或者直接拥抱 QUIC 协议
- 优化 TTI、FID、TBT 指标（优化耗时任务）
    - 使用 Web Worker 将耗时任务丢到子线程中，这样能让主线程在不卡顿的情况下处理 JS 任务
    - 调度任务 + 时间切片，这块技术在 React 16 中有使用到。简单来说就是给不同的任务分配优先级，然后将一段长任务切片，这样能尽量保证任务只在浏览器的空闲时间中执行而不卡顿主线程
- 优化 CLS 指标
    - 使用骨架屏给用户一个预期的内容框架，突兀的显示内容体验不会很好
    - 图片切勿不设置长宽，而是使用占位图给用户一个图片位置的预期
    - 不要在现有的内容中间插入内容，起码给出一个预留位置
- 代码优化
    - css 放在 head 里面：尽早的使 css 加载完成并执行完成。
    - js 放到 body 最下面：防止 js 阻塞 dom 解析。
    - 对 dom 查询进行缓存
    - 使用 DOMFragment 批量 DOM 操作。
    - css 选择器避免使用过多层级，避免使用标签选择器。
    - 频繁回流重绘的节点设置为单独的图层。使用 will-change。
    - 尽早执行 js：window.DOMContentLoaded：dom 渲染完即可执行，此时图片，视频可能还没有加载完。
    - 编写一些时间复杂度比较低的代码。
    - 节流防抖
    - 合理的加一些 loading

vue

vue 组件间通信

- props
- 自定义事件
- eventbus
- Vuex
- $parent/$ root
- $children
- $refs
- provide/inject

vue 父子组件生命周期钩子执行顺序

1. 加载渲染过程
   `父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount->子mounted->父mounted`
2. 子组件更新过程
   `父beforeUpdate->子beforeUpdate->子updated->父updated`
3. 父组件更新过程
   `父beforeUpdate->父updated`
4. 销毁过程
   `父beforeDestroy->子beforeDestroy->子destroyed->父destroyed`

vue 双向绑定实现原理

数据响应式 + 事件发布订阅

vue 源码调试技巧

搭建调试环境：

1. clone 源码，地址：https://github.com/vuejs/vue.git 版本:2.6.10
2. 安装依赖：npm install
3. 安装 rollup，Vue 的打包工具是 rollup：npm install -g rollup
4. 修改 package.json 中的 dev 打包脚本：增加 --sourcemap
   "dev": "rollup -w -c scripts/config.js --sourcemap --environment TARGET:web-full-dev"
5. 打包，执行开发脚本，输出最终我们要用的 vue.js：npm run dev
   打包成功之后 dist 下会生成一个全新的 vue.js，和它的 map 文件 vue.js.map
6. 编写测试文件
   examples/test/01-test.html
   把刚才打包的 vue.js 引进来，写一个 vue 程序，接下来就可以调试了。

调试技巧：

- 打开指定文件：ctrl+p
- 断点
- 单步执行：单步跳过函数/单步进入函数
- 查看调用栈：调用栈中可以很好的看到整个的函数执行的流程。

- 定位当前源文件所在位置：sources 代码上右键，Reveal in sidebar 选项。

vue 源码分析

1.根据打包命令找到打包的入口文件

核心功能：扩展 $mount

src/platforms/web/entry-runtime-with-compiler.js：

- 针对 web 平台的特点对 $mount 做扩展，扩展的就是跟编译相关的事。(功能扩展的方式值得学习)
  处理 render > template > el 选项：
  选项中如果有 render 直接调用 mount 执行挂载；如果有 template 或 el，将它们进行一定处理最后变成 template，然后将这个 template 执行模版解析和编译，最终得到 render 函数并将其放到选项中去。最后执行挂载操作。

2.寻找 Vue 构造函数

核心功能：定义 $mount, __patch__, 初始化全局 API(Vue.xxx), 定义 Vue, 初始化实例 API(Vue.prototype.xxx)

src/platforms/web/runtime/index.js：

- 安装 web 平台特有指令和组件；
- 在 Vue 原型上定义了补丁方法 Vue.prototype.__patch__：把虚拟 DOM 转换成真实 DOM。初始化的赋值和以后的更新都会用到这个 patch，也是 diff 算法发生的地方；
- 定义 $mount：它只做了一件事，就是把 el 做 DOM 查询，然后调用 mountComponent 执行挂载，将首次渲染的结果替换 el。

src/core/index.js：

- 初始化全局 API：Vue.util, Vue.set, Vue.delete, Vue.nextTick, Vue.use, Vue.mixin, Vue.extend, Vue.component, Vue.directive, Vue.filter

src/core/instance/index.js：

- 定义 Vue 构造函数：内部只执行了一行初始化方法 this._init()。
- 使用混入的方式定义 Vue 实例 API（这个混入的方式扩展构造函数原型值得学习）
  - initMixin(Vue)：定义了初始化方法 _init

- stateMixin(Vue)：定义了 $data$, props, set, delete,$watch
- eventsMixin(Vue)：定义了 on, once, off, emit
- lifecycleMixin(Vue)：定义了 _update, $forceUpdate$, destroy
- renderMixin(Vue)：定义了 $nextTick,_render

3.总体流程：

- 模版编译：编译的结果是得到 render 函数并放入配置中。
  - 解析：ast = parse(template.trim(), options)
    - HTML解析器
    - 文本解析器
    - 过滤器解析器。
  - 优化：optimize(ast, options)
    - 在 AST 中标记静态子树：patch 时，可以跳过静态子树，提高性能。
  - 生成：code = generate(ast, options)
    - 把 AST 转换成代码字符串，传入 new Function(code) 中得到 render 函数。
- 实例化：将配置传入构造函数中，实例化一个根组件（Vue）实例或自定义组件（VueComponent）实例。
  - 初始化 _init：
    - 合并选项
    - initLifecycle(vm)：声明组件实例的 $parent, $root, $children, $refs
    - initEvents(vm)：对父组件传入的自定义事件添加监听
    - initRender(vm)：声明了 $slots, $createElement$就是那个h，对 attrs, $listeners 做了响应化处理。
    - callHook(vm, 'beforeCreate')
    - initInjections(vm)：获取祖辈的注入数据
    - initState(vm)：初始化响应式数据 initProps, initMethods, initData, initComputed, initWatch
      - initData：数据响应式，有几个对象数据（包括data）就有几个 Observer 实例，dep 的数量是对象数据个数（包括 data） + data 内所有 key 的数量，几个组件就有几个 Watcher。
        - 数据命名冲突校验
        - 数据代理
        - 执行 observe，传入 data
          - 创建 Observer 实例
            - 创建对象数据的 dep。$set, array 那七个变更数组方法时会使用到这个的 dep 中存放的依赖来做通知更新。
            - 创建每个 key 对应的 dep。
            - 分别做数组和对象的响应化处理。

                - getter：分别对每个 key 的 dep 和对象数据的 dep 做依赖收集，收集的都是组件 Watcher
                - setter：劫持数据变化
                    - dep.notify()：通知更新
                    - 批量异步更新：将 dep 中收集的所有 Watcher 的更新函数批量异步的执行一遍。
                        - watcher.update()
                        - queueWatcher()
                        - nextTick()
                        - timerFunc()
                        - flushSchedulerQueue()
                        - watcher.run()
                        - watcher.get()
                        - updateComponent()
        - initProvide(vm)：给后代提供数据
        - callHook(vm, 'created')
        - 最后判断选项里如果有 el，自动执行 $mount。
- 挂载 $mount
    - 执行 mountComponent
        - callHook(vm, 'beforeMount')
        - 声明更新函数 updateComponent
            - 执行 _render
                - render（配置中的render)
                    - createElement：h 方法，传入 tag, data, children 等
                        - 原生标签：创建 vnode 并返回
                        - 自定义组件：createComponent
                            - 获取组件配置
                            - 根据组件配置，获取组件构造函数
                            - 安装组件管理钩子到该组件的 vnode 上。
                                - init：组件初始化，创建组件实例，挂载。patch 时执行 init。
                                - prepatch：组件更新之前执行，patch 之前的一些工作
                                - insert：组件创建完插入 dom 元素里，调用子组件的 mounted 生命周期
                                - destroy：组件销毁相关工作
                            - 创建 vnode 并返回
            - 执行 _update，传入 vnode。
                - patch
                    - new vnode 不存在就删除
                    - old vnode 不存在就新增
                        - createElm

        - 都存在
            - oldVnode 是原生标签
                - createElm：创建新节点，把 vnode 创建成 DOM 元素，然后递归创建子元素和子组件。
                    - createComponent：如果要创建的是组件，走这个流程
- 获取创建组件 vnode 时安装的 init 组件管理钩子并执行：创建组件实例并挂载。
- insert：子组件 DOM 树插入父组件的 DOM 树上。
                - 原生标签的创建：
- 把 vnode 创建成真实的 DOM，createChildren 递归创建子元素
- insert：子组件 DOM 树插入父组件的 DOM 树上。
            - oldVnode 不是原生标签 && 是同一个 vnode 节点
                - patchVnode：执行 diff 更新。
                  有孩子先比孩子调用 updateChildren，updateChildren 中还会调用 patchVnode，一直向下递归，将每个 vnode 节点都 patch 一遍。
                  每个节点比较的和更新的就是三件事：属性更新，文本更新，子节点更新：
                    - isPatchable(vnode)：节点本身的 patch 操作，属性更新。
                    - 都无子节点：只是文本的替换。
                    - 只有新有子节点：先清空老文本内容，然后为其新增子节点。
                    - 只有老有子节点：移除该节点的所有子节点。
                    - 新老均有子节点：对子节点进行 Diff 操作，调用 updateChildren。
- updateChildren
- 设置双指针，首尾都没有找到相同的节点还是要做双循环。最后根据新老 vnode 的节点剩余情况做相应的新增或删除工作。
- 找到相同的节点调用 patchVnode（递归：深度优先）
- 移动节点位置（实际的 dom 操作），移动指针做下一个节点的对比（同级比较）
            - invokeInsertHook：调用组件管理钩子 insert。里面调用了 mounted 生命周期钩子。
    - 创建组件 Watcher，传入 updateComponent
        - 执行 updateComponent
    - callHook(vm, 'mounted')

工程化

webpack 优化

分析工具

- 速度分析：使用 speed-measure-webpack-plugin
- 体积分析：使用 webpack-bundle-analyzer

构建速度优化

- 使用高版本的 webpack 和 Node.js
- 缩小构建目标
- 优化文件查找路径
- 多进程构建
- 多进程压缩代码
- 利用缓存提升二次构建速度

体积优化

- 代码、图片压缩
- Tree Shaking
- Scope Hoisting
- Code Splitting
- 动态 import 加载异步组件
- 使用 cdn 静态资源
- 动态 polyfill

webpack 原理

Tapable 为 webpack 插件提供了发布订阅的钩子。每个钩子代表一个关键的事件节点。

webpack 就是基于这种发布订阅的一系列的插件运行的事件流。

在 webpack 内部的 compiler 和 compilation 上面做 hooks 的调用。

插件有个 apply 方法，接收一个 compiler 参数。插件里面做 compiler 和 compilation 上的 hooks 的监听。

- 处理配置参数。
- 执行用户配置中的所有插件。
- 根据配置开启 webpack 内部的插件。
- 使用 loader-runner 运行 loaders 进行编译和分析依赖
- 将所有编译好的 js 代码放到 compilation 对象上的 modules 里面。

- 代码优化
- 将 modules 里的代码放到 compilation 对象的 assets 里面去
- 资源生成

手写一个 loader

loader

loader 是一个导出为声明式函数的 javascript 模块，接收资源返回资源：

```JavaScript
const loaderUtils = require("loader-utils");
module.exports = function(source) {
  // 参数获取
  const { name } = loaderUtils.getOptions(this);

  // 异常处理
  // 1.throw new Error('Error');
  // 2.this.callback(new Error('Error'), source);

  // 返回结果
  // 1.return source;
  // 2.this.callback(null, source, 1, 2); 可以返回多个值

  // 异步处理
  const callback = this.async();
  fs.readFile(path.join(__dirname, './demo.txt'), 'utf-8', (err, data) => {
    if (err) {
      callback(err, '');
    }
    callback(null, data);
  });

  // 缓存
  // webpack 中默认开启缓存，可以使用以下方法关闭缓存
  // 缓存生效条件：loader 的结果有确定的输出。有依赖的 loader 无法使用缓存。
  this.cacheable(false);

  // 文件输出
  const url = loaderUtils.interpolateName(this, "[name].[ext]", source);
  this.emitFile(url, source);
};
```

手写一个 plugin

插件是一个类，有一个 apply 方法。

JavaScript

```
// 将一段代码输出到文件里面就可以用 RawSource
const { RawSource } = require("webpack-sources");
class MyPlugin {
  constructor(options) {
    this.options = options;
  }
  apply(compiler) {
    // 插件处理逻辑

    // 插件的错误处理
    // 1.throw new Error('error');
    // 2.通过 compilation 对象的 warnings 和 errors 接收
    //   compilation.warnings.push("warning");
    //   compilation.errors.push("error");

    // 文件写入
    // webpack 的构建流程的文件生成是在 emit 阶段，所以在插件里监听 compiler emit 这个 hooks。
    // 监听这个 hook 之后我们可以获取到 compilation 对象
    // 然后只需要将最终要输出的内容设置到 compilation.assets 对象上面去就可以了
    // 最终webpack生成文件的时候会触发emit，然后读取compilation.assets上的资源内容并输出到磁盘目录
    const { path } = this.options;
    compiler.hooks.emit.tapAsync("MyPlugin", (compilation, callback) => {
      compilation.assets[path] = new RawSource("demo");
      callback();
    });
  }
}
module.exports = MyPlugin;
```