进阶篇

一、JS

1 谈谈变量提升

> 当执行 `JS` 代码时，会生成执行环境，只要代码不是写在函数中的，就是在全局执行环境中，函数中的代码会产生函数执行环境，只此两种执行环境。

```js
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：在生成执行环境时，会有两个阶段。第一个阶段是创建的阶段，`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

```js
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，在第一阶段内存也已经为他开辟好了空间，但是因为这个声明的特性导致了并不能在声明前使用

2 bind、call、apply 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

```js
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'])
```

> `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

3 如何实现一个 bind 函数

对于实现以下几个函数，可以从几个方面思考

- 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?

```js
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))
  }
}
```

4 如何实现一个 call 函数

```js
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
}
```

5 如何实现一个 apply 函数

```js
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
}
```

6 简单说下原型链?

- 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。

7 怎么判断对象类型

- 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。
- `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`

8 箭头函数的特点

```js
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

9 This

```js
function foo() {
    console.log(this.a)
}
var a = 1
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
```

10 async、await 优缺点

> `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。

```js
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
```

- 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，遇到 `await` 就会立即返回一个 `pending` 状态的 `Promise` 对象，暂时返回执行代码的控制权，使得函数外的代码得以继续执行，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

11 generator 原理

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

```js
// 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

```js
// 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();
```

```
      }
    }
  });
}
```

12 Promise

- `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`

13 如何实现一个 Promise

```js
// 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
  // 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);
  }
}
```

14 == 和 ===区别，什么情况用 ==

> 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

```js
// [] 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
```

> `===` 用于判断两者类型和值是否相同。 在开发中，对于后端返回的 `code` ，可以通过 `==` 去判断

15 基本数据类型和引用类型在存储上的差别

> 前者存储在栈上，后者存储在堆上

16 浏览器 Eventloop 和 Node 中的有什么区别

> 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

```js
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`。

```js
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 renderin`

> 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

- 执行同步代码，这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 `UI`
- 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

17 setTimeout 倒计时误差

> `JS` 是单线程的，所以 `setTimeout` 的误差其实是无法被完全解决的，原因有很多，可能是回调中的，有可能是浏览器中的各种事件导致。这也是为什么页面开久了，定时器会不准的原因，当然我们可以通过一定的办法去减少这个误差。

```js
// 以下是一个相对准备的倒计时实现
var period = 60 * 1000 * 60 * 2
var startTime = new Date().getTime();
var count = 0
var end = new Date().getTime() + period
var interval = 1000
var currentInterval = interval
```

```
function loop() {
  count++
  var offset = new Date().getTime() - (startTime + count * interval); // 代码执行所
  var diff = end - new Date().getTime()
  var h = Math.floor(diff / (60 * 1000 * 60))
  var hdiff = diff % (60 * 1000 * 60)
  var m = Math.floor(hdiff / (60 * 1000))
  var mdiff = hdiff % (60 * 1000)
  var s = mdiff / (1000)
  var sCeil = Math.ceil(s)
  var sFloor = Math.floor(s)
  currentInterval = interval - offset // 得到下一次循环所消耗的时间
  console.log('时：'+h, '分：'+m, '毫秒：'+s, '秒向上取整：'+sCeil, '代码执行时间：'+
  setTimeout(loop, currentInterval)
}

setTimeout(loop, currentInterval)
```

18 数组降维

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

> 如果想将一个多维数组彻底的降维，可以这样实现

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

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

19 深拷贝

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

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

```js
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)
复
```

> 在遇到函数、 `undefined` 或者 `symbol` 的时候，该对象也不能正常的序列化

```js
let a = {
    age: undefined,
    sex: Symbol('male'),
    jobs: function() {},
    name: 'yck'
```

```
}
let b = JSON.parse(JSON.stringify(a))
console.log(b) // {name: "yck"}
```

> 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

20 typeof 于 instanceof 区别

> `typeof` 对于基本类型，除了 `null` 都可以显示正确的类型

```js
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`

```js
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'
```

> `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`

```js
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__
    }
}
```

二、浏览器

1 cookie和localSrorage、session、indexDB 的区别

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

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

2 怎么判断页面是否加载完成?

- `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

3 如何解决跨域

> 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

```html
<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

```js
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

- `ORS` 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

```js
// 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')
    }
});
```

4 什么是事件代理

> 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

```html
<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

5 Service worker

> 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

```js
// 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

6 浏览器缓存

> 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

7 浏览器性能问题

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。

- 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。
- 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。
- 判断是否触发了 `media query`
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行 `requestAnimationFrame` 回调
- 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
- 更新界面
- 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。

减少重绘和回流

> 使用 `translate` 替代 `top`

```html
<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

```js
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。

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`

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

三、Webpack

1 优化打包速度

- 减少文件搜索范围
  - 比如通过别名
  - `loader` 的 `test` ， `include & exclude`

- `Webpack4` 默认压缩并行
- `Happypack` 并发调用
- `babel` 也可以缓存编译

2 Babel 原理

- 本质就是编译器，当代码转为字符串生成 `AST`，对 `AST` 进行转变最后再生成新的代码
- 分为三步：词法分析生成 `Token`，语法分析生成 `AST`，遍历 `AST`，根据插件变换相应的节点，最后把 `AST` 转换为代码

3 如何实现一个插件

- 调用插件 `apply` 函数传入 `compiler` 对象
- 通过 `compiler` 对象监听事件

比如你想实现一个编译结束退出命令的插件

```js
apply (compiler) {
  const afterEmit = (compilation, cb) => {
    cb()
    setTimeout(function () {
      process.exit(0)
    }, 1000)
  }

  compiler.plugin('after-emit', afterEmit)
}
}

module.exports = BuildEndPlugin
```