中高级前端大厂面试秘籍，为你保驾护航金三银四，直通大厂(上)

JavaScript

箭头函数

> 没有自己的 `this`，`arguments`，`super` 或 `new.target`。不能用作构造函数。

- 箭头函数不会创建自己的 this，它只会从自己的作用域链的上一层继承 this

  在箭头函数出现之前，每个新定义的函数都有它自己的 this 值（在构造函数的情况下是一个新对象，在严格模式的函数调用中为 undefined，如果该函数被作为“对象方法”调用则为基础对象等）

  > 由于 箭头函数没有自己的 this 指针，通过 call() 或 apply() 方法调用一个函数时，只能传递参数（不能绑定 this---译者注），他们的第一个参数会被忽略。

- 箭头函数不绑定 Arguments 对象。
- 箭头函数不能用作构造器，和 new 一起用会抛出错误。
- 箭头函数没有 prototype 属性。

fetch 取消

- AbortController
- Abortable fetch

symbol

- Symbol()

  Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。

- Symbol.for()

  接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。

  Symbol.for()与Symbol()这两种写法，都会生成新的 Symbol。它们的区别是，前者会被登记在全局环境中供搜索，后者不会。

- Symbol.keyFor()

  Symbol.keyFor方法返回一个已登记的 Symbol 类型值的key。

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

Symbol

1. 概述

ES5 的对象属性名都是字符串，这容易造成属性名
对象，但又想为这个对象添加新的方法（mixin 模
产生冲突。如果有一种机制，保证每个属性的名字
防止属性名的冲突。这就是 ES6 引入`Symbol`的原

ES6 引入了一种新的原始数据类型 上一章 表

JS 继承

- 原型链：本质是重写原型对象
  - 实现

```
function SuperType() {
  this.property = true
}

SuperType.prototype.getSuperValue = function() {
  return this.property
}

function SubType() {
  this.subProperty = false
}

// 继承了 SuperType
SubType.prototype = new SuperType()

SubType.prototype.constructor = SubType

SubType.prototype.getSubValue = function() {
  return this.subProperty
```

```
}

var inst = new SubType()
alert(inst.getSuperValue())
```

- 确定原型和实例关系

  1. instanceof
  2. isPrototypeOf()

- 问题

  1. 包含引用类型值的原型

```
function SuperType() {
  this.colors = ['red', 'blue', 'green']
}

function SubType() {}

// 继承了 SuperType
SubType.prototype = new SuperType()
var inst1 = new SubType()
inst1.colors.push('black')
console.log(inst1.colors) // 'red', 'blue', 'green', 'black'

var inst2 = new SubType()
console.log(inst2.colors) // 'red', 'blue', 'green', 'black'
```

  2. 在创建子类型实例的时候，不能向超类型的构造函数中传递参数。

- 借用构造函数

> 解决了原型中包含引用类型值所带来的问题

> 子类构造函数中向超类构造函数传递参数

  - 实现

```
function SuperType(name) {
  this.name = name
  this.colors = ['red', 'blue', 'green']
}

function SubType(name, age) {
  // 继承了 SuperType, 同时还传递了参数
  SuperType.call(this, name)
  // 实例属性
  this.age = age
}
```

```
var inst1 = new SubType('Nicholas', 29)
inst1.colors.push('black')
console.log(inst1.colors) // 'red', 'blue', 'green', 'black'

var inst2 = new SubType('Jerry', 27)
console.log(inst2.colors) // 'red', 'blue', 'green'
```

- 问题

1. 方法都在构造函数中定义，函数复用无从谈起。
2. 超类原型中定义的方法，子类不可见。

- 组合继承

> 将原型链 和 借用构造函数 结合到一起

- 实现

```
function SuperType(name) {
  this.name = name
  this.colors = ['red', 'blue', 'green']
}

SuperType.prototype.sayName = function() {
  console.log(this.name)
}

function SubType(name, age) {
  // 继承 SuperType 实例属性并传递参数
  SuperType.call(this, name)

  this.age = age
}

// 继承方法
SubType.prototype = new SuperType()
// 修复构造函数
SubType.prototype.constructor = SubType
SubType.prototype.sayAge = function() {
  console.log(this.age)
}

var inst1 = new SubType('Nicholas', 29)
inst1.colors.push('black')
console.log(inst1.colors) // 'red', 'blue', 'green', 'black'
inst1.sayName()
inst1.sayAge()

var inst2 = new SubType('Jerry', 27)
console.log(inst2.colors) // 'red', 'blue', 'green'
```

```
inst2.sayName()
inst2.sayAge()
```

- 问题

  1. 组合继承无论什么情况下，都会调用两次超类构造函数：

     - 一次是在创建子类原型的时候
     - 另一次是在子类构造函数内部调用超类构造函数。

- 原型式继承

> `Object.create()`规范化了原型式继承

- `Object.create(proto, [propertiesObject])`

  `Object.create()` 方法创建一个新对象，使用现有的对象来提供新创建的对象的`__proto__`。

  - proto

    新创建对象的原型对象。

  - propertiesObject

    可选。如果没有指定为 undefined，则是要添加到新创建对象的可枚举属性（即其自身定义的属性，而不是其原型链上的枚举属性）对象的属性描述符以及相应的属性名称。这些属性对应 `Object.defineProperties()` 的第二个参数。

  - 实现

```
if (typeof Object.create !== 'function') {
  Object.create = function(proto, propertiesObject) {
    if (typeof proto !== 'object' && typeof proto !==
'function') {
      throw new TypeError('Object prototype may only be an
Object: ' + proto)
    } else if (proto === null) {
      throw new Error(
        "This browser's implementation of Object.create is a
shim and doesn't support 'null' as the first argument."
      )
    }

    if (typeof propertiesObject != 'undefined')
      throw new Error(
        "This browser's implementation of Object.create is a
shim and doesn't support a second argument."
      )

    function F() {}
    F.prototype = proto
```

```
    return new F()
  }
}
```

- 问题

  在没有必要兴师动众的创建构造函数，而只是想让一个对象与另一个对象保持类似的情况下，原型式继承是完全可以胜任的。但是，**包含引用类型值的属性始终都会共享相应的值。**

- 寄生式继承

> 创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象。

  - 实现

```
function createAnother(original) {
  // 通过调用函数创建一个新对象
  var clone = Object.create(original)
  // 以某种方式来增强新对象
  clone.sayHi = function() {
    console.log('Hi')
  }
  return clone
}

var person = {
  name: 'Nicholas',
  friends: ['red', 'blue', 'green']
}
var anotherPerson = createAnother(person)
anotherPerson.sayHi()
```

  - 问题

    1. 不能做到函数复用而降低效率

- 寄生组合式继承

> 组合继承无论什么情况下，都会调用两次超类构造函数：

> 一次是在创建子类原型的时候
> 另一次是在子类构造函数内部调用超类构造函数。

```
// 组合继承🌰

function SuperType(name) {
  this.name = name
  this.colors = ['red', 'blue', 'green']
}
```

```
SuperType.prototype.sayName = function() {
  console.log(this.name)
}

function SubType(name, age) {
  SuperType.call(this, name) // 第二次调用 SuperType()

  this.age = age
}

SubType.prototype = new SuperType() // 第一次调用 SuperType()
SubType.prototype.constructor = SubType
SubType.prototype.sayAge = funciton() {
  console.log(this.age)
}
```

> 第一次调用在原型上有两个属性 name 和 colors
> 调用子类构造函数的时候，又会调用 超类构造函数， 又会在新对象上创建实例属性 name 和 colors，于是原型链上的两个同名属性就被屏蔽了

- 实现

```
function inheritPrototype(subType, superType) {
  var prototype = Object.create(superType.prototype) // 创建对象
  prototype.constructor = subType // 增强对象
  subType.prototype = prototype // 指定对象
}
```

```
function SuperType(name) {
  this.name = name
  this.colors = ['red', 'blue', 'green']
}

SuperType.prototype.sayName = function() {
  console.log(this.name)
}

function SubType(name, age) {
  SuperType.call(this, name) // 第二次调用 SuperType()

  this.age = age
}

inheritPrototype(SubType, SuperType)
SubType.prototype.sayAge = function() {
  console.log(this.age)
}
```

- 引用类型最理想的继承范式

Class 的继承

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

4. 类的 prototype 属性和__proto__属

大多数浏览器的 ES5 实现之中，每一个对象都有 `__pro` `prototype` 属性。Class 作为构造函数的语法糖，同时有 性，因此同时存在两条继承链。

（1）子类的 `__proto__` 属性，表示构造函数的继承，总

（2）子类 `prototype` 属性的 `__proto__` 属性，表示方法 `prototype` 属性。

```
class A {
}

class B extends A {
}

B.__proto__ === A // true
B.prototype.__proto__ === A.prototype // true
```

上面代码中，子类 `B` 的 `__proto__` 属性指向父类 `A`，子类 属性指向父类 `A` 的 `prototype` 属性

上一章 下一章

instanceof

> `instanceof` 运算符用于测试构造函数的 `prototype` 属性是否出现在对象的原型链中的任何位置

- 实现原理

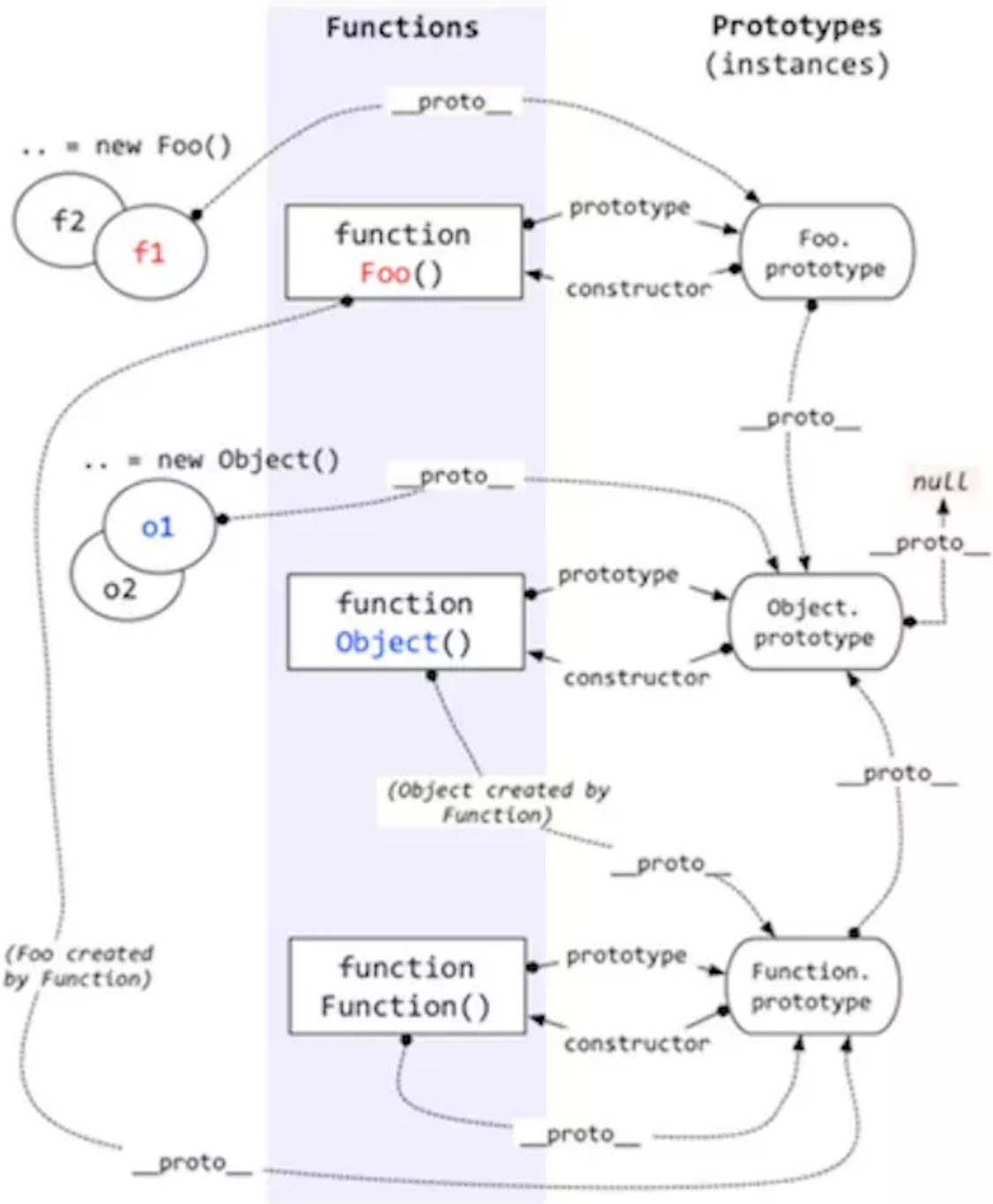
  instanceof 主要的实现原理就是只要右边变量的 prototype 在左边变量的原型链上即可

```
function new_instance_of(leftVaule, rightVaule) {
  let rightProto = rightVaule.prototype // 取右表达式的 prototype 值
  leftVaule = leftVaule.__proto__ // 取左表达式的__proto__值
  while (true) {
    if (leftVaule === null) {
      return false
    }
    if (leftVaule === rightProto) {
      return true
    }
    leftVaule = leftVaule.__proto__
  }
}
```

- 几个有趣的例子

```
function Foo() {}

Object instanceof Object // true
Function instanceof Function // true
Function instanceof Object // true
Foo instanceof Foo // false
Foo instanceof Object // true
Foo instanceof Function // true
```



- `Object instanceof Object`

Object 的 prototype 属性是 `Object.prototype`, 而由于 Object 本身是一个函数，由 Function 所创建，所以 `Object.__proto__` 的值是 `Function.prototype`，而 `Function.prototype` 的 `__proto__` 属性是 `Object.prototype`，所以我们可以判断出，`Object instanceof Object` 的结果是 true 。

typeof

typeof 在判断一个 object 的数据的时候只能告诉我们这个数据是 object, 而不能细致的具体到是哪一种 object

最好是用 typeof 来判断基本数据类型（包括 symbol）和 function，避免对 null 的判断。

```
typeof undefined // "undefined"
typeof true // "boolean"
typeof function() {} // "function"
typeof {} // "object"
typeof [] // "object"
typeof null // "object"
typeof new String('abc') // "object"
typeof new Date() // "object"
```

还有一个不错的判断类型的方法，就是 Object.prototype.toString

```
Object.prototype.toString.call(1) // "[object Number]"
Object.prototype.toString.call('hi') // "[object String]"
Object.prototype.toString.call({ a: 'hi' }) // "[object Object]"
Object.prototype.toString.call([1, 'a']) // "[object Array]"
Object.prototype.toString.call(true) // "[object Boolean]"
Object.prototype.toString.call(() => {}) // "[object Function]"
Object.prototype.toString.call(null) // "[object Null]"
Object.prototype.toString.call(undefined) // "[object Undefined]"
Object.prototype.toString.call(Symbol(1)) // "[object Symbol]"
```

```
let class2Type = {}
['Array', 'Date','RegExp', 'Error', 'Object'].forEach(type =>
class2Type[`[object ${type}]`] = type.toLowerCase())

function type(obj) {
  if(obj === null) return String(null)
  return typeof obj === 'object' ?
class2Type[Object.prototype.toString.call(obj)] || 'object' : typeof obj
}
```

数字千分位处理，正则和非正则都要实现(千位加逗号)

- numObj.toLocaleString([locales [, options]])

```
// 方法一
var num = 234982347.73
console.log(num.toLocaleString())

// 方法二
var num = 234982347.73
num.toString().replace(/^\d+/g, m => m.replace(/(\d{1,3})(?=
(?:\d{3})+$)/g, '$1,'))
```

- 正则

字符类别（Character Classes）	
字符	含义
`.`	(点号，小数点) 匹配任意单个字符，但是行结束符除外：`\n` `\r` `\u2028` 或 `\u2029`。 在字符集中，点(.)失去其特殊含义，并匹配一个字面点(.)。 需要注意的是，`m` 多行（multiline）标志不会改变点号的表现。因此为了匹配多行中的字符集，可使用`[^]` （当然你不是打算用在旧版本 IE 中），它将会匹配任意字符，包括换行符。 例如，`/.y/` 匹配 "yes make my day" 中的 "my" 和 "ay"，但是不匹配 "yes"。
`\d`	匹配任意阿拉伯数字。等价于`[0-9]`。 例如，`/\d/` 或 `/[0-9]/` 匹配 "B2 is the suite number." 中的 '2'。
`\D`	匹配任意一个不是阿拉伯数字的字符。等价于`[^0-9]`。 例如，`/\D/` 或 `/[^0-9]/` 匹配 "B2 is the suite number." 中的 'B'。
`\w`	匹配任意来自基本拉丁字母表中的字母数字字符，还包括下划线。等价于 `[A-Za-z0-9_]`。 例如，`/\w/` 匹配 "apple" 中的 'a'，"$5.28" 中的 '5' 和 "3D" 中的 '3'。
`\W`	匹配任意不是基本拉丁字母表中单词（字母数字下划线）字符的字符。等价于 `[^A-Za-z0-9_]`。 例如，`/\W/` 或 `/[^A-Za-z0-9_]/` 匹配 "50%" 中的 '%'。
`\s`	匹配一个空白符，包括空格、制表符、换页符、换行符和其他 Unicode 空格。 等价于 `[ \f\n\r\t\v\u00a0\u1680\u180e\u2000\u2001\u2002\u2003\u2004 \u2005\u2006\u2007\u2008 \u2009\u200a\u2028\u2029\u202f\u205f \u3000]`。 例如 `/\s\w*/` 匹配 "foo bar" 中的 ' bar'。
`\S`	匹配一个非空白符。等价于 `[^ \f\n\r\t\v\u00a0\u1680\u180e\u2000\u2001\u2002\u2003\u2004 \u2005\u2006 \u2007\u2008\u2009\u200a\u2028\u2029\u202f\u205f\u3000]` 。 例如，`/\S\w*/` 匹配 "foo bar" 中的 'foo'。
`\t`	匹配一个水平制表符（tab）
`\r`	匹配一个回车符（carriage return）
`\n`	匹配一个换行符（linefeed）
`\v`	匹配一个垂直制表符（vertical tab）

`\f`	匹配一个换页符（form-feed）
`[\b]`	匹配一个退格符（backspace）（不要与 `\b` 混淆）
`\0`	匹配一个 NUL 字符。不要在此后面跟小数点。
`\cX`	`X` 是 A - Z 的一个字母。匹配字符串中的一个控制字符。 例如，`/\cM/` 匹配字符串中的 control-M。
`\xhh`	匹配编码为 `hh` （两个十六进制数字）的字符。
`\uhhhh`	匹配 Unicode 值为 `hhhh` （四个十六进制数字）的字符。
`\`	对于那些通常被认为字面意义的字符来说，表示下一个字符具有特殊用处，并且不会被按照字面意义解释。 例如 `/b/` 匹配字符 'b'。在 b 前面加上一个反斜杠，即使用 `/\b/`，则该字符变得特殊，以为这匹配一个单词边界。 或 对于那些通常特殊对待的字符，表示下一个字符不具有特殊用途，会被按照字面意义解释。 例如，* 是一个特殊字符，表示匹配某个字符 0 或多次，如 `/a*/` 意味着 0 或多个 "a"。 为了匹配字面意义上的 `*` ，在它前面加上一个反斜杠，例如，`/a\*/` 匹配 'a*'。

字符集合（Character Sets）	
字符	含义
`[xyz]`	一个字符集合，也叫字符组。匹配集合中的任意一个字符。你可以使用连字符'-'指定一个范围。 例如，[abcd] 等价于 [a-d]，匹配"brisket"中的'b'和"chop"中的'c'。
`[^xyz]`	一个反义或补充字符集，也叫反义字符组。也就是说，它匹配任意不在括号内的字符。你也可以通过使用连字符 '-' 指定一个范围内的字符。 例如，`[^abc]` 等价于 `[^a-c]`。 第一个匹配的是 "bacon" 中的'o' 和 "chop" 中的 'h'。

边界（Boundaries）	
字符	含义
`^`	匹配输入开始。如果多行（multiline）标志被设为 true，该字符也会匹配一个断行（line break）符后的开始处。 例如，`/^A/` 不匹配 "an A" 中的 "A"，但匹配 "An A" 中的 "A"。
`$`	匹配输入结尾。如果多行（multiline）标志被设为 true，该字符也会匹配一个断行（line break）符的前的结尾处。 例如，`/t$/` 不匹配 "eater" 中的 "t"，但匹配 "eat" 中的 "t"。
`\b`	匹配一个零宽单词边界（zero-width word boundary），如一个字母与一个空格之间。 （不要和 `[\b]` 混淆） 例如，`/\bno/` 匹配 "at noon" 中的 "no"，`/ly\b/` 匹配 "possibly yesterday." 中的 "ly"。
`\B`	匹配一个零宽非单词边界（zero-width non-word boundary），如两个字母之间或两个空格之间。 例如，`/\Bon/` 匹配 "at noon" 中的 "on"，`/ye\B/` 匹配 "possibly yesterday." 中的 "ye"。

分组（Grouping）与反向引用（back references）	
字符	含义
`(x)`	匹配 `x` 并且捕获匹配项。 这被称为捕获括号（capturing parentheses）。 例如，`/(foo)/` 匹配且捕获 "foo bar." 中的 "foo"。被匹配的子字符串可以在结果数组的元素 `[1], ..., [n]` 中找到，或在被定义的 `RegExp` 对象的属性 `$1, ..., $9` 中找到。 捕获组（Capturing groups）有性能惩罚。如果不需再次访问被匹配的子字符串，最好使用非捕获括号（non-capturing

	parentheses），见下面。
`\n`	`n` 是一个正整数。一个反向引用（back reference），指向正则表达式中第 n 个括号（从左开始数）中匹配的子字符串。 例如，`/apple(,)\sorange\1/` 匹配 "apple, orange, cherry, peach." 中的 "apple,orange,"。一个更全面的例子在该表格下面。
`(?:x)`	匹配 `x` 不会捕获匹配项。这被称为非捕获括号（non-capturing parentheses）。匹配项不能够从结果数组的元素 `[1], ..., [n]` 或已被定义的 `RegExp` 对象的属性 `$1, ..., $9` 再次访问到。
数量词（Quantifiers）	
字符	含义
`x*`	匹配前面的模式 x 0 或多次。 例如，`/bo*/` 匹配 "A ghost booooed" 中的 "boooo"，"A bird warbled" 中的 "b"，但是不匹配 "A goat grunted"。
`x+`	匹配前面的模式 x 1 或多次。等价于 `{1,}`。 例如，`/a+/` 匹配 "candy" 中的 "a"，"caaaaaaandy" 中所有的 "a"。
`x*?` `x+?`	像上面的 * 和 + 一样匹配前面的模式 x，然而匹配是最小可能匹配。 例如，`/".*?"/` 匹配 '"foo" "bar"' 中的 '"foo"'，而 * 后面没有 ? 时匹配 '"foo" "bar"'。
`x?`	匹配前面的模式 x 0 或 1 次。 例如，`/e?le?/` 匹配 "angel" 中的 "el"，"angle" 中的 "le"。 如果在数量词 `*`、`+`、`?` 或 `{}`，任意一个后面紧跟该符号（?），会使数量词变为非贪婪（ non-greedy），即匹配次数最小化。反之，默认情况下，是贪婪的（greedy），即匹配次数最大化。 在使用于向前断言（lookahead assertions）时，见该表格中 `(?=)`、`(?!)` 和 `(?:)` 的说明。
`x(?=y)`	只有当 `x` 后面紧跟着 `y` 时，才匹配 `x`。例如，`/Jack(?=Sprat)/` 只有在 'Jack' 后面紧跟着 'Sprat' 时，才会匹配它。`/Jack(?=Sprat|Frost)/` 只有在 'Jack' 后面紧跟着 'Sprat' 或 'Frost' 时，才会匹配它。然而，'Sprat' 或 'Frost' 都不是匹配结果的一部分。
`x(?!y)`	只有当 `x` 后面不是紧跟着 `y` 时，才匹配 `x`。例如，`/\d+(?!\.)/` 只有当一个数字后面没有紧跟着一个小数点时，才会匹配该数字。 `/\d+(?!\.)/.exec("3.141")` 匹配 141 而不是 3.141。
`x|y`	匹配 `x` 或 `y` 例如，`/green|red/` 匹配 "green apple" 中的 'green'，"red apple." 中的 'red'。
`x{n}`	`n` 是一个正整数。前面的模式 x 连续出现 n 次时匹配。 例如，`/a{2}/` 不匹配 "candy," 中的 "a"，但是匹配 "caandy," 中的两个 "a"，且匹配 "caaandy." 中的前两个 "a"。
`x{n,}`	`n` 是一个正整数。前面的模式 x 连续出现至少 n 次时匹配。 例如，`/a{2,}/` 不匹配 "candy" 中的 "a"，但是匹配 "caandy" 和 "caaaaaaandy." 中所有的 "a"。
`x{n,m}`	`n` 和 `m` 为正整数。前面的模式 x 连续出现至少 n 次，至多 m 次时匹配。 例如，`/a{1,3}/` 不匹配 "cndy"，匹配 "candy," 中的 "a"，"caandy," 中的两个 "a"，匹配 "caaaaaaandy" 中的前面三个 "a"。注意，当匹配 "caaaaaaandy" 时，即使原始字符串拥有更多的 "a"，匹配项也是 "aaa"。
断言（Assertions）	
字符	含义
`x(?=y)`	仅匹配被y跟随的x。 举个例子，`/Jack(?=Sprat)/`，如果"Jack"后面跟着sprat，则匹配之。

	`/Jack(?=Sprat|Frost)/`，如果"Jack"后面跟着"Sprat"或者"Frost"，则匹配之。但是，"Sprat" 和"Frost" 都不会在匹配结果中出现。
`x(?!y)`	仅匹配不被y跟随的x。 举个例子，`/\d+(?!\.)/` 只会匹配不被点（.）跟随的数字。 `/\d+(?!\.)/.exec('3.141')` 匹配"141"，而不是"3.141"

- 标识符：
  - g: 全局
  - i: 忽略大小写
  - m: 多行模式， 在到达一行文本末尾时还会继续查找下一行中是否存在与模式匹配的项。
- 元字符：
  - `([{\^$|}?*+.])`
  - 所有的元字符都必须经过转义
- 使用 `RegExp` 构造函数
  - 由于 `RegExp` 构造函数的模式参数是个字符串，所以在某些情况下要对字符串进行双重转义

字面量模式	等价的字符串
`/\[bc\]at/`	`\\[bc\\]at`
`/\.at/`	`\\.at`
`/name\/age/`	`name\\/age`
`/\d.\d{1,2}/`	`\\d.\\d{1,2}`
`/\w\\hello\\123/`	`\\w\\\\hello\\\\123`

- ES5 明确规定， 使用正则表达式字面量必须像直接调用 `RegExp` 构造函数一样，每次都创建新的 `RegExp` 实例。
- 实例属性
  - global
  - ignoreCase
  - multiline
  - source: 正则表达式的字符串表示，按照**字面量形式**而非传入构造函数的字符串模式
  - lastIndex: 开始搜索下一个匹配项的字符位置，起始 0
- 实例方法
  - exec(): 专门为捕获组而设计，返回包含第一个匹配项信息的数组， 没有匹配项返回 null
    - 返回数组属性
      - 第一项是与整个模式匹配的字符串
      - 其他项是与模式中的捕获组匹配的字符串
      - index: 匹配项在字符串中的位置

        - input: 应用正则表达式的字符串
      - 模式中设置了 g 标志,每次也只返回一个匹配项
        - 同一个字符串多次调用 exec(),每次调用都会在字符串中继续查找新的匹配项
    - test(): 是否匹配
    - toLocaleString(): 返回正则表达式字面量
    - toString(): 返回正则表达式字面量

- 构造函数属性

长属性名	短属性名	说明
input	$_	最近一次要匹配的字符串
lastMatch	$&	最近一次匹配项
lastParen	$+	最近一次匹配的捕获组
leftContext	$`	input 字符串中 lastMatch 之前的文本
rightContext	$'	input字符串中lastMatch之后的文本
multiline	$*	布尔值，是否所有的表达式都使用多行模式
$1,$2,...,$9	存储第一到第九个捕获组，调用 exec()或 test()时，这些属性自动填充	

实现一个 bind 函数 *

bind()方法创建一个新的函数，在调用时设置 this 关键字为提供的值。并在调用新函数时，将给定参数列表作为原函数的参数序列的前若干项。

```
function.bind(thisArg[, arg1[, arg2[, ...]]])
```

- thisArg

  调用绑定函数时作为 this 参数传递给目标函数的值。 如果使用 new 运算符构造绑定函数，则忽略该值。当使用 bind 在 setTimeout 中创建一个函数（作为回调提供）时，作为 thisArg 传递的任何原始值都将转换为 object。**如果 bind 函数的参数列表为空，执行作用域的 this 将被视为新函数的 thisArg。**

- arg1, arg2, ...

  当目标函数被调用时，预先添加到绑定函数的参数列表中的参数。

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function(oThis) {
    if (typeof this !== 'function') {
      throw new TypeError('Function.prototype.bind - what is trying to be
bound is not callable')
    }
```

```
    var args = Array.prototype.slice.call(arguments, 1),
      functionToBind = this,
      functionBound = function() {
        var bindArgs = Array.prototype.slice.call(arguments)
        // this instanceof fBound === true时,说明返回的fBound被当做new的构造函数调用
        return functionToBind.apply(
          this instanceof functionBound ? this : oThis,
          // 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的
          args.concat(bindArgs)
        )
      }

    // 我们直接将 fBound.prototype = this.prototype，我们直接修改 fBound.prototype 的时候，也会直接修改绑定函数的 prototype。这个时候，我们可以通过一个空函数来进行中转：

    // 维护原型关系(原型链继承)
    var fNOP = function() {}
    if (this.prototype) {
      fNOP.prototype = this.prototype
    }

    functionBound.prototype = new fNOP()

    // functionbound.prototype = Object.create(this.prototype);

    return functionBound
  }
}
```

JavaScript 中的对象拷贝

- 浅拷贝
  - Object.assign()、扩展运算符(...)
    1. 复制对象的可枚举属性
    2. 可以拷贝方法，和循环引用
    3. 复制的嵌套属性是引用，共享
- 深拷贝
  - JSON.parse(JSON.stringify(obj))
    原型改变，不能复制对象方法，不能复制循环引用
  - 递归遍历属性，复制属性 Object.getOwnPropertyDescriptor

深入理解 new 操作符

```
  new Animal() {
    const obj = {};
```

```
    obj.__proto__ = Animal.prototype;

    const result = Animal.apply(obj, arguments);
    return typeof result === 'object' ? result : obj;

  }
```

深入理解 Babel 原理及其使用

前端基础进阶（二）：执行上下文详细图解

什么是作用域和执行上下文

Javascript 函数声明的优先级高于变量声明的优先级，但不会覆盖变量赋值

Set 和 Map 数据结构

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

Set 和 Map 数据结构

1. Set

基本用法

ES6 提供了新的数据结构 Set。它类似于数组，但是成

`Set` 本身是一个构造函数，用来生成 Set 数据结构。

```
const s = new Set();
```

上一章 下一章

异步解决方案

Promise

- 【剖析 Promise 内部结构，一步一步实现一个完整的、能通过所有 Test case 的 Promise 类】
- 深入 Promise(一)——Promise 实现详解

- 深入 Promise(二)——进击的 Promise
- 深入 Promise(三)——命名 Promise

- 实现

```
function Promise(executor) {
  var self = this
  self.status = 'pending'
  self.value = undefined
  self.onResolveCallback = []
  self.onRejectCallback = []

  function resolve(value) {
    if (value instanceof Promise) {
      return value.then(resolve, reject)
    }
    setTimeout(function() {
      if (self.status === 'pending') {
        self.status = 'fulfilled'
        self.value = value
        for (var i = 0; i < self.onResolveCallback.length; i++) {
          self.onResolveCallback[i](value)
        }
      }
    })
  }

  function reject(reason) {
    setTimeout(function() {
      if (self.status === 'pending') {
        self.status = 'rejected'
        self.value = reason
        for (var i = 0; i < self.onRejectCallback.length; i++) {
          self.onRejectCallback[i](reason)
        }
      }
    })
  }

  try {
    executor(resolve, reject)
  } catch (e) {
    reject(e)
  }
}

function resolvePromise(promise2, x, resolve, reject) {
  var then
  var thenCalledOrThrow = false

  if (promise2 === x) {
    return reject(new TypeError('Chaining cycle detected for
```

```
promise!''))
  }

  if (x instanceof Promise) {
    if (x.status === 'pending') {
      x.then(function(value) {
        resolvePromise(promise2, value, resolve, reject)
      }, reject)
    } else {
      x.then(resolve, reject)
    }
    return
  }

  if (x !== null && (typeof x === 'object' || typeof x ===
'function')) {
    try {
      then = x.then
      if (typeof then === 'function') {
        then.call(
          x,
          function rs(y) {
            if (thenCalledOrThrow) return
            thenCalledOrThrow = true
            return resolvePromise(promise2, y, resolve, reject)
          },
          function rj(r) {
            if (thenCalledOrThrow) return
            thenCalledOrThrow = true
            return reject(r)
          }
        )
      } else {
        resolve(x)
      }
    } catch (e) {
      if (thenCalledOrThrow) return
      thenCalledOrThrow = true
      return reject(e)
    }
  } else {
    resolve(x)
  }
}

Promise.prototype.then = function(onResolved, onRejected) {
  var self = this
  var promise2

  onResolved =
    typeof onResolved === 'function'
      ? onResolved
      : function(value) {
          return value
```

```
        }
    onRejected =
      typeof onRejected === 'function'
        ? onRejected
        : function(reason) {
            throw reason
        }

    if (self.status === 'fulfilled') {
      return (promise2 = new Promise(function(resolve, reject) {
        setTimeout(function() {
          try {
            var x = onResolved(self.value)
            resolvePromise(promise2, x, resolve, reject)
          } catch (e) {
            reject(e)
          }
        })
      }))
    }

    if (self.status === 'rejected') {
      return (promise2 = new Promise(function(resolve, reject) {
        setTimeout(function() {
          try {
            var x = onReject(self.value)
            resolvePromise(promise2, x, resolve, reject)
          } catch (e) {
            reject(e)
          }
        })
      }))
    }

    if (self.status === 'pending') {
      return (promise2 = new Promise(function(resolve, reject) {
        self.onResolvedCallback.push(function(value) {
          try {
            var x = onResolved(value)
            resolvePromise(promise2, x, resolve, reject)
          } catch (e) {
            reject(e)
          }
        })

        self.onRejectedCallback.push(function(reason) {
          try {
            var x = onReject(reason)
            resolvePromise(promise2, x, resolve, reject)
          } catch (e) {
            reject(e)
          }
        })
      }))
```

```
    }
  }

  Promise.prototype.catch = function(onReject) {
    return this.then(null, onReject)
  }

  Promise.deferred = Promise.defer = function() {
    var dfd = {}
    dfd.promise = new Promise(function(resolve, reject) {
      dfd.resolve = resolve
      dfd.reject = reject
    })
    return dfd
  }
```

generator

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

Generator 函数的语法

1. 简介

上一章 下一章

async/await

```
function spawn(genF) {
  return new Promise(function(resolve, reject) {
    const gen = genF();
```

```
    function step(nextF) {
      let next;
      try{
        next = nextF();
      }catch(err) {
        return reject(err);
      }
      if(next.done) {
        return resolve(next.value);
      }

      Promise.resolve(next.value).then(function(v) {
        step(function() { return gen.next(v); });
      }, function(err) {
        step(function() { return gen.throw(err); });
      });
    }

    step(function() { return gen.next(undefined); });
  })
}
```

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

4. async 函数的实现原理

async 函数的实现原理，就是将 Generator 函数和自动

```
async function fn(args) {
  // ...
}

// 等同于

function fn(args) {
  return spawn(function* () {
    // ...
  });
}
```

所有的`async`函数都可以写成上面的第二种形式，其中的

下面给出`spawn`函数的实现，基本就是前文自动执行器的

```
function spawn(genF) {
  return new Promise(function(resolve, reject
    const gen = genF();
    function step(nextF) {
```

上一章 下一章

promise 和 async 区别

防抖与节流

- 防抖

  将多次高频操作优化为只在最后一次执行，通常使用的场景是：用户输入，只需再输入完成后做一次输入校验即可。

  lodash debounce

- 节流

  每隔一段时间后执行一次，也就是降低频率，将高频操作优化成低频操作，通常使用场景: 滚动条事件或者 resize 事件，通常每隔 100~500 ms执行一次即可。

  lodash throttle

this 指向

import 和 require 的区别

- import 是关键字， 而 require 是个局部变量

  使用 require 的时候，其实会将 module 的代码进行包装，变成如下样子的代码：

```
function (exports, require, module, __filename, __dirname) {
  const m = 1;
  module.exports.m = m;
}
```

- ES6 模块的设计思想是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS 和 AMD 模块，都只能在运行时确定这些东西。

```
// CommonJS模块
let { stat, exists, readFile } = require('fs')

// 等同于
let _fs = require('fs')
let stat = _fs.stat
let exists = _fs.exists
let readfile = _fs.readfile
```

> 上面代码的实质是整体加载 fs 模块（即加载 fs 的所有方法），生成一个对象（_fs），然后再从这个对象上面读取 3 个方法。这种加载称为“运行时加载”，因为只有运行时才能得到这个对象，导致完全没办法在编译时做“静态优化”。

> ES6 模块不是对象，而是通过 export 命令显式指定输出的代码，再通过 import 命令输入。

```
// ES6模块
import { stat, exists, readFile } from 'fs'
```

> 上面代码的实质是从 fs 模块加载 3 个方法，其他方法不加载。这种加载称为“编译时加载”或者静态加载，即 ES6 可以在编译时就完成模块加载，效率要比 CommonJS 模块的加载方式高。当然，这也导致了没法引用 ES6 模块本身，因为它不是对象。

- export 语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

```
export default function() {}
=>
function a(){}
export { a as default }

================华丽分割线====================

import a from './d';
=>
import { default as a } from './d'
```

1. CommonJS 还是 ES6 Module 输出都可以看成是一个具备多个属性或者方法的对象；

- require

  理论上可以运用在代码的任何地方，甚至不需要赋值给某个变量之后再使用

```
require('./a')() // a模块是一个函数，立即执行a模块函数
var data = require('./a').data // a模块导出的是一个对象
var a = require('./a')[0] // a模块导出的是一个数组
```

ES6 模块与 CommonJS 模块的差异

ECMAScript 6 入门

作者：阮一峰

目录

2. ES6 模块与 CommonJS 模块的差异

讨论 Node 加载 ES6 模块之前，必须了解 ES6 模块与

它们有两个重大差异。

- CommonJS 模块输出的是一个值的拷贝，ES6
- CommonJS 模块是运行时加载，ES6 模块是编

第二个差异是因为 CommonJS 加载的是一个对象（即
有在脚本运行完才会生成。而 ES6 模块不是对象，它的
码静态解析阶段就会生成。

下面重点解释第一个差异。

CommonJS 模块输出的是值的拷贝，也就是说，一旦
不到这个值。请看下面这个模块文件`lib.js`的例子。

```
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
```

上一章 下一章

script 属性 defer 和 async 区别

defer 要等到整个页面在内存中正常渲染结束（DOM 结构完全生成，以及其他脚本执行完成），才会执行；async 一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，defer 是“渲染完再执行”，async 是“下载完就执行”。另外，如果有多个 defer 脚本，会按照它们在页面出现的顺序加载，而多个 async 脚本是不能保证加载顺序的。

浏览器

跨域

- 前端常见跨域解决方案（全）

跨页面通信的各种姿势

深入浅出浏览器渲染原理

Event loop

- 带你彻底弄懂Event Loop
- 浏览器事件循环机制（event loop）
- JavaScript 运行机制详解：再谈Event Loop

Linux IO模式及 select、poll、epoll详解

V8引擎中的垃圾回收机制

- 浅谈V8引擎中的垃圾回收机制

浏览器缓存

- 彻底搞懂浏览器缓存机制
- HTTP 缓存
- Cache-Control
    - no-cache: 告诉浏览器、缓存服务器，不管本地副本是否过期，使用资源副本前，一定要到源服务器进行副本有效性校验。

    - must-revalidate：告诉浏览器、缓存服务器，本地副本过期前，可以使用本地副本；本地副本一旦过期，必须去源服务器进行有效性校验。

DNS解析过程及DNS优化

前端安全

- 前端安全知多少
- 如何防止XSS攻击?
- 浅谈CSRF攻击方式

算法

- 前端笔试&面试爬坑系列---算法

CSS

BFC

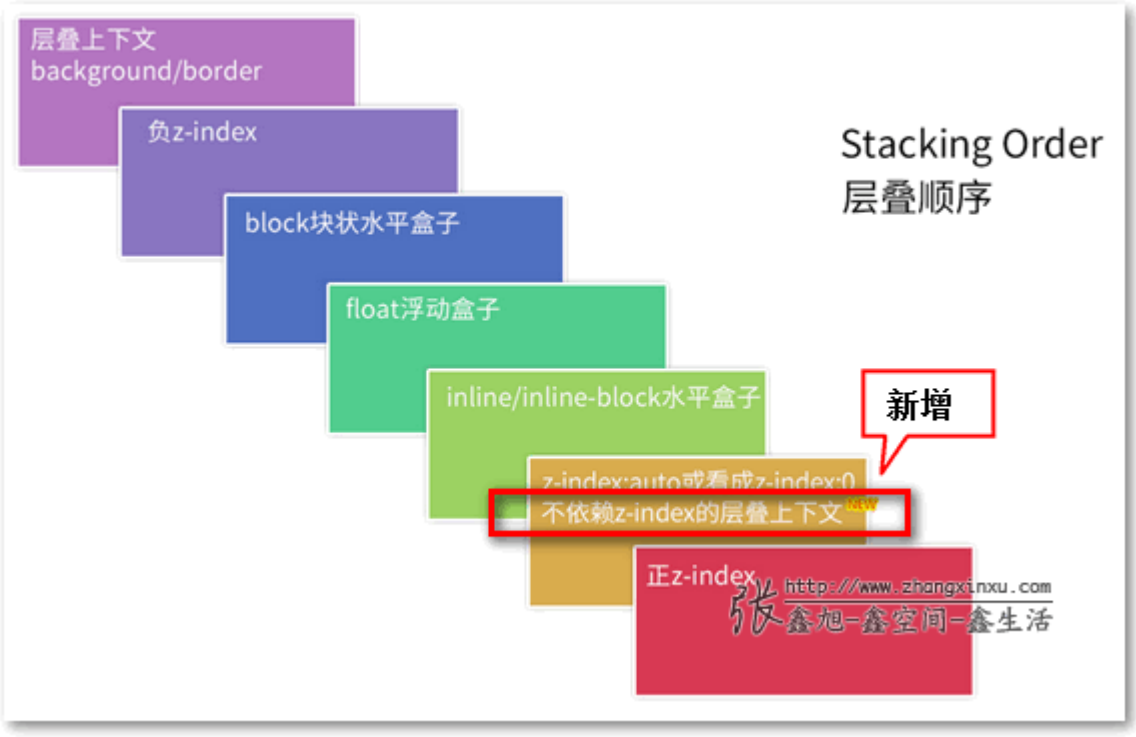
- BFC(块格式化上下文)
- 关于 CSS-BFC 深入理解
- 10 分钟理解 BFC 原理

清除浮动

- 清除浮动（clearfix）的常见方法
- clearfix（清除浮动）

层叠上下文

- 深入理解 CSS 中的层叠上下文和层叠顺序



居中方法

小tips:了解CSS/CSS3原生变量var

CSS实现长宽比的几种方案

- CSS 实现自适应正方形

```
  <div style="border: 1px solid; width: 10vmin; height: 10vmin;">
</div>

  <div style="border: 1px solid; width: 30%; height: 0; padding-
bottom: 30%;"></div>

  <style>
    div::after {
      content: '';
      display: block;
      margin-top: 100%;
    }
  </style>
  <div style="border: 1px solid; width: 30%; overflow: hidden;"></div>
```

- CSS3技巧之形状（椭圆）（border-raduis）

```
  <div style="width: 200px; height: 100px; border-raduis: 100px 50px;
border: 1px solid;"></div>
```

- 三角形实现

伪类与伪元素

css 布局

- css 网页的几种布局
- CSS 布局说——可能是最全的

网络层

http

- HTTP 协议
- HTTP 请求方法：GET、HEAD、POST、PUT、DELETE、CONNECT、OPTIONS、TRACE、PATCH
- HTTP 中 GET 与 POST 的区别

  GET和POST本质上就是TCP链接，并无差别。但是由于HTTP的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。

  1. GET在浏览器回退时是无害的，而POST会再次提交请求。
  2. GET产生的URL地址可以被Bookmark，而POST不可以。
  3. GET请求会被浏览器主动cache，而POST不会，除非手动设置。
  4. GET请求只能进行url编码，而POST支持多种编码方式。
  5. GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
  6. GET请求在URL中传送的参数是有长度限制的，而POST么有。
  7. 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
  8. GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
  9. GET参数通过URL传递，POST放在Request body中。
  10. **GET产生一个TCP数据包；POST产生两个TCP数据包。**
      对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；
      而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

https

- HTTPS 原理详解
- HTTPS

TCP

- 三次握手的误解与错误类比(RFC解读)

  TCP 需要 seq 序列号来做可靠重传或接收，而避免连接复用时无法分辨出 seq 是延迟或者是旧链接的 seq，因此需要三次握手来约定确定双方的 ISN（初始 seq 序列号）。

- TCP 三次握手
- TCP 三次握手、四次挥手
- 通俗大白话来理解TCP协议的三次握手和四次分手
- TCP的滑动窗口与拥塞窗口
- TCP 滑动窗口（发送窗口和接收窗口）
- 解析TCP之滑动窗口(动画演示)
- TCP-IP详解：滑动窗口（Sliding Window）
- TCP拥塞控制-慢启动、拥塞避免、快重传、快启动
- TCP-IP 详解: 慢启动和拥塞控制
- TCP 协议详解(慢启动,流量控制,阻塞控制之类)
- TCP 协议与 UDP 协议的区别

React/redux

vDom react 原理

diff 的原理

- react diff

setState

- react@16.x
  - Calling setState with null no longer triggers an update. This allows you to decide in an updater function if you want to re-render.
  - Calling setState directly in render always causes an update. This was not previously the case. Regardless, you should not be calling setState from render.
  - setState callback (second argument) now fires immediately after `componentDidMount / componentDidUpdate` instead of after all components have `rendered`.
- setState

合成事件（SyntheticEvent）

- SyntheticEvent object

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
```

```
    DOMEvent nativeEvent
    void preventDefault()
    boolean isDefaultPrevented()
    void stopPropagation()
    boolean isPropagationStopped()
    DOMEventTarget target
    number timeStamp
    string type
```

- Event Pooling

  SyntheticEvent 对象会被重用，并且在调用事件回调后，所有属性都将无效。**切勿异步调用访问 SyntheticEvent 对象**

  > 如果要以异步方式访问事件属性，则应在事件上调用 event.persist()，这将从池中删除合成事件，并允许用户代码保留对事件的引用。

- 捕获阶段添加事件

  事件处理程序由冒泡阶段的事件触发。要为捕获阶段注册事件处理程序，请将Capture附加到事件名称；例如，您可以使用 onClickCapture 来处理捕获阶段中的 click 事件，而不是使用 onClick。

- React合成事件和DOM原生事件混用须知

redux 基本组成和设计单向数据流

- redux 源码解读

Angular

- AngularJS 脏检查深入分析

```
/**
 * 藏检查的内部实现
 *
 * 每当我们将数据绑定到 UI 上，angular 就会向你的 watchList 上插入一个 $watch。
 * 只有当触发UI事件，ajax请求或者 timeout 延迟，才会触发脏检查。
 *
 * getter 和 setter 是Vue 采用的机制,我觉得他两个最大的区别就是 Angular 采用事件驱动,而Vue 采用数据驱动 .所以 Angular 是当界面事件 或者其他 来触发脏检查, 而Vue 是检测后台数据变化,一旦变化 被 setter 捕捉, 然后来触发 界面更新.
 */
function $Scope() {
  this.$$watchList = [];
}

$Scope.prototype.$watch = function(name, getNewValue, listener) {
 const watch = {
   name: name,
```

```
    getNewValue: getNewValue,
    listener: listener || function(){},
  };

  this.$$watchList.push(watch);
}

$Scope.prototype.$digest = function () {
  let dirty = true;
  let checkTimes = 0;
  while(dirty) {
    dirty = this.$digestOnce();
    checkTimes++;
    if(checkTimes > 10 && dirty) {
      throw new Error('检测次数超过10次');
    }
  }
}

$Scope.prototype.$digestOnce = function() {
  let dirty=false;
  const list = this.$$watchList;

  for(let i=0, len = list.length; i<len; i++ ) {
    const watch = list[i];
    const newValue = watch.getNewValue(this);  // 传入 scope 获取 scope 内值
    const oldValue = watch.last;

    if(newValue !== oldValue) {
      watch.listen(newValue, oldValue);
      dirty = true;
    }

    watch.last = newValue;
  }

  return dirty;
}

/*******************************************************/

const scope = new $Scope();
scope.first = 1;
scope.secode = 10;

scope.$watch('first', function(_scope){
  return _scope[this.name]; //  getNewValue 通过 watch.getNewValue 方式调用, this 指向 watch
}, function(newValue, oldValue) {
  scope.second++;
  console.log('first:      newValue:' + newValue + '-----' + 'oldValue:' + oldValue);
})
```

```
scope.$watch('second', function(_scope){
  return _scope[this.name];
}, function(newValue, oldValue){
  scope.first++;
  console.log('second:     newValue:' + newValue + '-----' + 'oldValue:' +
oldValue)
})

scope.$digest();
```

- 如何衡量一个人的 AngularJS 水平?
- 基于 getter 和 setter 撸一个简易的MVVM

NodeJS

NodeJS 的事件循环(Event Loop)

- 详解 JavaScript 中的 Event Loop（事件循环）机制
- Node.js 的事件循环(Event Loop)、Timer 和 process.nextTick()[翻译]

koa 的原理,继承

```
// application.js

module.exports = class Application extends Emitter {
  constructor() {
    super()
    this.proxy = false
    this.middleware = []
    this.env = process.env.NODE_ENV || 'development'
    this.context = Object.create(context)
    this.request = Object.create(request)
    this.response = Object.create(response)
  }

  listen(...args) {
    const server = http.createServer(this.callback())
    server.listen(...args)
  }

  use(fn) {
    if (typeof fn !== 'function') throw new TypeError('')
    if (isGeneratorFunction(fn)) {
      console.warn('')
      fn = convert(fn)
    }
    this.middleware.push(fn)
    return this
  }
```

```
  callback() {
    // 生成 handle fn
    fn = compose(this.middleware)

    // 添加默认的 error handle
    if (!this.listenCounter('error')) this.on('error', this.onerror)

    return (req, res) => {
      // 生成 ctx
      const ctx = this.createContext(req, res)
      return this.handleRequest(ctx, fn)
    }
  }

  handleRequest(ctx, fnMiddleware) {
    const res = ctx.res
    res.statusCode = 404
    const onerror = err => ctx.onerror(err)
    const handleResponse = () => respond(ctx)
    onFinished(res, onerror)
    fnMiddleware(ctx)
      .then(handleResponse)
      .catch(error => onerror)
  }

  createContext(req, res) {
    const context = Object.create(this.context)
    const request = (context.request = Object.create(this.request))
    const response = (context.response = Object.create(this.response))
    context.app = request.app = response.app = this
    context.req = request.req = response.req = req
    context.res = request.res = response.res = res
    request.ctx = response.ctx = context
    request.response = response
    response.request = request
    context.originalUrl = request.originalUrl = req.url
    context.state = {}
    return context
  }

  onerror(err) {
    if (!(err instanceof Error)) throw new TypeError(util.format('non-
error thrown: %j', err))

    if (404 == err.status || err.expose) return
    if (this.silent) return

    const msg = err.stack || err.toString()
    console.error()
    console.error(msg.replace(/^/gm, '  '))
    console.error()
  }
}
```

```
function respond(ctx) {
  // allow bypassing koa
  if (false === ctx.respond) return

  const res = ctx.res
  if (!ctx.writable) return

  let body = ctx.body
  const code = ctx.status

  // ignore body
  if (statuses.empty[code]) {
    // strip headers
    ctx.body = null
    return res.end()
  }

  if ('HEAD' == ctx.method) {
    if (!res.headersSent && isJSON(body)) {
      ctx.length = Buffer.byteLength(JSON.stringify(body))
    }
    return res.end()
  }

  // status body
  if (null == body) {
    if (ctx.req.httpVersionMajor >= 2) {
      body = String(code)
    } else {
      body = ctx.message || String(code)
    }
    if (!res.headersSent) {
      ctx.type = 'text'
      ctx.length = Buffer.byteLength(body)
    }
    return res.end(body)
  }

  // responses
  if (Buffer.isBuffer(body)) return res.end(body)
  if ('string' == typeof body) return res.end(body)
  if (body instanceof Stream) return body.pipe(res)

  // body: json
  body = JSON.stringify(body)
  if (!res.headersSent) {
    ctx.length = Buffer.byteLength(body)
  }
  res.end(body)
}
```

others

- 有没有使用过 css3 动画，介绍一下,怎么做，关键是怎么做的 CSS
- 单行文本溢出，多行文本溢出把代码实现写出来
- 闭包，平时在哪用到？ 立即执行函数解决闭包中访问变量的问题
- 算法 快拍 选择排序
- 实现一个构造函数 new 的时候每次加一
- react 性能优化
- 知不知道 css 性能优化
- chrome显示12px以下字体的解决方法
- 跨域
    - JSONP
    - window.name
    - document.domain
    - location.hash onhashChange
    - cors
    - postMessage
    - 服务器
- async 和 promise 区别 async 和 promise 都不会阻塞执行，await 只会对 async 函数内 await 之后的代码产生阻塞。 async 异常捕获用 try...catch, promise 直接用 catch(), try...catch 无法捕获 promise 异常 async...await 是 Generator 函数语法糖， co 模块实现是通过 Promise 包装的
- redux
- React 中组件间通信的几种方式
- React中受控与非受控组件
- 正则test, match, exec
    - test\exec 是正则的实例方法， match 是字符串的方法
    - test 匹配与否，返回Boolean
    - match\exec 捕获组， 如果匹配，返回数组，未匹配返回null
        - 返回数组第一项为正则匹配的整个字符串，后面为括号对应的捕获组，index是整个匹配从零开始的索引，Input 为被解析的原始字符串
- var, let 区别
    - 顶级作用域 var 声明变量是 window 的属性，let\const 声明变量不是 window 的属性，变量都可以在控制台访问。
    - let声明的变量拥有块级作用域，不存在变量提升

    - 暂时性死区

      只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

      ```
      var tmp = 123;

      if (true) {
        tmp = 'abc'; // ReferenceError
        let tmp;
      }
      ```

    - 不允许重复声明

- 浏览器安全，xss,csrf
- require 和 es6 module 区别
- commonjs 如何查找（node）
- 内存溢出
- CORS
- 居中方法
- http 与 Https 区别， https 加密解密
- 页面渲染 1000 个元素
- setState，diff
- 页面性能优化

```
const testData = {
    a_bbb: 123,
    a_g: [1, 2, 3, 4],
    a_d: {
        s: 2,
        s_d: 3
    },
    a_f: [1, 2, 3, {
        a_g: 5
    }],
    a_d_s: 1
}

/**
 * 将一个json数据的所有key从下划线改为驼峰
 *
 * @param {object | array} value 待处理对象或数组
```

```
 * @returns {object | array} 处理后的对象或数组
 */
function mapKeysToCamelCase(data) {

    /**
     * 如果是基本常量return
     */
    if(!isObject(data)) {
        return data
    }

    if(Array.isArray(data)) {
        return data.map(key => {
            return isObject(key)? mapKeysToCamelCase(key): key
        })
    }


    let obj={};

    Object.keys(data).forEach(key => {
        const _key = strToCamelCase(key);
        obj[_key] = mapKeysToCamelCase(data[key])
    })
    return obj;
}

function isObject(val) {
    const tp = Object.prototype.toString.call(val)
    return !['[object Number]','[object String]','[object
Boolean]','[object Null]','[object Undefined]'].includes(tp)
}

function strToCamelCase(key) {
    return (''+key).replace(/(_.{1})/g, (val) =>
val.slice(1).toUpperCase());
}

console.log(mapKeysToCamelCase(testData))
```