

目录

Introduction	1.1
--------------	-----

【 JS内功修炼 】

第一节: this/闭包/作用域

专业术语	3.1
执行上下文	3.2
作用域	3.3
闭包	3.4
this	3.5
相关面试题	3.6

第二节: 面向对象/原型链/继承

专业术语	4.1
面向对象	4.2
原型链	4.3
继承	4.4
相关面试题	4.5

第三节: es6深入理解

专业术语	5.1
babel	5.2
块级作用域	5.3
函数	5.4
扩展对象的功能性	5.5
解构	5.6
Set与Map	5.7
js的类	5.8
改进的数组功能	5.9
Promise与异步编程	5.10
代理和反射	5.11
用模块封装代码	5.12

async-await	5.13
Decorator装饰器	5.14

第四节: 模块化/浏览器事件模型

专业术语	6.1
模块化	6.2
浏览器事件模型	6.3

第五|六节: VUE基础

数据绑定	7.1
指令	7.2
计算属性	7.3
过滤器	7.4
Class与Style绑定	7.5
过渡	7.6
事件处理	7.7
生命周期	7.8
组件	7.9

第七节: VUE-ROUTER

router使用	8.1
动态路由匹配	8.2
嵌套路由	8.3
编程式的导航	8.4
命名路由	8.5
HTML5History模式	8.6
导航守卫	8.7
滚动行为	8.8
路由懒加载	8.9

第八节: VUE-CLI

介绍	9.1
安装	9.2
基础	9.3

开发	9.4
----	-----

第十四节: 微信小程序的开发及原理

小程序的诞生	10.1
小程序配置	10.2
小程序的运行时	10.3
小程序的开发	10.4

基础课: js基础

数组方法	11.1
对象方法	11.2
正则	11.3
git基础	11.4
代码优化	11.5
常用查错方法	11.6

第十七节: 小程序其他知识

支付宝小程序与百度小程序	12.1
taro	12.2
mpvue	12.3

第十九节: react高级

context	13.1
高阶组件	13.2
Refs转发	13.3
Portals	13.4
PropTypes	13.5
Hook	13.6

第二十节: redux

redux重要API	14.1
react-redux	14.2

第二十一节：react router

与v3版比较	15.1
Route匹配	15.2
api	15.3
服务器渲染	15.4

第二十三节|第二十四节：node基础

node特点	16.1
node安装	16.2
npm使用介绍	16.3
node内部工作原理解析	16.4

第二十五节：node与数据库

redis	17.1
sequelize	17.2

第二十六节：express

入门	18.1
----	------

第二十八节：hybrid开发

介绍	19.1
JS和客户端的通讯	19.2

第二十九节：React Native

环境搭建	20.1
开发项目	20.2
编译并运行	20.3

第三十一节：Flutter

环境安装	21.1
编写您的第一个 Flutter App	21.2

第三十四节：八大算法

八大算法	22.1
------	------

第三十六节：BFS和DFS算法解析

BFS	23.1
DFS	23.2
剪枝算法	23.3

第三十七节：设计模式

单例模式	24.1
工厂模式	24.2
观察者模式	24.3

第三十九节：web性能

网页性能指标	25.1
相关知识	25.2
常见的性能优化方法	25.3
PerformanceAPI	25.4
缓存	25.5
bigpipe	25.6
PWA	25.7

爪哇教育-前端课程

- 讲师：袁鑫
- 2019-8

JS内功修炼

专业术语

- 常量、变量、数据类型
- 形参、实参
- 匿名函数、具名函数、自执行函数
- 函数声明、函数表达式
- 堆、栈
- 同步、异步、进程、线程

执行上下文

当函数执行时，会创建一个称为执行上下文（execution context）的环境，分为创建和执行2个阶段

创建阶段

创建阶段，指函数被调用但还未执行任何代码时，此时创建了一个拥有3个属性的对象：

```
executionContext = {
  scopeChain: {}, // 创建作用域链 (scope chain)
  variableObject: {}, // 初始化变量、函数、形参
  this: {} // 指定this
}
```

代码执行阶段

代码执行阶段主要的工作是：1、分配变量、函数的引用，赋值。2、执行代码。

举个栗子

```
// 一段这样的代码
function demo(num) {
  var name = 'xiaowa';
  var getData = function getData() {};
  function c() {}
}
demo(100);

// 创建阶段大致这样，在这个阶段就出现了【变量提升(Hoisting)】
executionContext = {
  scopeChain: { ... },
  variableObject: {
```

```

arguments: { // 创建了参数对象
  0: 100,
  length: 1
},
num: 100, // 创建形参名称，赋值/或创建引用拷贝
c: pointer to function c() // 有内部函数声明的话，创建引用指向函数体
name: undefined, // 有内部声明变量a，初始化为undefined
getData: undefined // 有内部声明变量b，初始化为undefined
},
this: { ... }
}

// 代码执行阶段，在这个阶段主要是赋值并执行代码
executionContext = {
  scopeChain: { ... },
  variableObject: {
    arguments: {
      0: 100,
      length: 1
    },
    num: 100,
    c: pointer to function c()
    name: 'xiaowa', // 分配变量，赋值
    getData: pointer to function getData() // 分配函数的引用，赋值
  },
  this: { ... }
}

```

执行上下文栈

- 浏览器中的JS解释器是单线程的，相当于浏览器中同一时间只能做一个事情。
- 代码中只有一个全局执行上下文，和无数个函数执行上下文，这些组成了执行上下文栈（Execution Stack）。
- 一个函数的执行上下文，在函数执行完毕后，会被移出执行上下文栈。

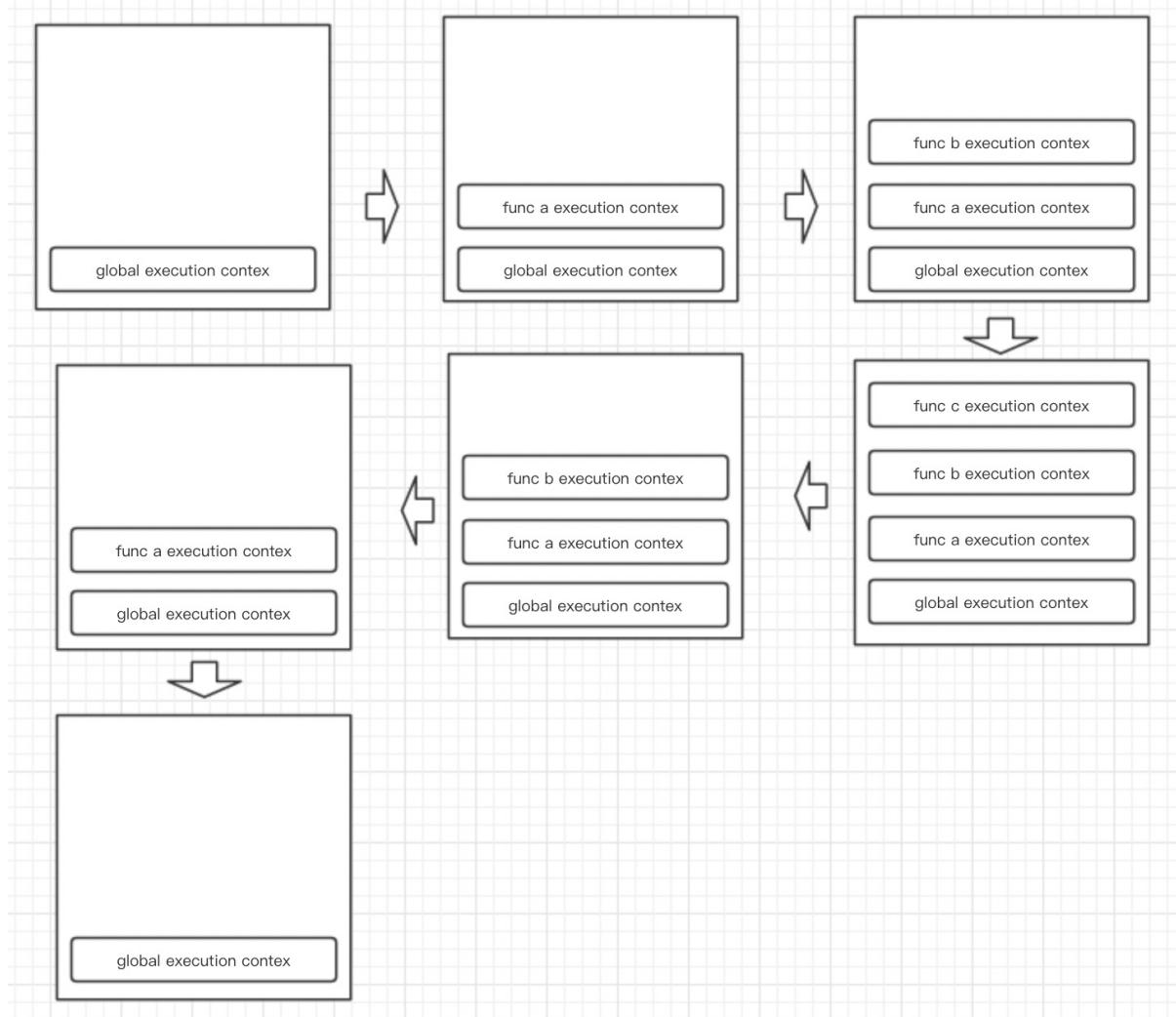
举个栗子

```

function c(){
  console.log('ok');
}
function b(){
  c();
}
function a(){
  b();
}
a();

```

这个栗子的执行上下文栈是这样的



作用域

js中有全局作用域、函数作用域，es6中又增加了块级作用域。作用域的最大用途就是隔离变量或函数，并控制他们的生命周期。作用域是在函数执行上下文创建时定义好的，不是函数执行时定义的。

举个栗子

```
// 不要看晕了哦~
function a () {
    return function b() {
        var myname = 'b';
        console.log(myname); // b
    }
}
function c() {
    var myname = 'c';
    b();
}
```

```

var b = a();
c();

// 去掉函数b中的myname声明后
function a () {
    return function b() {
        // var myname = 'b';
        console.log(myname); // 这里会报错
    }
}
function c() {
    var myname = 'c';
    b();
}
var b = a();
c();

```

作用域链

当一个块或函数嵌套在另一个块或函数中时，就发生了作用域的嵌套。在当前函数中如果js引擎无法找到某个变量，就会往上一级嵌套的作用域中去寻找，直到找到该变量或抵达全局作用域，这样的链式关系就称为作用域链(Scope Chain)

闭包

高级程序设计三中:闭包是指有权访问另外一个函数作用域中的变量的函数.可以理解为(能够读取其他函数内部变量的函数)

wiki百科的解释: [https://en.wikipedia.org/wiki/Closure_\(computer_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming))

In programming languages, a closure, also lexical closure or function closure, is a technique for implementing lexically scoped name binding in a language with first-class functions. Operationally, a closure is a record storing a function[a] together with an environment.[1] The environment is a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[b] Unlike a plain function, a closure allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

```

function outer() {
    var top = xxxx;
    function inner() {
        xxx.innerHTML = top;
    }
}

```

平时用在哪儿?

1、封装私有变量(amd的框架等都使用)

```
// 普通的定义类的方式
function Person() {
    this._attackVolume = 100;
}

Person.prototype = {
    attack(body) {
        body.bloodVolume -= this.attackVolume - body.defenseVolume;
    }
};

var person = new Person();
console.log(person._attackVolume);

// 工厂方法
function Person() {
    var _attackVolume = 100;
    return {
        attack() {
            body.bloodVolume -= attackVolume - body.defenseVolume;
        }
    };
}

var person = new Person();
console.log(person._attackVolume);
```

2、存储变量

```
// 封装的时候
function getListDataManager() {
    // 外层scope中定义一个变量
    let localData = null;

    return {
        getData() {
            // 里面的函数使用外层的变量，而且是反复使用
            if (localData) {
                return Promise.resolve(localData);
            }
            return fetch('xxxx')
                .then(data => localData = data.json());
        }
    };
}

// 用的时候
```

```
const listDataManager = getListDataManager();

button.onclick = () => {
    // 每次都会去获取数据，但是有可能是获取的缓存的数据
    text.innerHTML = listDataManager.getData();
};

window.onscroll = () => {
    // 每次都会去获取数据，但是有可能是获取的缓存的数据
    text.innerHTML = listDataManager.getData();
};
```

this

一共有5种场景。

场景1：函数直接调用时

```
function myfunc() {
    console.log(this) // this是window
}
var a = 1;
myfunc();
```

场景2：函数被别人调用时

```
function myfunc() {
    console.log(this) // this是对象a
}
var a = {
    myfunc: myfunc
};
a.myfunc();
```

场景3：new一个实例时

```
function Person(name) {
    this.name = name;
    console.log(this); // this是指实例p
}
var p = new Person('zhaowa');
```

场景4：apply、call、bind时

```
function getColor(color) {
    this.color = color;
    console.log(this);
}

function Car(name, color){
    this.name = name;    // this指的是实例car
    getColor.call(this, color); // 这里的this从原本的getColor, 变成了car
}

var car = new Car('卡车', '绿色');
```

场景5: 箭头函数时

```
// 复习一下场景1
var a = {
    myfunc: function() {
        setTimeout(function(){
            console.log(this); // this是a
        }, 0)
    }
};
a.myfunc();

// 稍微改变一下
var a = {
    myfunc: function() {
        var that = this;
        setTimeout(function(){
            console.log(that); // this是a
        }, 0)
    }
};
a.myfunc();

// 箭头函数
var a = {
    myfunc: function() {
        setTimeout(() => {
            console.log(this); // this是a
        }, 0)
    }
};
a.myfunc();
```

总结一下



- 1、对于直接调用的函数来说，不管函数被放在了什么地方，this都是window
- 2、对于被别人调用的函数来说，被谁点出来的，this就是谁
- 3、在构造函数中，类中(函数体中)出现的this.xxx=xxx中的this是当前类的一个实例
- 4、call、apply时，this是第一个参数。bind要优于call/apply哦，call参数多，apply参数少
- 5、箭头函数没有自己的this，需要看其外层的是否有函数，如果有，外层函数的this就是内部箭头函数的this，如果没有，则this是window

相关面试题

1. 考察this三板斧

1.1

```
function show () {
    console.log('this:', this);
}
var obj = {
    show: show
};
obj.show();

function show () {
    console.log('this:', this);
}

var obj = {
    show: function () {
        show();
    }
};
obj.show();
```

1.2

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
(0, obj.show)();
```

1.3

```
var obj = {
    sub: {
        show: function () {
            console.log('this:', this);
        }
    }
};
```

1.4

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var newobj = new obj.show();
```

1.5

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var newobj = new (obj.show.bind(obj))();
```

1.6

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var newobj = new (obj.show.bind(obj))();
```

1.7

```
var obj = {
    show: function () {
        console.log('this:', this);
    }
};
var elem = document.getElementById('book-search-results');
elem.addEventListener('click', obj.show);
elem.addEventListener('click', obj.show.bind(obj));
elem.addEventListener('click', function () {
```

```
    obj.show();
});
```

2. 作用域

2.1

```
var person = 1;
function showPerson() {
    var person = 2;
    console.log(person);
}
showPerson();
```

2.2

```
var person = 1;
function showPerson() {
    console.log(person);
    var person = 2;
}
showPerson();
```

2.3

```
var person = 1;
function showPerson() {
    console.log(person);

    var person = 2;
    function person() {}
}
showPerson();
```

2.4

```
var person = 1;
function showPerson() {
    console.log(person);

    function person() {}
    var person = 2;
}
showPerson();
```

2.5

```
for(var i = 0; i < 10; i++) {
    console.log(i);
}

for(var i = 0; i < 10; i++) {
    setTimeout(function(){
        console.log(i);
    }, 0);
}

for(var i = 0; i < 10; i++) {
    (function(i){
        setTimeout(function(){
            console.log(i);
        }, 0)
    })(i);
}

for(let i = 0; i < 10; i++) {
    console.log(i);
}
```

JS内功修炼

专业术语

- 类、封装、继承、多态
- 构造函数、实例、对象字面量
- 命名空间
- 内置对象、宿主对象、本地对象

面向对象

js里还是没有'类'，js的面向对象，还是基于原型的，无论是ES5/还是ES6，ES6中引入的class，只是基于原型继承模型的语法糖。

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

有兴趣的同学可以看一下ECMAScript的文档：

ECMA - 256页 - 开始定义类

ECMA - 260页 - 定义了什么是继承

创建对象的方法

方法1：工厂模式(没有对象识别)

举个栗子

```
function body() {  
    var o = new Object();  
    o._bloodVolume = 100;  
    o._attackVolume = 500;  
    return o;  
}  
  
var monster = body();  
var monster2 = body();
```

方法2：构造函数模式

js中可以使用方法(function)作为类，定义一个类与定义一个函数一致。使用new操作符，来创建新的对象。该function不显式的创造对象。

new的几个步骤：

- 1、创建一个新对象
- 2、将构造函数中的作用域指向该对象
- 3、执行构造函数中的代码
- 4、返回新对象

举个栗子

```
function Body() {  
    this._bloodVolume = 100;  
    this._attackVolume = 500;  
}  
  
var monster = new Body();
```

手写一个【new函数】

```
function Body() {  
    this._bloodVolume = 100;  
    this._attackVolume = 500;  
}  
  
function newOperation(constructFunc) {  
    const newObj = Object.create(null);  
    constructFunc.call(newObj);  
    return newObj;  
}  
  
var monster = newOperation(Body);
```

方法3：原型模式

我们创建的每个函数都有个prototype属性，这个属性是个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。那么prototype就是通过调用构造函数而创建的那个对象实例的原型对象。

举个栗子

```
function Body() {}  
Body.prototype._bloodVolume = 100;  
Body.prototype._attackVolume = 500;  
  
var monster = new Body();
```

什么是原型？

1. 真正的原型，在构造函数中

2. 我们每声明一个函数，浏览器就会在内存中创建一个对象，在这个对象中增加一个属性，叫 constructor指向我们的函数，并把我们的函数的prototype属性指向这个对象。
3. 用这个构造函数创建的对象都有一个不可访问的属性 [[prototype]]，这个属性就指向了构造函数的prototype，在浏览器中，支持使用 __proto__ 来访问这个对象。

使用原型对象的好处是，可以在创建出来的多个对象中，共享属性和方法。

手写一个【new函数】

```
function Body() {}
Body.prototype._bloodVolume = 100;
Body.prototype._attackVolume = 500;

function newOperation(constructFunc) {
    const newObj = Object.create(constructFunc.prototype);
    return newObj;
}

var monster = newOperation(Body);
```

这种模式下，每一个生成的对象都有一个属性[[Prototype]]，浏览器厂商在具体实现的时候，保留了一个 __proto__ 属性，用以让我们访问[[Prototype]]

[Prototype]的ECMA解释

19.2.4.3 prototype Function instances that can be used as a constructor have a prototype property. Whenever such a Function instance is created another ordinary object is also created and is the initial value of the function's prototype property. Unless otherwise specified, the value of the prototype property is used to initialize the [[Prototype]] internal slot of the object created when that function is invoked as a constructor.

方法4：组合模式

组合模式是构造函数和原型模式一起使用，构造函数模式用于定义实例属性，原型模式用于定义方法和共享的属性。

举个栗子

```
function Person() {
    this._attackVolume = 100;
}
Person.prototype = {
    attack(body) {
        body.bloodVolume -= this.attackVolume - body.defenseVolume;
    }
};
var hero = new Person();
```

手写一个【new函数】

```

function Person() {
    this._attackVolume = 100;
}
Person.prototype = {
    attack(body) {
        body.bloodVolume -= this.attackVolume - body.defenseVolume;
    }
};

function newOperation(constructFunc) {
    const newObj = Object.create(constructFunc.prototype);
    constructFunc.call(newObj);
    return newObj;
}

var hero = newOperation(Person);

```

es6的写法

```

// class expression
var Person = class {
    constructor(height, width) {}
}

// class declaration
class Person {
    constructor(height, width) {}
}

```

这里我们看下babel编译后的样子

原型链

利用js的原型模型，代码读取某个对象的某个属性的时候，都会按照属性名执行一次搜索，在实例中找到了则返回，如果没找到，则继续在当前实例的原型对象中搜索，直到找到为止。如果还没找到，则继续该原型对象的原型对象，以此类推，直到搜索到Object对象为止，这样就形成了一个原型指向的链条，专业术语称之为原型链。

继承

方法1：原型链继承

```

//父类型
function Body() {
    this.volumes = {
        _bloodVolume: 1000,

```

```

        _attackVolume: 500,
        _defenseVolume: 200
    );
}

Body.prototype.attacked = function (body) {
    this.volumes._bloodVolume -= body.getAttackVolume() - this.volumes._defenseVolume;
};

//子类型
function Monster() {};
Monster.prototype = new Body();
Monster.prototype.attacked = function () {
    this.volumes._bloodVolume -= 1;
};

var monster = new Monster();
var monster2 = new Monster();
monster.attacked();
console.log(monster2.volumes._bloodVolume);

```

注意：这种方式，monster改变`_bloodVolume`后，monster2的`_bloodVolume`也被改变了。

方法2：借用构造函数继承

```

// 父类
function Body() {
    this._bloodVolume = 1000;
    this._attackVolume = 500;
    this._defenseVolume = 200;
}
Body.prototype.attacked = function (body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
};

// 子类
function Monster() {
    Body.call(this);
};

var monster = new Monster();

```

注意：这种方式，Monster无法继承父类prototype上的方法和属性

方法3：原型链+借用构造函数的组合继承

```

// 父类
function Body() {

```

```

        this._bloodVolume = 1000;
        this._attackVolume = 500;
        this._defenseVolume = 200;
    }
Body.prototype.attacked = function (body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
};

// 子类
function Monster() {
    Body.call(this);
}
Monster.prototype = new Body();

var monster = new Monster();

```

方法4：寄生组合继承

```

// 父类
function Body() {
    this._bloodVolume = 1000;
    this._attackVolume = 500;
    this._defenseVolume = 200;
}
Body.prototype.attacked = function (body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
};

// 子类
function Monster() {
    this.name = 'asd';
    Body.call(this);
}
Monster.prototype = Object.create(Body.prototype);

var monster = new Monster();

```

方法5：es6的class

```

// 父类
class Body {
    constructor() {
        this._bloodVolume = 1000;
        this._attackVolume = 500;
        this._defenseVolume = 200;
    }

```

```
attacked(body) {
    this._bloodVolume -= body.getAttackVolume() - this._defenseVolume;
}
}

// 子类
class Monster extends Body {
    constructor() {
        super();
    }

    attacked() {
        this._bloodVolume -= 1;
    }
}

var monster = new Monster();
```

这里我们看下babel编译后的样子

相关面试题

1. 面向对象

1.1

```
function Person() {
    this.name = 1;
    return {};
}
var person = new Person();
console.log('name:', person.name);
```

1.2

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();
person.show();
```

1.3

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    name: 2,
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();

Person.prototype.show = function () {
    console.log('new show');
};

person.show();
```

1.4

```
function Person() {
    this.name = 1;
}
Person.prototype = {
    name: 2,
    show: function () {
        console.log('name is:', this.name);
    }
};
var person = new Person();
var person2 = new Person();

person.show = function () {
    console.log('new show');
};

person2.show();
person.show();
```

2、综合题

```
function Person() {
    this.name = 1;
}

Person.prototype = {
    name: 2,
```

```
show: function () {
    console.log('name is:', this.name);
}
};

Person.prototype.show();

(new Person()).show();
```

es6深入理解

专业术语

- 严格模式、非严格模式

babel

ECMAScript 6(ES6)的发展速度非常之快，但现代浏览器对ES6新特性支持度不高，所以要想在浏览器中直接使用ES6的新特性就得借助别的工具来实现。

babel原理

- babel 本质上就是在操作抽象语法树（Abstract Syntax Tree, AST）来完成代码的转译。
- babel的工作步骤：解析-转换-生成器
- babel 对于 AST 的遍历是深度优先遍历，对于 AST 上的每一个分支 babel 都会先向下遍历走到尽头，然后再向上遍历退出刚遍历过的节点，然后寻找下一个分支。

在线工具

<http://babeljs.io/repl/>

安装步骤

- npm install
 - "@babel/cli": "^7.5.5",
 - "@babel/core": "^7.5.5",
 - "@babel/preset-env": "^7.5.5",
 - "babel-plugin-proxy": "^1.1.0",
 - "babel-polyfill": "6.26.0",
- 安装语法插件
 - .babelrc 文件配置
 - babel src -d dist

语法插件

```
# ES2015转码规则
npm install --save-dev babel-preset-es2015

# react转码规则
```

```

npm install --save-dev babel-preset-react

# ES7不同阶段语法提案的转码规则（共有4个阶段），选装一个
npm install --save-dev babel-preset-stage-0
npm install --save-dev babel-preset-stage-1
npm install --save-dev babel-preset-stage-2
npm install --save-dev babel-preset-stage-3

```

.babelrc

```

{
  "presets": [
    [
      "@babel/env",
      {
        "targets": {
          "edge": "7",
          "firefox": "60",
          "chrome": "50",
          "safari": "11.1",
        },
        "useBuiltIns": "usage"
      }
    ]
  ],
  "plugins": ["proxy"]
}

```

块级作用域绑定

变量提升机制

在函数作用域或者全局作用域中，通过var声明的变量，无论在哪里声明，都会被当成在当前作用域顶部声明，这就是变量提升（Hoisting）

块级作用域

块级声明用于声明在制定块的作用域之外无法访问的变量。块级作用域存在于：

- 函数内部
- 块中（大括号之间的区域）

let声明

对变量的声明，不能变量提升。

举个栗子

```
if(condition) {  
    let value = 'bule';  
    console.log(value);  
} else {  
    // ...  
}  
console.log(value);
```

注意

1、禁止重复声明

```
var abc = 30;  
let abc = 40; // 会抛错  
  
var abc = 30;  
if (condition) {  
    let abc = 40; // 不会抛错  
}
```

2、块级不会提升

```
console.log(abc); // 会抛错 (临时死区Temporal Dead Zone)  
console.log(typeof abc); // 会抛错  
let abc = 30;  
  
console.log(abc); // undefined  
var abc = 30;
```

const声明

对常量的声明，同样也不会变量提升。目前的普遍推荐方式是：默认使用const，只有确实需要改变的变量时，才用let。

举个栗子

```
const name = 'zhaowa';
```

注意

1、有效的常量声明

```
const abc = 30; // 有效  
const abc; // 无效  
const abc = 40; // 重复声明，会报错
```

2、用const声明对象时，不允许修改绑定，但允许修改值

```
const persion = {
  age: 18
};
persion.age = 19; // 有效
```

循环中的块级作用域

长久以来，var声明让开发者在循环中创建函数变的异常困难，因为变量到了循环之外仍能访问。为了解决这个问题，开发者在循环中使用立即调用函数表达式。

举个栗子

```
var funcs = [];
for (var i = 0; i < 10; i++) {
  funcs.push(function() {
    console.log(i);
  });
}
funcs.forEach(function(func) {
  func(); // 输出10次10
});

// 使用自执行函数
var funcs = [];
for (var i = 0; i < 10; i++) {
  funcs.push(function(i) {
    return function() {console.log(i);}
  )(i);
}
funcs.forEach(function(func) {
  func(); // 输出0到9
});

// 使用let
var funcs = [];
for (let i = 0; i < 10; i++) {
  funcs.push(function() {
    console.log(i);
  });
}
funcs.forEach(function(func) {
  func(); // 输出0到9
});
```

全局块作用域绑定

let和const与var的另外一个区别是他们在全局作用域中的行为。这意味着var很可能无意间覆盖了一个已经存在的全局属性。

举个栗子

```
var RegExp = 'hello';
console.log(window.RegExp); // hello

let RegExp = 'hello';
console.log(RegExp); // hello
console.log(window.RegExp === RegExp); // false
```

函数

函数形参的默认值

举个栗子

```
// es5效果
function request(url, timeout, callback) {
    timeout = (typeof timeout !== 'undefined') ? timeout : 1000;
    callback = (typeof callback !== 'undefined') ? callback : function() {};
}

// es6效果
function request(url, timeout = 1000, callback = function() {}) {
    console.log(arguments.length); // 如果只传了url, 那arguments.length为1
}
```

注意

1、前面的参数默认值不能引用后面的参数值

```
function add(a,b=1){return a+b};
console.log(add(1,1)); //2

function addition(a=b,b){return a+b};
console.log(addition(undefined,1)); // 报错
```

处理无命名参数

在函数的命名参数前添加三个点(...)就标识这是一个不定参数，是个数组。

举个栗子

```
function checkArgs(...keys) {
    console.log(args.length); // 2
    console.log(arguments.length); // 2
    console.log(args[0], arguments[0]); // 1 1
    console.log(args[1], arguments[1]); // 2 2
```

```
    }
    checkArgs(1, 2);
```

注意

1、...必须是最后一个参数

```
function demo(object, ...keys, last) { // 报错
}
```

2、setter中不能使用...

```
let object = {
    set name(...value) { // 报错
        }
}
```

扩展运算符

执行函数时，传入展开后的参数

举个栗子

```
abc(...value, 0);
```

箭头函数

es6中的新特性，通过 `参数 => 函数体` 的方式调用。

特性

- 没有this、super、arguments, new.target绑定。this、super、arguments以及内部函数的new.target的值由所在的最近的外部非箭头函数来决定。
- 不能使用new来调用。箭头函数没有[[Construct]]方法，因此不能被用为构造函数，使用new调用函数会抛出错误。
- 没有原型。没有使用new，因此没有prototype属性。
- 不能修改this的绑定。不能通过call(),apply()以及bind()方法修改this。
- 没有arguments对象。需要通过命名参数和不定参数这两种形式访问函数的参数。
- 不允许使用重复的具名参数。箭头函数不允许拥有重复的具名参数，无论是否在严格模式下。

举个栗子

```
// es6
let sum = (num1, num2) => num1 + num2;
// es5
```

```

var sum = function(num1, num2) {
    return num1 + num2;
}

// es6
let person = ((name) => {
    return {
        getName: function() {
            return name;
        }
    };
})('zhaowa');
console.log(person.getName());
// es5
var person = function(name) {
    return {
        getName: function() {
            return name;
        }
    };
}('zhaowa');
console.log(person.getName());

```

注意

1、箭头函数没有arguments对象，但可以访问上一层函数的arguments对象

```

function outer() {
    return () => arguments[0];
}
var result = outer(1,2,3);
console.log(result()); // 1

```

2、如果需要返回对象的话，需要使用圆括号()将对象包裹起来，为了防止对象字面量被认为是函数体语句。

尾调用优化

主要是为了解决递归函数的爆栈问题，降低内存占用。需要满足以下条件：

- 尾调用不能引用当前栈帧中的变量。
- 进行尾调用的函数在尾调用返回结果后不能做额外任何操作。
- 尾调用的结果作为当前函数的返回值。

扩展对象的功能性

对象字面量语法扩展

举个栗子

1、属性初始值的简写

```
// es5
function demo(name, age) {
    return {
        name: name,
        age: age
    }
}

// es6
function demo(name, age) {
    return {
        name,
        age
    }
}
```

2、对象方法的简写

```
// es5
var demo = {
    name: 'zhaowa',
    getName: function() {
        console.log(this.name);
    }
}

// es6
let demo = {
    name: 'zhaowa',
    getName() {
        console.log(this.name);
    }
}
```

3、可计算属性名

```
// es5
var demo = {
    'first name': 'xiaowa'
};
console.log(demo['first name']);

// es6
let firstName = 'first name';
```

```
let demo = {
  [firstName]: 'xiao'
};
console.log(demo[firstName]);
```

es6新增方法

Object.is()

Object.assign()

重复的对象字面量属性

举个栗子

```
// es5
var person = {
  name: 'xiaowa',
  name: 'yuanxin' // 严格模式下报错
};

// es6
var person = {
  name: 'xiaowa',
  name: 'yuanxin' // 严格模式和非严格模式都正常
};
console.log(person.name); // yuanxin
```

增强对象原型

1、使用setPrototypeOf修改对象原型

```
let person = {
  getName(){
    return 'hello';
  }
}
let dog ={
  getName(){
    return 'world';
  }
}

let friend = Object.create(person);
console.log(friend.getName()); //hello
console.log(Object.getPrototypeOf(friend)===person); //true

Object.setPrototypeOf(friend,dog);
```

```
console.log(friend.getName()); //world
console.log(Object.getPrototypeOf(friend) === dog); //true
```

2、super的引用，来访问原型中的方法

```
let person = {
  getName(){
    return 'hello';
  }
}
let dog ={
  getName(){
    return super.getName() + ' world';
  }
}
Object.setPrototypeOf(dog, person);
console.log(dog.getName()); //hello world
```

解构

对象的解构

解构语法

举个栗子

```
let person ={
  name: 'hello',
  age: 18
}
let {name, age} = person;
console.log(name); //hello
console.log(age); //18
```

解构赋值与默认值

举个栗子

```
let person ={
  name: 'hello',
  age: 18
}
let name = 'world';
let age = 20;
({name, age, value = true} = person); // 这里是圆括号包裹
console.log(name); // hello
console.log(age); // 18
```

```
console.log(value); // true
```

为不同名的变量赋值

举个栗子

```
let person = {  
    name:'hello',  
    age:18  
};  
({name: localName, age: localAge, value = true} = person); // 等号是赋默认值, 冒号是为别名赋  
值  
console.log(localName); //hello  
console.log(localAge); //18  
console.log(value); // true
```

嵌套的对象解构

举个栗子

```
let person = {  
    name: 'zhaowa',  
    school: {  
        primary: {  
            start: 2000,  
            column: 2007  
        },  
        middle: {  
            start: 2007,  
            end: 2010  
        },  
        university: {  
            start: 2010,  
            end: 2014  
        }  
    }  
};  
let{school: {university}} = person;  
console.log(university.start); // 2010  
console.log(university.end); // 2014
```

数组的解构

解构语法

举个栗子

```
let arr = [1,2,3];
```

```
let [first, second] = arr;
console.log(first); // 1
console.log(second); // 2
```

解构赋值与默认值

举个栗子

```
let arr = [1, 2, 3];
[first, second, third = 100] = arr;
[first, second] = [second, first]; // 变量值互换
console.log(first); // 2
console.log(second); // 1
console.log(third); // 100
```

剩余项

举个栗子

```
let aaa = [1, 2, 3, 4, 5];
let [first, ...subArr] = aaa;
console.log(first); // 1
console.log(subArr.length); // 4
console.log(subArr[0]); // 2

// 深拷贝数组
let bbb = [1, 2, 3, 4, 5];
let [...clonedArr] = bbb;
console.log(clonedArr); // [1, 2, 3, 4, 5]
```

参数解构

对可选参数设置默认值

举个栗子

```
function setCookie(name, value,
{
  secure = false,
  path = "/",
  domain = "example.com",
  expires = new Date(Date.now() + 3600000000)
} = {}
) {
// ...
```

}

Set与Map

Set

ES6中新增了Set类型是一种有序列表，其中包含一些相互独立的非重复值。Set在存放对象时，实际上是存放的是对象的引用，如果所存储的对象被置为null，但是Set实例仍然存在的话，对象依然无法被垃圾回收器回收，从而无法释放内存。

举个栗子

```
// 创建、检验、删除
let set= new Set();
set.add(5);
set.add('5');
set.add(5);
console.log(set.size); // 2 只存放不重复的值
set.has(5); // true 检验某值是否存在
set.delete('5'); // 删除某值

// 与数组互转
let set = new Set([1, 2, 3]);
let arr = [...set];
console.log(arr); // [1,2,3]

let arr = [1, 2, 3];
let set = new Set(arr);
console.log(set.length); // 3

// forEach方法遍历
let set = new Set([1,2,3,3,3,3]);
set.forEach(function (value, key) { // 为了和Map一致，set的value和key的值一样
    console.log(value);
    console.log(key);
});

// 使用WeakSet解决无法垃圾回收的问题
let set = new WeakSet();
let key = {};
set.add(key);
console.log(set.has(key)); //true
set.delete(key);
console.log(set.has(key));
```

注意

- 对于Weak Set实例，若调用了add()方法时传入了非对象的参数，则会抛出错误。如果在has()或

者`delete()`方法中传入了非对象的参数则会返回`false`;

- `Weak Set`集合不可迭代，因此不能用于`for-of`循环；
- `Weak Set`集合不暴露任何迭代器（例如`keys()`与`values()`方法），所以无法通过程序本身来检测其中的内容；
- `Weak Set`集合不支持`forEach()`方法；
- `Weak Set`集合不支持`size`属性；

Map

ES6中提供了`Map`数据结构，能够存放键值对。

举个栗子

```
// set、get、has、delete、clear
let map = new Map(['year', 2018]);
map.set('name', 'zhaowa');
console.log(map.get('title')); // zhaowa
console.log(map.has('title')); // true
map.delete('title');
map.clear();
console.log(map.size); // 0

// forEach
let map = new Map([['title', 'hello world'], ['year', '2018']]);
map.forEach((value, key) => {
  console.log(value);
  console.log(key);
});

// Weak Map
let map = new WeakMap([[key, 'hello'], [key2, 'world']]);
```

js的类

回想一下es5如何创建类的？

注意

- 函数声明可以被提升，而类和`let`类似，不能被提升。真正执行声明语句之前，他们会一直存在于临时死区中。
- 类声明中的所有代码会自动运行在严格模式下，无法强行让代码脱离严格模式执行。
- 类的所有方法都是不可枚举的；
- 每个类都有一个名为`[[Constructor]]`的内部方法，通过`new`调用那些没有`[[Constructor]]`的方法会报错。
- 使用非`new`关键字调用构造函数会抛出错误；
- 在类中修改类名会抛出错误；

创建类

举个栗子

```
// 方式1: 类声明
class PersonClass{
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log(this.name);
    }
}
let person = new PersonClass("hello class");
person.sayName();

// 方式2: 类表达式
let PersonClass = class {
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log(this.name);
    }
}
let person = new PersonClass("hello class");
person.sayName(); //hello class

// 方式3: 当做参数传入
function createObj(classDef){
    return new classDef();
}
let person = createObj(class{
    sayName(){
        console.log('hello'); //hello
    }
});
person.sayName();

// 立即调用构造器
let person = new class{
    constructor(name){
        this.name = name;
    }
    sayName(){
        console.log(this.name);
    }
}('hello world');
person.sayName(); //hello world
```

访问器属性

自有属性需要在类构造器中创建，而类还允许创建访问器属性。为了创建一个getter，要使用get关键字，并要与后面的标识符之间留出空格；创建setter使用相同的方式，只需要将关键字换成set即可：

举个栗子

```
class PersonClass{
    constructor(name){
        this.name = name;
    }
    get name(){
        return name; //不要使用this.name会导致无限递归
    }

    set name(value){
        name=value; //不要使用this.value会导致无限递归
    }
}
let person = new PersonClass('hello');
console.log(person.name); // hello
person.name = 'world';
console.log(person.name); //world
let descriptor = Object.getOwnPropertyDescriptor(PersonClass.prototype, 'name');
console.log('get' in descriptor); //true
```

静态成员

ES6的类简化了静态成员的创建，只要在方法与访问器属性的名称前添加static关键字即可：

举个栗子

```
class PersonClass {
    // 等价于 PersonType 构造器
    constructor(name) {
        this.name = name;
    }
    static create(name) {
        return new PersonClass(name);
    }
}
let person = PersonClass.create("Nicholas");
```

类继承

使用关键字extends可以完成类继承，同时使用super关键字可以在派生类上访问到基类上的方法，包括构造器方法：

举个栗子

```

class Rec{
    constructor(width, height){
        this.width = width;
        this.height = height;
    }
    getArea(){
        return this.width * this.height;
    }
}
class Square extends Rec{
    constructor(width,height){
        super(width, height);
    }
}
let square = new Square(100, 100);
console.log(square.getArea()); //10000

```

注意

- 如果基类中包含了静态成员，那么这些静态成员在派生类中也是可以使用的。注意：静态成员只能通过类名进行访问，而不是使用对象实例进行访问；

从表达式中派生类

在ES6中最大的能力是从表达式中导出类的功能，只要一个表达式可以被解析成一个函数并具有`[[Constructor]]`属性以及原型，就可以用`extends`进行派生。由于`extends`后面能够接收任意类型的表达式，创造了更多的可能性，动态的确定类的继承目标。

举个栗子

```

let SerializableMixin = {
    serialize() {
        return JSON.stringify(this);
    }
};
let AreaMixin = {
    getArea() {
        return this.length * this.width;
    }
};
function mixin(...mixins) {
    var base = function() {};
    Object.assign(base.prototype, ...mixins);
    return base;
}
class Square extends mixin(AreaMixin, SerializableMixin) {
    constructor(length) {
        super();
    }
}

```

```

    this.length = length;
    this.width = length;
}
}

let x = new Square(3);
console.log(x.getArea()); // 9
console.log(x.serialize()); // {"length":3,"width":3}

```

继承内置对象

在ES6中能够通过`extends`继承JS中内置对象，例如：

举个栗子

```

class MyArray extends Array {
// 空代码块
}
let colors = new MyArray();
colors[0] = "red";
console.log(colors.length); // 1
colors.length = 0;
console.log(colors[0]); // undefined

```

改进的数组功能

创建数组

- `Array.of()`
- `Array.from()`

数组上所有的新方法

- `find()`和`findIndex()`
- `fill()`
- `copyWithin()`

Promise与异步编程

Promise基础

每个 Promise 都会经历一个短暂的生命周期，初始为进行中状态（pending），也被认为是未处理的（unsettled）。一旦异步操作结束，Promise就会被认为是已处理的（settled），操作结束后Promise可能会进入下面两种状态之一：

- `fulfilled`: Promise 异步操作成功完成

- rejected: 由于程序错误或者其他原因, Promise 的异步操作未成功完成

内部属性[[PromiseState]]被用来表示3种状态: "pending"、"fulfilled"、"rejected", 属性不会暴露在Promise 对象上, , 当改变状态时, 通过then()方法采取一些特定的行动。

举个栗子

```
promise.catch(function(err) {
    // 拒绝
    console.error(err);
});

promise.then(null, function(err) {
    // 拒绝
    console.error(err);
});

promise.then(function(res) {
    console.log(res);
}).catch(function(err) {
    console.error(err);
});
```

创建未完成的Promise

```
let promise = new Promise(function(resolve, reject) {
    console.log('hi, promise');
    resolve();
});
promise.then(() => {
    console.log('hi, then');
});
console.log('hi');
```

输出:
hi, promise
hi
hi then

创建已完成的Promise

有以下2种方式:

```
let promise = Promise.resolve('zhaowa');
promise.then(res => {
    console.log(res); // zhaowa
});
```

```
let reject = Promise.reject('yuanxin');
reject.catch(err => {
    console.log(err); // yuanxin
})
```

执行器错误

如果执行器内部抛出一个错误，则Promise的拒绝处理会被调用

```
let promise = new Promise(function(resolve, reject) {
    throw new Error('Error!');
});
promise.catch(function(msg) {
    console.log(msg); // error
});
```

串联Promise

每次调用then()方法或者catch()方法会返回另外一个Promise，只有当第一个被拒绝或者完成后，第二个才会被解决。下面看下串联的几种场景。

举个栗子

```
// 捕获错误
let p1 = new Promise(function(resolve, reject) {
    resolve('zhaowa');
});
p1.then(res => {
    console.log(res);
    throw new Error('Error!');
}).catch(err => {
    console.log(err);
});

// 传递值
let p1 = new Promise(function(resolve, reject) {
    resolve(100);
});
p1.then(res => res + 1)
.then(res => {
    console.log(res); // 101
});

// 传递promise
let p1 = new Promise(function(resolve, reject){
    resolve('zhaowa');
});
```

```
let p2 = new Promise(function(resolve, reject){  
    resolve('yuanxin');  
})  
p1.then(res => {  
    console.log(res); // zhaowa  
    return p2;  
}).then(res => {  
    console.log(res); // yuanxin  
});
```

响应多个Promise

监听多个Promise来决定下一步的操作。

举个栗子

```
//Promise.all()  
let p1 = new Promise(function(resolve, reject) {  
    resolve(1);  
});  
let p2 = new Promise(function(resolve, reject) {  
    resolve(2);  
});  
let p3 = new Promise(function(resolve, reject) {  
    resolve(3);  
});  
let p4 = Promise.all([p1, p2, p3]);  
p4.then([p1res, p2res, p3res] => {  
    // 需要所有的都成功，才走到这里  
});  
  
// Promise.race()  
let p1 = new Promise(function(resolve, reject) {  
    resolve(1);  
});  
let p2 = new Promise(function(resolve, reject) {  
    resolve(2);  
});  
let p3 = new Promise(function(resolve, reject) {  
    resolve(3);  
});  
let p4 = Promise.race([p1, p2, p3]);  
p4.then(res => {  
    // 只要有一个先成功，就会走到这里  
});
```

核心Promises / A +规范

2.1 Promise状态

一个promise必须处于三种状态之一： 请求态（pending）， 完成态（fulfilled）， 拒绝态（rejected）

2.1.1 当promise处于请求状态（pending）时

2.1.1.1 promise可以转为fulfilled或rejected状态

2.1.2 当promise处于完成状态（fulfilled）时

2.1.2.1 promise不能转为任何其他状态

2.1.2.2 必须有一个值，且此值不能改变

2.1.3 当promise处于拒绝状态（rejected）时

2.1.3.1 promise不能转为任何其他状态

2.1.3.2 必须有一个原因（reason），且此原因不能改变

2.2 then方法

promise必须提供then方法来存取它当前或最终的值或者原因。

promise的then方法接收两个参数：

promise.then(onFulfilled, onRejected)

复制代码

2.2.1 onFulfilled和onRejected都是可选的参数：

2.2.1.1 如果 onFulfilled不是函数，必须忽略

2.2.1.1 如果 onRejected不是函数，必须忽略

2.2.2 如果onFulfilled是函数：

2.2.2.1 此函数必须在promise 完成(fulfilled)后被调用，并把promise 的值作为它的第一个参数

2.2.2.2 此函数在promise完成(fulfilled)之前绝对不能被调用

2.2.2.2 此函数绝对不能被调用超过一次

2.2.3 如果onRejected是函数：

2.2.2.1 此函数必须在promise rejected后被调用，并把promise 的reason作为它的第一个参数

2.2.2.2 此函数在promise rejected之前绝对不能被调用

2.2.2.2 此函数绝对不能被调用超过一次

2.2.4 在执行上下文堆栈（execution context）仅包含平台代码之前，不得调用 onFulfilled和onRejected

3.1

2.2.5 onFulfilled和onRejected必须被当做函数调用(i.e. with no this value-->这里不会翻.....).

3.2

2.2.6 then可以在同一个promise里被多次调用

2.2.6.1 如果/当 promise 完成执行（fulfilled），各个相应的onFulfilled回调

必须根据最原始的then 顺序来调用

2.2.6.2 如果/当 promise 被拒绝（rejected），各个相应的onRejected回调

必须根据最原始的then 顺序来调用

2.2.7 then必须返回一个promise 3.3

```
promise2 = promise1.then(onFulfilled, onRejected);
```

复制代码

2.2.7.1 如果onFulfilled或onRejected返回一个值x，运行

Promise Resolution Procedure [[Resolve]](promise2, x) 2.3

2.2.7.2 如果onFulfilled或onRejected抛出一个异常e,promise2

必须被拒绝 (rejected) 并把e当作原因

2.2.7.3 如果onFulfilled不是一个方法，并且promise1已经完成 (fulfilled) , promise2必须使用与promise1相同的值来完成 (fulfilled)

2.2.7.4 如果onRejected不是一个方法，并且promise1已经被拒绝 (rejected) , promise2必须使用与promise1相同的原因来拒绝 (rejected)

2.3 Promise解决程序

promise解析过程 是一个抽象操作，它将promise和value作为输入，我们将其表示为[[Resolve]] (promise, x)。

如果x是thenable的，假设x的行为至少有点像promise，

它会尝试让promise采用x的状态。不然就会用x来完成promise

只要它们公开一个Promises / A +兼容的方法，对thenables的这种处理允许promise实现进行互操作，

它还允许Promises / A +实现使用合理的then方法“同化”不一致的实现。

运行[[Resolve]](promise, x),执行以下步骤:

2.3.1 如果promise和x引用同一个对象，则用TypeError作为原因拒绝 (reject) promise。

2.3.2 如果x是一个promise,采用promise的状态3.4

2.3.2.1 如果x是请求状态(pending),promise必须保持pending直到xfulfilled或rejected

2.3.2.2 如果x是完成态(fulfilled), 用相同的值完成fulfillpromise

2.3.2.2 如果x是拒绝态(rejected), 用相同的原因rejectpromise

2.3.3另外，如果x是个对象或者方法

2.3.3.1 让x作为x.then. 3.5

2.3.3.2 如果取回的x.then属性的结果为一个异常e,用e作为原因reject promise

2.3.3.3 如果then是一个方法，把x当作this来调用它，

第一个参数为 resolvePromise, 第二个参数为rejectPromise, 其中:

2.3.3.3.1 如果/当 resolvePromise被一个值y调用，运行 [[Resolve]](promise, y)

2.3.3.3.2 如果/当 rejectPromise被一个原因r调用，用r拒绝 (reject) promise

2.3.3.3.3 如果resolvePromise和 rejectPromise都被调用，或者对同一个参数进行多次调用，第一次调用执行，任何进一步的调用都被忽略

2.3.3.3.4 如果调用then抛出一个异常e,

2.3.3.3.4.1 如果resolvePromise或 rejectPromise已被调用，忽略。

2.3.3.3.4.2 或者， 用e作为reason拒绝 (reject) promise

2.3.3.4 如果then不是一个函数，用x完成(fulfill)promise

2.3.4 如果 x既不是对象也不是函数，用x完成(fulfill)promise

如果一个promise被一个thenable resolve,并且这个thenable参与了循环的thenable环，

[[Resolve]](promise, thenable)的递归特性最终会引起[[Resolve]](promise, thenable)再次被调用。

遵循上述算法会导致无限递归，鼓励（但不是必须）实现检测这种递归并用包含信息的TypeError作为reason拒绝 (reject) 3.6

代理和反射

代理可以拦截js引擎内部目标的底层对象操作，这些底层操作被拦截后会触发相应特定操作的陷阱函数。调用 new proxy() 可创建代替其他目标 (target) 对象的代理，它虚拟化了目标，所以二者看起来功能一致。

反射api以Reflect对象的形式出现，对象中方法的默认特性与相同的底层操作一致。

思考一下：vue里是如何做到修改数据后，视图就变化了？

创建代理

举个栗子

```
// 不使用任何陷阱的处理等于转发
let target = {};
let proxy = new Proxy(target, {});
proxy.msg = 'zhaowa';
console.log(proxy.msg); // zhaowa
console.log(target.msg); // zhaowa
```

常用陷阱

结合使用代理陷阱和反射api方法可以过滤一些操作，他们默认执行内置行为，只在某些条件下才会表现不同的行为。

代理陷阱：

- set
- get
- has
- deleteProperty
- getPrototypeOf
- setPrototypeOf
- isExtensible
- preventExtensions
- getOwnPropertyDescriptor
- defineProperty
- ownKeys
- apply
- construct

set陷阱

参数

- trapTarget: 用于接受属性（代理的目标）对象；

- key: 要写入的属性键;
- value: 被写入属性的值;
- receiver: 操作发生的对象 (通常是代理)

举个栗子

```
//set陷阱函数
let target = {
    name: 'target'
}
let proxy = new Proxy(target, {
    set(tarpTarget, key, value, receiver){
        if(!tarpTarget.hasOwnProperty(key)){
            if(isNaN(value)){
                throw new Error('属性必须是数字');
            }
        }
        return Reflect.set(tarpTarget, key, value, receiver);
    }
});
proxy.msg = 'hello proxy'; // 属性必须是数字

proxy.const = 1;
console.log(proxy.const); // 1
console.log(target.const); // 1

proxy.name = 'zhaowa';
console.log(proxy.name); // zhaowa 可以给已有属性赋值成非数字
console.log(target.name); // zhaowa
```

get陷阱

参数

- trapTarget: 被读取属性的原对象
- key: 要读取的属性键
- receiver: 操作发生的对象 (通常是代理)

举个栗子

```
let target = {
    name: 'hello world'
};
let proxy = new Proxy(target, {
    get(tarpTarget, key, receiver){
        if(!(key in tarpTarget)){ // 这里检查tarpTarget而不是receiver的原因，是为了避免
            receiver代理中有has陷阱
            throw new Error('不存在该对象');
        }
        return Reflect.get(tarpTarget, key, receiver);
    }
});
```

```
    }
});

console.log(proxy.name); //hello world
console.log(proxy.age); // Uncaught Error: 不存在该对象
```

has陷阱

参数

- trapTarget: 读取属性的对象
- key: 要检查的属性键

举个栗子

```
let target = {
  value: 42,
  name: 'target'
};
let proxy = new Proxy(target, {
  has(tarpTarget, key){
    if(Object.is(key, 'value')){
      return false;
    }
    Reflect.has(tarpTarget, key);
  }
});
console.log('value' in proxy); //false
```

delete陷阱

参数

- trapTarget: 读取属性的对象
- key: 要检查的属性键

举个栗子

```
let target = {
  name: "target",
  value: 42
};
let proxy = new Proxy(target, {
  deleteProperty(trapTarget, key) {
    if (key === "value") {
      return false; // 确保 value 属性不被删除
    } else {
      return Reflect.deleteProperty(trapTarget, key);
    }
  }
});
```

```
// 尝试删除 proxy.value
console.log("value" in proxy); // true
let result = delete proxy.value;
console.log(result); // false
```

可被撤销的代理

当 revoke() 函数被调用后，就不能再对该 proxy 对象进行更多操作

举个栗子

```
let target = {
  name: "target"
};
let { proxy, revoke } = Proxy.revocable(target, {});
console.log(proxy.name); // "target"
revoke();
// 抛出错误
console.log(proxy.name);
```

用模块封装代码

什么是模块

- 模块是自动运行在严格模式下并且没有本办法退出运行的js代码。
- 在模块顶部创建的变量不会自动被添加到全局作用域。
- 模块必须导出一些外部代码可以访问的元素。
- 模块可以从其他模块导入绑定。
- 在模块顶部， this 的值是 undefined
- 模块不支持html风格代码注释

导出基本语法

```
// 导出数据
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;
// 导出函数
export function sum(num1, num2) {
  return num1 + num2;
}
// 导出类
export class Rectangle {
  constructor(length, width) {
    this.length = length;
    this.width = width;
}
```

```
    }
}

// 此函数为模块私有
function subtract(num1, num2) {
    return num1 - num2;
}

// 定义一个函数.....
function multiply(num1, num2) {
    return num1 * num2;
}

// .....导出一个函数引用
export { multiply };
```

导入基本语法

```
// 导入多个绑定
import {sum, multiply, magicNumber} from "./example.js";
console.log(sum(1));

// 完全导入一个模块
import * as example from "./example.js";
console.log(example.sum(1, example.magicNumber)); // 8
console.log(example.multiply(1, 2)); // 2
```

导出和导入时重命名

```
function sum(num1, num2) {
    return num1 + num2;
}
export { sum as add };

import { add as sum } from './example.js'
console.log(typeof add); // "undefined"
console.log(sum(1, 2)); // 3
```

模块的默认值

```
// 不使用标识符
export default function(num1,num2){
    return num1+num2;
}

// 使用标识符
function sum(num1, num2) {
    return num1 + num2;
}
```

```

export default sum;

// 使用重命名语法
function sum(num1, num2) {
    return num1 + num2;
}
export {sum as default};

// 既导出了默认值，又导出非默认值
export let color = 'red';
export default function(num1,num2){
    return num1+num2;
}

```

导入默认值

```

// 只导入默认值
import sum from './example.js';

// 既导入默认值，又导入非默认值
import sum, { color } from './example.js';

// 对导入默认值重命名
import {default as sum, color} from './example.js'

```

对已导入的内容再导出

```

export {sum} from './example.js';
export { sum as add } from './example.js'
export * from './example.js';

```

async-await

async关键字

- 表明程序里面可能有异步过程：`async`关键字表明程序里面可能有异步过程，里面可以有`await`关键字；当然全部是同步代码也没关系，但是这样`async`关键字就显得多余
- 非阻塞：`async`函数里面如果有异步过程会等待，但是`async`函数本身会马上返回，不会阻塞当前线程，可以简单认为，`async`函数工作在主线程，同步执行，不会阻塞界面渲染，`async`函数内部由`await`关键字修饰的异步过程，会阻塞等待异步任务的完成再返回；
- `async`函数返回类型为Promise对象，相当于返回了`Promise.resolve()`；

await关键字

- await只能在async函数内部使用
- await关键字后面跟Promise对象
- await不处理异步error: await是不管异步过程的reject(error)消息的，async函数返回的这个Promise对象的catch函数负责统一抓取内部所有异步过程的错误
- await的结果：如果得到的不是一个Promise对象，那么await表达式的运算结果就是它等到的东西；如果它等到的是一个Promise对象，await就忙起来了，它会阻塞其后面的代码，等着Promise对象resolve，然后得到resolve的值，作为await表达式的运算结果；虽然是阻塞，但async函数调用并不会造成阻塞，它内部所有的阻塞都被封装在一个Promise对象中异步执行，这也正是await必须用在async函数中的原因

举个栗子

```
async function demo() {
  let data = await asyncFunction();
  // ...
  return data;
}
```

举个栗子

```
// promise与async/await对比
function takeLongTime(n) {
  return new Promise(resolve => {
    setTimeout(() => resolve(n + 200), n);
  });
}

function step1(n) {
  console.log(`step1 with ${n}`);
  return takeLongTime(n);
}

function step2(n) {
  console.log(`step2 with ${n}`);
  return takeLongTime(n);
}

function step3(n) {
  console.log(`step3 with ${n}`);
  return takeLongTime(n);
}

// async/await方式
async function doIt() {
  console.time("doIt");
  const time1 = 300;
  const time2 = await step1(time1);
  const time3 = await step2(time2);
  const result = await step3(time3);
}
```

```

        console.log(`result is ${result}`);
        console.timeEnd("doIt");
    }
doIt();

// promise方式
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
            console.log(`result is ${result}`);
            console.timeEnd("doIt");
        });
}

doIt();

```

Decorator装饰器

我们可以在不侵入原有代码的情况下，为代码增加一些额外的功能。一般都比较独立，不和原有逻辑耦合，只是做一层包装。

装饰器

- 首先它是一个函数。
- 这个函数会接收3个参数，分别是target、key和descriptor
- 它可以修改descriptor做一些额外的逻辑。

举个栗子

```

function memoize(target, key, descriptor) {
    ...
}
class Foo {
    @memoize;
    getFooById(id) {
        // ...
    }
}

```

装饰器的3个参数

举个栗子

```

function log(target, key, descriptor) {
    console.log(target);
    console.log(target.hasOwnProperty('constructor'));
    console.log(target.constructor);
    console.log(key);
    console.log(descriptor);
}

class Bar {
    @log;
    bar() {}
}

// {}

// true

// function Bar() { ... }

// bar

// {"enumerable":false,"configurable":true,"writable":true}

```

- key很明显就是当前方法名，我们可以推断出来用于属性的时候就是属性名
- descriptor显然是一个PropertyDescriptor，就是我们用于defineProperty时的那个东西。
- target是一个对象，然后是一个有constructor属性的对象，最后constructur指向的是Bar这个函数。所以这个就是Bar.prototype。

有几种装饰器

- 放在class上的“类装饰器”。
- 放在属性上的“属性装饰器”，这需要配合另一个Stage 0的类属性语法提案，或者只能放在对象字面量上了。
- 放在方法上的“方法装饰器”。
- 放在getter或setter上的“访问器装饰器”。

装饰器在什么时候执行

- 装饰器是在声明期就起效的，并不需要类进行实例化。类实例化并不会致使装饰器多次执行，因此不会对实例化带来额外的开销。
- 按编码时的声明顺序执行，并不会将属性、方法、访问器进行重排序。

举个栗子

```

function randomize(target, key, descriptor) {
    let raw = descriptor.initializer;
    descriptor.initializer = function() {
        let value = raw.call(this);
        value += '-' + Math.floor(Math.random() * 1e6);
        return value;
    };
}
class Alice {

```

```
@randomize;
name = 'alice';
}
console.log((new Alice()).name); // alice-776521
```

模块化

模块化的进化

1、全局方法时代

```
function m1(){
    //...
}
function m2(){
    //...
}
```

2、命名空间时代

```
let myModule = {
    data: 'www.baidu.com',
    foo() {
        console.log(`foo() ${this.data}`)
    },
    bar() {
        console.log(`bar() ${this.data}`)
    }
}
myModule.data = 'other data' //能直接修改模块内部的数据
myModule.foo() // foo() other data
```

3、匿名函数自调用时代

```
(function(window, $) {
    $('body').css('background', 'red');
    ...
})(window, $)
```

模块化的好处

- 避免命名冲突(减少命名空间污染)
- 更好的分离, 按需加载
- 更高复用性
- 高可维护性

模块化规范

CommonJS

Node 应用由模块组成，采用 CommonJS 模块规范。每个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。在服务器端，模块的加载是运行时同步加载的；

CommonJS模块的加载机制是，输入的是被输出的值的拷贝。也就是说，一旦输出一个值，模块内部的变化就影响不到这个值

使用方式

暴露模块：`module.exports = value`或`exports.xxx = value` 引入模块：`require(xxx)`,如果是第三方模块，`xxx`为模块名；如果是自定义模块，`xxx`为模块文件路径

举个栗子

```
// 输出
var x = 5;
var addX = function (value) {
    return value + x;
};
module.exports.x = x;
module.exports.addX = addX;

// 引入
var example = require('./example.js');//如果参数字符串以“./”开头，则表示加载的是一个位于相对路径
console.log(example.x); // 5
console.log(example.addX(1)); // 6
```

AMD

AMD规范则是非同步加载模块，允许指定回调函数。由于Node.js主要用于服务器编程，模块文件一般都已经存在于本地硬盘，所以加载起来比较快，不用考虑非同步加载的方式，所以CommonJS规范比较适用。但是，如果是浏览器环境，要从服务器端加载模块，这时就必须采用非同步模式，因此浏览器端一般采用AMD规范。

使用方式

定义暴露模块：

```
define(['module1', 'module2'], function(m1, m2){ return 模块 })
```

引入使用模块：

```
require(['module1', 'module2'], function(m1, m2){ 使用m1/m2 })
```

`require.js`

RequireJS是一个工具库，主要用于客户端的模块管理。它的模块管理遵守AMD规范，RequireJS的基本思想是，通过define方法，将代码定义为模块；通过require方法，实现代码的模块加载。

CMD

CMD规范专门用于浏览器端，模块的加载是异步的，模块使用时才会加载执行。其实只是一种推荐形式，CMD崇尚就近以来，AMD崇尚前置依赖，但是其实AMD的框架库也支持就近依赖。

使用方式

定义暴露模块：

```
define(['module1', 'module2'], function(m1, m2){ module.exports = 模块 })
```

引入使用模块：

```
require(['module1', 'module2'], function(m1, m2){ 使用m1/m2 })
```

seajs

CMD或者AMD的框架库，如何实现依赖分析的？（讲解seajs的代码）

ES6 Module

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。比如，CommonJS 模块就是对象，输入时必须查找对象属性。

使用方式

export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。（demo代码）

```
// 定义模块 math.js
var basicNum = 0;
var add = function (a, b) {
    return a + b;
};
export { basicNum, add };

// 引用模块
import { basicNum, add } from './math';
function test(ele) {
    ele.textContent = add(99 + basicNum);
}
```

ES6 模块与 CommonJS 模块的差异

- CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。
- CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

第一个差异：ES6 模块的运行机制与 CommonJS 不一样。ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

举个栗子

```
// lib.js
export let counter = 3;
export function incCounter() {
  counter++;
}
// main.js
import { counter, incCounter } from './lib';
console.log(counter); // 3
incCounter();
console.log(counter); // 4
```

第二个差异是因为 CommonJS 加载的是一个对象（即`module.exports`属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

说一说webpack

1. UMD如何定义

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD. Register as an anonymous module.
    define(['b'], factory);
  } else if (typeof module === 'object' && module.exports) {
    // Node. Does not work with strict CommonJS, but
    // only CommonJS-like environments that support module.exports,
    // like Node.
    module.exports = factory(require('b'));
  } else {
    // Browser globals (root is window)
    root.returnExports = factory(root.b);
  }
})(this, function (b) {
  //use b in some fashion.
  // Just return a value to define the module export.
  // This example returns an object, but the module
  // can return a function as the exported value.
  return {};
});
```

1. `require`到底会被编译成什么样子？

2. treeshaking

浏览器事件模型

DOM事件模型和事件流

一个事件发生后，会在子元素和父元素之间传播，分成三个阶段。

- 捕获阶段：事件从window对象自上而下向目标节点传播的阶段；
- 目标阶段：真正的目标节点正在处理事件的阶段；
- 冒泡阶段：事件从目标节点自下而上向window对象传播的阶段。

如何阻止冒泡？

```
event.stopPropagation()
```

阻止默认行为

```
e.preventDefault()
```

事件代理(事件委托)

由于事件会在冒泡阶段向上传播到父节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件。这种方法叫做事件的代理。

BOM 操作

- window.screen对象：包含有关用户屏幕的信息
- window.location对象：用于获得当前页面的地址(URL)，并把浏览器重定向到新的页面
- window.history对象：浏览历史的前进后退等
- window.navigator对象：常常用来获取浏览器信息、是否移动端访问等等

存储

sessionStorage、localStorage 和 cookie 之间的区别

- 共同点：都是保存在浏览器端，且都遵循同源策略。
- 不同点：在于生命周期与作用域的不同

作用域：

localStorage只要在相同的协议、相同的主机名、相同的端口下，就能读取/修改到同一份localStorage数据。 sessionStorage比localStorage更严苛一点，除了协议、主机名、端口外，还要求在同一窗口（也就是浏览器的标签页）下

生命周期：

`localStorage` 是持久化的本地存储，存储在其中的数据是永远不会过期的，使其消失的唯一办法是手动删除；而 `sessionStorage` 是临时性的本地存储，它是会话级别的存储，当会话结束（页面被关闭）时，存储内容也随之被释放。

VUE基础

vue是什么

一套用于构建用户界面的渐进式框架，用来开发web的前端库。通过简单的api提供高效的数据绑定和灵活的组件系统。

特点： 1、轻量级 2、数据绑定 3、指令：通过指令对应的表达式的值的变化，就可以修改对应的dom
4、插件化： ajax、router等都可以以插件形式加载

数据绑定

数据绑定是将数据和视图相关联，当数据发生变化时，可以自动更新视图。

举个栗子

```
// 文本插值
<p>{{text}}</p>

// 插入html
<p v-html="myhtml"></p>
myhtml: <span>我是一个html片段</span>

// 使用表达式
<p>{{number + 1}}</p>
```

指令

指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM

- v-if (v-else v-else-if)
- v-bind:xxx (:xxx)
- v-on:click (@click)
- v-for
- v-model
- v-slot

自定义指令

基本语法如下：

```
Vue.directive('echarts', {
  // 当被绑定的元素插入到 DOM 中时.....
```

```

    inserted: function (el) {
      // 处理DOM元素
    }
  });

// 外部使用时，直接声明即可
<div v-echarts></div>

```

计算属性

模板内的表达式常用于简单的运算，当其过长或逻辑复杂时，会难以维护。因此为了简化逻辑，当某个属性依赖其他属性的值时，我们可以使用计算属性。

举个栗子

```

<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>

var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // 计算属性的 getter
    reversedMessage: function () {
      // `this` 指向 vm 实例
      return this.message.split('').reverse().join('')
    }
  }
})

```

计算属性缓存

那么，既然使用methods就可以实现，为什么还需要计算属性呢？原因就是计算属性是基于它的依赖缓存的，一个计算属性所依赖的数据发生变化时，它才会重新取值，所以只有值不改变，计算属性就不会更新。使用计算属性还是methods取决于是否需要缓存，当便利大数组和做大量计算时，应当使用计算属性。

侦听器

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的侦听器。这就是为什么 Vue 通过 watch 选项提供了一个更通用的方法，来响应数据的变化。当需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的。

举个栗子

```

<div id="watch-example">
  <p>
    Ask a yes/no question:
    <input v-model="question">
  </p>
  <p>{{ answer }}</p>
</div>

var watchExampleVM = new Vue({
  el: '#watch-example',
  data: {
    question: '',
    answer: 'I cannot give you an answer until you ask a question!'
  },
  watch: {
    // 如果 `question` 发生改变，这个函数就会运行
    question: function (newQuestion, oldQuestion) {
      this.answer = 'Waiting for you to stop typing...'
      this.debouncedGetAnswer()
    }
  },
  methods: {
    ...
  }
})

```

过滤器

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化。过滤器可以用在两个地方：双花括号插值和 v-bind 表达式 (后者从 2.1.0+ 开始支持)。过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符号指示：

举个栗子

```

<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
你可以在一个组件的选项中定义本地的过滤器：

filters: {
  capitalize: function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
  }
}

```

Class与Style绑定

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是属性，所以我们可以用 v-bind 处理

class

举个栗子

```
<div class="static"
  v-bind:class="{ active: isActive, 'text-danger': hasError }"
></div>

// 数据
data: {
  isActive: true,
  hasError: false
}

// 结果渲染为:
<div class="static active"></div>
```

style

举个栗子

```
<div v-bind:style="styleObject"></div>

// 数据
data: {
  styleObject: {
    color: 'red',
    fontSize: '13px'
  }
}
```

过渡

在进入/离开的过渡中，会有 6 个 class 切换。

- v-enter: 定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。
- v-enter-active: 定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。

- **v-enter-to:** 2.1.8版及以上 定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 v-enter 被移除), 在过渡/动画完成之后移除。
- **v-leave:** 定义离开过渡的开始状态。在离开过渡被触发时立刻生效, 下一帧被移除。
- **v-leave-active:** 定义离开过渡生效时的状态。在整个离开过渡的阶段中应用, 在离开过渡被触发时立刻生效, 在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间, 延迟和曲线函数。
- **v-leave-to:** 2.1.8版及以上 定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 v-leave 被删除), 在过渡/动画完成之后移除。

事件处理

可以用 v-on 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码。

举个栗子

```
<div id="example-2">
  <!-- `greet` 是在下面定义的方法名 -->
  <button v-on:click="greet">Greet</button>
</div>

var example2 = new Vue({
  el: '#example-2',
  data: {
    name: 'Vue.js'
  },
  // 在 `methods` 对象中定义方法
  methods: {
    greet: function (event) {
      ...
    }
  }
})
```

事件修饰符

- .stop
- .prevent
- .capture
- .self
- .once
- .passive

按键修饰符

- .enter
- .tab
- .delete (捕获“删除”和“退格”键)
- .esc
- .space
- .up
- .down
- .left
- .right

系统修饰符

- .ctrl
- .alt
- .shift
- .meta

鼠标按钮修饰符

- .left
- .right
- .middle

生命周期

所有的生命周期钩子自动绑定 this 上下文到实例中，因此你可以访问数据，对属性和方法进行运算。这意味着你不能使用箭头函数来定义一个生命周期方法（例如 created: () => this.fetchTodos()）。这是因为箭头函数绑定了父上下文，因此 this 与你期待的 Vue 实例不同，this.fetchTodos 的行为未定义。

- beforeCreate
- created: 在实例创建完成后被立即调用。在这一步，实例已完成以下的配置：数据观测 (data observer)，属性和方法的运算，watch/event 事件回调
- beforeMount
- mounted: el 被新创建的 vm.\$el 替换，挂载到实例上去
- beforeUpdate
- updated: 数据更改导致的虚拟 DOM 重新渲染
- activated: keep-alive 组件激活时调用
- deactivated
- beforeDestroy
- destroyed: 实例销毁之后调用
- errorCaptured

组件

组件是可复用的 Vue 实例，且带有一个名字。data、computed、watch、methods 以及生命周期钩子等，组件也是有的。仅有的例外是没有 el 这样根实例特有的选项，并且data必须是个函数。

通过 Prop 向子组件传递数据

举个栗子

```
Vue.component('button-counter', {
  data: function () {
    props: ['title'],
    return {
      count: 0,
      name: title
    }
  },
  template: '<button v-on:click="count++">{{name}} clicked me {{ count }} times.</button>'
})
new Vue({ el: '#components-demo' })

<div id="components-demo">
  <button-counter title="zhaowa"></button-counter>
</div>
```

监听子组件事件

在我们开发子组件时，它的一些功能可能要求我们和父级组件进行沟通。

举个栗子

```
// 子组件
<button v-on:click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>

// 父组件中的使用
<blog-post
  v-on:enlarge-text="onEnlargeText"
></blog-post>

methods: {
  onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

组件的注册

全局注册是指，子组件在任何组件里都可以使用。局部注册是指，父组件只能使用自己内部配置过的子组件。

全局注册所有的组件意味着即便你已经不再使用一个组件了，它仍然会被包含在你最终的构建结果中。这造成了用户下载的 JavaScript 的无谓的增加。

prop

- prop 命名建议使用(短横线分隔命名) 命名
- 通常你希望每个 prop 都有指定的值类型。这时，你可以以对象形式列出 prop，这些属性的名称和值分别是 prop 各自的名称和类型：

```
props: {
  title: String,
  likes: Number,
  isPublished: Boolean,
  commentIds: Array,
  author: Object,
  callback: Function,
  contactsPromise: Promise // or any other constructor
}
```

举个栗子

```
<blog-post v-bind:likes="42"></blog-post>
<blog-post v-bind:is-published="false"></blog-post>
<blog-post v-bind:comment-ids="[234, 266, 273]"></blog-post>
<blog-post v-bind="post"></blog-post> // 传入整个post对象
```

单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个单向下行绑定：父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

prop 验证

可以为组件的 prop 指定验证要求，例如你知道的这些类型。如果有一个需求没有被满足，则 Vue 会在浏览器控制台中警告你。这在开发一个会被别人用到的组件时尤其有帮助。

举个栗子

```
Vue.component('my-component', {
  props: {
    // 基础的类型检查 (`null` 和 `undefined` 会通过任何类型验证)
```

```

propA: Number,
// 多个可能的类型
propB: [String, Number],
// 必填的字符串
propC: {
  type: String,
  required: true
},
// 带有默认值的数字
propD: {
  type: Number,
  default: 100
},
// 带有默认值的对象
propE: {
  type: Object,
  // 对象或数组默认值必须从一个工厂函数获取
  default: function () {
    return { message: 'hello' }
  }
},
// 自定义验证函数
propF: {
  validator: function (value) {
    // 这个值必须匹配下列字符串中的一个
    return ['success', 'warning', 'danger'].indexOf(value) !== -1
  }
}
})

```

自定义事件

在html中不区分大小写，所以建议始终使用 kebab-case 的事件名。

举个栗子

```

<my-component v-on:my-event="doSomething"></my-component>

this.$emit('my-event')

```

插槽

Vue 实现了一套内容分发的 API，将元素作为承载分发内容的出口。可以理解为开发了一个弹窗组件，但只是包含了弹窗的外壳，具体弹窗的内容是使用占位，当使用弹窗组件的地方传入了具体内容后，就替换的位置。

举个栗子

```
// navigation-link组件的内部
<a v-bind:href="url" class="nav-link">
  <slot></slot>
</a>

// 使用时
<navigation-link url="/profile">
  <span class="fa fa-user">我太难了</span>
  Your Profile
</navigation-link>

// 渲染后的效果
<a v-bind:href="url" class="nav-link">
  <span class="fa fa-user">我太难了</span>
  Your Profile
</a>
```

具名插槽

有时我们需要多个插槽，对于这样的情况，元素有一个特殊的特性：name。这个特性可以用来定义额外的插槽，一个不带 name 的出口会带有隐含的名字“default”。

在向具名插槽提供内容的时候，我们可以在一个

举个栗子

```
// 使用时
<base-layout>
  <template v-slot:header>
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template v-slot:footer>
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

动态组件 & 异步组件

在动态组件上使用 keep-alive

我们之前曾经在一个多标签的界面中使用 is 特性来切换不同的组件，当在这些组件之间切换的时候，你有时会想保持这些组件的状态，以避免反复重渲染导致的性能问题。

举个栗子

```
<keep-alive>
  <component :is="view"></component>
</keep-alive>
```

异步组件

在大型应用中，我们可能需要将应用分割成小一些的代码块，并且只在需要的时候才从服务器加载一个模块。为了简化，Vue 允许你以一个工厂函数的方式定义你的组件，这个工厂函数会异步解析你的组件定义。Vue 只有在这个组件需要被渲染的时候才会触发该工厂函数，且会把结果缓存起来供未来重渲染

举个栗子

```
Vue.component('async-webpack-example', function (resolve) {
  // 这个特殊的 `require` 语法将会告诉 webpack
  // 自动将你的构建代码切割成多个包，这些包
  // 会通过 Ajax 请求加载
  require(['./my-async-component'], resolve)
})
```

插件

使用插件

调用 `Vue.use('你的插件');`，如果没有在 `new Vue()` 之前调用的话，本次 `new Vue` 里面的组件是用不上的哦~~

```
Vue.use(Plugin);

new Vue({});
```

开发插件

```
let MyPlugin = {

  install: function (Vue, options) {
    // ...参见课上的内容
  }
}
```

mixin

全局注册一个混入，影响注册之后所有创建的每个 Vue 实例。插件作者可以使用混入，向组件注入自定义的行为

。不推荐在应用代码中使用。

```
Vue.mixin({  
  created() {  
    // xxxx  
  }  
})
```

VUE-ROUTER

router使用

使用 Vue.js，我们已经可以通过组合组件来组成应用程序，当你要把 Vue Router 添加进来，我们需要做的是，将组件 (components) 映射到路由 (routes)，然后告诉 Vue Router 在哪里渲染它们

举个栗子

```
// 路由匹配到的组件将渲染在这里
<div id="app">
  <router-view></router-view>
</div>

// 0. 如果使用模块化机制编程，导入Vue和VueRouter，要调用 Vue.use(VueRouter)

// 1. 定义（路由）组件。
// 可以从其他文件 import 进来
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. 定义路由
// 每个路由应该映射一个组件。 其中"component" 可以是
// 通过 Vue.extend() 创建的组件构造器，
// 或者，只是一个组件配置对象。
// 我们晚点再讨论嵌套路由。
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]

// 3. 创建 router 实例，然后传 `routes` 配置
// 你还可以传别的配置参数，不过先这么简单着吧。
const router = new VueRouter({
  routes // (缩写) 相当于 routes: routes
})

// 4. 创建和挂载根实例。
// 记得要通过 router 配置参数注入路由，
// 从而让整个应用都有路由功能
const app = new Vue({
  router
}).$mount('#app')
```

动态路由匹配

我们经常需要把某种模式匹配到的所有路由，全都映射到同个组件，比如用户信息组件，不同用户使用同一个组件。

举个栗子

```
const router = new VueRouter({
  routes: [
    // 动态路径参数 以冒号开头
    { path: '/user/:id', component: User }
  ]
});

const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
```

响应路由参数的变化

复用组件时，想对路由参数的变化作出响应的话，可以使用 watch 或者 beforeRouteUpdate

举个栗子

```
const User = {
  template: '...',
  watch: {
    '$route' (to, from) {
      // 对路由变化作出响应...
    }
  }
}

const User = {
  template: '...',
  beforeRouteUpdate (to, from, next) {
    // react to route changes...
    // don't forget to call next()
  }
}
```

捕获所有路由或 404 Not found 路由

当使用通配符路由时，请确保路由的顺序是正确的，也就是说含有通配符的路由应该放在最后

举个栗子

```
{
  // 会匹配所有路径
  path: '*'
```

```
}
```

嵌套路由

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各层组件

举个栗子

```
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User,
      children: [
        {
          // 当 /user/:id/profile 匹配成功,
          // UserProfile 会被渲染在 User 的 <router-view> 中
          path: 'profile',
          component: UserProfile
        },
        {
          // 当 /user/:id/posts 匹配成功
          // UserPosts 会被渲染在 User 的 <router-view> 中
          path: 'posts',
          component: UserPosts
        }
      ]
    }
  ]
})
```

编程式的导航

除了使用创建 a 标签来定义导航链接，我们还可以借助 router 的实例方法，通过编写代码来实现。

- router.push
- router.replace
- router.go

命名路由

有时候，通过一个名称来标识一个路由显得更方便一些，特别是在链接一个路由，或者是执行一些跳转的时候。你可以在创建 Router 实例的时候，在 routes 配置中给某个路由设置名称。

举个栗子

```
const router = new VueRouter({
  routes: [
```

```

    {
      path: '/user/:userId',
      name: 'user',
      component: User
    }
  ]
})

router.push({ name: 'user', params: { userId: 123 }})

```

HTML5History模式

vue-router 默认 hash 模式 —— 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。

举个栗子

```

const router = new VueRouter({
  mode: 'history', // hash和history两种模式
  routes: [...]
})

// history模式  http://yoursite.com/user/id
// hash模式    http://yoursite.com#/user/id

```

导航守卫

正如其名，vue-router 提供的导航守卫主要用来通过跳转或取消的方式守卫导航。有多种机会植入路由导航过程中：全局的，单个路由独享的，或者组件级的。

完整的导航解析流程

- 导航被触发。
- 在失活的组件里调用离开守卫。
- 调用全局的 beforeEach 守卫。
- 在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。
- 在路由配置里调用 beforeEnter。
- 解析异步路由组件。
- 在被激活的组件里调用 beforeRouteEnter。
- 调用全局的 beforeResolve 守卫 (2.5+)。
- 导航被确认。
- 调用全局的 afterEach 钩子。
- 触发 DOM 更新。
- 用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

举个栗子

```

// 全局
const router = new VueRouter({ ... })
router.beforeEach((to, from, next) => {
  // ...
})
router.afterEach((to, from) => {
  // ...
})

// 路由独享
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})

// 组件内
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不! 能! 获取组件实例 `this`
    // 因为当守卫执行前, 组件实例还没被创建
  },
  beforeRouteUpdate (to, from, next) {
    // 在当前路由改变, 但是该组件被复用时调用
    // 举例来说, 对于一个带有动态参数的路径 /foo/:id, 在 /foo/1 和 /foo/2 之间跳转的时候,
    // 由于会渲染同样的 Foo 组件, 因此组件实例会被复用。而这个钩子就会在这个情况下被调用。
    // 可以访问组件实例 `this`
  },
  beforeRouteLeave (to, from, next) {
    // 导航离开该组件的对应路由时调用
    // 可以访问组件实例 `this`
  }
}

```

next必须调用

- `next()`: 进行管道中的下一个钩子。如果全部钩子执行完了, 则导航的状态就是 `confirmed` (确认的)。
- `next(false)`: 中断当前的导航。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器后退按钮), 那么 URL 地址会重置到 `from` 路由对应的地址。
- `next('/')` 或者 `next({ path: '/' })`: 跳转到一个不同的地址。当前的导航被中断, 然后进行一个新的导

航。你可以向 next 传递任意位置对象，且允许设置诸如 replace: true、name: 'home' 之类的选项以及任何用在 router-link 的 to prop 或 router.push 中的选项。

- next(error): (2.4.0+) 如果传入 next 的参数是一个 Error 实例，则导航会被终止且该错误会被传递给 router.onError() 注册过的回调。

滚动行为

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。vue-router 能做到，而且更好，它让你可以自定义路由切换时页面如何滚动。

注意：这个功能只在支持 history.pushState 的浏览器中可用。

举个栗子

```
const router = new VueRouter({
  routes: [...],
  scrollBehavior (to, from, savedPosition) {
    // return 期望滚动到哪个的位置
  }
})
```

路由懒加载

当打包构建应用时，JavaScript 包会变得非常大，影响页面加载。如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了。

举个栗子

```
const Foo = () => import('./Foo.vue')

const router = new VueRouter({
  routes: [
    { path: '/foo', component: Foo }
  ]
})
```

VUE-CLI

vuecli介绍

Vue CLI 是一个基于 Vue.js 进行快速开发的完整系统，提供：

- 通过 @vue/cli 搭建交互式的项目脚手架。
- 通过 @vue/cli + @vue/cli-service-global 快速开始零配置原型开发。
- 一个运行时依赖 (@vue/cli-service)
- 一个丰富的官方插件集合，集成了前端生态中最好的工具。
- 一套完全图形化的创建和管理 Vue.js 项目的用户界面。

安装

```
npm install -g @vue/cli
```

基础

快速原型开发

```
npm install -g @vue/cli-service-global

// 启动一个服务器
vue serve xxx.vue

// 将目标文件构建成一个生产环境的包
vue build xxx.vue
```

创建一个项目

```
vue create mytest // 创建一个由 `vue-cli-service` 提供支持的新项目
```

你会被提示选取一个 preset。你可以选默认的包含了基本的 Babel + ESLint 设置的 preset，也可以选“手动选择特性”来选取需要的特性。这个默认的设置非常适合快速创建一个新项目的原型，而手动设置则提供了更多的选项，它们是面向生产的项目更加需要的。

使用图形化界面

```
vue ui
```

插件和预设配置

插件

Vue CLI 使用了一套基于插件的架构，`package.json`中依赖都是以 `@vue/cli-plugin-` 开头的。插件可以修改内部的 `webpack` 配置，也可以向 `vue-cli-service` 注入命令。在项目创建的过程中列出的特性，绝大部分都是通过插件来实现的。

```
vue add eslint // 在现有的项目中安装插件
```

预设配置

Vue CLI 预设配置是一个包含创建新项目所需的预定义选项和插件的 JSON 对象，让用户无需在命令提示中选择它们。在 `vue create` 过程中保存的预设配置会被放在你的 `home` 目录下的一个配置文件中 (`~/.vuerc`)。你可以通过直接编辑这个文件来调整、添加、删除保存好的配置。

这里有一个预设配置的示例：

```
{
  "useConfigFiles": true,
  "router": true,
  "vuex": true,
  "cssPreprocessor": "sass",
  "plugins": {
    "@vue/cli-plugin-babel": {},
    "@vue/cli-plugin-eslint": {
      "config": "airbnb",
      "lintOn": ["save", "commit"]
    }
  }
}
```

CLI 服务

在一个 Vue CLI 项目中，`@vue/cli-service` 安装了一个名为 `vue-cli-service` 的命令。你可以在 `npm scripts` 中以 `vue-cli-service`、或者从终端中以 `./node_modules/.bin/vue-cli-service` 访问这个命令。

```
{
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build"
  }
}

npm run serve
```

开发

浏览器兼容性

一个默认的 Vue CLI 项目会使用 `@vue/babel-preset-app`, 它通过 `@babel/preset-env` 和 `browserslist` 配置来决定项目需要的 polyfill。

默认情况下, 它会把 `useBuiltIns: 'usage'` 传递给 `@babel/preset-env`, 这样它会根据源代码中出现的语言特性自动检测需要的 polyfill。这确保了最终包里 polyfill 数量的最小化。然而, 这也意味着如果其中一个依赖需要特殊的 polyfill, 默认情况下 Babel 无法将其检测出来。

如果有依赖需要 polyfill, 你有几种选择:

- 如果该依赖基于一个目标环境不支持的 ES 版本撰写: 将其添加到 `vue.config.js` 中的 `transpileDependencies` 选项。这会为该依赖同时开启语法转换和根据使用情况检测 polyfill。
- 如果该依赖交付了 ES5 代码并显式地列出了需要的 polyfill: 你可以使用 `@vue/babel-preset-app` 的 `polyfills` 选项预包含所需要的 polyfill。注意 `es6.promise` 将被默认包含, 因为现在的库依赖 `Promise` 是非常普遍的。

```
// babel.config.js
module.exports = {
  presets: [
    ['@vue/app', {
      polyfills: [
        'es6.promise',
        'es6.symbol'
      ]
    }]
  ]
}
```

- 如果该依赖交付 ES5 代码, 但使用了 ES6+ 特性且没有显式地列出需要的 polyfill (例如 Vuetify): 请使用 `useBuiltIns: 'entry'` 然后在入口文件添加 `import '@babel/polyfill'`。这会根据 `browserslist` 目标导入所有 polyfill, 这样你就不用再担心依赖的 polyfill 问题了, 但是因为包含了一些没有用到的 polyfill 所以最终的包大小可能会增加。

现代模式

有了 Babel 我们可以兼顾所有最新的 ES2015+ 语言特性, 但也意味着我们需要交付转译和 polyfill 后的包以支持旧浏览器。这些转译后的包通常都比原生的 ES2015+ 代码会更冗长, 运行更慢。现如今绝大多数现代浏览器都已经支持了原生的 ES2015, 所以因为要支持更老的浏览器而为它们交付笨重的代码是一种浪费。

Vue CLI 提供了一个“现代模式”帮你解决这个问题。以如下命令为生产环境构建:

```
vue-cli-service build --modern
```

Vue CLI 会产生两个应用的版本：一个现代版的包，面向支持 ES modules 的现代浏览器，另一个旧版的包，面向不支持的旧浏览器。

HTML

Preload

是一种 resource hint，用来指定页面加载后很快会被用到的资源，所以在页面加载的过程中，我们希望在浏览器开始主体渲染之前尽早 preload。

默认情况下，一个 Vue CLI 应用会为所有初始化渲染需要的文件自动生成 preload 提示。

这些提示会被 `@vue/preload-webpack-plugin` 注入，并且可以通过 `chainWebpack` 的 `config.plugin('preload')` 进行修改和删除。

Prefetch

是一种 resource hint，用来告诉浏览器在页面加载完成后，利用空闲时间提前获取用户未来可能会访问的内容。

默认情况下，一个 Vue CLI 应用会为所有作为 `async chunk` 生成的 JavaScript 文件（通过动态 `import()` 按需 code splitting 的产物）自动生成 prefetch 提示。

这些提示会被 `@vue/preload-webpack-plugin` 注入，并且可以通过 `chainWebpack` 的 `config.plugin('prefetch')` 进行修改和删除。

举个栗子

```
// vue.config.js
module.exports = {
  chainWebpack: config => {
    // 移除 prefetch 插件
    config.plugins.delete('prefetch')

    // 或者
    // 修改它的选项：
    config.plugin('prefetch').tap(options => {
      options[0].fileBlacklist = options[0].fileBlacklist || []
      options[0].fileBlacklist.push(/myasyncRoute(.+?)\.js$/)
      return options
    })
  }
}
```

当 `prefetch` 插件被禁用时，你可以通过 webpack 的内联注释手动选定要提前获取的代码区块：

```
import(* webpackPrefetch: true */ './someAsyncComponent.vue')
```

webpack 的运行时会在父级区块被加载之后注入 `prefetch` 链接。

处理静态资源

静态资源可以通过两种方式进行处理：

- 在 JavaScript 被导入或在 template/CSS 中通过相对路径被引用。这类引用会被 webpack 处理。
当你在 JavaScript、CSS 或 *.vue 文件中使用相对路径（必须以 . 开头）引用一个静态资源时，该资源将会被包含进入 webpack 的依赖图中。在其内部，我们通过 file-loader 用版本哈希值和正确的公共基础路径来决定最终的文件路径，再用 url-loader 将小于 4kb 的资源内联，以减少 HTTP 请求的数量。
- 放置在 public 目录下或通过绝对路径被引用。这类资源将会直接被拷贝，而不会经过 webpack 的处理。

CSS 相关

Vue CLI 项目天生支持 PostCSS、CSS Modules 和包含 Sass、Less、Stylus 在内的预处理器。

- 所有编译后的 CSS 都会通过 css-loader 来解析其中的 url() 引用，并将这些引用作为模块请求来处理。这意味着你可以根据本地的文件结构用相对路径来引用静态资源。
- 你可以在创建项目的时候选择预处理器（Sass/Less/Stylus）。如果当时没有选好，内置的 webpack 仍然会被预配置为可以完成所有的处理。

webpack 相关

调整 webpack 配置最简单的方式就是在 vue.config.js 中的 configureWebpack 选项提供一个对象，该对象将会被 webpack-merge 合并入最终的 webpack 配置。

```
// vue.config.js
module.exports = {
  configureWebpack: {
    plugins: [
      new MyAwesomeWebpackPlugin()
    ]
  }
}
```

构建目标

当你运行 vue-cli-service build 时，你可以通过 --target 选项指定不同的构建目标。应用模式是默认的模式。在这个模式中：

- index.html 会带有注入的资源和 resource hint
- 第三方库会被分到一个独立包以便更好的缓存
- 小于 4kb 的静态资源会被内联在 JavaScript 中
- public 中的静态资源会被复制到输出目录中

部署

如果你独立于后端部署前端应用——也就是说后端暴露一个前端可访问的 API，然后前端实际上是纯静态应用。那么你可以将 `dist` 目录里构建的内容部署到任何静态文件服务器中，但要确保正确的 `publicPath`。

微信小程序的开发与原理

小程序的诞生

小程序并非凭空冒出来的一个概念。当微信中的 WebView 逐渐成为移动 Web 的一个重要入口时，微信就有相关的 JS-SDK 了。JS-SDK 解决了移动网页能力不足的问题，通过暴露微信的接口使得 Web 开发者能够拥有更多的能力，然而在更多的能力之外，JS-SDK 的模式并没有解决使用移动网页遇到的体验不良的问题。

微信面临的问题是如何设计一个比较好的系统，使得所有开发者在微信中都能获得比较好的体验。这个问题是之前的 JS-SDK 所处理不了的，需要一个全新的系统来完成，它需要使得所有的开发者都能做到：

- 快速的加载
- 更强大的能力
- 原生的体验
- 易用且安全的微信数据开放
- 高效和简单的开发

这就是小程序的由来。

小程序与普通网页开发的区别

小程序的主要开发语言是 JavaScript，小程序的开发同普通的网页开发相比有很大的相似性。对于前端开发者而言，从网页开发迁移到小程序的开发成本并不高，但是二者还是有些许区别的。

网页开发渲染线程和脚本线程是互斥的，这也是为什么长时间的脚本运行可能会导致页面失去响应，而在小程序中，二者是分开的，分别运行在不同的线程中。网页开发者可以使用到各种浏览器暴露出来的 DOM API，进行 DOM 选中和操作。而如上文所述，小程序的逻辑层和渲染层是分开的，逻辑层运行在 JSCore 中，并没有一个完整浏览器对象，因而缺少相关的DOM API和BOM API。这一区别导致了前端开发非常熟悉的一些库，例如 jQuery、Zepto 等，在小程序中是无法运行的。同时 JSCore 的环境同 NodeJS 环境也是不尽相同，所以一些 NPM 的包在小程序中也是无法运行的。

网页开发者需要面对的环境是各式各样的浏览器，PC 端需要面对 IE、Chrome、QQ 浏览器等，在移动端需要面对 Safari、Chrome 以及 iOS、Android 系统中的各式 WebView。而小程序开发过程中需要面对的是两大操作系统 iOS 和 Android 的微信客户端，以及用于辅助开发的小程序开发者工具，小程序中三大运行环境也是有所区别的

小程序配置

小程序开发目录结构：

```
|── app.js  
|── app.json  
|── app.wxss
```

```

├─ pages
|  ├─ index
|  |  ├─ index.wxml
|  |  ├─ index.js
|  |  ├─ index.json
|  |  └─ index.wxss
|  └─ logs
|      ├─ logs.wxml
|      └─ logs.js
└─ utils

```

JSON 语法

这里说一下小程序里JSON配置的一些注意事项。

JSON文件都是被包裹在一个大括号中 {}，通过key-value的方式来表达数据。JSON的Key必须包裹在一个双引号中，在实践中，编写 JSON 的时候，忘了给 Key 值加双引号或者是把双引号写成单引号是常见错误。

JSON的值只能是以下几种数据格式，其他任何格式都会触发报错，例如 JavaScript 中的 undefined。

- 数字，包含浮点数和整数
- 字符串，需要包裹在双引号中
- Bool值，true 或者 false
- 数组，需要包裹在方括号中 []
- 对象，需要包裹在大括号中 {}
- Null

还需要注意的是 JSON 文件中无法使用注释，试图添加注释将会引发报错。

app.json配置

app.json 是当前小程序的全局配置，包括了小程序的所有页面路径、界面表现、网络超时时间、底部 tab 等。

举个栗子

```
{
  "pages": [
    "pages/index/index",
    "pages/logs/logs"
  ],
  "window": {
    "backgroundTextStyle": "light",
    "navigationBarBackgroundColor": "#fff",
    "navigationBarTitleText": "WeChat",
    "navigationBarTextStyle": "black"
  }
}
```

我们简单说一下这个配置各个项的含义：

- **pages**字段 —— 用于描述当前小程序所有页面路径，这是为了让微信客户端知道当前你的小程序页面定义在哪个目录。
- **window**字段 —— 定义小程序所有页面的顶部背景颜色，文字颜色定义等。

页面配置

每一个小程序页面也可以使用同名.json文件来对本页面的窗口表现进行配置，页面中配置项会覆盖app.json的window中相同的配置项。

举个栗子

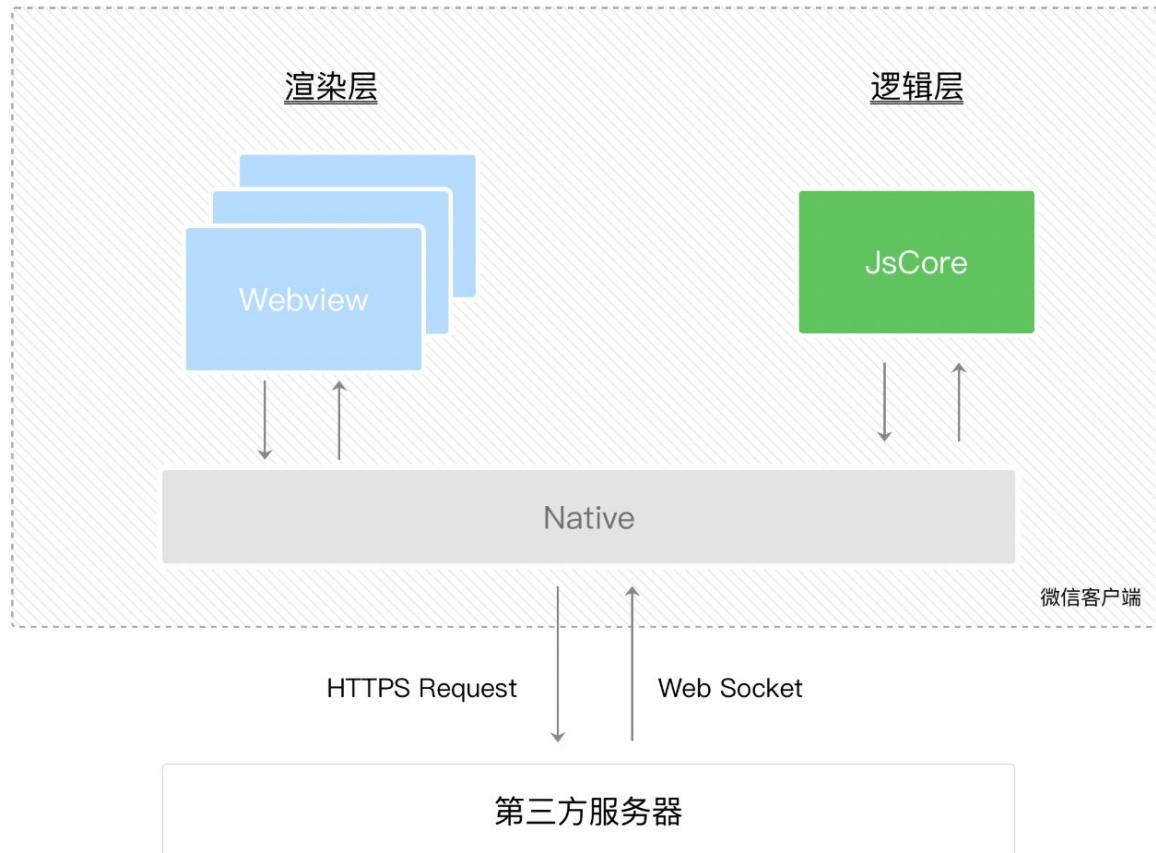
```
{  
  "navigationBarBackgroundColor": "#ffffff",  
  "navigationBarTextStyle": "black",  
  "navigationBarTitleText": "微信接口功能演示",  
  "backgroundColor": "#eeeeee",  
  "backgroundTextStyle": "light"  
}
```

小程序的运行时

渲染层和逻辑层

首先，我们来简单了解下小程序的运行环境。小程序的运行环境分成渲染层和逻辑层，其中 WXML 模板和 WXSS 样式工作在渲染层，JS 脚本工作在逻辑层。

小程序的渲染层和逻辑层分别由2个线程管理：渲染层的界面使用了WebView 进行渲染；逻辑层采用 JsCore 线程运行JS脚本。一个小程序存在多个界面，所以渲染层存在多个WebView线程，这两个线程的通信会经由微信客户端（下文中也会采用Native来代指微信客户端）做中转，逻辑层发送网络请求也经由Native转发。



逻辑层 App Service

小程序开发框架的逻辑层使用 JavaScript 引擎为小程序提供开发者 JavaScript 代码的运行环境以及微信小程序的特有功能。

逻辑层将数据进行处理后发送给视图层，同时接受视图层的事件反馈。

开发者写的所有代码最终将会打包成一份 JavaScript 文件，并在小程序启动的时候运行，直到小程序销毁。这一行为类似 ServiceWorker，所以逻辑层也称之为 App Service。

小程序运行机制

前台/后台状态

小程序启动后，界面被展示给用户，此时小程序处于前台状态。

当用户点击右上角胶囊按钮关闭小程序，或者按了设备 Home 键离开微信时，小程序并没有完全终止运行，而是进入了后台状态，小程序还可以运行一小段时间。

当用户再次进入微信或再次打开小程序，小程序又会从后台进入前台。但如果用户很久没有再进入小程序，或者系统资源紧张，小程序可能被销毁，即完全终止运行。

小程序启动

这样，小程序启动可以分为两种情况，一种是冷启动，一种是热启动。

冷启动：如果用户首次打开，或小程序销毁后被用户再次打开，此时小程序需要重新加载启动，即冷启动。热启动：如果用户已经打开过某小程序，然后在一定时间内再次打开该小程序，此时小程序并未被销毁，只是从后台状态进入前台状态，这个过程就是热启动。

小程序销毁时机

通常，只有当小程序进入后台一定时间，或者系统资源占用过高，才会被销毁。具体而言包括以下几种情形。

当小程序进入后台，可以会维持一小段时间的运行状态，如果这段时间内都未进入前台，小程序会被销毁。当小程序占用系统资源过高，可能会被系统销毁或被微信客户端主动回收。

- 在 iOS 上，当微信客户端在一定时间间隔内（目前是 5 秒）连续收到两次及以上系统内存告警时，会主动进行小程序的销毁，并提示用户「该小程序可能导致微信响应变慢被终止」。
- 建议小程序在必要时使用 `wx.onMemoryWarning` 监听内存告警事件，进行必要的内存清理。

退出状态

每当小程序可能被销毁之前，页面回调函数 `onSaveExitState` 会被调用。如果想保留页面中的状态，可以在这个回调函数中“保存”一些数据，下次启动时可以通过 `exitState` 获得这些已保存数据。

小程序的开发

在 JavaScript 的基础上，我们增加了一些功能，以方便小程序的开发：

- 增加 App 和 Page 方法，进行程序注册和页面注册。
- 增加 `getApp` 和 `getCurrentPages` 方法，分别用来获取 App 实例和当前页面栈。
- 提供丰富的 API，如微信用户数据，扫一扫，支付等微信特有能力。
- 提供模块化能力，每个页面有独立的作用域。

注意：小程序框架的逻辑层并非运行在浏览器中，因此 JavaScript 在 web 中一些能力都无法使用，如 `window`, `document` 等。

app

- 微信客户端在打开小程序之前，会把整个小程序的代码包下载到本地。紧接着通过 `app.json` 的 `pages` 字段就可以知道你当前小程序的所有页面路径。
- 写在 `app.json` 中 `pages` 字段的第一个页面就是这个小程序的首页（打开小程序看到的第一个页面）。
- 于是微信客户端就把首页的代码装载进来，通过小程序底层的一些机制，就可以渲染出这个首页。
- 小程序启动之后，在 `app.js` 定义的 App 实例的 `onLaunch` 回调会被执行
- 整个小程序只有一个 App 实例，是全部页面共享的

```
App({  
  onLaunch (options) {
```

```
// Do something initial when launch.  
},  
onShow (options) {  
    // Do something when show.  
},  
onHide () {  
    // Do something when hide.  
},  
onError (msg) {  
    console.log(msg)  
},  
globalData: 'I am global data'  
})
```

- 开发者可以通过 `getApp` 方法获取到全局唯一的 App 示例，获取App上的数据或调用开发者注册在 App 上的函数。

```
// xxx.js  
const appInstance = getApp()  
console.log(appInstance.globalData) // I am global data
```

page

小程序里边包含了不同类型的文件:

- .json 后缀的 JSON 配置文件
- .wxml 后缀的 WXML 模板文件
- .wxss 后缀的 WXSS 样式文件
- .js 后缀的 JS 脚本逻辑文件

在js文件中，Page 是一个页面构造器，这个构造器就生成了一个页面。在生成页面的时候，小程序框架会把 `data` 数据和 `index.wxml` 一起渲染出最终的结构，于是就得到了你看到的小程序的样子。

在渲染完界面之后，页面实例就会收到一个 `onLoad` 的回调，你可以在这个回调处理你的逻辑。

```
Page({  
    data: {  
        text: "This is page data."  
    },  
    onLoad: function(options) {  
        // 页面创建时执行  
    },  
    onShow: function() {  
        // 页面出现在前台时执行  
    },  
    onReady: function() {  
        // 页面首次渲染完毕时执行  
    },  
    onHide: function() {  
        // 页面隐藏时执行  
    }  
})
```

```
// 页面从前台变为后台时执行
},
onUnload: function() {
// 页面销毁时执行
},
onPullDownRefresh: function() {
// 触发下拉刷新时执行
},
onReachBottom: function() {
// 页面触底时执行
},
onShareAppMessage: function () {
// 页面被用户分享时执行
},
onPageScroll: function() {
// 页面滚动时执行
},
onResize: function() {
// 页面尺寸变化时执行
},
onTabItemTap(item) {
// tab 点击时执行
console.log(item.index)
console.log(item.pagePath)
console.log(item.text)
},
// 事件响应函数
viewTap: function() {
this.setData({
    text: 'Set some data for updating view.'
}, function() {
    // this is setData callback
})
},
// 自由数据
customData: {
hi: 'MINA'
}
})
```

框架的核心是一个响应的数据绑定系统，可以让数据与视图非常简单地保持同步。当做数据修改的时候，只需要在逻辑层修改数据，视图层就会做相应的更新。

举个栗子

```
<!-- This is our View -->
<view> Hello {{name}}! </view>
<button bindtap="changeName"> Click me! </button>

// Register a Page.
```

```
Page({
  data: {
    name: 'WeChat'
  },
  changeName: function(e) {
    // sent data change to view
    this.setData({
      name: 'MINA'
    })
  }
})
```

组件

小程序提供了丰富的基础组件给开发者，开发者可以像搭积木一样，组合各种组件拼合成自己的小程序。

就像 HTML 的 `div`, `p` 等标签一样，在小程序里边，你只需要在 WXML 写上对应的组件标签名字就可以把该组件显示在界面上，例如，你需要在界面上显示地图，你只需要这样写即可：

```
<map></map>
```

使用组件的时候，还可以通过属性传递值给组件，让组件可以以不同的状态去展现，例如，我们希望地图一开始的中心的经纬度是广州，那么你需要声明地图的 `longitude`（中心经度）和 `latitude`（中心纬度）两个属性：

```
<map longitude="广州经度" latitude="广州纬度"></map>
```

api

为了让开发者可以很方便的调起微信提供的能力，例如获取用户信息、微信支付等等，小程序提供了很多 API 给开发者去使用。需要注意的是：多数 API 的回调都是异步，你需要处理好代码逻辑的异步问题。

举个栗子

```
wx.getLocation({
  type: 'wgs84',
  success: (res) => {
    var latitude = res.latitude // 纬度
    var longitude = res.longitude // 经度
  }
})
```

自定义组件

开发者可以将页面内的功能模块抽象成自定义组件，以便在不同的页面中重复使用；也可以将复杂的页面拆分成多个低耦合的模块，有助于代码维护。自定义组件在使用时与基础组件非常相似。

开发自定义组件

类似于页面，一个自定义组件由 json wxml wxss js 4个文件组成。要编写一个自定义组件，首先需要在 json 文件中进行自定义组件声明（将 component 字段设为 true 可这一组文件设为自定义组件）：

```
{
  "component": true
}
```

在自定义组件的 js 文件中，需要使用 Component() 来注册组件，并提供组件的属性定义、内部数据和自定义方法。组件的属性值和内部数据将被用于组件 wxml 的渲染，其中，属性值是可由组件外部传入的。

```
Component({
  properties: {
    // 这里定义了innerText属性，属性值可以在组件使用时指定
    innerText: {
      type: String,
      value: 'default value',
    }
  },
  data: {
    // 这里是一些组件内部数据
    someData: {}
  },
  methods: {
    // 这里是一个自定义方法
    customMethod: function(){}
  }
})
```

使用自定义组件

使用已注册的自定义组件前，首先要在页面的 json 文件中进行引用声明。此时需要提供每个自定义组件的标签名和对应的自定义组件文件路径：

```
{
  "usingComponents": {
    "component-tag-name": "path/to/the/custom/component"
  }
}
```

这样，在页面的 wxml 中就可以像使用基础组件一样使用自定义组件。节点名即自定义组件的标签名，节点属性即传递给组件的属性值。

```
<view>
  <!-- 以下是对一个自定义组件的引用 -->
  <component-tag-name inner-text="Some text"></component-tag-name>
</view>
```

自定义组件的 wxml 节点结构在与数据结合之后，将被插入到引用位置内。

常见的 setData 操作错误

1. 频繁的去 setData

在我们分析过的一些案例里，部分小程序会非常频繁（毫秒级）的去setData，其导致了两个后果：

Android 下用户在滑动时会感觉到卡顿，操作反馈延迟严重，因为 JS 线程一直在编译执行渲染，未能及时将用户操作事件传递到逻辑层，逻辑层亦无法及时将操作处理结果及时传递到视图层；渲染有出现延时，由于 WebView 的 JS 线程一直处于忙碌状态，逻辑层到页面层的通信耗时上升，视图层收到的数据消息时距离发出时间已经过去了几百毫秒，渲染的结果并不实时；

1. 每次 setData 都传递大量新数据

由setData的底层实现可知，我们的数据传输实际是一次 evaluateJavascript 脚本过程，当数据量过大时会增加脚本的编译执行时间，占用 WebView JS 线程，

1. 后台态页面进行 setData

当页面进入后台态（用户不可见），不应该继续去进行setData，后台态页面的渲染用户是无法感受的，另外后台态页面去setData也会抢占前台页面的执行。

分包

某些情况下，开发者需要将小程序划分成不同的子包，在构建时打包成不同的分包，用户在使用时按需进行加载。

在构建小程序分包项目时，构建会输出一个或多个分包。每个使用分包小程序必定含有一个主包。所谓的主包，即放置默认启动页面/TabBar 页面，以及一些所有分包都需要用到公共资源/Javascript 脚本；而分包则是根据开发者的配置进行划分。

在小程序启动时，默认会下载主包并启动主包内页面，当用户进入分包内某个页面时，客户端会把对应分包下载下来，下载完成后再进行展示。

目前小程序分包大小有以下限制：

- 整个小程序所有分包大小不超过 8M
- 单个分包/主包大小不能超过 2M

使用分包

假设支持分包的小程序目录结构如下：

```
├── app.js
├── app.json
├── app.wxss
├── packageA
│   └── pages
│       ├── cat
│       └── dog
└── packageB
    └── pages
        ├── apple
        └── banana
├── pages
│   ├── index
│   └── logs
└── utils
```

开发者通过在 `app.json` `subpackages` 字段声明项目分包结构：

```
{
  "pages": [
    "pages/index",
    "pages/logs"
  ],
  "subpackages": [
    {
      "root": "packageA",
      "pages": [
        "pages/cat",
        "pages/dog"
      ]
    },
    {
      "root": "packageB",
      "name": "pack2",
      "pages": [
        "pages/apple",
        "pages/banana"
      ]
    }
  ]
}
```

打包原则

- 声明 `subpackages` 后，将按 `subpackages` 配置路径进行打包，`subpackages` 配置路径外的目录将被打包到 `app`（主包）中
- `app`（主包）也可以有自己的 `pages`（即最外层的 `pages` 字段）
- `subpackage` 的根目录不能是另外一个 `subpackage` 内的子目录
- `tabBar` 页面必须在 `app`（主包）内

js基础

数组方法

map

- 返回数组，原数组有多少位，map后还有多少位。
- 需要return

```
const newArray = [1, 2, 3, 4].map(item => item*2);
console.log(newArray); // [2,4,6,8]
```

或者

```
const newArray = [1, 2, 3, 4].map(item => {
    return item * 2;
});
console.log(newArray); // [2,4,6,8]
```

filter

- 返回数据，在原数组中筛选出合适的项，filter后的长度小于等于原数组
- 需要return

```
const newArray = [1, 2, 3, 4].filter(item => item > 2);
console.log(newArray); // [3,4]
```

find

- 返回原数组中的某一项，或者为空。遇到第一个合适的项后，就结束循环
- 需要return

```
const result = [1, 2, 3, 4].find(item => item > 2);
console.log(result); // 3
```

forEach

- 循环数组，return不是必须的

```
let result = [];
[1, 2, 3, 4].forEach(item => {
    if (item > 2) {
        result.push(item);
```

```
    }
});

console.log(result); // [3,4]
```

every

- 对数组中每一项运行给定函数，如果该函数的每一项都返回true，则返回true

```
const isNum = [1, 2, '3', 4].every(item => typeof item === 'number');
console.log(isNum); // false
```

some

- 对数组中每一项运行给定函数，如果该函数有一项都返回true，则返回true

```
const hasNum = [1, 2, '3', 4].some(item => typeof item === 'number');
console.log(hasNum); // true
```

for...of

- 允许遍历获得键值，但是不能循环对象

```
let arr = [1, 'a', 2, 3, 4, 'a'];
for(let i of arr) {
  console.log(i); // 1, a, 2, 3, 4, a
}
```

reduce

- 可以实现一个去重功能

```
let arr = [1,2,3,4,4,1]
let newArr = arr.reduce((pre, cur) => {
  if (!pre.includes(cur)) {
    return pre.concat(cur);
  } else {
    return pre;
  }
}, []);
console.log(newArr); // [1, 2, 3, 4]
```

对象方法

for-in遍历

- `for-in`是为遍历对象而设计的，不适用于遍历数组。

```
let obj = {
  a: '11',
  b: 2
};
for(let i in obj) {
  console.log(i); // a, b
  console.log(obj[i]); // '11', 2
}
```

Object.keys()

- 循环对象的key，返回数组

```
let obj = {
  a: '11',
  b: 2
};
let keys = Object.keys(obj); // [a, b]
```

Object.values()

- 循环对象的value，返回数组

```
let obj = {
  a: '11',
  b: 2
};
let values = Object.values(obj); // ['11', 2]
```

Object.getOwnPropertyNames()

- 返回一个数组，包含对象自身（不含继承）的所有属性名

```
var Person = function({name='none', age=18, height=170} = {}){
  this.name = name;
  this.age = age;
  this.height = height;
}

Person.prototype = {
  type: 'Animal'
}

var qiu = new Person()
```

```
// 将height属性设置为 不可枚举
Object.defineProperty(qiu, 'height', {
  enumerable: false
})

var keys = Object.getOwnPropertyNames(qiu);
console.log(keys); // ['name', 'age', 'height']
```

正则

使用正则表达式的方法

方法	描述
exec	一个在字符串中执行查找匹配的RegExp方法，它返回一个数组（未匹配到则返回null）。
test	一个在字符串中测试是否匹配的RegExp方法，它返回 true 或 false。
match	一个在字符串中执行查找匹配的String方法，它返回一个数组，在未匹配到时会返回null。
search	一个在字符串中测试匹配的String方法，它返回匹配到的位置索引，或者在失败时返回-1。
replace	一个在字符串中执行查找匹配的String方法，并且使用替换字符串替换掉匹配到的子字符串。

边界

字符	含义
^	匹配输入的开始。如果多行标志被设置为 true，那么也匹配换行符后紧跟的位置。例如，/^A/ 并不会匹配 "an A" 中的 'A'，但是会匹配 "An E" 中的 'A'。当 '^' 作为第一个字符出现在一个字符集合模式时，它将会有不同的含义。补充字符集合一节有详细介绍和示例。
\$	匹配输入的结束。如果多行标志被设置为 true，那么也匹配换行符前的位置。例如，/t\$/ 并不会匹配 "eater" 中的 't'，但是会匹配 "eat" 中的 't'。
\b	匹配一个词的边界。一个词的边界就是一个词不被另外一个“字”字符跟随的位置或者前面跟其他“字”字符的位置，例如在字母和空格之间。注意，匹配中不包括匹配的字边界。换句话说，一个匹配的词的边界的内容的长度是0。（不要和\b混淆了）
\B	匹配一个非单词边界。匹配如下几种情况：1、字符串第一个字符为非“字”字符。2、字符串最后一个字符为非“字”字符。3、两个单词字符之间。4、两个非单词字符之间。5、空字符串。例如，\B..匹配"noonday"中的'oo'，而\b..匹配"possibly yesterday"中的'yes'

字符类别

字符	含义

.	(小数点) 默认匹配除换行符之外的任何单个字符。例如, <code>/.\n/</code> 将会匹配 "nay, an apple is on the tree" 中的 'an' 和 'on', 但是不会匹配 'nay'。如果 <code>s("dotAll")</code> 标志位被设为 true, 它也会匹配换行符。
\d	匹配一个数字。等价于 [0-9]。例如, <code>\d/</code> 或者 <code>/[0-9]/</code> 匹配 "B2 is the suite number." 中的 '2'。
\D	匹配一个非数字字符。等价于 <code>^0-9</code> 。例如, <code>\D/</code> 或者 <code>/^0-9/</code> 匹配 "B2 is the suite number." 中的 'B'。
\s	匹配一个空白字符, 包括空格、制表符、换页符和换行符。例如, <code>\s\w*/</code> 匹配 "foo bar." 中的 ' bar'。
\S	匹配一个非空白字符。等价于 <code>^</code>
\w	匹配一个单字字符 (字母、数字或者下划线)。等价于 <code>[A-Za-z0-9_]</code> 。例如, <code>\w/</code> 匹配 "apple," 中的 'a', "\$5.28," 中的 '5' 和 "3D." 中的 '3'。
\W	匹配一个非单字字符。等价于 <code>A-Za-z0-9_</code> 。例如, <code>\W/</code> 或者 <code>/A-Za-z0-9_</code> 匹配 "50%." 中的 '%'。

量词

字符	含义
*	匹配前一个表达式 0 次或多次。等价于 <code>{0,}</code> 。例如, <code>/bo*/</code> 会匹配 "A ghost boooooed" 中的 'booooo' 和 "A bird warbled" 中的 'b', 但是在 "A goat grunted" 中不会匹配任何内容。
+	匹配前一个表达式 1 次或多次。等价于 <code>{1,}</code> 。例如, <code>/a+/</code> 会匹配 "candy" 中的 'a' 和 "aaaaaaaaandy" 中所有的 'a', 但是在 "cndy" 中不会匹配任何内容。
?	匹配前一个表达式 0 次或 1 次。等价于 <code>{0,1}</code> 。例如, <code>/e?le?/</code> 匹配 "angel" 中的 'el'、"angle" 中的 'le' 以及 "oslo" 中的 'l'。如果紧跟在任何量词 *、+、? 或 { } 的后面, 将会使量词变为非贪婪 (匹配尽量少的字符), 和缺省使用的贪婪模式 (匹配尽可能多的字符) 正好相反。例如, 对 "123abc" 使用 <code>\d+/?</code> 将会匹配 "123", 而使用 <code>\d+?/?</code> 则只会匹配到 "1"。还用于先行断言中, 如本表的 <code>x(?=y)</code> 和 <code>x(?!y)</code> 条目所述。
{n}	n 是一个正整数, 匹配了前一个字符刚好出现了 n 次。比如, <code>/a{2}/</code> 不会匹配 "candy" 中的 'a', 但是会匹配 "caandy" 中所有的 a, 以及 "caaandy" 中的前两个 'a'。
{n,}	n 是一个正整数, 匹配前一个字符至少出现了 n 次。例如, <code>/a{2,}/</code> 匹配 "aa", "aaaa" 和 "aaaaa" 但是不匹配 "a"。
{n,m}	n 和 m 都是整数。匹配前面的字符至少 n 次, 最多 m 次。如果 n 或者 m 的值是 0, 这个值被忽略。例如, <code>/a{1,3}/</code> 并不匹配 "cndy" 中的任意字符, 匹配 "candy" 中的 a, 匹配 "caandy" 中的前两个 a, 也匹配 "aaaaaaaaandy" 中的前三个 a。注意, 当匹配 "aaaaaaaaandy" 时, 匹配的值是 "aaa", 即使原始的字符串中有更多的 a。

断言

字符	含义
<code>x(?=y)</code>	匹配 'x' 仅仅当 'x' 后面跟着 'y'。这种叫做先行断言。
<code>(?<=y)x</code>	匹配 'x' 仅仅当 'x' 前面是 'y'。这种叫做后行断言。
<code>x(?!y)</code>	仅仅当 'x' 后面不跟着 'y' 时匹配 'x', 这被称为正向否定查找。

(?< !y)x	仅仅当'x'前面不是'y'时匹配'x'，这被称为反向否定查找。
----------	---------------------------------

组和范围

字符	含义
(x)	像下面的例子展示的那样，它会匹配 'x' 并且记住匹配项。其中括号被称为捕获括号。模式 <code>/foo (bar) \1 \2/</code> 中的 '(foo)' 和 '(bar)' 匹配并记住字符串 "foo bar foo bar" 中前两个单词。模式中的 \1 和 \2 表示第一个和第二个被捕获括号匹配的子字符串，即 foo 和 bar，匹配了原字符串中的后两个单词。注意 \1、\2、...、\n 是用在正则表达式的匹配环节，详情可以参阅后文的 \n 条目。而在正则表达式的替换环节，则要使用像 \$1、\$2、...、\$n 这样的语法，例如， <code>'bar foo'.replace(/(...)(...)/, '\$2 \$1')</code> 。\$& 表示整个用于匹配的原字符串。
(?:x)	匹配'x'但是不记住匹配项。这种括号叫作非捕获括号，使得你能够定义与正则表达式运算符一起使用的子表达式。看看这个例子/ <code>(?:foo){1,2}</code> /。如果表达式是 <code>/foo{1,2}/</code> ，{1,2}将只应用于'foo'的最后一个字符'o'。如果使用非捕获括号，则{1,2}会应用于整个 'foo' 单词。更多信息，可以参阅下文的 Using parentheses 条目。
[xyz]	一个字符集合。匹配方括号中的任意字符，包括转义序列。你可以使用破折号 (-) 来指定一个字符范围。对于点 (.) 和星号 (*) 这样的特殊符号在一个字符集中没有特殊的含义。他们不必进行转义，不过转义也是起作用的。例如，[abcd] 和 [a-d] 是一样的。他们都匹配"brisket"中的'b'，也都匹配"city"中的'c'。 <code>/[a-z.]+/</code> 和 <code>/[\w.]+/</code> 与字符串"test.i.ng"匹配。
xyz	一个反向字符集。也就是说，它匹配任何没有包含在方括号中的字符。你可以使用破折号 (-) 来指定一个字符范围。任何普通字符在这里都是起作用的。例如， <code>\abc</code> 和 <code>\a-c</code> 是一样的。他们匹配"brisket"中的'r'，也匹配"chop"中的'h'。

|x|y |匹配'x'或者'y'。例如，/green|red/匹配“green apple”中的‘green’和“red apple”中的‘red’

git基础

Git是一个分布式的版本控制系统，与集中式的版本控制系统不同的是，每个人都工作在通过克隆建立的本地版本库中。也就是说每个人都拥有一个完整的版本库，查看提交日志、提交、创建里程碑和分支、合并分支、回退等所有操作都直接在本地完成而不需要网络连接。

对于Git仓库来说，每个人都有一个独立完整的仓库，所谓的远程仓库或是服务器仓库其实也是一个仓库，只不过这台主机24小时运行，它是一个稳定的仓库，供他人克隆、推送，也从服务器仓库中拉取别人的提交。

Git生成SSH密钥

```
git config --global user.name "xxx" // 配置用户名
git config --global user.email "xxx@qq.com" // 配置邮箱
ssh-keygen -t rsa -C "xxx@qq.com" // 生成公钥和私钥，按3次Enter，不需要设置名称与密码
cat ~/.ssh/id_rsa.pub // 查看公钥
```

gitignore

.gitignore文件的作用是忽略那些不必要的提交，比如系统环境或程序运行时产生的文件。

git checkout 切换分支

```
git checkout branchName // 切换到名为branchName的分支  
git checkout -b newBranchName // 新建名为newBranchName的分支  
git checkout fileName // 回退文件名为fileName的文件
```

git add 将工作区的修改提交到暂存区

```
git add .  
git add src/*
```

git commit 将暂存区的修改提交到当前分支

```
git commit -m "xxxx" // 提交并写提交说明  
git commit --amend // 修改最近一次提交
```

有时候如果提交注释书写有误或者漏提文件，可以使用此命令。

对于漏提交的文件，需要git add到缓存区之后，git commit --amend才能将修改追加到最近的一次提交上。

git merge 合并某分支内容到当前分支

```
git merge branchName
```

git status 查看当前仓库的状态

```
git status -s
```

A: 本地新增的文件（服务器上没有）
C: 文件的一个新拷贝
D: 本地删除的文件（服务器上还在）
M: 红色为修改过未被添加进暂存区的，绿色为已经添加进暂存区的
R: 文件名被修改
T: 文件的类型被修改
U: 文件没有被合并(你需要完成合并才能进行提交)
X: 未知状态(很可能是遇到git的bug了，你可以向git提交bug report)
?: 未被git进行管理，可以使用git add fileName把文件添加进来进行管理

git diff 查看修改

```
git diff // 查看当前修改diff  
或者  
git log
```

```
git diff commit-id commit-id // 查看2次提交的diff
```

git pull 从远程更新代码

```
git pull origin dev // 将远端dev内容，更新到本地dev分支上
```

git push 将本地代码更新到远程分支上

```
git push origin dev // dev分支推送到远端  
git push origin --delete dev // 删除远程dev分支
```

git log 查看提交历史

```
git log
```

git reset 回退到某一个版本

```
git log // 获取所有操作历史  
git reset commit-id
```

git stash 保存某次修改，例如不想提交修改，但又需要切换分支

```
git stash // 所有未提交的修改都保存起来，用于后续恢复当前工作目录  
git stash pop // 默认回退到最新保存  
或者  
git stash // 所有未提交的修改都保存起来，用于后续恢复当前工作目录  
git stash list // 查看现有所有stash， 查看结果 stash@{0}: On changehost  
git stash pop stash@{0}
```

git revert 回退某个修改

git revert, 反转提交, 撤销一个提交的同时会创建一个新的提交, 也就是用一个新提交来消除一个历史提交所做的任何修改.

```
git revert commit-id // revert指定的一个commit  
git revert HEAD~3 // revert指定倒数第四个commit
```

revert过程有可能遇到冲突, 要么git revert --abort终止此次revert操作, 代码还原至revert命令前. 要么手动消除冲突(同普通的冲突解决), 然后add commit

git branch 分支操作

```
git branch // 查看本地分支  
git branch -a // 查看远端分支  
git branch -D branchName // 删除名为branchName的分支（前提是先要切换到其他分支）
```

git remote 查看关联的远程仓库的名称

代码优化

1、

```
// 优化前  
if (a && typeof a === 'string') {}  
// 优化后  
if (typeof a === 'string') {}
```

2、

```
// 优化前  
let arr = [1, 2, 3, 4].map(item => {  
    return item * 2;  
});  
// 优化后  
let arr = [1, 2, 3, 4].map(item => item * 2);
```

3、

```
// 优化前  
let a = true; // a为布尔  
if (a) {  
    return true;  
} else {  
    return false;  
}  
// 优化后  
return a;
```

4、

```
// 优化前  
var a = {  
    name: 'xiaowa',  
    age: 20  
};
```

```
var b = {  
    name: a.name,  
    age: a.age,  
    like: red  
};  
  
// 优化后  
var a = {  
    name: 'xiaowa',  
    age: 20  
};  
var b = {  
    ...a,  
    like: red  
}
```

5、

```
// 优化前  
let a = '';  
if (true) {  
    a = '1'  
} else {  
    a = '0'  
}  
// 优化后  
let a = '0';  
if (true) {  
    a = '1';  
}
```

常用查错方法

- 1、加 try-catch
- 2、加 setTimeout
- 3、用typeof查看类型
- 4、挂window上

小程序其他知识

支付宝小程序与百度小程序

百度小程序

- page由4个文件组成：js、swan、css、json
- api前缀为swan
- 提供很多ai相关能力

支付宝小程序

- page由4个文件组成：js、axml、acss、json
- api前缀为my
- 提供很多支付、资金、会员、营销等能力

taro

京东凹凸实验室开源的一款使用 React.js 开发微信小程序的前端框架。Taro 是一套遵循 React 语法规范的 多端开发 解决方案。

现如今市面上端的形态多种多样，Web、React-Native、微信小程序等各种端大行其道，当业务要求同时在不同的端都要求有所表现的时候，针对不同的端去编写多套代码的成本显然非常高，这时候只编写一套代码就能够适配到多端的能力就显得极为需要。

快速开发微信小程序

Taro 立足于微信小程序开发，众所周知小程序的开发体验并不是非常友好，比如小程序中无法使用 npm 来进行第三方库的管理，无法使用一些比较新的 ES 规范等等，针对小程序端的开发弊端，Taro 具有以下的优秀特性

- 支持使用 npm/yarn 安装管理第三方依赖
- 支持使用 ES7/ES8 甚至更新的 ES 规范，一切都可自行配置
- 支持使用 CSS 预编译器，例如 Sass 等
- 支持使用 Redux 进行状态管理
- 支持使用 MobX 进行状态管理
- 小程序 API 优化，异步 API Promise 化等等

支持多端开发转化

Taro 方案的初心就是为了打造一个多端开发的解决方案。目前 Taro 代码可以支持转换到 微信/百度/支付宝/字节跳动/QQ小程序、快应用、H5 端以及移动端（React Native）。

安装

```
npm install -g @tarojs/cli
```

项目目录结构

```

├── dist          编译结果目录
├── config        配置目录
│   ├── dev.js    开发时配置
│   ├── index.js  默认配置
│   └── prod.js   打包时配置
├── src           源码目录
│   ├── pages      页面文件目录
│   │   ├── index  index 页面目录
│   │   │   ├── index.js  index 页面逻辑
│   │   │   └── index.css  index 页面样式
│   ├── app.css    项目总通用样式
│   └── app.js     项目入口文件
└── package.json

```

举个栗子

```
// 入口文件默认是 src 目录下的 app.js。
import Taro, { Component } from '@tarojs/taro'
import Index from './pages/index'

import './app.scss'

class App extends Component {
  // 项目配置
  config = {
    pages: [
      'pages/index/index'
    ],
    window: {
      backgroundTextStyle: 'light',
      navigationBarBackgroundColor: '#fff',
      navigationBarTitleText: 'WeChat',
      navigationBarTextStyle: 'black'
    }
  }

  componentWillMount () {}

  componentDidMount () {}

  componentDidShow () {}
}
```

```
componentDidHide () {}

render () {
  return (
    <Index />
  )
}

// page页的配置

export default class Index extends Component {
  config = {
    navigationBarBackgroundColor: '#ffffff',
    navigationBarTextStyle: 'black',
    navigationBarTitleText: '首页',
    backgroundColor: '#eeeeee',
    backgroundTextStyle: 'light'
  }

  render () {
    return (
      <View className='index'>
        <Text>1</Text>
      </View>
    )
  }
}
```

项目配置

各类小程序平台均有自己的项目配置文件，例如

- 微信小程序，`project.config.json`
- 百度智能小程序，`project.swan.json`
- 头条小程序，`project.config.json`，文档暂无，大部分字段与微信小程序一致
- 支付宝小程序，暂无发现
- 快应用，`manifest.json`
- QQ 小程序，暂无发现

为了能够适配到各个小程序平台，满足不同小程序平台配置文件不同的情况，在 Taro 支持为各个小程序平台添加不同的项目配置文件。

通过 Taro 模板创建的项目都会默认拥有 `project.config.json` 这一项目配置文件，这个文件只能用于微信小程序，若要兼容到其他小程序平台，请按如下对应规则来增加相应平台的配置文件，其配置与各小程序平台要求的一致

生命周期函数

生命周期方法	说明
Page.onLoad	componentWillMount
onShow	componentDidShow
onHide	componentDidHide
onReady	componentDidMount
onUnload	componentWillUnmount
onError	componentDidCatchError
App.onLaunch	componentWillMount
Component.created	componentWillMount
attached	componentDidMount
ready	componentDidMount
detached	componentWillUnmount
moved	保留

组件的使用

```

import Taro, { Component } from '@tarojs/taro'
// 引入 map 组件
import { Map } from '@tarojs/components'

class App extends Component {
  onTap () {}
  render () {
    return (
      <Map onClick={this.onTap} />
    )
  }
}

```

api的使用

```

import Taro from '@tarojs/taro'

Taro.request({
  url: 'http://localhost:8080/test',
  data: {
    foo: 'foo',
    bar: 10
  },
}

```

```

    header: {
      'content-type': 'application/json'
    }
  })
  .then(res => console.log(res.data))

```

mpvue

美团团队开源的一款使用 Vue.js 开发微信小程序的前端框架。使用此框架，开发者将得到完整的 Vue.js 开发体验，同时为 H5 和小程序提供了代码复用的能力。使用 mpvue 开发小程序，你将在小程序技术体系的基础上获取到这样一些能力：

- 彻底的组件化开发能力：提高代码
- 完整的 Vue.js 开发体验
- 方便的 Vuex 数据管理方案：方便构建复杂应用
- 快捷的 webpack 构建机制：自定义构建策略、开发阶段 hotReload
- 支持使用 npm 外部依赖
- 使用 Vue.js 命令行工具 vue-cli 快速初始化项目
- H5 代码转换编译成小程序目标代码的能力

它的目标是：在未来最理想的状态下，可以一套代码可以直接跑在多端：WEB、微信小程序、支付宝小程序、Native（借助weex）。不过由于各个端之间都存在一些比较明显的差异性，从产品的层面上讲，不建议这么做，这个框架的官方他们对它的期望的也只是开发和调试体验的一致。

框架原理

mpvue 保留了 vue.runtime 核心方法，无缝继承了 Vue.js 的基础能力 mpvue-template-compiler 提供了将 vue 的模板语法转换到小程序的 wxml 语法的能力 修改了 vue 的建构配置，使之构建出符合小程序项目结构的代码格式： json/wxml/wxss/js 文件

快速开始

1. 初始化一个 mpvue 项目

现代前端开发框架和环境都是需要 Node.js 的，如果没有的话，请先下载 nodejs 并安装。

然后打开命令行工具：

```

# 1. 先检查下 Node.js 是否安装成功
$ node -v
v8.9.0

$ npm -v
5.6.0

# 2. 由于众所周知的原因，可以考虑切换源为 taobao 源
$ npm set registry https://registry.npm.taobao.org/

```

```
# 3. 全局安装 vue-cli  
# 一般是要 sudo 权限的  
$ npm install --global vue-cli@2.9  
  
# 4. 创建一个基于 mpvue-quickstart 模板的新项目  
# 新手一路回车选择默认就可以了  
$ vue init mpvue/mpvue-quickstart my-project  
  
# 5. 安装依赖, 走你  
$ cd my-project  
$ npm install  
$ npm run dev  
随着运行成功的回显之后, 可以看到本地多了个 dist 目录, 这个目录里就是生成的小程序相关代码。
```

2. 搭建小程序的开发环境

小程序自己有一个专门的微信开发者工具, 然后用微信扫描二维码登陆。至此小程序的开发环境差不多完成。

3. 调试开发 mpvue

选择 小程序项目 并依次填好需要的信息:

- 项目目录: 就是刚刚创建的项目目录 (非 dist 目录)
- AppID: 没有的话可以点选体验“小程序”, 只影响是否可以真机调试。
- 项目名称。

点击“确定”按钮后会跳到正式的开发页面, 点击“编辑器”按钮, 关闭自带的小程序编辑器。此时, 整个 mpvue 项目已经跑起来了。用自己趁手的编辑器 (或者IDE) 打开 my-project 中的 src 目录下的代码试试, 到此, 上手完毕。

react高级

context

在一个典型的 React 应用中，数据是通过 props 属性自上而下（由父及子）进行传递的，但这种做法对于某些类型的属性而言是极其繁琐的（例如：地区偏好，UI 主题），这些属性是应用程序中许多组件都需要的。Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

React.createContext

```
const MyContext = React.createContext(defaultValue);
```

创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 Provider 中读取到当前的 context 值。

只有当组件所处的树中没有匹配到 Provider 时，其 defaultValue 参数才会生效。这有助于在不使用 Provider 包装组件的情况下对组件进行测试。注意：将 undefined 传递给 Provider 时，消费组件的 defaultValue 不会生效。

Context.Provider

```
<MyContext.Provider value={/* 某个值 */}>
```

每个 Context 对象都会返回一个 Provider React 组件，它允许消费组件订阅 context 的变化。

Provider 接收一个 value 属性，传递给消费组件。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 value 值发生变化时，它内部的所有消费组件都会重新渲染。Provider 及其内部 consumer 组件都不受制于 shouldComponentUpdate 函数，因此当 consumer 组件在其祖先组件退出更新的情况下也能更新。

Class.contextType

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* 在组件挂载完成后，使用 MyContext 组件的值来执行一些有副作用的操作 */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
}
```

```

    }
    componentWillUnmount() {
      let value = this.context;
      /* ... */
    }
    render() {
      let value = this.context;
      /* 基于 MyContext 组件的值进行渲染 */
    }
  }
 MyClass.contextType = MyContext;

```

挂载在 class 上的 contextType 属性会被重赋值为一个由 React.createContext() 创建的 Context 对象。这能让你使用 this.context 来消费最近 Context 上的那个值。你可以在任何生命周期中访问到它，包括 render 函数中。

Context.Consumer

```

<MyContext.Consumer>
  {value => /* 基于 context 值进行渲染*/}
</MyContext.Consumer>

```

这里，React 组件也可以订阅到 context 变更。这能让你在函数式组件中完成订阅 context。

这需要函数作为子元素（function as a child）这种做法。这个函数接收当前的 context 值，返回一个 React 节点。传递给函数的 value 值等同于往上组件树离这个 context 最近的 Provider 提供的 value 值。如果没有对应的 Provider，value 参数等同于传递给 createContext() 的 defaultValue。

高阶组件

高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

具体而言，高阶组件是参数为组件，返回值为新组件的函数。

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

组件是将 props 转换为 UI，而高阶组件是将组件转换为另一个组件。

使用 HOC 解决横切关注点问题

假设有一个 CommentList 组件，它订阅外部数据源，用以渲染评论列表：

```

class CommentList extends React.Component {
  constructor(props) {
    super(props);
  }
}

```

```

this.handleChange = this.handleChange.bind(this);
this.state = {
  // 假设 "DataSource" 是个全局范围内的数据源变量
  comments: DataSource.getComments()
};
}

componentDidMount() {
  // 订阅更改
  DataSource.addChangeListener(this.handleChange);
}

componentWillUnmount() {
  // 清除订阅
  DataSource.removeChangeListener(this.handleChange);
}

handleChange() {
  // 当数据源更新时，更新组件状态
  this.setState({
    comments: DataSource.getComments()
  });
}

render() {
  return (
    <div>
      {this.state.comments.map((comment) => (
        <Comment comment={comment} key={comment.id} />
      ))}
    </div>
  );
}
}

```

稍后，编写了一个用于订阅单个博客帖子的组件，该帖子遵循类似的模式：

```

class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }
}

```

```

componentWillUnmount() {
  DataSource.removeChangeListener(this.handleChange);
}

handleChange() {
  this.setState({
    blogPost: DataSource.getBlogPost(this.props.id)
  });
}

render() {
  return <TextBlock text={this.state.blogPost} />;
}
}

```

CommentList 和 BlogPost 不同 - 它们在 DataSource 上调用不同的方法，且渲染不同的结果。但它们的大部分实现都是一样的：

- 在挂载时，向 DataSource 添加一个更改侦听器。
- 在侦听器内部，当数据源发生变化时，调用 setState。
- 在卸载时，删除侦听器。

你可以想象，在一个大型应用程序中，这种订阅 DataSource 和调用 setState 的模式将一次又一次地发生。我们需要一个抽象，允许我们在一个地方定义这个逻辑，并在许多组件之间共享它。这正是高阶组件擅长的地方。

我们可以编写一个创建组件的函数，比如 CommentList 和 BlogPost，订阅 DataSource。该函数将接受一个子组件作为它的其中一个参数，该子组件将订阅数据作为 prop。让我们调用函数 withSubscription：

```

const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);

```

第一个参数是被包装组件。第二个参数通过 DataSource 和当前的 props 返回我们需要的数据。

当渲染 CommentListWithSubscription 和 BlogPostWithSubscription 时， CommentList 和 BlogPost 将传递一个 data prop，其中包含从 DataSource 检索到的最新数据：

```

// 此函数接收一个组件...
function withSubscription(WrappedComponent, selectData) {
  // ...并返回另一个组件...
  return class extends React.Component {

```

```

constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
  this.state = {
    data: selectData(dataSource, props)
  };
}

componentDidMount() {
  // ...负责订阅相关的操作...
  dataSource.addChangeListener(this.handleChange);
}

componentWillUnmount() {
  dataSource.removeChangeListener(this.handleChange);
}

handleChange() {
  this.setState({
    data: selectData(dataSource, this.props)
  });
}

render() {
  // ... 并使用新数据渲染被包装的组件!
  // 请注意，我们可能还会传递其他属性
  return <WrappedComponent data={this.state.data} {...this.props} />;
}
};
}

```

请注意，HOC 不会修改传入的组件，也不会使用继承来复制其行为。相反，HOC 通过将组件包装在容器组件中来组成新组件。HOC 是纯函数，没有副作用。

被包装组件接收来自容器组件的所有 prop，同时也接收一个新的用于 render 的 data prop。HOC 不需要关心数据的使用方式或原因，而被包装组件也不需要关心数据是怎么来的。

因为 withSubscription 是一个普通函数，你可以根据需要对参数进行增添或者删除。例如，您可能希望使 data prop 的名称可配置，以进一步将 HOC 与包装组件隔离开来。或者你可以接受一个配置 shouldComponentUpdate 的参数，或者一个配置数据源的参数。因为 HOC 可以控制组件的定义方式，这一切都变得有可能。

与组件一样，withSubscription 和包装组件之间的契约完全基于之间传递的 props。这种依赖方式使得替换 HOC 变得容易，只要它们为包装的组件提供相同的 prop 即可。例如你需要改用其他库来获取数据的时候，这一点就很有用。

Refs 转发

Ref 转发是一个可选特性，其允许某些组件接收 ref，并将其向下传递（换句话说，“转发”它）给子组件。

转发 refs 到 DOM 组件

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// 你可以直接获取 DOM button 的 ref:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

这样，使用 FancyButton 的组件可以获取底层 DOM 节点 button 的 ref，并在必要时访问，就像其直接使用 DOM button 一样。

以下是对上述示例发生情况的逐步解释：

- 我们通过调用 React.createRef 创建了一个 React ref 并将其赋值给 ref 变量。
- 我们通过指定 ref 为 JSX 属性，将其向下传递给。
- React 传递 ref 给 forwardRef 内函数 (props, ref) => ...，作为其第二个参数。
- 我们向下转发该 ref 参数到 <button ref={ref}>，将其指定为 JSX 属性。
- 当 ref 挂载完成，ref.current 将指向 <button> DOM 节点。

高阶组件中转发 refs

这个技巧对高阶组件（也被称为 HOC）特别有用。让我们从一个输出组件 props 到控制台的 HOC 示例开始：

```
function logProps(WrappedComponent) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  }

  return LogProps;
}
```

“logProps” HOC 透传 (pass through) 所有 props 到其包裹的组件，所以渲染结果将是相同的。例如：我们可以使用该 HOC 记录所有传递到 “fancy button” 组件的 props：

```
class FancyButton extends React.Component {
  focus() {
    // ...
  }

  // ...
}

// 我们导出 LogProps，而不是 FancyButton。
// 虽然它也会渲染一个 FancyButton。
export default logProps(FancyButton);
```

上面的示例有一点需要注意：refs 将不会透传下去。这是因为 ref 不是 prop 属性。就像 key 一样，其被 React 进行了特殊处理。如果你对 HOC 添加 ref，该 ref 将引用最外层的容器组件，而不是被包裹的组件。

这意味着用于我们 FancyButton 组件的 refs 实际上将被挂载到 LogProps 组件：

```
import FancyButton from './FancyButton';

const ref = React.createRef();

// 我们导入的 FancyButton 组件是高阶组件 (HOC) LogProps。
// 尽管渲染结果将是一样的，
// 但我们的 ref 将指向 LogProps 而不是内部的 FancyButton 组件！
// 这意味着我们不能调用例如 ref.current.focus() 这样的方法
<FancyButton
  label="Click Me"
  handleClick={handleClick}
  ref={ref}
/>;
```

幸运的是，我们可以使用 React.forwardRef API 明确地将 refs 转发到内部的 FancyButton 组件。React.forwardRef 接受一个渲染函数，其接收 props 和 ref 参数并返回一个 React 节点。例如：

```
function logProps(Component) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }
  }

  render() {
    const {forwardedRef, ...rest} = this.props;
```

```

    // 将自定义的 prop 属性 “forwardedRef” 定义为 ref
    return <Component ref={forwardedRef} {...rest} />;
}

}

// 注意 React.forwardRef 回调的第二个参数 “ref”。
// 我们可以将其作为常规 prop 属性传递给 LogProps，例如 “forwardedRef”
// 然后它就可以被挂载到被 LogProps 包裹的子组件上。
return React.forwardRef((props, ref) => {
    return <LogProps {...props} forwardedRef={ref} />;
});
}

```

Portals

Portal 提供了一种将子节点渲染到存在于父组件以外的 DOM 节点的优秀的方案

通常来讲，当你从组件的 render 方法返回一个元素时，该元素将被挂载到 DOM 节点中离其最近的父节点：

```

render() {
    // React 挂载了一个新的 div，并且把子元素渲染其中
    return (
        <div>
            {this.props.children}
        </div>
    );
}

```

然而，有时候将子元素插入到 DOM 节点中的不同位置也是有好处的：

```

render() {
    // React 并*没有*创建一个新的 div。它只是把子元素渲染到 `domNode` 中。
    // `domNode` 是一个可以在任何位置的有效 DOM 节点。
    return ReactDOM.createPortal(
        this.props.children,
        domNode
    );
}

```

一个 portal 的典型用例是当父组件有 overflow: hidden 或 z-index 样式时，但你需要子组件能够在视觉上“跳出”其容器。例如，对话框、悬浮卡以及提示框：

PropTypes

随着你的应用程序不断增长，你可以通过类型检查捕获大量错误，React 也内置了一些类型检查的功能。要在组件的 props 上进行类型检查，你只需配置特定的 propTypes 属性：

```

import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};

```

Hook

Hook 是 React 16.8 的新增特性。它可以在不编写 class 的情况下使用 state 以及其他 React 特性。

Hook 解决了我们五年来编写和维护成千上万的组件时遇到的各种各样看起来不相关的问题。无论你正在学习 React，或每天使用，或者更愿尝试另一个和 React 有相似组件模型的框架，你都可能对这些问题似曾相识。

- 在组件之间复用状态逻辑很难
- 复杂组件变得难以理解
- 难以理解的 class

使用 State Hook

state 初始值为 { count: 0 }，当用户点击按钮后，我们通过调用 this.setState() 来增加 state.count。

```

import React, { useState } from 'react';

function Example() {
  // 声明一个叫 "count" 的 state 变量
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

// 等价的 class 示例

```

```

class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}

```

使用 Effect Hook

如果你熟悉 React class 的生命周期函数，你可以把 useEffect Hook 看做 componentDidMount, componentDidUpdate 和 componentWillUnmount 这三个函数的组合。

```

import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

```

在 React 组件中有两种常见副作用操作：需要清除的和不需要清除的。我们来更仔细地看一下他们之间的区别。

无需清除的effect

有时候，我们只想在 React 更新 DOM 之后运行一些额外的代码。比如发送网络请求，手动变更 DOM，记录日志，这些都是常见的无需清除的操作。因为我们在执行完这些操作之后，就可以忽略它们了。

```
// 在这个 class 中，我们需要在两个生命周期函数中编写重复的代码。
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}

import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```

        </button>
    </div>
);
}

```

useEffect 做了什么？通过使用这个 Hook，你可以告诉 React 组件需要在渲染后执行某些操作。React 会保存你传递的函数（我们将它称之为“effect”），并在执行 DOM 更新之后调用它。在这个 effect 中，我们设置了 document 的 title 属性，不过我们也可以执行数据获取或调用其他命令式的 API。

为什么在组件内部调用 useEffect？将 useEffect 放在组件内部让我们可以在 effect 中直接访问 count state 变量（或其他 props）。我们不需要特殊的 API 来读取它——它已经保存在函数作用域中。Hook 使用了 JavaScript 的闭包机制，而不用在 JavaScript 已经提供了解决方案的情况下，还引入特定的 React API。

useEffect 会在每次渲染后都执行吗？是的，默认情况下，它在第一次渲染之后和每次更新之后都会执行。（我们稍后会谈到如何控制它。）你可能会更容易接受 effect 发生在“渲染之后”这种概念，不用再去考虑“挂载”还是“更新”。React 保证了每次运行 effect 的同时，DOM 都已经更新完毕。

需要清除的 effect

在 React class 中，你通常会在 componentDidMount 中设置订阅，并在 componentWillUnmount 中清除它。就需要清除 effect。你会注意到 componentDidMount 和 componentWillUnmount 之间相互对应。使用生命周期函数迫使我们拆分这些逻辑代码，即使这两部分代码都作用于相同的副作用。

```

// class 的方式
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
}

handleStatusChange(status) {

```

```

    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}

// useEffect的方式
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

```

为什么要在 effect 中返回一个函数？这是 effect 可选的清除机制。每个 effect 都可以返回一个清除函数。如此可以将添加和移除订阅的逻辑放在一起。它们都属于 effect 的一部分。

React 何时清除 effect？React 会在组件卸载的时候执行清除操作。正如之前学到的，effect 在每次渲染的时候都会执行。这就是为什么 React 会在执行当前 effect 之前对上一个 effect 进行清除。

Hook 规则

只在最顶层使用 Hook

不要在循环，条件或嵌套函数中调用 Hook， 确保总是在你的 React 函数的最顶层调用他们。遵守这条规则，你就能确保 Hook 在每一次渲染中都按照同样的顺序被调用。这让 React 能够在多次的 useState 和 useEffect 调用之间保持 hook 状态的正确。

只在 React 函数中调用 Hook

不要在普通的 JavaScript 函数中调用 Hook。你可以：

- 在 React 的函数组件中调用 Hook
- 在自定义 Hook 中调用其他 Hook

Redux

Redux的设计理念很简单： Web 应用是一个状态机，视图与状态是一一对应的，应用的状态保存在一个对象中。

理解redux 之前，我们要知道几个概念， store, reducer, action.

store 掌管整个应用的状态, 整个应用只能有一个store。通过store.getState() 获取应用某个时间点的快照（状态），通过store.dispatch 分发action Redux 规定：一个 State 对应一个 View。只要 State 相同，View 就相同。你知道 State，就知道 View 是什么样，反之亦然。reducer 当store收到action，就要生成一个新的state, 这个计算过程就是reducer。reducer是一个纯函数，即相同的输入，一定得到相同的输出。action 用户不能直接修改state, state的变化必须是View 导致的，action 就是View 发出的一个通知，表示State要发生变化啦。action 是一个对象，type属性是必须的,表示action 的名称。

redux重要API

redux提供了几个重要的函数：

createStore函数

用来生成store

```
const store = createStore(reducer, initialState, enhancer);
```

bindActionCreators

将 action 包装成直接可被调用的函数，用户感知不到dispatch的存在

combineReducers

一个复杂的应用往往state 比较庞大，导致 Reducer 函数也比较庞大，因此如果能把reducer拆分成一个个独立的子Reducer, 最后再把他们合成一个大的reducer，处理起来就比较方便。而 combineReducers就是做这件事的，该函数根据state的key去执行响应的子Reducer，并将结果合并到最终的state对象里。

applyMiddleware

applyMiddleware(...middlewares) 引入中间件，比如我们经常使用的用于处理异步action的redux-thunk 中间件。实际上，中间件是一个函数，对store.dispatch函数进行了改造，在发出action和执行reducer之间，增加了一些其他的功能。

compose函数

compose是一个返回依次执行参数里面的方法的函数， 其内部是通过Array.prototype.reduceRight 函数实现的， 一般redux项目使用多个中间件时会用到。

react-redux

react-redux 提供了两个重要的函数， Provider 和 connect。

Provider组件

Provider其实是一个React 组件， 其原理是通过React组件的context 属性实现store 的传递， 进而拿到整个应用的state。

```
// Provider 实现原理
class Provider extends Component {
  getChildContext() {
    store: this.props.store
  }
  render() {
    const { children } = this.props
    return Children.only(children)
  }
}

Provider.childContextTypes = {
  store: React.PropTypes.object
}
```

上面store放在context里， 这样页面组件就可以通过context 拿到store

```
class PageComp extends Component {

  render() {
    const { store } = this.context;
    const state = store.getState();

    return (<div>{state.number || ''}</div>)
  }
}

PageComp.contextTypes = {
  store: React.PropTypes.object
}
```

connect 函数

connect函数是把 redux 的 dispatch 和 state 映射为 react 组件的 props中， 将页面里的组件与应用的状态state真正连接起来。

React router

与v3版比较

react router的重写导致和v3有很多不同的地方。主要有：

- 在react router v4里，路由不再是集中在一起的。它成了应用布局、UI的一部分。
- 浏览器用的router在react-router-dom里。所以，浏览器里使用的时候只需要import react-router-dom就可以。
- 新的概念BrowserRouter和HashRouter。他们各自服务于不同的情景下
- 不在使用{props.children}来处理嵌套的路由。
- v4的路由默认不再排他，会有多个匹配。而v3是默认排他的，只会有一个匹配被使用。

Route匹配

组件

有两个路由匹配组件：`<Route>` 和 `<Switch>`。

- 路由匹配是通过比较 `<Route>` 的 path 属性和当前地址的 pathname 来实现的。当一个匹配成功时，它将渲染其内容，当它不匹配时就会渲染 null。没有路径的 `<Route>` 将始终被匹配。你可以在任何你希望根据地址渲染内容的地方添加 `<Route>`。
- `<Switch>` 不是分组 `<Route>` 所必须的，但他通常很有用。一个 `<Switch>` 会遍历其所有的子 `<Route>` 元素，并仅渲染与当前地址匹配的第一个元素。

```
<Switch>
  <Route exact path="/" component={Home} />
  <Route path="/about" component={About} />
  <Route path="/contact" component={Contact} />
  {/* when none of the above match, <NoMatch> will be rendered */}
  <Route component={NoMatch} />
</Switch>
```

路由渲染属性

你有三个属性来给 `<Route>` 渲染组件: component , render, 和 children 。你可以查看 `<Route>` 文档来了解它们的更多信息，但在这我们将重点关注component 和 render 因为这几乎是你会用到的两个。component 应该在你想渲染现存组件时使用（React.Component 或一个无状态组件）。

render, 只有在必须将范围内的变量传递给要渲染的组件时才能使用。你不应该使用具有内联函数的 component 属性来传递范围内的变量，因为你将要不必要的卸载/重载组件。

```
const App = () => {
  const someVariable = true;
```

```

return (
  <Switch>
    {/* these are good */}
    <Route exact path="/" component={Home} />
    <Route
      path="/about"
      render={props => <About {...props} extra={someVariable} />}
    />
    {/* do not do this */}
    <Route
      path="/contact"
      component={props => <Contact {...props} extra={someVariable} />}
    />
  </Switch>
);
};

```

导航

React Router 提供了一个 `<Link>` 组件来在你的应用程序中创建链接。无论你在何处渲染一个 `<Link>`，都会在应用程序的 HTML 中渲染锚（`<a>`）。当你想强制导航时，你可以渲染一个 `<Redirect>`。当一个 `<Redirect>` 渲染时，它将使用它的 `to` 属性进行定向。

嵌套路由

`Route` 只是一个组件，就像 `div` 一样。因此，为了嵌入 `Route` 或 `div`，你只需...做到这一点。

```

const App = () => (
  <BrowserRouter>
    {/* here's a div */}
    <div>
      {/* here's a Route */}
      <Route path="/tacos" component={Tacos} />
    </div>
  </BrowserRouter>
);

// when the url matches `/tacos` this component renders
const Tacos = ({ match }) => (
  // here's a nested div
  <div>
    {/* here's a nested Route,
        match.url helps us make a relative path */}
    <Route path={match.url + "/carnitas"} component={Carnitas} />
  </div>
);

```

api

history

history 对象通常会具有以下属性和方法：

- length - (number 类型) history 堆栈的条目数
- action - (string 类型) 当前的操作(PUSH, REPLACE, POP)
- location - (object 类型) 当前的位置。location 会具有以下属性：
 - pathname - (string 类型) URL 路径
 - search - (string 类型) URL 中的查询字符串
 - hash - (string 类型) URL 的哈希片段
 - state - (object 类型) 提供给例如使用 push(path, state) 操作将 location 放入堆栈时的特定 location 状态。只在浏览器和内存历史中可用。
- push(path, [state]) - (function 类型) 在 history 堆栈添加一个新条目
- replace(path, [state]) - (function 类型) 替换在 history 堆栈中的当前条目
- go(n) - (function 类型) 将 history 堆栈中的指针调整 n
- goBack() - (function 类型) 等同于 go(-1)
- goForward() - (function 类型) 等同于 go(1)
- block(prompt) - (function 类型) 阻止跳转。(详见 history 文档)。

location

router 将在这几个地方为您提供一个 location 对象：

- Route component as this.props.location
- Route render as {({ location }) => ()}
- Route children as {({ location }) => ()}
- withRouter as this.props.location

它也可以在 history.location 找到，但是你不应该使用它，因为它是可变的，你可以在 history 文档中阅读更多内容。location 对象永远不会发生变化，因此你可以在生命周期钩子中使用它来确定何时导航，这对数据抓取和动画非常有用。

服务器渲染

在服务器上渲染有点儿不同，因为他们都是无状态的。基本思想是我们将 app 包装在一个无状态的 `<StaticRouter>` 中而不是 `<BrowserRouter>` 中。我们通过服务器传入请求的 URL，以便路由可以匹配，然后我们将在下面讨论 `<context>` 属性。

```
// 服务器端 (不是完整的代码)
import { createServer } from 'http'
import React from 'react'
import ReactDOMServer from 'react-dom/server'
import { StaticRouter } from 'react-router'
import App from './App'
```

```
createServer((req, res) => {
  const context = {}

  const html = ReactDOMServer.renderToString(
    <StaticRouter
      location={req.url}
      context={context}
    >
    <App/>
  </StaticRouter>
  )

  if (context.url) {
    res.writeHead(301, {
      Location: context.url
    })
    res.end()
  } else {
    res.write(`
      <!doctype html>
      <div id="app">${html}</div>
    `)
    res.end()
  }
}).listen(3000)

// 客户端
import ReactDOM from 'react-dom'
import { BrowserRouter } from 'react-router-dom'
import App from './App'

ReactDOM.render((
  <BrowserRouter>
    <App/>
  </BrowserRouter>
), document.getElementById('app'))
```

node

- node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境,让JavaScript的执行效率与低端的C语言的相近。
- node.js 使用了一个事件驱动、非阻塞式 I/O 的模型, 使其轻量又高效。
- node.js 的包管理器 npm, 是全球最大的开源库生态系统。

node特点

nodejs与浏览器中js的区别： 运行环境不同

- dom 与 file system、http server等
- window 与 global
- web platform api 与 nodejs api

事件驱动编程

事件驱动编程 (Event-driven programming) 是一种编程风格, 由事件来决定程序的执行流程, 事件由事件处理器 (event handler) 或事件回调 (event callback) 来处理, 事件回调是当某个特定事件发生时被调用的函数。

这种编程模型就叫事件驱动编程或异步编程。这是Node一个最明显的特性, 这种编程模型意味着当前进程在执行I/O操作时不会被阻塞, 因此, 多个I/O操作可以并行执行, 当操作完成后相应的回调函数就会被调用。

事件驱动编程底层依赖于事件循环 (event loop), 事件循环基本上是事件检测和事件处理器触发这两种函数不断循环调用的一个结构。在每次循环里, 事件循环机制需要检测发生了哪些事件, 当事件发生时, 它找到对应的回调函数并调用它。

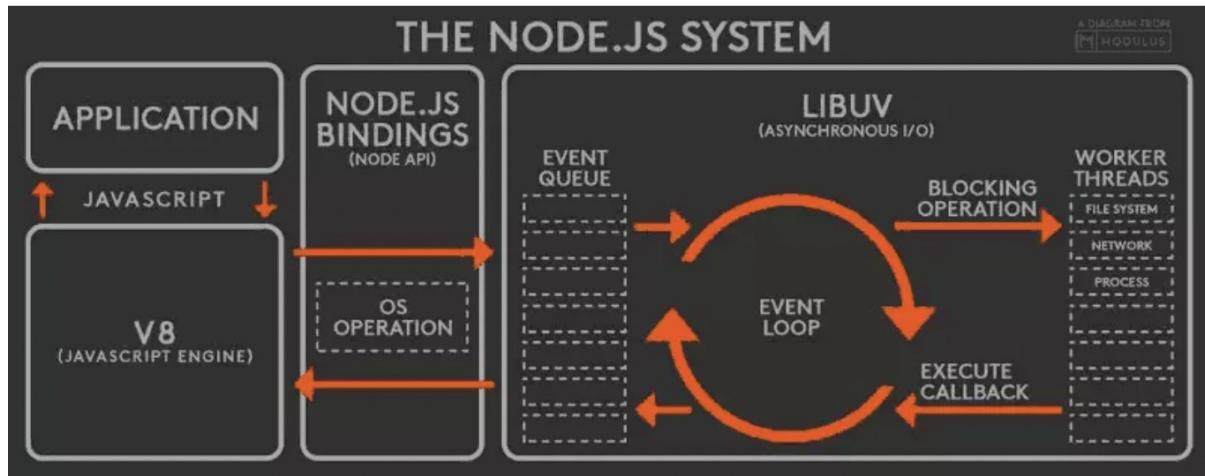
事件循环只是运行在进程内的一个线程, 当事件发生时, 事件处理器可以单独运行并且不会被中断, 也就是说:

- 1.在某个特定时刻最多有一个事件回调函数运行
- 2.任何事件处理器运行时都不会被中断

有了这个, 开发人员就可以不再为线程同步和并发修改共享内存这些事头疼了。

Node.js的Event Loop

- V8引擎解析JavaScript脚本。
- 解析后的代码, 调用Node API。
- libuv库负责Node API的执行。它将不同的任务分配给不同的线程, 形成一个Event Loop (事件循环), 以异步的方式将任务的执行结果返回给V8引擎。
- V8引擎再将结果返回给用户



阻塞与非阻塞

- 阻塞：就像单线程cpu一样，一个任务由多个小任务组成，但是只能一个任务接一个任务流程的往下走，谁在任务排序的前面就谁先执行，执行完了进行下一个，如果遇到错误，下面的小任务就不要做了，一直卡住。
- 非阻塞：就像多线程cpu一样，一个任务由多个小任务组成，可以分开线程来做，哪个线程做分配到的任务，完成了对应的任务就行，某个线程的任务没做完那就做报对应的错，其他的不受影响。

举个栗子

```
// 阻塞代码实例
var fs = require("fs");
var data = fs.readFileSync('input.txt');
console.log(data.toString());
console.log("程序执行结束!");

// 非阻塞代码实例
var fs = require("fs");
fs.readFile('input.txt', function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});
console.log("程序执行结束!');
```

node安装

使用安装包安装

在官网主页上可下载到Windows及Mac版本的安装包，安装完成后将自带npm工具（Node Package Manager，Node包管理器）

使用nvm安装

nvm是一个开源的node.js多版本管理Bash工具。

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh | bash
or
wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh | bash
```

nvm安装的最新版可能并不是Node官方推荐的稳定版本，建议参考官方推荐版本后进行版本指定安装：

```
nvm install 8.11.3
```

npm使用介绍

新版的nodejs已经集成了npm。"npm -v" 来测试是否成功安装

- 允许用户从npm服务器下载别人编写的第三方包到本地使用。
- 允许用户从npm服务器下载并安装别人编写的命令行程序到本地使用。
- 允许用户将自己编写的包或命令行程序上传到npm服务器供别人使用。

使用 npm 命令安装模块

```
npm install <Module Name>
```

举个栗子

```
//常用的 Node.js web框架模块 express:
npm install express

//express 包就放在了工程目录下的 node_modules 目录中，因此在代码中只需要通过 require('express')
的方式就好，无需指定第三方包路径。
var express = require('express');
```

全局&局部安装

```
npm install express      # 本地安装
npm install express -g  # 全局安装
```

- 全局安装：将安装包放在 /usr/local 下或者你 node 的安装目录。可以直接在命令行里使用。
- 局部安装：将安装包放在 ./node_modules 下（运行 npm 命令时所在的目录），如果没有 node_modules 目录，会在当前执行 npm 命令的目录下生成 node_modules 目录。
- 局部安装：可以通过 require() 来引入本地安装的包。

其他命令

```
npm uninstall express // 卸载模块
npm update express // 更新模块
npm search express // 搜索模块
```

node内部工作原理解析

在 github 中，我们可以看到 node 库的目录，其中：

- deps：包含了node所依赖的库；
- lib：包含了我们在项目中引入的用 javascript 定义的函数和模块；
- src：lib 库对应的C++实现。

在 Node 官方文档的 Dependencies 中，我们也可以看到一些具体的所依赖的库的作用。

- v8 引擎的作用就是将 js 转成 C++。
- libuv 用于在C++中处理并发和进程构建，具有跨平台和异步能力。
- c-ares：提供了异步处理 DNS 相关的能力。
- http_parser、OpenSSL、zlib 等：提供包括 http 解析、SSL、数据压缩等其他的能力。

接下来，我们就主要看两个依赖库：V8 和 Libuv。

V8

谷歌开源的 JavaScript 引擎，目的是使 JavaScript 能够在浏览器之外的地方运行。前面说过，Javascipt引擎是一个能够将 Javascript 语言转换成浏览器能够识别的低级语言或机器码的程序。

Libuv

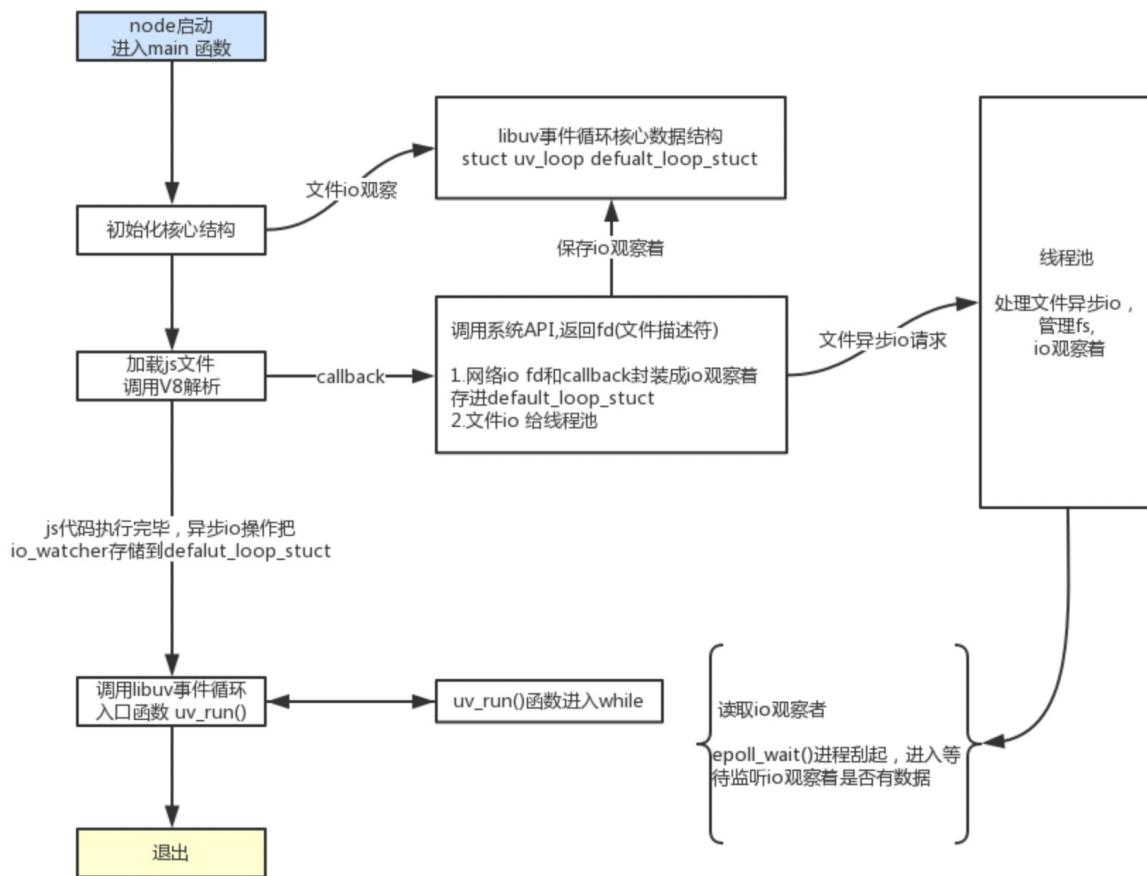
C++的开源项目，使 Node 能够访问操作系统的底层文件系统（file system），访问网络（networking）并且处理一些高并发相关的问题。

那么问题来了，既然 V8 能够让我们使用 JavaScript，libuv 给了我们一些操作系统、网络等层面的访问能力，我们还需要 Node 干嘛呢？

```
V8 大约70%由 C++ 实现，30%由 JavaScript 实现。
Libuv 100% 由 C++ 实现。
```

原因不难理解：

- 因为V8 和 libuv 都并非是用 JavaScript 写的，对于我们前端儿来说，写 C++ 是头疼的事情，而 Node 为我们提供了一个很好的接口，用来将 javascript 应用程序的 javascript 端与运行在我们计算机上的实际 c++ 关联起来，从而实际地解释和执行 javascript 代码。
- Node 封装了一系列的 API 供我们使用，并且提供了一致的接口。



Redis

Redis是一个基于BSD开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API.

- 1.Redis存储的是一个个的键值对
- 2.通常用Redis做缓存数据库
- 3.Redis的五种数据类型(字符串，哈希，链表，无序集合，有序集合)

mac下安装redis

```
// 下载redis数据库
wget http://download.redis.io/releases/redis-3.2.5.tar.gz

// 解压文件夹
tar xzf redis-3.2.5.tar.gz
cd redis-3.2.5

// 安装
make

// 运行redis
src/redis-server
```

安装 noderedis

```
npm install redis
```

在node中使用redis

nodejs 中的 redis 其实跟正常的终端使用redis 是一样的，只是他的结果值都是在一个回掉函数中。回掉函数的第一个参数是报错参数，第二个参数是返回的结果 如果没有报错，err 一般都是返回 null，如果有报错，第二个参数的返回结果和在终端执行的返回结果是一致的，不管是设置类的操作，还是获取数据的操作

```
var redis = require('redis');
var client = redis.createClient(6379, '127.0.0.1');
client.auth(123456); // 如果没有设置密码 是不需要这一步的
client.on('connect', function () {
    // set 语法
    client.set('name', 'long', function (err, data) {
        console.log(data)
    })
    // get 语法
})
```

```

client.get('name', function (err, data) {
  console.log(data)
})

client.lpush('class',1,function (err,data) {
  console.log(data)
})

client.lrange('class',0,-1,function (err,data) {
  console.log(data)
})
})
}

```

sequelize

什么是 ORM?

首先看下维基百科上的定义，ORM 是「对象关系映射」的翻译，英语全称为 Object Relational Mapping，它是一种程序设计技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的「虚拟对象数据库」。

随着面向对象软件开发方法的发展，ORM 的概念应运而生，它用来把对象模型表示的对象，映射到基于 SQL 的关系模型数据库结构中去。这样，我们在具体的操作实体数据库的时候，就不需要再去和复杂的 SQL 语句打交道，只需简单的操作实体对象的属性和方法，就可以达到操作数据库的效果。

ORM 技术是在对象和数据库之间提供了一条桥梁，前台的对象型数据和数据库中的关系型的数据通过这个桥梁来相互转化。

不同的编程语言，有不同的ORM框架。例如Java，它的ORM框架就有：Hibernate，Ibatis/Mybatis等等。在Node Web开发中，Sequelize 就是一款比较流行的 ORM 框架。

sequelize常用api

1、连接数据库

Name	Attribute
database	数据库名
username	数据库用户名.
password	密码
dialect	数据库方式
timezone	时区 默认'+00:00'
logging	日志
pool	连接池 min max idle

```

this.sequelize = new Sequelize({
  database: database,          //数据库名称
  username: username,         //数据库用户名
  password: password,        //数据库密码
  host: host,                //数据库主机IP localhost
  dialect: "mysql",           //数据库类型 'mysql'|'mariadb'|'sqlite'|'postgres'|'mssql',
  pool: {
    max: 5,                  //最大连接数
    min: 0,                  //最小连接数
    acquire: acquireTimeout,   //请求超时时间
    idle: 10000               //断开连接后，连接实例在连接池保持的时间
  },
  logging: this.logging //输出日志信息 true or false
});

```

2、创建表

```

this.tableModels.Users = this.sequelize.define('users', {
  mail: {
    type: Sequelize.STRING,      //数据类型 STRING,CHAR,INTEGER,FLOAT,FLOAT
    ,BOOLEAN,TEXT(不限长度)
    primaryKey: true,          // 主键 默认false
    allowNull: false,           // 是否可以为空 默认true
    defaultValue: ' 33'
  },
  name: { type: Sequelize.STRING, allowNull: false },
  password: { type: Sequelize.STRING(255) },
  authority: { type: Sequelize.INTEGER(1).UNSIGNED,values: ["1", "8"], }
}, {
  freezeTableName: true
});

```

3、处理数据

```

findAll()
findOne()
update()
bulkCreate()

```

Express

Express 是一个保持最小规模的灵活的 Node.js Web 应用程序开发框架，为 Web 和移动应用程序提供一组强大的功能。

入门

安装

```
npm install express --save
```

- body-parser用于处理 JSON, Raw, Text 和 URL 编码的数据。
- cookie-parser这是一个解析Cookie的工具。通过req.cookies可以取到传过来的cookie，并把它转成对象。
- multer用于处理 enctype="multipart/form-data"（设置表单的MIME编码）的表单数据。

```
npm install body-parser--save  
npm install cookie-parser--save  
npm install multer--save
```

request和response对象

Request 对象- request 对象表示 HTTP 请求，包含了请求查询字符串，参数，内容，HTTP 头部等属性。

```
req.app: 当callback为外部文件时，用req.app访问express的实例  
req.baseUrl: 获取路由当前安装的URL路径  
req.body / req.cookies: 获得「请求主体」 / Cookies  
req.fresh / req.stale: 判断请求是否还「新鲜」  
req.hostname / req.ip: 获取主机名和IP地址  
req.originalUrl: 获取原始请求URL  
req.params: 获取路由的parameters  
req.path: 获取请求路径  
req.protocol: 获取协议类型  
req.query: 获取URL的查询参数串  
req.route: 获取当前匹配的路由  
req.subdomains: 获取子域名  
req.accepts (): 检查请求的Accept头的请求类型  
req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages  
req.get (): 获取指定的HTTP请求头  
req.is (): 判断请求头Content-Type的MIME类型
```

Response 对象- response 对象表示 HTTP 响应，即在接收到请求时向客户端发送的 HTTP 响应数据。

```

res.app: 同req.app一样
res.append () : 追加指定HTTP头
res.set () 在res.append () 后将重置之前设置的头
res.cookie (name, value [, option]): 设置Cookie
option: domain / expires / httpOnly / maxAge / path / secure / signed
res.clearCookie () : 清除Cookie
res.download () : 传送指定路径的文件
res.get () : 返回指定的HTTP头
res.json () : 传送JSON响应
res.jsonp () : 传送JSONP响应
res.location () : 只设置响应的Location HTTP头, 不设置状态码或者close response
res.redirect () : 设置响应的Location HTTP头, 并且设置状态码302
res.send () : 传送HTTP响应
res.sendFile (path [, options] [, fn]): 传送指定路径的文件 -会自动根据文件extension设定Content-Type
res.set (): 设置HTTP头, 传入object可以一次设置多个头
res.status () : 设置HTTP状态码
res.type () : 设置Content-Type的MIME类型

```



路由

路由是指如何定义应用的端点（URLs）以及如何响应客户端的请求。路由（Routing）是由一个 URI（路径）和一个特定的 HTTP 方法（get、post、put、delete 等）组成的，涉及到应用如何响应客户端对某个网站节点的访问。

```

// Respond with Hello World! on the homepage:
app.get('/', function (req, res) {
  res.send('Hello World!')
})

// Respond to POST request on the root route (/), the application's home page:
app.post('/', function (req, res) {
  res.send('Got a POST request')
})

// Respond to a PUT request to the /user route:
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user')
})

// Respond to a DELETE request to the /user route:
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user')
})

```

利用 Express 托管静态文件

为了提供诸如图像、CSS 文件和 JavaScript 文件之类的静态文件，请使用 Express 中的 `express.static` 内置中间件函数。

例如，通过如下代码就可以将 `public` 目录下的图片、CSS 文件、JavaScript 文件对外开放访问了：

```
app.use(express.static('public'))
```

如果要使用多个静态资源目录，请多次调用 `express.static` 中间件函数：

```
app.use(express.static('public'))
app.use(express.static('files'))
```

如果要通过带有 `/static` 前缀地址来访问 `public` 目录中的文件了：

```
app.use('/static', express.static('public'))
```

中间件

应用层中间件

应用级中间键绑定到`app`对象使用`app.use`和`app.METHOD()`-需要处理http请求的方法，例如GET、PUT、POST，将之前的`get`或者`post`替换为`use`就行。

```
var express=require("express");
var app=express();

//匹配路由之前的操作
app.use(function(req,res,next){
    console.log("访问之前");
    next();
});

app.get("/",function(req,res){
    res.send("主页");
});

app.listen(8080);
```

路由中间件

```
var express=require("express");
var app = express();
var router=express.Router();
```

```
router.use("/",function(req,res,next){
    console.log("匹配前");
    next();
});

router.use("/user",function(req,res,next){
    console.log("匹配地址: ",req.originalUrl);
    next();
},function(req,res){
    res.send("用户登录");
});

app.use("/",router);
app.listen(8080);
```

hybrid

介绍

hybrid是客户端和前端的混合开发，需前端开发人员和客户端开发人员配合完成，某些环节可能涉及到server端。hybrid存在的核心意义在于快速迭代，无需审核。体验流畅，一套代码在ios和安卓端都可使用，减少开发成本。

移动应用开发

移动应用开发的方式，目前主要有三种：

- Native App：本地应用程序（原生App）
- Web App：网页应用程序（移动web）
- Hybrid App：混合应用程序（混合App）

file协议

浏览器打开本地文件，就是通过使用file协议

- file协议：本地文件，快
- http(s)协议：网络加载，慢

hybrid更新上线流程

- 分版本，有版本号，如201803211015
- 将静态文件压缩成zip包，上传到服务端
- 客户端每次启动，都去服务端检查版本号
- 如果服务端版本号大于客户端版本号，就去下载最新的zip包
- 下载完之后解压，覆盖原有文件

JS和客户端的通讯

- 1.js访问客户端能力，传递参数和回调函数
- 2.客户端通过回调函数返回内容
- 3.前端和客户端通讯的约定—schema协议
- 4.schema是内置上线的

schema协议

前端和客户端通讯的约定，就是一个接口形式

```
function abc(result) {
```

```
    console.log(result);
}
var message = 'weixin://dl/scan?callback=abc';
```

iframe调起端

```
function invokeScan() {
    var iframe = document.createElement('iframe');
    iframe.src = message;
    document.body.appendChild(iframe);
    document.body.removeChild(iframe);
}

document.getElementById('btn').addEventListener('click', function(){
    invokeScan(); // html调用schema协议
})
```

postMessage调起ios端

代码中的model是在app中使用WKWebViewConfiguration类中的方法注册

```
window.webkit.messageHandlers.model.postMessage(message);
```

prompt调起端

```
window.prompt(message);
```

location.href调起端

```
window.location.href = message;
```

自定义全局对象调起端

端上还可以通过自定义全局对象的方法，提供给前端调端使用。

端调用前端

端上通过调用schema中的回调方法，将返回给前端的结果，传入回调中。

React Native

环境搭建

必须安装的依赖有：Node、Watchman 和 React Native 命令行工具以及 Xcode。

虽然你可以使用任何编辑器来开发应用（编写 js 代码），但你仍然必须安装 Xcode 来获得编译 iOS 应用所需的工具和环境。

Node, Watchman

我们推荐使用Homebrew来安装 Node 和 Watchman。在命令行中执行下列命令安装：

```
brew install node // 版本需要v10以上  
brew install watchman
```

安装 react native 命令行工具

```
npm install -g yarn react-native-cli
```

安装完 yarn 后同理也要设置镜像源：

```
yarn config set registry https://registry.npm.taobao.org --global  
yarn config set disturl https://npm.taobao.org/dist --global
```

安装完 yarn 之后就可以用 yarn 替代 npm 了，例如用yarn代替npm install命令，用yarn add 某第三方库名代替npm install 某第三方库名。

安装Xcode

React Native 目前需要Xcode 10 或更高版本。你可以通过 App Store 或是到Apple 开发者官网上下载。这一步骤会同时安装 Xcode IDE、Xcode 的命令行工具和 iOS 模拟器。

开发项目

```
react-native init AwesomeProject
```

样式

在 React Native 中，你并不需要学习什么特殊的语法来定义样式。我们仍然是使用 JavaScript 来写样式。所有的核心组件都接受名为 style 的属性。这些样式名基本上是遵循了 web 上的 CSS 的命名，只是按照 JS 的语法要求使用了驼峰命名法，例如将 background-color 改为 backgroundColor。

style 属性可以是一个普通的 JavaScript 对象。

```
import React, { Component } from 'react';
import { StyleSheet, Text, View } from 'react-native';

export default class LotsOfStyles extends Component {
  render() {
    return (
      <View>
        <Text style={styles.red}>just red</Text>
        <Text style={styles.bigBlue}>just bigBlue</Text>
        <Text style={[styles.bigBlue, styles.red]}>bigBlue, then red</Text>
        <Text style={[styles.red, styles.bigBlue]}>red, then bigBlue</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  bigBlue: {
    color: 'blue',
    fontWeight: 'bold',
    fontSize: 30,
  },
  red: {
    color: 'red',
  },
});
```

组件和API

基础组件

大多数应用都会用到这里的基础组件。如果你是新手的话，那更应该先好好熟悉一下这些组件：

- View：搭建用户界面的最基础组件。
- Text：显示文本内容的组件。
- Image：显示图片内容的组件。
- TextInput：文本输入框。
- ScrollView：可滚动的容器视图。
- StyleSheet：提供类似CSS样式表的样式抽象层。

交互控件

提供一些常见的跨平台的交互控件。

- Button: 一个简单的跨平台的按钮控件。
- Picker: 在iOS和Android上调用各自原生的选择器控件。
- Slider: 滑动数值选择器。
- Switch: 开关控件。

列表视图

和一般化用途的ScrollView不同，下面的列表组件只会渲染当前屏幕可见的元素，这样有利于显示大量的数据。

- FlatList: 高性能的滚动列表组件。
- SectionList: 类似FlatList，但是多了分组显示。

iOS 独有的组件和 API

下面很多组件都是对常用的 UIKit 类的封装。

- ActionSheetIOS: 从设备底部弹出一个显示一个ActionSheet弹出框选项菜单或分享菜单。
- AlertIOS: 弹出一个提示对话框，还可以带有输入框。
- DatePickerIOS: 显示一个日期/时间选择器。
- ImagePickerIOS: 插入图片。
- ProgressViewIOS: 渲染一个UIProgressView进度条。
- PushNotificationIOS: 管理推送通知，包括权限处理和应用角标数字。
- SegmentedControlIOS: 渲染一个UISegmentedControl顶部选项卡布局
- TabBarIOS: 渲染一个UITabBarController底部选项卡布局。需要和TabBarIOS.Item搭配使用。

Android 独有的组件和 API

下面很多组件提供了对 Andriod 常用类的封装。

- BackHandler: 监听并处理设备上的返回按钮。
- DatePickerAndroid: 打开日期选择器。
- DrawerLayoutAndroid: 渲染一个DrawerLayout抽屉布局。
- PermissionsAndroid: 对Android 6.0引入的权限模型的封装。
- ProgressBarAndroid: 渲染一个ProgressBar进度条。
- TimePickerAndroid: 打开时间选择器。
- ToastAndroid: 弹出一个Toast提示框。
- ToolbarAndroid: 在顶部渲染一个Toolbar工具栏。
- ViewPagerAndroid: 可左右翻页滑动的视图容器。

其他

下面的组件可能适用于一些特定场景。

- ActivityIndicator: 显示一个圆形的正在加载的符号。
- Alert: 弹出一个提示框，显示指定的标题和信息。
- Animated: 易于使用和维护的动画库，可生成流畅而强大的动画。
- CameraRoll: 访问本地相册。

- Clipboard: 读写剪贴板内容。
- Dimensions: 获取设备尺寸。
- KeyboardAvoidingView: 一种视图容器, 可以随键盘升起而自动移动。
- Linking: 提供了一个通用的接口来调起其他应用或被其他应用调起。
- Modal: 一种简单的覆盖全屏的模态视图。
- PixelRatio: 可以获取设备的像素密度。
- RefreshControl: 此组件用在ScrollView及其衍生组件的内部, 用于添加下拉刷新的功能。
- StatusBar: 用于控制应用顶部状态栏样式的组件。
- WebView: 在原生视图中显示Web内容的组件。

编译并运行

```
react-native run-ios
```

react-native run-ios只是运行应用的方式之一。你也可以在 Xcode 中直接运行应用。注意0.60版本之后的主项目文件是.xcworkspace, 不是.xcodeproj。

在 iOS 设备上运行应用

1. 通过 USB 数据线连接设备 使用USB闪电数据线连接iOS设备到Mac。导航到工程的ios文件夹, 然后用Xcode打开.xcodeproj文件, 如果使用CocoaPods打开.xcworkspace。

如果这是第一次在iOS设备上运行app, 需要注册开发设备。从Xcode菜单栏打开Product菜单, 然后前往Destination。从列表中查找并选择设备。Xcode 将注册为开发设备。

1. 配置代码签名 如果没有Apple developer account, 先注册。

在Xcode Project导航中选择project, 然后选择main target (它应该和project共享同样的名字)。查找"General"标签。前往"Signing"并确保在"Team"下拉下选择了开发者账号或团队。tests target (以Tests结束, 在main target下面) 所做相同。

1. 编译并运行应用 如果一切设置正确, 设备会在Xcode toolbar 中被列为build target, 也会出现在设备面板里(⇧⌘2)。现在可以按下 Build and run 按钮(⌘R)或从Product菜单中选择Run。app会立刻启动在设备上。

flutte

环境安装

获取Flutter SDK

- 去flutter官网下载其最新可用的安装
- 解压安装包到你想安装的目录，如：

```
cd ~/development unzip ~/Downloads/flutter_macos_v0.5.1-beta.zip
```

- 添加flutter相关工具到path中：

```
export PATH= pwd /flutter/bin:$PATH
```

- 运行 flutter doctor

该命令检查您的环境并在终端窗口中显示报告。Dart SDK已经在捆绑在Flutter里了，没有必要单独安装Dart。仔细检查命令行输出以获取可能需要安装的其他软件或进一步需要执行的任务

第一次运行一个flutter命令（如flutter doctor）时，它会下载它自己的依赖项并自行编译。以后再运行就会快得多。

iOS 设置

安装 Xcode

要为iOS开发Flutter应用程序，您需要Xcode 7.2或更高版本：

配置Xcode命令行工具以使用新安装的Xcode版本 sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer 对于大多数情况，当您想要使用最新版本的Xcode时，这是正确的路径。如果您需要使用不同的版本，请指定相应路径。

确保Xcode许可协议是通过打开一次Xcode或通过命令sudo xcodebuild -license同意过了。

使用Xcode，您可以在iOS设备或模拟器上运行Flutter应用程序。

设置iOS模拟器

要准备在iOS模拟器上运行并测试您的Flutter应用，请按以下步骤操作：

- 在Mac上，通过Spotlight或使用以下命令找到模拟器：

```
open -a Simulator
```

- 通过检查模拟器 硬件>设备 菜单中的设置，确保您的模拟器正在使用64位设备（iPhone 5s或更高版本）。
- 根据您的开发机器的屏幕大小，模拟的高清屏iOS设备可能会使您的屏幕溢出。在模拟器的

Window> Scale 菜单下设置设备比例

- 运行 flutter run 启动您的应用。

编写您的第一个FlutterApp

替换 lib/main.dart。删除 lib / main.dart 中的所有代码，然后替换为下面的代码，它将在屏幕的中心显示“Hello World”。

```
import 'package:flutter/material.dart';

void main() => runApp(new MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new MaterialApp(
      title: 'Welcome to Flutter',
      home: new Scaffold(
        appBar: new AppBar(
          title: new Text('Welcome to Flutter'),
        ),
        body: new Center(
          child: new Text('Hello World'),
        ),
      ),
    );
  }
}
```

将一个 widget 传给 runApp 函数，该函数接受给定的 Widget 并使其成为 widget 树的根。

widget

Flutter Widget 采用现代响应式框架构建，这是从 React 中获得的灵感，中心思想是用 widget 构建你的 UI。Widget 描述了他们的视图在给定其当前配置和状态时应该看起来像什么。当 widget 的状态发生变化时，widget 会重新构建 UI，Flutter 会对比前后变化的不同，以确定底层渲染树从一个状态转换到下一个状态所需的最小更改（译者语：类似于 React/Vue 中虚拟 DOM 的 diff 算法）。

Flutter 有一套丰富、强大的基础 widget，其中以下是很常用的：

- Text：该 widget 可以创建一个带格式的文本。
- Row、Column：这些具有弹性空间的布局类 Widget 可让您在水平 (Row) 和垂直 (Column) 方向上创建灵活的布局。其设计是基于 Web 开发中的 Flexbox 布局模型。
- Stack：取代线性布局（译者语：和 Android 中的 LinearLayout 相似），Stack 允许子 widget 堆叠，你可以使用 Positioned 来定位他们相对于 Stack 的上下左右四条边的位置。Stacks 是基于 Web 开发中的绝对定位 (absolute positioning) 布局模型设计的。

- Container: Container 可让您创建矩形视觉元素。container 可以装饰为一个BoxDecoration, 如 background、一个边框、或者一个阴影。Container 也可以具有边距 (margins) 、填充 (padding)和应用于其大小的约束(constraints)。另外, Container可以使用矩阵在三维空间中对其进行变换。

使用 packages

要将包'css_colors'添加到应用中, 请执行以下操作

- 1、依赖它 打开 pubspec.yaml 文件, 然后在dependencies下添加css_colors:
- 2、安装它 在 terminal中: 运行 flutter packages get 或者 在 IntelliJ中: 点击pubspec.yaml文件顶部的'Packages Get'
- 3、导入它 在您的Dart代码中添加相应的import语句.

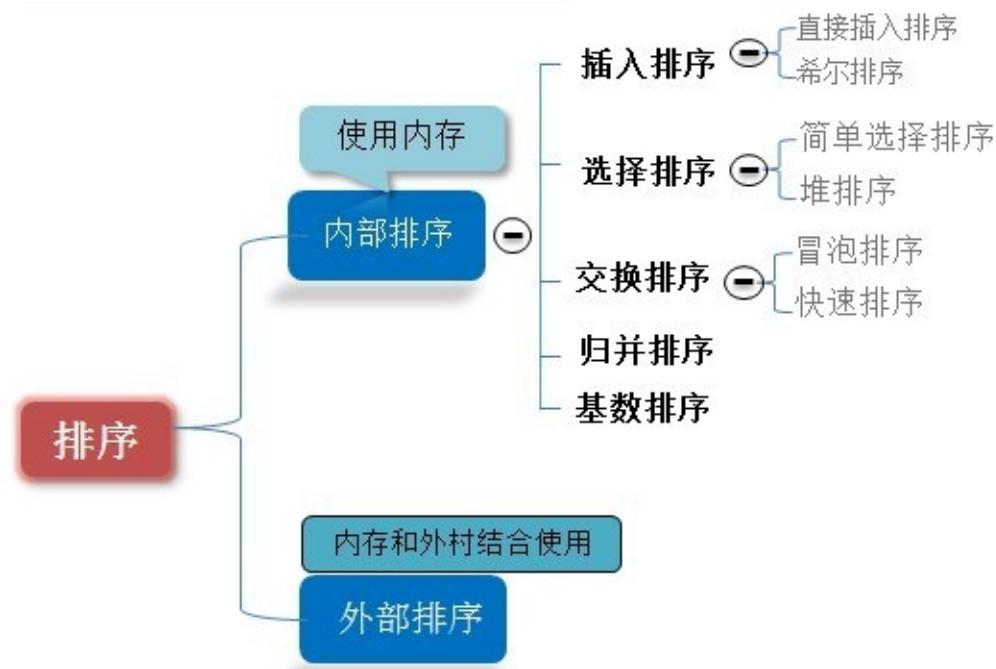
dart语法

- 在Dart中, 一切都是对象, 一切对象都是class的实例, 哪怕是数字类型、方法甚至null都是对象, 所有的对象都是继承自Object
- 虽然Dart是强类型语言, 但变量类型是可选的因为Dart可以自动推断变量类型
- Dart支持范型, List表示一个整型的数据列表, List则是一个对象的列表, 其中可以装任意对象
- Dart支持顶层方法 (如main方法), 也支持类方法或对象方法, 同时你也可以在方法内部创建方法
- Dart支持顶层变量, 也支持类变量或对象变量
- 跟Java不同的是, Dart没有public protected private等关键字, 如果某个变量以下划线 (_) 开头, 代表这个变量在库中是私有的, 具体可以看这里
- Dart中变量可以以字母或下划线开头, 后面跟着任意组合的字符或数字

八大算法

排序有内部排序和外部排序，内部排序是数据记录在内存中进行排序，而外部排序是因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存。

我们这里说说八大排序就是内部排序。



性能比较：

各种常用排序算法						
类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况		
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	不稳定
归并排序		$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中， r 代表关键字的基数， d 代表长度， n 代表关键字的个数

直接插入排序

将数组中的所有元素依次跟前面已经排好的元素相比较，如果选择的元素比已排序的元素小，则交换，直到全部元素都比较过。因此，从上面的描述中我们可以发现，直接插入排序可以用两个循环完成：

第一层循环：遍历待比较的所有数组元素 第二层循环：将本轮选择的元素(selected)与已经排好序的元素(ordered)相比较。如果：selected > ordered，那么将二者交换



```

function insertSort(arr) {
    for (let i = 1; i < arr.length; i++) {
        // 将待插入元素提取出来
        let temp = arr[i]
        let j
        for (j = i - 1; j >= 0; j--) {
            if (arr[j] > temp) {
                // 插入元素小于比较元素，比较元素则向后移动一位
                arr[j + 1] = arr[j]
            } else {
                // 否则，结束移位
                break
            }
        }
        // 将插入元素插入正确位置
        arr[j + 1] = temp
    }
    return arr
}
console.log(insertSort([7, 3, 4, 5, 10, 7, 8, 2]))

```

冒泡排序

对相邻的元素进行两两比较，顺序相反则进行交换，这样，每一趟会将最小或最大的元素“浮”到顶端，最终达到完全有序。（始终移动最大或最小的一个）

```

List sorted!
do
    swapped = false
    for i = 1 to indexOfLastUnsortedElement
        if leftElement > rightElement
            swap(leftElement, rightElement)
            swapped = true
    while swapped

```

```

function bubbleSort(arr) {
    for (let i = 0; i < arr.length; i++) {
        // 因为每次比较时都已经有i个元素沉下去了，所以j<arr.length-1-i
        for (let j = 0; j < arr.length - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                // 这里采用了解构赋值。如果一般做法，借助临时变量，则辅助空间是O(1)
                ;[arr[j], arr[j + 1]] = [arr[j + 1], arr[j]]
            }
        }
    }
    return arr
}
console.log(bubbleSort([7, 3, 4, 5, 10, 7, 8, 2]))

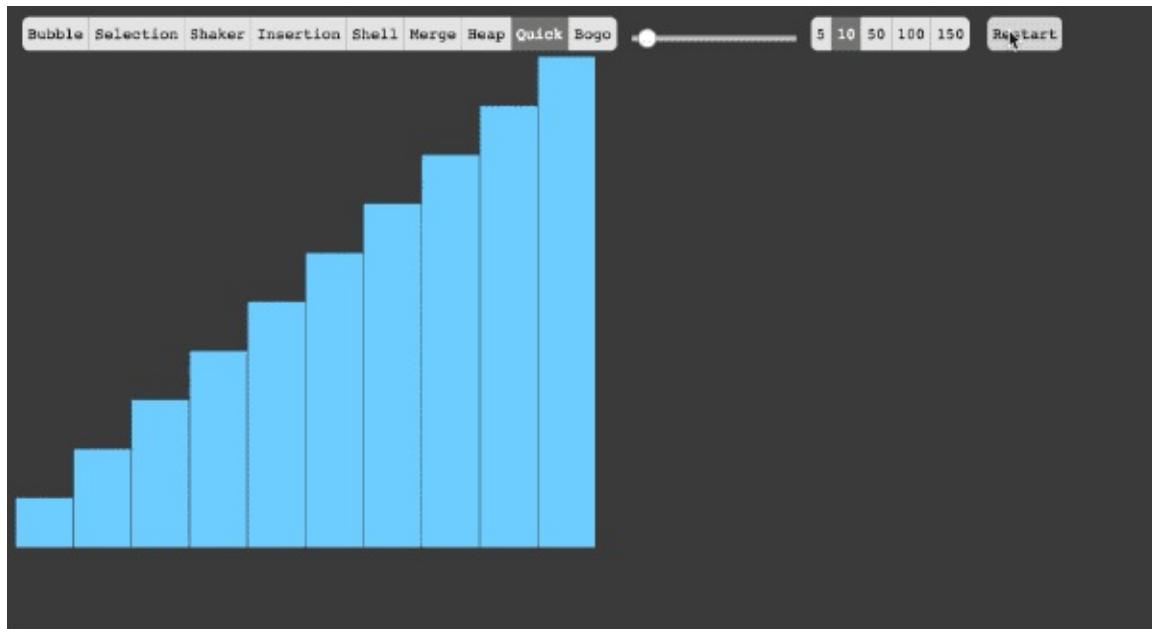
```

快速排序

快速排序的基本思想：挖坑填数+分治法

- 从序列当中选择一个基准数(pivot)，在这里我们选择序列当中第一个数最为基准数
- 将序列当中的所有数依次遍历，比基准数大的位于其右侧，比基准数小的位于其左侧

重复步骤1.2，直到所有子集当中只有一个元素为止。



```

let quicksort = function(arr) {
    if(arr.length <= 1) return arr;

    let pivot = Math.floor((arr.length -1)/2);
    let val = arr[pivot], less = [], more = [];

    arr.splice(pivot, 1);
    arr.forEach(function(e,i,a){
        e < val ? less.push(e) : more.push(e);
    });

    return (quicksort(less)).concat([val],quicksort(more))
}
console.log(quicksort([7, 3, 4, 5, 10, 7, 8, 2]))

```

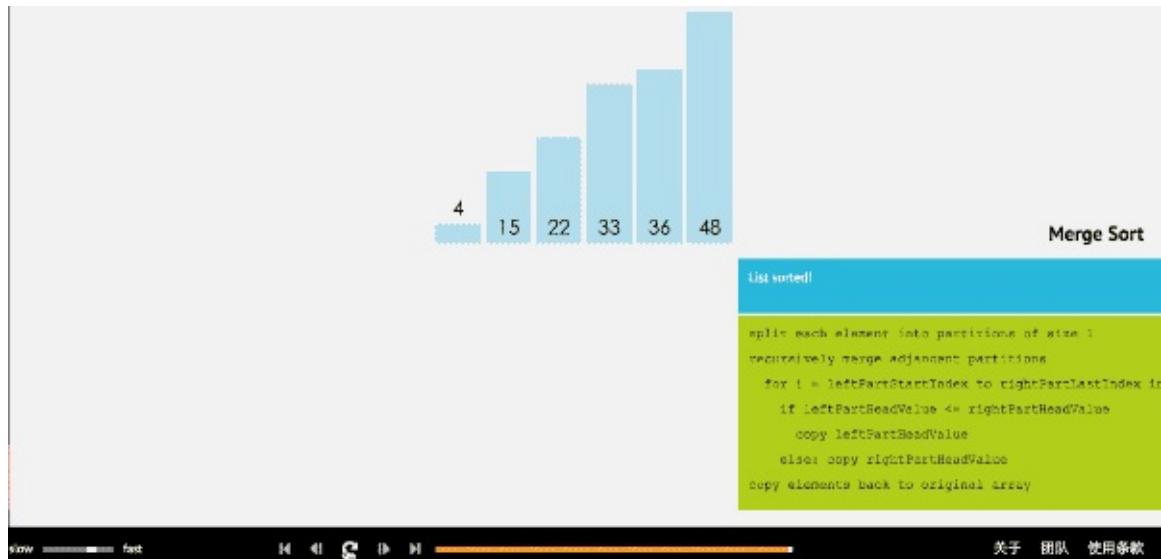
归并排序

归并排序是建立在归并操作上的一种有效的排序算法，该算法是采用分治法的一个典型的应用。它的基本操作是：将已有的子序列合并，达到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

归并排序其实要做两件事：

- 分解----将序列每次折半拆分
- 合并----将划分后的序列段两两排序合并

因此，归并排序实际上就是两个操作，拆分+合并



```

function merge(left, right) {
  let result = []
  while (left.length > 0 && right.length > 0) {
    if (left[0] < right[0]) {
      /*shift()方法用于把数组的第一个元素从其中删除，并返回第一个元素的值。*/
      result.push(left.shift())
    } else {
      result.push(right.shift())
    }
  }
  return result.concat(left).concat(right)
}
function mergeSort(arr) {
  if (arr.length == 1) {
    return arr
  }
  let middle = Math.floor(arr.length / 2),
    left = arr.slice(0, middle),
    right = arr.slice(middle)
  return merge(mergeSort(left), mergeSort(right))
}
console.log(mergeSort([7, 3, 4, 5, 10, 7, 8, 2]))

```

直接选择排序

简单选择排序的基本思想：比较+交换。

从待排序序列中，找到关键字最小的元素；如果最小元素不是待排序序列的第一个元素，将其和第一个元素互换；从余下的 N - 1 个元素中，找出关键字最小的元素，重复(1)、(2)步，直到排序结束。

因此我们可以发现，简单选择排序也是通过两层循环实现。

- 第一层循环：依次遍历序列当中的每一个元素
- 第二层循环：将遍历得到的当前元素依次与余下的元素进行比较，符合最小元素的条件，则交

换。

```

function directSelectSort(arr) {
    for (let i = 0; i < arr.length; i++) {
        let min = arr[i]
        let index = i
        for (let j = i + 1; j < arr.length; j++) {
            if (arr[j] < min) {
                // 找到最小值，并标注最小值索引，方便后续与元素arr[i]交换位置
                min = arr[j]
                index = j
            }
        }
        arr[index] = arr[i]
        arr[i] = min
    }
    return arr
}
console.log(directSelectSort([7, 3, 4, 5, 10, 7, 8, 2]))

```

希尔排序

先取一个小于n的整数d1作为第一个增量，把文件的全部记录分成d1个组。所有距离为d1的倍数的记录放在同一个组中。先在各组内进行直接插入排序；然后，取第二个增量d2< d1重复上述的分组和排序，直至所取的增量dt=1(dt< dt-1<...< d2< d1)，即所有记录放在同一组中进行直接插入排序为止。

```

function shellSort(arr) {
    let d = arr.length
    while (true) {
        d = Math.floor(d / 2)
        for (let x = 0; x < d; x++) {

```

```

        for (let i = x + d; i < arr.length; i = i + d) {
            let temp = arr[i]
            let j
            for (j = i - d; j >= 0 && arr[j] > temp; j = j - d) {
                arr[j + d] = arr[j]
            }
            arr[j + d] = temp
        }
    }
    if (d == 1) {
        break
    }
}
return arr
}
console.log(shellSort([7, 3, 4, 5, 10, 7, 8, 2]))

```

堆排序

堆的概念：

堆：本质是一种数组对象。特别重要的一点性质：任意的叶子节点小于（或大于）它所有的父节点。对此，又分为大顶堆和小顶堆，大顶堆要求节点的元素都要大于其孩子，小顶堆要求节点元素都小于其左右孩子，两者对左右孩子的大小关系不做任何要求。利用堆排序，就是基于大顶堆或者小顶堆的一种排序方法。下面，我们通过大顶堆来实现。

基本思想：

堆排序可以按照以下步骤来完成：

- 首先将序列构建称为大顶堆；（这样满足了大顶堆那条性质：位于根节点的元素一定是当前序列的最大值）
- 取出当前大顶堆的根节点，将其与序列末尾元素进行交换；（此时：序列末尾的元素为已排序的最大值；由于交换了元素，当前位于根节点的堆并不一定满足大顶堆的性质）对交换后的n-1个序列元素进行调整，使其满足大顶堆的性质；
- 重复2.3步骤，直至堆中只有1个元素为止

```

let len

function buildMaxHeap(arr) {
    //建立大根堆
    len = arr.length
    for (let i = Math.floor(len / 2); i >= 0; i--) {
        heapify(arr, i)
    }
}

function heapify(arr, i) {
    //堆调整
}

```

```

let left = 2 * i + 1,
right = 2 * i + 2,
largest = i

if (left < len && arr[left] > arr[largest]) {
    largest = left
}

if (right < len && arr[right] > arr[largest]) {
    largest = right
}

if (largest !== i) {
    // 解构赋值，交换变量
    ;[arr[i], arr[largest]] = [arr[largest], arr[i]]
    heapify(arr, largest)
}
}

function heapSort(arr) {
    buildMaxHeap(arr)

    for (let i = arr.length - 1; i > 0; i--) {
        ;[arr[0], arr[i]] = [arr[i], arr[0]]
        len--
        heapify(arr, 0)
    }
    return arr
}

console.log(heapSort([7, 3, 4, 5, 10, 7, 8, 2]))

```

基数排序

基数排序：通过序列中各个元素的值，对排序的N个元素进行若干趟的“分配”与“收集”来实现排序。

- 分配：我们将L[i]中的元素取出（位数不够可补0），首先确定其个位上的数字，根据该数字分配到与之序号相同的桶中
- 收集：当序列中所有的元素都分配到对应的桶中，再按照顺序依次将桶中的元素收集形成新的一个待排序列L[]

对新形成的序列L[]重复执行分配和收集元素中的十位、百位...直到分配完该序列中的最高位，则排序结束

The screenshot shows a radix sort visualization interface. At the top, there is a row of six boxes containing the numbers 1, 82, 127, 599, 622, and 743. Below this is a large, empty rectangular area for visualizing the sorting process. At the bottom, there is a control bar with the following elements from left to right: a 'slow' button, a 'fast' button, a left arrow, a right arrow, a search icon, another right arrow, a 'next' button, and a 'last' button. To the right of these buttons is a horizontal progress bar that is mostly orange. On the far right of the control bar are three small links: '关于' (About), '团队' (Team), and '使用条款' (Usage Terms).

```
// LSD Radix Sort
// helper function to get the last nth digit of a number
var getDigit = function(num,nth){
    // get last nth digit of a number
    var ret = 0;
    while(nth--){
        ret = num % 10
        num = Math.floor((num - ret) / 10)
    }
    return ret
}

// radixSort
function radixSort(arr){
    var max = Math.floor(Math.log10(Math.max.apply(Math,arr))),
        // get the length of digits of the max value in this array
        digitBuckets = [],
        idx = 0;

    for(var i = 0;i<max+1;i++){

        // rebuild the digit buckets according to this digit
        digitBuckets = []
        for(var j = 0;j<arr.length;j++){
            var digit = getDigit(arr[j],i+1);

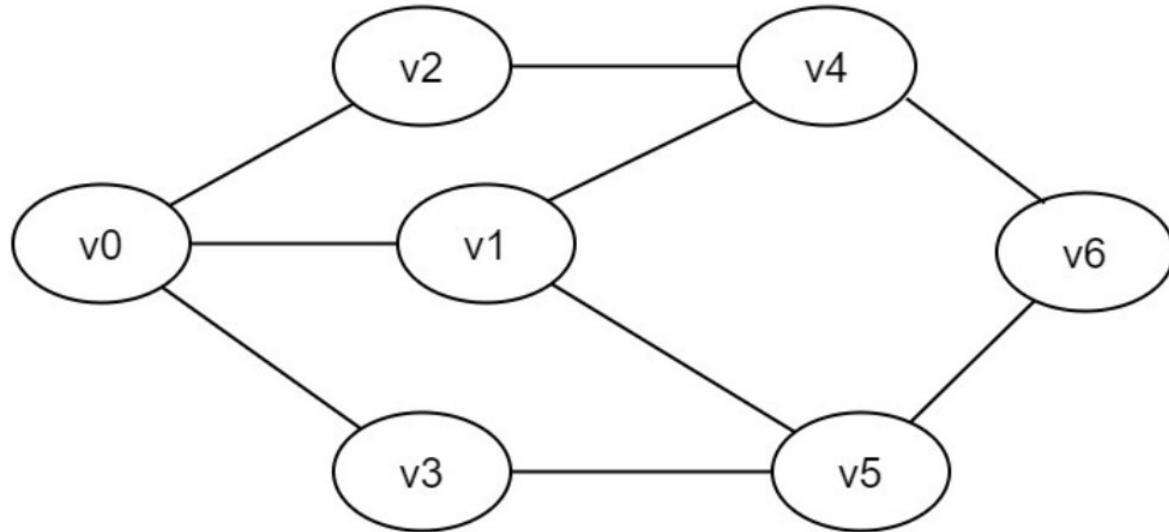
            digitBuckets[digit] = digitBuckets[digit] || [];
            digitBuckets[digit].push(arr[j]);
        }

        // rebuild the arr according to this digit
        arr = digitBuckets.flat();
    }
}
```

```
idx = 0
for(var t = 0; t< digitBuckets.length;t++){
    if(digitBuckets[t] && digitBuckets[t].length > 0){
        for(j = 0;j<digitBuckets[t].length;j++){
            arr[idx++] = digitBuckets[t][j];
        }
    }
}
return arr
}
console.log(radixSort([7, 3, 4, 5, 10, 7, 8, 2]))
```

BFS和DFS算法解析

图的遍历的定义：从图的某个顶点出发访问遍图中所有顶点，且每个顶点仅被访问一次



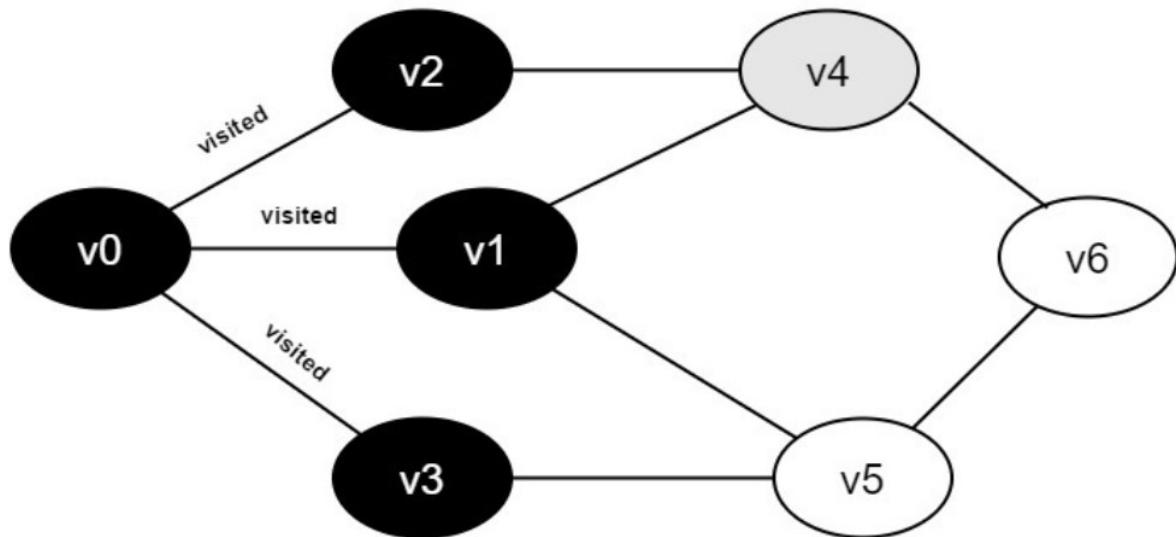
BFS

广度优先搜索类似于树的层次遍历过程。它需要借助一个队列来实现。要想遍历从 v_0 到 v_6 的每一个顶点，我们可以设 v_0 为第一层， v_1, v_2, v_3 为第二层， v_4, v_5 为第三层， v_6 为第四层，再逐个遍历每一层的每个顶点。

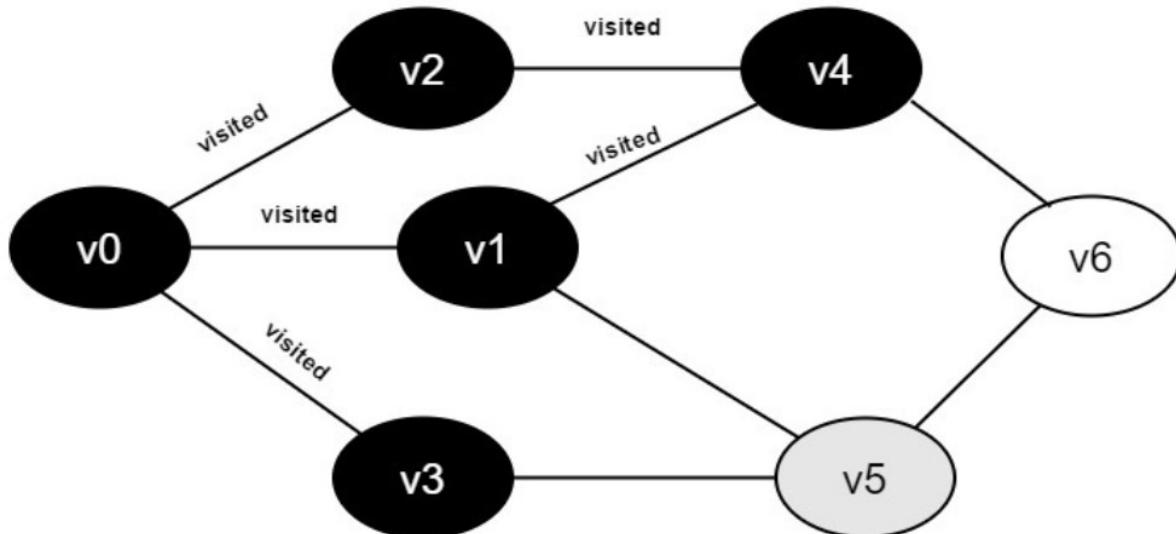
具体步骤：

- 1.准备工作：创建一个visited数组，用来记录已被访问过的顶点；创建一个队列，用来存放每一层的顶点；初始化图G。
- 2.从图中的 v_0 开始访问，将的visited[v_0]数组的值设置为true，同时将 v_0 入队。
- 3.只要队列不空，则重复如下操作：
 - (1)队头顶点 u 出队。
 - (2)依次检查 u 的所有邻接顶点 w ，若visited[w]的值为false，则访问 w ，并将visited[w]置为true，同时将 w 入队。

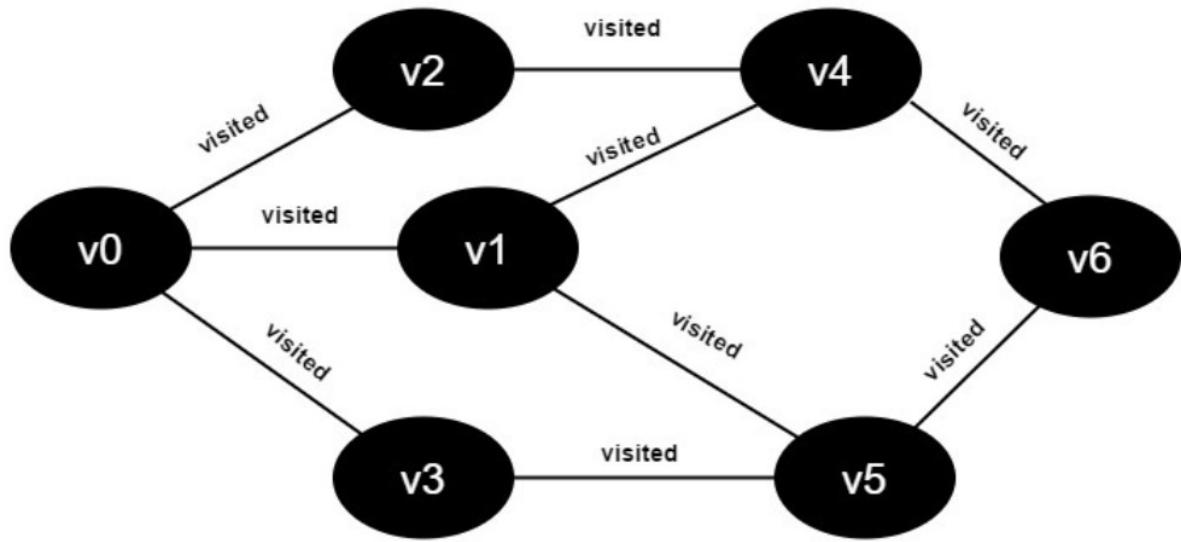
过程



v0的全部邻接点均已被访问完毕。将队头元素v2出队，开始访问v2的所有邻接点。开始访问v2邻接点v0，判断visited[0]，因为其值为1，不进行访问。继续访问v2邻接点v4，判断visited[4]，因为其值为0，访问v4



v2的全部邻接点均已被访问完毕。将队头元素v1出队，开始访问v1的所有邻接点。开始访问v1邻接点v0，因为visited[0]值为1，不进行访问。继续访问v1邻接点v4，因为visited[4]的值为1，不进行访问。继续访问v1邻接点v5，因为visited[5]值为0，访问v5



v5的全部邻接点均已被访问完毕，将队头元素v6出队，开始访问v6的所有邻接点。开始访问v6邻接点v4，因为visited[4]的值为1，不进行访问。继续访问v6邻接点v5，因为visited[5]的值为1，不进行访问。

DFS

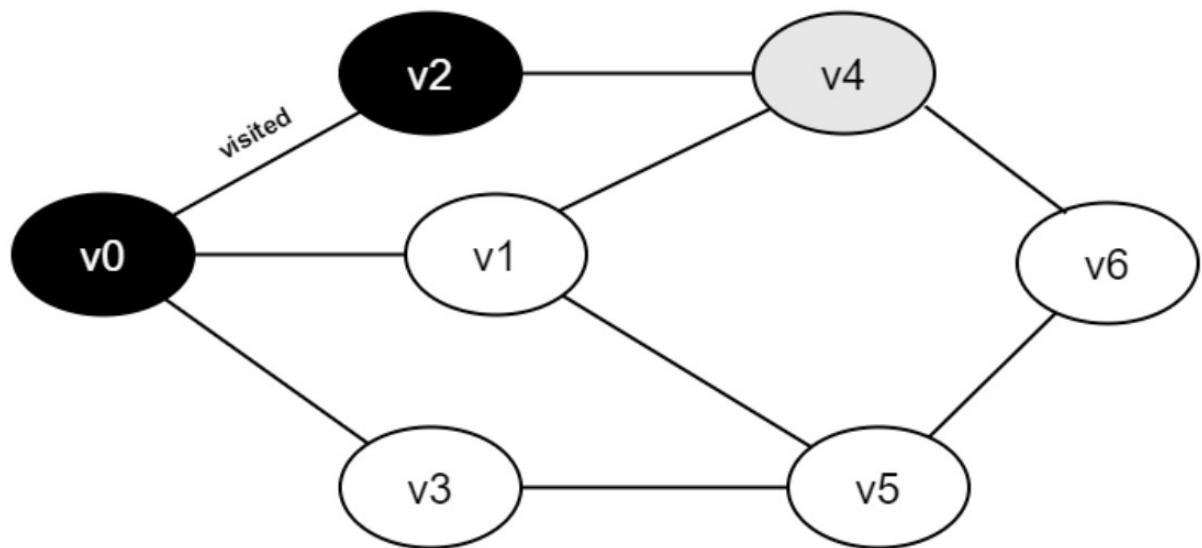
DFS遍历类似于树的先序遍历，是树的先序遍历的推广。

具体步骤：

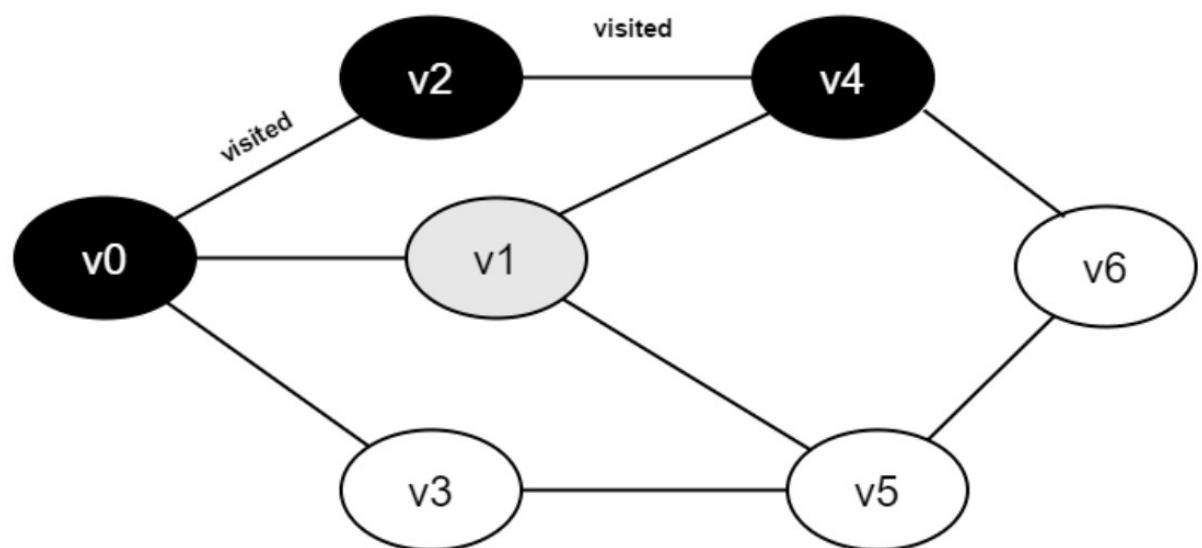
准备工作：创建一个visited数组，用于记录所有被访问过的顶点。

- 1.从图中v0出发，访问v0。
- 2.找出v0的第一个未被访问的邻接点，访问该顶点。以该顶点为新顶点，重复此步骤，直至刚访问过的顶点没有未被访问的邻接点为止。
- 3.返回前一个访问过的仍有未被访问邻接点的顶点，继续访问该顶点的下一个未被访问领接点。
- 4.重复2,3步骤，直至所有顶点均被访问，搜索结束。

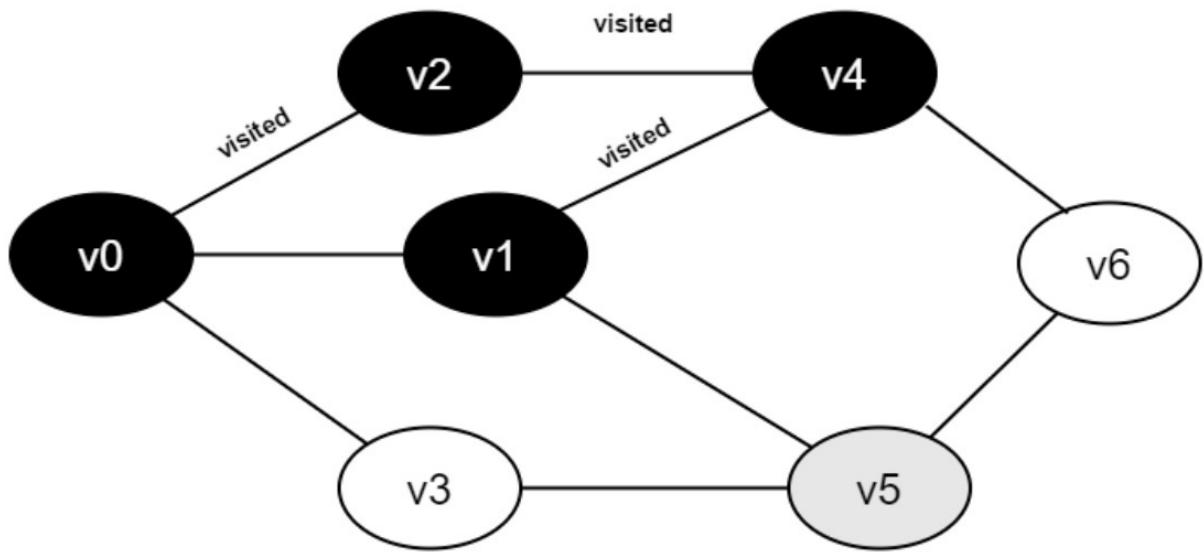
过程



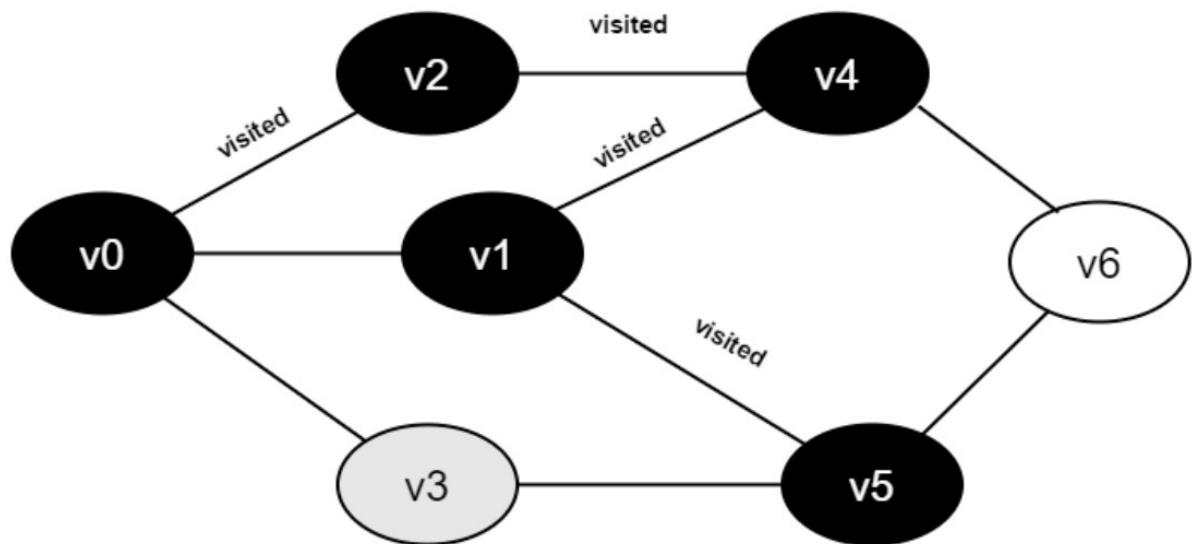
访问v2的邻接点v0, 判断visited[0], 其值为1, 不访问。继续访问v2的邻接点v4, 判断visited[4], 其值为0, 访问v4



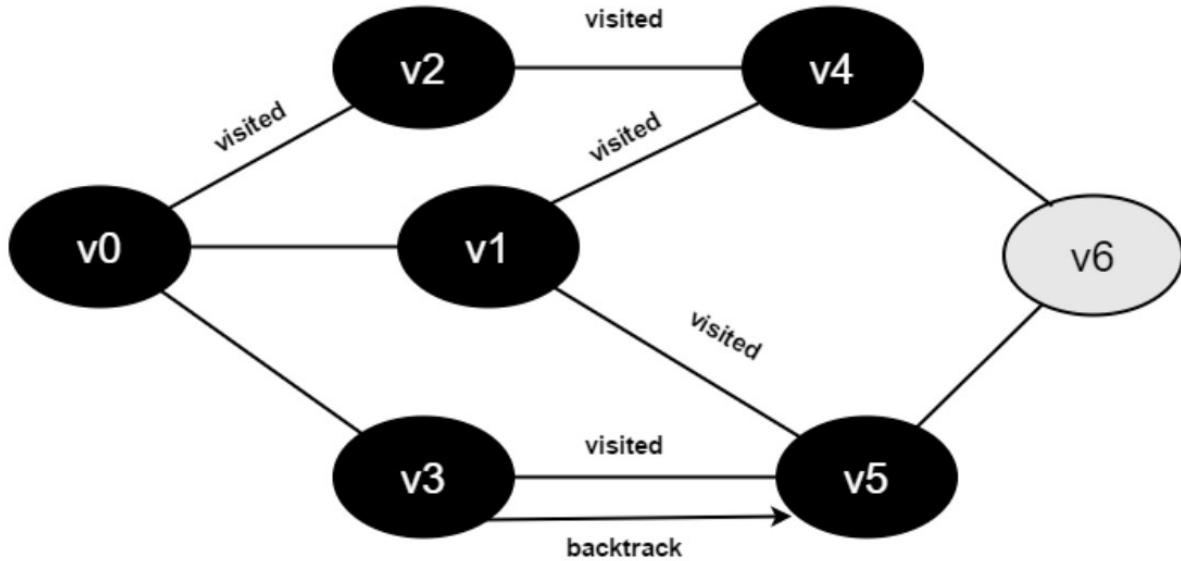
访问v4的邻接点v1, 判断visited[1], 其值为0, 访问v1



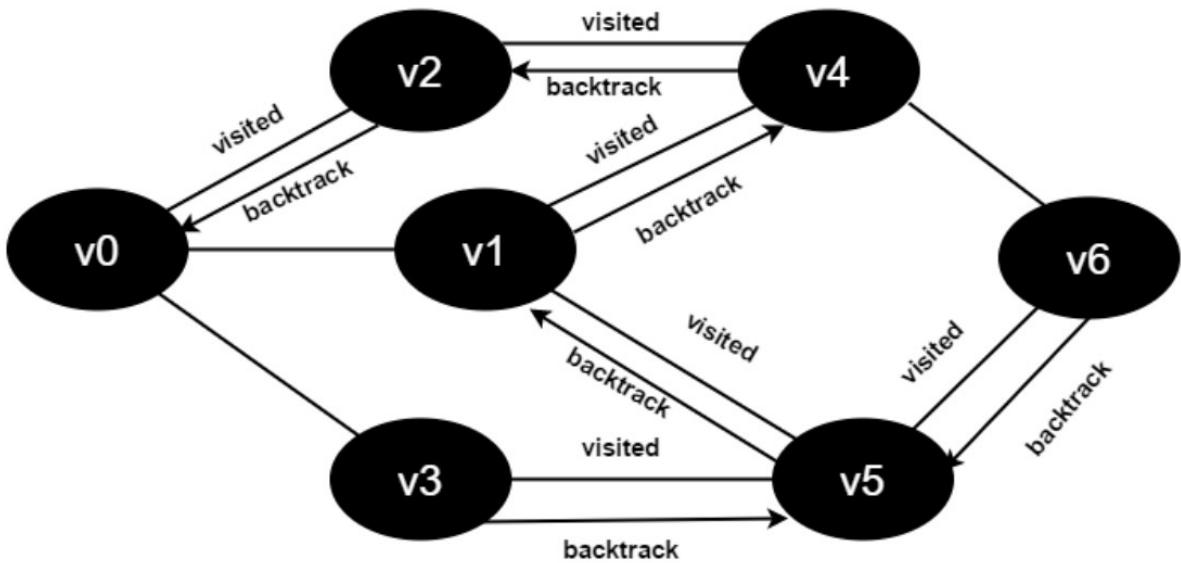
访问v1的邻接点v0，判断visited[0]，其值为1，不访问。继续访问v1的邻接点v4，判断visited[4]，其值为1，不访问。继续访问v1的邻接点v5，判断visited[5]，其值为0，访问v5。



访问v5的邻接点v1，判断visited[1]，其值为1，不访问。继续访问v5的邻接点v3，判断visited[3]，其值为0，访问v3。



访问v3的邻接点v0，判断visited[0]，其值为1，不访问。继续访问v3的邻接点v5，判断visited[5]，其值为1，不访问。v3所有邻接点均已被访问，回溯到其上一个顶点v5，遍历v5所有邻接点。访问v5的邻接点v6，判断visited[6]，其值为0，访问v6。



v5所有邻接点均已被访问，回溯到其上一个顶点v1。v1所有邻接点均已被访问，回溯到其上一个顶点v4，遍历v4剩余邻接点v6。v4所有邻接点均已被访问，回溯到其上一个顶点v2。v2所有邻接点均已被访问，回溯到其上一个顶点v1，遍历v1剩余邻接点v3。v1所有邻接点均已被访问，搜索结束。

剪枝算法

在搜索算法中优化中，剪枝就是通过某种判断，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”，故称剪枝。应用剪枝优化的核心问题是设计剪枝判断方法，即确定哪些枝条应当舍弃，哪些枝条应当保留的方法。

剪枝策略，通常应用在DFS 和 BFS 搜索算法中；就是寻找过滤条件，提前减少不必要的搜索路径。

设计模式

单例模式

核心思想是确保一个类只对应一个实例。每次调用构造函数时，返回指向同一个对象的指针。也就是说，我们只在第一次调用构造函数时创建新对象，之后调用返回时返回该对象即可。

```
function A(name){  
    var instance = this  
    this.name = name  
  
    //重写构造函数  
    A = function (){  
        return instance  
    }  
  
    // 第一种写法，这里实际上实现了一次原型链继承，如果不想这样实现，也可以直接指向原来的原型  
    A.prototype = this  
    // 第二种写法，直接指向旧的原型  
    A.prototype = this.constructor.prototype  
  
    instance = new A()  
  
    // 调整构造函数指针，这里实际上实现了一次原型链继承，如果不想这样实现，也可以直接指向原来的原型  
    instance.constructor = A  
  
    return instance  
}  
A.prototype.pro1 = "from protptype1"  
  
var a1 = new A()  
A.prototype.pro2 = "from protptype2"  
var a2= new A()  
  
console.log(a1.pro1)//from protptype1  
console.log(a1.pro2)//from protptype2  
console.log(a2.pro1)//from protptype1  
console.log(a2.pro2)//from protptype2
```

工厂模式

工厂模式是用来创建对象的一种最常用的设计模式。我们不暴露创建对象的具体逻辑，而是将逻辑封装在一个函数中，那么这个函数就可以被视为一个工厂。工厂模式根据抽象程度的不同可以分为：简单工厂，工厂方法和抽象工厂。

简单工厂模式

简单工厂模式又叫静态工厂模式，由一个工厂对象决定创建某一种产品对象类的实例。主要用来创建同一类对象。简单工厂只能作用于创建的对象数量较少，对象的创建逻辑不复杂时使用。

```

let UserFactory = function (role) {
    function User(opt) {
        this.name = opt.name;
        this.viewPage = opt.viewPage;
    }

    switch (role) {
        case 'superAdmin':
            return new User({ name: '超级管理员', viewPage: ['首页', '通讯录', '发现页', '应用数据', '权限管理'] });
            break;
        case 'admin':
            return new User({ name: '管理员', viewPage: ['首页', '通讯录', '发现页', '应用数据'] });
            break;
        case 'user':
            return new User({ name: '普通用户', viewPage: ['首页', '通讯录', '发现页'] });
            break;
        default:
            throw new Error('参数错误，可选参数:superAdmin、admin、user')
    }
}

//调用
let superAdmin = UserFactory('superAdmin');
let admin = UserFactory('admin')
let normalUser = UserFactory('user')

```

工厂方法模式

工厂方法模式的本意是将实际创建对象的工作推迟到子类中，这样核心类就变成了抽象类。

```

let UserFactory = function(role) {
    if(this instanceof UserFactory) {
        var s = new this[role]();
        return s;
    } else {
        return new UserFactory(role);
    }
}

//工厂方法函数的原型中设置所有对象的构造函数
UserFactory.prototype = {
    SuperAdmin: function() {

```

```

        this.name = "超级管理员",
        this.viewPage = ['首页', '通讯录', '发现页', '应用数据', '权限管理']
    },
    Admin: function() {
        this.name = "管理员",
        this.viewPage = ['首页', '通讯录', '发现页', '应用数据']
    },
    NormalUser: function() {
        this.name = '普通用户',
        this.viewPage = ['首页', '通讯录', '发现页']
    }
}

//调用
let superAdmin = UserFactory('SuperAdmin');
let admin = UserFactory('Admin')
let normalUser = UserFactory('NormalUser')

```

抽象工厂模式

上面介绍了简单工厂模式和工厂方法模式都是直接生成实例，但是抽象工厂模式不同，抽象工厂模式并不直接生成实例，而是用于对产品类簇的创建。在抽象工厂中，类簇一般用父类定义，并在父类中定义一些抽象方法，再通过抽象工厂让子类继承父类。所以，抽象工厂其实是实现子类继承父类的方法。

观察者模式

观察者模式又叫做发布—订阅模式，是我们最常用的设计模式之一。它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知和更新。观察者模式提供了一个订阅模型，其中对象订阅事件并在发生时得到通知，这种模式是事件驱动的编程基石，它有利于良好的面向对象的设计。

```

const event = {
    clientList: [],
    listen: function(key , fn) {
        if (this.clientList[key]) {
            this.clientList[key] = []
        }
        this.clientList[key].push(fn)
    },
    trigger: function() {
        const key = Array.prototype.shift.call(arguments)
        const fns = this.clientList[key]
        if (!fns || fns.length === 0 ) {
            return false
        }
        for (let i = 0, fn ;fn = fns[i++];) {
            fn.apply(this, arguments)
        }
    }
}

```

```
        }
    },
    remove : function(key , fn) {
        const fns = this.clientList[key]
        if (!fns) {
            return false
        }
        if (!fn) {
            fns && (fns.length = 0)
        } else {
            for (let l = fns.length - 1; l>=0; l--) {
                const _fn = fns[l]
                if ( _fn ===fn) {
                    fns.splice(l, 1)
                }
            }
        }
    }

const installEvent = (obj) => {
    for (let i in event) {
        obj[i] = event[i]
    }
}
```

web性能

网页性能指标

白屏时间

读取页面首字节时间 (ttfb - Time To First Byte) , 可以理解为用户拿到页面资源占用的时间。浏览器对html文档的解析和渲染是一个渐进的过程，一般在拿到首字节之后便会有内容绘制在页面上，正常网络状态下基本上白屏时间很短。

资源加载

浏览器在接收到服务器返回的 html 文档数据之后，会起一系列的线程去请求文档解析中遇到的各种资源，js脚本、CSS样式表、图片，以及发起异步请求。我们这里的资源认为是 js/css/图片，后面统计资源加载情况时，会统计这些资源的文件大小、文件数量、总的加载用时。ajax异步请求我们会另外进行统计。

用户可操作时间

在查阅相关资料时，会看到用户等待页面时间、用户可操作时间等概念，不同资料和文章的定义也不同，这里我们认为用户可操作时间就是用户可以进行页面操作的时间，此时 html 文档解析完成 (domContentLoadedEventEnd) 。另一种用户等待页面的时间，一般是按照页面加载完成的时间来统计 (loadEventEnd) 。但在我们这次的前端性能监控方案中，并不将其作为主要的监控指标。

首屏渲染时间

首屏时间的统计比较复杂，因为涉及图片资源的下载及异步请求等因素。有些资料统计中不计算图片的下载时间，但我们认为既然是首屏的展示，应当包括图片加载的完成。判断首屏图片加载完成的方法，这里不再详述，可以查阅相关文章。我们这次的前端性能分析方案中，并没有涉及到图片，而是关注页面初始化过程中的异步请求。

相关知识

影响网页性能的因素

- HTML 的解析和渲染
- 服务端处理的速度（负载均衡，缓存策略）
- 客户端带宽（网络状况）

浏览器解析渲染HTML页面的过程

- HTML 文档的解析和渲染是一个渐进的过程。为达到更好的用户体验，呈现引擎会力求尽快将内容显示在屏幕上。它不必等到整个 HTML 文档解析完毕，就会开始构建呈现树和设置布局。在不断接收和处理来自网络的其余内容的同时，呈现引擎会将部分内容解析并显示出来。
- 浏览器的预解析机制。

- HTML 文档的解析和渲染过程中，外部样式表和脚本顺序执行、并发加载。

JS 脚本会阻塞 HTML 文档的解析，包括 DOM 树的构建和渲染树的构建；CSS 样式表会阻塞渲染树的构建，但 DOM 树依然继续构建（除非遇到 `< script >` 标记时会立即解析并执行脚本，HTML 文档的解析将被阻塞，直到脚本执行完毕。如果脚本是外部的，那么解析过程会停止，直到从网络抓取资源并解析和执行完成后，再继续解析后续内容。但无论是哪种情况导致的阻塞，该加载的外部资源还是会加载，例如外部脚本、样式表和图片。HTML 文档的解析可能会被阻塞，但外部资源的加载不会被阻塞。

浏览器并发连接数

Chrome 浏览器的并发连接数为 6 个，超过限制数目的请求会被阻塞。

常见的性能优化方法

减少 HTTP请求数

- 从设计实现层面简化页面
- 合理设置 HTTP缓存
- 资源合并与压缩
- CSS Sprites
- Inline Images
- Lazy Load Images

将外部脚本置底

浏览器是可以并发请求的，这一特点使得其能够更快的加载资源，然而外链脚本在加载时却会阻塞其他资源，例如在脚本加载完成之前，它后面的图片、样式以及其他脚本都处于阻塞状态，直到脚本加载完成后才会开始加载。如果将脚本放在比较靠前的位置，则会影响整个页面的加载速度从而影响用户体验。

异步执行 inline脚本

inline脚本对性能的影响与外部脚本相比，是有过之而无不及。异步的方式有很多种，例如使用 `script` 元素的 `defer` 属性(存在兼容性问题和其他一些问题，例如不能使用 `document.write`)、使用 `setTimeout`，此外，在 HTML5 中引入了 Web Workers 的机制，恰恰可以解决此类问题。

Lazy Load Javascript

只有在需要加载的时候加载，在一般情况下并不加载信息内容。

将 CSS放在 HEAD中

减少不必要的 HTTP跳转

对于以目录形式访问的 HTTP链接，很多人都会忽略链接最后是否带'/'，假如你的服务器对此是区别对待的话，那么你也需要注意，这其中很可能隐藏了 301跳转，增加了多余请求。

避免重复的资源请求

这种情况主要是由于疏忽或页面由多个模块拼接而成，然后每个模块中请求了同样的资源时，会导致资源的重复请求

精简Javascript和CSS

使用CDN加速

CDN的全称是Content Delivery Network，即内容分发网络。CDN是构建在现有网络基础之上的智能虚拟网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。CDN的关键技术主要有内容存储和分发技术。

PerformanceAPI

能够实现对网页性能的监控，主要是依靠 Performance API。

```
Performance.timing
Performance.getEntries()
Performance.getEntriesByType()
Performance.now()
```

The screenshot shows the Chrome DevTools Performance tab. On the left, there's a summary panel with 8 messages (0 errors, 8 warnings, 0 info, 0 verbose). The main area displays a timeline with several performance entries. A detailed view of one entry shows the following properties:

```

memory: MemoryInfo {totalJSHeapSize: 19299648, usedJSHeapSize: 13914280, jsHeapSizeLimit: 2217857988}
navigation: PerformanceNavigation {type: 0, redirectCount: 0}
onresourceTimingbufferfull: null
timeOrigin: 1545637744172.0542
timing: PerformanceTiming {navigationStart: 1545637744173, unloadEventStart: 0, unloadEventEnd: 0, redirectStart: 0, redirectEnd: 0, connectEnd: 1545637744400, connectStart: 1545637744279, domComplete: 1545637747269, domContentLoadedEventEnd: 1545637746132, domContentLoadedEventStart: 1545637746129, domInteractive: 1545637746129, domLoading: 1545637744924, domainLookupEnd: 1545637744279, domainLookupStart: 1545637744207, fetchStart: 1545637744198, loadEventEnd: 1545637747272, loadEventStart: 1545637747269, navigationStart: 1545637744173, redirectEnd: 0, redirectStart: 0, requestStart: 1545637744440, responseEnd: 1545637745084, responseStart: 1545637744850, secureConnectionStart: 1545637744341, unloadEventEnd: 0, unloadEventStart: 0}
__proto__: PerformanceTiming
  
```

- DNS查询耗时：domainLookupEnd - domainLookupStart
- TCP链接耗时：connectEnd - connectStart
- request请求耗时：responseEnd - responseStart
- 解析dom树耗时：domComplete - domInteractive
- 白屏时间：responseStart - navigationStart

- domready时间 : domContentLoadedEventEnd - navigationStart
- onload时间 : loadEventEnd - navigationStart

缓存

存储对象

- 使用 localStorage 缓存常用的数据 现代化浏览器都支持 localStorage , 我们可以实现一个类似这样的功能, 当我们取一个数据的时候, 先去localStorage 中寻找, 没有再向服务器发送请求。夸张一点, 我们可以把图片、js文件存储到里面。淘宝真是物尽所用, 所有储存对象基本都用了。在 localStorage 存储了大量的数据, 包含图片、icon等等。当页面第二次加载的时候就不用去重复请求后台相应的资源了。(localStorage存储大约能存 4M)
- CDN域名不携带cookie cookie存储能带给后端, 所以主要是用来鉴别客户端的唯一性, 知道你是哪个用户。所以 cookie 中的数据不能无意义, 不能太大。我们的CDN域名最好和主域名分开, 这样在请求静态资源的时候就不会带上这个 cookie 了减少了请求头的大小, 减少了客户所需的流量。可能感觉微乎其微, 但是你假设你的 cookie 是 1k, 一天有 1万人访问, 访问静态资源请求 2 万此, 那么你就会白白消耗 1万K 的流量。
- 其余的存储对象都可以在适当的场景, 适当的使用。使用的时候要考虑兼容, 和最大储存容量。用的好是跑车, 用不好是拖拉机

cache-control

- max-age 指定缓存的最大有效时间, 时间之内再次请求资源, 不去发送http请求
- s-maxage 指定public的缓存的最大有效时间, 优先级高于max-age,会发送请求, 返回状态: 304
- private 用户所独有的缓存, 就是单一用户浏览器的缓存。
- public 公共缓存, 例如cdn的, 代理服务器的缓存。
- no-cache 指定缓存是否要发送http请求来询问服务器当前的缓存内容是否还有效, 搭配max-age=0使用, 有这个属性就会发送http请求询问服务器。
- no-store 完全不会存储

bigpipe

衡量一个页面的性能, 其中一个重要指标, 是这个页面“白屏”时间。这与yahoo的优化建议Flush the Buffer Early有相似之处。其实我们用传统的web技术就能实现这个优化, 只需要让webserver先输出一部分内容, 让client先渲染一部分, 耗时的部分后输出。

当server向client传递数据时, 我们很容易让server先传字符“a”,client渲染字符“a”, 过一秒后, 传递“b”时, 再渲染“b”。并不需要server把字符“a”、“b”都收集完后才整个传递给client端, 要知道等所有数据都收集完成, 这个时间是很漫长的, client会一直处于白屏, client也从来没这样要求过。在于开发者怎么优化了。

http 长连接

不是长连接时，只要server端或者client端close，两端都会知道这个连接已经断开了，需要数据通信时，再重新建立连接。但是长连接就不一样了，不能close。那两端总不能傻等着吧？要知道，如果不再需要数据传递，就应该把连接标示为空闲，因为长连接，这个连接要被其它需要数据通信的程序复用，传递其它数据。于是乎长连接下，增加了两个头

Content-Length 和 Transfer-Encoding: chunked

这两种方式都可以标识不再需要数据传递，这个连接可以放到空闲池，以备它用。

http1.0 在长连接时，必须要加

Content-Length

非长连接，直接close掉就行了。

http1.1长连接时，可以用

Content-Length 或 Transfer-Encoding: chunked

其中一个。两个同时使用时，Transfer-Encoding: chunked 优先级高。

PWA

PWA是一种理念，使用多种技术来增强web app的功能，可以让网站的体验变得更好，能够模拟一些原生功能，比如通知推送。在移动端利用标准化框架，让网页应用呈现和原生应用相似的体验。

PWA不能包含原生OS相关代码。PWA仍然是网站，只是在缓存、通知、后台功能等方面表现更好。

即使在不确定的网络条件下也不会受到影响。当用户从主屏幕启动时，service work可以立即加载渐进式Web应用程序，完全不受网络环境的影响。service work就像一个客户端代理，它控制缓存以及如何响应资源请求逻辑，通过预缓存关键资源，可以消除对网络的依赖，确保为用户提供即时可靠的体验。

PWA的实现

PWA并不是单指某一项技术，你更可以把它理解成是一种思想和概念，目的就是对标原生app，将Web网站通过一系列的Web技术去优化它，提升其安全性，性能，流畅性，用户体验等各方面指标，最后达到用户就像在用app一样的感觉。

- Web App Manifest
- Service Worker
- Cache API 缓存
- Push&Notification 推送与通知
- Background Sync 后台同步
- 响应式设计

Manifest实现添加至主屏幕

```

<head>
  <title>Minimal PWA</title>
  <meta name="viewport" content="width=device-width, user-scalable=no" />
  <link rel="manifest" href="manifest.json" />
  <link rel="stylesheet" type="text/css" href="main.css">
  <link rel="icon" href="/e.png" type="image/png" />
</head>

{
  "name": "Minimal PWA", // 必填 显示的插件名称
  "short_name": "PWA Demo", // 可选 在APP launcher和新的tab页显示, 如果没有设置, 则使用name
  "description": "The app that helps you understand PWA", // 用于描述应用
  "display": "standalone", // 定义开发人员对Web应用程序的首选显示模式。standalone模式会有单独的
  "start_url": "/", // 应用启动时的url
  "theme_color": "#313131", // 桌面图标的背景色
  "background_color": "#313131", // 为web应用程序预定义的背景颜色。在启动web应用程序和加载应用
  程序的内容之间创建了一个平滑的过渡。
  "icons": [ // 桌面图标, 是一个数组
    {
      "src": "icon/lowres.webp",
      "sizes": "48x48", // 以空格分隔的图片尺寸
      "type": "image/webp" // 帮助userAgent快速排除不支持的类型
    },
    {
      "src": "icon/lowres",
      "sizes": "48x48"
    },
    {
      "src": "icon/hd_hi.ico",
      "sizes": "72x72 96x96 128x128 256x256"
    },
    {
      "src": "icon/hd_hi.svg",
      "sizes": "72x72"
    }
  ]
}

```

service worker实现离线缓存

- HTTP缓存

Web 服务器可以使用 `Expires` 首部来通知 Web 客户端, 它可以使用资源的当前副本, 直到指定的“过期时间”。反过来, 浏览器可以缓存此资源, 并且只有在有效期满后才会再次检查新版本。使用 HTTP 缓存意味着你要依赖服务器来告诉你何时缓存资源和何时过期。

- service worker缓存

Service Workers 的强大在于它们拦截 HTTP 请求的能力 进入任何传入的 HTTP 请求，并决定想要如何响应。在你的 Service Worker 中，可以编写逻辑来决定想要缓存的资源，以及需要满足什么条件和资源需要缓存多久。一切尽归你掌控！

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello Caching World!</title>
  </head>
  <body>
    <!-- Image -->
    
    <!-- JavaScript -->
    <script async src="/js/script.js"></script>
    <script>
      // 注册 service worker
      if ('serviceWorker' in navigator) {
        navigator.serviceWorker.register('/service-worker.js', {scope: '/'})
          .then(function (registration) {
            // 注册成功
            console.log('ServiceWorker registration successful with scope: ', registration.scope);
          })
          .catch(function (err) {
            // 注册失败 :(
            console.log('ServiceWorker registration failed: ', err);
          });
      }
    </script>
  </body>
</html>

var cacheName = 'helloWorld';      // 缓存的名称
// install 事件，它发生在浏览器安装并注册 Service Worker 时
self.addEventListener('install', event => {
  /* event.waitUntil 用于在安装成功之前执行一些预装逻辑
   * 但是建议只做一些轻量级和非常重要的资源的缓存，减少安装失败的概率
   * 安装成功后 ServiceWorker 状态会从 installing 变为 installed */
  event.waitUntil(
    caches.open(cacheName)
      .then(cache => cache.addAll([
        // 如果所有的文件都成功缓存了，便会安装完成。如果任何文件下载失败了，那么安装过程也会随之失败。
        '/js/script.js',
        '/images/hello.png'
      ]))
  );
});

/**

```

为 `fetch` 事件添加一个事件监听器。接下来，使用 `caches.match()` 函数来检查传入的请求 URL 是否匹配当前缓存中存在的任何内容。如果存在的话，返回缓存的资源。

如果资源并不存在于缓存当中，通过网络来获取资源，并将获取到的资源添加到缓存中。

```
/*
self.addEventListener('fetch', function (event) {
  event.respondWith(
    caches.match(event.request)
    .then(function (response) {
      if (response) {
        return response;
      }
      var requestToCache = event.request.clone(); // 
      return fetch(requestToCache).then(
        function (response) {
          if (!response || response.status !== 200) {
            return response;
          }
          var responseToCache = response.clone();
          caches.open(cacheName)
            .then(function (cache) {
              cache.put(requestToCache, responseToCache);
            });
          return response;
        })
      );
    });
});
```

Service worker实现消息推送

- 步骤一、提示用户并获得他们的订阅详细信息
- 步骤二、将这些详细信息保存在服务器上
- 步骤三、在需要时发送任何消息