JavaScript 部分

1. JavaScript 有哪些数据类型，它们的区别？

JavaScript 共有八种数据类型，分别是 Undefined、Null、Boolean、Number、String、Object、Symbol、BigInt。

其中 Symbol 和 BigInt 是 ES6 中新增的数据类型：

●Symbol 代表创建后独一无二且不可变的数据类型，它主要是为了解决可能出现的全局变量冲突的问题。

●BigInt 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 BigInt 可以安全地存储和操作大整数，即使这个数已经超出了 Number 能够表示的安全整数范围。

这些数据可以分为原始数据类型和引用数据类型：

●栈：原始数据类型（Undefined、Null、Boolean、Number、String）

●堆：引用数据类型（对象、数组和函数）

两种类型的区别在于存储位置的不同：

●原始数据类型直接存储在栈（stack）中的简单数据段，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储；

●引用数据类型存储在堆（heap）中的对象，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。

堆和栈的概念存在于数据结构和操作系统内存中，在数据结构中：

●在数据结构中，栈中数据的存取方式为先进后出。

●堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。

在操作系统中，内存被分为栈区和堆区：

●栈区内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

●堆区内存一般由开发着分配释放，若开发者不释放，程序结束时可能由垃圾回收机制回收。

2. 数据类型检测的方式有哪些

（1）typeof

```
console.log(typeof 2);              // number
console.log(typeof true);           // boolean
console.log(typeof 'str');          // string
console.log(typeof []);             // object
console.log(typeof function(){});   // function
console.log(typeof {});             // object
console.log(typeof undefined);      // undefined
console.log(typeof null);           // object
```

其中数组、对象、null 都会被判断为 object，其他判断都正确。

（2）instanceof

instanceof 可以正确判断对象的类型，其内部运行机制是判断在其原型链中能否找到该类型的原型。

```
console.log(2 instanceof Number);                    // false
console.log(true instanceof Boolean);                // false
console.log('str' instanceof String);                // false

console.log([] instanceof Array);                    // true
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
```

可以看到，instanceof 只能正确判断引用数据类型，而不能判断基本数据类型。instanceof 运算符可以用来测试一个对象在其原型链中是否存在一个构造函数的 prototype 属性。

（3） constructor

```
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log(([]).constructor === Array); // true
console.log((function() {}).constructor === Function); // true
console.log(({}).constructor === Object); // true
```

constructor 有两个作用，一是判断数据的类型，二是对象实例通过 constrcutor 对象访问它的构造函数。需要注意，如果创建一个对象来改变它的原型，constructor 就不能用来判断数据类型了：

```
function Fn(){};

Fn.prototype = new Array();

var f = new Fn();

console.log(f.constructor===Fn);    // false
console.log(f.constructor===Array); // true
```

（4）Object.prototype.toString.call()

Object.prototype.toString.call() 使用 Object 对象的原型方法 toString 来判断数据类型：

```
var a = Object.prototype.toString;

console.log(a.call(2));
console.log(a.call(true));
console.log(a.call('str'));
console.log(a.call([]));
console.log(a.call(function(){}));
console.log(a.call({}));
console.log(a.call(undefined));
console.log(a.call(null));
```

同样是检测对象 obj 调用 toString 方法，obj.toString()的结果和 Object.prototype.toString.call(obj)的结果不一样，这是为什么？

这是因为 toString 是 Object 的原型方法，而 Array、function 等类型作为 Object 的实例，都重写了 toString 方法。不同的对象类型调用 toString 方法时，根据原型链的知识，调用的是对应的重写之后

的 toString 方法（function 类型返回内容为函数体的字符串，Array 类型返回元素组成的字符串…），而不会去调用 Object 上原型 toString 方法（返回对象的具体类型），所以采用 obj.toString() 不能得到其对象类型，只能将 obj 转换为字符串类型；因此，在想要得到对象的具体类型时，应该调用 Object 原型上的 toString 方法。

3. null 和 undefined 区别

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

undefined 代表的含义是未定义，null 代表的含义是空对象。一般变量声明了但还没有定义的时候会返回 undefined，null 主要用于赋值给一些可能会返回对象的变量，作为初始化。

undefined 在 JavaScript 中不是一个保留字，这意味着可以使用 undefined 来作为一个变量名，但是这样的做法是非常危险的，它会影响对 undefined 值的判断。我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。

当对这两种类型使用 typeof 进行判断时，Null 类型化会返回“object”，这是一个历史遗留的问题。当使用双等号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

4. intanceof 操作符的实现原理及实现

instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

```
function myInstanceof(left, right) {
  // 获取对象的原型
  let proto = Object.getPrototypeOf(left)
  // 获取构造函数的 prototype 对象
  let prototype = right.prototype;

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
    if (proto === prototype) return true;
    // 如果没有找到，就继续从其原型上找，Object.getPrototypeOf方法用来获取指定对象的原型
    proto = Object.getPrototypeOf(proto);
  }
}
```

5. 如何获取安全的 undefined 值？

因为 undefined 是一个标识符，所以可以被当作变量来使用和赋值，但是这样会影响 undefined 的正常判断。表达式 void ___ 没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值。因此可以用 void 0 来获得 undefined。

6. Object.is() 与比较操作符 “===”、“==” 的区别？

使用双等号（==）进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。

使用三等号（===）进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 false。

使用 Object.is 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 -0 和 +0 不再相等，两个 NaN 是相等的。

7. 什么是 JavaScript 中的包装类型？

在 JavaScript 中，基本类型是没有属性和方法的，但是为了便于操作基本类型的值，在调用基本类型的属性或方法时 JavaScript 会在后台隐式地将基本类型的值转换为对象，如：

```
const a = "abc";
a.length; // 3
a.toUpperCase(); // "ABC"
```

在访问'abc'.length 时，JavaScript 将'abc'在后台转换成 String('abc')，然后再访问其 length 属性。

JavaScript 也可以使用 Object 函数显式地将基本类型转换为包装类型：

```
var a = 'abc'
Object(a) // String {"abc"}
```

也可以使用 valueOf 方法将包装类型倒转成基本类型：

```
var a = 'abc'
var b = Object(a)
var c = b.valueOf() // 'abc'
```

看看如下代码会打印出什么：

```
var a = new Boolean( false );
if (!a) {
  console.log( "Oops" ); // never runs
}
```

答案是什么都不会打印，因为虽然包裹的基本类型是 false，但是 false 被包裹成包装类型后就成了对象，所以其非值为 false，所以循环体中的内容不会运行。

8. 为什么会有 BigInt 的提案？

JavaScript 中 Number.MAX_SAFE_INTEGER 表示最大安全数字，计算结果是 9007199254740991，即在这个数范围内不会出现精度丢失（小数除外）。但是一旦超过这个范围，js 就会出现计算不准确的情况，这在大数计算的时候不得不依靠一些第三方库进行解决，因此官方提出了 BigInt 来解决此问题。

9. 如何判断一个对象是空对象

使用 JSON 自带的.stringify 方法来判断：

```
if(Json.stringify(Obj) == '{}' ){
    console.log('空对象');
}
```

使用 ES6 新增的方法 Object.keys()来判断：

```
if(Object.keys(Obj).length < 0){
    console.log('空对象');
}
```

10. const 对象的属性可以修改吗

const 保证的并不是变量的值不能改动，而是变量指向的那个内存地址不能改动。对于基本类型的数据（数值、字符串、布尔值），其值就保存在变量指向的那个内存地址，因此等同于常量。

但对于引用类型的数据（主要是对象和数组）来说，变量指向数据的内存地址，保存的只是一个指针，const 只能保证这个指针是固定不变的，至于它指向的数据结构是不是可变的，就完全不能控制了。

11. 如果 new 一个箭头函数的会怎么样

箭头函数是 ES6 中的提出来的，它没有 prototype，也没有自己的 this 指向，更不可以使用 arguments 参数，所以不能 New 一个箭头函数。

new 操作符的实现步骤如下：

1.创建一个对象

2.将构造函数的作用域赋给新对象（也就是将对象的__proto__属性指向构造函数的 prototype 属性）

3.指向构造函数中的代码，构造函数中的 this 指向该对象（也就是为这个对象添加属性和方法）

4.返回新的对象

所以，上面的第二、三步，箭头函数都是没有办法执行的。

12. 箭头函数的 this 指向哪里？

箭头函数不同于传统 JavaScript 中的函数，箭头函数并没有属于自己的 this，它所谓的 this 是捕获其所在上下文的 this 值，作为自己的 this 值，并且由于没有属于自己的 this，所以是不会被 new 调用的，这个所谓的 this 也不会被改变。

可以用Babel 理解一下箭头函数:

```
// ES6
const obj = {
  getArrow() {
    return () => {
      console.log(this === obj);
    };
  }
}
```

转化后：

```
// ES5, 由 Babel 转译
var obj = {
  getArrow: function getArrow() {
    var _this = this;
    return function () {
      console.log(_this === obj);
    };
  }
};
```

13. 扩展运算符的作用及使用场景

（1）对象扩展运算符

对象的扩展运算符(...)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中。

```
let bar = { a: 1, b: 2 };
let baz = { ...bar }; // { a: 1, b: 2 }
```

上述方法实际上等价于:

```
let bar = { a: 1, b: 2 };
let baz = Object.assign({}, bar); // { a: 1, b: 2 }
```

Object.assign 方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。Object.assign 方法的第一个参数是目标对象，后面的参数都是源对象。(如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性)。

同样，如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

```
let bar = {a: 1, b: 2};
let baz = {...bar, ...{a:2, b: 4}};  // {a: 2, b: 4}
```

利用上述特性就可以很方便的修改对象的部分属性。在 redux 中的 reducer 函数规定必须是一个纯函数，reducer 中的 state 对象要求不能直接修改，可以通过扩展运算符把修改路径的对象都复制一遍，然后产生一个新的对象返回。

需要注意：扩展运算符对对象实例的拷贝属于浅拷贝。

（2）数组扩展运算符

数组的扩展运算符可以将一个数组转为用逗号分隔的参数序列，且每次只能展开一层数组。

```
console.log(...[1, 2, 3])
// 1 2 3
console.log(...[1, [2, 3, 4], 5])
// 1 [2, 3, 4] 5
```

下面是数组的扩展运算符的应用：

将数组转换为参数序列

```
function add(x, y) {
  return x + y;
}
const numbers = [1, 2];
add(...numbers) // 3
```

复制数组

```
const arr1 = [1, 2];
const arr2 = [...arr1];
```

要记住：扩展运算符(…)用于取出参数对象中的所有可遍历属性，拷贝到当前对象之中，这里参数对象是个数组，数组里面的所有对象都是基础数据类型，将所有基础数据类型重新拷贝到新的数组中。

合并数组

如果想在数组内合并数组，可以这样：

```
const arr1 = ['two', 'three'];
const arr2 = ['one', ...arr1, 'four', 'five'];
// ["one", "two", "three", "four", "five"]
```

扩展运算符与解构赋值结合起来，用于生成数组

```
const [first, ...rest] = [1, 2, 3, 4, 5];
first // 1
rest  // [2, 3, 4, 5]
```

需要注意：如果将扩展运算符用于数组赋值，只能放在参数的最后一位，否则会报错。

```
const [...rest, last] = [1, 2, 3, 4, 5];         // 报错
const [first, ...rest, last] = [1, 2, 3, 4, 5];  // 报错
```

将字符串转为真正的数组

```
[...'hello']    // [ "h", "e", "l", "l", "o" ]
```

任何 Iterator 接口的对象，都可以用扩展运算符转为真正的数组

比较常见的应用是可以将某些数据结构转为数组：

```
// arguments对象
function foo() {
  const args = [...arguments];
}
```

用于替换 es5 中的 Array.prototype.slice.call(arguments)写法。

使用 Math 函数获取数组中特定的值

```
const numbers = [9, 4, 7, 1];
Math.min(...numbers); // 1
Math.max(...numbers); // 9
```

14. Proxy 可以实现什么功能？

在 Vue3.0 中通过 Proxy 来替换原本的 Object.defineProperty 来实现数据响应式。

Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作。

```
let p = new Proxy(target, handler)
```

代表需要添加代理的对象，handler 用来自定义对象中的操作，比如可以用来自定义 set 或者 get 函数。

下面来通过 Proxy 来实现一个数据响应式：

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver)
    },
    set(target, property, value, receiver) {
      setBind(value, property)
      return Reflect.set(target, property, value)
    }
  }
  return new Proxy(obj, handler)
}
let obj = { a: 1 }
let p = onWatch(
  obj,
  (v, property) => {
    console.log(`监听到属性${property}改变为${v}`)
  },
  (target, property) => {
    console.log(`'${property}' = ${target[property]}`)
  }
)
p.a = 2 // 监听到属性a改变
p.a // 'a' = 2
```

在上述代码中，通过自定义 set 和 get 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了在对对象任何属性进行读写时发出通知。

当然这是简单版的响应式实现，如果需要实现一个 Vue 中的响应式，需要在 get 中收集依赖，在 set 派发更新，之所以 Vue3.0 要使用 Proxy 替换原本的 API 原因在于 Proxy 无需一层层递归为每个属性添加代理，一次即可完成以上操作，性能上更好，并且原本的实现有一些数据更新不能监听到，但是 Proxy 可以完美监听到任何方式的数据改变，唯一缺陷就是浏览器的兼容性不好。

15. 常用的正则表达式有哪些？

```
// (1) 匹配 16 进制颜色值
var regex = /#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})/g;

// (2) 匹配日期，如 yyyy-mm-dd 格式
var regex = /^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/;

// (3) 匹配 qq 号
var regex = /^[1-9][0-9]{4,10}$/g;

// (4) 手机号码正则
var regex = /^1[34578]\d{9}$/g;

// (5) 用户名正则
var regex = /^[a-zA-Z\$][a-zA-Z0-9_\$]{4,16}$/;
```

16. 对 JSON 的理解

JSON 是一种基于文本的轻量级的数据交换格式。它可以被任何的编程语言读取和作为数据格式来传递。

在项目开发中，使用 JSON 作为前后端数据交换的方式。在前端通过将一个符合 JSON 格式的数据结构序列化为

JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是应该注意的是 JSON 和 js 中的对象不是一回事，JSON 中对象格式更加严格，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

在 js 中提供了两个函数来实现 js 数据结构和 JSON 格式的转换处理，

JSON.stringify 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，可以调用这个函数将数据对象转化为 JSON 格式的字符串。

JSON.parse() 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当从后端接收到 JSON 格式的字符串时，可以通过这个方法来将其解析为一个 js 数据结构，以此来进行数据的访问。

17. JavaScript 脚本延迟加载的方式有哪些？

延迟加载就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

defer 属性：给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性

的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。

async 属性：给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。

动态创建 DOM 方式：动态创建 DOM 标签的方式，可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。

使用 setTimeout 延迟方法：设置一个定时器来延迟加载 js 脚本文件

让 JS 最后加载：将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。

18. 什么是 DOM 和 BOM?

DOM 指的是文档对象模型，它指的是把文档当做一个对象，这个对象主要定义了处理网页内容的方法和接口。

BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的法和接口。BOM 的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。window 对象含有 location 对象、navigator 对象、screen

对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

19. escape、encodeURI、encodeURIComponent 的区别

encodeURI 是对整个 URI 进行转义，将 URI 中的非法字符转换为合法字符，所以对于一些在 URI 中有特殊意义的字符不会进行转义。

encodeURIComponent 是对 URI 的组成部分进行转义，所以一些特殊字符也会得到转义。

escape 和 encodeURI 的作用相同，不过它们对于 unicode 编码为 0xff 之外字符的时候会有区别，escape 是直接在字符的 unicode 编码前加上 %u，而 encodeURI 首先会将字符转换为 UTF-8 的格式，再在每个字节前加上 %。

20. 对 AJAX 的理解，实现一个 AJAX 请求

AJAX 是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的 异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

创建 AJAX 请求的步骤：

创建一个 XMLHttpRequest 对象。

在这个对象上使用 open 方法创建一个 HTTP 请求，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。

在发起请求前，可以为这个对象添加一些信息和监听函数。比如说可以通过 setRequestHeader 方法来为请求添加头信息。还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发 onreadystatechange 事件，可以

通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候就可以通过 response 中的数据来对页面进行更新了。

当对象的属性和监听函数设置完成后，最后调用 sent 方法来向服务器发起请求，可以传入参数作为发送的数据体。

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", url, true);
// 设置状态监听函数
xhr.onreadystatechange = function() {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};
// 设置请求失败时的监听函数
xhr.onerror = function() {
  console.error(this.statusText);
};
// 设置请求头信息
xhr.responseType = "json";
xhr.setRequestHeader("Accept", "application/json");
// 发送 Http 请求
xhr.send(null);
```

使用 Promise 封装 AJAX：

```
// promise 封装实现：
function getJSON(url) {
  // 创建一个 promise 对象
  let promise = new Promise(function(resolve, reject) {
    let xhr = new XMLHttpRequest();
    // 新建一个 http 请求
    xhr.open("GET", url, true);
    // 设置状态的监听函数
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;
      // 当请求成功或失败时，改变 promise 的状态
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    // 设置错误监听函数
    xhr.onerror = function() {
      reject(new Error(this.statusText));
    };
    // 设置响应的数据类型
    xhr.responseType = "json";
    // 设置请求头信息
    xhr.setRequestHeader("Accept", "application/json");
    // 发送 http 请求
    xhr.send(null);
  });
  return promise;
}
```

21. 什么是尾调用，使用尾调用有什么好处？

尾调用指的是函数的最后一步调用另一个函数。代码执行是基于执行栈的，所以当在一个函数里调用另一个函数时，会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。使用尾调用的话，因为已经是函数的最后一步，所以这时可以不必再保留当前的执行上下文，从而节省了内存，这就是尾调用优化。但是 ES6 的尾调用优化只在严格模式下开启，正常模式是无效的。

22. ES6 模块与 CommonJS 模块有什么异同？

ES6 Module 和 CommonJS 模块的区别：

CommonJS 是对模块的浅拷贝，ES6 Module 是对模块的引用，即 ES6 Module 只存只读，不能改变其值，也就是指针指向不能变，类似 const；

import 的接口是 read-only（只读状态），不能修改其变量值。 即不能修改其变量的指针指向，但可以改变变量内部指针指向，可以对 commonJS 对重新赋值（改变指针指向），但是对 ES6 Module 赋值会编译报错。

ES6 Module 和 CommonJS 模块的共同点：

CommonJS 和 ES6 Module 都可以对引入的对象进行赋值，即对对象内部属性的值进行改变。

23. for...in 和 for...of 的区别

for…of 是 ES6 新增的遍历方式，允许遍历一个含有 iterator 接口的数据结构（数组、对象等）并且返回各项的值，和 ES3 中的 for…in 的区别如下

for…of 遍历获取的是对象的键值，for…in 获取的是对象的键名；

for… in 会遍历对象的整个原型链，性能非常差不推荐使用，而 for … of 只遍历当前对象不会遍历原型链；

对于数组的遍历，for…in 会返回数组中所有可枚举的属性(包括原型链上可枚举的属性)，for…of 只返回数组的下标对应的属性值；

总结：for...in 循环主要是为了遍历对象而生，不适用于遍历数组；for...of 循环可以用来遍历数组、类数组对象，字符串、Set、Map 以及 Generator 对象。

24. ajax、axios、fetch 的区别

（1）AJAX

Ajax 即“AsynchronousJavascriptAndXML”（异步 JavaScript 和 XML），是指一种创建交互式网页应用的网页开发技术。它是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。通过在后台与服务器进行少量数据交换，Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。传统的网页（不使用 Ajax）如果需要更新内容，必须重载整个网页页面。其缺点如下：

本身是针对 MVC 编程，不符合前端 MVVM 的浪潮

基于原生 XHR 开发，XHR 本身的架构不清晰

不符合关注分离（Separation of Concerns）的原则

配置和调用方式非常混乱，而且基于事件的异步模型不友好。

（2）Fetch

fetch 号称是 AJAX 的替代品，是在 ES6 出现的，使用了 ES6 中的 promise 对象。Fetch 是基于 promise 设计的。Fetch 的代码结构比

起 ajax 简单多。fetch 不是 ajax 的进一步封装，而是原生 js，没有使用 XMLHttpRequest 对象。

fetch 的优点：

语法简洁，更加语义化

基于标准 Promise 实现，支持 async/await

更加底层，提供的 API 丰富（request, response）

脱离了 XHR，是 ES 规范里新的实现方式

fetch 的缺点：

fetch 只对网络请求报错，对 400，500 都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。

fetch 默认不会带 cookie，需要添加配置项： fetch(url, {credentials: 'include'})

fetch 不支持 abort，不支持超时控制，使用 setTimeout 及 Promise.reject 的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费

fetch 没有办法原生监测请求的进度，而 XHR 可以

（3）Axios

Axios 是一种基于 Promise 封装的 HTTP 客户端，其特点如下：

浏览器端发起 XMLHttpRequests 请求

node 端发起 http 请求

支持 Promise API

监听请求和返回

对请求和返回进行转化

取消请求

自动转换 json 数据

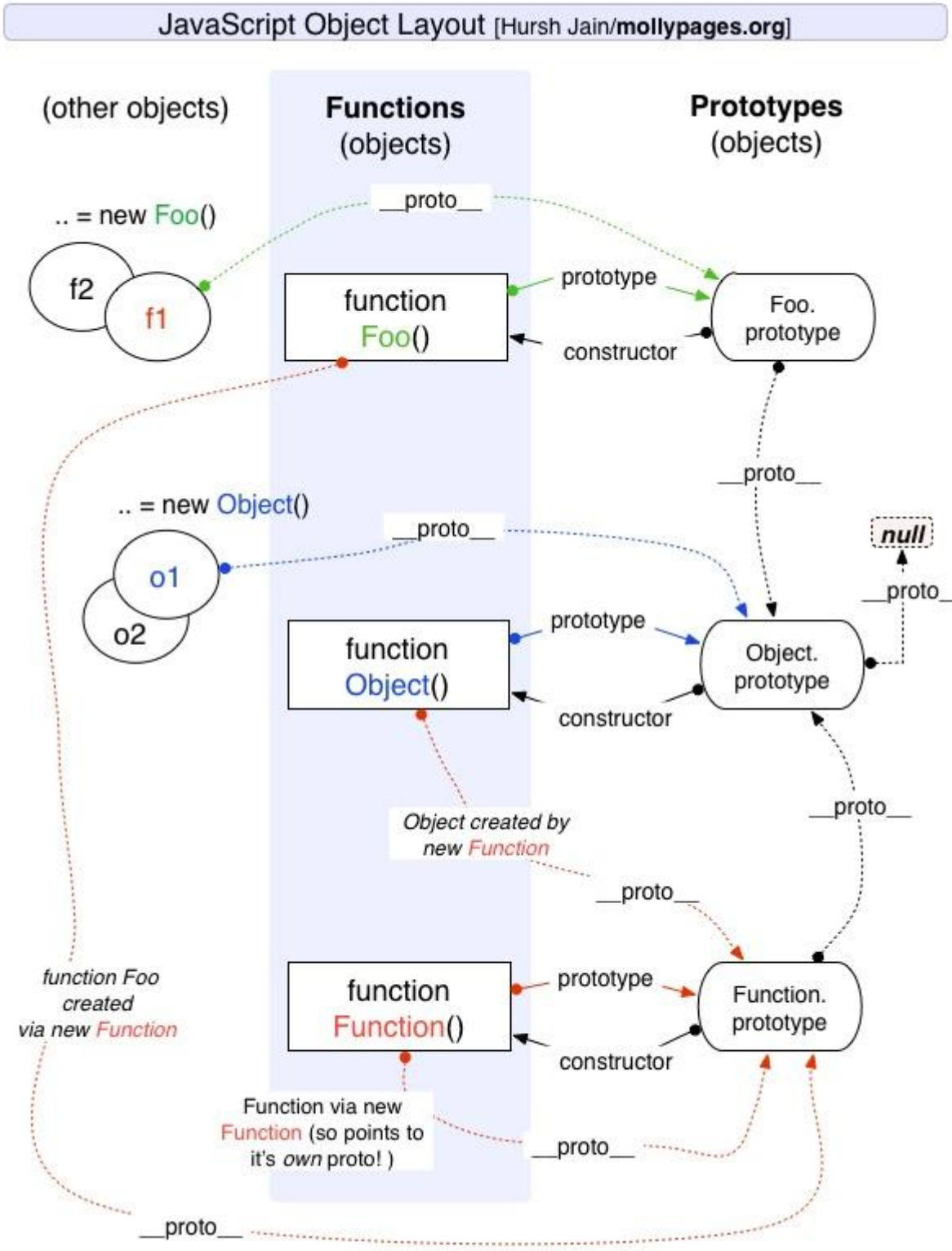
客户端支持抵御 XSRF 攻击

25. 对原型、原型链的理解

在 JavaScript 中是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性，它的属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说不应该能够获取到这个值的，但是现在浏览器中都实现了 __proto__ 属性来访问这个属性，但是最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，可以通过这个方法来获取对象的原型。

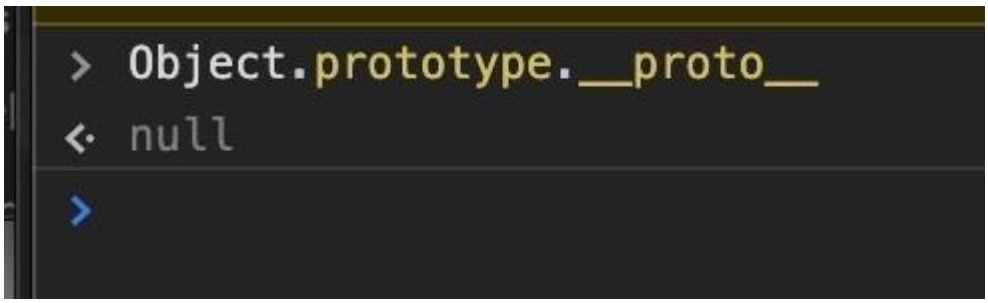
当访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是新建的对象为什么能够使用 toString() 等方法的原因。

特点：JavaScript 对象是通过引用来传递的，创建的每个新对象实体中并没有一份属于自己的原型副本。当修改原型时，与之相关的对象也会继承这一改变。



26. 原型链的终点是什么？如何打印出原型链的终点？

由于 Object 是构造函数，原型链终点 Object.prototype.__proto__，而 Object.prototype.__proto__=== null // true，所以，原型链的终点是 null。原型链上的所有原型都是对象，所有的对象最终都是由 Object 构造的，而 Object.prototype 的下一级是 Object.prototype.__proto__。



27. 对作用域、作用域链的理解

1）全局作用域和函数作用域

（1）全局作用域

最外层函数和最外层函数外面定义的变量拥有全局作用域

所有未定义直接赋值的变量自动声明为全局作用域

所有 window 对象的属性拥有全局作用域

全局作用域有很大的弊端，过多的全局作用域变量会污染全局命名空间，容易引起命名冲突。

（2）函数作用域

函数作用域声明在函数内部的变零，一般只有固定的代码片段可以访问到

作用域是分层的，内层作用域可以访问外层作用域，反之不行

2）块级作用域

使用 ES6 中新增的 let 和 const 指令可以声明块级作用域，块级作用域可以在函数中创建也可以在一个代码块中的创建（由{ }包裹的代码片段）

let 和 const 声明的变量不会有变量提升，也不可以重复声明

在循环中比较适合绑定块级作用域，这样就可以把声明的计数器变量限制在循环内部。

作用域链：

在当前作用域中查找所需变量，但是该作用域没有这个变量，那这个变量就是自由变量。如果在自己作用域找不到该变量就去父级作用域查找，依次向上级作用域查找，直到访问到 window 对象就被终止，这一层层的关系就是作用域链。

作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，可以访问到外层环境的变量和函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当查找一个变量时，如果当前执行环境中没有找到，可以沿着作用域链向后查找。

28. 对 this 对象的理解

this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this 的指向可以通过四种调用模式来判断。

第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。

第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。

第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。

第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

这四种方式，使用构造器调用模式的优先级最高，然后是 apply、call 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

29. call() 和 apply() 的区别？

它们的作用一模一样，区别仅在于传入参数的形式的不同。

apply 接受两个参数，第一个参数指定了函数体内 this 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。

call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被依次传入函数。

30. 异步编程的实现方式？

JavaScript 中的异步机制可以分为以下几种：

回调函数 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

Promise 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

generator 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

async 函数 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

31. 对 Promise 的理解

Promise 是异步编程的一种解决方案，它是一个对象，可以获取异步操作的消息,他的出现大大改善了异步编程的困境,避免了地狱回调，它比传统的解决方案回调函数和事件更合理和更强大。

所谓 Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

（1）Promise 的实例有三个状态:

Pending（进行中）

Resolved（已完成）

Rejected（已拒绝）

当把一件事情交给 promise 时，它的状态就是 Pending，任务完成了状态就变成了 Resolved、没有完成失败了就变成了 Rejected。

（2）Promise 的实例有两个过程：

pending -> fulfilled : Resolved（已完成）

pending -> rejected：Rejected（已拒绝）

注意：一旦从进行状态变成为其他状态就永远不能更改状态了。

Promise 的特点：

对象的状态不受外界影响。promise 对象代表一个异步操作，有三种状态，pending（进行中）、fulfilled（已成功）、rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态，这也是 promise 这个名字的由来——“承诺”；

一旦状态改变就不会再变，任何时候都可以得到这个结果。promise 对象的状态改变，只有两种可能：从 pending 变为 fulfilled，从 pending 变为 rejected。这时就称为 resolved（已定型）。如果改变已经发生了，你再对 promise 对象添加回调函数，也会立即得到这个结果。这与事件（event）完全不同，事件的特点是：如果你错过了它，再去监听是得不到结果的。

Promise 的缺点：

无法取消 Promise，一旦新建它就会立即执行，无法中途取消。

如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。

当处于 pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

总结：

Promise 对象是异步编程的一种解决方案，最早由社区提出。Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是 pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由 pending 转变 resolved 或者 rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。

状态的改变是通过 resolve() 和 reject() 函数来实现的，可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 then 方法，使用这个 then 方法可以为两个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

注意：在构造 Promise 的时候，构造函数内部的代码是立即执行的

32. Promise 解决了什么问题

在工作中经常会碰到这样一个需求，比如我使用 ajax 发一个 A 请求后，成功后拿到数据，需要把数据传给 B 请求；那么需要如下编写代码：

```
let fs = require('fs')
fs.readFile('./a.txt','utf8',function(err,data){
  fs.readFile(data,'utf8',function(err,data){
    fs.readFile(data,'utf8',function(err,data){
      console.log(data)
    })
  })
})
```

上面的代码有如下缺点：

后一个请求需要依赖于前一个请求成功后，将数据往下传递，会导致多个 ajax 请求嵌套的情况，代码不够直观。

如果前后两个请求不需要传递参数的情况下，那么后一个请求也需要前一个请求成功后再执行下一步操作，这种情况下，那么也需要如上编写代码，导致代码不够直观。

Promise 出现之后，代码变成这样：

```
let fs = require('fs')
function read(url){
  return new Promise((resolve,reject)=>{
    fs.readFile(url,'utf8',function(error,data){
      error && reject(error)
      resolve(data)
    })
  })
}
read('./a.txt').then(data=>{
  return read(data)
}).then(data=>{
  return read(data)
}).then(data=>{
  console.log(data)
})
```

这样代码看起了就简洁了很多，解决了地狱回调的问题。

33. 对 async/await 的理解

async/await 其实是 Generator 的语法糖，它能实现的效果都能用 then 链来实现，它是为优化 then 链而开发出来的。从字面上来看，async 是“异步”的简写，await 则为等待，所以很好理解 async 用于申明一个 function 是异步的，而 await 用于等待一个异步方法执行完成。当然语法上强制规定 await 只能出现在 asnyc 函数中，先来看看 async 函数返回了什么：

```
async function testAsy(){
    return 'hello world';
}
let result = testAsy();
console.log(result)
```

所以，async 函数返回的是一个 Promise 对象。async 函数（包含函数语句、函数表达式、Lambda 表达式）会返回一个 Promise 对象，如果在函数中 return 一个直接量，async 会把这个直接量通过 Promise.resolve() 封装成 Promise 对象。

async 函数返回的是一个 Promise 对象，所以在最外层不能用 await 获取其返回值的情况下，当然应该用原来的方式：then() 链来处理这个 Promise 对象，就像这样：

```
async function testAsy(){
    return 'hello world'
}
let result = testAsy()
console.log(result)
result.then(v=>{
    console.log(v)   // hello world
})
```

那如果 async 函数没有返回值，又该如何？很容易想到，它会返回 Promise.resolve(undefined)。

联想一下 Promise 的特点——无等待，所以在没有 await 的情况下执行 async 函数，它会立即执行，返回一个 Promise 对象，并且，绝不会阻塞后面的语句。这和普通返回 Promise 对象的函数并无二致。

注意：Promise.resolve(x) 可以看作是 new Promise(resolve => resolve(x)) 的简写，可以用于快速封装字面量对象或其他对象，将其封装成 Promise 实例。

34. async/await 的优势

单一的 Promise 链并不能发现 async/await 的优势，但是，如果需要处理由多个 Promise 组成的 then 链的时候，优势就能体现出来了（很有意思，Promise 通过 then 链来解决多层回调的问题，现在又用 async/await 来进一步优化它）。

假设一个业务，分多个步骤完成，每个步骤都是异步的，而且依赖于上一个步骤的结果。仍然用 setTimeout 来模拟异步操作：

```
/**
 * 传入参数 n，表示这个函数执行的时间（毫秒）
 * 执行的结果是 n + 200，这个值将用于下一步骤
 */
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);
}
```

现在用 Promise 方式来实现这三个步骤的处理：

```
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();
// c:\var\test>node --harmony_async_await .
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
// doIt: 1507.251ms
```

输出结果 result 是 step3() 的参数 700 + 200 = 900。doIt() 顺序执行了三个步骤，一共用了 300 + 500 + 700 = 1500 毫秒，和 console.time()/console.timeEnd() 计算的结果一致。

如果用 async/await 来实现呢，会是这样：

```
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();
```

结果和之前的 Promise 实现是一样的，但是这个代码看起来是不是清晰得多，几乎跟同步代码一样

35. async/await 对比 Promise 的优势

代码读起来更加同步，Promise 虽然摆脱了回调地狱，但是 then 的链式调用也会带来额外的阅读负担

Promise 传递中间值非常麻烦，而async/await几乎是同步的写法，非常优雅

错误处理友好，async/await 可以用成熟的 try/catch，Promise 的错误捕获非常冗余

调试友好，Promise 的调试很差，由于没有代码块，你不能在一个返回表达式的箭头函数中设置断点，如果你在一个.then 代码块中使用调试器的步进(step-over)功能，调试器并不会进入后续的.then 代码块，因为调试器只能跟踪同步代码的每一步。

36. 对象创建的方式有哪些？

一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js 和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是可以使用函数来进行模拟，从而产生出可复用的对象创建方式，常见的有以下几种：

（1）第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

（2）第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

（3）第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

（4）第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

（5）第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。

（6）第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

37. 对象继承的方式有哪些？

（1）第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

（2）第二种方式是使用借用构造函数的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

（3）第三种方式是组合继承，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

（4）第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 Object.create() 方法就是原型式继承的实现。缺点与原型链方式相同。

（5）第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是自定义类型时。缺点是没有办法实现函数的复用。

（6）第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。

38. 哪些情况会导致内存泄漏

以下四种情况会造成内存的泄漏：

意外的全局变量：由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。

被遗忘的计时器或回调函数：设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。

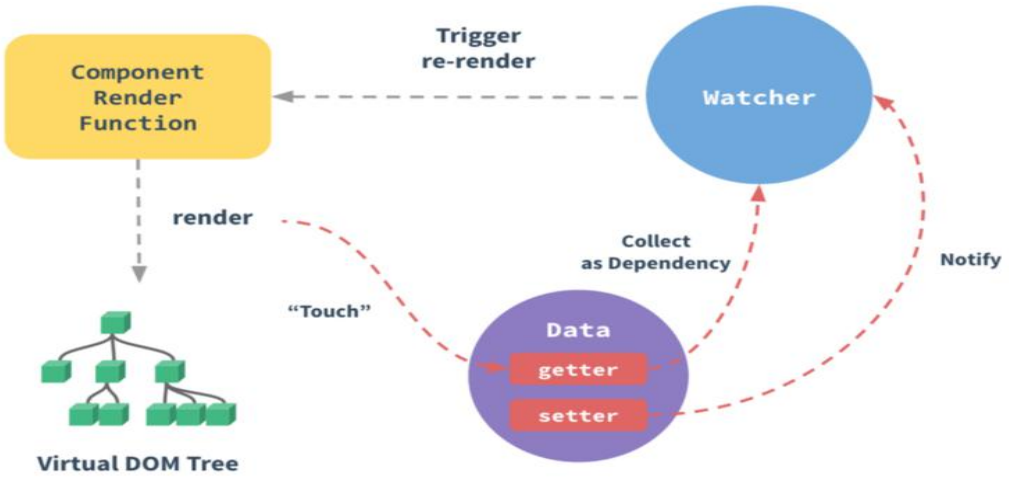
脱离 DOM 的引用：获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。

闭包：不合理的使用闭包，从而导致某些变量一直被留在内存当中。

VUE 部分

1. Vue 的基本原理

当一个 Vue 实例创建时，Vue 会遍历 data 中的属性，用 Object.defineProperty（vue3.0 使用 proxy ）将它们转为 getter/setter，并且在内部追踪相关依赖，在属性被访问和修改时通知变化。 每个组件实例都有相应的 watcher 程序实例，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。



2. 双向数据绑定的原理

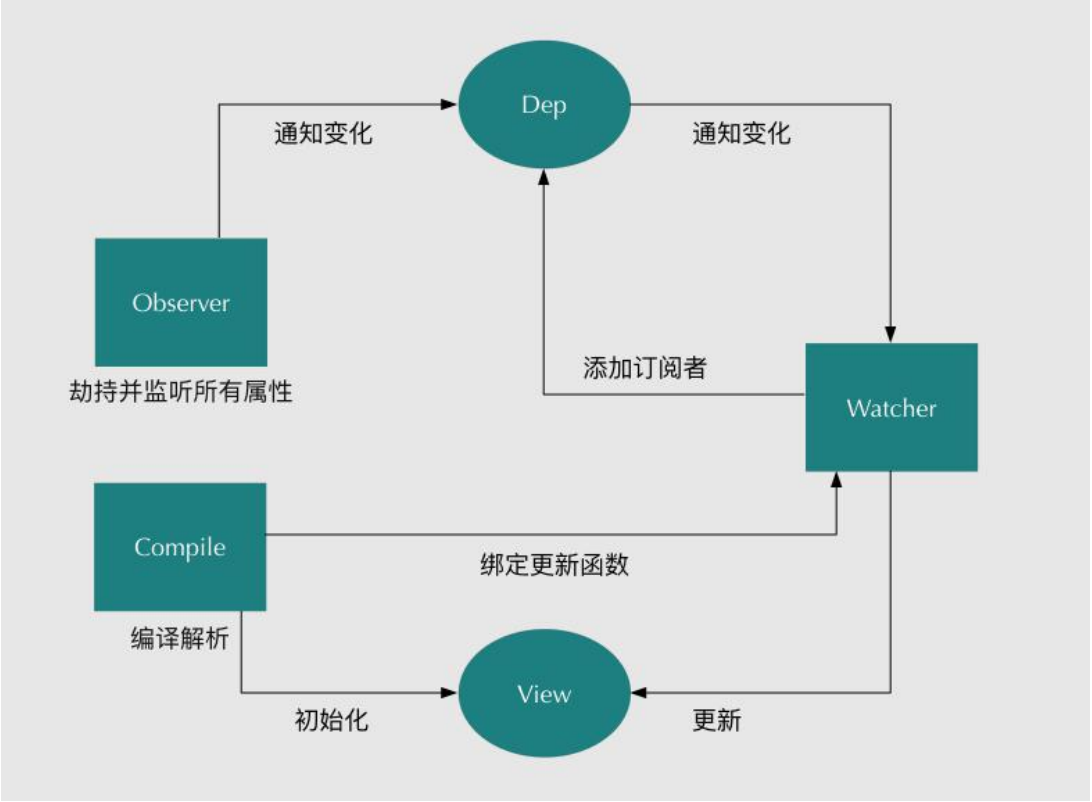
Vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty()来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。主要分为以下几个步骤：

1.需要 observe 的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter 和 getter 这样的话，给这个对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化

2.compile 解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图

3.Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要做的事情是: ①在自身实例化时往属性订阅器(dep)里面添加自己 ②自身必须有一个 update()方法 ③待属性变动 dep.notice()通知时，能调用自身的 update()方法，并触发 Compile 中绑定的回调，则功成身退。

4.MVVM 作为数据绑定的入口，整合 Observer、Compile 和 Watcher 三者，通过 Observer 来监听自己的 model 数据变化，通过 Compile 来解析编译模板指令，最终利用 Watcher 搭起 Observer 和 Compile 之间的通信桥梁,达到数据变化 -> 视图更新;视图交互变化(input) -> 数据 model 变更的双向绑定效果。



3. MVVM、MVC、MVP 的区别

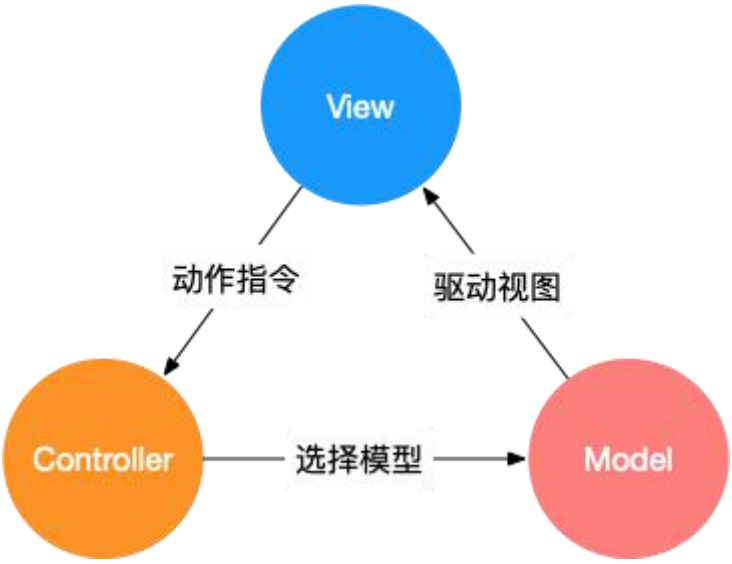
MVC、MVP 和 MVVM 是三种常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构，优化开发效率。

在开发单页面应用时，往往一个路由页面对应了一个脚本文件，所有的页面逻辑都在一个脚本文件里。页面的渲染、数据的获取，对用户事件的响应所有的应用逻辑都混合在一起，这样在开发简单项目时，可能看不出什么问题，如果项目变得复杂，那么整个文件就会变得冗长、混乱，这样对项目开发和后期的项目维护是非常不利的。

（1）MVC

MVC 通过分离 Model、View 和 Controller 的方式来组织代码结构。其中 View 负责页面的显示逻辑，Model 负责存储页面的业务数据，以及对相应数据的操作。并且 View 和 Model 应用了观察者模式，当 Model 层发生改变的时候它会通知有关 View 层更新页面。

Controller 层是 View 层和 Model 层的纽带，它主要负责用户与应用的响应操作，当用户与页面产生交互的时候，Controller 中的事件触发器就开始工作了，通过调用 Model 层，来完成对 Model 的修改，然后 Model 层再去通知 View 层更新。



（2）MVVM

MVVM 分为 Model、View、ViewModel：

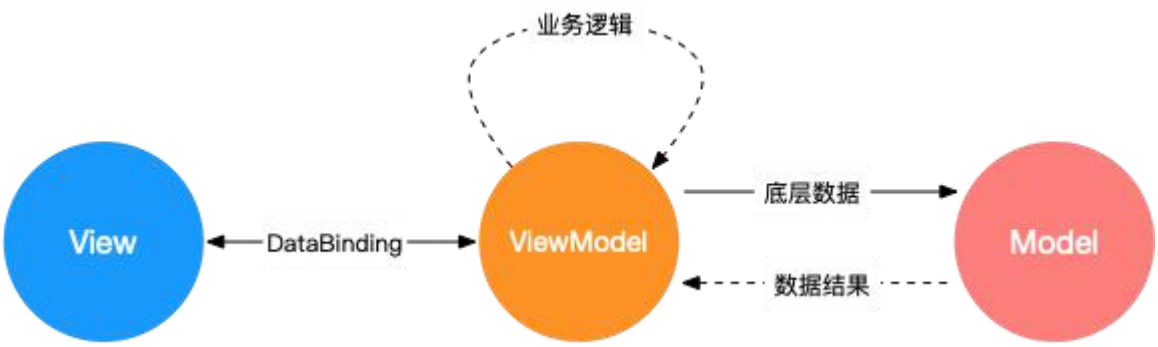
Model 代表数据模型，数据和业务逻辑都在 Model 层中定义；

View 代表 UI 视图，负责数据的展示；

ViewModel 负责监听 Model 中数据的改变并且控制视图的更新，处理用户交互操作；

Model 和 View 并无直接关联，而是通过 ViewModel 来进行联系的，Model 和 ViewModel 之间有着双向数据绑定的联系。因此当 Model 中的数据改变时会触发 View 层的刷新，View 中由于用户交互操作而改变的数据也会在 Model 中同步。

这种模式实现了 Model 和 View 的数据自动同步，因此开发者只需要专注于数据的维护操作即可，而不需要自己操作 DOM。



（3）MVP

MVP 模式与 MVC 唯一不同的在于 Presenter 和 Controller。在 MVC 模式中使用观察者模式，来实现当 Model 层数据发生变化的时候，通知 View 层的更新。这样 View 层和 Model 层耦合在一起，当项目逻辑变得复杂的时候，可能会造成代码的混乱，并且可能会对代码的复用性造成一些问题。MVP 的模式通过使用 Presenter 来实现对 View 层和 Model 层的解耦。MVC 中的 Controller 只知道 Model 的接口，因此它没有办法控制 View 层的更新，MVP 模式中，View 层的接口暴露给了 Presenter 因此可以在 Presenter 中将 Model 的变化和 View 的变化绑定在一起，以此来实现 View 和 Model 的同步更新。这样就实现了对 View 和 Model 的解耦，Presenter 还包含了其他的响应逻辑。

4. slot 是什么？有什么作用？原理是什么？

slot 又名插槽，是 Vue 的内容分发机制，组件内部的模板引擎使用 slot 元素作为承载分发内容的出口。插槽 slot 是子组件的一个模板标签元素，而这一个标签元素是否显示，以及怎么显示是由父组件决定的。slot 又分三类，默认插槽，具名插槽和作用域插槽。

默认插槽：又名匿名插槽，当 slot 没有指定 name 属性值的时候一个默认显示插槽，一个组件内只有有一个匿名插槽。

具名插槽：带有具体名字的插槽，也就是带有 name 属性的 slot，一个组件可以出现多个具名插槽。

作用域插槽：默认插槽、具名插槽的一个变体，可以是匿名插槽，也可以是具名插槽，该插槽的不同点是在子组件渲染作用域插槽时，可以将子组件内部的数据传递给父组件，让父组件根据子组件的传递过来的数据决定如何渲染该插槽。

实现原理：当子组件 vm 实例化时，获取到父组件传入的 slot 标签的内容，存放在 vm.$slot 中，默认插槽为 vm.$slot.default，具名插槽为 vm.$slot.xxx，xxx 为插槽名，当组件执行渲染函数时候，遇到 slot 标签，使用$slot 中的内容进行替换，此时可以为插槽传递数据，若存在数据，则可称该插槽为作用域插槽。

5. $nextTick 原理及作用

Vue 的 nextTick 其本质是对 JavaScript 执行原理 EventLoop 的一种应用。

nextTick 的核心是利用了如 Promise 、MutationObserver、setImmediate、setTimeout 的原生 JavaScript 方法来模拟对应的微/宏任务的实现，本质是为了利用 JavaScript 的这些异步回调任务队列来实现 Vue 框架中自己的异步回调队列。

nextTick 不仅是 Vue 内部的异步队列的调用方法，同时也允许开发者在实际项目中使用这个方法来满足实际应用中对 DOM 更新数据时机的后续逻辑处理

nextTick 是典型的将底层 JavaScript 执行原理应用到具体案例中的示例，引入异步更新队列机制的原因：

如果是同步更新，则多次对一个或多个属性赋值，会频繁触发 UI/DOM 的渲染，可以减少一些无用渲染

同时由于 VirtualDOM 的引入，每一次状态发生变化后，状态变化的信号会发送给组件，组件内部使用 VirtualDOM 进行计算得出需要更新的具体的 DOM 节点，然后对 DOM 进行更新操作，每次更新状态后的渲染过程需要更多的计算，而这种无用功也将浪费更多的性能，所以异步渲染变得更加至关重要

Vue 采用了数据驱动视图的思想，但是在一些情况下，仍然需要操作 DOM。有时候，可能遇到这样的情况，DOM1 的数据发生了变化，而 DOM2 需要从 DOM1 中获取数据，那这时就会发现 DOM2 的视图并没有更新，这时就需要用到了 nextTick 了。

由于 Vue 的 DOM 操作是异步的，所以，在上面的情况中，就要将 DOM2 获取数据的操作写在$nextTick 中。

```
this.$nextTick(() => {
    // 获取数据的操作...
})
```

所以，在以下情况下，会用到 nextTick：

在数据变化后执行的某个操作，而这个操作需要使用随数据变化而变化的 DOM 结构的时候，这个操作就需要方法在 nextTick()的回调函数中。

在 vue 生命周期中，如果在 created()钩子进行 DOM 操作，也一定要放在 nextTick()的回调函数中。

因为在 created()钩子函数中，页面的 DOM 还未渲染，这时候也没办法操作 DOM，所以，此时如果想要操作 DOM，必须将操作的代码放在 nextTick()的回调函数中。

6. Vue 单页应用与多页应用的区别

概念：

SPA 单页面应用（SinglePage Web Application），指只有一个主页面的应用，一开始只需要加载一次 js、css 等相关资源。所有内容都包含在主页面，对每一个功能模块组件化。单页应用跳转，就是切换相关组件，仅仅刷新局部资源。

MPA 多页面应用 （MultiPage Application），指有多个独立页面的应用，每个页面必须重复加载 js、css 等相关资源。多页应用跳转，需要整页资源刷新。

区别：

对比项 \ 模式	SPA	MPA
结构	一个主页面 + 许多模块的组件	许多完整的页面
体验	页面切换快，体验佳；当初次加载文件过多时，需要做相关的调优。	页面切换慢，网速慢的时候，体验尤其不好
资源文件	组件公用的资源只需要加载一次	每个页面都要自己加载公用的资源
适用场景	对体验度和流畅度有较高要求的应用，不利于 SEO（可借助 SSR 优化 SEO）	适用于对 SEO 要求较高的应用
过渡动画	Vue 提供了 transition 的封装组件，容易实现	很难实现
内容更新	相关组件的切换，即局部更新	整体 HTML 的切换，费钱（重复 HTTP 请求）
路由模式	可以使用 hash ，也可以使用 history	普通链接跳转
数据传递	因为单页面，使用全局变量就好（Vuex）	cookie 、localStorage 等缓存方案，URL 参数，调用接口保存等
相关成本	前期开发成本较高，后期维护较为容易	前期开发成本低，后期维护就比较麻烦，因为可能一个功能需要改很多地方

7. Vue 中封装的数组方法有哪些，其如何实现页面更新

在 Vue 中，对响应式处理利用的是 Object.defineProperty 对数据进行拦截，而这个方法并不能监听到数组内部变化，数组长度变化，数组的截取变化等，所以需要对这些操作进行 hack，让 Vue 能监听到其中的变化。

Vue 将被侦听的数组的变更方法进行了包裹，所以它们也将会触发视图更新。这些被包裹过的方法包括：

- `push()`
- `pop()`
- `shift()`
- `unshift()`
- `splice()`
- `sort()`
- `reverse()`

那 Vue 是如何实现让这些数组方法实现元素的实时更新的呢，下面是 Vue 中对这些方法的封装：

```
// 缓存数组原型
const arrayProto = Array.prototype;
// 实现 arrayMethods.__proto__ === Array.prototype
export const arrayMethods = Object.create(arrayProto);
// 需要进行功能拓展的方法
const methodsToPatch = [
  "push",
  "pop",
  "shift",
  "unshift",
  "splice",
  "sort",
  "reverse"
];

/**
 * Intercept mutating methods and emit events
 */
methodsToPatch.forEach(function(method) {
  // 缓存原生数组方法
  const original = arrayProto[method];
  def(arrayMethods, method, function mutator(...args) {
    // 执行并缓存原生数组功能
    const result = original.apply(this, args);
    // 响应式处理
    const ob = this.__ob__;
    let inserted;
    switch (method) {
    // push、unshift会新增索引，所以要手动observer
      case "push":
      case "unshift":
        inserted = args;
        break;
      // splice方法，如果传入了第三个参数，也会有索引加入，也要手动observer。
      case "splice":
        inserted = args.slice(2);
        break;
    }
    //
    if (inserted) ob.observeArray(inserted);// 获取插入的值，并设置响应式监听
    // notify change
    ob.dep.notify();// 通知依赖更新
    // 返回原生数组方法的执行结果
    return result;
  });
});
```

简单来说就是，重写了数组中的那些原生方法，首先获取到这个数组的__ob__，也就是它的 Observer 对象，如果有新的值，就调用 observeArray 继续对新的值观察变化（也就是通过 target__proto__ == arrayMethods 来改变了数组实例的型），然后手动调用 notify，通知渲染 watcher，执行 update。

8. Vue data 中某一个属性的值发生改变后，视图会立即同步执行重新渲染吗？

不会立即同步执行重新渲染。Vue 实现响应式并不是数据发生变化之后 DOM 立即变化，而是按一定的策略进行 DOM 的更新。Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化， Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。

如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环 tick 中，Vue 刷新队列并执行实际（已去重的）工作。

9. 简述 mixin、extends 的覆盖逻辑

（1）mixin 和 extends

mixin 和 extends 均是用于合并、拓展组件的，两者均通过 mergeOptions 方法实现合并。

mixins 接收一个混入对象的数组，其中混入对象可以像正常的实例对象一样包含实例选项，这些选项会被合并到最终的选项中。Mixin 钩子按照传入顺序依次调用，并在调用组件自身的钩子之前被调用。

extends 主要是为了便于扩展单文件组件，接收一个对象或构造函数。

属性名称	合并策略	对应合并函数
data	mixins/extends 只会将自己有的但是组件上没有内容混合到组件上，重复定义默认使用组件上的 如果data里的值是对象，将递归内部对象继续按照该策略合并	mergeDataOrFn, mergeData
provide	同上	mergeDataOrFn, mergeData
props	mixins/extends 只会将自己有的但是组件上没有内容混合到组件上	extend
methods	同上	extend
inject	同上	extend
computed	同上	extend
组件，过滤器，指令属性	同上	extend
el	同上	defaultStrat
propsData	同上	defaultStrat
watch	合并watch监控的回调方法 执行顺序是先mixins/extends里watch定义的回调，然后是组件的回掉	strats.watch
HOOKS 生命周期钩子	同一种钩子的回调函数会被合并成数组 执行顺序是先mixins/extends里定义的钩子函数，然后才是组件里定义的	mergeHook

（2）mergeOptions 的执行过程

规范化选项（normalizeProps、normalizelnject、normalizeDirectives)

对未合并的选项，进行判断

```
if(!child._base) {
    if(child.extends) {
        parent = mergeOptions(parent, child.extends, vm)
    }
    if(child.mixins) {
        for(let i = 0, l = child.mixins.length; i < l; i++){
            parent = mergeOptions(parent, child.mixins[i], vm)
        }
    }
}
```

10. 子组件可以直接改变父组件的数据吗？

子组件不可以直接改变父组件的数据。这样做主要是为了维护父子组件的单向数据流。每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。如果这样做了，Vue 会在浏览器的控制台中发出警告。

Vue 提倡单向数据流，即父级 props 的更新会流向子组件，但是反过来则不行。这是为了防止意外的改变父组件状态，使得应用的数据流变得难以理解，导致数据流混乱。如果破坏了单向数据流，当应用复杂时，debug 的成本会非常高。

只能通过 $emit 派发一个自定义事件，父组件接收到后，由父组件修改。

11. 对 React 和 Vue 的理解，它们的异同

相似之处：

都将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库；

都有自己的构建工具，能让你得到一个根据最佳实践设置的项目模板；

都使用了 Virtual DOM（虚拟 DOM）提高重绘性能；

都有 props 的概念，允许组件间的数据传递；

都鼓励组件化应用，将应用分拆成一个个功能明确的模块，提高复用性。

不同之处 ：

1）数据流

Vue 默认支持数据双向绑定，而 React 一直提倡单向数据流

2）虚拟 DOM

Vue2.x 开始引入"Virtual DOM"，消除了和 React 在这方面的差异，但是在具体的细节还是有各自的特点。

Vue 宣称可以更快地计算出 Virtual DOM 的差异，这是由于它在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。

对于 React 而言，每当应用的状态被改变时，全部子组件都会重新渲染。当然，这可以通过 PureComponent/shouldComponentUpdate 这个生命周期方法来进行控制，但 Vue 将此视为默认的优化。

3）组件化

React 与 Vue 最大的不同是模板的编写。

Vue 鼓励写近似常规 HTML 的模板。写起来很接近标准 HTML 元素，只是多了一些属性。

React 推荐你所有的模板通用 JavaScript 的语法扩展——JSX 书写。

具体来讲：React 中 render 函数是支持闭包特性的，所以 import 的组件在 render 中可以直接调用。但是在 Vue 中，由于模板中使用的数据都必须挂在 this 上进行一次中转，所以 import 一个组件完了之后，还需要在 components 中再声明下。

4）监听数据变化的实现原理不同

Vue 通过 getter/setter 以及一些函数的劫持，能精确知道数据变化，不需要特别的优化就能达到很好的性能

React 默认是通过比较引用的方式进行的，如果不优化（PureComponent/shouldComponentUpdate）可能导致大量不必要的 vDOM 的重新渲染。这是因为 Vue 使用的是可变数据，而 React 更强调数据的不可变。

5）高阶组件

react 可以通过高阶组件（HOC）来扩展，而 Vue 需要通过 mixins 来扩展。

高阶组件就是高阶函数，而 React 的组件本身就是纯粹的函数，所以高阶函数对 React 来说易如反掌。相反 Vue.js 使用 HTML 模板创建视图组件，这时模板无法有效的编译，因此 Vue 不能采用 HOC 来实现。

6）构建工具

两者都有自己的构建工具：

React ==> Create React APP

Vue ==> vue-cli

7）跨平台

React ==> React Native

Vue ==> Weex

12. Vue 的优点

轻量级框架：只关注视图层，是一个构建数据的视图集合，大小只有几十 kb ；

简单易学：国人开发，中文文档，不存在语言障碍 ，易于理解和学习；

双向数据绑定：保留了 angular 的特点，在数据操作方面更为简单；

组件化：保留了 react 的优点，实现了 html 的封装和重用，在构建单页面应用方面有着独特的优势；

视图，数据，结构分离：使数据的更改更为简单，不需要进行逻辑代码的修改，只需要操作数据就能完成相关操作；

虚拟 DOM：dom 操作是非常耗费性能的，不再使用原生的 dom 操作节点，极大解放 dom 操作，但具体操作的还是 dom 不过是换了另一种方式；

运行速度更快：相比较于 react 而言，同样是操作虚拟 dom，就性能而言， vue 存在很大的优势。

13. assets 和 static 的区别

相同点： assets 和 static 两个都是存放静态资源文件。项目中所需要的资源文件图片，字体图标，样式文件等都可以放在这两个文件下，这是相同点

不相同点：assets 中存放的静态资源文件在项目打包时，也就是运行 npm run build 时会将 assets 中放置的静态资源文件进行打包上传，所谓打包简单点可以理解为压缩体积，代码格式化。而压缩后的静态资源文件最终也都会放置在 static 文件中跟着 index.html 一同上传至服务器。static 中放置的静态资源文件就不会要走打包压缩格式化等流程，而是直接进入打包好的目录，直接上传至服务器。因为避免了压缩直接进行上传，在打包时会提高一定的效率，但是 static 中的资源文件由于没有进行压缩等操作，所以文件的体积也就相对于 assets 中打包后的文件提交较大点。在服务器中就会占据更大的空间。

建议： 将项目中 template 需要的样式文件 js 文件等都可以放置在 assets 中，走打包这一流程。减少体积。而项目中引入的第三方的资源文件如 iconfoont.css 等文件可以放置在 static 中，因为这些引入的第三方文件已经经过处理，不再需要处理，直接上传。

14. delete 和 Vue.delete 删除数组的区别

delete 只是被删除的元素变成了 empty/undefined 其他的元素的键值还是不变。

Vue.delete 直接删除了数组 改变了数组的键值。

15. Vue 模版编译原理

vue 中的模板 template 无法被浏览器解析并渲染，因为这不属于浏览器的标准，不是正确的 HTML 语法，所有需要将 template 转化成一个 JavaScript 函数，这样浏览器就可以执行这一个函数并渲染出对应的 HTML 元素，就可以让视图跑起来了，这一个转化的过程，就成为模板编译。模板编译又分三个阶段，解析 parse，优化 optimize，生成 generate，最终生成可执行函数 render。

解析阶段：使用大量的正则表达式对 template 字符串进行解析，将标签、指令、属性等转化为抽象语法树 AST。

优化阶段：遍历 AST，找到其中的一些静态节点并进行标记，方便在页面重渲染的时候进行 diff 比较时，直接跳过这一些静态节点，优化 runtime 的性能。

生成阶段：将最终的 AST 转化为 render 函数字符串。

16. vue 初始化页面闪动问题

使用 vue 开发时，在 vue 初始化之前，由于 div 是不归 vue 管的，所以我们写的代码在还没有解析的情况下会容易出现花屏现象，看到类似于{{message}}的字样，虽然一般情况下这个时间很短暂，但是还是有必要让解决这个问题的。

首先：在 css 里加上以下代码：

```
[v-cloak] {
    display: none;
}
```

如果没有彻底解决问题，则在根元素加上 style="display: none;" :style="{display: 'block'}"

17. MVVM 的优缺点?

优点:

分离视图（View）和模型（Model），降低代码耦合，提高视图或者逻辑的重用性: 比如视图（View）可以独立于 Model 变化和修改，一个 ViewModel 可以绑定不同的"View"上，当 View 变化的时候 Model 不可以不变，当 Model 变化的时候 View 也可以不变。你可以把一些视图逻辑放在一个 ViewModel里面，让很多 view 重用这段视图逻辑

提高可测试性: ViewModel 的存在可以帮助开发者更好地编写测试代码

自动更新 dom: 利用双向绑定,数据更新后视图自动更新,让开发者从繁琐的手动 dom 中解放

缺点:

Bug 很难被调试: 因为使用双向绑定的模式，当你看到界面异常了，有可能是你 View 的代码有 Bug，也可能是 Model 的代码有问题。数据绑定使得一个位置的 Bug 被快速传递到别的位置，要定位原始出问题的地方就变得不那么容易了。另外，数据绑定的声明是指令式地写在 View 的模版当中的，这些内容是没办法去打断点 debug 的

一个大的模块中 model 也会很大，虽然使用方便了也很容易保证了数据的一致性，当时长期持有，不释放内存就造成了花费更多的内存

对于大型的图形应用程序，视图状态较多，ViewModel 的构建和维护的成本都会比较高。

18. v-if 和 v-for 哪个优先级更高？如果同时出现，应如何优化？

v-for 优先于 v-if 被解析，如果同时出现，每次渲染都会先执行循环再判断条件，无论如何循环都不可避免，浪费了性能。

要避免出现这种情况，则在外层嵌套 template，在这一层进行 v-if 判断，然后在内部进行 v-for 循环。如果条件出现在循环内部，可通过计算属性提前过滤掉那些不需要显示的项。

19. 对 Vue 组件化的理解

1.组件是独立和可复用的代码组织单元。组件系统是 Vue 核心特性之一，它使开发者使用小型、独立和通常可复用的组件构建大型应用；

2.组件化开发能大幅提高应用开发效率、测试性、复用性等；

3.组件使用按分类有：页面组件、业务组件、通用组件；

4.vue 的组件是基于配置的，我们通常编写的组件是组件配置而非组件，框架后续会生成其构造函数，它们基于 VueComponent，扩展于 Vue；

5.vue 中常见组件化技术有：属性 prop，自定义事件，插槽等，它们主要用于组件通信、扩展等；6.合理的划分组件，有助于提升应用性能；

6.组件应该是高内聚、低耦合的；

7.遵循单向数据流的原则。

20. 对 vue 设计原则的理解

1.渐进式 JavaScript 框架：与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

2.易用性：vue 提供数据响应式、声明式模板语法和基于配置的组件系统等核心特性。这些使我们只需要关注应用的核心业务即可，只要会写 js、html 和 css 就能轻松编写 vue 应用。

3.灵活性：渐进式框架的最大优点就是灵活性，如果应用足够小，我们可能仅需要 vue 核心特性即可完成功能；随着应用规模不断扩大，我们才可能逐渐引入路由、状态管理、vue-cli 等库和工具，不管是应用体积还是学习难度都是一个逐渐增加的平和曲线。

4.高效性：超快的虚拟 DOM 和 diff算法使我们的应用拥有最佳的性能表现。追求高效的过程还在继续，vue3 中引入 Proxy 对数据响应式改进以及编译器中对于静态内容编译的改进都会让 vue 更加高效。

21. 说一下 Vue 的生命周期

Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模版、挂载 Dom -> 渲染、更新 -> 渲染、卸载 等一系列过程，称这是 Vue 的生命周期。

1.beforeCreate（创建前）：数据观测和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不能访问到 data、computed、watch、methods 上的方法和数据。

2.created（创建后）：实例创建完成，实例上配置的 options 包括 data、computed、watch、methods 等都配置完成，但是此时渲染得节点还未挂载到 DOM，所以不能访问到 $el 属性。

3.beforeMount（挂载前）：在挂载开始之前被调用，相关的 render 函数首次被调用。实例已完成以下的配置：编译模板，把 data 里面的数据和模板生成 html。此时还没有挂载 html 到页面上。

4.mounted（挂载后）：在 el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的 html 内容替换 el 属性指向的 DOM 对象。完成模板中的 html 渲染到 html 页面中。此过程中进行 ajax 交互。

5.beforeUpdate（更新前）：响应式数据更新时调用，此时虽然响应式数据更新了，但是对应的真实 DOM 还没有被渲染。

6.updated（更新后）：在由于数据更改导致的虚拟 DOM 重新渲染和打补丁之后调用。此时 DOM 已经根据响应式数据的变化更新了。调用时，组件 DOM 已经更新，所以可以执行依赖于 DOM 的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。

7.beforeDestroy（销毁前）：实例销毁之前调用。这一步，实例仍然完全可用，this 仍能获取到实例。

8.destroyed（销毁后）：实例销毁后调用，调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务端渲染期间不被调用。

另外还有 keep-alive 独有的生命周期，分别为 activated 和 deactivated。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 activated 钩子函数。

22. Vue 子组件和父组件执行顺序

加载渲染过程：

1.父组件 beforeCreate

2.父组件 created

3.父组件 beforeMount

4.子组件 beforeCreate

5.子组件 created

6.子组件 beforeMount

7.子组件 mounted

8.父组件 mounted

更新过程：

1. 父组件 beforeUpdate

2.子组件 beforeUpdate

3.子组件 updated

4.父组件 updated

销毁过程：

1. 父组件 beforeDestroy

2.子组件 beforeDestroy

3.子组件 destroyed

4.父组件 destoryed

23. created 和 mounted 的区别

created:在模板渲染成 html 前调用，即通常初始化某些属性值，然后再渲染成视图。

mounted:在模板渲染成 html 后调用，通常是初始化页面完成后，再对 html 的 dom 节点进行一些需要的操作。

4. 一般在哪个生命周期请求异步数据

我们可以在钩子函数 created、beforeMount、mounted 中进行调用，因为在这三个钩子函数中，data 已经创建，可以将服务端端返回的数据进行赋值。

推荐在 created 钩子函数中调用异步请求，因为在 created 钩子函数中调用异步请求有以下优点：

能更快获取到服务端数据，减少页面加载时间，用户体验更好；

SSR 不支持 beforeMount 、mounted 钩子函数，放在 created 中有助于一致性。

24. keep-alive 中的生命周期哪些

keep-alive 是 Vue 提供的一个内置组件，用来对组件进行缓存——在组件切换过程中将状态保留在内存中，防止重复渲染 DOM。

如果为一个组件包裹了 keep-alive，那么它会多出两个生命周期：deactivated、activated。同时，beforeDestroy 和 destroyed 就不会再被触发了，因为组件不会被真正销毁。

当组件被换掉时，会被缓存到内存中、触发 deactivated 生命周期；当组件被切回来时，再去缓存里找这个组件、触发 activated 钩子函数。

25. 路由的 hash 和 history 模式的区别

Vue-Router 有两种模式：hash 模式和 history 模式。默认的路由模式是 hash 模式。

1. hash 模式

简介： hash 模式是开发中默认的模式，它的 URL 带着一个#，例如：http://www.abc.com/#/vue，它的 hash 值就是#/vue。

特点：hash 值会出现在 URL 里面，但是不会出现在 HTTP 请求中，对后端完全没有影响。所以改变 hash 值，不会重新加载页面。这种模式的浏览器支持度很好，低版本的 IE 浏览器也支持这种模式。hash 路由被称为是前端路由，已经成为 SPA（单页面应用）的标配。

原理： hash 模式的主要原理就是 onhashchange()事件：

```
window.onhashchange = function(event){
  console.log(event.oldURL, event.newURL);
  let hash = location.hash.slice(1);
}
```

使用 onhashchange()事件的好处就是，在页面的 hash 值发生变化时，无需向后端发起请求，window 就可以监听事件的改变，并按规则加载相应的代码。除此之外，hash 值变化对应的 URL 都会被浏览器记

录下来，这样浏览器就能实现页面的前进和后退。虽然是没有请求后端服务器，但是页面的 hash 值和对应的 URL 关联起来了。

2. history 模式

简介： history 模式的 URL 中没有#，它使用的是传统的路由分发模式，即用户在输入一个 URL 时，服务器会接收这个请求，并解析这个 URL，然后做出相应的逻辑处理。

特点： 当使用 history 模式时，URL 就像这样：http://abc.com/user/id。相比 hash 模式更加好看。但是，history 模式需要后台配置支持。如果后台没有正确配置，访问时会返回 404。

API： history api 可以分为两大部分，切换历史状态和修改历史状态：

修改历史状态：包括了 HTML5 History Interface 中新增的 pushState() 和 replaceState() 方法，这两个方法应用于浏览器的历史记录栈，提供了对历史记录进行修改的功能。只是当他们进行修改时，虽然修改了 url，但浏览器不会立即向后端发送请求。如果要做到改变 url 但又不刷新页面的效果，就需要前端用上这两个 API。

切换历史状态： 包括 forward()、back()、go()三个方法，对应浏览器的前进，后退，跳转操作。

虽然 history 模式丢弃了丑陋的#。但是，它也有自己的缺点，就是在刷新页面的时候，如果没有相应的路由或资源，就会刷出 404 来。

如果想要切换到 history 模式，就要进行以下配置（后端也要进行配置）：

```
const router = new VueRouter({
  mode: 'history',
  routes: [...]
})
```

3. 两种模式对比

调用 history.pushState() 相比于直接修改 hash，存在以下优势:

pushState() 设置的新 URL 可以是与当前 URL 同源的任意 URL；而 hash 只可修改 # 后面的部分，因此只能设置与当前 URL 同文档的 URL；

pushState() 设置的新 URL 可以与当前 URL 一模一样，这样也会把记录添加到栈中；而 hash 设置的新值必须与原来不一样才会触发动作将记录添加到栈中；

pushState() 通过 stateObject 参数可以添加任意类型的数据到记录中；而 hash 只可添加短字符串；

pushState() 可额外设置 title 属性供后续使用。

hash 模式下，仅 hash 符号之前的 url 会被包含在请求中，后端如果没有做到对路由的全覆盖，也不会返回 404 错误；history 模式下，前端的 url 必须和实际向后端发起请求的 url 一致，如果没有对用的路由处理，将返回 404 错误。

hash 模式和 history 模式都有各自的优势和缺陷，还是要根据实际情况选择性的使用。

26. Vue-router 跳转和 location.href 有什么区别

使用 location.href= /url 来跳转，简单方便，但是刷新了页面；

使用 history.pushState(/url) ，无刷新页面，静态跳转；

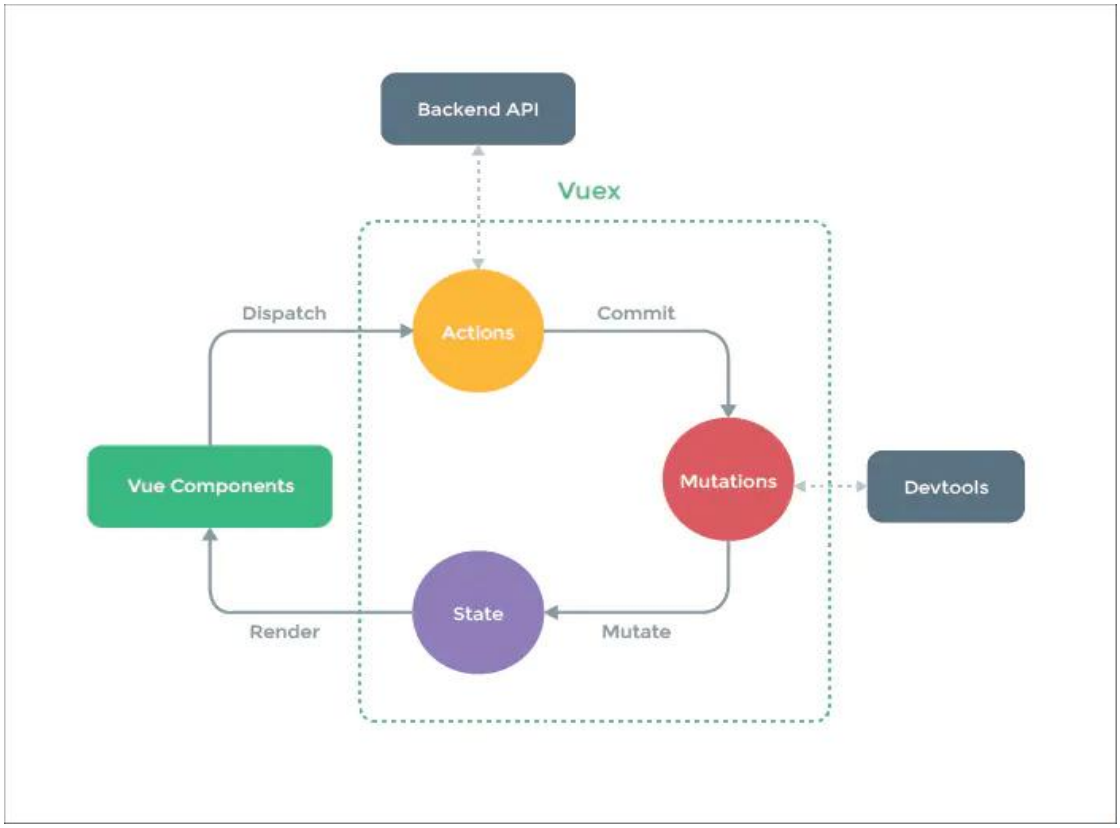
引进 router ，然后使用 router.push(/url) 来跳转，使用了 diff 算法，实现了按需加载，减少了 dom 的消耗。其实使用 router 跳转和使用 history.pushState() 没什么差别的，因为 vue-router 就是用了 history.pushState() ，尤其是在 history 模式下。

27. Vuex 的原理

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。“store” 基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。

Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。

改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样可以方便地跟踪每一个状态的变化。



Vuex 为 Vue Components 建立起了一个完整的生态圈，包括开发中的 API 调用一环。

（1）核心流程中的主要功能：

Vue Components 是 vue 组件，组件会触发（dispatch）一些事件或动作，也就是图中的 Actions;

在组件中发出的动作，肯定是想获取或者改变数据的，但是在 vuex 中，数据是集中管理的，不能直接去更改数据，所以会把这个动作提交（Commit）到 Mutations 中;

然后 Mutations 就去改变（Mutate）State 中的数据;

当 State 中的数据被改变之后，就会重新渲染（Render）到 Vue Components 中去，组件展示更新后的数据，完成一个流程。

（2）各模块在核心流程中的主要功能：

Vue Components： Vue 组件。HTML 页面上，负责接收用户操作等交互行为，执行 dispatch 方法触发对应 action 进行回应。

dispatch：操作行为触发方法，是唯一能执行 action 的方法。

actions： 操作行为处理模块。负责处理 Vue Components 接收到的所有交互行为。包含同步/异步操作，支持多个同名方法，按照注册的顺序依次触发。向后台 API 请求的操作就在这个模块中进行，包括触发其他 action 以及提交 mutation 的操作。该模块提供了 Promise 的封装，以支持 action 的链式触发。

commit：状态改变提交操作方法。对 mutation 进行提交，是唯一能执行 mutation 的方法。

mutations :状态改变操作方法。是 Vuex 修改 state 的唯一推荐方法，其他修改方式在严格模式下将会报错。该方法只能进行同步操作，且方法名只能全局唯一。操作之中会有一些 hook 暴露出来，以进行 state 的监控等。

state： 页面状态管理容器对象。集中存储 Vuecomponents 中 data 对象的零散数据，全局唯一，以进行统一的状态管理。页面显示所需的数据从该对象中进行读取，利用 Vue 的细粒度数据响应机制来进行高效的状态更新。

getters： state 对象读取方法。图中没有单独列出该模块，应该被包含在了 render 中，Vue Components 通过该方法读取全局 state 对象。

总结：

Vuex 实现了一个单向数据流，在全局拥有一个 State 存放数据，当组件要更改 State 中的数据时，必须通过 Mutation 提交修改信息，Mutation 同时提供了订阅者模式供外部插件调用获取 State 数据的更新。而当所有异步操作(常见于调用后端接口异步获取更新数据)或批量的同步操作需要走 Action ，但 Action 也是无法直接修改 State 的，还是需要通过 Mutation 来修改 State 的数据。最后，根据 State 的变化，渲染到视图上。

28. Vuex 和 localStorage 的区别

（1）最重要的区别

vuex 存储在内存中

localstorage 则以文件的方式存储在本地，只能存储字符串类型的数据，存储对象需要 JSON 的 stringify 和 parse 方法进行处理。 读取内存比读取硬盘速度要快

（2）应用场景

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。vuex 用于组件之间的传值。

localstorage 是本地存储，是将数据存储到浏览器的方法，一般是在跨页面传递数据时使用 。

Vuex 能做到数据的响应式，localstorage 不能

（3）永久性

刷新页面时 vuex 存储的值会丢失，localstorage 不会。

注意：对于不变的数据确实可以用 localstorage 可以代替 vuex，但是当两个组件共用一个数据源（对象或数组）时，如果其中一个组件改变了该数据源，希望另一个组件响应该变化时，localstorage 无法做到，原因就是区别 1。

29. Redux 和 Vuex 有什么区别，它们的共同思想

（1）Redux 和 Vuex 区别

Vuex 改进了 Redux 中的 Action 和 Reducer 函数，以 mutations 变化函数取代 Reducer，无需 switch，只需在对应的 mutation 函数里改变 state 值即可

Vuex 由于 Vue 自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的 State 即可

Vuex 数据流的顺序是：View 调用 store.commit 提交对应的请求到 Store 中对应的 mutation 函数->store 改变（vue 检测到数据变化自动渲染）

通俗点理解就是，vuex 弱化 dispatch，通过 commit 进行 store 状态的一次更变;取消了 action 概念，不必传入特定的 action 形式进行指定变更;弱化 reducer，基于 commit 参数直接对数据进行转变，使得框架更加简易;

（2）共同思想

单—的数据源

变化可以预测

本质上：redux 与 vuex 都是对 mvvm 思想的服务，将数据从视图中抽离的一种方案;

形式上：vuex 借鉴了 redux，将 store 作为全局的数据中心，进行 mode 管理;

30. 为什么要用 Vuex 或者 Redux

由于传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致代码无法维护。

所以需要把组件的共享状态抽取出来，以一个全局单例模式管理。在这种模式下，组件树构成了一个巨大的"视图"，不管在树的哪个位置，任何组件都能获取状态或者触发行为。

另外，通过定义和隔离状态管理中的各种概念并强制遵守一定的规则，代码将会变得更结构化且易维护。

31. Vuex 有哪几种属性？

有五种，分别是 State、 Getter、Mutation 、Action、 Module

state => 基本数据(数据源存放地)

getters => 从基本数据派生出来的数据

mutations => 提交更改数据的方法，同步

actions => 像一个装饰器，包裹 mutations，使之可以异步。

modules => 模块化 Vuex

32. Vuex 和单纯的全局对象有什么区别？

Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。

不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样可以方便地跟踪每一个状态的变化，从而能够实现一些工具帮助更好地了解我们的应用。

33. 为什么 Vuex 的 mutation 中不能做异步操作？

Vuex 中所有的状态更新的唯一途径都是 mutation，异步操作通过 Action 来提交 mutation 实现，这样可以方便地跟踪每一个状态的变化，从而能够实现一些工具帮助更好地了解我们的应用。

每个 mutation 执行完成后都会对应到一个新的状态变更，这样 devtools 就可以打个快照存下来,然后就可以实现 time-travel 了。

如果 mutation 支持异步操作，就没有办法知道状态是何时更新的，无法很好的进行状态的追踪，给调试带来困难。

34. Vue3.0 有什么更新

（1）监测机制的改变

3.0 将带来基于代理 Proxy 的 observer 实现，提供全语言覆盖的反应性跟踪。

消除了 Vue 2 当中基于 Object.defineProperty 的实现所存在的很多限制：

（2）只能监测属性，不能监测对象

检测属性的添加和删除；

检测数组索引和长度的变更；

支持 Map、Set、WeakMap 和 WeakSet。

（3）模板

作用域插槽，2.x 的机制导致作用域插槽变了，父组件会重新渲染，而 3.0 把作用域插槽改成了函数的方式，这样只会影响子组件的重新渲染，提升了渲染的性能。

同时，对于 render 函数的方面，vue3.0 也会进行一系列更改来方便习惯直接使用 api 来生成 vdom 。

（4）对象式的组件声明方式

vue2.x 中的组件是通过声明的方式传入一系列 option，和 TypeScript 的结合需要通过一些装饰器的方式来做，虽然能实现功能，但是比较麻烦。

3.0 修改了组件的声明方式，改成了类式的写法，这样使得和 TypeScript 的结合变得很容易

（5）其它方面的更改

支持自定义渲染器，从而使得 weex 可以通过自定义渲染器的方式来扩展，而不是直接 fork 源码来改的方式。

支持 Fragment（多个根节点）和 Protal（在 dom 其他部分渲染组建内容）组件，针对一些特殊的场景做了处理。

基于 tree shaking 优化，提供了更多的内置功能。

35. defineProperty 和 proxy 的区别

Vue 在实例初始化时遍历 data 中的所有属性，并使用 Object.defineProperty 把这些属性全部转为 getter/setter。这样当追踪数据发生变化时，setter 会被自动调用。

Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。

但是这样做有以下问题：

1.添加或删除对象的属性时，Vue 检测不到。因为添加或删除的对象没有在初始化进行响应式处理，只能通过$set 来调用 Object.defineProperty()处理。

2.无法监控到数组下标和长度的变化。

Vue3 使用 Proxy 来监控数据的变化。Proxy 是 ES6 中提供的功能，其作用为：用于定义基本操作的自定义行为（如属性查找，赋值，枚举，函数调用等）。相对于 Object.defineProperty()，其有以下特点：

1.Proxy 直接代理整个对象而非对象属性，这样只需做一层代理就可以监听同级结构下的所有属性变化，包括新增属性和删除属性。

2.Proxy 可以监听数组的变化。

36. Vue3.0 为什么要用 proxy？

在 Vue2 中， Object.defineProperty 会改变原始数据，而 Proxy 是创建对象的虚拟表示，并提供 set 、get 和 deleteProperty 等处理器，这些处理器可在访问或修改原始对象上的属性时进行拦截，有以下特点：

不需用使用 Vue.$set 或 Vue.$delete 触发响应式。

全方位的数组变化检测，消除了 Vue2 无效的边界情况。

支持 Map，Set，WeakMap 和 WeakSet。

Proxy 实现的响应式原理与 Vue2 的实现原理相同，实现方式大同小异：

get 收集依赖

Set、delete 等触发依赖

对于集合类型，就是对集合对象的方法做一层包装：原方法执行后执行依赖相关的收集或触发逻辑。

37. 虚拟 DOM 的解析过程

虚拟 DOM 的解析过程：

首先对将要插入到文档中的 DOM 树结构进行分析，使用 js 对象将其表示出来，比如一个元素对象，包含 TagName、props 和 Children

这些属性。然后将这个 js 对象树给保存下来，最后再将 DOM 片段插入到文档中。

当页面的状态发生改变，需要对页面的 DOM 的结构进行调整的时候，首先根据变更的状态，重新构建起一棵对象树，然后将这棵新的对象树和旧的对象树进行比较，记录下两棵树的的差异。

最后将记录的有差异的地方应用到真正的 DOM 树中去，这样视图就更新了。

38. DIFF 算法的原理

在新老虚拟 DOM 对比时：

首先，对比节点本身，判断是否为同一节点，如果不为相同节点，则删除该节点重新创建节点进行替换

如果为相同节点，进行 patchVnode，判断如何对该节点的子节点进行处理，先判断一方有子节点一方没有子节点的情况(如果新的 children 没有子节点，将旧的子节点移除)

比较如果都有子节点，则进行 updateChildren，判断如何对这些新老节点的子节点进行操作（diff 核心）。

匹配时，找到相同的子节点，递归比较子节点

在 diff 中，只对同层的子节点进行比较，放弃跨级的节点比较，使得时间复杂从 O(n^3)降低值 O(n)，也就是说，只有当新旧 children 都为多个子节点时才需要用核心的 Diff 算法进行同层级比较。

39. Vue 中 key 的作用

vue 中 key 值的作用可以分为两种情况来考虑：

第一种情况是 v-if 中使用 key。由于 Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。因此当使用 v-if 来实现元素切换的时候，如果切换前后含有相同类型的元素，那么这个元素就会被复用。如果是相同的 input 元素，那么切换前后用户的输入不会被清除掉，这样是不符合需求的。因此可以通过使用 key 来唯一的标识一个元素，这个情况下，使用 key 的元素不会被复用。这个时候 key 的作用是用来标识一个独立的元素。

第二种情况是 v-for 中使用 key。用 v-for 更新已渲染过的元素列表时，它默认使用“就地复用”的策略。如果数据项的顺序发生了改变，Vue 不会移动 DOM 元素来匹配数据项的顺序，而是简单复用此处的每个元素。因此通过为每个列表项提供一个 key 值，来以便 Vue 跟踪元素的身份，从而高效的实现复用。这个时候 key 的作用是为了高效的更新渲染虚拟 DOM。

key 是为 Vue 中 vnode 的唯一标记，通过这个 key，diff 操作可以更准确、更快速

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 a.key === b.key 对比中可以避免就地复用的情况。所以会更加准确。

更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快

React 部分

1. React 的事件和普通的 HTML 事件有什么不同？

区别：

对于事件名称命名方式，原生事件为全小写，react 事件采用小驼峰；

对于事件函数处理语法，原生事件为字符串，react 事件为函数；

react 事件不能采用 return false 的方式来阻止浏览器的默认行为，而必须要地明确地调用 preventDefault()来阻止默认行为。

合成事件是 react 模拟原生 DOM 事件所有能力的一个事件对象，其优点如下：

兼容所有浏览器，更好的跨平台；

将事件统一存放在一个数组，避免频繁的新增与删除（垃圾回收）。

方便 react 统一管理和事务机制。

事件的执行顺序为原生事件先执行，合成事件后执行，合成事件会冒泡绑定到 document 上，所以尽量避免原生事件与合成事件混用，如果原生事件阻止冒泡，可能会导致合成事件不执行，因为需要冒泡到 document 上合成事件才会执行。

2. React 组件中怎么做事件代理？它的原理是什么？

React 基于 Virtual DOM 实现了一个 SyntheticEvent 层（合成事件层），定义的事件处理器会接收到一个合成事件对象的实例，它符合 W3C 标准，且与原生的浏览器事件拥有同样的接口，支持冒泡机制，所有的事件都自动绑定在最外层上。

在 React 底层，主要对合成事件做了两件事：

事件委派：React 会把所有的事件绑定到结构的最外层，使用统一的事件监听器，这个事件监听器上维持了一个映射来保存所有组件内部事件监听和处理函数。

自动绑定：React 组件中，每个方法的上下文都会指向该组件的实例，即自动绑定 this 为当前组件。

3. React 高阶组件、Render props、hooks 有什么区别，为什么要不断迭代

这三者是目前 react 解决代码复用的主要方式：

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。

render props 是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术，更具体的说，render prop 是一个用于告知组件需要渲染什么内容的函数 prop。

通常，render props 和高阶组件只渲染一个子节点。让 Hook 来服务这个使用场景更加简单。这两种模式仍有用武之地，（例如，一个虚拟滚动条组件或许会有一个 renderltem 属性，或是一个可见的容器组件或许会有它自己的 DOM 结构）。但在大部分场景下，Hook 足够了，并且能够帮助减少嵌套。

（1）HOC

官方解释：

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。

简言之，HOC 是一种组件的设计模式，HOC 接受一个组件和额外的参数（如果需要），返回一个新的组件。HOC 是纯函数，没有副作用。

```
// hoc的定义
function withSubscription(WrappedComponent, selectData) {
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        data: selectData(DataSource, props)
      };
    }
    // 一些通用的逻辑处理
    render() {
      // ... 并使用新数据渲染被包装的组件!
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };

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

HOC 的优缺点：

优点： 逻辑服用、不影响被包裹组件的内部逻辑。

缺点： hoc 传递给被包裹组件的 props 容易和被包裹后的组件重名，进而被覆盖

（2）Render props

官方解释：

"render prop"是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术

具有 render prop 的组件接受一个返回 React 元素的函数，将 render 的渲染逻辑注入到组件内部。在这里，"render"的命名可以是任何其他有效的标识符。

```
// DataProvider组件内部的渲染逻辑如下
class DataProvider extends React.Components {
     state = {
      name: 'Tom'
    }

    render() {
      return (
          <div>
            <p>共享数据组件自己内部的渲染逻辑</p>
            { this.props.render(this.state) }
          </div>
      );
    }
}

// 调用方式
<DataProvider render={data => (
  <h1>Hello {data.name}</h1>
)}/>
```

由此可以看到，render props 的优缺点也很明显：

优点：数据共享、代码复用，将组件内的 state 作为 props 传递给调用者，将渲染逻辑交给调用者。

缺点：无法在 return 语句外访问数据、嵌套写法不够优雅

（3）Hooks

官方解释：

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。通过自定义 hook，可以复用代码逻辑。

```
// 自定义一个获取订阅数据的hook
function useSubscription() {
  const data = DataSource.getComments();
  return [data];
}
//
function CommentList(props) {
  const {data} = props;
  const [subData] = useSubscription();
    ...
}
// 使用
<CommentList data='hello' />
```

以上可以看出，hook 解决了 hoc 的 prop 覆盖的问题，同时使用的方式解决了 render props 的嵌套地狱的问题。hook 的优点如下：

使用直观；

解决 hoc 的 prop 重名问题；

解决 render props 因共享数据 而出现嵌套地狱的问题；

能在 return 之外使用数据的问题。

需要注意的是：hook 只能在组件顶层使用，不可在分支语句中使用。

总结：

Hoc、render props 和 hook 都是为了解决代码复用的问题，但是 hoc 和 render props 都有特定的使用场景和明显的缺点。hook 是 react16.8 更新的新的 API，让组件逻辑复用更简洁明了，同时也解决了 hoc 和 render props 的一些缺点。

4. Component, Element, Instance 之间有什么区别和联系？

元素：一个元素 element 是一个普通对象(plain object)，描述了对于一个 DOM 节点或者其他组件 component，你想让它在屏幕上呈现成什么样子。元素 element 可以在它的属性 props 中包含其他元素(译注:用于形成元素树)。创建一个 React 元素 element 成本很低。元素 element 创建之后是不可变的。

组件：一个组件 component 可以通过多种方式声明。可以是带有一个 render()方法的类，简单点也可以定义为一个函数。这两种情况下，它都把属性 props 作为输入，把返回的一棵元素树作为输出。

实例：一个实例 instance 是你在所写的组件类 component class 中使用关键字 this 所指向的东西(译注:组件实例)。它用来存储本地状态和响应生命周期事件很有用。

函数式组件(Functional component)根本没有实例 instance。类组件(Class component)有实例 instance，但是永远也不需要直接创建一个组件的实例，因为 React 帮我们做了这些。

5. React.createClass 和 extends Component 的区别有哪些？

React.createClass 和 extends Component 的 bai 区别主要在于：

（1）语法区别

createClass 本质上是一个工厂函数，extends 的方式更加接近最新的 ES6 规范的 class 写法。两种方式在语法上的差别主要体现在方法的定义和静态属性的声明上。

createClass 方式的方法定义使用逗号，隔开，因为 creatClass 本质上是一个函数，传递给它的是一个 Object；而 class 的方式定义方法时务必谨记不要使用逗号隔开，这是 ES6 class 的语法规范。

（2）propType 和 getDefaultProps

React.createClass：通过 proTypes 对象和 getDefaultProps()方法来设置和获取 props.

React.Component：通过设置两个属性 propTypes 和 defaultProps

（3）状态的区别

React.createClass：通过 getInitialState()方法返回一个包含初始值的对象

React.Component：通过 constructor 设置初始状态

（4）this 区别

React.createClass：会正确绑定 this

React.Component：由于使用了 ES6，这里会有些微不同，属性并不会自动绑定到 React 类的实例上。

（5）Mixins

React.createClass：使用 React.createClass 的话，可以在创建组件时添加一个叫做 mixins 的属性，并将可供混合的类的集合以数组的形式赋给 mixins。

如果使用 ES6 的方式来创建组件，那么 React mixins 的特性将不能被使用了。

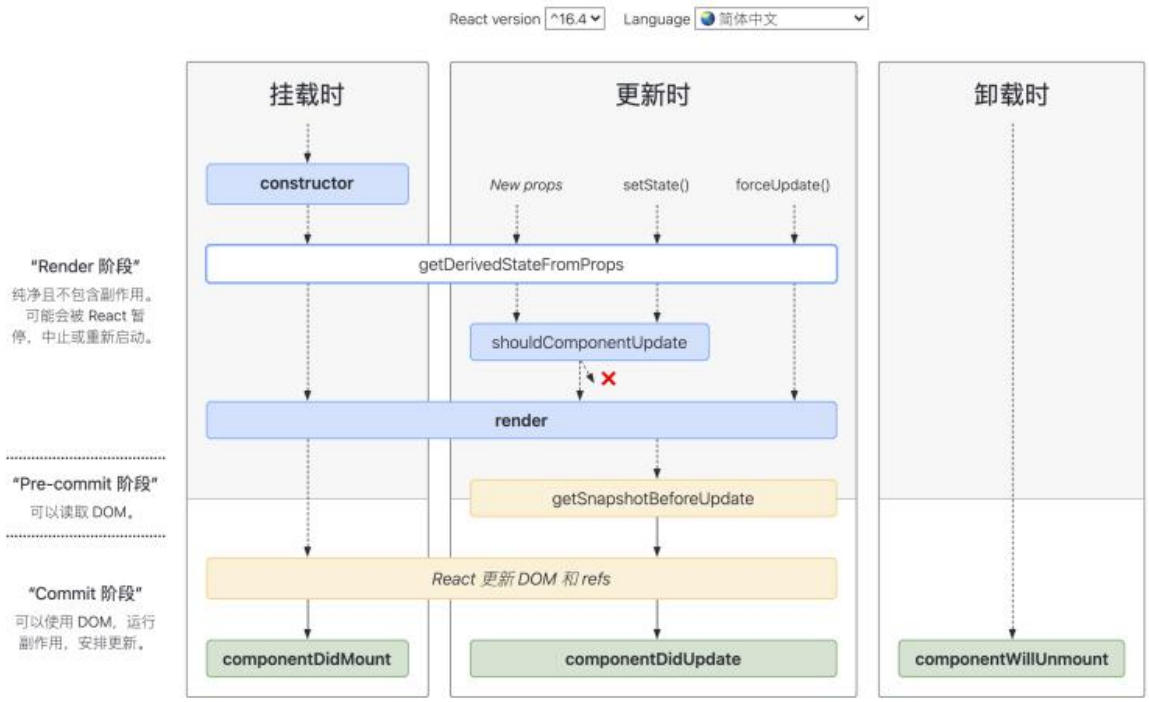
6. React 如何判断什么时候重新渲染组件？

组件状态的改变可以因为 props 的改变，或者直接通过 setState 方法改变。组件获得新的状态，然后 React 决定是否应该重新渲染组件。只要组件的 state 发生变化，React 就会对组件进行重新渲染。这是因为 React 中的 shouldComponentUpdate 方法默认返回 true，这就是导致每次更新都重新渲染的原因。

当 React 将要渲染组件时会执行 shouldComponentUpdate 方法来看它是否返回 true（组件应该更新，也就是重新渲染）。所以需要重写 shouldComponentUpdate 方法让它根据情况返回 true 或者 false 来告诉 React 什么时候重新渲染什么时候跳过重新渲染。

7. React 中可以在 render 访问 refs 吗？为什么？

不可以，render 阶段 DOM 还没有生成，无法获取 DOM。DOM 的获取需要在 pre-commit 阶段和 commit 阶段：



8. React setState 调用之后发生了什么？是同步还是异步？

（1）React 中 setState 后发生了什么

在代码中调用 setState 函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发调和过程(Reconciliation)。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个 UI 界面。

在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

如果在短时间内频繁 setState。React 会将 state 的改变压入栈中，在合适的时机，批量更新 state 和视图，达到提高性能的效果。

（2）setState 是同步还是异步的

假如所有 setState 是同步的，意味着每执行一次 setState 时（有可能一个同步代码中，多次 setState），都重新 vnode diff + dom 修改，这对性能来说是极为不好的。如果是异步，则可以把一个同步代码中的多个 setState 合并成一次组件更新。所以默认是异步的，但是在一些情况下是同步的。

setState 并不是单纯同步/异步的，它的表现会因调用场景的不同而不同。在源码中，通过 isBatchingUpdates 来判断 setState 是先存进 state 队列还是直接更新，如果值为 true 则执行异步操作，为 false 则直接更新。

异步：在 React 可以控制的地方，就为 true，比如在 React 生命周期事件和合成事件中，都会走合并操作，延迟更新的策略。

同步：在 React 无法控制的地方，比如原生事件，具体就是在 addEventListener 、setTimeout、setInterval 等事件中，就只能同步更新。

一般认为，做异步设计是为了性能优化、减少渲染次数：

setState 设计为异步，可以显著的提升性能。如果每次调用 setState 都进行一次更新，那么意味着 render 函数会被频繁调用，界面重新渲染，这样效率是很低的；最好的办法应该是获取到多个更新，之后进行批量更新；

如果同步更新了 state，但是还没有执行 render 函数，那么 state 和 props 不能保持同步。state 和 props 不能保持一致性，会在开发中产生很多的问题；

9. React 组件的 state 和 props 有什么区别？

（1）props

props 是一个从外部传进组件的参数，主要作为就是从父组件向子组件传递数据，它具有可读性和不变性，只能通过外部组件主动传入新的 props 来重新渲染子组件，否则子组件的 props 以及展现形式不会改变。

（2）state

state 的主要作用是用于组件保存、控制以及修改自己的状态，它只能在 constructor 中初始化，它算是组件的私有属性，不可通过外部访问和修改，只能通过组件内部的 this.setState 来修改，修改 state 属性会导致组件的重新渲染。

（3）区别

props 是传递给组件的（类似于函数的形参），而 state 是在组件内被组件自己管理的（类似于在一个函数内声明的变量）。

props 是不可修改的，所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。

state 是在组件中创建的，一般在 constructor 中初始化 state。state 是多变的、可以修改，每次 setState 都异步更新的。

10. React 中的 props 为什么是只读的？

this.props 是组件之间沟通的一个接口，原则上来讲，它只能从父组件流向子组件。React 具有浓重的函数式编程的思想。

提到函数式编程就要提一个概念：纯函数。它有几个特点：

给定相同的输入，总是返回相同的输出。

过程没有副作用。

不依赖外部状态。

this.props 就是汲取了纯函数的思想。props 的不可以变性就保证的相同的输入，页面显示的内容是一样的，并且不会产生副作用

11. React 中怎么检验 props？验证 props 的目的是什么？

React 为我们提供了 PropTypes 以供验证使用。当我们向 Props 传入的数据无效（向 Props 传入的数据类型和验证的数据类型不符）就会在控制台发出警告信息。它可以避免随着应用越来越复杂从而出现的问题。并且，它还可以让程序变得更易读。

```
import PropTypes from 'prop-types';

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

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

当然，如果项目汇中使用了 TypeScript，那么就可以不用 PropTypes 来校验，而使用 TypeScript 定义接口来校验 props。

2. React 废弃了哪些生命周期？为什么？

被废弃的三个函数都是在 render 之前，因为 fber 的出现，很可能因为高优先级任务的出现而打断现有任务导致它们会被执行多次。另外的一个原因则是，React 想约束使用者，好的框架能够让人不得已写出容易维护和扩展的代码，这一点又是从何谈起，可以从新增加以及即将废弃的生命周期分析入手

1) componentWillMount

首先这个函数的功能完全可以使用 componentDidMount 和 constructor 来代替，异步获取的数据的情况上面已经说明了，而如

果抛去异步获取数据，其余的即是初始化而已，这些功能都可以在 constructor 中执行，除此之外，如果在 willMount 中订阅事件，但在服务端这并不会执行 willUnMount 事件，也就是说服务端会导致内存泄漏所以 componentWilIMount 完全可以不使用，但使用者有时候难免因为各 种各样的情况在 componentWilMount 中做一些操作，那么 React 为了约束开发者，干脆就抛掉了这个 API

2) componentWillReceiveProps

在老版本的 React 中，如果组件自身的某个 state 跟其 props 密切相关的话，一直都没有一种很优雅的处理方式去更新 state，而是需要在 componentWilReceiveProps 中判断前后两个 props 是否相同，如果不同再将新的 props 更新到相应的 state 上去。这样做一来会破坏 state 数据的单一数据源，导致组件状态变得不可预测，另一方面也会增加组件的重绘次数。类似的业务需求也有很多，如一个可以横向滑动的列表，当前高亮的 Tab 显然隶属于列表自身的时，根据传入的某个值，直接定位到某个 Tab。为了解决这些问题，React 引入了第一个新的生命周期：getDerivedStateFromProps。它有以下的优点：

●getDSFP 是静态方法，在这里不能使用 this，也就是一个纯函数，开发者不能写出副作用的代码

●开发者只能通过 prevState 而不是 prevProps 来做对比，保证了 state 和 props 之间的简单关系以及不需要处理第一次渲染时 prevProps 为空的情况

●基于第一点，将状态变化（setState）和昂贵操作（tabChange）区分开，更加便于 render 和 commit 阶段操作或者说优化。

3) componentWillUpdate

与 componentWillReceiveProps 类似，许多开发者也会在 componentWillUpdate 中根据 props 的变化去触发一些回调 。 但不论是 componentWilReceiveProps 还 是 componentWilUpdate，都有可能在一次更新中被调用多次，也就是说写在这里的回调函数也有可能会被调用多次，这显然是不可取的。与 componentDidMount 类似， componentDidUpdate 也不存在这样的问题，一次更新中 componentDidUpdate 只会被调用一次，所以将原先写在 componentWillUpdate 中 的 回 调 迁 移 至 componentDidUpdate 就可以解决这个问题。

另外一种情况则是需要获取 DOM 元素状态，但是由于在 fber 中，render 可打断，可能在 wilMount 中获取到的元素状态很可能与实际需要的不同，这个通常可以使用第二个新增的生命函数的解决

getSnapshotBeforeUpdate(prevProps, prevState)

4) getSnapshotBeforeUpdate(prevProps, prevState)

返回的值作为 componentDidUpdate 的第三个参数。与 willMount 不同的是，getSnapshotBeforeUpdate 会在最终确定的 render 执行之前执行，也就是能保证其获取到的元素状态与 didUpdate 中获取到的元素状态相同。官方参考代码：

```jsx
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    // 我们是否在 list 中添加新的 items ？
    // 捕获滚动••位置以便我们稍后调整滚动位置。
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // 如果我们 snapshot 有值，说明我们刚刚添加了新的 items，
    // 调整滚动位置使得这些新 items 不会将旧的 items 推出视图。
    //（这里的 snapshot 是 getSnapshotBeforeUpdate 的返回值）
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

12. React 16.X 中 props 改变后在哪个生命周期中处理

在 getDerivedStateFromProps 中进行处理。

这个生命周期函数是为了替代 componentWillReceiveProps 存在的，所以在需要使用 componentWillReceiveProps 时，就可以考虑使用 getDerivedStateFromProps 来进行替代。

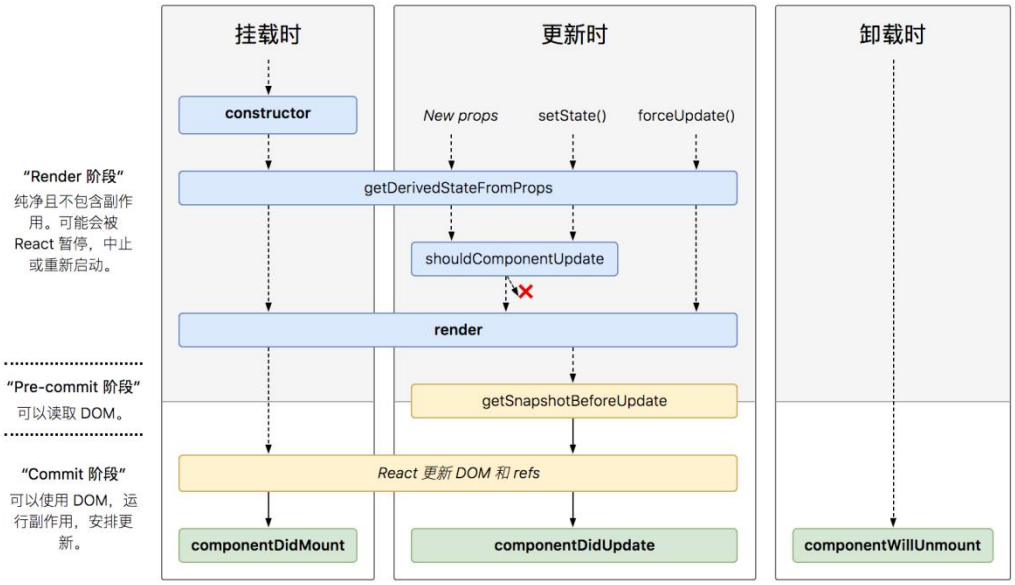
两者的参数是不相同的，而 getDerivedStateFromProps 是一个静态函数，也就是这个函数不能通过 this 访问到 class 的属性，也并不推荐直接访问属性。而是应该通过参数提供的 nextProps 以及 prevState 来进行判断，根据新传入的 props 来映射到 state。

需要注意的是，如果 props 传入的内容不需要影响到你的 state，那么就需要返回一个 null，这个返回值是必须的，所以尽量将其写到函数的末尾：

```
static getDerivedStateFromProps(nextProps, prevState) {
    const {type} = nextProps;
    // 当传入的type发生变化的时候，更新state
    if (type !== prevState.type) {
        return {
            type,
        };
    }
    // 否则，对于state不进行任何操作
    return null;
}
```

13. React 16 中新生命周期有哪些

关于 React16 开始应用的新生命周期：



可以看出，React16 自上而下地对生命周期做了另一种维度的解读：

Render 阶段：用于计算一些必要的状态信息。这个阶段可能会被 React 暂停，这一点和 React16 引入的 Fiber 架构（我们后面会重点讲解）是有关的；

Pre-commit 阶段：所谓“commit”，这里指的是“更新真正的 DOM 节点”这个动作。所谓 Pre-commit，就是说我在这个阶段其实还并没有去更新真实的 DOM，不过 DOM 信息已经是可以读取的了；

Commit 阶段：在这一步，React 会完成真实 DOM 的更新工作。Commit 阶段，我们可以拿到真实 DOM（包括 refs）。

与此同时，新的生命周期在流程方面，仍然遵循“挂载”、“更新”、“卸载”这三个广义的划分方式。它们分别对应到：

挂载过程：

constructor

getDerivedStateFromProps

render

componentDidMount

更新过程：

getDerivedStateFromProps

shouldComponentUpdate

render

getSnapshotBeforeUpdate

componentDidUpdate

卸载过程：

componentWillUnmount

14. React-Router 的实现原理是什么？

客户端路由实现的思想：

基于 hash 的路由：通过监听 hashchange 事件，感知 hash 的变化

改变 hash 可以直接通过 location.hash=xxx

基于 H5 history 路由：

改变 url 可以通过 history.pushState 和 resplaceState 等，会将 URL 压入堆栈，同时能够应用 history.go() 等 API

监听 url 的变化可以通过自定义事件触发实现

react-router 实现的思想：

基于 history 库来实现上述不同的客户端路由实现思想，并且能够保存历史记录等，磨平浏览器差异，上层无感知

通过维护的列表，在每次 URL 发生变化的回收，通过配置的 路由路径，匹配到对应的 Component，并且 render

15. react-router 里的 Link 标签和 a 标签的区别

从最终渲染的 DOM 来看，这两者都是链接，都是 标签，区别是：

<Link>是 react-router 里实现路由跳转的链接，一般配合<Route>使用，react-router 接管了其默认的链接跳转行为，区别于传统的页面跳转，<Link> 的“跳转”行为只会触发相匹配的<Route>对应的页面内容更新，而不会刷新整个页面。

<Link>做了 3 件事情:

有 onclick 那就执行 onclick

click 的时候阻止 a 标签默认事件

根据跳转 href(即是 to)，用 history (web 前端路由两种方式之一，history & hash)跳转，此时只是链接变了，并没有刷新页面而<a>

标签就是普通的超链接了，用于从当前页面跳转到 href 指向的另一个页面(非锚点情况)。

a 标签默认事件禁掉之后做了什么才实现了跳转?

```
let domArr = document.getElementsByTagName('a')
[...domArr].forEach(item=>{
    item.addEventListener('click',function () {
        location.href = this.href
    })
})
```

16. 对 Redux 的理解，主要解决什么问题

React 是视图层框架。Redux 是一个用来管理数据状态和 UI 状态的 JavaScript 应用工具。随着 JavaScript 单页应用（SPA）开发日趋复杂， JavaScript 需要管理比任何时候都要多的 state（状态），Redux 就是降低管理难度的。（Redux 支持 React、Angular、jQuery 甚至纯 JavaScript）。

在 React 中，UI 以组件的形式来搭建，组件之间可以嵌套组合。但 React 中组件间通信的数据流是单向的，顶层组件可以通过 props 属性向下层组件传递数据，而下层组件不能向上层组件传递数据，兄弟组件之间同样不能。这样简单的单向数据流支撑起了 React 中的数据可控性。

当项目越来越大的时候，管理数据的事件或回调函数将越来越多，也将越来越不好管理。管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。如果

这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等。state 的管理在大项目中相当复杂。

Redux 提供了一个叫 store 的统一仓储库，组件通过 dispatch 将 state 直接传入 store，不用通过其他的组件。并且组件通过 subscribe 从 store 获取到 state 的改变。使用了 Redux，所有的组件都可以从 store 中获取到所需的 state，他们也能从 store 获取到 state 的改变。这比组件之间互相传递数据清晰明朗的多。

主要解决的问题：

单纯的 Redux 只是一个状态机，是没有 UI 呈现的，react- redux 作用是将 Redux 的状态机和 React 的 UI 呈现绑定在一起，当你 dispatch action 改变 state 的时候，会自动更新页面。

17. Redux 状态管理器和变量挂载到 window 中有什么区别

两者都是存储数据以供后期使用。但是 Redux 状态更改可回溯——Time travel，数据多了的时候可以很清晰的知道改动在哪里发生，完整的提供了一套状态管理模式。

随着 JavaScript 单页应用开发日趋复杂，JavaScript 需要管理比任何时候都要多的 state （状态）。 这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI 状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。

管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地，可能会引起另一个 view 的变化。

直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。

如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等等。前端开发者正在经受前所未有的复杂性，难道就这么放弃了吗?当然不是。

这里的复杂性很大程度上来自于：我们总是将两个难以理清的概念混淆在一起：变化和异步。 可以称它们为曼妥思和可乐。如果把二者分开，能做的很好，但混到一起，就变得一团糟。一些库如 React 视图在视图层禁止异步和直接操作 DOM 来解决这个问题。美中不足的是，React 依旧把处理 state 中数据的问题留给了你。Redux 就是为了帮你解决这个问题。

18. Redux 和 Vuex 有什么区别，它们的共同思想

（1）Redux 和 Vuex 区别

Vuex 改进了 Redux 中的 Action 和 Reducer 函数，以 mutations 变化函数取代 Reducer，无需 switch，只需在对应的 mutation 函数里改变 state 值即可

Vuex 由于 Vue 自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的 State 即可

Vuex 数据流的顺序是：View 调用 store.commit 提交对应的请求到 Store 中对应的 mutation 函数->store 改变（vue 检测到数据变化自动渲染）

通俗点理解就是，vuex 弱化 dispatch，通过 commit 进行 store 状态的一次更变；取消了 action 概念，不必传入特定的 action 形式

进行指定变更；弱化 reducer，基于 commit 参数直接对数据进行转变，使得框架更加简易;

（2）共同思想

单一的数据源

变化可以预测

本质上： redux 与 vuex 都是对 mvvm 思想的服务，将数据从视图中抽离的一种方案。

19. Redux 中间件是怎么拿到 store 和 action? 然后怎么处理?

redux 中间件本质就是一个函数柯里化。redux applyMiddleware Api 源码中每个 middleware 接受 2 个参数， Store 的 getState 函数和 dispatch 函数，分别获得 store 和 action，最终返回一个函数。该函数会被传入 next 的下一个 middleware 的 dispatch 方法，并返回一个接收 action 的新函数，这个函数可以直接调用 next（action），或者在其他需要的时刻调用，甚至根本不去调用它。调用链中最后一个 middleware 会接受真实的 store 的 dispatch 方法作为 next 参数，并借此结束调用链。所以，middleware 的函数签名是（{ getState，dispatch }）=> next => action。

20. React Hooks 解决了哪些问题？

React Hooks 主要解决了以下问题：

（1）在组件之间复用状态逻辑很难

React 没有提供将可复用性行为“附加”到组件的途径（例如，把组件连接到 store）解决此类问题可以使用 render props 和 高阶组

件。但是这类方案需要重新组织组件结构，这可能会很麻烦，并且会使代码难以理解。由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件会形成“嵌套地狱”。尽管可以在 DevTools 过滤掉它们，但这说明了一个更深层次的问题：React 需要为共享状态逻辑提供更好的原生途径。

可以使用 Hook 从组件中提取状态逻辑，使得这些逻辑可以单独测试并复用。Hook 使我们在无需修改组件结构的情况下复用状态逻辑。这使得在组件间或社区内共享 Hook 变得更便捷。

（2）复杂组件变得难以理解

在组件中，每个生命周期常常包含一些不相关的逻辑。例如，组件常常在 componentDidMount 和 componentDidUpdate 中获取数据。但是，同一个 componentDidMount 中可能也包含很多其它的逻辑，如设置事件监听，而之后需在 componentWillUnmount 中清除。相互关联且需要对照修改的代码被进行了拆分，而完全不相关的代码却在同一个方法中组合在一起。如此很容易产生 bug，并且导致逻辑不一致。

在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。这也给测试带来了一定挑战。同时，这也是很多人将 React 与状态管理库结合使用的原因之一。但是，这往往会引入了很多抽象概念，需要你在不同的文件之间来回切换，使得复用变得更加困难。

为了解决这个问题，Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据），而并非强制按照生命周期划分。你还可以使用 reducer 来管理组件的内部状态，使其更加可预测。

（3）难以理解的 class

除了代码复用和代码管理会遇到困难外，class 是学习 React 的一大屏障。我们必须去理解 JavaScript 中 this 的工作方式，这与其他语言存在巨大差异。还不能忘记绑定事件处理器。没有稳定的语法提案，这些代码非常冗余。大家可以很好地理解 props，state 和自顶向下的数据流，但对 class 却一筹莫展。即便在有经验的 React 开发者之间，对于函数组件与 class 组件的差异也存在分歧，甚至还要区分两种组件的使用场景。

为了解决这些问题，Hook 使你在非 class 的情况下可以使用更多的 React 特性。 从概念上讲，React 组件一直更像是函数。而 Hook 则拥抱了函数，同时也没有牺牲 React 的精神原则。Hook 提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术

21. React Hook 的使用限制有哪些？

React Hooks 的限制主要有两条：

不要在循环、条件或嵌套函数中调用 Hook；

在 React 的函数组件中调用 Hook。

那为什么会有这样的限制呢？Hooks 的设计初衷是为了改进 React 组件的开发模式。在旧有的开发模式下遇到了三个问题。

组件之间难以复用状态逻辑。过去常见的解决方案是高阶组件、render props 及状态管理框架。

复杂的组件变得难以理解。生命周期函数与业务逻辑耦合太深，导致关联部分难以拆分。

人和机器都很容易混淆类。常见的有 this 的问题，但在 React 团队中还有类难以优化的问题，希望在编译优化层面做出一些改进。这

三个问题在一定程度上阻碍了 React 的后续发展，所以为了解决这三个问题，Hooks 基于函数组件开始设计。然而第三个问题决定了 Hooks 只支持函数组件。

那为什么不要在循环、条件或嵌套函数中调用 Hook 呢？因为 Hooks 的设计是基于数组实现。在调用时按顺序加入数组中，如果使用循环、条件或嵌套函数很有可能导致数组取值错位，执行错误的 Hook。当然，实质上 React 的源码里不是数组，是链表。

这些限制会在编码上造成一定程度的心智负担，新手可能会写错，为了避免这样的情况，可以引入 ESLint 的 Hooks 检查插件进行预防。

22. React diff 算法的原理是什么？

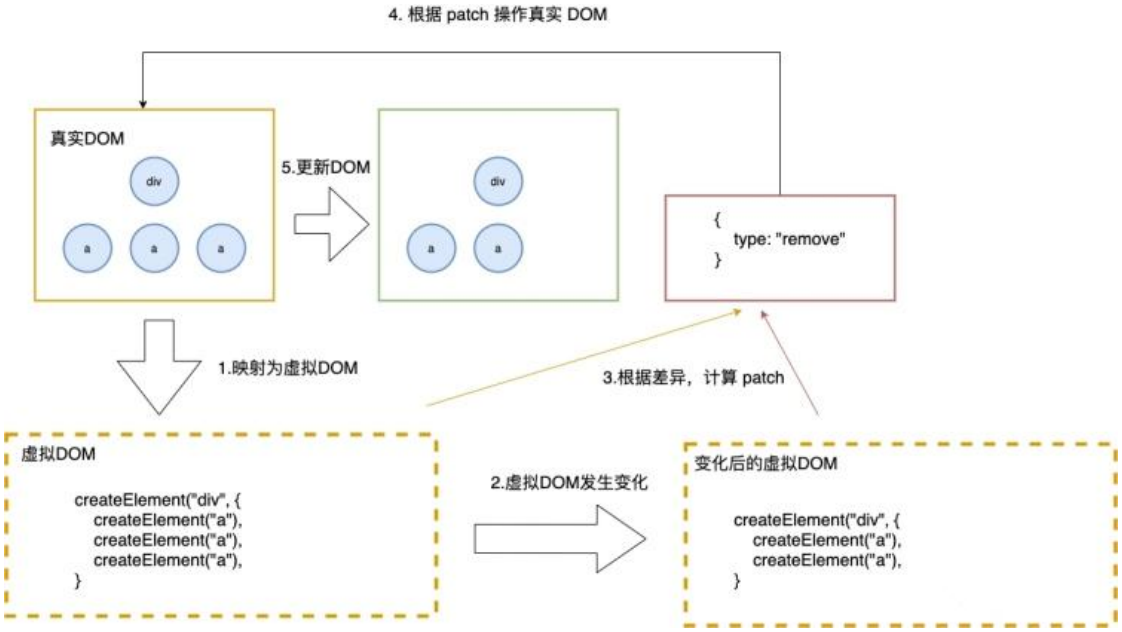
实际上，diff 算法探讨的就是虚拟 DOM 树发生变化后，生成 DOM 树更新补丁的方式。它通过对比新旧两株虚拟 DOM 树的变更差异，将更新补丁作用于真实 DOM，以最小成本完成视图更新。

具体的流程如下：

真实的 DOM 首先会映射为虚拟 DOM；

当虚拟 DOM 发生变化后，就会根据差距计算生成 patch，这个 patch 是一个结构化的数据，内容包含了增加、更新、移除等；

根据 patch 去更新真实的 DOM，反馈到用户的界面上。



一个简单的例子：

```
import React from 'react'
export default class ExampleComponent extends React.Component {
  render() {
    if(this.props.isVisible) {
       return <div className="visible">visbile</div>;
    }
     return <div className="hidden">hidden</div>;
  }
}
```

这里，首先假定 ExampleComponent 可见，然后再改变它的状态，让它不可见 。映射为真实的 DOM 操作是这样的，React 会创建一个 div 节点。

```
<div class="visible">visbile</div>
```

当把 visbile 的值变为 false 时，就会替换 class 属性为 hidden，并重写内部的 innerText 为 hidden。这样一个生成补丁、更新差异的过程统称为 diff 算法。

diff 算法可以总结为三个策略，分别从树、组件及元素三个层面进行复杂度的优化：

策略一：忽略节点跨层级操作场景，提升比对效率。（基于树进行对比）

这一策略需要进行树比对，即对树进行分层比较。树比对的处理手法是非常“暴力”的，即两棵树只对同一层次的节点进行比较，如果发现节点已经不存在了，则该节点及其子节点会被完全删除掉，不会用于进一步的比较，这就提升了比对效率。

策略二：如果组件的 class 一致，则默认为相似的树结构，否则默认为不同的树结构。（基于组件进行对比）

在组件比对的过程中：

如果组件是同一类型则进行树比对；

如果不是则直接放入补丁中。

只要父组件类型不同，就会被重新渲染。这也就是为什么 shouldComponentUpdate、PureComponent 及 React.memo 可以提高性能的原因。

策略三：同一层级的子节点，可以通过标记 key 的方式进行列表对比。（基于节点进行对比）

元素比对主要发生在同层级中，通过标记节点操作生成补丁。节点操作包含了插入、移动、删除等。其中节点重新排序同时涉及插入、移动、删除三个操作，所以效率消耗最大，此时策略三起到了至关重要的作用。通过标记 key 的方式，React 可以直接移动 DOM 节点，降低内耗。

23. React key 是干嘛用的 为什么要加？key 主要是解决哪一类问题的

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。

在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染此外，React 还需要借助 Key 值来判断元素与本地状态的关联关系。

注意事项：

key 值一定要和具体的元素一一对应；

尽量不要用数组的 index 去作为 key；

不要在 render 的时候用随机数或者其他操作给元素加上不稳定的 key，这样造成的性能开销比不加 key 的情况下更糟糕。

24. React 与 Vue 的 diff 算法有何不同？

diff 算法是指生成更新补丁的方式，主要应用于虚拟 DOM 树变化后，更新真实 DOM。所以 diff 算法一定存在这样一个过程：触发更新 → 生成补丁 → 应用补丁。

React 的 diff 算法，触发更新的时机主要在 state 变化与 hooks 调用之后。此时触发虚拟 DOM 树变更遍历，采用了深度优先遍历算法。但传统的遍历方式，效率较低。为了优化效率，使用了分治的方式。将单一节点比对转化为了 3 种类型节点的比对，分别是树、组件及元素，以此提升效率。

树比对：由于网页视图中较少有跨层级节点移动，两株虚拟 DOM 树只对同一层次的节点进行比较。

组件比对：如果组件是同一类型，则进行树比对，如果不是，则直接放入到补丁中。

元素比对：主要发生在同层级中，通过标记节点操作生成补丁，节点操作对应真实的 DOM 剪裁操作。

以上是经典的 React diff 算法内容。自 React 16 起，引入了 Fiber 架构。为了使整个更新过程可随时暂停恢复，节点与树分别采用了 FiberNode 与 FiberTree 进行重构。fiberNode 使用了双链表的结构，可以直接找到兄弟节点与子节点。整个更新过程由 current 与 workInProgress 两株树双缓冲完成。workInProgress 更新完成后，再通过修改 current 相关指针指向新节点。

Vue 的整体 diff 策略与 React 对齐，虽然缺乏时间切片能力，但这并不意味着 Vue 的性能更差，因为在 Vue 3 初期引入过，后期因为收益不高移除掉了。除了高帧率动画，在 Vue 中其他的场景几乎都可以使用防抖和节流去提高响应性能。

25. react 最新版本解决了什么问题，增加了哪些东西

React 16.x 的三大新特性 Time Slicing、Suspense、 hooks

Time Slicing（解决 CPU 速度问题）使得在执行任务的期间可以随时暂停，跑去干别的事情，这个特性使得 react 能在性能极其差的机器跑时，仍然保持有良好的性能

Suspense （解决网络 IO 问题）和 lazy 配合，实现异步加载组件。 能暂停当前组件的渲染， 当完成某件事以后再继续渲染，解决从 react 出生到现在都存在的「异步副作用」的问题，而且解决得非的优雅，使用的是 T 异步但是同步的写法，这是最好的解决异步问题的方式

提供了一个内置函数 componentDidCatch，当有错误发生时，可以友好地展示 fallback 组件；可以捕捉到它的子元素（包括嵌套子元素）抛出的异常；可以复用错误组件。

（1）React16.8

加入 hooks，让 React 函数式组件更加灵活，hooks 之前，React 存在很多问题：

在组件间复用状态逻辑很难

复杂组件变得难以理解，高阶组件和函数组件的嵌套过深。

class 组件的 this 指向问题

难以记忆的生命周期

hooks 很好的解决了上述问题，hooks 提供了很多方法

useState 返回有状态值，以及更新这个状态值的函数

useEffect 接受包含命令式，可能有副作用代码的函数。

useContext 接受上下文对象（从 React.createContext 返回的值）并返回当前上下文值，

useReducer useState 的替代方案。接受类型为 （state，action）=> newState 的 reducer，并返回与 dispatch 方法配对的当前状态。

useCalLback 返回一个回忆的 memoized 版本，该版本仅在其中一个输入发生更改时才会更改。纯函数的输入输出确定性 o useMemo 纯的一个记忆函数 o useRef 返回一个可变的 ref 对象，其 Current 属性被初始化为传递的参数，返回的 ref 对象在组件的整个生命周期内保持不变。

useImperativeMethods 自定义使用 ref 时公开给父组件的实例值

useMutationEffect 更新兄弟组件之前，它在 React 执行其 DOM 改变的同一阶段同步触发

useLayoutEffect DOM 改变后同步触发。使用它来从 DOM 读取布局并同步重新渲染

（2）React16.9

重命名 Unsafe 的生命周期方法。新的 UNSAFE_前缀将有助于在代码 review 和 debug 期间，使这些有问题的字样更突出废弃

javascrip:形式的 URL。以 javascript:开头的 URL 非常容易遭受攻击，造成安全漏洞。

废弃"Factory"组件。 工厂组件会导致 React 变大且变慢。

act（）也支持异步函数，并且你可以在调用它时使用 await。

使用 <React.ProfiLer> 进行性能评估。在较大的应用中追踪性能回归可能会很方便

（3）React16.13.0

支持在渲染期间调用 setState，但仅适用于同一组件

可检测冲突的样式规则并记录警告

废弃 unstable_createPortal，使用 CreatePortal

将组件堆栈添加到其开发警告中，使开发人员能够隔离 bug 并调试其程序，这可以清楚地说明问题所在，并更快地定位和修复错误。

26. 在 React 中页面重新加载时怎样保留数据？

这个问题就设计到了数据持久化，主要的实现方式有以下几种：

Redux:将页面的数据存储在 redux 中,在重新加载页面时,获取 Redux 中的数据;

data.js：使用 webpack 构建的项目，可以建一个文件，data.js，将数据保存 data.js 中，跳转页面后获取;

sessionStorge：在进入选择地址页面之前，componentWillUnMount 的时候，将数据存储到 sessionStorage 中，每次进入页面判断 sessionStorage 中有没有存储的那个值，有，则读取渲染数据；没有，则说明数据是初始化的状态。返回或进入除了选择地址以外的页面，清掉存储的 sessionStorage，保证下次进入是初始化的数据

history API：History API 的 pushState 函数可以给历史记录关联一个任意的可序列化 state,所以可以在路由 push 的时候将当前页面的一些信息存到 state 中，下次返回到这个页面的时候就能从 state 里面取出离开前的数据重新渲染。react-router 直接可以支持。这个方法适合一些需要临时存储的场景。

27. 为什么使用 jsx 的组件中没有看到使用 react 却需要引入 react?

本质上来说 JSX 是 React.createElement(component, props, ...children)方法的语法糖。在 React 17 之前，如果使用了 JSX，其实就是在使用 React， babel 会把组件转换为 CreateElement 形式。在 React 17 之后，就不再需要引入，因为 babel 已经可以帮我们自动引入 react。

28. Redux 中间件是什么？接受几个参数？柯里化函数两端的参数具体是什么？

Redux 的中间件提供的是位于 action 被发起之后，到达 reducer 之前的扩展点，换而言之，原本 view -→> action -> reducer -> store 的数据流加上中间件后变成了 view -> action -> middleware -> reducer -> store ，在这一环节可以做一些"副作用"的操作，如异步请求、打印日志等。

applyMiddleware 源码：

```
export default function applyMiddleware(...middlewares) {
    return createStore => (...args) => {
        // 利用传入的createStore和reducer和创建一个store
        const store = createStore(...args)
        let dispatch = () => {
            throw new Error()
        }
        const middlewareAPI = {
            getState: store.getState,
            dispatch: (...args) => dispatch(...args)
        }
        // 让每个 middleware 带着 middlewareAPI 这个参数分别执行一遍
        const chain = middlewares.map(middleware => middleware(middlewareAPI))
        // 接着 compose 将 chain 中的所有匿名函数，组装成一个新的函数，即新的 dispatch
        dispatch = compose(...chain)(store.dispatch)
        return {
            ...store,
            dispatch
        }
    }
}
```

从 applyMiddleware 中可以看出：

redux 中间件接受一个对象作为参数，对象的参数上有两个字段 dispatch 和 getState，分别代表着 Redux Store 上的两个同名函数。

柯里化函数两端一个是 middewares，一个是 store.dispatch

29. 组件通信的方式有哪些

父组件向子组件通讯：父组件可以向子组件通过传 props 的方式，向子组件进行通讯

子组件向父组件通讯：props+回调的方式，父组件向子组件传递 props 进行通讯，此 props 为作用域为父组件自身的函 数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中

兄弟组件通信：找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信

跨层级通信：Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言，对于跨越多层的全局数据通过 Context 通信再适合不过

发布订阅模式：发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入event 模块进行通信

全局状态管理工具：借助 Redux 或者 Mobx 等全局状态管理工具进行通信,这种工具会维护一个全局状态中心Store,并根据不同的事件产生新的状态

性能优化部分

1. 懒加载的概念

懒加载也叫做延迟加载、按需加载，指的是在长网页中延迟加载图片数据,是一种较好的网页性能优化的方式。在比较长的网页或应用中，如果图片很多，所有的图片都被加载出来，而用户只能看到可视窗口的那一部分图片数据，这样就浪费了性能。

如果使用图片的懒加载就可以解决以上问题。在滚动屏幕之前，可视化区域之外的图片不会进行加载，在滚动屏幕时才加载。这样使得网

页的加载速度更快，减少了服务器的负载。懒加载适用于图片较多，页面列表较长（长列表）的场景中。

2. 懒加载的特点

减少无用资源的加载：使用懒加载明显减少了服务器的压力和流量，同时也减小了浏览器的负担。

提升用户体验：如果同时加载较多图片，可能需要等待的时间较长，这样影响了用户体验，而使用懒加载就能大大的提高用户体验。

防止加载过多图片而影响其他资源文件的加载 ：会影响网站应用的正常使用。

3. 懒加载的实现原理

图片的加载是由 src 引起的，当对 src 赋值时，浏览器就会请求图片资源。根据这个原理，我们使用 HTML5 的 data-xxx 属性来储存图片的路径，在需要加载图片的时候，将 data-xxx 中图片的路径赋值给 src，这样就实现了图片的按需加载，即懒加载。

注意：data-xxx 中的 xxx 可以自定义，这里我们使用 data-src 来定义。

懒加载的实现重点在于确定用户需要加载哪张图片，在浏览器中，可视区域内的资源就是用户需要的资源。所以当图片出现在可视区域时，获取图片的真实地址并赋值给图片即可。

使用原生 JavaScript 实现懒加载：

知识点：

window.innerHeight 是浏览器可视区的高度

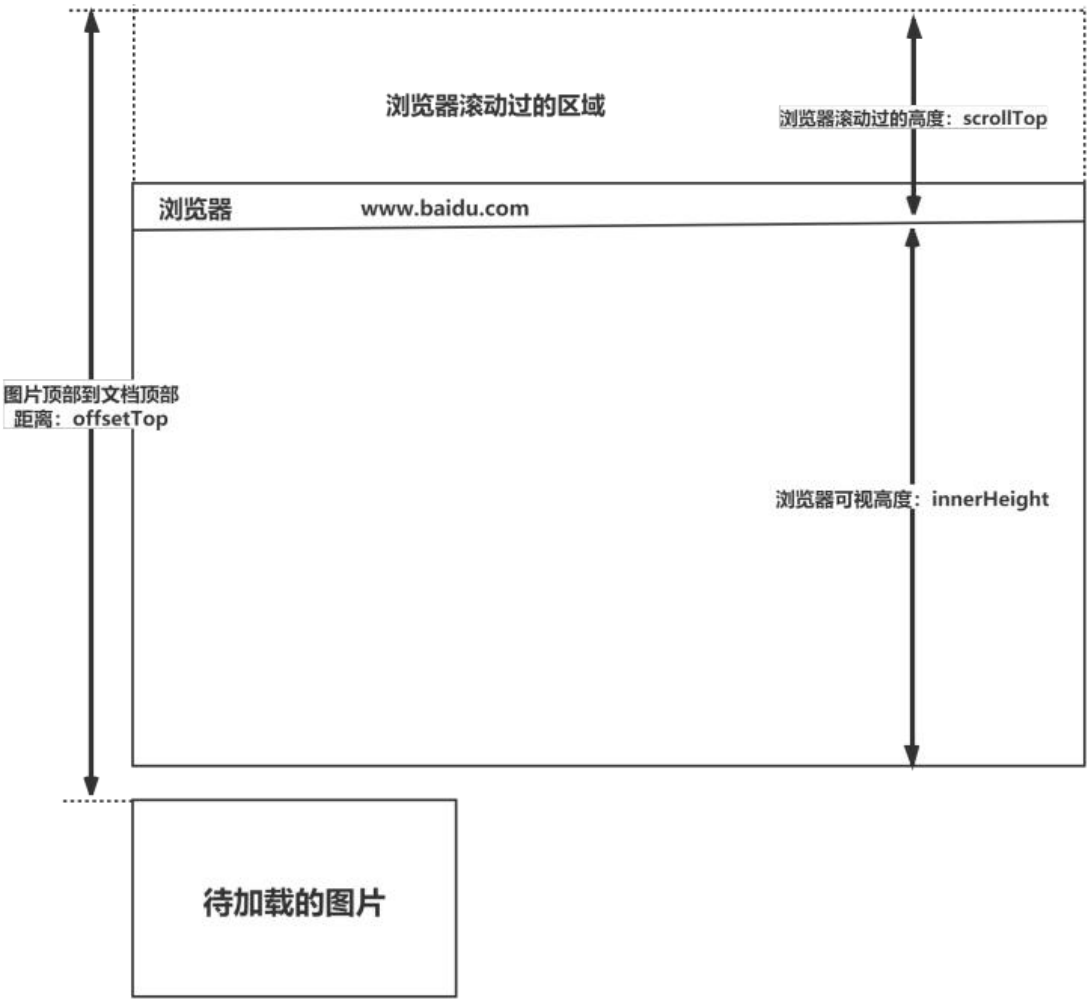
document.body.scrollTop

document.documentElement.scrollTop 是浏览器滚动的过的距离

imgs.offsetTop 是元素顶部距离文档顶部的高度（包括滚动条的距离）

图片加载条件：img.offsetTop < window.innerHeight + document.body.scrollTop;

图示：



代码实现：

```
<div class="container">
     <img src="loading.gif"  data-src="pic.png">
     <img src="loading.gif"  data-src="pic.png">
     <img src="loading.gif"  data-src="pic.png">
     <img src="loading.gif"  data-src="pic.png">
     <img src="loading.gif"  data-src="pic.png">
     <img src="loading.gif"  data-src="pic.png">
</div>
<script>
var imgs = document.querySelectorAll('img');
function lozyLoad(){
    var scrollTop = document.body.scrollTop || document.documentElement.scrollTop;
    var winHeight= window.innerHeight;
    for(var i=0;i < imgs.length;i++){
      if(imgs[i].offsetTop < scrollTop + winHeight ){
        imgs[i].src = imgs[i].getAttribute('data-src');
      }
    }
  }
  window.onscroll = lozyLoad();
</script>
```

4. 回流与重绘的概念及触发条件

（1）回流

当渲染树中部分或者全部元素的尺寸、结构或者属性发生变化时，浏览器会重新渲染部分或者全部文档的过程就称为回流。

下面这些操作会导致回流：

页面的首次渲染

浏览器的窗口大小发生变化

元素的内容发生变化

元素的尺寸或者位置发生变化

元素的字体大小发生变化

激活 CSS 伪类

查询某些属性或者调用某些方法

添加或者删除可见的 DOM 元素

在触发回流（重排）的时候，由于浏览器渲染页面是基于流式布局的，所以当触发回流时，会导致周围的 DOM 元素重新排列，它的影响范围有两种：

全局范围：从根节点开始，对整个渲染树进行重新布局

局部范围：对渲染树的某部分或者一个渲染对象进行重新布局

（2）重绘

当页面中某些元素的样式发生变化，但是不会影响其在文档流中的位置时，浏览器就会对元素进行重新绘制，这个过程就是重绘。

下面这些操作会导致重绘：

color、background 相关属性：background-color、background-image 等

outline 相关属性：outline-color、outline-width、text-decoration

border-radius、visibility、box-shadow

注意： 当触发回流时，一定会触发重绘，但是重绘不一定会引发回流。

5. 如何避免回流与重绘？

减少回流与重绘的措施：

操作 DOM 时，尽量在低层级的 DOM 节点进行操作

不要使用 table 布局， 一个小的改动可能会使整个 table 进行重新布局

使用 CSS 的表达式

不要频繁操作元素的样式，对于静态页面，可以修改类名，而不是样式。

使用 absolute 或者 fixed，使元素脱离文档流，这样他们发生变化就不会影响其他元素

避免频繁操作 DOM，可以创建一个文档片段 documentFragment，在它上面应用所有 DOM 操作，最后再把它添加到文档中

将元素先设置 display: none，操作结束后再把它显示出来。因为在 display 属性为 none 的元素上进行的 DOM 操作不会引发回流和重绘。

将 DOM 的多个读操作（或者写操作）放在一起，而不是读写操作穿插着写。这得益于浏览器的渲染队列机制。

浏览器针对页面的回流与重绘，进行了自身的优化——渲染队列

浏览器会将所有的回流、重绘的操作放在一个队列中，当队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会对队列进行批处理。这样就会让多次的回流、重绘变成一次回流重绘。

上面，将多个读操作（或者写操作）放在一起，就会等所有的读操作进入队列之后执行，这样，原本应该是触发多次回流，变成了只触发一次回流。

6. 如何优化动画？

对于如何优化动画，我们知道，一般情况下，动画需要频繁的操作 DOM，就就会导致页面的性能问题，我们可以将动画的 position 属性设置为 absolute 或者 fixed，将动画脱离文档流，这样他的回流就不会影响到页面了。

7. documentFragment 是什么？用它跟直接操作 DOM 的区别是什么？

MDN 中对 documentFragment 的解释：

DocumentFragment，文档片段接口，一个没有父对象的最小文档对象。它被作为一个轻量版的 Document 使用，就像标准的 document 一样，存储由节点（nodes）组成的文档结构。与 document 相比，最大的区别是 DocumentFragment 不是真实 DOM 树的一部分，它的变化不会触发 DOM 树的重新渲染，且不会导致性能等问题。

当我们把一个 DocumentFragment 节点插入文档树时，插入的不是 DocumentFragment 自身，而是它的所有子孙节点。在频繁的 DOM 操作时，我们就可以将 DOM 元素插入 DocumentFragment，之后一次性的将所有的子孙节点插入文档中。和直接操作 DOM 相比，将 DocumentFragment 节点插入 DOM 树时，不会触发页面的重绘，这样就大大提高了页面的性能。

8. 对节流与防抖的理解

函数防抖是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。

函数节流是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。

防抖函数的应用场景：

按钮提交场景：防止多次提交按钮，只执行最后提交的一次

服务端验证场景：表单验证需要服务端配合，只执行一段连续的输入事件的最后一次，还有搜索联想词功能类似生存环境请用lodash.debounce

节流函数的适用场景：

拖拽场景：固定时间内只执行一次，防止超高频次触发位置变动

缩放场景：监控浏览器 resize

动画场景：避免短时间内多次触发动画引起性能问题

9. 实现节流函数和防抖函数

函数防抖的实现：

```
function debounce(fn, wait) {
  var timer = null;

  return function() {
    var context = this,
      args = [...arguments];

    // 如果此时存在定时器的话，则取消之前的定时器重新记时
    if (timer) {
      clearTimeout(timer);
      timer = null;
    }

    // 设置定时器，使事件间隔指定事件后执行
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, wait);
  };
}
```

函数节流的实现：

```
// 时间戳版
function throttle(fn, delay) {
  var preTime = Date.now();

  return function() {
    var context = this,
      args = [...arguments],
      nowTime = Date.now();

    // 如果两次时间间隔超过了指定时间，则执行函数。
    if (nowTime - preTime >= delay) {
      preTime = Date.now();
      return fn.apply(context, args);
    }
  };
}

// 定时器版
function throttle (fun, wait){
  let timeout = null
  return function(){
    let context = this
    let args = [...arguments]
    if(!timeout){
      timeout = setTimeout(() => {
        fun.apply(context, args)
        timeout = null
      }, wait)
    }
  }
}
```

10. 如何对项目中的图片进行优化？

1.不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 CSS 去代替。

2.对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 CDN 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。

3.小图使用 base64 格式

4.将多个图标文件整合到一张图片中（雪碧图）

5.选择正确的图片格式：

对于能够显示 WebP 格式的浏览器尽量使用 WebP 格式。因为 WebP 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好

小图使用 PNG，其实对于大部分图标这类图片，完全可以使用 SVG 代替

照片使用 JPEG

11. 常见的图片格式及使用场景

（1）BMP，是无损的、既支持索引色也支持直接色的点阵图。这种图片格式几乎没有对数据进行压缩，所以 BMP 格式的图片通常是较大的文件。

（2）GIF 是无损的、采用索引色的点阵图。采用 LZW 压缩算法进行编码。文件小，是 GIF 格式的优点，同时，GIF 格式还具有支持动画以及透明的优点。但是 GIF 格式仅支持 8bit 的索引色，所以 GIF 格式适用于对色彩要求不高同时需要文件体积较小的场景。

（3）JPEG 是有损的、采用直接色的点阵图。JPEG 的图片的优点是采用了直接色，得益于更丰富的色彩，JPEG 非常适合用来存储照片，与 GIF 相比，JPEG 不适合用来存储企业 Logo、线框类的图。因为有损压缩会导致图片模糊，而直接色的选用，又会导致图片文件较 GIF 更大。

（4）PNG-8 是无损的、使用索引色的点阵图。PNG 是一种比较新的图片格式，PNG-8 是非常好的 GIF 格式替代者，在可能的情况下，应该尽可能的使用 PNG-8 而不是 GIF，因为在相同的图片效果下，PNG-8 具有更小的文件体积。除此之外，PNG-8 还支持透明度的调节，而 GIF 并不支持。除非需要动画的支持，否则没有理由使用 GIF 而不是 PNG-8。

（5）PNG-24 是无损的、使用直接色的点阵图。PNG-24 的优点在于它压缩了图片的数据，使得同样效果的图片，PNG-24 格式的文件大小要比 BMP 小得多。当然，PNG24 的图片还是要比 JPEG、GIF、PNG-8 大得多。

（6）SVG 是无损的矢量图。SVG 是矢量图意味着 SVG 图片由直线和曲线以及绘制它们的方法组成。当放大 SVG 图片时，看到的还是线和曲线，而不会出现像素点。这意味着 SVG 图片在放大时，不会失真，所以它非常适合用来绘制 Logo、Icon 等。

（7）WebP 是谷歌开发的一种新图片格式，WebP 是同时支持有损和无损压缩的、使用直接色的点阵图。从名字就可以看出来它是为 Web 而生的，什么叫为 Web 而生呢？就是说相同质量的图片，WebP 具有更小的文件体积。现在网站上充满了大量的图片，如果能够降低每一个图片的文件大小，那么将大大减少浏览器和服务器之间的数据传输量，进而降低访问延迟，提升访问体验。目前只有 Chrome 浏览器和 Opera 浏览器支持 WebP 格式，兼容性不太好。

在无损压缩的情况下，相同质量的 WebP 图片，文件大小要比 PNG 小 26%；

在有损压缩的情况下，具有相同图片精度的 WebP 图片，文件大小要比 JPEG 小 25%~34%；

WebP 图片格式支持图片透明度，一个无损压缩的 WebP 图片，如果要支持透明度只需要 22%的格外文件大小。

12. 如何用webpack 来优化前端性能？

用webpack 优化前端性能是指优化 webpack 的输出结果，让打包的最终结果在浏览器运行快速高效。

压缩代码：删除多余的代码、注释、简化代码的写法等等方式。可以利用webpack的 UglifyJsPlugin 和 ParallelUglifyPlugin 来压缩JS文件， 利用 cssnano （css-loader?minimize）来压缩 css

利用CDN 加速: 在构建过程中，将引用的静态资源路径修改为 CDN 上对应的路径。可以利用webpack 对于 output 参数和各 loader 的 publicPath 参数来修改资源路径

Tree Shaking: 将代码中永远不会走到的片段删除掉。可以通过在启动 webpack 时追加参数 --optimize-minimize 来实现

Code Splitting: 将代码按路由维度或者组件分块(chunk),这样做到按需加载,同时可以充分利用浏览器缓存

提取公共第三方库: SplitChunksPlugin 插件来进行公共模块抽取,利用浏览器缓存可以长期缓存这些无需频繁变动的公共代码

13. 如何提高webpack 的构建速度？

1.多入口情况下，使用 CommonsChunkPlugin 来提取公共代码

2.通过 externals 配置来提取常用库

3.利用 DllPlugin 和 DllReferencePlugin 预编译资源模块 通过 DllPlugin 来对那些我们引用但是绝对不会修改的 npm 包来进行预编译，再通过 DllReferencePlugin 将预编译的模块加载进来。

4.使用 Happypack 实现多线程加速编译

5.使用 webpack-uglify-parallel 来提升 uglifyPlugin 的压缩速度。 原理上 webpack-uglify-parallel 采用了多核并行压缩来提升压缩速度

6.使用 Tree-shaking 和 Scope Hoisting 来剔除多余代码

前端工程化部分

1. webpack 与 grunt、gulp 的不同？

Grunt、Gulp 是基于任务运行的工具： 它们会自动执行指定的任务，就像流水线，把资源放上去然后通过不同插件进行加工，它们包含活跃的社区，丰富的插件，能方便的打造各种工作流。

Webpack 是基于模块化打包的工具: 自动化处理模块，webpack 把一切当成模块，当 webpack 处理应用程序时，它会递归地构建一个依赖关系图 (dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle。

因此这是完全不同的两类工具,而现在主流的方式是用npm script 代替 Grunt、Gulp，npm script 同样可以打造任务流。

2. webpack、rollup、parcel 优劣？

webpack 适用于大型复杂的前端站点构建: webpack 有强大的 loader 和插件生态,打包后的文件实际上就是一个立即执行函数，这个立即执行函数接收一个参数,这个参数是模块对象,键为各个模块的路径,值为模块内容。立即执行函数内部则处理模块之间的引用，执行模块等,这种情况更适合文件依赖复杂的应用开发。

rollup 适用于基础库的打包，如 vue、d3 等: Rollup 就是将各个模块打包进一个文件中，并且通过 Tree-shaking 来删除无用的代码,可以最大程度上降低代码体积,但是 rollup 没有 webpack 如此多的的如代码分割、按需加载等高级功能，其更聚焦于库的打包，因此更适合库的开发。

parcel 适用于简单的实验性项目：他可以满足低门槛的快速看到效果,但是生态差、报错信息不够全面都是他的硬伤，除了一些玩具项目或者实验项目不建议使用。

3. 有哪些常见的 Loader？

file-loader：把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件

url-loader：和 file-loader 类似，但是能在文件很小的情况下以 base64 的方式把文件内容注入到代码中去

source-map-loader：加载额外的 Source Map 文件，以方便断点调试

image-loader：加载并且压缩图片文件

babel-loader：把 ES6 转换成 ES5

css-loader：加载 CSS，支持模块化、压缩、文件导入等特性

style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS。

eslint-loader：通过 ESLint 检查 JavaScript 代码

注意：在 Webpack 中，loader 的执行顺序是从右向左执行的。因为 webpack 选择了 compose 这样的函数式编程方式，这种方式的表达式执行是从右向左的。

4. 有哪些常见的 Plugin？

define-plugin：定义环境变量

html-webpack-plugin：简化 html文件创建

uglifyjs-webpack-plugin：通过 UglifyES 压缩 ES6 代码

webpack-parallel-uglify-plugin: 多核压缩，提高压缩速度

webpack-bundle-analyzer: 可视化 webpack 输出文件的体积

mini-css-extract-plugin: CSS 提取到单独的文件中，支持按需加载

5. bundle，chunk，module 是什么？

bundle：是由 webpack 打包出来的文件；

chunk：代码块，一个 chunk 由多个模块组合而成，用于代码的合并和分割；

module：是开发中的单个模块，在 webpack 的世界，一切皆模块，一个模块对应一个文件，webpack 会从配置的 entry 中递归开始找出所有依赖的模块。

6. Loader 和 Plugin 的不同？

不同的作用:

Loader 直译为"加载器"。Webpack 将一切文件视为模块，但是 webpack 原生是只能解析 js文件，如果想将其他文件也打包的话，就会用到 loader 。 所以 Loader 的作用是让 webpack 拥有了加载和解析非 JavaScript文件的能力。

Plugin 直译为"插件"。Plugin 可以扩展 webpack 的功能，让 webpack 具有更多的灵活性。 在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

不同的用法:

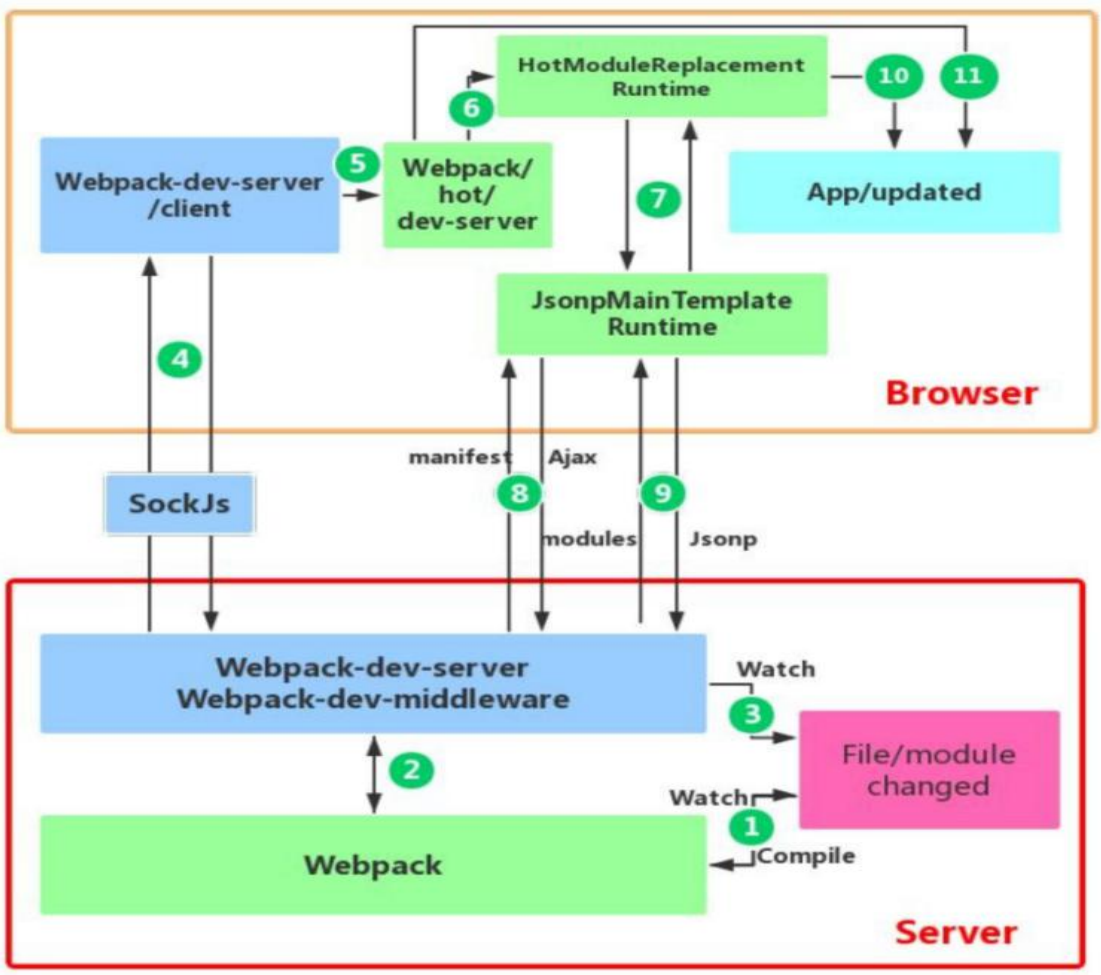
Loader 在 module.rules 中配置，也就是说他作为模块的解析规则而存在。 类型为数组，每一项都是一个 Object ，里面描述了对于什么类型的文件（ test ），使用什么加载(loader)和使用的参数（ options ）

Plugin 在 plugins 中单独配置。类型为数组,每一项是一个 plugin 的实例，参数都通过构造函数传入。

7. webpack 热更新的实现原理？

webpack 的热更新又称热替换（Hot Module Replacement），缩写为 HMR。 这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

原理：



首先要知道 server 端和 client 端都做了处理工作：

第一步，在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文

件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中。

第二步是 webpack-dev-server 和 webpack 之间的接口交互，而在这一步，主要是 dev-server 的中间件 webpack- dev-middleware 和 webpack 之间的交互，webpack-dev-middleware 调用 webpack 暴露的 API 对代码变化进行监 控，并且告诉 webpack，将代码打包到内存中。

第三步是 webpack-dev-server 对文件变化的一个监控，这一步不同于第一步，并不是监控代码变化重新打包。当我们在配置文件中配置了 devServer.watchContentBase 为 true 的时候，Server 会监听这些配置文件夹中静态文件的变化，变化后会通知浏览器端对应用进行 live reload。注意，这儿是浏览器刷新，和 HMR 是两个概念。

第四步也是 webpack-dev-server 代码的工作，该步骤主要是通过 sockjs（webpack-dev-server 的依赖）在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端，同时也包括第三步中 Server 监听静态文件变化的信息。浏览器端根据这些 socket 消息进行不同的操作。当然服务端传递的最主要信息还是新模块的 hash 值，后面的步骤根据这一 hash 值来进行模块热替换。

webpack-dev-server/client 端并不能够请求更新的代码，也不会执行热更模块操作，而把这些工作又交回给了 webpack，

webpack/hot/dev-server 的工作就是根据 webpack-dev-server/client 传给它的信息以及 dev-server 的配置决定是刷新浏览器呢还是进行模块热更新。当然如果仅仅是刷新浏览器，也就没有后面那些步骤了。

HotModuleReplacement.runtime 是客户端 HMR 的中枢，它接收到上一步传递给他的新模块的 hash 值，它通过 JsonpMainTemplate.runtime 向 server 端发送 Ajax 请求，服务端返回一个 json，该 json 包含了所有要更新的模块的 hash 值，获取到更新列表后，该模块再次通过 jsonp 请求，获取到最新的模块代码。这就是上图中 7、8、9 步骤。

而第 10 步是决定 HMR 成功与否的关键步骤，在该步骤中，HotModulePlugin 将会对新旧模块进行对比，决定是否更新模块，在决定更新模块后，检查模块之间的依赖关系，更新模块的同时更新模块间的依赖引用。

最后一步，当 HMR 失败后，回退到 live reload 操作，也就是进行浏览器刷新来获取最新打包代码。

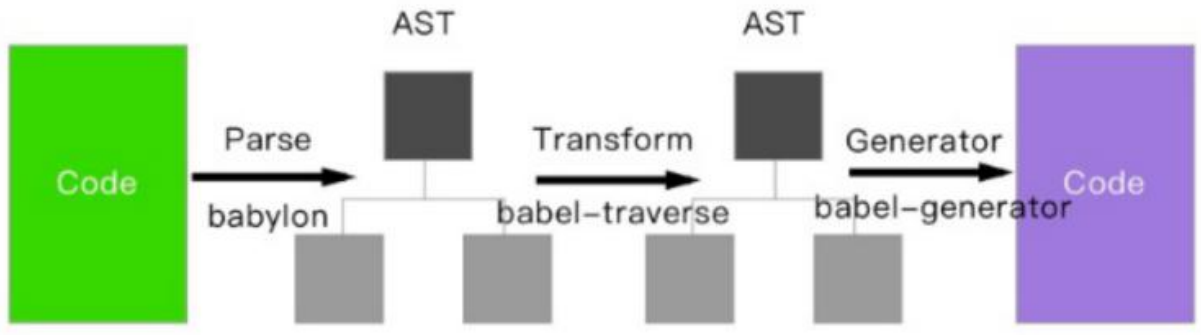
8. Babel 的原理是什么？

babel 的转译过程也分为三个阶段，这三步具体是：

解析 Parse: 将代码解析生成抽象语法树（AST），即词法分析与语法分析的过程；

转换 Transform: 对于 AST 进行变换一系列的操作，babel 接受得到 AST 并通过 babel-traverse 对其进行遍历，在此过程中进行添加、更新及移除等操作；

生成 Generate：将变换后的 AST 再转换为 JS 代码，使用到的模块是 babel-generator。



9. git 和 svn 的区别

git 和 svn 最大的区别在于 git 是分布式的，而 svn 是集中式的。因此我们不能再离线的情况下使用 svn。如果服务器出现问题，就没有办法使用 svn 来提交代码。

svn 中的分支是整个版本库的复制的一份完整目录，而 git 的分支是指针指向某次提交，因此 git 的分支创建更加开销更小并且分支上的变化不会影响到其他人。svn 的分支变化会影响到所有的人。

svn 的指令相对于 git 来说要简单一些，比 git 更容易上手。

GIT 把内容按元数据方式存储，而 SVN 是按文件：因为 git 目录是处于个人机器上的一个克隆版的版本库，它拥有中心版本库上所有的东西，例如标签，分支，版本记录等。

GIT 分支和 SVN 的分支不同：svn 会发生分支遗漏的情况，而 git 可以同一个工作目录下快速的在几个分支间切换，很容易发现未被合并的分支，简单而快捷的合并这些文件。

GIT 没有一个全局的版本号，而 SVN 有

GIT 的内容完整性要优于 SVN：GIT 的内容存储使用的是 SHA-1 哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏

10. 经常使用的 git 命令？

```
git init                    // 新建 git 代码库
git add                     // 添加指定文件到暂存区
git rm                      // 删除工作区文件，并且将这次删除放入暂存区
git commit -m [message]     // 提交暂存区到仓库区
git branch                  // 列出所有分支
git checkout -b [branch]    // 新建一个分支，并切换到该分支
git status                  // 显示有变更文件的状态
```

11. git pull 和 git fetch 的区别

git fetch 只是将远程仓库的变化下载下来，并没有和本地分支合并。

git pull 会将远程仓库的变化下载下来，并和当前分支合并。

12. git rebase 和 git merge 的区别

git merge 和 git rebase 都是用于分支合并，关键在 commit 记录的处理上不同：

git merge 会新建一个新的 commit 对象，然后两个分支以前的 commit 记录都指向这个新 commit 记录。这种方法会保留之前每个分支的 commit 历史。

git rebase 会先找到两个分支的第一个共同的 commit 祖先记录，然后将提取当前分支这之后的所有 commit 记录，然后将这个 commit 记录添加到目标分支的最新提交后面。经过这个合并后，两个分支合并后的 commit 记录就变为了线性的记录了。

浏览器部分

1. 什么是 XSS 攻击？

（1）概念

XSS 攻击指的是跨站脚本攻击，是一种代码注入攻击。攻击者通过在网站注入恶意脚本，使之在用户的浏览器上运行，从而盗取用户的信息如 cookie 等。

XSS 的本质是因为网站没有对恶意代码进行过滤，与正常的代码混合在一起了，浏览器没有办法分辨哪些脚本是可信的，从而导致了恶意代码的执行。

攻击者可以通过这种攻击方式可以进行以下操作：

获取页面的数据，如 DOM、cookie、localStorage；

DOS 攻击，发送合理请求，占用服务器资源，从而使用户无法访问服务器；

破坏页面结构；

流量劫持（将链接指向某网站）；

（2）攻击类型

XSS 可以分为存储型、反射型和 DOM 型：

存储型指的是恶意脚本会存储在目标服务器上，当浏览器请求数据时，脚本从服务器传回并执行。

反射型指的是攻击者诱导用户访问一个带有恶意代码的 URL 后，服务器端接收数据后处理，然后把带有恶意代码的数据发送到浏览器端，浏览器端解析这段带有 XSS 代码的数据后当做脚本执行，最终完成 XSS 攻击。

DOM 型指的通过修改页面的 DOM 节点形成的 XSS。

1）存储型 XSS 的攻击步骤：

1.攻击者将恶意代码提交到目标网站的数据库中。

2.用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器。

3.用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。

4.恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等。

2）反射型 XSS 的攻击步骤：

1.攻击者构造出特殊的 URL，其中包含恶意代码。

2.用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。

3.用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。

4.恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。 由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

3）DOM 型 XSS 的攻击步骤：

1.攻击者构造出特殊的 URL，其中包含恶意代码。

2.用户打开带有恶意代码的 URL。

3.用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。

4.恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

2. 如何防御 XSS 攻击？

可以看到 XSS 危害如此之大，那么在开发网站时就要做好防御措施，具体措施如下：

可以从浏览器的执行来进行预防，一种是使用纯前端的方式，不用服务器端拼接后返回（不使用服务端渲染）。另一种是对需要插入到 HTML 中的代码做好充分的转义。对于 DOM 型的攻击，主要是前端脚本的不可靠而造成的，对于数据获取渲染和字符串拼接的时候应该对可能出现的恶意代码情况进行判断。

使用 CSP ，CSP 的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行，从而防止恶意代码的注入攻击。

1.CSP 指的是内容安全策略，它的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截由浏览器自己来实现。

2.通常有两种方式来开启 CSP，一种是设置 HTTP 首部中的 Content-Security-Policy，一种是设置 meta 标签的方式 <meta http-equiv="Content-Security-Policy">

对一些敏感信息进行保护，比如 cookie 使用 http-only，使得脚本无法获取。也可以使用验证码，避免脚本伪装成用户执行一些操作。

3. 什么是 CSRF 攻击？

（1）概念

CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态，绕过后台的用户验证，冒充用户向服务器执行一些操作。

CSRF 攻击的本质是利用 cookie 会在同源请求中携带发送给服务器的特点，以此来实现用户的冒充。

（2）攻击类型

常见的 CSRF 攻击有三种：

GET 类型的 CSRF 攻击，比如在网站中的一个 img 标签里构建一个请求，当用户打开这个网站的时候就会自动发起提交。

POST 类型的 CSRF 攻击，比如构建一个表单，然后隐藏它，当用户进入页面时，自动提交这个表单。

链接类型的 CSRF 攻击，比如在 a 标签的 href 属性里构建一个请求，然后诱导用户去点击。

4. 如何防御 CSRF 攻击？

CSRF 攻击可以使用以下方法来防护：

进行同源检测，服务器根据 http 请求头中 origin 或者 referer 信息来判断请求是否为允许访问的站点，从而对请求进行过滤。当 origin 或者 referer 信息都不存在的时候，直接阻止请求。这种方式的缺点是有些情况下 referer 可以被伪造，同时还会把搜索引擎的链接也给屏蔽了。所以一般网站会允许搜索引擎的页面请求，但是相应的页面请求这种请求方式也可能被攻击者给利用。（Referer 字段会告诉服务器该网页是从哪个页面链接过来的）

使用 CSRF Token 进行验证，服务器向用户返回一个随机数 Token ，当网站再次发起请求时，在请求参数中加入服务器端返回的 token ，然后服务器对这个 token 进行验证。这种方法解决了使用 cookie 单一验证方式时，可能会被冒用的问题，但是这种方法存在一个缺点就是，我们需要给网站中的所有请求都添加上这个 token，操作比较繁琐。还有一个问题是一般不会只有一台网站服务器，如果请求经过负载平衡转移到了其他的服务器，但是这个服务器的 session 中没有保留这个 token 的话，就没有办法验证了。这种情况可以通过改变 token 的构建方式来解决。

对 Cookie 进行双重验证，服务器在用户访问网站页面时，向请求域名注入一个 Cookie，内容为随机字符串，然后当用户再次向服务器发送请求的时候，从 cookie 中取出这个字符串，添加到 URL 参数中，然后服务器通过对 cookie 中的数据和参数中的数据进行比较，来进行验证。使用这种方式是利用了攻击者只能利用 cookie，但是不能访问获取 cookie 的特点。并且这种方法比 CSRF Token 的方法更加方便，并且不涉及到分布式访问的问题。这种方法的缺点是如果网站存在 XSS 漏洞的，那么这种方式会失效。同时这种方式不能做到子域名的隔离。

在设置 cookie 属性的时候设置 Samesite ，限制 cookie 不能作为被第三方使用，从而可以避免被攻击者利用。Samesite 一共有两种模式，一种是严格模式，在严格模式下 cookie 在任何情况下都不可能作为第三方 Cookie 使用，在宽松模式下，cookie 可以被请求是 GET 请求，且会发生页面跳转的请求所使用。

5. 有哪些可能引起前端安全的问题?

跨站脚本 (Cross-Site Scripting, XSS): 一种代码注入方式, 为了与 CSS 区分所以被称作 XSS。早期常见于网络论坛, 起因是网站没有对用户的输入进行严格的限制, 使得攻击者可以将脚本上传到帖子让其他人浏览到有恶意脚本的页面, 其注入方式很简单包括但不限于 JavaScript / CSS / Flash 等;

iframe 的滥用: iframe 中的内容是由第三方来提供的，默认情况下他们不受控制，他们可以在 iframe 中运行JavaScirpt 脚本、Flash 插件、弹出对话框等等，这可能会破坏前端用户体验;

跨站点请求伪造（Cross-Site Request Forgeries，CSRF）: 指攻击者通过设置好的陷阱，强制对已完成认证的用户进行非预期的个人信息或设定信息等某些状态更新，属于被动攻击

恶意第三方库: 无论是后端服务器应用还是前端应用开发，绝大多数时候都是在借助开发框架和各种类库进行快速开发，一旦第三方库被植入恶意代码很容易引起安全问题。

6. 网络劫持有哪几种，如何防范？

网络劫持分为两种:

（1）DNS 劫持: (输入京东被强制跳转到淘宝这就属于 dns 劫持)

DNS 强制解析：通过修改运营商的本地 DNS 记录，来引导用户流量到缓存服务器

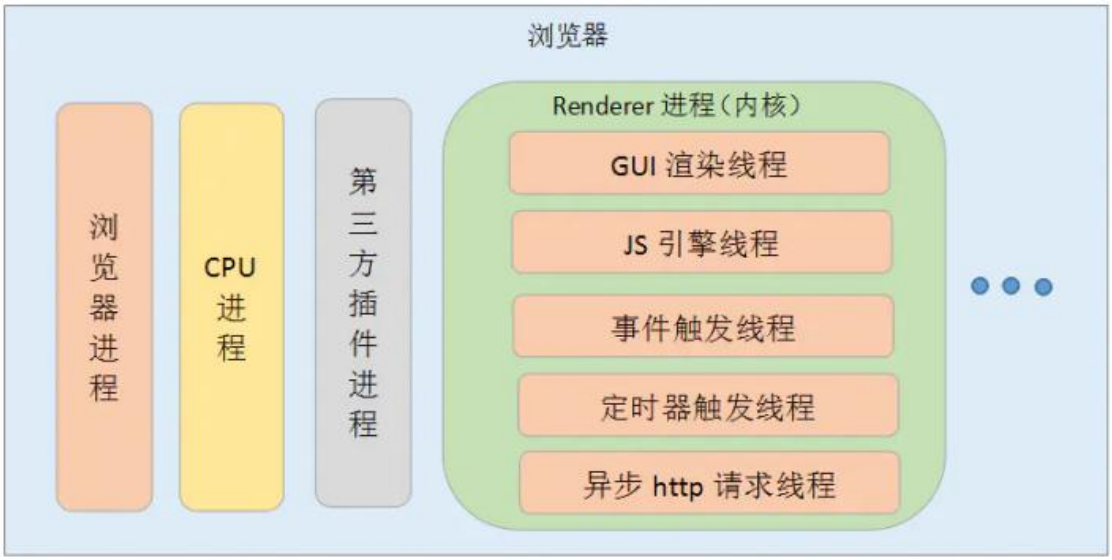
302 跳转的方式：通过监控网络出口的流量，分析判断哪些内容是可以进行劫持处理的,再对劫持的内存发起 302 跳转的回复，引导用户获取内容

（2）HTTP 劫持：(访问谷歌但是一直有贪玩蓝月的广告),由于 http 明文传输,运营商会修改你的 http 响应内容(即加广告)

（3）DNS 劫持由于涉嫌违法，已经被监管起来，现在很少会有 DNS 劫持，而http 劫持依然非常盛行，最有效的办法就是全站 HTTPS，将 HTTP 加密，这使得运营商无法获取明文，就无法劫持你的响应内容。

7. 浏览器渲染进程的线程有哪些

浏览器的渲染进程的线程总共有五种：



（1）GUI 渲染线程

负责渲染浏览器页面，解析 HTML、CSS，构建 DOM 树、构建 CSSOM 树、构建渲染树和绘制页面；当界面需要重绘或由于某种操作引发回流时，该线程就会执行。

注意：GUI 渲染线程和 JS 引擎线程是互斥的，当 JS 引擎执行时 GUI 线程会被挂起，GUI 更新会被保存在一个队列中等到 JS 引擎空闲时立即被执行。

（2）JS 引擎线程

JS 引擎线程也称为 JS 内核，负责处理 Javascript 脚本程序，解析 Javascript 脚本，运行代码；JS 引擎线程一直等待着任务队列中任务的到来，然后加以处理，一个 Tab 页中无论什么时候都只有一个 JS 引擎线程在运行 JS 程序；

注意：GUI 渲染线程与 JS 引擎线程的互斥关系，所以如果 JS 执行的时间过长，会造成页面的渲染不连贯，导致页面渲染加载阻塞。

（3）时间触发线程

时间触发线程属于浏览器而不是 JS 引擎，用来控制事件循环；当 JS 引擎执行代码块如 setTimeOut 时（也可是来自浏览器内核的其他线程,如鼠标点击、AJAX 异步请求等），会将对应任务添加到事件触发线程中；当对应的事件符合触发条件被触发时，该线程会把事件添加到待处理队列的队尾，等待 JS 引擎的处理；

注意：由于 JS 的单线程关系，所以这些待处理队列中的事件都得排队等待 JS 引擎处理（当 JS 引擎空闲时才会去执行）；

（4）定时器触发进程

定时器触发进程即 setInterval 与 setTimeout 所在线程；浏览器定时计数器并不是由 JS 引擎计数的，因为 JS 引擎是单线程的，如果处于阻塞线程状态就会影响记计时的准确性；因此使用单独线程来计时并触发定时器，计时完毕后，添加到事件队列中，等待 JS 引擎空闲后执行，所以定时器中的任务在设定的时间点不一定能够准时执行，定时器只是在指定时间点将任务添加到事件队列中；

注意：W3C 在 HTML 标准中规定，定时器的定时时间不能小于 4ms，如果是小于 4ms，则默认为 4ms。

（5）异步 http 请求线程

XMLHttpRequest 连接后通过浏览器新开一个线程请求；

检测到状态变更时，如果设置有回调函数，异步线程就产生状态变更事件，将回调函数放入事件队列中，等待 JS 引擎空闲后执行；

8. 僵尸进程和孤儿进程是什么？

孤儿进程：父进程退出了，而它的一个或多个进程还在运行，那这些子进程都会成为孤儿进程。孤儿进程将被 init 进程(进程号为 1)所收养，并由 init 进程对它们完成状态收集工作。

僵尸进程：子进程比父进程先结束，而父进程又没有释放子进程占用的资源，那么子进程的进程描述符仍然保存在系统中，这种进程称之为僵死进程。

9. 如何实现浏览器内多个标签页之间的通信?

实现多个标签页之间的通信，本质上都是通过中介者模式来实现的。因为标签页之间没有办法直接通信，因此我们可以找一个中介者，让

标签页和中介者进行通信，然后让这个中介者来进行消息的转发。通信方法如下：

使用 websocket 协议，因为 websocket 协议可以实现服务器推送，所以服务器就可以用来当做这个中介者。标签页通过向服务器发送数据，然后由服务器向其他标签页推送转发。

使用 ShareWorker 的方式，shareWorker 会在页面存在的生命周期内创建一个唯一的线程，并且开启多个页面也只会使用同一个线程。这个时候共享线程就可以充当中介者的角色。标签页间通过共享一个线程，然后通过这个共享的线程来实现数据的交换。

使用 localStorage 的方式,我们可以在一个标签页对 localStorage 的变化事件进行监听，然后当另一个标签页修改数据的时候，我们就可以通过这个监听事件来获取到数据。这个时候 localStorage 对象就是充当的中介者的角色。

使用 postMessage 方法，如果我们能够获得对应标签页的引用，就可以使用 postMessage 方法，进行通信。

10. 对浏览器的缓存机制的理解

浏览器缓存的全过程：

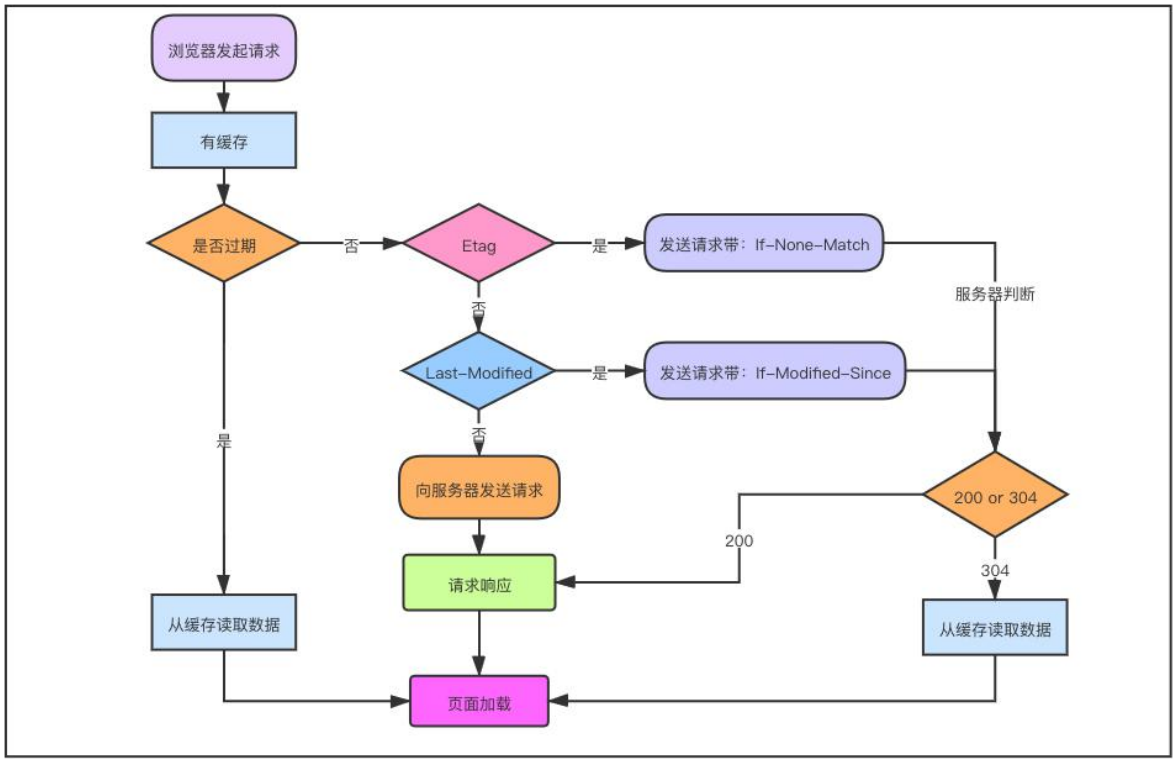
浏览器第一次加载资源，服务器返回 200，浏览器从服务器下载资源文件，并缓存资源文件与 response header，以供下次加载时对比使用；

下一次加载资源时，由于强制缓存优先级较高，先比较当前时间与上一次返回 200 时的时间差，如果没有超过 cache-control 设置的 max-age，则没有过期，并命中强缓存，直接从本地读取资源。如果浏览器不支持 HTTP1.1，则使用 expires 头判断是否过期；

如果资源已过期，则表明强制缓存没有被命中，则开始协商缓存，向服务器发送带有 If-None-Match 和 If-Modified-Since 的请求；

服务器收到请求后，优先根据 Etag 的值判断被请求的文件有没有做修改，Etag 值一致则没有修改，命中协商缓存，返回 304；如果不一致则有改动，直接返回新的资源文件带上新的 Etag 值并返回 200；

如果服务器收到的请求没有 Etag 值，则将 If-Modified-Since 和被请求文件的最后修改时间做比对，一致则命中协商缓存，返回 304；不一致则返回新的 last-modified 和文件并返回 200；



很多网站的资源后面都加了版本号，这样做的目的是：每次升级了 JS 或 CSS 文件后，为了防止浏览器进行缓存，强制改变版本号，客户端浏览器就会重新下载新的 JS 或 CSS 文件 ，以保证用户能够及时获得网站的最新更新。

11. 协商缓存和强缓存的区别

（1）强缓存

使用强缓存策略时，如果缓存资源有效，则直接使用缓存资源，不必再向服务器发起请求。

强缓存策略可以通过两种方式来设置，分别是 http 头信息中的 Expires 属性和 Cache-Control 属性

（1）服务器通过在响应头中添加 Expires 属性，来指定资源的过期时间。在过期时间以内，该资源可以被缓存使用，不必再向服务器发送请求。这个时间是一个绝对时间，它是服务器的时间，因此可能存在这样的问题，就是客户端的时间和服务器端的时间不一致，或者用户可以对客户端时间进行修改的情况，这样就可能会影响缓存命中的结果。

（2）Expires 是 http1.0 中的方式，因为它的一些缺点，在 HTTP 1.1 中提出了一个新的头部属性就是 Cache-Control 属性，它提供了对资源的缓存的更精确的控制。它有很多不同的值，

Cache-Control 可设置的字段：

public：设置了该字段值的资源表示可以被任何对象（包括：发送请求的客户端、代理服务器等等）缓存。这个字段值不常用，一般还是使用 max-age=来精确控制；

private：设置了该字段值的资源只能被用户浏览器缓存，不允许任何代理服务器缓存。在实际开发当中，对于一些含有用户信息的 HTML，通常都要设置这个字段值，避免代理服务器(CDN)缓存；

no-cache：设置了该字段需要先和服务端确认返回的资源是否发生了变化，如果资源未发生变化，则直接使用缓存好的资源；

no-store：设置了该字段表示禁止任何缓存，每次都会向服务端发起新的请求，拉取最新的资源；

max-age=：设置缓存的最大有效期，单位为秒；

s-maxage=：优先级高于 max-age=，仅适用于共享缓存(CDN)，优先级高于 max-age 或者 Expires 头；

max-stale[=]：设置了该字段表明客户端愿意接收已经过期的资源，但是不能超过给定的时间限制。

一般来说只需要设置其中一种方式就可以实现强缓存策略，当两种方式一起使用时，Cache-Control 的优先级要高于 Expires。

no-cache 和 no-store 很容易混淆：

no-cache 是指先要和服务器确认是否有资源更新，在进行判断。也就是说没有强缓存，但是会有协商缓存；

no-store 是指不使用任何缓存，每次请求都直接从服务器获取资源。

（2）协商缓存

如果命中强制缓存，我们无需发起新的请求，直接使用缓存内容，如果没有命中强制缓存，如果设置了协商缓存，这个时候协商缓存就会发挥作用了。

上面已经说到了，命中协商缓存的条件有两个：

max-age=xxx 过期了

值为 no-store

使用协商缓存策略时，会先向服务器发送一个请求，如果资源没有发生修改，则返回一个 304 状态，让浏览器使用本地的缓存副本。如果资源发生了修改，则返回修改后的资源。

协商缓存也可以通过两种方式来设置，分别是 http 头信息中的 Etag 和 Last-Modified 属性。

（1）服务器通过在响应头中添加 Last-Modified 属性来指出资源最后一次修改的时间，当浏览器下一次发起请求时，会在请求头中添加一个 If-Modified-Since 的属性，属性值为上一次资源返回时的 Last-Modified 的值。当请求发送到服务器后服务器会通过这个属性来和资源的最后一次的修改时间来进行比较，以此来判断资源是否做了修改。如果资源没有修改，那么返回 304 状态，让客户端使用本地的缓存。如果资源已经被修改了，则返回修改后的资源。使用这种方法有一个缺点，就是 Last-Modified 标注的最后修改时间只能精确到秒级，如果某些文件在 1 秒钟以内，被修改多次的话，那么文件已将改变了但是 Last-Modified 却没有改变，这样会造成缓存命中的不准确。

（2）因为 Last-Modified 的这种可能发生的不准确性，http 中提供了另外一种方式，那就是 Etag 属性。服务器在返回资源的时候，在头信息中添加了 Etag 属性，这个属性是资源生成的唯一标识符，当资源发生改变的时候，这个值也会发生改变。在下一次资源请求时，浏览器会在请求头中添加一个 If-None-Match 属性，这个属性的值就是上次返回的资源的 Etag 的值。服务接收到请求后会根据这个值来和资源当前的 Etag 的值来进行比较，以此来判断资源是否发生改变，是否需要返回资源。通过这种方式，比 Last-Modified 的方式更加精确。

当 Last-Modified 和 Etag 属性同时出现的时候，Etag 的优先级更高。使用协商缓存的时候，服务器需要考虑负载平衡的问题，因此多个服务器上资源的 Last-Modified 应该保持一致，因为每个服务器

上 Etag 的值都不一样，因此在考虑负载平衡时，最好不要设置 Etag 属性。

总结：

强缓存策略和协商缓存策略在缓存命中时都会直接使用本地的缓存副本，区别只在于协商缓存会向服务器发送一次请求。它们缓存不命中时，都会向服务器发送请求来获取资源。在实际的缓存机制中，强缓存策略和协商缓存策略是一起合作使用的。浏览器首先会根据请求的信息判断，强缓存是否命中，如果命中则直接使用资源。如果不命中则根据头信息向服务器发起请求，使用协商缓存，如果协商缓存命中的话，则服务器不返回资源，浏览器直接使用本地资源的副本，如果协商缓存不命中，则浏览器返回最新的资源给浏览器。

12. 点击刷新按钮或者按 F5、按 Ctrl+F5 （强制刷新）、地址栏回车有什么区别？

点击刷新按钮或者按 F5：浏览器直接对本地的缓存文件过期，但是会带上 If-Modifed-Since，If-None-Match，这就意味着服务器会对文件检查新鲜度，返回结果可能是 304，也有可能是 200。

用户按 Ctrl+F5（强制刷新）：浏览器不仅会对本地文件过期，而且不会带上 If-Modifed-Since，If-None-Match，相当于之前从来没有请求过，返回结果是 200。

地址栏回车： 浏览器发起请求，按照正常流程，本地检查是否过期，然后服务器检查新鲜度，最后返回内容。

13. 常见的浏览器内核比较

Trident:这种浏览器内核是 IE 浏览器用的内核,因为在早期 IE 占有大量的市场份额,所以这种内核比较流行,以前有很多网页也是根据这个内核的标准来编写的,但是实际上这个内核对真正的网页标准支持不是很好。但是由于 IE 的高市场占有率,微软也很长时间没有更新 Trident 内核,就导致了 Trident 内核和 W3C 标准脱节。还有就是 Trident 内核的大量 Bug 等安全问题没有得到解决,加上一些专家学者公开自己认为 IE 浏览器不安全的观点,使很多用户开始转向其他浏览器。

Gecko:这是 Firefox 和 Flock 所采用的内核,这个内核的优点就是功能强大、丰富,可以支持很多复杂网页效果和浏览器扩展接口,但是代价是也显而易见就是要消耗很多的资源,比如内存。

Presto:Opera 曾经采用的就是 Presto 内核,Presto 内核被称为公认的浏览网页速度最快的内核,这得益于它在开发时的天生优势,在处理 JS 脚本等脚本语言时,会比其他的内核快 3 倍左右,缺点就是为了达到很快的速度而丢掉了一部分网页兼容性。

Webkit:Webkit 是 Safari 采用的内核,它的优点就是网页浏览速度较快,虽然不及 Presto 但是也胜于 Gecko 和 Trident,缺点是对于网页代码的容错性不高,也就是说对网页代码的兼容性较低,会使一些编写不标准的网页无法正确显示。WebKit 前身是 KDE 小组的 KHTML 引擎,可以说 WebKit 是 KHTML 的一个开源的分支。

Blink:谷歌在 Chromium Blog 上发表博客,称将与苹果的开源浏览器核心 Webkit 分道扬镳,在 Chromium 项目中研发 Blink 渲染引擎(即浏览器核心),内置于 Chrome 浏览器之中。其实 Blink 引擎就是 Webkit 的一个分支,就像 webkit 是 KHTML 的分支一样。Blink 引擎现在是谷歌公司与 Opera Software 共同研发,上面提到

过的，Opera 弃用了自己的 Presto 内核，加入 Google 阵营，跟随谷歌一起研发 Blink。

14. 浏览器的渲染过程

浏览器渲染主要有以下步骤：

首先解析收到的文档，根据文档定义构建一棵 DOM 树，DOM 树是由 DOM 元素及属性节点组成的。

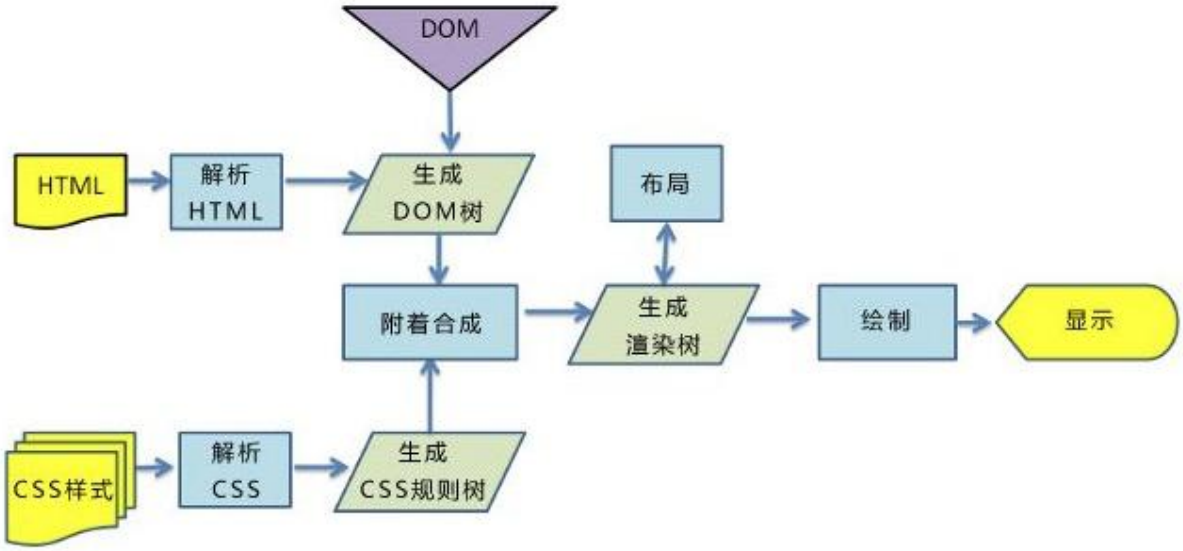
然后对 CSS 进行解析，生成 CSSOM 规则树。

根据 DOM 树和 CSSOM 规则树构建渲染树。渲染树的节点被称为渲染对象，渲染对象是一个包含有颜色和大小等属性的矩形，渲染对象和 DOM 元素相对应，但这种对应关系不是一对一的，不可见的 DOM 元素不会被插入渲染树。还有一些 DOM 元素对应几个可见对象，它们一般是一些具有复杂结构的元素，无法用一个矩形来描述。

当渲染对象被创建并添加到树中，它们并没有位置和大小，所以当浏览器生成渲染树以后，就会根据渲染树来进行布局（也可以叫做回流）。这一阶段浏览器要做的事情是要弄清楚各个节点在页面中的确切位置和大小。通常这一行为也被称为“自动重排”。

布局阶段结束后是绘制阶段，遍历渲染树并调用渲染对象的 paint 方法将它们的内容显示在屏幕上，绘制使用 UI 基础组件。

大致过程如图所示：



注意：这个过程是逐步完成的，为了更好的用户体验，渲染引擎将会尽可能早的将内容呈现到屏幕上，并不会等到所有的 html 都解析完成之后再去构建和布局 render 树。它是解析完一部分内容就显示一部分内容，同时，可能还在通过网络下载其余内容。

15. 渲染过程中遇到 JS 文件如何处理？

JavaScript 的加载、解析与执行会阻塞文档的解析，也就是说，在构建 DOM 时，HTML 解析器若遇到了 JavaScript，那么它会暂停文档的解析，将控制权移交给 JavaScript 引擎，等 JavaScript 引擎运行完毕，浏览器再从中断的地方恢复继续解析文档。也就是说，如果想要首屏渲染的越快，就越不应该在首屏就加载 JS 文件，这也是都建议将 script 标签放在 body 标签底部的原因。当然在当下，并不是说 script 标签必须放在底部，因为你可以给 script 标签添加 defer 或者 async 属性。

16. 前端储存的方式有哪些？

cookies： 在 HTML5 标准前本地储存的主要方式，优点是兼容性好，请求头自带 cookie方便，缺点是大小只有 4k，自动请求头加入cookie

浪费流量，每个 domain 限制 20 个 cookie，使用起来麻烦，需要自行封装；

localStorage：HTML5 加入的以键值对(Key-Value)为标准的方式，优点是操作方便，永久性储存（除非手动删除），大小为 5M，兼容 IE8+ ；

sessionStorage:与 localStorage 基本类似,区别是 sessionStorage 当页面关闭后会被清理，而且与 cookie、localStorage 不同，他不能在所有同源窗口中共享，是会话级别的储存方式；

Web SQL：2010 年被 W3C 废弃的本地数据库数据存储方案，但是主流浏览器（火狐除外）都已经有了相关的实现，web sql 类似于 SQLite，是真正意义上的关系型数据库,用sql 进行操作,当我们用JavaScript 时要进行转换，较为繁琐；

IndexedDB：是被正式纳入HTML5 标准的数据库储存方案,它是 NoSQL 数据库，用键值对进行储存，可以进行快速读取操作，非常适合 web 场景，同时用JavaScript 进行操作会非常便。

17. 事件是什么？事件模型？

事件是用户操作网页时发生的交互动作，比如 click/move， 事件除了用户触发的动作外，还可以是文档加载，窗口滚动和大小调整。事件被封装成一个 event 对象，包含了该事件发生时的所有相关信息（ event 的属性）以及可以对事件进行的操作（ event 的方法）。

事件是用户操作网页时发生的交互动作或者网页本身的一些操作,现代浏览器一共有三种事件模型：

DOM0 级事件模型，这种模型不会传播，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现,它可以在网页中直接定义

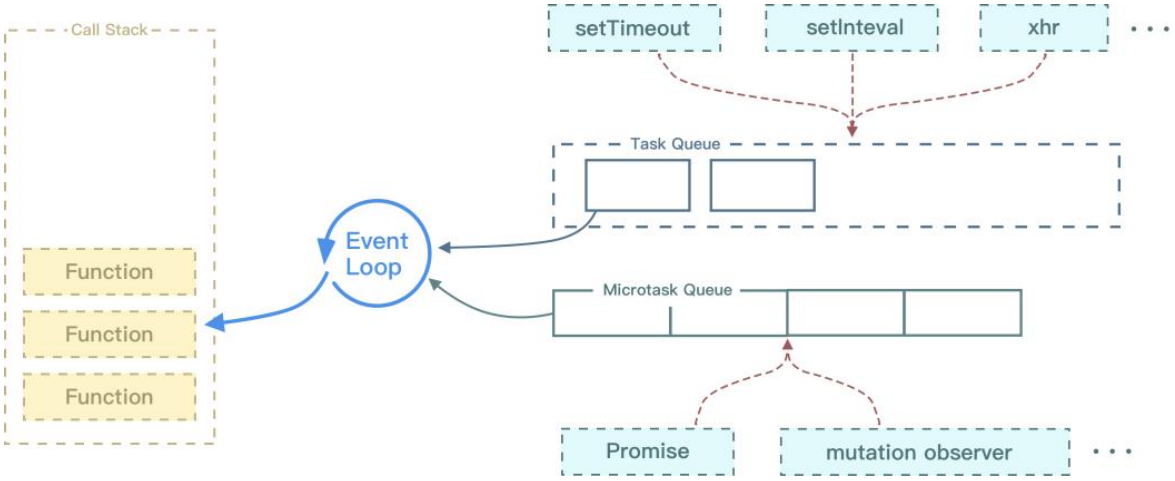
监听函数，也可以通过 js 属性来指定监听函数。所有浏览器都兼容这种方式。直接在 dom 对象上注册事件名称，就是 DOM0 写法。

IE 事件模型，在该事件模型中，一次事件共有两个过程，事件处理阶段和事件冒泡阶段。事件处理阶段会首先执行目标元素绑定的监听事件。然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到 document，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。这种模型通过 attachEvent 来添加监听函数，可以添加多个监听函数，会按顺序依次执行。

DOM2 级事件模型，在该事件模型中，一次事件共有三个过程，第一个过程是事件捕获阶段。捕获指的是事件从 document 一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和 IE 事件模型的两个阶段相同。这种事件模型，事件绑定的函数是 addEventListener，其中第三个参数可以指定事件是否在捕获阶段执行。

18. 对事件循环的理解

因为 js 是单线程运行的，在代码执行时，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。在执行同步代码时，如果遇到异步事件，js 引擎并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务。当异步事件执行完毕后，再将异步事件对应的回调加入到一个任务队列中等待执行。任务队列可以分为宏任务队列和微任务队列，当当前执行栈中的事件执行完毕后，js 引擎首先会判断微任务队列中是否有任务可以执行，如果有就将微任务队首的事件压入栈中执行。当微任务队列中的任务都执行完成后再去执行宏任务队列中的任务。



Event Loop 执行顺序如下所示：

首先执行同步代码，这属于宏任务

当执行完所有同步代码后，执行栈为空，查询是否有异步代码需要执行

执行所有微任务

当执行完所有微任务后，如有必要会渲染页面

然后开始下一轮 Event Loop，执行宏任务中的异步代码

计算机网络部分

1. GET 和 POST 的请求的区别

Post 和 Get 是 HTTP 请求的两种方法，其区别如下：

应用场景：GET 请求是一个幂等的请求，一般 Get 请求用于对服务器资源不会产生影响的场景，比如说请求一个网页的资源。而 Post 不是一个幂等的请求，一般用于对服务器资源会产生影响的情景，比如注册用户这一类的操作。

是否缓存：因为两者应用场景不同，浏览器一般会对 Get 请求缓存，但很少对 Post 请求缓存。

发送的报文格式：Get 请求的报文中实体部分为空，Post 请求的报文中实体部分一般为向服务器发送的数据。

安全性：Get 请求可以将请求的参数放入 url 中向服务器发送，这样的做法相对于 Post 请求来说是不太安全的，因为请求的 url 会被保留在历史记录中。

请求长度：浏览器由于对 url 长度的限制，所以会影响 get 请求发送数据时的长度。这个限制是浏览器规定的，并不是 RFC 规定的。

参数类型：post 的参数传递支持更多的数据类型。

2. POST 和 PUT 请求的区别

PUT 请求是向服务器端发送数据，从而修改数据的内容，但是不会增加数据的种类等，也就是说无论进行多少次 PUT 操作，其结果并没有不同。（可以理解为时更新数据）

POST 请求是向服务器端发送数据，该请求会改变数据的种类等资源，它会创建新的内容。（可以理解为是创建数据）

3. 常见的 HTTP 请求头和响应头

HTTP Request Header 常见的请求头：

Accept:浏览器能够处理的内容类型

Accept-Charset:浏览器能够显示的字符集

Accept-Encoding：浏览器能够处理的压缩编码

Accept-Language：浏览器当前设置的语言

Connection：浏览器与服务器之间连接的类型

Cookie：当前页面设置的任何 Cookie

Host：发出请求的页面所在的域

Referer：发出请求的页面的 URL

User-Agent：浏览器的用户代理字符串

HTTP Responses Header 常见的响应头：

Date：表示消息发送的时间，时间的描述格式由 rfc822 定义

server:服务器名称

Connection：浏览器与服务器之间连接的类型

Cache-Control：控制 HTTP 缓存

content-type:表示后面的文档属于什么 MIME 类型

常见的 Content-Type 属性值有以下四种：

（1）application/x-www-form-urlencoded：浏览器的原生 form 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。该种方式提交的数据放在 body 里面，数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL 转码。

（2）multipart/form-data：该种方式也是一个常见的 POST 提交方式，通常表单上传文件时使用该种方式。

（3）application/json：服务器消息主体是序列化后的 JSON 字符串。

（4）text/xml：该种方式主要用来提交 XML 格式的数据。

4. 常见的 HTTP 请求方法

GET: 向服务器获取数据；

POST：将实体提交到指定的资源，通常会造成服务器资源的修改；

PUT：上传文件，更新数据；

DELETE：删除服务器上的对象；

HEAD：获取报文首部，与 GET 相比，不返回报文主体部分；

OPTIONS：询问支持的请求方法，用来跨域请求；

CONNECT：要求在与代理服务器通信时建立隧道，使用隧道进行 TCP 通信；

TRACE: 回显服务器收到的请求，主要用于测试或诊断。

5. HTTP 1.1 和 HTTP 2.0 的区别

二进制协议：HTTP/2 是一个二进制协议。在 HTTP/1.1 版中，报文的头信息必须是文本（ASCII 编码），数据体可以是文本，也可以是二进制。HTTP/2 则是一个彻底的二进制协议，头信息和数据体都是二进制，并且统称为"帧"，可以分为头信息帧和数据帧。 帧的概念是它实现多路复用的基础。

多路复用：HTTP/2 实现了多路复用，HTTP/2 仍然复用 TCP 连接，但是在一个连接里，客户端和服务器都可以同时发送多个请求或回应，而且不用按照顺序一一发送，这样就避免了"队头堵塞"的问题。

数据流：HTTP/2 使用了数据流的概念，因为 HTTP/2 的数据包是不按顺序发送的，同一个连接里面连续的数据包，可能属于不同的请求。因此，必须要对数据包做标记，指出它属于哪个请求。HTTP/2 将每个请求或回应的所有数据包，称为一个数据流。每个数据流都有一个独一无二的编号。数据包发送时，都必须标记数据流 ID ，用来区分它属于哪个数据流。

头信息压缩：HTTP/2 实现了头信息压缩，由于 HTTP 1.1 协议不带状态，每次请求都必须附上所有信息。所以，请求的很多字段都是重复的，比如 Cookie 和 User Agent ，一模一样的内容，每次请求都必须附带，这会浪费很多带宽，也影响速度。HTTP/2 对这一点做了优化，引入了头信息压缩机制。一方面，头信息使用 gzip 或 compress 压缩后再发送；另一方面，客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就能提高速度了。

服务器推送：HTTP/2 允许服务器未经请求，主动向客户端发送资源，这叫做服务器推送。使用服务器推送提前给客户端推送必要的资源，这样就可以相对减少一些延迟时间。这里需要注意的是 http2 下服务器主动推送的是静态资源，和 WebSocket 以及使用 SSE 等方式向客户端发送即时数据的推送是不同的。

6. HTTP 和 HTTPS 协议的区别

HTTP 和 HTTPS 协议的主要区别如下：

HTTPS 协议需要 CA 证书，费用较高；而 HTTP 协议不需要；

HTTP 协议是超文本传输协议，信息是明文传输的，HTTPS 则是具有安全性的 SSL 加密传输协议；

使用不同的连接方式，端口也不同，HTTP 协议端口是 80，HTTPS 协议端口是 443；

HTTP 协议连接很简单，是无状态的；HTTPS 协议是有 SSL 和 HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 HTTP 更加安全。

7. HTTP2 的头部压缩算法是怎样的？

HTTP2 的头部压缩是 HPACK 算法。在客户端和服务器两端建立“字典”，用索引号表示重复的字符串，采用哈夫曼编码来压缩整数和字符串，可以达到 50%~90%的高压缩率。

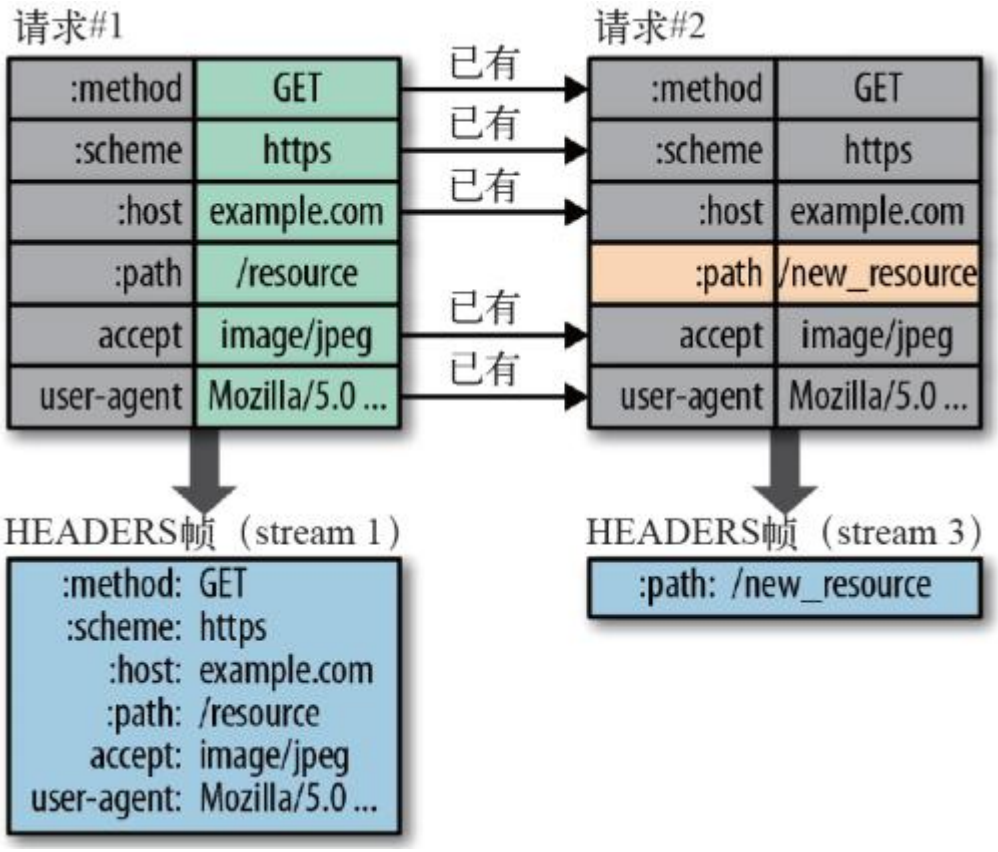
具体来说:

在客户端和服务器端使用“首部表”来跟踪和存储之前发送的键值对，对于相同的数据，不再通过每次请求和响应发送；

首部表在 HTTP/2 的连接存续期内始终存在，由客户端和服务器共同渐进地更新；

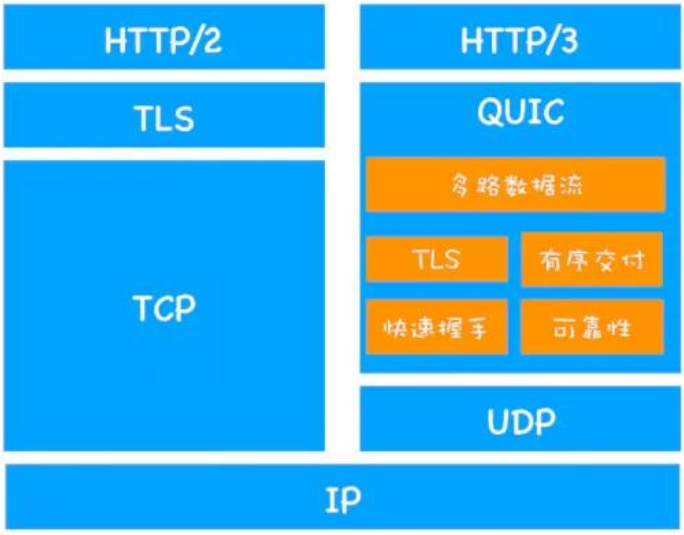
每个新的首部键值对要么被追加到当前表的末尾，要么替换表中之前的值。

例如下图中的两个请求， 请求一发送了所有的头部字段，第二个请求则只需要发送差异数据，这样可以减少冗余数据，降低开销。



8. 说一下 HTTP 3.0

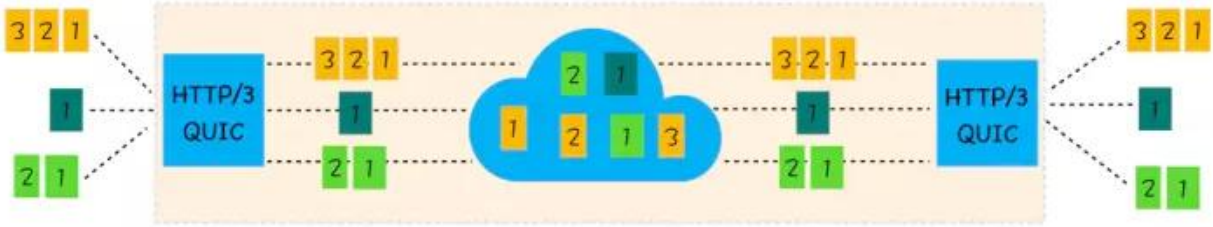
HTTP/3 基于 UDP 协议实现了类似于 TCP 的多路复用数据流、传输可靠性等功能，这套功能被称为 QUIC 协议。



1.流量控制、传输可靠性功能：QUIC 在 UDP 的基础上增加了一层来保证数据传输可靠性，它提供了数据包重传、拥塞控制、以及其他一些 TCP 中的特性。

2.集成 TLS 加密功能：目前 QUIC 使用 TLS1.3，减少了握手所花费的 RTT 数。

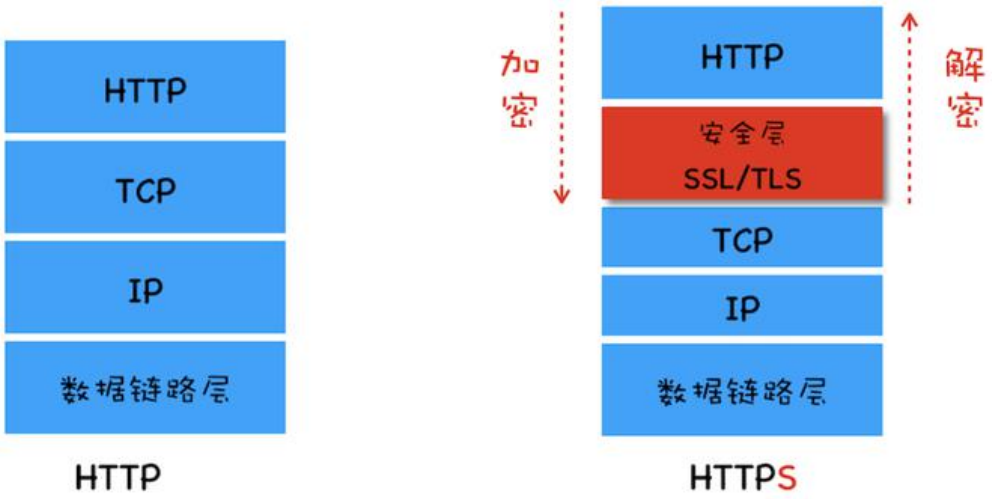
3.多路复用：同一物理连接上可以有多个独立的逻辑数据流，实现了数据流的单独传输，解决了 TCP 的队头阻塞问题。



4.快速握手：由于基于 UDP，可以实现使用 0～1 个 RTT 来建立连接。

9. 什么是 HTTPS 协议？

超文本传输安全协议（Hypertext Transfer Protocol Secure，简称：HTTPS）是一种通过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，利用 SSL/TLS 来加密数据包。HTTPS 的主要目的是提供对网站服务器的身份认证，保护交换数据的隐私与完整性。



HTTP 协议采用明文传输信息，存在信息窃听、信息篡改和信息劫持的风险，而协议 TLS/SSL 具有身份验证、信息加密和完整性校验的功能，可以避免此类问题发生。

安全层的主要职责就是对发起的 HTTP 请求的数据进行加密操作 和对接收到的 HTTP 的内容进行解密操作。

10. HTTPS 通信（握手）过程

HTTPS 的通信过程如下：

1.客户端向服务器发起请求，请求中包含使用的协议版本号、生成的一个随机数、以及客户端支持的加密方法。

2.服务器端接收到请求后，确认双方使用的加密方法、并给出服务器的证书、以及一个服务器生成的随机数。

3.客户端确认服务器证书有效后，生成一个新的随机数，并使用数字证书中的公钥，加密这个随机数，然后发给服 务器。并且还会提供一个前面所有内容的 hash 的值，用来供服务器检验。

4.服务器使用自己的私钥，来解密客户端发送过来的随机数。并提供前面所有内容的 hash 值来供客户端检验。

5.客户端和服务器端根据约定的加密方法使用前面的三个随机数，生成对话秘钥，以后的对话过程都使用这个秘钥来加密信息。

11. DNS 完整的查询过程

DNS 服务器解析域名的过程：

首先会在浏览器的缓存中查找对应的 IP 地址，如果查找到直接返回，若找不到继续下一步

将请求发送给本地 DNS 服务器，在本地域名服务器缓存中查询，如果查找到，就直接将查找结果返回，若找不到继续下一步

本地 DNS 服务器向根域名服务器发送请求，根域名服务器会返回一个所查询域的顶级域名服务器地址

本地 DNS 服务器向顶级域名服务器发送请求，接受请求的服务器查询自己的缓存，如果有记录，就返回查询结果，如果没有就返回相关的下一级的权威域名服务器的地址

本地 DNS 服务器向权威域名服务器发送请求，域名服务器返回对应的结果

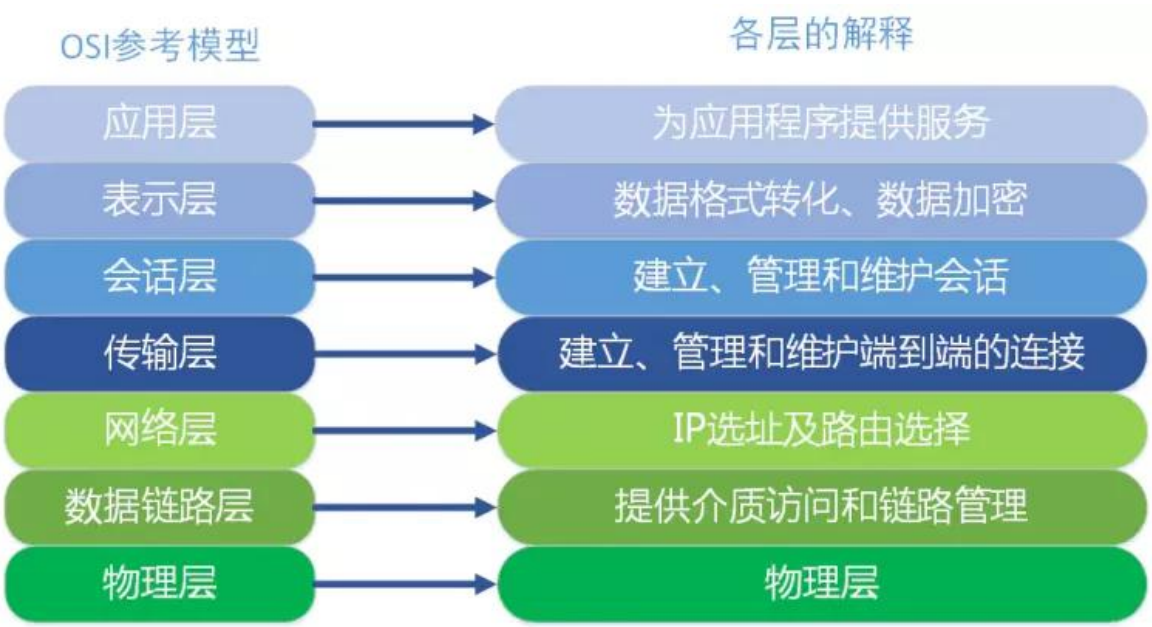
本地 DNS 服务器将返回结果保存在缓存中，便于下次使用

本地 DNS 服务器将返回结果返回给浏览器

比如要查询 www.baidu.com 的 IP 地址，首先会在浏览器的缓存中查找是否有该域名的缓存，如果不存在就将请求发送到本地的 DNS 服务器中，本地 DNS 服务器会判断是否存在该域名的缓存，如果不存在，则向根域名服务器发送一个请求，根域名服务器返回负责 .com 的顶级域名服务器的 IP 地址的列表。然后本地 DNS 服务器再向其中一个负责 .com 的顶级域名服务器发送一个请求，负责 .com 的顶级域名服务器返回负责 .baidu 的权威域名服务器的 IP 地址列表。然后本地 DNS 服务器再向其中一个权威域名服务器发送一个请求，最后权威域名服务器返回一个对应的主机名的 IP 地址列表。

12. OSI 七层模型

ISO 为了更好的使网络应用更为普及，推出了 OSI 参考模型。



（1）应用层

OSI 参考模型中最靠近用户的一层，是为计算机用户提供应用接口，也为用户直接提供各种网络服务。我们常见应用层的网络服务协议有：HTTP，HTTPS，FTP，POP3、SMTP 等。

在客户端与服务器中经常会有数据的请求，这个时候就是会用到 http(hyper text transfer protocol)(超文本传输协议)或者 https.在后端设计数据接口时，我们常常使用到这个协议。

FTP 是文件传输协议，在开发过程中，个人并没有涉及到，但是我想，在一些资源网站，比如百度网盘``迅雷应该是基于此协议的。

SMTP 是 simple mail transfer protocol（简单邮件传输协议）。在一个项目中，在用户邮箱验证码登录的功能时，使用到了这个协议。

（2）表示层

表示层提供各种用于应用层数据的编码和转换功能,确保一个系统的应用层发送的数据能被另一个系统的应用层识别。如果必要，该层可提供一种标准表示形式,用于将计算机内部的多种数据格式转换成通信中采用的标准表示形式。数据压缩和加密也是表示层可提供的转换功能之一。

在项目开发中，为了方便数据传输，可以使用 base64 对数据进行编解码。如果按功能来划分，base64 应该是工作在表示层。

（3）会话层

会话层就是负责建立、管理和终止表示层实体之间的通信会话。该层的通信由不同设备中的应用程序之间的服务请求和响应组成。

（4）传输层

传输层建立了主机端到端的链接，传输层的作用是为上层协议提供端到端的可靠和透明的数据传输服务，包括处理差错控制和流量控制等问题。该层向高层屏蔽了下层数据通信的细节，使高层用户看到的只是在两个传输实体间的一条主机到主机的、可由用户控制和设定的、可靠的数据通路。我们通常说的，TCP UDP 就是在这一层。端口号既是这里的“端”。

（5）网络层

本层通过 IP 寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点，正确无误地按照地址传送给目的端的运输层。就是通常说的 IP 层。这一层就是我们经常说的 IP 协议层。IP 协议是 Internet 的基础。我们可以这样理解，网络层规定了数据包的传输路线，而传输层则规定了数据包的传输方式。

（6）数据链路层

将比特组合成字节,再将字节组合成帧,使用链路层地址 (以太网使用 MAC 地址)来访问介质,并进行差错检测。

网络层与数据链路层的对比，通过上面的描述，我们或许可以这样理解,网络层是规划了数据包的传输路线,而数据链路层就是传输路线。不过，在数据链路层上还增加了差错控制的功能。

（7）物理层

实际最终信号的传输是通过物理层实现的。通过物理介质传输比特流。规定了电平、速度和电缆针脚。常用设备有（各种物理设备）集线器、中继器、调制解调器、网线、双绞线、同轴电缆。这些都是物理层的传输介质。

OSI 七层模型通信特点：对等通信

对等通信，为了使数据分组从源传送到目的地，源端 OSI 模型的每一层都必须与目的端的对等层进行通信，这种通信方式称为对等层通信。在每一层通信过程中，使用本层自己协议进行通信。

13. TCP 的三次握手和四次挥手

（1）三次握手

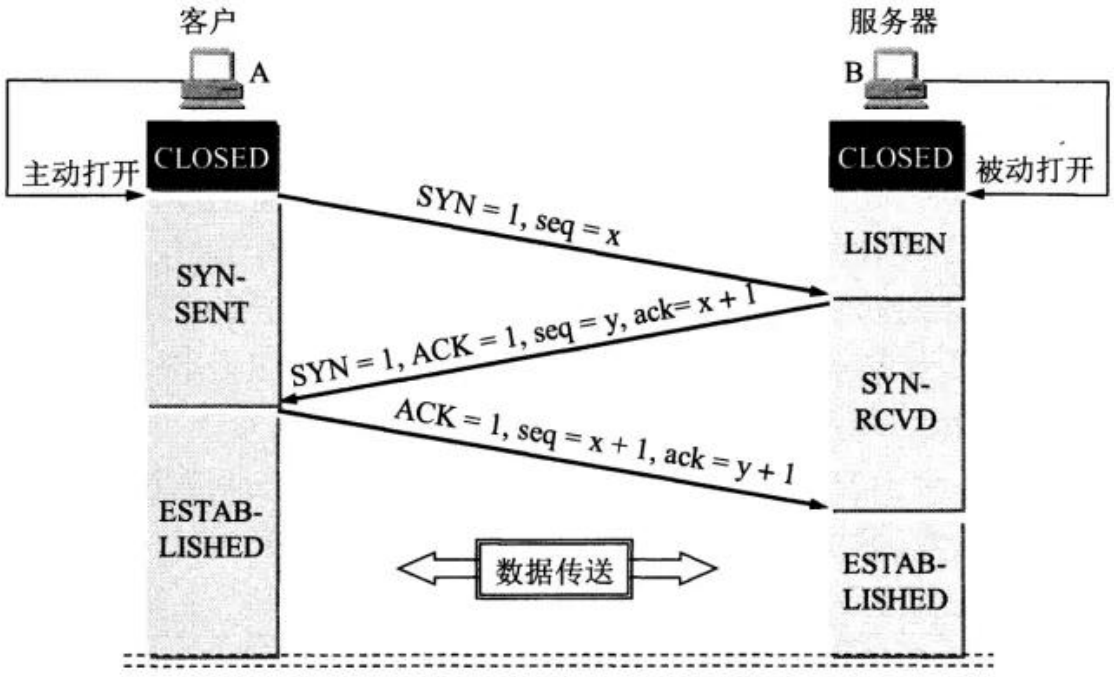


图 5-28 用三报文握手建立 TCP 连接

三次握手（Three-way Handshake）其实就是指建立一个 TCP 连接时，需要客户端和服务器总共发送 3 个包。进行三次握手的主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化序列号为后面的可靠性传送做准备。实质上其实就是连接服务器指定端口，建立 TCP 连接，并同步连接双方的序列号和确认号，交换 TCP 窗口大小信息。

刚开始客户端处于 Closed 的状态，服务端处于 Listen 状态。

第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 ISN，此时客户端处于 SYN_SEND 状态。

首部的同步位 SYN=1，初始序号 seq=x，SYN=1 的报文段不能携带数据，但要消耗掉一个序号。

第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN。同时会把客户端的 ISN + 1 作为ACK 的值,表示自己已经收到了客户端的 SYN，此时服务器处于 SYN_REVD 的状态。

在确认报文段中 SYN=1，ACK=1，确认号 ack=x+1，初始序号 seq=y

第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN + 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立起了连接。

确认报文段 ACK=1，确认号 ack=y+1，序号 seq=x+1（初始为 seq=x，第二个报文段所以要+1），ACK 报文段可以携带数据，不携带数据则不消耗序号。

那为什么要三次握手呢？两次不行吗？

为了确认双方的接收能力和发送能力都正常

如果是用两次握手，则会出现下面这种情况：

如客户端发出连接请求，但因连接请求报文丢失而未收到确认，于是客户端再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接，客户端共发出了两个连接请求报文段，其

中第一个丢失，第二个到达了服务端，但是第一个丢失的报文段只是在某些网络结点长时间滞留了，延误到连接释放以后的某个时间才到达服务端，此时服务端误认为客户端又发出一次新的连接请求，于是就向客户端发出确认报文段，同意建立连接，不采用三次握手，只要服务端发出确认，就建立新的连接了，此时客户端忽略服务端发来的确认，也不发送数据，则服务端一致等待客户端发送数据，浪费资源。

简单来说就是以下三步：

第一次握手：客户端向服务端发送连接请求报文段。该报文段中包含自身的数据通讯初始序号。请求发送后，客户端便进入 SYN-SENT 状态。

第二次握手：服务端收到连接请求报文段后，如果同意连接，则会发送一个应答，该应答中也会包含自身的数据通讯初始序号，发送完成后便进入 SYN-RECEIVED 状态。

第三次握手：当客户端收到连接同意的应答后，还要向服务端发送一个确认报文。客户端发完这个报文段后便进入 ESTABLISHED 状态，服务端收到这个应答后也进入 ESTABLISHED 状态，此时连接建立成功。

TCP 三次握手的建立连接的过程就是相互确认初始序号的过程，告诉对方，什么样序号的报文段能够被正确接收。 第三次握手的作用是客户端对服务器端的初始序号的确认。如果只使用两次握手，那么服务器就没有办法知道自己的序号是否 已被确认。同时这样也是为了防止失效的请求报文段被服务器接收，而出现错误的情况。

（2）四次挥手

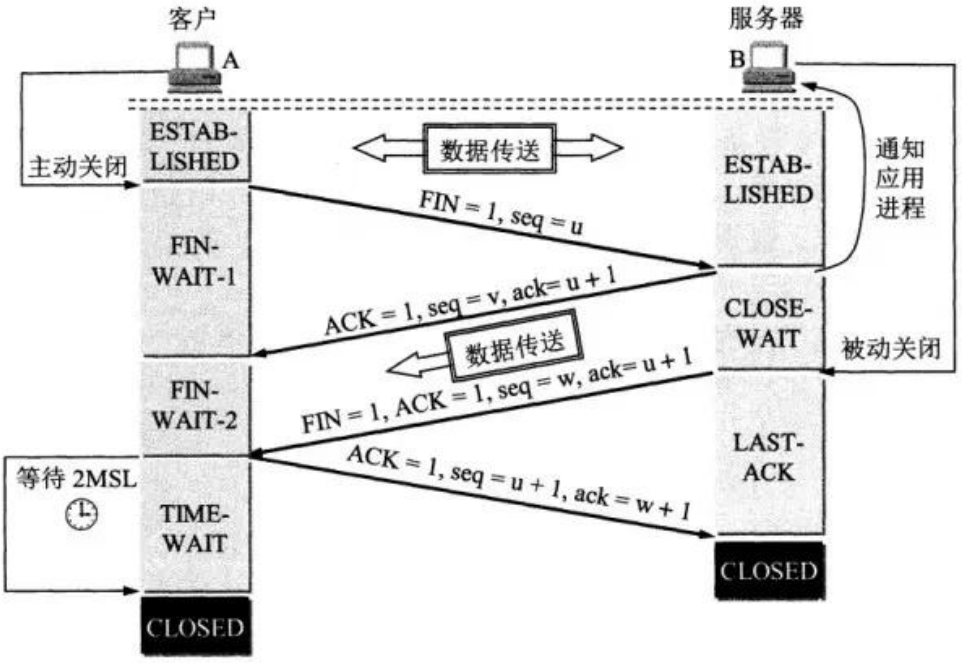


图 5-29　TCP 连接释放的过程

刚开始双方都处于 ESTABLISHED 状态，假如是客户端先发起关闭请求。四次挥手的过程如下：

第一次挥手： 客户端会发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于 FIN_WAIT1 状态。

即发出连接释放报文段（FIN=1，序号 seq=u），并停止再发送数据，主动关闭 TCP 连接，进入 FIN_WAIT1（终止等待 1）状态，等待服务端的确认。

第二次挥手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 +1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 CLOSE_WAIT 状态。

即服务端收到连接释放报文段后即发出确认报文段（ACK=1，确认号 ack=u+1，序号 seq=v），服务端进入 CLOSE_WAIT（关闭等待）状态，此时的 TCP 处于半关闭状态，客户端到服务端的连接释放。客户端收

到服务端的确认后，进入 FIN_WAIT2（终止等待 2）状态，等待服务端发出的连接释放报文段。

第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 LAST_ACK 的状态。

即服务端没有要向客户端发出的数据，服务端发出连接释放报文段（FIN=1，ACK=1，序号 seq=w，确认号 ack=u+1），服务端进入 LAST_ACK（最后确认）状态，等待客户端的确认。

第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 +1 作为自己 ACK 报文的序列号值，此时客户端处于 TIME_WAIT 状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态，服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。

即客户端收到服务端的连接释放报文段后，对此发出确认报文段（ACK=1，seq=u+1，ack=w+1），客户端进入 TIME_WAIT（时间等待）状态。此时 TCP 未释放掉，需要经过时间等待计时器设置的时间 2MSL 后，客户端才进入 CLOSED 状态。

那为什么需要四次挥手呢？

因为当服务端收到客户端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中 ACK 报文是用来应答的，SYN 报文是用来同步的。但是关闭连接时，当服务端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉客户端，“你发的 FIN 报文我收到了”。只有等到我服务端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送，故需要四次挥手。

简单来说就是以下四步：

第一次挥手：若客户端认为数据发送完成，则它需要向服务端发送连接释放请求。

第二次挥手：服务端收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进入 CLOSE_WAIT 状态，此时表明客户端到服务端的连接已经释放，不再接收客户端发的数据了。但是因为 TCP 连接是双向的，所以服务端仍旧可以发送数据给客户端。

第三次挥手：服务端如果此时还有没发完的数据会继续发送，完毕后会向客户端发送连接释放请求，然后服务端便进入 LAST-ACK 状态。

第四次挥手：客户端收到释放请求后，向服务端发送确认应答，此时客户端进入 TIME-WAIT 状态。该状态会持续 2MSL（最大段生存期，指报文段在网络中生存的时间，超时会被抛弃） 时间，若该时间段内没有服务端的重发请求的话，就进入 CLOSED 状态。当服务端收到确认应答后，也便进入 CLOSED 状态。

TCP 使用四次挥手的原因是因为 TCP 的连接是全双工的，所以需要双方分别释放到对方的连接，单独一方的连接释放，只代 表不能再向对方发送数据，连接处于的是半释放的状态。

最后一次挥手中，客户端会等待一段时间再关闭的原因，是为了防止发送给服务器的确认报文段丢失或者出错，从而导致服务器 端不能正常关闭。

HTML 部分

1. 对 HTML 语义化的理解

语义化是指根据内容的结构化（内容语义化），选择合适的标签（代码语义化）。通俗来讲就是用正确的标签做正确的事情。

语义化的优点如下：

对机器友好，带有语义的文字表现力丰富，更适合搜索引擎的爬虫爬取有效信息，有利于 SEO。除此之外，语义类还支持读屏软件，根据文章可以自动生成目录；

对开发者友好，使用语义类标签增强了可读性，结构更加清晰，开发者能清晰的看出网页的结构，便于团队的开发与维护。

常见的语义化标签：

```
<header></header> 头部

<nav></nav> 导航栏

<section></section> 区块（有语义化的div）

<main></main> 主要区域

<article></article> 主要内容

<aside></aside> 侧边栏

<footer></footer> 底部
```

2. DOCTYPE(文档类型) 的作用

DOCTYPE 是 HTML5 中一种标准通用标记语言的文档类型声明，它的目的是告诉浏览器（解析器）应该以什么样（html 或 xhtml）的文档类型定义来解析文档，不同的渲染模式会影响浏览器对 CSS 代码甚至 JavaScript 脚本的解析。它必须声明在 HTML文档的第一行。

浏览器渲染页面的两种模式（可通过 document.compatMode 获取，比如，语雀官网的文档类型是 CSS1Compat）：

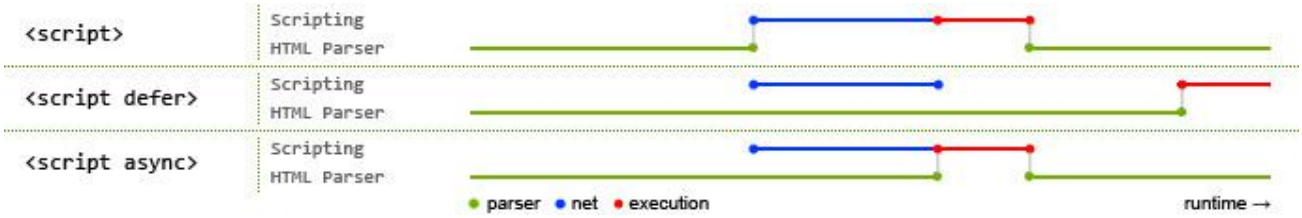
CSS1Compat：标准模式（Strick mode），默认模式，浏览器使用 W3C 的标准解析渲染页面。在标准模式中，浏览器以其支持的最高标准呈现页面。

BackCompat：怪异模式(混杂模式)(Quick mode)，浏览器使用自己的怪异模式解析渲染页面。在怪异模式中，页面以一种比较宽松的向后兼容的方式显示。

3. script 标签中 defer 和 async 的区别

如果没有 defer 或 async 属性,浏览器会立即加载并执行相应的脚本。它不会等待后续加载的文档元素，读取到就会开始加载和执行，这样就阻塞了后续文档的加载。

下图可以直观的看出三者之间的区别:



其中蓝色代表 js 脚本网络加载时间，红色代表 js 脚本执行时间，绿色代表 html 解析。

defer 和 async 属性都是去异步加载外部的 JS 脚本文件，它们都不会阻塞页面的解析，其区别如下：

执行顺序：多个带 async 属性的标签，不能保证加载的顺序；多个带 defer 属性的标签，按照加载顺序执行；

脚本是否并行执行：async 属性，表示后续文档的加载和执行与 js 脚本的加载和执行是并行进行的，即异步执行；defer 属性，加载后续文档的过程和 js 脚本的加载(此时仅加载不执行)是并行进行的

(异步)，js 脚本需要等到文档所有元素解析完成之后才执行，DOMContentLoaded 事件触发执行之前。

4. 行内元素有哪些？块级元素有哪些？ 空(void)元素有那些？

行内元素有：a b span img input select strong;

块级元素有：div ul ol li dl dt dd h1 h2 h3 h4 h5 h6 p;

空元素，即没有内容的 HTML 元素。空元素是在开始标签中关闭的，也就是空元素没有闭合标签：

常见的有：
、<hr>、<img>、<input>、<link>、<meta>;

鲜见的有:<area>、<base>、<col>、<colgroup>、<command>、<embed>、<keygen>、<param>、<source>、<track>、<wbr>。

5. 浏览器是如何对 HTML5 的离线储存资源进行管理和加载？

在线的情况下，浏览器发现 html 头部有 manifest 属性，它会请求 manifest 文件，如果是第一次访问页面 ，那么浏览器就会根据 manifest 文件的内容下载相应的资源并且进行离线存储。如果已经访问过页面并且资源已经进行离线存储了，那么浏览器就会使用离线的资源加载页面，然后浏览器会对比新的 manifest 文件与旧的 manifest 文件，如果文件没有发生改变，就不做任何操作，如果文件改变了，就会重新下载文件中的资源并进行离线存储。

离线的情况下，浏览器会直接使用离线存储的资源。

6. Canvas 和 SVG 的区别

（1）SVG:

SVG 可缩放矢量图形（Scalable Vector Graphics）是基于可扩展标记语言 XML 描述的 2D 图形的语言，SVG 基于 XML 就意味着 SVG DOM 中的每个元素都是可用的，可以为某个元素附加 Javascript 事件处理器。在 SVG 中，每个被绘制的图形均被视为对象。如果 SVG 对象的属性发生变化，那么浏览器能够自动重现图形。

其特点如下：

不依赖分辨率

支持事件处理器

最适合带有大型渲染区域的应用程序（比如谷歌地图）

复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）

不适合游戏应用

（2）Canvas：

Canvas 是画布，通过 Javascript 来绘制 2D 图形，是逐像素进行渲染的。其位置发生改变，就会重新进行绘制。

其特点如下：

依赖分辨率

不支持事件处理器

弱的文本渲染能力

能够以 .png 或 .jpg 格式保存结果图像

最适合图像密集型的游戏，其中的许多对象会被频繁重绘

注：矢量图，也称为面向对象的图像或绘图图像，在数学上定义为一系列由线连接的点。矢量文件中的图形元素称为对象。每个对象都是

一个自成一体的实体，它具有颜色、形状、轮廓、大小和屏幕位置等属性。

7. 说一下 HTML5 drag API

dragstart：事件主体是被拖放元素，在开始拖放被拖放元素时触发。

darg：事件主体是被拖放元素，在正在拖放被拖放元素时触发。

dragenter：事件主体是目标元素，在被拖放元素进入某元素时触发。

dragover：事件主体是目标元素，在被拖放在某元素内移动时触发。

dragleave：事件主体是目标元素，在被拖放元素移出目标元素是触发。

drop:事件主体是目标元素,在目标元素完全接受被拖放元素时触发。

dragend：事件主体是被拖放元素，在整个拖放操作结束时触发。

CSS 部分

1. display 的 block、inline 和 inline-block 的区别

（1）block：会独占一行，多个元素会另起一行，可以设置 width、height、margin 和 padding 属性；

（2）inline：元素不会独占一行，设置 width、height 属性无效。但可以设置水平方向的 margin 和 padding 属性，不能设置垂直方向的 padding 和 margin；

（3）inline-block：将对象设置为 inline 对象，但对象的内容作为 block 对象呈现，之后的内联对象会被排列在同一行内。

对于行内元素和块级元素，其特点如下：

（1）行内元素

设置宽高无效；

可以设置水平方向的 margin 和 padding 属性，不能设置垂直方向的 padding 和 margin；

不会自动换行；

（2）块级元素

可以设置宽高；

设置 margin 和 padding 都有效；

可以自动换行；

多个块状，默认排列从上到下。

2. link 和@import 的区别

两者都是外部引用 CSS 的方式，它们的区别如下：

link 是 XHTML 标签，除了加载 CSS 外，还可以定义 RSS 等其他事务；@import 属于 CSS 范畴，只能加载 CSS。

link 引用 CSS 时，在页面载入时同时加载；@import 需要页面网页完全载入以后加载。

link 是 XHTML 标签，无兼容问题；@import 是在 CSS2.1 提出的，低版本的浏览器不支持。

link 支持使用 Javascript 控制 DOM 去改变样式；而@import 不支持。

3. CSS3 中有哪些新特性

新增各种 CSS 选择器 （: not(.input)：所有 class 不是“input”的节点）

圆角 （border-radius:8px）

多列布局 （multi-column layout）

阴影和反射 （Shadoweflect）

文字特效 （text-shadow）

文字渲染 （Text-decoration）

线性渐变 （gradient）

旋转 （transform）

增加了旋转,缩放,定位,倾斜,动画,多背景

4. 对 CSSSprites 的理解

CSSSprites（精灵图），将一个页面涉及到的所有图片都包含到一张大图中去,然后利用 CSS 的 background-image,background-repeat,background-position 属性的组合进行背景定位。

优点：

利用 CSS Sprites 能很好地减少网页的 http 请求，从而大大提高了页面的性能，这是 CSS Sprites 最大的优点；

CSS Sprites 能减少图片的字节，把 3 张图片合并成 1 张图片的字节总是小于这 3 张图片的字节总和。

缺点：

在图片合并时，要把多张图片有序的、合理的合并成一张图片，还要留好足够的空间，防止板块内出现不必要的背景。在宽屏及高分辨率下的自适应页面，如果背景不够宽，很容易出现背景断裂；

CSSSprites 在开发的时候相对来说有点麻烦，需要借助 photoshop 或其他工具来对每个背景单元测量其准确的位置。

维护方面：CSS Sprites 在维护的时候比较麻烦，页面背景有少许改动时，就要改这张合并的图片，无需改的地方尽量不要动，这样避免改动更多的 CSS，如果在原来的地方放不下，又只能（最好）往下加图片，这样图片的字节就增加了，还要改动 CSS。

5. CSS 优化和提高性能的方法有哪些？

加载性能：

（1）css 压缩：将写好的 css 进行打包压缩，可以减小文件体积。

（2）css 单一样式：当需要下边距和左边距的时候，很多时候会选择使用 margin:top 0 bottom 0；但 margin-bottom:bottom;margin-left:left;执行效率会更高。

（3）减少使用@import，建议使用 link，因为后者在页面加载时一起加载，前者是等待页面加载完成之后再进行加载。

选择器性能：

（1）关键选择器（key selector）。选择器的最后面的部分为关键选择器（即用来匹配目标元素的部分）。CSS 选择符是从右到左进行匹配的。当使用后代选择器的时候，浏览器会遍历所有子元素来确定是否是指定的元素等等；

（2）如果规则拥有 ID 选择器作为其关键选择器，则不要为规则增加标签。过滤掉无关的规则（这样样式系统就不会浪费时间去匹配它们了）。

（3）避免使用通配规则，如*{}计算次数惊人，只对需要用到的元素进行选择。

（4）尽量少的去对标签进行选择，而是用 class。

（5）尽量少的去使用后代选择器，降低选择器的权重值。后代选择器的开销是最高的，尽量将选择器的深度降到最低，最高不要超过三层，更多的使用类来关联每一个标签元素。

（6）了解哪些属性是可以通过继承而来的，然后避免对这些属性重复指定规则。

渲染性能：

（1）慎重使用高性能属性：浮动、定位。

（2）尽量减少页面重排、重绘。

（3）去除空规则：｛｝。空规则的产生原因一般来说是为了预留样式。去除这些空规则无疑能减少 css 文档体积。

（4）属性值为 0 时，不加单位。

（5）属性值为浮动小数 0.**，可以省略小数点之前的 0。

（6）标准化各种浏览器前缀：带浏览器前缀的在前。标准属性在后。

（7）不使用@import 前缀，它会影响 css 的加载速度。

（8）选择器优化嵌套，尽量避免层级过深。

（9）css 雪碧图，同一页面相近部分的小图标，方便使用，减少页面的请求次数，但是同时图片本身会变大，使用时，优劣考虑清楚，再使用。

（10）正确使用 display 的属性，由于 display 的作用，某些样式组合会无效，徒增样式体积的同时也影响解析性能。

（11）不滥用 web 字体。对于中文网站来说 WebFonts 可能很陌生，国外却很流行。web fonts 通常体积庞大，而且一些浏览器在下载 web fonts 时会阻塞页面渲染损伤性能。

可维护性、健壮性：

（1）将具有相同属性的样式抽离出来，整合并通过 class 在页面中进行使用，提高 css 的可维护性。

（2）样式与内容分离：将 css 代码定义到外部 css 中。

6. 对 CSS 工程化的理解

CSS 工程化是为了解决以下问题：

1.宏观设计：CSS 代码如何组织、如何拆分、模块结构怎样设计？

2.编码优化：怎样写出更好的 CSS？

3.构建：如何处理我的 CSS，才能让它的打包结果最优？

4.可维护性：代码写完了，如何最小化它后续的变更成本？如何确保任何一个同事都能轻松接手？

以下三个方向都是时下比较流行的、普适性非常好的 CSS 工程化实践：

预处理器：Less、 Sass 等；

重要的工程化插件： PostCss；

Webpack loader 等 。

基于这三个方向，可以衍生出一些具有典型意义的子问题，这里我们逐个来看：

（1）预处理器：为什么要用预处理器？它的出现是为了解决什么问题？

预处理器，其实就是 CSS 世界的“轮子”。预处理器支持我们写一种类似 CSS、但实际并不是 CSS 的语言，然后把它编译成 CSS 代码：

那为什么写 CSS 代码写得好好的，偏偏要转去写“类 CSS”呢？这就和本来用 JS 也可以实现所有功能，但最后却写 React 的 jsx 或者 Vue 的模板语法一样——为了爽！要想知道有了预处理器有多爽，首先要知道的是传统 CSS 有多不爽。随着前端业务复杂度的提高，前端工程中对 CSS 提出了以下的诉求：

1.宏观设计上：我们希望能优化 CSS 文件的目录结构，对现有的 CSS 文件实现复用；

2.编码优化上：我们希望能写出结构清晰、简明易懂的 CSS，需要它具有一目了然的嵌套层级关系，而不是无差别的一铺到底写法；我们希望它具有变量特征、计算能力、循环能力等等更强的可编程性，这样我们可以少写一些无用的代码；

3.可维护性上：更强的可编程性意味着更优质的代码结构，实现复用意味着更简单的目录结构和更强的拓展能力，这两点如果能做到，自然会带来更强的可维护性。

这三点是传统 CSS 所做不到的，也正是预处理器所解决掉的问题。预处理器普遍会具备这样的特性：

嵌套代码的能力，通过嵌套来反映不同 css 属性之间的层级关系 ；

支持定义 css 变量；

提供计算函数；

允许对代码片段进行 extend 和 mixin；

支持循环语句的使用；

支持将 CSS 文件模块化，实现复用。

（2）PostCss：PostCss 是如何工作的？我们在什么场景下会使用 PostCss？

PostCss 仍然是一个对 CSS 进行解析和处理的工具，它会对 CSS 做这样的事情：

它和预处理器的不同就在于，预处理器处理的是 类 CSS，而 PostCss 处理的就是 CSS 本身。Babel 可以将高版本的 JS 代码转换为低版本的 JS 代码。PostCss 做的是类似的事情：它可以编译尚未被浏览器广泛支持的先进的 CSS 语法，还可以自动为一些需要额外兼容的语法增加前缀。更强的是，由于 PostCss 有着强大的插件机制，支持各种各样的扩展，极大地强化了 CSS 的能力。

PostCss 在业务中的使用场景非常多：

提高 CSS 代码的可读性：PostCss 其实可以做类似预处理器能做的工作；

当我们的 CSS 代码需要适配低版本浏览器时，PostCss 的 Autoprefixer 插件可以帮助我们自动增加浏览器前缀；

允许我们编写面向未来的 CSS：PostCss 能够帮助我们编译 CSS next 代码；

（3）Webpack 能处理 CSS 吗？如何实现？

Webpack 能处理 CSS 吗：

Webpack 在裸奔的状态下，是不能处理 CSS 的，Webpack 本身是一个面向 JavaScript 且只能处理 JavaScript 代码的模块化打包工具；

Webpack 在 loader 的辅助下，是可以处理 CSS 的。

如何用 Webpack 实现对 CSS 的处理：

Webpack 中操作 CSS 需要使用的两个关键的 loader：css-loader 和 style-loader

注意，答出“用什么”有时候可能还不够，面试官会怀疑你是不是在背答案，所以你还需要了解每个 loader 都做了什么事情：

css-loader：导入 CSS 模块，对 CSS 代码进行编译处理；

style-loader：创建 style 标签，把 CSS 内容写入标签。

在实际使用中，css-loader 的执行顺序一定要安排在 style-loader 的前面。因为只有完成了编译过程，才可以对 css 代码进行插入；若提前插入了未编译的代码，那么 webpack 是无法理解这坨东西的，它会无情报错。

7. 常见的 CSS 布局单位

常用的布局单位包括像素（px），百分比（%），em，rem，vw/vh。

（1）像素（px）是页面布局的基础，一个像素表示终端（电脑、手机、平板等）屏幕所能显示的最小的区域，像素分为两种类型：CSS像素和物理像素：

CSS像素：为web开发者提供，在CSS中使用的一个抽象单位；

物理像素：只与设备的硬件密度有关，任何设备的物理像素都是固定的。

（2）百分比（%），当浏览器的宽度或者高度发生变化时，通过百分比单位可以使得浏览器中的组件的宽和高随着浏览器的变化而变化，从而实现响应式的效果。一般认为子元素的百分比相对于直接父元素。

（3）em和rem相对于px更具灵活性，它们都是相对长度单位，它们之间的区别：em相对于父元素，rem相对于根元素。

em： 文本相对长度单位。相对于当前对象内文本的字体尺寸。如果当前行内文本的字体尺寸未被人为设置，则相对于浏览器的默认字体尺寸(默认16px)。(相对父元素的字体大小倍数)。

rem： rem是CSS3新增的一个相对单位，相对于根元素（html元素）的font-size的倍数。作用：利用rem可以实现简单的响应式布局，可以利用html元素中字体的大小与屏幕间的比值来设置font-size的值，以此实现当屏幕分辨率变化时让元素也随之变化。

（4）vw/vh是与视图窗口有关的单位，vw表示相对于视图窗口的宽度，vh表示相对于视图窗口高度，除了vw和vh外，还有vmin和vmax两个相关的单位。

vw：相对于视窗的宽度，视窗宽度是100vw；

vh：相对于视窗的高度，视窗高度是 100vh；

vmin：vw 和 vh 中的较小值；

vmax：vw 和 vh 中的较大值；

vw/vh 和百分比很类似，两者的区别：

百分比（%）：大部分相对于祖先元素，也有相对于自身的情况比如（border-radius、translate 等）

vw/vm：相对于视窗的尺寸

8. 水平垂直居中的实现

利用绝对定位，先将元素的左上角通过 top:50%和 left:50%定位到页面的中心，然后再通过 translate 来调整元素的中心点到页面的中心。该方法需要考虑浏览器兼容问题。

```
.parent {
    position: relative;
}

.child {
    position: absolute;
    left: 50%;
    top: 50%;
    transform: translate(-50%,-50%);
}
```

利用绝对定位，设置四个方向的值都为 0，并将 margin 设置为 auto，由于宽高固定，因此对应方向实现平分，可以实现水平和垂直方向上的居中。该方法适用于盒子有宽高的情况：

```
.parent {
    position: relative;
}

.child {
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    margin: auto;
}
```

利用绝对定位，先将元素的左上角通过 top:50%和 left:50%定位到页面的中心，然后再通过 margin 负值来调整元素的中心点到页面的中心。该方法适用于盒子宽高已知的情况

```
.parent {
    position: relative;
}

.child {
    position: absolute;
    top: 50%;
    left: 50%;
    margin-top: -50px;     /* 自身 height 的一半 */
    margin-left: -50px;    /* 自身 width 的一半 */
}
```

使用 flex 布局，通过 align-items:center 和 justify-content:center 设置容器的垂直和水平方向上为居中对齐，然后它的子元素也可以实现垂直和水平的居中。该方法要考虑兼容的问题，该方法在移动端用的较多：

```
.parent {
    display: flex;
    justify-content:center;
    align-items:center;
}
```

9. 对 BFC 的理解，如何创建 BFC

先来看两个相关的概念：

Box: Box 是 CSS 布局的对象和基本单位，一个页面是由很多个 Box 组成的，这个 Box 就是我们所说的盒模型。

Formatting context：块级上下文格式化，它是页面中的一块渲染区域，并且有一套渲染规则，它决定了其子元素将如何定位，以及和其他元素的关系和相互作用。

块格式化上下文（Block Formatting Context，BFC）是 Web 页面的可视化 CSS 渲染的一部分，是布局过程中生成块级盒子的区域，也是浮动元素与其他元素的交互限定区域。

通俗来讲：BFC 是一个独立的布局环境，可以理解为一个容器，在这个容器中按照一定规则进行物品摆放，并且不会影响其它环境中的物品。如果一个元素符合触发 BFC 的条件，则 BFC 中的元素布局不受外部影响。

创建 BFC 的条件：

根元素：body；

元素设置浮动：float 除 none 以外的值；

元素设置绝对定位：position (absolute、fixed)；

display 值为：inline-block、table-cell、table-caption、flex 等；

overflow 值为：hidden、auto、scroll；

BFC 的特点：

垂直方向上，自上而下排列，和文档流的排列方式一致。

在 BFC 中上下相邻的两个容器的 margin 会重叠

计算 BFC 的高度时，需要计算浮动元素的高度

BFC 区域不会与浮动的容器发生重叠

BFC 是独立的容器，容器内部元素不会影响外部元素

每个元素的左 margin 值和容器的左 border 相接触

BFC 的作用：

解决 margin 的重叠问题：由于 BFC 是一个独立的区域，内部的元素和外部的元素互不影响，将两个元素变为两个 BFC，就解决了 margin 重叠的问题。

解决高度塌陷的问题：在对子元素设置浮动后，父元素会发生高度塌陷，也就是父元素的高度变为 0。解决这个问题，只需要把父元素变成一个 BFC。常用的办法是给父元素设置 overflow:hidden。

创建自适应两栏布局：可以用来创建自适应两栏布局：左边的宽度固定，右边的宽度自适应。

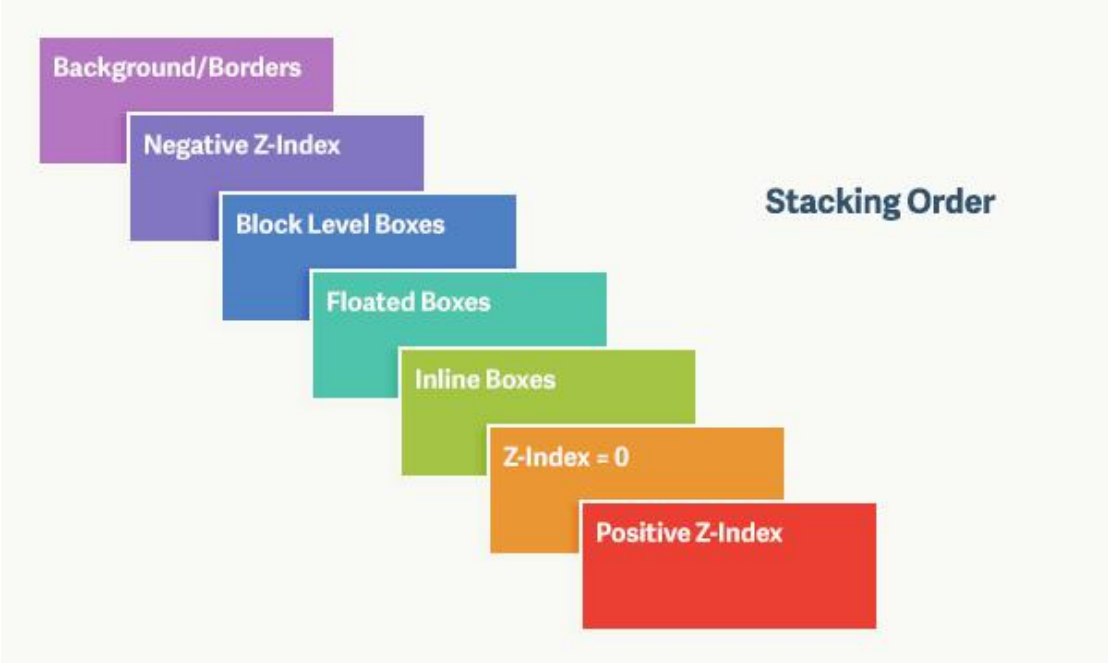
```
.left{
    width: 100px;
    height: 200px;
    background: red;
    float: left;
}
.right{
    height: 300px;
    background: blue;
    overflow: hidden;
}

<div class="left"></div>
<div class="right"></div>
```

左侧设置 float:left，右侧设置 overflow: hidden。这样右边就触发了 BFC，BFC 的区域不会与浮动元素发生重叠，所以两侧就不会发生重叠，实现了自适应两栏布局。

10. 元素的层叠顺序

层叠顺序，英文称作 stacking order，表示元素发生层叠时有着特定的垂直显示顺序。下面是盒模型的层叠规则：



对于上图，由上到下分别是：

（1）背景和边框：建立当前层叠上下文元素的背景和边框。

（2）负的 z-index：当前层叠上下文中，z-index 属性值为负的元素。

（3）块级盒：文档流内非行内级非定位后代元素。

（4）浮动盒：非定位浮动元素。

（5）行内盒：文档流内行内级非定位后代元素。

（6）z-index:0：层叠级数为 0 的定位元素。

（7）正 z-index：z-index 属性值为正的定位元素。

注意：当定位元素 z-index:auto，生成盒在当前层叠上下文中的层级为 0，不会建立新的层叠上下文，除非是根元素。

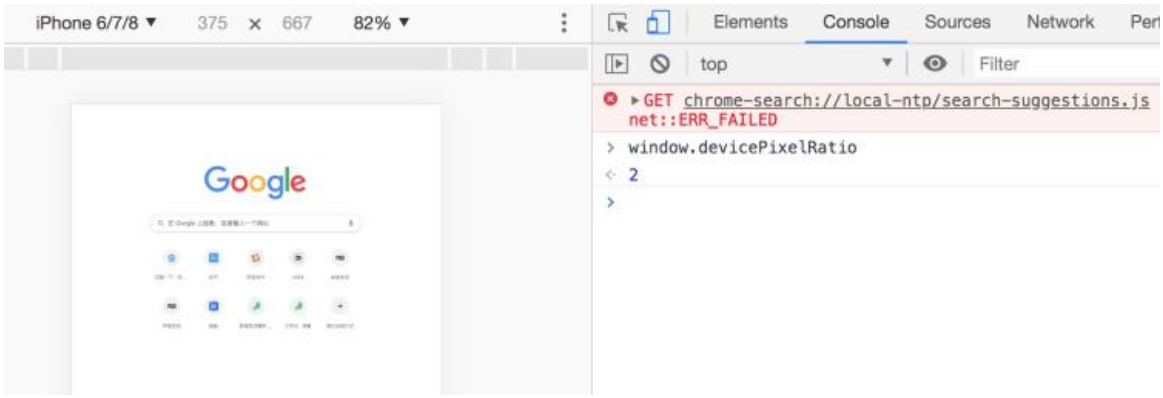
11. 如何解决 1px 问题？

1px 问题指的是：在一些 Retina 屏幕 的机型上，移动端页面的 1px 会变得很粗，呈现出不止 1px 的效果。原因很简单——CSS 中的 1px

并不能和移动设备上的 1px 划等号。它们之间的比例关系有一个专门的属性来描述：

```
window.devicePixelRatio = 设备的物理像素 / CSS像素。
```

打开 Chrome 浏览器，启动移动端调试模式，在控制台去输出这个 devicePixelRatio 的值。这里选中 iPhone6/7/8 这系列的机型，输出的结果就是 2：



这就意味着设置的 1px CSS 像素，在这个设备上实际会用 2 个物理像素单元来进行渲染，所以实际看到的一定会比 1px 粗一些。

解决 1px 问题的三种思路：

思路一：直接写 0.5px

如果之前 1px 的样式这样写：

```
border:1px solid #333
```

可以先在 JS 中拿到 window.devicePixelRatio 的值，然后把这个值通过 JSX 或者模板语法给到 CSS 的 data 里，达到这样的效果（这里用 JSX 语法做示范）：

```
<div id="container" data-device={{window.devicePixelRatio}}></div>
```

然后就可以在 CSS 中用属性选择器来命中 devicePixelRatio 为某一值的情况，比如说这里尝试命中 devicePixelRatio 为 2 的情况：

```
#container[data-device="2"] {
  border:0.5px solid #333
}
```

直接把 1px 改成 1/devicePixelRatio 后的值，这是目前为止最简单的一种方法。这种方法的缺陷在于兼容性不行，IOS 系统需要 8 及以上的版本，安卓系统则直接不兼容。

思路二：伪元素先放大后缩小

这个方法的可行性会更高，兼容性也更好。唯一的缺点是代码会变多。

思路是先放大、后缩小：在目标元素的后面追加一个 ::after 伪元素，让这个元素布局为 absolute 之后、整个伸展开铺在目标元素上，然后把它的宽和高都设置为目标元素的两倍，border 值设为 1px。接着借助 CSS 动画特效中的放缩能力，把整个伪元素缩小为原来的 50%。此时，伪元素的宽高刚好可以和原有的目标元素对齐，而 border 也缩小为了 1px 的二分之一，间接地实现了 0.5px 的效果。

代码如下：

```
#container[data-device="2"] {
    position: relative;
}
#container[data-device="2"]::after{
      position:absolute;
      top: 0;
      left: 0;
      width: 200%;
      height: 200%;
      content:"";
      transform: scale(0.5);
      transform-origin: left top;
      box-sizing: border-box;
      border: 1px solid #333;
    }
}
```
