请简述JavaScript中的this

JS 中的 `this` 是一个相对复杂的概念，不是简单几句能解释清楚的。粗略地讲，函数的调用方式决定了 `this` 的值。我阅读了网上很多关于 `this` 的文章，**Arnav Aggrawal** 写的比较清楚。`this` 取值符合以下规则：

在调用函数时使用 `new` 关键字，函数内的 `this` 是一个全新的对象。 如果 `apply、call` 或 `bind` 方法用于调用、创建一个函数，函数内的 `this` 就是作为参数传入这些方法的对象。 当函数作为对象里的方法被调用时，函数内的 `this` 是调用该函数的对象。比如当 `obj.method()` 被调用时，函数内的 `this` 将绑定到 `obj` 对象。 如果调用函数不符合上述规则，那么 `this` 的值指向全局对象（`global object`）。浏览器环境下 `this` 的值指向 `window` 对象，但是在严格模式下(`'use strict'`)，`this` 的值为 `undefined`。 如果符合上述多个规则，则较高的规则（1 号最高，4 号最低）将决定 `this` 的值。 如果该函数是 `ES2015` 中的箭头函数，将忽略上面的所有规则，`this` 被设置为它被创建时的上下文。

说说你对 AMD 和 CommonJS 的了解。

它们都是实现模块体系的方式，直到 `ES2015` 出现之前，`JavaScript` 一直没有模块体系。`CommonJS` 是同步的，而 `AMD（Asynchronous Module Definition）` 从全称中可以明显看出是异步的。`CommonJS` 的设计是为服务器端开发考虑的，而 `AMD` 支持异步加载模块，更适合浏览器。

我发现 `AMD` 的语法非常冗长，`CommonJS` 更接近其他语言 `import` 声明语句的用法习惯。大多数情况下，我认为 `AMD` 没有使用的必要，因为如果把所有 `JavaScript` 都捆绑进一个文件中，将无法得到异步加载的好处。此外，`CommonJS` 语法上更接近 `Node` 编写模块的风格，在前后端都使用 `JavaScript` 开发之间进行切换时，语境的切换开销较小。

我很高兴看到 `ES2015` 的模块加载方案同时支持同步和异步，我们终于可以只使用一种方案了。虽然它尚未在浏览器和 `Node` 中完全推出，但是我们可以使用代码转换工具进行转换。

请解释下面代码为什么不能用作 IIFE：function foo(){ }();，需要作出哪些修改才能使其成为 IIFE?

IIFE（Immediately Invoked Function Expressions）代表立即执行函数。`JavaScript` 解析器将 `function foo(){ }();` 解析成 `function foo(){ }和();`。其中，前者是函数声明；后者（一对括号）是试图调用一个函数，却没有指定名称，因此它会抛出 `Uncaught SyntaxError: Unexpected token )` 的错误。

修改方法是：再添加一对括号，形式上有两种：`(function foo(){ })()` 和 `(function foo(){ }())`。以上函数不会暴露到全局作用域，如果不需要在函数内部引用自身，可以省略函数的名称。

你可能会用到 `void` 操作符：`void function foo(){ }();`。但是，这种做法是有问题的。表达式的值是 `undefined`，所以如果你的 `IIFE` 有返回值，不要用这种做法。例如

```
javascript复制代码const foo = void (function bar() {
  return 'foo';
})();

console.log(foo); // undefined
```

null、undefined和未声明变量之间有什么区别？如何检查判断这些状态值？

当你没有提前使用 `var、let` 或 `const` 声明变量，就为一个变量赋值时，该变量是未声明变量（`undeclared variables`）。未声明变量会脱离当前作用域，成为全局作用域下定义的变量。在严格模式下，给未声明的变量赋值，会抛出 `ReferenceError` 错误。和使用全局变量一样，使用未声明变量也是非常不好的做法，应当尽可能避免。要检查判断它们，需要将用到它们的代码放在 `try/catch` 语句中。

```
scss复制代码function foo() {
  x = 1; // 在严格模式下，抛出 ReferenceError 错误
}

foo();
console.log(x); // 1
```

当一个变量已经声明，但没有赋值时，该变量的值是 `undefined`。如果一个函数的执行结果被赋值给一个变量，但是这个函数却没有返回任何值，那么该变量的值是 `undefined`。要检查它，需要使用严格相等（`===`）；或者使用 `typeof`，它会返回 `'undefined'` 字符串。请注意，不能使用非严格相等（`==`）来检查，因为如果变量值为 `null`，使用非严格相等也会返回 `true`。

```
javascript复制代码var foo;
console.log(foo); // undefined
console.log(foo === undefined); // true
console.log(typeof foo === 'undefined'); // true

console.log(foo == null); // true. 错误，不要使用非严格相等!

function bar() {}
var baz = bar();
console.log(baz); // undefined
```

`null` 只能被显式赋值给变量。它表示空值，与被显式赋值 `undefined` 的意义不同。要检查判断 `null` 值，需要使用严格相等运算符。请注意，和前面一样，不能使用非严格相等（`==`）来检查，因为如果变量值为 `undefined`，使用非严格相等也会返回 `true`。

```
ini复制代码var foo = null;
console.log(foo === null); // true

console.log(foo == undefined); // true. 错误，不要使用非严格相等!
```

作为一种个人习惯，我从不使用未声明变量。如果定义了暂时没有用到的变量，我会在声明后明确地给它们赋值为 `null`

什么是闭包（closure），为什么使用闭包？

闭包是函数和声明该函数的词法环境的组合。词法作用域中使用的域，是变量在代码中声明的位置所决定的。闭包是即使被外部函数返回，依然可以访问到外部（封闭）函数作用域的函数。

为什么使用闭包？

- 利用闭包实现数据私有化或模拟私有方法。这个方式也称为模块模式（`module pattern`）。

- 部分参数函数（`partial applications`）柯里化（`currying`）.

请说明.forEach循环和.map()循环的主要区别，它们分别在什么情况下使用?

为了理解两者的区别，我们看看它们分别是做什么的。

forEach

- 遍历数组中的元素。
- 为每个元素执行回调。
- 无返回值。

```
ini复制代码const a = [1, 2, 3];
const doubled = a.forEach((num, index) => {
  // 执行与 num、index 相关的代码
});

// doubled = undefined
```

map

- 遍历数组中的元素
- 通过对每个元素调用函数，将每个元素"映射（map）"到一个新元素，从而创建一个新数组。

```
ini复制代码const a = [1, 2, 3];
const doubled = a.map((num) => {
  return num * 2;
});

// doubled = [2, 4, 6]
```

`.forEach` 和 `.map()` 的主要区别在于 `.map()` 返回一个新的数组。如果你想得到一个结果，但不想改变原始数组，用 `.map()`。如果你只需要在数组上做迭代修改，用 `forEach`。

匿名函数的典型应用场景是什么?

匿名函数可以在 IIFE 中使用，来封装局部作用域内的代码，以便其声明的变量不会暴露到全局作用域。

```
javascript复制代码(function () {
  // 一些代码。
})();
```

匿名函数可以作为只用一次，不需要在其他地方使用的回调函数。当处理函数在调用它们的程序内部被定义时，代码具有更好地自闭性和可读性，可以省去寻找该处理函数的函数体位置的麻烦。

```
javascript复制代码setTimeout(function () {
  console.log('Hello world!');
}, 1000);
```

匿名函数可以用于函数式编程或 Lodash（类似于回调函数）。

```
c复制代码const arr = [1, 2, 3];
const double = arr.map(function (el) {
  return el * 2;
});
console.log(double); // [2, 4, 6]
```

.call和.apply有什么区别?

`.call` 和 `.apply` 都用于调用函数，第一个参数将用作函数内 `this` 的值。然而，`.call` 接受逗号分隔的参数作为后面的参数，而 `.apply` 接受一个参数数组作为后面的参数。一个简单的记忆方法是，从 `call` 中的 C 联想到逗号分隔（`comma-separated`），从 `apply` 中的 A 联想到数组（`array`）。

```
javascript复制代码function add(a, b) {
  return a + b;
}

console.log(add.call(null, 1, 2)); // 3
console.log(add.apply(null, [1, 2])); // 3
```

请说明Function.prototype.bind的用法。

摘自MDN：

> bind()方法创建一个新的函数, 当被调用时，将其 this 关键字设置为提供的值，在调用新函数时，在任何提供之前提供一个给定的参数序列。

根据我的经验，将 `this` 的值绑定到想要传递给其他函数的类的方法中是非常有用的。在 `React` 组件中经常这样做。

请尽可能详细地解释 Ajax。

Ajax（asynchronous JavaScript and XML）是使用客户端上的许多 Web 技术，创建异步 Web 应用的一种 Web 开发技术。借助 Ajax，Web 应用可以异步（在后台）向服务器发送数据和从服务器检索数据，而不会干扰现有页面的显示和行为。通过将数据交换层与表示层分离，Ajax 允许网页和扩展 Web 应用程序动态更改内容，而无需重新加载整个页面。实际上，现在通常将 XML 替换为 JSON，因为 JavaScript 对 JSON 有原生支持优势。

`XMLHttpRequest API` 经常用于异步通信。此外还有最近流行的 `fetch API`。

使用 Ajax 的优缺点分别是什么?

优点

- 交互性更好。来自服务器的新内容可以动态更改，无需重新加载整个页面。
- 减少与服务器的连接，因为脚本和样式只需要被请求一次。
- 状态可以维护在一个页面上。JavaScript 变量和 DOM 状态将得到保持，因为主容器页面未被重新加载。
- 基本上包括大部分 SPA 的优点。

缺点

- 动态网页很难收藏。
- 如果 JavaScript 已在浏览器中被禁用，则不起作用。
- 有些网络爬虫不执行 JavaScript，也不会看到 JavaScript 加载的内容。

- 基本上包括大部分 SPA 的缺点

请说明 JSONP 的工作原理，它为什么不是真正的 Ajax?

JSONP（带填充的 JSON）是一种通常用于绕过 Web 浏览器中的跨域限制的方法，因为 Ajax 不允许跨域请求。

JSONP 通过 ` 标签发送跨域请求，通常使用 callback 查询参数，例如：https://example.com?callback=printData。 然后服务器将数据包装在一个名为 printData ` 的函数中并将其返回给客户端。

```
xml复制代码<!-- https://mydomain.com -->
<script>
  function printData(data) {
    console.log(`My name is ${data.name}!`);
  }
</script>

<script src="https://example.com?callback=printData"></script>
css复制代码// 文件加载自 https://example.com?callback=printData
printData({name: 'Yang Shun'});
```

客户端必须在其全局范围内具有 `printData` 函数，并且在收到来自跨域的响应时，该函数将由客户端执行。

JSONP 可能具有一些安全隐患。由于 JSONP 是纯 JavaScript 实现，它可以完成 JavaScript 所能做的一切，因此需要信任 JSONP 数据的提供者。

现如今，跨来源资源共享（CORS） 是推荐的主流方式，JSONP 已被视为一种比较 hack 的方式。

请解释变量提升（hoisting）。

变量提升（hoisting）是用于解释代码中变量声明行为的术语。使用 `var` 关键字声明或初始化的变量，会将声明语句“提升”到当前作用域的顶部。 但是，只有声明才会触发提升，赋值语句（如果有的话）将保持原样。我们用几个例子来解释一下。

```
javascript复制代码// 用 var 声明得到提升
console.log(foo); // undefined
var foo = 1;
console.log(foo); // 1

// 用 let/const 声明不会提升
console.log(bar); // ReferenceError: bar is not defined
let bar = 2;
console.log(bar); // 2
```

函数声明会使函数体提升，但函数表达式（以声明变量的形式书写）只有变量声明会被提升。

```
javascript复制代码// 函数声明
console.log(foo); // [Function: foo]
foo(); // 'FOOOOO'
function foo() {
  console.log('FOOOOO');
}
console.log(foo); // [Function: foo]
```

```
// 函数表达式
console.log(bar); // undefined
bar(); // Uncaught TypeError: bar is not a function
var bar = function () {
  console.log('BARRRR');
};
console.log(bar); // [Function: bar]
```

请描述事件冒泡。

当一个事件在 DOM 元素上触发时，如果有事件监听器，它将尝试处理该事件，然后事件冒泡到其父级元素，并发生同样的事情。最后直到事件到达祖先元素。事件冒泡是实现事件委托的原理（event delegation）。

==和===的区别是什么

`==` 是抽象相等运算符，而 `===` 是严格相等运算符。`==` 运算符是在进行必要的类型转换后，再比较。`===` 运算符不会进行类型转换，所以如果两个值不是相同的类型，会直接返回 `false` 。使用 `==` 时，可能发生一些特别的事情，例如：

```
ini复制代码1 == '1'; // true
1 == [1]; // true
1 == true; // true
0 == ''; // true
0 == '0'; // true
0 == false; // true
```

我的建议是从不使用 `==` 运算符，除了方便与 `null` 或 `undefined` 比较时，`a == null` 如果 `a` 为 `null` 或 `undefined` 将返回 `true`

```
ini复制代码var a = null;
console.log(a == null); // true
console.log(a == undefined); // true
```

请解释关于 JavaScript 的同源策略。

同源策略可防止 JavaScript 发起跨域请求。源被定义为 URI、主机名和端口号的组合。此策略可防止页面上的恶意脚本通过该页面的文档对象模型，访问另一个网页上的敏感数据。

你对 Promises 及其 polyfill 的掌握程度如何?

掌握它的工作原理。`Promise` 是一个可能在未来某个时间产生结果的对象：操作成功的结果或失败的原因（例如发生网络错误）。 `Promise` 可能处于以下三种状态之一：`fulfilled`、`rejected` 或 `pending`。 用户可以对 `Promise` 添加回调函数来处理操作成功的结果或失败的原因。

一些常见的 `polyfill` 是 `$.deferred`、Q 和 Bluebird，但不是所有的 polyfill 都符合规范。ES2015 支持 Promises，现在通常不需要使用 polyfills。

Promise代替回调函数有什么优缺点?

优点

- 避免可读性极差的回调地狱。
- 使用.then()编写的顺序异步代码，既简单又易读。
- 使用Promise.all()编写并行异步代码变得很容易。

缺点

- 轻微地增加了代码的复杂度（这点存在争议）。
- 在不支持 ES2015 的旧版浏览器中，需要引入 polyfill 才能使用。

请解释同步和异步函数之间的区别。

同步函数阻塞，而异步函数不阻塞。在同步函数中，语句完成后，下一句才执行。在这种情况下，程序可以按照语句的顺序进行精确评估，如果其中一个语句需要很长时间，程序的执行会停滞很长时间。

异步函数通常接受回调作为参数，在调用异步函数后立即继续执行下一行。回调函数仅在异步操作完成且调用堆栈为空时调用。诸如从 Web 服务器加载数据或查询数据库等重负载操作应该异步完成，以便主线程可以继续执行其他操作，而不会出现一直阻塞，直到费时操作完成的情况（在浏览器中，界面会卡住）。

什么是事件循环？调用堆栈和任务队列之间有什么区别？

事件循环是一个单线程循环，用于监视调用堆栈并检查是否有工作即将在任务队列中完成。如果调用堆栈为空并且任务队列中有回调函数，则将回调函数出队并推送到调用堆栈中执行。

使用let、var和const创建变量有什么区别？

用 `var` 声明的变量的作用域是它当前的执行上下文，它可以是嵌套的函数，也可以是声明在任何函数外的变量。`let` 和 `const` 是块级作用域，意味着它们只能在最近的一组花括号（function、if-else 代码块或 for 循环中）中访问。

```
javascript复制代码function foo() {
  // 所有变量在函数中都可访问
  var bar = 'bar';
  let baz = 'baz';
  const qux = 'qux';

  console.log(bar); // bar
  console.log(baz); // baz
  console.log(qux); // qux
}

console.log(bar); // ReferenceError: bar is not defined
console.log(baz); // ReferenceError: baz is not defined
console.log(qux); // ReferenceError: qux is not defined
ini复制代码if (true) {
  var bar = 'bar';
  let baz = 'baz';
  const qux = 'qux';
}

// 用 var 声明的变量在函数作用域上都可访问
console.log(bar); // bar
// let 和 const 定义的变量在它们被定义的语句块之外不可访问
console.log(baz); // ReferenceError: baz is not defined
console.log(qux); // ReferenceError: qux is not defined
```

`var` 会使变量提升，这意味着变量可以在声明之前使用。`let`和 `const` 不会使变量提升，提前使用会报错。

```ini复制代码
console.log(foo); // undefined

var foo = 'foo';

console.log(baz); // ReferenceError: can't access lexical declaration 'baz'
before initialization

let baz = 'baz';

console.log(bar); // ReferenceError: can't access lexical declaration 'bar'
before initialization

const bar = 'bar';
```

用 `var` 重复声明不会报错，但 `let` 和 `const` 会。

```ini复制代码
var foo = 'foo';
var foo = 'bar';
console.log(foo); // "bar"

let baz = 'baz';
let baz = 'qux'; // Uncaught SyntaxError: Identifier 'baz' has already been
declared
```

`let` 和 `const` 的区别在于：`let` 允许多次赋值，而 `const` 只允许一次。

```ini复制代码
// 这样不会报错。
let foo = 'foo';
foo = 'bar';

// 这样会报错。
const baz = 'baz';
baz = 'qux';
```

你能给出一个使用箭头函数的例子吗，箭头函数与其他函数有什么不同

一个很明显的优点就是箭头函数可以简化创建函数的语法，我们不需要在箭头函数前面加上 `function` 关键词。并且箭头函数的 `this` 会自动绑定到当前作用域的上下文中，这和普通的函数不一样。普通函数的 `this` 是在执行的时候才能确定的。箭头函数的这个特点对于回调函数来说特别有用，特别对于 `React` 组件而言。

高阶函数（higher-order）的定义是什么?

高阶函数是将一个或多个函数作为参数的函数，它用于数据处理，也可能将函数作为返回结果。高阶函数是为了抽象一些重复执行的操作。一个典型的例子是 `map`，它将一个数组和一个函数作为参数。`map` 使用这个函数来转换数组中的每个元素，并返回一个包含转换后元素的新数组。JavaScript 中的其他常见示例是 `forEach`、`filter` 和 `reduce`。高阶函数不仅需要操作数组的时候会用到，还有许多

函数返回新函数的用例。`Function.prototype.bind` 就是一个例子。

Map 示例

假设我们有一个由名字组成的数组，我们需要将每个字符转换为大写字母。

```ini复制代码const names = ['irish', 'daisy', 'anna'];
```

不使用高阶函数的方法是这样：

```ini复制代码const transformNamesToUppercase = function (names) {
  const results = [];
  for (let i = 0; i < names.length; i++) {
    results.push(names[i].toUpperCase());
  }
  return results;
};
transformNamesToUppercase(names); // ['IRISH', 'DAISY', 'ANNA']
```

使用 `.map(transformerFn)` 使代码更简明

```javascript复制代码const transformNamesToUppercase = function (names) {
  return names.map((name) => name.toUpperCase());
};
transformNamesToUppercase(names); // ['IRISH', 'DAISY', 'ANNA']
```

请给出一个解构（destructuring）对象或数组的例子。

解构是 ES6 中新功能，它提供了一种简洁方便的方法来提取对象或数组的值，并将它们放入不同的变量中。

数组解构

```arduino复制代码// 变量赋值
const foo = ['one', 'two', 'three'];

const [one, two, three] = foo;
console.log(one); // "one"
console.log(two); // "two"
console.log(three); // "three"
```

```ini复制代码// 变量交换
let a = 1;
let b = 3;

[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1
```

对象解构

```
arduino复制代码// 变量赋值
const o = {p: 42, q: true};
const {p, q} = o;

console.log(p); // 42
console.log(q); // true
```

你能举出一个柯里化函数（curry function）的例子吗？它有哪些好处？

柯里化（currying）是一种模式，其中具有多个参数的函数被分解为多个函数，当被串联调用时，将一次一个地累积所有需要的参数。这种技术帮助编写函数式风格的代码，使代码更易读、紧凑。值得注意的是，对于需要被 curry 的函数，它需要从一个函数开始，然后分解成一系列函数，每个函数都需要一个参数。

```
scss复制代码function curry(fn) {
  if (fn.length === 0) {
    return fn;
  }

  function _curried(depth, args) {
    return function (newArgument) {
      if (depth - 1 === 0) {
        return fn(...args, newArgument);
      }
      return _curried(depth - 1, [...args, newArgument]);
    };
  }

  return _curried(fn.length, []);
}

function add(a, b) {
  return a + b;
}

var curriedAdd = curry(add);
var addFive = curriedAdd(5);

var result = [0, 1, 2, 3, 4, 5].map(addFive); // [5, 6, 7, 8, 9, 10]
```

meta viewport 是做什么用的，怎么写？

```
ini复制代码<meta name="viewport" content="width=device-width, initial-scale=1,
maximum-scale=1, minimum-scale=1" />
```

目的 是为了在移动端不让用户缩放页面使用的

解释每个单词的含义

- with=device-width 将布局视窗（layout viewport）的宽度设置为设备屏幕分辨率的宽度
- initial-scale=1 页面初始缩放比例为屏幕分辨率的宽度
- maximum-scale=1 指定用户能够放大的最大比例

- minimum-scale=1 指定用户能够缩小的最大比例

浏览器乱码的原因是什么?如何解决?

编码格式不匹配

浏览器打开网页时,需要根据网页源代码的编码格式来解码。如果网页的编码格式与浏览器的编码格式不匹配,就会出现乱码。比如,网页的编码格式为 UTF-8,而浏览器的编码格式是 GB2312,那么就会出现乱码。

网页编码错误

在编写网页的时候,如果编码出现错误,也会导致浏览器打开网页时出现乱码。比如,在写 HTML 代码时,如果忘记给中文字符指定编码格式,就会出现乱码。

字体缺失

有些网页会使用比较特殊的字体,如果浏览器中没有这个字体,就会找不到对应的字符,从而出现乱码。

iframe 有那些优点和缺点?

优点

- 可以在页面上独立显示一个页面或者内容,不会与页面其他元素产生冲突。
- 可以在多个页面中重用同一个页面或者内容,可以减少代码的冗余。
- 加载是异步的,页面可以在不等待 iframe 加载完成的情况下进行展示。
- 方便地实现跨域访问

缺点

- 搜索引擎可能无法正确解析 iframe 中的内容
- 会阻塞主页面的 onload 事件
- 和主页面共享连接池,影响页面并行加载

HTML5 新特性

- 语义化标签
- 增强型表单(如可以通过 input 的 type 属性指定类型是 color 还是 date 或者 url 等)
- 媒体元素标签(video,audio)
- canvas,svg
- svg 绘图
- 地理等位(navigator.geolocation.getCurrentPosition(callback))
- 拖放 API(给标签元素设置属性 draggable 值为 true,能够实现对目标元素的拖动)
- Web Worker(可以开启一个子线程运行脚本)
- Web Storage(即 sessionStorage 与 localStorage)
- Websocket(双向通信协议,可以让浏览器接收服务端的请求)
- dom 查询(document.querySelector()和 document.querySelectorAll())

如何使用HTML5中的Canvas元素绘制图形?

Canvas元素允许在网页上使用JavaScript绘制图形和动画。以下是一个简单的绘制矩形的示例:

```ini复制代码<canvas id="myCanvas" width="200" height="200"></canvas>
<script>
  var canvas = document.getElementById("myCanvas");
  var ctx = canvas.getContext("2d");
  ctx.fillStyle = "red";
  ctx.fillRect(50, 50, 100, 100);
</script>
```

在这个示例中，使用 `document.getElementById()` 方法获取 `Canvas` 元素，并通过 `getContext("2d")` 获取2D绘图上下文。 然后，使用 `fillStyle` 属性设置填充颜色，`fillRect()` 方法绘制一个矩形。

什么是data-属性?

在 JavaScript 框架变得流行之前，前端开发者经常使用 `data-` 属性，把额外数据存储在 DOM 自身中。当时没有其他 Hack 手段（比如使用非标准属性或 DOM 上额外属性）。这样做是为了将自定义数据存储到页面或应用中，对此没有其他更适当的属性或元素。

而现在，不鼓励使用 `data-` 属性。原因之一是，用户可以通过在浏览器中利用检查元素，轻松地修改属性值，借此修改数据。数据模型最好存储在 JavaScript 本身中，并利用框架提供的数据绑定，使之与 DOM 保持更新。

请描述 cookie、sessionStorage 和 localStorage 的区别。

	cookie	localStorage	sessionStorage
由谁初始化	客户端或服务器，服务器可以使用 `Set-Cookie` 请求头。	客户端	客户端
过期时间	手动设置	永不过期	当前页面关闭时
在当前浏览器会话（browser sessions）中是否保持不变	取决于是否设置了过期时间	是	否
是否随着每个 HTTP 请求发送给服务器	是，Cookies 会通过 `Cookie` 请求头，自动发送给服务器	否	否
容量（每个域名）	4kb	5MB	5MB
访问权限	任意窗口	任意窗口	当前页面窗口

请描述 script、script async 和 script defer 的区别。

- `` - HTML 解析中断，脚本被提取并立即执行。执行结束后，HTML 解析继续。
- `` - 脚本的提取、执行的过程与 HTML 解析过程并行，脚本执行完毕可能在 HTML 解析完毕之前。当脚本与页面上其他脚本独立时，可以使用async`，比如用作页面统计分析。

- `` - 脚本仅提取过程与 HTML 解析过程并行，脚本的执行将在 HTML 解析完毕后进行。如果有多个含 defer` 的脚本，脚本的执行顺序将按照在 document 中出现的位置，从上到下顺序执行。

注意：没有 `src` 属性的脚本，`async` 和 `defer` 属性会被忽略。

为什么最好把 CSS 的 link 标签放在 head 之间？为什么最好把 JS 的 script 标签恰好放在 body 之前，有例外情况吗？

把 `标签放在` 之间是规范要求的内容。此外，这种做法可以让页面逐步呈现，提高了用户体验。将样式表放在文档底部附近，会使许多浏览器（包括 Internet Explorer）不能逐步呈现页面。一些浏览器会阻止渲染，以避免在页面样式发生变化时，重新绘制页面中的元素。这种做法可以防止呈现给用户空白的页面或没有样式的内容。

把 `标签恰好放在` 之前

脚本在下载和执行期间会阻止 HTML 解析。把 `` 标签放在底部，保证 HTML 首先完成解析，将页面尽早呈现给用户。

例外情况是当你的脚本里包含 `document.write()` 时。但是现在，`document.write()` 不推荐使用。同时，将 `标签放在底部，意味着浏览器不能开始下载脚本，直到整个文档 `（document）` 被解析。也许，对此比较好的做法是，` 使用 `defer` 属性，放在 `` 中。

什么是渐进式渲染（progressive rendering）？

渐进式渲染是用于提高网页性能（尤其是提高用户感知的加载速度），以尽快呈现页面的技术。

在以前互联网带宽较小的时期，这种技术更为普遍。如今，移动终端的盛行，而移动网络往往不稳定，渐进式渲染在现代前端开发中仍然有用武之地。

一些举例：

- 图片懒加载——页面上的图片不会一次性全部加载。当用户滚动页面到图片部分时，JavaScript 将加载并显示图像。
- 确定显示内容的优先级（分层次渲染）——为了尽快将页面呈现给用户，页面只包含基本的最少量的 CSS、脚本和内容，然后可以使用延迟加载脚本或监听 `DOMContentLoaded` / `load` 事件加载其他资源和内容。
- 异步加载 HTML 片段——当页面通过后台渲染时，把 HTML 拆分，通过异步请求，分块发送给浏览器。

CSS 选择器的优先级是如何计算的？

浏览器通过优先级规则，判断元素展示哪些样式。优先级通过 4 个维度指标确定，我们假定以 `a、b、c、d` 命名，分别代表以下含义：

- `a` 表示是否使用内联样式（inline style）。如果使用，`a` 为 1，否则为 0。
- `b` 表示 ID 选择器的数量。
- `c` 表示类选择器、属性选择器和伪类选择器数量之和。
- `d` 表示标签（类型）选择器和伪元素选择器之和。

优先级的结果并非通过以上四个值生成一个得分，而是每个值分开比较。`a、b、c、d` 权重从左到右，依次减小。判断优先级时，从左到右，一一比较，直到比较出最大值，即可停止。所以，如果 `b` 的值不同，那么 `c` 和 `d` 不管多大，都不会对结果产生影响。比如 `0，1，0，0` 的优先级高于 `0，0，10，10`。

当出现优先级相等的情况时，最晚出现的样式规则会被采纳。如果你在样式表里写了相同的规则（无论是在该文件内部还是其它样式文件中），那么最后出现的（在文件底部的）样式优先级更高，因此会被采纳。

在写样式时，我会使用较低的优先级，这样这些样式可以轻易地覆盖掉。尤其对写 UI 组件的时候更为重要，这样使用者就不需要通过非常复杂的优先级规则或使用 `!important` 的方式，去覆盖组件的样式了。

重置（resetting）CSS 和 标准化（normalizing）CSS 的区别是什么？你会选择哪种方式，为什么？

- **重置（Resetting）**： 重置意味着除去所有的浏览器默认样式。对于页面所有的元素，像 margin、padding、font-size这些样式全部置成一样。你将必须重新定义各种元素的样式。
- **标准化（Normalizing）**： 标准化没有去掉所有的默认样式，而是保留了有用的一部分，同时还纠正了一些常见错误。

当需要实现非常个性化的网页设计时，我会选择重置的方式，因为我要写很多自定义的样式以满足设计需求，这时候就不再需要标准化的默认样式了。

请阐述Float定位的工作原理。

浮动（float）是 CSS 定位属性。浮动元素从网页的正常流动中移出，但是保持了部分的流动性，会影响其他元素的定位（比如文字会围绕着浮动元素）。这一点与绝对定位不同，绝对定位的元素完全从文档流中脱离。

CSS 的 `clear` 属性通过使用 `left、right、both`，让该元素向下移动（清除浮动）到浮动元素下面。

如果父元素只包含浮动元素，那么该父元素的高度将塌缩为 0。我们可以通过清除（clear）从浮动元素后到父元素关闭前之间的浮动来修复这个问题。

有一种 hack 的方法，是自定义一个 `.clearfix` 类，利用伪元素选择器 `::after` 清除浮动。另外还有一些方法，比如添加空的 `` ` `` 和设置浮动元素父元素的overflow属性。与这些方法不同的是，clearfix` 方法，只需要给父元素添加一个类，定义如下：

```
css复制代码.clearfix::after {
  content: '';
  display: block;
  clear: both;
}
```

值得一提的是，把父元素属性设置为 `overflow: auto` 或 `overflow: hidden`，会使其内部的子元素形成块格式化上下文（Block Formatting Context），并且父元素会扩张自己，使其能够包围它的子元素。

请阐述z-index属性，并说明如何形成层叠上下文（stacking context）。

CSS 中的 `z-index` 属性控制重叠元素的垂直叠加顺序。`z-index` 只能影响 `position` 值不是 `static` 的元素。

没有定义 `z-index` 的值时，元素按照它们出现在 DOM 中的顺序堆叠（层级越低，出现位置越靠上）。非静态定位的元素（及其子元素）将始终覆盖静态定位（static）的元素，而不管 HTML 层次结构如何。

层叠上下文是包含一组图层的元素。 在一组层叠上下文中，其子元素的 `z-index` 值是相对于该父元素而不是 document root 设置的。每个层叠上下文完全独立于它的兄弟元素。如果元素 B 位于元素 A 之上，则即使元素 A 的子元素 C 具有比元素 B 更高的z-index值，元素 C 也永远不会在元素 B 之上.

每个层叠上下文是自包含的：当元素的内容发生层叠后，整个该元素将会在父层叠上下文中按顺序进行层叠。少数 CSS 属性会触发一个新的层叠上下文，例如opacity小于 1，filter不是none，transform不是none。

每个层叠上下文是自包含的：当元素的内容发生层叠后，整个该元素将会在父层叠上下文中按顺序进行层叠。少数 CSS 属性会触发一个新的层叠上下文，例如 `opacity` 小于 1，`filter` 不是 `none`，`transform` 不是 `none`。

请阐述块格式化上下文（Block Formatting Context）及其工作原理。

块格式上下文（BFC）是 Web 页面的可视化 CSS 渲染的部分，是块级盒布局发生的区域，也是浮动元素与其他元素交互的区域。

一个 HTML 盒（Box）满足以下任意一条，会创建块格式化上下文：

- float的值不是none.
- position的值不是static或relative.
- display的值是table-cell、table-caption、inline-block、flex、或inline-flex。
- overflow的值不是visible。

在 BFC 中，每个盒的左外边缘都与其包含的块的左边缘相接。

两个相邻的块级盒在垂直方向上的边距会发生合并（collapse）。

有哪些清除浮动的技术，都适用哪些情况?

- 空div方法：

  。
- Clearfix 方法：上文使用.clearfix类已经提到。
- overflow: auto或overflow: hidden方法：上文已经提到。

在大型项目中，我会使用 Clearfix 方法，在需要的地方使用 `.clearfix`。设置 `overflow: hidden` 的方法可能使其子元素显示不完整，当子元素的高度大于父元素时。

请解释什么是精灵图（css sprites），以及如何实现?

精灵图，也称雪碧图。因常见碳酸饮料雪碧的英文名也是 Sprite，因此也有人会使用雪碧图的非正式译名。

精灵图是把多张图片整合到一张上的图片。它被运用在众多使用了很多小图标的网站上（Gmail 在使用）。实现方法：

1. 使用生成器将多张图片打包成一张精灵图，并为其生成合适的 CSS。
2. 每张图片都有相应的 CSS 类，该类定义了 `background-image`、`background-position` 和 `background-size` 属性。
3. 使用图片时，将相应的类添加到你的元素中。

好处：

- 减少加载多张图片的 HTTP 请求数（一张精灵图只需要一个请求）。但是对于 HTTP2 而言，加载多张图片不再是问题。
- 提前加载资源，防止在需要时才在开始下载引发的问题，比如只出现在:hover伪类中的图片，不会出现闪烁。

如何解决不同浏览器的样式兼容性问题?

- 在确定问题原因和有问题的浏览器后，使用单独的样式表，仅供出现问题的浏览器加载。这种方法需要使用服务器端渲染。
- 使用已经处理好此类问题的库，比如 Bootstrap。
- 使用 autoprefixer 自动生成 CSS 属性前缀。
- 使用 Reset CSS 或 Normalize.css。

如何为功能受限的浏览器提供页面？ 使用什么样的技术和流程?

- 优雅的降级：为现代浏览器构建应用，同时确保它在旧版浏览器中正常运行。
- 渐进式增强：构建基于用户体验的应用，但在浏览器支持时添加新增功能。
- 利用 caniuse.com 检查特性支持。
- 使用 autoprefixer 自动生成 CSS 属性前缀。
- 使用 Modernizr进行特性检测。

有什么不同的方式可以隐藏内容（使其仅适用于屏幕阅读器）？

这些方法与可访问性（a11y）有关。

- width: 0; height: 0：使元素不占用屏幕上的任何空间，导致不显示它。
- position: absolute; left: -99999px： 将它置于屏幕之外。
- text-indent: -9999px：这只适用于block元素中的文本。
- Metadata： 例如通过使用 Schema.org，RDF 和 JSON-LD。
- WAI-ARIA：如何增加网页可访问性的 W3C 技术规范。

即使 WAI-ARIA 是理想的解决方案，我也会采用绝对定位方法，因为它具有最少的注意事项，适用于大多数元素，而且使用起来非常简单。

你熟悉制作 SVG 吗?

是的，你可以使用内联 CSS、嵌入式 CSS 部分或外部 CSS 文件对形状进行着色（包括指定对象上的属性）。在网上大部分 SVG 使用的是内联 CSS，不过每个类型都有优点和缺点。

通过设置 `fill` 和 `stroke` 属性，可以完成基本着色操作。`fill` 可以设置内部的颜色，`stroke` 可以设置周围绘制的线条的颜色。你可以使用与 `HTML` 中使用的 CSS 颜色命名方案相同的 CSS 颜色命名方案：颜色名称（即`red`）、RGB 值（即`rgb(255,0,0)`）、十六进制值、RGBA 值等等。

```ini复制代码<rect
  x="10"
  y="10"
  width="100"
  height="100"
  stroke="blue"
  fill="purple"
  fill-opacity="0.5"
  stroke-opacity="0.8"
/>
```

编写高效的 CSS 应该注意什么?

首先，浏览器从最右边的选择器，即关键选择器（key selector），向左依次匹配。根据关键选择器，浏览器从 `DOM` 中筛选出元素，然后向上遍历被选元素的父元素，判断是否匹配。选择器匹配语句链越短，浏览器的匹配速度越快。避免使用标签和通用选择器作为关键选择器，因为它们会匹配大量的元素，浏览器必须要进行大量的工作，去判断这些元素的父元素们是否匹配。

BEM (Block Element Modifier)原则上建议为独立的 CSS 类命名，并且在需要层级关系时，将关系也体现在命名中，这自然会使选择器高效且易于覆盖。

搞清楚哪些 CSS 属性会触发重新布局（reflow）、重绘（repaint）和合成（compositing）。在写样式时，避免触发重新布局的可能。

使用 CSS 预处理的优缺点分别是什么?

优点

- 提高 CSS 可维护性。
- 易于编写嵌套选择器。
- 引入变量，增添主题功能。可以在不同的项目中共享主题文件。
- 通过混合（Mixins）生成重复的 CSS。
- 将代码分割成多个文件。不进行预处理的 CSS，虽然也可以分割成多个文件，但需要建立多个 HTTP 请求加载这些文件。

缺点

- 需要预处理工具。
- 重新编译的时间可能会很慢。

描述伪元素及其用途。

CSS 伪元素是添加到选择器的关键字，去选择元素的特定部分。它们可以用于装饰（`:first-line`，`:first-letter`）或将元素添加到标记中（与 content:...组合），而不必修改标记（`:before`，`:after`）。

- :first-line和:first-letter可以用来修饰文字。
- 上面提到的.clearfix方法中，使用clear: both来添加不占空间的元素。
- 使用:before和after展示提示中的三角箭头。鼓励关注点分离，因为三角被视为样式的一部分，而不是真正的 DOM。如果不使用额外的 HTML 元素，- 只用 CSS 样式绘制三角形是不太可能的。

说说你对盒模型的理解，以及如何告知浏览器使用不同的盒模型渲染布局。

CSS 盒模型描述了以文档树中的元素而生成的矩形框，并根据排版模式进行布局。每个盒子都有一个内容区域（例如文本，图像等）以及周围可选的 `padding`、`border` 和 `margin` 区域。

CSS 盒模型负责计算：

- 块级元素占用多少空间。
- 边框是否重叠，边距是否合并。
- 盒子的尺寸。

盒模型有以下规则：

- 块级元素的大小由 `width`、`height`、`padding`、`border` 和 `margin` 决定。
- 如果没有指定 `height`，则块级元素的高度等于其包含子元素的内容高度加上 `padding`（除非有浮动元素，请参阅下文）。
- 如果没有指定 `width`，则非浮动块级元素的宽度等于其父元素的宽度减去父元素的 `padding`。
- 元素的 `height` 是由内容的 `height` 来计算的。
- 元素的 `width` 是由内容的 `width` 来计算的。
- 默认情况下，`padding` 和 `border` 不是元素 `width` 和 `height` 的组成部分。

* { box-sizing: border-box; }会产生怎样的效果?

- 元素默认应用了 `box-sizing: content-box`，元素的宽高只会决定内容（content）的大小。
- `box-sizing: border-box` 改变计算元素 `width` 和 `height` 的方式，`border` 和 `padding` 的大小也将计算在内。
- 元素的height = 内容（content）的高度 + 垂直方向的padding + 垂直方向border的宽度
- 元素的width = 内容（content）的宽度 + 水平方向的padding + 水平方向border的宽度

display的属性值都有哪些?

`none`, `block`, `inline`, `inline-block`, `table`, `table-row`, `table-cell`, `list-item`.

inline和inline-block有什么区别?

我把block也加入其中，为了获得更好的比较。

	block	**inline-block**	**inline**
大小	填充其父容器的宽度。	取决于内容。	取决于内容。
定位	从新的一行开始，并且不允许旁边有 HTML 元素（除非是 `float`）	与其他内容一起流动，并允许旁边有其他元素。	与其他内容一起流动，并允许旁边有其他元素。
能否设置 `width` 和 `height`	能	能	不能。 设置会被忽略。
可以使用 `vertical-align` 对齐	不可以	可以	可以
边距（margin）			只有水平方向存在。垂直方向会被忽

和填充（padding）	各个方向都存在 **block**	各个方向都存在 **inline-block**	略。尽管 `border` 和 `padding` 在 `content` 周围，但垂直方向上的空间取决于 `'line-height'`
浮动（float）	-	-	就像一个 `block` 元素，可以设置垂直边距和填充。

relative、fixed、absolute 和 static 四种定位有什么区别?

经过定位的元素，其 `position` 属性值必然是 `relative`、`absolute`、`fixed` 或 `sticky`。

- `static`：默认定位属性值。该关键字指定元素使用正常的布局行为，即元素在文档常规流中当前的布局位置。此时 top, right, bottom, left 和 z-index 属性无效。
- `relative`：该关键字下，元素先放置在未添加定位时的位置，再在不改变页面布局的前提下调整元素位置（因此会在此元素未添加定位时所在位置留下空白）。
- `absolute`：不为元素预留空间，通过指定元素相对于最近的非 static 定位祖先元素的偏移，来确定元素位置。绝对定位的元素可以设置外边距（margins），且不会与其他边距合并。
- `fixed`：不为元素预留空间，而是通过指定元素相对于屏幕视口（viewport）的位置来指定元素位置。元素的位置在屏幕滚动时不会改变。打印时，元素会出现在的每页的固定位置。fixed 属性会创建新的层叠上下文。当元素祖先的 transform 属性非 none 时，容器由视口改为该祖先。
- `sticky`：盒位置根据正常流计算(这称为正常流动中的位置)，然后相对于该元素在流中的 flow root（BFC）和 containing block（最近的块级祖先元素）定位。在所有情况下（即便被定位元素为 `table` 时），该元素定位均不对后续元素造成影响。当元素 B 被粘性定位时，后续元素的位置仍按照 B 未定位时的位置来确定。`position: sticky` 对 table 元素的效果与 `position: relative` 相同。

你了解 CSS Flexbox 和 Grid 吗?

了解。Flexbox 主要用于一维布局，而 Grid 则用于二维布局。

Flexbox 解决了 CSS 中的许多常见问题，例如容器中元素的垂直居中，粘性定位（sticky）的页脚等。Bootstrap 和 Bulma 基于 Flexbox，这是创建布局的推荐方式。我之前曾使用过 Flexbox，但在使用 `flex-grow` 时遇到了一些浏览器不兼容问题（Safari），我必须使用 `inline-blocks` 和手动计算百分比宽度，来重写我的代码，这种体验不是很好。

`Grid` 创建基于栅格的布局，是迄今为止最直观的方法（最好是！），但目前浏览器支持并不广泛。

请解释在编写网站时，响应式与移动优先的区别。

请注意，这两种方法不是互斥的。

使一个网站响应意味着网站会根据设备屏幕的尺寸会自行调整一些元素的尺寸和功能，通常是通过 CSS 媒体查询的视口宽度，例如，使字体在小屏幕上变小。

```
arduino复制代码@media (min-width: 601px) {
  .my-class {
    font-size: 24px;
  }
}

@media (max-width: 600px) {
  .my-class {
    font-size: 12px;
  }
}
```

移动优先策略同样也指的是响应式，但是它建议我们应该默认定义移动设备的所有样式，仅仅添加特定的规则用来适配其他设备，下面是前一个例子：

```
arduino复制代码.my-class {
  font-size: 12px;
}

@media (min-width: 600px) {
  .my-class {
    font-size: 24px;
  }
}
```

移动优先策略有 2 大优势：

- 在移动设备上有更好的性能，因为应用于它们的规则无需针对任何媒体查询的验证。
- 它让你强制编写与响应 CSS 规则相关的更干净的代码。

响应式设计与自适应设计有何不同?

响应式设计和自适应设计都以提高不同设备间的用户体验为目标，根据视窗大小、分辨率、使用环境和控制方式等参数进行优化调整。

响应式设计的适应性原则：网站应该凭借一份代码，在各种设备上都有良好的显示和使用效果。响应式网站通过使用媒体查询，自适应栅格和响应式图片，基于多种因素进行变化，创造出优良的用户体验。就像一个球通过膨胀和收缩，来适应不同大小的篮圈。

自适应设计更像是渐进式增强的现代解释。与响应式设计单一地去适配不同，自适应设计通过检测设备和其他特征，从早已定义好的一系列视窗大小和其他特性中，选出最恰当的功能和布局。与使用一个球去穿过各种的篮筐不同，自适应设计允许使用多个球，然后根据不同的篮筐大小，去选择最合适的一个。

什么情况下，用translate()而不用绝对定位？什么时候，情况相反。

`translate()` 是 `transform` 的一个值。改变 `transform` 或 `opacity` 不会触发浏览器重新布局（reflow）或重绘（repaint），只会触发复合（compositions）。而改变绝对定位会触发重新布局，进而触发重绘和复合。`transform` 使浏览器为元素创建一个 GPU 图层，但改变绝对定位会使用到 CPU。 因此 `translate()` 更高效，可以缩短平滑动画的绘制时间。

当使用 `translate()` 时，元素仍然占据其原始空间（有点像 `position: relative`），这与改变绝对定位不同。

实现简易版 Promise

以下是一个基本的 Promise 实现：

```
javascript复制代码class MyPromise {
  constructor(executor) {
    this.status = 'pending';
    this.value = undefined;
    this.onResolveCallbacks = [];
    this.onRejectCallbacks = [];

    const resolve = (value) => {
      if (this.status === 'pending') {
        this.status = 'fulfilled';
        this.value = value;
        this.onResolveCallbacks.forEach((callback) => callback(this.value));
      }
    };

    const reject = (reason) => {
      if (this.status === 'pending') {
        this.status = 'rejected';
        this.value = reason;
        this.onRejectCallbacks.forEach((callback) => callback(this.value));
      }
    };

    try {
      executor(resolve, reject);
    } catch (error) {
      reject(error);
    }
  }

  then(onFulfilled, onRejected) {
    if (this.status === 'fulfilled') {
      onFulfilled(this.value);
    } else if (this.status === 'rejected') {
      onRejected(this.value);
    } else {
      this.onResolveCallbacks.push(onFulfilled);
      this.onRejectCallbacks.push(onRejected);
    }
  }
}

// 示例用法
const promise = new MyPromise((resolve, reject) => {
  // 异步操作，比如请求数据
  setTimeout(() => {
    resolve('成功');
    // 或者 reject('失败');
  }, 1000);
});

promise.then(
  (value) => {
```

```
    console.log('成功:', value);
  },
  (reason) => {
    console.log('失败:', reason);
  }
);
```

这是一个非常基本的 Promise 实现，仅用于演示目的。在实际应用中，要考虑更多的细节和错误处理。现代 JavaScript 已经内置了 Promise，通常不需要手动实现它。

实现函数节流

函数节流是一种控制函数执行频率的技术，确保函数在一定时间间隔内最多执行一次。以下是一个简单的 JavaScript 函数节流的实现：

```
javascript复制代码function throttle(func, delay) {
  let timerId;
  let lastExecTime = 0;

  return function (...args) {
    const now = Date.now();
    if (now - lastExecTime >= delay) {
      func.apply(this, args);
      lastExecTime = now;
    } else {
      clearTimeout(timerId);
      timerId = setTimeout(() => {
        func.apply(this, args);
        lastExecTime = Date.now();
      }, delay);
    }
  };
}
```

上述 `throttle` 函数接受两个参数：`func` 是要节流的函数，`delay` 是执行的时间间隔（以毫秒为单位）。

使用这个节流函数，您可以包装需要进行节流的函数，以确保它们不会在短时间内被频繁执行。例如：

```
javascript复制代码// 原始函数，可能会频繁触发
function handleResize() {
  console.log('窗口大小改变了');
}

// 使用节流包装后的函数
const throttledResize = throttle(handleResize, 200); // 200毫秒的节流间隔

// 监听窗口大小改变事件，使用节流函数
window.addEventListener('resize', throttledResize);
```

现在，`handleResize` 函数将在 200 毫秒内最多执行一次，无论窗口大小改变多频繁。这有助于减少频繁的函数调用，提高性能。

实现函数防抖

函数防抖是一种控制函数执行频率的技术，确保函数在一定时间间隔内只执行一次。以下是一个简单的 JavaScript 函数防抖的实现：

```
javascript复制代码function debounce(func, delay) {
  let timerId;

  return function (...args) {
    clearTimeout(timerId);
    timerId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}
```

上述 `debounce` 函数接受两个参数：`func` 是要防抖的函数，`delay` 是等待的时间间隔（以毫秒为单位）。

使用这个防抖函数，您可以包装需要进行防抖的函数，以确保它们只在一定时间间隔后被执行。例如：

```
javascript复制代码// 原始函数，可能会频繁触发
function handleInput(value) {
  console.log('输入值为:', value);
}

// 使用防抖包装后的函数
const debouncedInput = debounce(handleInput, 300); // 300毫秒的防抖间隔

// 监听输入事件，使用防抖函数
document.querySelector('input').addEventListener('input', (event) => {
  debouncedInput(event.target.value);
});
```

现在，`handleInput` 函数将在用户停止输入 300 毫秒后执行，从而减少了频繁的函数调用，提高了性能。

实现观察者模式

观察者模式是一种设计模式，其中一个主题（被观察者）维护了一个观察者列表，并在状态变化时通知观察者。以下是一个简单的 JavaScript 观察者模式的实现：

```
javascript复制代码class Subject {
  constructor() {
    this.observers = [];
  }

  addObserver(observer) {
    this.observers.push(observer);
  }

  removeObserver(observer) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }

  notify(data) {
    this.observers.forEach(observer => observer.update(data));
```

```
  }
}

class Observer {
  constructor(name) {
    this.name = name;
  }

  update(data) {
    console.log(`${this.name} 收到更新，数据为:`, data);
  }
}

// 示例用法
const subject = new Subject();

const observer1 = new Observer('观察者1');
const observer2 = new Observer('观察者2');

subject.addObserver(observer1);
subject.addObserver(observer2);

subject.notify('新数据更新了'); // 观察者1 收到更新，数据为: 新数据更新了
                                // 观察者2 收到更新，数据为: 新数据更新了

subject.removeObserver(observer1);

subject.notify('又有新数据更新了'); // 只有观察者2会收到更新
```

上述代码创建了一个简单的观察者模式实现，包括一个主题类 `Subject` 和一个观察者类 `Observer`。主题可以添加、移除观察者，并在状态变化时通知所有观察者。

在示例中，我们创建了一个主题 `subject`，并添加了两个观察者 `observer1` 和 `observer2`。当主题状态发生变化时，它会通知所有观察者。

这只是一个基本的示例，实际应用中，您可能需要更复杂的实现以满足特定需求。

实现发布订阅模式

订阅者模式也被称为发布-订阅模式，它是一种设计模式，其中一个主题（发布者）维护了一个订阅者列表，并在事件发生时通知所有订阅者。以下是一个简单的 JavaScript 订阅者模式的实现：

```
javascript复制代码class Publisher {
  constructor() {
    this.subscribers = [];
  }

  subscribe(subscriber) {
    this.subscribers.push(subscriber);
  }

  unsubscribe(subscriber) {
    this.subscribers = this.subscribers.filter(sub => sub !== subscriber);
  }

  publish(eventData) {
    this.subscribers.forEach(subscriber => subscriber.notify(eventData));
```

```
  }
}

class Subscriber {
  constructor(name) {
    this.name = name;
  }

  notify(eventData) {
    console.log(`${this.name} 收到通知，事件数据为:`, eventData);
  }
}

// 示例用法
const publisher = new Publisher();

const subscriber1 = new Subscriber('订阅者1');
const subscriber2 = new Subscriber('订阅者2');

publisher.subscribe(subscriber1);
publisher.subscribe(subscriber2);

publisher.publish('新事件发生了'); // 订阅者1 收到通知，事件数据为: 新事件发生了
                                   // 订阅者2 收到通知，事件数据为: 新事件发生了

publisher.unsubscribe(subscriber1);

publisher.publish('又有新事件发生了'); // 只有订阅者2会收到通知
```

在上述代码中，我们创建了一个简单的订阅者模式实现，包括一个发布者类 `Publisher` 和一个订阅者类 `Subscriber`。发布者可以添加、移除订阅者，并在事件发生时通知所有订阅者。

示例中，我们创建了一个发布者 `publisher`，并添加了两个订阅者 `subscriber1` 和 `subscriber2`。当发布者发布事件时，它会通知所有订阅者。

这只是一个基本的示例，实际应用中，您可以根据需要扩展订阅者模式以满足特定需求。

实现 new 关键字

要实现 JavaScript 中 `new` 操作符的基本功能，您可以编写一个函数，该函数接受构造函数和构造函数参数，并返回一个新的对象实例。以下是一个示例的实现：

```
javascript复制代码function myNew(constructor, ...args) {
  // 创建一个新对象，并将其原型指向构造函数的原型
  const obj = Object.create(constructor.prototype);

  // 调用构造函数，将新对象绑定到构造函数的上下文中
  const result = constructor.apply(obj, args);

  // 如果构造函数返回的是一个对象，则返回该对象；否则返回新创建的对象
  return typeof result === 'object' ? result : obj;
}
```

然后，您可以使用 `myNew` 函数来模拟 `new` 操作符的行为。例如：

```
javascript复制代码function Person(name, age) {
  this.name = name;
  this.age = age;
}

// 使用 myNew 模拟 new 操作符
const person1 = myNew(Person, 'Alice', 30);
const person2 = myNew(Person, 'Bob', 25);

console.log(person1); // 输出: Person { name: 'Alice', age: 30 }
console.log(person2); // 输出: Person { name: 'Bob', age: 25 }
```

这个 `myNew` 函数首先创建一个新对象 `obj`，然后将新对象的原型指向构造函数 `constructor` 的原型。接下来，它调用构造函数，并将新对象绑定到构造函数的上下文中。最后，它检查构造函数的返回值，如果是对象则返回该对象，否则返回新创建的对象。

这是一个简单的 `new` 操作符的模拟实现，实际上，`new` 还涉及到原型链等更复杂的特性，但这个示例可以演示基本的原理。

实现 DeepClone

深拷贝（deep clone）是一种在复制对象时，不仅复制对象本身，还递归复制对象内部所有嵌套的对象和属性的操作。以下是一个简单的 JavaScript 深拷贝的实现示例：

```
javascript复制代码function deepClone(obj, hash = new WeakMap()) {
  // 如果是基本数据类型或 null，则直接返回
  if (obj === null || typeof obj !== 'object') {
    return obj;
  }

  // 如果已经拷贝过这个对象，则直接返回之前的拷贝结果，防止循环引用
  if (hash.has(obj)) {
    return hash.get(obj);
  }

  // 根据对象的类型创建新的对象
  const clone = Array.isArray(obj) ? [] : {};

  // 将新对象添加到哈希表
  hash.set(obj, clone);

  // 递归拷贝对象的属性
  for (const key in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, key)) {
      clone[key] = deepClone(obj[key], hash);
    }
  }

  return clone;
}
```

这个 `deepClone` 函数可以深度复制包括对象、数组和嵌套结构在内的复杂数据类型。它使用了一个哈希表 `hash` 来防止循环引用，确保不会陷入无限递归。

示例用法：

```
javascript复制代码const originalObj = {
  name: 'John',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'New York'
  }
};

const clonedObj = deepClone(originalObj);

console.log(clonedObj); // 输出深拷贝后的对象
console.log(originalObj === clonedObj); // 输出 false，说明是不同的对象
```

请注意，这只是一个简单的深拷贝实现示例，实际应用中可能需要更复杂的处理，以应对各种数据类型和情况。

实现函数 Curry

函数柯里化（Currying）是一种将接受多个参数的函数转换为一系列接受单个参数的函数的技术。以下是一个简单的 JavaScript 函数柯里化的实现示例：

```
javascript复制代码function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args);
    } else {
      return function (...moreArgs) {
        return curried.apply(this, args.concat(moreArgs));
      };
    }
  };
}
```

这个 `curry` 函数接受一个函数 `fn`，然后返回一个柯里化后的函数。当柯里化后的函数被调用时，它将检查传入的参数数量是否足够执行原始函数 `fn`。如果参数足够，它会直接调用 `fn`；如果参数不够，它将返回一个新的函数，等待更多参数传入，并持续追加参数，直到参数足够。

示例用法：

```
javascript复制代码function add(a, b, c) {
  return a + b + c;
}

const curriedAdd = curry(add);

console.log(curriedAdd(1)(2)(3)); // 输出 6
console.log(curriedAdd(1, 2)(3)); // 输出 6
console.log(curriedAdd(1)(2, 3)); // 输出 6
```

在示例中，我们首先使用 `curry` 函数将 `add` 函数柯里化，然后可以通过多种方式调用 `curriedAdd` 来实现加法操作。

这只是一个简单的函数柯里化的实现示例，实际应用中，您可能需要更复杂的处理，以应对不同的函数和参数情况。

实现 Call

`call` 是 JavaScript 中用于调用函数的方法，它允许您指定函数内部的 `this` 值并传递参数。以下是一个简单的 JavaScript `call` 方法的模拟实现：

```
javascript复制代码Function.prototype.myCall = function (context, ...args) {
  // 如果没有传递上下文对象，则使用全局对象（浏览器环境下为 window）
  context = context || globalThis;

  // 将当前函数作为上下文对象的一个属性
  const uniqueKey = Symbol('uniqueKey');
  context[uniqueKey] = this;

  // 调用函数，并传递参数
  const result = context[uniqueKey](...args);

  // 删除临时属性
  delete context[uniqueKey];

  return result;
};
```

这个模拟的 `myCall` 方法可以添加到 `Function.prototype` 上，以使所有函数都能够调用它。它接受一个上下文对象 `context` 和一系列参数 `args`。

示例用法：

```
javascript复制代码function greet(greeting) {
  console.log(`${greeting}, ${this.name}`);
}

const person = { name: 'Alice' };

// 使用 myCall 来调用 greet 函数，并指定上下文对象为 person
greet.myCall(person, 'Hello'); // 输出: Hello, Alice
```

在示例中，我们通过 `myCall` 方法来调用 `greet` 函数，并指定 `person` 对象作为上下文对象，这使得 `this` 在函数内部指向了 `person` 对象。

请注意，这只是一个简单的 `call` 方法模拟实现，实际的 `call` 方法还可以处理更多参数和特殊情况。

实现数组拍平

在 JavaScript 中，您可以使用递归或循环来实现数组的拍平（Flatten）。以下是一些拍平数组的方法：

递归方法：

```
javascript复制代码function flattenArray(arr) {
  let result = [];

  for (let i = 0; i < arr.length; i++) {
    if (Array.isArray(arr[i])) {
      // 如果当前元素是数组，递归拍平
      result = result.concat(flattenArray(arr[i]));
    } else {
```

```
      // 如果不是数组，直接添加到结果数组中
      result.push(arr[i]);
    }
  }

  return result;
}

const nestedArray = [1, [2, [3, 4], 5], 6];
const flattenedArray = flattenArray(nestedArray);
console.log(flattenedArray); // 输出: [1, 2, 3, 4, 5, 6]
```

使用 `reduce` 方法：

```
javascript复制代码function flattenArray(arr) {
  return arr.reduce(function (flat, toFlatten) {
    return flat.concat(Array.isArray(toFlatten) ? flattenArray(toFlatten) :
toFlatten);
  }, []);
}

const nestedArray = [1, [2, [3, 4], 5], 6];
const flattenedArray = flattenArray(nestedArray);
console.log(flattenedArray); // 输出: [1, 2, 3, 4, 5, 6]
```

使用ES6的 `Array.flat` 方法：

```
javascript复制代码const nestedArray = [1, [2, [3, 4], 5], 6];
const flattenedArray = nestedArray.flat(Infinity);
console.log(flattenedArray); // 输出: [1, 2, 3, 4, 5, 6]
```

上述方法都可以将嵌套的数组拍平成一个一维数组。选择哪种方法取决于您的项目需求和对兼容性的要求。如果您的环境支持ES6的 `Array.flat` 方法，那是最简单的方式。如果需要兼容旧的环境，可以使用递归或 `reduce` 方法。

封装 Hooks 定时器

要封装一个可以在多个组件中共享的自定义Hooks定时器，您可以创建一个名为 `useTimer` 的自定义Hooks。以下是一个示例：

```
javascript复制代码import { useState, useEffect } from 'react';

function useTimer(initialCount = 0, interval = 1000) {
  const [count, setCount] = useState(initialCount);

  useEffect(() => {
    const timer = setInterval(() => {
      setCount((prevCount) => prevCount + 1);
    }, interval);

    // 在组件卸载时清除定时器
    return () => {
      clearInterval(timer);
    };
  }, [interval]);
```

```
  return count;
}

export default useTimer;
```

这个`useTimer`自定义Hooks接受两个参数：`initialCount`（初始计数值，默认为0）和`interval`（定时器间隔，默认为1000毫秒）。它返回一个表示定时器计数值的`count`状态变量。

您可以在多个组件中使用`useTimer`来创建定时器。以下是一个示例：

```
javascript复制代码import React from 'react';
import useTimer from './useTimer'; // 导入自定义Hooks

function TimerComponent() {
  const count = useTimer(); // 使用自定义Hooks创建定时器

  return (
    <div>
      <h1>定时器示例</h1>
      <p>计数：{count}</p>
    </div>
  );
}

export default TimerComponent;
```

在上述示例中，我们导入了自定义Hooks`useTimer`，然后在`TimerComponent`组件中使用它创建了一个定时器。每个使用`useTimer`的组件都会独立拥有自己的定时器，但它们可以共享相同的定时器逻辑。

您可以在需要的多个组件中使用`useTimer`来创建和管理定时器，以便在整个应用程序中实现共享的定时器功能。

Vue.js 与其他前端框架（如React和Angular）相比有什么优势和区别?

1. 简单性和易用性：

Vue.js 是一款轻量级框架，容易学习和上手。它提供了直观的API和清晰的文档，使开发者可以迅速构建应用程序。 React 和 Angular 在某些方面更复杂，需要更多的学习成本。

1. 渐进式框架：

Vue.js 被称为渐进式框架，允许你逐步采用它的特性。这意味着你可以在现有项目中集成Vue.js，而不必一次性重写整个应用。 React 和 Angular 在集成到现有项目时可能需要更多的工作。

1. 双向数据绑定：

Vue.js 提供了直接的双向数据绑定，使数据在视图和模型之间保持同步。这使得开发人员更容易管理应用程序的状态。 React 和 Angular 也支持数据绑定，但它们的实现方式略有不同。

1. 组件化开发：

Vue.js、React 和 Angular 都鼓励组件化开发，但Vue.js在这方面表现出色。Vue组件的定义非常简单，易于复用和维护。 React 使用JSX来创建组件，Angular使用模板。这些框架的组件系统也很强大，但可能需要更多的配置。

1. 生态系统和社区：

React 和 Angular 有庞大的生态系统和活跃的社区支持，有丰富的第三方库和插件。 Vue.js 的生态系统也在不断壮大，虽然相对较小，但社区也非常积极。

1. 性能：

Vue.js 在性能方面表现良好，具有虚拟DOM机制，可以高效地更新视图。 React 也使用虚拟DOM，性能也很出色。Angular 在某些情况下可能需要更多的性能优化工作。

1. 工具和生态系统：

Vue.js 提供了一些强大的工具，如Vue CLI，用于快速搭建项目，并与Vue Router和Vuex等官方库集成。 React 和 Angular 也有类似的工具和库，但Vue的工具生态系统在某些方面更加直观和易用。

1. 使用案例：

Vue.js 适用于中小型应用程序和单页面应用程序（SPA），以及需要快速原型开发的项目。 React 和 Angular 适用于各种规模的应用，包括大型企业级应用。总之，选择使用哪个前端框架取决于项目的需求和团队的偏好。Vue.js在简单性、易用性和渐进式开发方面具有优势，适合许多项目，但React和Angular在大型应用和企业级项目中也有其优势。

Vue实例与组件之间的区别是什么？它们如何进行通信？

Vue.js 中的 Vue 实例（Vue Instance）和组件（Components）是两个不同的概念，它们之间有一些重要的区别，同时也有不同的方式来进行通信。

1. Vue 实例（Vue Instance）：

- Vue 实例是 Vue.js 的核心概念之一。它是一个独立的 Vue 对象，用来管理应用的状态、行为和生命周期。
- 通常，一个 Vue 应用的根实例会被创建，它管理整个应用的数据和方法。你可以使用 `new Vue()` 来创建一个 Vue 实例。

2. 组件（Components）：

- 组件是 Vue.js 中的可复用的代码块，用于构建用户界面。每个组件都有自己的状态、行为和模板。
- 组件可以像标签一样在模板中使用，允许你构建复杂的用户界面，将界面分解成可维护的部分。
- 通过 `Vue.component` 或使用单文件组件 (`.vue` 文件) 的方式定义组件。

通信方式：

在 Vue.js 中，Vue 实例和组件之间可以通过以下方式进行通信：

1. Props（属性）：

- 父组件可以通过 props 向子组件传递数据。子组件通过 props 接收数据并在自己的模板中使用。
- 这是一种单向数据流的方式，父组件向子组件传递数据。

2. 自定义事件：

- 子组件可以通过触发自定义事件来向父组件通知事件发生。父组件可以监听这些事件并执行相应的操作。
- 这是一种从子组件到父组件的通信方式。

3. 状态管理（如Vuex）：

- 对于大型应用程序，可以使用状态管理库如 Vuex 来管理应用的状态。它提供了一个集中的状态存储，所有组件都可以访问和修改其中的数据。
- 这是一种跨组件通信的高级方式。

4. 依赖注入：

- Vue.js 提供了依赖注入机制，允许你在祖先组件中注册一些数据，然后在后代组件中访问这些数据，而不需要通过 props 一层层传递。
- 依赖注入通常用于一些全局配置或主题样式的传递。

总结： Vue 实例是整个应用的根对象，而组件是应用中的可复用模块。它们之间的通信主要通过 props 和自定义事件来实现，但对于更复杂的状态管理，可以使用 Vuex 或其他状态管理库。

Vue中的声明周期钩子函数是什么？它们的执行顺序是怎样的?

Vue.js 中的生命周期钩子函数是一组特定的函数，它们允许你在组件的不同生命周期阶段执行代码。这些钩子函数可以用于执行初始化、数据加载、DOM 操作等任务。Vue 组件的生命周期钩子函数按照以下顺序执行：

1. **beforeCreate（创建前）：**
    - 在组件实例被创建之前立即调用。
    - 此时组件的数据和事件还未初始化。
2. **created（创建后）：**
    - 在组件实例被创建后立即调用。
    - 组件的数据已经初始化，但此时还未挂载到 DOM。
3. **beforeMount（挂载前）：**
    - 在组件挂载到 DOM 之前立即调用。
    - 此时模板编译完成，但尚未将组件渲染到页面上。
4. **mounted（挂载后）：**
    - 在组件挂载到 DOM 后立即调用。
    - 此时组件已经渲染到页面上，可以进行 DOM 操作。
5. **beforeUpdate（更新前）：**
    - 在组件数据更新之前立即调用。
    - 在此钩子函数内，你可以访问之前的状态，但此时尚未应用最新的数据。
6. **updated（更新后）：**
    - 在组件数据更新后立即调用。
    - 此时组件已经重新渲染，可以进行 DOM 操作。
7. **beforeDestroy（销毁前）：**
    - 在组件销毁之前立即调用。
    - 此时组件仍然可用，你可以执行一些清理工作。
8. **destroyed（销毁后）：**
    - 在组件销毁后立即调用。
    - 此时组件已经被完全销毁，不再可用。

这些生命周期钩子函数允许你在不同的阶段执行代码，以满足应用程序的需求。例如，在 `created` 钩子中可以进行数据初始化，而在 `mounted` 钩子中可以进行 DOM 操作。请注意，不同的生命周期钩子适合不同的用途，应根据需要选择合适的钩子函数来执行相应的任务。

Vue的双向数据绑定是如何实现的？请举例说明。

Vue.js 的双向数据绑定是通过其特有的响应式系统来实现的。这个系统使用了ES6的Proxy对象或者Object.defineProperty()方法，以便在数据变化时通知视图进行更新。这意味着当你修改数据模型时，与之相关联的视图会自动更新，反之亦然。

下面是一个简单的示例，演示了如何在Vue.js中实现双向数据绑定：

HTML模板：

```
html复制代码<div id="app">
  <input v-model="message" type="text">
  <p>{{ message }}</p>
</div>
```

Vue实例的JavaScript代码：

```
javascript复制代码new Vue({
  el: '#app',
  data: {
    message: 'Hello, Vue!'
  }
})
```

在这个示例中，我们使用了`v-model`指令将`元素与Vue实例中的message属性双向绑定。这意味着当你在输入框中输入文本时，message的值会自动更新，同时当message`的值变化时，文本也会自动更新。

当你在输入框中输入文字时，Vue会自动将输入的值更新到`message`属性中，因此实现了从视图到数据的更新。反过来，如果你在JavaScript代码中修改了`message`属性的值，视图中的文本也会自动更新，实现了从数据到视图的更新。

这种双向数据绑定使得数据与视图保持同步，大大简化了前端开发中处理用户输入和数据展示的任务。

Vue中的计算属性和观察者的作用是什么？它们有什么区别？

在Vue.js中，计算属性（Computed Properties）和观察者（Watchers）都用于处理数据的变化，但它们有不同的作用和用途。

计算属性（Computed Properties）：

计算属性是Vue.js中的一种属性类型，它的值是基于其他数据属性计算而来的，类似于一个函数。计算属性的主要作用是将计算逻辑封装起来，以便在模板中直接引用，而且它们具有缓存机制，只有在依赖的数据发生变化时才会重新计算。

主要特点和作用：

- 用于派生或计算基于现有数据属性的值。
- 具有缓存机制，只有在相关数据发生变化时才会重新计算，提高性能。
- 在模板中可以像普通属性一样直接引用。
- 计算属性一般用于简单的数据转换、筛选、格式化等操作。

示例：

```
vue复制代码<template>
  <div>
    <p>{{ fullName }}</p>
  </div>
</template>

<script>
export default {
```

```
  data() {
    return {
      firstName: 'John',
      lastName: 'Doe'
    }
  },
  computed: {
    fullName() {
      return this.firstName + ' ' + this.lastName;
    }
  }
}
</script>
```

观察者（Watchers）：

观察者是Vue.js中的一种方式，用于在数据变化时执行自定义的异步或开销较大的操作。你可以监听一个或多个数据属性的变化，并在数据变化时执行特定的函数。

主要特点和作用：

- 用于在数据变化时执行自定义的操作，例如异步请求或复杂的数据处理。
- 不具有缓存机制，每次数据变化都会触发执行。
- 需要手动编写观察者函数来处理数据变化。
- 可以监听多个数据属性的变化。

示例：

```
vue复制代码<template>
  <div>
    <p>{{ message }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      value: 'Initial Value',
      message: ''
    }
  },
  watch: {
    value(newValue, oldValue) {
      // 在value属性变化时执行的操作
      this.message = 'Value changed: ' + newValue;
    }
  }
}
</script>
```

区别：

1. **计算属性**主要用于对数据的转换和派生，具有缓存机制，只有在相关数据变化时才会重新计算，适合用于简单的数据处理。它们在模板中可以像普通属性一样直接引用。
2. **观察者**用于在数据变化时执行自定义的操作，没有缓存机制，每次数据变化都会触发执行。适合处理复杂的异步操作或需要监听多个数据变化的情况。

根据具体的需求，你可以选择使用计算属性或观察者来处理数据变化。通常，计算属性是首选，因为它们更简单且性能更高，而只有在需要特殊处理数据变化时才使用观察者。

谈谈你对Vue组件的理解。如何创建一个Vue组件?

Vue 组件是 Vue.js 应用中的可复用模块，它将一个页面拆分成多个独立的部分，每个部分有自己的状态、模板和行为。组件化是 Vue.js 的核心概念之一，它使前端开发更加模块化、可维护和可重用。

创建一个 Vue 组件的基本步骤如下：

1. **定义组件：** 首先，你需要定义一个 Vue 组件。组件可以使用 `Vue.component` 方法或者使用单文件组件（.vue 文件）来定义。以下是一个使用 `Vue.component` 定义组件的示例：

```
vue复制代码Vue.component('my-component', {
  // 组件的选项
  template: '<div>This is a custom component</div>'
})
```

1. **在模板中使用组件：** 一旦定义了组件，你可以在父组件的模板中使用它。例如：

```
vue复制代码<template>
  <div>
    <my-component></my-component>
  </div>
</template>
```

1. **传递数据给组件：** 你可以通过组件的 props 来传递数据给组件，使组件可以接收外部数据并在模板中使用。例如：

```
vue复制代码<template>
  <div>
    <my-component :message="message"></my-component>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello from parent component'
    }
  }
}
</script>
```

在组件内部，你可以使用 `props` 来接收这个数据，并在模板中使用它：

```
vue复制代码<template>
  <div>
    <p>{{ message }}</p>
  </div>
</template>

<script>
export default {
  props: ['message']
}
</script>
```

1. **组件的生命周期：** 组件也具有生命周期钩子函数，允许你在不同的生命周期阶段执行代码。这些钩子函数包括 `beforeCreate`、`created`、`beforeMount`、`mounted` 等，用于执行初始化、数据加载、DOM 操作等任务。
2. **自定义事件：** 组件之间可以通过自定义事件进行通信。子组件可以触发自定义事件，而父组件可以监听这些事件并执行相应的操作。
3. **组件之间的通信：** 除了 props 和自定义事件，你还可以使用 Vuex 这样的状态管理工具来实现组件之间的通信和数据共享。

总之，Vue 组件是 Vue.js 应用中的核心概念之一，它使前端开发更加模块化和可维护，允许你将界面拆分成多个可复用的部分，每个部分都有自己的状态和行为。创建和使用组件是 Vue.js 开发中的重要部分，帮助你构建更高效和可维护的前端应用程序。

Vue中的指令是什么？列举一些常用的指令，并简要介绍它们的作用。

在 Vue.js 中，指令（Directives）是一种特殊的 token，可以在模板中使用，以表示对 DOM 元素的行为。指令以 `v-` 开头，后面跟着指令的名称，例如 `v-bind`、`v-if` 等。指令用于将模板中的数据与 DOM 元素进行绑定，控制元素的显示、隐藏、渲染和行为等。

以下是一些常用的 Vue 指令以及它们的作用：

1. `v-bind`：
   - 作用：用于绑定元素的属性，将元素的属性值与 Vue 实例的数据进行绑定。
   - 示例：``
2. `v-model`：
   - 作用：用于实现表单元素与 Vue 实例数据的双向绑定，使用户输入能够自动更新数据，反之亦然。
   - 示例：``
3. `v-for`：
   - 作用：用于循环渲染一个数组或对象的数据，生成多个元素。
   - 示例：`{{ item }}`
4. `v-if` / `v-else-if` / `v-else`：
   - 作用：用于根据条件控制元素的显示和隐藏，类似于JavaScript中的条件语句。
   - 示例：`This is shown`
5. `v-show`：
   - 作用：用于根据条件控制元素的显示和隐藏，不同于`v-if`，它是通过 CSS 的 `display` 属性来控制，不会销毁和重新创建元素。
   - 示例：`This is shown`
6. `v-on`：

    - 作用：用于监听 DOM 事件，并在事件触发时执行指定的方法。
    - 示例：`Click me`
7. `v-pre`：
    - 作用：跳过此元素和其子元素的编译过程，直接将其作为原始HTML输出。
    - 示例：`{{ message }}`
8. `v-cloak`：
    - 作用：在元素和Vue实例之间保持隐藏，直到Vue编译完成。
    - 示例：`{{ message }}`
9. `v-once`：
    - 作用：只渲染元素和组件一次，不再进行响应式更新。
    - 示例：`{{ message }}`

这些指令使你能够轻松地在模板中操作 DOM 元素，根据数据的变化实现视图的动态更新。每个指令都有自己的特定作用，让你能够以声明性的方式定义页面的交互和逻辑。你可以根据需要在模板中使用这些指令，从而构建强大的 Vue.js 应用程序。

Vuex是什么？它的作用是什么？请描述Vuex应用程序的基本结构。

Vuex 是一个专为 Vue.js 应用程序开发的状态管理库。它主要用于管理 Vue.js 应用中的共享状态（如数据、状态、配置信息等），以便更好地组织、维护和跟踪应用中的数据流。Vuex 的核心思想是将应用中的状态集中存储在一个全局的 store 中，使得状态的变化可预测且可维护。

Vuex 的主要作用包括：

1. **集中式状态管理：** Vuex 允许将应用的状态存储在一个单一的地方，称为 store。这个 store 是一个响应式的状态树，多个组件可以共享并访问这个状态，而不需要通过 props 层层传递数据。
2. **状态变化可追踪：** Vuex 使用了严格的状态变化追踪机制，每次状态发生变化时都会有明确的记录和日志，方便开发者追踪和调试应用。
3. **组件通信：** Vuex 提供了一种统一的方式来管理组件之间的通信。组件可以通过提交 mutations 来修改状态，也可以通过派发 actions 来触发异步操作，并且这些操作都是可预测且可控制的。
4. **中间件：** Vuex 支持中间件，可以在状态变化时执行一些额外的逻辑，例如日志记录、数据持久化等。

一个基本的 Vuex 应用程序通常包括以下组件：

- **State（状态）：** 存储应用程序的状态数据，通常是一个 JavaScript 对象。
- **Mutations（突变）：** 用于修改状态的方法。每个 mutation 都有一个类型（type）和一个处理函数，用来执行实际的状态修改操作。
- **Actions（动作）：** 类似于 mutations，但是它可以包含异步操作，通常用于处理与服务器交互、数据获取等。Actions 负责提交 mutations 来修改状态。
- **Getters（计算属性）：** 用于从状态中派生出一些新的数据，类似于计算属性，可以被组件直接使用。
- **Store（存储）：** 将状态、mutations、actions、getters 集中管理的对象，是 Vuex 的核心。

下面是一个简单的 Vuex 应用程序的基本结构示例：

```
javascript复制代码import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

const store = new Vuex.Store({
```

```
  state: {
    count: 0
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    }
  },
  actions: {
    incrementAsync(context) {
      setTimeout(() => {
        context.commit('increment')
      }, 1000)
    }
  },
  getters: {
    doubleCount(state) {
      return state.count * 2
    }
  }
})

export default store
```

在上述示例中，我们定义了一个包含状态、突变、动作和计算属性的 Vuex store。这个 store 可以在 Vue 组件中被引用，并用于管理和操作应用程序的状态。 Vuex 的使用可以极大地简化状态管理和组件通信，特别是在大型应用程序中。

Vue Router是什么？它的作用是什么？请描述Vue Router的基本使用方法。

Vue Router 是 Vue.js 官方的路由管理库，用于构建单页应用程序（SPA）。它允许你在 Vue 应用中实现页面之间的导航、路由跳转和 URL 的管理。Vue Router 的主要作用是将不同的视图组件与应用的不同路由（URL 地址）进行关联，从而实现页面之间的切换和导航。

Vue Router 的基本使用方法包括以下步骤：

1. **安装 Vue Router：** 首先，在你的 Vue.js 项目中安装 Vue Router。你可以使用 npm 或 yarn 进行安装：

   ```
   bash复制代码npm install vue-router
   # 或者
   yarn add vue-router
   ```

2. **创建路由配置：** 在你的项目中创建一个路由配置文件，通常命名为 `router.js`，并导入 Vue 和 Vue Router：

   ```
   javascript复制代码import Vue from 'vue'
   import VueRouter from 'vue-router'

   Vue.use(VueRouter)
   ```

```
const routes = [
  {
    path: '/',          // 路由路径
    component: Home     // 对应的视图组件
  },
  {
    path: '/about',
    component: About
  }
  // 其他路由配置
]

const router = new VueRouter({
  routes // 使用配置文件中的路由规则
})

export default router
```

3. **创建视图组件：** 为每个路由路径创建对应的视图组件。这些组件可以是普通的 Vue 组件，例如 `Home.vue` 和 `About.vue`。
4. **在根组件中使用 Router：** 在根 Vue 实例中使用 Vue Router，通常是在 `main.js` 中：

```
javascript复制代码import Vue from 'vue'
import App from './App.vue'
import router from './router' // 导入路由配置

new Vue({
  el: '#app',
  router, // 使用路由配置
  render: h => h(App)
})
```

5. **使用 `和`：** 在模板中使用 `标签来创建导航链接，使用` 标签来渲染当前路由的视图组件。例如：

```
vue复制代码<template>
  <div>
    <router-link to="/">Home</router-link>
    <router-link to="/about">About</router-link>

    <router-view></router-view>
  </div>
</template>
```

6. **导航和路由跳转：** 你可以使用 `` ` 来实现路由导航，也可以在组件中使用 `` this.$router.push()`` 方法来进行编程式的路由跳转。

这些是 Vue Router 的基本使用方法。它允许你在 Vue.js 应用中轻松实现页面之间的导航和路由切换，使单页应用程序的开发更加方便和可维护。通过定义路由配置和关联视图组件，你可以构建出丰富的单页应用程序，将不同的视图组件与不同的 URL 路由进行关联。

Vue2 和 Vue3 的区别?

Vue.js 2 和 Vue.js 3 之间存在一些重要的区别和改进。以下是一些主要的区别和特点：

1. **性能优化：**

- Vue 3 在底层进行了许多性能优化，包括虚拟 DOM 的升级，使其更快速和高效。
- Vue 3 引入了懒加载（Lazy Loading）和静态提升（Static Hoisting）等优化策略，进一步提高了性能。

2. **Composition API：**
   - Vue 3 引入了 Composition API，这是一个基于函数的 API，可以更灵活地组织和重用组件逻辑。
   - Composition API 允许开发者按功能划分代码，提高了代码的可读性和维护性。
3. **更小的包体积：**
   - Vue 3 的核心库体积更小，因此加载更快。
   - Vue 3 支持按需加载，使得只引入需要的功能，进一步减小包体积。
4. **Teleport：**
   - Vue 3 引入了 Teleport，允许将组件的内容渲染到 DOM 中的任何位置，这在处理模态框、弹出菜单等场景中非常有用。
5. **Fragments：**
   - Vue 3 支持 Fragments，允许组件返回多个根元素，而不需要额外的容器元素。
6. **全局 API 的修改：**
   - Vue 3 对全局 API 进行了一些修改，使其更符合现代 JavaScript 的标准。
   - 例如，`Vue.component` 现在改为 `app.component`，`Vue.directive` 改为 `app.directive`，`Vue.mixin` 改为 `app.mixin`。
7. **新的生命周期钩子：**
   - Vue 3 引入了新的生命周期钩子，如 `onBeforeMount` 和 `onBeforeUpdate`，以提供更精确的控制和更好的性能优化机会。
8. **TypeScript 支持改进：**
   - Vue 3 对 TypeScript 的支持更加完善，提供了更好的类型推断和类型检查。
9. **响应式系统的改进：**
   - Vue 3 对响应式系统进行了改进，提供了更好的 TypeScript 支持，并且更加高效。

总的来说，Vue.js 3 在性能、开发体验和可维护性等方面都有显著的改进。然而，Vue 2 仍然是一个稳定的版本，具有广泛的生态系统和支持，开发者可以根据项目需求来选择使用哪个版本。如果你正在开始一个新项目，Vue 3 可能是一个更好的选择，因为它具备了许多优势和改进。如果你正在维护一个 Vue 2 项目，也可以考虑逐渐迁移到 Vue 3，以获得性能和开发体验上的改进。

你能列举一些 Vue3 中的新特性吗?

以下是 Vue.js 3 中一些重要的新特性和改进：

1. **Composition API：** Composition API 是 Vue 3 最引人注目的新特性之一。它允许你按功能划分代码，将相关的代码逻辑组织在一起，提高了可维护性和代码复用性。
2. **Teleport：** Teleport 是一个新的特性，允许你将组件的内容渲染到 DOM 中的其他位置。这对于创建模态框、弹出菜单等组件非常有用。
3. **Fragments：** Vue 3 支持 Fragments，允许一个组件返回多个根元素，而不需要额外的包装容器元素。
4. **全局 API 的修改：** Vue 3 对全局 API 进行了一些修改，使其更符合现代 JavaScript 的标准。例如，`Vue.component` 现在改为 `app.component`。
5. **性能优化：** Vue 3 在底层进行了许多性能优化，包括虚拟 DOM 的升级，懒加载和静态提升等策略，使应用程序更快速和高效。
6. **响应式系统改进：** Vue 3 对响应式系统进行了改进，提供了更好的 TypeScript 支持和更高效的响应式数据追踪。
7. **TypeScript 支持：** Vue 3 对 TypeScript 的支持更加完善，提供了更好的类型推断和类型检查。
8. **更小的包体积：** Vue 3 的核心库体积更小，加载更快，并且支持按需加载，减小了包体积。

9. **生命周期钩子的改进：** Vue 3 引入了新的生命周期钩子，如 `onBeforeMount` 和 `onBeforeUpdate`，提供了更精确的控制和性能优化的机会。
10. **Suspense：** Vue 3 支持 Suspense 特性，允许你优雅地处理异步组件的加载状态，提供更好的用户体验。
11. **自定义渲染器：** Vue 3 允许你创建自定义渲染器，这使得你可以在不同的目标环境中使用 Vue，例如服务器端渲染（SSR）或原生应用。
12. **V-model 的改进：** Vue 3 改进了 v-model 的语法，使其更加灵活，可以用于自定义组件的双向绑定。

这些新特性和改进使 Vue.js 3 成为一个更加强大、高效和灵活的前端框架，有助于开发者构建更优秀的单页应用和用户界面。

请解释 Composition API 是什么以及它的优势是什么?

Composition API 是 Vue.js 3 中引入的一种新的组件组织方式，它允许你按功能划分和组织组件的代码逻辑。这是一种基于函数的 API 风格，与传统的 Options API 相对立，它的主要优势包括：

1. **更灵活的代码组织：** Composition API 允许你将一个组件的代码逻辑分成多个功能相关的部分，每个部分都是一个独立的函数。这使得代码更加清晰，易于维护和测试。你可以更容易地重用代码逻辑，将其应用于多个组件。
2. **更好的类型推断：** Composition API 配合 TypeScript 使用时，

Vue 3 中有哪些性能优化措施?

Vue 3 在性能优化方面引入了许多新特性和改进，以提高应用程序的性能。以下是一些Vue 3 中的性能优化措施：

1. **虚拟 DOM 重写**：Vue 3 的虚拟 DOM 实现进行了重写，使其更快速和轻量化。这意味着渲染和更新性能更高。
2. **静态树提升**：Vue 3 可以检测静态的子树，并将其提升为静态 vnode，以避免不必要的重新渲染和对比操作。
3. **树懒加载**：Vue 3 支持树懒加载，只在需要时才会渲染子组件，减少了初始渲染的负担。
4. **更好的事件处理**：Vue 3 采用了更高效的事件监听和处理方式，提高了事件处理性能。
5. **编译器优化**：Vue 3 的模板编译器进行了优化，生成更有效的渲染函数，减少了运行时的开销。
6. **Fragment 和 Teleport**：Vue 3 引入了Fragment和Teleport，这些特性可以帮助你更有效地组织你的组件，减少不必要的嵌套和渲染节点。
7. **Suspense**：Vue 3 中的Suspense特性允许你在异步组件加载时显示占位符，这有助于提高用户体验，同时减少了不必要的渲染。
8. **响应式系统重写**：Vue 3 的响应式系统进行了重写，使其更快速和可扩展。它采用了Proxy代理，比Vue 2的Object.defineProperty更高效。
9. **Composition API**：Vue 3 引入了Composition API，允许你更灵活地组织和重用代码，这有助于提高代码的性能和可维护性。
10. **Tree-Shaking**：由于Vue 3采用了ES模块的方式组织代码，因此Webpack等构建工具可以更容易地进行Tree-Shaking，只包含应用程序实际使用的代码，减小了包的大小。

这些性能优化措施使Vue 3成为一个更快速和高效的前端框架，有助于构建更具响应性和流畅性的Web应用程序。但请注意，性能优化也取决于你的具体应用程序和使用方式，因此在实际项目中，你可能需要进一步的性能分析和调整。

什么是 Teleport 和 Fragments，它们在 Vue 3 中的作用是什么?

在 Vue 3 中，Teleport 和 Fragments 是两个新的特性，它们分别用于改善组件的渲染结构和渲染位置的控制。以下是它们的作用和用法：

1. **Teleport（传送门）**：
   - **作用**：Teleport 允许你将组件的内容渲染到 DOM 结构的不同位置，而不受父组件的限制。这对于处理模态框、对话框、通知消息等需要在页面的不同位置渲染的情况非常有用。
   - **用法**：你可以在模板中使用 `` ` 元素，并将其 to ` `` 属性设置为一个目标选择器，以指定内容应该被渲染到哪个 DOM 元素中。例如：

     ```
     vue复制代码<template>
       <div>
         <button @click="showModal">Show Modal</button>
         <teleport to="#modal-container">
           <Modal v-if="isModalVisible" @close="closeModal" />
         </teleport>
       </div>
     </template>
     ```

     在上面的示例中，Modal 组件的内容会被渲染到页面中具有 `id="modal-container"` 的 DOM 元素内部。

2. **Fragments（片段）**：
   - **作用**：Fragments 允许你在不引入额外的 DOM 元素的情况下，将多个子元素包裹在一个父元素中。这有助于减少 DOM 结构的嵌套，使代码更清晰和简洁。
   - **用法**：你可以使用 `元素或 Vue 3 提供的特殊语法` v-fragment ` 来创建一个 Fragment。例如：

     ```
     vue复制代码<template>
       <div>
         <p>Paragraph 1</p>
         <p>Paragraph 2</p>
         <v-fragment>
           <p>Paragraph 3</p>
           <p>Paragraph 4</p>
         </v-fragment>
       </div>
     </template>
     ```

     在上面的示例中，`包裹了两个` 元素，但最终渲染的 DOM 结构中并不会包含额外的父元素。

Teleport 和 Fragments 是 Vue 3 中的两个强大的工具，它们有助于更灵活、更清晰地管理组件的渲染结构，同时提高了代码的可读性和维护性。这两个特性在处理复杂布局和可复用组件时尤其有用。

Vue 3 中对全局 API 进行了哪些修改？如何使用这些修改后的 API？

Vue 3 对全局 API 进行了一些修改，以提供更好的性能和功能。以下是一些主要的修改和如何使用这些修改后的 API：

1. **创建 Vue 实例**：
   - **修改前（Vue 2）**：在 Vue 2 中，你可以使用 `new Vue()` 创建根 Vue 实例。
   - **修改后（Vue 3）**：在 Vue 3 中，你可以使用 `createApp()` 来创建应用实例，例如：

```
javascript复制代码import { createApp } from 'vue';
const app = createApp(App);
app.mount('#app');
```

2. **全局组件的注册**：
   - **修改前（Vue 2）**：在 Vue 2 中，你可以使用 `Vue.component()` 全局注册组件。
   - **修改后（Vue 3）**：在 Vue 3 中，你可以使用 `app.component()` 注册全局组件，例如：

```
javascript复制代码app.component('my-component', MyComponent);
```

3. **过滤器**：
   - **修改前（Vue 2）**：Vue 2 中支持过滤器，但在 Vue 3 中已经移除了过滤器的概念。你可以使用计算属性或方法来代替过滤器的功能。
4. **混入（Mixins）**：
   - **修改前（Vue 2）**：在 Vue 2 中，你可以使用 `mixins` 选项来混入组件的选项。
   - **修改后（Vue 3）**：在 Vue 3 中，你可以使用 `mix` 函数来实现类似的功能，例如：

```
javascript复制代码import { defineComponent, ref, mix } from 'vue';

const mixin = {
  data() {
    return {
      message: 'Hello from mixin'
    };
  }
};

const MyComponent = defineComponent({
  mixins: [mixin],
  setup() {
    const count = ref(0);
    return {
      count
    };
  },
  template: `
    <div>
      {{ message }}
      {{ count }}
    </div>
  `
});
```

5. **自定义指令**：
   - **修改前（Vue 2）**：在 Vue 2 中，你可以使用 `Vue.directive()` 注册全局自定义指令。
   - **修改后（Vue 3）**：在 Vue 3 中，你可以使用 `app.directive()` 注册全局自定义指令，例如：

```
javascript复制代码app.directive('my-directive', {
  // 自定义指令的定义
});
```

这些是一些主要的全局 API 修改。在 Vue 3 中，全局 API 的使用方式与 Vue 2 有一些不同，因此在迁移项目或编写新的 Vue 3 代码时，需要注意这些变化。你需要根据具体的情况来使用新的 API，以确保你的应用能够充分利用 Vue 3 的功能和性能优势。

请解释 Vue 3 中的响应式系统是如何工作的?

Vue 3 的响应式系统是其核心功能之一，它允许你在应用程序中实现数据与视图的自动同步。下面是Vue 3中的响应式系统如何工作的简要解释：

1. **初始化**：
   - 当你创建一个Vue 3组件或应用程序时，Vue会初始化一个响应式系统的实例。
2. **数据定义**：
   - 你通过在组件的 `setup` 函数中创建响应式数据。这可以通过 `ref`、`reactive`、或 `computed` 来实现。
3. **数据依赖追踪**：
   - 当组件渲染时，Vue 会自动追踪数据属性的依赖关系。这意味着Vue知道哪些数据属性被用于渲染视图。
4. **响应式依赖收集**：
   - Vue 会在组件渲染期间收集数据属性的依赖，构建一个依赖关系图。
5. **数据变更时触发**：
   - 当响应式数据属性发生变化时，Vue 会通知依赖于该数据属性的视图更新。
6. **批量更新**：
   - Vue 3 会将多个数据变更的通知进行批处理，以最小化 DOM 更新操作，提高性能。
7. **异步更新队列**：
   - Vue 3 使用微任务队列（如 `Promise` 或 `nextTick`）来处理数据更新，确保在同一事件循环中的多次数据变更只触发一次视图更新。
8. **视图更新**：
   - 一旦数据变更通知到视图，Vue 3 会重新渲染相关的组件部分，使其与最新的数据保持同步。
9. **计算属性和侦听器**：
   - Vue 3 允许你使用计算属性（`computed`）和侦听器（`watch`）来处理数据的派生和监听变化，这些特性也依赖于响应式系统来工作。

总的来说，Vue 3的响应式系统通过数据依赖追踪和自动的视图更新机制，实现了数据与视图之间的自动同步。这使得开发者可以更专注于数据的处理，而不必手动操作DOM，提高了开发效率并改善了代码的可维护性。

Ref 和 Reactive 的区别是什么?

`ref` 和 `reactive` 是 Vue 3 中用于创建响应式数据的两种不同方式，它们有一些重要的区别：

1. **引用类型**：
   - `ref`：`ref` 用于创建单个响应式数据。它将一个普通的 JavaScript 值（如数字、字符串等）包装在一个具有 `.value` 属性的对象中，使其成为响应式数据。
   - `reactive`：`reactive` 用于创建一个包含多个属性的响应式对象。它接受一个普通 JavaScript 对象，并返回一个响应式代理对象，这个代理对象可以让对象内的属性变成响应式数据。
2. **访问方式**：
   - `ref`：你可以通过 `.value` 属性来访问 `ref` 中的值。例如：`myRef.value`。

- `reactive`：你可以直接访问 `reactive` 对象内的属性。例如：`myReactiveObj.someProperty`。

3. **用途**：
   - `ref`：通常用于包装基本数据类型，如数字、字符串、布尔值等，或者用于包装需要通过 `.value` 来更新的数据。
   - `reactive`：通常用于创建包含多个属性的响应式数据对象，比如复杂的配置对象或组件的状态。
4. **示例**：
   - 使用 `ref` 创建响应式数据：

```
javascript复制代码import { ref } from 'vue';

const count = ref(0); // 创建一个包装数字的 ref
```

   - 使用 `reactive` 创建响应式对象：

```
javascript复制代码import { reactive } from 'vue';

const person = reactive({
  name: 'Alice',
  age: 30
}); // 创建一个包含多个属性的响应式对象
```

总的来说，`ref` 用于创建单个响应式数据，通常用于包装基本数据类型。而 `reactive` 用于创建包含多个属性的响应式对象，通常用于复杂的数据结构或组件状态的管理。选择使用哪种方式取决于你的具体需求和数据结构。

Vue 3 对 TypeScript 的支持有哪些改进？如何在 Vue 3 中使用 TypeScript?

Vue 3 对 TypeScript 的支持有很多改进，使得在使用 TypeScript 和 Vue 3 结合开发变得更加流畅和类型安全。以下是一些关键的改进和使用 TypeScript 的指南：

1. 类型推断和类型声明：

- Vue 3 提供了更强大的类型推断，允许你获得更准确的类型检查。
- Vue 3 本身附带了 TypeScript 声明文件，因此你不需要额外安装声明文件。

2. 单文件组件：

- 单文件组件（.vue 文件）中的 `` 部分可以使用 TypeScript 编写。
- 你可以为组件的 `props`、`data`、`methods` 等部分添加类型声明，以获得更好的类型检查。

3. 提供更多的类型定义：

- Vue 3 提供了丰富的类型定义，包括用于 `ref`、`reactive`、`computed`、`watch`、`provide`、`inject` 等功能的类型定义。

4. Composition API：

- Vue 3 的 Composition API 具有强大的 TypeScript 支持，可以更容易地编写可复用的逻辑。
- 使用 `defineComponent` 函数可以轻松定义类型安全的组件。

5. 类型安全的 Props：

- 在组件中，可以使用 `PropType` 来定义 props 的类型。
- 使用 TypeScript 的可选属性和默认值来确保 props 的类型安全。

6. 自动化类型推断：

- Vue 3 可以自动推断许多属性的类型，减少了手动添加类型声明的需要。

7. 类型安全的钩子函数：

- Vue 3 支持类型安全的生命周期钩子函数，如 `onMounted`、`onUpdated` 等。

8. TypeScript 装饰器支持：

- Vue 3 支持 TypeScript 装饰器，可以用于创建 mixin、自定义指令等。

9. 丰富的 TypeScript 文档：

- Vue 3 文档中提供了丰富的 TypeScript 示例和说明，方便开发者更好地了解如何在 Vue 3 中使用 TypeScript。

使用 TypeScript 的指南：

1. 安装 Vue 3：确保你的项目中安装了 Vue 3 和 TypeScript。
2. 创建组件：使用 `.vue` 文件或者 Composition API 来创建组件，可以添加类型声明来定义组件的 props 和数据。
3. 利用编辑器支持：使用支持 TypeScript 的编辑器（如 VS Code）来获得更好的类型检查和自动补全。
4. 遵循 Vue 3 文档：查阅 Vue 3 的官方文档，其中有关于如何使用 TypeScript 的详细说明和示例。

总的来说，Vue 3 提供了强大的 TypeScript 支持，使得在 Vue 3 项目中使用 TypeScript 变得更加容易和可靠。你可以利用这些功能来提高代码质量、可维护性和开发效率。

请解释 Vue 3 中如何创建自定义指令和自定义组件。

Vue 3 中新增了一些生命周期钩子函数，以扩展组件的生命周期管理和逻辑。以下是新增的生命周期钩子以及它们的用途：

1. **beforeMount**（新增）：
   - 用途：在组件挂载之前调用。在此阶段，虚拟 DOM 已经准备好，但尚未渲染到真实 DOM 中。可用于执行一些准备工作。
2. **beforeUpdate**（新增）：
   - 用途：在组件更新之前调用。在此阶段，虚拟 DOM 已经更新，但尚未渲染到真实 DOM 中。可用于执行更新前的准备工作。
3. **updated**（新增）：
   - 用途：在组件更新之后调用。在此阶段，组件的数据已经同步到视图中。可用于执行一些与更新后的 DOM 相关的操作。
4. **beforeUnmount**（新增）：
   - 用途：在组件卸载之前调用。在此阶段，组件仍然完全可用。可用于执行一些清理工作。
5. **unmounted**（新增）：
   - 用途：在组件卸载之后调用。在此阶段，组件的所有资源已被释放，不再可用。可用于执行一些最终的清理工作。

这些新增的生命周期钩子函数主要用于更细粒度的生命周期管理，允许你在组件不同生命周期阶段执行特定的操作。例如，你可以在 `beforeMount` 钩子中执行一些与渲染前准备相关的操作，或者在 `updated` 钩子中执行一些与更新后 DOM 操作相关的任务。

除了新增的生命周期钩子，Vue 3 仍然支持 Vue 2 中的其他生命周期钩子，如 `created`、`mounted`、`beforeDestroy` 和 `destroyed` 等。这些生命周期钩子允许你更灵活地管理组件的生命周期，以满足不同的需求。

什么时候使用状态管理器?

从项目的整体架构来看，要选择适合项目背景的极速。如果项目背景不适合使用状态管理器，那就没有一定的必要性去使用，比如微信小程序等，可以从以下几个维度来看

用户的使用方式复杂

- 不同身份的用户有不同的使用方式（比如普通用户和管理员）
- 多个用户之间可以协作
- 与服务器大量交互，或者使用了WebSocket
- View要从多个来源获取数据

从组件角度看

- 某个组件的状态，需要共享
- 某个状态需要在任何地方都可以拿到
- 一个组件需要改变全局状态
- 一个组件需要改变另一个组件的状态

什么渲染劫持?

什么是渲染劫持，渲染劫持的概念是控制组件从另一个组件输出的能力，当然这个概念一般和react中的高阶组件（HOC）放在一起解释比较有明了。

高阶组件可以在render函数中做非常多的操作，从而控制原组件的渲染输出，只要改变了原组件的渲染，我们都将它称之为一种渲染劫持。

实际上，在高阶组件中，组合渲染和条件渲染都是渲染劫持的一种，通过反向继承，不仅可以实现以上两点，还可以增强由原组件 render 函数产生的 React元素。

实际的操作中通过操作 state、props 都可以实现渲染劫持

怎么实现React组件的国际化呢?

依赖于 i18next 的方案，对于庞大的业务项目有个很蛋疼的问题，那就是 json 文件的维护。每次产品迭代都需要增加新的配置，那么这份配置由谁来维护，怎么维护，都会有很多问题，而且如果你的项目要支持几十个国家的语言，那么这几十份文件又怎么维护。

所以现在大厂比较常用的方案是，使用 AST，每次开发完新版本，通过 AST 去扫描所有的代码，找出代码中的中文，以中文为 key，调用智能翻译服务，去帮项目自动生成 json 文件。这样，再也不需要人为去维护 json 文件，一切都依赖工具进行自动化。目前已经有大厂开源，比如滴滴的 di18n，阿里的 kiwi

React如何进行代码拆分？拆分的原则是什么?

我认为 react 的拆分前提是代码目录设计规范，模块定义规范，代码设计规范，符合程序设计的一般原则，例如高内聚、低耦合等等。

在我们的react项目中：

- 在 api 层面我们单独封装，对外暴露 http 请求的结果。

- 数据层我们使用的 mobx 封装处理异步请求和业务逻辑处理。
- 试图层，尽量使用 mobx 层面的传递过来的数据，修改逻辑。
- 静态类型的资源单独放置
- 公共组件、高阶组件、插件单独放置
- 工具类文件单独放置

React中在哪捕获错误?

官网例子:

```
javascript复制代码class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染能够显示降级后的 UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // 你同样可以将错误日志上报给服务器
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // 你可以自定义降级后的 UI 并渲染
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

使用

```
xml复制代码<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

但是错误边界不会捕获:

```
csharp复制代码try{}catch(err){}
///异步代码（例如 setTimeout 或 requestAnimationFrame 回调函数）
///服务端渲染
///它自身抛出来的错误（并非它的子组件）
```

为什么说React中的props是只读的?

保证react的单向数据流的设计模式，使状态更可预测。如果允许自组件修改，那么一个父组件将状态传递给好几个子组件，这几个子组件随意修改，就完全不可预测，不知道在什么地方修改了状态，所以我们必须像纯函数一样保护 props 不被修改

怎样使用Hooks获取服务端数据?

```
javascript复制代码import React, { useState, useEffect } from 'react';
import axios from 'axios';
function App() {
  const [data, setData] = useState({ hits: [] });
  useEffect(async () => {
    const result = await axios(
      'https://api/url/to/data',
    );
    setData(result.data);
  });
  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}
export default App;
```

使用Hooks要遵守哪些原则?

1. 只在最顶层使用 Hook

不要在循环，条件或嵌套函数中调用 Hook， 确保总是在你的 React 函数的最顶层调用他们。

1. 只在 React 函数中调用 Hook

不要在普通的 JavaScript 函数中调用 Hook。你可以：

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

React Fiber它的目的是解决什么问题?

React15 的 `StackReconciler` 方案由于递归不可中断问题，如果 Diff 时间过长（JS计算时间），会造成页面 UI 的无响应（比如输入框）的表现，`vdom` 无法应用到 `dom` 中。

为了解决这个问题，React16 实现了新的基于 `requestIdleCallback` 的调度器（因为 `requestIdleCallback` 兼容性和稳定性问题，自己实现了 `polyfill`），通过任务优先级的思想，在高优先级任务进入的时候，中断 `reconciler`。

为了适配这种新的调度器，推出了 `FiberReconciler`，将原来的树形结构（vdom）转换成 Fiber 链表的形式（child/sibling/return），整个 Fiber 的遍历是基于循环而非递归，可以随时中断。

更加核心的是，基于 Fiber 的链表结构，对于后续（React 17 lane 架构）的异步渲染和 （可能存在的）worker 计算都有非常好的应用基础

说出几点你认为的React最佳实践

参考官网

React为什么要搞一个Hooks?

官网回答：

动机

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

在组件之间复用状态逻辑很难

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）。如果你使用过 React 一段时间，你也许会熟悉一些解决此类问题的方案，比如 render props 和 高阶组件。但是这类方案需要重新组织你的组件结构，这可能会很麻烦，使你的代码难以理解。如果你在 React DevTools 中观察过 React 应用，你会发现由 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”。尽管我们可以在 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。

你可以使用 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 使你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。

复杂组件变得难以理解

我们经常维护一些组件，组件起初很简单，但是逐渐会被状态逻辑和副作用充斥。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。

难以理解的 class

除了代码复用和代码管理会遇到困难外，我们还发现 class 是学习 React 的一大屏障。你必须去理解 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。

另外，React 已经发布五年了，我们希望它能在下一个五年也与时俱进。就像 Svelte，Angular，Glimmer等其它的库展示的那样，组件预编译会带来巨大的潜力。尤其是在它不局限于模板的时候。最近，我们一直在使用 Prepack 来试验 component folding，也取得了初步成效。但是我们发现使用 class 组件会无意中鼓励开发者使用一些让优化措施无效的方案。class 也给目前的工具带来了一些问题。例如，class 不能很好的压缩，并且会使热重载出现不稳定的情况。因此，我们想提供一个使代码更易于优化的 API。

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

状态管理解决了什么问题?

专注 view 层

React 官网是这么简介的。JavaScript library for building user interfaces.专注 view 层 的特点决定了它不是一个全能框架，相比 angular 这种全能框架，React 功能较简单，单一。比如说没有前端路由，没有状态管理，没有一站式开发文档等。

f(state) = view

react 组件是根据 state （或者 props）去渲染页面的，类似于一个函数，输入 state，输出 view。不过这不是完整意义上的 MDV（Model Driven View），没有完备的 model 层。顺便提一句，感觉现在的组件化和 MDV 在前端开发中正火热，大势所趋...

state 自上而下流向、Props 只读

从我们最开始写 React 开始，就了解这条特点了。state 流向是自组件从外到内，从上到下的，而且传递下来的 props 是只读的，如果你想更改 props，只能上层组件传下一个包装好的 setState 方法。不像 angular 有 ng-model, vue 有 v-model， 提供了双向绑定的指令。React 中的约定就是这样，你可能觉得这很繁琐，不过 state 的流向却更清晰了，单向数据流在大型 spa 总是要讨好一些的。

这些特点决定了，React 本身是没有提供强大的状态管理功能的，原生大概是三种方式。

函数式组件有没有生命周期?

它没有提供生命周期概念，不像 class 组件继承 React.component，可以让你使用生命周期以及特意强调相关概念

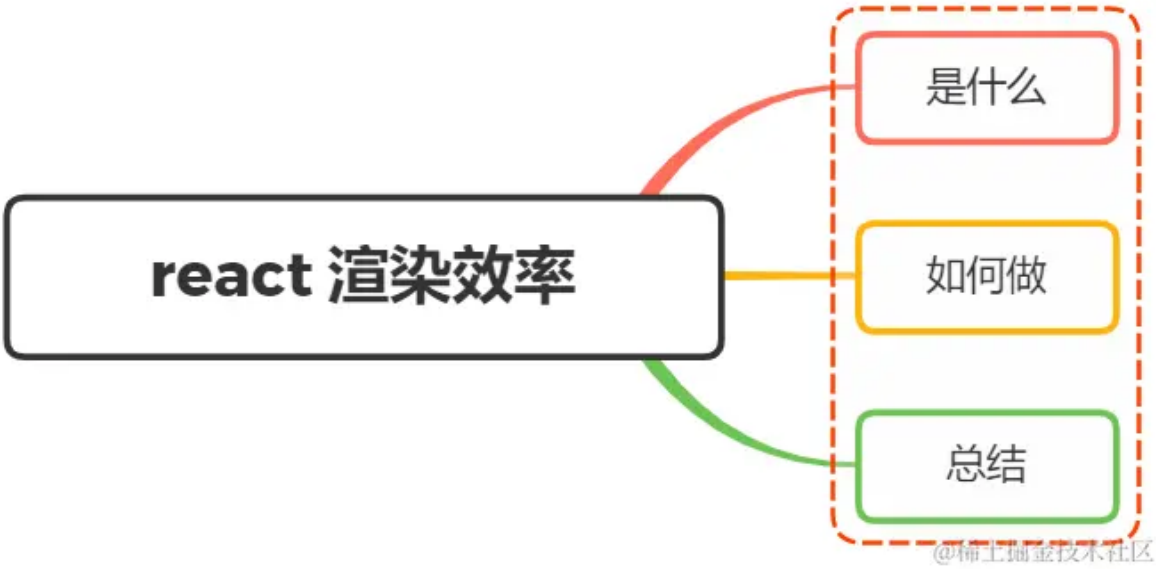
immutable的原理是什么?

使用字典树持久化数据结构，更新时可优化对象生成逻辑，降低成本

怎么防止HTML被转义?

```
dangerouslySetInnerHTML
```

说说你是如何提高组件的渲染效率的



是什么

react 基于虚拟 DOM 和高效 Diff算法的完美配合，实现了对 DOM最小粒度的更新，大多数情况下，React对 DOM的渲染效率足以我们的业务日常

复杂业务场景下，性能问题依然会困扰我们。此时需要采取一些措施来提升运行性能，避免不必要的渲染则是业务中常见的优化手段之一

如何做

类组件：

- 继承PureComponent
- 使用shouldComponentUpdate优化

函数组件：

- memo模拟PureComponent
- 使用useMemo缓存变量
- 使用useCallback缓存函数
- 循环添加key, key最好用数组项的唯一值，不推荐用 index

总结

在实际开发过程中，前端性能问题是一个必须考虑的问题，随着业务的复杂，遇到性能问题的概率也在增高

除此之外，建议将页面进行更小的颗粒化，如果一个过大，当状态发生修改的时候，就会导致整个大组件的渲染，而对组件进行拆分后，粒度变小了，也能够减少子组件不必要的渲染

说说对高阶组件（HOC）的理解?

高阶函数（Higher-order function），至少满足下列一个条件的函数

- 接受一个或多个函数作为输入
- 输出一个函数

在React中，高阶组件即接受一个或多个组件作为参数并且返回一个组件，本质也就是一个函数，并不是一个组件

```ini复制代码const EnhancedComponent = highOrderComponent(WrappedComponent);
```

上述代码中，该函数接受一个组件 `WrappedComponent` 作为参数，返回加工过的新组件 `EnhancedComponent`

高阶组件的这种实现方式，本质上是一个装饰者设计模式

说说对React refs 的理解?

Refs 在计算机中称为弹性文件系统（英语：Resilient File System，简称ReFS）

React 中的 Refs提供了一种方式，允许我们访问 DOM节点或在 render方法中创建的 React元素

本质为ReactDOM.render()返回的组件实例，如果是渲染组件则返回的是组件实例，如果渲染dom则返回的是具体的dom节点

class

```scala复制代码class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref="myref" />;
  }
}
```

hooks

```javascript复制代码function App(props) {
  const myref = useRef()
  return (
    <>
      <div ref={myref}></div>
    </>
  )
}
```

背景

自我介绍下，四年工作经验，头两年全栈开发，后两年专职做前端，目前已达到高级前端工程师水平，经历过三家公司。第一家公司，电商行业，做阿里 `ISV` 供应商，为淘宝卖家服务，也是我第一次接触百万 `UV` 级别的产品，在第一家公司呆了两年，由于达到技术瓶颈期，遂跳槽，第二家公司，航运物流行业，呆了六个月（工作强度对我来说，是真的高），身体不适，没有同意转正。目前这家，担任项目管理和前端组长，两个角色，目前呆了两年，做了很多东西，把自己的一些想法跟大家聊一聊。

入职时的环境

这是一家做保险和金融行业的公司，属于传统行业的科技公司，有点外包的性质，当然，也有自己的 `SaaS` 产品，由于是传统行业的公司，技术栈相对互联网公司来说，稍微落后一点。我刚来的时候，上一个前端要辞职了，然后做对接工作（告诉我，有啥问题，直接搜代码），我算是接盘侠，前任留下的屎山，其他的，大概有以下几点：

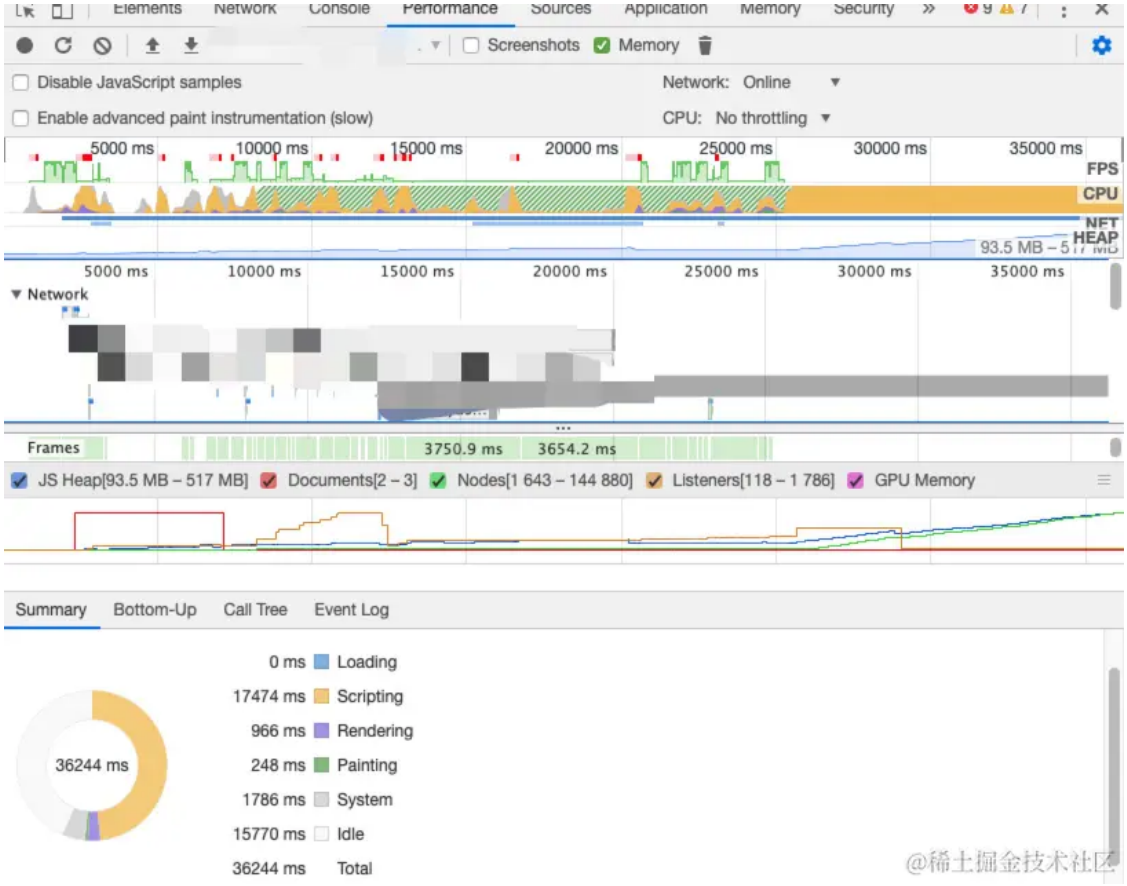
前端组 4 个人

其中一个归 CTO（做后端） 管，另外两个在广东，我入职的时候，也没有确认，到底要不要带人。我来的时候，就已经在了，后面我领导跟我说，要带下他们，我当时压根就没有带人的想法，也是个坑。

历史项目有很多个，都是基于一套从 GitHub 上弄过来的项目框架

- 没有前端工程化体系，开发周期长，开发质量差，维护困难
- 前后端混合项目，剥离前端代码没有剥离干净，后端很多文件都在，不知道重不重要，前端代码运行在服务端，每次修改一行代码，看效果，需要拖到服务器上进行编译，编译大概 1-2 分钟左右，非常痛苦。
- 完全熟悉该项目的人员已离职（技术和产品），项目交接没有处理好。
- 业务逻辑非常混乱，没有相关的产品流程图，全凭记忆。
- 服务器上运行的 `Node` 版本非常低，到现在还是 `8.x`，各种低版本的库都在，比如 `Ant Design` 用的 `3.6.2`，在项目开发中遇到穿梭框无法进行树状显示(代码一摸一样，在高版本 `3.19.2`，可以显示)。又比如还有这种 `"translate.js": "git+https://github.com/MichelSimonot/translate.js.git”`
- 尝试过升级库和卸载其他库，各种报错。
- 代码缺乏注释，一个文件几千行，对 `React`， `Redux` 使用，欠缺理解。
- 有过一次”爆炸”，此项目如果再继续迭代下去，随时可能继续”爆炸”，现在已经是在踩雷开发阶段。

  在 2019 年 10 月 18 号，24：00 发生生产事故，事故表现为，操作特定页面，浏览器崩溃，卡死。

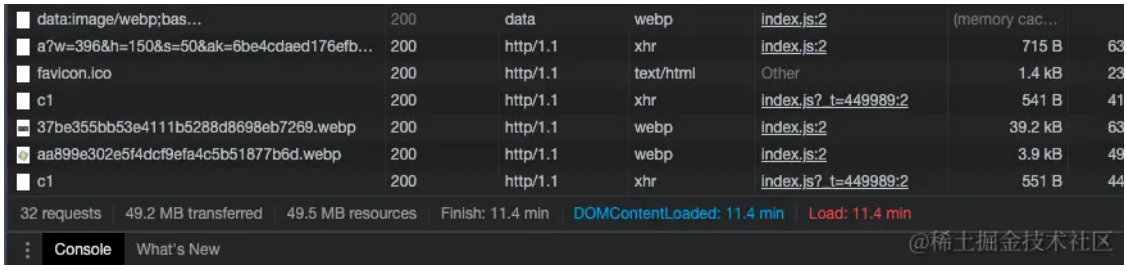


脚本执行时间非常长，后面经查，是由以下代码引起

```
php复制代码actions.getAgentListByPage({
    companyId: currentAgent.companyId,
    pageIndex: 1,
    pageSize: 20000,
    searchProvince: currentAgent.province,
    searchCity: currentAgent.city
})
```

页面很多地方存在请求 ***pageSize：20000*** 的情况，该代码是由前任前端编写，具体为何写出这样的代码，原因未知，处理方案给到后端解决，前端配合加入 `workbench` 字段，凌晨 1 点左右得到解决。

- 一套项目上，运行着两套系统。
- 打包出来的项目代码体积有 **49.5MB**，页面首次加载耗时 **11.4 min**



基于以上的原因，向领导提出过重构，没有同意，我认为可能有两个方面的顾虑，

- 从人力资源条件来讲，并不允许。
- 从公司战略角度来讲，能挣钱的项目就是好项目。但是，这并不影响我建设前端工程化体系。

项目人员能力较弱

- 测试同学报备 BUG，没有记录可复现步骤。
- 任务管理工具平台没有真正利用起来，相关项目需求，BUG 没有整理起来放在上面。
- 产品不理解大概的技术实现，没有把产品文档梳理，留存下来，不理解客户的真正需求，以至于技术实现比较鸡肋。

前后端接口对接，没有相关的文档

产品画的原形 和 UI 设计稿不规范

列举了以上的这些点，烂摊子太多了，好在有一个点，领导的支持力度还不错，看我是如何突围的。

明确自己的任务

前端技术建设的核心目的，是为了提高开发效率，保证开发质量，为保障项目高质量按时交付，同时兼顾考虑中长期研发实际情况，结合团队实际能力，为未来做技术储备，为业务发展提供更多的可能性，大概将自己的分为以下四类

- **基础架构设计**，主要目的是从架构层面出发，通过流程化设计，规避常见问题，提高开发效率。
- **工程化设计**，与代码强相关，主要目的是提高代码质量，增强代码的长期可维护性，降低开发时间和成本。
- **团队管理**，通过合理有效的团队管理，提高团队人效比，为未来项目研发、技术发展，进行人才储备、技术研发。
- **项目管理**，进行合理的项目管理，合适的工时排期和迭代计划，提高项目交付质量和效率。

如何解决

首先，要对现有的问题进行梳理归纳，按照问题的优先级进行排序，然后，分阶段性目标进行实现，对于上面的问题，我大概整理了一张表格

问题	优先级	成本	目标
如何打造前端工程化体系	p0	高	提升整个前端团队的开发效率、按时交付、保证交付质量。
如何进行团队管理	p0	中	进行人才储备，提高团队人员技术能力
如何进行项目管理	p1	中	掌控全局，知道项目下的人都在做什么，资源协调

团队管理

人员管理

- **初来乍到**，首先就是跟大家一起聊聊天，了解他/她的想法，以及个人情况、技术能力、兴趣爱好、性格特点等。
- **团建聚餐**，经常请大家喝奶茶/咖啡，不定时的组织活动，通常是聚餐（个人出钱），为下面的工作，好开展。
- **导师帮带**，新人进来后，安排一个人带着他，答复常见问题，由简单的需求再到核心模块的负责，一点一点施展压力。
- **新人适应**，负责安排新成员的发展方向，并在新人入职的前几周，了解项目框架和开发模式，再安排其做基于现有页面的优化，帮助其了解不同人负责的业务。

- **责任划分**，明确团队里人员定位，并使其知晓，根据成员能力不同，态度不同，安排适合其的任务。
- **前端周会**，每周一次，组织大家开前端周会，在这个会上，过下大家目前手头上的事情，有没有遇到什么问题，需要协调的一些资源，进度把控等。
- **技术分享**，不定时的前端技术分享，主题不限，并把相关分享后的资料，上传到前端文档管理，方便日后的人员进行查看。

权限管理

主要是指代码权限控制，目的是确保代码安全，问题可控可避免可追溯

具体管理举措有以下几条：

- **公司仓库**，代码属于公司财产，对代码进行权限隔离，启用内网 `GitLab`，默认关闭所有外网访问权限，针对每个项目，按实际需要给开发赋予指定权限。
- **提交权限**，允许开发在自己仓库下提交，但涉及到公司仓库的合并，需要发起 `PR`，然后在组长进行 `CR` 后，才能提交到主仓库。
- **发布权限**，对于将要发布到生产环境，权限给到组长，只允许组长进行发布。

前后端接口对接

前后端开发联调有一个严重问题，就是后端接口变动或者字段改动时，没有在事前事后通知相应前端开发，测试人员，导致效率底下，并且会出现各种异常情况。

因此，通过梳理开发流程，出接口文档，作为对接标准。

我们使用 `apiDoc` 来作为前后端联调标准。

File – 上传文件接口 1.0.0

上传文件接口

POST

```
/api/file
```

Request body

字段	类型	描述
agentId	String	账号id
loginName	String	账号登录名
fullName	String	账号姓名
orgAccount	String	机构id
orgAccountName	String	机构名称
tenant	String	租户id
tenantName	String	租户名称
tenantBucket	String	租户桶名
tenantCloudType	String	租户云存储名,AWS/COS
isPublic	String	是否个人文件,个人文件: 0, 公共文件: 1

Request file

字段	类型	描述
file	File	上传的文件

但在实际情况中，还是会有一些接口文档和实际接口不符的情况发生，导致一些问题产生，这个我们也在思考。

前端工程化体系

刚入职的时候，由于上面的项目框架问题太多，之前也尝试过解决，但，解决不了，领导也意识到了这点，而且也有新项目进来，就让我重新搞一套项目框架。所以，我自研了一套基于 `Webpack` 的项目框架和工程化体系，做这件事的目的，就如我上面提到过的一样，提升整个前端团队的开发效率、按时交付、保证交付质量。

基础架构设计

Git 分支管理规范化

我们使用的是 `Git Flow` 分支管理策略

`Git Flow` 最开始是由 `Vincent Driessen` 发行并广受欢迎，这个模型是在 2010 年构思出来的，而现在距今已有 10 多年了，而 `Git` 本身才诞生不久。在过去的十年中，`Git Flow` 在许多软件团队中非常流行

分支命名规范

- master 分支：master 分支只有一个，名称即为 master。GitHub 现在叫 main
- develop 分支：develop 分支只有一个，名称即为 develop
- feature 分支：feature/<功能名>，例如：feature/login，以便其他人可以看到你的工作
- hotfix 分支：hotfix/日期，例如：hotfix/0104

分支说明

- master || main 分支：存储正式发布的产品，`master || main` 分支上的产品要求随时处于可部署状态。`master || main` 分支只能通过与其他分支合并来更新内容，禁止直接在 `master || main` 分支进行修改。
- develop 分支：汇总开发者完成的工作成果，`develop` 分支上的产品可以是缺失功能模块的半成品，但是已有的功能模块不能是半成品。`develop` 分支只能通过与其他分支合并来更新内容，禁止直接在 `develop` 分支进行修改。
- feature 分支：当要开发新功能时，从 master 分支创建一个新的 `feature` 分支，并在 `feature` 分支上进行开发。开发完成后，需要将该 `feature` 分支合并到 `develop` 分支，最后删除该 `feature` 分支。
- release 分支：当 `develop` 分支上的项目准备发布时，从 `develop` 分支上创建一个新的 `release` 分支，新建的 `release` 分支只能进行质量测试、bug 修复、文档生成等面向发布的任务，不能再添加功能。这一系列发布任务完成后，需要将 `release` 分支合并到 `master` 分支上，并根据版本号为 `master` 分支添加 `tag`，然后将 `release` 分支创建以来的修改合并回 `develop` 分支，最后删除 `release` 分支。
- hotfix 分支：当 `master` 分支中的产品出现需要立即修复的 bug 时，从 `master` 分支上创建一个新的 `hotfix` 分支，并在 `hotfix` 分支上进行 BUG 修复。修复完成后，需要将 `hotfix` 分支合并到 `master` 分支和 `develop` 分支，并为 `master` 分支添加新的版本号 `tag`，最后删除 `hotfix` 分支。

提交信息规范

提交信息应该描述“做了什么”和“这么做的原因”，必要时还可以加上“造成的影响”，主要由 **3** 个部分组成：`Header`、`Body` 和 `Footer`。

Header Header 部分只有 1 行，格式为`():`。

type 用于说明提交的类型，共有 8 个候选值：

1. feat：新功能（feature）
2. fix：问题修复
3. docs：文档
4. style：调整格式（不影响代码运行）

5. refactor：重构
6. test：增加测试
7. chore：构建过程或辅助工具的变动
8. revert：撤销以前的提交
9. scope 用于说明提交的影响范围，内容根据具体项目而定。

subject 用于概括提交内容。

Body 省略

Footer 省略

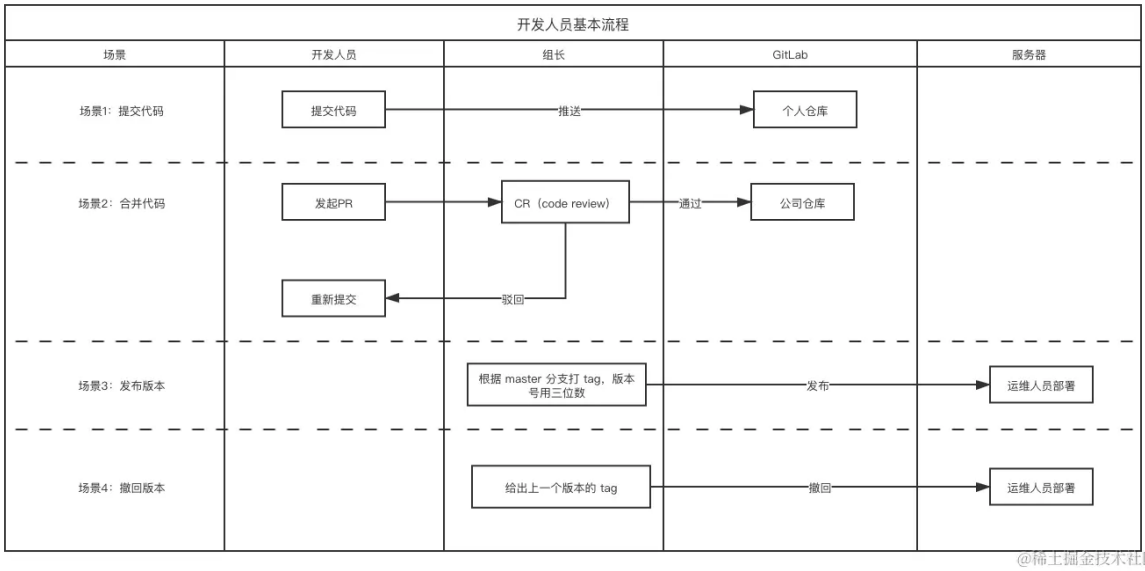
.vscode	Update settings.json	6 months ago
src	chore: 注释登录权限自动跳转	11 days ago
.babelrc	feat: import vite replace webpack	19 days ago
.eslintrc.js	feat: import vite replace webpack	19 days ago
.gitignore	init	7 months ago
LICENSE	chore: update readme.md	7 months ago
README.md	chore: update README.md	11 days ago
declaration.d.ts	chore: format file	3 months ago
index.html	feat: import vite replace webpack	19 days ago
package.json	feat: import vite replace webpack	19 days ago
prettier.config.js	Update prettier.config.js	6 months ago
stylelint.config.js	fix: fix lint-staged invalid	3 months ago
tsconfig.json	feat: import vite replace webpack	19 days ago
vite.config.ts	feat: import vite replace webpack	19 days ago

这样做起来的好处，这个项目下：

- 对于分支，每个人在做什么，我看分支就清楚。
- 对于修改内容，看前缀就知道这个文件改动了什么。
- 对于版本迭代，看 `Tag` 都上线了什么内容。

总之，一目了然。

开发人员基本流程



在这个流程中，开发人员只对个人仓库拥有可控权，无法直接改变公司仓库代码，当需要提交到公司仓库下时，需要发起 `PR` 请求，经过组长 `CR` 后，将其代码合并到公司仓库下。

主分支代码和线上代码进行隔离，由组长将指定版本的 `Tag` 发布到生产环境，再通过运营人员直接从 `GitLab` 上拉取指定的 `Tag`，然后打包发布。

通过以上流程，前端代码能保证高质量，高稳定性的状态，运行在服务器端。

工程化设计

要根据实际业务情况和团队规模，技术水平来做，关键是要形成一个闭环，所谓闭环就是从零开始到上线再到迭代的全链路，有很多节点，这些节点需要根据实际情况进行设计，避免过度设计。

定制 Webpack 项目框架

为何不是 create-react-app

create-react-app 是基于 `webpack` 的打包层方案，包含 `build、dev、lint` 等，他在打包层把体验做到了极致，但是不包含路由，不是框架，也不支持配置。所以，如果大家想基于他修改部分配置，或者希望在打包层之外也做技术收敛时，就会遇到困难。

为何不是 umi

umi 提供的功能很多，这也导致它太过于臃肿。而且你还要去学它的封装化配置，而不是学原生第三方库的配置，如果你只想要一些简单的功能，追求更高的可玩性，哪 umi 不太适合。

所以，我自己定制了一套脚手架，实现了以下功能：

- **快速上手**，只要了解 React、Mobx、Webpack 和 React Router，就可以快速搭建中后台管理平台
- **路由系统**，基于 react-router 实现的路由系统
- **Loading**，不需要重复写组件 Loading 判断
- **国际化**，基于 react-intl-universal 实现的国际化
- **网络请求**，基于 axios 实现的请求拦截
- **页面交互**，基于 mobx 实现的数据交互方式
- **UI**，使用业界最著名的 ant-design
- **代码规范校验**，使用 eslint、pre-commit、lint-staged、prettier、stylelint
- **模拟请求数据**，基于 mockjs 实现
- **打包工具**，目前最流行的 Webpack

解决了以下的问题：

- 约束开发人员代码规范
- 方便提供给其他开发使用标准的脚手架，并提供技术支持

完成整个编码过程的一个闭环：

- 编码前：编码规范，最佳实践
- 编码中：自研项目框架、代码校验
- 编码后：发布部署工具 JenKins，手动发布或 CI/CD

这些节点要视实际情况，以最小成本去做，然后逐步升级。比如编码规范，我们是采用业界比较著名的 `Airbnb JavaScript` 代码规范，搭配 `eslint、pre-commit、lint-staged、prettier、stylelint` 去进行约束。

这套项目框架，目前开发体验非常爽，在我司多个产品线上，投入使用，并已开源，**框架地址**，演示页面比较少，大佬们觉得不错的话，可以给个 Star 🌟，也欢迎给项目提 issues ~

业务场景

我们是做 `ToB` 业务，存在页面上大量使用表单的场景，所以，把我们的表单页面做成可配置化，实现了大部分页面表单配置化，减少前端人力资源投入。

针对公司的实际业务场景，其他子系统不会特别复杂，页面也不会多，共享一套账号体系，这里采用的思路是只有一个项目，不分主从系统，通过 `Webpack` 配置多页面，不同的子系统进入的首页内容不一样，加载内容不一样，菜单导航，则通过后端对每个租户进行区分，来做到租户看到的菜单系统不一样。

如果子系统特别复杂，有主从系统概念，可以考虑使用微服务设计，这里不做过多介绍。

静态资源

除了业务代码以外，前端还会有一些公共静态资源，例如 `React` 资源，`Ant Design` 资源，`BizCharts` 资源，以及一些图片文件等。

对于这些文件，是所有项目所共享的，假如这些文件分散在各个项目里，既没必要，也容易导致不同项目依赖文件不统一。

我们是放在 `S3` 上，做 `CDN` 静态资源加速，然后前端项目通过引入 `url` 来使用这些资源，这样可以减轻自己的服务器网络带宽消耗。

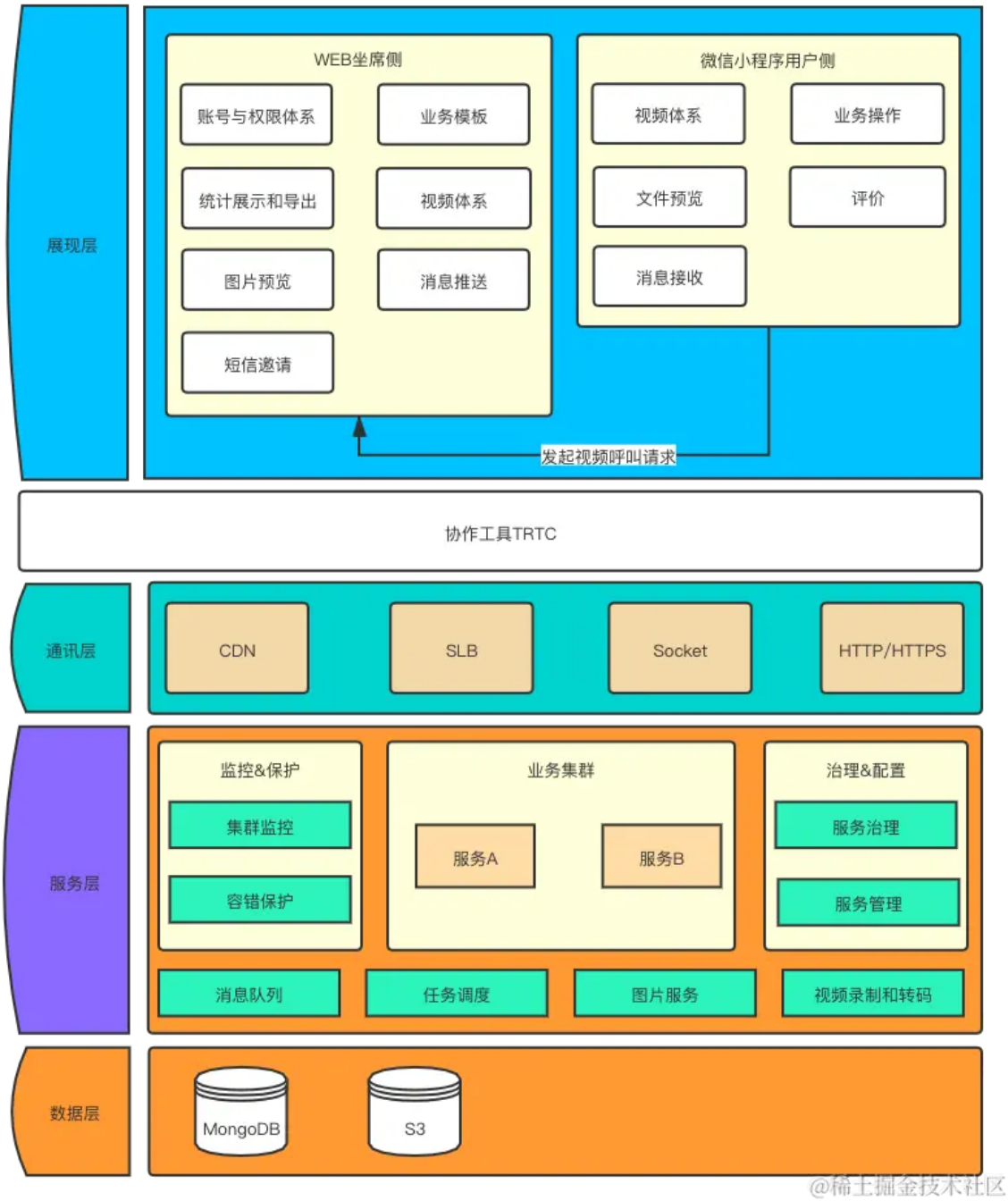
项目管理

- **任务分配**，产品把相关的需求，经过讨论，可行性分析，通过项目管理工具，放到迭代计划中，录入开发工时，测试工时。
- **文档管理**，采用项目管理工具自带的文档，要求做到文档可以团队编辑，可以查看到编辑历史。
- **项目周会**，过大家手上目前的迭代进度，遇到的问题，需要协调的资源，风险控制等。
- **项目复盘**，复盘首先是要做的是事实陈述，开始诊断、分析存在差异的原因，找出导致成功或失败的根本原因后进行规律总结。明白为什么会成功、哪些关键行为起了作用，这些行为有没有适用条件，对于提高后续行动的成功率有没有价值。

熟悉产品线业务

所谓技术服务业务，找产品了解现有的业务流程以及痛点，甚至未来要做的一些产品规划，好的技术架构，要考虑各种各样的业务场景，怎样才能结合业务的复杂度，设计出颗粒度更加细化的组件。

画出产品架构图



提升相关人员的能力

产品人员

需求频繁且混乱，决策摇摆不定、动辄推倒重来。市面上一个好的产品经理是很贵的，没个三四万是拿不下一个真正靠谱的能抗住复杂产品线的产品经理，但是很多公司老板是不愿意花这个钱，一般就会招个工作一两年的产品经理先过来，顶个位置把这个工具给做出来就行了。恰恰因为这样一个认知导致产品经理这一层他既没话语权，又不能让自己闲着，所以层出不穷的需求全堆上来了，而对于公司长久型的产品架构就把控不住，如果一个产品经理无法起到，上对客户负责，下对开发负责，不会对所有需求进行筛选，把需求只会丢给开发，不会进行工时把控和质量把控。甚至对现有产品有什么功能，都不了解，那么就不是一个合格的产品。

所以对产品经理的要求非常严格，因为一个公司，如果战略方向把握住了，那么核心是要看产品，能否把握住市场方向，非常关键。这样才能决定你是否能占有市场，由于我司是做一个 `ToB SaaS` 化的平台，所以，必须要求产品经理清楚的了解客户实际需求，需求背后的实际场景，提炼出来哪些是共性的需求，哪些是客户定制化的需求，然后再讨论，再进行落地实际的开发。

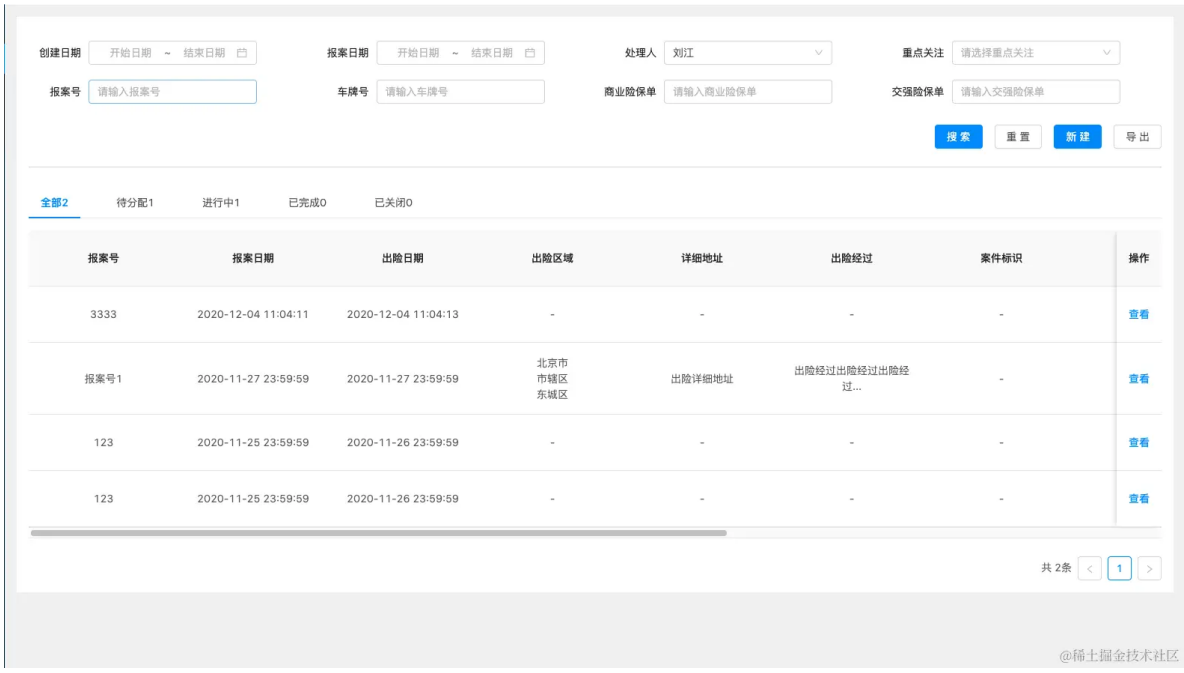
测试人员

对测试人员，尽量覆盖全所有场景，保证核心流程畅通，要求能找到复现步骤，提高开发解决 BUG 的效率。

设计规范

由于我司采用的是 `Ant Design` UI 库，所以设计标准，尽量都是按照 `Ant Design` 现成组件和样式来做，避免开发二次修改，参考这个链接 Ant Design 设计原则

某个列表页



普通的列表，和设计，产品都约定好，上面是筛选，下面是按钮，底部是表格展示。

某个详情页

详情页大量会使用到表单，所以直接使用 `Ant Design` 的 `From` 表单组件。

表单每行放多少个，都是以 `Ant Design` 组件来的。

这样带来的好处就是尽量避免定制化的开发，所有列表和详情都是按照这种风格来进行开发。

执行成果

开发效率

组内人员开发效率，较之前提升了一倍左右，普通的列表页面（搜索、展示、弹窗），包含接口联调 + 自测，大概 1 天左右完成，详情页面，复杂一点的表单交互，表单组件联动，大概在 2 天左右完成，包含接口联调 + 自测，目前我们也在探索 `Vite`，`Snowpack`，极大的提升开发体验。

系统情况

`SaaS` 系统，首次无缓存加载耗时 **3.22s**，三个系统（30 多个页面，14 个公用组件）打包出来的体积在 **9.3MB**

?sentry_key=28d1091a7e144f5eb...	200	h2	fetch	vendors.js:2	193 B	200
pingd?dm=service-support-test.ik...	200	http/1.1	text/html	vendors.js:2	239 B	42
7dfac08d683b81d4755c4f28ae93...	200	http/1.1	jpeg	1.9003ed1.css	402 kB	275
?sentry_key=28d1091a7e144f5eb...	200	h2	fetch	vendors.js:2	278 B	91

25 requests | 9.3 MB transferred | 9.3 MB resources | Finish: 3.22 s | DOMContentLoaded: 2.17 s | Load: 2.43 s

当然还有优化空间

设计规范

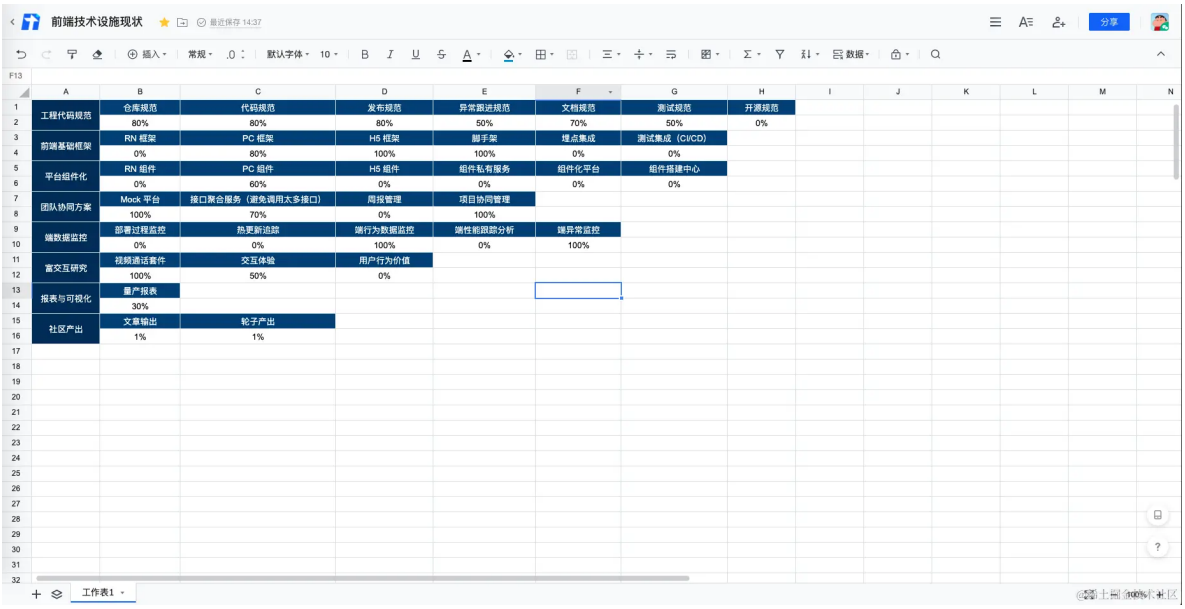
目前大部分页面不需要设计资源投入，尽量按照 `Ant Design` 设计标准和我们自定的 UI 标准风格来做，减少设计人员的工作投入。

项目文档

目前所有的产品文档，技术文档都非常规范，可以溯源，以及当时在什么样的场景下，为什么要做出这样的解决方案。

总结

上面这些，包含其他的，大概花了一年多的时间，建设完成，我们目前的基建状况如下表所示



如果你觉得对你有帮助或启发，欢迎点赞留言。