2024.04.15 更新前端面试问题总结（20 道题）

中级开发者相关问题【共计 11 道题】

681.CSS 中的 display 属性有哪些值？【热度: 593】【CSS】【出题公司: TOP100 互联网】

CSS 中的 `display` 属性是一个非常重要的属性，它用于设置一个元素的显示类型。这个属性决定了元素是如何显示以及与其他元素如何交互。以下是一些常见的 `display` 属性值及其含义：

1. `none`：元素不会被显示。
2. `block`：元素显示为块级元素，此类元素会新起一行。
3. `inline`：元素不会新起一行，其宽度只占据它的内容宽度。
4. `inline-block`：元素横排显示，但是同时具备块级元素的特性，比如可以设置宽高。
5. `flex`：元素会变成弹性容器（flex container），其子元素会成为弹性项（flex items）。这个值允许使用弹性盒子布局（flexbox）。
6. `grid`：元素会变成网格容器，其子元素会成为网格项。它开启了网格布局。
7. `table`、`table-row`、`table-cell` 等：这些值让元素表现得像表格元素一样。
8. `list-item`：元素会表现为列表项（像 `<li>` 元素一样）。

另外还有一些新的、较少使用或是实验性的 `display` 属性值，例如：

- `inline-flex`: 使元素的内容为弹性容器，与 `flex` 相似，但是元素自身会像 `inline` 元素一样排列。
- `inline-grid`: 类似于 `grid`, 但是元素自身表现为 `inline` 级别。

CSS3 引入了更多复杂的布局模式，包括上述的 `flex` 和 `grid` 以及其他的一些属性值。根据您使用的 CSS 版本，可能还有更多其他属性值存在。这些布局模式为网页布局提供了更为强大和灵活的控制手段。

682.CSS 属性值计算 - calc【热度: 320】【CSS】【出题公司: TOP100 互联网】

是的，CSS 支持计算值，这可以通过 `calc()` 函数实现。`calc()` 允许你进行数学运算，计算 CSS 属性值。这个功能非常有力，因为它可以混合使用不同的单位，并且可以用在几乎任何需要数值的地方。

以下是 `calc()` 函数的一些应用示例：

1. **基本运算**：可以执行加 (`+`)、减 (`-`)、乘 (`*`) 和除 (`/`) 四种基本运算。

```
.element {
  width: calc(100% - 80px);
}
```

混合单位：`calc()` 函数可以混合使用像素、百分比、em、rem 等单位。

```
.element {
  margin-top: calc(50vh - 50px);
}
```

嵌套：你可以在 `calc()` 里面嵌套另一个 `calc()` 。

```
.element {
  width: calc(100% - calc(50px + 2em));
}
```

环境变量：结合 CSS 变量 (Custom Properties) 使用。

```
:root {
  --main-padding: 10px;
}

.element {
  padding: calc(var(--main-padding) * 2);
}
```

动态值调整：用于某些动态大小的调整。

```
.element {
  position: absolute;
  bottom: calc(50% - 20px);
}
```

当使用 `calc()` 时有一些规则需要注意，例如:

- 运算符之间需要有空格。`calc(50% -50px)` 是无效的，而 `calc(50% - 50px)` 是有效的。
- 不能进行 0 除运算，也就是说分母不能为 0。
- 在进行乘法和除法时，至少有一个值必须是数值（即不带单元的数字）。

总的来说，`calc()` 是一个强大的 CSS 工具，可以在响应式设计和复杂布局管理中提供极大的帮助。

683.void 和 never 区别【热度: 410】【TypeScript】【出题公司: 阿里巴巴】

关键词：typescript void、typescript never、void 和 never 区别

在 TypeScript 中，`void` 和 `never` 类型都代表没有值，但用途和含义却有所不同。以下是它们之间的主要区别：

1. **void**：
2. `void` 类型用于标记函数没有任何返回值。这意味着函数可能执行了一些操作但是没有返回任何内容。这不同于返回 `undefined` 或 `null`，尽管在没有明确返回值时，JavaScript 函数默认返回 `undefined`。
3. 如果一个函数的返回类型是 `void`，它可能有一个 `return` 语句，但 `return` 语句不能返回任何值（或者根本就没有 `return` 语句）。
4. 例子：

```
function greet(): void {
  console.log("Hello, World!");
}
```

1. 这个函数打印一个字符串到控制台，但不返回任何值。
2. **never**：
3. `never` 类型表示永远不会返回任何值。它通常用于两种情况：函数总是抛出一个错误，这样就不会有返回值；或者函数有一个无法达到的终点，比如无限循环。
4. 例子：

```
function throwError(errorMsg: string): never {
  throw new Error(errorMsg);
}

function infiniteLoop(): never {
```

```
6   while (true) {}
7 }
8
```

这两个函数都不会正常结束：`throwError` 函数会抛出异常，而 `infiniteLoop` 函数会永远循环。在这两种情况中，返回类型 `never` 正确地表明函数不会有任何返回执行路径。

总结来说，`void` 用于没有返回值的函数，这意味着函数的执行结束后不会给调用者任何值；而 `never` 表示函数永远不会有一个正常的结束，因此不会给调用者任何机会获得它的返回值。它们在类型系统中表达了不同的概念和意图。

687.JS 严格模式作用是啥？【热度: 530】【JavaScript】【出题公司: 腾讯】

JavaScript 的严格模式（"use strict"）是一种限制 JavaScript 语法的方式，目的是为了捕捉一些常见的编码错误，同时阻止或者抛出错误对于一些保险不良行为的使用。它通过在脚本或函数的顶部添加一个特殊的声明来启用，声明如下：

```
1 "use strict";
```

以下是严格模式的一些主要作用：

1. **消除 JavaScript 语法的一些静默错误**：在非严格模式下，这些错误不会抛出异常，可能导致开发者难以发现。启用严格模式后，这些错误会被抛出。
2. **修复可能使 JavaScript 引擎难以优化代码的一些错误**：有些代码模式可能是引擎优化的障碍。严格模式下这些模式会被禁止，有时候能提升代码的性能。
3. **禁止使用某些语法**：这些语法在未来的 ECMAScript 版本中可能会有其他的用途。比如，未来的 ECMAScript 版本可能会使用 `implements`，`interface`，`let`，`package`，`private`，`protected`，`public`，`static`，和 `yield` 作为关键词。
4. **为未来新版 JS 做准备**：有些旧语法或行为在严格模式下已被废弃，确保代码兼容未来语言的演变。

以下是“严格模式”相比于“非严格模式”的一些具体区别：

- 在严格模式下，试图给未声明的变量赋值会抛出错误。
- 删除不可删除的属性会抛错（如 `delete Object.prototype;`）。
- 要求函数的参数名唯一。
- 禁用 `with` 语句。
- `eval` 在严格模式下有自己的作用域，且不会引入新的变量。

- 禁止 `this` 关键字指向全局对象。
- 不能使用八进制语法（例如，`042` 这样的八进制数字会抛错）。
- 不能给不可写属性赋值，也不能给只读属性赋值（getter-only）。
- 禁止在非构造函数中使用 `delete` 删除变量（否则会被忽略而不是抛错）。

总的来说，严格模式使得 JavaScript 语言有了更严谨的错误检查体系，促使开发者编写出更安全、更有利于维护和运行效率的代码。

689.ES6 中的 Set 对象该如何遍历【热度: 410】【JavaScript】【出题公司: TOP100 互联网】

Set 遍历

在 JavaScript 中，`Set` 对象表示值的集合，在这个集合中每个值只出现一次。`Set` 对象是可迭代的，因此你可以使用多种方法来遍历它：

1. **for...of 循环**:

可以使用 `for...of` 循环来遍历 `Set`。

```
let mySet = new Set([1, 2, 3, 4, 5]);

for (let value of mySet) {
  console.log(value); // 输出: 1, 2, 3, 4, 5
}
```

1. **forEach 方法**:

`Set` 对象有一个 `forEach` 方法，就像 `Array` 一样。你可以提供一个回调函数，该函数将对 `Set` 中的每个元素执行。

```
mySet.forEach((value) => {
  console.log(value); // 输出: 1, 2, 3, 4, 5
});
```

1. **扩展运算符 (...)**:

扩展运算符可以将 `Set` 对象转换为数组。

```
let array = [...mySet];
// 现在可以使用数组的遍历方法
for (let i = 0; i < array.length; i++) {
```

```
  console.log(array[i]); // 输出: 1, 2, 3, 4, 5
}

```

1. **Array.from 方法**:

`Array.from` 方法可以将 `Set` 对象转化为数组。

```
let array = Array.from(mySet);
// 现在可以使用数组的遍历方法
array.forEach((value) => {
  console.log(value); // 输出: 1, 2, 3, 4, 5
});
```

1. **keys(), values(), entries() 方法**:

尽管 `Set` 对象没有键名只有键值， `keys()` 和 `values()` 方法的行为事实上是一样的，它们都会返回一个新的可迭代对象。 `entries()` 方法也存在于 `Set` 上，但由于 `Set` 没有键名，它返回的迭代器将会为每个值提供一个[value, value]形式的数组。

```
for (let value of mySet.keys()) {
  console.log(value); // 输出: 1, 2, 3, 4, 5
}

for (let value of mySet.values()) {
  console.log(value); // 输出: 1, 2, 3, 4, 5
}

for (let entry of mySet.entries()) {
  console.log(entry); // 输出: [1, 1], [2, 2], [3, 3], [4, 4], [5, 5]
}

```

使用哪种方法取决于你的个人喜好和具体的场景。但最常用的可能是 `for...of` 循环和 `forEach` 方法。

690.ES6 中的 Map 对象该如何遍历【热度: 411】【JavaScript】【出题公司: TOP100 互联网】

Map 遍历

在 JavaScript 中，`Map` 对象当然可以被遍历。`Map` 对象持有键值对，任何值(对象或者原始值) 都可以作为一个键或一个值。你可以使用 `Map` 对象的几种方法遍历其中的键值对。

以下是几种遍历 Map 对象的方法：

1. **使用 `forEach()` 方法：**

`Map` 对象有一个 `forEach` 方法，你可以像遍历数组一样使用它来遍历 `Map`。`forEach` 方法会按照插入顺序遍历 Map 对象。

```
let myMap = new Map();
myMap.set("a", "alpha");
myMap.set("b", "beta");
myMap.set("g", "gamma");

myMap.forEach((value, key) => {
  console.log(key + " = " + value);
});

```

1. **使用 `for...of` 循环：**

你可以使用 `for...of` 循环来遍历 `Map` 对象的键值对(`entries`)，键(`keys`)或值(`values`)。

- 遍历 `Map` 的键值对:

```
for (let [key, value] of myMap) {
  console.log(key + " = " + value);
}
```

- 遍历 `Map` 的键:

```
for (let key of myMap.keys()) {
  console.log(key);
}
```

- 遍历 `Map` 的值:

```
for (let value of myMap.values()) {
  console.log(value);
}
```

1. **使用扩展运算符：**

你还可以使用扩展运算符来将 `Map` 对象的键值对、键或值转换为数组。

- 键值对数组:

```
let keyValueArray = [...myMap];
console.log(keyValueArray);
```

- 键数组:

```
let keysArray = [...myMap.keys()];
console.log(keysArray);
```

- 值数组:

```
let valuesArray = [...myMap.values()];
console.log(valuesArray);
```

每种方法的使用取决于你的具体需求。通常， `for...of` 和 `forEach()` 会用得更多，因为它们可以直接操作键和值。

693.介绍一下 TS 中的泛型【热度: 118】【TypeScript】【出题公司: TOP100 互联网】

关键词：TS 泛型

TypeScript 的泛型是一种工具，它能够使代码更加灵活，能够适配多种类型而非单一的类型。泛型可以创建可重用的组件，这些组件可以支持多种类型的数据，而不失去类型检查时的安全性。

泛型的基本概念

在 TypeScript 中, 泛型使用一个类型变量，常见的类型变量有 `T` , `U` , `V` 等。通过类型变量，你可以创建一个可以适应任何类型的组件（比如函数、接口或类）。类型变量像是函数或类的一个特殊参数，但这个参数是类型而非具体的值。

泛型的使用场景

1. **函数**：你可以创建一个泛型函数，该函数可以接受任意类型的参数，同时保证输入参数和返回参数类型相同：

```
function identity<T>(arg: T): T {
  return arg;
}
```

这里 `T` 用作类型变量，可以捕获用户提供的类型（比如 `number`），然后这个类型将被用于函数的参数和返回类型。

1. **接口**：使用泛型定义接口可以创建可用于多种类型的接口。

```
interface GenericIdentityFn<T> {
  (arg: T): T;
}

function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: GenericIdentityFn<number> = identity;

```

这里 `GenericIdentityFn` 接口定义了一个属性，它是一个接收 `T` 类型参数并返回 `T` 类型的函数。

1. **type**：`type` 关键字可以用来创建类型别名，它确实支持泛型。你可以为类型别名定义泛型参数，然后在使用该类型别名时指定具体的类型。

下面是使用泛型的类型别名的例子：

```
// 这里定义了一个带有泛型参数 T 的类型别名
type Container<T> = {
  value: T;
};

// 可以这样使用类型别名
let numberContainer: Container<number> = { value: 1 };
let stringContainer: Container<string> = { value: "Hello" };

// 使用类型别名定义函数类型
type ReturnFunction<T> = () => T;

// 这个函数返回一个数字
let myFunction: ReturnFunction<number> = () => 42;

```

```
// 使用带有两个参数的泛型
type KeyValue<K, V> = {
  key: K;
  value: V;
};

let keyValue: KeyValue<string, number> = { key: "testKey", value: 123 };

```

通过使用泛型，`type` 可以定义灵活的类型别名，使得别名能够用于各种不同的数据类型，同时保持类型的安全性。这使得你可以在类型别名中使用泛型来捕获传递给别名的类型信息。

1. **类**：泛型也可以用于类定义中，使得类可以灵活地与多种类型协作。

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
  return x + y;
};

```

这里，`GenericNumber<T>` 类具有一个类型为 `T` 的属性 `zeroValue` 和一个用两个 `T` 类型参数返回 `T` 类型的方法 `add`。

泛型约束

有时你可能希望对泛型进行限制，只允许使用满足特定接口的类型。这称为泛型约束。

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length); // Now we know it has a .length property, so no more error
  return arg;
}

```

在这里，我们约束了类型 `T` 必须遵从 `Lengthwise` 接口，确保传入的类型具有 `length` 属性。

泛型中使用类型参数

你还可以在泛型中使用类型参数本身。

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}

let x = { a: 1, b: 2, c: 3 };

getProperty(x, "a"); // Okay
getProperty(x, "m"); // Error: Argument of type '"m"' isn't assignable to '"a" | "b" | "c"'

```

在这个示例中，`getProperty` 函数有两个参数：`obj` 和 `key`，`obj` 是对象 `T`，`key` 是 `T` 中键的集合 `keyof T` 的成员。

通过泛型，TypeScript 允许你在保持类型安全的同时创建灵活，可适用于多种类型的组件。这样你就能够写出更加通用且易于复用的代码。

694.TS 中 any 和 unknow 的区别【热度: 412】【TypeScript】【出题公司: TOP100 互联网】

在 TypeScript 中，`any` 和 `unknown` 都代表可以赋予任何类型的值，但它们在使用上有明显的不同。

any 类型

1. **最不安全的类型**：`any` 类型是 TypeScript 类型系统的逃逸舱口，使用 `any` 可以让任何表达式绕过类型检查，转而采用 JavaScript 动态类型的行为。
2. **类型放弃**：当你把一个值声明为 `any` 类型，你本质上在告诉 TypeScript 编译器：“信任我，我知道我在做什么。”编译器不会对 `any` 类型的值进行类型检查，这意味着你可以对它执行任何操作，无论它的实际类型。

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean

notSure.ifItExists(); // okay, toExist might exist at runtime
```

```
6
```

上述代码没有错误，因为 `notSure` 被声明为 `any` 类型。

unknown 类型

1. **类型安全的 any**：与 `any` 相比， `unknown` 类型是类型安全的。它标志着一个值可以是任何类型，但与 `any` 不同的是，当值被声明为 `unknown` 时，你无法对该值执行任意的操作。
2. **需要断言或类型细化**：在对 `unknown` 类型的值执行大部分操作之前，你需要使用类型断言或类型守卫来细化类型。这迫使开发者更积极地处理 `unknown` 类型的值，因此可以防止潜在的错误。

```
let unsure: unknown = 4;
unsure = "maybe a string instead";
unsure = false; // okay, still uncertain

// unsure.ifItExists(); // Error: Object is of type 'unknown'.
// 下面是对unknown类型进行类型断言的示例
(unsure as string).length; // okay, we have asserted that unsure is a string

// 或者使用类型守卫
if (typeof unsure === "string") {
  console.log(unsure.length); // okay, we have checked it's a string
}

```

如你所见，你不能像处理 `any` 类型那样直接调用 unknown 类型的方法或属性，必须先进行类型检查。

总结

`unknown` 类型是 `any` 类型的类型安全对应物。当不确定一个值的类型时应首选使用 `unknown` 。这样，你会被迫在对该值执行操作之前进行适当的类型检查。这增加了一层类型安全性，可以帮助避免错误。相比之下， `any` 类型则完全放弃了类型检查，通常应该尽量避免。

695.Proxy 和 Reflect 有什么关系？【热度: 361】【JavaScript】【出题公司: 滴滴】

`Proxy` 和 `Reflect` 是 ES6 (ECMAScript 2015) 中引入的两个不同的构造函数，它们密切相关，通常在某些操作中一起使用。

1. **Proxy**： `Proxy` 对象用于定义基本操作的自定义行为，例如属性查找、赋值、枚举、函数调用等。当你对一个 `Proxy` 对象执行这些操作时，你可以拦截并重新定义这些操作的行为。

2. 下面是一些你可以使用 `Proxy` 拦截的操作:
   - `get`：读取属性值
   - `set`：设置属性值
   - `has`：`in` 操作符
   - `deleteProperty`：`delete` 操作符
   - `apply`：调用一个函数
   - 诸如此类的其他捕获器（handlers）
3. **Reflect**：`Reflect` 对象与 `Proxy` 捕获器（handlers）的方法一一对应。其目的是提供默认行为，对相应的对象操作进行默认的行为操作。在很多情况下，`Reflect` 的方法与对应的直接对象操作是相同的。
4. 这里是一些 `Reflect` 提供的方法的例子：
   - `Reflect.get()`：获取对象属性的值，类似于 `obj[prop]`
   - `Reflect.set()`：设置对象属性的值，类似于 `obj[prop] = value`
   - `Reflect.has()`：类似于 `prop in obj`
   - `Reflect.deleteProperty()`：类似于 `delete obj[prop]`
   - `Reflect.apply()`：调用一个函数
   - 其他与 `Proxy` 捕获器相对应的方法

两者的关系：`Proxy` 和 `Reflect` 的关系体现在它们共同协作时。在 `Proxy` 的捕获器函数中，开发者可以调用对应的 `Reflect` 方法，以实现默认的行为，同时加入自己的操纵和侧面逻辑。`Reflect` 方法提供了一种方便的方式来保持默认行为，而不需要手动编写这些语义。

例如，当在 `Proxy` 捕获器中捕获属性的读取行为时，使用 `Reflect.get()` 可以非常容易地调用相应对象的默认读取行为：

```
let obj = {
  a: 1,
  b: 2,
  c: 3,
};

let p = new Proxy(obj, {
  get(target, prop, receiver) {
    console.log(`读取了属性 ${prop}`);
    return Reflect.get(target, prop, receiver); // 调用默认操作
  },
  set(target, prop, value, receiver) {
    console.log(`将属性 ${prop} 设置为 ${value}`);
```

```
    return Reflect.set(target, prop, value, receiver); // 调用默认操作
  },
});

console.log(p.a); // 读取了属性 a，返回 1
p.b = 4; // 将属性 b 设置为 4

```

上面的例子中，通过 `Reflect` 对象的方法，我们不仅可以保持默认的 `get` 和 `set` 行为，还可以在这个过程之前或之后添加自己的逻辑。这样的设计使得代理行为的实现既安全又易于管理。

总而言之， `Proxy` 和 `Reflect` 共同提供了一种强大的机制来拦截和定义基本的 JavaScript 操作，`Reflect` 能提供操纵对象的默认方法，而 `Proxy` 则允许我们根据需要来定义这些操作的新行为。

697.JS 的加载会阻塞浏览器渲染吗？【热度: 243】【浏览器】【出题公司: TOP100 互联网】

关键词：浏览器渲染阻塞、JS 阻塞浏览器渲染

JavaScript 的加载、解析和执行默认情况下会阻塞浏览器的渲染过程。这是因为浏览器渲染引擎和 JavaScript 引擎是单线程的，并且二者共享同一个线程。JavaScript 在执行时会阻止 DOM 构建，因为 JavaScript 可能会修改 DOM 结构（例如添加、修改或删除节点）。出于这个原因，浏览器必须暂停 DOM 的解析和渲染，直到 JavaScript 执行完成。

默认情况下，当浏览器遇到一个 `<script>` 标签时，会立即停止解析 HTML，转而下载和执行脚本，然后再继续 HTML 的解析和渲染。这意味着在 HTML 文档中的 JavaScript 脚本的下载和执行过程中，页面的渲染是被阻塞的。

不过，你可以用下面几种方法调整脚本的加载和执行行为，以减少对浏览器渲染过程的阻塞：

1. **异步脚本（async）**： 在 `<script>` 标签中使用 `async` 属性可以使得脚本的加载变成异步操作。当使用 `async` 属性时，浏览器会在后台进行下载，但脚本的执行还是会阻塞 DOM 渲染。

```
<script async src="script.js"></script>
```

1. 使用 `async` 时，脚本会在下载完成后尽快执行，这可能会在文档解析完成之前或之后。
2. **延迟脚本（defer）**： `defer` 属性使得脚本在 HTML 解析完成之后、DOMContentLoaded 事件触发之前执行，不阻塞 HTML 的解析。

```
<script defer src="script.js"></script>
```

1. 使用 `defer`，脚本的执行顺序将按照它们在 DOM 中出现的顺序执行。
2. **动态脚本加载**： 你可以使用 JavaScript 动态创建 `<script>` 元素并添加到 DOM 中，这允许你控制脚本的加载和执行时机。

```
var script = document.createElement("script");
script.src = "script.js";
document.body.appendChild(script);
```

1. **移动脚本位置**： 将脚本放在 HTML 的底部，即 `<body>` 标签关闭之前，而不是放在 `<head>` 中，可以让页面内容先加载显示，从而减少用户对加载过程的可感知时间。

现代 Web 开发中通常推荐使用 `async` 或 `defer` 属性，提高页面加载性能，尤其是对于那些需要从外部服务器加载的大型 JavaScript 库来说尤为关键。

698.开发过程中为什么会选择使用 ts , 相比于 JS 开发， 有哪些优点？【热度: 318】【TypeScript】【出题公司: 阿里巴巴】

关键词：TS 开发优势

使用 TypeScript（简称 TS）而不是纯 JavaScript（JS）进行开发，是因为 TypeScript 提供了一些特性和优势，这些可以帮助开发者提高代码质量、可维护性以及开发效率。以下是使用 TypeScript 的一些主要优点：

1. **静态类型检查**： TypeScript 最大的优点是其静态类型系统。静态类型检查意味着错误（如类型错误、未定义的属性或函数）可以在代码运行之前被发现，通常在编写代码的过程中或编译阶段即被 IDE 捕获，这有助于减少运行时错误。
2. **代码智能与自动补全**： 由于类型注解，开发者在使用 IDE 或编辑器时可以得到更好的代码智能提示，包括自动补全、函数签名信息、跳转到定义等，从而提高开发效率。
3. **易于重构**： 类型安全意味着重构更加安全、更少风险。IDE 可以轻松做到像重命名变量、函数、类的成员等操作，并确保所有的引用都得到更新。
4. **更好的协作**： 类型系统可以作为代码中变量、函数和模块的文档说明，这使得团队协作时代码的意图更加清晰，尤其是在大型项目或多人协作环境中。
5. **更丰富的语言特性**： TypeScript 支持最新的和即将推出的 ECMAScript 特性，并添加了它自己的额外特性（如类型注解、接口、泛型、枚举等）。
6. **搭配现代框架**： 许多现代前端框架对 TypeScript 有良好支持。比如 Angular 是通过 TypeScript 开发的，Vue 和 React 也对 TypeScript 有很好的支持。
7. **避免隐式强制类型转换**： JavaScript 由于其动态类型的特性，在进行运算时可能会发生隐式类型转换，导致非预期结果。TypeScript 通过在编译阶段强制类型检查，减少这类问题的发生。

8. **增加大型项目的可管理性**： 对于大型、复杂的项目，TypeScript 的类型系统可以帮助代码更好地组织，易于理解和维护。

尽管 TypeScript 带来了许多优点，但它也需要一定的学习投入，并且搭建项目初始化时可能更加复杂。然而，对于需要长期维护和多人协作的项目，以及对类型安全有专门要求的应用，TypeScript 提供的长期益处远大于短期的劣势。

高级开发者相关问题【共计 9 道题】

684.Eslint 代码检查的过程是啥？【热度: 111】【web 应用场景】【出题公司: 阿里巴巴】

关键词：eslint 代码检测、eslint 代码检测 执行过程

ESLint 是一个插件化的静态代码分析工具，用于识别 JavaScript 代码中的问题。它在代码质量和编码风格方面有助于保持一致性。代码检查的过程通常如下：

1. **配置**： 首先需要为 ESLint 提供一套规则，这些规则可以在 `.eslintrc` 配置文件中定义，或者在项目的 `package.json` 文件中的 `eslintConfig` 字段里指定。规则可以继承自一套已有的规则集，如 `eslint:recommended` ，或者可以是一个流行的样式指南，如 `airbnb` 。也可以是自定义的规则集。
2. **解析**： 当运行 ESLint 时，它会使用一个解析器（如 `espree` ，默认的解析器）来解析代码，将代码转换成一个抽象语法树（AST）。AST 是代码结构的一个树状表示，能让 ESLint 理解代码的语义结构。
3. **遍历**： 一旦代码被转换成 AST，ESLint 则会遍历该树。它会查找树的每个节点，检查是否有任何规则适用于该节点。在遍历过程中，如果发现违反了某项规则，ESLint 将记录一个问题（通常称为“lint 错误”）。
4. **报告**： 在遍历完整个 AST 之后，ESLint 会生成一份报告。这份报告详细说明了它在代码中找到的任何问题。这些问题会被分类为错误或警告，根据配置设置的不同，某些问题可能会阻止构建过程或者被忽略。
5. **修复**： 对于某些类型的问题，ESLint 提供了自动修复的功能。这意味着你可以让 ESLint 尝试自动修复它所发现的问题，不需人工干预。
6. **集成**： ESLint 可以集成到 IDE 中，这样就可以在代码编写过程中即时提供反馈。它也可以被集成到构建工具如 Webpack 或任务运行器 Grunt、Gulp 中，作为构建过程或提交代码到版本控制系统前的一个步骤。

通过以上步骤，ESLint 帮助开发者在编码过程中遵循一致的风格和避免出现潜在的错误。

686.应用上线后， 怎么通知用户刷新当前页面？【热度: 466】【web 应用场景】【出题公司: 美团】

关键词：静态资源更新、页面版本更新、服务端推送

关键词：静态资源更新、页面版本更新

这个话题非常的有意思，问题的答案是比较开发的，这里仅代表作者本人的个人经验来做回答。 当然也可以自行去搜集掘金上的大佬们的博文。

首先第一个问题

用户在没有页面刷新的情况下， 如何去感知前端静态资源已经发生了更新？

首先要做静态资源版本管理。 这个版本直接给到 html 模板即可， 其他 link 打包的资源还是以哈希 code 作为文件名称后缀。

就类似于这样子的

```
xxx.1.0.0.html --> vender.hash_1.js、 vender.hash_2.js、 vender.hash_3.js、 vender.hash_1.css

xxx.1.0.1.html --> vender.hash_a.js、 vender.hash_b.js、 vender.hash_c.js、 vender.hash_d.css

```

如何主动推送给客户端

这个实现方式就非常的多了，我这里建议让服务端来做处理

因为我们前端静态资源打包之后， 大多数会上传到云存储服务器上， 或者甚至是 服务器本地 也行。 这个时候， 后端给一个定时任务， 比如 1 分钟去执行一次， 看看是否有新的 html 版本的内容生成。 如果有新的 html 版本内容生成， 且当前用户访问的还是旧版本， 那么直接发一个服务端信息推送即可（SSE 允许服务器推送数据到浏览器）。

这样做成本是最低的， 甚至可以说是一劳永逸。 前端是没有任何负债， 没有任何性能问题。

那是否还有别的处理方式呢？ 当然是有的。

1. **WebSockets**：

通过 WebSocket 连接，服务器可以实时地向客户端发送消息，包括静态资源更新的通知。收到消息后，客户端可以采取相应的措施，比如显示一个提示信息让用户选择是否重新加载页面。

1. **Service Workers(推荐)**：

Service workers 位于浏览器和网络之间，可以控制页面的资源缓存。它们也可用于检测资源更新，当检测到静态资源更新时，可以通过推送通知或在网站上显示更新提示。

1. **轮询**：

客户端用 JavaScript 定时发送 HTTP 请求到服务器，查询版本信息。如果检测到新版本，可以提醒用户或自动刷新资源。

在绝大多数情况下，使用 Service Workers 可能是最稳妥的做法，因为它不仅提供了资源缓存和管理的能力，而且也可以在后台做资源更新的检查，即使用户没有开启网页也能实现通知和更新的功能。当然，选择哪种方案还需考虑应用的需求、用户体验和实现复杂度等因素。

688.JS 严格模式为什么会禁用 with 语句？【热度: 210】【JavaScript】【出题公司: TOP100 互联网】

作者备注：

实话实说， 这个问题真的很冷门。 如果有面试官问到这个问题了，感觉就是坏。

但是作为一个知识点儿， 还是有一丢丢意思。 所以顺手就记录下来了。

在 JavaScript 中，严格模式禁用了 with 语句，主要是出于以下三个原因：

1. 性能问题：使用 with 语句会为 JavaScript 解释器带来优化难题。当使用 with 语句时，解释器在编译阶段无法确定对象属性的作用域，因此无法在编译时进行优化。这意味着在执行时需要做额外的作用域查找，可能会降低代码的执行效率。
2. 代码可读性和维护性：with 语句可以将一个对象的所有属性和方法直接引入到当前作用域中，这可能会带来潜在的命名冲突。如果一个属性在 with 语句内部和外部作用域都有定义，编写和维护代码的人员可能会对此感到困惑。因此，这种语句的使用可以使代码的可读性和维护性降低。
3. 编码错误可能性：with 语句改变了正常的作用域链查找规则，这可能会导致意外的变量分配。例如，如果 with 对象不包含某个属性，那么它可能意外地引用或创建一个全局变量，导致难以追踪的错误。

其中前两个原因还是比较好理解的，第三个原因， 「编码错误可能性」就需要好好解释下了：

这里 with 语法， 我就不过多讲解了哈。 如果不知道语法的同学， 我这儿丢一个传送门：

下面的例子展示了 `with` 语句如何导致潜在的编码错误：

考虑下面的对象和 `with` 语句：

```
var person = {
  name: "Alice",
  age: 25,
};

function updatePerson(person) {
  with (person) {
    name = "Bob"; // 意图是更新person的name属性
    age = 30; // 意图是更新person的age属性
  }
}

updatePerson(person);

```

```
15 console.log(person); // 输出: { name: 'Bob', age: 30 }, 这里看起来没问题
16
```

看起来这段代码没有问题，并且确实更新了 `person` 对象；但问题出现在如果 `with` 中的属性并不存在于对象中：

```
1  var person = {
2    name: "Alice",
3    age: 25,
4  };
5
6  function createNewPerson() {
7    var name = "Charlie";
8    var age = 20;
9
10   with (person) {
11     name = "David"; // 本意是更新person的name属性
12     age = 35; // 本意是更新person的age属性
13     // 由于person没有phone属性，所以这将创建一个全局变量phone
14     phone = "123-456-7890";
15   }
16
17   // 调用者可能预期这里的name和age还是'Charlie'和20 - 因为 with 预期是更改 person 的属性;
18   console.log(name, age); // 输出: 'David' 35, 而非'Charlie', 20
19 }
20
21 createNewPerson();
22
23 console.log(window.phone); // 输出: '123-456-7890'
24
```

在这个例子里：

- `name` 和 `age` 都是局部变量，但它们被 `with(person)` 覆盖了，因为 `person` 对象确实有这样的属性。
- `phone` 属性不在 `person` 对象中， `with` 语句创建了一个全局变量 `phone` 。

这展示了 `with` 语句如何引入两个潜在的陷阱：

1. **局部变量被意外覆盖：** 函数内部的 `name` 和 `age` 变量被覆盖，因为 `with` 语句使得 `person` 对象的属性在作用域链中的优先级高于局部变量。
2. **意外的全局变量：** 因为 `person` 对象中没有 `phone` 属性，所以 `phone` 变成了一个全局变量。

这些情况可能会导致难以追踪的错误和未预期的副作用，这正是为何严格模式中不允许使用 `with` 语句的原因之一。在严格模式中，代码会因试图使用 `with` 而抛出语法错误，上述的误导性行为就不会发生。

691.Webpack 项目中通过 script 标签引入资源，在项目中如何处理？【热度: 100】【工程化】【出题公司: TOP100 互联网】

> 也是作者无意中看到的一个有意思的问题。
>
> 虽然有意思， 但是没有任何价值， 如果说在项目中遇到过的， 而且处理过的同学， 肯定知道怎么回答。
>
> 但是压根没有碰到过得， 就算是你工作十年的老油条也是干望着。 所有没有任何面试价值。
>
> 故此， 可以当做科普来看看就行。

在使用 Webpack 打包的项目中，通常资源（如 JavaScript、CSS、图片等）会被 Webpack 处理，因为 Webpack 的设计初衷就是将所有资源视为模块，并进行有效的管理和打包。但有时候可能需要通过 `<script>` 标签直接引入资源，这通常有两种情况：

1. **在 HTML 文件中直接引入：** 可以在项目的 HTML 文件中直接使用 `<script>` 标签来引入外部资源：

```
<!-- 若要使用 CDN 上托管的库 -->
<script src="https://cdn.example.com/library.js"></script>
```

这种方法简单直接，但要记住，由于这些资源不会被 Webpack 处理，它们不会被包含在 Webpack 的依赖图中，并且也不会享受到 Webpack 的各种优化。

使用 Webpack 管理： 如果想要 Webpack 来处理这些通过 `<script>` 引入的资源，可以使用几种插件和加载器：

- `html-webpack-plugin` 可以帮助你生成一个 HTML 文件，并在文件中自动引入 Webpack 打包后的 bundles。
- `externals` 配置允许你将一些依赖排除在 Webpack 打包之外，但还是可以通过 `require` 或 `import` 引用它们。
- `script-loader` 可以将第三方全局变量注入的库当作模块来加载使用。

例如，使用 `html-webpack-plugin` 和 `externals` ，你可以将一个库配置为 external，然后通过 `html-webpack-plugin` 将其引入：

```
// webpack.config.js 文件
const HtmlWebpackPlugin = require("html-webpack-plugin");

```

```
module.exports = {
  // ...
  externals: {
    libraryName: "LibraryGlobalVariable",
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: "src/index.html",
      scriptLoading: "blocking", // 或者 'defer'
    }),
  ],
};

```

然后，在你的 `index.html` 模板文件中可以这样引入资源：

```
<script src="https://cdn.example.com/library.js"></script>
```

使用 `externals` 的方法能让你在 Webpack 打包的模块代码中用正常的 `import` 或 `require` 语句来引用那个全局变量：

```
// 你的 JavaScript 代码文件中
import Library from "libraryName"; // 虽然定义了external，Webpack依然会处理这个import
```

应根据项目需求和现有的架构来决定使用哪种方法。上述两种方法中，第二种可以更好地利用 Webpack 的功能，第一种则更加简单直接。

692.在 Babel 里，stage0、stage1、stage2 和 stage3 分别代表什么含义？【工程化】【出题公司: TOP100 互联网】

无意中看到别人一个面试问题， 个人感觉问这个问题的面试官， 不是蠢就是坏。

没有任何面试价值， 无法考察候选人水平。

仅仅作为科普类型参考 - 热度为 0

Babel 是一个流行的 JavaScript 编译器，它允许开发者使用新的语言特性，然后将它们编译成可以在当前和低版本的浏览器或环境中运行的代码。

在 Babel 里，stage0、stage1、stage2 和 stage3 这些术语指的是 ECMAScript 提案的不同阶段。ECMAScript 是 JavaScript 语言的标准化规范，新的特性进入标准之前会通过几个阶段的提案。

这些阶段表示了一个特性在正式成为 ECMAScript 标准的一部分之前的成熟度。这个过程有一个官方的 5 个阶段流程，即从 Stage 0（strawman）到 Stage 4（finished）。下面是这些阶段的含义：

- **Stage 0 - Strawman（稻草人阶段）：**初始阶段，任何尚未被 TC39（ECMAScript 的标准化组织）官方审议的提案都属于这里。这些都是某个委员或者社区成员提交的想法，还不算是正式的提案。
- **Stage 1 - Proposal（提案阶段）：**这个阶段的特性是值得进一步探讨的。它们需要有一个形式化的提案和一个负责人。在这个阶段，主要是确定问题和解决方案，以及进行初步探讨。
- **Stage 2 - Draft（草案阶段）：**一旦一个提案到达这个阶段，它就被认为是初步规格的草案。特性的描述应该足够具体和详细，并且有初步的实现。这个阶段通常需要提案的规格文本和至少一种实验性实现。
- **Stage 3 - Candidate（候选阶段）：**在候选阶段，提案的规格已经基本完成，并且需要更多的用户反馈来发现潜在问题。通常在这个阶段，实现者和开发者开始在生产环境中尝试使用这些特性，发现问题并提出改善建议。
- **Stage 4 - Finished（完成阶段）：**当一个提案达到这个阶段，它已经准备好被集成到下一个版本的 ECMAScript 标准中了。这意味着它已经获得了多个独立环境的实现，通过了综合的可行性和稳定性测试，并且已经被 TC39 委员会接受。

开发者们可以根据特性的稳定性和自己的需求，选择使用 Babel 的哪个阶段的预设。然而，请注意，使用较低阶段的提案特性在生产环境中是有风险的，因为它们还没有被完全确定并可能会在将来发生变更。

696.浏览器对队头阻塞有什么优化？【热度: 368】【网络、浏览器】【出题公司: 滴滴】

关键词：队头阻塞优化

队头阻塞（Head-of-Line Blocking，缩写 HoLB）问题主要发生在网络通信中，特别是在使用 HTTP/1.1 和以前版本时，在一个 TCP 连接中同一时间只能处理一个请求。即使后续的请求已经准备好在客户端，它们也必须等待当前处理中的请求完成后才能被发送。这会延迟整个页面或应用的网络请求，降低性能。

现代浏览器和协议已经实施了多种优化措施来减少或解决队头阻塞问题：

1. **HTTP/2**： 为了解决 HTTP/1.x 的诸多问题，包括队头阻塞问题，HTTP/2 引入了多路复用（multiplexing）功能。这允许在同一 TCP 连接上同时传输多个独立的请求-响应消息。与 HTTP/1.1 相比，HTTP/2 在同一个连接上可以并行处理多个请求，大大减少了队头阻塞的问题。
2. **服务器推送**： HTTP/2 还引入了服务器推送（server push）功能，允许服务器主动发送多个响应到客户端，而不需要客户端明确地为每个资源提出请求。这提高了页面加载的速度，因为相关资源可以被预先发送而无需等待浏览器请求。
3. **域名分散（Domain Sharding）**： 这种技术常用于 HTTP/1.1 中，通过创建多个子域，使得浏览器可以同时开启更多的 TCP 连接来加载资源。虽然这种方法可以在一定程度上减轻队头阻塞，但它增加了复杂性，并且在 HTTP/2 中由于多路复用功能变得不再必要。

4. **连接重用（Connection Reuse）**： 这是 HTTP/1.1 中的一个特性，即持久连接（Persistent Connections），允许在一次 TCP 连接中发送和接收多个 HTTP 请求和响应，而无需开启新的连接，从而减少了 TCP 握手的开销并提升了效率。
5. **资源优化**： 减少资源的大小通过压缩（如 GZIP），优化图片，减少 CSS 和 JavaScript 文件的大小等，可以减少队头阻塞的影响，因为小资源文件传输更快。
6. **优先级设置**： HTTP/2 允许设置资源的加载优先级，使得关键资源（如 HTML，CSS，JavaScript）可以比不那么重要的资源（如图片，广告）更早加载。
7. **预加载**： 浏览器可以通过使用 `<link rel="preload">` 标签预加载关键资源，例如字体文件和关键脚本，这样可以确保它们在主要内容加载之前已经准备好。
8. **HTTP/3 和 QUIC 协议**： HTTP/3 是未来的推进方向，它基于 QUIC 协议，一个在 UDP 之上的新传输层协议，旨在进一步减少延迟，解决 TCP/IP 协议的队头阻塞问题。

总的来说，HTTP/2 的特性如多路复用、服务器推送和优先级设置都有助于减少队头阻塞。而 HTTP/3 的引入可能会在未来为网络通信带来根本性的变化。在使用 HTTP/2、HTTP/3 和浏览器级别的优化时，网页开发者也需注意资源加载优化的最佳实践，以更全面地应对队头阻塞问题。

699.你在开发过程中， 使用过哪些 TS 的特性或者能力？ 【热度: 670】【TypeScript】【出题公司: 阿里巴巴】

关键词：TS 特性和能力

> 这个为何被作者列为了高阶范围的问题
> 原因如下：大多数纯写业务的同学， 可能就用到了 `interface` 和 `type` 最多加上 `enum` 和 `泛型`；
> TS 很多牛掰的特性， 是伴随做复杂的工具库， 或者处理复杂业务场景来的。

这里直接上干货：

1. **Utility Types（工具类型）**：
   - **Partial<T>**: 将类型 T 的所有属性变为可选。
   - **Required<T>**: 将类型 T 的所有属性变为必选。
   - **Readonly<T>**: 将类型 T 的所有属性变为只读。
   - **Record<K, T>**: 创建一个具有指定键类型 K 和值类型 T 的新对象类型。
   - **Pick<T, K>**: 从类型 T 中选择指定属性 K 形成新类型。
   - **Omit<T, K>**: 从类型 T 中排除指定属性 K 形成新类型。
   - **Exclude<T, U>**: 从类型 T 中排除可以赋值给类型 U 的类型。
   - **Extract<T, U>**: 从类型 T 中提取可以赋值给类型 U 的类型。
   - **NonNullable<T>**: 从类型 T 中排除 null 和 undefined 类型。
   - **ReturnType<T>**: 获取函数类型 T 的返回类型。

- **Parameters<T>**: 获取函数类型 T 的参数类型组成的元组类型。

2. **条件判定类型**：
   - **Conditional Types（条件类型）**: 根据类型关系进行条件判断生成不同的类型。
   - **Distribute Conditional Types（分布式条件类型）**: 分发条件类型，允许条件类型在联合类型上进行分发。
3. **Mapped Types（映射类型）**：根据已有类型创建新类型，通过映射类型可以生成新的类型结构。
4. **Template Literal Types（模板文字类型）**：使用字符串模板创建新类型。
5. **类型推断关键字**：
   - **keyof 关键字**：关键字允许在泛型条件类型中推断类型变量。
   - **instanceof**：运算符用于检查对象是否是特定类的实例。
   - **in**：用于检查对象是否具有特定属性。
   - **type guards**：类型守卫是自定义的函数或条件语句，用于在代码块内缩小变量的类型范围。
   - **as**：用于类型断言，允许将一个变量断言为特定的类型。

700.babel 核心库有哪些？【热度: 35】【工程化】【出题公司: 小红书】

关键词：babel 核心库

Babel 是一个 JavaScript 编译器，主要用于将 ES6 及以上版本的代码转换为向后兼容的 JavaScript 语法，以便在当前和旧版浏览器或环境中执行。核心的 Babel 库主要包括：

1. **@babel/core**: 这是 Babel 编译器的核心包，提供了 Babel 的主要转换引擎。它包含了解析、转换和生成代码的主要功能。几乎所有的 Babel 操作都需要这个模块作为基础。
2. **@babel/cli**: 这是 Babel 的命令行接口，通过它可以在终端或命令提示符中运行 Babel。它允许你执行转编译操作，如将 ES6 代码转换为 ES5。
3. **@babel/preset-env**: 这是一个智能预设，允许你使用最新的 JavaScript，而不必管理语法转换。`@babel/preset-env` 会根据你的目标环境（比如特定版本的浏览器或 Node.js），自动决定使用哪些 Babel 插件和 polyfills。
4. **@babel/polyfill** (现在已经被废弃，推荐使用 `core-js` 和 `regenerator-runtime`): 早期 Babel 版本中用于模拟完整的 ES2015+环境的包。它的目的是在全局范围内添加填充以模拟较新的环境。从 Babel 7.4.0 开始，建议直接包括 `core-js` 和 `regenerator-runtime`，因为这提供了更好的模块化和按需加载功能。
5. **babel-loader**: 这是 Babel 的一个 webpack 插件，可以将 Babel 集成到 webpack 构建过程中，使得你可以使用 webpack 来处理和打包使用了新版 JavaScript 语法的文件。
6. **@babel/plugin-transform-runtime**: 这个插件用于复用 Babel 注入的辅助代码，以节省代码大小，并能够在不污染全局环境的情况下使用新语言特性的 polyfills。

除了这些核心库外，还有许多可用的 Babel 插件，以支持各种 JavaScript 语法和特性（比如装饰器、类属性等）。这些插件可以按需引入，配置在 Babel 的配置文件（通常是 `.babelrc` 或 `babel.config.js`）中。这些插件的命名通常遵循 `@babel/plugin-` 的格式。

701.[React] 为什么 react 组件，都必须要申明一个 `import React from 'react';`【热度: 115】【web 框架、工程化】【出题公司: 小红书】

关键词：babel 编译 react

首先要知道一个事情： **JSX 是无法直接运行在浏览器环境**。

原因

JSX 语法不能直接被浏览器解析和运行，因此需要插件 `@babel/plugin-transform-react-jsx` 来转换语法，使之能够在浏览器或任何 JavaScript 环境中执行。

所以 React 组件需要引入 `React` 的一个主要原因是：在组件中使用 JSX 时，JSX 语法最终会被 Babel 编译成使用 `React.createElement` 方法的 JavaScript 代码。也就是说，任何使用 JSX 的 React 组件的背后都隐含了 `React.createElement` 的调用。

例如，当你编写如下的 JSX 代码：

```
const MyComponent = () => {
  return <div>Hello, World!</div>;
};
```

Babel 会将这段 JSX 编译为如下的 JavaScript 代码：

```
const MyComponent = () => {
  return React.createElement("div", null, "Hello, World!");
};
```

由于编译后的代码调用了 `React.createElement`，因此你需要在文件顶部导入 `React` 对象才能使用它。即使你在组件中并没有直接使用 `React` 对象，编译后的代码依赖于 `React` 的运行时。

Babel 7.0+ / React 17+ ， 可以不再需要 import React

在 Babel 7.0 版本之后， `@babel/plugin-transform-react-jsx` 插件还支持一个自动模式，它可以自动引入 JSX 转换所需的 `React` 包，无需手动在每个文件中添加 `import React from 'react'`。

注意，随着 React 17 的新 JSX 变换，它们引入了一个新的 JSX 转换方式，这在新的 Babel 插件 `@babel/plugin-transform-react-jsx` 和 `@babel/preset-react` 中得到了支持。这意味着在写 JSX 时，你不再需要导入 React。这个插件现在接收一个 `{ runtime: 'automatic' }` 选项来启用这一特性。

举个例子，在使用新的 JSX 转换之后，编译器将会自动引入 JSX 的运行时库，而不是 React，例如对于一个使用了新转换的 `MyComponent` 的组件:

```
// React 17+ 及支持新JSX转换的环境，可以不需要显式写这行
// import React from 'react';

const MyComponent = () => {
  return <div>Hello, World!</div>;
};


```

在新的转换下，你会看到类似 `import { jsx as _jsx } from 'react/jsx-runtime'` 的东西或者类似的别名，被自动插入到转译后的文件中，而不再是直接的 `React.createElement` 调用。这就是为什么在新版本的 React 中，你可能不再需要手动导入 React 了。

补充一个细节知识点： plugin-transform-react-jsx 和 @babel/preset-react` 是啥关系

它们是包含关系： `@babel/preset-react` 包括了 `@babel/plugin-transform-react-jsx`

`@babel/plugin-transform-react-jsx` 和 `@babel/preset-react` 都是 Babel 插件，它们在处理 React 项目中的 JSX 代码方面有关联，但它们的用途和包含的内容有所不同。

1. **@babel/plugin-transform-react-jsx**: 这是一个特定的 Babel 插件，它的功能就是将 JSX 语法转换为 `React.createElement` 调用。随着 React 17 的更新，它还允许使用新的 JSX 转换，无需导入 React 就可以使用 JSX。这意味着，在文件中不再需要 `import React from 'react'` 语句了，就可以使用 JSX。
2. 这个插件通常用于开发者想要精细控制某个具体转换功能时。如果你只需要转换 JSX 语法，但不需要处理其他与 React 相关的转换或优化，你可能会单独使用这个插件。
3. **@babel/preset-react**: 这是一个 Babel 预设，它是一组 Babel 插件的集合，旨在为 React 项目提供所需的全部 Babel 插件。 `@babel/preset-react` 包括了 `@babel/plugin-transform-react-jsx` ，但它还包含了其他一些插件，如处理 React 的显示名称的 `@babel/plugin-transform-react-display-name` ，以及为开发模式和生产模式添加/删除某些代码的插件。

4. 预设的好处是简化了配置过程。开发者可以在 Babel 的配置中一次性添加 `@babel/preset-react`，而不是单独添加每一个与 React 相关的 Babel 插件。此外，预设将维护这些插件的正确版本和顺序，这有助于避免潜在的配置错误。