**前端工程化**

## 工程化最新抄录 ~ 前端应用 CICD 有哪些方式实现【热度: 120】#1029

**说说你对前端工程化的理解**

**什么是前端工程化**

前端工程化是指将前端开发过程中的一系列流程和工具进行规范和自动化，从而提高开发效率、减少重复劳动、降低出错率。前端工程化的目标是让前端开发更高效、更优质。

**1.1 前端工程化的定义**

前端工程化是指在前端开发过程中，将前端开发的流程、工具和规范化，并使用相关技术实现自动化，包括但不限于代码编写、测试、构建、部署等环节，以提高前端开发效率、提高代码质量和可维护性。

**1.2 为什么需要前端工程化**

前端工程化能够极大地提高开发效率，提高代码质量和可维护性，减少出错率和重复工作。随着前端开发项目越来越复杂，需要开发的功能越来越多，手动进行前端开发将面临越来越大的挑战。

而采用前端工程化的方式，可以极大地减轻前端开发的工作负担，让开发人员更加专注于业务逻辑的开发。

**1.3 什么是模块化开发？请解释CommonJS和ES Modules的区别。**

模块化开发是一种将代码划分为独立、可复用的模块的开发方法。通过模块化，可以将复杂的代码拆分为小块，每个模块负责特定的功能，使代码更易于管理、测试和维护。模块化开发提供了一种组织代码的方式，通过明确的接口和依赖关系，实现代码的解耦和复用。

在 JavaScript 中，有两种常用的模块化规范：CommonJS和ES Modules（ESM）。

CommonJS 是一种用于服务器端的模块化规范，它在 Node.js 中广泛使用。CommonJS 通过 require() 和 module.exports 实现模块的导入和导出。在 CommonJS 中，每个模块都是一个独立的文件，通过 require() 导入其他模块的输出，然后通过 module.exports 或 exports 导出模块的功能。这种同步的加载方式适用于服务器端的场景。

ES Modules 是 ECMAScript 官方定义的模块化规范，在现代浏览器和最新的 Node.js 版本中原生支持。ES Modules 使用 import 和 export 关键字来导入和导出模块。与 CommonJS 不同，ES Modules 支持异步加载，适用于浏览器端的模块化开发。

ES Modules 还支持命名导入和导出、导入和导出多个值、动态导入等更多的语法特性，使得模块的管理更加灵活和强大。

**主要区别如下：**

* CommonJS 是同步加载模块的规范，适用于服务器端，而 ES Modules 支持异步加载模块，适用于浏览器端。
* CommonJS 使用 require() 和 module.exports，而 ES Modules 使用 import 和 export。
* ES Modules 具有更多的语法特性，如命名导入导出、多个导入导出、动态导入等。

**前端工程化的核心概念**

前端工程化的核心概念包括**模块化、打包构建、自动化部署、自动化测试和持续集成**等。

**2.1 模块化**

模块化是指将一个大的应用程序划分成多个小的模块，每个模块都有自己的功能和特点，可以独立开发、测试和维护。常见的模块化方案有 CommonJS、ES6 模块、AMD 等。

**2.2 打包构建**

打包构建是指将多个模块组合起来，生成可以在浏览器中运行的代码。打包构建的过程包括代码压缩、文件合并、资源管理等，常见的打包构建工具有 webpack、rollup 等。

**2.3 自动化部署**

自动化部署是指将打包构建后的代码部署到生产环境或测试环境中的自动化过程。自动化部署可以减少手动部署的错误和工作量，同时也可以缩短部署的时间。常见的自动化部署工具有 Jenkins、Travis CI 等。

**2.4 自动化测试**

前端工程化的另一个重要概念是自动化测试。自动化测试是指使用自动化工具对代码进行测试，以确保它们在开发过程中不会出现问题，并且在部署到生产环境之前也不会出现问题。

自动化测试可以分为两类：单元测试和端到端测试。单元测试是指测试应用程序中最小的可测试单元，例如一个函数或一个类。端到端测试是指测试应用程序的整个流程，包括用户界面和后端逻辑。

自动化测试的优势在于它可以提高开发效率和代码质量。它可以帮助开发人员在更早的阶段发现问题，并且可以确保代码的正确性，减少代码中的错误和缺陷。

**2.5 持续集成**

持续集成是指在应用程序开发过程中，将代码的改变频繁地集成到共享代码库中，并且每次集成都会进行自动化构建和自动化测试。这样可以确保代码的稳定性和质量，并且能够更快地检测和修复错误。

持续集成的优势在于它可以提高开发效率、加速代码部署和减少错误。它可以使团队更加协作，提高产品质量，并且可以更快地响应客户的需求。

**2.6 前端工程化的主要工具**

实现前端工程化需要使用多种工具。以下是一些常见的前端工程化工具：

包管理工具：npm、Yarn、Bower

构建工具：Webpack、Rollup、Parcel、Gulp、Grunt

自动化测试工具：Jest、Mocha、Karma、Cypress、Puppeteer

集成工具：Travis CI、Jenkins、CircleCI、GitLab CI/CD、GitHub Actions

这些工具可以帮助开发人员自动化完成各种任务，如安装和管理依赖项、打包和压缩代码、运行自动化测试和部署代码。

**前端工程化的应用**

前端工程化在现代Web开发中已经成为标配。下面是一些常见的前端工程化方案：

* 前端模块化：CommonJS、AMD、ES6模块化等
* 打包构建：Webpack、Rollup等
* 自动化部署：Jenkins、Travis CI等
* 自动化测试：Jest、Mocha、Karma等
* 持续集成：Jenkins、Travis CI等
* React项目的打包构建：使用Webpack将多个模块打包成一个文件，并进行优化和压缩，减少页面加载时间和提高性能。
* Vue.js项目的自动化部署：使用Travis CI实现自动化测试和部署，自动构建并部署代码到服务器，减少手动操作，提高效率。
* Angular项目的自动化测试：使用Jest和Karma进行自动化测试，覆盖率高，能够及时发现代码中的问题，提高代码质量。

**总结**

前端工程化是现代前端开发的标配，通过模块化、打包构建、自动化部署、自动化测试和持续集成等解决方案，可以提高开发效率、代码质量和团队协作效率，降低开发成本。

掌握前端工程化需要先掌握基本的前端技术，然后学习相关的工具和框架，多做实战项目进行实践，才能不断提升。

## 前端模块化发展历程？

前端模块化是指在前端开发中，通过模块化的方式组织代码，将代码按照一定规则分割成不同的模块，便于管理和维护。

前端模块化的发展历程如下：

* 早期，前端开发采用的是全局变量的方式进行开发，即将所有代码都放在一个文件中，通过全局变量进行交互。这种方式的问题在于，代码量较大，代码耦合度高，不易维护。
* 后来，前端开发采用了命名空间的方式进行组织代码，即将代码放在一个命名空间下，通过命名空间进行交互。这种方式解决了全局变量带来的问题，但是在开发大型应用时，仍然存在代码耦合度高、依赖管理不便等问题。
* 2009年，CommonJS提出了一种新的模块化规范，即将每个模块封装在一个独立的文件中，通过require和exports进行模块之间的依赖管理和导出。这种方式解决了前两种方式带来的问题，但是由于该规范是同步加载模块，不适用于浏览器环境。
* 2011年，AMD规范提出，即异步模块定义规范，采用异步的方式加载模块，可以在浏览器环境下使用。该规范主要是通过require和define方法进行模块之间的依赖管理和导出。
* 2013年，CommonJS和AMD的创始人合并了两种规范，提出了新的规范——CommonJS 2.0规范。该规范在CommonJS 1.0的基础上，增加了异步加载的功能，使其可以在浏览器环境下使用。
* 2014年，ES6（即ECMAScript 2015）正式发布，引入了模块化的支持，即通过import和export语句进行模块之间的依赖管理和导出。ES6的模块化规范具有更好的可读性、可维护性和性能优势，已成为前端开发的主流方式。
* 同时，还有一些第三方库，如RequireJS、SeaJS等，提供了更加灵活和可扩展的模块化方式，使得前端开发的模块化更加便捷和高效。

## AMD和CMD模块化有和区别?

AMD（Asynchronous Module Definition）和CMD（Common Module Definition）都是JavaScript模块化方案。它们的主要区别在于对依赖的处理方式上不同。

AMD是在require.js推广过程中诞生的，它的特点是提前执行，强调依赖前置。也就是说，在定义模块时就需要声明其所有依赖的模块。它的语法如下：

define(['dependency1', 'dependency2'], function(dependency1, dependency2) {

// 模块的定义

});

CMD是在Sea.js推广过程中诞生的，它和AMD非常相似，但是更加懒惰，是依赖就近，延迟执行。也就是说，在模块中需要用到依赖时，才去引入依赖。它的语法如下：

define(function(require, exports, module) {

var dependency1 = require('dependency1');

var dependency2 = require('dependency2');

// 模块的定义

});

简单来说，AMD是提前执行、依赖前置，CMD是延迟执行、依赖就近。两种模块化方案各有优缺点，选择哪种模块化方案需要根据实际情况和个人偏好进行考虑。

## 前端模块化的演进过程

### 1 函数作为块

最开始的时候，是以函数为块来编程，因为函数有自己的作用域，相对比较独立

### 2 namespace模式

使用对象作为独立块编程

### 3 IIFE模式

使用立即执行函数来创建块，可以形成独立的作用域，外面无法访问，借助window对象来向外暴露接口

### 4 CommonJS

最开始出现的模块化方案是在node.js中实现的。node中的模块化方案是根据CommonJS规范实现的。

CommonJS规定每个文件就是一个模块，以同步的方式引入其他模块

### 5 AMD和Require.js

CommonJS模块之前是同步引入的，这在服务端是没有什么问题的，因为文件都是保存在硬盘中，读取文件的速度是非常快的，同步加载带来的阻塞基本可以忽略不计。

但是如果在浏览器中使用CommonJS的话，因为js文件是存在服务端需要请求获取，所以同步的方式加载会极大的阻塞页面，显然是不可取的。

于是诞生了AMD（Asynchronous Module Definition）规范，一种异步加载的模块方案，使用回调函数来实现。require.js实现了AMD的规范。

#### Require.js的特点

* 依赖模块的代码都是放在回调函数中，等待模块都加载完成才执行这个回调函数，执行顺序可以保证
* 内部加载其他模块的时候，使用的是动态添加script标签的方式来实现动态加载
* 内部需要缓存模块暴露出来的接口，避免多次执行

**AMD推崇的是依赖前置，提前执行。**

### 6 CMD和Sea.js

由于require.js自身的一些问题存在，所以后来在国内（玉伯）诞生了CMD（Common Module Definition）和Sea.js。

CMD结合了CommonJS和AMD的特点，也是一种异步模块的方案，提倡就近依赖，延迟执行。

需要用到某个模块的时候，才用require引入，模块内部的代码也是在被引入的时候才会执行，声明的时候并没有执行。

语法设计上比较像CommonJS

### 7 ES Module

ES6开始，在语法标准上实现了模块化功能。简称ES Module

ES Module是一种静态依赖的模块化方案，模块与模块之间的依赖关系是在编译期完成连接的。

前面所说的三种方案都是动态模块化方案，依赖模块都是动态引入的，而且模块都是一个对象。而ES Module中，模块不是一个对象，模块与模块之间也不是动态引入的，而且编译期间静态引入的，所以无法实现条件加载

## 测试前端代码覆盖率一般有什么手段?【热度: 550】

前端代码的测试覆盖率通常是指衡量在测试过程中有多少代码被执⾏了的⼀个指标。测试覆盖率有助于了解测试的全⾯性，以下是测试前端代码覆盖率常⽤的⼿段：

1. 单元测试：

◦ 使⽤测试框架（例如Jest,Mocha,Jasmine等）编写单元测试。

◦ 利⽤测试框架或插件⽣成覆盖率报告（例如Istanbul/nyc⼯具可以与这些框架集成以⽣成覆盖率数据）。

2. 集成测试：

◦ 使⽤测试⼯具（⽐如Cypress,Selenium等）编写集成测试来模拟⽤户操作。

◦ 通常这些⼯具也⽀持收集代码覆盖率信息。

3. ⼿动测试与覆盖率⼯具结合：

◦ 在⼿动测试过程中，可以开启浏览器的覆盖率⼯具（如ChromeDevTools中的CoverageTab）记录覆盖率。

◦ 可以通过浏览器扩展程序或者⾃动化脚本来启动这些⼯具。

4. 测试覆盖率服务：

◦ 使⽤像Codecov或Coveralls这样的服务，在CI/CD流程中集成覆盖率测试和报告。

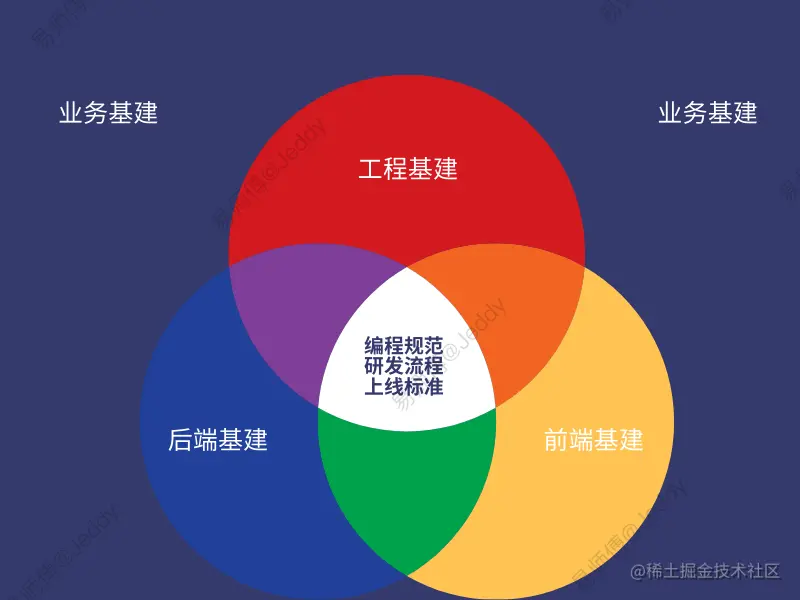
**如何从 0 到 1 搭建前端基建【热度: 404】**

**关键词：前端基建**

有一个非常经典的文章， 直接参考即可： [非大厂的我们，要如何去搞前端基建?](https://juejin.cn/post/7144881028661723167)

这里简单总结一下文章里面的要点

1. **什么是基建？**



**2.为什么要做前端基建？**

业务复用；

提升研发效率；

规范研发流程；

团队技术提升；

团队的技术影响力；

开源建设；

**3.前端基建如何推动落地？**

要合适的同学（资源）

要解决的问题（问题）

要解决问题方案计划书（方案）

要具体执行的步骤（执行）

**技术基建四大特性（切记）**

技术的健全性

基建的稳定性

研发的效率性

业务的体验性

**4.前端基建都有什么？**

前端规范（Standard）

前端文档（Document）

前端项目模板管理（Templates）

前端脚手架（CLI）

前端组件库（UI Design）

前端响应式设计 or 自适应设计

前端工具库（类 Hooks / Utils）

前端工具自动化（Tools）

接口数据聚合（BFF）

前端 SSR 推进

前端自动化构建部署（CI/CD）

全链路前端监控/数据埋点系统

前端可视化平台

前端性能优化

前端低代码平台搭建

微前端（Micro App）

**[React]为什么react组件，都必须要申明一个import React from 'react'?【热度: 115】**

**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。

**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。

**@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。

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。

**@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。

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。

在实践中，大多数开发 React 应用的开发者会使用 @babel/preset-react 因为它提供了一个即插即用的 Babel 环境，无需担心各个插件的具体细节。但是也有些情况下，为了更细致的优化和控制，开发者可能会选择手动添加特定的插件，包括 @babel/plugin-transform-react-jsx。

**为什么Vite速度比 Webpack快?【热度: 382】**

**关键词：vite 编译速度、vite 速度 与 webpack 速度**

**1、开发模式的差异**

在开发环境中，Webpack 是先打包再启动开发服务器，而 Vite 则是直接启动，然后再按需编译依赖文件。（大家可以启动项目后检查源码 Sources 那里看到）

这意味着，当使用 Webpack 时，所有的模块都需要在开发前进行打包，这会增加启动时间和构建时间。

而 Vite 则采用了不同的策略，它会在请求模块时再进行实时编译，这种按需动态编译的模式极大地缩短了编译时间，特别是在大型项目中，文件数量众多，Vite 的优势更为明显。

**2、对ES Modules的支持**

现代浏览器本身就支持 ES Modules，会主动发起请求去获取所需文件。Vite充分利用了这一点，将开发环境下的模块文件直接作为浏览器要执行的文件，而不是像 Webpack 那样先打包，再交给浏览器执行。这种方式减少了中间环节，提高了效率。

**3、底层语言的差异**

Webpack 是基于 Node.js 构建的，而 Vite 则是基于 esbuild 进行预构建依赖。esbuild 是采用 Go 语言编写的，Go 语言是纳秒级别的，而 Node.js 是毫秒级别的。因此，Vite 在打包速度上相比Webpack 有 10-100 倍的提升。

**什么是预构建依赖？**

预构建依赖通常指的是在项目启动或构建之前，对项目中所需的依赖项进行预先的处理或构建。这样做的好处在于，当项目实际运行时，可以直接使用这些已经预构建好的依赖，而无需再进行实时的编译或构建，从而提高了应用程序的运行速度和效率。

**4、热更新的处理**

在 Webpack 中，当一个模块或其依赖的模块内容改变时，需要重新编译这些模块。

而在 Vite 中，当某个模块内容改变时，只需要让浏览器重新请求该模块即可，这大大减少了热更新的时间。

**总结**

总的来说，Vite 之所以比 Webpack 快，主要是因为它采用了不同的开发模式、充分利用了现代浏览器的 ES Modules 支持、使用了更高效的底层语言，并优化了热更新的处理。这些特点使得 Vite在大型项目中具有显著的优势，能够快速启动和构建，提高开发效率。

**vite和webpack在热更新上有啥区别?【热度: 530】**

**关键词：热更新区别**

Vite 和 Webpack 在热更新上有一些区别：

* 模块级别的热更新：Vite 使用浏览器原生的 ES 模块系统，可以实现模块级别的热更新，即只更新修改的模块，而不需要刷新整个页面。这样可以提供更快的开发迭代速度。而在 Webpack 中，热更新是基于文件级别的，需要重新构建并刷新整个页面。
* 开发环境下的无构建：Vite 在开发环境下不会对代码进行打包构建，而是直接利用浏览器原生的模块导入功能，通过 HTTP 服务器提供模块的即时响应。这样可以避免了构建和重新编译的时间，更快地反映出代码的修改。而在 Webpack 中，每次修改代码都需要重新构建和编译，耗费一定的时间。
* 构建环境下的优化：尽管 Vite 在开发环境下不进行打包构建，但在生产环境下，它会通过预构建的方式生成高性能的静态资源，以提高页面加载速度。而 Webpack 则通过将所有模块打包成 bundle 文件，进行代码压缩和优化，以及使用各种插件和配置来优化构建结果。

总的来说，Vite 在热更新上比 Webpack 更加快速和精细化，能够在开发过程中提供更好的开发体验和更快的反馈速度。但是，Webpack 在构建环境下有更多的优化和功能，适用于更复杂的项目需求。

以下是 Vite 和 Webpack 在热更新方面的对比表格：

|  |  |  |
| --- | --- | --- |
| 特点 | Vite | Webpack |
| 实时热更新 | 支持模块级别的热更新，即只更新修改的模块，无需刷新整个页面 | 支持文件级别的热更新，修改任何文件都会触发整个应用的重新构建和刷新 |
| 构建速度 | 在开发环境下，利用浏览器原生的模块导入功能，不需要进行打包构建，启动速度更快 | 需要进行打包构建，每次修改代码都需要重新构建和编译，相对较慢 |
| 开发体验 | 提供更好的开发体验，修改代码后快速反馈，无需等待全量构建 | 反馈速度较慢，需要等待每次构建和编译完成 |
| 适用场景 | 适用于中小型项目，追求开发效率的前端项目 | 适用于大型项目，有复杂需求和更多构建优化的前端项目 |

**esbuild和rollup都是vite的基础依赖，那么他们有啥不同?【热度: 129】**

**关键词：esbuild 和 rollup 区别**

esbuild 和 Rollup 都是 Vite 的基础依赖，但它们在 Vite 中担负着不同的角色和任务。

esbuild：**esbuild 是一个快速、可扩展的 JavaScript 打包器，它被用作 Vite 的默认构建工具**。esbuild 的主要任务是将源代码转换为浏览器可以理解的代码，同时还支持压缩、代码分割、按需加载等功能。esbuild 利用其高性能的构建能力，实现了快速的开发服务器和热模块替换。

Rollup：**Rollup 是一个 JavaScript 模块打包工具，也是 Vite 的另一个基础依赖**。在 Vite 中，Rollup 主要用于生产构建阶段。它通过静态分析模块依赖关系，将多个模块打包为一个或多个最终的输出文件。Rollup 支持多种输出格式，如 ES 模块、CommonJS、UMD 等，可以根据项目的需要进行配置。

尽管 esbuild 和 Rollup 都是 Vite 的基础依赖，但它们的分工是不同的。**esbuild 用于开发服务器阶段，通过实时编译和提供模块来实现快速的冷启动和热模块替换。而 Rollup 用于生产构建阶段，将源代码打包为最终可发布的文件，以用于部署到生产环境**。这样的分工使得 Vite 在开发过程中能够快速响应变化，并在构建过程中生成高效的最终输出文件。

**vite编译器有啥特点?【热度: 237】**

**关键词：vite 编译器特点**

Vite 是一个基于现代浏览器原生 ES 模块导入功能的开发工具和构建系统。与传统的打包工具相比，Vite 具有以下几个特点：

* 快速冷启动：Vite 采用了一种新的开发服务器，利用浏览器原生的 ES 模块导入功能，无需提前构建和打包，可以实现快速的冷启动，并在浏览器中按需编译和加载代码。这种特性使得开发时的重新加载速度非常快，提高了开发效率。
* 按需编译：Vite 通过解析导入的模块路径，只编译当前需要的模块，而不是像传统的打包工具一样对整个项目进行全量编译。这种按需编译的方式减少了不必要的重复编译和构建时间，提高了构建速度。
* 零配置：Vite 提供了一种零配置的开发体验，无需繁琐的配置文件，可以快速开始项目的开发。Vite 默认支持常见的前端开发场景，如 Vue、React、TypeScript 等，开发者可以通过简单的配置进行个性化定制。
* 原生 ES 模块支持：Vite 利用浏览器原生的 ES 模块导入功能，可以直接在浏览器中引入 ES 模块，无需经过任何编译和转换，提供了更好的开发体验和更高的性能。
* 插件化：Vite 的构建系统采用了插件化的架构，开发者可以根据需求选择和配置不同的插件来扩展 Vite 的功能。Vite 提供了丰富的插件生态系统，使得开发者可以定制化地满足项目需求。

**vite编译器的组成部分【热度: 335】**

**关键词：vite 编译器组成部分**

Vite 编译器的主要组成部分包括：

* esbuild：一个快速的 JavaScript 打包器，用于在开发阶段进行实时编译。esbuild 提供了快速的冷启动和热模块替换功能，能够极大地加快开发环境的构建速度。
* Rollup：一个强大的 JavaScript 模块打包器，在生产构建阶段使用。Rollup 能够将源代码打包为最终可发布的文件，支持代码分割、Tree Shaking 等优化技术，生成更小、更高效的代码包。
* 前端开发服务器：Vite 还提供了一个内置的开发服务器，用于提供开发环境下的静态文件服务和构建工具集成。这个服务器能够利用 esbuild 实现快速的编译和热模块替换，使开发者在开发过程中可以快速地预览和调试代码。
* 插件系统：Vite 通过插件系统来扩展其功能。开发者可以编写自定义的插件，用于处理特定的文件类型、引入额外的功能或者定制构建过程。插件系统使得 Vite 能够与各种前端框架和工具集成，并提供更灵活的开发体验。

**vite涉及到了哪些底层原理?【热度: 510】**

Vite涉及到以下⼏个底层原理：

* 1. ES模块：Vite使⽤了ES模块来管理和加载模块。ES模块是JavaScript的标准模块系统，相⽐于传统的CommonJS或AMD，ES模块具有更好的静态分析能⼒和更⾼的性能。Vite通过使⽤浏览器原⽣的ES模块加载器，可以实现按需加载和快速构建。
* 2. HTTP/2：Vite借助于现代浏览器的HTTP/2⽀持来实现更⾼效的资源加载。HTTP/2⽀持多路复⽤，可以同时请求多个资源，避免了传统的HTTP/1中的队头阻塞问题，加快了资源加载速度。
* 3. 编译器：Vite使⽤了⾃定义的编译器来处理开发时的模块解析和转换。它能够识别模块的依赖关系，并将模块转换为浏览器可直接执⾏的代码。Vite的编译器⽀持热模块替换（HMR），可以在代码修改时⾃动更新浏览器中的⻚⾯，提⾼开发效率。
* 4. 中间件：Vite使⽤了基于Koa框架的中间件来处理开发服务器。通过中间件，Vite可以拦截和处理开发时的HTTP请求，并根据请求的路径返回相应的模块⽂件。中间件还可以处理各种开发时的特殊需求，如代理API请求、路由转发等。

Vite基于ES模块、HTTP/2、⾃定义编译器和中间件等底层原理，实现了快速的模块加载和开发体验。这些原理的运⽤使得Vite在开发环境下能够提供更快的构建速度和更好的开发体验。

**[Webpack] Webpack vs Vite的核心差异【热度: 620】**

**关键词：Webpack vs Vite 差异**

**构建速度:**

Webpack: Webpack的构建速度相对较慢，尤其在大型项目中，因为它需要分析整个依赖图，进行多次文件扫描和转译。

Vite: Vite以开发模式下的极速构建著称。它利用ES模块的特性，只构建正在编辑的文件，而不是整个项目。这使得它在开发环境下几乎是即时的。

**开发模式:**

Webpack: Webpack通常使用热模块替换（HMR）来实现快速开发模式，但配置相对复杂。

Vite: Vite的开发模式非常轻量且快速，支持HMR，但无需额外配置，因为它默认支持。

**配置复杂度:**

Webpack: Webpack的配置相对复杂，特别是在处理不同类型的资源和加载器时。

Vite: Vite鼓励零配置，使得项目起步非常简单，但同时也支持自定义配置，使其适用于复杂项目。

**插件生态:**

Webpack: Webpack拥有庞大的插件生态系统，适用于各种不同的需求。

Vite: Vite也有相当数量的插件，但相对较小，因为它的开发模式和构建方式减少了对一些传统插件的需求。

**编译方式:**

Webpack: Webpack使用了多种加载器和插件来处理不同类型的资源，如JavaScript、CSS、图片等。

Vite: Vite利用ES模块原生支持，使用原生浏览器导入来处理模块，不需要大规模的编译和打包。

**应用场景:**

Webpack: 适用于复杂的大型项目，特别是需要大量自定义配置和复杂构建管道的项目。

Vite: 更适用于小到中型项目，或者需要快速开发原型和小型应用的场景。

**打包原理:**

Webpack: Webpack的打包原理是将所有资源打包成一个或多个bundle文件，通常是一个JavaScript文件。

Vite: Vite的打包原理是保持开发时的模块化结构，使用浏览器原生的导入机制，在生产环境中进行代码分割和优化。

**优缺点:**

**Webpack:**

优点：灵活、强大、适用于复杂场景、庞大的插件生态。

缺点：构建速度较慢、配置复杂、开发体验不如Vite流畅。

**Vite:**

优点：极快的开发构建速度、零配置启动、原生ES模块支持、适用于小型项目和快速原型开发。

缺点：插件生态相对较小、不太适用于复杂大型项目。

**[Vue]响应式为何要从 Object.defineProperty改为proxy?【热度: 352】**

Vue在早期版本中使⽤了 Object.defineProperty 来实现响应式系统。但是，在 Object.defineProperty 中存在⼀些限制和局限性，导致在某些场景下⽆法完全满⾜需求。因此，Vue在最新的版本中引⼊了 Proxy 来替代 Object.defineProperty 。

以下是⼀些 Proxy 相对于 Object.defineProperty 的优势：

* 1. 功能更强⼤： Proxy 可以代理整个对象，⽽ Object.defineProperty 只能对已存在的属性进⾏拦截。使⽤ Proxy 可以在对象级别上进⾏拦截、代理、验证等操作。
* 2. 更易于使⽤和理解： Proxy 提供了⼀组更直观和易于理解的API，使开发者可以更容易地创建和管理代理。
* 3. 性能优化： Proxy 针对属性的访问和修改都提供了更佳的性能优化。⽽Object.defineProperty 在拦截属性访问和修改时会有⼀定的性能损耗。
* 4. 更好的嵌套⽀持： Proxy 可以代理嵌套对象的属性，⽽ Object.defineProperty 只能对顶层对象的属性进⾏拦截。

总的来说， Proxy 相对于 Object.defineProperty 在功能上更强⼤、使⽤更便捷、性能更优，并且在更复杂的场景下也能提供更好的⽀持。因此，Vue在新版本中选择了使⽤Proxy 来实现响应式系统

**将静态资源缓存在本地的方式有哪些?【热度: 584)]**

**关键词：静态资源缓存本地**

浏览器可以使用以下几种方式将前端静态资源缓存在本地：

* HTTP缓存：浏览器通过设置HTTP响应头中的Cache-Control或Expires字段来指定资源的缓存策略。常见的缓存策略有：no-cache（每次都请求服务器进行验证）、no-store（不缓存资源）、max-age（设置资源缓存的最大时间）等。浏览器根据这些缓存策略来决定是否将资源缓存在本地。
* ETag/If-None-Match：服务器可以通过在响应头中添加ETag字段，用于标识资源的版本号。当浏览器再次请求资源时，会将上次请求返回的ETag值通过If-None-Match字段发送给服务器，由服务器判断资源是否发生了变化。如果资源未发生变化，服务器会返回304 Not Modified状态码，浏览器则直接使用本地缓存的资源。
* Last-Modified/If-Modified-Since：服务器可以通过在响应头中添加Last-Modified字段，用于标识资源的最后修改时间。浏览器再次请求资源时，会将上次请求返回的Last-Modified值通过If-Modified-Since字段发送给服务器。服务器根据资源的最后修改时间判断资源是否发生了变化，如果未发生变化，则返回304 Not Modified状态码，浏览器使用本地缓存的资源。
* Service Worker缓存：使用Service Worker可以将前端资源缓存在浏览器的Service Worker缓存中。Service Worker是运行在浏览器后台的脚本，它可以拦截和处理网络请求，因此可以将前端资源缓存起来，并在离线状态下提供缓存的资源。
* LocalStorage或IndexedDB：对于一些小的静态资源，可以将其存储在浏览器的LocalStorage或IndexedDB中。这些存储方式是浏览器提供的本地存储机制，可以将数据以键值对的形式存储在浏览器中，从而实现缓存的效果。

**如何将静态资源缓存在 LocalStorage或IndexedDB**

以下是一个使用LocalStorage将静态资源缓存的示例代码：

// 定义一个数组，包含需要缓存的静态资源的URL

var resources = [

'https://example.com/css/style.css',

'https://example.com/js/main.js',

'https://example.com/images/logo.png'

];

// 遍历资源数组，将资源请求并存储在LocalStorage中

resources.forEach(function(url) {

// 发起资源请求

fetch(url)

.then(function(response) {

// 检查请求是否成功

if (!response.ok) {

throw new Error('Request failed: ' + response.status);

}

// 将响应数据存储在LocalStorage中

return response.text();

})

.then(function(data) {

// 将资源数据存储在LocalStorage中，以URL作为键名

localStorage.setItem(url, data);

console.log('Resource cached: ' + url);

})

.catch(function(error) {

console.error(error);

});

});

以下是一个使用IndexedDB将静态资源缓存的示例代码：

// 打开或创建一个IndexedDB数据库

var request = indexedDB.open('myDatabase', 1);

// 创建或更新数据库的对象存储空间

request.onupgradeneeded = function(event) {

var db = event.target.result;

var objectStore = db.createObjectStore('resources', { keyPath: 'url' });

objectStore.createIndex('url', 'url', { unique: true });

};

// 成功打开数据库后，将资源请求并存储在IndexedDB中

request.onsuccess = function(event) {

var db = event.target.result;

var transaction = db.transaction('resources', 'readwrite');

var objectStore = transaction.objectStore('resources');

resources.forEach(function(url) {

// 发起资源请求

fetch(url)

.then(function(response) {

// 检查请求是否成功

if (!response.ok) {

throw new Error('Request failed: ' + response.status);

}

// 将响应数据存储在IndexedDB中

return response.blob();

})

.then(function(data) {

// 创建一个资源对象，以URL作为键名

var resource = { url: url, data: data };

// 将资源对象存储在IndexedDB中

objectStore.put(resource);

console.log('Resource cached: ' + url);

})

.catch(function(error) {

console.error(error);

});

});

// 完成事务

transaction.oncomplete = function() {

console.log('All resources cached in IndexedDB.');

};

transaction.onerror = function(event) {

console.error('Transaction error:', event.target.error);

};

};

以上代码仅为示例，实际应用中需要根据具体的需求进行相应的优化和错误处理。

**为什么现代前端应用需要打包工具进行打包编译?【热度:1,588】**

现代前端应⽤需要打包⼯具进⾏打包编译的主要原因有以下⼏点：

1. 模块化管理：现代前端应⽤通常采⽤模块化的开发⽅式，将代码划分为多个模块，每个模块具有独⽴的功能和依赖关系。打包⼯具可以将这些模块进⾏分析，将它们打包成⼀个或多个静态⽂件，⽅便管理和维护。

2. 解决浏览器兼容性问题：不同的浏览器对于JavaScript和CSS的⽀持程度不同，⽽且随着新特性的不断出现，旧版浏览器可能⽆法完全⽀持。打包⼯具可以通过转译、压缩和兼容性处理等⼿段，将当前前端代码转化为浏览器可识别和运⾏的代码，解决兼容性问题。

3. 静态资源处理和优化：现代前端应⽤涉及⼤量的静态资源，如图⽚、字体等。打包⼯具可以对这些资源进⾏处理和优化，如图⽚压缩、字体⽂件打包等，以减⼩资源⽂件的体积，提⾼⻚⾯的加载速度和性能。

4. 代码分割和按需加载：打包⼯具可以将应⽤程序拆分成多个⼩块，实现代码分割和按需加载。这样可以实现懒加载，只在需要时加载特定的代码块，提⾼⻚⾯的加载速度。

5. 开发环境⽀持：打包⼯具通常提供开发服务器和热模块替换（HMR）等功能，⽅便开发⼈员进⾏开发和调试。开发服务器可以实时预览代码变化，HMR可以在修改代码后只替换修改的部分，⽽不是整个⻚⾯刷新，提⾼开发效率。

6. 提升性能：打包⼯具可以通过代码优化、压缩和混淆等技术⼿段，减⼩⽂件体积，提升应⽤程序的加载速度和执⾏效率。

7. ⽀持多种前端技术：现代前端应⽤通常使⽤多种前端技术和语⾔，如JavaScript、CSS、TypeScript、Sass等。打包⼯具可以集成这些技术，并提供相应的编译、转译和处理功能，使开发⼈员能够更轻松地使⽤这些技术。

8. ⾃动化⼯作流程：打包⼯具可以配合其他构建⼯具和⾃动化任务运⾏器，如Webpack配合Grunt或Gulp，实现⾃动化的构建和部署流程。这可以减少⼿动操作，提⾼开发效率和代码质量。

9. 第三⽅库管理：现代前端应⽤通常使⽤⼤量的第三⽅库和框架，这些库可能包含多个⽂件和依赖关系。打包⼯具可以⾃动管理这些库的依赖关系，并将它们打包为单个⽂件，减少⽹络请求和提⾼代码的可维护性。

10. ⾼度可定制化：打包⼯具通常提供丰富的插件和配置选项，允许开发⼈员根据项⽬需求进⾏定制。可以灵活配置打包过程中的各种处理和优化⽅式，以满⾜项⽬的具体需求。

总结-现代前端应⽤需要打包⼯具进⾏打包编译的原因是为了：实现模块化管理、解决兼容性问题、静态资源处理和优化、代码分割和按需加载、开发环境⽀持、性能提升、多技术⽀持、⾃动化⼯作流程、第三⽅库管理和可定制化等⽅⾯的需求。

**如何理解前端工程化?**

前端工程化是指将前端开发过程中的各种工具、技术和流程进行规范化和自动化，以提高前端开发效率、代码质量和团队协作能力的一种方法。

前端工程化主要涵盖以下几个方面：

* 代码管理：使用版本控制系统（如Git）对前端项目进行代码管理，实现代码的版本控制、协作开发和代码回滚等功能。
* 代码规范：制定和遵循统一的前端代码规范，包括缩进、命名、注释等，以提高代码的可读性和可维护性。
* 模块化开发：使用模块化的开发方式，将前端代码拆分为独立的模块，以便于复用和维护。
* 构建工具：使用构建工具（如Webpack、Gulp）对前端代码进行构建、打包、压缩和优化等处理，提高开发效率和应用性能。
* 自动化测试：编写自动化测试用例，使用测试工具（如Jest、Mocha）进行自动化测试，以确保代码质量和功能的稳定性。
* 持续集成和部署：使用持续集成工具（如Jenkins、Travis CI）将代码自动构建、测试和部署到服务器，加快代码交付速度和减少人工操作。
* 性能优化：使用性能优化工具（如Webpack、Lighthouse）来分析和改进前端应用程序的性能，包括文件加载、资源压缩、缓存策略等方面。
* 文档和知识管理：编写和维护开发文档、API 文档以及团队知识库，方便团队成员之间的沟通和知识分享。

通过前端工程化的实践，可以减少重复劳动、提高代码质量、加速开发流程和降低维护成本，从而更好地满足项目需求和提升用户体验。

**如何理解前端架构?**

前端架构是指在开发前端应用程序时，为了提高开发效率、代码可维护性和可扩展性，将前端代码组织和设计的一种架构体系。它涵盖了前端项目的整体结构、代码组织方式、技术选型、模块化开发、数据管理、状态管理、网络请求、路由管理等方面的规划和设计。

前端架构的目标是使前端开发更加规范、高效和可持续。良好的前端架构可以帮助团队成员更好地协作、降低维护成本、提高开发效率、减少代码冗余和bug等。

以下是一些常见的前端架构概念和思想：

* 分层架构：将前端应用程序划分为不同的层次，如视图层、业务逻辑层、数据层等，以实现各个层次的解耦和职责清晰。
* 组件化开发：将界面拆分为独立的可复用组件，通过组合不同的组件来构建页面，提高代码的可维护性和可复用性。
* 模块化开发：将代码按照功能或业务模块进行拆分，每个模块都有独立的职责和功能，便于团队协作和代码维护。
* 数据管理和状态管理：使用状态管理库（如Redux、Vuex）来管理应用程序的状态和数据流，使得数据的变化和传递更加可控和可预测。
* 路由管理：使用路由库（如React Router、Vue Router）来管理前端路由，实现页面之间的跳转和导航。
* 统一风格和规范：制定和遵循统一的编码规范、命名规范、目录结构等，以便于不同开发者之间的协作和代码的统一性。
* 自动化构建和部署：使用构建工具（如Webpack、Rollup）和自动化部署工具（如Jenkins、Travis CI）来提高开发效率和代码交付速度。
* 性能优化和项目优化：通过代码分割、懒加载、缓存、CDN 加速、前端性能监控等手段，提升应用程序的性能和用户体验。

前端架构是一种组织和规划前端代码的方法论，旨在提高前端开发的效率和质量，同时也要根据具体项目的需求和规模来选择和适配合适的架构方案。

**前端架构和前端工程化有什么区别?**

区别

前端架构和前端工程化是两个不同的概念，但它们之间有一些相互关联的特点。

**前端架构**是指在前端开发中，对整个前端应用程序的组织结构、模块划分、框架选择等方面的设计和规划。前端架构的目标是为了提高代码的可维护性、可扩展性和可重用性，以及优化前端应用程序的性能和用户体验。常见的前端架构包括MVC（Model-View-Controller）、MVVM（Model-View-ViewModel）等。

**前端工程化**是指使用各种工具、技术和流程对前端开发过程进行管理和优化，以提高开发效率、代码质量和团队协作能力。前端工程化的目标是通过规范化和自动化的方式，解决前端开发中的重复劳动、低效率、代码质量不稳定等问题。前端工程化包括代码管理、代码规范、模块化开发、构建工具、自动化测试、持续集成和部署、性能优化、文档和知识管理等方面。

虽然前端架构和前端工程化是两个不同的概念，但它们之间存在一些相似的目标和方法。**前端架构关注的是前端应用程序的结构和设计，而前端工程化关注的是前端开发的流程和工具的使用。前端架构可以通过前端工程化的方式实现，而前端工程化可以提供支持和保障，以实现良好的前端架构。**

**前端基建涉及到哪些方面【热度: 5,782】**

**关键词：前端工程化建设、前端基建**

前端基建是指在前端开发过程中，为提高开发效率、代码质量和团队协作而构建的一些基础设施和工具。下面是前端基建可以做的一些事情：

**脚手架工具**

开发和维护一个通用的脚手架工具，可以帮助团队快速初始化项目结构、配置构建工具、集成常用的开发依赖等。

**组件库**

开发和维护一个内部的组件库，包含常用的UI组件、业务组件等，提供给团队成员复用，减少重复开发的工作量。

**构建工具和打包工具**

搭建和维护一套完善的构建和打包工具链，包括使用Webpack、Parcel等工具进行代码的压缩、合并、打包等工具，优化前端资源加载和性能。

**自动化测试工具**

引入自动化测试工具，如Jest、Mocha等，编写和维护测试用例，进行单元测试、集成测试、UI测试等，提高代码质量和可靠性。

**文档工具**

使用工具如JSDoc、Swagger等，生成项目的API文档、接口文档等，方便团队成员查阅和维护。

**Git工作流**

制定和规范团队的Git工作流程，使用版本控制工具管理代码，方便团队协作和代码回退。

**性能监控和优化**

引入性能监控工具，如Lighthouse、Web Vitals等，对项目进行性能分析，优化网页加载速度、响应时间等。

**工程化规范**

制定并推广团队的代码规范、目录结构规范等，提高代码的可读性、可维护性和可扩展性。

**持续集成和部署**

搭建持续集成和部署系统，如Jenkins、Travis CI等，实现代码的自动构建、测试和部署，提高开发效率和代码质量。

**项目文档和知识库**

建立一个内部的项目文档和知识库，记录项目的技术细节、开发经验、常见问题等，方便团队成员查阅和学习。

**代码质量工具**

引入代码质量工具，如ESLint、Prettier等，对代码进行静态分析和格式化，提高代码的一致性和可读性。

**国际化支持**

为项目添加国际化支持，可以通过引入国际化库，如i18next、vue-i18n等，实现多语言的切换和管理。

**错误监控和日志收集**

引入错误监控工具，如Sentry、Bugsnag等，实时监控前端错误，并收集错误日志，方便进行问题排查和修复。

**前端性能优化工具**

使用工具如WebPageTest、Chrome DevTools等，对项目进行性能分析和优化，提高页面加载速度、响应时间等。

**缓存管理**

考虑合理利用浏览器缓存和服务端缓存，减少网络请求，提升用户访问速度和体验。

**移动端适配**

针对移动端设备，采用响应式设计或使用CSS媒体查询等技术，实现移动端适配，保证页面在不同尺寸的设备上有良好的显示效果。

**安全防护**

对项目进行安全审计，使用安全防护工具，如CSP（Content Security Policy）、XSS过滤等，保护网站免受常见的安全攻击。

**性能优化指标监控**

监控和分析关键的性能指标，如页面加载时间、首次渲染时间、交互响应时间等，以便及时发现和解决性能问题。

**前端日志分析**

使用日志分析工具，如ELK（Elasticsearch、Logstash、Kibana）等，对前端日志进行收集和分析，了解用户行为和页面异常情况。

**跨平台开发**

考虑使用跨平台开发框架，如React Native、Flutter等，实现一套代码在多个平台上复用，提高开发效率。

**编辑器配置和插件**

为团队提供统一的编辑器配置文件，包括代码格式化、语法高亮、代码自动补全等，并推荐常用的编辑器插件，提高开发效率。

**文档生成工具**

使用工具如Docusaurus、VuePress等，为项目生成漂亮的文档网站，方便团队成员查阅和维护项目文档。

**Mock数据和接口管理**

搭建一个Mock服务器，用于模拟后端接口数据，方便前端开发和测试，同时可以考虑使用接口管理工具，如Swagger等，方便接口的定义和调试。

**前端监控和统计**

引入前端监控工具，如Google Analytics、百度统计等，收集用户访问数据和行为信息，用于分析和优化用户体验。

**移动端调试工具**

使用工具如Eruda、VConsole等，帮助在移动端设备上进行调试和错误排查，提高开发效率。

**自动化部署**

配置自动化部署流程，将项目的代码自动部署到服务器或云平台，减少人工操作，提高发布效率和稳定性。

**前端团队协作工具**

使用团队协作工具，如GitLab、Bitbucket等，提供代码托管、项目管理、任务分配和团队沟通等功能，增强团队协作效率。

**前端培训和知识分享**

组织定期的前端培训和技术分享会，让团队成员相互学习和交流，推动技术的共享和提升。

**客户端性能优化**

针对移动端应用，可以使用工具如React Native Performance、Weex等，进行客户端性能优化，提高应用的响应速度和流畅度。

**技术选型和评估**

定期评估和研究前端技术的发展趋势，选择适用的技术栈和框架，以保持项目的竞争力和可持续发展。

**统一的状态管理**

引入状态管理工具，如Redux、Vuex等，帮助团队管理前端应用的状态，提高代码的可维护性和可扩展性。

**前端日志记录**

引入前端日志记录工具，如log4javascript、logrocket等，记录前端应用的运行日志，方便排查和解决问题。

**前端代码扫描**

使用静态代码扫描工具，如SonarQube、CodeClimate等，对前端代码进行扫描和分析，发现潜在的问题和漏洞。

**前端数据可视化**

使用数据可视化工具，如ECharts、Chart.js等，将数据以图表或图形的形式展示，增强数据的可理解性和可视化效果。

**前端容灾和故障处理**

制定容灾方案和故障处理流程，对前端应用进行监控和预警，及时处理和恢复故障，提高系统的可靠性和稳定性。

**前端安全加固**

对前端应用进行安全加固，如防止XSS攻击、CSRF攻击、数据加密等，保护用户数据的安全性和隐私。

**前端版本管理**

建立前端代码的版本管理机制，使用工具如Git、SVN等，管理和追踪代码的变更，方便团队成员之间的协作和版本控制。

**前端数据缓存**

考虑使用Local Storage、Session Storage等技术，对一些频繁使用的数据进行缓存，提高应用的性能和用户体验。

**前端代码分割**

使用代码分割技术，如Webpack的动态导入（Dynamic Import），将代码按需加载，减少初始加载的资源大小，提高页面加载速度。

**前端性能监测工具**

使用性能监测工具，如WebPageTest、GTmetrix等，监测前端应用的性能指标，如页面加载时间、资源加载时间等，进行性能优化。

**如何搭建一套灰度系统?【热度: 1,226】**

**关键词：灰度上线**

这个是一个非常复杂的话题， 没法直接给出答案， 进提供一些实现的思路：

**什么是灰度**

灰度系统可以把流量划分成多份，一份走新版本代码，一份走老版本代码。

而且灰度系统支持设置流量的比例，比如可以把走新版本代码的流程设置为 5%，没啥问题再放到 10%，50%，最后放到 100% 全量。

这样可以把出现问题的影响降到最低。

而且灰度系统不止这一个用途，比如产品不确定某些改动是不是要的，就要做 AB 实验，也就是要把流量分成两份，一份走 A 版本代码，一份走 B 版本代码。

**实现思路**

* 后端支持：灰度上线需要后端的支持，通过后端的灰度发布控制，可以将不同版本的前端应用分配给不同用户。
* 搭建网关层： 支持一部分用户分发到 A 版本， 一部分用户分发到 B 版本 （通常使用 nginx 搭建）。
* 版本管控机制： 使用版本控制系统（如Git、package.version、hash version 等）来管理不同版本的前端应用代码。在灰度上线时，可以根据需要切换到特定的版本。
* 动态路由：通过动态路由配置，将用户请求导向不同版本的前端应用。例如，可以使用Nginx或其他反向代理服务器来实现动态路由。
* 流量染色：使用Cookie或Session来控制用户的灰度版本访问。可以通过设置不同的Cookie值或Session标记，将用户分配到不同的灰度版本。
* 更复杂的漏量配置： 例如要根据部门、权限、角色等方式来开放灰度；可以使用让用户访问应用的时候， 查询其权限和角色， 根据权限和角色来分发不同的页面路由。

**如何理解研发流程和研发效率，如何保障研发效率**

**保障研发效率**

研发流程指的是从需求调研、设计、开发、测试、发布、维护等一系列环节组成的整个研发过程。它是实现软件产品的必经之路，可以帮助团队规范化、标准化研发流程，提高研发效率、降低研发成本、提高产品质量和用户满意度。

研发效率指的是在研发流程中，完成同样的工作所需要的时间和成本，也就是研发效率越高，就可以在同样的时间内完成更多的工作，并在更短的时间内推出产品，提高团队的竞争力和市场占有率。

**保障研发效率可以从以下几个方面入手：**

* 优化研发流程：通过优化整个研发流程，缩短产品上线周期，降低研发成本，提高产品质量和用户满意度。
* 确定明确的目标和任务：团队需要清楚地了解产品的目标和任务，明确每个人的责任和任务，以便更好地完成工作。
* 提供优秀的工具和环境：为团队提供高效的工具和优秀的开发环境，帮助开发者更好地完成工作，提高工作效率。
* 加强团队协作和沟通：团队成员之间需要建立良好的沟通和协作机制，通过有效的沟通和协作，提高工作效率和质量。
* 持续学习和提高技能：团队成员需要不断学习新知识和提高技能，接受新技术，以便更好地完成工作，提高研发效率。

**具体讲一下如何优化研发流程**

**优化研发流程**可以从以下几个方面入手：

* 需求管理：建立明确的需求管理机制，包括需求收集、需求筛选、需求优化、需求变更管理等，以确保需求的准确性、完整性、一致性和可追溯性。
* 设计管理：建立明确的设计管理机制，包括设计评审、设计文档管理、设计变更管理等，以确保设计的合理性、可行性、可维护性和可扩展性。
* 开发管理：建立高效的开发管理机制，包括任务分配、代码管理、代码审查、编码规范管理等，以确保开发的效率、质量和一致性。
* 测试管理：建立严格的测试管理机制，包括测试计划、测试用例管理、测试环境管理、缺陷管理等，以确保产品的质量和稳定性。
* 发布管理：建立有效的发布管理机制，包括版本控制、发布计划、发布测试、发布文档等，以确保产品的稳定性和用户满意度。
* 迭代管理：建立迭代管理机制，通过不断的迭代和优化，提高产品的质量和用户满意度，保持团队的创新和活力。
* 数据分析：建立数据分析机制，通过数据分析和用户反馈，不断优化产品和流程，提高团队的效率和竞争力。

优化研发流程需要建立标准化、规范化的流程和管理机制，不断优化和改善流程，并建立有效的沟通和协作机制，以提高团队的效率和产品质量，满足用户的需求和期望。

**如何保证上述机制能够正确推进下去？**

为了确保上述机制能够正确推进下去，可以采取以下措施：

* 建立标准化的流程和管理机制，并将其纳入团队的日常工作流程中，确保每个团队成员都能够遵循。
* 建立监控和评估机制，对各项机制进行定期的检查、评估和反馈，并根据实际情况进行调整和改进。
* 建立协作和沟通机制，促进团队成员之间的协作和沟通，确保各项机制能够得到正确的执行和落实。
* 建立培训和学习机制，定期培训团队成员，提高其专业技能和管理能力，增强其执行机制的能力和信心。
* 采用项目管理工具和平台，对各项机制进行集中管理和监控，确保团队成员能够及时获取和共享必要的信息和资源。

要保证各项机制能够正确推进下去，需要建立完善的流程和管理机制，并通过监控、评估、协作、沟通、培训和技术支持等措施，确保团队成员能够正确执行和落实机制，从而提高团队的效率和竞争力。

**确定明确需求目标和需求内容**

首要任务是对需求进行拆解拆分，需要进行需求拆分来更好地管理和实现项目目标。以下是对大型项目进行需求拆分的建议：

对于一个大型项目，需要进行需求拆分来更好地管理和实现项目目标。以下是**对大型项目进行需求拆分的建议**：

* 确定项目范围：明确项目的目标和范围，明确项目所需的主要功能和特性。
* 列出项目需求：将项目所需的各种需求列出来，包括功能需求、性能需求、安全需求、可靠性需求等。
* 进行需求分类：将需求进行分类，可以按照功能、用户、业务流程等方式进行分类。
* 制定需求文档：根据需求分类，制定详细的需求文档，包括需求描述、优先级和验收标准等。
* 制定项目计划：根据需求文档，制定项目计划，包括任务分解、时间安排和资源分配等。
* 协同开发：在开发过程中，需要协同进行开发，进行需求变更和调整。
* 进行验收和测试：在项目结束时，进行验收和测试，确保项目满足客户需求和要求。

**SSR了解多少【热度:486】**

**关键词：SSR 原理、SSR 实现**

**SSR 原理是啥**

服务器端渲染（Server-Side Rendering，SSR）是一种前端渲染方式，其核心原理是在服务器端将动态生成的 HTML 页面发送给客户端，以便客户端在接收到页面时直接渲染显示，而不是在客户端使用 JavaScript 动态生成页面。

核心原理如下：

* 客户端发起请求：当用户访问一个 SSR 应用的页面时，客户端会向服务器发起请求。
* 服务器处理请求：服务器接收到请求后，根据请求的路径和参数，获取对应的数据。
* 数据获取和页面渲染：在服务器端，通过调用后端数据接口或其他数据源获取页面所需的数据。获取到数据后，服务器使用模板引擎或渲染框架将数据填充到页面模板中，生成完整的 HTML 页面。
* HTML 页面返回给客户端：服务器将生成的 HTML 页面作为响应返回给客户端。
* 客户端渲染：客户端接收到服务器返回的 HTML 页面后，直接渲染显示页面内容。由于服务器已经将数据填充到了页面中，客户端无需再进行数据获取和页面渲染的过程，提升了页面的加载速度和用户体验。

SSR 的核心原理是在服务器端生成完整的 HTML 页面，并将其发送给客户端，使客户端能够更快地显示页面内容。相比于传统的客户端渲染（CSR），SSR 可以改善首次加载时的白屏时间和搜索引擎抓取等方面的问题。同时，SSR 也可以更好地支持 SEO（搜索引擎优化）和提供更好的性能体验给用户。

**实现方案**

前端实现服务器端渲染（SSR）的方案有以下几种：

* 基于 Node.js 的框架：使用 Node.js 的框架（如Express、Koa、Nest.js等）来构建服务器端应用程序，并在服务器端进行页面渲染。通过在服务器上运行 JavaScript 代码，将渲染好的页面直接返回给客户端。
* 框架提供的 SSR 功能：一些前端框架（如Next.js、Nuxt.js、Angular Universal等）提供了内置的服务器端渲染功能，可以更方便地实现 SSR。这些框架会负责处理路由、数据预取和页面渲染等工作，并将渲染好的页面返回给客户端。
* 预渲染：使用预渲染技术将静态页面提前生成，并部署到服务器上。在用户请求页面时，直接返回预渲染好的 HTML 页面，然后再由客户端接管页面的交互。这种方式适用于内容不经常变动或不需要动态数据的页面。
* 后端代理：通过将前端应用程序的请求代理到服务器端，然后在服务器端进行页面渲染，并将渲染好的页面返回给客户端。这种方式适用于在现有的后端服务中添加 SSR 功能，而无需重写整个应用程序。

需要根据具体的项目需求、技术栈和框架选择合适的 SSR 实现方案。每种方案都有其优点和限制，综合考虑性能、开发体验、部署成本和维护复杂度等因素来做出决策。

**CSR、SSR、SSG、NSR、ESR、ISR 都是啥？【热度: 3,492】**

CSR、SSR、SSG、NSR、ESR、ISR 都是啥？

根据不同的构建、渲染过程有不同的优劣势和适用情况。

* 现代 UI 库加持下常用的 CSR、
* 具有更好 SEO 效果的 SSR (SPR)、
* 转换思路主打构建时生成的 SSG、
* 大架构视野之上的 ISR、DPR，
* 还有更少听到的 NSR、ESR。

**CSR(Client Side Rendering)**

页面托管服务器只需要对页面的访问请求响应一个如下的空页面

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<!-- metas -->

<title></title>

<link rel="shortcut icon" href="xxx.png" />

<link rel="stylesheet" href="xxx.css" />

</head>

<body>

<div id="root"><!-- page content --></div>

<script src="xxx/filterXss.min.js"></script>

<script src="xxx/x.chunk.js"></script>

<script src="xxx/main.chunk.js"></script>

</body>

</html>

页面中留出一个用于填充渲染内容的视图节点 (div#root)，并插入指向项目编译压缩后的

* JS Bundle 文件的 script 节点
* 指向 CSS 文件的 link.stylesheet 节点等。

浏览器接收到这样的文档响应之后，会根据文档内的链接加载脚本与样式资源，并完成以下几方面主要工作：

1. 执行脚本
2. 进行网络访问以获取在线数据
3. 使用 DOM API 更新页面结构
4. 绑定交互事件
5. 注入样式

以此完成整个渲染过程。

CSR 模式有以下几方面优点：

* UI 库支持
* 前后端分离
* 服务器负担轻

**SSR (Server Side Rendering)**

SSR 的概念，即与 CSR 相对地，在服务端完成大部分渲染工作， 服务器在响应站点访问请求的时候，就已经渲染好可供呈现的页面。

像 React、Vue 这样的 UI 生态巨头，其实都有一个关键的 Virtual DOM (or VDOM) 概念,先自己建模处理视图表现与更新、再批量调 DOM API 完成视图渲染更新。这就带来了一种 SSR 方案：

VDOM 是自建模型，是一种抽象的嵌套数据结构，也就可以在 Node 环境（或者说一切服务端环境）下跑起来，把原来的视图代码拿来在服务端跑，通过 VDOM 维护，再在最后拼接好字符串作为页面响应，生成文档作为响应页面，此时的页面内容已经基本生成完毕，把逻辑代码、样式代码附上，则可以实现完整的、可呈现页面的响应。

**SSR优点**

* 呈现速度和用户体验佳
* SEO 友好

**SSR缺点**

* 引入成本高
* 将视图渲染的工作交给了服务器做，引入了新的概念和技术栈（如 Node）
* 响应时间长
* SSR 在完成访问响应的时候需要做更多的计算和生成工作
* 关键指标 TTFB (Time To First Byte) 将变得更大
* 首屏交互不佳
* 虽然 SSR 可以让页面请求响应后更快在浏览器上渲染出来
* 但在首帧出现，需要客户端加载激活的逻辑代码（如事件绑定）还没有初始化完毕的时候，其实是不可交互的状态

**SSR-React 原理**

VDOM

同构

双端对比

**几大概念：**

VDOM

同构

双端对比

renderToString()

renderToStaticMarkup()

ReactDOMServer.renderToStaticMarkup(element)

仅仅是为了将组件渲染为html字符串，不会带有data-react-checksum属性

**SPR (Serverless Pre-Rendering)**

无服务预渲染，这是 Serverless 话题之下的一项渲染技术。SPR 是指在 SSR 架构下通过预渲染与缓存能力，将部分页面转化为静态页面，以避免其在服务器接收到请求的时候频繁被渲染的能力，同时一些框架还支持设置静态资源过期时间，以确保这部分“静态页面”也能有一定的即时性。

**SSG (Static Site Generation)**

它与 CSR 一样，只需要页面托管，不需要真正编写并部署服务端，页面资源在编译完成部署之前就已经确定；

但它又与 SSR 一样，属于一种 Prerender 预渲染操作，即在用户浏览器得到页面响应之前，页面内容和结构就已经渲染好了。

当然形式和特征来看，它更接近 SSR。

SSG 模式，把原本日益动态化、交互性增强的页面，变成了大部分已经填充好，托管在页面服务 / CDN 上的静态页面

**NSR (Native Side Rendering)**

Native 就是客户端，万物皆可分布式，可以理解为这就是一种分布式的 SSR，不过这里的渲染工作交给了客户端去做而不是远端服务器。在用户即将访问页面的上级页面预取页面数据，由客户端缓存 HTML 结构，以达到用户真正访问时快速响应的效果。

NSR 见于各种移动端 + Webview 的 Hybrid 场景，是需要页面与客户端研发协作的一种优化手段。

**ESR (Edge Side Rendering)**

Edge 就是边缘，类比前面的各种 XSR，ESR 就是将渲染工作交给边缘服务器节点，常见的就是 CDN 的边缘节点。这个方案主打的是边缘节点相比核心服务器与用户的距离优势，利用了 CDN 分级缓存的概念，渲染和内容填充也可以是分级进行并缓存下来的。

ESR 之下静态内容与动态内容是分流的，

1. 边缘 CDN 节点可以将静态页面内容先响应给用户
2. 然后再自己发起动态内容请求，得到核心服务器响应之后再返回给用户

是在大型网络架构下非常极致的一种优化，但这也就依赖更庞大的技术基建体系了。

**ISR (Incremental Site Rendering)**

增量式网站渲染，就是对待页面内容小刀切，有更细的差异化渲染粒度，能渐进、分层地进行渲染。

常见的选择是：

* 对于重要页面如首屏、访问量较大的直接落地页，进行预渲染并添加缓存，保证最佳的访问性能；
* 对于次要页面，则确保有兜底内容可以即时 fallback，再将其实时数据的渲染留到 CSR 层次完成，同时触发异步缓存更新。

对于“异步缓存更新”，则需要提到一个常见的内容缓存策略：Stale While Revalidate，CDN 对于数据请求始终首先响应缓存内容，如果这份内容已经过期，则在响应之后再触发异步更新——这也是对于次要元素或页面的缓存处理方式。

## 幽灵依赖是什么?

"幽灵依赖"（Ghost Dependency）是指在项目的node\_modules目录中存在但未被实际使用的依赖包。

在使用 npm 或者其他包管理工具安装依赖包时，有时会出现安装了一些不需要的或者不正确的依赖包的情况。这些依赖包在项目中没有被显式地引用或使用，但仍然存在于node\_modules目录中，占用了项目的存储空间。

幽灵依赖可能会产生以下问题：

* 占用存储空间：未使用的依赖包会增加项目的体积，占用存储空间。对于大型项目或频繁部署的项目来说，这可能会造成不必要的存储资源浪费。
* 增加构建时间：未使用的依赖包可能会增加构建过程中的解析和处理时间，导致构建过程变慢。这会影响开发人员的开发效率和项目的部署速度。
* 潜在的安全风险：未使用的依赖包可能包含漏洞或安全风险，但由于没有使用，可能没有及时更新或修复这些问题，增加了项目的安全隐患。

为了解决幽灵依赖的问题，可以采取以下措施：

* 定期检查依赖：定期检查项目的依赖，识别和删除未使用的依赖包。可以使用工具如npm-check-unused、depcheck等来帮助检测和清理未使用的依赖。
* 精简依赖：审查项目的依赖关系，仅安装和保留必要的依赖包。避免过度依赖，只安装项目所需的模块，减少项目体积和构建时间。
* 更新依赖包：确保项目中使用的依赖包都是最新版本，并及时更新已知的安全漏洞和问题。这可以通过定期检查依赖包的更新和使用工具如npm audit来实现。

通过处理幽灵依赖，可以提高项目的整洁性、性能和安全性，并减少不必要的开销和风险。

### pnpm 是如何解决幽灵依赖问题的

pnpm 是一个基于 npm 的包管理工具，它采用了一种称为"**快速硬链接**（Fast Hard Links）"的机制来解决幽灵依赖问题。

传统的 npm 或 yarn 安装依赖时，每个项目都会在node\_modules目录下创建依赖包的副本。这导致了大量的重复文件，尤其是对于多个项目都使用同一依赖包时。

而 pnpm 通过使用快速硬链接机制，在全局的存储位置（默认为~/.pnpm-store）只保存一份依赖包，而不是为每个项目都复制一份。这样就避免了幽灵依赖问题，减少了存储空间的占用。

当使用 pnpm 安装依赖时，它会在项目的node\_modules目录下创建一个.modules.yaml文件，记录项目所需的依赖包和版本信息。实际的依赖包文件通过硬链接指向全局存储位置中的依赖包。这意味着不同项目之间可以共享相同的依赖包，但每个项目都拥有自己的依赖版本。

通过这种方式，pnpm 解决了幽灵依赖的问题，同时减少了存储空间的使用。它还具有一些其他的优点，如更快的安装速度、更少的网络传输和更好的缓存利用率。

需要注意的是，pnpm 仍然会将项目中的所有依赖安装在node\_modules目录下，但它使用硬链接的方式避免了重复文件的复制，从而解决了幽灵依赖问题。

## 如何组织 monorepo 工程?

[pnpm + workspace + changesets 构建你的 monorepo 工程](https://juejin.cn/post/7098609682519949325)

[现代 Monorepo 工程技术选型，聊聊我的思考](https://juejin.cn/post/7102452341210611720)

[前端工程化之多个项目如何同时高效管理 — monorepo](https://juejin.cn/post/6985336835459252260)

## 如何将JavaScript代码解析成抽象语法树(AST【热度: 1,169】

**关键词：解析为 AST、抽象语法树、AST 词法分析、AST 语法分析**

### 如何将JavaScript代码解析成抽象语法树

要将JavaScript代码解析成抽象语法树（Abstract Syntax Tree，AST），你可以使用工具或库来实现。以下是几种常用的方法：

1.Esprima: Esprima 是一个流行的JavaScript解析器，它可以将JavaScript代码解析成AST。你可以使用它的 JavaScript API 来将代码解析成AST对象。

const esprima = require('esprima');

const code = 'var x = 5;';

const ast = esprima.parseScript(code);

console.log(ast);

2.Acorn: Acorn 是另一个广泛使用的JavaScript解析器，它也可以将JavaScript代码解析成AST。你可以使用它的 JavaScript API 来解析代码并获取AST对象。

const acorn = require('acorn');

const code = 'var x = 5;';

const ast = acorn.parse(code, { ecmaVersion: 2020 });

console.log(ast);

3.Babel: Babel 是一个功能强大的JavaScript编译器，它可以将代码转换为AST，并提供了丰富的插件系统，用于转换和操作AST。你可以使用 Babel 的 API 来解析代码并获取AST对象。

const babelParser = require('@babel/parser');

const code = 'const x = 5;';

const ast = babelParser.parse(code, { sourceType: 'module' });

console.log(ast);

这些工具和库都可以将JavaScript代码解析成AST对象，从而使你能够对代码进行进一步的分析、转换或处理。你可以根据自己的需求选择其中之一，并根据其文档了解更多关于解析选项和AST节点的信息。

### JavaScript代码解析成抽象语法树的原理是什么

JavaScript代码解析成抽象语法树（Abstract Syntax Tree，AST）的过程涉及以下几个主要步骤：

1. 词法分析（Lexical Analysis）：词法分析器（Lexer）将源代码拆分成词法单元（tokens），比如变量名、关键字、操作符、标点符号等。它根据一组定义好的规则（词法规范）来识别和分类这些词法单元。
2. 语法分析（Syntax Analysis）：语法分析器（Parser）接收词法分析器生成的词法单元，并根据语法规则构建AST。语法分析器使用上下文无关文法（Context-Free Grammar）来定义语言的语法规则，它通过递归下降、LR(1) 等算法来处理这些规则，以确定输入是否符合语法规则并生成相应的AST。
3. 构建AST：在语法分析的过程中，语法分析器根据语法规则构建AST。AST是一个树状结构，其中每个节点表示源代码中的一个语法结构，如表达式、语句、函数等。不同节点类型代表不同的语法结构，它们之间通过父子关系和兄弟关系来表示源代码的层次结构和逻辑关联。
4. 后续处理：生成AST后，可以进行进一步的处理和分析。这可能包括语义分析、类型推断、符号解析、代码优化等。这些步骤可以根据具体的需求和工具进行。

总结：将JavaScript代码解析成AST的过程是通过词法分析器将源代码拆分成词法单元，然后语法分析器根据语法规则构建AST。AST提供了对代码结构的抽象表示，便于进一步分析、转换和操作代码。

## [React] react是如何实现页面的快速响应?【热度: 696】

**关键词：react 快速响应实现、react 可中断更新、react IO瓶颈、react CPU瓶颈**

### react 是如何实现快速响应的？

我们日常使用App，浏览网页时，有两类场景会制约快速响应：

当遇到大计算量的操作或者设备性能不足使页面掉帧，导致卡顿。

发送网络请求后，由于需要等待数据返回才能进一步操作导致不能快速响应。

这两类场景可以概括为：

* CPU的瓶颈
* IO的瓶颈

CPU的瓶颈

主流浏览器刷新频率为60Hz，即每（1000ms / 60Hz）16.6ms浏览器刷新一次。

我们知道，JS可以操作DOM，GUI渲染线程与JS线程是互斥的。所以JS脚本执行和浏览器布局、绘制不能同时执行。

在每16.6ms时间内，需要完成如下工作： **JS脚本执行 ----- 样式布局 ----- 样式绘制**

当JS执行时间过长，超出了16.6ms，这次刷新就没有时间执行样式布局和样式绘制了。

比如我们可以通过一个循环， 渲染列表 3000 个组件， 那么这种渲染时间， 就肯定是远超过 16.6 ms 的， 页面就会感觉到卡顿。

如何解决这个问题呢？

答案是：**在浏览器每一帧的时间中，预留一些时间给JS线程，React利用这部分时间更新组件（可以看到，在源码中，预留的初始时间是5ms）**。

源码位置：

[https://github.com/facebook/react/blob/1fb18e22ae66fdb1dc127347e169e73948778e5a/packages/scheduler/src/forks/SchedulerHostConfig.default.js#L119](https://github.com/facebook/react/blob/1fb18e22ae66fdb1dc127347e169e73948778e5a/packages/scheduler/src/forks/SchedulerHostConfig.default.js" \l "L119)

当预留的时间不够用时，React将线程控制权交还给浏览器使其有时间渲染UI，React则等待下一帧时间到来继续被中断的工作。

这种将长任务分拆到每一帧中，像蚂蚁搬家一样一次执行一小段任务的操作，被称为时间切片（time slice）

**所以，解决CPU瓶颈的关键是实现时间切片，而时间切片的关键是：将同步的更新变为可中断的异步更新。**

### IO的瓶颈

网络延迟是前端开发者无法解决的。如何在网络延迟客观存在的情况下，减少用户对网络延迟的感知？

简单点儿来说， 就是在点击页面跳转的是时候提前去加载下一个页面的内容。 或者在当前页面 hold .5s 左右时间， 利用这个时间去加载下一个页面的内容。

从而达到下一个页面的快速交互

React实现了 Suspense 功能及配套的 hook——useDeferredValue。

而在源码内部，为了支持这些特性，**同样需要将同步的更新变为可中断的异步更新**。

## package.json配置了解多少?【热度:747】

package.json常见配置分类：

* **描述配置**
* **文件配置**
* **脚本配置**
* **依赖配置**
* **发布配置**
* **系统配置**
* **第三方配置**

**package.json 作用**：存储一切与项目相关的配置，例如项目基本信息、外界访问项目的方式、项目内置脚本、项目依赖等。

**描述配置**

主要是项目的基本信息，包括名称，版本，描述，仓库，作者等，部分会展示在 npm 官网上。

**文件配置**

包括项目所包含的文件，以及入口等信息。

**脚本配置**

{

// npm run {scripts} / yarn {scripts} 等命令行方式启动预设置的脚本

"scripts": {

"build": "webpack"

},

// 设置 scripts 里的脚本在运行时的参数

"config": {

"port": "3001"

},

}

**依赖配置**

项目依赖其他包引用的相关信息。

**发布配置**

主要是和项目发布相关的配置。

private

如果是私有项目，不希望发布到公共 npm 仓库上，可以将 private 设为 true。

"private": true

publishConfig

顾名思义，publishConfig 就是 npm 包发布时使用的配置。

比如在安装依赖时指定了 registry 为 taobao 镜像源，但发布时希望在公网发布，就可以指定 publishConfig.registry。

"publishConfig": {

"registry": "https://registry.npmjs.org/"

}

**系统配置**

和项目关联的系统配置，比如 node 版本或操作系统兼容性之类。这些要求只会起到提示警告的作用，即使用户的环境不符合要求，也不影响安装依赖包。

**第三方配置**

一些第三方库或应用在进行某些内部处理时会依赖这些字段，使用它们时需要安装对应的第三方库。

## 语义化版本SemVer (Semantic Versioning)了解多少?

Semantic Versioning（语义化版本）是一种为软件组件定义版本号的规范。它使用“major.minor.patch”的格式来表示版本号。其中：

* Major（主版本号）：当你做了不兼容的 API 修改时，你需要更新主版本号。
* Minor（次版本号）：当你做了向下兼容的功能性新增时，你需要更新次版本号。
* Patch（修订号）：当你做了向下兼容的问题修正时，你需要更新修订号。

Semantic Versioning 的目的是为了让软件版本号的变化具有可读性和可预测性，这样用户就可以通过版本号来了解软件包的更新内容和影响。

### 版本更新

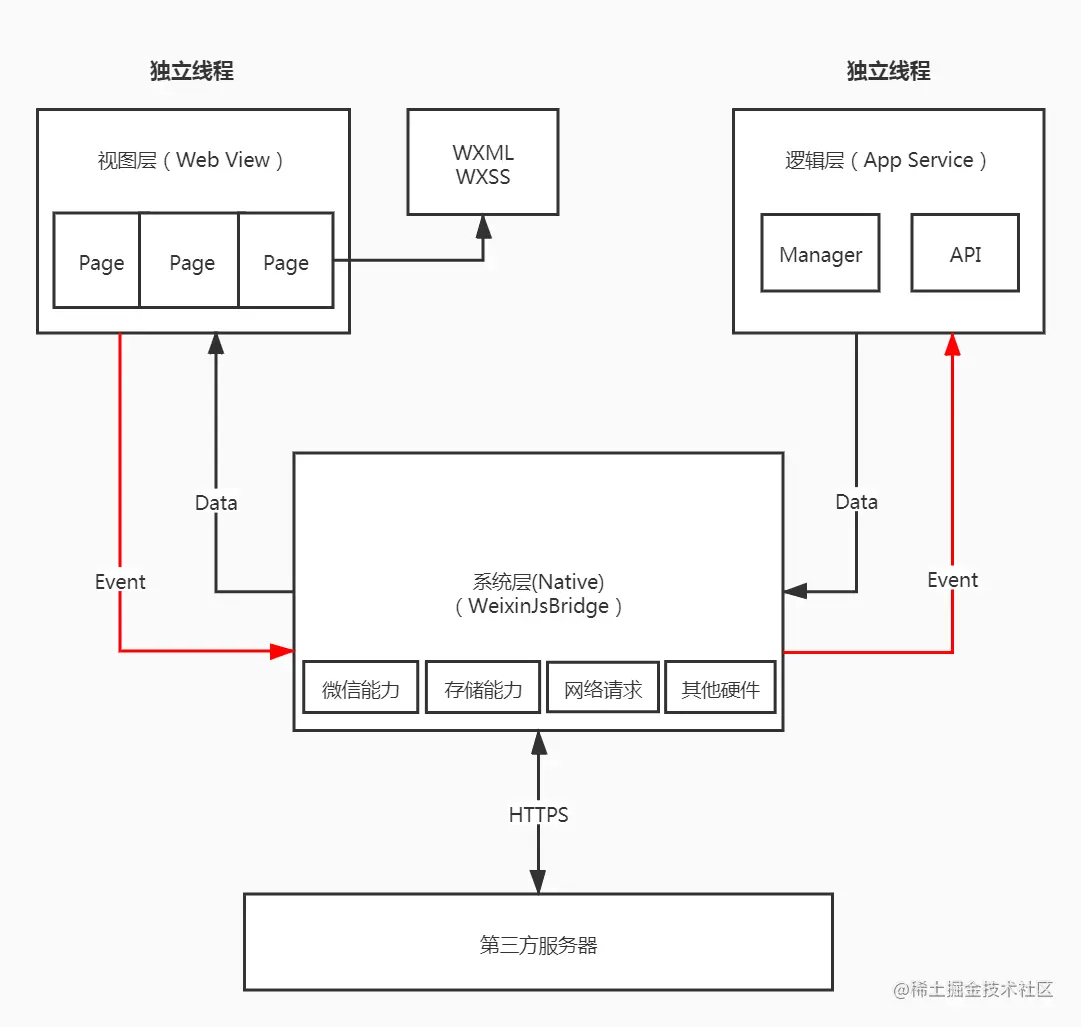
在升级版本时，常常使用一些符号来指定允许升级的范围，其中包括 ^ 和 ~ 等。

* ^ 表示向后兼容地升级版本号，只允许升级到次版本号或修订版本号，不允许升级到主版本号。
* ~ 表示只允许升级到修订版本号，不允许升级到次版本号或主版本号。

例如，对于版本号为 1.2.3：

* ^1.2.3 允许升级到 1.2.4、1.3.0 等修订号或次版号的版本，但不允许升级到 2.0.0。
* ~1.2.3 只允许升级到 1.2.4、1.2.5 等修订版本号的版本，但不允许升级到 1.3.0、2.0.0 等更高的版本。

## 小程序的大概原理?



要了解小程序架构原理， 要从以下几个方面入手探索

宿主环境

执行环境

小程序整体架构

运行机制

更新机制

数据通信机制

登录机制

性能方向问题

JSCore

[微信小程序架构原理基础解析](https://juejin.cn/post/6976805521407868958)

## 前端单页应用History路由模式,需要如何配置nginx ?

前端单页应用使用 History 路由模式，需要将路由的控制交给前端，而不是后端处理，因此需要配置 nginx 让其将所有请求都指向 index.html 页面，让前端来处理路由。

具体配置可以参考以下 nginx 配置：

bashCopy codeserver {

listen 80;

server\_name example.com; # 域名或 IP 地址

location / {

root /path/to/your/app; # 项目根目录

index index.html;

try\_files $uri $uri/ /index.html; # 将所有请求都指向 index.html 页面

}

}

以上配置中，try\_files 指令会尝试将请求映射到对应的文件，如果找不到，则会将请求转发到 /index.html 页面，由前端来处理路由。同时，root 指令需要指向前端应用的根目录。

## 实现JS沙盒的方式有哪些?

微前端已经成为前端领域比较火爆的话题，在技术方面，微前端有一个始终绕不过去的话题就是前端沙箱

### 什么是沙箱

Sandboxie(又叫沙箱、沙盘)即是一个虚拟系统程序，允许你在沙盘环境中运行浏览器或其他程序，因此运行所产生的变化可以随后删除。它创造了一个类似沙盒的独立作业环境，在其内部运行的程序并不能对硬盘产生永久性的影响。 在网络安全中，沙箱指在隔离环境中，用以测试不受信任的文件或应用程序等行为的工具

简单来说沙箱（sandbox）就是与外界隔绝的一个环境，内外环境互不影响，外界无法修改该环境内任何信息，沙箱内的东西单独属于一个世界。

### JavaScript 的沙箱

对于 JavaScript 来说，沙箱并非传统意义上的沙箱，它只是一种语法上的 Hack 写法，沙箱是一种安全机制，把一些不信任的代码运行在沙箱之内，使其不能访问沙箱之外的代码。当需要解析或着执行不可信的 JavaScript 的时候，需要隔离被执行代码的执行环境的时候，需要对执行代码中可访问对象进行限制，通常开始可以把 JavaScript 中处理模块依赖关系的闭包称之为沙箱。

### JavaScript 沙箱实现

我们大致可以把沙箱的实现总体分为两个部分：

* 构建一个闭包环境
* 模拟原生浏览器对象

具体内容见 [实现JS沙盒的方式有哪些?](https://github.com/pro-collection/interview-question/issues/145)

## 不用使用vue-cli，如何创建一个完整的vue工程?

主要涉及到的话题如下：

vue 工程初始化

测试集成

UI 库绑定、基础组件使用

开发流程

代码规范（甚至包含 commit 规范）

多人协作与工作流

构建问题

上线流程

线上日志与用户反馈问题排查

性能保证

## 使用虚拟DOM一定会比直接操作真实DOM快吗?

大家惯有的思维模式下，我们普遍的认为，虚拟DOM一定会比原生DOM要快的多。

但实际上并不是这样。

仅从React的角度来说 : **React的官网可从来都没有说过虚拟DOM会比原生操作DOM更快。**

虚拟DOM和原生操作DOM谁更快这个问题。如果要我来回答的话，一定是原生DOM比虚拟DOM更快性能更好。

值得注意的是，**虚拟DOM并不是比原生DOM快，更确切的来说，虚拟DOM是比操作不当的原生DOM快**。实际上，如果对原生DOM的操作得当的话，原生DOM的性能一定优于虚拟DOM。

### 虚拟DOM为什么而存在

其最核心的思想是**提升开发效率而非提升性能**

使用 React/Vue 这些框架的时候，我们不需要去考虑对DOM的操作，只需要关心数据的改变。我们以前还在使用JQ的时候，数据改变之后我们需要调用$("#id").append(node)等操作去手动追加DOM。而在使用React/Vue之后，我们只需要关心数据的改变。至于对DOM的一系列动作，在我们的数据改变之后，React/Vue会为我们代劳。这极大程度的提升了我们的开发效率。也是React/Vue的核心思想和初衷。

至于很多人都说，虚拟DOM会比操作原生DOM更快，这个说法并不全面。比如，首次渲染或者所有节点都需要进行更新的时候。这个时候采用虚拟DOM会比直接操作原生DOM多一重构建虚拟DOM树的操作。这会更大的占用内存和延长渲染时间。

#### 举个例子

**首次渲染 不采用虚拟DOM的步骤**

* 浏览器接受绘制指令
* 创建所有节点

**首次渲染 采用虚拟DOM的步骤**

* 浏览器接受绘制指令
* 创建虚拟DOM
* 创建所有节点

不难发现，在首次渲染的时候，采用虚拟DOM会比不采用虚拟DOM要多一个**创建虚拟DOM**的步骤。

**注意:虚拟DOM的存在，并不是免费的，比对新旧虚拟DOM树的差异会带来一定的性能开销。**

**虚拟DOM的优势在于我们更新节点时候。它会检查哪些节点需要更新。尽量复用已有DOM，减少DOM的删除和重新创建。并且这些操作我们是可以通过自己手动操作javascript底层api实现的。只是我们手动操作会非常耗费我们的时间和精力。这个工作由虚拟DOM代劳，会让我们开发更快速便捷。**

### 框架的意义

我们需要知道:不论是React还是Vue或者是Angular。这些框架本身，都是基于原生的基础上创造的。它们，底层依赖的还是javascript，并不是一门新的语言。在他们的底层逻辑下。我们使用框架所做出的一切行为，都会被框架转化为对原生DOM的操作。

**框架，只是一个转化语法的工具。**

既然原生DOM可以创造出这些框架。当然我们使用原生DOM自然是可以写出比这些框架更好的性能。

但是:为什么对原生DOM进行操作的性能明明可以比使用框架更好。为什么大家都在使用框架，而没有人去直接对原生DOM进行操作。

这背后涉及成本和普适性。

如果我们直接去操作真实DOM,当然，我们可以做到在性能上比虚拟DOM更快。但问题是，技术水准能做到这个地步的人，又有多少人呢。不说比虚拟DOM快。即使是做到和虚拟DOM不分上下的性能，拥有这种水平的前端玩家，也是寥寥无几。

**基于这样的客观情况下，框架的出现解决了这个问题。**

**框架存在的意义**: 在为我们提供只需要关注数据的前提下。框架本身已经做好了底层原理上的性能优化（包括但不限于,对DOM的调用,算法上的优化）已经是高度封装。这样就可以让我们使用一些简单的较为容易理解的技术去做我们原本做不到的事情。 这其实就像调用网上的第三方包，某一个功能，自己写是写不出来，写出来性能也不会很好。但是同样的功能，我们去网上引入其他大神已经封装完成的第三方包。我们就会用，功能就可以实现并且性能上也过得去。

如果让大家直接对DOM进行操作完成比框架更优秀的性能。这绝不是大多数人可以做到的。让大多数可以接受，框架需要做的，就是让大多数人使用尽量使用简单的技术，完成相对困难的操作。这是普适性。

并且，如果完成同一个性能效果，需要我们去精通原生javascript和学习框架上的一些简单的API和结构。明显后者的学习成本更低。如果说使用框架我们所能完成的某一阶段的性能所需要的学习成本是2个月的话。 那么学习javascript完成同一阶段的性能可能需要一年。

框架的初衷就是让用户使用尽量简单的技术，完成相对复杂的工作并提升一定的性能 （这其中包括但不限于:可维护性，可复用性，渲染效率等） 。这样，即使我们的水平不是很高，使用框架以后。项目在性能上也能过得去。

### 总结

* 虚拟DOM不一定会比操作原生DOM更快。
* 虚拟DOM的优势在于节点进行改动的时候尽量减少开销
* React从来没说过虚拟DOM会比原生更快。
* 框架的本质是提升开发效率，让我们的注意力更集中于数据

## 前端应用 CICD 有哪些方式实现【热度: 120】

### 一、使用 Jenkins

#### 持续集成：

* Jenkins 可以监听代码仓库（如 Git）的变化，当有新的代码提交时，自动触发构建任务。
* 对于前端项目，可以配置 Jenkins 执行构建命令，如使用 npm 或 yarn 安装依赖、运行构建脚本等。
* 例如，可以创建一个自由风格的项目，配置源代码管理为你的 Git 仓库地址，并在构建步骤中添加“Execute shell”，输入构建命令，如npm install && npm run build。

#### 持续部署：

* 构建成功后，Jenkins 可以将构建生成的静态文件部署到目标服务器上。
* 可以使用插件（如 Publish Over SSH）将文件传输到远程服务器，并执行部署脚本。
* 例如，配置插件连接到目标服务器，设置部署目录，然后在构建后操作中选择“Send build artifacts over SSH”，指定要传输的文件和目标服务器信息。

### 二、使用 GitLab CI/CD

#### 持续集成：

* 在.gitlab-ci.yml文件中定义一系列的阶段（stages）和任务（jobs）。
* 当代码推送到 GitLab 仓库时，GitLab Runner 会自动执行这些任务。
* 对于前端项目，可以定义一个build job，在其中执行构建命令。
* 例如：

stages:

- build

build:

stage: build

script:

- npm install

- npm run build

#### 持续部署：

* 可以在.gitlab-ci.yml中定义deploy job，将构建生成的静态文件部署到服务器上。
* 可以使用 SSH 密钥或其他部署工具来实现部署。
* 例如：

stages:

- build

- deploy

build:

stage: build

script:

- npm install

- npm run build

deploy:

stage: deploy

script:

- scp -r dist/\* user@server:/path/to/deploy

### 三、使用 GitHub Actions

#### 持续集成：

* 在.github/workflows目录下创建一个 YAML 文件来定义工作流。
* 当代码推送到 GitHub 仓库时，GitHub Actions 会自动执行工作流中的任务。
* 对于前端项目，可以在工作流中执行构建命令。
* 例如：

name: CI/CD for Frontend App

on:

push:

branches:

- main

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Install dependencies

run: npm install

- name: Build

run: npm run build

#### 持续部署：

* 可以在工作流中添加部署步骤，使用 SSH、FTP 等方式将静态文件部署到服务器上。
* 或者使用云服务提供商的部署服务，如 AWS Amplify、Netlify 等。
* 例如：

name: CI/CD for Frontend App

on:

push:

branches:

- main

jobs:

build:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Install dependencies

run: npm install

- name: Build

run: npm run build

deploy:

needs: build

runs-on: ubuntu-latest

steps:

- name: Deploy to Server

run: scp -r dist/\* user@server:/path/to/deploy

### 四、使用 Travis CI

#### 持续集成：

* 在项目根目录下创建一个.travis.yml文件来定义构建配置。
* 当代码推送到支持的代码仓库（如 GitHub）时，Travis CI 会自动触发构建。
* 对于前端项目，可以在配置文件中指定构建命令。
* 例如：

language: node\_js

node\_js:

- 12

script:

- npm install

- npm run build

#### 持续部署：

* 可以在构建成功后，使用部署工具或脚本将静态文件部署到服务器上。
* 例如，可以在.travis.yml中添加部署步骤，使用 SSH 或其他方式进行部署。

language: node\_js

node\_js:

- 12

script:

- npm install

- npm run build

after\_success:

- scp -r dist/\* user@server:/path/to/deploy

## 前端工程里面，如何检测并避免循环依赖【热度: 359】

**关键词：循环依赖检测**

### 一、检测循环依赖

#### 手动审查代码：

* 仔细检查项目中的模块导入关系。查看每个模块的导入语句，确定是否存在一个模块被另一个模块导入，而后者又反过来导入前者的情况。
* 对于大型项目，可以使用工具辅助手动审查，如代码编辑器的搜索功能或一些专门的代码分析工具，来快速查找可能的循环依赖。

#### 使用静态分析工具：

* 有一些静态分析工具可以帮助检测循环依赖。例如，ESLint 的插件eslint-plugin-import可以配置规则来检测模块之间的循环依赖。
* 使用这些工具可以在开发过程中自动检测循环依赖，并给出明确的错误提示，帮助开发者及时发现和修复问题。

### 二、避免循环依赖

#### 优化模块设计：

* 重新审视项目的模块结构，确保模块之间的依赖关系清晰且单向。避免设计出相互依赖的模块结构。
* 例如，如果模块 A 和模块 B 相互依赖，可以考虑将它们共同依赖的部分提取出来，创建一个新的模块 C，然后让 A 和 B 分别依赖于 C，而不是直接相互依赖。

#### 采用依赖注入：

* 依赖注入是一种设计模式，可以帮助减少模块之间的直接依赖。通过将依赖项作为参数传递给函数或构造函数，可以在运行时动态地注入依赖项，而不是在模块内部直接导入它们。
* 这样可以降低模块之间的耦合度，减少循环依赖的可能性。

#### 延迟加载：

* 在某些情况下，可以使用延迟加载（lazy loading）来避免循环依赖。延迟加载是指在实际需要的时候才加载模块，而不是在应用程序启动时就加载所有模块。
* 例如，可以使用动态导入（dynamic import）来实现延迟加载。这样可以避免在模块加载时立即出现循环依赖的问题，因为模块只有在需要的时候才会被加载。

#### 建立清晰的模块边界：

* 每个模块应该有明确的职责和功能范围。避免模块之间的职责重叠，这样可以减少模块之间的相互依赖。
* 同时，尽量保持模块的独立性，使得模块可以独立开发、测试和维护。

## 部署一个 node 应用要考虑哪些因素【热度: 98】

### 一、选择服务器和环境

#### 服务器选择：

* 可以选择云服务器提供商（如阿里云、腾讯云、AWS 等）的虚拟服务器或容器服务。根据应用的需求和预算，确定服务器的配置（CPU、内存、存储等）。
* 考虑服务器的地理位置和网络连接质量，以确保用户能够快速访问应用。

#### 操作系统：

* 常见的选择是 Linux 发行版，如 Ubuntu、CentOS 等。Linux 系统具有稳定性、安全性和资源效率高的特点，适合部署服务器应用。

### 二、代码准备和优化

#### 代码审查和测试：

* 在部署之前，确保代码经过严格的审查和测试。包括单元测试、集成测试和端到端测试，以确保应用的功能和性能符合要求。
* 检查代码中的潜在安全漏洞，如 SQL 注入、跨站脚本攻击（XSS）等，并采取相应的防范措施。

#### 性能优化：

对 Node.js 应用进行性能优化，以提高在生产环境中的响应速度和吞吐量。可以采用以下优化措施：

* 优化数据库查询，避免不必要的查询和复杂的连接。
* 使用缓存技术，如内存缓存或分布式缓存，减少重复计算和数据库访问。
* 压缩和优化静态资源，如 CSS、JavaScript 和图片，减少网络传输时间。
* 配置适当的日志级别，避免过多的日志输出影响性能。

### 三、部署方式

#### 手动部署：

直接将代码上传到服务器，并手动安装和配置所需的依赖项和运行环境。这种方式适用于小型项目或临时部署，但不够自动化和可重复。

步骤如下：

* 使用 SSH 连接到服务器。
* 将代码上传到服务器，可以使用 FTP、SCP 或 Git 等方式。
* 在服务器上安装 Node.js 和所需的依赖项（使用npm install或yarn install）。
* 配置应用的环境变量，如数据库连接字符串、端口号等。
* 启动应用（使用node app.js或通过 PM2 等进程管理工具）。

#### 持续集成/持续部署（CI/CD）：

* 使用 CI/CD 工具实现自动化的部署流程。这种方式可以提高部署的效率和可靠性，减少人为错误。
* 常见的 CI/CD 工具包括 Jenkins、Travis CI、CircleCI 等。以下是一般的部署流程：
* 将代码托管在版本控制系统（如 Git）中。
* 配置 CI/CD 工具，设置构建和部署的触发器（如代码提交、定时任务等）。
* 在 CI/CD 管道中，进行代码构建、测试和打包。
* 将打包后的应用部署到服务器，可以使用 SSH、Docker 等方式。
* 进行自动化的测试和验证，确保应用在部署后正常运行。

### 四、运行和监控

#### 进程管理：

* 使用进程管理工具来确保 Node.js 应用在服务器上持续运行。常见的进程管理工具包括 PM2、Forever 等。
* 这些工具可以监控应用的运行状态，自动重启应用在崩溃或出现错误时，并提供日志管理和性能监控功能。

#### 监控和日志记录：

* 配置应用的日志记录，以便在生产环境中跟踪应用的运行状态和错误。可以使用日志管理工具，如 Logstash、ELK Stack 等，集中收集和分析日志。
* 同时，使用监控工具来实时监测应用的性能指标，如 CPU 使用率、内存占用、响应时间等。常见的监控工具包括 New Relic、Datadog、Prometheus 等。

#### 安全措施：

采取必要的安全措施来保护生产环境中的 Node.js 应用。包括：

* 配置防火墙，限制对服务器的访问。
* 使用 HTTPS 加密通信，保护用户数据的安全。
* 定期更新服务器和应用的软件包，以修复安全漏洞。
* 实施访问控制，限制对敏感资源的访问。

## commonjs 模块化中 module.exports 与 exports 有什么区别， 该如何选择【热度: 194】

### 一、module.exports

#### 本质：

module.exports是一个对象，它代表当前模块要导出的内容。可以将任意类型的值（如函数、对象、字符串等）赋值给module.exports来决定模块导出的内容。

#### 作用范围和灵活性：

可以完全覆盖模块的导出内容。例如，可以直接将一个全新的对象赋值给module.exports，从而完全替换模块原本可能通过exports添加的属性。

适合需要导出复杂数据结构或多个不同类型的值的情况。例如，可以导出一个包含多个函数和变量的对象。

### 二、exports

#### 本质：

exports最初是一个对module.exports的引用。这意味着通过exports添加的属性实际上是添加到了module.exports所代表的对象上。

#### 局限性：

如果直接将一个值赋值给exports，它将不再是对module.exports的引用，而是变成一个独立的变量。此时，模块的导出内容将变为这个新的值，而不是原本期望的通过添加属性到exports来构建的导出对象。

例如：

exports = function () {

console.log("This is a new function.");

};

在这种情况下，模块将不再导出之前可能通过exports.xxx = yyy添加的属性，而是只导出这个新的函数。

### 三、选择建议\*\*：

#### 简单模块导出单个值：

如果模块只需要导出一个简单的值，如一个函数或一个字符串，可以使用module.exports直接赋值的方式。例如：

module.exports = function add(a, b) {

return a + b;

};

#### 复杂模块构建导出对象：

当模块需要导出多个相关的值或功能时，可以先使用exports添加属性，最后确保module.exports指向一个包含所有需要导出内容的对象。例如：

exports.foo = function () {

console.log("foo function.");

};

exports.bar = "bar value";

module.exports = exports; // 确保 module.exports 和 exports 指向同一个对象

#### 避免混淆和错误：

理解module.exports和exports的区别非常重要，以避免在导出模块内容时出现意外的结果。尽量明确使用module.exports或遵循使用exports的正确方式，避免直接赋值给exports而导致错误的导出行为。

## 对于已经 import 但未实际使用的模块使用 webpack 还会对它打包吗【热度: 108】

在使用 Webpack 进行打包时，对于已经导入（import）但未实际使用的模块，**通常情况下不会被打包进去，但这也取决于具体的配置和使用场景**。

### 一、默认行为

在默认情况下，Webpack 会进行“树摇”（tree shaking）操作。树摇是一种优化技术，它可以检测和移除未被使用的代码。如果一个模块被导入但在代码中没有被实际使用，Webpack 会尝试识别并排除这个模块，以减小最终打包文件的大小。

import { someFunction } from "moduleA";

// 如果 someFunction 在后续代码中没有被调用，Webpack 可能会在打包时排除 moduleA 中未被使用的部分。

### 二、特殊情况

然而，在某些情况下，即使模块未被直接使用，也可能会被打包进去：

**1.副作用**：如果导入的模块有副作用（例如修改全局状态、注册事件监听器等），即使模块中的具体函数或变量没有被直接使用，Webpack 也可能不会将其排除。因为这些副作用可能对应用的运行产生影响。

例如：

import "moduleB"; // moduleB 可能会在导入时执行一些初始化操作，有副作用。

**2.动态导入**：如果使用动态导入（例如通过 import()函数），Webpack 可能无法在打包时确定哪些模块会被实际使用，因此可能会保留一些未被直接使用的模块，以确保在运行时能够正确加载所需的模块。

例如：

const moduleName = "moduleC";

import(moduleName).then((module) => {

// 使用动态导入的模块。

});

**3.第三方库的依赖**：如果导入的模块被其他模块间接依赖，即使在当前模块中没有直接使用，也可能会被打包进去。例如，一个第三方库可能依赖于某个特定的模块，而你的代码只导入了这个第三方库，Webpack 可能会将这个被依赖的模块也打包进来。

## 前端需要加载一个大体积的文件时， 一般有哪些优化思路【热度: 594】

### 一、文件压缩

#### 服务器端压缩：

在服务器上配置文件压缩功能，如使用 Gzip 或 Brotli 压缩算法对文件进行压缩后再传输。这样可以显著减少文件的大小，降低传输时间。

例如，在 Nginx 服务器中，可以通过配置开启 Gzip 压缩：

gzip on;

gzip\_comp\_level 6;

gzip\_types text/plain text/css application/javascript application/json image/svg+xml;

#### 客户端解压缩：

现代浏览器通常支持对 Gzip 和 Brotli 压缩的文件进行自动解压缩。当浏览器接收到压缩后的文件时，会自动解压缩并使用。

无需额外的客户端代码，浏览器会自动处理压缩文件的解压缩过程，提高文件加载速度。

### 二、文件分割与懒加载

#### 文件分割：

将大体积文件分割成多个较小的文件。例如，对于一个大型的 JavaScript 库，可以将其拆分成多个模块，根据需要逐步加载。

这样可以避免一次性加载整个大文件，减少初始加载时间。

例如，使用 Webpack 等构建工具可以将代码分割成多个 chunk，根据路由或特定条件进行加载。

#### 懒加载：

对于不是立即需要的文件或资源，可以采用懒加载的方式。当用户实际需要使用该资源时，再进行加载。

例如，对于图片、视频等资源，可以在用户滚动到可视区域时再进行加载，避免在页面初始加载时加载所有资源。

对于 JavaScript 模块，可以使用动态导入（dynamic import）的方式实现懒加载：

const loadModule = async () => {

const module = await import("./largeModule.js");

// 使用加载的模块

};

### 三、缓存策略

#### 浏览器缓存：

设置合理的缓存策略，让浏览器缓存已经加载过的文件。这样，当用户再次访问时，可以直接从缓存中读取文件，而无需再次从服务器下载。

可以通过设置 HTTP 响应头来控制缓存，例如：

location / {

add\_header Cache-Control "max-age=3600";

}

上述配置将设置文件的缓存时间为 1 小时。

#### 缓存更新机制：

当文件内容发生变化时，需要确保浏览器能够获取到最新的版本。可以通过在文件名中添加版本号或哈希值来实现缓存更新。

例如，将文件名改为largeFile\_v1.2.js或largeFile\_abc123.js，当文件内容变化时，更新版本号或哈希值，浏览器会认为这是一个新的文件并进行下载。

### 四、优化加载顺序

#### 关键资源优先加载：

确定哪些资源是页面加载的关键资源，优先加载这些资源。对于大体积文件，如果不是关键资源，可以延迟加载。

例如，对于一个图片库应用，先加载页面的基本结构和导航部分，图片可以在用户交互时再进行加载。

#### 异步加载：

使用异步加载的方式加载大体积文件。例如，对于 JavaScript 文件，可以使用<script async>标签或动态创建<script>标签并插入到页面中进行异步加载。

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

这样可以避免阻塞页面的渲染，提高用户体验。

### 五、CDN 加速

#### 使用内容分发网络（CDN）：

将大体积文件托管在 CDN 上，利用 CDN 的分布式节点，可以让用户从离自己最近的节点获取文件，减少网络延迟，提高加载速度。

例如，将图片、视频、静态文件等托管在 CDN 上，通过 CDN 的 URL 进行访问。

#### CDN 缓存：

CDN 通常会对文件进行缓存，进一步提高文件的加载速度。当文件内容发生变化时，需要及时更新 CDN 上的缓存。

可以通过设置 CDN 的缓存策略或使用版本号等方式来管理 CDN 缓存。

## 前端部署应该要考虑哪些问题【热度: 310】

**关键词：前端部署**

在前端部署到生产环境时，有多个关键问题需要解决：

### 一、高可用性与故障应对

生产环境中，单点故障是必须考虑的问题。若一台物理机突然崩溃，应用需依然能够正常运行。这时可以了解 Kubernetes（k8s）的 Deployment，它能够有效地应对单点故障，确保应用的高可用性。

### 二、热更新与用户体验

热更新也是重要环节。当需要更新应用时，要做到让用户无感知。同样，k8s 的 Deployment 提供了热更新的功能，能够在不影响用户使用的情况下完成应用的更新。

### 三、版本回退机制

如果上线的新版本出现问题，需要有时光回溯的能力，能够快速回退到之前的稳定版本。

### 四、负载均衡

负载均衡能够确保流量均匀分配到各个服务器上，提高系统的性能和稳定性。使用 Nginx 等工具可以实现负载均衡，满足生产环境的需求。

### 五、自动扩缩容

当流量突然增大时，系统需要能够自动扩容以应对压力。反之，流量减少时也应能自动缩容以节省资源。

### 六、数据库部署与运维

数据库部署不建议在 k8s 上进行，可选择本地部署 MySQL 集群或直接上云。同时，还需要具备基本的数据运维能力，如日志收集（ELK 或 EFK）和性能监控等。

### 总结

要将前端应用成功部署到生产环境，需要解决上述多个问题。自动化 pipeline 也是关键，包括 Jenkins、GitLab、k8s 等工具的组合使用。将这一套组合拳运用好，即使不是非常厉害的高手，达到中级水平，也能在中小型公司中轻松应对各种部署需求，实现高效的生产环境部署和运维。

## monorepo 工程有哪些工具架构， 该如何选型【热度: 450】

**关键词：monorepo 技术选项**

### 工具推荐

| 工具 | Turborepo | Rush | Nx | Lerna | Pnpm Workspace |
| --- | --- | --- | --- | --- | --- |
| 依赖管理 | ❌ | ✅ | ❌ | ❌ | ✅ |
| 版本管理 | ❌ | ✅ | ❌ | ✅ | ❌ |
| 增量构建 | ✅ | ✅ | ✅ | ❌ | ❌ |
| 插件扩展 | ✅ | ✅ | ✅ | ❌ | ❌ |
| 云端缓存 | ✅ | ✅ | ✅ | ❌ | ❌ |
| Stars | 20.4K | 4.9K | 17K | 34.3K | 22.7K |

### 工具选型

选择合适的 Monorepo 管理工具对于确保项目的顺利进行是至关重要的。Monorepo 管理工具可以帮助你高效地管理项目依赖、统一代码风格、简化开发流程等。在进行 Monorepo 工具选型时，需要考虑几个重要的因素：

#### 1. 技术栈的兼容性

* Lerna：与任何技术栈兼容性都很好，特别是与前端项目协同工作时。它对 NPM 和 Yarn 都有良好支持，适用于需要独立版本管理或频繁发布的项目。
* Yarn Workspaces：特别适合使用 Yarn 作为包管理器的 JavaScript 或 TypeScript 项目。它非常适合团队中包之间有很多交叉依赖的情形。
* Nx：支持多种前端和后端框架，如 Angular、React、NestJS 等。如果项目采用多技术栈，Nx 提供了一套完整的解决方案，包含了构建、测试和 linting 等一站式服务。
* Rush：同样适用于大型项目，兼容任何 NPM 包管理器，如 NPM、Yarn、pnpm。Rush 提供了灵活的版本控制策略，非常适合需要精细控制包版本策略的场景。
* pnpm Workspaces：具有高效的节点模块解析机制，非常注重节省磁盘空间及速度优化。如果磁盘空间和安装速度是关键考虑因素，pnpm 会是一个不错的选择。

#### 2. 项目的规模和复杂度

对于大型或复杂项目，Nx 和 Rush 提供了更多的高级特性，比如增量构建、依赖图可视化等，可以有效提升大团队的开发效率。

对于中小型项目，Lerna、Yarn Workspaces 或 pnpm Workspaces 可能更易上手，配置和管理也较为简单。

#### 3. 构建、测试和部署的需求

如果项目需要复杂的构建、测试流程，Nx 提供了一些很好的工具来优化这一过程。Nx 可以智能地只重新构建受影响的项目，节省 CI/CD 的时间和资源。

Rush 强调在大型仓库中提供稳定而灵活的版本策略和发布管理，对于需要精细控制不同环境部署的项目非常有用。

#### 4. 团队协作和代码共享的便利性

所有这些工具都支持代码共享和重用，但是Nx 和 Rush 在支持大型团队和多项目协作方面有一些额外的优势，如更智能的依赖管理和版本控制。

#### 5. 社区支持和文档

Nx 拥有强大的社区支持和丰富的文档、教程，非常适合于新技术栈的团队。

Lerna 和 Yarn Workspaces 受众广泛，网上有很多资源和案例，学习曲线相对平缓。

推荐策略

如果你的项目非常关注于构建效率和对多种技术栈的支持，Nx 是非常好的选择。如果你更关心包的独立发布和版本管理，Lerna 和 Rush 可以满足你的需求。而对于那些偏好 Yarn 并且注重依赖管理的项目来说，Yarn Workspaces 提供了一套简单直接的解决方案。如果磁盘空间和安装速度是你的主要考虑，不妨试试 pnpm Workspaces。