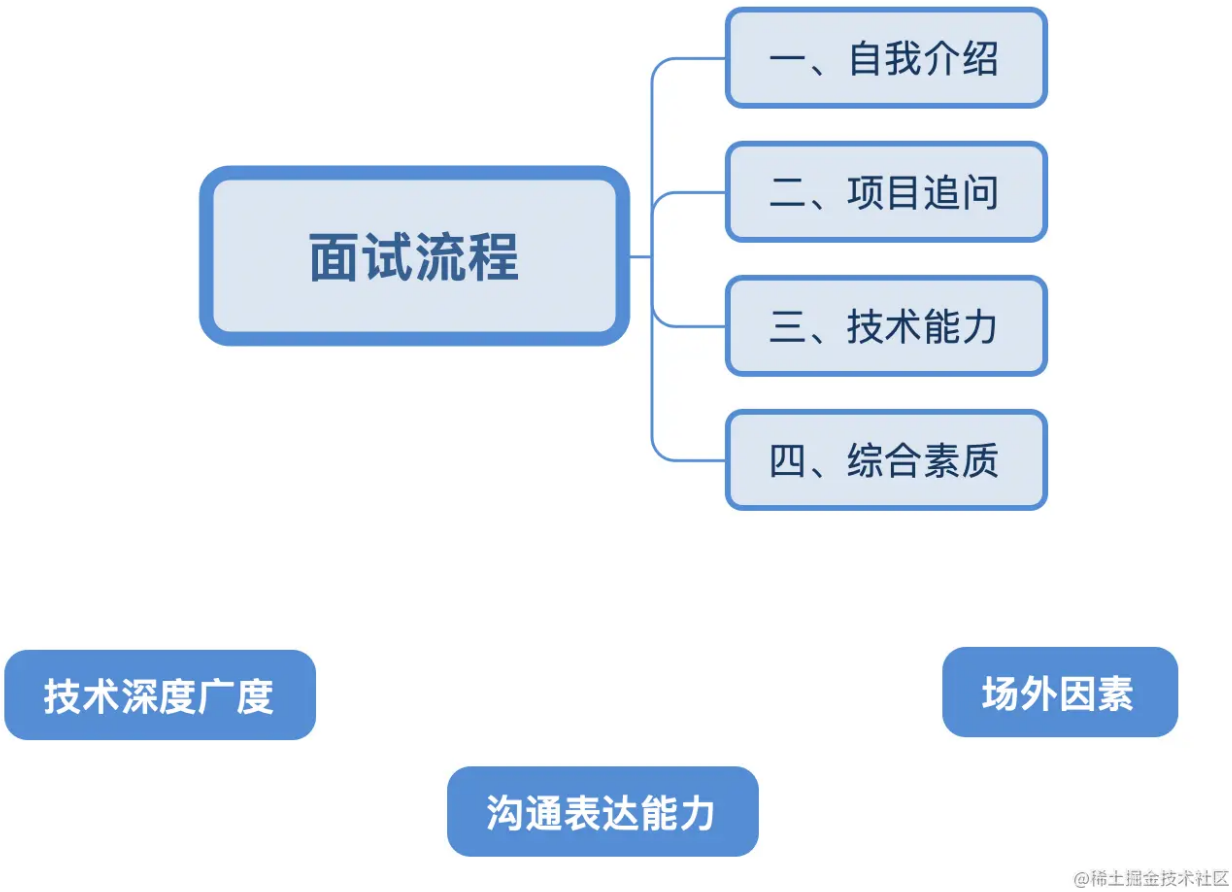
2022凛冬之时三年经验前端面经

今年的就业形式简直一片黑暗，本着明年会比今年还差的“侥幸心理”，我还是毫不犹豫地选择裸辞了，历经一个半月的努力，收到了还算不错的 offer，薪资和平台都有比较大的提升，但还是和自己的心理预期有着很大的差距。所以得出最大的结论就是：不要裸辞！不要裸辞！不要裸辞！因为面试期间带给人的压力，和现实与理想的落差对心理的摧残是不可估量的，在这样一个环境苟着是最好的选择。

接下来总结一般情况下前端面试中会经历的以下四个阶段和三个决定因素：



作为前端人员，技术的深度广度是排在第一位的，三年是一个分割线，一定要在这个时候找准自己的定位和方向。

其次良好的沟通表达能力、着装和表现等场外因素能提高面试官对你的认可度。

有的人技术很牛逼，但是面试时让面试官觉得不爽，觉得你盛气逼人/形象邋遢/自以为是，就直接不要你，那是最得不偿失的。

以下是我整个面试准备以及被问到过的问题的一个凝练，因为在面试过程中，和面试官的交流很大程度并不是简单的背书，最好是要将知识通过自己的总结和凝练表达出来，再根据面试官的提问临场发挥将之补足完善。

一、自我介绍

面试官让你自我介绍，而且不限定自我介绍的范围，肯定是面试官想从你的自我介绍中了解到你，所以介绍一定要保证简短和流畅，面对不同的面试官，自我介绍的内容可以是完全一样的，所以提前准备好说辞很重要，并且一定要注意：不要磕磕巴巴，要自信！**流畅的表达和沟通能力**，同样是面试官会对候选人考核点之一。我也曾当过面试官，自信大方的候选人，往往更容易受到青睐。

1、个人介绍（基本情况），主要的简历都有了，这方面一定要短

2、个人擅长什么，包括技术上的和非技术上的。技术上可以了解你的转场，非技术可以了解你这个人

3、做过的项目，捡最核心的项目说，不要把所有项目像背书一样介绍

4、自己的一些想法、兴趣或者是观点，甚至包括自己的职业规划。这要是给面试官一个感觉：热衷于"折腾"或者"思考"

> 示例：
>
> 面试官您好，我叫xxx，xx年毕业于xx大学，自毕业以来一直从事着前端开发的相关工作。
>
> 我擅长的技术栈是 vue 全家桶，对 vue2 和 vue3 在使用上和源码都有一定程度的钻研；打包工具对 webpack 和 vite 都比较熟悉；有从零到一主导中大型项目落地的经验和能力。
>
> 在上家公司主要是xx产品线负责人的角色，主要职责是。。。。。。
>
> 除了开发相关工作，还有一定的技术管理经验：比如担任需求评审、UI/UE交互评审评委，负责开发排期、成员协作、对成员代码进行review、组织例会等等
>
> 平常会在自己搭建的博客上记录一些学习文章或者学习笔记，也会写一些原创的技术文章发表到掘金上，获得过xx奖。

总的来说自我介绍尽量控制在3 - 5分钟之间，简明扼要为第一要义，其次是突出自己的能力和长处。

对于普通的技术面试官来说，自我介绍只是习惯性的面试前的开场白，一般简历上列举的基本信息已经满足他们对你的基本了解了。但是对于主管级别的面试官或者Hr，会看重你的性格、行为习惯、抗压能力等等综合能力。所以要让自己在面试过程尽可能表现的积极向上，爱好广泛、喜欢持续学习，喜欢团队合作，可以无条件加班等等。当然也不是说让你去欺骗，只是在现在这种环境中，这些"侧面能力"也是能在一定程度提升自己竞争力的法宝。

二、项目挖掘

在目前这个行情，当你收到面试通知时，有很大概率是因为你的项目经验和所招聘的岗位比较符合。 所以在项目的准备上要额外上心，比如：

1. 对项目中使用到的**技术**的深挖
2. 对项目**整体设计**思路的把控
3. 对项目**运作流程**的管理
4. **团队协作**的能力等等。

这些因人而异就不做赘述了，根据自己的情况好好挖掘即可。

三、个人

先说个人，当你通过了技术面试，到了主管和hr这一步，不管你当前的技术多牛逼，他们会额外考察你个人的潜力、学习能力、性格与团队的磨合等软实力，这里列出一些很容易被问到的：

为什么跳槽?

直接从个人发展入手表现出自己的上进心：

1. 一直想去更大的平台，不但有更好的技术氛围，而且学到的东西也更多
2. 想扩展一下自己的知识面，之前一致是做x端的 xx 产品，技术栈比较单一一点，相对xx进行学习。

3. 之前的工作陷入了舒适圈，做来做去也就那些东西，想要换个平台扩宽自己的技术广度，接触和学习一些新的技术体系，为后续的个人发展更有利

讲讲你和普通前端，你的亮点有哪些?

1、善于规划和总结，我会对自己经手的项目进行一个全面的分析，一个是业务拆解，对个各模块的业务通过脑图进行拆解；另一个就是对代码模块的拆解，按功能去区分各个代码模块。再去进行开发。我觉得这是很多只会进行盲目业务开发的前端做不到的

2、喜欢专研技术，平常对 vue 的源码一直在学习，也有输出自己的技术文章，像之前写过一篇逐行精读 teleport 的源码，花了大约有三十个小时才写出来的，对每一行源码的功能和作用进行了解读（但是为啥阅读和点赞这么低）。

你有什么缺点?

性子比较沉，更偏内向一点，所以我也会尝试让自己变得外向一点。

一个是要开各种评审会，作为前端代表需要我去准备各种材料和进行发言。

所以在团队内做比较多的技术分享，每周主持例会，也让我敢于去表达和探讨。

最近有关注什么新技术吗?

1. 包依赖管理工具 pnpm(不会重复安装依赖，非扁平的node_modules结构，符号链接方式添加依赖包)
2. 打包工具 vite （极速的开发环境）
3. flutter （Google推出并开源的移动应用程序（App）开发框架，主打跨平台、高保真、高性能）
4. rust（听说是js未来的基座）
5. turbopack，webpack的继任者，说是比 vite快10倍，webpack快700倍，然后尤雨溪亲自验证其实并没有比 vite 快10倍
6. webcomponents

你是偏向于走各个方向探索还是一直向某个方向研究下去?

我对个人的规划是这样的：

3 - 5 年在提高自己的技术深度的同时，扩宽自己的知识面，就是深度和广度上都要有提升，主要是在广度上，充分对大前端有了认知才能更好的做出选择

5 - 7 年就是当有足够的知识积累之后再选择某一个感兴趣方向深研下去，争取成为那个领域的专家

团队规模，团队规范和开发流程

这个因人而异，如实准备即可，因为不同规模团队的研发模式差别是很大的。

代码 review 的目标

1、最注重的是代码的可维护性（变量命名、注释、函数单一性原则等）

2、扩展性：封装能力（组件、代码逻辑是否可复用、可扩展性）

3、ES 新特性（es6+ 、ES2020, ES2021 可选链、at）

4、函数使用规范（比如遇到用 map 拿来当 forEach 用的）

5、性能提升，怎样运用算法，写出更加优雅，性能更好的代码

如何带领团队的

我在上家公司是一个技术管理的角色。

0、**落实开发规范**，我在公司内部 wiki 上有发过，从命名、最佳实践到各种工具库的使用。新人进来前期我会优先跟进他们的代码质量

1、**团队分工**：每个人单独负责一个产品的开发，然后公共模块一般我会指定某几个人开发

2、**代码质量保证**：每周会review他们的代码，也会组织交叉 review 代码，将修改结果输出文章放到 wiki中

3、**组织例会**：每周组织例会同步各自进度和风险，根据各自的进度调配工作任务

4、**技术分享**：还会组织不定时的技术分享。一开始就是单纯的我做分享，比如微前端的体系，ice stark 的源码

5、**公共需求池**：比如webpack5/vite的升级；vue2.7的升级引入setup语法糖；pnpm的使用；拓扑图性能优化

6、**优化专项**：在第一版产品出来之后，我还发起过性能优化专项，首屏加载性能，打包体积优化；让每个人去负责对应的优化项

对加班怎么看?

我觉得加班一般会有两种情况：

一是项目进度比较紧，那当然以项目进度为先，毕竟大家都靠这个吃饭

二是自身能力问题，对业务不熟啊或者引入一个全新的技术栈，那么我觉得不仅要加班跟上，还要去利用空闲时间抓紧学习，弥补自己的不足

有什么兴趣爱好?

我平常喜欢阅读，就是在微信阅读里读一些心理学、时间管理、还有一些演讲技巧之类的书

然后是写文章，因为我发现单纯的记笔记很容易就忘了，因为只是记载别人的内容，而写自己的原创文章，在这个过程中能将知识非常高的比例转换成自身的东西，所以除了自个发掘金的文章，我也经常会对项目的产出有文章输出到 wiki 上

其他爱好就是和朋友约着打篮球、唱歌

四、技术

技术面试一定要注意：**简明扼要，详略得当，不懂的就说不懂。**因为在面试过程中是一个和面试官面对面交流的过程，没有面试官会喜欢一个絮絮叨叨半天说不到重点候选人，同时在说话过程中，听者会被动的忽略自己不感兴趣的部分，所以要着重突出某个技术的核心特点，并围绕着核心适当展开。

大厂基本都会通过算法题来筛选候选人，算法没有捷径，只能一步一步地刷题再刷题，这方面薄弱的要提前规划进行个学习了。

技术面过程主要会对前端领域相关的技术进行提问，一般面试官会基于你的建立，而更多的是，面试官基于他之前准备好的面试题，或者所在项目组比较熟悉的技术点进行提问，因为都

是未知数，所以方方面面都还是要求比较足的。

如果想进入一个中大型且发展前景不错的公司，并不是照着别人的面经背一背就能糊弄过去的，这里作出的总结虽然每一条都很简短，但都是我对每一个知识点进行全面学习后才提炼出来的部分核心知识点，所以不惧怕面试官的“发散一下思维”。

JS篇

什么是原型/原型链?

原型的本质就是一个**对象**。

当我们在创建一个构造函数之后，这个函数会默认带上一个 `prototype` 属性，而这个属性的值就指向这个函数的原型对象。

这个原型对象是用来为通过该构造函数创建的实例对象提供共享属性，即**用来实现基于原型的继承和属性的共享**

所以我们通过构造函数**创建的实例对象**都会从这个函数的原型对象上继承上面具有的属性

当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还查不到，就去找原型的原型，一直找到最顶层为止（最顶层就是 `Object.prototype` 的原型，值为null）。

所以**通过原型一层层相互关联的链状结构就称为原型链。**

什么是闭包?

定义：闭包是指**引用了其他函数作用域中变量的函数**，通常是在嵌套函数中实现的。

从技术角度上所有 js 函数都是闭包。

从实践角度来看，满足以下俩个条件的函数算闭包

1. 即使创建它的上下文被销毁了，它依然存在。（比如从父函数中返回）
2. 在代码中引用了自由变量（在函数中使用的既不是函数参数也不是函数局部变量的变量称作自由变量）

使用场景：

- 创建私有变量

  vue 中的data，需要是一个闭包，保证每个data中数据唯一，避免多次引用该组件造成的data共享

- 延长变量的生命周期

  一般函数的词法环境在函数返回后就被销毁，但是闭包会保存对创建时所在词法环境的引用，即便创建时所在的执行上下文被销毁，但创建时所在词法环境依然存在，以达到延长变量的生命周期的目的

应用

- 柯里化函数
- 例如计数器、延迟调用、回调函数等

this 的指向

在绝大多数情况下，函数的调用方式决定了 `this` 的值（运行时绑定）

1、全局的this非严格模式指向window对象，严格模式指向 undefined

2、对象的属性方法中的this 指向对象本身

3、apply、call、bind 可以变更 this 指向为第一个传参

4、箭头函数中的this指向它的父级作用域，它自身不存在 this

浏览器的事件循环?

js 代码执行过程中，会创建对应的执行上下文并压入执行上下文栈中。

如果遇到异步任务就会将任务挂起，交给其他线程去处理异步任务，当异步任务处理完后，会将回调结果加入事件队列中。

当执行栈中所有任务执行完毕后，就是主线程处于闲置状态时，才会从事件队列中取出排在首位的事件回调结果，并把这个回调加入执行栈中然后执行其中的代码，如此反复，这个过程就被称为事件循环。

事件队列分为了**宏任务队列**和微任务队列，在当前执行栈为空时，主线程回先查看微任务队列是否有事件存在，存在则依次执行微任务队列中的事件回调，直至微任务队列为空；不存在再去宏任务队列中处理。

常见的宏任务有 `setTimeout()` 、 `setInterval()` 、 `setImmediate()` 、I/O、用户交互操作，UI渲染

常见的微任务有 `promise.then()` 、 `promise.catch()` 、 `new MutationObserver` 、 `process.nextTick()`

宏任务和微任务的本质区别

1. 宏任务有明确的异步任务需要执行和回调，需要其他异步线程支持
2. 微任务没有明确的异步任务需要执行，只有回调，不需要其他异步线程支持。

javascript中数据在栈和堆中的存储方式

1、基本数据类型大小固定且操作简单，所以放入栈中存储

2、引用数据类型大小不确定，所以将它们放入堆内存中，让它们在申请内存的时候自己确定大小

3、这样分开存储可以使内存占用最小。栈的效率高于堆

4、栈内存中变量在执行环境结束后会立即进行垃圾回收，而堆内存中需要变量的所有引用都结束才会被回收

讲讲v8垃圾回收

1、根据对象的存活时间将内存的垃圾回收进行不同的分代，然后对不同分代采用不同的回收算法

2、新生代采用空间换时间的 scavenge 算法：整个空间分为两块，变量仅存在其中一块，回收的时候将存活变量复制到另一块空间，不存活的回收掉，周而复始轮流操作

3、老生代使用标记清除和标记整理，标记清除：遍历所有对象标记标记可以访问到的对象（活着的），然后将不活的当做垃圾进行回收。回收完后避免内存的断层不连续，需要通过标记整理将活着的对象往内存一端进行移动，移动完成后再清理边界内存

函数调用的方法

1、普通 `function` 直接使用 `()` 调用并传参，如： `function test(x, y) { return x + y}` ，`test(3, 4)`

2、作为对象的一个属性方法调用，如： `const obj = { test: function (val) { return val } }` ，`obj.test(2)`

3、使用 `call` 或 `apply` 调用，更改函数 this 指向，也就是更改函数的执行上下文

4、 `new` 可以间接调用构造函数生成对象实例

defer和async的区别

一般情况下，当执行到 script 标签时会进行下载 + 执行两步操作，这两步会阻塞 HTML 的解析；

async 和 defer 能将script的**下载阶段**变成异步执行（和 html解析同步进行）；

async下载完成后会立即执行js，此时会阻塞HTML解析；

defer会等全部HTML解析完成且在DOMContentLoaded 事件之前执行。

浏览器事件机制

DOM 事件流三阶段：

1. **捕获阶段**：事件最开始由不太具体的节点最早接受事件, 而最具体的节点（触发节点）最后接受事件。为了让事件到达最终目标之前拦截事件。

   比如点击一个div，则 click 事件会按这种顺序触发: document => `<html>` => `<body>` => `<div>` ，即由 document 捕获后沿着 DOM 树依次向下传播，**并在各节点上触发捕获事件**，直到到达实际目标元素。

2. **目标阶段**

   当事件到达目标节点的，事件就进入了目标阶段。**事件在目标节点上被触发**(执行事件对应的函数)，然后会逆向回流，直到传播至最外层的文档节点。

3. **冒泡阶段**

   事件在目标元素上触发后，会继续随着 DOM 树一层层往上冒泡，直到到达最外层的根节点。

所有事件都要经历捕获阶段和目标阶段，但有些事件会跳过冒泡阶段，比如元素获得焦点 focus 和失去焦点 blur 不会冒泡

扩展一

e.target 和 e.currentTarget 区别?

- `e.target` 指向触发事件监听的对象。
- `e.currentTarget` 指向添加监听事件的对象。

例如：

```javascript 复制代码
<ul>
  <li><span>hello 1</span></li>
</ul>

let ul = document.querySelectorAll('ul')[0]
let aLi = document.querySelectorAll('li')
ul.addEventListener('click',function(e){
  let oLi1 = e.target
  let oLi2 = e.currentTarget
  console.log(oLi1)   //  被点击的li
  console.log(oLi2)   // ul
  console.og(oLi1===oLi2)  // false
})
```

给 ul 绑定了事件，点击其中 li 的时候，target 就是被点击的 li， currentTarget 就是被绑定事件的 ul

事件冒泡阶段（上述例子），`e.currenttarget` 和 `e.target` 是不相等的，但是在事件的目标阶段，`e.currenttarget` 和 `e.target` 是相等的

作用：

`e.target` 可以用来实现事件委托，该原理是通过事件冒泡（或者事件捕获）给父元素添加事件监听，e.target指向引发触发事件的元素

扩展二

addEventListener 参数

语法：

```scss
addEventListener(type, listener);
addEventListener(type, listener, options || useCapture);
```

- type: 监听事件的类型，如：'click'/'scroll'/'focus'
- listener: 必须是一个实现了 `EventListener` 接口的对象，或者是一个函数。当监听的事件类型被触发时，会执行
- options：指定 listerner 有关的可选参数对象
  - capture: 布尔值，表示 listener 是否在事件捕获阶段传播到 EventTarget 时触发
  - once：布尔值，表示 listener 添加之后最多调用一次，为 true 则 listener 在执行一次后会移除
  - passive: 布尔值，表示 listener 永远不会调用 `preventDefault()`
  - signal：可选，`AbortSignal`，当它的 `abort()` 方法被调用时，监听器会被移除
- useCapture：布尔值，默认为 false，listener 在事件冒泡阶段结束时执行，true 则表示在捕获阶段开始时执行。作用就是更改事件作用的时机，方便拦截/不被拦截。

Vue篇

vue和react的区别

1、数据可变性

- React 推崇函数式编程，数据不可变以及单向数据流，只能通过 `setState` 或者 `onchange` 来实现视图更新
- Vue 基于数据可变，设计了响应式数据，通过监听数据的变化自动更新视图

2、写法

- React 推荐使用 jsx + inline style的形式，就是 all in js

- Vue 是单文件组件（SFC）形式，在一个组件内分模块(tmplate/script/style)，当然vue也支持jsx形式，可以在开发vue的ui组件库时使用

3、diff 算法

- Vue2采用双端比较，Vue3采用快速比较
- `react`主要使用diff队列保存需要更新哪些DOM，得到patch树，再统一操作批量更新DOM。，需要使用`shouldComponentUpdate()`来手动优化react的渲染。

扩展：了解 react hooks吗

组件类的写法很重，层级一多很难维护。

函数组件是纯函数，不能包含状态，也不支持生命周期方法，因此无法取代类。

React Hooks 的设计目的，就是加强版函数组件，完全不使用"类"，就能写出一个全功能的组件

React Hooks 的意思是，组件尽量写成纯函数，如果需要外部功能和副作用，就用钩子把外部代码"钩"进来。

vue组件通信方式

- props / $emit
- ref / $refs
- *parent*/root
- attrs / listeners
- eventBus / vuex / pinia / localStorage / sessionStorage / Cookie / window
- provide / inject

vue 渲染列表为什么要加key?

Vue 在处理更新同类型 vnode 的一组子节点（比如v-for渲染的列表节点）的过程中，为了减少 DOM 频繁创建和销毁的性能开销：

对没有 key 的子节点数组更新是通过**就地更新**的策略。它会通过对比新旧子节点数组的长度，先以比较短的那部分长度为基准，将新子节点的那一部分直接 patch 上去。然后再判断，如果是新子节点数组的长度更长，就直接将新子节点数组剩余部分挂载；如果是新子节点数组更

短，就把旧子节点多出来的那部分给卸载掉）。**所以如果子节点是组件或者有状态的 DOM 元素，原有的状态会保留，就会出现渲染不正确的问题。**

有 key 的子节点更新是调用的 `patchKeyedChildren`，这个函数就是大家熟悉的实现核心 diff 算法的地方，大概流程就是同步头部节点、同步尾部节点、处理新增和删除的节点，最后用求解最长递增子序列的方法区处理未知子序列。是为了**最大程度实现对已有节点的复用，减少 DOM 操作的性能开销**，同时避免了就地更新带来的子节点状态错误的问题。

综上，如果是用 v-for 去遍历常量或者子节点是诸如纯文本这类没有"状态"的节点，是可以使用不加 key 的写法的。但是实际开发过程中更推荐统一加上 key，能够实现更广泛场景的同时，避免了可能发生的状态更新错误，我们一般可以使用 ESlint 配置 key 为 v-for 的必需元素。

想详细了解这个知识点的可以去看看我之前写的文章：v-for 到底为啥要加上 key?

vue3 相对 vue2的响应式优化

vue2使用的是 `Object.defineProperty` 去监听对象属性值的变化，但是它不能监听对象属性的新增和删除，所以需要使用 `$set` 、 `$delete` 这种语法糖去实现，这其实是一种设计上的不足。

所以 vue3 采用了 `proxy` 去实现响应式监听对象属性的增删查改。

其实从api的原生性能上 `proxy` 是比 `Object.defineProperty` 要差的。

而 vue 做的响应式性能优化主要是在将嵌套层级比较深的对象变成响应式的这一过程。

vue2的做法是在组件初始化的时候就递归执行 `Object.defineProperty` 把子对象变成响应式的；

而vue3是在访问到子对象属性的时候，才会去将它转换为响应式。这种延时定义子对象响应式会对性能有一定的提升

Vue 核心diff流程

前提：当同类型的 vnode 的子节点都是一组节点（数组类型）的时候，

步骤：会走核心 diff 流程

Vue3是快速选择算法

- 同步头部节点
- 同步尾部节点
- 新增新的节点
- 删除多余节点
- 处理未知子序列（贪心 + 二分处理最长递增子序列）

Vue2是双端比较算法

在新旧字节点的头尾节点，也就是四个节点之间进行对比，找到可复用的节点，不断向中间靠拢的过程

diff目的：diff 算法的目的就是为了尽可能地复用节点，减少 DOM 频繁创建和删除带来的性能开销

vue双向绑定原理

基于 MVVM 模型，viewModel(业务逻辑层)提供了**数据变化后更新视图**和**视图变化后更新数据**这样一个功能，就是传统意义上的双向绑定。

Vue2.x 实现双向绑定核心是通过三个模块：Observer监听器、Watcher订阅者和Compile编译器。

首先监听器会监听所有的响应式对象属性，编译器会将模板进行编译，找到里面动态绑定的响应式数据并初始化视图；watchr 会去收集这些依赖；当响应式数据发生变更时Observer就会通知 Watcher；watcher接收到监听器的信号就会执行更新函数去更新视图；

vue3的变更是数据劫持部分使用了porxy 替代 Object.defineProperty，收集的依赖使用组件的副作用渲染函数替代watcher

v-model 原理

vue2 v-model 原理剖析

V-model 是用来监听用户事件然后更新数据的语法糖。

其本质还是单向数据流，内部是通过绑定元素的 value 值向下传递数据，然后通过绑定 input 事件，向上接收并处理更新数据。

单向数据流：父组件传递给子组件的值子组件不能修改，只能通过emit事件让父组件自个改。

```ini 复制代码
// 比如
<input v-model="sth" />
// 等价于
<input :value="sth" @input="sth = $event.target.value" />
```

给组件添加 `v-model` 属性时，默认会把 `value` 作为组件的属性，把 `input` 作为给组件绑定事件时的事件名:

```xml 复制代码
// 父组件
<my-button v-model="number"></my-button>

// 子组件
<script>
export default {
  props: {
    value: Number, //  属性名必须是 value
  },

  methods: {
    add() {
      this.$emit('input', this.value + 1) // 事件名必须是 input
    },
  }
}
</script>
```

如果想给绑定的 value 属性和 input 事件换个名称呢？可以这样：

在 Vue 2.2 及以上版本，你可以在定义组件时通过 model 选项的方式来定制 prop/event：

```xml 复制代码
<script>
export default {
  model: {
    prop: 'num', // 自定义属性名
    event: 'addNum' // 自定义事件名
  }
}
```

vue3 v-model 原理

实现和 vue2 基本一致

```ini
<Son v-model="modalValue"/>
```

等同于

```ruby
<Son v-model="modalValue"/> <Son :modalValue="modalValue" @update:modalValue="modalUpdate=$event.tar
```

自定义 model 参数

```javascript
<Son v-model:visible="visible"/>
setup(props, ctx){
    ctx.emit("update:visible", false)
}
```

vue 响应式原理

不管vue2 还是 vue3，响应式的核心就是观察者模式 + 劫持数据的变化，在访问的时候做依赖收集和在修改数据的时候执行收集的依赖并更新数据。具体点就是：

vue2 的话采用的是 `Object.definePorperty` 劫持对象的 get 和 set 方法，每个组件实例都会在渲染时初始化一个 watcher 实例，它会将组件渲染过程中所接触的响应式变量记为依赖，并且保存了组件的更新方法 update。当依赖的 setter 触发时，会通知 watcher 触发组件的 update 方法，从而更新视图。

Vue3 使用的是 ES6 的 proxy，proxy 不仅能够追踪属性的获取和修改，还可以追踪对象的增删，这在 vue2中需要 *set*/delete 才能实现。然后就是收集的依赖是用组件的副作用渲染函数替代 watcher 实例。

性能方面，从原生 api 角度，proxy 这个方法的性能是不如 Object.property，但是 vue3 强就强在一个是上面提到的可以追踪对象的增删，第二个是对嵌套对象的处理上是访问到具体属性才会把那个对象属性给转换成响应式，而 vue2 是在初始化的时候就递归调用将整个对象和他的属性都变成响应式，这部分就差了。

扩展一

vue2 通过数组下标更改数组视图为什么不会更新？

尤大：性能不好

注意：vue3 是没问题的

why 性能不好?

我们看一下响应式处理:

```scss
export class Observer {
  this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
      // 这里对数组进行单独处理
      if (hasProto) {
        protoAugment(value, arrayMethods)
      } else {
        copyAugment(value, arrayMethods, arrayKeys)
      }
      this.observeArray(value)
    } else {
      // 对对象遍历所有键值
      this.walk(value)
    }
  }
  walk (obj: Object) {
    const keys = Object.keys(obj)
    for (let i = 0; i < keys.length; i++) {
      defineReactive(obj, keys[i])
    }
  }
  observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])
    }
  }
}
```

对于对象是通过 `Object.keys()` 遍历全部的键值，对数组只是 `observe` 监听已有的元素，所以通过下标更改不会触发响应式更新。

理由是数组的键相较对象多很多，当数组数据大的时候性能会很拉胯。所以不开放

computed 和 watch

Computed 的大体实现和普通的响应式数据是一致的，不过加了延时计算和缓存的功能：

在访问computed对象的时候，会触发 getter ，初始化的时候将 computed 属性创建的 watcher （vue3是副作用渲染函数）添加到与之相关的响应式数据的依赖收集器中（dep），然后根据里面一个叫 dirty 的属性判断是否要收集依赖，不需要的话直接返回上一次的计算结果，需要的话就执行更新重新渲染视图。

watchEffect?

watchEffect会自动收集回调函数中响应式变量的依赖。并在首次自动执行

推荐在大部分时候用 `watch` 显式的指定依赖以避免不必要的重复触发，也避免在后续代码修改或重构时不小心引入新的依赖。 `watchEffect` 适用于一些逻辑相对简单，依赖源和逻辑强相关的场景（或者懒惰的场景 ）

$nextTick 原理?

vue有个机制，更新 DOM 是异步执行的，当数据变化会产生一个异步更行队列，要等异步队列结束后才会统一进行更新视图，所以改了数据之后立即去拿 dom 还没有更新就会拿不到最新数据。所以提供了一个 nextTick 函数，它的回调函数会在DOM 更新后立即执行。

nextTick 本质上是个异步任务，由于事件循环机制，异步任务的回调总是在同步任务执行完成后才得到执行。所以源码实现就是根据环境创建异步函数比如 Promise.then（浏览器不支持promise就会用MutationObserver，浏览器不支持MutationObserver就会用setTimeout），然后调用异步函数执行回调队列。

所以项目中不使用$nextTick的话也可以直接使用Promise.then或者SetTimeout实现相同的效果

Vue 异常处理

1、全局错误处理： `Vue.config.errorHandler`

```
Vue.config.errorHandler = function(err, vm, info) {};
```

如果在组件渲染时出现运行错误，错误将会被传递至全局 `Vue.config.errorHandler` 配置函数(如果已设置)。

比如前端监控领域的 sentry，就是利用这个钩子函数进行的 vue 相关异常捕捉处理

2、**全局警告处理**：`Vue.config.warnHandler`

```ini 复制代码
Vue.config.warnHandler = function(msg, vm, trace) {};
```

注意：仅在开发环境生效

像在模板中引用一个没有定义的变量，它就会有warning

3、**单个vue 实例错误处理**：`renderError`

```javascript 复制代码
const app = new Vue({
    el: "#app",
    renderError(h, err) {
        return h("pre", { style: { color: "red" } }, err.stack);
    }
});
```

和组件相关，只适用于开发环境，这个用处不是很大，不如直接看控制台

4、**子孙组件错误处理**：`errorCaptured`

```typescript 复制代码
Vue.component("cat", {
    template: `<div><slot></slot></div>`,
    props: { name: { type: string } },
    errorCaptured(err, vm, info) {
        console.log(`cat EC: ${err.toString()}\ninfo: ${info}`);
        return false;
    }
});
```

注：只能在组件内部使用，用于捕获子孙组件的错误，一般可以用于组件开发过程中的错误处理

5、**终极错误捕捉**：`window.onerror`

```ini 复制代码
window.onerror = function(message, source, line, column, error) {};
```

它是一个全局的异常处理函数，可以抓取所有的 JavaScript 异常

Vuex 流程 & 原理

Vuex 利用 vue 的mixin 机制，在beforeCreate 钩子前混入了 vuexinit 方法，这个方法实现了将 store 注入 vue 实例当中，并注册了 store 的引用属性 *store*，所以可以使用‘*this*.store.xxx`去引入vuex中定义的内容。

然后 state 是利用 vue 的 data，通过 `new Vue({data: {$$state: state}}` 将 state 转换成响应式对象，然后使用 computed 函数实时计算 getter

Vue.use函数里面具体做了哪些事

概念

可以通过全局方法 `Vue.use()` 注册插件，并能阻止多次注册相同插件，它需要在 `new Vue` 之前使用。

该方法第一个参数必须是 `Object` 或 `Function` 类型的参数。如果是 `Object` 那么该 `Object` 需要定义一个 `install` 方法；如果是 `Function` 那么这个函数就被当做 `install` 方法。

`Vue.use()` 执行就是执行 `install` 方法，其他传参会作为 `install` 方法的参数执行。

所以** `Vue.use()` 本质就是执行需要注入插件的 `install` 方法**。

源码实现

javascript 复制代码

```
export function initUse (Vue: GlobalAPI) {
 Vue.use = function (plugin: Function | Object) {
  const installedPlugins = (this._installedPlugins || (this._installedPlugins = []))
  // 避免重复注册
  if (installedPlugins.indexOf(plugin) > -1) {
   return this
  }
  // 获取传入的第一个参数
  const args = toArray(arguments, 1)
  args.unshift(this)
  if (typeof plugin.install === 'function') {
   // 如果传入对象中的install属性是个函数则直接执行
   plugin.install.apply(plugin, args)
  } else if (typeof plugin === 'function') {
   // 如果传入的是函数，则直接（作为install方法）执行
   plugin.apply(null, args)
  }
```

```
  // 将已经注册的插件推入全局installedPlugins中
  installedPlugins.push(plugin)
  return this
 }
}
```

使用方式

```
javascript 复制代码
installedPlugins import Vue from 'vue'
import Element from 'element-ui'
Vue.use(Element)
```

怎么编写一个vue插件

要暴露一个 `install` 方法，第一个参数是 `Vue` 构造器，第二个参数是一个可选的配置项对象

```
javascript 复制代码
Myplugin.install = function(Vue, options = {}) {
  // 1、添加全局方法或属性
  Vue.myGlobalMethod = function() {}
  // 2、添加全局服务
  Vue.directive('my-directive', {
    bind(el, binding, vnode, pldVnode) {}
  })
  // 3、注入组件选项
  Vue.mixin({
    created: function() {}
  })
  // 4、添加实例方法
  Vue.prototype.$myMethod = function(methodOptions) {}
}
```

CSS篇

什么是 BFC

Block Formatting context，块级格式上下文

BFC 是一个独立的渲染区域，相当于一个容器，在这个容器中的样式布局不会受到外界的影响。

比如浮动元素、绝对定位、overflow 除 visble 以外的值、display 为 inline/tabel-cells/flex 都能构建 BFC。

常常用于解决

1. 处于同一个 BFC 的元素外边距会产生重叠（此时需要将它们放在不同 BFC 中）；
2. 清除浮动（float），使用 BFC 包裹浮动的元素即可
3. 阻止元素被浮动元素覆盖，应用于两列式布局，左边宽度固定，右边内容自适应宽度（左边float，右边 overflow）

伪类和伪元素及使用场景

伪类

伪类即：当元素处于特定状态时才会运用的特殊类

开头为冒号的选择器，用于选择处于特定状态的元素。比如 `:first-child` 选择第一个子元素； `:hover` 悬浮在元素上会显示； `:focus` 用键盘选定元素时激活； `:link` + `:visted` 点击过的链接的样式; `:not` 用于匹配不符合参数选择器的元素； `:fist-child` 匹配元素的第一个子元素； `:disabled` 匹配禁用的表单元素

伪元素

伪元素用于创建一些不在文档树中的元素，并为其添加样式。比如说，我们可以通过 `::before` 来在一个元素前增加一些文本，并为这些文本添加样式。虽然用户可以看到这些文本，但是这些文本实际上不在文档树中。示例：

`::before` 在被选元素前插入内容。需要使用 content 属性来指定要插入的内容。被插入的内容实际上不在文档树中

```css
h1:before {
    content: "Hello ";
}
```

`::first-line` 匹配元素中第一行的文本

src 和 href 区别

- href是Hypertext Reference的简写，表示超文本引用，指向网络资源所在位置。href 用于在当前文档和引用资源之间确立联系
- src是source的简写，目的是要把文件下载到html页面中去。src 用于替换当前内容
- 浏览器解析方式

  当浏览器遇到href会并行下载资源并且不会停止对当前文档的处理。(同时也是为什么建议使用 link 方式加载 CSS，而不是使用 @import 方式)

  当浏览器解析到src ，会暂停其他资源的下载和处理，直到将该资源加载或执行完毕。(这也是script标签为什么放在底部而不是头部的原因)

不定宽高元素的水平垂直居中

- flex

```scss
<div class="wrapper flex-center">
  <p>horizontal and vertical</p>
</div>

.wrapper {
    width: 900px;
    height: 300px;
    border: 1px solid #ccc;
}
.flex-center {  // 注意是父元素
    display: flex;
    justify-content: center;  // 主轴（竖线）上的对齐方式
    align-items: center;      // 交叉轴（横轴）上的对齐方式
}
```

- flex + margin

```css
<div class="wrapper">
  <p>horizontal and vertical</p>
</div>

.wrapper {
    width: 900px;
    height: 300px;
    border: 1px solid #ccc;
```

```
    display: flex;
}
.wrapper > p {
    margin: auto;
}
```

- Transform + absolute

```
<div class="wrapper">
    <img src="test.png">
</div>

.wrapper {
    width: 300px;
    height: 300px;
    border: 1px solid #ccc;
    position: relative;
}
.wrapper > img {
    position: absolute;
    left: 50%;
    top: 50%;
    tansform: translate(-50%, -50%)
}
```

注：使用该方法只适用于行内元素(a、img、label、br、select等)（宽度随元素的内容变化而变化），用于块级元素（独占一行）会有问题，left/top 的50%是基于图片最左侧的边来移动的，tanslate会将多移动的图片自身的半个长宽移动回去，就实现了水平垂直居中的效果

- display: table-cell

```
<div class="wrapper">
  <p>absghjdgalsjdbhaksldjba</p>
</div>

.wrapper {
    width: 900px;
    height: 300px;
    border: 1px solid #ccc;
    display: table-cell;
    vertical-align: middle;
    text-align: center;
}
```

浏览器和网络篇

跨页面通信的方法?

这里分了同源页面和不同源页面的通信。

不同源页面可以通过 iframe 作为一个桥梁，因为 iframe 可以指定 origin 来忽略同源限制，所以可以在每个页面都嵌入同一个 iframe 然后监听 iframe 中传递的 message 就可以了。

同源页面的通信大致分为了三类：广播模式、共享存储模式和口口相传模式

第一种广播模式，就是可以通过 BroadCast Channel、Service Worker 或者 localStorage 作为广播，然后去监听广播事件中消息的变化，达到页面通信的效果。

第二种是共享存储模式，我们可以通过Shared Worker 或者 IndexedDB，创建全局共享的数据存储。然后再通过轮询去定时获取这些被存储的数据是否有变更，达到一个的通信效果。像常见cookie 也可以作为实现共享存储达到页面通信的一种方式

最后一种是口口相传模式，这个主要是在使用 window.open 的时候，会返回被打开页面的 window 的引用，而在被打开的页面可以通过 window.opener 获取打开它的页面的 window 点引用，这样，多个页面之间的 window 是能够相互获取到的，传递消息的话通过 postMessage 去传递再做一个事件监听就可以了

详细说说 HTTP 缓存

在浏览器第一次发起请求服务的过程中，会根据响应报文中的缓存标识决定是否缓存结果，是否将缓存标识和请求结果存入到浏览器缓存中。

HTTP 缓存分为强制缓存和协商缓存两类。

强制缓存就是请求的时候浏览器向缓存查找这次请求的结果，这里分了三种情况，没查找到直接发起请求（和第一次请求一致）；查找到了并且缓存结果还没有失效就直接使用缓存结果；查找到但是缓存结果失效了就会使用协商缓存。

强制缓存有 Expires 和 Cache-control 两个缓存标识，Expires 是http/1.0 的字段，是用来指定过期的具体的一个时间（如 Fri, 02 Sep 2022 08:03:35 GMT），当服务器时间和浏览器时间不一致的话，就会出现问题。所以在 http1.1 添加了 cache-control 这个字段，它的值规定了

缓存的范围（public/private/no-cache/no-store），也可以规定缓存在xxx时间内失效（max-age=xxx）是个相对值，就能避免了 expires带来的问题。

协商缓存就是强制缓存的缓存结果失效了，浏览器携带缓存标识向服务器发起请求，有服务器通过缓存标识决定是否使用缓存的过程。

控制协商缓存的字段有 last-modified / if-modified-since 和 Etag / if-none-match，后者优先级更高。

大致过程就是通过请求报文传递 last-modified 或 Etag 的值给服务器与服务器中对应值作对比，若和响应报文中的 if-modified-since 或 if-none-match 结果一致，则协商缓存有效，使用缓存结果，返回304；否则失效，重新请求结果，返回200

输入 URL 到页面展现的全过程

用户输入一段内容后，浏览器会先去判断这段内容是搜索内容还是 URL ，是搜索内容的话就会接合默认的搜索引擎生成 URL，比如 google 浏览器是goole.com/search?xxxx，如果是 URL 会拼接协议，比如 http/https。当页面没有监听 beforeupload 时间或者同意了继续执行流程，浏览器图标栏会进入加载中的状态。

接下来浏览器进程会通过 IPC 进程间通信将 URL 请求发送给网络进程，网络进程会先去缓存中查找该资源，如果有则拦截请求并直接200返回，没有的话会进入网络请求流程。

网络请求流程是网络进程请求 DNS 服务器返回域名对应的IP和端口号（如果这些之前有缓存也是直接返回缓存结果），如果没有端口号，http默认为80，https默认为443，如果是https还需要建立 TLS 安全连接创建加密的数据通道。

接着就是 TCP 三次握手建立浏览器和服务器连接，然后进行数据传输，数据传输完成四次挥手断开连接，如果设置了 `connection: keep-alive` 就可以一直保持连接。

网络进程将通过TCP获取的数据包进行解析，首先是根据响应头的content-type来判断数据类型，如果是字节流或者文件类型的话，会交给下载管理器进行下载，这时候导航流程就结束了。如果是 text/html 类型，就会通知到浏览器进程获取文档进行渲染。

浏览器进程获取到渲染的通知，会根据当前页面和新输入的页面判断是否是同一个站点，是的话就复用之前网页创建的渲染进程，否则的话会新创建一个单独的渲染进程。

浏览器进程将“提交文档”的消息给渲染进程，渲染进程接收到消息就会和网络进程建立传输数据的通道，数据传输完成后就返回“确认提交”的信息给浏览器进程。

浏览器接收到渲染进程的“确认提交”的消息后，就会更新浏览器的页面状态：安全状态、地址栏URL、前进后退的历史消息，并更新 web页面，此时页面是空白页面（白屏）。

页面渲染过程（重点记忆）

最后是渲染进程对文档进行页面解析和子资源加载，渲染进程会将 HTML 转换成 DOM 树结构，将 css 转换成 styleSeets （ CSSOM）。然后复制 DOM 树过滤掉不显示的元素创建基本的渲染树，接着计算每个 DOM 节点的样式和计算每个节点的位置布局信息构建成布局树。

具有层叠上下文或者需要要裁剪的地方会独立创建图层，这就是分层，最终会形成一个分层树，渲染进程会给每个图层生成绘制列表并提交给合成线程，合成线程将图层分成图块（避免一次性绘制图层所有内容，可以根据图块优先需渲染视口部分），并在光栅化线程池中将图块转换成位图。

转换完毕后合成线程发送绘制图块命令 DrawQuard 给浏览器进程，浏览器根据 DrawQuard 消息生成页面，并显示到浏览器上。

速记：

浏览器的渲染进程将 html 解析成 dom树，将 css 解析成 cssom 树，然后会先复制一份 DOM 树过滤掉不显示的元素（比如 display: none），再和 cssom 结合进行计算每个 dom 节点的布局信息构建成一个布局树。

布局树生成完毕就会根据图层的层叠上下文或者裁剪部分进行分层，形成一个分层树。

渲染进程再将每个图层生成绘制列表并提交给合成线程，合成线程为了避免一次性渲染，就是分块渲染，会将图层分成图块，并通过光栅化线程池将图块转换成位图。

转换完毕后，合成线程将绘制图块的命令发送给浏览器进行显示

TCP 和 UDP 的区别

UDP 是**用户数据包协议**（User Dataprogram Protocol），IP 通过 IP 地址信息把数据包传送给指定电脑后，UDP 可以通过端口号把数据包分发给正确的程序。UDP 可以校验数据是否正确，但没有重发的机制，只会丢弃错误的数据包，同时 UDP 在发送之后无法确认是否到达目的地。UDP 不能保证数据的可靠性，但是传输的速度非常快，通常运用于在线视频、互动游戏这些不那么严格保证数据完整性的领域。

TCP 是为了解决 UDP 的数据容易丢失，且无法正确组装数据包二引入的传输控制协议（Transmission Control Protocol），**是一种面向连接的，可靠的，基于字节流的传输层通信协议。**TCP 在处理数据包丢失的情况，提供了重传机制；并且 TCP 引入了数据包排序机制，可以将乱序的数据包组合成完整的文件。

TCP 头除了包含目标端口和本机端口号外，还提供了用于排序的序列号，以便接收端通过序号来重排数据包。

TCP 和 UDP 的区别

一个 TCP 连接的生命周期会经历链接阶段，数据传输和断开连接阶段三个阶段。

连接阶段

用来建立客户端和服务器之间的链接，通过三次握手用来确认客户端、服务端相互之间的数据包收发能力。

1、客户端先发送 SYN 报文用来确认服务端能够发数据，并进入 SYN_SENT 状态等待服务端确认

2、服务端收到 SYN 报文，会向客户端发送一个 ACK 确认报文，同时服务端也会向客户端发送 SYN 报文用来确认客户端是否能够发送数据，此时服务端进入 SYN_RCVD 状态

3、客户端接收到 ACK + SYN 的报文，就会向服务端发送数据包并进入 ESTABLISHED 状态（建立连接）；服务端接收到客户端发送的 ACK 包也会进入 ESTABLISHED 状态，完成三次握手

传输数据阶段

该阶段，接收端需要对每个包进行确认操作；

所以当发送端发送了一个数据包之后，在规定时间内没有接收到接收端反馈的确认消息，就会判断为包丢失，从而触发重发机制；

一个大文件在传输过程中会分为很多个小数据包，数据包到达接收端后会根据 TCP 头中的序号为其排序，保证数据的完整。

断开连接阶段

通过四次挥手，来保证双方的建立的连接能够断开

1、客户端向服务器发起 FIN 包，并进入 FIN_WAIT_1 状态

2、服务端收到 FIN 包，发出确认包 ACK，并带上自己的序号，服务端进入 CLOSE_WAIT 状态。这时候客户端已经没有数据要发给服务端了，但是服务端如果有数据要发给客户端，客户端还是需要接收。客户端收到 ACK 后进入 FIN_WAIT_2 状态

3、服务端数据发送完毕后，向客户端发送 FIN 包，此时服务器进入 LAST_ACK 状态

4、客户端收到 FIN 包发出确认包 ACK ，此时客户端进入 TIME_WAIT 状态，等待 2 MSL 后进入 CLOSED 状态；服务端接收到客户端的 ACK 后就进入 CLOSED 状态了。

对于四次挥手，因为 TCP 是全双工通信，在主动关闭方发送 FIN 包后，接收端可能还要发送数据，不能立即关闭服务器端到客户端的数据通道，所以也就不能将服务器端的 `FIN` 包与对客户端的 `ACK` 包合并发送，只能先确认 `ACK`，然后服务器待无需发送数据时再发送 `FIN` 包，所以四次挥手时必须是四次数据包的交互

Content-length 了解吗?

Content-length 是 http 消息长度，用十进制数字表示的字节的数目。

如果 content-length > 实际长度，服务端/客户端读取到消息队尾时会继续等待下一个字节，会出现无响应超时的情况

如果 content-length < 实际长度，首次请求的消息会被截取，然后会导致后续的数据解析混乱。

当不确定content-length的值应该使用Transfer-Encoding: chunked，能够将需要返回的数据分成多个数据块，直到返回长度为 0 的终止块

跨域常用方案

什么是跨域?

协议 + 域名 + 端口号均相同时则为同域，任意一个不同则为跨域

解决方案

1、 传统的jsonp：利用 `<script>` 标签没有跨域限制的特点，仅支持 get接口，应该没有人用这个了

2、 一般使用 cors（跨域资源共享）来解决跨域问题，浏览器在请求头中发送origin字段指明请求发起的地址，服务端返回Access-control-allow-orign，如果一致的话就可以进行跨域访问

3、 Iframe 解决主域名相同，子域名不同的跨域请求

4、 浏览器关闭跨域限制的功能

5、 http-proxy-middleware 代理

预检

补充：http会在跨域的时候发起一次**预检**请求，“需预检的请求”要求必须首先使用OPTIONS方法发起一个预检请求到服务器，以获知服务器是否允许该实际请求。“预检请求”的使用，可以避免跨域请求对服务器的用户数据产生未预期的影响。

withCredentials为 true不会产生预请求；content-type为application/json会产生预请求；设置了用户自定义请求头会产生预检请求；delete方法会产生预检请求；

XSS 和 CSRF

xss基本概念

Xss (Cross site scripting)跨站脚本攻击，为了和 css 区别开来所以叫 xss

Xss 指黑客向 html 或 dom 中注入恶意脚本，从而在用户浏览页面的时候利用注入脚本对用户实施攻击的手段

恶意脚本可以做到：窃取 cookie 信息、监听用户行为（比如表单的输入）、修改DOM（比如伪造登录界面骗用户输入账号密码）、在页面生成浮窗广告等

恶意脚本注入方式：

1. 存储型 xss

   黑客利用站点漏洞将恶意 js 代码提交到站点服务器，用户访问页面就会导致恶意脚本获取用户的cookie等信息。

2. 反射性 xss

   用户将一段恶意代码请求提交给 web 服务器，web 服务器接收到请求后将恶意代码反射到浏览器端

3. 基于 DOM 的 xss 攻击

   通过网络劫持在页面传输过程中更改 HTML 内容

前两种属于服务端漏洞，最后一种属于前端漏洞

防止xss攻击的策略

1、服务器对输入脚本进行过滤或者转码，比如将 `code:<script>alert('你被xss攻击了')</script>` 转换成 `code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;`

2、充分利用内容安全策略 CSP(content-security-policy)，可以通过 http 头信息的 content-security-policy 字段控制可以加载和执行的外部资源；或者通过html的meta 标签 `<meta http-equiv="Content-Security-Policy" content="script-src 'self'; object-src 'none'; style-src cdn.example.org third-party.org; child-src https:">`

3、cookie设置为 http-only, cookie 就无法通过 `document.cookie` 来读取

csrf基本概念

Csrf（cross site request forgery）跨站请求伪造，指黑客引导用户访问黑客的网站。

CSRF 是指黑客引诱用户打开黑客的网站，在黑客的网站中，利用用户的登录状态发起的跨站请求。简单来讲，CSRF 攻击就是黑客利用了用户的登录状态，并通过第三方的站点来做一些坏事。

Csrf 攻击场景

1. 自动发起 get 请求

   比如黑客网站有个图片:

```ini
<img src="https://time.geekbang.org/sendcoin?user=hacker&number=100">
```

黑客将转账的请求接口隐藏在 img 标签内，欺骗浏览器这是一张图片资源。当该页面被加载时，浏览器会自动发起 img 的资源请求，如果服务器没有对该请求做判断的话，那么服务器就会认为该请求是一个转账请求，于是用户账户上的 100 极客币就被转移到黑客的账户上去了。

2. 自动发起 post 请求

   黑客在页面中构建一个隐藏的表单，当用户点开链接后，表单自动提交

3. 引诱用户点击链接

   比如页面上放了一张美女图片，下面放了图片下载地址，而这个下载地址实际上是黑客用来转账的接口，一旦用户点击了这个链接，那么他的极客币就被转到黑客账户上了

防止csrf方法

1、设置 cookie 时带上SameSite: strict/Lax选项

2、验证请求的来源站点，通过 origin 和 refere 判断来源站点信息

3、csrf token，浏览器发起请求服务器生成csrf token，发起请求前会验证 csrf token是否合法。第三方网站肯定是拿不到这个token。我们的 csrf token 是前后端约定好后写死的。

websocket

websocket是一种支持双向通信的协议，就是服务器可以主动向客户端发消息，客户端也可以主动向服务器发消息。

它是基于 HTTP 协议来建立连接的的，与http协议的兼容性很好，所以能通过 HTTP 代理服务器；没有同源限制。

WebSocket 是一种事件驱动的协议，这意味着可以将其用于真正的实时通信。与 HTTP 不同（必须不断地请求更新），而使用 websockets，更新在可用时就会立即发送

当连接终止时，WebSockets 不会自动恢复，这是应用开发中需要自己实现的机制，也是存在许多客户端开源库的原因之一。

像webpack和vite的devServer就使用了websocket实现热更新

Post 和 Get 区别

- **应用场景：** (GET 请求是一个**幂等**的请求)一般 Get 请求用于对服务器资源不会产生影响的场景，比如说请求一个网页的资源。(而 Post 不是一个**幂等**的请求)一般用于对服务器资源会产生影响的情景，比如注册用户这一类的操作。（**幂等是指一个请求方法执行多次和仅执行一次的效果完全相同**）
- **是否缓存：** 因为两者应用场景不同，浏览器一般会对 Get 请求缓存，但很少对 Post 请求缓存。
- **传参方式不同：** Get 通过查询字符串传参，Post 通过请求体传参。
- **安全性：** Get 请求可以将请求的参数放入 url 中向服务器发送，这样的做法相对于 Post 请求来说是不太安全的，因为请求的 url 会被保留在历史记录中。
- **请求长度：** 浏览器由于对 url 长度的限制，所以会影响 get 请求发送数据时的长度。这个限制是浏览器规定的，并不是 RFC 规定的。
- **参数类型：** get参数只允许ASCII字符，post 的参数传递支持更多的数据类型(如文件、图片)。

性能优化篇

性能优化有哪些手段

1. 从缓存的角度
    - 将一些不常变的大数据通过localstorage/sessionStorage/indexdDB 进行读取
    - 活用 http 缓存（强缓存和协商缓存），将内容存储在内存或者硬盘中，减少对服务器端的请求
2. 网络方面比较常用的是静态资源使用 CDN
3. 打包方面
    - 路由的按需加载
    - 优化打包后资源的大小
    - 开启 gzip 压缩资源
    - 按需加载三方库
4. 代码层面
    - 减少不必要的请求，删除不必要的代码

- 避免耗时过长的js处理阻塞主线程（耗时且无关 DOM 可以丢到web worker去处理或者拆分成小的任务）
- 图片可以使用懒加载的方式，长列表使用虚拟滚动

5. 首屏速度提升

   - 代码压缩，减少打包的静态资源体积(Terser plugin/MiniCssExtratplugin)
   - 路由懒加载，首屏就只会请求第一个路由的相关资源
   - 使用 cdn加速第三方库，我们是toB的产品，会需要部署在内网，所以一般不用，toC用的多
   - ssr 服务端渲染，由服务器直接返回拼接好的html页面

6. vue 常见的性能优化方式

   - 图片懒加载： vue-lazyLoad
   - 虚拟滚动
   - 函数式组件
   - v-show/ keep-alive 复用 dom
   - deffer延时渲染组件（requestIdleCallback）
   - 时间切片 time slicing

前端监控 SDK 技术要点

1. 可以通过 `window.performance` 获取各项性能指标数据

2. 完整的前端监控平台包括：数据采集和上报、数据整理和存储、数据展示

3. 网页性能指标：

   - FP（first-paint）从页面加载到第一个像素绘制到屏幕上的时间
   - FCP（first-contentful-paint），从页面加载开始到页面内容的任何部分在屏幕上完成渲染的时间
   - LCP(largest-contentful-paint)，从页面加载到最大文本或图像元素在屏幕上完成渲染的时间

4. 以上指标可以通过PerformanceObserver获取

5. 首屏渲染时间计算：通过 `MutationObserver` 监听 `document` 对象的属性变化

如何减少回流、重绘，充分利用 GPU 加速渲染?

首先应该避免直接使用 DOM API 操作 DOM，像 vue react 虚拟 DOM 让对 DOM 的多次操作合并成了一次。

1. 样式集中改变，好的方式是使用动态 class

2. 读写操作分离，避免读后写，写后又读

```ini
// bad 强制刷新 触发四次重排+重绘
div.style.left = div.offsetLeft + 1 + 'px';
div.style.top = div.offsetTop + 1 + 'px';
div.style.right = div.offsetRight + 1 + 'px';
div.style.bottom = div.offsetBottom + 1 + 'px';


// good 缓存布局信息 相当于读写分离 触发一次重排+重绘
var curLeft = div.offsetLeft;
var curTop = div.offsetTop;
var curRight = div.offsetRight;
var curBottom = div.offsetBottom;

div.style.left = curLeft + 1 + 'px';
div.style.top = curTop + 1 + 'px';
div.style.right = curRight + 1 + 'px';
div.style.bottom = curBottom + 1 + 'px';
```

   原来的操作会导致四次重排，读写分离之后实际上只触发了一次重排，这都得益于浏览器的渲染队列机制：

   > 当我们修改了元素的几何属性，导致浏览器触发重排或重绘时。它会把该操作放进渲染队列，等到队列中的操作到了一定的数量或者到了一定的时间间隔时，浏览器就会批量执行这些操作。

3. 使用 `display: none` 后元素不会存在渲染树中，这时对它进行各种操作，然后更改 display 显示即可（示例：向2000个div中插入一个div）

4. 通过 documentFragment 创建 dom 片段，在它上面批量操作 dom ，操作完后再添加到文档中，这样只有一次重排（示例：一次性插入2000个div）

5. 复制节点在副本上操作然后替换它

6. 使用 BFC 脱离文档流，重排开销小

Css 中的`transform`、`opacity`、`filter`、`will-change`能触发硬件加速

大图片优化的方案

1. 优化请求数
    - 雪碧图，将所有图标合并成一个独立的图片文件，再通过`background-url`和`backgroun-position`来显示图标
    - 懒加载，尽量只加载用户正则浏览器或者即将浏览的图片。最简单使用监听页面滚动判断图片是否进入视野；使用 intersection Observer API；使用已知工具库；使用css的`background-url`来懒加载
    - base64，小图标或骨架图可以使用内联 base64因为 base64相比普通图片体积大。注意首屏不需要懒加载，设置合理的占位图避免抖动。
2. 减小图片大小
    - 使用合适的格式比如WebP、svg、video替代 GIF 、渐进式 JPEG
    - 削减图片质量
    - 使用合适的大小和分辨率
    - 删除冗余的图片信息
    - Svg 压缩
3. 缓存

代码优化

1. 非响应式变量可以定义在`created`钩子中使用 this.xxx 赋值
2. 访问局部变量比全局变量块，因为不需要切换作用域
3. 尽可能使用`const`声明变量，注意数组和对象
4. 使用 v8 引擎时，运行期间，V8 会将创建的对象与隐藏类关联起来，以追踪它们的属性特征。能够共享相同隐藏类的对象性能会更好，v8 会针对这种情况去优化。所以为了贴合“共享隐藏类”，我们要避免”先创建再补充“式的**动态属性复制以及动态删除属性**（使用

delete关键字）。即尽量在构造函数/对象中一次性声明所有属性。属性删除时可以设置为null，这样可以保持隐藏类不变和继续共享。

5. 避免内存泄露的方式

    - 尽可能少创建全局变量
    - 手动清除定时器
    - 少用闭包
    - 清除 DOM 引用
    - 弱引用

6. 避免强制同步，在修改 DOM 之前查询相关值

7. 避免布局抖动（一次 JS 执行过程中多次执行强制布局和抖动操作），尽量不要在修改 DOM 结构时再去查询一些相关值

8. 合理利用 css 合成动画，如果能用 css 处理就交给 css。因为合成动画会由合成线程执行，不会占用主线程

9. 避免频繁的垃圾回收，优化存储结构，避免小颗粒对象的产生

前端工程化篇

webpack的执行流程和生命周期

webpack 是为现代 JS 应用提供静态资源打包功能的 bundle。

核心流程有三个阶段: 初始化阶段、构建阶段和生成阶段

1、初始化阶段，会从配置文件、配置对象和Shell参数中读取初始化的参数并与默认配置结合成最终的参数，之以及创建 compiler 编译器对象和初始化它的运行环境

2、构建阶段，编译器会执行它的 run()方法开始编译的过程，其中会先确认 entry 入口文件，从入口文件开始搜索和入口文件有直接或者简介关联的所有文件创建依赖对象，之后再根据依赖对象创建 module 对象，这时候会使用 loader 将模块转换标准的 js 内容，再调用 js 的解释器将内容转换成 AST 对象，再从 AST 中找到该模块依赖的模块，递归本步骤知道所有入口依赖文件都经过了本步骤的处理。最后完成模块编译，得到了每个模块被翻译的内容和他们之间的关系依赖图（dependency graph），这个依赖图就是项目所有用到的模块的映射关系。

3、生成阶段，将编译后的 module 组合成 chunk ， 再把每个 chunk 转换成一个单独的文件输出到文件列表，确定好输出内容后，根据配置确定输出路径和文件名，就把文件内容写入文件系统

webpack的plugin 和loader

loader

webpack只能理解 JS 和 JSON 文件，loader 本质上就是个转换器，能将其他类型的文件转换成 webpack 识别的东西

loader 会在 webpack 的构建阶段将依赖对象创建的 module 转换成标准的 js 内容的东西。比如 vue-loader 将vue文件转换成 js 模块，图片字体通过 url-loader 转换成 data URL，这些 webpack 能够识别的东西。

可以在 module.rules 中配置不同的 loader 解析不同的文件

plugin

插件本质是一个带有 apply 函数的类`class myPlugin { apply(compiler) {} }`,这个apply 函数有个参数 compiler 是webpack 初始化阶段生成的编译器对象，可以调用编译器对象中的 hooks 注册各种钩子的回调这些 hooks 是贯穿整个编译的生命周期。所以开发者可以通过钩子回调在里面插入特定的代码，实现特定的功能。

比如stylelint plugin可以指定 stylelint 的需要检查文件类型和文件范围；HtmlWebpackPlugin 用来生成打包后的模板文件；MiniCssExtactPlugin会将所有的css提取成独立的chunks,stylelintplugin可以在开发阶段提供样式的检查功能。

webpack的hash策略

MiniCssExtractPlugin 对于浏览器来说，一方面期望每次请求页面资源时，获得的都是最新的资源；一方面期望在资源没有发生变化时，能够复用缓存对象。这个时候，使用文件名+文件哈希值的方式，就可以实现只要通过文件名，就可以区分资源是否有更新。而webpack就内置了hash计算方法，对生成文件的可以在输出文件中添加hash字段。

Webpack 内置 hash 有三种

- **hash**: 项目每次构建都会生成一个hash，和整个项目有关，项目任意地方有改变就会改变

hash会更据每次工程的内容进行计算，很容易造成不必要的hash变更，不利于版本管理。一般来说，没有什么机会直接使用hash。

- **content hash**: 和单个文件的内容相关。指定文件的内容发生改变，就会改变hash，内容不变hash 值不变

  对于css文件来说，一般会使用MiniCssExtractPlugin将其抽取为一个单独的css文件。

  此时可以使用contenthash进行标记，确保css文件内容变化时，可以更新hash。

- **chunk hash**：和webpack打包生成的chunk相关。每一个entry，都会有不同的hash。

  一般来说，针对于输出文件，我们使用chunkhash。

  因为webpack打包后，最终每个entry文件及其依赖会生成单独的一个js文件。

  此时使用chunkhash，能够保证整个打包内容的更新准确性。

扩展：file-loader 的 hash 可能有同学会表示有以下疑问。

明明经常看到在处理一些图片，字体的file-loader的打包时，使用的是[name]_[hash:8].[ext]

但是如果改了其他工程文件，比如index.js，生成的图片hash并没有变化。

这里需要注意的是，file-loader的hash字段，这个loader自己定义的占位符，和webpack的内置hash字段并不一致。

这里的hash是使用md4等hash算法，对文件内容进行hash。

所以只要文件内容不变，hash还是会保持一致。

vite原理

Vite 主要由两个部分组成

1. 开发环境

   Vite 利用浏览器去解析 imports，在服务器端按需编译返回，完全跳过了打包这个概念，服务器随起随用（就相当于把我们在开发的文件转换成 ESM 格式直接发送给浏览器）

当浏览器解析 import HelloWorld from './components/HelloWorld.vue' 时，会向当前域名发送一个请求获取对应的资源（ESM支持解析相对路径）,浏览器直接下载对应的文件然后解析成模块记录（打开 network 面板可以看到响应数据都是 ESM 类型的 js）。然后实例化为模块分配内存，按照导入导出语句建立模块和内存的映射关系。最后运行代码。

vite 会启动一个 koa 服务器拦截浏览器对 ESM 的请求，通过请求路径找到目录下对应的文件并处理成 ESM 格式返回给客户端。

vite的热加载是在客户端和服务端之间建立了 websocket 连接，代码修改后服务端发送消息通知客户端去请求修改模块的代码，完成热更新，就是改了哪个 view 文件就重新请求那个文件，这样保证了热更新速度不受项目大小影响。

开发环境会使用 esbuild 对依赖进行个预构建缓存，第一次启动会慢一点，后面的启动会直接读取缓存

2. 生产环境

使用 rollup 来构建代码，提供指令可以用来优化构建过程。缺点就是开发环境和生产环境可能不一致;

webpack 和 vite 对比

Webpack 的热更新原理简单来说就是，一旦发生某个依赖（比如 `a.js` ）改变，就将这个依赖所处的 `module` 的更新，并将新的 `module` 发送给浏览器重新执行。每次热更新都会重新生成 `bundle` 。试想如果依赖越来越多，就算只修改一个文件，理论上热更新的速度也会越来越慢

Vite 利用浏览器去解析 imports，在服务器端按需编译返回，完全跳过了打包这个概念，服务器随起随用，热更新是在客户端和服务端之间建立了 websocket 连接，代码修改后服务端发送消息通知客户端去请求修改模块的代码，完成热更新，就是改了哪个文件就重新请求那个文件，这样保证了热更新速度不受项目大小影响。

所以vite目前的最大亮点在于开发体验上，服务启动快、热更新快，明显地优化了开发者体验，生产环境因为底层是 rollup ， rollup更适合小的代码库，从扩展和功能上都是不如 webpack 的，可以使用vite作为一个开发服务器dev server使用，生产打包用webpack这样的模式。

做过哪些 wewbpack 的优化

0、升级 webpack 版本，3升4，实测是提升了几十秒的打包速度

1、splitChunksPlugin 抽离共用模块输出单独的chunks ,像一些第三方的依赖库，可以单独拆分出来，避免单个chunks过大。

2、DllPlugin 作用同上，这个依赖库相当于从业务代码中剥离出来，只有依赖库自身版本变化才会重新打包，提升打包速度

3、loaders的运行是同步的，同各模块会执行全部的loaders

- 可以使用oneOf,只要匹配上对应的loader就不会继续执行loader
- 使用 happyPack 将loader的同步执行转换成并行比如（style-loader,css-loader,less-loader合并起来执行）

4、exclude/include 指定匹配范围；alias指定路径别名;

5、cache-loader持久化存储；

6、ESM项目开启 useExport标记，使用 tree-shaking

7、exclude/include 指定匹配范围；alias指定路径别名;cache-loader持久化存储；

8、terserPlugin 可以提供代码压缩，去除注释、去除空格的功能；MiniCssExtractPlugin 压缩 css

npm run 执行过程

0、在 package.json 文件中可以定义 script 配置项，里面可以定义运行脚本的键和值

1、在 npm install 的时候，npm 会读取配置将**执行脚本软链接**到 `node_modules/.bin` 目录下，**同时将 `./bin` 加入当环境变量 `$PATH` 中**，所以如果在全局直接运行该命令会去全局目录里找，可能会找不到该命令就会报错。比如 npm run start，他是执行的webpack-dev-server带上参数

2、还有一种情况，就是单纯的执行脚本命令，比如 npm run build，实际运行的是 node build.js，即使用 node 执行 build.js 这个文件

ESM和CJS 的区别

ES6

- ES6模块是引用，重新赋值会编译报错，不能修改其变量的指针指向，但可以改变内部属性的值；
- ES6模块中的值属于动态只读引用。
- 对于只读来说，即不允许修改引入变量的值，import的变量是只读的，不论是基本数据类型还是复杂数据类型。当模块遇到import命令时，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。
- 对于动态来说，原始值发生变化，import 加载的值也会发生变化。不论是基本数据类型还是复杂数据类型。
- 循环加载时，ES6模块是动态引用。只要两个模块之间存在某个引用，代码就能够执行。

CommonJS

- CommonJS模块是拷贝（浅拷贝），可以重新赋值，可以修改指针指向；
- 对于基本数据类型，属于复制。即会被模块缓存。同时，在另一个模块可以对该模块输出的变量重新赋值。
- 对于复杂数据类型，属于浅拷贝。由于两个模块引用的对象指向同一个内存空间，因此对该模块的值做修改时会影响另一个模块。 当使用require命令加载某个模块时，就会运行整个模块的代码。
- 当使用require命令加载同一个模块时，不会再执行该模块，而是取到缓存之中的值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。
- 当循环加载时，脚本代码在require的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。

设计模式篇

代理模式

代理模式：为对象提供一个代用品或占位符，以便控制对它的访问

例如实现图片懒加载的功能，先通过一张 `loading` 图占位，然后通过异步的方式加载图片，等图片加载好了再把完成的图片加载到 `img` 标签里面

装饰者模式

装饰者模式的定义：在不改变对象自身的基础上，在程序运行期间给对象动态地添加方法

通常运用在原有方法维持不变，在原有方法上再挂载其他方法来满足现有需求。

像 typescript 的装饰器就是一个典型的装饰者模式，还有 vue 当中的 mixin

单例模式

保证一个类仅有一个实例，并提供一个访问它的全局访问点。实现的方法为先判断实例存在与否，如果存在则直接返回，如果不存在就创建了再返回，这就确保了一个类只有一个实例对象

比如 ice stark 的子应用，一次只保证渲染一个子应用

观察者模式和发布订阅模式

1、虽然两种模式都存在订阅者和发布者（具体观察者可认为是订阅者、具体目标可认为是发布者），但是观察者模式是由具体目标调度的，而发布/订阅模式是统一由调度中心调的，所以观察者模式的订阅者与发布者之间是存在依赖的，而发布/订阅模式则不会。

2、两种模式都可以用于松散耦合，改进代码管理和潜在的复用。

3、在观察者模式中，观察者是知道Subject的，Subject一直保持对观察者进行记录。然而，在发布订阅模式中，发布者和订阅者不知道对方的存在。它们只有通过消息代理进行通信

4、观察者模式大多数时候是同步的，比如当事件触发，Subject就会去调用观察者的方法。而发布-订阅模式大多数时候是异步的（使用消息队列）

其他

正则表达式

正则表达式是什么数据类型?

是对象，`let re = /ab+c/` 等价于 `let re = new RegExp('ab+c')`

正则贪婪模式和非贪婪模式?

量词

`*`：0或多次； `?`：0或1次； `+`：1到多次； `{m}`：出现m次； `{m,}`：出现至少m次； `{m,n}`：出现m到n次

贪婪模式

正则中，表示次数的量词默认是贪婪的，会尽可能匹配最大长度，比如 `a*` 会从第一个匹配到a的时候向后匹配尽可能多的a，直到没有a为止。

非贪婪模式

在量词后面加 `?` 就能变成非贪婪模式，非贪婪即找出长度最小且满足条件的

贪婪& 非贪婪特点

贪婪模式和非贪婪模式，都需要发生**回溯**才能完成相应的功能

独占模式

独占模式和贪婪模式很像，独占模式会尽可能多地去匹配，如果匹配失败就结束，不会进行回溯，这样的话就比较节省时间。

写法：量词后面使用 `+`

优缺点：独占模式性能好，可以减少匹配的时间和 cpu 资源；但是某些情况下匹配不到，比如：

	正则	文本	结果
贪婪模式	a{1,3}ab	aaab	匹配
非贪婪模式	a{1,3}?ab	aaab	匹配
独占模式	a{1,3}+ab	aaab	不匹配

a{1,3}+ab 去匹配 aaab 字符串，a{1,3}+ 会把前面三个 a 都用掉，并且不会回溯

常见正则匹配

操作符	说明	实例

.	表示任何单个字符			
[ ]	字符集，对单个字符给出范围	`[abc]` 表示 a、b、c, [a-z]表示 a-z 的单个字符		
[^]	非字符集，对单个字符给出排除范围	`[^abc]` 表示非a或b或c的单个字符		
–	前一个字符零次或无限次扩展	`abc_` 表示 ab、abc、abcc、abccc 等		
`	`	左右表达式的任意一个	`abc	def`表示 abc、def
$	匹配字符串结尾	`abc$` 表示 abc 且在一个字符串结尾		
( )	分组标记内部只能使用	`(abc)` 表示 abc,`(abc	def)`表示 abc、def	
\D	非数字			
\d	数字，等价于0-9			
\S	可见字符			
\s	空白字符(空格、换行、制表符等等)			
\W	非单词字符			
\w	单词字符，等价于[a-z0-9A-Z_]			
	匹配字符串开头	`^abc` 表示 abc 且在一个字符串的开头		
{m,n}	扩展前一个字符 m 到 n 次	`ab{1,2}c` 表示 abc、abbc		
{m}	扩展前一个字符 m 次	`ab{2}c` 表示 abbc		
操作符	**说明**	**实例**		
	匹配前一个字符至少m			

{m,}	次	
?	前一个字符 0 次或 1 次扩展	`abc?` 表示 ab、abc

单元测试

概念：前端自动化测试领域的， 用来验证独立的代码片段能否正常工作

1、可以直接用 Node 中自带的 assert 模块做断言：如果当前程序的某种状态符合 assert 的期望此程序才能正常执行，否则直接退出应用。

```ini
function multiple(a, b) {
    let result = 0;
    for (let i = 0; i < b; ++i)
        result += a;
    return result;
}

const assert = require('assert');
assert.equal(multiple(1, 2), 3));
```

2、常见单测工具：Jest，使用示例

```scss
const sum = require('./sum');

describe('sum function test', () => {
  it('sum(1, 2) === 3', () => {
    expect(sum(1, 2)).toBe(3);
  });

  // 这里 test 和 it 没有明显区别，it 是指：it should xxx, test 是指 test xxx
  test('sum(1, 2) === 3', () => {
    expect(sum(1, 2)).toBe(3);
  });
})
```

babel原理和用途

babel 用途

- 转义 esnext、typescript 到目标环境支持 js （高级语言到到低级语言叫编译，高级语言到高级语言叫转译）
- 代码转换（taro）
- 代码分析（模块分析、tree-shaking、linter 等）

bebel 如何转换的?

对源码字符串 parse 生成 AST，然后对 AST 进行增删改，然后输出目标代码字符串

转换过程

1. parse 阶段：首先使用 `@babel/parser` 将源码转换成 AST
2. transform 阶段：接着使用 `@babel/traverse` 遍历 AST，并调用 visitor 函数修改 AST，修改过程中通过 `@babel/types` 来创建、判断 AST 节点；使用 `@babel/template` 来批量创建 AST
3. generate 阶段：使用 `@babel/generate` 将 AST 打印为目标代码字符串，期间遇到代码错误位置时会用到 `@babel/code-frame`

好的，差不多就是这些了，祝大家早日找到心仪的工作~