2025.4

1. 说一下Vue生命周期

vue 的生命周期就像是组件从出生到退休的整个过程，分成四个阶段，包括：创建阶段、挂载阶段、更新阶段和销毁阶段，在这些生命周期过程中，会自动执行一些函数，也就是生命周期钩子函数

- 创建阶段：
  - beforeCreate：数据和事件都还没有初始化
  - created：数据初始化完了，可以发请求了（但页面还没渲染）

    > 这个钩子函数常用，可以在其中发请求拿数据，如项目中购物车页面，可以在 created 中发请求拿到购物车页面商品列表的数据存到 vuex 之后再渲染
    >
    > 为什么这里数据一定要存到 vuex ，直接存到页面的 data 中用不行吗?
    >
    > 答：不是不能存在 data，而是因为购物车数据具有全局共享的特性，放 vuex 更合理，方便管理和同步
    >
    > 要不要存vuex的关键是看它是不是**多个组件共享的核心数据**
    >
    > 商品结算页面需要的商品id集需要vuex中selCartList
    >
    > 商品详细页及底部tabber导航栏购物车图标右上角显示数量需要cartTotal
    >
    > Vuex 是存储在**内存中的临时数据**，页面刷新会导致 JS 重新加载，也就意味着 Vuex 的 state 被**重置了**

- 挂载阶段：
  - beforeMount：模版已编译，还没挂到页面上
  - mounted：模版已经全部挂载到页面上，DOM 可以操作了

    > 我在项目里还用 `mounted` 做过页面滚动位置的还原功能，比如用户刷新页面，我会在刷新前记录 scrollTop，在 `mounted` 中读取这个值并用 `window.scrollTo()` 定位回原来的位置。这个操作要放在 `mounted`，因为这时候 DOM 已经挂载好了，页面才能滚得动。
    >
    > this.$refs.xxx 获取目标标签，需要等 DOM 渲染完毕之后，也就是在 mounted

- 更新阶段：
  - beforeUpdate：数据更新了，但 DOM 还没变
  - updated：DOM 跟着数据更新了
- 销毁阶段：
  - beforeDestroy：实例准备销毁了，可以在这里做一些清理

    > 可以清理定时器，释放这些资源提升性能

> 原因：
>
> 定时器（`setInterval` 或 `setTimeout`）是由 **浏览器环境管理的异步任务**，它和 Vue 组件本身没有直接绑定关系。
>
> 所以，即使组件销毁了，**定时器依然活着！** 它还会继续执行，只不过它要执行的回调函数可能会引用已经被销毁的 DOM 或数据，**这时候就会出问题了：**
>
> - 报错（访问不存在的 DOM）
> - 内存泄漏（函数引用旧数据，无法被释放）
> - 性能浪费（一直在执行没意义的逻辑）

- destroyed：组件彻底没了

2. 说一下组件通信

- 最常用的 父子通信：
  - Vue2 Options API 中的写法

    父传子：通过`props`

```
<Child :msg="老父亲要传的数据">

props: {
    msg: {
        type: String,
        default: 'hello'
        // 对于 String、Number、Boolean 这类值类型，可以直接写默认值
        // 如果默认值是对象必须用箭头函数返回
        // default: () => {
        //   return {}
        // }
        // 因为对象是引用数据类型，这样写会导致所有使用这个组件的实例，都会共享
        // 同一个对象作为默认值，可能会造成数据污染
        // 而使用箭头函数每一次都会调用函数而返回生成一个新的空对象
        // 利用了 函数作用域 + 闭包特性
    }
}
```

    子传父：通过`$emit`

```
this.$emit('update', '儿子要传给老父亲的数据')

<Child @update="handleUpdate">

handleUpdate (msg) {
   console.log(msg)
}
```

- Vue3 composition API 中的写法

  父传子：通过`defineProps`

```
<Child :msg="老父亲要传的数据"></Child>

// 设置 setup 之后，props 没地方写了
<script setup>
    const props = defineProps({
        msg: {
            type: String,
            default: 'hello'
        }
    })
</script>
```

  子传父：通过`defineEmits`

```
<SonCom :changeMsg="changeFn"></SonCom>
changeFn (msg) {
    this.msg = msg
}

// 设置 setup 之后，没有 this 了
<script setup>
    const emit = defineEmits(['changeMsg'])
    emit('changeMsg', '儿子想传给老父亲的数据')
</script>
```

- ref 和 $refs

  利用 ref 和 $refs 可以用于 获取 DOM 元素，或 组件实例

```
<BaseForm ref="baseForm"></BaseForm>

this.$refs.baseForm.组件方法()

// 比如来重置或者清空表单，表单是一个子组件
// 父组件
handleGet () {
    console.log(this.$refs.getValues())
}
handleReset () {
    this.$refs.resetValues()
}
// 子组件（表单组件）
```

```
getValues () {
    // 收集表单数据，返回一个对象
    return {
        account: this.account,
        password: this.password
    }
}
resetValues () {
    // 重置表单
    this.account = ''
    this.password = ''
}
```

- provide / inject —— 跨层组件通信（爷孙、跨多层）
    - Vue2 Options API 中的写法

      父组件 provide 提供数据

      ```
      export default {
          provide () {
              return {
                  color: this.color, // 普通类型（非响应式）
                  userInfo: this.userInfo // 复杂类型（响应式）
              }
          }
      }
      ```

      子 / 孙组件 inject 取值使用

      ```
      <div>{{ color }} - {{ userInfo.name }}<div>

      export default {
          inject: ['color', 'userInfo']
      }
      ```

    - Vue3 composition API 中的写法

      父组件 provide 提供数据

      ```
      <script setup>
          import { provide } from 'vue'
          const message = ref('hello')
          provide('msg', message)
          // 跨层传递函数 => 用来给子孙后代修改数据
          provide('changeMsg', (newMsg) => {
              message.value = newMsg
      ```

```
    })
</script>
```

子 / 孙组件 inject 取值使用

```
<script setup>
    import { inject } from 'vue'
    const msg = inject('msg')
    // 接收用来修改数据的函数
    const changeMsg = inject('changeMsg')
</script>
```

- EventBus（事件总线）

  简易的消息传递，可以用 EventBus，需要借助一个中转站 Bus（空的 Vue 实例）

  utils/EventBus.js

  ```
  import Vue form 'vue'
  const Bus = new Vue()
  export default Bus
  ```

  A组件（接收方）监听 Bus 实例的事件

  ```
  // created中数据已经初始化完了，created执行的很早，最好一进页面就进行监听，监听的越早越好
  created () {
      Bus.$on('sendMsg', (msg) => {
          console.log(msg)
          this.msg = msg
      })
  }
  ```

  B组件（发送方）触发 Bus 实例的事件

  ```
  Bus.$emit('sendMsg', '这是一条发给A组件的消息')
  ```

- vuex —— 全局状态管理

  适合用于多个组件都要用到的数据，比如购物车数据、用户信息、权限状态token等。

  > “**只要是多个组件都要用到的数据，就该放进 vuex**”

3. 说一下Vuex

Vuex 就是 Vue 的 **状态管理工具**，是专门用来解决组件之间数据共享的问题

它有五个核心概念：

- state：存储数据（就像 data）
- getter：计算属性（可以对 state 做加工）
- mutations：添加方法修改 state 中的数据（唯一方式）
- actions：写异步代码，主要用来发请求，也可以提交 mutations

> 购物车中，在 actions 中通过 getCartAction 发请求后，通过 commit 提交 setCartList mutations 方法更新 state 中的数据

4. Vue的基本原理

Vue 的核心是：**数据响应式 + 模版编译 + 视图更新机制**

1. 数据劫持（响应式）

   Vue 用 `Object.defineProperty()` 把 `data` 中的属性变成 "响应式的"，能监听数据的读写

2. 依赖收集（Dep 和 Watcher）
3. 模版编译（Compile）
4. 视图更新（patch + diff）

5.说一下双向绑定的原理

6.使用 Object.defineProperty() 来进行数据劫持有什么缺点？#

在对一些属性进行操作时，使用这种方法无法拦截，比如通过下标方式修改数组数据或者给对象新增属性，这都不能触发组件的重新渲染，因为 Object.defineProperty() 不能拦截到这些操作。更精确的来说，对于数组而言，大部分操作都是拦截不到的，只是 Vue 内部通过重写函数的方式解决了这个问题。

在 Vue3 中已经不使用这种方式了，而是通过使用 Proxy 对对象进行代理，从而实现数据劫持。它的好处是可以完美的监听到任何方式的数据改变，包括对象增删属性、通过数组下标更改数据。

7.说一下 diff 算法

由于 Vue 使用了 **虚拟DOM**，diff算法就是**比较新旧虚拟DOM的差异**，然后**只更新真正发生变化的那部分DOM。**diff算法采用 双端对比 + 同层比较

8.MVVM、MVC、MVP的区别

MVC、MVP 和 MVVM 是三种常见的软件架构设计模式

9.知道哪些 vue 指令

> 指令就是一些带有 **v-前缀** 的特殊**标签属性**

v-if / v-else 条件渲染（这个是直接 创建/销毁 DOM）

v-show 控制元素的显示隐藏（这个 DOM 还在）

v-for 列表渲染 基于数据循环，多次渲染整个元素

v-model 双向数据绑定，是一个语法糖，相当与 :value + @input

v-html 设置元素的 innerHTML

v-on 用来注册事件的 v-on: => @

> v-on:click="count++" => @click="count++"

v-bind 动态绑定属性

> v-bind:src="imgUrl" => :src="imgUrl"

v-slot 通常设置在 `<template v-slot="test">` 上，配合 `<slot name="test">` ，可以准确地将内容插入到子组件指定的位置上，也能接收子组件给的数据

> v-slot:插槽名 => #插槽名

父组件 `App.vue`

```
<template>
    <div>
        <Mytable>
            <!--
              <template>
              可以用来加 v-slot:header 找对应的 <slot>
                        #header
              也可以来加 #default="obj" 接收 <slot> 传来的数据
            -->
            <!-- <template #test="obj"> -->
            <template v-slot:test="obj">
                <button @click="del(obj.row.id)">删除</button>
            </template>
        </Mytable>
    </div>
</template>
```

子组件 `MyTable.vue`

```
<template>
    <table>
        <thead>
            <tr>
                <th>序号</th>
```

```
                <th>姓名</th>
                <th>年纪</th>
                <th>操作</th>
            </tr>
        </thead>
        <tbady>
            <tr v-for="(item, index) in data" :key="item.id">
                <td>{{ index + 1}}</td>
                <td>{{ item.name }}</td>
                <td>{{ item.age }}</td>
                <td>
                  <slot name="test" :row="item" msg="测试文本"></slot>
                </td>
            </tr>
        </tbady>
    </table>
</template>
```

10.说一下自定义指令

自定义指令：子级定义的指令，可以封装一些 dom 操作，扩展额外功能

```
// 全局注册
// 代码写在 main.js 中
Vue.directive('指令名', {
    // inserted 是自定义指令内置的生命周期钩子
    // 会在指令所在的元素被插入到页面中时触发
    "inserted" (el) {
        // el 就是指令所绑定的元素
        el.focus()
    }
})

<input v-指令名 type="text">

// 局部注册
// 写在对应的组件内，只能在该组件内使用
directives: {
    "指令名": {
        inserted (sl) {
            el.focus()
        }
    }
}

<input v-指令名 type="text">
```

11.说一下路由传参

- 动态路由传参

```
// 配置路由
{
    path: '/user/:id'
    component: user
}
// 跳转传参
this.$router.push(`/user/${id}`)
// 获取参数
this.$route.params.id
```

- 查询参数传参

```
// 跳转传参
this.$router.push(`/user?username=${username}}`)
// 获取参数
this.$route.query.username
```

12.说一下节流防抖

> 利用 lodash 中提供的节流防抖函数快速处理

节流 throttle：单位时间内，频繁触发事件，只执行一次（冷却CD）

使用场景：

- 按钮点击放连点
- 页面滚动到底加载更多。给滚动事件（scroll）加节流，每 100 ms执行一次检测是否到底

```
// lodash
box.addEventListenr('scroll', _.throttle(scroll, 500))
```

防抖 debounce：单位时间内，频繁触发事件，只执行最后一次（回城）

使用场景：

- 搜索框搜索输入。只需用户最后一次输入完，再发送请求
- 手机号、邮箱验证输入检测

```
// lodash
box.addEventListener('input', _.debounce(input, 500))
```

13.说一下 this

全局作用域，浏览器中 this 指向 window，Node.js 中 this 指向 global

普通函数，谁调用函数，this 就是谁

箭头函数不绑定 this，看外层

用 new 调用构造函数时，this 指向新创建的对象

```
// 事件监听中
// button元素，调用了回调函数
document.querySelector('button').addEventListener('click', function () {
  console.log(this); // 这个 this 指向触发事件的元素 button
})

// 箭头函数中
const obj = {
  name: 'GPT',
  sayHi: () => {
    console.log(this.name);
  }
};
obj.sayHi(); // undefined

// 构造函数中
function Person(name) {
  this.name = name;
}
const p = new Person('小明');
console.log(p.name); // 小明

// 函数直接调用
function show() {
  console.log(this);
}
show(); // 浏览器中 -> window

// 对象方法中
const obj = {
  name: 'GPT',
  sayHi() {
    console.log(this.name);
  }
};
obj.sayHi(); // 输出 GPT

// 全局作用域
console.log(this) // window
```

14.怎么阻止默认事件

```
<a href="http://baidu.com"></a>
```

```
document.querySelector('a').addEventListener('click', (e) => {
    // 阻止元素默认行为，这里是阻止a标签默认跳转
    e.preventDefault()
})
```

15.怎么阻止事件冒泡

```
son.addEventListener('click', () => {
    alert('我是儿子')
    // 此方法可以阻断事件流动传播，不光在冒泡阶段有效，捕获阶段也有效
    e.stopPropagation()
})
```

16.数组的常用方法

- `push()` 尾部添加元素

```
let arr = [1, 2]
arr.push(3)
console.log(arr) // [1, 2, 3]
```

- `pop()` 删除最后一个元素

```
let arr = [1, 2 ,3]
arr.pop()
console.log(arr) // [1, 2]
```

- `unshift()` 头部添加元素

```
let arr = [2, 3]
arr.unshift(1)
console.log(arr) // [1, 2, 3]
```

- `shift()` 删除第一个元素

```
let arr = [1, 2, 3]
arr.shift()
console.log(arr) // [2, 3]
```

- `splice()` 数组增 / 删 / 改

```
let arr = [1, 2, 3, 4]

// 删除 2 个，从索引 1 开始
arr.splice(1, 2) // 删除了 [2, 3]
console.log(arr) // [1, 4]

// 插入元素
arr.splice(1, 0, 'a', 'b') // 从索引 1 开始，删除 0 个，插入 2 个
console.log(arr) // [1, 'a', 'b', 4]

// 替换元素
arr.splice(2, 1, 'X')
console.log(arr) // [1, 'a', 'X', 4]
```

- `concat()` 合并数组

> 不改原数组，返回新数组
>
> 展开运算符 `...` 也可以合并
>
> ```
> let arr = [...arr1, ...arr2]
> ```

```
let arr1 = [1, 2]
let arr2 = [3, 4]
let newArr = arr1.concat(arr2)
console.log(newArr) // [1, 2, 3, 4]
```

- `slice(start, end)` 数组截取

> 从索引 start 开始，到索引 end（不含）

```
let arr = [1, 2, 3, 4]
let res = arr.slice(1, 3)
console.log(res) // [2, 3]
```

- `join()` 把数组变成字符串

```
let arr = ['a', 'b', 'c']
console.log(arr.join()) // a,b,c
console.log(arr.join('-')) // a-b-c
```

- `includes()` 是否包含某值

```
let arr = [1, 2, 3]
arr.includes(2) // true
```

- `indexOf` 查找元素索引

```
let arr = ['a', 'b', 'c']
arr.indexOf('b') // 1
```

17.数组去重知道哪些方法

- Set + 展开运算符

> Set 是 JS 的一种集合类型，用来存储**唯一值**（不会重复）
>
> 可以是数字、字符串、对象，啥都能存
>
> ```
> const s = new Set([1, 2, 3, 3]);
> console.log(s); // Set(3) {1, 2, 3}
> ```
>
> new 出来的是 Set(3) {1, 2, 3} 这种形式，需要用展开运算符展开

```
const arr = [1, 2, 2, 3]
const newArr = [...new Set(arr)]
```

- filter + indexOf

```
const arr = [1, 2, 3, 3, 4]
const newArr = arr.filter((item, index) => arr.indexOf(item) === index)
```

- reduce + includes

```
const arr = [1, 2, 2, 3, 4]
const newArr = arr.reduce((prev, cur) => {
    if(!prev.includes(cur)) prev.push(cur)
    return prev
}, [])
```

18.字符串操作方法

- `length` 获取字符串长度

```
const str = "hello"
console.log(str.length) // 5
```

- `charAt(index)` 获取指定索引的字符

```
"hello".charAt(1) // "e"
```

- `slice(start, end)` 截取字符串的片段，不包含 end 索引的字符

```
"abcdefg".slice(1, 4) // "bcd"
"abcdefg".slice(-3) // "efg"
```

- `indexOf(substr)` 返回首次出现的位置，找不到返回 -1

```
"hello world".indexOf("o") // 4
```

- `includes(substr)` 是否包含某个字符串，返回布尔值

```
"hello".includes("el") // true
```

- `replace(原内容, 替换内容)` 替换第一个匹配项

```
"abcabc".replace("a", "X") // Xbcabc
```

- `toUpperCase` / `toLowerCase` 转换大小写

```
"AbC".toLowerCase() // "abc"
```

- `trim()` 清除空格

```
"  hello  ".trim() // "hello"
```

- `split(分隔符)` 把字符串拆成数组

```
"1,2,3".split(",") // ["1", "2", "3"]
```

19.怎么判断一个数是不是素数，逻辑怎么写?

素数的定义：大于1，且除了1和它本身外没有其他因数的自然数

```
const count = 7
let i;
for(i = 2; i < count; i++)
{
    if( count % i === 0) break
}
if(i === count){
    console.log('素数')
} else {
    console.log('不是素数')
}
```

20.怎么清除浮动

> **浮动：**
>
> - 浮动属性 float，left 表示左浮动，right 表示右浮动（浮动属性是设置给子盒子的，也就是想要其靠左或者靠右的盒子）
> - 特点：
>   - 浮动后的盒子顶对齐
>   - 浮动后的盒子具有行内块特点
>   - 父级宽度不够，浮动的子级会换行
>   - 浮动后的盒子脱标

问题场景：浮动元素会脱标，如果父级没有高度，子级无法撑开父级高度（可能导致页面布局错乱）

解决方法：清除浮动（不是删除浮动效果而是清除浮带来的影响）

- 额外标签法**（不推荐）**

  给父元素内容的最后添加一个块级元素，设置 css 属性 clear: both（会增加额外的标签，可能影响页面结构）

- 单伪元素法

```
.clearfix::after {
    content: "";
    display: block;
    clear: both;
}
```

- 双伪元素法**（推荐）**

```
/* before 解决外边距塌陷问题 */
/* 在父级开头加一个标签，隔离了有 margin 的子级盒子和父级盒子 */
.clearfix::before,
.clearfix::after {
    content: "";
    display: table
}

.clear::after {
    clear: both;
}
```

- overflow

  父元素添加 css 属性 overflow: hidden

  > 设置 overflow: hidden 后，浏览器会去寻找父级盒子的边缘位置，然后发现父级盒子大小出现问题，顺便就盒子大小问题解决

21.div怎么垂直居中

- flex 居中

```
<div class="box">
    <div class="son"></div>
</div>

.box {
    display: flex;
    justify-content: center;
    align-items: center;
}
```

- 定位 + transform

```
<div class="box">
    <div class="son"></div>
</div>

.box {
    position: relative;
}
.box .son {
    position: aboulate;
    left: 50%;
    right: 50%;
    tranform(-50%, -50%);
}
```

> 通常使用 margin: 0 auto; 实现水平居中
>
> margin-left 和 margin-right 自动均分剩余空间

22.怎么画0.5px的线

可以使用`scaleY(0.5)`进行缩放

```
<div class="half-line"></div>

<style>
    .half-line {
        height: 1px;
        background-color: black;
        transform: scaleY(0.5);
    }
</style>
```

其实对于一些高分屏设备，可以直接给div设置 background-color: black; height: 0.5px; 就可以直接在屏幕上显示一条黑色的0.5px的线

但是为了追求兼容性推荐用`scaleY(0.5)`

> 什么是低分屏高分屏，是分辨率高吗?
>
> 低分屏是指设备像素比比较低的屏幕。也就是说，一个 css像素（1px）在物理屏幕上映射的物理像素数量比较少，这种屏幕就称为“低分屏”。
>
> 低分屏（DPR = 1）：1个css像素（1px）就是1个物理像素，没办法再细了，所以 0.5px 也只能在屏幕上用1个物理像素显示，显示的效果跟 1px 相同
>
> 高分屏（DPR = 2 以上）：每个 css像素 可以映射多个物理像素 1px 可能等于 2x2=4 个物理像素（DPR = 2），甚至更多，这就有可能渲染出更细的线

23.说一下HTML语义化

HTML语义化就是用**合适的标签表达内容的意义**，比如：

不语义化写法	语义化写法
`<div id="header">`	`<header>`
`<div id="nav">`	`<nav>`
`<div id="main">`	`<main>`

主要优点：

1. 对机器友好，带有语义的文字表现力丰富，更适合搜索引擎的爬虫爬取有效信息，利于SEO

> **SEO = Search Engine Optimization（搜索引擎优化）**
>
> 简单说就是：
>
> > 让你的网站在百度、Google、Bing 这些搜索引擎上更容易被找到、排名更靠前。
>
> SEO 涉及很多方面，比如页面加载速度、结构清晰、关键词优化、移动端适配，但其中一个非常重要的点就是——**HTML 语义化结构**！

2. 对开发者友好，使用语义化标签增强了可读性，开发者能一眼看出网页的结构

24.说一下盒模型

CSS3中的盒模型有一下两种：标准盒模型、怪异盒模型

盒模型都是由四个部分组成的，从内到外分别是content、padding、border和margin

标准盒模型和怪异盒模型的区别在于设置 width 和 height 时，所对应的范围不同：

- 标准盒模型的 width 和 height 属性的范围只包含了 content
- 怪异盒模型的 width 和 height 属性的范围包含了content、padding和border

可以通过修改元素的 box-sizing 属性来改变元素的盒模型：

- box-sizing: content-box 表示标准盒模型（默认值）
- box-sizing: border-box 表示怪异盒模型（IE盒模型）

25.两个异步请求数据操作怎么合并

- 使用 `Promise.all()`

  并发请求，等待所有请求都完成，如果有一个失败整个Promise.all就会调用 .catch()

```
Promise.all([p1, p2, p3]).then(res => {
    console.log(res)
}).catch(err => {
    console.log(err)
})
```

- 使用 `async/await` 顺序请求

  一个请求完成之后再发起下一个

```
async function getData() {
  try {
    const token = await getUserToken() // 如果这里出错，直接跳到catch
    const data = await getUserData(token) // 如果getUserToken成功但这里出错，
也跳到catch
    console.log(data)
  } catch (err) {
```

```
    console.error('请求失败: ', err) // 捕获第一个发生的错误
  }
}
```

> try-catch
>
> - 只能捕获第一个发生的错误
> - 一旦某个await失败，后续代码不会执行

- 使用 `async/await` 并发请求

```
async function getData() {
  try {
    // 这种情况JS引擎会自动处理解构
    const [info, posts] = await Promise.all([getUserInfo(), getUserPosts()])
    console.log(info, posts)
  } catch (err) {
    console.error('有请求失败了: ', err)
  }
}
```

> 控制台直接打印 Promise.all 可能的结果
>
> ```
> 所有Promise都成功
> Promise { <pending> }  // 如果 Promise 还未完成
> Promise { [ "成功1", "成功2", "成功3" ] }  // 如果 Promise 已完成
> 如果有一个失败
> Promise { <rejected> '错误' }
>  (控制台可能附带警告: Uncaught (in promise) 错误
> ```

26.v-model 是谁的语法糖

:value 和 @input

27.query和params区别

> query：查询参数 params：路径参数

- params 是路由路径里的参数，比如 `/user/:id`，而 query 是 URL 问号后面的参数
- 必须在路由里声明 `:id`，也就是路由要写成 `/user/:id` 才支持 params

28.vue怎么获取dom

- Vue2 Options API

```

```

```
<template>
  <div ref="box">我是一个盒子</div>
</template>

export default {
    // 模版已经全部挂载到页面上，可以操作DOM了
    mounted () {
        console.log(this.$refs.box) // 原生DOM元素
    }
}

// 获取多个DOM元素
<ul>
  <li v-for="(item, index) in list" :key="index" :ref="'items'">{{ item }}
</li>
</ul>

mounted() {
  // 会是一个数组
  console.log(this.$refs.items) // [li, li, li...]
}
```

- Vue3 Composition API

```
<template>
    <div ref="box">Hello</div>
    <div ref="boxRef">Hello</div>
</template>

<script setup>
    import { ref, onMounted } from 'vue'
    // 这里定义的变量名要跟上面 ref 中的一致
    const box = ref(null)
    const boxRef = ref(null)
    onMounted(() => {
        // 两个div内文字的颜色都会生效
        console.log(box.value)
        box.value.style.color = "red"
        console.log(boxRef.value)
        boxRef.value.style.color = "blue"
    })
</script>
```

29.flex布局怎么把元素搞到右上角

```
<div class="container">
  <div class="box">右上角元素</div>
</div>
```

```
.container {
    display: flex;
    justify-content: flex-end; /* 横向靠右 */
    align-items: flex-start;
    height: 200px;
    width: 200px;
    background: pink;
}
.box {
    width: 50px;
    height: 50px;
    background: blue;
}
```

flex 布局：

- 主轴对齐方式

  justify-content

  flex-start 弹性盒子从**起点**开始依次排列

  flex-end 弹性盒子从**终点**开始依次排列

  center 弹性盒子沿主轴**居中**排列

  space-between 两个盒子先靠左右两边，空白间距均分在盒子之间

  space-around 两端出现空白间距，视觉效果：弹性盒子之间的间距是两端间距的两倍

  space-evently 两端同样出现空白间距，不过弹性盒子之间的间距与两端间距相等

- 侧轴对齐方式

  align-items 设置给所有盒子（给弹性容器设置）

  align-self 只设置给一个盒子（给那个盒子单独设置）

  flex-start 弹性盒子从**起点**开始依次排列

  flex-end 弹性盒子从**终点**开始依次排列

  stretch 沿着侧轴线**拉伸至铺满容器**（不能设置侧轴方向的尺寸）

  center 沿侧轴**居中**排列

- 修改主轴方向

  flex-direction: column 改变主轴方向为垂直方向，侧轴自动变换为水平方向

  flex-direction: row 改变主轴方向为水平方向，侧轴自动变换为垂直方向

- 弹性伸缩比

> 默认情况下，主轴方向尺寸是靠内容撑开的，侧轴默认拉伸

控制弹性盒子主轴方向的尺寸，因为默认是靠内容撑开的

A盒子 flex: 1 B盒子 flex: 2 C盒子 flex: 1

占空白比例 1/4 1/2 1/4

- 弹性换行

  flex-wrap: nowrap 不换行（默认）

  flex-wrap: wrap 换行

- 行对齐方式

  align-content

  flex-start 弹性盒子从**起点**开始依次排列

  flex-end 弹性盒子从**终点**开始依次排列

  center 弹性盒子沿主轴**居中**排列

  space-between 两个盒子先靠左右两边，空白间距均分在盒子之间

  space-around 两端出现空白间距，视觉效果：弹性盒子之间的间距是两端间距的两倍

  space-evently 两端同样出现空白间距，不过弹性盒子之间的间距与两端间距相等

> align-items 和 align-content 的区别：
>
> align-items 是对单行生效的；align-content 是对多行生效的，需要开启弹性换行wrap

30.promise有几种状态，会不会改变

Promise 有三种状态：

- 待定（pending）：初始状态，既没有被兑现，也没有被拒绝
- 已兑现（fulfilled）：操作成功完成 => 执行 .then() 回调
- 已拒绝（rejected）：操作失败 => 执行 .catch() 回调

Promise对象一旦被兑现 / 拒绝 就是已经敲定了，状态无法再被改变了

31.async和await解决什么问题

使异步代码看起来像同步代码一样，更清晰自然

以下以后再深挖，再挖下去没完没了了！！！

async 函数返回的是一个 Promise 对象

.then() ?

32.var、let和const的区别

- 作用域：var 是函数作用域，而 let 和 const 是块级作用域
- 变量提升：var 会提升，但值是 undefined，let 和 const 不存在变量提升

```
console.log(a); // 输出 undefined（不是报错）
var a = 5;
```

等价于

```
var a;           // 变量声明被提升
console.log(a);  // undefined
a = 5;           // 赋值在原地
```

- 重复声明：var 声明变量时，可以重复声明变量，后声明的同名变量会覆盖之前声明的变量。let 和 const 不允许重复声明变量
- 重新赋值：var 和 let 可以重新赋值，const 不可以
- 初识值设置：在变量声明时，var 和 let 可以不用设置初始值。而 const 声明变量必须设置初始值
- 全局挂载：浏览器的全局对象是 window，Node.js 的全局对象是 global。var 声明的变量为全局变量，并且会将该变量添加为全局对象的属性，但是 let 和 const 不会

> 全局挂载就是把某个变量、方法、插件、组件等，挂在全局对象上（比如 Vue 的 app.config.globalProperties、或浏览器的 window）来让所有地方都能访问到

```
let a = 10;
console.log(window.a); // undefined
var b = 20;
console.log(window.b); // 20
```

> 什么是作用域?
>
> 作用域是变量的可访问范围
>
> **js 中常见的三种作用域：**
>
> 全局作用域：整个文件都能访问
>
> 函数作用域：变量只能在函数内部访问
>
> 块级作用域：{}代码块内有效

```
var a = 1;

function test() {
    var b = 2;
```

```
    if(1){
        var c = 3;
    }
    console.log(a); // 1: 全局变量能访问
    console.log(b); // 2: 函数内部能访问自己的变量
    console.log(c); // 3: var 是函数作用域，块级作用域不会限制它
}
test()
console.log(b); // 报错: b 只在函数内部有效
console.log(c); // 报错: c 只在函数内部有效
```

块级作用域例子:

```
if (1) {
  let x = 10;
}
console.log(x); // 报错: let 有块级作用域
```

33.const定义对象里面的属性值能不能改

可以改对象里面的属性，但是不能改整个对象

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

但对于引用类型的数据（主要是对象和数组）来说，变量指向数据的内存地址，保存的只是一个指针，const只能保证这个指针是固定不变的，不能控制它指向的数据不变

```
const obj = {
  name: 'Anvar',
  age: 20
}

// 修改属性成功
obj.age = 21

// 添加属性成功
obj.gender = 'male'

// 重新赋值（换整个对象）就报错
// obj 存储的那个地址变了
obj = {
  name: 'GPT'
}
```

34.常见状态码

常见的状态码主要分为几类：2xx 表示请求成功，3xx 表示重定向，4xx 是客户端错误，5xx 是服务端错误。

比如我们常见的 200 表示请求成功，404 是找不到资源，401 是未授权，500 是服务器报错。如果涉及到缓存会用到 304，重定向会用到 301 或 302

35.对同步和异步的理解

同步代码：逐行执行，需原地等待结果后，才能向下执行

异步代码：调用后耗时，不阻塞代码继续执行（不必原地等待），在将来完成后触发一个回调函数

36.常见的宏任务，微任务

- 宏任务：浏览器执行的异步代码

  setTimeout / setInterval、ajax请求、用户交互事件（注册点击事件）

- 微任务：JS引擎执行的异步代码

  Promise对象.then()

  > Promise对象 本身是同步的，而then和catch回调函数是异步的

```
console.log(1)
// 宏任务 => 进入宏任务队列
setTimeout(() => {
    console.log(2)
}, 0)
// Promise对象 本身是同步的，在Promise对象创建时，里面的代码就会执行了
const p = new Promise((resolve, reject) => {
    console.log(3)
    resolve(4)
})
// 微任务 => 进入微任务队列（优先级大于宏任务）
p.then(result => {
    console.log(result)
})
console.log(5)

// 结果：1 3 5 4 2
```

37.怎么判断两个数组相等

先比较长度，如果长度相等在比较每一项值是否一致

```
function isEqual (arr1, arr2) {
    if(arr1.length !== arr2.length) return false
    return arr1.every((item, index) => item === arr2[index])
}
```

38.ES6遍历数组的方法

```
const arr1 = ['pink', 'blue', 'red']
const arr2 = [1, 2, 3, 4, 5, 6, 7, 8]

// forEach
arr1.forEach((item, index) => {
    console.log(item)
    console.log(index)
})
// map
// 最终会返回一个新数组 ['pink老师', 'blue老师', 'red老师']
const newArr1 = arr1.map((item,index) => {
    console.log(item)
    console.log(index)
    return item + '老师'
})
// filter
// 最终会返回一个满足条件的数组 [4, 5, 6, 7, 8]
const newArr2 = arr2.filter((item, index) => {
    return item > 3
})
// some
// 有一个元素满足条件就返回 true
// 数组 arr2 中存在元素 大于6，所以最终返回 true
const res1 = arr2.some((item, index) => {
    console.log(item)
    console.log(index)
    return item > 6
})
// every
// 所有元素都满足条件才会返回 true
// 数组 arr2 中存在元素 不满足 >6 的条件，所以最终返回 false
const res2 = arr2.every((item, index) => {
    console.log(item)
    console.log(index)
    return item > 6
})
// find
// 返回第一个满足条件的元素
// re3 最终结果为 6
const res3 = arr2.find((item, index) => {
    return item > 5
})
// reduce
// 累加
// res4 最终结果为 36
const res4 = arr2.reduce((prev, item) => {
    return prev + item
} , 0)
```

39.前端怎么给数据加密

我了解几种加密方式

1. **Base 64 编码**（这个不能叫加密，只是编码）
2. **MD5 加密**（这种方式不可逆，只能加密不能解密）
3. **AES 对称加密**（这种可以解密）

40.两个html文件怎么传值

- url 参数传值

```
<!-- a.html -->
<!-- 这种路径的写法需要两个文件在同一个文件夹下 -->
<a href="b.html?name=Tom&age=18">跳转</a>
```

```
const urlParams = new URLSearchParams(location.search);
console.log(urlParams.get('name')); // "Tom"
```

> **前端路径三种写法:**
>
> 1. **相对路径**
>
>   - `b.html` ➜ 表示「当前路径下」找 `b.html`
>   - `./b.html` ➜ 一样，表示「当前目录」
>   - `../b.html` ➜ 表示「上一级目录」
>
> 2. **绝对路径（站内）**
>
>    ```
>    <a href="/b.html"></a>
>    ```
>
>   - 这会从**网站根目录**去找 `/b.html`
>   - 假设你的网站部署在 `http://localhost:3000/`
>
>   它会请求 `http://localhost:3000/b.html`
>
> 3. **完整路径（站外）**
>
>    ```
>    <a href="https://example.com/b.html"></a>
>    ```
>
>   跳转到外部网站的 b.html，这个就跟你文件夹没关系了。

> `location.search` **是啥?**
>
> `location` 是浏览器提供的全局对象，代表当前页面的 URL 信息
>
> `location.search` 就是从 **问号 ? 开始到结尾**的参数串
>
> `new URLSearchParams()` **是啥?**
>
> 这是一个内置类，专门用来处理 `?参数=值` 这种字符串，里面也封装了一些方法，将`?参数=值` 这种字符串放进去它会立刻**解析成键值对格式**

- localStorage / sessionStorage

```
// a.html
localStorage.setItem('msg', 'hello');
```

```
// b.html
let msg = localStorage.getItem('msg');
```

- cookie（跨页面共享数据）
- postMessage（跨窗口通信）

41.vue3了解多少

42.说一下 js 变量提升

js 在执行前，会把 var 声明的变量提前到作用域顶部，但不会提升赋值部分，函数声明也会被整体提升（包括函数体），这就是 js 的变量提升

> 由于 var 声明的变量没有提升赋值部分，而函数声明整体提升包括函数体，这导致变量提升的结果，可以在 var 变量初始化之前访问该变量，但返回的是 undefined。函数声明之前可以调用该函数

这里涉及一个 TDZ 暂时性死区的概念

暂时性死区（TDZ）是指:

> 在 let 和 const 声明的变量被创建后、被声明之前，在这段时间内如果访问这个变量，就会抛出 `ReferenceError`

换句话说:

- js 在进入作用域时会提前"知道"你用了 let / const 声明了变量（尽管代码还没有执行到那里）
- 但是在真正执行到那一行代码之前，它**不让你用**
- 从**作用域开始**到**变量声明之前**这一段禁止访问的区域，就是 TDZ

```
{
  // 这里是块级作用域的开始
```

```
  // JS 进入这个块级作用域后
  // 编译阶段（预处理）：知道你后面会声明 let a 于是给这个变量分配了内存
  // 创建了变量 a，但没初始化（TDZ 开始），这个阶段变量 a 不能用
  console.log(a); // 报错：此时还在 TDZ 中
  // ReferenceError: Cannot access 'a' before initialization
  let a = 10;     // 代码执行到这里：声明 + 初始化完成，TDZ 结束
  console.log(a); // 输出 10
}
```

> 区分创建变量、声明变量、变量初始化：
>
> js 执行代码之前，会先经历**预处理**阶段（编译阶段）这时候：
>
> 1. 创建变量：js 引擎会扫描作用域，发现 let / const 声明的变量后会“预先创建”这个变量，给这个变量分配内存
> 2. 未声明之前：此时 js 进入作用域，但还没有读到声明赋值那行代码，处在 TDZ 中
> 3. 真正执行代码时：当 js 读到`let x = 10`这一行，此时完成了 声明变量 + 变量初始化，之后这个变量才能访问

概念	发生时间	作用
创建变量	编译阶段（预处理）	JS 引擎知道有这个变量，分配了内存，但还不能用
声明变量	写代码那一行	程序执行到 `let x` 这时才算真正声明
初始化变量	同一行的赋值部分	给变量赋初始值，比如 `= 10`

43.说一下外边距合并和塌陷问题

外边距合并

场景：垂直排列的兄弟元素，上下 margin 会合并

现象： 取两个 margin 中的较大值生效

外边距塌陷

场景：父子级的标签，子级在添加 上外边距 会产生塌陷问题

现象：导致父级一起向下移动

解决方法：

- 取消子级 margin，父级设置 padding**（推荐）**

  > 配合 box-sizing: border-box

- 父级设置 overflow: hidden（溢出隐藏）

  > 浏览器会去寻找父级盒子范围的边界位置，找到之后把内容多出的部分修剪掉。在寻找边界位置的过程中，浏览器发现父级盒子位置出现了问题，然后顺便就把盒子位置修正了

- 父级设置 border-top

  > 原理与 overflow: hidden 相同，给父级盒子加上边界需要先找到盒子的边缘位置，在寻找盒子位置的过程中发现了位置的问题然后将问题解决

44.`$route 和 $router` 的区别

$route 是路由信息对象，包括当前路径、查询参数、路由名字等路由信息参数

```
this.$route.path        // 当前路径，比如 '/user/123'
this.$route.params      // 动态路由参数，比如 { id: '123' }
this.$route.query       // 查询参数，比如 { keyword: 'vue' }
this.$route.name        // 路由名字
this.$route.fullPath    // 完整路径 '/user/123?keyword=vue'
```

$router 是路由实例对象，包括了路由跳转方法等

```
this.$router.push()      // 编程式跳转（前进）
this.$router.go(-1)      // 后退一步
```

45.事件流

事件流指的是事件完整执行过程中的流动路径，会经历两个阶段，分别是捕获阶段、冒泡阶段

简单来说：捕获阶段是 从父到子，冒泡阶段是 从子到父

实际工作都是使用事件冒泡为主

`e.stopPropagation()` 可以阻断事件传播，不光在冒泡阶段有效，捕获阶段也有效

- **事件捕获**

  当一个元素触发事件后，会从DOM的根元素开始依次向下调用所有元素的**同名事件**

  ```
  // 需要设置捕获属性开启事件捕获，点击子盒子后，会依次弹出 我是爷爷、我是爸爸、我是儿子（从外到内）
  <div class="father">
      <div class="son"></div>
  </div>

  const father = document.querySelector('.father')
  const son = document.querySelector('.son')
  document.addEventListener('click', () => {
      alert('我是爷爷')
      // 阻止捕获（不常用）
      // 会跳过设置 stopPropagation 的事件
      // 点击子盒子后，会依次弹出 我是爸爸、我是儿子
      e.stopPropagation()
  ```

```
}, true)
father.addEventListener('click', () => {
    // 如果在这儿阻止捕获
    // 点击子盒子后，会依次弹出 我是爷爷、我是儿子
    alert('我是爸爸')
}, true)
son.addEventListener('click', () => {
    alert('我是儿子')
}, true)
```

- **事件冒泡（默认）**

  当一个元素触发事件后，会依次向上调用所有元素的**同名事件**

```
// 默认冒泡，点击子盒子后，会依次弹出 我是儿子、我是爸爸、我是爷爷（从内到外）
<div class="father">
    <div class="son"></div>
</div>

const father = document.querySelector('.father')
const son = document.querySelector('.son')
document.addEventListener('click', () => {
    alert('我是爷爷')
})
father.addEventListener('click', () => {
    alert('我是爸爸')
})
son.addEventListener('click', (e) => {
    alert('我是儿子')
    // 阻止冒泡
    // 点击子盒子后只会弹出 我是儿子
    e.stopPropagation()
})
```

46.说说你对 Promise 的理解

Promise 是 JavaScript 中用于处理异步操作的对象。它代表一个尚未完成但会在未来完成的操作结果。

- Promise 的状态

> 作用：状态改变后，调用关联的处理函数
>
> 注意：Promise对象一旦被兑现 / 拒接 就是已敲定了，状态无法再被改变了
>
> Promise对象刚创建时状态为 pending，但是对象创建时，回调函数里的代码就会执行了

  - 待定（pending）：初始状态，既没有被兑现，也没有被拒接
  - 已兑现（fulfilled）：操作成功完成 => 执行 .then() 回调
  - 已拒绝（rejected）：操作失败 => 执行 .catch() 回调

- Promise 的方法

47.JavaScript代码执行顺序

1. 从上到下将同步代码放入**调用栈**执行，遇到异步代码（宏任务/微任务）交给**宿主环境（浏览器）**，异步代码有结果则其回调函数进入**对应队列**
2. 当调用栈空闲时，先清空微任务队列内所有微任务的回调函数，然后执行宏任务队列内第一个宏任务的回调函数（微任务优先级大于宏任务）
3. 之后再检查微任务队列内是否有待执行的代码，如果有则清空微任务队列，如果没有则继续执行下一个宏任务的回调函数
4. 依次循环

> js代码是由JS引擎在执行
>
> 执行代码靠JS引擎，浏览器提供环境和webAPI
>
> 只要调用栈空闲，先去看看微任务队列，再去看宏任务队列。因为执行宏任务的回调函数也是在调用栈，这就意味着每执行一个宏任务的回调函数JS引擎都会去微任务队列看看

48.回调函数地狱

概念：在回调函数中嵌套回调函数，一直嵌套下去就形成了回调函数地狱

缺点：可读性差，异常无法捕获，耦合性严重，牵一发而动全身

```
// 回调函数套回调函数形成回调函数地狱
// 1. 获取默认第一个省份的名字
axios({url: 'http://hmajax.itheima.net/api/province'}).then(result => {
    const pname = result.data.list[0]
    document.querySelector('.province').innerHTML = pname
    // 2. 获取第一个城市的名字
    axios({url: 'http://hmajax.itheima.net/api/city', params: { pname
}}).then(result => {
        const cname = result.data.list[0]
        document.querySelector('.city').innerHTML = cname
        // 3. 获取第一个地区的名字
        // 故意写错 url 地址
        axios({url: 'http://hmajax.itheima.net/api/area1', params: { pname, cname
}}).then(result => {
            const areaName = result.data.list[0]
            document.querySelector('.area').innerHTML = areaName
        })
    })
}).catch(error => {
    // 不能捕获到内部的错误
    console.dir(error)
})
```

Promise链式调用解决回调函数地狱

```
// 1. 获取默认第一个省份的名字
axios({url: 'http://hmajax.itheima.net/api/province'}).then(result => {
    const pname = result.data.list[0]
    document.querySelector('.province').innerHTML = pname
    // 2. 获取第一个城市的名字
    return axios({url: 'http://hmajax.itheima.net/api/city', params: { pname }})
}).then(result => {
    const cname = result.data.list[0]
    document.querySelector('.city').innerHTML = cname
    // 3. 获取第一个地区的名字
    // 故意写错 url 地址
    return axios({url: 'http://hmajax.itheima.net/api/area1', params: { pname,
cname }})
}).then(result => {
        const areaName = result.data.list[0]
        document.querySelector('.area').innerHTML = areaName
}).catch(error => {
    // 上面任何一个 .then() 中出现了问题都可以被捕获到
    console.dir(error)
})
```

更直观化一下：

```
// axios底层是xhr外面包了一层Promise，本质也是一个Promise对象
// 内部请求成功就 resolve()，请求失败就 reject()
// 所以 axios 可以 .then() 和 .catch()
// axios底层逻辑后面再挖
axios()
  .then()
  .then()
  .catch(error => {
    console.error('出错了', err);
  })
```

为什么说这样就解决了回调函数地狱问题?

回调函数地狱的问题：可读性差，异常无法捕获，耦合性严重

而Promise链式调用更直观，每一步逻辑清晰；Promise 也提供了统一的错误处理机制 .catch()，可以捕获到链式调用中的任何错误

Promise对象的 .catch() 为什么能捕获到链式调用中的任何错误?

- 代码从上往下执行，一旦有 `.then()` 里抛出异常或者返回一个 rejected 的 Promise
- 这个错误就会顺着链条往后冒泡
- 这个错误遇到的第一个 `.catch()` 会接管这个错误

```
Promise.resolve()
  .then(() => {
    console.log('第一个then');
    return 123;
  })
  .then(() => {
    console.log('第二个then');
    throw new Error('这里出错了');
  })
  .then(() => {
    console.log('第三个then');
  })
  .catch(err => {
    console.log('被catch到的错误：', err.message);
  })

// 输出结果
// 第一个then
// 第二个then
// 被catch到的错误：  这里出错了
```

第三个 `.then()` 根本没执行，因为在第二个 `.then()` 里抛出错误后，就直接跳到 `.catch()` 了

```
// 快速创建一个“成功状态”(fulfilled)的 Promise 对象
const p = Promise.resolve(123)
// 这行代码等价于：
const p = new Promise((resolve, reject) => {
  resolve(123);
})
```

49.async / await 对比 Promise 的优势是什么?

- Promise 解决了回调函数地狱，但是 `.then().then().then()` 看起来还是没有那么清晰，async / await 使异步代码看起来像同步代码一样，更清晰自然
- Promise处理错误用 `.catch()` ，async / await 用 `try-catch` ，两者都可以统一捕获错误，而后者避免了深层次的嵌套，直接包住所有 await ，更接近同步代码逻辑，可读性更好

50.Promise.all 和 Promise.race 分别有哪些使用场景？有什么区别?

Promise.all()

- 概念语法：合并多个 Promise 对象，当所有 Promise 都成功时 => .then() ，如果有任何一个 Promise 失败 => .catch()

```
const p = Promise.all([Promise1, Promise2, Promise3])
p.then(result => {
    // 包含所有Promise的成功结果
```

```
    // 结果数组的顺序和合并时的顺序一致
    console.log(result)
}).catch(error => {
    // 第一个失败的Promise对象抛出的异常
    console.dir(error)
})
```

- 使用场景：当需要同一时间显示多个请求的结果时，就需要把多个请求合并。例如，在一个天气预报网站的页面同时显示了多个地区的天气情况，就需要用 Promise.all() 合并每个地区的天气请求

Promise.race()

- 概念语法：多个 Promise 对象谁先返回结果（无论成功或失败），就用谁的结果

```
const p1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("p1 成功了")
    }, 2000)
})
const p2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject("p2 失败了")
    }, 3000)
})
const p3 = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject("p3 失败了")
    }, 1000)
})

const p = Promise.race([p1, p2, p3])
p.then(result => {
    console.log(result)
}).catch(error => {
    // p3 最快
    console.dir(error) // p3 失败了
})
```

- 使用场景：请求超时控制

  假设有一个异步请求，但是不希望它耗时太长，如果超过一定时间没有响应，就自动取消并返回超时错误。

```
// 模拟一个网络请求，随机 1-5 秒返回数据
function fetchData () {
    return new Promise((resolve) => {
        const delay = 1000 + Math.random() * 4000
        setTimeout(() => {
            resolve("获取数据成功")
```

```
        }, delay)
    })
}
// 创建一个3秒后会拒绝的Promise
function timeoutPromise (timeout) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject(new Error("请求超时"))
        }, timeout)
    })
}
// Promise.race() 会返回最先完成的Promise
// 如果请求在3秒内完成，则返回请求结果
// 如果3秒后请求还没完成，则返回超时结果
Promise.race([fetchData, timeoutPromise(3000)]).then(res => {
    // 如果请求快（<3秒）
    console.log(res) // 获取数据成功
}).catch(err => {
    // 如果请求慢（>3秒）
    console.log(err.messsage) // 请求超时
})
```