面试题

自我介绍

- 面试官你好，我叫艾合麦提，2016年毕业于西安电子科技大学，有6年的工作经验，3份工作经历。
- 第一份工作是在华为，在华为从事过web前端和java后端的开发，前端模块参与过os系统中告警，设备面板等模块的开发
- 独立负责部门内部面板系统的前后端开发。当时用的前端技术栈是React
- 第二份工作是在腾讯云，参与腾讯会议前端相关开发，参与过官网，内嵌h5页面，用户管理部分的预定会议，云录制模块的开发
- 第三份工作是在凯捷咨询，是一个外企，参与developer portal一站式服务平台的前端开发，用的技术栈是react

Javascript相关

1. 介绍 js 的基本数据类型

> js 一共有六种基本数据类型，分别是 Undefined、Null、Boolean、Number、String，还有在 ES6 中新增的 Symbol 类型，代表创建后独一无二且不可变的数据类型，它的出现我认为主要是为了解决可能出现的**全局变量冲突**的问题。

2. JavaScript 有几种类型的值？你能画一下他们的内存图吗？

> js 可以分为两种类型的值，一种是基本数据类型，一种是复杂数据类型。
>
> 基本数据类型：Undefined、Null、Boolean、Number、String，还有在 ES6 中新增的 Symbol 类型
>
> 复杂数据类型指的是 Object 类型，所有其他的如 Array、Date 等数据类型都可以理解为 Object 类型的子类。
>
> 两种类型间的主要区别是它们的存储位置不同，基本数据类型的值直接保存在栈中，而复杂数据类型的值保存在堆中，通过使用在栈中保存对应的指针来获取堆中的值。

3. 什么是堆？什么是栈？它们之间有什么区别和联系？

> 堆和栈的概念存在于数据结构中和操作系统内存中。
>
> 在数据结构中，栈中数据的存取方式为**先进后出**。而堆是一个优先队列，是按优先级来进行排序的.
>
> 在操作系统中，内存被分为栈区和堆区。 栈区内存由**编译器自动分配释放**，存放函数的**参数值，局部变量的值**等。其操作方式类似于数据结构中的栈。
>
> 堆区内存一般由程序员分配释放，若程序员不释放，**程序结束**时可能由垃圾回收机制回收。

4. 内部属性 [[Class]] 是什么？

> 所有 typeof 返回值为 "object" 的对象（如数组）都包含一个内部属性 [[Class]]
>
> 这个属性无法直接访问，一般通过 Object.prototype.toString(..) 来查看。

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

5. 介绍 js 有哪些内置对象?

> js 中的内置对象主要指的是在**程序执行前**存在**全局作用域**里的由 **js 定义的**一些全局值属性、函数和用来实例化其他对象的构造函数对象。
>
> 一般我们经常用到的如全局变量值 NaN、undefined，全局函数如 parseInt()、parseFloat() 用来实例化对象的构造函数如 Date、Object 等，还有提供数学计算的单体内置对象如 Math 对象。

6. undefined 与 undeclared 的区别?

> 已在作用域中**声明但还没有赋值**的变量，是 undefined 的。相反，还没有在作用域中声明过的变量，是 undeclared 的。
>
> 对于 undeclared 变量的引用，浏览器会报引用错误，如 ReferenceError: b is not defined 。 但是我们可以使用 typeof 的安全防范机制来避免报错，因为对于 undeclared（或者 not defined ）变量，typeof 会返回 "undefined"。

7. null 和 undefined 的区别?

> 首先 Undefined 和 Null 都是**基本数据类型**，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。
>
> undefined 代表的含义是**未定义**，null 代表的含义是**空对象**。一般变量声明了但还没有定义的时候会返回 undefined，null 主要用于赋值给一些可能会返回对象的变量，作为初始化。

8. 如何获取安全的 undefined 值?

> 按惯例我们用 void 0 来获得 undefined。

9. 说几条写 JavaScript 的基本规范?

> 在平常项目开发中，我们遵守一些这样的基本规范，比如说：
>
> （1）一个函数作用域中**所有的变量声明应该尽量提到函数首部**，用一个 var 声明，不允许出现两个连续的 var 声明，声明时如果**变量没有值**，应该给该变量赋值**对应类型的初始值**，便于他人阅读代码时，能够一目了然的知道变量对应的类型值。
>
> （2）代码中出现地址、时间等字符串时需要使用**常量**代替。
>
> （3）在进行比较的时候吧，尽量使用'===', '!=='代替'==', '!='。
>
> （4）不要在**内置对象的原型上添加方法**，如 Array, Date。
>
> （5）switch 语句必须带有 default 分支。
>
> （6）for 循环必须使用大括号。
>
> （7）if 语句必须使用大括号。

10. JavaScript 原型，原型链？ 有什么特点?

> 在 js 中我们是使用**构造函数**来**新建一个对象**的，每一个构造函数的内部都有一个 prototype 属性值，这个**属性值是一个对象**，这个对象包含了可以由该**构造函数的所有实例**共享的**属性和方法**。当我们使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。
>
> 一般来说我们是不应该能够获取到这个值的，但是现在浏览器中都实现了 **proto** 属性来让我们访问这个属性，但是我们最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 **Object.getPrototypeOf()** 方法，我们可以通过这个方法来获取对象的原型。
>
> 当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的**原型对象里找这个属性**，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是**原型链**的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是我们新建的对象为什么能够使用 toString() 等方法的原因。
>
> 特点:
>
> JavaScript 对象是通过**引用**来传递的，我们创建的每个**新对象实体**中并**没有一份属于自己的原型副本**。当我们修改原型时，与之相关的对象也会继承这一改变。

11. js 获取原型的方法

> p.proto p.constructor.prototype Object.getPrototypeOf(p)

```
function Person() {
    this.name = 'aaa'
}

let p = new Person()
p.__proto__
p.construnctor.prototype
Person.prototype
Object.getPrototypeOf(p)
```

12. 在 js 中不同进制数字的表示方式

- 以 0X、0x 开头的表示为十六进制。
- 以 0、0O、0o 开头的表示为八进制。
- 以 0B、0b 开头的表示为二进制格式。

13. js 中整数的安全范围是多少?

> 安全整数指的是，在这个范围内的整数转化为二进制存储的时候不会出现精度丢失，能够被“安全”呈现的最大整数是 2^53 - 1，在 ES6 中被定义为 **Number.MAX_SAFE_INTEGER, Number.MIN_SAFE_INTEGER**

14. typeof NaN 的结果是什么?

> NaN 意指“不是一个数字”（not a number），用于指出数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果”。
>
> typeof NaN; // "number"

15. isNaN 和 Number.isNaN 函数的区别?

> 函数 isNaN(numner) 接收参数后，会尝试将这个**参数**转换**为数值**，任何不能被转换为数值的的值都会返回 true，因此非数字值传入也会返回 true ，会影响 NaN 的判断。

> 函数 Number.isNaN 会**首先**判断**传入参数是否为数字**，如果是数字再继续判断是否为 NaN ，这种方法对于 NaN 的判断更为准确。

16. Array 构造函数只有一个参数值时的表现?

> Array 构造函数只带一个数字参数的时候，该参数会被作为**数组的预设长度（length）**，而非只充当数组中的一个元素。
>
> 这样创建出来的只是一个**空数组**，只不过它的 length 属性被设置成了指定的值。
>
> 构造函数 Array(..) 不要求必须带 new 关键字。不带时，它会被自动补上。

17. 其他值到字符串的转换规则?

> （1）Null 和 Undefined 类型 ，null 转换为 "null"，undefined 转换为 "undefined"，
>
> （2）Boolean 类型，true 转换为 "true"，false 转换为 "false"。
>
> （3）Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。
>
> （4）Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。
>
> （4）对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString()（Object.prototype.toString()）来返回内部属性 [[Class]] 的值，如"[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会调用该方法并使用其返回值。

18. 其他值到数字值的转换规则?

> （1）Undefined 类型的值转换为 NaN。
>
> （2）Null 类型的值转换为 0。
>
> （3）Boolean 类型的值，true 转换为 1，false 转换为 0。
>
> （4）String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。
>
> （5）Symbol 类型的值不能转换为数字，会报错。
>
> （6）对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。

19. 其他值到布尔类型的值的转换规则?

> 以下这些是假值：

- undefined
- null
- false
- +0、-0 和 NaN
- ""

> 假值的布尔强制类型转换结果为 false。 从逻辑上说，假值列表以外的都应该是真值。

20. {} 和 [] 的 valueOf 和 toString 的结果是什么?

> {} 的 valueOf 结果为 {} ，toString 的结果为 "[object Object]"
>
> [] 的 valueOf 结果为 [] ，toString 的结果为 ""

21. ~ 操作符的作用?

> ~ 返回 2 的补码，并且 ~ 会将数字转换为 32 位整数，因此我们可以使用 ~ 来进行取整操作。
>
> ~x 大致等同于 -(x+1)。

22. ∘ 操作符什么时候用于字符串的拼接?

> 简单来说就是，如果 + 的其中一个操作数是**字符串**（或者通过以上步骤最终得到字符串），则执行字符串拼接，否则执行数字加法。

23. javascript 创建对象的几种方式?

> 我了解到的方式有这么几种：
>
> （1）第一种是工厂模式，工厂模式的主要工作原理是用**函数来封装创建对象的细节**，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是**创建出来的对象无法和某个类型联系起来**，它只是简单的封装了复用代码
>
> （2）第二种是**构造函数模式**。js 中每一个函数都可以作为构造函数，只要一个函数是**通过 new 来调用的，那么我们就可以把它称为构造函数**。执行构造函数**首先会创建一个对象**，然后将对象的**原型指向构造函数的 prototype 属性**，然后将执行上下文中的 **this 指向这个对象**，**最后再执行整个函数**，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此我们可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系.
>
> （3）第三种模式是**原型模式**，因为每一个函数都有一个 **prototype** 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此我们可以使用原型对象来添加公用属性和方法，从而实现代码的复用。
>
> （4）第四种模式是**组合使用构造函数模式和原型模式**，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此我们可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。？（5）第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。
>
> （6）第六种模式是**寄生构造函数模式**，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

24. JavaScript 继承的几种实现方式?

> 我了解的 js 中实现继承的几种方式有：
>
> （1）第一种是以**原型链**的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

```
// Game类
function Game() {
    this.name = 'lol'
}
Game.prototype.getName = function() {
    return this.name;
}
// LOL类
function LOL() {}
LOL.prototype = new Game();
```

```
LOL.prototype.constructor = LOL;
const game = new LOL();
// 本质：重写原型对象方式，将父对象的属性方法，作为子对象原型对象的属性和方法
```

（2）第二种方式是使用借用**构造函数**的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

```
function Game(arg) {
    this.name = 'lol';
    this.skin = ['s'];
}
Game.prototype.getName = function() {
    return this.name;
}

// LOL类
function LOL(arg) {
    Game.call(this, arg);
}

const game3 = new LOL('arg');
// 解决了共享属性的问题 + 子向父传参问题
```

（3）第三种方式是**组合继承**，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

```
function Game(arg) {
    this.name = 'lol';
    this.skin = ['s'];
}
Game.prototype.getName = function() {
    return this.name;
}

// LOL类
function LOL(arg) {
    Game.call(this, arg);
}
LOL.prototype = new Game();
LOL.prototype.constructor = LOL;

const game3 = new LOL();
```

（4）第四种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。

```
function Game(arg) {
    this.name = 'lol';
    this.skin = ['s'];
}
Game.prototype.getName = function() {
    return this.name;
}

// LOL类
function LOL(arg) {
    Game.call(this, arg);
```

```
}
LOL.prototype = Object.create(Game.prototype);
LOL.prototype.constructor = LOL;
```

25. Javascript 的作用域链?

> 作用域链的作用是保证对**执行环境有权访问的所有变量和函数的有序访问**，通过作用域链，我们可以访问到外层环境的变量和函数。
>
> 作用域链的本质上是一个**指向**变量对象的指针列表**。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的**前端**始终都是**当前执行上下文的变量对象**。全局执行上下文的变量对象（也就是**全局对象**）始终是作用域链的**最后一个对象**。
>
> 当我们查找一个变量时，如果当前执行环境中没有找到，我们可以沿着作用域链向后查找。

26. 谈谈 This 对象的理解。

> this 是执行上下文中的一个属性，它指向**最后一次调用这个方法**的对象。在实际开发中，this 的指向可以通过四种调用模式来判断。 1.第一种是**函数调用模式**，当一个函数不是一个对象的属性时，直接作为函数来调用时，**this 指向全局对象**。
>
> 第二种是**方法调用模式**，如果一个函数作为一个**对象的方法**来调用时，this 指向**这个对象**。
>
> 第三种是**构造器调用模式**，如果一个函数用 new 调用时，函数执行前会新创建一个对象，**this 指向这个新创建的对象**。
>
> 第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以**显示的指定调用函数的 this 指向**。其中
>
> apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。
>
> call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。
>
> bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变
>
> 这四种方式，使用构造器调用模式的优先级最高，然后是 apply 、 call 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

26. 什么是 DOM 和 BOM?

> DOM 指的是**文档对象模型**，它指的是把文档当做一个对象来对待，这个对象主要定义了**处理网页内容的方法和接口**。
>
> BOM 指的是**浏览器对象模型**，它指的是把浏览器当做一个对象来对待，这个对象主要定义了**与浏览器进行交互的法和接口**。
>
> BOM的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局）对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。
>
> window 对象含有 location 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对象的子对象。

27. 写一个通用的事件侦听器函数。

```
const EventUtils = {
  // 视能力分别使用dom0||dom2||IE方式 来绑定事件
  // 添加事件
  addEvent: function(element, type, handler) {
    if (element.addEventListener) {
      element.addEventListener(type, handler, false);
    } else if (element.attachEvent) {
      element.attachEvent("on" + type, handler);
    } else {
      element["on" + type] = handler;
    }
  },

  // 移除事件
  removeEvent: function(element, type, handler) {
    if (element.removeEventListener) {
      element.removeEventListener(type, handler, false);
    } else if (element.detachEvent) {
      element.detachEvent("on" + type, handler);
    } else {
      element["on" + type] = null;
    }
  },

  // 获取事件目标
  getTarget: function(event) {
    return event.target || event.srcElement;
  },

  // 获取 event 对象的引用，取到事件的所有信息，确保随时能使用 event
  getEvent: function(event) {
    return event || window.event;
  },

  // 阻止事件（主要是事件冒泡，因为 IE 不支持事件捕获）
  stopPropagation: function(event) {
    if (event.stopPropagation) {
      event.stopPropagation();
    } else {
      event.cancelBubble = true;
    }
  },

  // 取消事件的默认行为
  preventDefault: function(event) {
    if (event.preventDefault) {
      event.preventDefault();
    } else {
      event.returnValue = false;
    }
  }
};
```

28. 事件是什么？IE 与火狐的事件机制有什么区别？ 如何阻止冒泡？

> 1.事件是用户**操作网页**时发生的**交互动作**，比如 click/move，
>
> 事件除了用户触发的动作外，还可以是文档加载，窗口滚动和大小调整。
>
> 事件被封装成一个 event 对象，包含了该事件发生时的所有相关信息（ event 的属性）以及可以对事件进行的操作（ event 的方法）。
>
> 2.事件处理机制：IE 支持事件冒泡、Firefox 同时支持两种事件模型，也就是：**事件冒泡和事件捕获**。
>
> 3.event.stopPropagation() 或者 ie 下的方法 event.cancelBubble = true;

29. 三种事件模型是什么？

> 事件是用户操作网页时发生的交互动作或者网页本身的一些操作，现代浏览器一共有三种事件模型。

> 第一种事件模型是最早的 **DOM0 级模型**，这种模型**不会传播**，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现，它可以在网页中**直接定义监听函数**，也可以通过 js 属性来指定**监听函数**。这种方式是所有浏览器都兼容的。
>
> 第二种事件模型是 **IE 事件模型**，在该事件模型中，一次事件共有**两个过程**，事件**处理**阶段，和事件**冒泡**阶段。事件处理阶段会**首先执行目标元素绑定的监听事件**。然后是事件冒泡阶段，冒泡指的是事件从**目标元素冒泡到 document**，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。这种模型通过 attachEvent 来添加监听函数，可以添加多个监听函数，会按顺序依次执行。
>
> 第三种是 **DOM2 级事件模型**，在该事件模型中，一次事件共有三个过程，第一个过程是事件捕**获阶段**。捕获指的是事件从 **document 一直向下传播到目标元素**，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和 IE 事件模型的两个阶段相同。这种事件模型，事件绑定的函数是 addEventListener，其中第三个参数可以指定事件是否在捕获阶段执行。

30. 事件委托是什么?

> 事件委托本质上是利用了**浏览器事件冒泡的机制**。因为事件在冒泡过程中会上传到父节点，并且父节点可以通过事件对象获取到目标节点，因此可以把子**节点的监听函数定义在父节点上**，由**父节点的监听函数统一处理多个子元素的事件**，这种方式称为事件代理。
>
> 使用事件代理我们可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理我们还可以实现事件的动态绑定，比如说新增了一个子节点，我们并不需要单独地为它添加一个监听事件，它所发生的事件会交给父元素中的监听函数来处理。

31. 什么是闭包，为什么要用它?

> 闭包是指**有权访问另一个函数作用域中变量的函数**，创建闭包的最常见的方式就是在**一个函数内创建另一个函数**，创建的函数可以访问到当前函数的局部变量。
>
> 闭包有两个常用的用途。
>
> 1. 使我们在函数外部能够访问到函数内部的变量。通过使用闭包，我们可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建**私有变量**。

> 2. 使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

32. 如何判断一个对象是否属于某个类?

> 1. 使用 instanceof 运算符来判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

> 2. 通过对象的 constructor 属性来判断，对象的 constructor 属性指向该对象的构造函数，但是这种方式不是很安全，因为 constructor 属性可以被改写。
>
> 3.如果需要判断的是某个内置的引用类型的话，可以使用 Object.prototype.toString() 方法来打印对象的 [[Class]] 属性来进行判断。

33. instanceof 的作用?

> instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。

```
function myInstanceof(left, right) {
  let proto = Object.getPrototypeOf(left), // 获取对象的原型
    prototype = right.prototype; // 获取构造函数的 prototype 对象

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
    if (proto === prototype) return true;
```

```
    proto = Object.getPrototypeOf(proto);
  }
}
```

34. new 操作符具体干了什么呢？如何实现？

> 1）首先创建了一个新的空对象
>
> 2）设置原型，将对象的原型设置为函数的 prototype 对象。
>
> 3）让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）
>
> 4）判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。

```
// 实现:

function objectFactory() {
  let newObject = null,
    constructor = Array.prototype.shift.call(arguments),
    result = null;

  // 参数判断
  if (typeof constructor !== "function") {
    console.error("type error");
    return;
  }

  // 新建一个空对象，对象的原型为构造函数的 prototype 对象
  newObject = Object.create(constructor.prototype);

  // 将 this 指向新建对象，并执行函数
  result = constructor.apply(newObject, arguments);

  // 判断返回对象
  let flag =
    result && (typeof result === "object" || typeof result === "function");

  // 判断返回结果
  return flag ? result : newObject;
}

// 使用方法
// objectFactory(构造函数, 初始化参数);
```

35. js 延迟加载的方式有哪些？

- defer 属性
- async 属性
- 动态创建 DOM 方式
- 使用 setTimeout 延迟方法
- 让 JS 最后加载

> js 的加载、解析和执行会阻塞页面的渲染过程，因此我们希望 js 脚本能够尽可能的延迟加载，提高页面的**渲染速度**。

> 我了解到的几种方式是：
>
> 第一种方式是我们一般采用的是将 js 脚本放在**文档的底部**，来使 js 脚本尽可能的在最后来加载执行。
>
> 第二种方式是给 js 脚本添加**defer** 属性，这个属性会让**脚本的加载与文档的解析同步解析**，然后在**文档解析完成后再执行这个脚本文件**，这样的话就能使页面的渲染**不被阻塞**。

第三种方式是给 js 脚本添加 **async** 属性，这个属性会**使脚本异步加载**，不会阻塞页面的解析过程，

第四种方式是**动态创建 DOM 标签的方式**，我们可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。

36. Ajax 是什么? 如何创建一个 Ajax?

它是 Asynchronous JavaScript and XML 的缩写，指的是**通过 JavaScript 的异步通信**，从服务器获取 XML 文档从中**提取数据**，再更新当前网页的**对应部分，而不用刷新整个网页**

```
const SERVER_URL = "/server";
let xhr = new XMLHttpRequest();
// 创建 Http 请求
xhr.open("GET", SERVER_URL, true);
// 设置状态监听函数
xhr.onreadystatechange = function() {
  if (this.readyState !== 4) return;
  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};
// 设置请求失败时的监听函数
xhr.onerror = function() {
  console.error(this.statusText);
};

// 设置请求头信息
xhr.responseType = "json";
xhr.setRequestHeader("Accept", "application/json");

// 发送 Http 请求
xhr.send(null);
```

37. 谈一谈浏览器的缓存机制?

浏览器的缓存机制指的是**通过在一段时间内保留已接收到的 web 资源的一个副本**，如果在资源的**有效时间**内，发起了对这个资源的**再一次请求**，那么浏览器会直接使用**缓存的副本**，而不是向服务器发起请求。

web 资源的缓存策略一般由服务器来指定，可以分为两种，分别是**强缓存策略和协商缓存策略**。

使用**强缓存策略**时，如果缓存资源有效，则直接使用缓存资源，不必再向服务器发起请求。

强缓存策略可以通过两种方式来设置，分别是 http 头信息中的 **Expires** 属性和 **Cache-Control** 属性。

服务器通过在响**应头中添加 Expires 属性**，来指定资源的过期时间。在过期时间以内，该资源可以被缓存使用，不必再向服务器发送请求。这个时间是一个**绝对时间**

Expires 是 http1.0 中的方式，在 http 1.1 中提出了一个新的头部属性就是 > Cache-Control 属性，

它提供了对资源的缓存的更精确的控制。它有很多不同的值，常用的比如我们可以通过设置 **max-age** 来指定资源能够被缓存的时间的大小，这是一个**相对的时间**，它会根据这个时间的大小和资源第一次请求时的时间来计算出资源过期的时间，因此相对于 Expires来说，这种方式更加有效一些。常用的还有比如 private ， 用来规定资源只能被客户端缓存，不能够代理服务器所缓存。 还有如 no-store ， 用来指定资源不能够被缓存， no-cache 代表该资源能够被缓存，但是立即失效，每次都需要向服务器发起请求。

一般来说只需要设置其中一种方式就可以实现强缓存策略，当两种方式一起使用时，Cache-Control 的优先级要高于 Expires 。

> 使用协商缓存策略时，会先向**服务器发送一个请求**，如果**资源没有发生修改**，则返回一个 **304** 状态，让浏览器使用本地的缓存副本。
>
> 如果资源发生了修改，则返回修改后的资源。协商缓存也可以通过两种方式来设置，分别是 http 头信息中的 **Etag** 和 **Last-Modified** 属性。
>
> 服务器通过在响应头中添加 **Last-Modified** 属性来指出资源**最后一次修改的时间**，当浏览器下一次发起请求时，会在请求头中添加一个 **If-Modified-Since** 的属性，属性值为**上一次资源返回时的 Last-Modified 的值**。当请求发送到服务器后服务器会通过这个属性来和资源的最后一次的修改时间来进行比较，以此来判断资源是否做了修改。
>
> 如果资源没有修改，那么返回 304 状态，让客户端使用本地的缓存。
>
> 如果资源已经被修改了，则返回修改后的资源。使用这种方法有一个缺点，就是 Last-Modified 标注的最后修改时间只能精确到**秒级**，如果某些文件在1秒钟以内，被修改多次的话，那么文件已将改变了但是 Last-Modified 却没有改变，这样会造成缓存命中的不准确。
>
> 因为 Last-Modified 的这种可能发生的不准确性，http 中提供了另外一种方式，那就是 **Etag** 属性。服务器在返回资源的时候，在头信息中添加了 Etag 属性，这个属性是资源生成的**唯一标识符**，当**资源发生改变的时候，这个值也会发生改变**。在下一次资源请求时，浏览器会在**请求头中添加一个 If-None-Match** 属性，这个属性的值就是上次返回的资源的 Etag 的值。服务接收到请求后会根据这个值来和资源当前的 Etag 的值来进行比较，以此来判断资源是否发生改变，是否需要返回资源。通过这种方式，比 Last-Modified 的方式更加精确。
>
> 当 Last-Modified 和 Etag 属性同时出现的时候，Etag 的优先级更高。
>
> 使用协商缓存的时候，服务器需要考虑负载平衡的问题，因此多个服务器上资源的 Last-Modified 应该保持一致，因为每个服务器上 Etag 的值都不一样，因此在考虑负载平衡时，最好不要设置 Etag 属性。
>
> 强缓存策略和协商缓存策略在缓存命中时都会直接使用本地的缓存副本，区别只在于**协商缓存会向服务器发送一次请求**。

38. 同步和异步的区别?

> 同步指的是当一个进程在执行某个请求的时候，如果这个请求需要等待一段时间才能返回，那么这个进程会一**直等待下去**，直到消息返回为止再继续向下执行。
>
> 异步指的是当一个进程在执行某个请求的时候，如果这个请求需要等待一段时间才能返回，这个时候进程会**继续往下执行**，不会阻塞等待消息的返回，当消息返回时系统再通知进程进行处理。

39. 什么是浏览器的同源政策?

> 一个域下的 js 脚本在未经允许的情况下，不能够访问另一个域的内容。这里的同源的指的是**两个域的协议、域名、端口号必须相同**，否则则不属于同一个域。
>
> 同源政策主要限制了三个方面
>
> 第一个是当前域下的 js 脚本不能够访问其他域下的 cookie、localStorage 和 > indexDB。
>
> 第二个是当前域下的 js 脚本不能够操作访问操作其他域下的 DOM。
>
> 第三个是当前域下 ajax 无法发送跨域请求。
>
> 同源政策的目的主要是为了保证用户的信息安全，它只是对 js 脚本的一种限制，并不是对浏览器的限制，对于一般的 img、或者script 脚本请求都不会有跨域的限制，这是因为这些操作都不会通过响应结果来进行可能出现安全问题的操作。

40. 如何解决跨域问题?

> 解决跨域的方法我们可以根据我们想要实现的目的来划分。
>
> 首先我们如果只是想要实现主域名下的不同子域名的跨域操作，我们可以使用设置 document.domain 来解决。
>
> （1）将 **document.domain 设置为主域名**，来实现**相同子域名的跨域操作，这个时候主域名下的 cookie 就能够被子域名所访问**。如果是想要解决不同跨域窗口间的通信问题，比如说一个页面想要和页面的中的不同源的 iframe 进行通信的问题，我们可以使用 **location.hash 或者 window.name 或者 postMessage** 来解决。
>
> （2）使用 location.hash 的方法，我们可以在主页面动态的修改 iframe 窗口的 hash 值，然后在 iframe 窗口里实现监听函数来实现这样一个单向的通信。因为在 iframe 是没有办法访问到不同源的父级窗口的，所以我们不能直接修改父级窗口的 hash 值来实现通信，我们可以在 iframe 中再加入一个 iframe，这个 iframe 的内容是和父级页面同源的，所以我们可以 window.parent.parent 来修改最顶级页面的 src，以此来实现双向通信。
>
> 如果是像解决 ajax 无法提交跨域请求的问题，我们可以使用 **jsonp、cors、websocket 协议、服务器代理来解决问题**。

> （3）使用 jsonp 来实现跨域请求，它的主要原理是通过动态构建 script 标签来实现跨域请求，因为浏览器对 script 标签的引入没有跨域的访问限制 。通过在请求的 url 后指定一个回调函数，然后服务器在返回数据的时候，构建一个 json 数据的包装，这个包装就是回调函数，然后返回给前端，前端接收到数据后，因为请求的是脚本文件，所以会直接执行，这样我们先前定义好的回调函数就可以被调用，从而实现了跨域请求的处理。这种方式只能用于 get 请求。
>
> （4）使用 CORS 的方式，CORS 是一个 W3C 标准，全称是"跨域资源共享"。CORS 需要**浏览器和服务器同时支持**。浏览器将 CORS 请求分成两类：简单请求和非简单请求。对于简单请求，浏览器直接发出 CORS 请求。具体来说，就是会在头信息之中，增加一个 Origin 字段。Origin 字段用来说明本次请求来自哪个源。服务器根据这个值，决定是否同意这次请求。对于如果 Origin 指定的源，不在许可范围内，服务器会返回一个正常的 HTTP 回应。浏览器发现，这个回应的头信息没有包含 Access-Control-Allow-Origin 字段，就知道出错了，从而抛出一个错误，ajax 不会收到响应信息。如果成功的话会包含一些以 Access-Control- 开头的字段。

> 非简单请求，浏览器会先发出一次预检请求，来判断该域名是否在服务器的白名单中，如果收到肯定回复后才会发起请求。

> （5）使用 websocket 协议，这个协议没有同源限制。

> （6）使用服务器来代理跨域的访问请求，就是有跨域的请求操作时发送请求给后端，让后端代为请求，然后最后将获取的结果发返回。

41. 简单谈一下 cookie ?

> cookie 是服务器提供的一种用于维护**会话状态信息的数据**，通过**服务器发送到浏览器，浏览器保存在本地**，当下一次有**同源的请求时，将保存的 cookie 值添加到请求头部，发送给服务端**。这可以用来实现记录**用户登录状态**等功能。cookie 一般可以存储 4k 大小的数据，并且只能够被同源的网页所共享访问。
>
> 服务器端可以使用 Set-Cookie 的响应头部来配置 cookie 信息。一条cookie 包括了5个属性值 expires、domain、path、secure、HttpOnly。
>
> 其中 expires 指定了 cookie 失效的时间， domain 是域名、 path是路径，domain 和 path 一起限制了 cookie 能够被哪些 url 访问。> secure 规定了 cookie 只能在确保安全的情况下传输，HttpOnly 规定了这个 cookie 只能被服务器访问，不能使用 js 脚本访问。

42. 模块化开发怎么做?

> 一个模块是实现一个**特定功能的一组方法**。在最开始的时候，js 只实现一些简单的功能，所以并没有模块的概念
>
> 但随着程序越来越复杂，代码的模块化开发变得越来越重要。
>
> 由于函数具有独立作用域的特点，最原始的写法是**使用函数来作为模块**，几个函数作为一个模块，但是这种方式容易造成**全局变量的污染**，并且模块间没有联系。
>
> 后面提出了**对象写法**，通过将函数作为一个对象的方法来实现，这样解决了直接使用函数作为模块的一些缺点，但是这种办法会暴露所有的所有的模块成员，外部代码可以修改内部属性的值。
>
> 现在最常用的是**立即执行函数的写法**，通过**利用闭包**来实现模块私有作用域的建立，同时不会对全局作用域造成污染。

43. js 的几种模块规范?

> js 中现在比较成熟的有四种模块加载方案。
>
> 第一种是 **CommonJS** 方案，它通过 **require 来引入模块，通过 module.exports 定义模块的输出接口**。这种模块加载方案是**服务器端的解决方案**，它是以**同步的方式**来引入模块的，因为在服务端文件都存储在本地磁盘，所以读取非常快，所以以同步的方式加载没有问题。但如果是在浏览器端，由于模块的加载是使用网络请求，因此使用异步加载的方式更加合适。
>
> 第二种是 **AMD** 方案，这种方案采用**异步加载**的方式来加载模块，模块的加载不影响后面语句的执行，所有依赖这个模块的语句都**定义在一个回调函数里**，等到加载完成后再执行回调函数。require.js 实现了 AMD 规范。
>
> 第三种是 CMD 方案，这种方案和 AMD 方案都是为了解决异步模块加载的问题，sea.js 实现了 CMD 规范。它和 require.js的区别在于模块定义时对依赖的处理不同和对依赖模块的执行时机的处理不同。参考 60
>
> 第四种方案是 **ES6** 提出的方案，使用 import 和 export 的形式来导入导出模块。这种方案和上面三种方案都不同。

44. AMD 和 CMD 规范的区别?

```
// CMD
define(function(require, exports, module) {
  var a = require("./a");
  a.doSomething();
  // 此处略去 100 行
  var b = require("./b"); // 依赖可以就近书写
  b.doSomething();
  // ...
});

// AMD 默认推荐
define(["./a", "./b"], function(a, b) {
  // 依赖必须一开始就写好
  a.doSomething();
  // 此处略去 100 行
  b.doSomething();
  // ...
});
```

45. ES6 模块与 CommonJS 模块、AMD、CMD 的差异。

> CommonJS 模块输出的是一个**值的拷贝**，一旦输出一个值，模块内部的变化就影响不到这个值.
>
> ES6 模块输出的是**值的引用**。JS 引擎对脚本静态分析的时候，遇到模块加载命令 import，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

> CommonJS 模块是运行时加载，CommonJS 模块就是对象,即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”
>
> ES6 模块是编译时输出接口。ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

46. requireJS 的核心原理是什么？（如何动态加载的？如何避免多次加载的？如何 缓存的？）

> require.js 的核心原理是通过**动态创建 script 脚本来异步引入模块**，然后对每个脚本的 **load 事件进行监听**，如果每个脚本都加载完成了，再调用回调函数。

47. ECMAScript6 怎么写 class，为什么会出现 class 这种东西?

> 在我看来 ES6 新添加的 class 只是为了补充 js 中缺少的一些**面向对象语言**的特性，但本质上来说它只是一种语法糖，不是一个新的东西，其背后还是原型继承的思想。通过加入 class 可以有利于我们更好的组织代码。
>
> 在 class 中添加的方法，其实是添加在类的原型上的。

48. documen.write 和 innerHTML 的区别?

- document.write 的内容会代替整个文档内容，会重写整个页面。
- innerHTML 的内容只是替代指定元素的内容，只会重写页面中的部分内容。

49. call() 和 .apply() 的区别?

> 它们的作用一模一样，区别仅在于传入参数的形式的不同。
>
> apply 接受**两个参数**，第一个参数**指定了函数体内 this 对象的指向**，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。
>
> call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被依次传入函数。

50. 数组和对象有哪些原生方法，列举一下?

```
数组和字符串的转换方法：toString()、toLocalString()、join() 其中 join() 方法可以指定转换为字符串时的分隔符。

数组尾部操作的方法 pop() 和 push()，push 方法可以传入多个参数。

数组首部操作的方法 shift() 和 unshift() 重排序的方法 reverse() 和 sort()，sort() 方法可以传入一个函数来进行比较，传入前后两个值，如果返回值为正数，则交换两个参数的位置。

数组连接的方法 concat() ，返回的是拼接好的数组，不影响原数组。

数组截取办法 slice()，用于截取数组中的一部分返回，不影响原数组。

数组插入方法 splice()，影响原数组查找特定项的索引的方法，indexOf() 和 lastIndexOf() 迭代方法 every()、some()、filter()、map() 和 forEach() 方法

数组归并方法 reduce() 和 reduceRight() 方法
```

51. JavaScript 中的作用域与变量声明提升?

> 变量提升的表现是，无论我们在函数中何处位置声明的变量，好像都被提升到了函数的首部，我们可以在变量声明前访问到而不会报错。
>
> 造成变量声明提升的本质原因是 js 引擎在代码**执行前有一个解析的过程**，创建了执行上下文，初始化了一些代码执行时需要用到的对象。当我们访问一个变量时，我们会到当前执行上下文中的作用域链中去

> 查找，而作用域链的首端指向的是当前执行上下文的变量对象，这个变量对象是执行上下文的一个属性，它包含了函数的形参、所有的函数和变量声明，这个对象的是在代码解析的时候创建的。这就是会出现变量声明提升的根本原因。

52. 如何编写高性能的 Javascript ?

- 使用位运算代替一些简单的四则运算。
- 避免使用过深的嵌套循环。
- 不要使用未定义的变量。
- 当需要多次访问数组长度时，可以用变量保存起来，避免每次都会去进行属性查找。

53. 哪些操作会造成内存泄漏?

- 意外的全局变量
- 被遗忘的计时器或回调函数
- 脱离 DOM 的引用
- 闭包

> 第一种情况是我们由于**使用未声明的变量**，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。

> 第二种情况是我们设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
>
> 第三种情况是我们获取一个 DOM 元素的引用，而后面这个元素被删除，由于我们一直保留了对这个元素的引用，所以它也无法被回收。
>
> 第四种情况是不合理的使用闭包，从而导致某些变量一直被留在内存当中。

54. 什么是“前端路由”？什么时候适合使用“前端路由”？“前端路由”有哪些优点和缺点?

> （1）什么是前端路由?
>
> 前端路由就是把不同路由对应不同的内容或页面的任务交给前端来做，之前是通过服务端根据 url 的不同返回不同的页面实现的。
>
> （2）什么时候使用前端路由?
>
> 在单页面应用，大部分页面结构不变，只改变部分内容的使用
>
> （3）前端路由有什么优点和缺点?
>
> 优点：用户体验好，不需要每次都从服务器全部获取，快速展现给用户
>
> 缺点：单页面无法记住之前滚动的位置，无法在前进，后退的时候记住滚动的位置
>
> 前端路由一共有两种实现方式，一种是通过 hash 的方式，一种是通过使用 pushState 的方式。

55. 介绍一下 js 的节流与防抖?

```
// 函数防抖：  在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。

// 函数节流：  规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。

// 函数防抖的实现
function debounce(fn, wait) {
  var timer = null;

  return function() {
    var context = this,
```

```
      args = arguments;

    // 如果此时存在定时器的话，则取消之前的定时器重新记时
    if (timer) {
      clearTimeout(timer);
      timer = null;
    }

    // 设置定时器，使事件间隔指定事件后执行
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, wait);
  };
}

// 函数节流的实现;
function throttle(fn, delay) {
  var preTime = Date.now();

  return function() {
    var context = this,
      args = arguments,
      nowTime = Date.now();

    // 如果两次时间间隔超过了指定时间，则执行函数。
    if (nowTime - preTime >= delay) {
      preTime = Date.now();
      return fn.apply(context, args);
    }
  };
}
```

56. js 的事件循环是什么?

> 因为 js 是单线程运行的，在代码执行的时候，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。在执行同步代码的时候，如果遇到了异步事件，js 引擎并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务。当异步事件执行完毕后，再将异步事件对应的回调加入到与当前执行栈中不同的另一个任务队列中等待执行。任务队列可以分为**宏任务对列和微任务对列**，当当前**执行栈**中的事件执行完毕后，js 引擎首先会判断**微任务对列中**是否有任务可以执行，**如果有就将微任务队首的事件压入栈中执行**。当**微任务对列中的任务都执行完成后再去判断宏任务对列中的任务**。
>
> 微任务包括了 promise 的回调、node 中的 process.nextTick 、对 Dom 变化监听的 MutationObserver。
>
> 宏任务包括了 script 脚本的执行、setTimeout ， setInterval, setImmediate 一类的定时事件，还有如 I/O 操作、UI 渲染等。

57. js 中的深浅拷贝实现?

```
// 浅拷贝的实现;

function shallowCopy(object) {
  // 只拷贝对象
  if (!object || typeof object !== "object") return;
  // 根据 object 的类型判断是新建一个数组还是对象
  let newObject = Array.isArray(object) ? [] : {};
  // 遍历 object，并且判断是 object 的属性才拷贝
  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] = object[key];
    }
  }
  return newObject;
}

// 深拷贝的实现;
function deepCopy(object) {
  if (!object || typeof object !== "object") return;
  let newObject = Array.isArray(object) ? [] : {};
  for (let key in object) {
```

```
    if (object.hasOwnProperty(key)) {
      newObject[key] =
        typeof object[key] === "object" ? deepCopy(object[key]) : object[key];
    }
  }
  return newObject;
}
```

57. 手写 call、apply 及 bind 函数

```
// call函数实现
Function.prototype.myCall = function(context) {
  // 判断调用对象
  if (typeof this !== "function") {
    console.error("type error");
  }
  // 获取参数
  let args = [...arguments].slice(1),
    result = null;
  // 判断 context 是否传入，如果未传入则设置为 window
  context = context || window;
  // 将调用函数设为对象的方法
  context.fn = this;
  // 调用函数
  result = context.fn(...args);
  // 将属性删除
  delete context.fn;
  return result;
};

// apply 函数实现
Function.prototype.myApply = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }
  let result = null;
  // 判断 context 是否存在，如果未传入则为 window
  context = context || window;
  // 将函数设为对象的方法
  context.fn = this;
  // 调用方法
  if (arguments[1]) {
    result = context.fn(...arguments[1]);
  } else {
    result = context.fn();
  }
  // 将属性删除
  delete context.fn;
  return result;
};

// bind 函数实现
Function.prototype.myBind = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }
  // 获取参数
  var args = [...arguments].slice(1),
    fn = this;
  return function Fn() {
    // 根据调用方式，传入不同绑定值
    return fn.apply(
      this instanceof Fn ? this : context,
      args.concat(...arguments)
    );
  };
};
```

58. 函数柯里化的实现

> 函数柯里化指的是一种将使用多个**参数的**一个函数转换成一系列使用一个参数的函数的技术。

```
function curry(fn, args) {
  // 获取函数需要的参数长度
  let length = fn.length;
  args = args || [];
  return function() {
    let subArgs = args.slice(0);
    // 拼接得到现有的所有参数
    for (let i = 0; i < arguments.length; i++) {
      subArgs.push(arguments[i]);
    }
    // 判断参数的长度是否已经满足函数所需参数的长度
    if (subArgs.length >= length) {
      // 如果满足，执行函数
      return fn.apply(this, subArgs);
    } else {
      // 如果不满足，递归返回科里化的函数，等待参数的传入
      return curry.call(this, fn, subArgs);
    }
  };
}

// es6 实现
function curry(fn, ...args) {
  return fn.length <= args.length ? fn(...args) : curry.bind(null, fn, ...args);
}
```

59. 什么是 XSS 攻击？如何防范 XSS 攻击？

> XSS 攻击指的是跨站脚本攻击，是一种代码注入攻击。攻击者通过在网站注入恶意脚本，使之在用户的浏览器上运行，从而盗取用户的信息如 cookie 等。
>
> XSS 的本质是因为网站没有对恶意代码进行过滤，与正常的代码混合在一起了，浏览器没有办法分辨哪些脚本是可信的，从而导致了恶意代码的执行。
>
> XSS 一般分为存储型、反射型和 DOM 型。
>
> 存储型指的是恶意代码提交到了网站的数据库中，当用户请求数据的时候，服务器将其拼接为 HTML 后返回给了用户，从而导致了恶意代码的执行。
>
> 反射型指的是攻击者构建了特殊的 URL，当服务器接收到请求后，从 URL 中获取数据，拼接到 HTML 后返回，从而导致了恶意代码的执行。
>
> DOM 型指的是攻击者构建了特殊的 URL，用户打开网站后，js 脚本从 URL 中获取数据，从而导致了恶意代码的执行。
>
> XSS 攻击的预防可以从两个方面入手，一个是恶意代码提交的时候，一个是浏览器执行恶意代码的时候。
>
> 对于第一个方面，如果我们对存入数据库的数据都进行的转义处理，但是一个数据可能在多个地方使用，有的地方可能不需要转义，由于我们没有办法判断数据最后的使用场景，所以直接在输入端进行恶意代码的处理，其实是不太可靠的。
>
> 因此我们可以从浏览器的执行来进行预防，一种是使用纯前端的方式，不用服务器端拼接后返回。另一种是对需要插入到 HTML 中的代码做好充分的转义。对于 DOM 型的攻击，主要是前端脚本的不可靠而造成的，我们对于数据获取渲染和字符串拼接的时候应该对可能出现的恶意代码情况进行判断。
>
> 还有一些方式，比如使用 CSP ，CSP 的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行，从而防止恶意代码的注入攻击。

> 还可以对一些敏感信息进行保护，比如 cookie 使用 http-only ，使得脚本无法获取。也可以使用验证码，避免脚本伪装成用户执行一些操作。

60. 什么是 CSRF 攻击？如何防范 CSRF 攻击?

> CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态，绕过后台的用户验证，冒充用户向服务器执行一些操作。
>
> CSRF 攻击的本质是利用了 cookie 会在同源请求中携带发送给服务器的特点，以此来实现用户的冒充。
>
> 一般的 CSRF 攻击类型有三种：
>
> 第一种是 GET 类型的 CSRF 攻击，比如在网站中的一个 img 标签里构建一个请求，当用户打开这个网站的时候就会自动发起提 交。
>
> 第二种是 POST 类型的 CSRF 攻击，比如说构建一个表单，然后隐藏它，当用户进入页面时，自动提交这个表单。
>
> 第三种是链接类型的 CSRF 攻击，比如说在 a 标签的 href 属性里构建一个请求，然后诱导用户去点击。
>
> CSRF 可以用下面几种方法来防护：
>
> 第一种是同源检测的方法，服务器根据 http 请求头中 origin 或者 referer 信息来判断请求是否为允许访问的站点，从而对请求进行过滤。当 origin 或者 referer 信息都不存在的时候，直接阻止。这种方式的缺点是有些情况下 referer 可以被伪造。还有就是我们这种方法同时把搜索引擎的链接也给屏蔽了，所以一般网站会允许搜索引擎的页面请求，但是相应的页面请求这种请求方式也可能被攻击者给利用。
>
> 第二种方法是使用 CSRF Token 来进行验证，服务器向用户返回一个随机数 Token ，当网站再次发起请求时，在请求参数中加入服务器端返回的 token ，然后服务器对这个 token 进行验证。这种方法解决了使用 cookie 单一验证方式时，可能会被冒用的问题，但是这种方法存在一个缺点就是，我们需要给网站中的所有请求都添加上这个 token，操作比较繁琐。还有一个问题是一般不会只有一台网站服务器，如果我们的请求经过负载平衡转移到了其他的服务器，但是这个服务器的 session 中没有保留这个 token 的话，就没有办法验证了。这种情况我们可以通过改变 token 的构建方式来解决。
>
> 第三种方式使用双重 Cookie 验证的办法，服务器在用户访问网站页面时，向请求域名注入一个 Cookie，内容为随机字符串，然后当用户再次向服务器发送请求的时候，从 cookie 中取出这个字符串，添加到 URL 参数中，然后服务器通过对 cookie 中的数据和参数中的数据进行比较，来进行验证。使用这种方式是利用了攻击者只能利用 cookie，但是不能访问获取 cookie 的特点。并且这种方法比 CSRF Token 的方法更加方便，并且不涉及到分布式访问的问题。这种方法的缺点是如果网站存在 XSS 漏洞的，那么这种方式会失效。同时这种方式不能做到子域名的隔离。
>
> 第四种方式是使用在设置 cookie 属性的时候设置 Samesite ，限制 cookie 不能作为被第三方使用，从而可以避免被攻击者利用。Samesite 一共有两种模式，一种是严格模式，在严格模式下 cookie 在任何情况下都不可能作为第三方 Cookie 使用，在宽松模式下，cookie 可以被请求是 GET 请求，且会发生页面跳转的请求所使用。

71. SQL 注入攻击?

> SQL 注入攻击指的是攻击者在 HTTP 请求中注入恶意的 SQL 代码，服务器使用参数构建数据库 SQL 命令时，恶意 SQL 被一起构造，破坏原有 SQL 结构，并在数据库中执行，达到编写程序时意料之外结果的攻击行为。

72. 什么是 MVVM？比之 MVC 有什么区别？什么又是 MVP ？

> MVC、MVP 和 MVVM 是三种常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构，优化我们的开发效率。

MVC 通过分离 Model、View 和 Controller 的方式来组织代码结构。其中 View 负责页面的显示逻辑，Model 负责存储页面的业务数据，以及对相应数据的操作。并且 View 和 Model 应用了观察者模式，当 Model 层发生改变的时候它会通知有关 View 层更新页面。Controller 层是 View 层和 Model 层的纽带，它主要负责用户与应用的响应操作，当用户与页面产生交互的时候，Controller 中的事件触发器就开始工作了，通过调用 Model 层，来完成对 Model 的修改，然后 Model 层再去通知 View 层更新。

MVP 模式与 MVC 唯一不同的在于 Presenter 和 Controller。在 MVC 模式中我们使用观察者模式，来实现当 Model 层数据发生变化的时候，通知 View 层的更新。这样 View 层和 Model 层耦合在一起，当项目逻辑变得复杂的时候，可能会造成代码的混乱，并且可能会对代码的复用性造成一些问题。MVP 的模式通过使用 Presenter 来实现对 View 层和 Model 层的解耦。MVC 中的Controller 只知道 Model 的接口，因此它没有办法控制 View 层的更新，MVP 模式中，View 层的接口暴露给了 Presenter 因此我们可以在 Presenter 中将 Model 的变化和 View 的变化绑定在一起，以此来实现 View 和 Model 的同步更新。这样就实现了对 View 和 Model 的解耦，Presenter 还包含了其他的响应逻辑。

MVVM 模式中的 VM，指的是 ViewModel，它和 MVP 的思想其实是相同的，不过它通过双向的数据绑定，将 View 和 Model 的同步更新给自动化了。当 Model 发生变化的时候，ViewModel 就会自动更新；ViewModel 变化了，View 也会更新。这样就将 Presenter 中的工作给自动化了。我了解过一点双向数据绑定的原理，比如 vue 是通过使用数据劫持和发布订阅者模式来实现的这一功能。

73. vue 双向数据绑定原理?

vue 通过使用双向数据绑定，来实现了 View 和 Model 的同步更新。vue 的双向数据绑定主要是通过使用**数据劫持和发布订阅者模式**来实现的。

首先我们通过 Object.defineProperty() 方法来对 Model 数据各个属性添加访问器属性，以此来实现数据的劫持，因此当 Model 中的数据发生变化的时候，我们可以通过配置的 setter 和 getter 方法来实现对 View 层数据更新的通知。

数据在 html 模板中一共有两种绑定情况，一种是使用 v-model 来对 value 值进行绑定，一种是作为文本绑定，在对模板引擎进行解析的过程中。

如果遇到元素节点，并且属性值包含 v-model 的话，我们就从 Model 中去获取 v-model 所对应的属性的值，并赋值给元素的 value 值。然后给这个元素设置一个监听事件，当 View 中元素的数据发生变化的时候触发该事件，通知 Model 中的对应的属性的值进行更新。

如果遇到了绑定的文本节点，我们使用 Model 中对应的属性的值来替换这个文本。对于文本节点的更新，我们使用了发布订阅者模式，属性作为一个主题，我们为这个节点设置一个订阅者对象，将这个订阅者对象加入这个属性主题的订阅者列表中。当 Model 层数据发生改变的时候，Model 作为发布者向主题发出通知，主题收到通知再向它的所有订阅者推送，订阅者收到通知后更改自己的数据。

74. Object.defineProperty 介绍?

Object.defineProperty 函数一共有三个参数，

第一个参数是需要定义属性的对象，

第二个参数是需要定义的属性，

第三个是该属性描述符。一个属性的描述符有四个属性，分别是 value 属性的值，writable 属性是否可写，enumerable 属性是否可枚举，configurable 属性是否可配置修改。

75. 什么是 Virtual DOM? 为什么 Virtual DOM 比原生 DOM 快?

我对 Virtual DOM 的理解是，

首先对我们将要插入到文档中的 DOM 树**结构进行分析**，使用 js 对象将其表示出来，比如一个**元素对象，包含 TagName、props 和 Children 这些属性**。然后我们将这个 js 对象树给保存下来，最后再将

> DOM 片段插入到文档中。
>
> 当页面的状态发生改变，我们需要对页面的 DOM 的结构进行调整的时候，我们首先根据**变更的状态**，重新构建起一棵对象树，然后将这棵**新的对象树和旧的对象树进行比较**，记录下两棵树的的差异。
>
> 最后将记录的有差异的地方应用到真正的 DOM 树中去，这样视图就更新了。
>
> 我认为 Virtual DOM 这种方法对于我们需要有大量的 DOM 操作的时候，能够很好的提高我们的操作效率， 通过在操作前确定需要做的最小修改，尽可能的减少 DOM 操作带来的重流和重绘的影响。
>
> 其实 Virtual DOM 并不一定比我们真实的操作 DOM 要快，这种方法的目的是为了提高我们开发时的可维护性，在任意的情况下，都能保证一个尽量小的性能消耗去进行操作。

76. 如何比较两个 DOM 树的差异?

> 两个树的完全 diff 算法的时间复杂度为 O(n^3) ，但是在前端中，我们很少会跨层级的移动元素，所以我们只需要比较同一层级的元素进行比较，这样就可以将算法的时间复杂度降低为 O(n)。
>
> 算法首先会对**新旧两棵树进行一个深度优先的遍历**，这样每个节点都会有一个序号。在**深度遍历**的时候，每遍历到一个节点，我们就将这个节点和新的树中的节点进行比较，如果有差异，则将**这个差异记录到一个对象中**。
>
> 在对列表元素进行对比的时候，由于 TagName 是重复的，所以我们不能使用这个来对比。我们需要给每一个子节点加上一个 key，列表对比的时候使用 key 来进行比较，这样我们才能够复用老的 DOM 树上的节点。

77. 谈谈你对 webpack 的看法

> 我当时使用 webpack 的一个最主要原因是为了简化页面依赖的管理，并且通过将其打包为一个文件来降低页面加载时请求的资源数。
>
> 我认为 webpack 的主要原理是，它**将所有的资源都看成是一个模块**，并且把页面逻辑当作一个整体，通过一个给定的入口文件，webpack 从这个文件开始，找到所有的依赖文件，将各个依赖文件模块通过 loader 和 plugins 处理后，然后打包在一起，最后输出一个浏览器可识别的 JS 文件。
>
> Webpack 具有四个核心的概念，分别是 Entry（入口）、Output（输出）、loader 和 Plugins（插件）。
>
> Entry 是 webpack 的入口起点，它指示 webpack 应该从哪个模块开始着手，来作为其构建内部依赖图的开始。
>
> Output 属性告诉 webpack 在哪里输出它所创建的打包文件，也可指定打包文件的名称，默认位置为 ./dist。
>
> loader 可以理解为 webpack 的**编译器**，它使得 webpack 可以处理一些非 JavaScript 文件。在对 loader 进行配置的时候，test 属性，标志有哪些后缀的文件应该被处理，是一个正则表达式。use 属性，指定 test 类型的文件应该使用哪个 loader 进行预处理。常用的 loader 有 css-loader、style-loader 等。
>
> 插件可以用于执行范围更广的任务，包括打包、优化、压缩、搭建服务器等等，要使用一个插件，一般是先使用 npm 包管理器进行安装，然后在配置文件中引入，最后将其实例化后传递给 plugins 数组属性。
>
> 使用 webpack 的确能够提供我们对于项目的管理，但是它的缺点就是调试和配置起来太麻烦了。

78. 谈一谈你理解的函数式编程?

> 简单说，"函数式编程"是一种"编程范式"（programming paradigm），也就是如何编写程序的方法论。
>
> 它具有以下特性：闭包和高阶函数、惰性计算、递归、函数是"第一等公民"、只用"表达式"。

79. 异步编程的实现方式?

> js 中的异步机制可以分为以下几种：

> 第一种最常见的是使用**回调函数**的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。
>
> 第二种是 Promise 的方式，使用 Promise 的方式可以将嵌套的回调函数作为**链式调用**。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。
>
> 第三种是使用 **generator** 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部我们还可以将执行权转移回来。当我们遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕的时候我们再将执行权给转移回来。因此我们在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式我们需要考虑的问题是何时将函数的控制权转移回来，因此我们需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。
>
> 第四种是使用 async 函数的形式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此我们可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

80. get 请求传参长度的误区

> 误区：我们经常说 get 请求参数的大小存在限制，而 post 请求的参数大小是无限制的。
>
> 实际上 HTTP 协议从未规定 GET/POST 的请求长度限制是多少。对 get 请求参数的限制是来源与浏览器或web 服务器，浏览器或 web 服务器限制了 url 的长度。为了明确这个概念，我们必须再次强调下面几点:

- HTTP 协议未规定 GET 和 POST 的长度限制
- GET 的最大长度显示是因为浏览器和 web 服务器限制了 URI 的长度
- 不同的浏览器和 WEB 服务器，限制的最大长度不一样
- 要支持 IE，则最大长度为 2083byte，若只支持 Chrome，则最大长度 8182byte

81. URL 和 URI 的区别?

- URI: Uniform Resource Identifier 指的是统一资源标识符
- URL: Uniform Resource Location 指的是统一资源定位符
- URN: Universal Resource Name 指的是统一资源名称

> URI 指的是**统一资源标识符**，用唯一的标识来确定一个资源，它是一种抽象的定义，也就是说，不管使用什么方法来定义，只要能唯一的**标识一个资源**，就可以称为 URI。
>
> URL 指的是统一资源**定位符**，URN 指的是统一资源名称。URL 和 URN 是 URI 的子集，URL 可以理解为**使用地址来标识资源**，URN 可以理解为使用名称来标识资源。

82. get 和 post 请求在缓存方面的区别

> get 请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以可以使用缓存。
>
> post 不同，post 做的一般是修改和删除的工作，所以必须与数据库交互，所以不能使用缓存。因此 get 请求适合于请求缓存。

83. 图片的懒加载和预加载

> 预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。
>
> 懒加载：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。

> 两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。 懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

84. 为什么使用 setTimeout 实现 setInterval？怎么模拟？

> setInterval 的作用是每隔一段指定时间执行一个函数，但是这个执行不是真的到了时间立即执行，它真正的作用是每隔一段时间将**事件加入事件队列中**去，只有当当前的执行栈为空的时候，才能去从事件队列中取出事件执行。所以可能会出现这样的情况，就是当前执行栈执行的时间很长，导致事件队列里边积累多个定时器加入的事件，当执行栈结束的时候，这些事件会依次执行，因此就不能到间隔一段时间执行的效果。
>
> 针对 setInterval 的这个缺点，我们可以使用 setTimeout 递归调用来模拟 setInterval，这样我们就确保了只有一个事件结束了，我们才会触发下一个定时器事件，这样解决了 setInterval 的问题。

```
// 思路是使用递归函数，不断地去执行 setTimeout 从而达到 setInterval 的效果

function mySetInterval(fn, timeout) {
  // 控制器，控制定时器是否继续执行
  var timer = {
    flag: true
  };

  // 设置递归函数，模拟定时器执行。
  function interval() {
    if (timer.flag) {
      fn();
      setTimeout(interval, timeout);
    }
  }

  // 启动定时器
  setTimeout(interval, timeout);

  // 返回控制器
  return timer;
}
```

85. let 和 const 的注意点？

- 声明的变量只在声明时的代码块内有效
- 不存在声明提升
- 存在暂时性死区，如果在变量声明前使用，会报错
- 不允许重复声明，重复声明会报错

86. 什么是 rest 参数？

> rest 参数（形式为...变量名），用于获取函数的**多余参数**。

87. Set 和 WeakSet 结构？

- ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。
- WeakSet 结构与 Set 类似，也是不重复的值的集合。但是 WeakSet 的成员只能是对象，而不能是其他类型的值。WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用

88. 什么是 Proxy ？

> Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”，即对编程语言进行编程。
>
> Proxy 可以理解成，在**目标对象之前架设一层“拦截”**，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

89. 什么是 Promise 对象，什么是 Promises/A+ 规范?

> Promise 对象是异步编程的一种解决方案，最早由社区提出。Promises/A+ 规范是 JavaScript Promise 的标准，规定了一个 Promise 所必须具有的特性。
>
> Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。
>
> 一个 Promise 实例有三种状态，分别是 pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由 pending 转变 resolved 或者 rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。状态的改变是通过 resolve() 和 reject() 函数来实现的，
>
> 我们可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 then 方法，使用这个 then 方法可以为两个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

90. 手写一个 Promise

```
const PENDING = "pending";
const FULFILLED = "fulfilled";
const REJECTED = "rejected";

class NPromise {
  FULFILLED_CALLBACK_LIST = [];
  REJECTED_CALLBACK_LIST = [];
  _status = PENDING;

  constructor(fn) {
    this.status = PENDING;
    this.value = null;
    this.reason = null;

    try {
      // fn: (resolve, reject) => {}
      fn(this.resolve.bind(this), this.reject.bind(this));
    } catch (e) {
      this.reject(e);
    }
  }
  // getter 和 setter 来属性的监听和代理
  get status() {
    return this._status;
  }

  set status(newStatus) {
    this._status = newStatus;
    switch (newStatus) {
      case FULFILLED: {
        this.FULFILLED_CALLBACK_LIST.forEach((callback) => {
          callback(this.value);
        });
        break;
      }
      case REJECTED: {
        this.REJECTED_CALLBACK_LIST.forEach((callback) => {
          callback(this.reason);
        });
        break;
      }
    }
  }

  resolve(value) {
    if (this.status === PENDING) {
      this.value = value;
      this.status = FULFILLED;
    }
  }

  reject(reason) {
    if (this.status === PENDING) {
      this.reason = reason;
```

```
    this.status = REJECTED;
  }
}

then(onFulfilled, onRejected) {
  const realOnFulfilled = this.isFunction(onFulfilled)
    ? onFulfilled
    : (value) => {
        return value;
      };
  const realOnRejected = this.isFunction(onRejected)
    ? onRejected
    : (reason) => {
        throw reason;
      };

  const promise2 = new NPromise((resolve, reject) => {
    const fulfilledMicroTask = () => {
      queueMicrotask(() => {
        try {
          const x = realOnFulfilled(this.value);
          this.resolvePromise(promise2, x, resolve, reject);
        } catch (error) {
          reject(error);
        }
      });
    };

    const rejectedMicroTask = () => {
      queueMicrotask(() => {
        try {
          const x = realOnRejected(this.reason);
          this.resolvePromise(promise2, x, resolve, reject);
        } catch (error) {
          reject(error);
        }
      });
    };

    switch (this.status) {
      case FULFILLED: {
        fulfilledMicroTask();
        break;
      }
      case REJECTED: {
        rejectedMicroTask();
        break;
      }
      case PENDING: {
        this.FULFILLED_CALLBACK_LIST.push(realOnFulfilled);
        this.REJECTED_CALLBACK_LIST.push(realOnRejected);
      }
    }
  });
  return promise2;
}

isFunction(params) {
  return typeof params === "function";
}

resolvePromise(promise2, x, resolve, reject) {
  if (promise2 === x) {
    return reject(new TypeError("the promise & return value are the same"));
  }
  if (x instanceof NPromise) {
    queueMicrotask(() => {
      x.then((y) => {
        this.resolvePromise(promise2, y, resolve, reject);
      }, reject);
    });
  } else if (typeof x === "object" || this.isFunction(x)) {
    if (x === null) {
      return resolve(x);
    }
    let then = null;
    try {
      then = x.then;
```

```
      } catch (error) {
        return reject(error);
      }

      if (this.isFunction(then)) {
        let called = false;
        try {
          then.call(
            x,
            (y) => {
              if (called) {
                return;
              }
              called = true;
              this.resolvePromise(promise2, y, resolve, reject);
            },
            (r) => {
              if (called) {
                return;
              }
              called = true;
              reject(r);
            }
          );
        } catch (error) {
          if (called) {
            return;
          }
          reject(error);
        }
      } else {
        resolve(x);
      }
    } else {
      resolve(x);
    }
  }

  catch(onRejected) {
    this.then(null, onRejected);
  }

  static resolve(value) {
    if (value instanceof NPromise) {
      return value;
    }
    return new NPromise((resolve) => {
      resolve(value);
    });
  }

  static reject(reason) {
    return new NPromise((resolve, reject) => {
      reject(reason);
    });
  }
}

const test = new NPromise((resolve, rejest) => {
    // setTimeout(() => {
    //     resolve(111)
    // }, 100)
    resolve(111)
}).then((value) => {
    console.log(value)
})

console.log(test);
setTimeout(() => {
  console.log(test);
}, 8000);
```

91. 单例模式模式是什么?

> 单例模式保证了全局只有**一个实例来被访问**。比如说常用的如弹框组件的实现和全局状态的实现。

92. Vue 的各个生命阶段是什么?

Vue 一共有8个生命阶段，分别是创建前、创建后、加载前、加载后、更新前、更新后、销毁前和销毁后，每个阶段对应了一个生命周期的钩子函数。

（1）beforeCreate 钩子函数，在实例初始化之后，在数据监听和事件配置之前触发。因此在这个事件中我们是获取不到 data 数据的。

（2）created 钩子函数，在实例创建完成后触发，此时可以访问 data、methods 等属性。但这个时候组件还没有被挂载到页面中去，所以这个时候访问不到 $el 属性。一般我们可以在这个函数中进行一些页面初始化的工作，比如通过 ajax 请求数据来对页面进行初始化。

（3）beforeMount 钩子函数，在组件被挂载到页面之前触发。在 beforeMount 之前，会找到对应的 template，并编译成 render 函数。

（4）mounted 钩子函数，在组件挂载到页面之后触发。此时可以通过 DOM API 获取到页面中的 DOM 元素。

（5）beforeUpdate 钩子函数，在响应式数据更新时触发，发生在虚拟 DOM 重新渲染和打补丁之前，这个时候我们可以对可能会被移除的元素做一些操作，比如移除事件监听器。

（6）updated 钩子函数，虚拟 DOM 重新渲染和打补丁之后调用。

（7）beforeDestroy 钩子函数，在实例销毁之前调用。一般在这一步我们可以销毁定时器、解绑全局事件等。

（8）destroyed 钩子函数，在实例销毁之后调用，调用后，Vue 实例中的所有东西都会解除绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

当我们使用 keep-alive 的时候，还有两个钩子函数，分别是 activated 和 deactivated 。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 actived 钩子函数。

93. Vue 组件间的参数传递方式?

（1）父子组件间通信

第一种方法是子组件通过 props 属性来接受父组件的数据，然后父组件在子组件上注册监听事件，子组件通过 emit 触发事件来向父组件发送数据。

第二种是通过 ref 属性给子组件设置一个名字。父组件通过 $refs 组件名来获得子组件，子组件通过 $parent 获得父组件，这样也可以实现通信。

第三种是使用 provider/inject，在父组件中通过 provider 提供变量，在子组件中通过 inject 来将变量注入到组件中。不论子组件有多深，只要调用了 inject 那么就可以注入 provider 中的数据。

（2）兄弟组件间通信

第一种是使用 eventBus 的方法，它的本质是通过创建一个空的 Vue 实例来作为消息传递的对象，通信的组件引入这个实例，通信的组件通过在这个实例上监听和触发事件，来实现消息的传递。

第二种是通过 $parent.$refs 来获取到兄弟组件，也可以进行通信。

（3）任意组件之间

使用 eventBus ，其实就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。

如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候采用上面这一些方法可能不利于项目的维护。这个时候可以使用 vuex ，vuex 的思想就是将这一些公共的数据抽离出来，将它作为一个全局的变量来管理，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

94. *route*和router 的区别?

> *route*是“路由信息对象”，包括*path*，*params*，*hash*，*query*，*fullPath*，*matched*，*name*等路由信息参数
> router 是“路由实例”对象包括了路由的跳转方法，钩子函数等。

95. 开发中常用的几种 Content-Type ?

> （1）application/x-www-form-urlencoded

> 浏览器的原生 form 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。该种方式提交的数据放在 body 里面，数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL转码。
>
> （2）multipart/form-data
>
> 该种方式也是一个常见的 POST 提交方式，通常表单上传文件时使用该种方式。
>
> （3）application/json
>
> 告诉服务器消息主体是序列化后的 JSON 字符串。
>
> （4）text/xml
>
> 该种方式主要用来提交 XML 格式的数据。

96. 使用闭包实现每隔一秒打印 1,2,3,4

```
for (let i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i);
  }, i * 1000);
}
```

97.手写一个 jsonp

```
function jsonp(url, params, callback) {
  // 判断是否含有参数
  let queryString = url.indexOf("?") === "-1" ? "?" : "&";
  // 添加参数
  for (var k in params) {
    if (params.hasOwnProperty(k)) {
      queryString += k + "=" + params[k] + "&";
    }
  }
  // 处理回调函数名
  let random = Math.random()
      .toString()
      .replace(".", ""),
    callbackName = "myJsonp" + random;
  // 添加回调函数
  queryString += "callback=" + callbackName;
  // 构建请求
  let scriptNode = document.createElement("script");
  scriptNode.src = url + queryString;
  window[callbackName] = function() {
    // 调用回调函数
    callback(...arguments);
    // 删除这个引入的脚本
    document.getElementsByTagName("head")[0].removeChild(scriptNode);
  };
  // 发起请求
  document.getElementsByTagName("head")[0].appendChild(scriptNode);
}
```

98. EventEmitter 实现

```
class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(event, callback) {
    let callbacks = this.events[event] || [];
```

```
    callbacks.push(callback);
    this.events[event] = callbacks;
    return this;
  }
  off(event, callback) {
    let callbacks = this.events[event];
    this.events[event] = callbacks && callbacks.filter(fn => fn !== callback);

    return this;
  }
  emit(event, ...args) {
    let callbacks = this.events[event];
    callbacks.forEach(fn => {
      fn(...args);
    });

    return this;
  }
  once(event, callback) {
    let wrapFun = function(...args) {
      callback(...args);

      this.off(event, wrapFun);
    };
    this.on(event, wrapFun);

    return this;
  }
}
```

99. Object.assign()

> Object.assign() 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

CSS

1. 盒模型

> 页面渲染时，dom 元素所采用的 布局模型。可通过box-sizing进行设置。根据计算宽高的区域可分为：

- content-box (W3C 标准盒模型)
- border-box (IE 盒模型)
- padding-box (FireFox 曾经支持)
- margin-box (浏览器未实现)

2. BFC

> 块级格式化上下文，是一个独立的渲染区域，让处于 BFC 内部的元素与外部的元素相互隔离，使内外元素的定位不会相互影响。

> 触发条件:

- 根元素
- position: absolute/fixed
- display: inline-block / table
- float 元素
- ovevflow !== visible

> 规则:

- 属于同一个 BFC 的两个相邻 Box 垂直排列
- 属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠

- BFC 中子元素的 margin box 的左边， 与包含块 (BFC) border box的左边相触 (子元素 absolute 除外)
- BFC 的区域不会与 float 的元素区域重叠
- 计算 BFC 的高度时，浮动子元素也参与计算
- 文字层不会被浮动层覆盖，环绕于周围

3. 居中布局

> 水平居中

- 行内元素: text-align: center
- 块级元素: margin: 0 auto
- absolute + transform
- flex + justify-content: center

> 垂直居中

- line-height: height
- absolute + transform
- flex + align-items: center
- table

> 水平垂直居中

- absolute + transform
- flex + justify-content + align-items

4. 选择器优先级

> !important > 行内样式 > #id > .class > tag > * > 继承 > 默认
>
> 选择器 从右往左 解析

5. 去除浮动影响，防止父级高度塌陷

- 通过增加尾元素清除浮动
    - :after /
      : clear: both
- 创建父级 BFC
- 父级设置高度

6. link 与 @import 的区别

- link功能较多，可以定义 RSS，定义 Rel 等作用，而@import只能用于加载 css
- 当解析到link时，页面会同步加载所引的 css，而@import所引用的 css 会等到页面加载完才被加载
- @import需要 IE5 以上才能使用
- link可以使用 js 动态引入，@import不行

6. CSS预处理器(Sass/Less/Postcss)

> CSS预处理器的原理: 是将类 CSS 语言通过 Webpack 编译 转成浏览器可读的真正 CSS。在这层编译之上，便可以赋予 CSS 更多更强大的功能，常用功能:

- 嵌套
- 变量
- 循环语句
- 条件语句

- 自动前缀
- 单位转换
- mixin复用

7. CSS动画

> transition: 过渡动画

- transition-property: 属性
- transition-duration: 间隔
- transition-timing-function: 曲线
- transition-delay: 延迟 常用钩子: transitionend

> animation / keyframes

- animation-name: 动画名称，对应@keyframes
- animation-duration: 间隔
- animation-timing-function: 曲线
- animation-delay: 延迟
- animation-iteration-count: 次数

```
infinite: 循环动画
animation-direction: 方向
alternate: 反向播放
animation-fill-mode: 静止模式
forwards: 停止时，保留最后一帧
backwards: 停止时，回到第一帧
both: 同时运用 forwards / backwards
```

常用钩子: animationend

> 动画属性: 尽量使用动画属性进行动画，能拥有较好的性能表现

- translate
- scale
- rotate
- skew
- opacity
- color

其他常见

1. CP三次握手

> 建立连接前，客户端和服务端需要通过握手来确认对方:
>
> 客户端发送 syn(同步序列编号) 请求，进入 syn_send 状态，等待确认
>
> 服务端接收并确认 syn 包后发送 syn+ack 包，进入 syn_recv 状态
>
> 客户端接收 syn+ack 包后，发送 ack 包，双方进入 established 状态

2. TCP四次挥手

> 客户端 -- FIN --> 服务端， FIN—WAIT
>
> 服务端 -- ACK --> 客户端， CLOSE-WAIT
>
> 服务端 -- ACK,FIN --> 客户端， LAST-ACK
>
> 客户端 -- ACK --> 服务端，CLOSED

React

1. Fiber

> React 的核心流程可以分为两个部分:
>
> reconciliation (调度算法，也可称为 render):

- 更新 state 与 props；
- 调用生命周期钩子；
- 生成 virtual dom: 这里应该称为 Fiber Tree 更为符合；
- 通过新旧 vdom 进行 diff 算法，获取 vdom change；
- 确定是否需要重新渲染

> commit:

- 如需要，则操作 dom 节点更新；

要了解 Fiber，我们首先来看为什么需要它?

- **问题**: 随着应用变得越来越庞大，整个更新渲染的过程开始变得吃力，大量的组件渲染会导致主进程长时间被占用，导致一些**动画或高频操作出现卡顿和掉帧的情况**。而关键点，便是 **同步阻塞**。在之前的调度算法中，React 需要实例化每个类组件，生成一颗组件树，使用 **同步递归** 的方式进行遍历渲染，而这个过程最大的问题就是**无法暂停和恢复**。
- **解决方案**: 解决同步阻塞的方法，通常有两种: 异步 与 **任务分割**。而 React Fiber 便是为了实现任务分割而诞生的.

> 主要是 将原先同步更新渲染的任务分割成一个个独立的 小任务单位，根据不同的优先级，将小任务分散到浏览器的空闲时间执行，充分利用主进程的事件循环机制。

> 链表树遍历算法: 通过 节点保存与映射，便能够随时地进行 停止和重启，这样便能达到实现任务分割的基本前提；

1、首先通过不断遍历子节点，到树末尾； 2、开始通过 sibling 遍历兄弟节点； 3、return 返回父节点，继续执行2； 4、直到 root 节点后，跳出遍历；

> 任务分割，React 中的渲染更新可以分成两个阶段:

reconciliation 阶段: vdom 的数据对比，是个适合拆分的阶段，比如对比一部分树后，先暂停执行个动画调用，待完成后再回来继续比对。 Commit 阶段: 将 change list 更新到 dom 上，并不适合拆分，才能保持数据与 UI 的同步。否则可能由于阻塞 UI 更新，而导致数据更新和 UI 不一致的情况。

> 分散执行: 任务分割后，就可以把小任务单元分散到浏览器的空闲期间去排队执行，而实现的关键是两个新API: requestIdleCallback 与 requestAnimationFrame

低优先级的任务交给**requestIdleCallback**处理，这是个浏览器提供的事件循环空闲期的回调函数，需要pollyfill，而且拥有 deadline 参数，限制执行事件，以继续切分任务； 高优先级的任务交给requestAnimationFrame处理；

> 优先级策略: 文本框输入 > 本次调度结束需完成的任务 > 动画过渡 > 交互反馈 > 数据更新 > 不会显示但以防将来会显示的任务

2. 生命周期

> 在新版本中，React 官方对生命周期有了新的 变动建议:

- 使用getDerivedStateFromProps 替换 componentWillMount 与 componentWillReceiveProps；
- 使用getSnapshotBeforeUpdate替换componentWillUpdate；
- 避免使用componentWillReceiveProps；

其实该变动的原因，正是由于上述提到的 Fiber。首先，从上面我们知道 React 可以分成 reconciliation 与 commit 两个阶段，对应的生命周期如下:

> reconciliation:

- componentWillMount
- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate

> commit:

- componentDidMount
- componentDidUpdate
- componentWillUnmount

> 在 Fiber 中，reconciliation 阶段进行了任务分割，涉及到 暂停 和 重启，因此可能会导致 reconciliation 中的生命周期函数在一次更新渲染循环中被 多次调用 的情况，产生一些意外错误

```
class Component extends React.Component {
  // 替换 `componentWillReceiveProps` ,
  // 初始化和 update 时被调用
  // 静态函数，无法使用 this
  static getDerivedStateFromProps(nextProps, prevState) {}

  // 判断是否需要更新组件
  // 可以用于组件性能优化
  shouldComponentUpdate(nextProps, nextState) {}

  // 组件被挂载后触发
  componentDidMount() {}

  // 替换 componentWillUpdate
  // 可以在更新之前获取最新 dom 数据
  getSnapshotBeforeUpdate() {}

  // 组件更新后调用
  componentDidUpdate() {}

  // 组件即将销毁
  componentWillUnmount() {}

  // 组件已销毁
  componentDidUnmount() {}
}
```

> 使用建议:

- 在constructor初始化 state；
- 在componentDidMount中进行事件监听，并在componentWillUnmount中解绑事件；
- 在componentDidMount中进行数据的请求，而不是在componentWillMount； 需要根据 props 更新 state 时，使用getDerivedStateFromProps(nextProps, prevState)；

3. setState

> setState并不是单纯的异步或同步，这其实与调用时的环境相关:
>
> 在 合成事件 和 生命周期钩子(除 componentDidUpdate) 中，setState是"异步"的；
>
> 原因: 因为在setState的实现中，有一个判断: 当更新策略正在事务流的执行中时，该组件更新会被推入dirtyComponents队列中等待执行；否则，开始执行batchedUpdates队列更新；
>
> 在生命周期钩子调用中，更新策略都处于更新之前，组件仍处于事务流中，而componentDidUpdate是在更新之后，此时组件已经不在事务流中了，因此则会同步执行；
>
> 在合成事件中，React 是基于 事务流完成的事件委托机制 实现，也是处于事务流中；

> 问题: 无法在setState后马上从this.state上获取更新后的值。

> 解决: 如果需要马上同步去获取新值，setState其实是可以传入第二个参数的。setState(updater, callback)，在回调中即可获取最新值；

> 在 原生事件 和 setTimeout 中，setState是同步的，可以马上获取更新后的值；
>
> 原因: 原生事件是浏览器本身的实现，与事务流无关，自然是同步；而setTimeout是放置于定时器线程中延后执行，此时事务流已结束，因此也是同步；

4. HOC(高阶组件)

> HOC(Higher Order Componennt) 是在 React 机制下社区形成的一种组件模式，在很多第三方开源库中表现强大。

- 高阶组件不是组件，是 增强函数，可以输入一个元组件，返回出一个新的增强组件；
- 高阶组件的主要作用是 代码复用，操作 状态和参数；

> 用法

- 属性代理 (Props Proxy): 返回出一个组件，它基于被包裹组件进行 功能增强；

```
function proxyHoc(Comp) {
        return class extends React.Component {
                render() {
                        const newProps = {
                                name: 'tayde',
                                age: 1,
                        }
                        return <Comp {...this.props} {...newProps} />
                }
        }
}
```

- 包裹组件: 可以为被包裹元素进行一层包装，

```
function withMask(Comp) {
  return class extends React.Component {
      render() {
                  return (
                      <div>
                                  <Comp {...this.props} />
                                        <div style={{
                                          width: '100%',
                                          height: '100%',
                                          backgroundColor: 'rgba(0, 0, 0, .6)',
                                  }}
                          </div>
```

```
                )
            }
        }
    }
```

5. React Hooks

> React 中通常使用 类定义 或者 函数定义 创建组件:
>
> 在类定义中，我们可以使用到许多 React 特性，例如 state、 各种组件生命周期钩子等，但是在函数定义中，我们却无能为力，因此 React 16.8 版本推出了一个新功能 (React Hooks)，通过它，可以更好的在函数定义组件中使用 React 特性

> 好处:

- 1、跨组件复用: 其实 render props / HOC 也是为了复用，相比于它们，Hooks 作为官方的底层 API，最为轻量，而且改造成本小，不会影响原来的组件层次结构和传说中的嵌套地狱；
- 2、类定义更为复杂:
    - 不同的生命周期会使逻辑变得分散且混乱，不易维护和管理；
    - 时刻需要关注this的指向问题；
    - 代码复用代价高，高阶组件的使用经常会使整个组件树变得臃肿；
- 3、状态与UI隔离: 正是由于 Hooks 的特性，状态逻辑会变成更小的粒度，并且极容易被抽象成一个自定义 Hooks，组件中的状态和 UI 变得更为清晰和隔离

> 注意:

- 避免在 循环/条件判断/嵌套函数 中调用 hooks，保证调用顺序的稳定；
- 只有 函数定义组件 和 hooks 可以调用 hooks，避免在 类组件 或者 普通函数 调用；
- 不能在useEffect中使用useState，React 会报错提示；
- 类组件不会被替换或废弃，不需要强制改造类组件，两种方式能并存