

准备：简历编写和面试前准备

一般来说，跳槽找工作要经历投递简历、准备面试、面试和谈 offer 四个阶段。其中面试题目会因你的等级和职位而异，从入门级到专家级，广度和深度都会有所增加。不过，不管什么级别和职位，面试题目一般都可以分类为理论知识、算法、项目细节、技术视野、开放性题、工作案例等内容。接下来重点来说下简历和知识点梳理的方法。

准备一份合适的简历

首先，什么样子的简历更加容易拿到面试邀请？笔者作为一名在 BAT 中待过两家的面试官，见过各种各样的简历，先说一下一些比较不受欢迎的简历：

1. **招聘网站上的简历**：有些简历是 HR 直接从某招聘网站直接下载下来的，格式统一，而且对于自己的技能还有自己打分，这类简历有可能是候选人根本就没自己精心准备简历，而是网站根据他填写的内容自动生成的，遇到这样的简历笔者一定会让 HR 或者候选人更新一份简历给我。
2. **太花俏的简历**：有人觉得简历花俏一点会让人眼前一亮，但是公司招聘的是前端不是视觉设计，所以如果找的不是视觉设计工作，还是工工整整的简历会比较受欢迎，而且太花俏的简历有可能让人第一感觉是华而不实，并不是关注候选人的技能。
3. **造假或者描述太出格的简历**：看到你简历的人可能是不懂技术的 HR，也可能是专业领域的大牛，如果数据造假或者夸大其实，这样很容易就让人给卡掉。

那么，怎样的简历才是好的简历呢？

技术型简历的重要组成部分

一份合适的技术型简历最重要的三部分是：

1. 个人掌握的技能，是否有岗位需要用到的技能，及其技能掌握的熟练程度：熟悉、了解还是精通
2. 项目经历，项目经历是否对现在岗位有用或者有重叠，是否能够驾驭大型项目
3. 实习经历，对于没有经验的应届生来说，实习经历是很重要的部分，是否有大公司或者具体项目的实习经历是筛选简历的重要参考

技术型简历一般不要太花俏，关键要语言表达通顺清楚，让语言准确和容易理解，在 HR 筛选简历的时候，可以瞬间抓住他的眼球。另外如果有一些特殊奖项，也可以在简历中突出出来，比如：季度之星、最佳个人之类的奖项，应届生会有优秀毕业生、全额奖学金等。

推荐使用 PDF 版本的简历

一般来说简历会有 Word、Markdown 等版本，这里笔者推荐使用 PDF 版本的简历，主要原因如下：

1. 内容丰富，布局调整方便
2. 字体等格式有保障，你不知道收到你简历的人用的是什么环境，PDF 版本不会因为不同操作系统等原因而受限
3. 便于携带和传播，始终存一份简历在手机或者邮箱内，随时发送
4. 不容易被涂改

一般 Windows 系统的 Word、Mac 系统的 Pages 都支持导出 PDF 格式的文件，原稿可以保存到云端或者 iCloud，方便以后修改。

虽然我们是 Web 前端工程师，笔者还是不推荐使用 HTML 格式的简历，HTML 版本的简历容易受浏览器等环境因素影响，而且接收方不一定是技术人员，你做的炫酷的效果也不一定会被看到。

简历最好要有针对性地来写

简历是「敲门砖」，笔者建议根据你想要找的公司、岗位和职位描述来有针对性地写简历。尤其是个人技能和项目（实习）经验部分，要根据岗位要求来写，这样才能增加受邀面试的机会。

举个例子：好友给你推荐了百度地图部门的一个高级 Web 前端工程师工作，并且把职位描述（JD）发给你了，里面有要求哪些技能，用到哪些技术，还有加分项。那么你写简历就应该思考自己有没有这些技能。如果没有 JD，那么至少你应该知道：地图部门肯定做一些跟地图相关的工作，如果恰巧你之前研究过地图定位，了解 HTML5 Geolocation 定位接口，那么你可以在简历里提一下。

很多时候我们并不知道简历会被谁看到，也不知道简历会被朋友/猎头投递到什么公司或者职位，那么这样的简历应该是一种「通用简历」。所谓通用简历，应该是与你找的职位和期望的级别相匹配的简历，比如想找大概 T4 水平的 Web 前端工作，那么你就应该在简历体现出来自己的技能能够达到 T4 的水平。不要拿着一两年前的简历去找工作，前端这两年发展速度很快，只靠一两年前简历上面「精通、熟悉」的库和框架，可能已经找不到工作了。

所以，写简历也是个技术活，而且是一个辛苦活！不要用千篇一律的模板！

简历是面试时「点菜」用的菜单

简历除了是「敲门砖」之外，还是供面试官提问用的「菜单」。面试官会从你简历上面写的技能、项目进行提问。所以简历是候选人「反客为主」的重要工具，这也是笔者一直提到的：**不要造假或者描述太出格**，而应该实事求是地写简历。简历中的技能和项目都要做好知识点梳理，尽量多地梳理出面试官可能问到的问题，并且想出怎么回答应对，**千万不要在简历上自己给自己挖坑**。

案例：记得有一个候选人，写的工作时间段有问题，简历上写在 2015 年 3 月到 2017 年 4 月在 A 公司工作，但是面试自我介绍的时候说自己在 A 公司工作了一年，这就有可能让面试官认为个人工作经历存在造假可能。不要小看细节！

另外简历中不要出现错误的单词拼写，注意单词的大小写，比如 `jQuery` 之类。

拿到面试邀请之后做的准备工作

当有公司邀请你去面试的时候，应该针对性地做一些功课。

了解部门和团队

了解部门做的事情，团队用的技术栈，前文提到这部分信息一般从 JD 当中就可以看到，如果 JD 并没有这些信息，那么可以根据面试的部门搜索下，总会找到一些零星的信息，如果实在没有任何信息，就准备岗位需要的通用技术。

了解面试官

通过邀请电话或者面试邀请邮件，可以找到面试官信息。通过这些信息查找面试官技术博客、GitHub 等，了解面试官最近关注的技术和擅长的技术，因为面试官往往会在面试的过程中问自己擅长的技术。

面试中出现的常规问题

对于面试中出现的常规问题要做好准备，比如：介绍下自己，为什么跳槽，面试最后一般会问有什么要问的。

介绍自己

介绍自己时，切忌从自己大学实习一直到最新公司全部毫无侧重地介绍，这些在简历当中都有，最好的方式是在介绍中铺垫自己的技术特长、做的项目，引导面试官问自己准备好的问题。

为什么跳槽

这个问题一定要慎重和认真思考，诚实回答。一般这个问题是想评估你入职能够待多长时间，是否能够融入团队。

每个人跳槽前肯定想了很多原因，最终才走出这一步，不管现在工作怎样，**切忌抱怨，不要吐槽，更不要说和现在领导不和睦之类的话**。多从自身发展找原因，可以表达寻找自己心目中的好的技术团队氛围和平台机会，比如：个人遇见了天花板，希望找个更好的发展机会。

利用脑图来梳理知识点

对于统一校招类的面试，要重点梳理前端的所有知识点，校招面试一般是为了做人才储备，所以看的是候选人的可塑性和学习能力；对于社招类面试，则看重的是业务能力和 JD 匹配程度，所以要针对性地整理前端知识点，针对性的内容包括：项目用到的技术细节、个人技能部分需要加强或提升的常考知识点。

所以，不仅仅简历要针对性地来写，知识点也要根据自己的经历、准备的简历、公司和职位描述来针对性地梳理。每个读者的技术能力和工作经历不同，因而知识点梳理大纲也不同，本小册重点介绍如何梳理自己的面试知识点，并且对一些常考的题目进行解答，起到知识点巩固和讲解的作用。

基础知识来自于自己平时的储备，一般对着一本系统的书籍或者自己平时的笔记过一遍即可，但是提到自己做到的项目是没有固定的复习套路的，而且围绕项目可以衍生出来各种问题，都需要了解，项目讲清楚对于候选人也特别重要。基础是固定的，任何人经过一段时间都可以学完的，但是项目经历是实打实的经验。

对于项目的复习和准备，笔者建议是**列思维导图（脑图）**，针对自己重点需要讲的项目，列出用到的技术点（知识点），介绍背景、项目上线后的收益以及后续优化点。这是第一层，第二层就是针对技术点（知识点）做各种发散的问题。

小结

本小节希望你得到下面的知识：

1. 找工作之前应该准备一份合适的工作简历
2. 工作简历可以针对性地来写
3. 收到面试邀请之后应该去了解 JD 和涉及公司部门的基本情况
4. 利用思维导图来梳理知识点

一面 1：ES 基础知识点与高频考题解析

JavaScript 是 ECMAScript 规范的一种实现，本小节重点梳理下 ECMAScript 中的常考知识点，然后就一些容易出现的题目进行解析。

知识点梳理

- 变量类型
 - JS 的数据类型分类和判断
 - 值类型和引用类型
- 原型与原型链（继承）
 - 原型和原型链定义
 - 继承写法
- 作用域和闭包
 - 执行上下文

- this
 - 闭包是什么
- 异步
 - 同步 vs 异步
 - 异步和单线程
 - 前端异步的场景
- ES6/7 新标准的考查
 - 箭头函数
 - Module
 - Class
 - Set 和 Map
 - Promise
- ◦ *

变量类型

JavaScript 是一种弱类型脚本语言，所谓弱类型指的是定义变量时，不需要什么类型，在程序运行过程中会自动判断类型。

ECMAScript 中定义了 6 种原始类型：

- Boolean
- String
- Number
- Null
- Undefined
- Symbol (ES6 新定义)

注意：原始类型不包含 Object。

题目：类型判断用到哪些方法？

typeof

`typeof xxx` 得到的值有以下几种类型：`undefined` `boolean` `number` `string` `object` `function`、`symbol`，比较简单，不再一一演示了。这里需要注意的有三点：

- `typeof null` 结果是 `object`，实际这是 `typeof` 的一个 bug，`null` 是原始值，非引用类型
- `typeof [1, 2]` 结果是 `object`，结果中没有 `array` 这一项，引用类型除了 `function` 其他的全部都是 `object`
- `typeof Symbol()` 用 `typeof` 获取 `symbol` 类型的值得到的是 `symbol`，这是 ES6 新增的知识点

instanceof

用于实例和构造函数的对应。例如判断一个变量是否是数组，使用 `typeof` 无法判断，但可以使用 `[1, 2] instanceof Array` 来判断。因为，`[1, 2]` 是数组，它的构造函数就是 `Array`。同理：

```
function Foo(name) {
  this.name = name
}
var foo = new Foo('bar')
console.log(foo instanceof Foo) // true
```

值类型 vs 引用类型

除了原始类型，ES 还有引用类型，上文提到的 `typeof` 识别出来的类型中，只有 `object` 和 `function` 是引用类型，其他都是值类型。

根据 JavaScript 中的变量类型传递方式，又分为**值类型**和**引用类型**，值类型变量包括 Boolean、String、Number、Undefined、Null，引用类型包括了 Object 类的所有，如 Date、Array、Function 等。在参数传递方式上，值类型是按值传递，引用类型是按共享传递。

下面通过一个小题目，来看下两者的主要区别，以及实际开发中需要注意的地方。

```
// 值类型
var a = 10
var b = a
b = 20
console.log(a) // 10
console.log(b) // 20
```

上述代码中，`a` `b` 都是值类型，两者分别修改赋值，相互之间没有任何影响。再看引用类型的例子：

```
// 引用类型
var a = {x: 10, y: 20}
var b = a
b.x = 100
b.y = 200
console.log(a) // {x: 100, y: 200}
console.log(b) // {x: 100, y: 200}
```

上述代码中，`a` `b` 都是引用类型。在执行了 `b = a` 之后，修改 `b` 的属性值，`a` 的也跟着变化。因为 `a` 和 `b` 都是引用类型，指向了同一个内存地址，即两者引用的是同一个值，因此 `b` 修改属性时，`a` 的值随之改动。

再借助题目进一步讲解一下。

说出下面代码的执行结果，并分析其原因。

```
function foo(a){
  a = a * 10;
}
function bar(b){
  b.value = 'new';
}
var a = 1;
var b = {value: 'old'};
foo(a);
bar(b);
console.log(a); // 1
console.log(b); // value: new
```

通过代码执行，会发现：

- a 的值没有发生改变
- 而 b 的值发生了改变

这就是因为 `Number` 类型的 a 是按值传递的，而 `Object` 类型的 b 是按共享传递的。

JS 中这种设计的原因是：按值传递的类型，复制一份存入栈内存，这类类型一般不占用太多内存，而且按值传递保证了其访问速度。按共享传递的类型，是复制其引用，而不是整个复制其值（C 语言中的指针），保证过大的对象等不会因为不停复制内容而造成内存的浪费。

引用类型经常会在代码中按照下面的写法使用，或者说**容易不知不觉中造成错误**！

```
var obj = {
  a: 1,
  b: [1,2,3]
}
var a = obj.a
var b = obj.b
a = 2
b.push(4)
console.log(obj, a, b)
```

虽然 obj 本身是个引用类型的变量（对象），但是内部的 a 和 b 一个是值类型一个是引用类型，a 的赋值不会改变 obj.a，但是 b 的操作却会反映到 obj 对象上。

原型和原型链

JavaScript 是基于原型的语言，原型理解起来非常简单，但却特别重要，下面还是通过题目来理解下 JavaScript 的原型概念。

题目：如何理解 JavaScript 的原型

对于这个问题，可以从下面这几个要点来理解和回答，**下面几条必须记住并且理解**

- 所有的引用类型（数组、对象、函数），都具有对象特性，即可自由扩展属性（`null` 除外）
- 所有的引用类型（数组、对象、函数），都有一个 `__proto__` 属性，属性值是一个普通的对象
- 所有的函数，都有一个 `prototype` 属性，属性值也是一个普通的对象
- 所有的引用类型（数组、对象、函数），`__proto__` 属性值指向它的构造函数的 `prototype` 属性值

通过代码解释一下，大家可自行运行以下代码，看结果。

```
// 要点一：自由扩展属性
var obj = {}; obj.a = 100;
var arr = []; arr.a = 100;
function fn () {}
fn.a = 100;

// 要点二：__proto__
console.log(obj.__proto__);
console.log(arr.__proto__);
console.log(fn.__proto__);

// 要点三：函数有 prototype
console.log(fn.prototype)
```

```
// 要点四：引用类型的 __proto__ 属性值指向它的构造函数的 prototype 属性值
console.log(obj.__proto__ === Object.prototype)
```

原型

先写一个简单的代码示例。

```
// 构造函数
function Foo(name, age) {
    this.name = name
}
Foo.prototype.alertName = function () {
    alert(this.name)
}
// 创建示例
var f = new Foo('zhangsan')
f.printName = function () {
    console.log(this.name)
}
// 测试
f.printName()
f.alertName()
```

执行 `printName` 时很好理解，但是执行 `alertName` 时发生了什么？这里再记住一个重点 **当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的 `__proto__`（即它的构造函数的 `prototype`）中寻找**，因此 `f.alertName` 就会找到 `Foo.prototype.alertName`。

那么如何判断这个属性是不是对象本身的属性呢？使用 `hasOwnProperty`，常用的地方是遍历一个对象的时候。

```
var item
for (item in f) {
    // 高级浏览器已经在 for in 中屏蔽了来自原型的属性，但是这里建议大家还是加上这个判断，保证程序的健壮性
    if (f.hasOwnProperty(item)) {
        console.log(item)
    }
}
```

题目：如何理解 JS 的原型链

原型链

还是接着上面的示例，如果执行 `f.toString()` 时，又发生了什么？

```
// 省略 N 行

// 测试
f.printName()
f.alertName()
f.toString()
```

因为 `f` 本身没有 `toString()`，并且 `f.__proto__`（即 `Foo.prototype`）中也没有 `toString`。这个问题还是得拿出刚才那句话——**当试图得到一个对象的某个属性时，如果这个对象本身没有这个属性，那么会去它的 `__proto__`（即它的构造函数的 `prototype`）中寻找。**

如果在 `f.__proto__` 中没有找到 `toString`，那么就继续去 `f.__proto__.__proto__` 中寻找，因为 `f.__proto__` 就是一个普通的对象而已嘛！

- `f.__proto__` 即 `Foo.prototype`，没有找到 `toString`，继续往上找
- `f.__proto__.__proto__` 即 `Foo.prototype.__proto__`。 `Foo.prototype` 就是一个普通的对象，因此 `Foo.prototype.__proto__` 就是 `Object.prototype`，在这里可以找到 `toString`
- 因此 `f.toString` 最终对应到了 `Object.prototype.toString`

这样一直往上找，你会发现是一个链式的结构，所以叫做“原型链”。如果一直找到最上层都没有找到，那么就宣告失败，返回 `undefined`。最上层是什么 —— `Object.prototype.__proto__ === null`

原型链中的 `this`

所有从原型或更高级原型中得到、执行的方法，其中的 `this` 在执行时，就指向了当前这个触发事件执行的对象。因此 `printName` 和 `alertName` 中的 `this` 都是 `f`。

作用域和闭包

作用域和闭包是前端面试中，最可能考查的知识点。例如下面的题目：

题目：现在有个 HTML 片段，要求编写代码，点击编号为几的链接就 `alert` 弹出其编号

```
<ul>
  <li>编号1，点击我请弹出1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
```

一般不知道这个题目用闭包的话，会写出下面的代码：

```
var list = document.getElementsByTagName('li');
for (var i = 0; i < list.length; i++) {
  list[i].addEventListener('click', function(){
    alert(i + 1)
  }, true)
}
```

实际上执行才会发现始终弹出的是 6，这时候就应该通过闭包来解决：


```
var list = document.getElementsByTagName('li');
for (var i = 0; i < list.length; i++) {
  list[i].addEventListener('click', function(i){
    return function(){
      alert(i + 1)
    }
  })(i), true)
}
```

要理解闭包，就需要我们从「执行上下文」开始讲起。

执行上下文

先讲一个关于 **变量提升** 的知识点，面试中可能会遇见下面的问题，很多候选人都回答错误：

题目：说出下面执行的结果（这里笔者直接注释输出了）

```
console.log(a)  // undefined
var a = 100

fn('zhangsan')  // 'zhangsan' 20
function fn(name) {
  age = 20
  console.log(name, age)
  var age
}

console.log(b); // 这里报错
// Uncaught ReferenceError: b is not defined
b = 100;
```

在一段 JS 脚本（即一个 `<script>` 标签中）执行之前，要先解析代码（所以说 JS 是解释执行的脚本语言），解析的时候会先创建一个 **全局执行上下文** 环境，先把代码中即将执行的（内部函数的不算，因为你不知道函数何时执行）变量、函数声明都拿出来。变量先暂时赋值为 `undefined`，函数则先声明好可使用。这一步做完了，然后再开始正式执行程序。再次强调，这是在代码执行之前才开始的工作。

我们来看下上面的面试小题目，为什么 `a` 是 `undefined`，而 `b` 却报错了，实际 JS 在代码执行之前，要「全文解析」，发现 `var a`，知道有个 `a` 的变量，存入了执行上下文，而 `b` 没有找到 `var` 关键字，这时候没有在执行上下文提前「占位」，所以代码执行的时候，提前报到的 `a` 是有记录的，只不过值暂时还没有赋值，即为 `undefined`，而 `b` 在执行上下文没有找到，自然会报错（没有找到 `b` 的引用）。

另外，一个函数在执行之前，也会创建一个 **函数执行上下文** 环境，跟 **全局上下文** 差不多，不过 **函数执行上下文** 中会多出 `this`、`arguments` 和函数的参数。参数和 `arguments` 好理解，这里的 `this` 咱们需要专门讲解。

总结一下：

- 范围：一段 `<script>`、js 文件或者一个函数
- 全局上下文：变量定义，函数声明
- 函数上下文：变量定义，函数声明，`this`，`arguments`

this

先搞明白一个很重要的概念 —— **this 的值是在执行的时候才能确认，定义的时候不能确认！** 为什么呢 —— 因为 **this** 是执行上下文环境的一部分，而执行上下文需要在代码执行之前确定，而不是定义的时候。看如下例子

```
var a = {
  name: 'A',
  fn: function () {
    console.log(this.name)
  }
}
a.fn() // this === a
a.fn.call({name: 'B'}) // this === {name: 'B'}
var fn1 = a.fn
fn1() // this === window
```

this 执行会有不同，主要集中在几个场景中

- 作为构造函数执行，构造函数中
- 作为对象属性执行，上述代码中 `a.fn()`
- 作为普通函数执行，上述代码中 `fn1()`
- 用于 `call` `apply` `bind`，上述代码中 `a.fn.call({name: 'B'})`

下面再来讲解下什么是作用域和作用域链，作用域链和作用域也是常考的题目。

题目：如何理解 JS 的作用域和作用域链

作用域

ES6 之前 JS 没有块级作用域。例如

```
if (true) {
  var name = 'zhangsan'
}
console.log(name)
```

从上面的例子可以体会到作用域的概念，作用域就是一个独立的地盘，让变量不会外泄、暴露出去。上面的 `name` 就被暴露出去了，因此，**JS 没有块级作用域，只有全局作用域和函数作用域。**

```
var a = 100
function fn() {
  var a = 200
  console.log('fn', a)
}
console.log('global', a)
fn()
```

全局作用域就是最外层的作用域，如果我们写了很多行 JS 代码，变量定义都没有用函数包括，那么它们就全部都在全局作用域中。这样的坏处就是很容易撞车、冲突。

```
// 张三写的代码中
var data = {a: 100}

// 李四写的代码中
var data = {x: true}
```

这就是为何 jQuery、Zepto 等库的源码，所有的代码都会放在 `(function(){...})()` 中。因为放在里面的所有变量，都不会被外泄和暴露，不会污染到外面，不会对其他的库或者 JS 脚本造成影响。这是函数作用域的一个体现。

附：ES6 中开始加入了块级作用域，使用 `let` 定义变量即可，如下：

```
if (true) {
    let name = 'zhangsan'
}
console.log(name) // 报错，因为let定义的name是在if这个块级作用域
```

作用域链

首先认识一下什么叫做 **自由变量**。如下代码中，`console.log(a)` 要得到 `a` 变量，但是在当前的作用域中没有定义 `a`（可对比一下 `b`）。当前作用域没有定义的变量，这成为 **自由变量**。自由变量如何得到——向父级作用域寻找。

```
var a = 100
function fn() {
    var b = 200
    console.log(a)
    console.log(b)
}
fn()
```

如果父级也没呢？再一层一层向上寻找，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是 **作用域链**。

```
var a = 100
function F1() {
    var b = 200
    function F2() {
        var c = 300
        console.log(a) // 自由变量，顺作用域链向父作用域找
        console.log(b) // 自由变量，顺作用域链向父作用域找
        console.log(c) // 本作用域的变量
    }
    F2()
}
F1()
```

闭包

讲完这些内容，我们再来看一个例子，通过例子来理解闭包。

```
function F1() {  
    var a = 100  
    return function () {  
        console.log(a)  
    }  
}  
var f1 = F1()  
var a = 200  
f1()
```

自由变量将从作用域链中寻找，但是 **依据的是函数定义时的作用域链，而不是函数执行时**，以上这个例子就是闭包。闭包主要有两个应用场景：

- **函数作为返回值**，上面的例子就是
- **函数作为参数传递**，看以下例子

```
function F1() {  
    var a = 100  
    return function () {  
        console.log(a)  
    }  
}  
function F2(f1) {  
    var a = 200  
    console.log(f1())  
}  
var f1 = F1()  
F2(f1)
```

至此，对应着「作用域和闭包」这部分一开始的点击弹出 alert 的代码再看闭包，就很好理解了。

异步

异步和同步也是面试中常考的内容，下面笔者来讲解下同步和异步的区别。

同步 vs 异步

先看下面的 demo，根据程序阅读起来表达的意思，应该是先打印 100，1秒钟之后打印 200，最后打印 300。但是实际运行根本不是那么回事。

```
console.log(100)
setTimeout(function () {
  console.log(200)
}, 1000)
console.log(300)
```

再对比以下程序。先打印 100，再弹出 200（等待用户确认），最后打印 300。这个运行效果就符合预期要求。

```
console.log(100)
alert(200) // 1秒钟之后点击确认
console.log(300)
```

这两到底有何区别？—— 第一个示例中间的步骤根本没有阻塞接下来程序的运行，而第二个示例却阻塞了后面程序的运行。前面这种表现就叫做 **异步**（后面这个叫做 **同步**），即**不会阻塞后面程序的运行**。

异步和单线程

JS 需要异步的根本原因是 **JS 是单线程运行的**，即在同一时间只能做一件事，不能“一心二用”。

一个 Ajax 请求由于网络比较慢，请求需要 5 秒钟。如果是同步，这 5 秒钟页面就卡死在这里啥也干不了了。异步的话，就好很多了，5 秒等待就等待了，其他事情不耽误做，至于那 5 秒钟等待是网速太慢，不是因为 JS 的原因。

讲到单线程，我们再来看个真题：

题目：讲解下面代码的执行过程和结果

```
var a = true;
setTimeout(function(){
  a = false;
}, 100)
while(a){
  console.log('while执行了')
}
```

这是一个很有迷惑性的题目，不少候选人认为 100ms 之后，由于 a 变成了 false，所以 while 就中止了，实际不是这样，因为 JS 是单线程的，所以进入 while 循环之后，没有「时间」（线程）去跑定时器了，所以这个代码跑起来是个死循环！

前端异步的场景

- 定时 `setTimeout` `setInterval`
- 网络请求，如 `Ajax` `` 加载

Ajax 代码示例

```
console.log('start')
$.get('./data1.json', function (data1) {
  console.log(data1)
})
console.log('end')
```

img 代码示例（常用于打点统计）

```
console.log('start')
var img = document.createElement('img')
// 或者 img = new Image()
img.onload = function () {
  console.log('loaded')
  img.onload = null
}
img.src = '/xxx.png'
console.log('end')
```

ES6/7 新标准的考查

题目：ES6 箭头函数中的 `this` 和普通函数中的有什么不同

箭头函数

箭头函数是 ES6 中新的函数定义形式，`function name(arg1, arg2) {...}` 可以使用 `(arg1, arg2) => {...}` 来定义。示例如下：

```
// JS 普通函数
var arr = [1, 2, 3]
arr.map(function (item) {
  console.log(index)
  return item + 1
})

// ES6 箭头函数
const arr = [1, 2, 3]
arr.map((item, index) => {
  console.log(index)
  return item + 1
})
```

箭头函数存在的意义，第一写起来更加简洁，第二可以解决 ES6 之前函数执行中 `this` 是全局变量的问题，看如下代码

```
function fn() {
  console.log('real', this) // {a: 100}，该作用域下的 this 的真实值
  var arr = [1, 2, 3]
```

```
// 普通 JS
arr.map(function (item) {
  console.log('js', this) // window 。普通函数，这里打印出来的是全局变量，令人费解
  return item + 1
})
// 箭头函数
arr.map(item => {
  console.log('es6', this) // {a: 100} 。箭头函数，这里打印的就是父作用域的 this
  return item + 1
})
}
fn.call({a: 100})
```

题目：ES6 模块化如何使用？

Module

ES6 中模块化语法更加简洁，直接看示例。

如果只是输出一个唯一的对象，使用 `export default` 即可，代码如下

```
// 创建 util1.js 文件，内容如
export default {
  a: 100
}

// 创建 index.js 文件，内容如
import obj from './util1.js'
console.log(obj)
```

如果想要输出许多个对象，就不能用 `default` 了，且 `import` 时候要加 `{...}`，代码如下

```
// 创建 util2.js 文件，内容如
export function fn1() {
  alert('fn1')
}
export function fn2() {
  alert('fn2')
}

// 创建 index.js 文件，内容如
import { fn1, fn2 } from './util2.js'
fn1()
fn2()
```

题目：ES6 class 和普通构造函数的区别

class

class 其实一直是 JS 的关键字（保留字），但是一直没有正式使用，直到 ES6。ES6 的 class 就是取代之前构造函数初始化对象的形式，从语法上更加符合面向对象的写法。例如：

JS 构造函数的写法

```
function MathHandle(x, y) {
  this.x = x;
  this.y = y;
}

MathHandle.prototype.add = function () {
  return this.x + this.y;
};

var m = new MathHandle(1, 2);
console.log(m.add())
```

用 ES6 class 的写法

```
class MathHandle {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  add() {
    return this.x + this.y;
  }
}

const m = new MathHandle(1, 2);
console.log(m.add())
```

注意以下几点，全都是关于 class 语法的：

- class 是一种新的语法形式，是 `class Name {...}` 这种形式，和函数的写法完全不一样
- 两者对比，构造函数函数体的内容要放在 class 中的 `constructor` 函数中，`constructor` 即构造器，初始化实例时默认执行
- class 中函数的写法是 `add() {...}` 这种形式，并没有 `function` 关键字

使用 class 来实现继承就更加简单了，至少比构造函数实现继承简单很多。看下面例子

JS 构造函数实现继承

```
// 动物
function Animal() {
  this.eat = function () {
    console.log('animal eat')
  }
}

// 狗
function Dog() {
  this.bark = function () {
```



```
        console.log('dog bark')
    }
}
Dog.prototype = new Animal()
// 哈士奇
var hashiqi = new Dog()
```

ES6 class 实现继承

```
class Animal {
  constructor(name) {
    this.name = name
  }
  eat() {
    console.log(`${this.name} eat`)
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name)
    this.name = name
  }
  say() {
    console.log(`${this.name} say`)
  }
}

const dog = new Dog('哈士奇')
dog.say()
dog.eat()
```

注意以下两点：

- 使用 `extends` 即可实现继承，更加符合经典面向对象语言的写法，如 Java
- 子类的 `constructor` 一定要执行 `super()`，以调用父类的 `constructor`

题目：ES6 中新增的数据类型有哪些？

Set 和 Map

Set 和 Map 都是 ES6 中新增的数据结构，是对当前 JS 数组和对象这两种重要数据结构的扩展。由于是新增的数据结构，目前尚未被大规模使用，但是作为前端程序员，提前了解是必须做到的。先总结一下两者最关键的地方：

- Set 类似于数组，但数组可以允许元素重复，Set 不允许元素重复
- Map 类似于对象，但普通对象的 key 必须是字符串或者数字，而 Map 的 key 可以是任何数据类型

Set

Set 实例不允许元素有重复，可以通过以下示例证明。可以通过一个数组初始化一个 Set 实例，或者通过 `add` 添加元素，元素不能重复，重复的会被忽略。

```
// 例1
const set = new Set([1, 2, 3, 4, 4]);
console.log(set) // Set(4) {1, 2, 3, 4}

// 例2
const set = new Set();
[2, 3, 5, 4, 5, 8, 8].forEach(item => set.add(item));
for (let item of set) {
  console.log(item);
}
// 2 3 5 4 8
```

Set 实例的属性和方法有

- `size`：获取元素数量。
- `add(value)`：添加元素，返回 Set 实例本身。
- `delete(value)`：删除元素，返回一个布尔值，表示删除是否成功。
- `has(value)`：返回一个布尔值，表示该值是否是 Set 实例的元素。
- `clear()`：清除所有元素，没有返回值。

```
const s = new Set();
s.add(1).add(2).add(2); // 添加元素

s.size // 2

s.has(1) // true
s.has(2) // true
s.has(3) // false

s.delete(2);
s.has(2) // false

s.clear();
console.log(s); // Set(0) {}
```

Set 实例的遍历，可使用如下方法

- `keys()`：返回键名的遍历器。
- `values()`：返回键值的遍历器。不过由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以 `keys()` 和 `values()` 返回结果一致。
- `entries()`：返回键值对的遍历器。
- `forEach()`：使用回调函数遍历每个成员。

```
let set = new Set(['aaa', 'bbb', 'ccc']);

for (let item of set.keys()) {
  console.log(item);
}
// aaa
// bbb
// ccc
```

```
for (let item of set.values()) {
  console.log(item);
}
// aaa
// bbb
// ccc

for (let item of set.entries()) {
  console.log(item);
}
// ["aaa", "aaa"]
// ["bbb", "bbb"]
// ["ccc", "ccc"]

set.forEach((value, key) => console.log(key + ' : ' + value))
// aaa : aaa
// bbb : bbb
// ccc : ccc
```

Map

Map 的用法和普通对象基本一致，先看一下它能用非字符串或者数字作为 key 的特性。

```
const map = new Map();
const obj = {p: 'Hello world'};

map.set(obj, 'OK')
map.get(obj) // "OK"

map.has(obj) // true
map.delete(obj) // true
map.has(obj) // false
```

需要使用 `new Map()` 初始化一个实例，下面代码中 `set` `get` `has` `delete` 顾名思义（下文也会演示）。其中，`map.set(obj, 'OK')` 就是用对象作为的 key（不光可以是对象，任何数据类型都可以），并且后面通过 `map.get(obj)` 正确获取了。

Map 实例的属性和方法如下：

- `size`：获取成员的数量
- `set`：设置成员 key 和 value
- `get`：获取成员属性值
- `has`：判断成员是否存在
- `delete`：删除成员
- `clear`：清空所有

```
const map = new Map();
map.set('aaa', 100);
map.set('bbb', 200);

map.size // 2
```

```
map.get('aaa') // 100

map.has('aaa') // true

map.delete('aaa')
map.has('aaa') // false

map.clear()
```

Map 实例的遍历方法有：

- `keys()`：返回键名的遍历器。
- `values()`：返回键值的遍历器。
- `entries()`：返回所有成员的遍历器。
- `forEach()`：遍历 Map 的所有成员。

```
const map = new Map();
map.set('aaa', 100);
map.set('bbb', 200);

for (let key of map.keys()) {
  console.log(key);
}
// "aaa"
// "bbb"

for (let value of map.values()) {
  console.log(value);
}
// 100
// 200

for (let item of map.entries()) {
  console.log(item[0], item[1]);
}
// aaa 100
// bbb 200

// 或者
for (let [key, value] of map.entries()) {
  console.log(key, value);
}
// aaa 100
// bbb 200
```

Promise

`Promise` 是 CommonJS 提出来的这一种规范，有多个版本，在 ES6 当中已经纳入规范，原生支持 `Promise` 对象，非 ES6 环境可以用类似 `Bluebird`、`Q` 这类库来支持。

`Promise` 可以将回调变成链式调用写法，流程更加清晰，代码更加优雅。

简单归纳下 Promise：**三个状态、两个过程、一个方法**，快速记忆方法：**3-2-1**

三个状态：`pending`、`fulfilled`、`rejected`

两个过程：

- `pending`→`fulfilled` (`resolve`)
- `pending`→`rejected` (`reject`)

一个方法：`then`

当然还有其他概念，如 `catch`、`Promise.all/race`，这里就不展开了。

关于 ES6/7 的考查内容还有很多，本小节就不逐一介绍了，如果想继续深入学习，可以在线看《[ES6入门](#)》。

小结

本小节主要总结了 ES 基础语法中面试经常考查的知识点，包括之前就考查较多的原型、异步、作用域，以及 ES6 的一些新内容，这些知识点希望大家都要掌握。

一面 2：JS-Web-API 知识点与高频考题解析

除 ES 基础之外，Web 前端经常会用到一些跟浏览器相关的 API，接下来我们一起梳理一下。

知识点梳理

- BOM 操作
- DOM 操作
- 事件绑定
- Ajax
- 存储
- ◦ *

BOM

BOM（浏览器对象模型）是浏览器本身的一些信息的设置和获取，例如获取浏览器的宽度、高度，设置让浏览器跳转到哪个地址。

- `navigator`
- `screen`
- `location`
- `history`

这些对象就是一堆非常简单粗暴的 API，没任何技术含量，讲起来一点意思都没有，大家去 MDN 或者 w3school 这种网站一查就都明白了。面试的时候，面试官基本不会出太多这方面的题目，因为只要基础知识过关了，这些 API 即便你记不住，上网一查也知道了。下面列举一下常用功能的代码示例

获取浏览器特性（即俗称的 UA）然后识别客户端，例如判断是不是 Chrome 浏览器

```
var ua = navigator.userAgent
var isChrome = ua.indexOf('Chrome')
console.log(isChrome)
```

获取屏幕的宽度和高度

```
console.log(screen.width)
console.log(screen.height)
```

获取网址、协议、path、参数、hash 等

```
// 例如当前网址是 https://juejin.im/timeline/frontend?a=10&b=10#some
console.log(location.href) // https://juejin.im/timeline/frontend?
a=10&b=10#some
console.log(location.protocol) // https:
console.log(location.pathname) // /timeline/frontend
console.log(location.search) // ?a=10&b=10
console.log(location.hash) // #some
```

另外，还有调用浏览器的前进、后退功能等

```
history.back()
history.forward()
```

DOM

题目：DOM 和 HTML 区别和联系

什么是 DOM

讲 DOM 先从 HTML 讲起，讲 HTML 先从 XML 讲起。XML 是一种可扩展的标记语言，所谓可扩展就是它可以描述任何结构化的数据，它是一棵树！

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
  <other>
    <a></a>
    <b></b>
  </other>
</note>
```

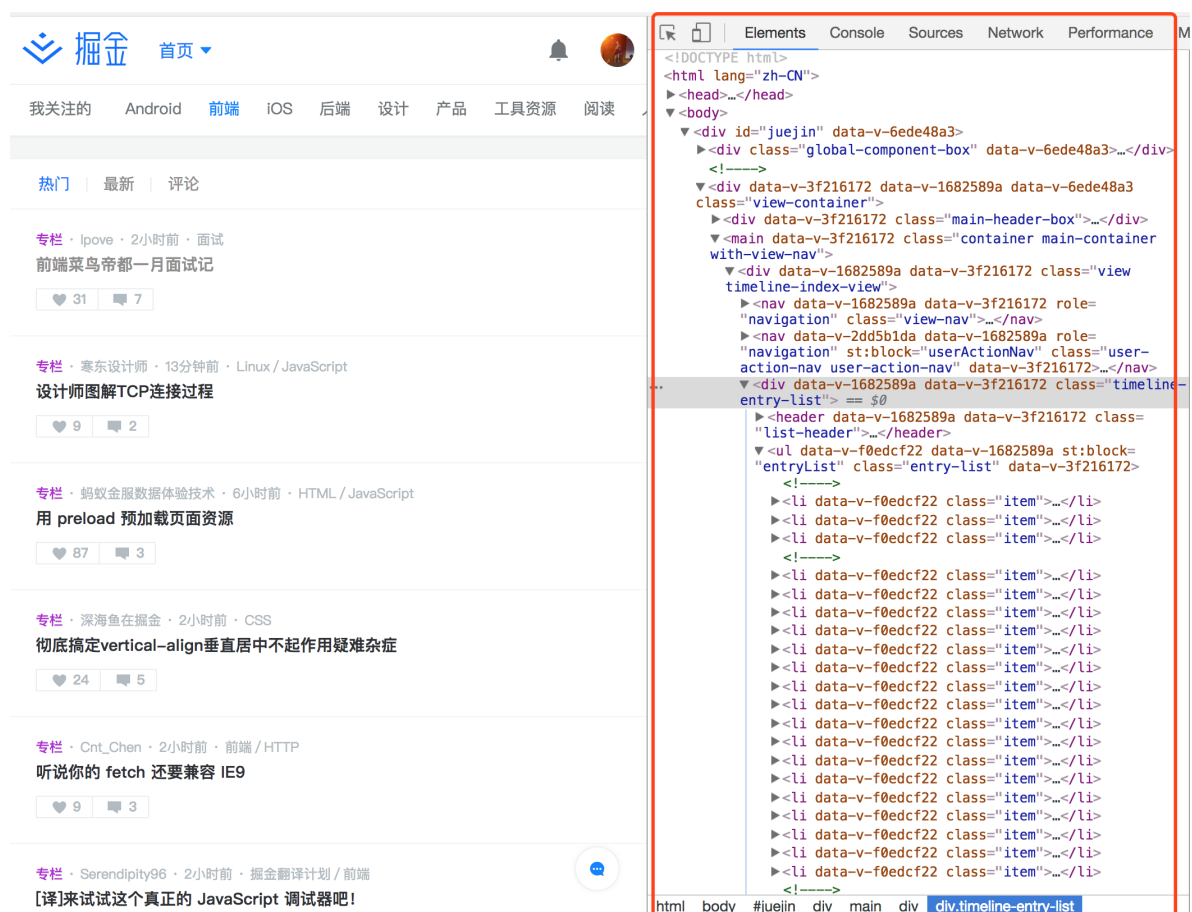
HTML 是一个有既定标签标准的 XML 格式，标签的名字、层级关系和属性，都被标准化（否则浏览器无法解析）。同样，它也是一棵树。

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <div>
    <p>this is p</p>
  </div>
</body>
</html>
```

我们开发完的 HTML 代码会保存到一个文档中（一般以 .html 或者 .htm 结尾），文档放在服务器上，浏览器请求服务器，这个文档被返回。因此，最终浏览器拿到的是一个文档而已，文档的内容就是 HTML 格式的代码。

但是浏览器要把这个文档中的 HTML 按照标准渲染成一个页面，此时浏览器就需要将这堆代码处理成自己能理解的东西，也得处理成 JS 能理解的东西，因为还得允许 JS 修改页面内容呢。

基于以上需求，浏览器就需要把 HTML 转变成 DOM，HTML 是一棵树，DOM 也是一棵树。对 DOM 的理解，可以暂时先抛开浏览器的内部因素，先从 JS 着手，即可以认为 DOM 就是 JS 能识别的 HTML 结构，一个普通的 JS 对象或者数组。



The screenshot shows a web browser interface with a list of articles on the left and the browser's developer tools on the right. The developer tools show the DOM tree with a red box highlighting the 'div.timeline-entry-list' element, which contains a list of article items.

获取 DOM 节点

最常用的 DOM API 就是获取节点，其中常用的获取方法如下面代码示例：

```
// 通过 id 获取
var div1 = document.getElementById('div1') // 元素
```

```
// 通过 tagname 获取
var divList = document.getElementsByTagName('div') // 集合
console.log(divList.length)
console.log(divList[0])

// 通过 class 获取
var containerList = document.getElementsByClassName('container') // 集合

// 通过 CSS 选择器获取
var pList = document.querySelectorAll('p') // 集合
```

题目：property 和 attribute 的区别是什么？

property

DOM 节点就是一个 JS 对象，它符合之前讲述的对象特征——可扩展属性，因为 DOM 节点本质上也是一个 JS 对象。因此，如下代码所示，`p` 可以有 `style` 属性，有 `className` `nodeName` `nodeType` 属性。注意，**这些都是 JS 范畴的属性，符合 JS 语法标准的。**

```
var pList = document.querySelectorAll('p')
var p = pList[0]
console.log(p.style.width) // 获取样式
p.style.width = '100px' // 修改样式
console.log(p.className) // 获取 class
p.className = 'p1' // 修改 class

// 获取 nodeName 和 nodeType
console.log(p.nodeName)
console.log(p.nodeType)
```

attribute

property 的获取和修改，是直接改变 JS 对象，而 attribute 是直接改变 HTML 的属性，两种有很大的区别。attribute 就是对 HTML 属性的 get 和 set，和 DOM 节点的 JS 范畴的 property 没有关系。

```
var pList = document.querySelectorAll('p')
var p = pList[0]
p.getAttribute('data-name')
p.setAttribute('data-name', 'juejin')
p.getAttribute('style')
p.setAttribute('style', 'font-size:30px;')
```

而且，get 和 set attribute 时，还会触发 DOM 的查询或者重绘、重排，频繁操作会影响页面性能。

题目：DOM 操作的基本 API 有哪些？

DOM 树操作

新增节点


```
var div1 = document.getElementById('div1')

// 添加新节点
var p1 = document.createElement('p')
p1.innerHTML = 'this is p1'
div1.appendChild(p1) // 添加新创建的元素

// 移动已有节点。注意，这里是“移动”，并不是拷贝
var p2 = document.getElementById('p2')
div1.appendChild(p2)
```

获取父元素

```
var div1 = document.getElementById('div1')
var parent = div1.parentElement
```

获取子元素

```
var div1 = document.getElementById('div1')
var child = div1.childNodes
```

删除节点

```
var div1 = document.getElementById('div1')
var child = div1.childNodes
div1.removeChild(child[0])
```

还有其他操作的API，例如获取前一个节点、获取后一个节点等，但是面试过程中经常考到的就是上面几个。

事件

事件绑定

普通的事件绑定写法如下：

```
var btn = document.getElementById('btn1')
btn.addEventListener('click', function (event) {
  // event.preventDefault() // 阻止默认行为
  // event.stopPropagation() // 阻止冒泡
  console.log('clicked')
})
```

为了编写简单的事件绑定，可以编写通用的事件绑定函数。这里虽然比较简单，但是会随着后文的讲解，来继续完善和丰富这个函数。

```
// 通用的事件绑定函数
function bindEvent(elem, type, fn) {
    elem.addEventListener(type, fn)
}
var a = document.getElementById('link1')
// 写起来更加简单了
bindEvent(a, 'click', function(e) {
    e.preventDefault() // 阻止默认行为
    alert('clicked')
})
```

最后，如果面试被问到 IE 低版本兼容性问题，我劝你果断放弃这份工作机会。现在互联网流量都在 App 上，IE 占比越来越少，再去为 IE 浪费青春不值得，要尽量去做 App 相关的工作。

题目：什么是事件冒泡？

事件冒泡

```
<body>
  <div id="div1">
    <p id="p1">激活</p>
    <p id="p2">取消</p>
    <p id="p3">取消</p>
    <p id="p4">取消</p>
  </div>
  <div id="div2">
    <p id="p5">取消</p>
    <p id="p6">取消</p>
  </div>
</body>
```

对于以上 HTML 代码结构，要求点击 p1 时候进入激活状态，点击其他任何 <p> 都取消激活状态，如何实现？代码如下，注意看注释：

```
var body = document.body
bindEvent(body, 'click', function (e) {
    // 所有 p 的点击都会冒泡到 body 上，因为 DOM 结构中 body 是 p 的上级节点，事件会沿着 DOM 树向上冒泡
    alert('取消')
})

var p1 = document.getElementById('p1')
bindEvent(p1, 'click', function (e) {
    e.stopPropagation() // 阻止冒泡
    alert('激活')
})
```

如果我们在 p1 div1 body 中都绑定了事件，它是会根据 DOM 的结构来冒泡，从下到上挨个执行的。但是我们使用 e.stopPropagation() 就可以阻止冒泡

题目：如何使用事件代理？有何好处？

事件代理

我们设定一种场景，如下代码，一个 `<div>` 中包含了若干个 `<a>`，而且还能继续增加。那如何快捷方便地为所有 `<a>` 绑定事件呢？

```
<div id="div1">
  <a href="#">a1</a>
  <a href="#">a2</a>
  <a href="#">a3</a>
  <a href="#">a4</a>
</div>
<button>点击增加一个 a 标签</button>
```

这里就会用到事件代理。我们要监听 `<a>` 的事件，但要把具体的事件绑定到 `<div>` 上，然后看事件的触发点是不是 `<a>`。

```
var div1 = document.getElementById('div1')
div1.addEventListener('click', function (e) {
  // e.target 可以监听到触发点击事件的元素是哪一个
  var target = e.target
  if (e.nodeName === 'A') {
    // 点击的是 <a> 元素
    alert(target.innerHTML)
  }
})
```

我们现在完善一下之前写的通用事件绑定函数，加上事件代理。

```
function bindEvent(elem, type, selector, fn) {
  // 这样处理，可接收两种调用方式 bindEvent(div1, 'click', 'a', function () {...})
  // 和 bindEvent(div1, 'click', function () {...}) 这两种
  if (fn == null) {
    fn = selector
    selector = null
  }

  // 绑定事件
  elem.addEventListener(type, function (e) {
    var target
    if (selector) {
      // 有 selector 说明需要做事件代理
      // 获取触发时间的元素，即 e.target
      target = e.target
      // 看是否符合 selector 这个条件
      if (target.matches(selector)) {
        fn.call(target, e)
      }
    } else {
      // 无 selector，说明不需要事件代理
    }
  })
}
```

```
        fn(e)
      }
    })
  }
}
```

然后这样使用，简单很多。

```
// 使用代理，bindEvent 多一个 'a' 参数
var div1 = document.getElementById('div1')
bindEvent(div1, 'click', 'a', function (e) {
  console.log(this.innerHTML)
})

// 不使用代理
var a = document.getElementById('a1')
bindEvent(div1, 'click', function (e) {
  console.log(a.innerHTML)
})
```

最后，使用代理的优点如下：

- 使代码简洁
- 减少浏览器的内存占用
- ○ *

Ajax

XMLHttpRequest

题目：手写 XMLHttpRequest 不借助任何库

这是很多奇葩的、个性的面试官经常用的手段。这种考查方式存在很多争议，但是你不能完全说它是错误的，毕竟也是考查对最基础知识的掌握情况。

```
var xhr = new XMLHttpRequest()
xhr.onreadystatechange = function () {
  // 这里的函数异步执行，可参考之前 JS 基础中的异步模块
  if (xhr.readyState == 4) {
    if (xhr.status == 200) {
      alert(xhr.responseText)
    }
  }
}
xhr.open("GET", "/api", false)
xhr.send(null)
```

当然，使用 jQuery、Zepto 或 Fetch 等库来写就更加简单了，这里不再赘述。

状态码说明

上述代码中，有两处状态码需要说明。`xhr.readyState` 是浏览器判断请求过程中各个阶段的，`xhr.status` 是 HTTP 协议中规定的不同结果的返回状态说明。

`xhr.readyState` 的状态码说明：

- 0 -代理被创建，但尚未调用 `open()` 方法。
- 1 -`open()` 方法已经被调用。
- 2 -`send()` 方法已经被调用，并且头部和状态已经可获得。
- 3 -下载中，`responseText` 属性已经包含部分数据。
- 4 -下载操作已完成

题目：HTTP 协议中，response 的状态码，常见的有哪些？

`xhr.status` 即 HTTP 状态码，有 2xx 3xx 4xx 5xx 这几种，比较常用的有以下几种：

- 200 正常
- 3xx
 - 301 永久重定向。如 `http://xxx.com` 这个 GET 请求（最后没有 /），就会被 301 到 `http://xxx.com/`（最后是 /）
 - 302 临时重定向。临时的，不是永久的
 - 304 资源找到但是不符合请求条件，不会返回任何主体。如发送 GET 请求时，head 中有 `If-Modified-Since: xxx`（要求返回更新时间是 xxx 时间之后的资源），如果此时服务器端资源未更新，则会返回 304，即不符合要求
- 404 找不到资源
- 5xx 服务器端出错了

看完要明白，为何上述代码中要同时满足 `xhr.readyState == 4` 和 `xhr.status == 200`。

Fetch API

目前已经有一个获取 HTTP 请求更加方便的 API：`Fetch`，通过 `Fetch` 提供的 `fetch()` 这个全局函数方法可以很简单地发起异步请求，并且支持 `Promise` 的回调。但是 `Fetch` API 是比较新的 API，具体使用的时候还需要查查 [caniuse](#)，看下其浏览器兼容情况。

看一个简单的例子：

```
fetch('some/api/data.json', {
  method: 'POST', //请求类型 GET、POST
  headers: {}, // 请求的头信息，形式为 Headers 对象或 ByteString
  body: {}, //请求发送的数据 blob、BufferSource、FormData、URLSearchParams (get 或 head 方法中不能包含 body)
  mode: '', //请求的模式，是否跨域等，如 cors、no-cors 或 same-origin
  credentials: '', //cookie 的跨域策略，如 omit、same-origin 或 include
  cache: '', //请求的 cache 模式：default、no-store、reload、no-cache、force-cache 或 only-if-cached
}).then(function(response) { ... });
```

`Fetch` 支持 `headers` 定义，通过 `headers` 自定义可以方便地实现多种请求方法（PUT、GET、POST 等）、请求头（包括跨域）和 `cache` 策略等；除此之外还支持 `response`（返回数据）多种类型，比如支持二进制文件、字符串和 `formData` 等。

跨域

题目：如何实现跨域？

浏览器中有 **同源策略**，即一个域下的页面中，无法通过 Ajax 获取到其他域的接口。例如有一个接口

`http://m.juejin.com/course/ajaxcourse/recom?cid=459`，你自己的一个页面

`http://www.yourname.com/page1.html` 中的 Ajax 无法获取这个接口。这正是命中了“同源策略”。如果浏览器哪些地方忽略了同源策略，那就是浏览器的安全漏洞，需要紧急修复。

url 哪些地方不同算作跨域？

- 协议
- 域名
- 端口

但是 HTML 中几个标签能逃避过同源策略——`<script src="xxx">`、``、`<link href="xxxx">`，这三个标签的 `src/href` 可以加载其他域的资源，不受同源策略限制。

因此，这使得这三个标签可以做一些特殊的事情。

- `` 可以做打点统计，因为统计方并不一定是同域的，在讲解 JS 基础知识异步的时候有过代码示例。除了能跨域之外，`` 几乎没有浏览器兼容问题，它是一个非常古老的标签。
- `<script>` 和 `<link>` 可以使用 CDN，CDN 基本都是其他域的连接。
- 另外 `<script>` 还可以实现 JSONP，能获取其他域接口的信息，接下来马上讲解。

但是请注意，所有的跨域请求方式，最终都需要信息提供方来做出相应的支持和改动，也就是要经过信息提供方的同意才行，否则接收方是无法得到它们的信息的，浏览器是不允许的。

解决跨域 - JSONP

首先，有一个概念你要明白，例如访问 `http://coding.m.juejin.com/classindex.html` 的时候，服务器端就一定有一个 `classindex.html` 文件吗？——不一定，服务器可以拿到这个请求，动态生成一个文件，然后返回。同理，`<script src="http://coding.m.juejin.com/api.js">` 也不一定加载一个服务器端的静态文件，服务器也可以动态生成文件并返回。OK，接下来正式开始。

例如我们的网站和掘金网，肯定不是一个域。我们需要掘金网提供一个接口，供我们来获取。首先，我们在自己的页面这样定义

```
<script>
window.callback = function (data) {
  // 这是我们跨域得到信息
  console.log(data)
}
</script>
```

然后掘金网给我提供了一个 `http://coding.m.juejin.com/api.js`，内容如下（之前说过，服务器可动态生成内容）

```
callback({x:100, y:200})
```

最后我们在页面中加入 `<script src="http://coding.m.juejin.com/api.js"></script>`，那么这个 js 加载之后，就会执行内容，我们就得到内容了。

解决跨域 - 服务器端设置 http header

这是需要在服务器端设置的，作为前端工程师我们不用详细掌握，但是要知道有这么个解决方案。而且，现在推崇的跨域解决方案是这一种，比 JSONP 简单许多。

```
response.setHeader("Access-Control-Allow-Origin", "http://m.juejin.com/"); //
第二个参数填写允许跨域的域名称，不建议直接写 "*"
response.setHeader("Access-Control-Allow-Headers", "X-Requested-With");
response.setHeader("Access-Control-Allow-Methods",
"PUT,POST,GET,DELETE,OPTIONS");

// 接收跨域的cookie
response.setHeader("Access-Control-Allow-Credentials", "true");
```

存储

题目：cookie 和 localStorage 有何区别？

cookie

cookie 本身不是用来做服务器端存储的（计算机领域有很多这种“狗拿耗子”的例子，例如 CSS 中的 float），它是设计用来在服务器和客户端进行信息传递的，因此我们的每个 HTTP 请求都带着 cookie。但是 cookie 也具备浏览器端存储的能力（例如记住用户名和密码），因此就被开发者用上了。

使用起来也非常简单，`document.cookie =` 即可。

但是 cookie 有它致命的缺点：

- 存储量太小，只有 4KB
- 所有 HTTP 请求都带着，会影响获取资源的效率
- API 简单，需要封装才能用

localStorage 和 sessionStorage

后来，HTML5 标准就带来了 `sessionStorage` 和 `localStorage`，先拿 `localStorage` 来说，它是专门为了浏览器端缓存而设计的。其优点有：

- 存储量增大到 5MB
- 不会带到 HTTP 请求中
- API 适用于数据存储 `localStorage.setItem(key, value)` `localStorage.getItem(key)`

`sessionStorage` 的区别就在于它是根据 session 过去时间而实现，而 `localStorage` 会永久有效，应用场景不同。例如，一些需要及时失效的重要信息放在 `sessionStorage` 中，一些不重要但是不经常设置的信息，放在 `localStorage` 中。

另外告诉大家一个小技巧，针对 `localStorage.setItem`，使用时尽量加入到 `try-catch` 中，某些浏览器是禁用这个 API 的，要注意。

小结

本小节总结了 W3C 标准中 Web-API 部分，面试中常考的知识点，这些也是日常开发中最常用的 API 和知识。

一面 3：CSS-HTML 知识点与高频考题解析

CSS 和 HTML 是网页开发中布局相关的组成部分，涉及的内容比较多和杂乱，本小节重点介绍下常考的知识点。

知识点梳理

- 选择器的权重和优先级
- 盒模型
 - 盒子大小计算
 - margin 的重叠计算
- 浮动 float
 - 浮动布局概念
 - 清理浮动
- 定位 position
 - 文档流概念
 - 定位分类
 - fixed 定位特点
 - 绝对定位计算方式
- flex 布局
- 如何实现居中对齐？
- 理解语义化
- CSS3 动画
- 重绘和回流
- ◦ *

选择器的权重和优先级

CSS 选择器有很多，不同的选择器的权重和优先级不一样，对于一个元素，如果存在多个选择器，那么就需要根据权重来计算其优先级。

权重分为四级，分别是：

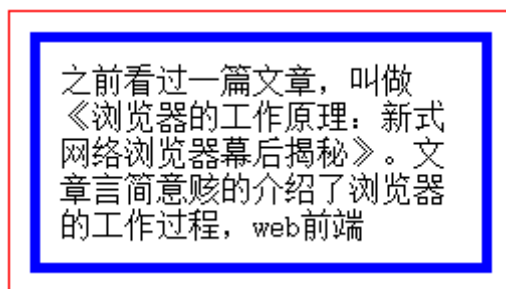
1. 代表内联样式，如 `style="xxx"`，权值为 1000；
2. 代表 ID 选择器，如 `#content`，权值为 100；
3. 代表类、伪类和属性选择器，如 `.content`、`:hover`、`[attribute]`，权值为 10；
4. 代表元素选择器和伪元素选择器，如 `div`、`p`，权值为 1。

需要注意的是：通用选择器 (*)、子选择器 (>) 和相邻同胞选择器 (+) 并不在这四个等级中，所以他们的权值都为 0。 权重值大的选择器其优先级也高，相同权重的优先级又遵循后定义覆盖前面定义的情况。

盒模型

什么是“盒子”

初学 CSS 的朋友，一开始学 CSS 基础知识的时候一定学过 `padding` `border` 和 `margin`，即内边距、边框和外边距。它们三者就构成了一个“盒子”。就像我们收到的快递，本来买了一部小小的手机，收到的却是那么大一个盒子。因为手机白色的包装盒和手机机器之间有间隔层（内边距），手机白色盒子有厚度，虽然很薄（边框），盒子和快递箱子之间还有一层泡沫板（外边距）。这就是一个典型的盒子。

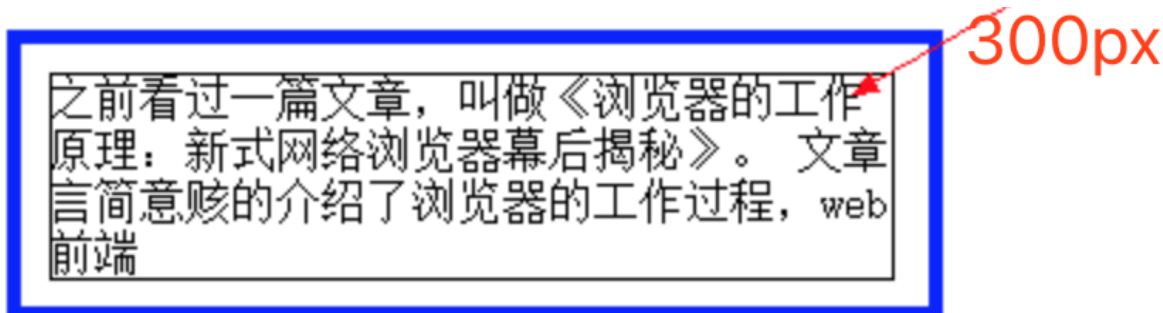


如上图，真正的内容就是这些文字，文字外围有 10px 的内边距，5px 的边框，10px 的外边距。看到盒子了吧？

题目：盒子模型的宽度如何计算

固定宽度的盒子

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px;">
  之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》，
  文章言简意赅的介绍了浏览器的工作过程，web前端
</div>
```



如上图，得到网页效果之后，我们可以用截图工具来量一下文字内容的宽度。发现，文字内容的宽度刚好是 300px，也就是我们设置的宽度。

因此，在盒子模型中，我们设置的宽度都是内容宽度，不是整个盒子的宽度。而整个盒子的宽度是：
(内容宽度 + border 宽度 + padding 宽度 + margin 宽度) 之和。这样我们改四个中的其中一个，都会导致盒子宽度的改变。这对我们来说不友好。

没关系，这个东西不友好早就有人发现了，而且已经解决，下文再说。

充满父容器的盒子

默认情况下，`div` 是 `display:block`，宽度会充满整个父容器。如下图：

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px;">
  之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》，
  文章言简意赅的介绍了浏览器的工作过程，web前端
  之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》，
  文章言简意赅的介绍了浏览器的工作过程，web前端
</div>
```

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》。文章言简意赅的介绍了浏览器的工作过程，web前端

但是别忘记，这个 div 是个盒子模型，它的整个宽度包括（内容宽度 + border 宽度 + padding 宽度 + margin 宽度），整个的宽度充满父容器。

问题就在这里。如果父容器宽度不变，我们手动增大 margin、border 或 padding 其中一项的宽度值，都会导致内容宽度的减少。极端情况下，如果内容的宽度压缩到不能再压缩了（例如一个字的宽度），那么浏览器会强迫增加父容器的宽度。这可不是我们想要看到的。

包裹内容的盒子

这种情况下比较简单，内容的宽度按照内容计算，盒子的宽度将在内容宽度的基础上再增加（padding 宽度 + border 宽度 + margin 宽度）之和。

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px;">
    之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》
</div>
```

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》

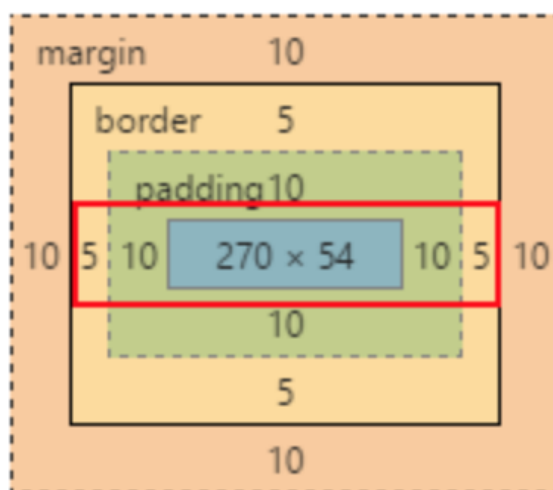
box-sizing:border-box

前面提到，为盒子模型设置宽度，结果只是设置了内容的宽度，这个不合理。如何解决这一问题？答案就是为盒子指定样式：box-sizing:border-box。

```
<div style="padding:10px; border:5px solid blue; margin: 10px; width:300px; box-
sizing:border-box;">
    之前看过一篇文章，叫做《浏览器工作原理：新式网络浏览器幕后揭秘》
</div>
```

之前看过一篇文章，叫做《浏览器的工作原理：新式网络浏览器幕后揭秘》

```
element.style {  
  padding: 10px;  
  border: 5px solid blue;  
  margin: 10px;  
  width: 300px;  
  box-sizing: border-box;  
}
```



上图中，为div设置了 `box-sizing: border-box` 之后，300px 的宽度是内容 + padding + 边框的宽度（不包括 margin），这样就比较符合我们的实际要求了。建议大家在为系统写 CSS 时候，第一个样式是：

```
* {  
  box-sizing: border-box;  
}
```

大名鼎鼎的 Bootstrap 也把 `box-sizing: border-box` 加入到它的 * 选择器中，我们为什么不这样做呢？

纵向 margin 重叠

这里提到 margin，就不得不提一下 margin 的这一特性——纵向重叠。如 `<p>` 的纵向 margin 是 16px，那么两个 `<p>` 之间纵向的距离是多少？——按常理来说应该是 $16 + 16 = 32\text{px}$ ，但是答案仍然是 16px。因为纵向的 margin 是会重叠的，如果两者不一样大的话，大的会把小的“吃掉”。

浮动 float

float 用于网页布局比较多，使用起来也比较简单，这里总结了一些比较重要、需要注意的知识点，供大家参考。

误解和误用

float 被设计出来的初衷是用于文字环绕效果，即一个图片一段文字，图片 `float: left` 之后，文字会环绕图片。

```
<div>
  
  一段文字一段文字一段文字一段文字一段文字一段文字一段文字一段文字一段文字
</div>
```

但是，后来大家发现结合 `float + div` 可以实现之前通过 `table` 实现的网页布局，因此就被“误用”于网页布局了。

题目：为何 `float` 会导致父元素塌陷？

破坏性

```
<div style='border:1px solid blue; padding:3px;'>
  
</div>
```



```
<div style='border:1px solid blue; padding:3px;'>
  
</div>
```



`float` 的**破坏性**——`float` 破坏了父标签的原本结构，使得父标签出现了坍塌现象。导致这一现象的最根本原因在于：**被设置了 `float` 的元素会脱离文档流**。其根本原因在于 `float` 的设计初衷是解决文字环绕图片的问题。大家要记住 `float` 的这个影响。

包裹性

包裹性也是 `float` 的一个非常重要的特性，大家用 `float` 时一定要熟知这一特性。咱们还是先从小例子看起：

没有 `float`

`float:left`

如上图，普通的 `div` 如果没有设置宽度，它会撑满整个屏幕，在之前的盒子模型那一节也讲到过。而如给 `div` 增加 `float:left` 之后，它突然变得紧凑了，宽度发生了变化，把内容中的三个字包裹了——这就是包裹性。为 `div` 设置了 `float` 之后，其宽度会自动调整为包裹住内容宽度，而不是撑满整个父容器。

注意，此时 `div` 虽然体现了包裹性，但是它的 `display` 样式是没有变化的，还是 `display: block`。

`float` 为什么要具有包裹性？其实答案还是得从 `float` 的设计初衷来寻找，`float` 是被设计用于实现文字环绕效果的。文字环绕图片比较好理解，但是如果想要让文字环绕一个 `div` 呢？此时 `div` 不被“包裹”起来的话，就无法实现环绕效果了。

清空格

`float` 还有一个大家可能不是很熟悉的特性——清空格。按照惯例，咱还是先举例子说明。

```
<div style="border: 2px solid blue; padding:3px;">
  
  
  
  
</div>
```



加上 `float:left` 之后:



上面第一张图中，正常的 `img` 中间是会有空格的，因为多个 `img` 标签会有换行，而浏览器识别换行为空格，这也是很正常的。第二张图中，为 `img` 增加了 `float:left` 的样式，这就使得 `img` 之间没有了空格，4 个 `img` 紧紧挨着。

如果大家之前没注意，现在想想之前写过的程序，是不是有这个特性。为什么 `float` 适合用于网页排版（俗称“砌砖头”）？就是因为 `float` 排版出来的网页严丝合缝，中间连个苍蝇都飞不进去。

“清空格”这一特性的根本原因是 `float` 会导致节点脱离文档流结构。它都不属于文档流结构了，那么它身边的什么换行、空格就都和它没了关系，它就尽量往一边靠拢，能靠多近就靠多近，这就是清空格的本质。

题目：手写 `clearfix`

clearfix

清除浮动的影响，一般使用的样式如下，统称 `clearfix` 代码。所有 `float` 元素的父容器，一般情况下都应该加 `clearfix` 这个 class。

```
.clearfix:after {
  content: '';
  display: table;
  clear: both;
}
.clearfix {
  *zoom: 1; /* 兼容 IE 低版本 */
}
```

```
<div class="clearfix">
  
  
</div>
```

小结

float 的设计初衷是解决文字环绕图片的问题，后来误打误撞用于做布局，因此有许多不合适或者需要注意的地方，上文基本都讲到了需要的知识点。如果是刚开始接触 float 的同学，学完上面的基础知识之后，还应该做一些练习实战一下 —— 经典的“圣杯布局”和“双飞翼布局”。这里就不再展开讲了，网上资料非常多，例如[浅谈面试中常考的两种经典布局——圣杯与双飞翼](#)（此文最后两张图清晰地展示了这两种布局）。

定位 position

position 用于网页元素的定位，可设置 static/relative/absolute/fixed 这些值，其中 static 是默认值，不用介绍。

题目：relative 和 absolute 有何区别？

relative

相对定位 relative 可以用一个例子很轻松地演示出来。例如我们写 4 个 `<p>`，出来的样子大家不用看也能知道。

```
<p>第一段文字</p>
<p>第二段文字</p>
<p>第三段文字</p>
<p>第四段文字</p>
```

第一段文字

第二段文字

第三段文字

第四段文字

然后我们在第三个 `<p>` 上面，加上 `position: relative` 并且设置 `left` 和 `top` 值，看这个 `<p>` 有什么变化。

```
<p>第一段文字</p>
<p>第二段文字</p>
<p style="position: relative; top: 10px; left: 10px">第三段文字</p>
<p>第四段文字</p>
```

第一段文字

第二段文字

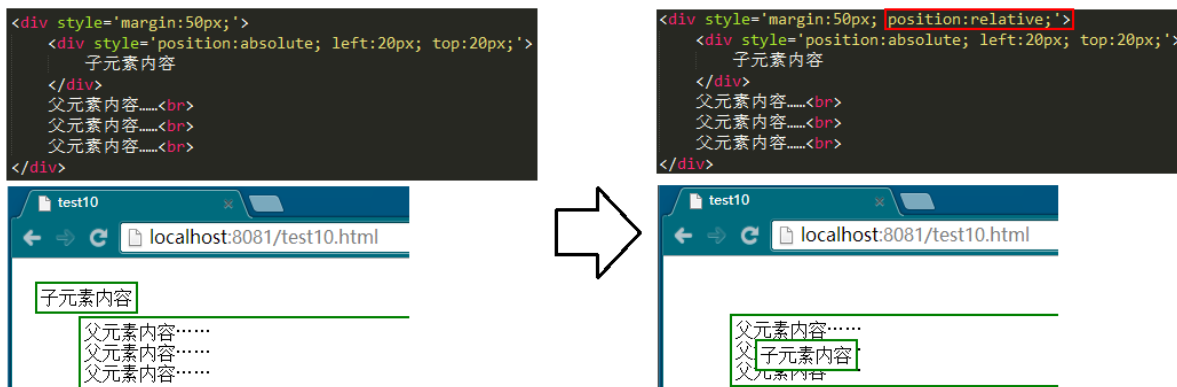
第三段文字

第四段文字

上图中，大家应该要识别出两个信息（相信大部分人会忽略第二个信息）

- 第三个 `<p>` 发生了位置变化，分别向右向下移动了10px；
- 其他的三个 `<p>` 位置没有发生变化，这一点也很重要。

可见，**relative 会导致自身位置的相对变化，而不会影响其他元素的位置、大小**。这是 relative 的要点之一。还有第二个要点，就是 relative 产生一个新的定位上下文。下文有关于定位上下文的详细介绍，这里可以先通过一个例子来展示一下区别：



注意看这两图的区别，下文将有解释。

absolute

还是先写一个基本的 demo。

```
<p>第一段文字</p>
<p>第二段文字</p>
<p style="background: yellow">第三段文字</p>
<p>第四段文字</p>
```

第一段文字

第二段文字

第三段文字

第四段文字

然后，我们把第三个 `<p>` 改为 `position: absolute;`，看看会发生什么变化。

第一段文字

第二段文字

第四段文字

第三段文字

从上面的结果中，我们能看出几点信息：

- absolute 元素脱离了文档结构。和 relative 不同，其他三个元素的位置重新排列了。只要元素会脱离文档结构，它就会产生破坏性，导致父元素坍塌。（此时你应该能立刻想起来，float 元素也会脱离文档结构。）

- absolute 元素具有“包裹性”。之前 `<p>` 的宽度是撑满整个屏幕的，而此时 `<p>` 的宽度刚好是内容的宽度。
- absolute 元素具有“跟随性”。虽然 absolute 元素脱离了文档结构，但是它的位置并没有发生变化，还是老老实实在它原本的位置，因为我们此时没有设置 top、left 的值。
- absolute 元素会悬浮在页面上方，会遮挡住下方的页面内容。

最后，通过给 absolute 元素设置 top、left 值，可自定义其内容，这个都是平时比较常用的了。这里需要注意的是，设置了 top、left 值时，元素是相对于最近的定位上下文来定位的，而不是相对于浏览器定位。

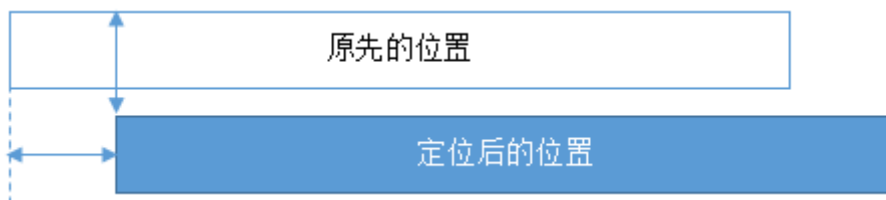
fixed

其实 fixed 和 absolute 是一样的，唯一的区别在于：absolute 元素是根据最近的定位上下文确定位置，而 fixed 根据 window（或者 iframe）确定位置。

题目：relative、absolute 和 fixed 分别依据谁来定位？

定位上下文

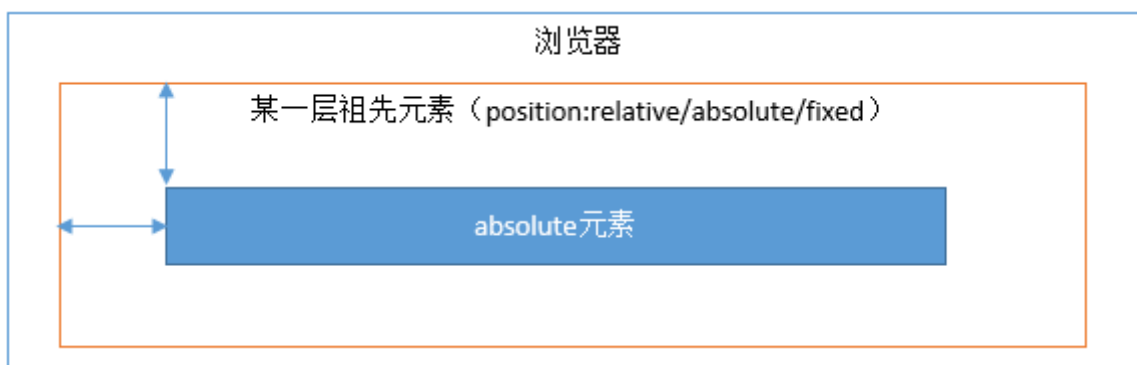
relative 元素的定位永远是相对于元素自身位置的，和其他元素没关系，也不会影响其他元素。



fixed 元素的定位是相对于 window（或者 iframe）边界的，和其他元素没有关系。但是它具有破坏性，会导致其他元素位置的变化。



absolute 的定位相对于前两者要复杂许多。如果为 absolute 设置了 top、left，浏览器会根据什么去确定它的纵向和横向的偏移量呢？答案是浏览器会递归查找该元素的所有父元素，如果找到一个设置了 `position:relative/absolute/fixed` 的元素，就以该元素为基准定位，如果没找到，就以浏览器边界定位。如下两个图所示：





flex布局

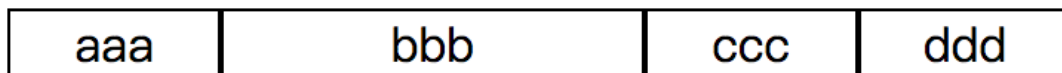
布局的传统解决方案基于盒子模型，依赖 `display` 属性 + `position` 属性 + `float` 属性。它对于那些特殊布局非常不方便，比如，垂直居中（下文会专门讲解）就不容易实现。在目前主流的移动端页面中，使用 flex 布局能更好地完成需求，因此 flex 布局的知识是必须要掌握的。

基本使用

任何一个容器都可以使用 flex 布局，代码也很简单。

```
<style type="text/css">
  .container {
    display: flex;
  }
  .item {
    border: 1px solid #000;
    flex: 1;
  }
</style>

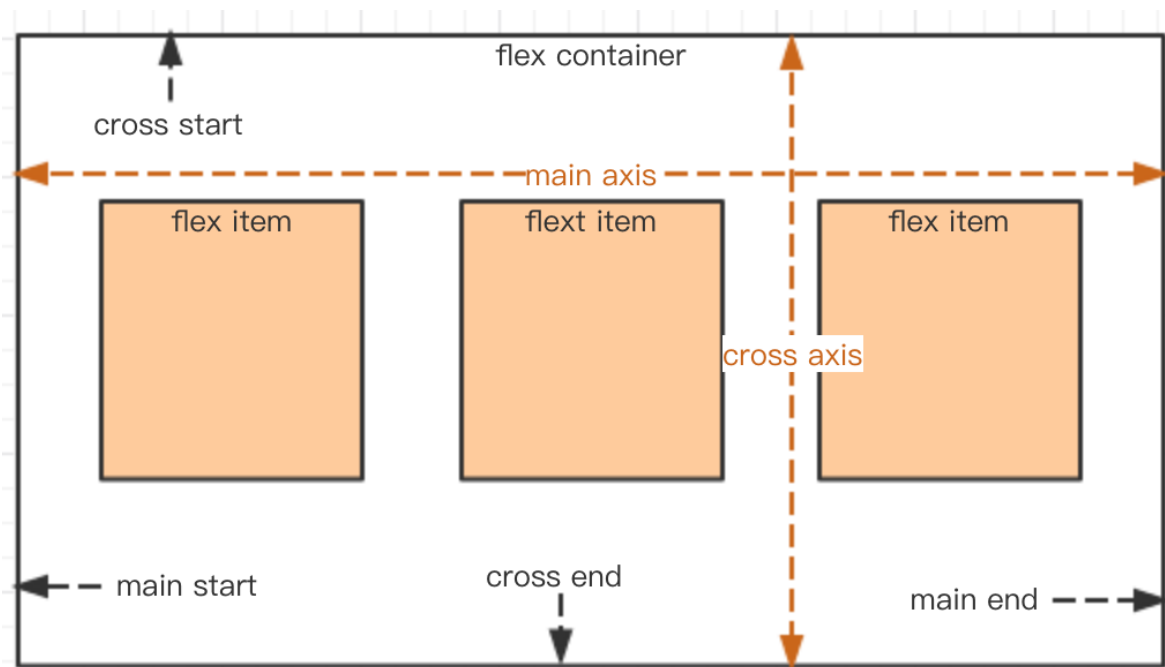
<div class="container">
  <div class="item">aaa</div>
  <div class="item" style="flex: 2">bbb</div>
  <div class="item">ccc</div>
  <div class="item">ddd</div>
</div>
```



注意，第三个 `<div>` 的 `flex: 2`，其他的 `<div>` 的 `flex: 1`，这样第二个 `<div>` 的宽度就是其他的 `<div>` 的两倍。

设计原理

设置了 `display: flex` 的元素，我们称为“容器”（flex container），其所有的子节点我们称为“成员”（flex item）。容器默认存在两根轴：水平的主轴（main axis）和垂直的交叉轴（cross axis）。主轴的开始位置（与边框的交叉点）叫做 main start，结束位置叫做 main end；交叉轴的开始位置叫做 cross start，结束位置叫做 cross end。项目默认沿主轴排列。单个项目占据的主轴空间叫做 main size，占据的交叉轴空间叫做 cross size。



将以上文字和图片结合起来，再详细看一遍，这样就能理解 flex 的设计原理，才能更好地实际使用。

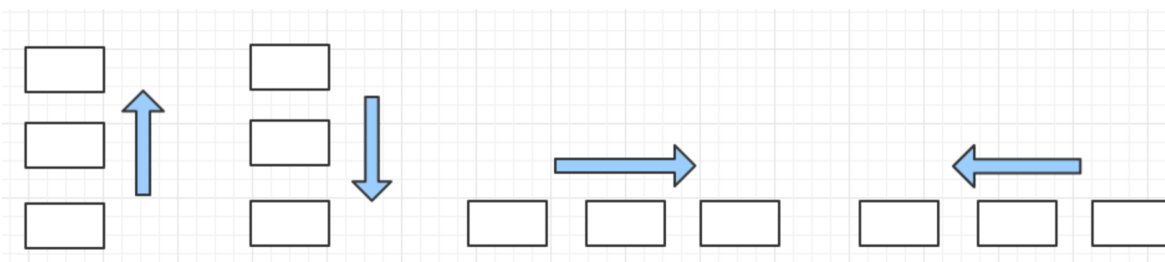
设置主轴的方向

`flex-direction` 可决定主轴的方向，有四个可选值：

- `row`（默认值）：主轴为水平方向，起点在左端。
- `row-reverse`：主轴为水平方向，起点在右端。
- `column`：主轴为垂直方向，起点在上沿。
- `column-reverse`：主轴为垂直方向，起点在下沿。

```
.box {
  flex-direction: column-reverse | column | row | row-reverse;
}
```

以上代码设置的主轴方向，将依次对应下图：

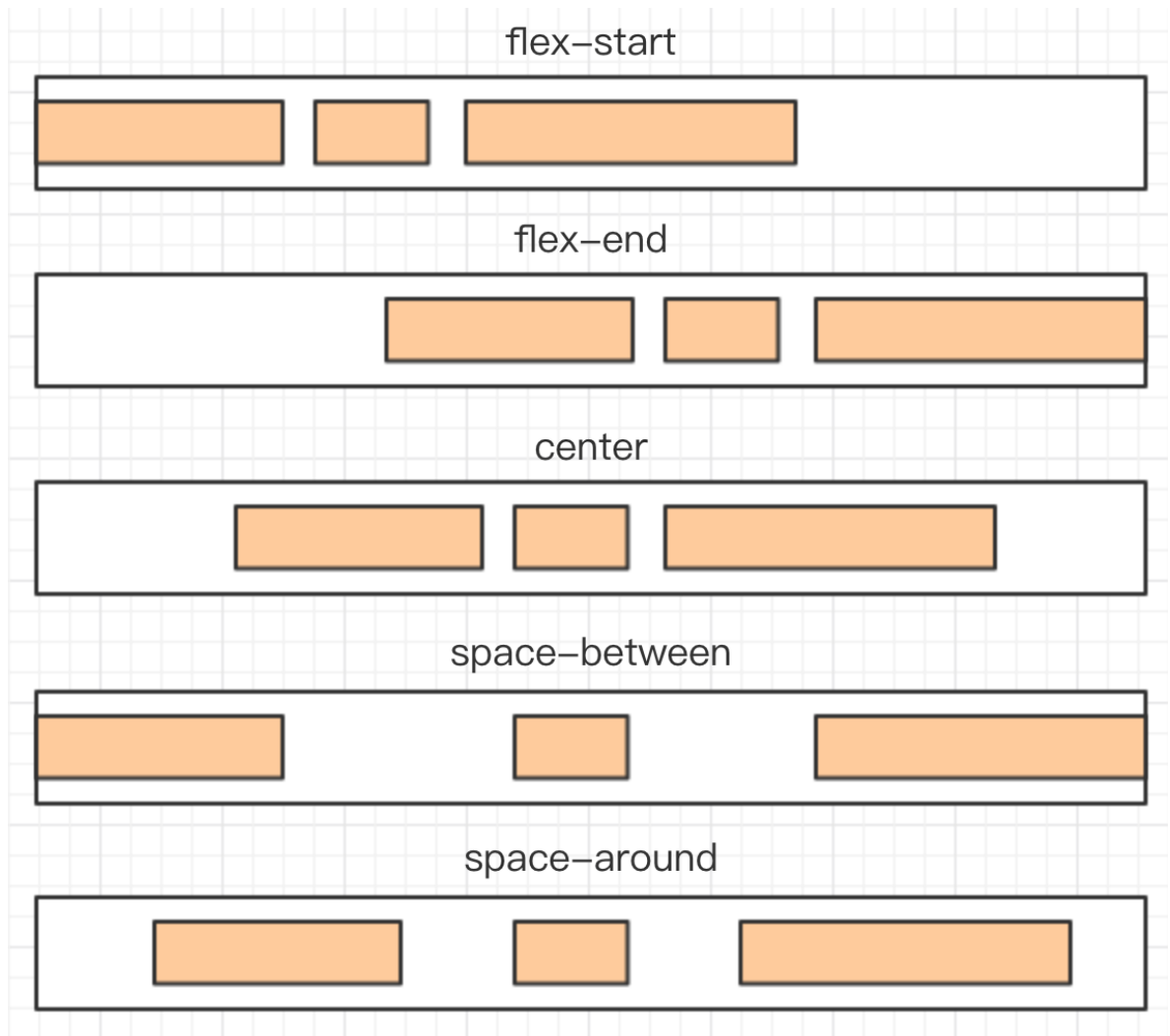


设置主轴的对齐方式

`justify-content` 属性定义了项目在主轴上的对齐方式，值如下：

- `flex-start`（默认值）：向主轴开始方向对齐。
- `flex-end`：向主轴结束方向对齐。
- `center`：居中。
- `space-between`：两端对齐，项目之间的间隔都相等。
- `space-around`：每个项目两侧的间隔相等。所以，项目之间的间隔比项目与边框的间隔大一倍。

```
.box {  
  justify-content: flex-start | flex-end | center | space-between | space-around;  
}
```

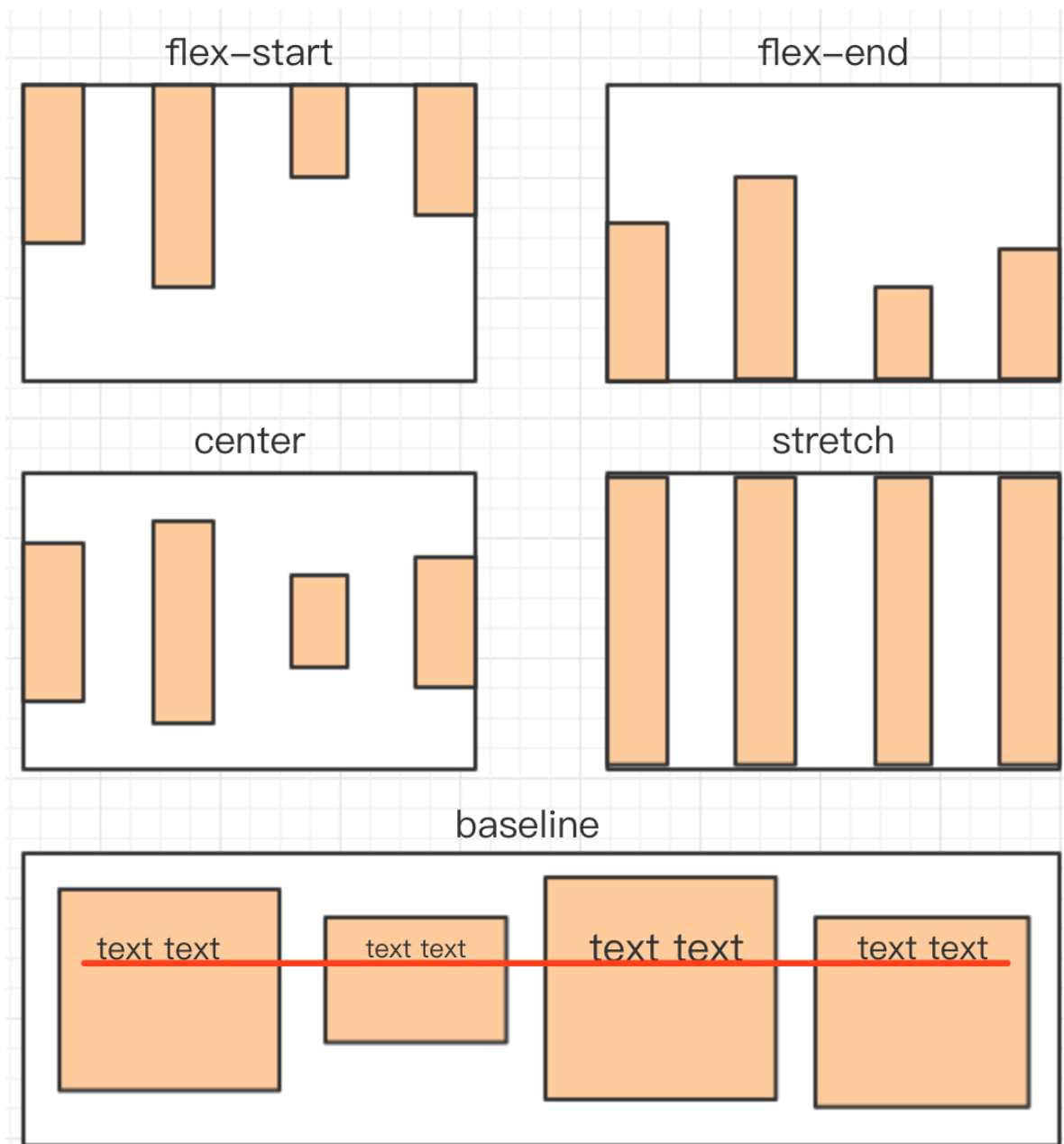


交叉轴的对齐方式

`align-items` 属性定义项目在交叉轴上如何对齐，值如下：

- `flex-start`：交叉轴的起点对齐。
- `flex-end`：交叉轴的终点对齐。
- `center`：交叉轴的中点对齐。
- `baseline`：项目的第一行文字的基线对齐。
- `stretch`（默认值）：如果项目未设置高度或设为 `auto`，将占满整个容器的高度。

```
.box {  
  align-items: flex-start | flex-end | center | baseline | stretch;  
}
```



如何实现居中对齐？

题目：如何实现水平居中？

水平居中

inline 元素用 `text-align: center;` 即可，如下：

```
.container {  
  text-align: center;  
}
```

block 元素可使用 `margin: auto;`，PC 时代的很多网站都这么搞。

```
.container {
  text-align: center;
}
.item {
  width: 1000px;
  margin: auto;
}
```

绝对定位元素可结合 `left` 和 `margin` 实现，但是必须知道宽度。

```
.container {
  position: relative;
  width: 500px;
}
.item {
  width: 300px;
  height: 100px;
  position: absolute;
  left: 50%;
  margin: -150px;
}
```

题目：如何实现垂直居中？

垂直居中

`inline` 元素可设置 `line-height` 的值等于 `height` 值，如单行文字垂直居中：

```
.container {
  height: 50px;
  line-height: 50px;
}
```

绝对定位元素，可结合 `left` 和 `margin` 实现，但是必须知道尺寸。

- 优点：兼容性好
- 缺点：需要提前知道尺寸

```
.container {
  position: relative;
  height: 200px;
}
.item {
  width: 80px;
  height: 40px;
  position: absolute;
  left: 50%;
  top: 50%;
  margin-top: -20px;
}
```

```
margin-left: -40px;
}
```

绝对定位可结合 `transform` 实现居中。

- 优点：不需要提前知道尺寸
- 缺点：兼容性不好

```
.container {
  position: relative;
  height: 200px;
}
.item {
  width: 80px;
  height: 40px;
  position: absolute;
  left: 50%;
  top: 50%;
  transform: translate(-50%, -50%);
  background: blue;
}
```

绝对定位结合 `margin: auto`，不需要提前知道尺寸，兼容性好。

```
.container {
  position: relative;
  height: 300px;
}
.item {
  width: 100px;
  height: 50px;
  position: absolute;
  left: 0;
  top: 0;
  right: 0;
  bottom: 0;
  margin: auto;
}
```

其他的解决方案还有，不过没必要掌握太多，能说出上文的这几个解决方案即可。

理解语义化

题目：如何理解 HTML 语义化？

所谓“语义”就是为了更易读懂，这要分两部分：

- 让人（写程序、读程序）更易读懂
- 让机器（浏览器、搜索引擎）更易读懂

让人更易读懂

对于人来说，代码可读性、语义化就是一个非常广泛的概念了，例如定义 JS 变量的时候使用更易读懂的名称，定义 CSS class 的时候也一样，例如 `length` `list` 等，而不是使用 `a` `b` 这种谁都看不懂的名称。

不过我们平常考查的“语义化”并不会考查这么广义、这么泛的问题，而是考查 HTML 的语义化，是为了更好地让机器读懂 HTML。

让机器更易读懂

HTML 符合 XML 标准，但又和 XML 不一样——HTML 不允许像 XML 那样自定义标签名称，HTML 有自己规定的标签名称。问题就在这里——HTML 为何要自己规定那么多标签名称呢，例如 `p` `div` `h1` `ul` 等——就是为了语义化。其实，如果你精通 CSS 的话，你完全可以全部用 `<div>` 标签来实现所有的网页效果，其他的 `p` `h1` `ul` 等标签可以一个都不用。但是我们不推荐这么做，这样做就失去了 HTML 语义化的意义。

拿搜索引擎来说，爬虫下载到我们的网页的 HTML 代码，它如何更好地去理解网页的内容呢？——就是根据 HTML 既定的标签。`h1` 标签就代表是标题；`p` 里面的就是段落详细内容，权重肯定没有标题高；`ul` 里面就是列表；`strong` 就是加粗的强调的内容……如果我们不按照 HTML 语义化来写，全部都用 `<div>` 标签，那搜索引擎将很难理解我们网页的内容。

为了加强 HTML 语义化，HTML5 标准中又增加了 `header` `section` `article` 等标签。因此，书写 HTML 时，语义化是非常重要的，否则 W3C 也没必要辛辛苦苦制定出这些标准来。

CSS3 动画

CSS3 可以实现动画，代替原来的 Flash 和 JavaScript 方案。

首先，使用 `@keyframes` 定义一个动画，名称为 `testAnimation`，如下代码，通过百分比来设置不同的 CSS 样式，规定动画的变化。所有的动画变化都可以这么定义出来。

```
@keyframes testAnimation
{
  0%    {background: red; left:0; top:0;}
  25%   {background: yellow; left:200px; top:0;}
  50%   {background: blue; left:200px; top:200px;}
  75%   {background: green; left:0; top:200px;}
  100%  {background: red; left:0; top:0;}
}
```

然后，针对一个 CSS 选择器来设置动画，例如针对 `div` 元素设置动画，如下：

```
div {  
  width: 100px;  
  height: 50px;  
  position: absolute;  
  
  animation-name: myfirst;  
  animation-duration: 5s;  
}
```

`animation-name` 对应到动画名称, `animation-duration` 是动画时长, 还有其他属性:

- `animation-timing-function`: 规定动画的速度曲线。默认是 `ease`
- `animation-delay`: 规定动画何时开始。默认是 0
- `animation-iteration-count`: 规定动画被播放的次数。默认是 1
- `animation-direction`: 规定动画是否在下一周期逆向地播放。默认是 `normal`
- `animation-play-state`: 规定动画是否正在运行或暂停。默认是 `running`
- `animation-fill-mode`: 规定动画执行之前和之后如何给动画的目标应用, 默认是 `none`, 保留在最后一帧可以用 `forwards`

题目: CSS 的 `transition` 和 `animation` 有何区别?

首先 `transition` 和 `animation` 都可以做动效, 从语义上来理解, `transition` 是过渡, 由一个状态过渡到另一个状态, 比如高度 100px 过渡到 200px; 而 `animation` 是动画, 即更专业做动效的, `animation` 有帧的概念, 可以设置关键帧 `keyframe`, 一个动画可以由多个关键帧多个状态过渡组成, 另外 `animation` 也包含上面提到的多个属性。

重绘和回流

重绘和回流是面试题经常考的题目, 也是性能优化当中应该注意的点, 下面笔者简单介绍下。

- **重绘**: 指的是当页面中的元素不脱离文档流, 而简单地进行样式的变化, 比如修改颜色、背景等, 浏览器重新绘制样式
- **回流**: 指的是处于文档流中 DOM 的尺寸大小、位置或者某些属性发生变化时, 导致浏览器重新渲染部分或全部文档的情况

相比之下, **回流要比重绘消耗性能开支更大**。另外, 一些属性的读取也会引起回流, 比如读取某个 DOM 的高度和宽度, 或者使用 `getComputedStyle` 方法。在写代码的时候要避免回流和重绘。比如在笔试中可能会遇见下面的题目:

题目: 找出下面代码的优化点, 并且优化它

```
var data = ['string1', 'string2', 'string3'];  
for(var i = 0; i < data.length; i++){  
  var dom = document.getElementById('list');  
  dom.innerHTML += '<li>' + data[i] + '</li>';  
}
```

上面的代码在循环中每次都获取 `dom`, 然后对其内部的 HTML 进行累加 `li`, 每次都会操作 DOM 结构, 可以改成使用 `documentFragment` 或者先遍历组成 HTML 的字符串, 最后操作一次 `innerHTML`。

小结

本小节总结了 CSS 和 HTML 常考的知识点，包括 CSS 中比较重要的定位、布局的知识，也介绍了一些 CSS3 的知识点概念和题目，以及 HTML 的语义化。

一面 4：从容应对算法题目

由冯·诺依曼机组成我们知道：数据存储和运算是计算机工作的主要内容。`程序=数据结构+算法`，所以计算机类工程师必须掌握一定的数据结构和算法知识。

知识点梳理

- 常见的数据结构
 - 栈、队列、链表
 - 集合、字典、散列集
- 常见算法
 - 递归
 - 排序
 - 枚举
- 算法复杂度分析
- 算法思维
 - 分治
 - 贪心
 - 动态规划
- 高级数据结构
 - 树、图
 - 深度优先和广度优先搜索

本小节会带领大家快速过一遍数据结构和算法，重点讲解一些常考、前端会用到的算法和数据结构。

数据结构

数据结构决定了数据存储的空间和时间效率问题，数据的写入和提取速度要求也决定了应该选择怎样的数据结构。

根据对场景需求的不同，我们设计不同的数据结构，比如：

- 读得多的数据结构，应该想办法提高数据的读取效率，比如 IP 数据库，只需要写一次，剩下的都是读取；
- 读写都多的数据结构，要兼顾两者的需求平衡，比如 LRU Cache 算法。

算法是数据加工处理的方式，一定的算法会提升数据的处理效率。比如有序数组的二分查找，要比普通的顺序查找快很多，尤其是在处理大量数据的时候。

数据结构和算法是程序开发的通用技能，所以在任何面试中都可能会遇见。随着近几年 AI、大数据、小游戏越来越火，Web 前端职位难免会跟数据结构和算法打交道，面试中也会出现越来越多的算法题目。学习数据结构和算法也能够帮助我们打开思路，突破技能瓶颈。

前端常遇见的数据结构问题

现在我来梳理下前端常遇见的数据结构：

- 简单数据结构（必须理解掌握）
 - 有序数据结构：栈、队列、链表，有序数据结构省空间（存储空间小）
 - 无序数据结构：集合、字典、散列表，无序数据结构省时间（读取时间快）
- 复杂数据结构
 - 树、堆
 - 图

对于简单数据结构，在 ES 中对应的是数组（Array）和对象（Object）。可以想一下，数组的存储是有序的，对象的存储是无序的，但是我要在对象中根据 key 找到一个值是立即返回的，数组则需要查找的过程。

这里我通过一个真实面试题目来说明介绍下数据结构设计。

题目：使用 ECMAScript (JS) 代码实现一个事件类 `Event`，包含下面功能：绑定事件、解绑事件和派发事件。

在稍微复杂点的页面中，比如组件化开发的页面，同一个页面由两三个人来开发，为了保证组件的独立性和降低组件间耦合度，我们往往使用「订阅发布模式」，即组件间通信使用事件监听和派发的方式，而不是直接相互调用组件方法，这就是题目要求写的 `Event` 类。

这个题目的核心是一个事件类型对应回调函数的数据设计。为了实现绑定事件，我们需要一个 `_cache` 对象来记录绑定了哪些事件。而事件发生的时候，我们需要从 `_cache` 中读取出来事件回调，依次执行它们。一般页面中事件派发（读）要比事件绑定（写）多。所以我们设计的数据结构应该尽量地能够在事件发生时，更加快速地找到对应事件的回调函数们，然后执行。

经过这样一番考虑，我简单写了下代码实现：

```
class Event {
  constructor() {
    // 存储事件的数据结构
    // 为了查找迅速，使用了对象（字典）
    this._cache = {};
  }
  // 绑定
  on(type, callback) {
    // 为了按类查找方便和节省空间，
    // 将同一类型事件放到一个数组中
    // 这里的数组是队列，遵循先进先出
    // 即先绑定的事件先触发
    let fns = (this._cache[type] = this._cache[type] || []);
    if (fns.indexOf(callback) === -1) {
      fns.push(callback);
    }
    return this;
  }
  // 触发
  trigger(type, data) {
    let fns = this._cache[type];
    if (Array.isArray(fns)) {
      fns.forEach((fn) => {
        fn(data);
      });
    }
    return this;
  }
}
```

```

// 解绑
off(type, callback) {
  let fns = this._cache[type];
  if (Array.isArray(fns)) {
    if (callback) {
      let index = fns.indexOf(callback);
      if (index !== -1) {
        fns.splice(index, 1);
      }
    } else {
      //全部清空
      fns.length = 0;
    }
  }
  return this;
}
}

// 测试用例
const event = new Event();
event.on('test', (a) => {
  console.log(a);
});
event.trigger('test', 'hello world');

event.off('test');
event.trigger('test', 'hello world');

```

类似于树、堆、图这些高级数据结构，前端一般也不会考查太多，但是它们的查找方法却常考，后面介绍。高级数据应该平时多积累，好好理解，比如理解了堆是什么样的数据结构，在面试中遇见的「查找最大的 K 个数」这类算法问题，就会迎刃而解。

算法的效率是通过算法复杂度来衡量的

算法的好坏可以通过算法复杂度来衡量，算法复杂度包括时间复杂度和空间复杂度两个。时间复杂度由于好估算、好评估等特点，是面试中考查的重点。空间复杂度在面试中考查得不多。

常见的时间复杂度有：

- 常数阶 $O(1)$
- 对数阶 $O(\log N)$
- 线性阶 $O(n)$
- 线性对数阶 $O(n \log N)$
- 平方阶 $O(n^2)$
- 立方阶 $O(n^3)$
- k 次方阶 $O(n^k)$
- 指数阶 $O(2^n)$

随着问题规模 n 的不断增大，上述时间复杂度不断增大，算法的执行效率越低。

一般做算法复杂度分析的时候，遵循下面的技巧：

1. 看看有几重循环，一般来说一重就是 $O(n)$ ，两重就是 $O(n^2)$ ，以此类推
2. 如果有二分，则为 $O(\log N)$
3. 保留最高项，去除常数项

题目：分析下面代码的算法复杂度（为了方便，我已经在注释中加了代码分析）

```
let i = 0; // 语句执行一次
while (i < n) { // 语句执行 n 次
  console.log(`Current i is ${i}`); // 语句执行 n 次
  i++; // 语句执行 n 次
}
```

根据注释可以得到，算法复杂度为 $1 + n + n + n = 1 + 3n$ ，去除常数项，为 $O(n)$ 。

```
let number = 1; // 语句执行一次
while (number < n) { // 语句执行 logN 次
  number *= 2; // 语句执行 logN 次
}
```

上面代码 while 的跳出判断条件是 $number < n$ ，而循环体内 number 增长速度是 (2^n) ，所以循环代码实际执行 $\log N$ 次，复杂度为： $1 + 2 * \log N = O(\log N)$

```
for (let i = 0; i < n; i++) { // 语句执行 n 次
  for (let j = 0; j < n; j++) { // 语句执行 n^2 次
    console.log('I am here!'); // 语句执行 n^2 次
  }
}
```

上面代码是两个 for 循环嵌套，很容易得出复杂度为： $O(n^2)$

人人都要掌握的基础算法

枚举和递归是最最简单的算法，也是复杂算法的基础，人人都应该掌握！枚举相对比较简单，我们重点说下递归。

递归由下面两部分组成：

1. 递归主体，就是要循环解决问题的代码
2. 递归的跳出条件，递归不能一直递归下去，需要完成一定条件后跳出

关于递归有个经典的面试题目是：

实现 JS 对象的深拷贝

什么是深拷贝？

「深拷贝」就是在拷贝数据的时候，将数据的所有引用结构都拷贝一份。简单的说就是，在内存中存在两个数据结构完全相同又相互独立的数据，将引用型类型进行复制，而不是只复制其引用关系。

分析下怎么做「深拷贝」：

1. 首先假设深拷贝这个方法已经完成，为 deepClone
2. 要拷贝一个数据，我们肯定要去遍历它的属性，如果这个对象的属性仍是对象，继续使用这个方法，如此往复

```
function deepClone(o1, o2) {
  for (let k in o2) {
    if (typeof o2[k] === 'object') {
      o1[k] = {};
    }
  }
}
```

```

        deepClone(o1[k], o2[k]);
    } else {
        o1[k] = o2[k];
    }
}
}
// 测试用例
let obj = {
    a: 1,
    b: [1, 2, 3],
    c: {}
};
let emptyObj = Object.create(null);
deepClone(emptyObj, obj);
console.log(emptyObj.a == obj.a);
console.log(emptyObj.b == obj.b);

```

递归容易造成爆栈，尾部调用可以解决递归的这个问题，Chrome 的 V8 引擎做了尾部调用优化，我们在写代码的时候也要注意尾部调用写法。递归的爆栈问题可以通过将递归改写成枚举的方式来解决，就是通过 `for` 或者 `while` 来代替递归。

我们在使用递归的时候，要注意做优化，比如下面的题目。

题目：求斐波那契数列（兔子数列）1,1,2,3,5,8,13,21,34,55,89...中的第 n 项

下面的代码中 `count` 记录递归的次数，我们看下两种差异性的代码中的 `count` 的值：

```

let count = 0;
function fn(n) {
    let cache = {};
    function _fn(n) {
        if (cache[n]) {
            return cache[n];
        }
        count++;
        if (n == 1 || n == 2) {
            return 1;
        }
        let prev = _fn(n - 1);
        cache[n - 1] = prev;
        let next = _fn(n - 2);
        cache[n - 2] = next;
        return prev + next;
    }
    return _fn(n);
}

let count2 = 0;
function fn2(n) {
    count2++;
    if (n == 1 || n == 2) {
        return 1;
    }
    return fn2(n - 1) + fn2(n - 2);
}

console.log(fn(20), count); // 6765 20

```

```
console.log(fn2(20), count2); // 6765 13529
```

快排和二分查找

前端中面试排序和查找的可能性比较小，因为 JS 引擎已经把这些常用操作优化得很好了，可能项目中你费劲写的一个排序方法，都不如 `Array.sort` 速度快且代码少。因此，掌握快排和二分查找就可以了。

快排和二分查找都基于一种叫做「分治」的算法思想，通过对数据进行分类处理，不断降低数量级，实现 $O(\log N)$ （对数级别，比 $O(n)$ 这种线性复杂度更低的一种，快排核心是二分法的 $O(\log N)$ ，实际复杂度为 $O(N * \log N)$ ）的复杂度。

快速排序

快排大概的流程是：

1. 随机选择数组中的一个数 A，以这个数为基准
2. 其他数字跟这个数进行比较，比这个数小的放在其左边，大的放到其右边
3. 经过一次循环之后，A 左边为小于 A 的，右边为大于 A 的
4. 这时候将左边和右边的数再递归上面的过程

具体代码如下：

```
// 划分操作函数
function partition(array, left, right) {
  // 用index取中间值而非splice
  const pivot = array[Math.floor((right + left) / 2)]
  let i = left
  let j = right

  while (i <= j) {
    while (compare(array[i], pivot) === -1) {
      i++
    }
    while (compare(array[j], pivot) === 1) {
      j--
    }
    if (i <= j) {
      swap(array, i, j)
      i++
      j--
    }
  }
  return i
}

// 比较函数
function compare(a, b) {
  if (a === b) {
    return 0
  }
  return a < b ? -1 : 1
}

function quick(array, left, right) {
```

```

    let index
    if (array.length > 1) {
        index = partition(array, left, right)
        if (left < index - 1) {
            quick(array, left, index - 1)
        }
        if (index < right) {
            quick(array, index, right)
        }
    }
    return array
}
function quickSort(array) {
    return quick(array, 0, array.length - 1)
}

// 原地交换函数，而非用临时数组
function swap(array, a, b) {
    ;[array[a], array[b]] = [array[b], array[a]]
}
const Arr = [85, 24, 63, 45, 17, 31, 96, 50];
console.log(quickSort(Arr));
// 本版本来自: https://juejin.im/post/5af4902a6fb9a07abf728c40#heading-12

```

二分查找

二分查找法主要是解决「在一堆有序的数中找出指定的数」这类问题，不管这些数是一维数组还是多维数组，只要有序，就可以用二分查找来优化。

二分查找是一种「分治」思想的算法，大概流程如下：

1. 数组中排在中间的数字 A，与要找的数字比较大小
2. 因为数组是有序的，所以： a) A 较大则说明要查找的数字应该从前半部分查找 b) A 较小则说明应该从查找数字的后半部分查找
3. 这样不断查找缩小数量级（扔掉一半数据），直到找完数组为止

题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

```

function Find(target, array) {
    let i = 0;
    let j = array[i].length - 1;
    while (i < array.length && j >= 0) {
        if (array[i][j] < target) {
            i++;
        } else if (array[i][j] > target) {
            j--;
        } else {
            return true;
        }
    }
    return false;
}

//测试用例

```

```
console.log(Find(10, [
  [1, 2, 3, 4],
  [5, 9, 10, 11],
  [13, 20, 21, 23]
]));
```

另外笔者在面试中遇见过下面的问题：

题目：现在我有一个 1~1000 区间中的正整数，需要你猜下这个数字是几，你只能问一个问题：大了还是小了？问需要猜几次才能猜对？

拿到这个题目，笔者想到的就是电视上面有个「猜价格」的购物节目，在规定时间内猜对价格就可以把实物抱回家。所以问题就是让面试官不停地回答我猜的数字比这个数字大了还是小了。这就是二分查找！

猜几次呢？其实这个问题就是个二分查找的算法时间复杂度问题，二分查找的时间复杂度是 $O(\log N)$ ，所以求 $\log 1000$ 的解就是猜的次数。我们知道 $2^{10}=1024$ ，所以可以快速估算出： $\log 1000$ 约等于 10，最多问 10 次就能得到这个数！

面试遇见不会的算法问题怎么办

面试的时候，在遇见算法题目的时候，应该揣摩面试官的意图，听好关键词，比如：有序的数列做查找、要求算法复杂度是 $O(\log N)$ 这类一般就是用二分的思想。

一般来说算法题目的解题思路分以下四步：

1. 先降低数量级，拿可以计算出来的情况（数据）来构思解题步骤
2. 根据解题步骤编写程序，优先将特殊情况做好判断处理，比如一个大数组的问题，如果数组为两个数长度的情况
3. 检验程序正确性
4. 是否可以优化（由浅到深），有能力的话可以故意预留优化点，这样可以体现个人技术能力

正则匹配解题

很多算法题目利用 ES 语法的特性来回答更加简单，比如正则匹配就是常用的一种方式。笔者简单通过几个真题来汇总下正则的知识点。

题目：字符串中第一个出现一次的字符

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符「go」时，第一个只出现一次的字符是「g」。当从该字符流中读出前六个字符「google」时，第一个只出现一次的字符是「l」。

这个如果用纯算法来解答需要遍历字符串，统计每个字符出现的次数，然后按照字符串的顺序来找出第一次出现一次的字符，整个过程比较繁琐，如果用正则就简单多了。


```
function find(str){
  for (var i = 0; i < str.length; i++) {
    let char = str[i]
    let reg = new RegExp(char, 'g');
    let l = str.match(reg).length
    if(l===1){
      return char
    }
  }
}
```

当然，使用 `indexOf/lastIndexOf` 也是一个取巧的方式。再来看一个千分位问题。

题目：将 `1234567` 变成 `1,234,567`，即千分位标注

这个题目可以用算法直接来解，如果候选人使用正则来回答，这样主动展现了自己其他方面的优势，即使不是算法解答出来的，面试官一般也不会太难为他。这道题目可以利用正则的「零宽断言」(`?=exp`)，意思是它断言自身出现的位置的后面能匹配表达式 `exp`。数字千分位的特点是，第一个逗号后面数字的个数是3的倍数，正则：`/(\d{3})+$/`；第一个逗号前最多可以有 1~3 个数字，正则：`/\d{1,3}/`。加起来就是 `/\d{1,3}(\d{3})+$/`，分隔符要从前往后加。

对于零宽断言的详细介绍可以阅读「[零宽断言](#)」这篇文章。

```
function exchange(num) {
  num += ''; //转成字符串
  if (num.length <= 3) {
    return num;
  }

  num = num.replace(/(\d{1,3})(?=(\d{3})+)/g, (v) => {
    console.log(v)
    return v + ',';
  });
  return num;
}

console.log(exchange(1234567));
```

当然上面讲到的多数是算法题目取巧的方式，下面这个题目是纯正则考查，笔者在面试的过程中碰见过，这里顺便提一下。

题目，请写出下面的代码执行结果

```
var str = 'google';
var reg = /o/g;
console.log(reg.test(str))
console.log(reg.test(str))
console.log(reg.test(str))
```

代码执行后，会发现，最后一个不是为 `true`，而是 `false`，这是因为 `reg` 这个正则有个 `g`，即 `global` 全局的属性，这种情况下 `lastIndex` 就发挥作用了，可以看下面的代码执行结果就明白了。

```
console.log(reg.test(str), reg.lastIndex)
console.log(reg.test(str), reg.lastIndex)
console.log(reg.test(str), reg.lastIndex)
```

实际开发中也会犯这样的错误，比如为了减少变量每次都重新定义，会把用到的变量提前定义好，这样在使用的时候容易掉进坑里，比如下面代码：

```
(function(){
  const reg = /o/g;
  function isHasO(str){
    // reg.lastIndex = 0; 这样就可以避免这种情况
    return reg.test(str)
  }
  var str = 'google';
  console.log(isHasO(str))
  console.log(isHasO(str))
  console.log(isHasO(str))
})();
```

小结

本小节介绍了数据结构和算法的关系，作为普通的前端也应该学习数据结构和算法知识，并且顺带介绍了下正则匹配。具体来说，本小节梳理了以下几部分数据结构和算法知识点：

1. 经常用到的数据结构有哪些，它们的特点有哪些
2. 递归和枚举是最基础的算法，必须牢牢掌握
3. 排序里面理解并掌握快速排序算法，其他排序算法可以根据个人实际情况大概了解
4. 有序查找用二分查找
5. 遇见不会的算法问题，先缩小数量级，然后分析推导

当然算法部分还有很多知识，比如动态规划这些算法思想，还有图和树常用到的广度优先搜索和深度优先搜索。这些知识在前端面试和项目中遇见得不多，感兴趣的读者可以在梳理知识点的时候根据个人情况自行决定是否复习。

一面 5：浏览器相关知识点与高频考题解析

Web 前端工程师写的页面要跑在浏览器里面，所以面试中也会出现很多跟浏览器相关的面试题目。

知识点梳理

- 浏览器加载页面和渲染过程
- 性能优化
- Web 安全

本小节会从浏览器的加载过程开始讲解，然后介绍如何进行性能优化，最后介绍下 Web 开发中常见的安全问题和预防。

加载页面和渲染过程

可将加载过程和渲染过程分开说。回答问题的时候，关键要抓住核心的要点，把要点说全面，稍加解析即可，简明扼要不拖沓。

题目：浏览器从加载页面到渲染页面的过程

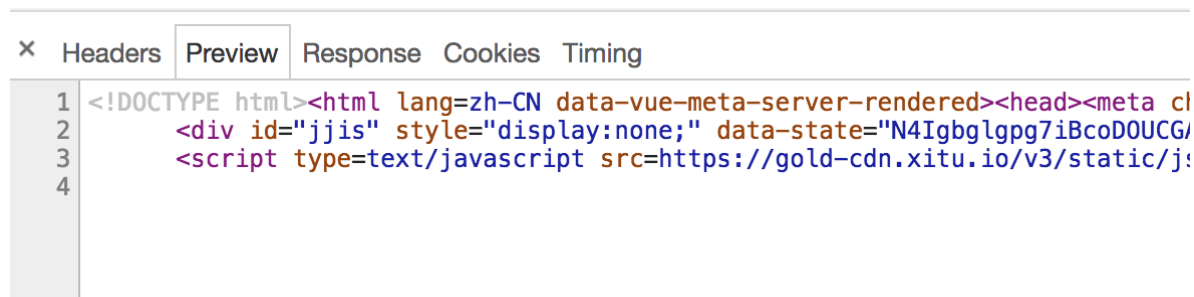
加载过程

要点如下：

- 浏览器根据 DNS 服务器得到域名的 IP 地址
- 向这个 IP 的机器发送 HTTP 请求
- 服务器收到、处理并返回 HTTP 请求
- 浏览器得到返回内容

例如在浏览器输入 `https://juejin.im/timeline`，然后经过 DNS 解析，`juejin.im` 对应的 IP 是 `36.248.217.149`（不同时间、地点对应的 IP 可能会不同）。然后浏览器向该 IP 发送 HTTP 请求。

server 端接收到 HTTP 请求，然后经过计算（向不同的用户推送不同的内容），返回 HTTP 请求，返回的内容如下：



其实就是一堆 HTML 格式的字符串，因为只有 HTML 格式浏览器才能正确解析，这是 W3C 标准的要求。接下来就是浏览器的渲染过程。

渲染过程

要点如下：

- 根据 HTML 结构生成 DOM 树
- 根据 CSS 生成 CSSOM
- 将 DOM 和 CSSOM 整合形成 RenderTree
- 根据 RenderTree 开始渲染和展示
- 遇到 `<script>` 时，会执行并阻塞渲染

上文中，浏览器已经拿到了 server 端返回的 HTML 内容，开始解析并渲染。最初拿到的内容就是一堆字符串，必须先结构化成计算机擅长处理的基本数据结构，因此要把 HTML 字符串转化成 DOM 树——树是最基本的数据结构之一。

解析过程中，如果遇到 `<link href= "... ">` 和 `<script src= "... ">` 这种外链加载 CSS 和 JS 的标签，浏览器会异步下载，下载过程和上文中下载 HTML 的流程一样。只不过，这里下载下来的字符串是 CSS 或者 JS 格式的。

浏览器将 CSS 生成 CSSOM，再将 DOM 和 CSSOM 整合成 RenderTree，然后针对 RenderTree 即可进行渲染了。大家可以想一下，有 DOM 结构、有样式，此时就能满足渲染的条件了。另外，这里也可以解释一个问题——**为何要将 CSS 放在 HTML 头部？**——这样会让浏览器尽早拿到 CSS 尽早生成 CSSOM，然后在解析 HTML 之后可一次性生成最终的 RenderTree，渲染一次即可。如果 CSS 放在 HTML 底部，会出现渲染卡顿的情况，影响性能和体验。

最后，渲染过程中，如果遇到 `<script>` 就停止渲染，执行 JS 代码。因为浏览器渲染和 JS 执行共用一个线程，而且这里必须是单线程操作，多线程会产生渲染 DOM 冲突。待 `<script>` 内容执行完之后，浏览器继续渲染。最后再思考一个问题 —— **为何要将 JS 放在 HTML 底部？** —— JS 放在底部可以保证让浏览器优先渲染完现有的 HTML 内容，让用户先看到内容，体验好。另外，JS 执行如果涉及 DOM 操作，得等待 DOM 解析完成才行，JS 放在底部执行时，HTML 肯定都解析成了 DOM 结构。JS 如果放在 HTML 顶部，JS 执行的时候 HTML 还没来得及转换为 DOM 结构，可能会报错。

关于浏览器整个流程，百度的多益大神有更加详细的文章，推荐阅读下：《[从输入 URL 到页面加载完成的过程中都发生了什么事情？](#)》。

性能优化

性能优化的题目也是面试常考的，这类题目有很大的扩展性，能够扩展出来很多小细节，而且对个人的技术视野和业务能力有很大的挑战。这部分笔者会重点讲下常用的性能优化方案。

题目：总结前端性能优化的解决方案

优化原则和方向

性能优化的原则是**以更好的用户体验为标准**，具体就是实现下面的目标：

1. 多使用内存、缓存或者其他方法
2. 减少 CPU 和 GPU 计算，更快展现

优化的方向有两个：

- 减少页面体积，提升网络加载
- 优化页面渲染

减少页面体积，提升网络加载

- 静态资源的压缩合并（JS 代码压缩合并、CSS 代码压缩合并、雪碧图）
- 静态资源缓存（资源名称加 MD5 戳）
- 使用 CDN 让资源加载更快

优化页面渲染

- CSS 放前面，JS 放后面
- 懒加载（图片懒加载、下拉加载更多）
- 减少 DOM 查询，对 DOM 查询做缓存
- 减少 DOM 操作，多个操作尽量合并在一起执行（`DocumentFragment`）
- 事件节流
- 尽早执行操作（`DOMContentLoaded`）
- 使用 SSR 后端渲染，数据直接输出到 HTML 中，减少浏览器使用 JS 模板渲染页面 HTML 的时间

详细解释

静态资源的压缩合并

如果不合并，每个都会走一遍之前介绍的请求过程

```
<script src="a.js"></script>
<script src="b.js"></script>
<script src="c.js"></script>
```

如果合并了，就走一遍请求过程

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

静态资源缓存

通过链接名称控制缓存

```
<script src="abc_1.js"></script>
```

只有内容改变的时候，链接名称才会改变

```
<script src="abc_2.js"></script>
```

这个名称不用手动改，可通过前端构建工具根据文件内容，为文件名称添加 MD5 后缀。

使用 CDN 让资源加载更快

CDN 会提供专业的加载优化方案，静态资源要尽量放在 CDN 上。例如：

```
<script src="https://cdn.bootcss.com/zepto/1.0rc1/zepto.min.js"></script>
```

使用 SSR 后端渲染

可一次性输出 HTML 内容，不用在页面渲染完成之后，再通过 Ajax 加载数据、再渲染。例如使用 smarty、Vue SSR 等。

CSS 放前面，JS 放后面

上文讲述浏览器渲染过程时已经提过，不再赘述。

懒加载

一开始先给为 `src` 赋值成一个通用的预览图，下拉时候再动态赋值成正式的图片。如下，`preview.png` 是预览图片，比较小，加载很快，而且很多图片都共用这个 `preview.png`，加载一次即可。待页面下拉，图片显示出来时，再去替换 `src` 为 `data-realsrc` 的值。

```

```

另外，这里为何要用 `data-` 开头的属性值？—— 所有 HTML 中自定义的属性，都应该用 `data-` 开头，因为 `data-` 开头的属性浏览器渲染的时候会忽略掉，提高渲染性能。

DOM 查询做缓存

两段代码做一下对比：

```
var pList = document.getElementsByTagName('p') // 只查询一个 DOM，缓存在 pList 中了
var i
for (i = 0; i < pList.length; i++) {
}
```

```
var i
for (i = 0; i < document.getElementsByTagName('p').length; i++) { // 每次循环，都会查询 DOM，耗费性能
}
```

总结：DOM 操作，无论查询还是修改，都是非常耗费性能的，应尽量减少。

合并 DOM 插入

DOM 操作是非常耗费性能的，因此插入多个标签时，先插入 Fragment 然后再统一插入 DOM。

```
var listNode = document.getElementById('list')
// 要插入 10 个 li 标签
var frag = document.createDocumentFragment();
var x, li;
for(x = 0; x < 10; x++) {
    li = document.createElement("li");
    li.innerHTML = "List item " + x;
    frag.appendChild(li); // 先放在 frag 中，最后一次性插入到 DOM 结构中。
}
listNode.appendChild(frag);
```

事件节流

例如要在文字改变时触发一个 change 事件，通过 keyup 来监听。使用节流。

```
var textarea = document.getElementById('text')
var timeoutId
textarea.addEventListener('keyup', function () {
    if (timeoutId) {
        clearTimeout(timeoutId)
    }
    timeoutId = setTimeout(function () {
        // 触发 change 事件
    }, 100)
})
```

尽早执行操作

```
window.addEventListener('load', function () {
    // 页面的全部资源加载完才会执行，包括图片、视频等
})
document.addEventListener('DOMContentLoaded', function () {
    // DOM 渲染完即可执行，此时图片、视频还可能没有加载完
})
```

性能优化怎么做

上面提到的都是性能优化的单个点，性能优化项目具体实施起来，应该按照下面步骤推进：

1. 建立性能数据收集平台，摸底当前性能数据，通过性能打点，将上述整个页面打开过程消耗时间记录下来
2. 分析耗时较长时间段原因，寻找优化点，确定优化目标
3. 开始优化
4. 通过数据收集平台记录优化效果
5. 不断调整优化点和预期目标，循环2~4步骤

性能优化是个长期的事情，不是一蹴而就的，应该本着先摸底、再分析、后优化的原则逐步来做。

Web 安全

题目：前端常见的安全问题有哪些？

Web 前端的安全问题，能回答出下文的两个问题，这个题目就能基本过关了。开始之前，先说一个最简单的攻击方式——SQL 注入。

上学的时候就有一个「SQL注入」的攻击方式。例如做一个系统的登录界面，输入用户名和密码，提交之后，后端直接拿到数据就拼接 SQL 语句去查询数据库。如果在输入时进行了恶意的 SQL 拼装，那么最后生成的 SQL 就会有漏洞。但是现在稍微大型一点的系统，都不会这么做，从提交登录信息到最后拿到授权，要经过层层验证。因此，SQL 注入都只出现在比较低端小型的系统上。

XSS (Cross Site Scripting, 跨站脚本攻击)

这是前端最常见的攻击方式，很多大型网站（如 Facebook）都被 XSS 攻击过。

举个例子，我在一个博客网站正常发表一篇文章，输入汉字、英文和图片，完全没有问题。但是如果我写的是恶意的 JS 脚本，例如获取到 `document.cookie` 然后传输到自己的服务器上，那我这篇博客的每一次浏览都会执行这个脚本，都会把访客 cookie 中的信息偷偷传递到我的服务器上。

其实原理上就是黑客通过某种方式（发布文章、发表评论等）将一段特定的 JS 代码隐蔽地输入进去。然后别人再看这篇文章或者评论时，之前注入的这段 JS 代码就执行了。**JS 代码一旦执行，那可就不受控制了，因为它跟网页原有的 JS 有同样的权限**，例如可以获取 server 端数据、可以获取 cookie 等。于是，攻击就这样发生了。

XSS的危害

XSS 的危害相当大，如果页面可以随意执行别人不安全的 JS 代码，轻则会让页面错乱、功能缺失，重则会造成用户的信息泄露。

比如早些年社交网站经常爆出 XSS 蠕虫，通过发布的文章内插入 JS，用户访问了感染不安全 JS 注入的文章，会自动重新发布新的文章，这样的文章会通过推荐系统进入到每个用户的文章列表面前，很快会造成大规模的感染。

还有利用获取 cookie 的方式，将 cookie 传入入侵者的服务器上，入侵者就可以模拟 cookie 登录网站，对用户的信息进行篡改。

XSS的预防

那么如何预防 XSS 攻击呢？—— 最根本的方式，就是对用户输入的内容进行验证和替换，需要替换的字符有：

```
& 替换为: &amp;
< 替换为: &lt;
> 替换为: &gt;
" 替换为: &quot;
' 替换为: &#x27;
/ 替换为: &#x2f;
```

替换了这些字符之后，黑客输入的攻击代码就会失效，XSS 攻击将不会轻易发生。

除此之外，还可以通过对 cookie 进行较强的控制，比如对敏感的 cookie 增加 `http-only` 限制，让 JS 获取不到 cookie 的内容。

CSRF（Cross-site request forgery，跨站请求伪造）

CSRF 是借用了当前操作者的权限来偷偷地完成某个操作，而不是拿到用户的信息。

例如，一个支付类网站，给他人转账的接口是 `http://buy.com/pay?touid=999&money=100`，而这个接口在使用时没有任何密码或者 token 的验证，只要打开访问就直接给他人转账。一个用户已经登录了 `http://buy.com`，在选择商品时，突然收到一封邮件，而这封邮件正文有这么一行代码 ``，他访问了邮件之后，其实就已经完成了购买。

CSRF 的发生其实是借助了一个 cookie 的特性。我们知道，登录了 `http://buy.com` 之后，cookie 就会有登录过的标记了，此时请求 `http://buy.com/pay?touid=999&money=100` 是会带着 cookie 的，因此 server 端就知道已经登录了。而如果在 `http://buy.com` 去请求其他域名的 API 例如 `http://abc.com/api` 时，是不会带 cookie 的，这是浏览器的同源策略的限制。但是—— **此时在其他域名的页面中，请求 `http://buy.com/pay?touid=999&money=100`，会带着 `buy.com` 的 cookie，这是发生 CSRF 攻击的理论基础。**

预防 CSRF 就是加入各个层级的权限验证，例如现在的购物网站，只要涉及现金交易，肯定要输入密码或者指纹才行。除此之外，敏感的接口使用 `POST` 请求而不是 `GET` 也是很重要的。

小结

本小节总结了前端运行环境（即浏览器）的一些常考查知识点，包括页面加载过程、如何性能优化以及需要注意的安全问题。

一面 6：开发环境相关知识点与高频考题解析

工程师的开发环境决定其开发效率，常用的开发环境配置也是面试考查点之一。

知识点梳理

- IDE

- Git
- Linux 基础命令
- 前端构建工具
- 调试方法

本小节会重点介绍 Git 的基本用法、代码部署和开发中常用的 Linux 命令，然后以 webpack 为例介绍下前端构建工具，最后介绍怎么抓包解决线上问题。这些都是日常开发和面试中常用到的知识。

IDE

题目：你平时都使用什么 IDE 编程？有何提高效率的方法？

前端最常用的 IDE 有 [Webstorm](#)、[Sublime](#)、[Atom](#) 和 [VSCode](#)，我们可以分别去它们的官网看一下。

Webstorm 是最强大的编辑器，因为它拥有各种强大的插件和功能，但是我没有用过，因为它收费。不是我舍不得花钱，而是因为我感觉免费的 Sublime 已经够我用了。跟面试官聊到 Webstorm 的时候，没用过没事儿，但一定要知道它：第一，强大；第二，收费。

Sublime 是我日常用的编辑器，第一它免费，第二它轻量、高效，第三它插件非常多。用 Sublime 一定要安装各种插件配合使用，可以去网上搜一下“sublime”常用插件的安装以及用法，还有它的各种快捷键，并且亲自使用它。这里就不一一演示了，网上的教程也很傻瓜式。

Atom 是 GitHub 出品的编辑器，跟 Sublime 差不多，免费并且插件丰富，而且跟 Sublime 相比风格上还有些小清新。但是我用过几次就不用了，因此它打开的时候会比较慢，卡一下才打开。当然总体来说也是很好用的，只是个人习惯问题。

VSCode 是微软出品的轻量级（相对于 Visual Studio 来说）编辑器，微软做 IDE 那是出了名的好，出了名的大而全，因此 VSCode 也有上述 Sublime 和 Atom 的各种优点，但是我也是因为个人习惯问题（本人不愿意尝试没有新意的东西），用过几次就不用了。

总结一下：

- 如果你要走大牛、大咖、逼格的路线，就用 Webstorm
- 如果你走普通、屌丝、低调路线，就用 Sublime
- 如果你走小清新、个性路线，就用 VSCode 或者 Atom
- 如果你面试，最好有一个用的熟悉，其他都会一点

最后注意：千万不要说你使用 Dreamweaver 或者 notepad++ 写前端代码，会被人鄙视的。如果你不做 .NET 也不要 Visual Studio，不做 Java 也不要 Eclipse。

Git

你此前做过的项目一定要用过 Git，而且必须是命令行，如果没用过，你自己也得恶补一下。对 Git 的基本应用比较熟悉的同学，可以跳过这一部分了。macOS 自带 Git，Windows 需要安装 Git 客户端，去 [Git 官网](#) 下载即可。

国内比较好的 Git 服务商有 coding.net，国外有大名鼎鼎的 GitHub，但是有时会有网络问题，因此建议大家注册一个 coding.net 账号然后创建项目，来练练手。

题目：常用的 Git 命令有哪些？如何使用 Git 多人协作开发？

常用的 Git 命令

首先，通过 `git clone <项目远程地址>` 下载下来最新的代码，例如 `git clone git@git.coding.net:username/project-name.git`，默认会下载 master 分支。

然后修改代码，修改过程中可以通过 `git status` 看到自己的修改情况，通过 `git diff <文件名>` 可查阅单个文件的差异。

最后，将修改的内容提交到远程服务器，做如下操作

```
git add .
git commit -m "xxx"
git push origin master
```

如果别人也提交了代码，你想同步别人提交的内容，执行 `git pull origin master` 即可。

如何多人协作开发

多人协作开发，就不能使用 `master` 分支了，而是要每个开发者单独拉一个分支，使用 `git checkout -b <branchname>`，运行 `git branch` 可以看到本地所有的分支名称。

自己的分支，如果想同步 `master` 分支的内容，可运行 `git merge master`。切换分支可使用 `git checkout <branchname>`。

在自己的分支上修改了内容，可以将自己的分支提交到远程服务器

```
git add .
git commit -m "xxx"
git push origin <branchname>
```

最后，待代码测试没问题，再将自己分支的内容合并到 `master` 分支，然后提交到远程服务器。

```
git checkout master
git merge <branchname>
git push origin master
```

关于 SVN

关于 SVN 笔者的态度和针对 IE 低版本浏览器的态度一样，你只需要查询资料简单了解一下。面试的时候可能会问到，但你只要熟悉了 Git 的操作，面试官不会因为你不熟悉 SVN 而难为你。前提是你要知道一点 SVN 的基本命令，自己上网一查就行。

不过 SVN 和 Git 的区别你得了解。SVN 是每一步操作都离不开服务器，创建分支、提交代码都需要连接服务器。而 Git 就不一样了，你可以在本地创建分支、提交代码，最后再一起 push 到服务器上。因此，Git 拥有 SVN 的所有功能，但是却比 SVN 强大得多。（Git 是 Linux 的创始人 Linus 发明的东西，因此也倍得推崇。）

Linux 基础命令

目前互联网公司的线上服务器都使用 Linux 系统，测试环境为了保证和线上一致，肯定也是使用 Linux 系统，而且都是命令行的，没有桌面，不能用鼠标操作。因此，掌握基础的 Linux 命令是非常必要的。下面总结一些最常用的 Linux 命令，建议大家在真实的 Linux 系统下亲自试一下。

关于如何得到 Linux 系统，有两种选择：第一，在自己电脑的虚拟机中安装一个 Linux 系统，例如 Ubuntu/CentOS 等，下载这些都不用花钱；第二，花钱去阿里云等云服务商租一个最便宜的 Linux 虚拟机。推荐第二种。一般正式入职之后，公司都会给你分配开发机或者测试机，给你账号和密码，你自己可以远程登录。

题目：常见 linux 命令有哪些？

登录

入职之后，一般会有现有的用户名和密码给你，你拿来之后直接登录就行。运行 `ssh name@server` 然后输入密码即可登录。

目录操作

- 创建目录 `mkdir <目录名称>`
- 删除目录 `rm <目录名称>`
- 定位目录 `cd <目录名称>`
- 查看目录文件 `ls ll`
- 修改目录名 `mv <目录名称> <新目录名称>`
- 拷贝目录 `cp <目录名称> <新目录名称>`

文件操作

- 创建文件 `touch <文件名称> vi <文件名称>`
- 删除文件 `rm <文件名称>`
- 修改文件名 `mv <文件名称> <新文件名称>`
- 拷贝文件 `cp <文件名称> <新文件名称>`

文件内容操作

- 查看文件 `cat <文件名称> head <文件名称> tail <文件名称>`
- 编辑文件内容 `vi <文件名称>`
- 查找文件内容 `grep '关键字' <文件名称>`
- o *

前端构建工具

构建工具是前端工程化中不可缺少的一环，非常重要，而在面试中却有其特殊性——**面试官会通过询问构建工具的作用、目的来询问你对构建工具的了解，只要这些你都知道，不会再追问细节。**因为，在实际工作中，真正能让你编写构建工具配置文件的机会非常少，一个项目就配置一次，后面就很少改动了。而且，如果是大众使用的框架（如 React、Vue 等），还会直接有现成的脚手架工具，一键创建开发环境，不用手动配置。

题目：前端为何要使用构建工具？它解决了什么问题？

何为构建工具

“构建”也可理解为“编译”，就是将开发环境的代码转换成运行环境代码的过程。**开发环境的代码是为了更好地阅读，而运行环境的代码是为了更快地执行，两者目的不一样，因此代码形式也不一样。**例如，开发环境写的 JS 代码，要通过混淆压缩之后才能放在线上运行，因为这样代码体积更小，而且对代码执行不会有任何影响。总结一下需要构建工具处理的几种情况：

- **处理模块化**: CSS 和 JS 的模块化语法, 目前都无法被浏览器兼容。因此, 开发环境可以使用既定的模块化语法, 但是需要构建工具将模块化语法编译为浏览器可识别形式。例如, 使用 webpack、Rollup 等处理 JS 模块化。
- **编译语法**: 编写 CSS 时使用 Less、Sass, 编写 JS 时使用 ES6、TypeScript 等。这些标准目前也都无法被浏览器兼容, 因此需要构建工具编译, 例如使用 Babel 编译 ES6 语法。
- **代码压缩**: 将 CSS、JS 代码混淆压缩, 为了让代码体积更小, 加载更快。

构建工具介绍

最早普及使用的构建工具是 [Grunt](#), 不久又被 [Gulp](#) 给追赶上。Gulp 因其简单的配置以及高效的性能而被大家所接受, 也是笔者个人比较推荐的构建工具之一。如果你做一些简单的 JS 开发, 可以考虑使用。

如果你的项目比较复杂, 而且是多人开发, 那么你就需要掌握目前构建工具届的神器 —— webpack。不过神器也有一个缺点, 就是学习成本比较高, 需要拿出专门的时间来专心学习, 而不是三言两语就能讲完的。我们下面就演示一下 webpack 最简单的使用, 全面的学习还得靠大家去认真查阅相关文档, 或者参考专门讲解 webpack 的教程。

webpack 演示

接下来我们演示一下 webpack 处理模块化和混淆压缩代码这两个基本功能。

首先, 你需要安装 Node.js, 没有安装的可以去 [Node.js 官网](#) 下载并安装。安装完成后运行如下命令来验证是否安装成功。

```
node -v
npm -v
```

然后, 新建一个目录, 进入该目录, 运行 `npm init`, 按照提示输入名称、版本、描述等信息。完成之后, 该目录下出现了一个 `package.json` 文件, 是一个 JSON 文件。

接下来, 安装 webpack, 运行 `npm i --save-dev webpack`, 网络原因需要耐心等待几分钟。

接下来, 编写源代码, 在该目录下创建 `src` 文件夹, 并在其中创建 `app.js` 和 `dt.js` 两个文件, 文件内容分别是:

```
// dt.js 内容
module.exports = {
  getDateNow: function () {
    return Date.now()
  }
}

// app.js 内容
var dt = require('./dt.js')
alert(dt.getDateNow())
```

然后, 再返回上一层目录, 新建 `index.html` 文件 (该文件和 `src` 属于同一层级), 内容是

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>test</title>
</head>
<body>
  <div>test</div>

  <script src='./dist/bundle.js'></script>
</body>
</html>
```

然后，编写 webpack 配置文件，新建 `webpack.config.js`，内容是

```
const path = require('path');
const webpack = require('webpack');
module.exports = {
  context: path.resolve(__dirname, './src'),
  entry: {
    app: './app.js',
  },
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: 'bundle.js',
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        //supresses warnings, usually from module minification
        warnings: false
      }
    }),
  ],
};
```

总结一下，目前项目的文件目录是：

```
src
  +-- app.js
  +-- dt.js
index.html
package.json
webpack.config.js
```

接下来，打开 `package.json`，然后修改其中 `scripts` 的内容为：

```
"scripts": {
  "start": "webpack"
}
```

在命令行中运行 `npm start`，即可看到编译的结果，最后在浏览器中打开 `index.html`，即可弹出 `Date.now()` 的值。

总结

最后再次强调，**深刻理解构建工具存在的价值，比你多会一些配置代码更加有意义，特别是对于应对面试来说。**

调试方法

调试方法这块被考查最多的就是如何进行抓包。

题目：如何抓取数据？如何使用工具来配置代理？

PC 端的网页，我们可以通过 Chrome、Firefox 等浏览器自带的开发者工具来查看网页的所有网络请求，以帮助排查 bug。这种监听、查看网络请求的操作称为**抓包**。

针对移动端的抓包工具，Mac 系统下推荐使用 Charles 这个工具，首先 [下载](#) 并安装，打开。Windows 系统推荐使用 [Fiddler](#)，下载安装打开。两者使用基本一致，下面以 Charles 为例介绍。

接下来，将安装好 Charles 的电脑和要抓包的手机，连接到同一个网络（一般为公司统一提供的内网，由专业网络工程师搭建），保证 IP 段相同。然后，将手机设置网络代理（每种不同手机如何设置网络代理，网上都有傻瓜式教程），代理的 IP 为电脑的 IP，代理的端口为 8888。然后，Charles 可能会有一个弹框提示是否允许连接代理，这里选择“允许”即可。这样，使用手机端访问的网页或者联网的请求，Charles 就能监听到了。

在开发过程中，经常用到抓包工具来做代理，将线上的地址代理到测试环境，Charles 和 Fiddler 都可实现这个功能。以 Charles 为例，点击菜单栏中 Tools 菜单，然后二级菜单中点击 Map Remote，会弹出配置框。首先，选中 Enable Map Remote 复选框，然后点击 Add 按钮，添加一个代理项。例如，如果要将线上的 `https://www.aaa.com/api/getuser?name=xxx` 这个地址代理到测试地址

`http://168.1.1.100:8080/api/getuser?name=xxx`，配置如下图

Edit Mapping

Map From

Protocol: https

Host: www.aaa.com

Port: 443

Path: /api/getuser

Query: ?name=xxx

Map To

Protocol: http

Host: 168.1.1.100

Port: 8080

Path: /api/getuser

Query: ?name=xxx

☐ Preserve Host header

To map from a path and its subdirectories you must end the path with a *. To map an entire host leave the path blank.

?

Cancel

OK

小结

本小节总结了前端开发环境常考查的知识，这些知识也是前端程序员必须掌握的，否则会影响开发效率。

二面 1：如何回答常见的软技能问题

面试是个技术活，不仅仅是技术，各种软技能的面试技巧也是非常重要的，尤其是程序员一般对于自己的软技能不是很看重，其实软技能才是决定你职场能够走多远的关键。

程序员应该具备的软技能

程序员除了业务技能外，应该具有下面的软技能：

1. 韧性：抗压能力，在一定项目压力下能够迎难而上，比如勇于主动承担和解决技术难题
2. 责任心：对于自己做过的项目，能够出现 bug 之类主动解决
3. 持续学习能力：IT 行业是个需要不断充电的行业，尤其 Web 前端这些年一直在巨变，所以持续学习能力很重要
4. 团队合作能力：做项目不能个人英雄主义，应该融入团队，跟团队一起打仗
5. 交流沟通能力：经常会遇见沟通需求和交互设计的工作，应该乐于沟通分享

另外在《软技能：代码之外的生存指南》这本书里提到了下面一些软技能：

1. 职业
2. 自我营销
3. 学习能力
4. 提升工作效率
5. 理财
6. 健身
7. 积极的人生观

常见的软技能问题和提升

回答软技能类的问题，应该注意在回答过程中体现自己具备的软技能。下面列举几个常见的软技能类的问题。

回想下你遇见过最难打交道的同事，你是如何跟他沟通的

一般来说，工作中总会遇见一两个自己不喜欢的人，这种情况应该尽量避免冲突，从自己做起慢慢让对方感觉到自己的合作精神。

所以，遇见难打交道的同事，不要急于上报领导，应该自己主动多做一些事情，比如规划好工作安排，让他选择自己做的事情，有了结论记得发邮件确认下来，这样你们的领导和其他成员都会了解到工作的安排，在鞭策对方的同时，也做到了职责明确。在项目当中，多主动检查项目进展，提前发现逾期的问题。

重点是突出：自己主动沟通解决问题的意识，而不是遇见问题就找领导。

当你被分配一个几乎不可能完成的任务时，你会怎么做

这种情况下，一般通过下面方式来解决：

1. 自己先查找资料，寻找解决方案，评估自己需要怎样的资源来完成，需要多长时间
2. 能不能借助周围同事来解决问题
3. 拿着分析结果跟上级反馈，寻求帮助或者资源

突出的软技能：分析和解决问题，沟通寻求帮助。

业余时间都做什么？除了写码之外还有什么爱好

这类问题也是面试官的高频问题，「一个人的业余时间决定了他的未来」，如果回答周末都在追剧打游戏之类的，未免显得太不上进。

一般来说，推荐下面的回答：

周末一般会有三种状态：

1. 和朋友一起去做做运动，也会聚会聊天，探讨下新技术之类的；
2. 也会看一些书籍充充电，比如我最近看的 xx，有什么的想法；
3. 有时候会闷在家用最近比较火的技术做个小项目或者实现个小功能之类的。

这样的回答，既能表现自己阳光善于社交沟通的一面，又能表现自己的上进心。

小结

本小节介绍了程序员除了业务技术能力之外应该日常修炼的软技能，在面试中，软技能会被以各种形式问起，候选人应该先了解有哪些软技能可以修炼，才能在回答软技能问题的时候，尽量提到自己具备的软技能。

二面 2：如何介绍项目及应对项目细节追问

一个标准的面试流程中，肯定会在一面二面中问到你具体做过的项目，然后追问项目的细节。这类问题往往会通过下面形式来提问：

1. 发现你简历的一个项目，直接让你介绍下这个项目
2. 让你回忆下你做过的项目中，最值得分享（最大型/最困难/最能体现技术能力/最难忘）的
3. 如果让你设计 xx 系统/项目，你会怎么着手干

这类跟项目相关的综合性问题，既能体现候选人的技术水平、业务水平和架构能力，也能够辨别候选人是不是真的做过项目，还能够发现候选人的一些软技能。

下面分享下，遇见这类问题应该怎样回答。

怎样介绍自己做过的一个项目

按照第 1 小节说的，简历当中的项目，你要精挑细选，既要体现技术难度，又要想好细节。具体要介绍一个项目（包括梳理一个项目），可以按照下面几个阶段来做。

1. 介绍项目背景

这个项目为什么做，当初大的环境背景是什么？还是为了解决一个什么问题而设立的项目？背景是很重要的，如果不了解背景，一上来就听一个结论性的项目，面试官可能对于项目的技术选型、技术难度会有理解偏差，甚至怀疑是否真的有过这样的项目。

比如一上来就说：我们的项目采用了「backbone」来做框架，然后。。。而「backbone」已经是三四年前比较新鲜的技术，现在会有更好的选择方案，如果不介绍项目的时间背景，面试官肯定一脸懵逼。

2. 承担角色

项目涉及的人员角色有哪些，自己在其中扮演的角色是什么？

这里候选往往会自己给自己挖坑，比如把自己在项目中起到的作用夸大等。一般来说，面试官细节追问的时候，如果候选人能够把细节或者技术方案等讲明白、讲清楚，不管他是真的做过还是跟别人做过，或者自己认真思考过，都能体现候选人的技术水平和技术视野。前提还是在你能够兜得住的可控范围之内做适当的「美化」。

3. 最终的结果和收益

项目介绍过程中，应该介绍项目最终的结果和收益，比如项目最后经过多久的开发上线了，上线后的数据是怎样的，是否达到预期，还是带来了新的问题，遇见了问题自己后续又是怎样补救的。

4. 有始有终：项目总结和反思

有总结和反思，才会有进步。项目做完了往往会有一些心得和体会，这时候应该跟面试官说出来。在梳理项目的总结和反思时，可以按照下面的列表来梳理：

- 收获有哪些？

- 是否有做得不足的地方，怎么改进？
- 是否具有可迁移性？

比如，之前详细介绍了某个项目，这个项目当时看来没有什么问题，但是现在有更好的解决方案了，候选人就应该在这里提出来：现在看来，这个项目还有 xx 的问题，我可以通过 xx 的方式来解决。

再比如：做这个项目的时候，你做得比较出彩的地方，可以迁移到其他项目中直接使用，小到代码片段，大到解决方案，总会有你值得总结和梳理的地方。

介绍完项目总结这部分，也可以引导面试官往自己擅长的领域思考。比如上面提到项目中的问题，可以往你擅长的方面引导，即使面试官没有问到，你也介绍到了。

按照上面的四段体介绍项目，会让面试官感觉候选人有清晰的思路，对整个项目也有理解和想法，还能够总结反思项目的收益和问题，可谓「一箭三雕」。

没有做过大型项目怎么办

对于刚刚找工作的应届生，或者面试官让你进行一个大型项目的设计，候选人可能没有类似的经验。这时候不要用「我不会、没做过」一句话就带过。

如果是实在没有项目可以说，那么可以提自己日常做的练手项目，或者看到一个解决方案的文章/书，提到的某个项目，抒发下自己的想法。

如果是对于面试官提出来需要你设计的项目/系统，可以按照下面几步思考：

1. 有没有遇见过类似的项目
2. 有没有读过类似解决方案的文章
3. 项目能不能拆解，拆解过程中能不能发现自己做过的项目可以用
4. 项目解决的问题是什么，这类问题有没有更好的解决方案

总之，切记不要一句「不知道、没做过」就放弃，每一次提问都是自己表现的机会。

项目细节和技术点的追问

介绍项目的过程中，面试官可能会追问技术细节，所以我们在准备面试的时候，应该尽量把技术细节梳理清楚，技术细节包括：

1. 技术选型方案：当时做技术选型所面临的状况
2. 技术解决方案：最终确定某种技术方案的原因，比如：选择用 Vue 而没有用 React 是什么原因？
3. 项目数据和收益
4. 项目中最难的地方
5. 遇见的坑：如使用某种框架遇见哪些坑

一般来说，做技术选型的时候需要考虑下面几个因素：

1. 时代：现在比较火的技术是什么，为什么火起来，解决了什么问题，能否用到我的项目中？
2. 团队：个人或者团队对某种技术的熟悉程度是怎样的，学习成本又是怎样的？
3. 业务需求：需求是怎样的，能否套用现在的成熟解决方案/库来快速解决？
4. 维护成本：一个解决方案的是否再能够 cover 住的范围之内？

在项目中遇见的数据和收益应该做好跟踪，保证数据的真实性和可信性。另外，遇见的坑可能是面试官问得比较多的，尤其现在比较火的一些技术（Vue、React、webpack），一般团队都在使用，所以一定要提前准备下。

小结

本小节介绍了面试中关于项目类问题的回答方法，介绍项目要使用四段体的方式，从背景、承担角色、收益效果和总结反思四个部分来介绍项目。

准备这个面试环节的时候，利用笔者一直提倡的「思维导图」法，好好回顾和梳理自己的项目。

HR 面：谈钱不伤感情

当你顺利通过面试，最后 HR 面试主要有两大环节：

1. 了解候选人是否在岗位、团队、公司文化等方面能够跟要求匹配，并且能够长期服务
2. 谈薪资

匹配度考查

很多情况下 HR 并不懂技术，但是也会问你项目上的问题，这时候其实是考查候选人对自己所做项目和技术的能力。「检验一个人是否掌握一个专业知识，看他能不能把专业知识通俗易懂地对一个外行讲明白」。在面对 HR 询问项目或者技术点的细节时，你应该尽量通俗易懂地将知识点讲明白。怎样做到通俗易懂？笔者建议多作类比，跟生活中常见的或者大家都明白的知识作对比。举个例子，讲解「减少 Cookie 对页面打开速度优化有效果」的时候，笔者会这样来介绍：

你应该知道平时上传文件（比如头像）要比下载文件慢，这是因为网络上行带宽要比下行带宽窄，HTTP 请求的时候其实是双向的，先上传本地的信息，包括要访问的网址、本地携带的一些 Cookie 之类的数据，这些数据因为是上行（从用户手中发到服务器，给服务器上的代码使用），本来上行带宽小，所以对速度的影响更大。因此在 HTTP 上行请求中，减少 Cookie 的大小等方式可以有效提高打开速度，尤其是在带宽不大的网络环境中，比如手机的 2G 弱网络环境。

如果他还是不太清楚，那么带宽、上行、下行这些概念都可以类比迅雷下载这个场景，一解释应该就明白了。

HR 面试还会通过一些问题，判断你与公司文化是否契合，比如阿里的 HR 有政委体系，会严格考查候选人是否符合公司企业文化。针对这类问题应该在回答过程中体现出自己阳光正能量的一面，不要抱怨前公司，抱怨前领导，多从自身找原因和不足，谦虚谨慎。

谈薪资

谈 offer 的环节并不轻松，包括笔者在内往往在这个阶段「折了兵」。不会谈 offer 往往会遇见这样的情况：

1. 给你多少就要多少，不会议价
2. 谈一次被打击一次，最后越来越没有底气

尤其是很多不专业的 HR 以压低工资待遇为自己的首要目标，把候选人打击得不行不行的。一个不够满意的 offer 往往导致入职后出现抱怨，本小节重点讲下如何谈到自己中意的 offer。

准确定位和自我估值

在准备跳槽时，每个人肯定会对自己有一个预估，做好足够的心理准备。下面谈下怎么对自己的薪酬做个评估。一般来说跳槽的薪水是根据现在薪酬的基础上浮 15~30%，具体看个人面试的情况。对于应届毕业生，大公司基本都有标准薪水，同期的应届生差别不会特别大。

除了上面的方法，还应该按照公司的技术职级进行估值。每个公司都有对应的技术职级，不同的技术职级薪酬范围是固定的，如果是小公司，则可以参考大公司的职级范围来确定薪资范围。

根据职级薪资范围和自己现在薪酬基础上浮后的薪酬，做个比较，取其较高的结果。

当然如果面试结果很好，你可以适当地提高下薪酬预期。除了这种情况，应该针对不同的性质来对 offer 先做好不同的估值。这里的预期估值只是心理预期，也就是自己的「底牌」。

所谓不同性质的 offer 指的是：

1. 是否是自己真心喜欢的工作岗位：如果是自己真心喜欢的工作岗位，比如对于个人成长有利，或者希望进入某个公司部门，从事某个专业方向的工作，而你自己对于薪酬又不是特别在意，这时候可以适当调低薪酬预期，以拿到这个工作机会为主。
2. 是否只是做 backup 的岗位：面试可能不止面试一家，对于不是特别喜欢的公司部门，那么可以把这个 offer 做为 backup，后面遇见喜欢的公司可以以此基础来谈薪水。

这时候分两种情况：如果面试结果不是很好，这种情况应该优先拿到 offer，所以可以适当降低期望薪酬；如果面试结果很好，这种情况应该多要一些薪酬，增加的薪酬可以让你加入这家公司也心里很舒服。

对于自己真正的目标职位，面试之前应该先找 backup 岗位练练手，一是为了找出面试的感觉，二是为了拿到几个 offer 做好 backup。

关于如何客观评估自己的身价，有篇知乎的帖子比较专业，有时间可以读一下：

[如何在跳槽前客观地评估自己的身价？](#)

跟 HR 沟通的技巧

跟 HR 沟通的时候，不要夸大现在的薪酬，HR 知道的信息往往会超出你的认知，尤其大公司还会有背景调查，所以不要撒谎，实事求是。

跟 HR 沟通的技巧有以下几点：

不要急于出价

不要急于亮出自己的底牌，一旦你说出一个薪酬范围，自己就不能增加薪酬了，还给了对方砍价的空间。而且一个不合理的价格反而会让对方直接放弃。所以不要着急出价，先让对方出价。

同时，对于公司级别也是，不要一开始就奔着某个目标去面试，这样会加大面试的难度，比如：

目标是拿到阿里 P7 的职位，不要说不给 P7 我就不去面试之类的，这样的要求会让对方一开始就拿 P7 的标准来面试，可能会找 P8+ 的面试官来面试你，这样会大大提升面试难度。

要有底气足够自信

要有底气，自信，自己按照上面的估值盘算好了想要的薪酬，那么应该有底气地说出来，并且给出具体原因，比如：

1. 我已经对贵公司的薪酬范围和级别有了大概的了解，我现在的水平大概范围是多少
2. 现在公司很快就有调薪机会，自己已经很久没有调薪，年前跳槽会损失年终奖等情况
3. 现在我已经有个公司多少 K 的 offer

如果 HR 表示你想要的薪酬不能满足，这时候你应该给出自己评估的依据，是根据行业职级标准还是自己现有薪酬范围，这样做到有理有据。

谈好 offer 就要尽快落实

对于已经谈拢的薪酬待遇，一定要 HR 以发邮件 offer 的形式来确认。

小结

本小节详细谈了 HR 轮面试的两大环节，重点介绍了谈 offer 的一些技巧，希望对你有所帮助和启发。

其他：面试注意事项

除了前面小节中提到的一些面试中应该注意的问题，本小节再整理一些面试中应该注意的事项。

1. 注意社交礼仪

虽然说 IT 行业不怎么注重工作环境，上下级也没有繁文缛节，但是在面试中还是应该注意一些社交礼仪的。像进门敲门、出门关门、站着迎人这类基本礼仪还是要做的。

舒适但不随意的着装

首先着装方面，不要太随意，也不要太正式，太正式的衣服可能会使人紧张，所以建议穿自己平时喜欢的衣服，关键是干净整洁。

约个双方都舒服的面试时间

如果 HR 打电话预约面试时间，记得一定要约个双方都舒服的时间，宁愿请假也要安排好面试时间。

有个case：前几天有个朋友说为了给公司招人，晚上住公司附近酒店，原因是候选人为了不耽误现在公司的工作，想在 10 点之前按时上班，预约的面试时间是早上 8 点。这样对于面试官来说增加了负担，心里肯定不会特别舒服，可能会影响候选人的面试结果。

面试时间很重要，**提前十分钟到面试地点**，熟悉下环境，做个登记之类的，留下个守时的好印象。如果因为堵车之类的原因不能按时到达，则要在约定时间之前电话通知对方。

2. 面试后的提问环节

面试是一个双向选择的事情，所以面试后一般会有提问环节。在提问环节，候选人最好不要什么都不问，更不要只问薪水待遇、是否加班之类的问题。

其实这个时候可以反问面试官了解团队情况、团队做的业务、本职位具体做的工作、工作的规划，甚至一些数据（可能有些问题不会直面回答）。

还可以问一些关于公司培训机会和晋升机会之类的问题。如果是一些高端职位，则可以问一下：自己的 leader 想把这个职位安排给什么样的人，希望多久的时间内可以达到怎样的水平。

3. 面试禁忌

- 不要对老东家有太多埋怨和负面评价
- 不要有太多负面情绪，多表现自己阳光的一面
- 不要夸大其词，尤其是数据方面
- 不要贬低任何人，包括自己之前的同事，比如有人喜欢说自己周围同事多么的差劲，来突出自己的优秀
- 不要过多争辩。你是来展现自己胜任能力的，不是来证明面试官很蠢的

4. 最好自己带电脑

有些面试会让候选人直接笔试，或者直接去小黑板上面画图写代码，这种笔试的时候会非常痛苦，我经常见单词拼写错误的候选人，这种情况最好是自己带着电脑，直接在自己熟悉的 IDE 上面编写。

这里应该注意，自己带电脑可能也有弊端。如果你对自己的开发环境和电脑足够熟悉，操作起来能够得心应手，那么可以带着；如果你本身电脑操作就慢，比如 Linux 命令不熟悉，打开命令行忘记了命令，这种情况下会被减分。带与不带自己根据自己情况权衡。

5. 面试后的总结和思考

- 面试完了多总结自己哪里做得不好，哪里做得好，都记录下来，后续扬长避短
- 通过面试肯定亲身体会到了公司团队文化、面试官体现出来的技术能力、专业性以及职位将来所做的事情，跟自己预期是否有差距，多个 offer 的话多做对比

每次面试应该都有所收获，毕竟花费了时间和精力。即使面不上也可以知道自己哪方面做得不好，继续加强。

小结

本小节重点谈了面试中应该注意的一些细节问题，细节虽小，但是大家应该注意。

总结与补充说明

恭喜你，学完了本小册。下面来总结下本小册的内容，并补充一些遗漏的内容。

总结

本小册主要带领大家从准备简历开始，逐步梳理技术面试知识点和非技术面试常考问题，最后介绍了一些谈 offer 之类的面试技巧。下面带领大家根据准备、技术面试、非技术面试和 HR 面试四部分，回顾一下每部分的要点。

准备阶段

简历准备：

1. 简历要求尽量平实，不要太花俏
2. 格式推荐 PDF
3. 内容包含：个人技能、项目经验和实习经验
4. 简历应该针对性来写
5. 简历提到的项目、技能都要仔细回想细节，挖掘可能出现的面试题

拿到面邀之后准备：

1. 开场问题：自我介绍、离职原因等
2. 了解面试官、了解公司和部门做的事情
3. 知识梳理推荐使用思维导图

技术面部分

集中梳理了 ECMAScript 基础、JS-Web-API、CSS 和 HTML、算法、浏览器和开发环境六大部分内容，并且就一些高频考题进行讲解。

非技术面试部分

主要从软技能和项目介绍两个部分来梳理。在软技能方面，介绍了工程师从业人员应该具有的软技能，并且通过几个面试真题介绍了怎么灵活应对面试官；在项目介绍小节，推荐按照项目背景、承担角色、项目收益和项目总结反思四步来介绍，并且继续推荐使用思维导图方式来梳理项目的细节。

HR 面

在小册最后，介绍了 HR 面试应该注意的问题，重点分享了作为一个 Web 前端工程师怎么对自己进行估值，然后跟 HR 进行沟通，拿到自己可以接受的 offer。

最后还介绍了一些面试注意事项，在面试整个流程中，太多主观因素，细节虽小也可能决定候选人面试的结果。

补充说明

本着通用性和面试门槛考虑的设计，本小册对于一些前端进阶和框架类的问题没有进行梳理，没有涉及的内容主要有：

1. Node.js部分
2. 类库：Zepto、jQuery、React、Vue 和 Angular 等
3. 移动开发

下面简单展开下上面的内容。

Node.js部分

Node.js 涉及的知识点比较多，而且比较偏后端和工具性，如果用 Node.js 来做 Server 服务，需要补充大量的后端知识和运维知识，这里帮助梳理下知识点：

- Node 开发环境
 - npm 操作
 - package.json
- Node 基础 API 考查
 - file system
 - Event
 - 网络
 - child process
- Node 重点和难点
 - 事件和异步理解
 - Stream 相关概念
 - Buffer 相关概念
 - domain
 - vm
 - cluster
 - 异常调优
- Server 相关
 - 库
 - Koa
 - Express
 - 数据库
 - MongoDB
 - MySQL
 - Redis
 - 运维部署
 - Nginx
 - 进程守候
 - 日志

Node 的出现让前端可以做的事情更多，除了做一些 Server 的工作以外，Node 在日常开发中可以做一些工具来提升效率，比如常见的前端构建工具目前都是 Node 来编写的，而我们在研发中，一些类似 Mock、本地 server、代码实时刷新之类的功能，都可以使用 Node 来自己实现。

前端框架（库）

jQuery 和 Zepto 分别是应用在 PC 和移动上面的库，大大降低了前端开发人员的门槛，很多前端工程师都是从写 jQuery 代码开始的。jQuery 和 Zepto 这两个库对外的 API 都是相同的。在面试的时候可能会问到一些具体代码的实现，比如下面两个问题：

题目：谈谈 jQuery 的 delegate 和 bind 有什么区别；`window.onload` 和 `$(document).ready` 有什么区别

这实际上都是 JS-Web-API 部分基础知识的实际应用：

- delegate 是事件代理（委托），bind 是直接绑定事件
- onload 是浏览器部分的全部加载完成，包括页面的图片之类资源；ready 则是 `DOMContentLoaded` 事件，比 onload 提前一些

下面再说下比较火的 Angular、React 和 Vue。

为什么会出现 Angular、React 和 Vue 这种库？

理解为什么会出现一种新技术，以及新技术解决了什么问题，才能够更好地选择和运用新技术，不至于落入「喜新厌旧」的怪圈。

首先在互联网用户界面和交互越来越复杂的阶段，这些 `MV*` 库是极大提升了开发效率，比如在数据流为主的后台系统，每天打交道最多的就是数据的增删改查，这时候如果使用这些库，可以将注意力转移到数据本身来，而不再是页面交互，从而极大地提升开发效率和沟通成本。

React 还有个很好的想法是 React Native，只需要写一套代码就可以实现 Web、安卓、iOS 三端相同的效果，但是在实际使用和开发中会有比较大的坑。而且就像 Node 一样，前端用 Node 写 Server 可能需要用到的后端知识要比前端知识多，想要写好 React Native，客户端的知识也是必不可少的。React Native 和 Node 都是拓展了 Web 前端工程师可以走的路，既可以向后又可以向前，所谓「全栈」。

Angular、React 和 Vue 各自的特点

- AngularJS 有着诸多特性，最为核心的是 MVVM、模块化、自动化双向数据绑定、语义化标签、依赖注入等
- React 是一个为数据提供渲染为 HTML 视图的开源 JavaScript 库，最大特点是引入 Virtual DOM，极大提升数据修改后 DOM 树的更新速度，而且也有 React Native 来做客户端开发
- Vue.js 作为后起前端框架，借鉴了 Angular、React 等现代前端框架/库的诸多特点，并取得了相当不错的成绩。

一定要用这些库吗？

目前这些库的确解决了实际开发中很多问题，但是这种「三足鼎立」的状况不是最终态，会是阶段性产物。从长远来说，好的想法和点子终究会体现在语言本身特性上来，即通过这些库的想法来推动标准的改进，比如 jQuery 的很多选择器 API，最终都被 CSS3 和 HTML5 接纳和实现，也就有了后来的 Zepto。

另外，以展现交互为主的项目 **不太推荐** 使用这类库，本身库的性能和体积就对页面造成极大的负担，比如笔者使用 Vue 做纯展现为主的项目，性能要比页面直出 HTML 慢。纯展现页面指的是那些以展现为主、用户交互少的页面，如文章列表页、文章详情页等。

如果是数据交互较多的页面，例如后台系统这类对性能要求不多而数据交互较多的页面，**推荐使用**。

另外，不管是什么库和框架，我们最终应该学习的是编程思维，比如分层、性能优化等，考虑视图层、组件化和工程效率问题。相信随着 ES 标准发展、摩尔定律（硬件）和浏览器的演进，目前这些问题和状况都会得到改善。

关于三者的学习资料就不补充了，因为实在是太火了，随便搜索一下就会找到。

移动开发

这里说的移动开发指的是做的项目是面向移动端的，比如 HTML5 页面、小程序等。做移动开发用的也是前面几个小节梳理的基础知识，唯一不同的是工程师面向的浏览器是移动端的浏览器或者固定的 Webview，所以会跟普通的 PC 开发有所不同。除了最基础的 JSBridge 概念之外，这里笔者重点列出以下几点：

1. 移动端更加注重性能和体验，因为移动端设备和网络都比 PC 的差一些
2. 交互跟 PC 不同，比如 touch 事件
3. 浏览器和固定的 Webview 带来了更多兼容性的问题，如微信 webview、安卓浏览器和 iOS 浏览器
4. 调试技巧更多，在 Chrome 内开发完页面，放到真机需要再调试一遍，或者需要真机配合才能实现页面的完整功能

后记

小册梳理了很多知识点，但是限于笔者精力、小册篇幅和新知识的不断涌现，难免会有考虑不到的地方，还请大家按照我在第一节提到的思维导图的方式，自己列脑图进行梳理。

最后，祝每个人都拿到满意的 offer！