

目录

| | |
|------------|------|
| 前言 | 1.1 |
| HTML面试题 | 1.2 |
| CSS面试题 | 1.3 |
| JS面试题 | 1.4 |
| HTTP面试题 | 1.5 |
| TCP面试题 | 1.6 |
| DOM面试题 | 1.7 |
| 浏览器原理面试题 | 1.8 |
| 前端工程化面试题 | 1.9 |
| React面试题 | 1.10 |
| Vue面试题 | 1.11 |
| 前端安全面试题 | 1.12 |
| webpack面试题 | 1.13 |
| 算法面试题 | 1.14 |
| 字符串类笔试题 | 1.15 |
| JS笔试题 | 1.16 |
| 如何应对HR面 | 1.17 |
| 如何应对项目面 | 1.18 |
| 如何写简历 | 1.19 |

面试官到底想看什么样的简历？

本手册是基于《前端面试与进阶指南》的精简版，可用于快速突击前端面试的知识点。

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



HTML基础

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

- doctype(文档类型) 的作用是什么?
- 这三种模式的区别是什么? (接上一问追问)
- HTML、XML 和 XHTML 有什么区别?
- 什么是data-属性?
- 你对HTML语义化的理解?
- HTML5与HTML4的不同之处
- 有哪些常用的meta标签?
- src和href的区别?
- 知道img的srcset的作用是什么? (追问)
- 还有哪一个标签能起到跟srcset相似作用? (追问)
- script标签中defer和async的区别?
- 有几种前端储存的方式?
- 这些方式的区别是什么? (追问)

本章是HTML考点的非重难点，因此我们采用简略回答的方式进行撰写，所以不会有太多详细的解释。

我们约定，每个问题后我们标记『 』的为高频面试题

doctype的作用是什么?

DOCTYPE是html5标准网页声明，且必须声明在HTML文档的第一行。来告知浏览器的解析器用什么文档标准解析这个文档，不同的渲染模式会影响到浏览器对于 CSS 代码甚至 JavaScript 脚本的解析

文档解析类型有：

- BackCompat：怪异模式，浏览器使用自己的怪异模式解析渲染页面。（如果没有声明DOCTYPE，默认就是这个模式）
- CSS1Compat：标准模式，浏览器使用W3C的标准解析渲染页面。

IE8还有一种介乎于上述两者之间的近乎标准的模式，但是基本淘汰了。

这三种模式的区别是什么?

- 标准模式(standards mode)：页面按照 HTML 与 CSS 的定义渲染
- 怪异模式(quirks mode)模式：会模拟更旧的浏览器的行为
- 近乎标准(almost standards)模式：会实施了一种表单元格尺寸的怪异行为（与IE7之前的单元格布局方式一致），除此之外符合标准定义

HTML、XHTML、XML有什么区别

- HTML(超文本标记语言): 在html4.0之前HTML先有实现再有标准，导致HTML非常混乱和松散
- XML(可扩展标记语言): 主要用于存储数据和结构，可扩展，大家熟悉的JSON也是相似的作用，但是更加轻量高效，所以XML现在市场越来越小了
- XHTML(可扩展超文本标记语言): 基于上面两者而来，W3C为了解决HTML混乱问题而生，并基于此诞生了HTML5，开头加入 <!DOCTYPE html> 的做法因此而来，如果不加就是兼容混乱的HTML，加了就是标准模式。

什么是data-属性？

HTML的数据属性，用于将数据储存于标准的HTML元素中作为额外信息,我们可以通过js访问并操作它，来达到操作数据的目的。

```
<article
  id="electriccars"
  data-columns="3"
  data-index-number="12314"
  data-parent="cars">
...
</article>
```

前端框架出现之后，这种方法已经不流行了

你对HTML语义化的理解？

语义化是指使用恰当语义的html标签，让页面具有良好的结构与含义，比如 `<p>` 标签就代表段落，`<article>` 代表正文内容等等。

语化化的好处主要有两点：

- 开发者友好：使用语义类标签增强了可读性，开发者也能够清晰地看出网页的结构，也更为便于团队的开发和维护
- 机器友好：带有语义的文字表现力丰富，更适合搜索引擎的爬虫爬取有效信息，语义类还可以支持读屏软件，根据文章可以自动生成目录

这对于简书、知乎这种富文本类的应用很重要，语义化对于其网站的内容传播有很大的帮助，但是对于功能性的web软件重要性大打折扣，比如一个按钮、Skeleton这种组件根本没有对应的语义，也不需要什么SEO。

HTML5与HTML4的不同之处

- 文件类型声明 (`<!DOCTYPE>`) 仅有一型：`<!DOCTYPE HTML>`。
- 新的解析顺序：不再基于SGML。
- 新的元素： `section`, `video`, `progress`, `nav`, `meter`, `time`, `aside`, `canvas`, `command`, `datalist`, `details`, `embed`, `figcaption`, `figure`, `footer`, `header`, `hgroup`, `keygen`, `mark`, `output`, `rp`, `rt`, `ruby`, `source`, `summary`, `wbr`。
- `input`元素的新类型：`date`, `email`, `url`等等。
- 新的属性：`ping`（用于`a`与`area`）, `charset`（用于`meta`）, `async`（用于`script`）。
- 全域属性：`id`, `tabindex`, `repeat`。
- 新的全域属性：`contenteditable`, `contextmenu`, `draggable`, `dropzone`, `hidden`, `spellcheck`。
- 移除元素：`acronym`, `applet`, `basefont`, `big`, `center`, `dir`, `font`, `frame`, `frameset`, `isindex`, `noframes`, `strike`, `tt`

有哪些常用的meta标签？

meta标签由name和content两个属性来定义，来描述一个HTML网页文档的属性，例如作者、日期和时间、网页描述、关键词、页面刷新等，除了一些http标准规定了一些name为大家使用的共识，开发者也可以自定义name。

- `charset`, 用于描述HTML文档的编码形式

```
<meta charset="UTF-8" >
```

- `http-equiv`, 顾名思义，相当于http的文件头作用,比如下面的代码就可以设置http的缓存过期日期

```
<meta http-equiv="expires" content="Wed, 20 Jun 2019 22:33:00 GMT">
```

- `viewport`, 移动前端最熟悉不过, Web开发人员可以控制视口的大小和比例

```
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
```

- `apple-mobile-web-app-status-bar-style`, 开发过PWA应用的开发者应该很熟悉, 为了自定义评估工具栏的颜色。

```
<meta name="apple-mobile-web-app-status-bar-style" content="black-translucent">
```

src和href的区别?

- `src`是指向外部资源的位置, 指向的内容会嵌入到文档中当前标签所在的位置, 在请求`src`资源时会将其指向的资源下载并应用到文档内, 如js脚本, `img`图片和`frame`等元素。当浏览器解析到该元素时, 会暂停其他资源的下载和处理, 知道将该资源加载、编译、执行完毕, 所以一般js脚本会放在底部而不是头部。
- `href`是指向网络资源所在位置 (的超链接), 用来建立和当前元素或文档之间的连接, 当浏览器识别到它他指向的文件时, 就会并行下载资源, 不会停止对当前文档的处理。

知道的srcset的作用是什么? (追问)

可以设计响应式图片, 我们可以使用两个新的属性`srcset` 和 `sizes`来提供更多额外的资源图像和提示, 帮助浏览器选择正确的一个资源。

`srcset` 定义了我们允许浏览器选择的图像集, 以及每个图像的大小。

`sizes` 定义了一组媒体条件 (例如屏幕宽度) 并且指明当某些媒体条件为真时, 什么样的图片尺寸是最佳选择。

所以, 有了这些属性, 浏览器会:

- 查看设备宽度
- 检查 `sizes` 列表中哪个媒体条件是第一个为真
- 查看给予该媒体查询的槽大小
- 加载 `srcset` 列表中引用的最接近所选的槽大小的图像

`srcset`提供了根据屏幕条件选取图片的能力

```

```

还有哪一个标签能起到跟srcset相似作用? (追问)

`<picture>` 元素通过包含零或多个 `<source>` 元素和一个 `` 元素来为不同的显示/设备场景提供图像版本。浏览器会选择最匹配的子 `<source>` 元素, 如果没有匹配的, 就选择 `` 元素的 `src` 属性中的URL。然后, 所选图像呈现在 `` 元素占据的空间中

`picture`同样可以通过不同设备来匹配不同的图像资源

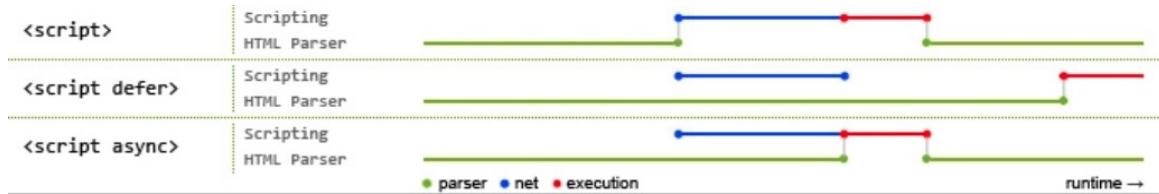
```
<picture>
```

```
<source srcset="/media/examples/surfer-240-200.jpg"
       media="(min-width: 800px)">

</picture>
```

script标签中defer和async的区别？

- **defer**: 浏览器指示脚本在文档被解析后执行，script被异步加载后并不会立刻执行，而是等待文档被解析完毕后执行。
- **async**: 同样是异步加载脚本，区别是脚本加载完毕后立即执行，这导致async属性下的脚本是乱序的，对于script有先后依赖关系的情况，并不适用。



蓝色线代表网络读取，红色线代表执行时间，这两都是针对脚本的；绿色线代表HTML解析

有几种前端储存的方式？

cookies、localStorage、sessionStorage、Web SQL、IndexedDB

这些方式的区别是什么？（追问）

- **cookies**: 在HTML5标准前本地储存的主要方式，优点是兼容性好，请求头自带cookie方便，缺点是大小只有4k，自动请求头加入cookie浪费流量，每个domain限制20个cookie，使用起来麻烦需要自行封装
- **localStorage**: HTML5加入的以键值对(Key-Value)为标准的方式，优点是操作方便，永久性储存（除非手动删除），大小为5M，兼容IE8+
- **sessionStorage**: 与localStorage基本类似，区别是sessionStorage当页面关闭后会被清理，而且与cookie、localStorage不同，他不能在所有同源窗口中共享，是会话级别的储存方式
- **Web SQL**: 2010年被W3C废弃的本地数据库数据存储方案，但是主流浏览器（火狐除外）都已经有了相关的实现，web sql类似于SQLite，是真正意义上的关系型数据库，用sql进行操作，当我们用JavaScript时要进行转换，较为繁琐。
- **IndexedDB**: 是被正式纳入HTML5标准的数据库储存方案，它是NoSQL数据库，用键值对进行储存，可以进行快速读取操作，非常适合web场景，同时用JavaScript进行操作会非常方便。

参考链接：

1. [src与href](#)
2. [语义化](#)
3. [defer和async的区别](#)
4. [响应式图片MDN](#)
5. [张鑫旭-srcset释义](#)
6. [picture元素-MDN](#)

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



CSS基础

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

- CSS选择器的优先级是怎样的?
- link和@import的区别?
- 有哪些方式 (CSS) 可以隐藏页面元素?
- em\px\rem区别?
- 块级元素水平居中的方法?
- CSS有几种定位方式?
- 如何理解z-index?
- 如何理解层叠上下文?
- 清除浮动有哪些方法?
- 你对css-sprites的理解
- 你对媒体查询的理解?
- 你对盒模型的理解?
- 标准盒模型和怪异盒模型有什么区别?
- 谈谈对BFC(Block Formatting Context)的理解?
- 为什么有时候人们用translate来改变位置而不是定位?
- 伪类和伪元素的区别是什么?
- 你对flex的理解?
- 关于CSS的动画与过渡问题

本章是CSS考点的非重难点，因此我们采用简略回答的方式进行撰写，所以不会有太多详细的解释。

我们约定，每个问题后我们标记『 』的为高频面试题

CSS选择器的优先级是怎样的?

CSS选择器的优先级是：内联 > ID选择器 > 类选择器 > 标签选择器

到具体的计算层面，优先级是由 A 、 B、 C、 D 的值来决定的，其中它们的值计算规则如下：

- A 的值等于 1 的前提是存在内联样式, 否则 A = 0;
- B 的值等于 ID选择器 出现的次数;
- C 的值等于 类选择器 和 属性选择器 和 伪类 出现的总次数;
- D 的值等于 标签选择器 和 伪元素 出现的总次数 。

就比如下面的选择器，它不存在内联样式，所以A=0,不存在id选择器B=0,存在一个类选择器C=1,存在三个标签选择器D=3，那么最终计算结果为: {0, 0, 1 ,3}

```
ul ol li .red {
  ...
}
```

按照这个结算方式，下面的计算结果为: {0, 1, 0, 0}

```
#red {
}
```

我们的比较优先级的方式是从A到D去比较值的大小，A、B、C、D权重从左到右，依次减小。判断优先级时，从左到右，一一比较，直到比较出最大值，即可停止。

比如第二个例子的B与第一个例子的B相比， $1 > 0$,接下来就不需要比较了，第二个选择器的优先级更高。

link和@import的区别？

- link属于XHTML标签，而@import是CSS提供的。
- 页面被加载时，link会同时被加载，而@import引用的CSS会等到页面被加载完再加载。
- import只在IE 5以上才能识别，而link是XHTML标签，无兼容问题。
- link方式的样式权重高于@import的权重。
- 使用dom控制样式时的差别。当使用javascript控制dom去改变样式的时候，只能使用link标签，因为@import不是dom可以控制的。

有哪些方式（CSS）可以隐藏页面元素？

- opacity:0：本质上是将元素的透明度将为0，就看起来隐藏了，但是依然占据空间且可以交互
- visibility:hidden：与上一个方法类似的效果，占据空间，但是不可以交互了
- overflow:hidden：这个只隐藏元素溢出的部分，但是占据空间且不可交互
- display:none：这个是彻底隐藏了元素，元素从文档流中消失，既不占据空间也不交互，也不影响布局
- z-index:-9999：原理是将层级放到底部，这样就被覆盖了，看起来隐藏了
- transform: scale(0,0)：平面变换，将元素缩放为0，但是依然占据空间，但不可交互

还有一些靠绝对定位把元素移到可视区域外，或者用clip-path进行裁剪的操作过于Hack，就不提了。

em\px\rem区别？

- px：绝对单位，页面按精确像素展示。
- em：相对单位，基准点为父节点字体的大小，如果自身定义了font-size按自身来计算（浏览器默认字体是16px），整个页面内1em不是一个固定的值。
- rem：相对单位，可理解为“root em”，相对根节点html的字体大小来计算，CSS3新加属性，chrome/firefox/IE9+支持

块级元素水平居中的方法？

如果使用Hack的话，水平居中的方法非常多，我们只介绍主流的，奇葩的见拓展阅读

margin:0 auto 方法

```
.center{
    height: 200px;
    width: 200px;
    margin: 0 auto;
    border: 1px solid red;
}
<div class="center">水平居中</div>
```

flex布局，目前主流方法

```
.center{
    display: flex;
    justify-content: center;
```

```

    }
<div class="center">
    <div class="flex-div">1</div>
    <div class="flex-div">2</div>
</div>

```

table方法

```

.center{
    display:table;
    margin:0 auto;
    border:1px solid red;
}
<div class="center">水平居中</div>

```

还有一些通过position+(margin|transform)等方法的不一样列举了，非重点没必要。

拓展阅读: [16种方法实现水平居中垂直居中](#)

CSS有几种定位方式？

- static: 正常文档流定位，此时 top, right, bottom, left 和 z-index 属性无效，块级元素从上往下纵向排布，行级元素从左向右排列。
- relative: 相对定位，此时的『相对』是相对于正常文档流的位置。
- absolute: 相对于最近的非 static 定位祖先元素的偏移，来确定元素位置，比如一个绝对定位元素它的父级、和祖父级元素都为relative，它会相对他的父级而产生偏移。
- fixed: 指定元素相对于屏幕视口（viewport）的位置来指定元素位置。元素的位置在屏幕滚动时不会改变，比如那种回到顶部的按钮一般都是用此定位方式。
- sticky: 粘性定位，特性近似于relative和fixed的合体，其在实际应用中的近似效果就是IOS通讯录滚动的时候的『顶屁股』。

文字描述很难理解，可以直接看代码

See the Pen [bPVNj](#) by Iwobi (@xiaomuzhu) on [CodePen](#).

如何理解z-index?

CSS 中的z-index属性控制重叠元素的垂直叠加顺序，默认元素的z-index为0，我们可以修改z-index来控制元素的图层位置，而且z-index只能影响设置了position值的元素。

我们可以把视图上的元素认为是一摞书的层叠，而人眼是俯视的视角，设置z-index的位置，就如同设置某一本在这摞书中的位置。

- 顶部: 最接近观察者
- ...
 - 3 层
 - 2 层
 - 1 层
 - 0 层 默认层
 - -1 层
 - -2 层
 - -3 层
- ...
- 底层: 距离观察者最远



可以结合这个例子理解z-index

See the Pen [xowqjG](#) by lwobi (@xiaomuzhu) on [CodePen](#).

如何理解层叠上下文？

是什么？

层叠上下文是HTML元素的三维概念，这些HTML元素在一条假想的相对于面向（电脑屏幕的）视窗或者网页的用户的z轴上延伸，HTML元素依据其自身属性按照优先级顺序占用层叠上下文的空间。

如何产生？

触发一下条件则会产生层叠上下文：

- 根元素 (HTML),
- z-index 值不为 "auto" 的绝对/相对定位,
- 一个 z-index 值不为 "auto" 的 flex 项目 (flex item), 即：父元素 display: flex|inline-flex,
- opacity 属性值小于 1 的元素 (参考 the specification for opacity) ,
- transform 属性值不为 "none" 的元素,
- mix-blend-mode 属性值不为 "normal" 的元素,
- filter 值不为 "none" 的元素,
- perspective 值不为 "none" 的元素,
- isolation 属性被设置为 "isolate" 的元素,
- position: fixed
- 在 will-change 中指定了任意 CSS 属性, 即便你没有直接指定这些属性的值 (参考 这篇文章)
- -webkit-overflow-scrolling 属性被设置 "touch" 的元素

拓展阅读：[层叠上下文-张鑫旭](#)

清除浮动有哪些方法？

- 空div方法：`<div style="clear:both;"></div>`
- Clearfix 方法：上文使用.clearfix类已经提到
- overflow: auto或overflow: hidden方法，使用BFC

在flex已经成为布局主流之后，浮动这种东西越来越少见了，毕竟它的副作用太大

你对css sprites的理解，好处是什么？

是什么？

雪碧图也叫CSS精灵，是一CSS图像合成技术，开发人员往往将小图标合并在一起之后的图片称作雪碧图。

如何操作？

使用工具（PS之类的）将多张图片打包成一张雪碧图，并为其生成合适的 CSS。每张图片都有相应的 CSS 类，该类定义了background-image、background-position和background-size属性。使用图片时，将相应的类添加到你的元素中。

好处：

- 减少加载多张图片的 HTTP 请求数（一张雪碧图只需要一个请求）
- 提前加载资源

不足：

- CSS Sprite维护成本较高，如果页面背景有少许改动，一般就要改这张合并的图片
- 加载速度优势在http2开启后荡然无存，HTTP2多路复用，多张图片也可以重复利用一个连接通道搞定

你对媒体查询的理解？

是什么

媒体查询由一个可选的媒体类型和零个或多个使用媒体功能的限制了样式表范围的表达式组成，例如宽度、高度和颜色。媒体查询，添加自CSS3，允许内容的呈现针对一个特定范围的输出设备而进行裁剪，而不必改变内容本身，非常适合web网页应对不同型号的设备而做出对应的响应适配。

如何使用？

媒体查询包含一个可选的媒体类型和，满足CSS3规范的条件下，包含零个或多个表达式，这些表达式描述了媒体特征，最终会被解析为true或false。如果媒体查询中指定的媒体类型匹配展示文档所使用的设备类型，并且所有的表达式的值都是true，那么该媒体查询的结果为true.那么媒体查询内的样式将会生效。

```
<!-- link元素中的CSS媒体查询 -->
<link rel="stylesheet" media="(max-width: 800px)" href="example.css" />

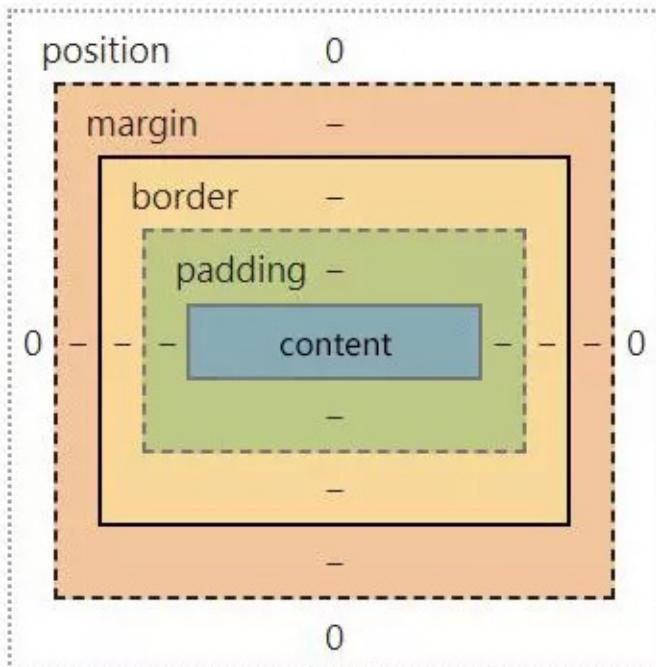
<!-- 样式表中的CSS媒体查询 -->
<style>
@media (max-width: 600px) {
    .facet_sidebar {
        display: none;
    }
}</style>
```

拓展阅读：[深入理解CSS Media媒体查询](#)

你对盒模型的理解

是什么？

当对一个文档进行布局（lay out）的时候，浏览器的渲染引擎会根据标准之一的 CSS 基础框盒模型（CSS basic box model），将所有元素表示为一个个矩形的盒子（box）。CSS 决定这些盒子的大小、位置以及属性（例如颜色、背景、边框尺寸...）。



盒模型由content（内容）、padding（内边距）、border（边框）、margin（外边距）组成。

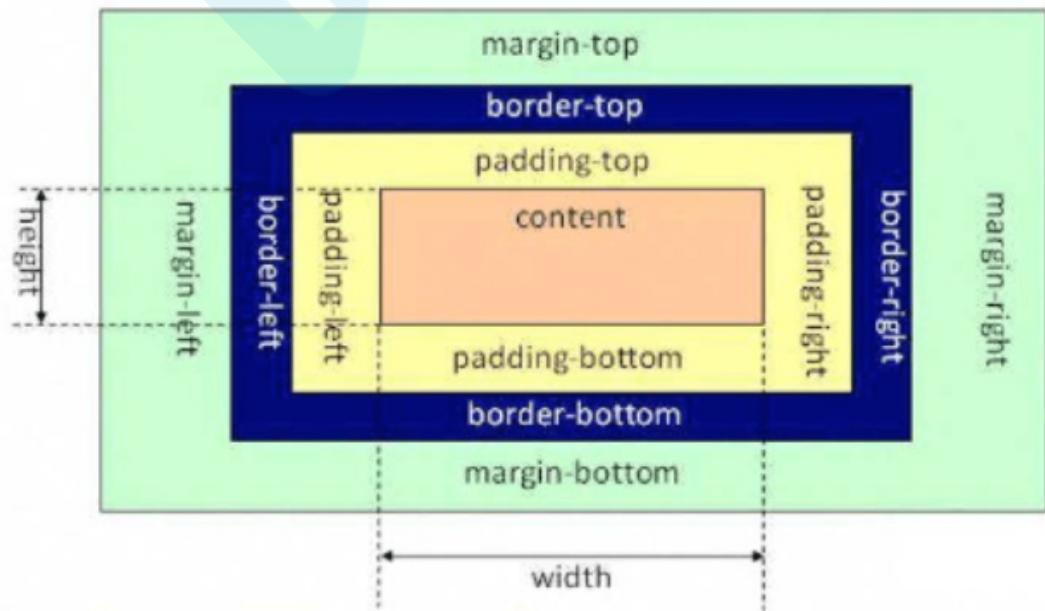
标准盒模型和怪异盒模型有什么区别？

在W3C标准下，我们定义元素的width值即为盒模型中的content的宽度值，height值即为盒模型中的content的高度值。

因此，标准盒模型下：

元素的宽度 = margin-left + border-left + padding-left + width + padding-right + border-right + margin-right

■ 标准盒子模型

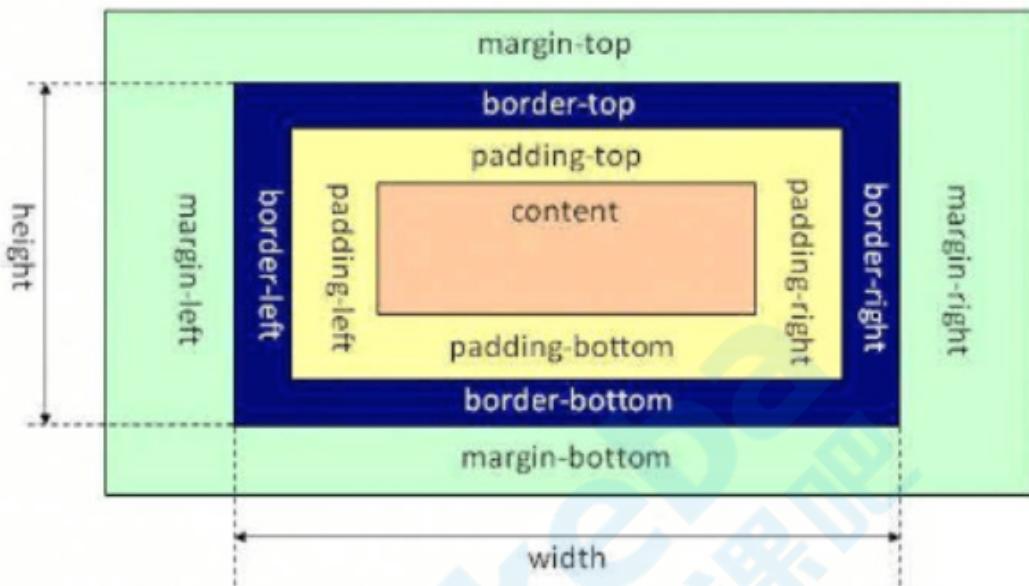


而IE怪异盒模型（IE8以下）width的宽度并不是content的宽度，而是border-left + padding-left + content的宽度值 + padding-right + border-right之和，height同理。

在怪异盒模型下：

元素占据的宽度 = margin-left + width + margin-right

■ IE盒子模型



虽然现代浏览器默认使用W3C的标准盒模型，但是在不少情况下怪异盒模型更好用，于是W3C在css3中加入 `box-sizing`。

```
box-sizing: content-box // 标准盒模型
box-sizing: border-box // 怪异盒模型
box-sizing: padding-box // 火狐的私有模型，没人用
```

此演示来源于拓展阅读文章

See the Pen [LKpyzz](#) by lwobi (@xiaomuzhu) on [CodePen](#).

拓展阅读：[深入理解盒模型](#)

谈谈对BFC的理解

是什么？

书面解释：BFC(Block Formatting Context)这几个英文拆解

- Box: CSS布局的基本单位，Box是CSS布局的对象和基本单位，直观点来说，就是一个页面是由很多个Box组成的，实际就是上个问题说的盒模型
- Formatting context: 块级上下文格式化，它是页面中的一块渲染区域，并且有一套渲染规则，它决定了其子元素将如何定位，以及和其他元素的关系和相互作用

简而言之，它是一块独立的区域，让处于BFC内部的元素与外部的元素互相隔离

如何形成？

BFC触发条件：

- 根元素，即HTML元素
- position: fixed/absolute
- float 不为none
- overflow不为visible
- display的值为inline-block、table-cell、table-caption

作用是什么？

防止margin发生重叠

See the Pen [NZGjYQ](#) by Iwobi (@xiaomuzhu) on CodePen.

两栏布局，防止文字环绕等

See the Pen [XLmRPM](#) by Iwobi (@xiaomuzhu) on CodePen.

防止元素塌陷

See the Pen [VJvbEd](#) by lwobi (@xiaomuzhu) on [CodePen](#).

拓展阅读: [深入理解BFC](#)

为什么有时候人们用translate来改变位置而不是定位?

translate()是transform的一个值。改变transform或opacity不会触发浏览器重新布局（reflow）或重绘（repaint），只会触发复合（compositions）。而改变绝对定位会触发重新布局，进而触发重绘和复合。transform使浏览器为元素创建一个GPU图层，但改变绝对定位会使用到CPU。因此translate()更高效，可以缩短平滑动画的绘制时间。

而translate改变位置时，元素依然会占据其原始空间，绝对定位就不会发生这种情况。

拓展阅读:[CSS3 3D transform变换-张鑫旭](#)

伪类和伪元素的区别是什么?

是什么?

伪类（pseudo-class）是一个以冒号(:)作为前缀，被添加到一个选择器末尾的关键字，当你希望样式在特定状态下才被呈现到指定的元素时，你可以往元素的选择器后面加上对应的伪类。

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

区别

其实上文已经表达清楚两者区别了，伪类是通过在元素选择器上加入伪类改变元素状态，而伪元素通过对元素的操作进行对元素的改变。

我们通过 `p::before` 对这段文本添加了额外的元素，通过 `p:first-child` 改变了文本的样式。

See the Pen [qzOXxO](#) by lwobi (@xiaomuzhu) on [CodePen](#).

| 拓展阅读：[伪类与伪元素](#)

你对flex的理解？

web应用有不同设备尺寸和分辨率，这时需要响应式界面设计来满足复杂的布局需求，Flex弹性盒模型的优势在于开发人员只是声明布局应该具有的行为，而不需要给出具体的实现方式，浏览器负责完成实际布局，当布局涉及到不定宽度，分布对齐的场景时，就要优先考虑弹性盒布局

> 具体用法移步阮一峰的[flex语法](#)、[flex实战](#)，讲得非常通俗易懂，而且我们一两句话说不清楚。

关于CSS的动画与过渡问题

[深入理解CSS动画animation](#)

[深入理解CSS过渡transition](#)

参考资料：

1. [盒模型](#)

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》：配套于本指南的突击手册，关注公众号回复「fed」获取



JavaScript基础

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

终于到了大家最擅长的JavaScript部分, 相比于HTML和CSS笔者写起JavaScript要顺手很多, 虽然前端有三剑客的说法, 但是实际应用中基本就是JavaScript为绝对主导, 尤其是在工程化的今天。

所以JavaScript才是前端基础面试中的重中之重, 在这部分我们会加入一个新的部分就是原理性的解释。

比如, 我们会有一个面试问题『解释下变量提升?』, 在本章下我们会有一个简短的解释, 但是不会解释原理性的东西, 因为『简短的解释』是给面试官听的, 『原理性的』是给自己解释的, 原理性的解释会在相关问题下连接到其他各个原理性详解的章节。

再说一下为什么会有『原理详解』这一part, 本项目并不仅想作为面试季帮助大家突击的一个清单, 更想做的是帮助大家梳理前端的各个知识点, 并把知识点讲透彻, 这才是真正对每个开发者有成长的事情。

此外, 如果不懂原理, 很容易被较真的面试官追问, 一下就原形毕露了, 所以如果你不懂原理, 建议阅读原理部分, 如果你已经懂了, 可以看简答部分作为梳理即可。

我们约定, 每个问题后我们标记『 』的为高频面试题

本章索引

- - 1. js基础
 - 谈谈你对原型链的理解?
 - 如何判断是否是数组?
 - ES6模块与CommonJS模块有什么区别?
 - 聊一聊如何在JavaScript中实现不可变对象?
 - JavaScript的参数是按照什么方式传递的?
 - js有哪些类型?
 - 为什么会有BigInt的提案?
 - null与undefined的区别是什么?
 - 0.1+0.2为什么不等于0.3?
 - 类型转换的规则有哪些?
 - 类型转换的原理是什么?
- - 1. js机制
 - 解释下变量提升?
 - 一段JavaScript代码是如何执行的?
 - JavaScript的作用域链理解吗?
 - 谈一谈你对this的了解?
 - 箭头函数的this指向哪里?
 - 理解闭包吗?
- - 1. js内存
 - 讲讲JavaScript垃圾回收是怎么做的?
 - JavaScript的基本类型和复杂类型是储存在哪里的?
- - 1. 异步
 - async/await 是什么?

- `async/await` 相比于Promise的优势?

解释下变量提升?

JavaScript引擎的工作方式是，先解析代码，获取所有被声明的变量，然后再一行一行地运行。这造成的结果，就是所有的变量的声明语句，都会被提升到代码的头部，这就叫做变量提升 (hoisting)。

```
console.log(a) // undefined

var a = 1

function b() {
    console.log(a)
}
b() // 1
```

上面的代码实际执行顺序是这样的：

第一步：引擎将 `var a = 1` 拆解为 `var a = undefined` 和 `a = 1`，并将 `var a = undefined` 放到最顶端，`a = 1` 还在原来的位置

这样一来代码就是这样：

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

a = 1

function b() {
    console.log(a)
}
b() // 1
```

第二步就是执行，因此js引擎一行一行从上往下执行就造成了当前的结果，这就叫变量提升。

原理详解请移步[,预解释与变量提升](#)

一段JavaScript代码是如何执行的?

此部分涉及概念较多，请移步[JavaScript执行机制](#)

理解闭包吗?

这个问题其实在问：

1. 闭包是什么？
2. 闭包有什么作用？

闭包是什么

MDN的解释：闭包是函数和声明该函数的词法环境的组合。

按照我的理解就是：闭包 = 『函数』 和 『函数体内可访问的变量总和』

举个简单的例子：

```
(function() {
    var a = 1;
    function add() {
        var b = 2

        var sum = b + a
        console.log(sum); // 3
    }
    add()
})()
```

add 函数本身，以及其内部可访问的变量，即 `a = 1`，这两个组合在一起就被称为闭包，仅此而已。

闭包的作用

闭包最大的作用就是隐藏变量，闭包的一大特性就是内部函数总是可以访问其所在的外部函数中声明的参数和变量，即使在其外部函数被返回（寿命终结）了之后

基于此特性，JavaScript可以实现私有变量、特权变量、储存变量等

我们就以私有变量举例，私有变量的实现方法很多，有靠约定的（变量名前加_），有靠Proxy代理的，也有靠Symbol这种新数据类型的。

但是真正广泛流行的其实是使用闭包。

```
function Person(){
    var name = 'cxk';
    this.getName = function(){
        return name;
    }
    this.setName = function(value){
        name = value;
    }
}

const cxk = new Person()

console.log(cxk.getName()) //cxk
cxk.setName('jntm')
console.log(cxk.getName()) //jntm
console.log(name) //name is not defined
```

函数体内的 `var name = 'cxk'` 只有 `getName` 和 `setName` 两个函数可以访问，外部无法访问，相对于将变量私有化。

JavaScript的作用域链理解吗？

JavaScript属于静态作用域，即声明的作用域是根据程序正文在编译时就确定的，有时也称为词法作用域。

其本质是JavaScript在执行过程中会创造可执行上下文，可执行上下文中的词法环境中含有外部词法环境的引用，我们可以通过这个引用获取外部词法环境的变量、声明等，这些引用串联起来一直指向全局的词法环境，因此形成了作用域链。



原理详解请移步[JavaScript执行机制](#)

ES6模块与CommonJS模块有什么区别?

ES6 Module和CommonJS模块的区别：

- CommonJS是对模块的浅拷贝，ES6 Module是对模块的引用,即ES6 Module只存只读，不能改变其值，具体点就是指针指向不能变，类似const
- import的接口是read-only（只读状态），不能修改其变量值。即不能修改其变量的指针指向，但可以改变变量内部指针指向,可以对commonJS对重新赋值（改变指针指向），但是对ES6 Module赋值会编译报错。

ES6 Module和CommonJS模块的共同点：

- CommonJS和ES6 Module都可以对引入的对象进行赋值，即对对象内部属性的值进行改变。

详解请移步[ES6模块与CommonJS模块的差异](#)

js有哪些类型?

JavaScript的类型分为两大类，一类是原始类型，一类是复杂(引用) 类型。

原始类型:

- boolean
- null
- undefined
- number
- string
- symbol

复杂类型:

- Object

还有一个没有正式发布但即将被加入标准的原始类型BigInt。

为什么会有BigInt的提案?

JavaScript中Number.MAX_SAFE_INTEGER表示最大安全数字,计算结果是9007199254740991，即在这个数范围内不会出现精度丢失（小数除外）。

但是一旦超过这个范围，js就会出现计算不准确的情况，这在大数计算的时候不得不依靠一些第三方库进行解决，因此官方提出了BigInt来解决此问题。

null与undefined的区别是什么?

null表示为空，代表此处不应该有值的存在，一个对象可以是null，代表是个空对象，而null本身也是对象。

undefined表示『不存在』，JavaScript是一门动态类型语言，成员除了表示存在的空值外，还有可能根本就不存在（因为不存在只在运行期才知道），这就是undefined的意义所在。

0.1+0.2为什么不等于0.3?

```
> 0.1 + 0.2
< 0.30000000000000004
```

JS 的 Number 类型遵循的是 IEEE 754 标准，使用的是 64 位固定长度来表示。

IEEE 754 浮点数由三个域组成，分别为 sign bit (符号位)、exponent bias (指数偏移值) 和 fraction (分数值)。64 位中，sign bit 占 1 位，exponent bias 占 11 位，fraction 占 52 位。

通过公式表示浮点数的值 **value = sign x exponent x fraction**

**

当一个数为正数，sign bit 为 0，当为负数时，sign bit 为 1.

以 0.1 转换为 IEEE 754 标准表示为例解释一下如何求 exponent bias 和 fraction。转换过程主要经历 3 个过程：

1. 将 0.1 转换为二进制表示
2. 将转换后的二进制通过科学计数法表示
3. 将通过科学计数法表示的二进制转换为 IEEE 754 标准表示

将 0.1 转换为二进制表示

回顾一下一个数的小数部分如何转换为二进制。一个数的小数部分，乘以 2，然后取整数部分的结果，再用计算后的小数部分重复计算，直到小数部分为 0。

因此 0.1 转换为二进制表示的过程如下：

| 小数 | x2 的结果 | 整数部分 |
|-----|--------|------|
| 0.1 | 0.2 | 0 |
| 0.2 | 0.4 | 0 |
| 0.4 | 0.8 | 0 |
| 0.8 | 1.6 | 1 |
| 0.6 | 1.2 | 1 |
| 0.2 | 0.4 | 0 |
| 0.4 | 0.8 | 0 |
| 0.8 | 1.6 | 1 |
| 0.6 | 1.2 | 1 |
| ... | ... | ... |

得到 0.1 的二进制表示为 0.00011... (无限重复 0011)

通过科学计数法表示

0.00011... (无限重复 0011) 通过科学计数法表示则是 $1.10011001... (\text{无限重复 } 1001) \times 2$

转换为 IEEE 754 标准表示

当经过科学计数法表示之后，就可以求得 exponent bias 和 fraction 了。

exponent bias (指数偏移值) 等于 双精度浮点数固定偏移值 (2-1) 加上指数实际值(即 2 中的 -4) 的 11 位二进制表示。为什么是 11 位？因为 exponent bias 在 64 位中占 11 位。

因此 0.1 的 exponent bias 等于 $1023 + (-4) = 1019$ 的 11 位二进制表示，即 011 1111 1011。

再来获取 0.1 的 fraction，fraction 就是 $1.10011001... (\text{无限重复 } 1001)$ 中的小数位，由于 fraction 占 52 位所以抽取 52 位小数，1001... (中间有 11 个 1001)...1010 (请注意最后四位，是 1010 而不是 1001，因为四舍五入有进位，这个进位就是造成 0.1 + 0.2 不等于 0.3 的原因)

```
0      011 1111 1011  1001... ( 11 x 1001 ) ... 1010
(sign bit) (exponent bias)      (fraction)
```

此时如果将这个数转换为十进制，可以发现值已经变为 0.10000000000000005551115123126 而不是 0.1 了，因此这个计算精度就出现了问题。

类型转换的规则有哪些？

在 if 语句、逻辑语句、数学运算逻辑、== 等情况下都可能出现隐式类型转换。

| 原始值 | 转化为数值类型 | 转化为字符串类型 | 转化为 Boolean 类型 |
|------------------|-----------|-------------------|----------------|
| false | 0 | "false" | false |
| true | 1 | "true" | true |
| 0 | 0 | "0" | false |
| 1 | 1 | "1" | true |
| "0" | 0 | "0" | true |
| "1" | 1 | "1" | true |
| NaN | NaN | "NaN" | false |
| Infinity | Infinity | "Infinity" | true |
| -Infinity | -Infinity | "-Infinity" | true |
| "" | 0 | "" | false |
| "20" | 20 | "20" | true |
| "twenty" | NaN | "twenty" | true |
| [] | 0 | "" | true |
| [20] | 20 | "20" | true |
| [10,20] | NaN | "10,20" | true |
| ["twenty"] | NaN | "twenty" | true |
| ["ten","twenty"] | NaN | "ten,twenty" | true |
| function(){} | NaN | "function(){}" | true |
| {} | NaN | "[object Object]" | true |
| null | 0 | "null" | false |
| undefined | NaN | "undefined" | false |

类型转换的原理是什么？

类型转换指的是将一种类型转换为另一种类型,例如:

```
var b = 2;
var a = String(b);
console.log(typeof a); //string
```

当然,类型转换分为显式和隐式,但是不管是隐式转换还是显式转换,都会遵循一定的原理,由于JavaScript是一门动态类型的语言,可以随时赋予任意值,但是各种运算符或条件判断中是需要特定类型的,因此JavaScript引擎会在运算时为变量设定类型.

这看起来很美好,JavaScript引擎帮我们搞定了 `类型` 的问题,但是引擎毕竟不是ASI(超级人工智能),它的很多动作会跟我们预期相去甚远,我们可以从一到面试题开始.

```
{ } + [] //0
```

答案是0

是什么原因造成了上述结果呢?那么我们得从ECMA-262中提到的转换规则和抽象操作说起,有兴趣的童鞋可以仔细阅读下这浩如烟海的[语言规范](#),如果没这个耐心还是往下看.

这是JavaScript中类型转换可以从原始类型转为引用类型,同样可以将引用类型转为原始类型,转为原始类型的抽象操作为 `ToPrimitive`,而后续更加细分的操作为: `ToNumber` `ToBoolean`。

为了更深入的探究JavaScript引擎是如何处理代码中类型转换问题的,就需要看 ECMA-262详细的规范,从而探究其内部原理,我们从这段内部原理示意代码开始.

```
// ECMA-262, section 9.1, page 30. Use null/undefined for no hint,
// (1) for number hint, and (2) for string hint.
function ToPrimitive(x, hint) {
    // Fast case check.
    if (IS_STRING(x)) return x;
    // Normal behavior.
    if (!IS_SPEC_OBJECT(x)) return x;
    if (IS_SYMBOL_WRAPPER(x)) throw MakeTypeError(kSymbolToPrimitive);
    if (hint == NO_HINT) hint = (IS_DATE(x)) ? STRING_HINT : NUMBER_HINT;
    return (hint == NUMBER_HINT) ? DefaultNumber(x) : DefaultString(x);
}

// ECMA-262, section 8.6.2.6, page 28.
function DefaultNumber(x) {
    if (!IS_SYMBOL_WRAPPER(x)) {
        var valueOf = x.valueOf;
        if (IS_SPEC_FUNCTION(valueOf)) {
            var v = %_CallFunction(x, valueOf);
            if (IsPrimitive(v)) return v;
        }

        var toString = x.toString;
        if (IS_SPEC_FUNCTION(toString)) {
            var s = %_CallFunction(x, toString);
            if (IsPrimitive(s)) return s;
        }
    }
    throw MakeTypeError(kCannotConvertToPrimitive);
}

// ECMA-262, section 8.6.2.6, page 28.
function DefaultString(x) {
    if (!IS_SYMBOL_WRAPPER(x)) {
        var toString = x.toString;
        if (IS_SPEC_FUNCTION(toString)) {
            var s = %_CallFunction(x, toString);
            if (IsPrimitive(s)) return s;
        }

        var valueOf = x.valueOf;
        if (IS_SPEC_FUNCTION(valueOf)) {
            var v = %_CallFunction(x, valueOf);
            if (IsPrimitive(v)) return v;
        }
    }
    throw MakeTypeError(kCannotConvertToPrimitive);
}
```

上面代码的逻辑是这样的:

1. 如果变量为字符串, 直接返回.
2. 如果 `!IS_SPEC_OBJECT(x)`, 直接返回.
3. 如果 `IS_SYMBOL_WRAPPER(x)`, 则抛出异常.
4. 否则会根据传入的 `hint` 来调用 `DefaultNumber` 和 `DefaultString`, 比如如果为 `Date` 对象, 会调用 `DefaultString`.
5. `DefaultNumber`: 首先 `x.valueOf`, 如果为 `primitive`, 则返回 `valueOf` 后的值, 否则继续调用 `x.toString`, 如果为 `primitive`, 则返回 `toString` 后的值, 否则抛出异常
6. `DefaultString`: 和 `DefaultNumber` 正好相反, 先调用 `toString`, 如果不是 `primitive` 再调用 `valueOf`.

那讲了实现原理，这个 `ToPrimitive` 有什么用呢？实际很多操作会调用 `ToPrimitive`，比如加、相等或比较操。在进行加操作时会将左右操作数转换为 `primitive`，然后进行相加。

下面来个实例，`({}) + 1`（将`{}`放在括号中是为了内核将其认为一个代码块）会输出啥？可能日常写代码并不会这样写，不过网上出过类似的面试题。

加操作只有左右运算符同时为 `String` 或 `Number` 时会执行对应的 `%_StringAdd` 或 `%NumberAdd`，下面看下 `({}) + 1` 内部会经过哪些步骤：

`{}` 和 `1` 首先会调用 `ToPrimitive`，`{}` 会走到 `DefaultNumber`，首先会调用 `valueOf`，返回的是 `Object { }`，不是 `primitive` 类型，从而继续走到 `toString`，返回 `[object Object]`，是 `string` 类型最后加操作，结果为 `[object object]1` 再比如有人问你 `[] + 1` 输出啥时，你可能知道应该怎么去计算了，先对 `[]` 调用 `ToPrimitive`，返回空字符串，最后结果为“1”。

谈谈你对原型链的理解？

这个问题关键在于两个点，一个是原型对象是什么，另一个是原型链是如何形成的

原型对象

绝大部分的函数(少数内建函数除外)都有一个 `prototype` 属性,这个属性是原型对象用来创建新对象实例,而所有被创建的对象都会共享原型对象,因此这些对象便可以访问原型对象的属性。

例如 `hasOwnProperty()` 方法存在于 `Object` 原型对象中,它便可以被任何对象当做自己的方法使用.

用法：`object.hasOwnProperty(propertyName)`

`hasOwnProperty()` 函数的返回值为 `Boolean` 类型。如果对象 `object` 具有名称为 `propertyName` 的属性，则返回 `true`，否则返回 `false`。

```
var person = {
  name: "Messi",
  age: 29,
  profession: "football player"
};
console.log(person.hasOwnProperty("name")); //true
console.log(person.hasOwnProperty("hasOwnProperty")); //false
console.log(Object.prototype.hasOwnProperty("hasOwnProperty")); //true
```

由以上代码可知, `hasOwnProperty()` 并不存在于 `person` 对象中,但是 `person` 依然可以拥有此方法.

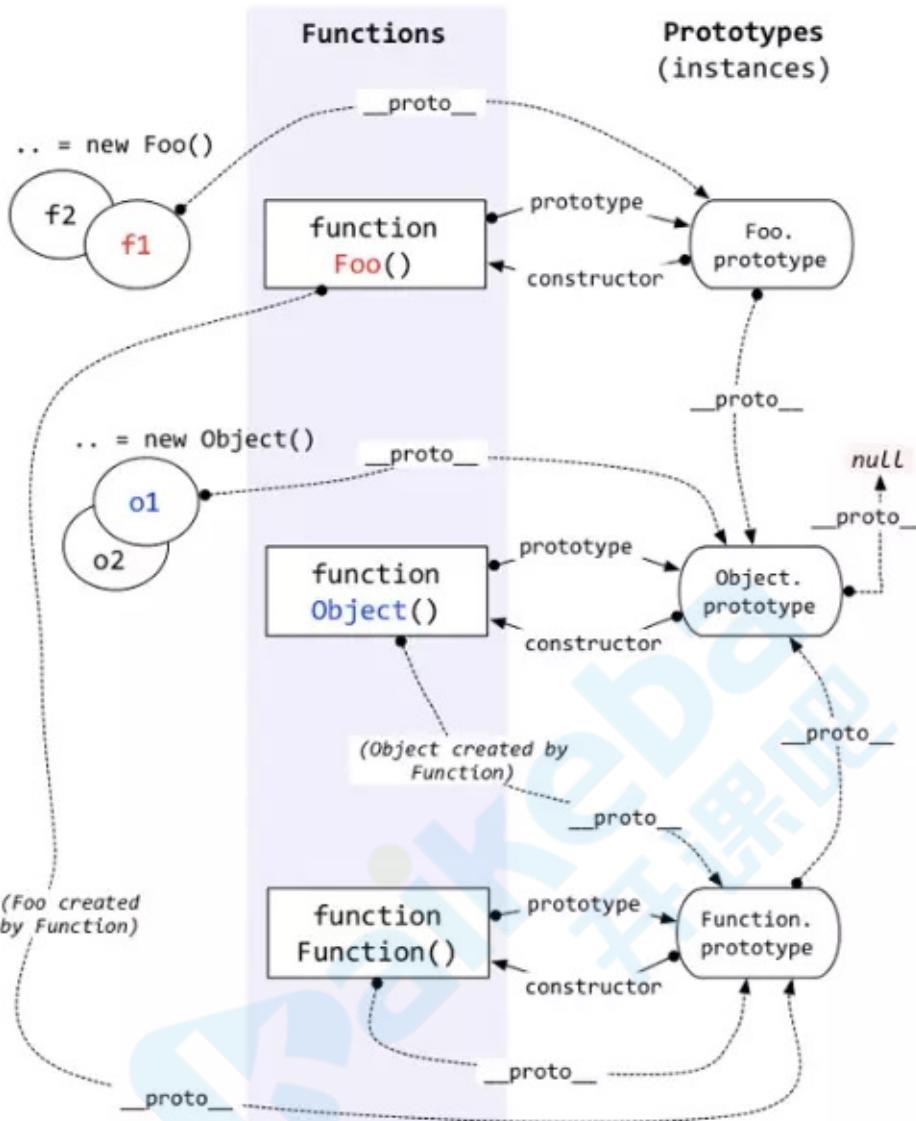
所以 `person` 对象是如何找到 `Object` 对象中的方法的呢?靠的是原型链。

原型链

原因是每个对象都有 `__proto__` 属性，此属性指向该对象的构造函数的原型。

对象可以通过 `__proto__` 与上游的构造函数的原型对象连接起来，而上游的原型对象也有一个 `__proto__`，这样就形成了原型链。

经典原型链图



如何判断是否是数组?

es6中加入了新的判断方法

```
if (Array.isArray(value)) {
    return true;
}
```

在考虑兼容性的情况下可以用`toString`的方法

```
if(!Array.isArray){
    Array.isArray = function(arg){
        return Object.prototype.toString.call(arg)=='[object Array]'
    }
}
```

谈一谈你对this的了解？

this的指向不是在编写时确定的,而是在执行时确定的，同时，this不同的指向在于遵循了一定的规则。

首先，在默认情况下，this是指向全局对象的，比如在浏览器就是指向window。

```
name = "Bale";

function sayName () {
    console.log(this.name);
};

sayName(); //Bale
```

其次，如果函数被调用的位置存在上下文对象时，那么函数是被隐式绑定的。

```
function f() {
    console.log( this.name );
}

var obj = {
    name: "Messi",
    f: f
};

obj.f(); //被调用的位置恰好被对象obj拥有，因此结果是Messi
```

再次，显示改变this指向，常见的方法就是call、apply、bind

以bind为例：

```
function f() {
    console.log( this.name );
}
var obj = {
    name: "Messi",
};

var obj1 = {
    name: "Bale"
};

f.bind(obj)(); //Messi ,由于bind将obj绑定到f函数上后返回一个新函数，因此需要再在后面加上括号进行执行，这是bind与apply和call的区别
```

最后，也是优先级最高的绑定 new 绑定。

用 new 调用一个构造函数，会创建一个新对象，在创造这个新对象的过程中，新对象会自动绑定到Person对象的this上，那么 this 自然就指向这个新对象。

```
function Person(name) {
    this.name = name;
    console.log(name);
}

var person1 = new Person('Messi'); //Messi
```

绑定优先级: new绑定 > 显式绑定 > 隐式绑定 > 默认绑定

那么箭头函数的this指向哪里？

箭头函数不同于传统JavaScript中的函数，箭头函数并没有属于自己的this，它的所谓的this是捕获其所在上下文的 this 值，作为自己的 this 值，并且由于没有属于自己的this，而箭头函数是不会被new调用的，这个所谓的this也不会被改变。

我们可以用Babel理解一下箭头函数：

```
// ES6
const obj = {
  getArrow() {
    return () => {
      console.log(this === obj);
    };
  }
}
```

转化后

```
// ES5, 由 Babel 转译
var obj = {
  getArrow: function getArrow() {
    var _this = this;
    return function () {
      console.log(_this === obj);
    };
  }
};
```

async/await是什么？

async 函数，就是 Generator 函数的语法糖，它建立在Promises上，并且与所有现有的基于Promise的API兼容。

1. Async—声明一个异步函数(async function someName(){...})
2. 自动将常规函数转换成Promise，返回值也是一个Promise对象
3. 只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数
4. 异步函数内部可以使用await
 1. Await—暂停异步的功能执行(var result = await someAsyncCall();)）
 2. 放置在Promise调用之前，await强制其他代码等待，直到Promise完成并返回结果
 3. 只能与Promise一起使用，不适用与回调
 4. 只能在async函数内部使用

async/await相比于Promise的优势？

- 代码读起来更加同步，Promise虽然摆脱了回调地狱，但是then的链式调用也会带来额外的阅读负担
- Promise传递中间值非常麻烦，而async/await几乎是同步的写法，非常优雅
- 错误处理友好，async/await可以用成熟的try/catch，Promise的错误捕获非常冗余
- 调试友好，Promise的调试很差，由于没有代码块，你不能在一个返回表达式的箭头函数中设置断点，如果你在一个.then代码块中使用调试器的步进(step-over)功能，调试器并不会进入后续的.then代码块，因为调试器只能跟踪同步代码的『每一步』。

JavaScript的参数是按照什么方式传递的?

基本类型传递方式

由于js中存在复杂类型和基本类型,对于基本类型而言,是按值传递的.

```
var a = 1;
function test(x) {
  x = 10;
  console.log(x);
}
test(a); // 10
console.log(a); // 1
```

虽然在函数 `test` 中 `a` 被修改,并没有影响到外部 `a` 的值,基本类型是按值传递的.

复杂类型按引用传递?

我们将外部 `a` 作为一个对象传入 `test` 函数.

```
var a = {
  a: 1,
  b: 2
};
function test(x) {
  x.a = 10;
  console.log(x);
}
test(a); // { a: 10, b: 2 }
console.log(a); // { a: 10, b: 2 }
```

可以看到,在函数体内被修改的 `a` 对象也同时影响到了外部的 `a` 对象,可见复杂类型是按引用传递的.

可是如果再做一个实验:

```
var a = {
  a: 1,
  b: 2
};
function test(x) {
  x = 10;
  console.log(x);
}
test(a); // 10
console.log(a); // { a: 1, b: 2 }
```

外部的 `a` 并没有被修改,如果是按引用传递的话,由于共享同一个堆内存, `a` 在外部也会表现为 `10` 才对. 此时的复杂类型同时表现出了 按值传递 和 按引用传递 的特性.

按共享传递

复杂类型之所以会产生这种特性,原因就是在传递过程中,对象 `a` 先产生了一个 副本`a`,这个 副本`a` 并不是深克隆得到的 副本`a`, 副本`a` 地址同样指向对象 `a` 指向的堆内存.



因此在函数体中修改 `x=10` 只是修改了 副本`a`, `a` 对象没有变化. 但是如果修改了 `x.a=10` 是修改了两者指向的同一堆内存,此时对象 `a` 也会受到影响.

有人讲这种特性叫做传递引用,也有一种说法叫做按共享传递.

聊一聊如何在JavaScript中实现不可变对象?

实现不可变数据有三种主流的方法

1. 深克隆, 但是深克隆的性能非常差, 不适合大规模使用
2. Immutable.js, Immutable.js是自成一体的一套数据结构, 性能良好, 但是需要学习额外的API
3. immer, 利用Proxy特性, 无需学习额外的api, 性能良好

| 原理详解请移步[实现JavaScript不可变数据](#)

JavaScript的基本类型和复杂类型是储存在哪里的?

基本类型储存在栈中, 但是一旦被闭包引用则成为常住内存, 会储存在内存堆中。

复杂类型会储存在内存堆中。

| 原理解析请移步[JavaScript内存管理](#)

讲讲JavaScript垃圾回收是怎么做的?

此过程比较复杂, 请看详细解析。

| 原理解析请移步[JavaScript内存管理](#)

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



HTTP协议

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

HTTP有哪些方法?

- HTTP1.0定义了三种请求方法: GET, POST 和 HEAD方法
- HTTP1.1新增了五种请求方法: OPTIONS, PUT, DELETE, TRACE 和 CONNECT

这些方法的具体作用是什么?

- GET: 通常用于请求服务器发送某些资源
- HEAD: 请求资源的头部信息, 并且这些头部与 HTTP GET 方法请求时返回的一致. 该请求方法的一个使用场景是在下载一个大文件前先获取其大小再决定是否要下载, 以此可以节约带宽资源
- OPTIONS: 用于获取目的资源所支持的通信选项
- POST: 发送数据给服务器
- PUT: 用于新增资源或者使用请求中的有效负载替换目标资源的表现形式
- DELETE: 用于删除指定的资源
- PATCH: 用于对资源进行部分修改
- CONNECT: HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器
- TRACE: 回显服务器收到的请求, 主要用于测试或诊断

GET和POST有什么区别?

- 数据传输方式不同: GET请求通过URL传输数据, 而POST的数据通过请求体传输。
- 安全性不同: POST的数据因为在请求主体内, 所以有一定的安全性保证, 而GET的数据在URL中, 通过历史记录, 缓存很容易查到数据信息。
- 数据类型不同: GET只允许 ASCII 字符, 而POST无限制
- GET无害: 刷新、后退等浏览器操作GET请求是无害的, POST可能重复提交表单
- 特性不同: GET是安全 (这里的安全是指只读特性, 就是使用这个方法不会引起服务器状态变化) 且幂等 (幂等的概念是指同一个请求方法执行多次和仅执行一次的效果完全相同), 而POST是非安全非幂等

PUT和POST都是给服务器发送新增资源, 有什么区别?

PUT 和POST方法的区别是,PUT方法是幂等的: 连续调用一次或者多次的效果相同 (无副作用), 而POST方法是非幂等的。

除此之外还有一个区别, 通常情况下, PUT的URI指向是具体单一资源, 而POST可以指向资源集合。

举个例子, 我们在开发一个博客系统, 当我们要创建一篇文章的时候往往用 `POST https://www.jianshu.com/articles`, 这个请求的语义是, 在articles的资源集合下创建一篇新的文章, 如果我们多次提交这个请求会创建多个文章, 这是非幂等的。

而 `PUT https://www.jianshu.com/articles/820357430` 的语义是更新对应文章下的资源 (比如修改作者名称等), 这个URI指向的就是单一资源, 而且是幂等的, 比如你把『刘德华』修改成『蔡徐坤』, 提交多少次都是修改成『蔡徐坤』

ps: 『POST表示创建资源, PUT表示更新资源』这种说法是错误的, 两个都能创建资源, 根本区别就在于幂等性

PUT和PATCH都是给服务器发送修改资源，有什么区别？

PUT和PATCH都是更新资源，而PATCH用来对已知资源进行局部更新。

比如我们有一篇文章的地址 <https://www.jianshu.com/articles/820357430> ,这篇文章的可以表示为:

```
article = {
    author: 'dxy',
    creationDate: '2019-6-12',
    content: '我写文章像蔡徐坤',
    id: 820357430
}
```

当我们要修改文章的作者时，我们可以直接发送 `PUT https://www.jianshu.com/articles/820357430`，这个时候的数据应该是:

```
{
    author: '蔡徐坤',
    creationDate: '2019-6-12',
    content: '我写文章像蔡徐坤',
    id: 820357430
}
```

这种直接覆盖资源的修改方式应该用put，但是你觉得每次都带有这么多无用的信息，那么可以发送 `PATCH https://www.jianshu.com/articles/820357430`，这个时候只需要:

```
{
    author: '蔡徐坤',
}
```

http的请求报文是什么样的？

请求报文有4部分组成:

- 请求行
- 请求头部
- 空行
- 请求体



- 请求行包括：请求方法字段、URL字段、HTTP协议版本字段。它们用空格分隔。例如，`GET /index.html`
`HTTP/1.1.`
- 请求头部:请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔
- User-Agent: 产生请求的浏览器类型。

- Accept: 客户端可识别的内容类型列表。
- Host: 请求的主机名, 允许多个域名同处一个IP地址, 即虚拟主机。
- 请求体: post put等请求携带的数据

Request

Raw Params Headers Hex

请求行
POST /web5/index.php HTTP/1.1

请求头
Host: 123.206.87.240:8002
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:63.0) Gecko/20100101 Firefox/63.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Referer: http://123.206.87.240:8002/web5/
Content-Type: application/x-www-form-urlencoded
Content-Length: 22
Connection: close
Upgrade-Insecure-Requests: 1

flag=abc&submit=Submit

请求正文

http的响应报文是什么样的?

请求报文有4部分组成:

- 响应行
- 响应头
- 空行
- 响应体



- 响应行: 由协议版本, 状态码和状态码的原因短语组成, 例如 `HTTP/1.1 200 OK`。
- 响应头: 响应部首组成
- 响应体: 服务器响应的数据

聊一聊HTTP的部首有哪些?

内容很多，重点看标『』内容

通用首部字段（General Header Fields）：请求报文和响应报文两方都会使用的首部

- Cache-Control 控制缓存
- Connection 连接管理、逐条首部
- Upgrade 升级为其他协议
- via 代理服务器的相关信息
- Wrining 错误和警告通知
- Transfor-Encoding 报文主体的传输编码格式
- Trailer 报文末端的首部一览
- Pragma 报文指令
- Date 创建报文的日期

请求首部字段（Reauest Header Fields）：客户端向服务器发送请求的报文时使用的首部

- Accept 客户端或者代理能够处理的媒体类型
- Accept-Encoding 优先可处理的编码格式
- Accept-Language 优先可处理的自然语言
- Accept-Charset 优先可以处理的字符集
- If-Match 比较实体标记（ETage）
- If-None-Match 比较实体标记（ETage）与 If-Match相反
- If-Modified-Since 比较资源更新时间（Last-Modified）
- If-Unmodified-Since 比较资源更新时间（Last-Modified），与 If-Modified-Since相反
- If-Rnages 资源未更新时发送实体byte的范围请求
- Range 实体的字节范围请求
- Authorization web的认证信息
- Proxy-Authorization 代理服务器要求web认证信息
- Host 请求资源所在服务器
- From 用户的邮箱地址
- User-Agent 客户端程序信息
- Max-Forwrads 最大的逐跳次数
- TE 传输编码的优先级
- Referer 请求原始放的url
- Expect 期待服务器的特定行为

响应首部字段（Response Header Fields）：从服务器向客户端响应时使用的字段

- Accept-Ranges 能接受的字节范围
- Age 推算资源创建经过时间
- Location 令客户端重定向的URI
- vary 代理服务器的缓存信息
- ETag 能够表示资源唯一资源的字符串
- WWW-Authenticate 服务器要求客户端的验证信息
- Proxy-Authenticate 代理服务器要求客户端的验证信息
- Server 服务器的信息
- Retry-After 和状态码503一起使用的首部字段，表示下次请求服务器的时间

实体首部字段（Entiy Header Fields）：针对请求报文和响应报文的实体部分使用首部

- Allow 资源可支持http请求的方法
- Content-Language 实体的资源语言
- Content-Encoding 实体的编码格式
- Content-Length 实体的大小（字节）
- Content-Type 实体媒体类型

- Content-MD5 实体报文的摘要
- Content-Location 代替资源的uri
- Content-Ranges 实体主体的位置返回
- Last-Modified 资源最后的修改资源
- Expires 实体主体的过期资源

聊一聊HTTP的状态码有哪些？

2XX 成功

- 200 OK, 表示从客户端发来的请求在服务器端被正确处理
- 201 Created 请求已经被实现，而且有一个新的资源已经依据请求的需要而建立
- 202 Accepted 请求已接受，但是还没执行，不保证完成请求
- 204 No content, 表示请求成功，但响应报文不含实体的主体部分
- 206 Partial Content, 进行范围请求

3XX 重定向

- 301 moved permanently, 永久性重定向，表示资源已被分配了新的 URL
- 302 found, 临时性重定向，表示资源临时被分配了新的 URL
- 303 see other, 表示资源存在着另一个 URL，应使用 GET 方法获取资源
- 304 not modified, 表示服务器允许访问资源，但因发生请求未满足条件的情况
- 307 temporary redirect, 临时重定向，和302含义相同

4XX 客户端错误

- 400 bad request, 请求报文存在语法错误
- 401 unauthorized, 表示发送的请求需要有通过 HTTP 认证的认证信息
- 403 forbidden, 表示对请求资源的访问被服务器拒绝
- 404 not found, 表示在服务器上没有找到请求的资源
- 408 Request timeout, 客户端请求超时
- 409 Conflict, 请求的资源可能引起冲突

5XX 服务器错误

- 500 internal sever error, 表示服务器端在执行请求时发生了错误
- 501 Not Implemented 请求超出服务器能力范围，例如服务器不支持当前请求所需要的某个功能，或者请求是服务器不支持的某个方法
- 503 service unavailable, 表明服务器暂时处于超负载或正在停机维护，无法处理请求
- 505 http version not supported 服务器不支持，或者拒绝支持在请求中使用的 HTTP 版本

同样是重定向307, 303, 302的区别？

302是http1.0的协议状态码，在http1.1版本的时候为了细化302状态码又出来了两个303和307。

303明确表示客户端应当采用get方法获取资源，他会把POST请求变为GET请求进行重定向。307会遵照浏览器标准，不会从post变为get。

HTTP的keep-alive是什么的？

在早期的HTTP/1.0中，每次http请求都要创建一个连接，而创建连接的过程需要消耗资源和时间，为了减少资源消耗，缩短响应时间，就需要重用连接。在后来的HTTP/1.0以及HTTP/1.1中，引入了重用连接的机制，就是在http请求头中加入Connection: keep-alive来告诉对方这个请求响应完成后不要关闭，下一次咱们还用这个请求继续交流。协议规定

HTTP/1.0如果想要保持长连接，需要在请求头中加上Connection: keep-alive。

keep-alive的优点：

- 较少的CPU和内存的使用（由于同时打开的连接的减少了）
- 允许请求和应答的HTTP管线化
- 降低拥塞控制（TCP连接减少了）
- 减少了后续请求的延迟（无需再进行握手）
- 报告错误无需关闭TCP连

为什么有了HTTP为什么还要HTTPS？

https是安全版的http，因为http协议的数据都是明文进行传输的，所以对于一些敏感信息的传输就很不安全，HTTPS就是为了解决HTTP的不安全而生的。

HTTPS是如何保证安全的？

过程比较复杂，我们得先理解两个概念

对称加密：即通信的双方都使用同一个密钥进行加解密，比如特务接头的暗号，就属于对称加密

对称加密虽然很简单性能也好，但是无法解决首次把密钥发给对方的问题，很容易被黑客拦截密钥。

非对称加密：

1. 私钥 + 公钥 = 密钥对
2. 即用私钥加密的数据，只有对应的公钥才能解密，用公钥加密的数据，只有对应的私钥才能解密
3. 因为通信双方的手里都有一套自己的密钥对，通信之前双方会先把自己的公钥都先发给对方
4. 然后对方再拿着这个公钥来加密数据响应给对方，等到到了对方那里，对方再用自己的私钥进行解密

非对称加密虽然安全性更高，但是带来的问题是速度很慢，影响性能。

解决方案：

那么结合两种加密方式，将对称加密的密钥使用非对称加密的公钥进行加密，然后发送出去，接收方使用私钥进行解密得到对称加密的密钥，然后双方可以使用对称加密来进行沟通。

此时又带来一个问题，中间人问题：

如果此时在客户端和服务器之间存在一个中间人，这个中间人只需要把原本双方通信互发的公钥，换成自己的公钥，这样中间人就可以轻松解密通信双方所发送的所有数据。

所以这个时候需要一个安全的第三方颁发证书（CA），证明身份的身份，防止被中间人攻击。

证书中包括：签发者、证书用途、使用者公钥、使用者私钥、使用的HASH算法、证书到期时间等

连接是安全的

您发送给这个网站的信息（例如密码或信用卡号）不会外泄。[了解详情](#)

 证书 (有效)

[显示证书 / 由DigiCert SHA2 High Assurance](#)

但是问题来了，如果中间人篡改了证书，那么身份证明是不是就无效了？这个证明就白买了，这个时候需要一个新的技术，数字签名。

数字签名就是用CA自带的HASH算法对证书的内容进行HASH得到一个摘要，再用CA的私钥加密，最终组成数字签名。

当别人把他的证书发过来的时候，我再用同样的Hash算法，再次生成消息摘要，然后用CA的公钥对数字签名解密，得到CA创建的消息摘要，两者一比，就知道中间有没有被人篡改了。

这个时候就能最大程度保证通信的安全了。

HTTP2相对于HTTP1.x有什么优势和特点？

二进制分帧

帧：HTTP/2 数据通信的最小单位消息：指 HTTP/2 中逻辑上的 HTTP 消息。例如请求和响应等，消息由一个或多个帧组成。

流：存在于连接中的一个虚拟通道。流可以承载双向消息，每个流都有一个唯一的整数ID

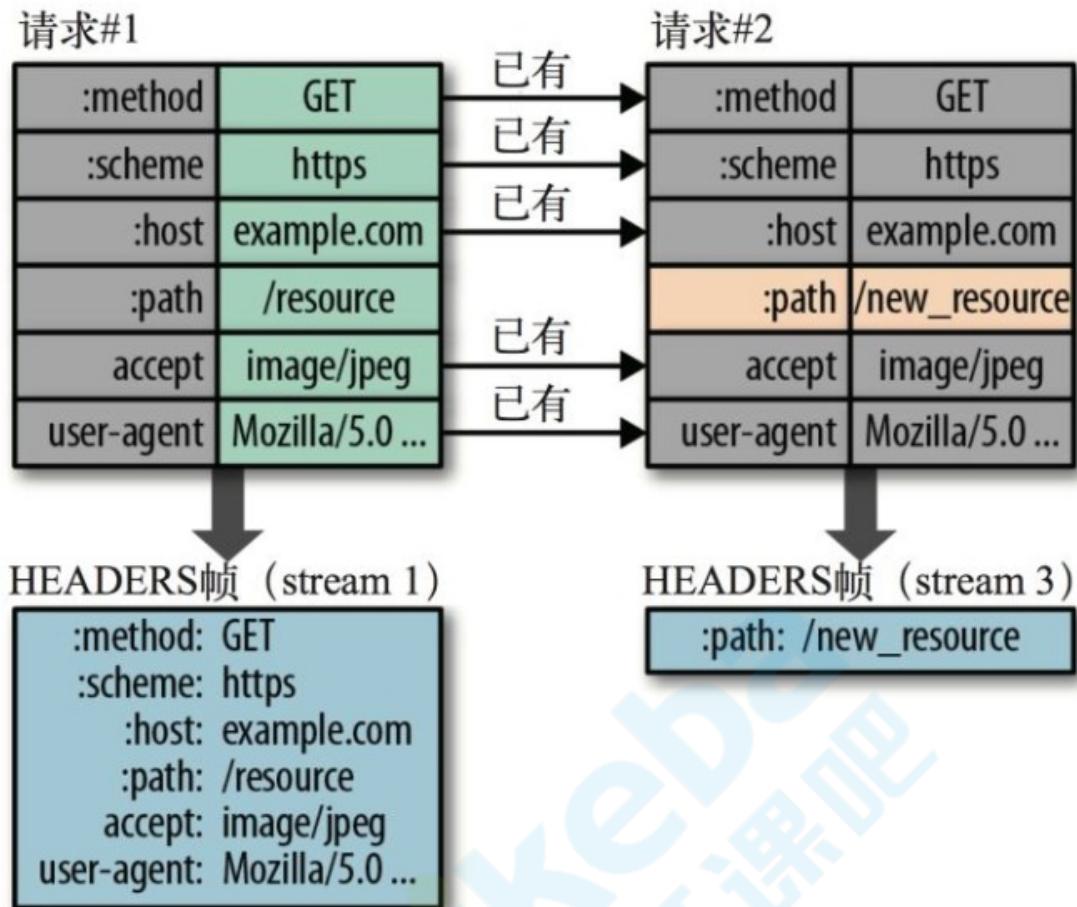
HTTP/2 采用二进制格式传输数据，而非 HTTP 1.x 的文本格式，二进制协议解析起来更高效。

头部压缩

HTTP/1.x会在请求和响应中重复地携带不常改变的、冗长的头部数据，给网络带来额外的负担。

- HTTP/2在客户端和服务端使用“首部表”来跟踪和存储之前发送的键-值对，对于相同的数据，不再通过每次请求和响应发送
- 首部表在HTTP/2的连接存续期内始终存在，由客户端和服务端共同渐进地更新；
- 每个新的首部键-值对要么被追加到当前表的末尾，要么替换表中之前的值。

你可以理解为只发送差异数据，而不是全部发送，从而减少头部的信息量



服务器推送

服务端可以在发送页面HTML时主动推送其它资源，而不用等到浏览器解析到相应位置，发起请求再响应。例如服务端可以主动把JS和CSS文件推送给客户端，而不需要客户端解析HTML时再发送这些请求。

服务端可以主动推送，客户端也有权利选择是否接收。如果服务端推送的资源已经被浏览器缓存过，浏览器可以通过发送RST_STREAM帧来拒收。主动推送也遵守同源策略，服务器不会随便推送第三方资源给客户端。

多路复用

HTTP 1.x 中，如果想并发多个请求，必须使用多个 TCP 链接，且浏览器为了控制资源，还会对单个域名有 6-8个的TCP链接请求限制。

HTTP2中：

- 同域名下所有通信都在单个连接上完成。
- 单个连接可以承载任意数量的双向数据流。
- 数据流以消息的形式发送，而消息又由一个或多个帧组成，多个帧之间可以乱序发送，因为根据帧首部的流标识可以重新组装

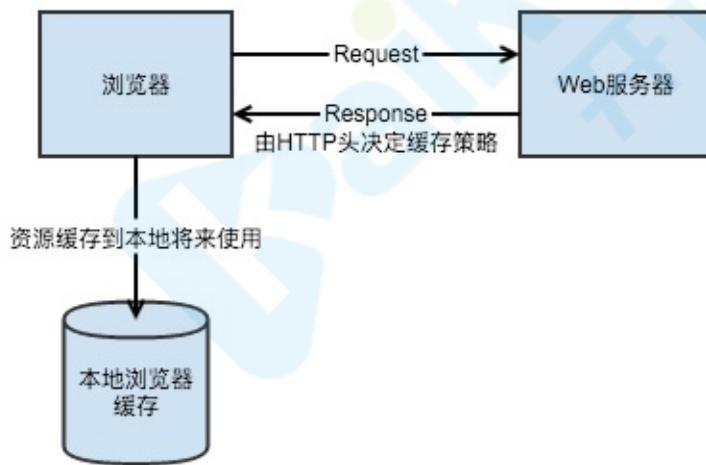
| | | | | | | | | | | | |
|--------------------|----------------------|----------------------|----|----------------------|-------------|----------------------|--------|----------------------|--|----------------------|--|
| | v2-20465930c... | 200 | h2 | jpeg | vendor.5... | 1.5 KB | 24 ... | Low | | | |
| | v2-026f5adaf7... | 200 | h2 | jpeg | vendor.5... | 2.0 KB | 23 ... | Low | | | |
| | a0471aa99_s.jpg | 200 | h2 | jpeg | vendor.5... | 1009 B | 24 ... | Low | | | |
| | da8e974dc_s.jpg | 200 | h2 | jpeg | vendor.5... | 829 B | 21 ... | Low | | | |
| | https://pic4.zhim... | 200 | h2 | jpeg | vendor.5... | 1.1 KB | 18 ... | Low | | | |
| | v2-2c3af13e32... | 200 | h2 | jpeg | vendor.5... | 1.3 KB | 19 ... | Low | | | |
| | v2-863cb45c9... | 200 | h2 | jpeg | vendor.5... | 1.2 KB | 31 ... | Low | | | |
| | v2-ab7f157fdc... | 200 | h2 | jpeg | vendor.5... | 1.1 KB | 17 ... | Low | | | |
| | 839d556b20f2... | 200 | h2 | jpeg | vendor.5... | 1.3 KB | 41 ... | Low | | | |
| 更多 | | 按列筛选 | | 按行筛选 | | 按列筛选 | | 按行筛选 | | 按列筛选 | |

拓展阅读: [HTTP/2特性及其在实际应用中的表现](#)

HTTP的缓存的过程是怎样的?

通常情况下的步骤是:

1. 客户端向服务器发出请求, 请求资源
2. 服务器返回资源, 并通过响应头决定缓存策略
3. 客户端根据响应头的策略决定是否缓存资源 (这里假设是), 并将响应头与资源缓存下来
4. 在客户端再次请求且命中资源的时候, 此时客户端去检查上次缓存的缓存策略, 根据策略的不同、是否过期等判断是直接读取本地缓存还是与服务器协商缓存



什么时候会触发强缓存或者协商缓存?

强缓存

强缓存离不开两个响应头 `Expires` 与 `Cache-Control`

- `Expires`: `Expires`是http1.0提出的一个表示资源过期时间的header, 它描述的是一个绝对时间, 由服务器返回, `Expires`受限于本地时间, 如果修改了本地时间, 可能会造成缓存失效

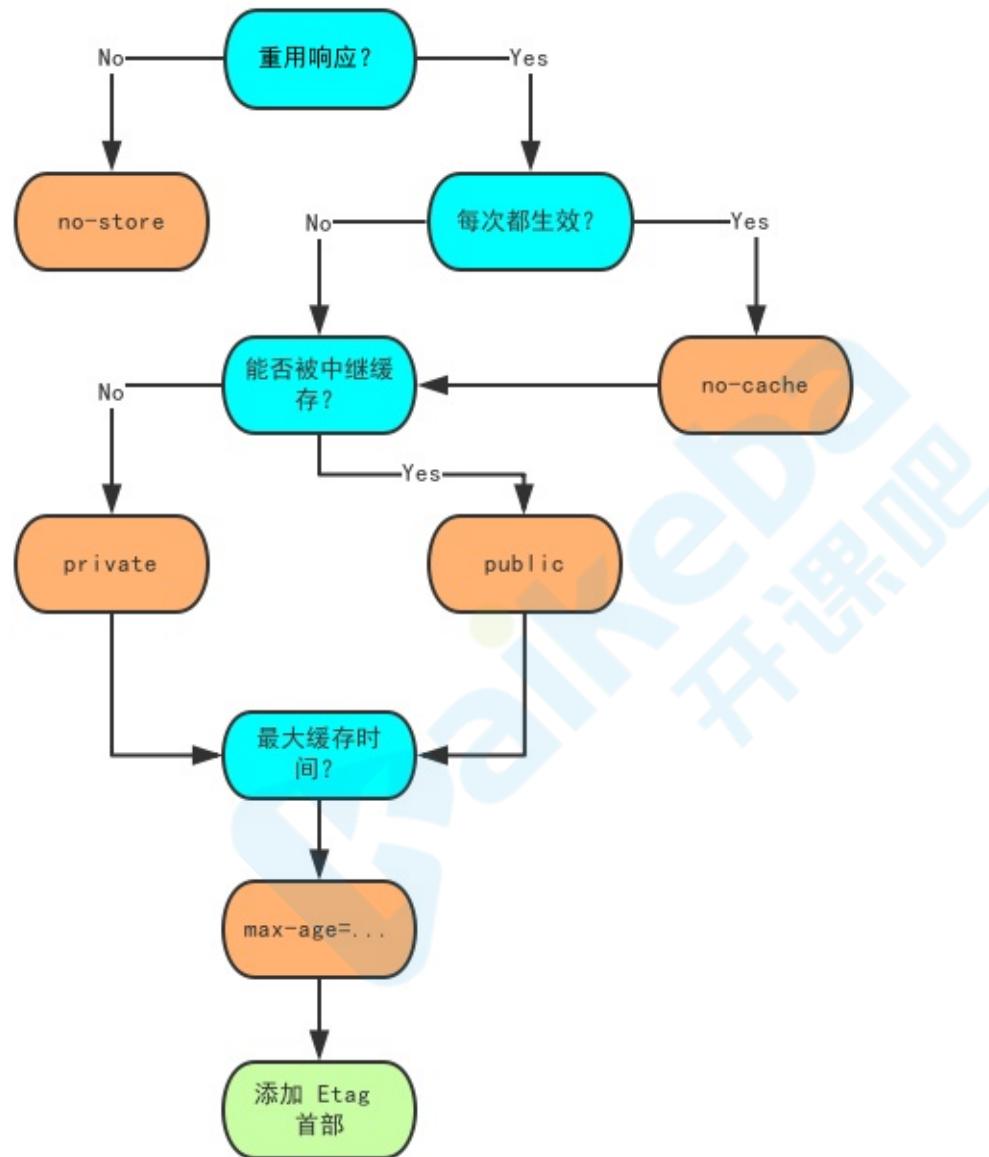
`Expires: Wed, 11 May 2018 07:20:00 GMT`

- `Cache-Control`: `Cache-Control` 出现于 HTTP / 1.1, 优先级高于 `Expires`, 表示的是相对时间

```
Cache-Control: max-age=315360000
```

目前主流的做法使用 Cache-Control 控制缓存，除了 max-age 控制过期时间外，还有一些不得不提

- Cache-Control: public 可以被所有用户缓存，包括终端和CDN等中间代理服务器
- Cache-Control: private 只能被终端浏览器缓存，不允许中继缓存服务器进行缓存
- Cache-Control: no-cache, 先缓存本地，但是在命中缓存之后必须与服务器验证缓存的新鲜度才能使用
- Cache-Control: no-store, 不会产生任何缓存



在缓存有效期内命中缓存，浏览器会直接读取本地的缓存资源，当缓存过期之后会与服务器进行协商。

协商缓存

当第一次请求时服务器返回的响应头中没有Cache-Control和Expires或者Cache-Control和Expires过期抑或它的属性设置为no-cache时，那么浏览器第二次请求时就会与服务器进行协商。

如果缓存和服务端资源的最新版本是一致的，那么就无需再次下载该资源，服务端直接返回304 Not Modified 状态码，如果服务器发现浏览器中的缓存已经是旧版本了，那么服务器就会把最新资源的完整内容返回给浏览器，状态码就是200 Ok。

服务器判断缓存是否是新鲜的方法就是依靠HTTP的另外两组信息

Last-Modified/If-Modified-Since

客户端首次请求资源时，服务器会把资源的最新修改时间 `Last-Modified:Thu, 19 Feb 2019 08:20:55 GMT` 通过响应部首发送给客户端，当再次发送请求时，客户端将服务器返回的修改时间放在请求头 `If-Modified-Since:Thu, 19 Feb 2019 08:20:55 GMT` 发送给服务器，服务器再跟服务器上的对应资源进行比对，如果服务器的资源更新，那么返回最新的资源，此时状态码200，当服务器资源跟客户端的请求的报首时间一致，证明客户端的资源是最新的，返回304状态码，表示客户端直接用缓存即可。

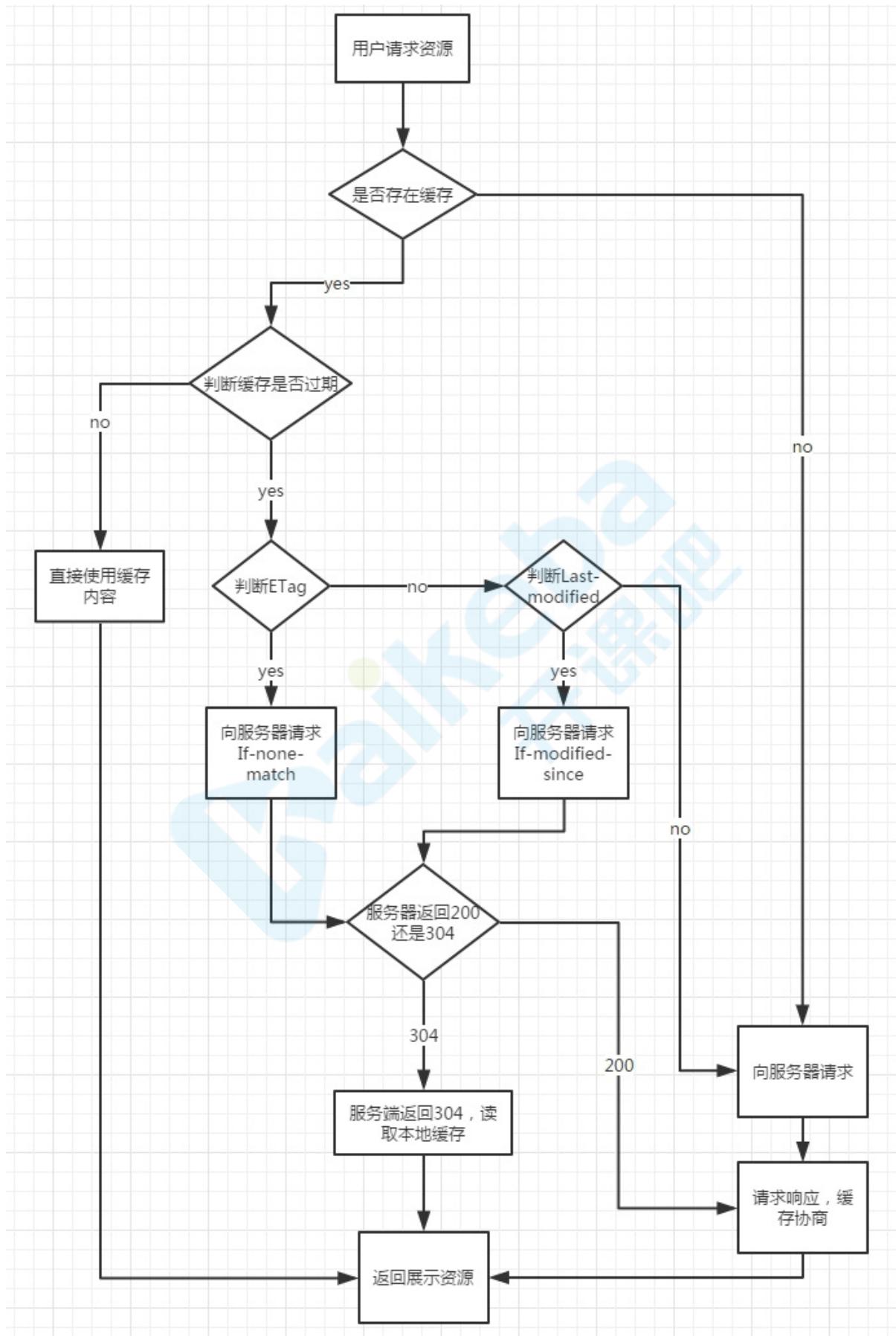
ETag/If-None-Match

ETag的流程跟Last-Modified是类似的，区别就在于ETag是根据资源内容进行hash，生成一个信息摘要，只要资源内容有变化，这个摘要就会发生巨变，通过这个摘要信息比对，即可确定客户端的缓存资源是否为最新，这比Last-Modified的精确度要更高。

响应头

```
cache-control: public, max-age=31536000
content-disposition: inline; filename="0.251196205261781ad15
9.js"; filename*=utf-8' '0.251196205261781ad159.js
content-encoding: gzip
content-length: 191469
content-transfer-encoding: binary
content-type: application/javascript
date: Thu, 13 Jun 2019 04:23:33 GMT
eagleid: 2aec23a515605289782858519e
etag: "FtINJcR0KB0bgECZ9VoQAn3tz2t6.gz"
last-modified: Thu, 13 Jun 2019 04:22:37 GMT
```

因此整体的缓存流程图如下：



图片来源于博客

TODO:

http的整个流程，涉及tcp/ip协议

参考：

- 图解HTTP
- HTTP权威指南
- [HTTP缓存策略](#)

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



TCP面试题

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

TCP的面试题通常情况下前端不会涉及太多，此章主要面对node.js工程师。

TCP 的特性

- TCP 提供一种面向连接的、可靠的字节流服务
- 在一个 TCP 连接中，仅有两方进行彼此通信。广播和多播不能用于 TCP
- TCP 使用校验和，确认和重传机制来保证可靠传输
- TCP 给数据分节进行排序，并使用累积确认保证数据的顺序不变和非重复
- TCP 使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行拥塞控制

请简述TCP\UDP的区别

| 协议 | 连接性 | 双工性 | 可靠性 | 有序性 | 有界性 | 拥塞控制 | 传输速度 | 量级 | 头部大小 |
|-----|-------------------------------|--------------|----------------------|-------------------------|----------------|------|------|----|-------------|
| TCP | 面向连接 (Connection oriented) | 全双工 (1:1) | 可靠 (重传机制) | 有序 (通过 SYN排 序) | 无, 有粘包 情况 | 有 | 慢 | 低 | 20~60 字节 |
| UDP | 无连接 (Connection less) | n:m | 不可靠 (丢包后 数据丢失) | 无序 | 有消息边 界, 无粘包 | 无 | 快 | 高 | 8字节 |

UDP socket 支持 n 对 m 的连接状态, 在[官方文档](#)中有写到在 `dgram.createSocket(options[, callback])` 中的 option 可以指定 `reuseAddr` 即 `SO_REUSEADDR` 标志. 通过 `SO_REUSEADDR` 可以简单的实现 n 对 m 的多播特性(不过仅在支持多播的系统上才有).

TCP粘包是怎么回事，如何处理？

默认情况下, TCP 连接会启用延迟传送算法 (Nagle 算法), 在数据发送之前缓存他们. 如果短时间有多个数据发送, 会缓冲到一起作一次发送 (缓冲大小见 `socket.bufferSize`), 这样可以减少 IO 消耗提高性能.

如果是传输文件的话, 那么根本不用处理粘包的问题, 来一个包拼一个包就好了. 但是如果是多条消息, 或者是别的用途的数据那么就需要处理粘包.

可以参见网上流传比较广的一个例子, 连续调用两次 `send` 分别发送两段数据 `data1` 和 `data2`, 在接收端有以下几种常见的:

- A. 先接收到 `data1`, 然后接收到 `data2`.
- B. 先接收到 `data1` 的部分数据, 然后接收到 `data1` 余下的部分以及 `data2` 的全部.
- C. 先接收到了 `data1` 的全部数据和 `data2` 的部分数据, 然后接收到了 `data2` 的余下的数据.
- D. 一次性接收到了 `data1` 和 `data2` 的全部数据.

其中的 BCD 就是我们常见的粘包的情况. 而对于处理粘包的问题, 常见的解决方案有:

1. 多次发送之前间隔一个等待时间
2. 关闭 Nagle 算法
3. 进行封包/拆包

方案1

只需要等上一段时间再进行下一次 send 就好, 适用于交互频率特别低的场景. 缺点也很明显, 对于比较频繁的场景而言传输效率实在太低. 不过几乎不用做什么处理.

方案2

关闭 Nagle 算法, 在 Node.js 中你可以通过 `socket.setNoDelay()` 方法来关闭 Nagle 算法, 让每一次 send 都不缓冲直接发送.

该方法比较适用于每次发送的数据都比较大 (但不是文件那么大), 并且频率不是特别高的场景. 如果是每次发送的数据量比较小, 并且频率特别高的, 关闭 Nagle 纯属自废武功.

另外, 该方法不适用于网络较差的情况, 因为 Nagle 算法是在服务端进行的包合并情况, 但是如果短时间内客户端的网络情况不好, 或者应用层由于某些原因不能及时将 TCP 的数据 recv, 就会造成多个包在客户端缓冲从而粘包的情况. (如果是在稳定的机房内部通信那么这个概率是比较小可以选择忽略的)

方案3

封包/拆包是目前业内常见的解决方案了. 即给每个数据包在发送之前, 于其前/后放一些有特征的数据, 然后收到数据的时候根据特征数据分割出来各个数据包.

为什么udp不会粘包?

1.TCP协议是面向流的协议， UDP是面向消息的协议

UDP段都是一条消息， 应用程序必须以消息为单位提取数据， 不能一次提取任意字节的数据

2.UDP具有保护消息边界，在每个UDP包中就有了消息头（消息来源地址，端口等信息），这样对于接收端来说就容易进行区分处理了。传输协议把数据当作一条独立的消息在网上传输，接收端只能接收独立的消息。接收端一次只能接收发送端发出的一个数据包,如果一次接受数据的大小小于发送端一次发送的数据大小，就会丢失一部分数据，即使丢失，接受端也不会分两次去接收

如何理解 TCP backlog?

本文来自[How TCP backlog works in Linux](#)

当应用程序调用 `listen` 系统调用让一个 `socket` 进入 `LISTEN` 状态时, 需要指定一个参数: `backlog`。这个参数经常被描述为, 新连接队列的长度限制。



`tcp-state-diagram.png`

由于 TCP 建立连接需要进行3次握手, 一个新连接在到达 `ESTABLISHED` 状态可以被 `accept` 系统调用返回给应用程序前, 必须经过一个中间状态 `SYN RECEIVED` (见上图)。这意味着, TCP/IP 协议栈在实现 `backlog` 队列时, 有两种不同的选择:

1. 仅使用一个队列, 队列规模由 `listen` 系统调用 `backlog` 参数指定。当协议栈收到一个 `SYN` 包时, 响应 `SYN/ACK` 包并且将连接加进该队列。当相应的 `ACK` 响应包收到后, 连接变为 `ESTABLISHED` 状态, 可以向应用程序返回。这意味着队列里的连接可以有两种不同的状态: `SEND RECEIVED` 和 `ESTABLISHED`。只有后一种连接才能被 `accept` 系统调用返回给应用程序。
2. 使用两个队列——`SYN` 队列(待完成连接队列)和 `accept` 队列(已完成连接队列)。状态为 `SYN RECEIVED` 的连接进入 `SYN` 队列, 后续当状态变更为 `ESTABLISHED` 时移到 `accept` 队列(即收到3次握手中最后一个 `ACK` 包)。顾名思义, `accept` 系统调用就只是简单地从 `accept` 队列消费新连接。在这种情况下, `listen` 系统调用 `backlog` 参数决定 `accept` 队列的最大规模。

历史上，起源于 BSD 的 TCP 实现使用第一种方法。这个方案意味着，但 backlog 限制达到，系统将停止对 SYN 包响应 SYN/ACK 包。通常，协议栈只是丢弃 SYN 包(而不是回一个 RST 包)以便客户端可以重试(而不是异常退出)。

TCP/IP详解 卷3 第 14.5 节中有提到这一点。书中作者提到，BSD 实现虽然使用了两个独立的队列，但是行为跟使用一个队列并没什么区别。

在 Linux 上，情况有所不同，情况 listen 系统调用 man 文档页：

The behavior of the backlog argument on TCP sockets changed with Linux 2.2. Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using /proc/sys/net/ipv4/tcp_max_syn_backlog. When syncookies are enabled there is no logical maximum length and this setting is ignored. 意思是，backlog 参数的行为在 Linux 2.2 之后有所改变。现在，它指定了等待 accept 系统调用的已建立连接队列的长度，而不是待完成连接请求数。待完成连接队列长度由 /proc/sys/net/ipv4/tcp_max_syn_backlog 指定；在 syncookies 启用的情况下，逻辑上没有最大值限制，这个设置便被忽略。

也就是说，当前版本的 Linux 实现了第二种方案，使用两个队列——一个 SYN 队列，长度系统级别可设置以及一个 accept 队列长度由应用程序指定。

现在，一个需要考虑的问题是在 accept 队列已满而一个已完成新连接需要用 SYN 队列移动到 accept 队列(收到3次握手最后一个 ACK 包)，这个实现方案是什么行为。这种情况下，由 net/ipv4/tcp_minisocks.c 中 tcp_check_req 函数处理：

```
child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb, req, NULL);
if (child == NULL)
    goto listen_overflow;
```

对于 IPv4，第一行代码实际上调用的是 net/ipv4/tcp_ipv4.c 中的 tcp_v4_syn_recv_sock 函数，代码如下：

```
if (sk_acceptq_is_full(sk))
    goto exit_overflow;
```

可以看到，这里会检查 accept 队列的长度。如果队列已满，跳到 exit_overflow 标签执行一些清理工作、更新 /proc/net/netstat 中的统计项 ListenOverflows 和 ListenDrops，最后返回 NULL。这会触发 tcp_check_req 函数跳到 listen_overflow 标签执行代码。

```
listen_overflow:
if (!sysctl_tcp_abort_on_overflow) {
    inet_rsk(req)->acked = 1;
    return NULL;
}
```

很显然，除非 /proc/sys/net/ipv4/tcp_abort_on_overflow 被设置为 1 (这种情况下发送一个 RST 包)，实现什么都没做。总结一下：Linux 内核协议栈在收到3次握手最后一个 ACK 包，确认一个新连接已完成，而 accept 队列已满的情况下，会忽略这个包。一开始您可能会对此感到奇怪——别忘了 SYN RECEIVED 状态下有一个计时器实现：如果 ACK 包没有收到(或者是我们讨论的忽略)，协议栈会重发 SYN/ACK 包(重试次数由 /proc/sys/net/ipv4/tcp_synack_retries 决定)。看以下抓包结果就非常明显——一个客户正尝试连接一个已经达到其最大 backlog 的 socket：

```
0.000 127.0.0.1 -> 127.0.0.1 TCP 74 53302 > 9999 [SYN] Seq=0 Len=0
0.000 127.0.0.1 -> 127.0.0.1 TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 Len=0
0.000 127.0.0.1 -> 127.0.0.1 TCP 66 53302 > 9999 [ACK] Seq=1 Ack=1 Len=0
0.000 127.0.0.1 -> 127.0.0.1 TCP 71 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
0.207 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
0.623 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
1.199 127.0.0.1 -> 127.0.0.1 TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 Len=0
1.199 127.0.0.1 -> 127.0.0.1 TCP 66 [TCP Dup ACK 6#1] 53302 > 9999 [ACK] Seq=6 Ack=1 Len=0
1.455 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
3.123 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
```

```

3.399 127.0.0.1 -> 127.0.0.1 TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 Len=0
3.399 127.0.0.1 -> 127.0.0.1 TCP 66 [TCP Dup ACK 10#1] 53302 > 9999 [ACK] Seq=6 Ack=1 Len=0
6.459 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
7.599 127.0.0.1 -> 127.0.0.1 TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 Len=0
7.599 127.0.0.1 -> 127.0.0.1 TCP 66 [TCP Dup ACK 13#1] 53302 > 9999 [ACK] Seq=6 Ack=1 Len=0
13.131 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
15.599 127.0.0.1 -> 127.0.0.1 TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 Len=0
15.599 127.0.0.1 -> 127.0.0.1 TCP 66 [TCP Dup ACK 16#1] 53302 > 9999 [ACK] Seq=6 Ack=1 Len=0
26.491 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
31.599 127.0.0.1 -> 127.0.0.1 TCP 74 9999 > 53302 [SYN, ACK] Seq=0 Ack=1 Len=0
31.599 127.0.0.1 -> 127.0.0.1 TCP 66 [TCP Dup ACK 19#1] 53302 > 9999 [ACK] Seq=6 Ack=1 Len=0
53.179 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
106.491 127.0.0.1 -> 127.0.0.1 TCP 71 [TCP Retransmission] 53302 > 9999 [PSH, ACK] Seq=1 Ack=1 Len=5
106.491 127.0.0.1 -> 127.0.0.1 TCP 54 9999 > 53302 [RST] Seq=1 Len=0

```

由于客户端的 TCP 实现在收到多个 SYN/ACK 包时，认为 ACK 包已经丢失了并且重传它。如果在 SYN/ACK 重试次数达到限制前，服务端应用从 accept 队列接收连接，使得 backlog 减少，那么协议栈会处理这些重传的 ACK 包，将连接状态从 SYN RECEIVED 变更到 ESTABLISHED 并且将其加入 accept 队列。否则，正如以上包跟踪所示，客户读会收到一个 RST 包宣告连接失败。

在客户端看来，第一次收到 SYN/ACK 包之后，连接就会进入 ESTABLISHED 状态。如果这时客户端首先开始发送数据，那么数据也会被重传。好在 TCP 有慢启动机制，在服务端还没进入 ESTABLISHED 之前，客户端能发送的数据非常有限。

相反，如果客户端一开始就在等待服务端，而服务端 backlog 没能减少，那么最后的结果是连接在客户端看来是 ESTABLISHED 状态，但在服务端看来是 CLOSED 状态。这也就是所谓的半开连接。

有一点还没讨论的是：man listen 中提到每次收到新 SYN 包，内核往 SYN 队列追加一个新连接(除非该队列已满)。事实并非如此，net/ipv4/tcp_ipv4.c 中 tcp_v4_conn_request 函数负责处理 SYN 包，请看以下代码：

```

if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1) {
    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERRFLOWS);
    goto drop;
}

```

可以看到，在 accept 队列已满的情况下，内核会强制限制 SYN 包的接收速率。如果有大量 SYN 包待处理，它们其中的一些会被丢弃。这样看来，就完全依靠客户端重传 SYN 包了，这种行为跟 BSD 实现一样。

下结论前，需要再研究以下 Linux 这种实现方式跟 BSD 相比有什么优势。Stevens 是这样说的：

在 accept 队列已满或者 SYN 队列已满的情况下，backlog 会达到限制。第一种情况经常发生在服务器或者服务器进程非常繁忙的情况下，进程没法足够快地调用 accept 系统调用从中取出已完成连接。后者是 HTTP 服务器经常面临的问题，在服务端客户端往返时间非常长的时候(相对于连接到达速率)，因为新 SYN 包在往返时间内都会占据一个连接对象。大多数情况下 accept 队列都是空的，因为一旦有一个新连接进入队列，阻塞等待的 accept 系统调用将返回，然后连接从队列中取出。

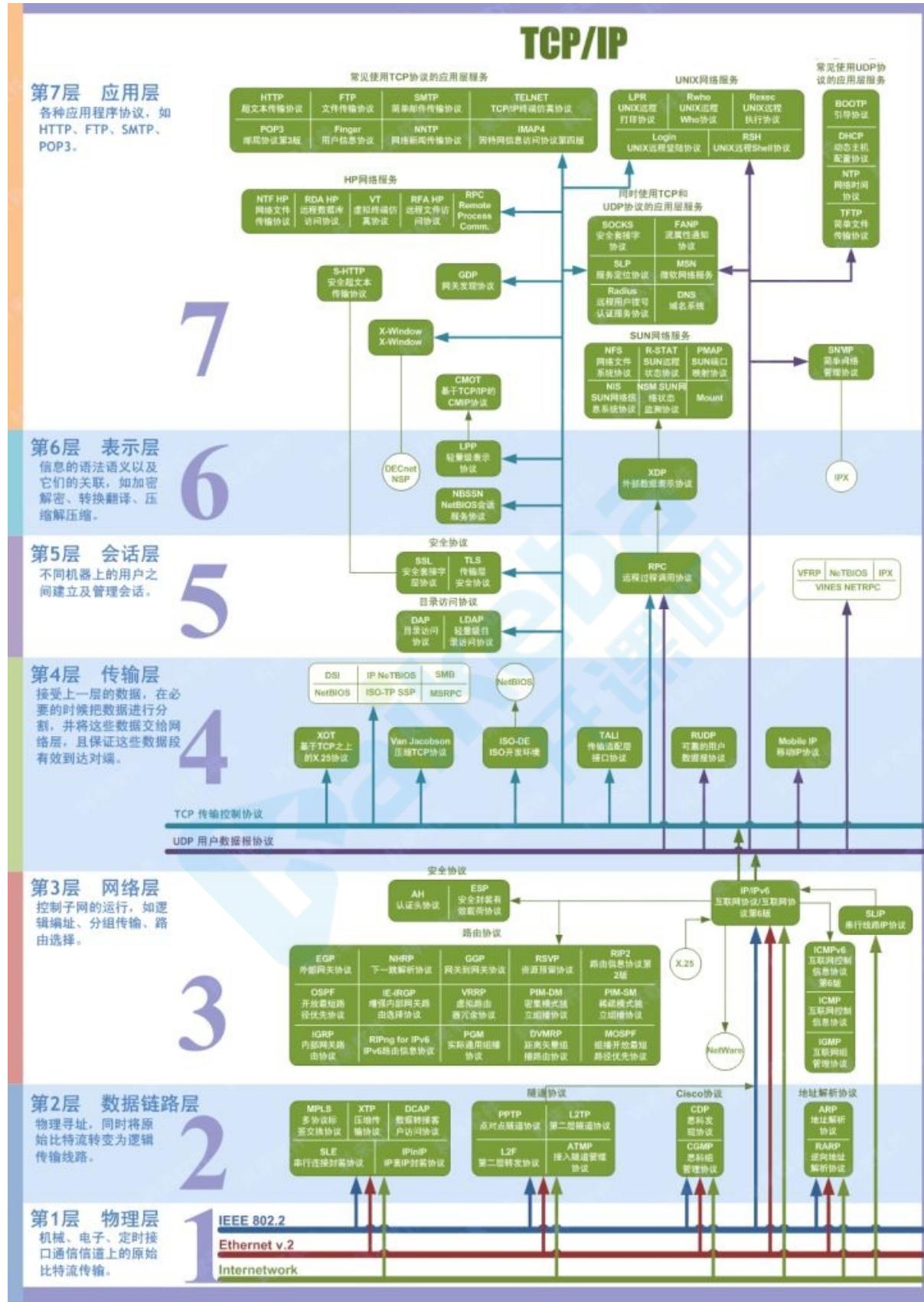
Stevens 建议的解决方案是简单地调大 backlog。但有个问题是，应用程序在调优 backlog 参数时，不仅需要考虑自身对新连接的处理逻辑，还需要考虑网络状况，包括往返时间等。Linux 实现实际上分成两部分：应用程序只负责调解 backlog 参数，确保 accept 调用足够快以免 accept 队列被塞满；系统管理员则根据网络状况调节 /proc/sys/net/ipv4/tcp_max_syn_backlog，各司其职。

常用端口号与对应的服务

| 端口 | 作用说明 |
|----|--|
| 21 | 21端口主要用于FTP (File Transfer Protocol, 文件传输协议) 服务。 |
| 23 | 23端口主要用于Telnet (远程登录) 服务，是Internet上普遍采用的登录和仿真程序。 |
| 25 | 25端口为SMTP (Simple Mail Transfer Protocol, 简单邮件传输协议) 服务器所开放，主要用于发送邮件，如今绝大多数邮件服务器都使用该协议。 |

| | |
|---------|--|
| 53 | 53端口为DNS（Domain Name Server, 域名服务器）服务器所开放，主要用于域名解析，DNS服务在NT系统中使用的最为广泛。 |
| 67、68 | 67、68端口分别是为Bootp服务的Bootstrap Protocol Server（引导程序协议服务端）和Bootstrap Protocol Client（引导程序协议客户端）开放的端口。 |
| 69 | TFTP是Cisco公司开发的一个简单文件传输协议，类似于FTP。 |
| 79 | 79端口是为Finger服务开放的，主要用于查询远程主机在线用户、操作系统类型以及是否缓冲区溢出等用户的详细信息。 |
| 80 | 80端口是为HTTP（HyperText Transport Protocol，超文本传输协议）开放的，这是上网冲浪使用最多的协议，主要用于在WWW（World WideWeb，万维网）服务上传输信息的协议。 |
| 99 | 99端口是用于一个名为“Metagram Relay”（亚对策延时）的服务，该服务比较少见，一般是用不到的。 |
| 109、110 | 109端口是为POP2（Post Office Protocol Version 2，邮局协议2）服务开放的，110端口是为POP3（邮件协议3）服务开放的，POP2、POP3都是主要用于接收邮件的。 |
| 111 | 111端口是SUN公司的RPC（Remote ProcedureCall，远程过程调用）服务所开放的端口，主要用于分布式系统中不同计算机的内部进程通信，RPC在多种网络服务中都是很重要的组件。 |
| 113 | 113端口主要用于Windows的“Authentication Service”（验证服务）。119端口：119端口是为“Network News TransferProtocol”（网络新闻组传输协议，简称NNTP）开放的。 |
| 135 | 135端口主要用于使用RPC（Remote Procedure Call，远程过程调用）协议并提供DCOM（分布式组件对象模型）服务。 |
| 137 | 137端口主要用于“NetBIOS Name Service”（NetBIOS名称服务）。 |
| 139 | 139端口是为“NetBIOS Session Service”提供的，主要用于提供Windows文件和打印机共享以及Unix中的Samba服务。 |
| 143 | 143端口主要是用于“Internet Message Access Protocol”v2（Internet消息访问协议，简称IMAP）。 |
| 161 | 161端口是用于“Simple Network Management Protocol”（简单网络管理协议，简称SNMP）。 |
| 443 | 443端口即网页浏览端口，主要是用于HTTPS服务，是提供加密和通过安全端口传输的另一种HTTP。 |
| 554 | 554端口默认情况下用于“Real Time Streaming Protocol”（实时流协议，简称RTSP）。 |
| 1024 | 1024端口一般不固定分配给某个服务，在英文中的解释是“Reserved”（保留）。 |
| 1080 | 1080端口是Socks代理服务使用的端口，大家平时上网使用的WWW服务使用的是HTTP协议的代理服务。 |
| 1755 | 1755端口默认情况下用于“Microsoft Media Server”（微软媒体服务器，简称MMS）。 |
| 4000 | 4000端口是用于大家经常使用的QQ聊天工具的，再细说就是为QQ客户端开放的端口，QQ服务端使用的端口是8000。 |
| 5554 | 在今年4月30日就报道出现了一种针对微软lsass服务的新蠕虫病毒——震荡波（Worm.Sasser），该病毒可以利用TCP 5554端口开启一个FTP服务，主要被用于病毒的传播。 |
| 5632 | 5632端口是被大家所熟悉的远程控制软件pcAnywhere所开启的端口。 |
| 8080 | 8080端口同80端口，是被用于WWW代理服务的，可以实现网页。 |

说一说OSI七层模型

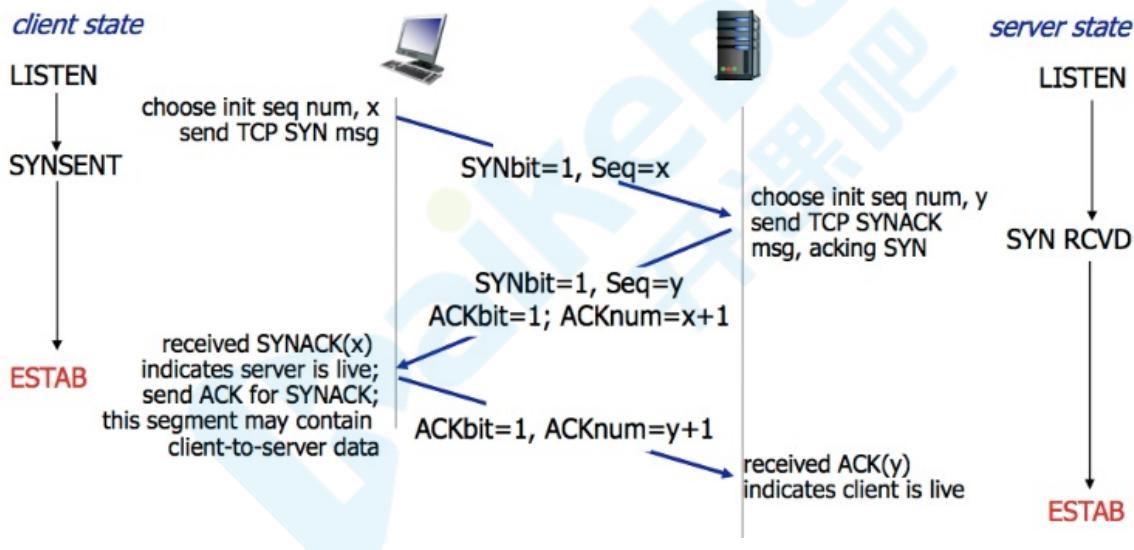


讲一下三次握手？

所谓三次握手(Three-way Handshake), 是指建立一个 TCP 连接时, 需要客户端和服务器总共发送3个包。三次握手的目的是连接服务器指定端口, 建立 TCP 连接, 并同步连接双方的序列号和确认号, 交换 TCP 窗口大小信息。在 socket 编程中, 客户端执行 `connect()` 时。将触发三次握手。

- 第一次握手(SYN=1, seq=x):
客户端发送一个 TCP 的 SYN 标志位置1的包, 指明客户端打算连接的服务器的端口, 以及初始序号 X, 保存在包头的序列号(Sequence Number)字段里。
发送完毕后, 客户端进入 `SYN_SEND` 状态。
- 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1):
服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为1。服务器端选择自己 ISN 序列号, 放到 Seq 域里, 同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加1, 即X+1。发送完毕后, 服务器端进入 `SYN_RECV` 状态。
- 第三次握手(ACK=1, ACKnum=y+1)
客户端再次发送确认包(ACK), SYN 标志位为0, ACK 标志位为1, 并且把服务器发来 ACK 的序号字段+1, 放在确定字段中发送给对方, 并且在数据段放写ISN的+1
发送完毕后, 客户端进入 `ESTABLISHED` 状态, 当服务器端接收到这个包时, 也进入 `ESTABLISHED` 状态, TCP 握手结束。

三次握手的过程的示意图如下:



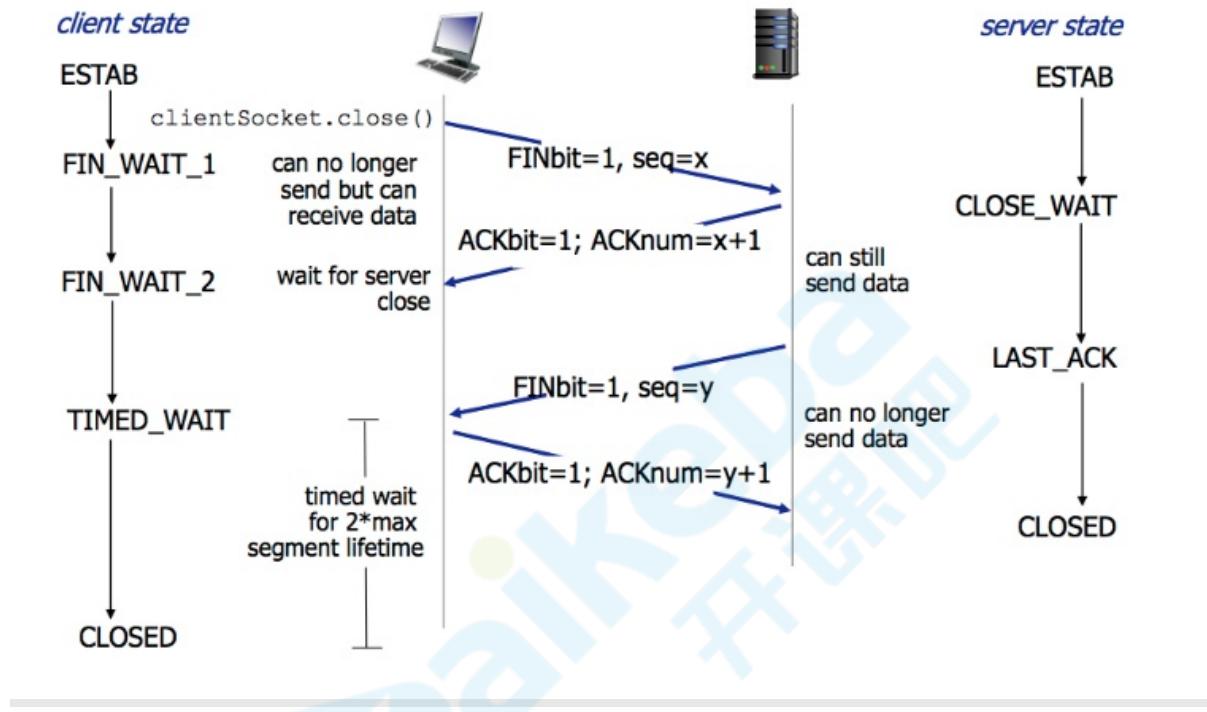
讲一下四次握手?

TCP 的连接的拆除需要发送四个包, 因此称为四次挥手(Four-way handshake), 也叫做改进的三次握手。客户端或服务器均可主动发起挥手动作, 在 socket 编程中, 任何一方执行 `close()` 操作即可产生挥手操作。

- 第一次挥手(FIN=1, seq=x)
假设客户端想要关闭连接, 客户端发送一个 FIN 标志位置为1的包, 表示自己已经没有数据可以发送了, 但是仍然可以接受数据。
发送完毕后, 客户端进入 `FIN_WAIT_1` 状态。
- 第二次挥手(ACK=1, ACKnum=x+1)
服务器端确认客户端的 FIN 包, 发送一个确认包, 表明自己接受到了客户端关闭连接的请求, 但还没有准备好关闭连接。
发送完毕后, 服务器端进入 `CLOSE_WAIT` 状态, 客户端接收到这个确认包之后, 进入 `FIN_WAIT_2` 状态, 等待服务器端关闭连接。
- 第三次挥手(FIN=1, seq=y)
服务器端准备好关闭连接时, 向客户端发送结束连接请求, FIN 置为1。

- 发送完毕后，服务器端进入 `LAST_ACK` 状态，等待来自客户端的最后一个ACK。
- 第四次挥手($ACK=1$, $ACKnum=y+1$)
 - 客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 `TIME_WAIT` 状态，等待可能出现的要求重传的ACK包。
 - 服务器端接收到这个确认包之后，关闭连接，进入 `CLOSED` 状态。
 - 客户端等待了某个固定时间（两个最大段生命周期， $2MSL$, 2 Maximum Segment Lifetime）之后，没有收到服务器端的ACK，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 `CLOSED` 状态。

四次挥手的示意图如下：



参考：

1. 饿了么面试
2. TCP

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



DOM

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

- DOM的事件模型是什么?
- DOM的事件流是什么?
- 什么是事件委托?

前端框架大行其道的今天，我们直接操作DOM的时候变得更少了，因此不妨复习一下DOM的基本知识

DOM的事件模型是什么？

DOM之事件模型分脚本模型、内联模型(同类一个，后者覆盖)、动态绑定(同类多个)

```

<body>
<!--行内绑定：脚本模型-->
<button onclick="javascrp:alert('Hello')">Hello1</button>
<!--内联模型-->
<button onclick="showHello()">Hello2</button>
<!--动态绑定-->
<button id="btn3">Hello3</button>
</body>
<script>
/*DOM0：同一个元素，同类事件只能添加一个，如果添加多个，
 * 后面添加的会覆盖之前添加的*/
function showHello() {
  alert("Hello");
}
var btn3 = document.getElementById("btn3");
btn3.onclick = function () {
  alert("Hello");
}
/*DOM2：可以给同一个元素添加多个同类事件*/
btn3.addEventListener("click",function () {
  alert("hello1");
});
btn3.addEventListener("click",function () {
  alert("hello2");
})
if (btn3.attachEvent){
  /*IE*/
  btn3.attachEvent("onclick",function () {
    alert("IE Hello1");
  })
} else {
  /*W3C*/
  btn3.addEventListener("click",function () {
    alert("W3C Hello");
  })
}
</script>

```

DOM的事件流是什么？

事件就是文档或浏览器窗口中发生的一些特定的交互瞬间，而事件流(又叫事件传播)描述的是从页面中接收事件的顺序。

事件冒泡

事件冒泡(event bubbling)，即事件开始时由最具体的元素(文档中嵌套层次最深的那个节点)接收，然后逐级向上传播到较为不具体的节点。

看如下例子：

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
<body>
<div></div>
</body>
</html>
```

如果单击了页面中的 `<div>` 元素，那么这个click事件沿DOM树向上传播，在每一级节点上都会发生，按照如下顺序传播：

1. div
2. body
3. html
4. document

事件捕获

事件捕获的思想是不太具体的节点应该更早接收到事件，而最具体的节点应该最后接收到事件。事件捕获的用意在于在事件到达预定目标之前就捕获它。

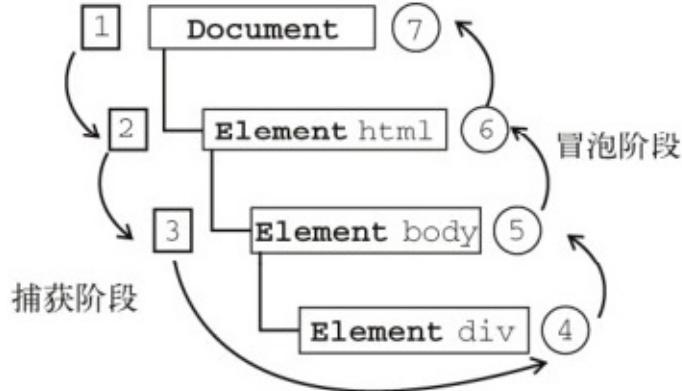
还是以上一节的html结构为例：

在事件捕获过程中，`document`对象首先接收到click事件，然后事件沿DOM树依次向下，一直传播到事件的实际目标，即 `<div>` 元素

1. document
2. html
3. body
4. div

事件流

事件流又称为事件传播，DOM2级事件规定的事件流包括三个阶段：事件捕获阶段(capture phase)、处于目标阶段(target phase)和事件冒泡阶段(bubbling phase)。



触发顺序通常为

1. 进行事件捕获，为截获事件提供了机会
2. 实际的目标接收到事件
3. 冒泡阶段，可以在这个阶段对事件做出响应

什么是事件委托

事件委托就是利用事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。

在绑定大量事件的时候往往选择事件委托。

```

<ul id="parent">
  <li class="child">one</li>
  <li class="child">two</li>
  <li class="child">three</li>
  ...
</ul>

<script type="text/javascript">
  //父元素
  var dom= document.getElementById('parent');

  //父元素绑定事件，代理子元素的点击事件
  dom.onclick= function(event) {
    var event= event || window.event;
    var curTarget= event.target || event.srcElement;

    if (curTarget.tagName.toLowerCase() == 'li') {
      //事件处理
    }
  }
</script>

```

优点：

- 节省内存占用，减少事件注册
- 新增子对象时无需再次对其绑定事件，适合动态添加元素

局限性：

- focus、blur 之类的事件本身没有事件冒泡机制，所以无法委托
- mousemove、mouseout 这样的事件，虽然有事件冒泡，但是只能不断通过位置去计算定位，对性能消耗高，不适合事件委托

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



浏览器与新技术

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

- 常见的浏览器内核有哪些?
- 浏览器的主要组成部分是什么?
- 浏览器是如何渲染UI的?
- 浏览器如何解析css选择器?
- DOM Tree是如何构建的?
- 浏览器重绘与重排的区别?
- 如何触发重排和重绘?
- 如何避免重绘或者重排?
- 前端如何实现即时通讯?
- 什么是浏览器同源策略?
- 如何实现跨域?

本章关于浏览器原理部分的内容主要来源于[浏览器工作原理](#), 这是一篇很长的文章, 可以算上一本小书了, 有精力的非常建议阅读。

常见的浏览器内核有哪些?

| 浏览器/RunTime | 内核 (渲染引擎) | JavaScript 引擎 |
|-------------|-----------------------------------|------------------------|
| Chrome | Blink (28~) Webkit (Chrome 27) | V8 |
| FireFox | Gecko | SpiderMonkey |
| Safari | Webkit | JavaScriptCore |
| Edge | EdgeHTML | Chakra(for JavaScript) |
| IE | Trident | Chakra(for JScript) |
| PhantomJS | Webkit | JavaScriptCore |
| Node.js | - | V8 |

浏览器的主要组成部分是什么?

1. **用户界面** - 包括地址栏、前进/后退按钮、书签菜单等。除了浏览器主窗口显示的您请求的页面外，其他显示的各个部分都属于用户界面。
2. **浏览器引擎** - 在用户界面和呈现引擎之间传送指令。
3. **呈现引擎** - 负责显示请求的内容。如果请求的内容是 HTML，它就负责解析 HTML 和 CSS 内容，并将解析后的内容显示在屏幕上。
4. **网络** - 用于网络调用，比如 HTTP 请求。其接口与平台无关，并为所有平台提供底层实现。
5. **用户界面后端** - 用于绘制基本的窗口小部件，比如组合框和窗口。其公开了与平台无关的通用接口，而在底层使用操作系统的用户界面方法。
6. **JavaScript 解释器**。用于解析和执行 JavaScript 代码。
7. **数据存储**。这是持久层。浏览器需要在硬盘上保存各种数据，例如 Cookie。新的 HTML 规范 (HTML5) 定义了“**网络数据库**”，这是一个完整（但是轻便）的浏览器内数据库。



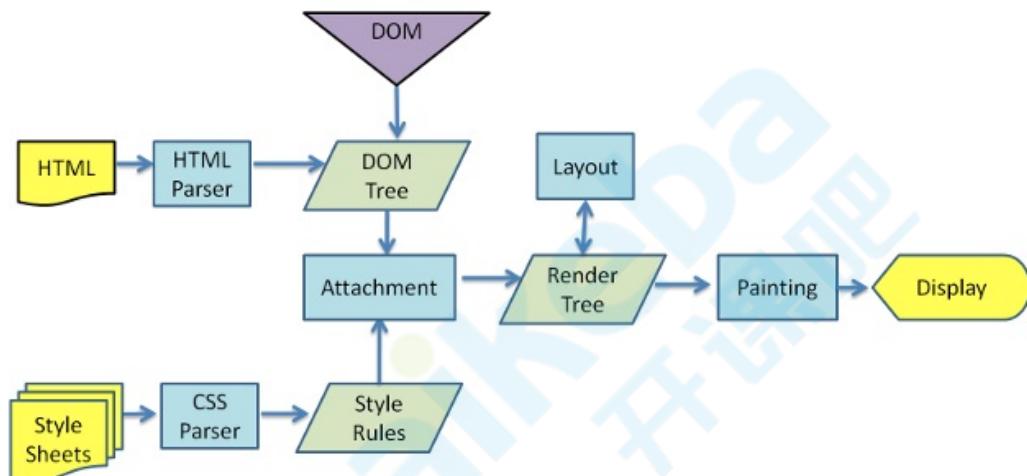
图：浏览器的主要组件。

值得注意的是，和大多数浏览器不同，Chrome 浏览器的每个标签页都分别对应一个呈现引擎实例。每个标签页都是一个独立的进程。

浏览器是如何渲染UI的？

1. 浏览器获取HTML文件，然后对文件进行解析，形成DOM Tree
2. 与此同时，进行CSS解析，生成Style Rules
3. 接着将DOM Tree与Style Rules合成为 Render Tree
4. 接着进入布局（Layout）阶段，也就是为每个节点分配一个应出现在屏幕上的确切坐标
5. 随后调用GPU进行绘制（Paint），遍历Render Tree的节点，并将元素呈现出来

主流程示例



图：WebKit 主流程

浏览器如何解析css选择器？

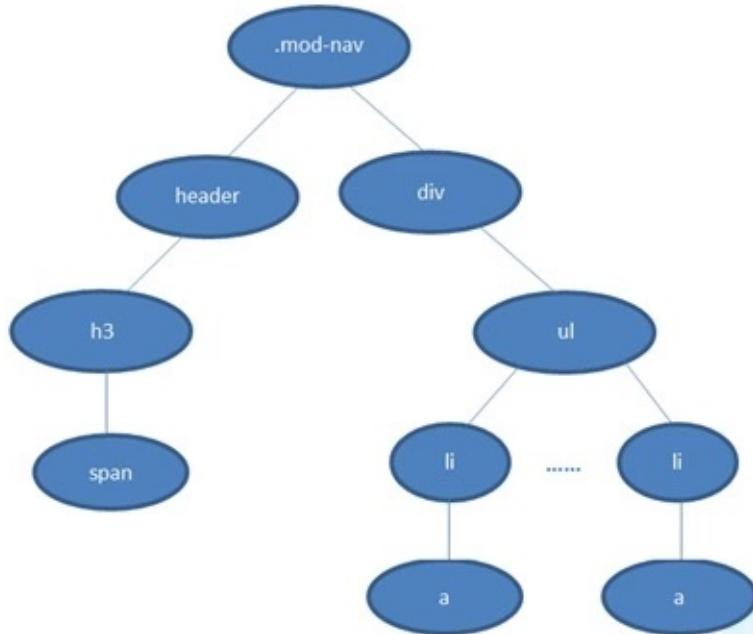
浏览器会『从右往左』解析CSS选择器。

我们知道DOM Tree与Style Rules合成为 Render Tree，实际上是需要将Style Rules附着到DOM Tree上，因此需要根据选择器提供的信息对DOM Tree进行遍历，才能将样式附着到对应的DOM元素上。

以下这段css为例

```
.mod-nav h3 span {font-size: 16px;}
```

我们对应的DOM Tree 如下



若从左向右的匹配，过程是：

1. 从 .mod-nav 开始，遍历子节点 header 和子节点 div
2. 然后各自向子节点遍历。在右侧 div 的分支中
3. 最后遍历到叶子节点 a，发现不符合规则，需要回溯到 ul 节点，再遍历下一个 li-a，一颗DOM树的节点动不动上千，这种效率很低。

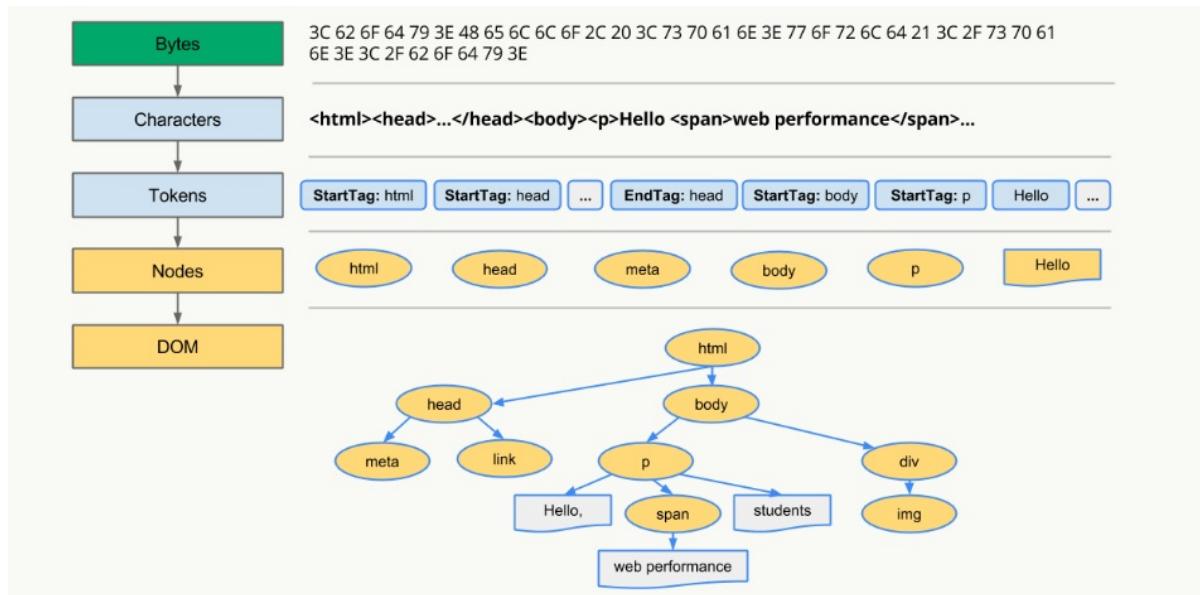
如果从右至左的匹配：

1. 先找到所有的最右节点 span，对于每一个 span，向上寻找节点 h3
2. 由 h3再向上寻找 class=mod-nav 的节点
3. 最后找到根元素 html 则结束这个分支的遍历。

后者匹配性能更好，是因为从右向左的匹配在第一步就筛选掉了大量的不符合条件的最右节点（叶子节点）；而从左向右的匹配规则的性能都浪费在了失败的查找上面。

DOM Tree是如何构建的？

1. 转码: 浏览器将接收到的二进制数据按照指定编码格式转化为HTML字符串
2. 生成Tokens: 之后开始parser，浏览器会将HTML字符串解析成Tokens
3. 构建Nodes: 对Node添加特定的属性，通过指针确定 Node 的父、子、兄弟关系和所属 treeScope
4. 生成DOM Tree: 通过node包含的指针确定的关系构建出DOM Tree



浏览器重绘与重排的区别?

- 重排: 部分渲染树 (或者整个渲染树) 需要重新分析并且节点尺寸需要重新计算, 表现为重新生成布局, 重新排列元素
- 重绘: 由于节点的几何属性发生改变或者由于样式发生改变, 例如改变元素背景色时, 屏幕上的部分内容需要更新, 表现为某些元素的外观被改变

单单改变元素的外观, 肯定不会引起网页重新生成布局, 但当浏览器完成重排之后, 将会重新绘制受到此次重排影响的部分

重排和重绘代价是高昂的, 它们会破坏用户体验, 并且让UI展示非常迟缓, 而相比之下重排的性能影响更大, 在两者无法避免的情况下, 一般我们宁可选择代价更小的重绘。

『重绘』不一定会出现『重排』, 『重排』必然会出现『重绘』。

如何触发重排和重绘?

任何改变用来构建渲染树的信息都会导致一次重排或重绘:

- 添加、删除、更新DOM节点
- 通过display: none隐藏一个DOM节点-触发重排和重绘
- 通过visibility: hidden隐藏一个DOM节点-只触发重绘, 因为没有几何变化
- 移动或者给页面中的DOM节点添加动画
- 添加一个样式表, 调整样式属性
- 用户行为, 例如调整窗口大小, 改变字号, 或者滚动。

如何避免重绘或者重排?

集中改变样式

我们往往通过改变class的方式来集中改变样式

```
// 判断是否是黑色系样式
const theme = isDark ? 'dark' : 'light'
```

```
// 根据判断来设置不同的class
ele.setAttribute('className', theme)
```

使用DocumentFragment

我们可以通过`createDocumentFragment`创建一个游离于DOM树之外的节点，然后在此节点上批量操作，最后插入DOM树中，因此只触发一次重排

```
var fragment = document.createDocumentFragment();

for (let i = 0; i < 10; i++) {
  let node = document.createElement("p");
  node.innerHTML = i;
  fragment.appendChild(node);
}

document.body.appendChild(fragment);
```

提升为合成层

将元素提升为合成层有以下优点：

- 合成层的位图，会交由 GPU 合成，比 CPU 处理要快
- 当需要 repaint 时，只需要 repaint 本身，不会影响到其他的层
- 对于 transform 和 opacity 效果，不会触发 layout 和 paint

提升合成层的最好方式是使用 CSS 的 will-change 属性：

```
#target {
  will-change: transform;
}
```

关于合成层的详解请移步[无线性能优化：Composite](#)

前端如何实现即时通讯？

短轮询

短轮询的原理很简单，每隔一段时间客户端就发出一个请求，去获取服务器最新的数据，一定程度上模拟实现了即时通讯。

- 优点：兼容性强，实现非常简单
- 缺点：延迟性高，非常消耗请求资源，影响性能

comet

comet有两种主要实现手段，一种是基于 AJAX 的长轮询（long-polling）方式，另一种是基于 Iframe 及 htmlfile 的流（streaming）方式，通常被叫做长连接。

具体两种手段的操作方法请移步[Comet技术详解：基于HTTP长连接的Web端实时通信技术](#)

长轮询优缺点：

- 优点：兼容性好，资源浪费较小
- 缺点：服务器hold连接会消耗资源，返回数据顺序无保证，难于管理维护

长连接优缺点：

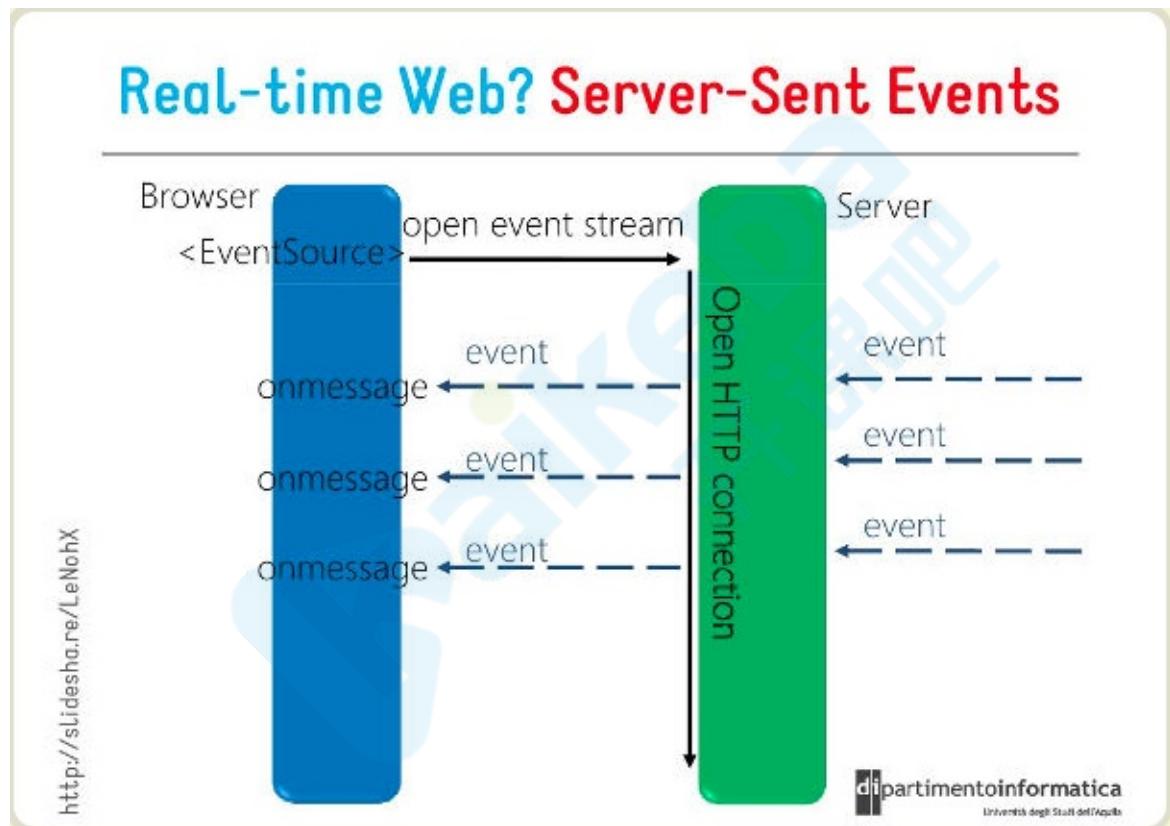
- 优点：兼容性好，消息即时到达，不发无用请求
- 缺点：服务器维护长连接消耗资源

SSE

使用指南请看[Server-Sent Events 教程](#)

SSE (Server-Sent Event, 服务端推送事件) 是一种允许服务端向客户端推送新数据的HTML5技术。

- 优点：基于HTTP而生，因此不需要太多改造就能使用，使用方便，而websocket非常复杂，必须借助成熟的库或框架
- 缺点：基于文本传输效率没有websocket高，不是严格的双向通信，客户端向服务端发送请求无法复用之前的连接，需要重新发出独立的请求

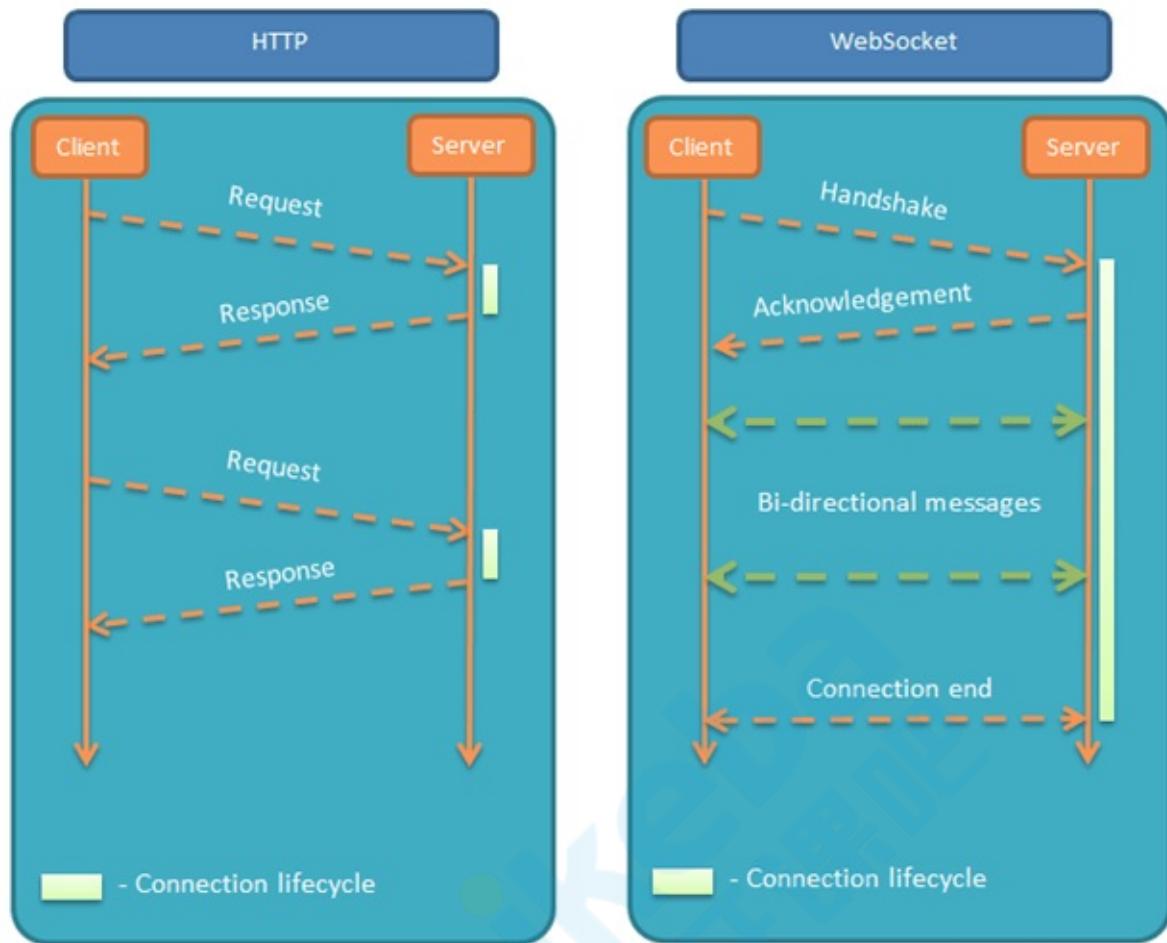


Websocket

使用指南请看[WebSocket 教程](#)

Websocket是一个全新的、独立的协议，基于TCP协议，与http协议兼容、却不会融入http协议，仅仅作为html5的一部分，其作用就是在服务器和客户端之间建立实时的双向通信。

- 优点：真正意义上的实时双向通信，性能好，低延迟
- 缺点：独立与http的协议，因此需要额外的项目改造，使用复杂度高，必须引入成熟的库，无法兼容低版本浏览器



Web Worker

后面性能优化部分会用到，先做了解

Web Worker 的作用，就是为 JavaScript 创建多线程环境，允许主线程创建 Worker 线程，将一些任务分配给后者运行

[Web Worker教程](#)

Service workers

后面性能优化部分会用到，先做了解

Service workers 本质上充当 Web 应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理，创建有效的离线体验。

[Service workers教程](#)

什么是浏览器同源策略？

同源策略限制了从同一个源加载的文档或脚本如何与来自另一个源的资源进行交互。这是一个用于隔离潜在恶意文件的重要安全机制。

同源是指“协议+域名+端口”三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

下表给出了相对 <http://store.company.com/dir/page.html> 同源检测的示例：

| URL | 结果 | 原因 |
|---|----|------------------------|
| http://store.company.com/dir2/other.html | 成功 | 只有路径不同 |
| http://store.company.com/dir/inner/another.html | 成功 | 只有路径不同 |
| https://store.company.com/secure.html | 失败 | 不同协议 (https和http) |
| http://store.company.com:81/dir/etc.html | 失败 | 不同端口 (http:// 80是默认的) |
| http://news.company.com/dir/other.html | 失败 | 不同域名 (news和store) |

浏览器中的大部分内容都是受同源策略限制的，但是以下三个标签可以不受限制：

-
- <link href=XXX>
- <script src=XXX>

如何实现跨域？

跨域是个比较古老的命题了，历史上跨域的实现手段有很多，我们现在主要介绍三种比较主流的跨域方案，其余的方案我们就不深入讨论了，因为使用场景很少，也没必要记这么多奇技淫巧。

最经典的跨域方案jsonp

jsonp本质上是一个Hack，它利用 `<script>` 标签不受同源策略限制的特性进行跨域操作。

jsonp优点：

- 实现简单
- 兼容性非常好

jsonp的缺点：

- 只支持get请求（因为 `<script>` 标签只能get）
- 有安全性问题，容易遭受xss攻击
- 需要服务端配合jsonp进行一定程度的改造

jsonp的实现：

```
function JSONP({
  url,
  params,
  callbackKey,
  callback
}) {
  // 在参数里制定 callback 的名字
  params = params || {}
  params[callbackKey] = 'jsonpCallback'
  // 预留 callback
  window.jsonpCallback = callback
  // 拼接参数字符串
  const paramKeys = Object.keys(params)
  const paramString = paramKeys
    .map(key => `${key}=${params[key]}`)
    .join('&')
  // 插入 DOM 元素
  const script = document.createElement('script')
  script.setAttribute('src', `${url}?${paramString}`)
  document.body.appendChild(script)
}

JSONP({
  url: 'http://s.weibo.com/ajax/jsonp/suggestion',
  params: {
```

```

        key: 'test',
    },
    callbackKey: '_cb',
    callback(result) {
        console.log(result.data)
    }
})

```

最流行的跨域方案cors

cors是目前主流的跨域解决方案，跨域资源共享(CORS) 是一种机制，它使用额外的 HTTP 头来告诉浏览器 让运行在一个 origin (domain) 上的Web应用被准许访问来自不同源服务器上的指定的资源。当一个资源从与该资源本身所在的服务器不同的域、协议或端口请求一个资源时，资源会发起一个跨域 HTTP 请求。

如果你用express，可以这样在后端设置

```

//CORS middleware
var allowCrossDomain = function(req, res, next) {
    res.header('Access-Control-Allow-Origin', 'http://example.com');
    res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE');
    res.header('Access-Control-Allow-Headers', 'Content-Type');

    next();
}

//...
app.configure(function() {
    app.use(express.bodyParser());
    app.use(express.cookieParser());
    app.use(express.session({ secret: 'cool beans' }));
    app.use(express.methodOverride());
    app.use(allowCrossDomain);
    app.use(app.router);
    app.use(express.static(__dirname + '/public'));
});

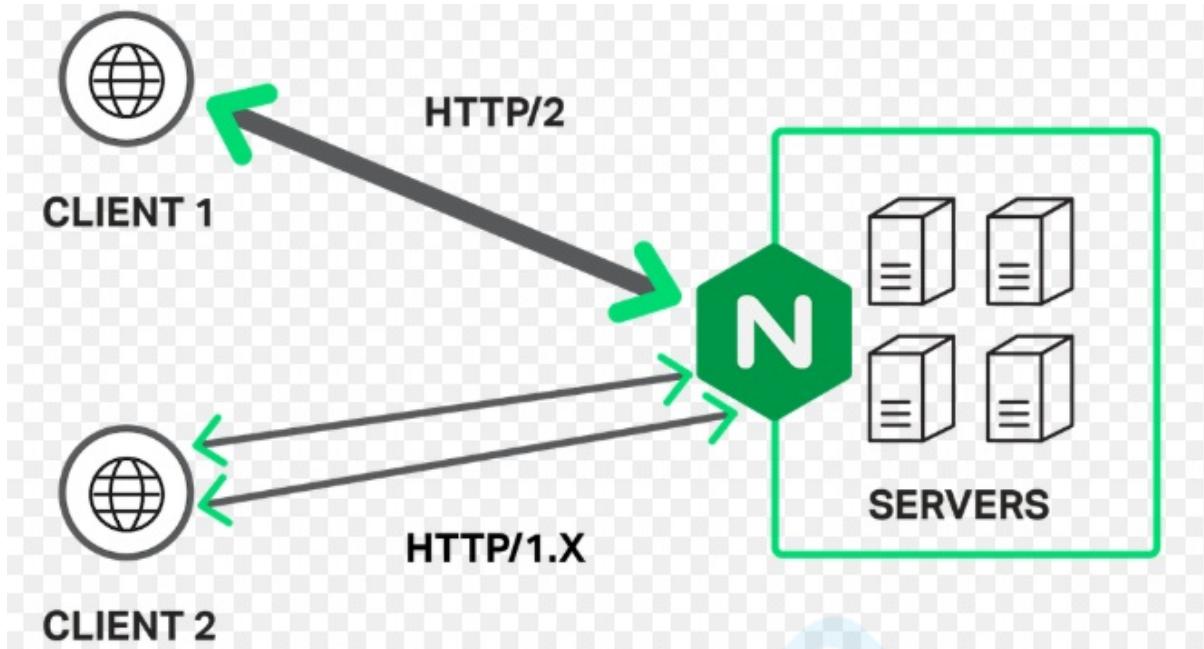
```

在生产环境中建议用成熟的开源中间件解决问题。

最方便的跨域方案Nginx

nginx是一款极其强大的web服务器，其优点就是轻量级、启动快、高并发。

现在的新项目中nginx几乎是首选，我们用node或者java开发的服务通常都需要经过nginx的反向代理。



反向代理的原理很简单，即所有客户端的请求都必须先经过nginx的处理，nginx作为代理服务器再将请求转发给node或者java服务，这样就规避了同源策略。

```
#进程，可更具cpu数量调整
worker_processes 1;

events {
    #连接数
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;

    sendfile on;

    #连接超时时间，服务器会在这个时间过后关闭连接。
    keepalive_timeout 10;

    # gzip压缩
    gzip on;

    # 直接请求nginx也是会报跨域错误的这里设置允许跨域
    # 如果代理地址已经允许跨域则不需要这些，否则报错(虽然这样nginx跨域就没意义了)
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Headers X-Requested-With;
    add_header Access-Control-Allow-Methods GET,POST,OPTIONS;

    # srever模块配置是http模块中的一个子模块，用来定义一个虚拟访问主机
    server {
        listen 80;
        server_name localhost;

        # 根路径指到index.html
        location / {
            root html;
            index index.html index.htm;
        }

        # localhost/api 的请求会被转发到192.168.0.103:8080
        location /api {
            rewrite ^/api/(.*)$ /$1 break; # 去除本地接口/api前缀，否则会出现404
        }
    }
}
```

```
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_pass http://192.168.0.103:8080; # 转发地址
}

# 重定向错误页面到/50x.html
error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root   html;
}

}

}
```

其它跨域方案

1. HTML5 XMLHttpRequest 有一个API，postMessage()方法允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本档、多窗口、跨域消息传递。
2. WebSocket 是一种双向通信协议，在建立连接之后，WebSocket 的 server 与 client 都能主动向对方发送或接收数据，连接建立好了之后 client 与 server 之间的双向通信就与 HTTP 无关了，因此可以跨域。
3. window.name + iframe：window.name属性值在不同的页面（甚至不同域名）加载后依旧存在，并且可以支持非常长的 name 值，我们可以利用这个特点进行跨域。
4. location.hash + iframe：a.html欲与c.html跨域相互通信，通过中间页b.html来实现。三个页面，不同域之间利用iframe的location.hash传值，相同域之间直接js访问来通信。
5. document.domain + iframe：该方式只能用于二级域名相同的情况下，比如 a.test.com 和 b.test.com 适用于该方式，我们只需要给页面添加 document.domain ='test.com' 表示二级域名都相同就可以实现跨域，两个页面都通过js强制设置document.domain为基础主域，就实现了同域。

其余方案来源于[九种跨域方式](#)

参考文章：

- [为什么 CSS 选择器解析的时候是从右往左？](#)

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



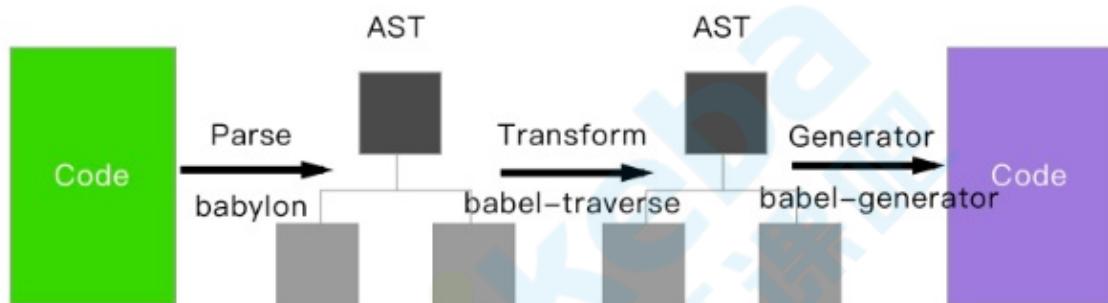
前端工程化

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

Babel的原理是什么?

babel 的转译过程也分为三个阶段, 这三步具体是:

- 解析 Parse: 将代码解析生成抽象语法树(即AST), 即词法分析与语法分析的过程
- 转换 Transform: 对于 AST 进行变换一系列的操作, babel 接受得到 AST 并通过 babel-traverse 对其进行遍历, 在此过程中进行添加、更新及移除等操作
- 生成 Generate: 将变换后的 AST 再转换为 JS 代码, 使用到的模块是 babel-generator



更具体的原理可以移步[如何写一个babel](#)

如何写一个babel插件?

Babel解析成AST，然后插件更改AST，最后由Babel输出代码

那么Babel的插件模块需要你暴露一个function, function内返回visitor

```

module.exports = function(babel){
  return {
    visitor: {
    }
  }
}
  
```

visitor是对各类型的AST节点做处理的地方，那么我们怎么知道Babel生成了的AST有哪些节点呢？

很简单，你可以把Babel转换的结果打印出来，或者这里有传送门: [AST explorer](#)

这里我们看到 `const result = 1 + 2` 中的 `1 + 1` 是一个 `BinaryExpression` 节点，那么在visitor中，我们就处理这个节点

```

var babel = require('babel-core');
var t = require('babel-types');
const visitor = {
  BinaryExpression(path) {
    const node = path.node;
  }
}
  
```

```

let result;
// 判断表达式两边，是否都是数字
if (t.isNumericLiteral(node.left) && t.isNumericLiteral(node.right)) {
    // 根据不同的操作符作运算
    switch (node.operator) {
        case "+":
            result = node.left.value + node.right.value;
            break;
        case "-":
            result = node.left.value - node.right.value;
            break;
        case "*":
            result = node.left.value * node.right.value;
            break;
        case "/":
            result = node.left.value / node.right.value;
            break;
        case "***":
            let i = node.right.value;
            while (--i) {
                result = result || node.left.value;
                result = result * node.left.value;
            }
            break;
        default:
    }
}
// 如果上面的运算有结果的话
if (result !== undefined) {
    // 把表达式节点替换成number字面量
    path.replaceWith(t.numericLiteral(result));
}
};

module.exports = function (babel) {
    return {
        visitor
    };
}

```

插件写好了，我们运行下插件试试

```

const babel = require("babel-core");
const result = babel.transform("const result = 1 + 2;", {
    plugins:[
        require("./index")
    ]
});
console.log(result.code); // const result = 3;

```

与预期一致，那么转换 `const result = 1 + 2 + 3 + 4 + 5;` 呢？

结果是: `const result = 3 + 3 + 4 + 5;`

这就奇怪了，为什么只计算了 `1 + 2` 之后，就没有继续往下运算了？

我们看一下这个表达式的AST树



你会发现Babel解析成表达式里面再嵌套表达式。

```
表达式( 表达式( 表达式( 表达式(1 + 2) + 3) + 4) + 5)
```

而我们的判断条件并不符合所有的，只符合 `1 + 2`

```
// 判断表达式两边，是否都是数字
```

```
if (t.isNumericLiteral(node.left) && t.isNumericLiteral(node.right)) {}
```

那么我们得改一改

第一次计算 $1 + 2$ 之后，我们会得到这样的表达式

```
表达式( 表达式( 表达式(3 + 3) + 4) + 5)
```

其中 $3 + 3$ 又符合了我们的条件， 我们通过向上递归的方式遍历父级节点

又转换成这样：

```
表达式( 表达式(6 + 4) + 5)
表达式(10 + 5)
15
```

```
// 如果上面的运算有结果的话
if (result !== undefined) {
    // 把表达式节点替换成number字面量
    path.replaceWith(t.numericLiteral(result));
    let parentPath = path.parentPath;
    // 向上遍历父级节点
    parentPath && visitor.BinaryExpression.call(this, parentPath);
}
```

到这里，我们就得出了结果 `const result = 15;`

那么其他运算呢：

```
const result = 100 + 10 - 50 >>> const result = 60;
const result = (100 / 2) + 50 >>> const result = 100;
const result = (((100 / 2) + 50 * 2) / 50) ** 2 >>> const result = 9;
```

项目地址

上述答案来源于cnode帖子

更详实的教程移步[官方的插件教程](#)

你的git工作流是怎样的？

GitFlow 是由 Vincent Driessen 提出的一个 git操作流程标准。包含如下几个关键分支：

| 名称 | 说明 |
|---------|---|
| master | 主分支 |
| develop | 主开发分支，包含确定即将发布的代码 |
| feature | 新功能分支，一般一个新功能对应一个分支，对于功能的拆分需要比较合理，以避免一些后面不必要的代码冲突 |
| release | 发布分支，发布时候用的分支，一般测试时候发现的 bug 在这个分支进行修复 |
| hotfix | hotfix 分支，紧急修 bug 的时候用 |

GitFlow 的优势有如下几点：

- 并行开发：GitFlow 可以很方便的实现并行开发：每个新功能都会建立一个新的 `feature` 分支，从而和已经完成的功能隔离开来，而且只有在新功能完成开发的情况下，其对应的 `feature` 分支才会合并到主开发分支上（也就

是我们经常说的 `develop` 分支）。另外，如果你正在开发某个功能，同时又有一个新的功能需要开发，你只需要提交当前 `feature` 的代码，然后创建另外一个 `feature` 分支并完成新功能开发。然后再切回之前的 `feature` 分支即可继续完成之前功能的开发。

- 协作开发：GitFlow 还支持多人协同开发，因为每个 `feature` 分支上改动的代码都只是为了让某个新的 `feature` 可以独立运行。同时我们也很容易知道每个人都在干嘛。
- 发布阶段：当一个新 `feature` 开发完成的时候，它会被合并到 `develop` 分支，这个分支主要用来暂时保存那些还没有发布的内容，所以如果需要再开发新的 `feature`，我们只需要从 `develop` 分支创建新分支，即可包含所有已经完成的 `feature`。
- 支持紧急修复：GitFlow 还包含了 `hotfix` 分支。这种类型的分支是从某个已经发布的 `tag` 上创建出来并做一个紧急的修复，而且这个紧急修复只影响这个已经发布的 `tag`，而不会影响到你正在开发的新 `feature`。

然后就是 GitFlow 最经典的几张流程图，一定要理解：



`feature` 分支都是从 `develop` 分支创建，完成后再合并到 `develop` 分支上，等待发布。



当需要发布时，我们从 `develop` 分支创建一个 `release` 分支



然后这个 `release` 分支会发布到测试环境进行测试，如果发现问题就在这个分支直接进行修复。在所有问题修复之前，我们会不停的重复发布->测试->修复->重新发布->重新测试这个流程。

发布结束后，这个 `release` 分支会合并到 `develop` 和 `master` 分支，从而保证不会有代码丢失。



`master` 分支只跟踪已经发布的代码，合并到 `master` 上的 commit 只能来自 `release` 分支和 `hotfix` 分支。

`hotfix` 分支的作用是紧急修复一些 Bug。

它们都是从 `master` 分支上的某个 `tag` 建立，修复结束后再合并到 `develop` 和 `master` 分支上。

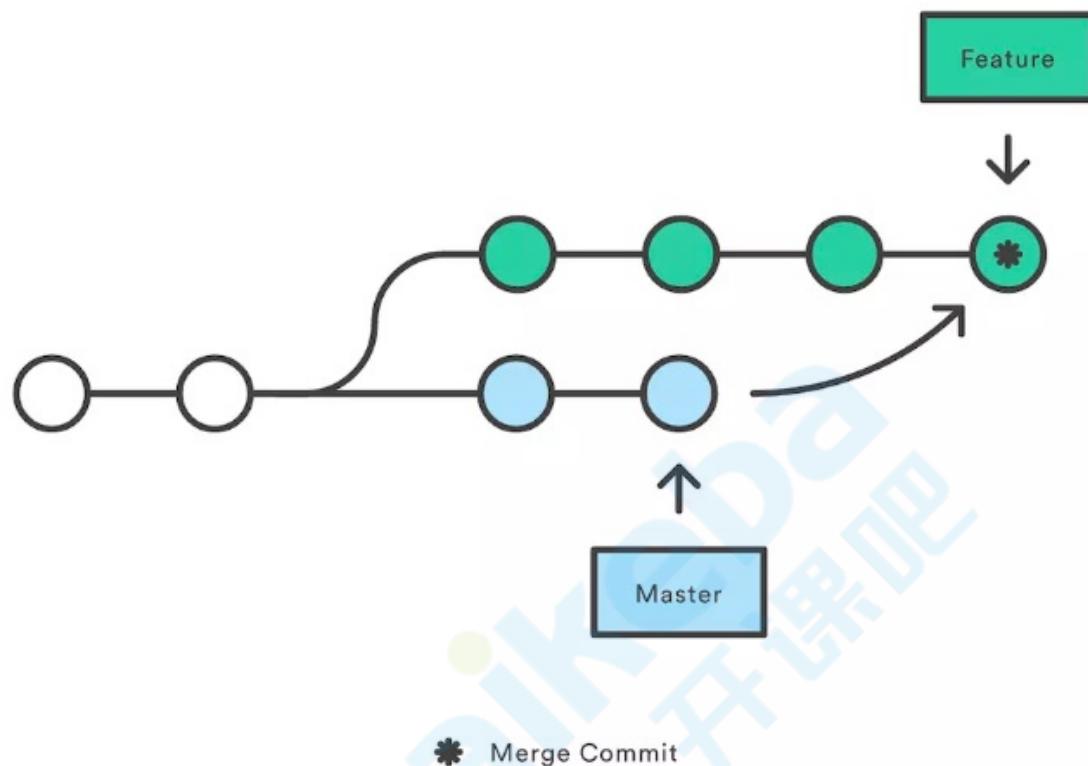
更多工作流可以参考阮老师的[Git 工作流程](#)

rebase 与 merge 的区别？

git rebase 和 git merge 一样都是用于从一个分支获取并且合并到当前分支。

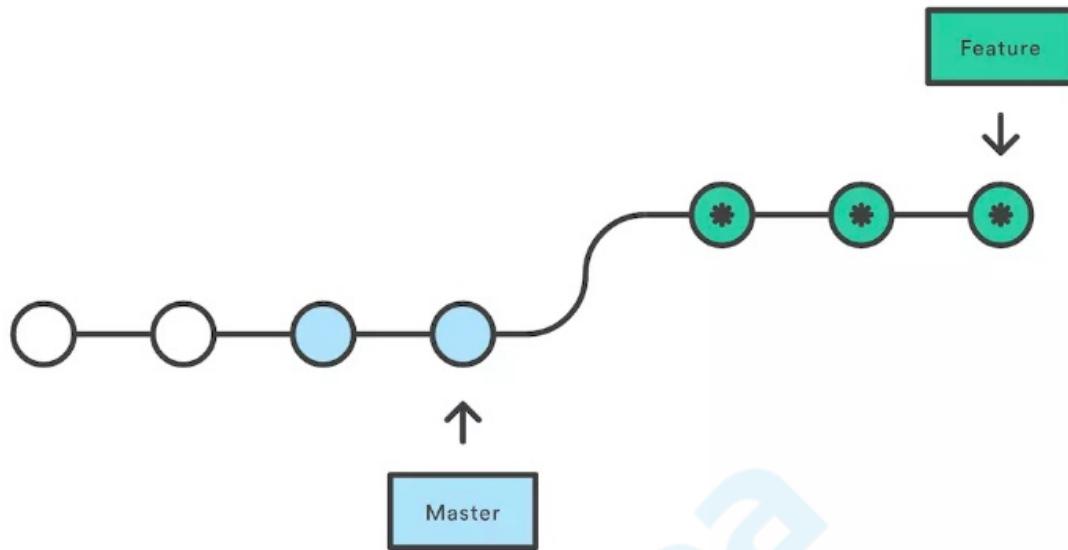
假设一个场景，就是我们开发的[feature/todo]分支要合并到master主分支，那么用rebase或者merge有什么不同呢？

Merging master into the feature branch



- **merge** 特点：自动创建一个新的commit 如果合并的时候遇到冲突，仅需要修改后重新commit
- 优点：记录了真实的commit情况，包括每个分支的详情
- 缺点：因为每次merge会自动产生一个merge commit，所以在使用一些git 的GUI tools，特别是commit比较频繁时，看到分支很杂乱。

Rebasing the feature branch onto master



- rebase 特点：会合并之前的commit历史
- 优点：得到更简洁的项目历史，去掉了merge commit
- 缺点：如果合并出现代码问题不容易定位，因为re-wrote了history

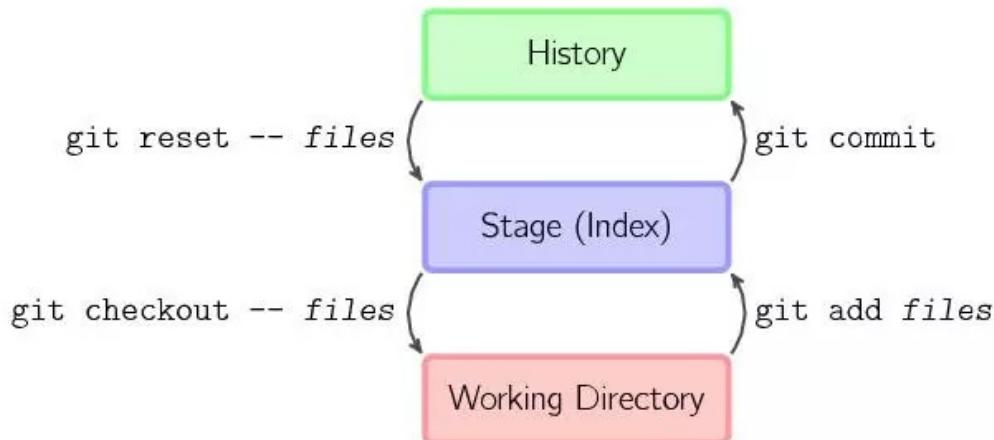
因此,当需要保留详细的合并信息的时候建议使用git merge, 特别是需要将分支合并进入master分支时; 当发现自己修改某个功能时, 频繁进行了git commit提交时, 发现其实过多的提交信息没有必要时, 可以尝试git rebase.

git reset、git revert 和 git checkout 有什么区别

这个问题同样也需要先了解 git 仓库的三个组成部分：工作区（Working Directory）、暂存区（Stage）和历史记录区（History）。

- 工作区：在 git 管理下的正常目录都算是工作区，我们平时的编辑工作都是在工作区完成
- 暂存区：临时区域。里面存放将要提交文件的快照
- 历史记录区：git commit 后的记录区

三个区的转换关系以及转换所使用的命令：



git reset、git revert 和 git checkout 的共同点：用来撤销代码仓库中的某些更改。

然后是不同点：

首先，从 commit 层面来说：

- git reset 可以将一个分支的末端指向之前的一个 commit。然后再下次 git 执行垃圾回收的时候，会把这个 commit 之后的 commit 都扔掉。git reset 还支持三种标记，用来标记 reset 指令影响的范围：
 - --mixed：会影响到暂存区和历史记录区。也是默认选项
 - --soft：只影响历史记录区
 - --hard：影响工作区、暂存区和历史记录区

注意：因为 git reset 是直接删除 commit 记录，从而会影响到其他开发人员的分支，所以不要在公共分支（比如 develop）做这个操作。

- git checkout 可以将 HEAD 移到一个新的分支，并更新工作目录。因为可能会覆盖本地的修改，所以执行这个指令之前，你需要 stash 或者 commit 暂存区和工作区的更改。
- git revert 和 git reset 的目的是一样的，但是做法不同，它会以创建新的 commit 的方式来撤销 commit，这样能保留之前的 commit 历史，比较安全。另外，同样因为可能会覆盖本地的修改，所以执行这个指令之前，你需要 stash 或者 commit 暂存区和工作区的更改。

然后，从文件层面来说：

- git reset 只是把文件从历史记录区拿到暂存区，不影响工作区的内容，而且不支持 --mixed、--soft 和 --hard。
- git checkout 则是把文件从历史记录拿到工作区，不影响暂存区的内容。
- git revert 不支持文件层面的操作。

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取





React面试题

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

React最新的生命周期是怎样的?

React 16之后有三个生命周期被废弃(但并未删除)

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

官方计划在17版本完全删除这三个函数, 只保留UNSAFE_前缀的三个函数, 目的是为了向下兼容, 但是对于开发者而言应该尽量避免使用他们, 而是使用新增的生命周期函数替代它们

目前React 16.8 + 的生命周期分为三个阶段, 分别是挂载阶段、更新阶段、卸载阶段

挂载阶段:

- constructor: 构造函数, 最先被执行, 我们通常在构造函数里初始化state对象或者给自定义方法绑定this
- getDerivedStateFromProps: static getDerivedStateFromProps(nextProps, prevState), 这是个静态方法, 当我们接收到新的属性想去修改我们state, 可以使用getDerivedStateFromProps
- render: render函数是纯函数, 只返回需要渲染的东西, 不应该包含其它的业务逻辑, 可以返回原生的DOM、React组件、Fragment、Portals、字符串和数字、Boolean和null等内容
- componentDidMount: 组件装载之后调用, 此时我们可以获取到DOM节点并操作, 比如对canvas, svg的操作, 服务器请求, 订阅都可以写在这个里面, 但是记得在componentWillUnmount中取消订阅

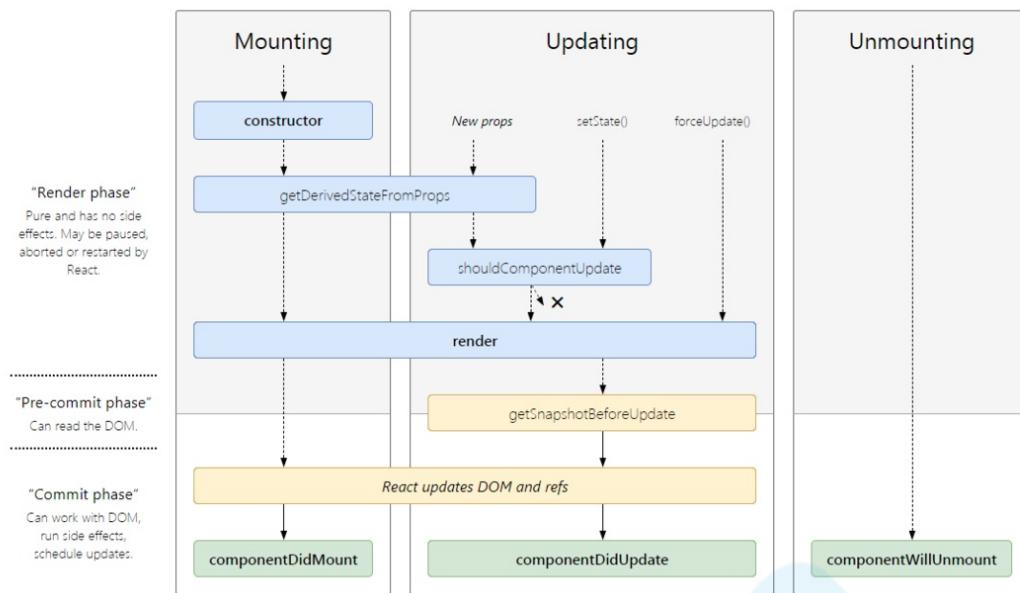
更新阶段:

- getDerivedStateFromProps: 此方法在更新个挂载阶段都可能会调用
- shouldComponentUpdate: shouldComponentUpdate(nextProps, nextState), 有两个参数nextProps和nextState, 表示新的属性和变化之后的state, 返回一个布尔值, true表示会触发重新渲染, false表示不会触发重新渲染, 默认返回true, 我们通常利用此生命周期来优化React程序性能
- render: 更新阶段也会触发此生命周期
- getSnapshotBeforeUpdate: getSnapshotBeforeUpdate(prevProps, prevState), 这个方法在render之后, componentDidUpdate之前调用, 有两个参数prevProps和prevState, 表示之前的属性和之前的state, 这个函数有一个返回值, 会作为第三个参数传给componentDidUpdate, 如果你不想要返回值, 可以返回null, 此生命周期必须与componentDidUpdate搭配使用
- componentDidUpdate: componentDidUpdate(prevProps, prevState, snapshot), 该方法在getSnapshotBeforeUpdate方法之后被调用, 有三个参数prevProps, prevState, snapshot, 表示之前的props, 之前的state, 和snapshot. 第三个参数是getSnapshotBeforeUpdate返回的, 如果触发某些回调函数时需要用到DOM元素的状态, 则将对比或计算的过程迁移至getSnapshotBeforeUpdate, 然后在componentDidUpdate中统一触发回调或更新状态。

卸载阶段:

- componentWillUnmount: 当我们的组件被卸载或者销毁了就会调用, 我们可以在这个函数里去清除一些定时器, 取消网络请求, 清理无效的DOM元素等垃圾清理工作

Show less common lifecycles



一个查看react生命周期的[网站](#)

React的请求应该放在哪个生命周期中？

React的异步请求到底应该放在哪个生命周期里,有人认为在 `componentWillMount` 中可以提前进行异步请求,避免白屏,其实这个观点是有问题的.

由于JavaScript中异步事件的性质,当您启动API调用时,浏览器会在此期间返回执行其他工作。当React渲染一个组件时,它不会等待`componentWillMount`它完成任何事情 - React继续前进并继续render,没有办法“暂停”渲染以等待数据到达。

而且在 `componentWillMount` 请求会有一系列潜在的问题,首先,在服务器渲染时,如果在 `componentWillMount` 里获取数据, `fetch data`会执行两次,一次在服务端一次在客户端,这造成了多余的请求,其次,在React 16进行React Fiber重写后, `componentWillMount` 可能在一次渲染中多次调用.

目前官方推荐的异步请求是在 `componentDidMount` 中进行.

如果有特殊需求需要提前请求,也可以在特殊情况下在 `constructor` 中请求:

react 17之后 `componentWillMount` 会被废弃,仅仅保留 `UNSAFE_componentWillMount`

setState到底是异步还是同步?

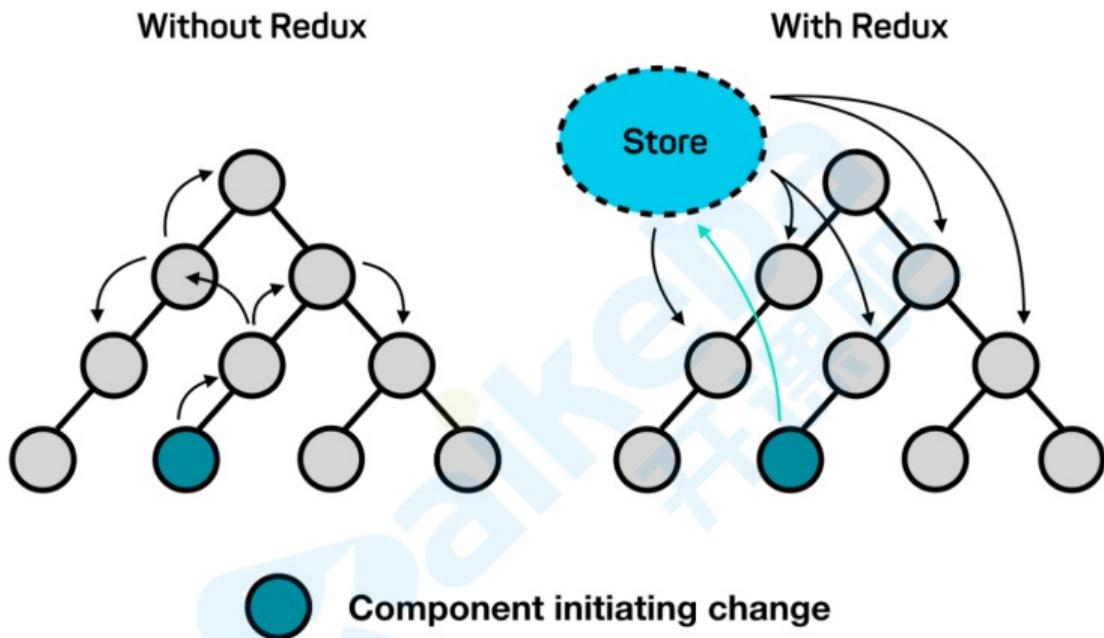
先给出答案: 有时表现出异步,有时表现出同步

- `setState` 只在合成事件和钩子函数中是“异步”的, 在原生事件和 `setTimeout` 中都是同步的。
- `setState` 的“异步”并不是说内部由异步代码实现, 其实本身执行的过程和代码都是同步的, 只是合成事件和钩子函数的调用顺序在更新之前, 导致在合成事件和钩子函数中没法立马拿到更新后的值, 形成了所谓的“异步”, 当然可以通过第二个参数 `setState(partialState, callback)` 中的 `callback` 拿到更新后的结果。
- `setState` 的批量更新优化也是建立在“异步”(合成事件、钩子函数)之上的, 在原生事件和`setTimeout`中不会批量更新, 在“异步”中如果对同一个值进行多次 `setState`, `setState` 的批量更新策略会对其进行覆盖, 取最后一次的执行, 如果是同时 `setState` 多个不同的值, 在更新时会对其进行合并批量更新。

React组件通信如何实现？

React组件间通信方式：

- 父组件向子组件通讯：父组件可以向子组件通过传 props 的方式，向子组件进行通讯
- 子组件向父组件通讯：props+回调的方式,父组件向子组件传递props进行通讯，此props为作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中
- 兄弟组件通信：找到这两个兄弟节点共同的父节点,结合上面两种方式由父节点转发信息进行通信
- 跨层级通信：Context 设计目的是为了共享那些对于一个组件树而言是“全局”的数据，例如当前认证的用户、主题或首选语言,对于跨多层的全局数据通过 Context 通信再适合不过
- 发布订阅模式：发布者发布事件，订阅者监听事件并做出反应,我们可以通过引入event模块进行通信
- 全局状态管理工具：借助Redux或者Mobx等全局状态管理工具进行通信,这种工具会维护一个全局状态中心Store,并根据不同的事件产生新的状态



React有哪些优化性能手段？

性能优化的手段很多时候是通用的详情见[前端性能优化加载篇](#)

React如何进行组件/逻辑复用？

抛开已经被官方弃用的Mixin,组件抽象的技术目前有三种比较主流：

- 高阶组件:
 - 属性代理
 - 反向继承
- 渲染属性
- react-hooks

组件复用详解见[组件复用](#)

mixin、hoc、render props、react-hooks的优劣如何？

Mixin的缺陷:

- 组件与 Mixin 之间存在隐式依赖 (Mixin 经常依赖组件的特定方法, 但在定义组件时并不知道这种依赖关系)
- 多个 Mixin 之间可能产生冲突 (比如定义了相同的state字段)
- Mixin 倾向于增加更多状态, 这降低了应用的可预测性 (The more state in your application, the harder it is to reason about it.) , 导致复杂度剧增
- 隐式依赖导致依赖关系不透明, 维护成本和理解成本迅速攀升:
 - 难以快速理解组件行为, 需要全盘了解所有依赖 Mixin 的扩展行为, 及其之间的相互影响
 - 组合自身的方法和state字段不敢轻易删改, 因为难以确定有没有 Mixin 依赖它
 - Mixin 也难以维护, 因为 Mixin 逻辑最后会被打平合并到一起, 很难搞清楚一个 Mixin 的输入输出

HOC相比Mixin的优势:

- HOC通过外层组件通过 Props 影响内层组件的状态, 而不是直接改变其 State不存在冲突和互相干扰,这就降低了耦合度
- 不同于 Mixin 的打平+合并, HOC 具有天然的层级结构 (组件树结构) , 这又降低了复杂度

HOC的缺陷:

- 扩展性限制: HOC 无法从外部访问子组件的 State因此无法通过shouldComponentUpdate滤掉不必要的更新,React 在支持 ES6 Class 之后提供了React.PureComponent来解决这个问题
- Ref 传递问题: Ref 被隔断,后来的React.forwardRef 来解决这个问题
- Wrapper Hell: HOC可能出现多层包裹组件的情况,多层抽象同样增加了复杂度和理解成本
- 命名冲突: 如果高阶组件多次嵌套,没有使用命名空间的话会产生冲突,然后覆盖老属性
- 不可见性: HOC相当于在原有组件外层再包装一个组件,你压根不知道外层的包装是啥,对于你是黑盒

Render Props优点:

- 上述HOC的缺点Render Props都可以解决

Render Props缺陷:

- 使用繁琐: HOC使用只需要借助装饰器语法通常一行代码就可以进行复用,Render Props无法做到如此简单
- 嵌套过深: Render Props虽然摆脱了组件多层嵌套的问题,但是转化为了函数回调的嵌套

React Hooks优点:

- 简洁: React Hooks解决了HOC和Render Props的嵌套问题,更加简洁
- 解耦: React Hooks可以更方便地把 UI 和状态分离,做到更彻底的解耦
- 组合: Hooks 中可以引用另外的 Hooks形成新的Hooks,组合变化万千
- 函数友好: React Hooks为函数组件而生,从而解决了类组件的几大问题:
 - this 指向容易错误
 - 分割在不同声明周期中的逻辑使得代码难以理解和维护
 - 代码复用成本高 (高阶组件容易使代码量剧增)

React Hooks缺陷:

- 额外的学习成本 (Functional Component 与 Class Component 之间的困惑)
- 写法上有限制 (不能出现在条件、循环中) , 并且写法限制增加了重构成本
- 破坏了 PureComponent、React.memo浅比较的性能优化效果 (为了取最新的props和state, 每次render()都要重新创建事件处函数)
- 在闭包场景可能会引用到旧的state、props值

- 内部实现上不直观（依赖一份可变的全局状态，不再那么“纯”）
- React.memo并不能完全替代shouldComponentUpdate（因为拿不到 state change，只针对 props change）

关于react-hooks的评价来源于官方[react-hooks RFC](#)

你是如何理解fiber的？

React Fiber 是一种基于浏览器的单线程调度算法。

React 16之前，reconciliation 算法实际上是递归，想要中断递归是很困难的，React 16 开始使用了循环来代替之前的递归。

Fiber：一种将 reconciliation（递归 diff），拆分成无数个小任务的算法；它随时能够停止，恢复。停止恢复的时机取决于当前的一帧（16ms）内，还有没有足够的时间允许计算。

你对 Time Slice的理解？

时间分片

- React 在渲染（render）的时候，不会阻塞现在的线程
- 如果你的设备足够快，你会感觉渲染是同步的
- 如果你设备非常慢，你会感觉还算是灵敏的
- 虽然是异步渲染，但是你将会看到完整的渲染，而不是一个组件一行行的渲染出来
- 同样书写组件的方式

也就是说，这是React背后在做的事情，对于我们开发者来说，是透明的，具体是什么样的效果呢？



有图表三个图表，有一个输入框，以及上面的三种模式

这个组件非常的巨大，而且在输入框每次输入东西的时候，就会进去一直在渲染。为了更好的看到渲染的性能，Dan为我们做了一个表。

我们先看看，同步模式：



同步模式下，我们都知道，我们没输入一个字符，React就开始渲染，当React渲染一颗巨大的树的时候，是非常卡的，所以才会有shouldUpdate的出现，在这里Dan也展示了，这种卡！

我们再来看看第二种（Debounced模式）：



Debounced模式简单的来说，就是延迟渲染，比如，当你输入完成以后，再开始渲染所有的变化。

这么做的坏处就是，至少不会阻塞用户的输入了，但是依然有非常严重的卡顿。

切换到异步模式：



异步渲染模式就是不阻塞当前线程，继续跑。在视频里可以看到所有的输入，表面上都会是原谅色的。

时间分片正是基于可随时打断、重启的Fiber架构，可打断当前任务，优先处理紧急且重要的任务，保证页面的流畅运行。

redux的工作流程？

首先，我们看下几个核心概念：

- Store：保存数据的地方，你可以把它看成一个容器，整个应用只能有一个Store。
- State：Store对象包含所有数据，如果想得到某个时点的数据，就要对Store生成快照，这种时点的数据集合，就叫

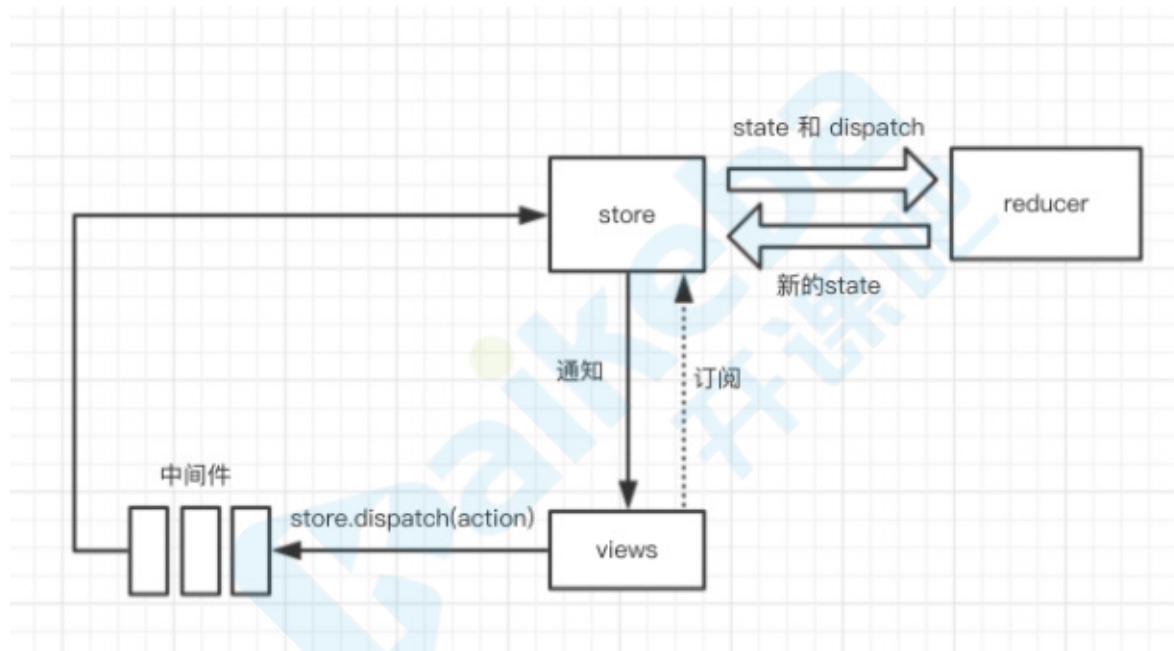
做State。

- Action: State的变化，会导致View的变化。但是，用户接触不到State，只能接触到View。所以，State的变化必须是View导致的。Action就是View发出的通知，表示State应该要发生变化了。
- Action Creator: View要发送多少种消息，就会有多少种Action。如果都手写，会很麻烦，所以我们定义一个函数来生成Action，这个函数就叫Action Creator。
- Reducer: Store收到Action以后，必须给出一个新的State，这样View才会发生变化。这种State的计算过程就叫做Reducer。Reducer是一个函数，它接受Action和当前State作为参数，返回一个新的State。
- dispatch: 是View发出Action的唯一方法。

然后我们过下整个工作流程：

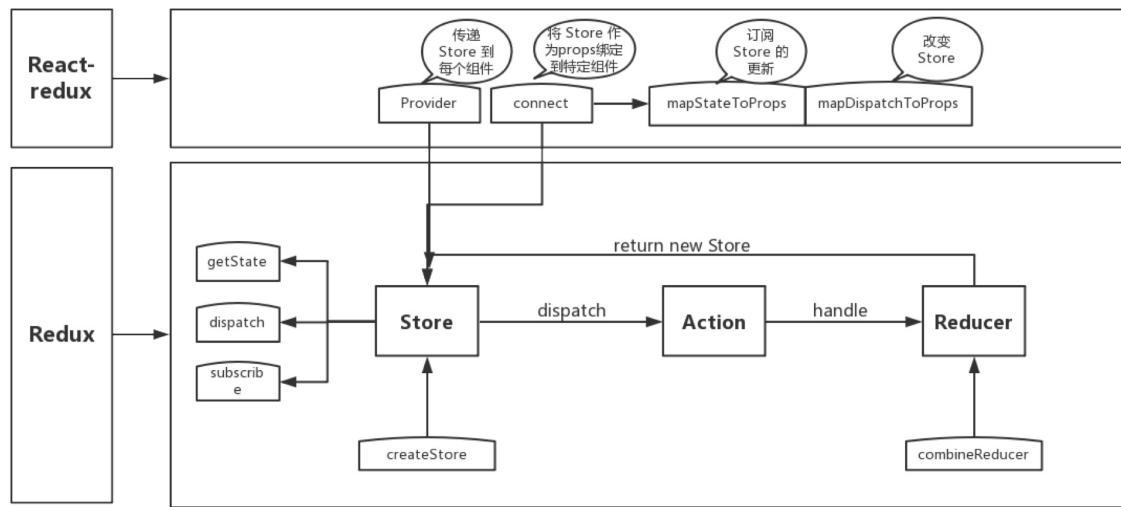
1. 首先，用户（通过View）发出Action，发出方式就用到了dispatch方法。
2. 然后，Store自动调用Reducer，并且传入两个参数：当前State和收到的Action，Reducer会返回新的State
3. State一旦有变化，Store就会调用监听函数，来更新View。

到这儿为止，一次用户交互流程结束。可以看到，在整个流程中数据都是单向流动的，这种方式保证了流程的清晰。



react-redux是如何工作的？

- Provider: Provider的作用是从最外部封装了整个应用，并向connect模块传递store
- connect: 负责连接React和Redux
 - 获取state: connect通过context获取Provider中的store，通过store.getState()获取整个store tree 上所有state
 - 包装原组件: 将state和action通过props的方式传入到原组件内部wrapWithConnect返回一个ReactComponent对象Connect，Connect重新render外部传入的原组件WrappedComponent，并把connect中传入的mapStateToProps, mapDispatchToProps与组件上原有的props合并后，通过属性的方式传给WrappedComponent
 - 监听store tree变化: connect缓存了store tree中state的状态，通过当前state状态和变更前state状态进行比较，从而确定是否调用 this.setState() 方法触发Connect及其子组件的重新渲染



redux与mobx的区别?

两者对比:

- redux将数据保存在单一的store中, mobx将数据保存在分散的多个store中
- redux使用plain object保存数据, 需要手动处理变化后的操作; mobx适用observable保存数据, 数据变化后自动处理响应的操作
- redux使用不可变状态, 这意味着状态是只读的, 不能直接去修改它, 而是应该返回一个新的状态, 同时使用纯函数; mobx中的状态是可变的, 可以直接对其进行修改
- mobx相对来说比较简单, 在其中有很多的抽象, mobx更多的使用面向对象的编程思维; redux会比较复杂, 因为其中的函数式编程思想掌握起来不是那么容易, 同时需要借助一系列的中间件来处理异步和副作用
- mobx中有更多的抽象和封装, 调试会比较困难, 同时结果也难以预测; 而redux提供能够进行时间回溯的开发工具, 同时其纯函数以及更少的抽象, 让调试变得更加容易

场景辨析:

基于以上区别,我们可以简单得分析一下两者的不同使用场景.

mobx更适合数据不复杂的应用: mobx难以调试,很多状态无法回溯,面对复杂度高的应用时,往往力不从心.

redux适合有回溯需求的应用: 比如一个画板应用、一个表格应用, 很多时候需要撤销、重做等操作, 由于redux不可变的特性, 天然支持这些操作.

mobx适合短平快的项目: mobx上手简单,样板代码少,可以很大程度上提高开发效率.

当然mobx和redux也并不一定是非此即彼的关系,你也可以在项目中用redux作为全局状态管理,用mobx作为组件局部状态管理器来用.

redux中如何进行异步操作?

当然,我们可以在 `componentDidMount` 中直接进行请求无须借助redux.

但是在一定规模的项目中,上述方法很难进行异步流的管理,通常情况下我们会借助redux的异步中间件进行异步处理.

redux异步流中间件其实有很多,但是当下主流的异步中间件只有两种redux-thunk、redux-saga, 当然redux-observable可能也有资格占据一席之地,其余的异步中间件不管是社区活跃度还是npm下载量都比较差了.

redux异步中间件之间的优劣？

redux-thunk优点:

- 体积小: redux-thunk的实现方式很简单,只有不到20行代码
- 使用简单: redux-thunk没有引入像redux-saga或者redux-observable额外的范式,上手简单

redux-thunk缺陷:

- 样板代码过多: 与redux本身一样,通常一个请求需要大量的代码,而且很多都是重复性质的
- 耦合严重: 异步操作与redux的action偶合在一起,不方便管理
- 功能孱弱: 有一些实际开发中常用的功能需要自己进行封装

redux-saga优点:

- 异步解耦: 异步操作被转移到单独 saga.js 中, 不再是掺杂在 action.js 或 component.js 中
- action摆脱thunk function: dispatch 的参数依然是一个纯粹的 action (FSA), 而不是充满“黑魔法”thunk function
- 异常处理: 受益于 generator function 的 saga 实现, 代码异常/请求失败 都可以直接通过 try/catch 语法直接捕获处理
- 功能强大: redux-saga提供了大量的Saga 辅助函数和Effect 创建器供开发者使用,开发者无须封装或者简单封装即可使用
- 灵活: redux-saga可以将多个Saga可以串行/并行组合起来,形成一个非常实用的异步flow
- 易测试, 提供了各种case的测试方案, 包括mock task, 分支覆盖等等

redux-saga缺陷:

- 额外的学习成本: redux-saga不仅在使用难以理解的 generator function,而且有数十个API,学习成本远超redux-thunk,最重要的是你的额外学习成本是只服务于这个库的,与redux-observable不同,redux-observable虽然也有额外学习成本但是背后是rxjs和一整套思想
- 体积庞大: 体积略大,代码近2000行, min版25KB左右
- 功能过剩: 实际上并发控制等功能很难用到,但是我们依然需要引入这些代码
- ts支持不友好: yield无法返回TS类型

redux-observable优点:

- 功能最强: 由于背靠rxjs这个强大的响应式编程的库,借助rxjs的操作符,你可以几乎做任何你能想到的异步处理
- 背靠rxjs: 由于有rxjs的加持,如果你已经学习了rxjs,redux-observable的学习成本并不高,而且随着rxjs的升级redux-observable也会变得更强大

redux-observable缺陷:

- 学习成本奇高: 如果你不会rxjs,则需要额外学习两个复杂的库
- 社区一般: redux-observable的下载量只有redux-saga的1/5,社区也不够活跃,在复杂异步流中间件这个层面redux-saga仍处于领导地位

关于redux-saga与redux-observable的详细比较可见[此链接](#)

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



Vue面试题

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

Vue框架部分我们会涉及一些高频且有一定探讨价值的面试题,我们不会涉及一些非常初级的在官方文档就能查看的纯记忆性质的面试题,比如:

- vue常用的修饰符?
- vue-cli 工程常用的 npm 命令有哪些?
- vue中 keep-alive 组件的作用?

首先,上述类型的面试题在文档中可查,没有比官方文档更权威的答案了,其次这种问题没有太大价值,除了考察候选人的记忆力,最后,这种面试题只要用过vue的都知道,没有必要占用我们的篇幅.

我们的问题并不多,但是难度可能会高一些,如果你真的搞懂了这些问题,在绝大多数情况下会有举一反三的效果,可以说基本能拿下Vue相关的所有重要知识点了.

你对MVVM的理解?

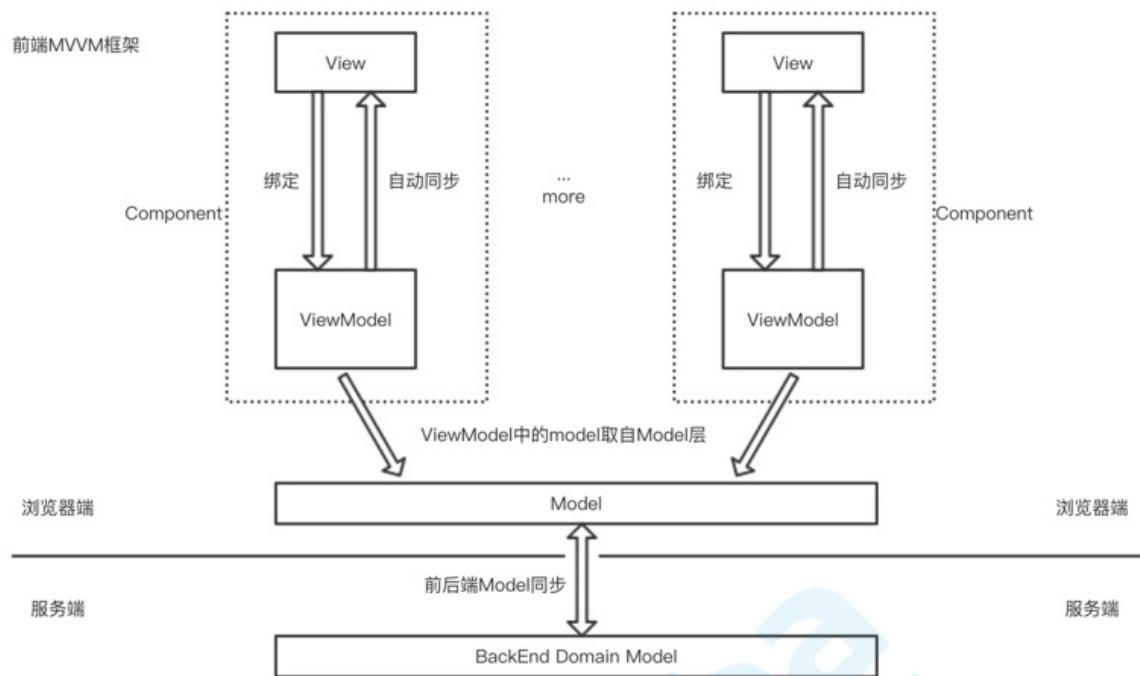
MVVM是什么?

MVVM 模式, 顾名思义即 Model-View-ViewModel 模式。它萌芽于2005年微软推出的基于 Windows 的用户界面框架 WPF , 前端最早的 MVVM 框架 knockout 在2010年发布。

Model 层: 对应数据层的域模型, 它主要做域模型的同步。通过 Ajax/fetch 等 API 完成客户端和服务端业务 Model 的同步。在层间关系里, 它主要用于抽象出 ViewModel 中视图的 Model。

View 层:作为视图模板存在, 在 MVVM 里, 整个 View 是一个动态模板。除了定义结构、布局外, 它展示的是 ViewModel 层的数据和状态。View 层不负责处理状态, View 层做的是 数据绑定的声明、指令的声明、事件绑定的声明。

ViewModel 层:把 View 需要的层数据暴露, 并对 View 层的数据绑定声明、指令声明、事件绑定声明 负责, 也就是处理 View 层的具体业务逻辑。ViewModel 底层会做好绑定属性的监听。当 ViewModel 中数据变化, View 层会得到更新; 而当 View 中声明了数据的双向绑定 (通常是表单元素) , 框架也会监听 View 层 (表单) 值的变化。一旦值变化, View 层绑定的 ViewModel 中的数据也会得到自动更新。



MVVM的优缺点？

优点:

- 分离视图（View）和模型（Model），降低代码耦合，提高视图或者逻辑的重用性: 比如视图（View）可以独立于 Model变化和修改，一个ViewModel可以绑定不同的"View"上，当View变化的时候Model不可以不变，当Model变化的时候View也可以不变。你可以把一些视图逻辑放在一个ViewModel里面，让很多view重用这段视图逻辑
- 提高可测试性: ViewModel的存在可以帮助开发者更好地编写测试代码
- 自动更新dom: 利用双向绑定,数据更新后视图自动更新,让开发者从繁琐的手动dom中解放

缺点:

- Bug很难被调试: 因为使用双向绑定的模式，当你看到界面异常了，有可能是你View的代码有Bug，也可能是Model的代码有问题。数据绑定使得一个位置的Bug被快速传递到别的位置，要定位原始出问题的地方就变得不容易了。另外，数据绑定的声明是指令式地写在View的模版当中的，这些内容是没办法去打断点debug的
- 一个大的模块中model也会很大，虽然使用方便了也很容易保证了数据的一致性，当时长期持有，不释放内存就造成了花费更多的内存
- 对于大型的图形应用程序，视图状态较多，ViewModel的构建和维护的成本都会比较高

你对Vue生命周期的理解？

生命周期是什么

Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模板、挂载Dom -> 渲染、更新 -> 渲染、卸载等一系列过程，我们称这是Vue的生命周期。

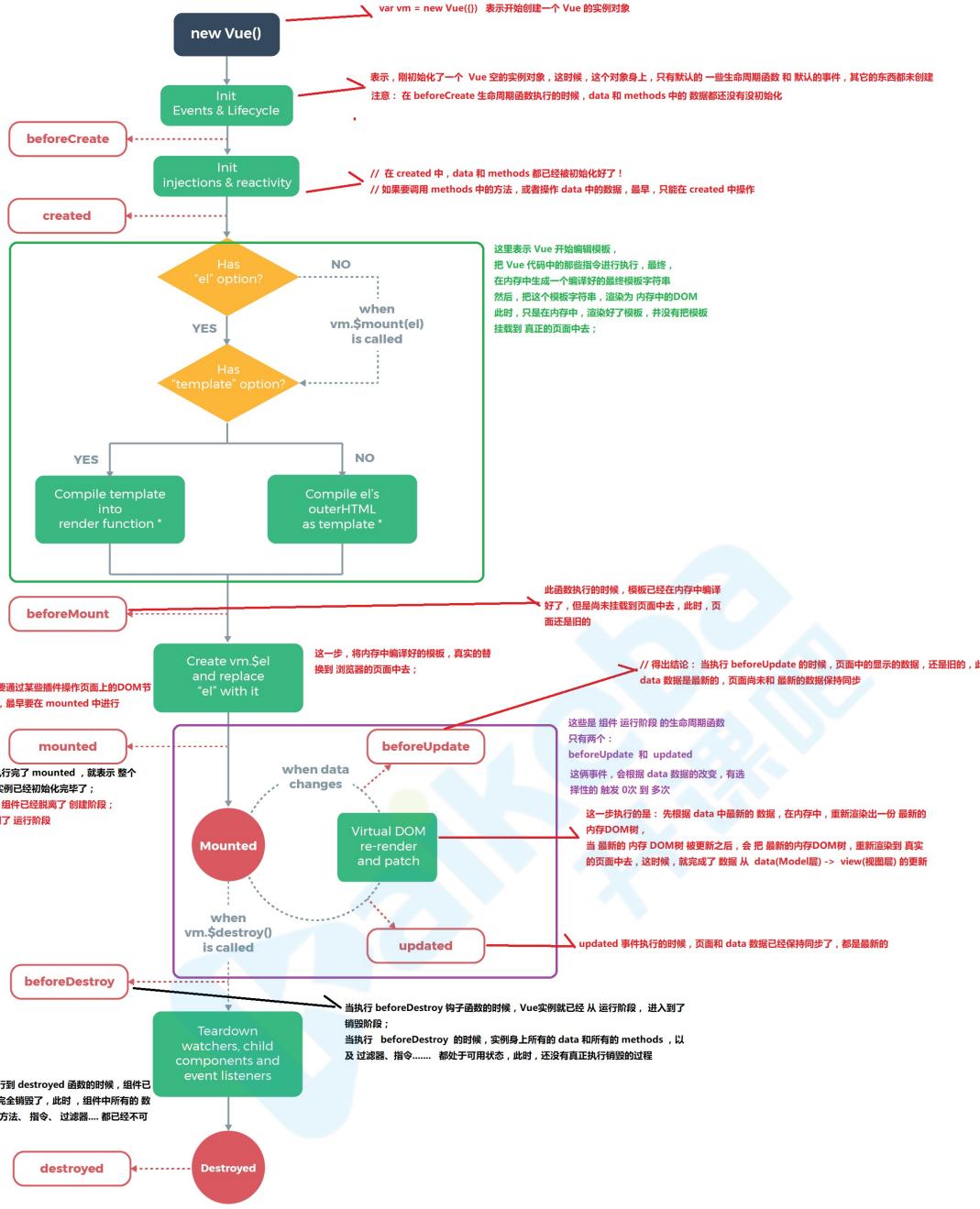
各个生命周期的作用

| 生命周期 | 描述 |
|--------------|---------------------|
| beforeCreate | 组件实例被创建之初，组件的属性生效之前 |

| | |
|---------------|--|
| created | 组件实例已经完全创建，属性也绑定，但真实dom还没有生成，\$el 还不可用 |
| beforeMount | 在挂载开始之前被调用：相关的 render 函数首次被调用 |
| mounted | el 被新创建的 vm.\$el 替换，并挂载到实例上去之后调用该钩子 |
| beforeUpdate | 组件数据更新之前调用，发生在虚拟 DOM 打补丁之前 |
| update | 组件数据更新之后 |
| activated | keep-alive专属，组件被激活时调用 |
| deactivated | keep-alive专属，组件被销毁时调用 |
| beforeDestory | 组件销毁前调用 |
| destoryed | 组件销毁后调用 |

生命周期示意图





异步请求适合在哪个生命周期调用？

官方实例的异步请求是在mounted生命周期中调用的，而实际上也可以在created生命周期中调用。

Vue组件如何通信？

Vue组件通信的方法如下：

- props/\$emit+v-on: 通过props将数据自上而下传递，而通过\$emit和v-on来向上传递信息。
- EventBus: 通过EventBus进行信息的发布与订阅
- vuex: 是全局数据管理库，可以通过vuex管理全局的数据流
- \$attrs/\$listeners: Vue2.4中加入的\$attrs/\$listeners可以进行跨级的组件通信

- `provide/inject`: 以允许一个祖先组件向其所有子孙后代注入一个依赖，不论组件层次有多深，并在起上下游关系成立的时间里始终生效，这成为了跨组件通信的基础

还有一些用`solt`插槽或者`ref`实例进行通信的，使用场景过于有限就不赘述了。

详细可以参考这篇文章[vue中8种组件通信方式](#)，不过太偏门的通信方式根本不会用到，单做知识点了解即可

computed和watch有什么区别？

`computed`:

1. `computed` 是计算属性，也就是计算值，它更多用于计算值的场景
2. `computed` 具有缓存性，`computed` 的值在`getter`执行后是会缓存的，只有在它依赖的属性值改变之后，下一次获取 `computed` 的值时才会重新调用对应的`getter`来计算
3. `computed` 适用于计算比较消耗性能的计算场景

`watch`:

1. 更多的是「观察」的作用，类似于某些数据的监听回调，用于观察 `props` `$emit` 或者本组件的值，当数据变化时来执行回调进行后续操作
2. 无缓存性，页面重新渲染时值不变化也会执行

小结：

1. 当我们要进行数值计算，而且依赖于其他数据，那么把这个数据设计为`computed`
2. 如果你需要在某个数据变化时做一些事情，使用`watch`来观察这个数据变化

Vue是如何实现双向绑定的？

利用 `Object.defineProperty` 劫持对象的访问器，在属性值发生变化时我们可以获取变化，然后根据变化进行后续响应，在 vue3.0 中通过 `Proxy` 代理对象进行类似的操作。

```
// 这是将要被劫持的对象
const data = {
  name: '',
};

function say(name) {
  if (name === '古天乐') {
    console.log('给大家推荐一款超好玩的游戏');
  } else if (name === '渣渣辉') {
    console.log('戏我演过很多，可游戏我只玩贪玩懒月');
  } else {
    console.log('来做我的兄弟');
  }
}

// 遍历对象，对其属性值进行劫持
Object.keys(data).forEach(function(key) {
  Object.defineProperty(data, key, {
    enumerable: true,
    configurable: true,
    get: function() {
      console.log('get');
    },
    set: function(newVal) {
      // 当属性值发生变化时我们可以进行额外操作
      console.log(`大家好，我系${newVal}`);
      say(newVal);
    },
  });
});
```

```

    });
    });

    data.name = '渣渣辉';
    //大家好,我系渣渣辉
    //戏我演过很多,可游戏我只玩贪玩懒月

```

详细实现见[Proxy比defineproperty优劣对比?](#)

Proxy与Object.defineProperty的优劣对比?

Proxy的优势如下:

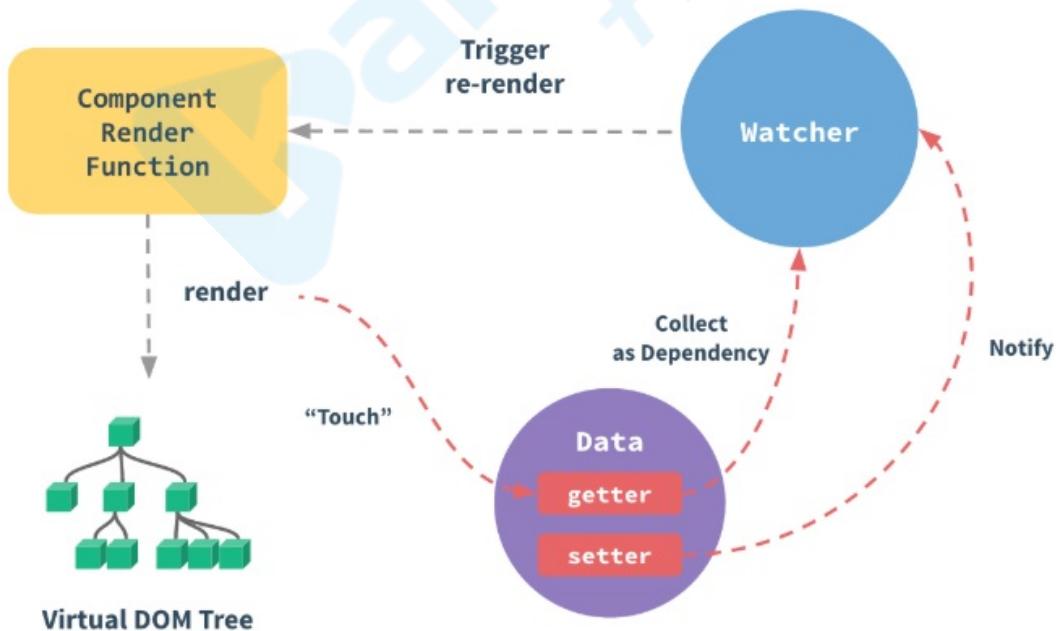
- Proxy可以直接监听对象而非属性
- Proxy可以直接监听数组的变化
- Proxy有多达13种拦截方法,不限于apply、ownKeys、deleteProperty、has等等是Object.defineProperty不具备的
- Proxy返回的是一个新对象,我们可以只操作新的对象达到目的,而Object.defineProperty只能遍历对象属性直接修改
- Proxy作为新标准将受到浏览器厂商重点持续的性能优化,也就是传说中的新标准的性能红利

Object.defineProperty的优势如下:

- 兼容性好,支持IE9

详细实现见[Proxy比defineproperty优劣对比?](#)

你是如何理解Vue的响应式系统的?



响应式系统简述:

- 任何一个 Vue Component 都有一个与之对应的 Watcher 实例。
- Vue 的 data 上的属性会被添加 getter 和 setter 属性。
- 当 Vue Component render 函数被执行的时候, data 上会被触碰(touch), 即被读, getter 方法会被调用, 此时 Vue 会

- 去记录此 Vue component 所依赖的所有 data。(这一过程被称为依赖收集)
- data 被改动时 (主要是用户操作), 即被写, setter 方法会被调用, 此时 Vue 会去通知所有依赖于此 data 的组件去调用他们的 render 函数进行更新。

深入响应式系统

既然Vue通过数据劫持可以精准探测数据变化,为什么还需要虚拟DOM进行diff检测差异?

考点: Vue的变化侦测原理

前置知识: 依赖收集、虚拟DOM、响应式系统

现代前端框架有两种方式侦测变化,一种是pull一种是push

pull: 其代表为React,我们可以回忆一下React是如何侦测到变化的,我们通常会用 `setState` API显式更新,然后React会进行一层层的Virtual Dom Diff操作找出差异,然后Patch到DOM上,React从一开始就不知道到底是哪发生了变化,只是知道「有变化了」,然后再进行比较暴力的Diff操作查找「哪发生变化了」, 另外一个代表就是Angular的脏检查操作。

push: Vue的响应式系统则是push的代表,当Vue程序初始化的时候就会对数据data进行依赖的收集,一但数据发生变化,响应式系统就会立刻得知,因此Vue是一开始就知道是「在哪发生了变化了」,但是这又会产生一个问题,如果你熟悉Vue的响应式系统就知道,通常一个绑定一个数据就需要一个Watcher,一但我们的绑定细粒度过高就会产生大量的Watcher,这会带来内存以及依赖追踪的开销,而细粒度过低会无法精准侦测变化,因此Vue的设计是选择中等细粒度的方案,在组件级别进行push侦测的方式,也就是那套响应式系统,通常我们会第一时间侦测到发生变化的组件,然后在组件内部进行Virtual Dom Diff获取更加具体的差异,而Virtual Dom Diff则是pull操作,Vue是push+pull结合的方式进行变化侦测的.

Vue为什么没有类似于React中shouldComponentUpdate的生命周期?

考点: Vue的变化侦测原理

前置知识: 依赖收集、虚拟DOM、响应式系统

根本原因是Vue与React的变化侦测方式有所不同

React是pull的方式侦测变化,当React知道发生变化后,会使用Virtual Dom Diff进行差异检测,但是很多组件实际上是肯定不会发生变化的,这个时候需要用`shouldComponentUpdate`进行手动操作来减少diff,从而提高程序整体的性能.

Vue是push+push的方式侦测变化的,在一开始就知道那个组件发生了变化,因此在push的阶段并不需要手动控制diff,而组件内部采用的diff方式实际上是可以引入类似于`shouldComponentUpdate`相关生命周期的,但是通常合理大小的组件不会有过多量的diff,手动优化的价值有限,因此目前Vue并没有考虑引入`shouldComponentUpdate`这种手动优化的生命周期.

Vue中的key到底有什么用?

`key` 是为Vue中的vnode标记的唯一id,通过这个key,我们的diff操作可以更准确、更快速

diff算法的过程中,先会进行新旧节点的首尾交叉对比,当无法匹配的时候会用新节点的 `key` 与旧节点进行比对,然后超出差异.

diff程可以概括为: oldCh和newCh各有两个头尾的变量StartIdx和EndIdx, 它们的2个变量相互比较, 一共有4种比较方式。如果4种比较都没匹配, 如果设置了key, 就会用key进行比较, 在比较的过程中, 变量会往中间靠, 一旦`StartIdx > EndIdx`表明oldCh和newCh至少有一个已经遍历完了, 就会结束比较,这四种比较方式就是首、尾、旧尾新头、旧头新尾.

- 准确:如果不加 key ,那么vue会选择复用节点(Vue的就地更新策略),导致之前节点的状态被保留下,会产生一系列的bug.
- 快速: key的唯一性可以被Map数据结构充分利用,相比于遍历查找的时间复杂度O(n),Map的时间复杂度仅仅为O(1).

```

60  function createKeyToOldIdx (children, beginIdx, endIdx) {
61    let i, key
62    const map = {}
63    for (i = beginIdx; i <= endIdx; ++i) {
64      key = children[i].key
65      if (isDef(key)) map[key] = i
66    }
67    return map
68  }

```

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取



前端安全面试题

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

有哪些可能引起前端安全的问题?

- 跨站脚本 (Cross-Site Scripting, XSS): 一种代码注入方式, 为了与 CSS 区分所以被称作 XSS. 早期常见于网络论坛, 起因是网站没有对用户的输入进行严格的限制, 使得攻击者可以将脚本上传到帖子让其他人浏览到有恶意脚本的页面, 其注入方式很简单包括但不限于 JavaScript / VBScript / CSS / Flash 等
- iframe的滥用: iframe中的内容是由第三方来提供的, 默认情况下他们不受我们的控制, 他们可以在iframe中运行 JavaScript脚本、Flash插件、弹出对话框等等, 这可能会破坏前端用户体验
- 跨站点请求伪造 (Cross-Site Request Forgery, CSRF) : 指攻击者通过设置好的陷阱, 强制对已完成认证的用户进行非预期的个人信息或设定信息等某些状态更新, 属于被动攻击
- 恶意第三方库: 无论是后端服务器应用还是前端应用开发, 绝大多数时候我们都是在借助开发框架和各种类库进行快速开发,一旦第三方库被植入恶意代码很容易引起安全问题,比如event-stream的恶意代码事件,2018年11月21日,名为 FallingSnow的用户在知名JavaScript应用库event-stream在github Issue中发布了针对植入的恶意代码的疑问, 表示event-stream中存在用于窃取用户数字钱包的恶意代码

XSS分为哪几类?

根据攻击的来源, XSS 攻击可分为存储型、反射型和 DOM 型三种。

- 存储区: 恶意代码存放的位置。
- 插入点: 由谁取得恶意代码, 并插入到网页上。

存储型 XSS

存储型 XSS 的攻击步骤:

1. 攻击者将恶意代码提交到目标网站的数据库中。
2. 用户打开目标网站时, 网站服务端将恶意代码从数据库取出, 拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行, 混在其中的恶意代码也被执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站, 或者冒充用户的行为, 调用目标网站接口执行攻击者指定的操作。

这种攻击常见于带有用户保存数据的网站功能, 如论坛发帖、商品评论、用户私信等。

反射型 XSS

反射型 XSS 的攻击步骤:

1. 攻击者构造出特殊的 URL, 其中包含恶意代码。
2. 用户打开带有恶意代码的 URL 时, 网站服务端将恶意代码从 URL 中取出, 拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行, 混在其中的恶意代码也被执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站, 或者冒充用户的行为, 调用目标网站接口执行攻击者指定的操作。

反射型 XSS 跟存储型 XSS 的区别是: 存储型 XSS 的恶意代码存在数据库里, 反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能, 如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效, 攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见。

DOM 型 XSS

DOM 型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL。
3. 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。
4. 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

如何预防XSS？

XSS 攻击有两大要素：

1. 攻击者提交恶意代码。
2. 浏览器执行恶意代码。

针对第一个要素：我们是否能够在用户输入的过程，过滤掉用户输入的恶意代码呢？

输入过滤

在用户提交时，由前端过滤输入，然后提交到后端。这样做是否可行呢？

答案是不可行。一旦攻击者绕过前端过滤，直接构造请求，就可以提交恶意代码了。

那么，换一个过滤时机：后端在写入数据库前，对输入进行过滤，然后把“安全的”内容，返回给前端。这样是否可行呢？

我们举一个例子，一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7`。

问题是：在提交阶段，我们并不确定内容要输出到哪里。

这里的“不确定内容要输出到哪里”有两层含义：

1. 用户的输入内容可能同时提供给前端和客户端，而一旦经过了 `escapeHTML()`，客户端显示的内容就变成了乱码(`5 < 7`)。
2. 在前端中，不同的位置所需的编码也不同。

- 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：

```
<div title="comment">5 &lt; 7</div>
```

- 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、alert 等

所以，输入侧过滤能够在某些情况下解决特定的 XSS 问题，但会引入很大的不确定性和乱码问题。在防范 XSS 攻击时应避免此类方法

当然，对于明确的输入类型，例如数字、URL、电话号码、邮件地址等等内容，进行输入过滤还是必要的

既然输入过滤并非完全可靠，我们就要通过“防止浏览器执行恶意代码”来防范 XSS。这部分分为两类：

- 防止 HTML 中出现注入

- 防止 JavaScript 执行时，执行恶意代码

预防存储型和反射型 XSS 攻击

存储型和反射型 XSS 都是在服务端取出恶意代码后，插入到响应 HTML 里的，攻击者刻意编写的“数据”被内嵌到“代码”中，被浏览器所执行。

预防这两种漏洞，有两种常见做法：

- 改成纯前端渲染，把代码和数据分隔开。
- 对 HTML 做充分转义。

纯前端渲染

纯前端渲染的过程：

1. 浏览器先加载一个静态 HTML，此 HTML 中不包含任何跟业务相关的数据。
2. 然后浏览器执行 HTML 中的 JavaScript。
3. JavaScript 通过 Ajax 加载业务数据，调用 DOM API 更新到页面上。

在纯前端渲染中，我们会明确的告诉浏览器：下面要设置的内容是文本（`.innerText`），还是属性

（`.setAttribute`），还是样式（`.style`）等等。浏览器不会被轻易的被欺骗，执行预期外的代码了。

但纯前端渲染还需注意避免 DOM 型 XSS 漏洞（例如 `onload` 事件和 `href` 中的 `javascript:xxx` 等，请参考下文“预防 DOM 型 XSS 攻击”部分）。

在很多内部、管理系统中，采用纯前端渲染是非常合适的。但对于性能要求高，或有 SEO 需求的页面，我们仍然要面对拼接 HTML 的问题。

转义 HTML

如果拼接 HTML 是必要的，就需要采用合适的转义库，对 HTML 模板各处插入点进行充分的转义。

常用的模板引擎，如 doT.js、ejs、FreeMarker 等，对于 HTML 转义通常只有一个规则，就是把 `& < > " ' /` 这几个字符转义掉，确实能起到一定的 XSS 防护作用，但并不完善：

|XSS 安全漏洞|简单转义是否有防护作用|--|HTML 标签文字内容|有|HTML 属性值|有||CSS 内联样式|无||内联 JavaScript|无||内联 JSON|无||跳转链接|无|

所以要完善 XSS 防护措施，我们要使用更完善更细致的转义策略。

例如 Java 工程里，常用的转义库为 `org.owasp.encoder`。以下代码引用自 [org.owasp.encoder 的官方说明](#)。

```

<!-- HTML 标签内文字内容 -->
<div><%= Encode.forHtml(UNTRUSTED) %></div>
<!-- HTML 标签属性值 -->
<input value="<%= Encode.forHtml(UNTRUSTED) %>" />
<!-- CSS 属性值 -->
<div style="width:<= Encode.forCssString(UNTRUSTED) %>">
<!-- CSS URL -->
<div style="background:<= Encode.forCssUrl(UNTRUSTED) %>">
<!-- JavaScript 内联代码块 -->
<script>
  var msg = "<%= Encode.forJavaScript(UNTRUSTED) %>";
  alert(msg);
</script>
<!-- JavaScript 内联代码块内嵌 JSON -->
<script>
  var __INITIAL_STATE__ = JSON.parse('<%= Encoder.forJavaScript(data.toJson()) %>');
</script>
<!-- HTML 标签内联监听器 -->
<button
  onclick="alert('<%= Encode.forJavaScript(UNTRUSTED) %>');">
  click me
</button>

```

```

<!-- URL 参数 -->
<a href="/search?value=<%= Encode.forURIComponent(UNTRUSTED) %>&order=1#top">
<!-- URL 路径 -->
<a href="/page/<%= Encode.forURIComponent(UNTRUSTED) %>">
<!--
URL.
注意: 要根据项目情况进行过滤, 禁止掉 "javascript:" 链接、非法 scheme 等
-->
<a href='<%= urlValidator.isValid(UNTRUSTED) ?
    Encode.forHtml(UNTRUSTED) :
    "/404"
%>'>
    link
</a>

```

可见, HTML 的编码是十分复杂的, 在不同的上下文里要使用相应的转义规则。

预防 DOM 型 XSS 攻击

DOM 型 XSS 攻击, 实际上就是网站前端 JavaScript 代码本身不够严谨, 把不可信的数据当作代码执行了。

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 时要特别小心, 不要把不可信的数据作为 HTML 插到页面上, 而应尽量使用 `.textContent`、`.setAttribute()` 等。

如果用 Vue/React 技术栈, 并且不使用 `v-html` / `dangerouslySetInnerHTML` 功能, 就在前端 render 阶段避免 `innerHTML`、`outerHTML` 的 XSS 隐患。

DOM 中的内联事件监听器, 如 `location`、`onclick`、`onerror`、`onload`、`onmouseover` 等, `<a>` 标签的 `href` 属性, JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等, 都能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API, 很容易产生安全隐患, 请务必避免。

```

<!-- 内联事件监听器中包含恶意代码 -->

<!-- 链接内包含恶意代码 -->
<a href="UNTRUSTED">1</a>
<script>
// setTimeout()/setInterval() 中调用恶意代码
setTimeout("UNTRUSTED")
setInterval("UNTRUSTED")
// location 调用恶意代码
location.href = 'UNTRUSTED'
// eval() 中调用恶意代码
eval("UNTRUSTED")
</script>

```

如果项目中有用到这些的话, 一定要避免在字符串中拼接不可信数据。

其他 XSS 防范措施

虽然在渲染页面和执行 JavaScript 时, 通过谨慎的转义可以防止 XSS 的发生, 但完全依靠开发的谨慎仍然是不够的。以下介绍一些通用的方案, 可以降低 XSS 带来的风险和后果。

Content Security Policy

严格的 CSP 在 XSS 的防范中可以起到以下的作用:

- 禁止加载外域代码, 防止复杂的攻击逻辑
- 禁止外域提交, 网站被攻击后, 用户的数据不会泄露到外域

- 禁止内联脚本执行（规则较严格，目前发现 GitHub 使用）
- 禁止未授权的脚本执行（新特性，Google Map 移动版在使用）
- 合理使用上报可以及时发现 XSS，利于尽快修复问题

输入内容长度控制

对于不受信任的输入，都应该限定一个合理的长度。虽然无法完全防止 XSS 发生，但可以增加 XSS 攻击的难度。

其他安全措施

- HTTP-only Cookie: 禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
- 验证码: 防止脚本冒充用户提交危险操作。

过滤 Html 标签能否防止 XSS？请列举不能的情况？

用户除了上传

```
<script>alert('xss');</script>
```

还可以使用图片 url 等方式来上传脚本进行攻击

```
<table background="javascript:alert(/xss/)"></table>

```

还可以使用各种方式来回避检查，例如空格，回车，Tab

```

```

还可以通过各种编码转换 (URL 编码, Unicode 编码, HTML 编码, ESCAPE 等) 来绕过检查

```
<img%20src=%22javascript:alert('xss');%22>

```

CSRF是什么？

CSRF (Cross-site request forgery) 跨站请求伪造：攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求。利用受害者的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目的。

一个典型的CSRF攻击有着如下的流程：

- 受害者登录 a.com，并保留了登录凭证（Cookie）
- 攻击者引诱受害者访问了 b.com
- b.com 向 a.com 发送了一个请求： a.com/act=xx 浏览器会默认携带a.com的Cookie
- a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求
- a.com以受害者的名义执行了act=xx
- 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作

CSRF的攻击类型？

GET类型的CSRF

GET类型的CSRF利用非常简单，只需要一个HTTP请求，一般会这样利用：

```
https://awps-assets.meituan.net/mit-x/blog-images-bundle-2018b/ff0cdbee.example/withdraw?amount=10000&for=hacker
```

在受害者访问含有这个img的页面后，浏览器会自动向 `http://bank.example/withdraw?account=xiaoming&amount=10000&for=hacker` 发出一次HTTP请求。`bank.example`就会收到包含受害者登录信息的一次跨域请求。

POST类型的CSRF

这种类型的CSRF利用起来通常使用的是一个自动提交的表单，如：

```
<form action="http://bank.example/withdraw" method=POST>
<input type="hidden" name="account" value="xiaoming" />
<input type="hidden" name="amount" value="10000" />
<input type="hidden" name="for" value="hacker" />
</form>
<script> document.forms[0].submit(); </script>
```

访问该页面后，表单会自动提交，相当于模拟用户完成了一次POST操作。

POST类型的攻击通常比GET要求更加严格一点，但仍并不复杂。任何个人网站、博客，被黑客上传页面的网站都有可能是发起攻击的来源，后端接口不能将安全寄托在仅允许POST上面。

链接类型的CSRF

链接类型的CSRF并不常见，比起其他两种用户打开页面就中招的情况，这种需要用户点击链接才会触发。这种类型通常是在论坛中发布的图片中嵌入恶意链接，或者以广告的形式诱导用户中招，攻击者通常会以比较夸张的词语诱骗用户点击，例如：

```
<a href="http://test.com/csrf/withdraw.php?amount=1000&for=hacker" target="_blank">
重磅消息！
<a/>
```

由于之前用户登录了信任的网站A，并且保存登录状态，只要用户主动访问上面的这个PHP页面，则表示攻击成功。

如何预防CSRF？

CSRF通常从第三方网站发起，被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性。

CSRF的两个特点：

- CSRF（通常）发生在第三方域名。
- CSRF攻击者不能获取到Cookie等信息，只是使用。

针对这两点，我们可以专门制定防护策略，如下：

- 阻止不明外域的访问
 - 同源检测
 - Samesite Cookie
- 提交时要求附加本域才能获取的信息
 - CSRF Token
 - 双重Cookie验证

因此我们可以针对性得进行预防

同源检测

既然CSRF大多来自第三方网站，那么我们就直接禁止外域（或者不受信任的域名）对我们发起请求：

- 使用Origin Header确定来源域名：在部分与CSRF有关的请求中，请求的Header中会携带Origin字段，如果Origin存在，那么直接使用Origin中的字段确认来源域名就可以
- 使用Referer Header确定来源域名：根据HTTP协议，在HTTP头中有一个字段叫Referer，记录了该HTTP请求的来源地址

CSRF Token

CSRF的另一个特征是，攻击者无法直接窃取到用户的信息（Cookie，Header，网站内容等），仅仅是冒用Cookie中的信息。

而CSRF攻击之所以能够成功，是因为服务器误把攻击者发送的请求当成了用户自己的请求。那么我们可以要求所有的用户请求都携带一个CSRF攻击者无法获取到的Token。服务器通过校验请求是否携带正确的Token，来把正常的请求和攻击的请求区分开，也可以防范CSRF的攻击：

CSRF Token的防护策略分为三个步骤：

- 将CSRF Token输出到页面中
- 页面提交的请求携带这个Token
- 服务器验证Token是否正确

双重Cookie验证

在会话中存储CSRF Token比较繁琐，而且不能在通用的拦截上统一处理所有的接口

那么另一种防御措施是使用双重提交Cookie。利用CSRF攻击不能获取到用户Cookie的特点，我们可以要求Ajax和表单请求携带一个Cookie中的值

双重Cookie采用以下流程：

- 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（例如 csrfcookie=v8g9e4ksfhw）。
- 在前端向后端发起请求时，取出Cookie，并添加到URL的参数中（接上例 POST <https://www.a.com/comment?csrfcookie=v8g9e4ksfhw>）。
- 后端接口验证Cookie中的字段与URL参数中的字段是否一致，不一致则拒绝。

Samesite Cookie属性

Google起草了一份草案来改进HTTP协议，那就是为Set-Cookie响应头新增Samesite属性，它用来标明这个Cookie是个“同站Cookie”，同站Cookie只能作为第一方Cookie，不能作为第三方Cookie，Samesite有两个属性值：

- Samesite=Strict：这种称为严格模式，表明这个Cookie在任何情况下都不可能作为第三方Cookie
- Samesite=Lax：这种称为宽松模式，比 Strict 放宽了点限制，假如这个请求是这种请求且同时是个GET请求，则这个Cookie可以作为第三方Cookie

网络劫持有哪几种？

网络劫持一般分为两种：

- DNS劫持：（输入京东被强制跳转到淘宝这就属于dns劫持）
 - DNS强制解析：通过修改运营商的本地DNS记录，来引导用户流量到缓存服务器
 - 302跳转的方式：通过监控网络出口的流量，分析判断哪些内容是可以进行劫持处理的，再对劫持的内存发起302

跳转的回复，引导用户获取内容

- HTTP劫持：(访问谷歌但是一直有贪玩蓝月的广告),由于http明文传输,运营商会修改你的http响应内容(即加广告)

如何应对网络劫持？

DNS劫持由于涉嫌违法,已经被监管起来,现在很少会有DNS劫持,而http劫持依然非常盛行.

最有效的办法就是全站HTTPS,将HTTP加密,这使得运营商无法获取明文,就无法劫持你的响应内容.

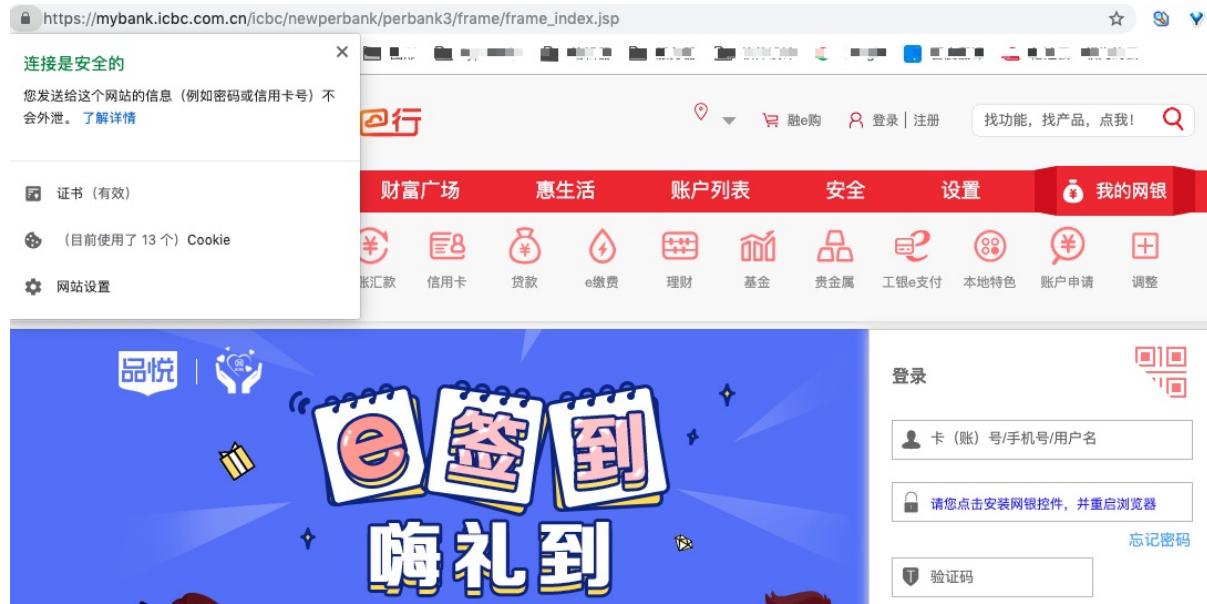
HTTPS一定是安全的吗？

非全站HTTPS并不安全

以国内的工商银行为例



工商银行的首页不支持HTTPS



而工商银行的网银页面是支持HTTPS的

可能有人会问,登录页面支持HTTPS不就行了,首页又没有涉及账户信息.

其实这是非常不安全的行为,黑客会利用这一点进行攻击,一般是以以下流程:

1. 用户在首页点击「登录」, 页面跳转到有https的网银页面,但此时由于首页是http请求,所以是明文的,这就会被黑客劫持
2. 黑客劫持用户的跳转请求,将https网银页面地址转换为http的地址再发送给银行

用户 <== HTTP ==> 黑客 <== HTTPS ==> 银行

1. 此时如果用户输入账户信息,那么会被中间的黑客获取,此时的账号密码就被泄露了

好在是工商银行的网银页面应该是开启了hsts和pre load,只支持https,因此上述攻击暂时是无效的.

中间人攻击

中间人 (Man-in-the-middle attack, MITM) 是指攻击者与通讯的两端分别创建独立的联系, 并交换其所收到的数据, 使通讯的两端认为他们正在通过一个私密的连接与对方直接对话, 但事实上整个会话都被攻击者完全控制. 在中间人攻击中, 攻击者可以拦截通讯双方的通话并插入新的内容.

一般的过程如下:

- 客户端发送请求到服务端, 请求被中间人截获
- 服务器向客户端发送公钥
- 中间人截获公钥, 保留在自己手上. 然后自己生成一个【伪造的】公钥, 发给客户端
- 客户端收到伪造的公钥后, 生成加密hash值发给服务器
- 中间人获得加密hash值, 用自己的私钥解密获得真秘钥, 同时生成假的加密hash值, 发给服务器
- 服务器用私钥解密获得假密钥, 然后加密数据传输给客户端

HTTPS中间人攻击实践

强烈建议阅读下面两篇前端安全文章:

[前端安全系列（一）：如何防止XSS攻击？](#)

前端安全系列（二）：如何防止CSRF攻击？

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



webpack面试题

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

webpack是事实上的前端打包标准,相关的面试题也是面试的热点.

webpack与grunt、gulp的不同?

Grunt、Gulp是基于任务运行的工具:

它们会自动执行指定的任务,就像流水线,把资源放上去然后通过不同插件进行加工,它们包含活跃的社区,丰富的插件,能方便的打造各种工作流。

Webpack是基于模块化打包的工具:

自动化处理模块,webpack把一切当成模块,当 webpack 处理应用程序时,它会递归地构建一个依赖关系图(dependency graph),其中包含应用程序需要的每个模块,然后将所有这些模块打包成一个或多个 bundle。

因此这是完全不同的两类工具,而现在主流的方式是用npm script代替Grunt、Gulp,npm script同样可以打造任务流.

webpack、rollup、parcel优劣?

- webpack适用于大型复杂的前端站点构建: webpack有强大的loader和插件生态,打包后的文件实际上就是一个立即执行函数,这个立即执行函数接收一个参数,这个参数是模块对象,键为各个模块的路径,值为模块内容。立即执行函数内部则处理模块之间的引用,执行模块等,这种情况更适合文件依赖复杂的应用开发.
- rollup适用于基础库的打包,如vue、d3等: Rollup 就是将各个模块打包进一个文件中,并且通过 Tree-shaking 来删除无用的代码,可以最大程度上降低代码体积,但是rollup没有webpack如此多的如代码分割、按需加载等高级功能,其更聚焦于库的打包,因此更适合库的开发.
- parcel适用于简单的实验性项目: 他可以满足低门槛的快速看到效果,但是生态差、报错信息不够全面都是他的硬伤,除了一些玩具项目或者实验项目不建议使用

```
(function(modules) { // webpackBootstrap
  /* ... */
})({  

  "./src/hello.js": (function(module, __webpack_exports__, __webpack_require__)  

    "use strict";  

    eval("__webpack_require__.r(__webpack_exports__);\\nlet a = 1; // 无用代码\\n");  

  }),  

  "./src/main.js": (function(module, __webpack_exports__, __webpack_require__)  

    "use strict";  

    eval("__webpack_require__.r(__webpack_exports__);\\n/* harmony import */ v  

  })
});
```

```
'use strict';

var hello = 'hello world!';

// src/main.js
function main () {
  console.log(hello);
}

module.exports = main;
```

有哪些常见的Loader?

- file-loader: 把文件输出到一个文件夹中, 在代码中通过相对 URL 去引用输出的文件
- url-loader: 和 file-loader 类似, 但是能在文件很小的情况下以 base64 的方式把文件内容注入到代码中去
- source-map-loader: 加载额外的 Source Map 文件, 以方便断点调试
- image-loader: 加载并且压缩图片文件
- babel-loader: 把 ES6 转换成 ES5
- css-loader: 加载 CSS, 支持模块化、压缩、文件导入等特性
- style-loader: 把 CSS 代码注入到 JavaScript 中, 通过 DOM 操作去加载 CSS。
- eslint-loader: 通过 ESLint 检查 JavaScript 代码

有哪些常见的Plugin?

- define-plugin: 定义环境变量
- html-webpack-plugin: 简化html文件创建
- uglifyjs-webpack-plugin: 通过 uglifyES 压缩 es6 代码
- webpack-parallel-uglify-plugin: 多核压缩,提高压缩速度
- webpack-bundle-analyzer: 可视化webpack输出文件的体积
- mini-css-extract-plugin: CSS提取到单独的文件中,支持按需加载

分别介绍bundle, chunk, module是什么

- bundle: 是由webpack打包出来的文件
- chunk: 代码块, 一个chunk由多个模块组合而成, 用于代码的合并和分割
- module: 是开发中的单个模块, 在webpack的世界, 一切皆模块, 一个模块对应一个文件, webpack会从配置的 entry中递归开始找出所有依赖的模块

Loader和Plugin的不同?

不同的作用:

- **Loader**直译为"加载器"。Webpack将一切文件视为模块, 但是webpack原生是只能解析js文件, 如果想将其他文件也打包的话, 就会用到 loader 。 所以Loader的作用是让webpack拥有了加载和解析非JavaScript文件的能力。
- **Plugin**直译为"插件"。Plugin可以扩展webpack的功能, 让webpack具有更多的灵活性。在 Webpack 运行的生命周期中会广播出许多事件, Plugin 可以监听这些事件, 在合适的时机通过 Webpack 提供的 API 改变输出结果。

不同的用法:

- **Loader**在 `module.rules` 中配置，也就是说他作为模块的解析规则而存在。类型为数组，每一项都是一个 `Object`，里面描述了对于什么类型的文件 (`test`)，使用什么加载(`loader`)和使用的参数 (`options`)
- **Plugin**在 `plugins` 中单独配置。类型为数组，每一项是一个 `plugin` 的实例，参数都通过构造函数传入。

webpack的构建流程是什么？

Webpack 的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

1. 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；
2. 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 `run` 方法开始执行编译；
3. 确定入口：根据配置中的 `entry` 找出所有的入口文件；
4. 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；
5. 完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；
6. 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
7. 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

在以上过程中，Webpack 会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变 Webpack 的运行结果。

来源于[深入浅出webpack第五章](#)

拓展阅读[细说 webpack 之流程篇](#)

是否写过Loader和Plugin？描述一下编写loader或plugin的思路？

Loader像一个"翻译官"把读到的源文件内容转义成新的文件内容，并且每个Loader通过链式操作，将源文件一步步翻译成想要的样子。

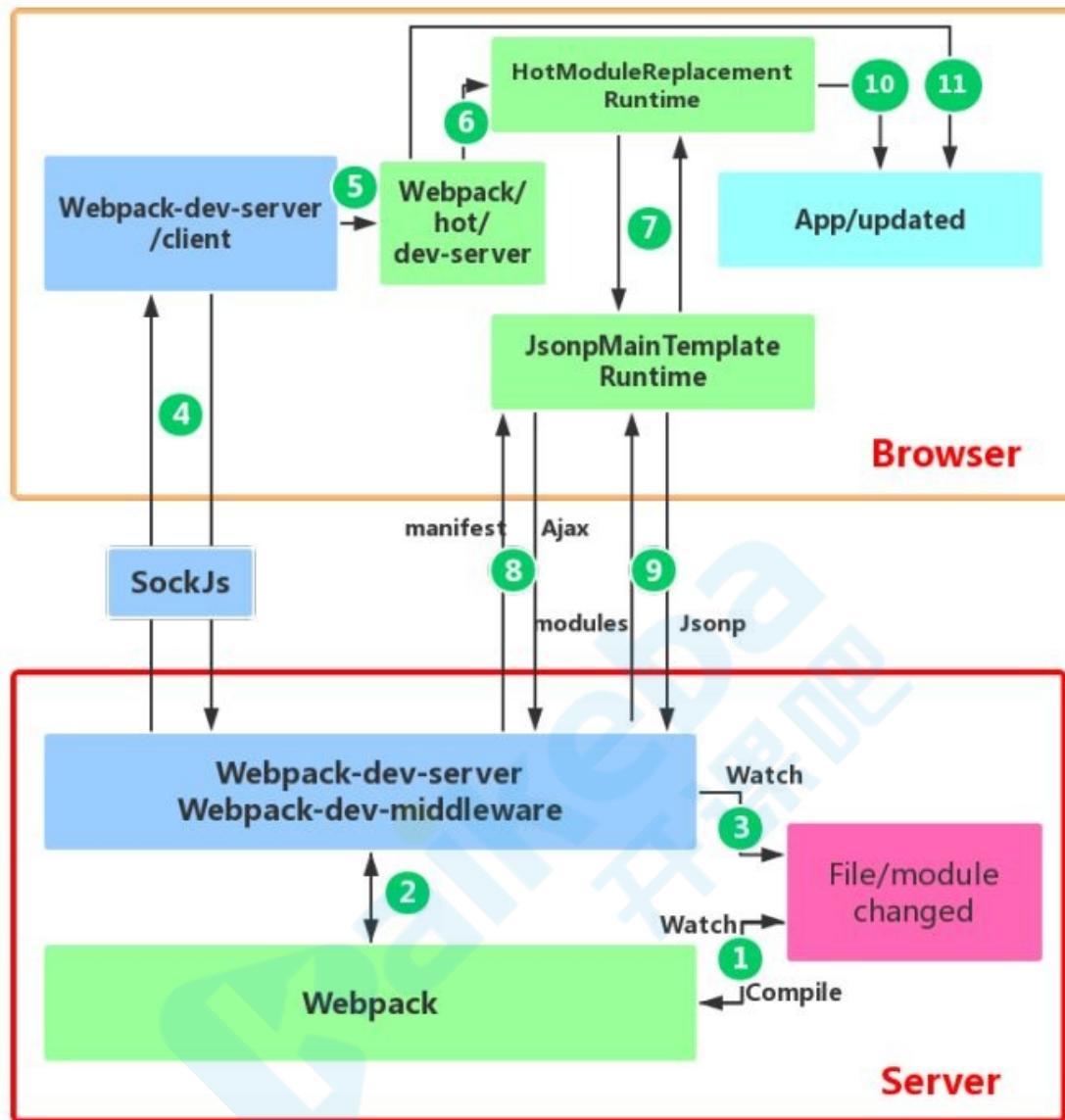
编写Loader时要遵循单一原则，每个Loader只做一种"转义"工作。每个Loader拿到的是源文件内容 (`source`)，可以通过返回值的方式将处理后的内容输出，也可以调用 `this.callback()` 方法，将内容返回给webpack。还可以通过 `this.async()` 生成一个 `callback` 函数，再用这个callback将处理后的内容输出出去。此外 webpack 还为开发者准备了开发loader的工具函数集—— `loader-utils`。

相对于Loader而言，Plugin的编写就灵活了许多。webpack在运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

webpack的热更新是如何做到的？说明其原理？

webpack的热更新又称热替换（Hot Module Replacement），缩写为HMR。这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

原理：



首先要知道server端和client端都做了处理工作

1. 第一步，在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中。
2. 第二步是 webpack-dev-server 和 webpack 之间的接口交互，而在这一步，主要是 dev-server 的中间件 webpack-dev-middleware 和 webpack 之间的交互，webpack-dev-middleware 调用 webpack 暴露的 API 对代码变化进行监控，并且告诉 webpack，将代码打包到内存中。
3. 第三步是 webpack-dev-server 对文件变化的一个监控，这一步不同于第一步，并不是监控代码变化重新打包。当我们在配置文件中配置了 `devServer.watchContentBase` 为 true 的时候，Server 会监听这些配置文件夹中静态文件的变化，变化后会通知浏览器端对应用进行 live reload。注意，这儿是浏览器刷新，和 HMR 是两个概念。
4. 第四步也是 webpack-dev-server 代码的工作，该步骤主要是通过 sockjs (webpack-dev-server 的依赖) 在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端，同时也包括第三步中 Server 监听静态文件变化的信息。浏览器端根据这些 socket 消息进行不同的操作。当然服务端传递的最主要信息还是新模块的 hash 值，后面的步骤根据这一 hash 值来进行模块热替换。
5. webpack-dev-server/client 端并不能够请求更新的代码，也不会执行热更模块操作，而把这些工作又交回给了 webpack，webpack/hot/dev-server 的工作就是根据 webpack-dev-server/client 传给它的信息以及 dev-server 的

配置决定是刷新浏览器呢还是进行模块热更新。当然如果仅仅是刷新浏览器，也就没有后面那些步骤了。

6. HotModuleReplacement.runtime 是客户端 HMR 的中枢，它接收到上一步传递给他的新模块的 hash 值，它通过 JsonpMainTemplate.runtime 向 server 端发送 Ajax 请求，服务端返回一个 json，该 json 包含了所有要更新的模块的 hash 值，获取到更新列表后，该模块再次通过 jsonp 请求，获取到最新的模块代码。这就是上图中 7、8、9 步骤。
7. 而第 10 步是决定 HMR 成功与否的关键步骤，在该步骤中，HotModulePlugin 将会对新旧模块进行对比，决定是否更新模块，在决定更新模块后，检查模块之间的依赖关系，更新模块的同时更新模块间的依赖引用。
8. 最后一步，当 HMR 失败后，回退到 live reload 操作，也就是进行浏览器刷新来获取最新打包代码。

详细原理解析来源于知乎饿了么前端[Webpack HMR 原理解析](#)

如何用webpack来优化前端性能？

用webpack优化前端性能是指优化webpack的输出结果，让打包的最终结果在浏览器运行快速高效。

- 压缩代码:删除多余的代码、注释、简化代码的写法等等方面。可以利用webpack的 `uglifyjsPlugin` 和 `ParallelUglifyPlugin` 来压缩JS文件，利用 `cssnano` (`css-loader?minimize`) 来压缩css
- 利用CDN加速: 在构建过程中，将引用的静态资源路径修改为CDN上对应的路径。可以利用webpack对于 `output` 参数和各loader的 `publicPath` 参数来修改资源路径
- Tree Shaking: 将代码中永远不会走到的片段删除掉。可以通过在启动webpack时追加参数 `--optimize-minimize` 来实现
- Code Splitting: 将代码按路由维度或者组件分块(chunk),这样做到按需加载,同时可以充分利用浏览器缓存
- 提取公共第三方库: `SplitChunksPlugin`插件来进行公共模块抽取,利用浏览器缓存可以长期缓存这些无需频繁变动的公共代码

详解可以参照[前端性能优化-加载](#)

如何提高webpack的打包速度？

- `happypack`: 利用进程并行编译loader,利用缓存来使得 rebuild 更快,遗憾的是作者表示已经不会继续开发此项目,类似的替代者是`thread-loader`
- **外部扩展(externals)**: 将不怎么需要更新的第三方库脱离webpack打包，不被打入bundle中，从而减少打包时间,比如jQuery用script标签引入
- `dll`: 采用webpack的 `DllPlugin` 和 `DllReferencePlugin` 引入dll，让一些基本不会改动的代码先打包成静态资源,避免反复编译浪费时间
- 利用缓存: `webpack.cache`、`babel-loader.cacheDirectory`、`HappyPack.cache` 都可以利用缓存提高rebuild效率
- 缩小文件搜索范围: 比如`babel-loader`插件,如果你的文件仅存在于src中,那么可以 `include: path.resolve(__dirname, 'src')` ,当然绝大多数情况下这种操作的提升有限,除非不小心build了node_modules文件

实战文章推荐[使用webpack4提升180%编译速度 Tool](#)

如何提高webpack的构建速度？

1. 多入口情况下，使用 `CommonsChunkPlugin` 来提取公共代码
2. 通过 `externals` 配置来提取常用库
3. 利用 `DllPlugin` 和 `DllReferencePlugin` 预编译资源模块 通过 `DllPlugin` 来对那些我们引用但是绝对不会修改的npm包来进行预编译，再通过 `DllReferencePlugin` 将预编译的模块加载进来。
4. 使用 `Happypack` 实现多线程加速编译
5. 使用 `webpack-uglify-parallel` 来提升 `uglifyPlugin` 的压缩速度。原理上 `webpack-uglify-parallel` 采用了多核并行压缩来提升压缩速度

6. 使用 Tree-shaking 和 Scope Hoisting 来剔除多余代码

怎么配置单页应用？怎么配置多页应用？

单页应用可以理解为webpack的标准模式，直接在 entry 中指定单页应用的入口即可，这里不再赘述

多页应用的话，可以使用webpack的 AutoWebPlugin 来完成简单自动化的构建，但是前提是项目的目录结构必须遵守他预设的规范。多页应用中要注意的是：

- 每个页面都有公共的代码，可以将这些代码抽离出来，避免重复的加载。比如，每个页面都引用了同一套css样式表
- 随着业务的不断扩展，页面可能会不断的追加，所以一定要让入口的配置足够灵活，避免每次添加新页面还需要修改构建配置

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号程序员面试官,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取



算法面试题

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

算法相关的题在前端面试中的比重越来越高,当然最有效的方法是去LeetCode上刷题,关于JavaScript版的LeetCode解题思路可以参考此项目[leetcode题解](#), [记录自己的leetcode解题之路](#)

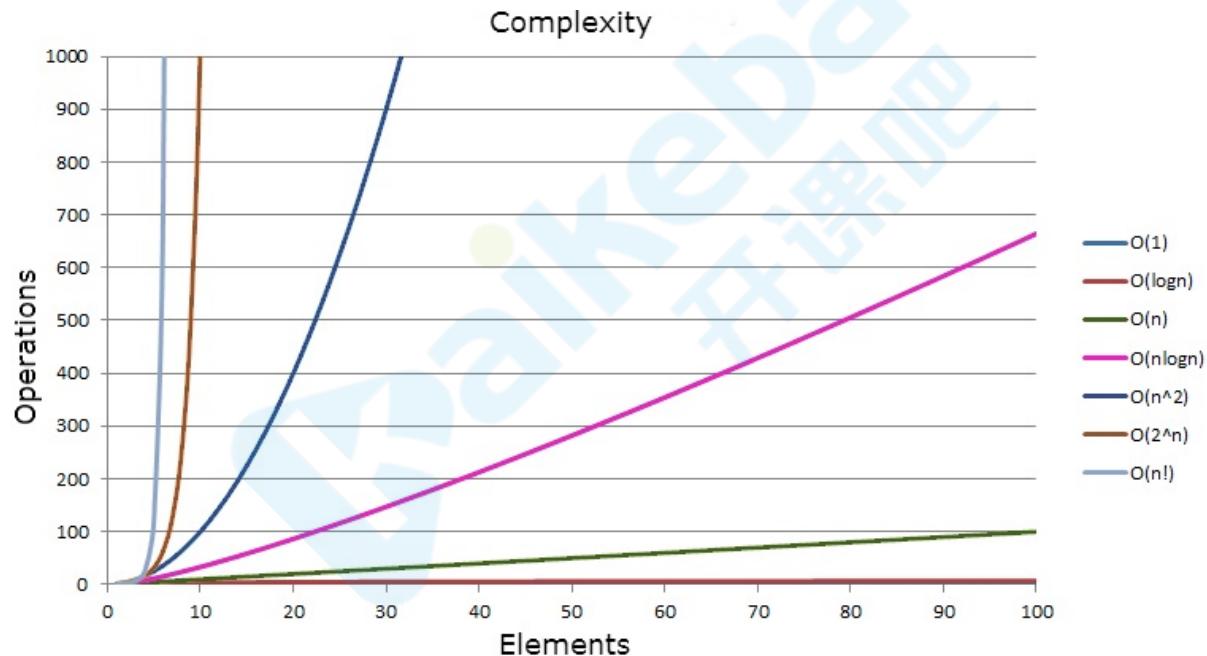
如何分析时间复杂度?

当问题规模即要处理的数据增长时, 基本操作要重复执行的次数必定也会增长, 那么我们关心地是这个执行次数以什么样的数量级增长。

我们用大O表示法表示一下常见的时间复杂度量级:

常数阶 $O(1)$ 线性阶 $O(n)$ 对数阶 $O(\log n)$ 线性对数阶 $O(n \log n)$ 平方阶 $O(n^2)$

当然还有指数阶和阶乘阶这种非常极端的复杂度量级, 我们就不讨论了。



$O(1)$

传说中的常数阶的复杂度, 这种复杂度无论数据规模n如何增长, 计算时间是不变的。

举一个简单的例子:

```
const increment = n => n++
```

不管n如何增长, 都不会影响到这个函数的计算时间, 因此这个代码的时间复杂度都是 $O(1)$ 。

$O(n)$

线性复杂度, 随着数据规模n的增长, 计算时间也会随着n线性增长。

典型的 $O(n)$ 的例子就是线性查找。

```

const linearSearch = (arr, target) => {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i
    }
  }
  return -1
}

```

线性查找的时间消耗与输入的数组数量 n 成一个线性比例，随着 n 规模的增大，时间也会线性增长。

O(logn)

对数复杂度，随着问题规模 n 的增长，计算时间也会随着 n 对数级增长。

典型的例子是二分查找法。

```

functions binarySearch(arr, target) {
  let max = arr.length - 1
  let min = 0
  while (min <= max) {
    let mid = Math.floor((max + min) / 2)
    if (target < arr[mid]) {
      max = mid - 1
    } else if (target > arr[mid]) {
      min = mid + 1
    } else {
      return mid
    }
  }
  return -1
}

```

在二分查找法的代码中，通过while循环，成2倍数的缩减搜索范围，也就是说需要经过 $\log_2 n$ 次即可跳出循环。

事实上在实际项目中， $O(\log n)$ 是一个非常好的时间复杂度，比如当 $n=100$ 的数据规模时，二分查找只需要7次，线性查找需要100次，这对于计算机而言差距不大，但是当有10亿的数据规模的时候，二分查找依然只需要30次，而线性查找需要惊人的10亿次， $O(\log n)$ 时间复杂度的算法随着数据规模的增大，它的优势就越明显。

O(nlogn)

线性对数复杂度，随着数据规模 n 的增长，计算时间也会随着 n 呈线性对数级增长。

这其中典型代表就是归并排序，我们会在对应小节详细分析它的复杂度。

```

const mergeSort = array => {
  const len = array.length
  if (len < 2) {
    return len
  }

  const mid = Math.floor(len / 2)
  const first = array.slice(0, mid)
  const last = array.slice(mid)

  return merge(mergeSort(first), mergeSort(last))

  function merge(left, right) {
    var result = []
    while (left.length && right.length) {
      if (left[0] <= right[0]) {
        result.push(left.shift())
      } else {
        result.push(right.shift())
      }
    }
    return result.concat(left.concat(right))
  }
}

```

```

        result.push(left.shift());
    } else {
        result.push(right.shift());
    }
}

while (left.length)
    result.push(left.shift());

while (right.length)
    result.push(right.shift());
return result;
}
}

```

O(n²)

平方级复杂度，典型情况是当存在双重循环的时候，即把 O(n) 的代码再嵌套循环一遍，它的时间复杂度就是 O(n²) 了，代表应用是冒泡排序算法。

```

function bubbleSort(arra){
    var temp;

    for(var i=0;i<arra.length;i++){
        for(var j=0;j<arra.length-i-1;j++){
            if(arra[j]>arra[j+1]){
                temp=arra[j];
                arra[j]=arra[j+1];
                arra[j+1]=temp;
            }
        }
    };
    return arra;
}

```

排序算法

排序算法有很多种,我们只讲最具代表性的几种算法：冒泡排序、希尔排序、归并排序、快速排序

| 排序算法 | 平均时间复杂度 | 最好情况 | 最坏情况 | 空间复杂度 | 排序方式 | 稳定性 |
|------|-----------------|-----------------|-----------------|-------------|-----------|-----|
| 冒泡排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | In-place | 稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | In-place | 不稳定 |
| 插入排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | In-place | 稳定 |
| 希尔排序 | $O(n \log n)$ | $O(n \log^2 n)$ | $O(n \log^2 n)$ | $O(1)$ | In-place | 不稳定 |
| 归并排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | Out-place | 稳定 |
| 快速排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | $O(\log n)$ | In-place | 不稳定 |
| 堆排序 | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(1)$ | In-place | 不稳定 |
| 计数排序 | $O(n + k)$ | $O(n + k)$ | $O(n + k)$ | $O(k)$ | Out-place | 稳定 |
| 桶排序 | $O(n + k)$ | $O(n + k)$ | $O(n^2)$ | $O(n + k)$ | Out-place | 稳定 |
| 基数排序 | $O(n \times k)$ | $O(n \times k)$ | $O(n \times k)$ | $O(n + k)$ | Out-place | 稳定 |

排序算法主体内容采用的是[十大经典排序算法总结（JavaScript描述）](#),更详细的内容可以移步,因为作者的内容与教科书上的内容有较大冲突,因此我们重写了快速排序部分的内容,以教科书为准,因此建议重点读一下本文的快速排序部分.

冒泡排序 (Bubble Sort)

实现思路:

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

实现:

```
function bubbleSort(arr) {
    var len = arr.length;
    for (var i = 0; i < len; i++) {
        for (var j = 0; j < len - 1 - i; j++) {
            if (arr[j] > arr[j+1]) {
                var temp = arr[j+1];
                arr[j+1] = arr[j];
                arr[j] = temp;
            }
        }
    }
    return arr;
}
```

改进1: 设置一标志性变量pos,用于记录每趟排序中最后一次进行交换的位置。由于pos位置之后的记录均已交换到位,故在进行下一趟排序时只要扫描到pos位置即可。

```
function bubbleSort2(arr) {
    console.time('改进后冒泡排序耗时');
```

```

var i = arr.length-1; //初始时,最后位置保持不变
while ( i > 0) {
    var pos= 0; //每趟开始时,无记录交换
    for (var j= 0; j< i; j++)
        if (arr[j]> arr[j+1]) {
            pos= j; //记录交换的位置
            var tmp = arr[j]; arr[j]=arr[j+1];arr[j+1]=tmp;
        }
    i= pos; //为下一趟排序作准备
}
console.timeEnd('改进后冒泡排序耗时');
return arr;
}

```

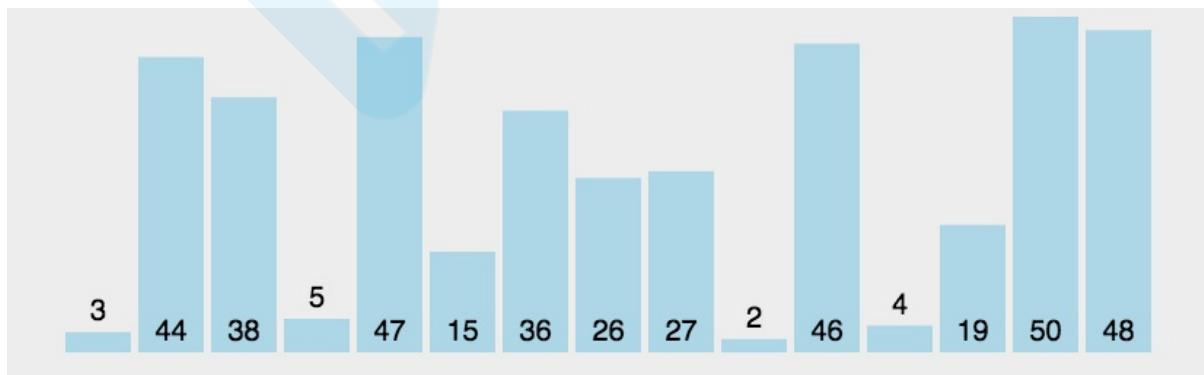
改进2: 传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值,我们考虑利用在每趟排序中进行正向和反向两遍冒泡的方法一次可以得到两个最终值(最大者和最小者),从而使排序趟数几乎减少了一半。

```

function bubbleSort3(arr3) {
    var low = 0;
    var high= arr.length-1; //设置变量的初始值
    var tmp,j;
    console.time('2.改进后冒泡排序耗时');
    while (low < high) {
        for (j= low; j< high; ++j) //正向冒泡,找到最大者
            if (arr[j]> arr[j+1]) {
                tmp = arr[j]; arr[j]=arr[j+1];arr[j+1]=tmp;
            }
        --high; //修改high值, 前移一位
        for (j=high; j>low; --j) //反向冒泡,找到最小者
            if (arr[j]<arr[j-1]) {
                tmp = arr[j]; arr[j]=arr[j-1];arr[j-1]=tmp;
            }
        ++low; //修改low值,后移一位
    }
    console.timeEnd('2.改进后冒泡排序耗时');
    return arr3;
}

```

动画:



希尔排序(Shell Sort)

1959年Shell发明; 第一个突破 $O(n^2)$ 的排序算法; 是简单插入排序的改进版; 它与插入排序的不同之处在于, 它会优先比较距离较远的元素。希尔排序又叫缩小增量排序

算法简介

希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。动态定义间隔序列的算法是《算法（第4版）》的合著者Robert Sedgewick提出的。

算法描述和实现

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

1. 选择一个增量序列 t_1, t_2, \dots, t_k , 其中 $t_i > t_j$, $t_k = 1$;
2. 按增量序列个数 k , 对序列进行 k 趟排序;
3. 每趟排序, 根据对应的增量 t_i , 将待排序列分割成若干长度为 m 的子序列, 分别对各子表进行直接插入排序。仅增量因子为 1 时, 整个序列作为一个表来处理, 表长度即为整个序列的长度。

Javascript代码实现：

```
function shellSort(arr) {
    var len = arr.length,
        temp,
        gap = 1;
    console.time('希尔排序耗时:');
    while(gap < len/5) {           //动态定义间隔序列
        gap = gap*5+1;
    }
    for (gap; gap > 0; gap = Math.floor(gap/5)) {
        for (var i = gap; i < len; i++) {
            temp = arr[i];
            for (var j = i-gap; j >= 0 && arr[j] > temp; j-=gap) {
                arr[j+gap] = arr[j];
            }
            arr[j+gap] = temp;
        }
    }
    console.timeEnd('希尔排序耗时:');
    return arr;
}
var arr=[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48];
console.log(shellSort(arr));//[2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]
```

希尔排序图示（图片来源网络）：

希尔排序过程

```

592 401 874 141 348 72 911 887 820 283
592 —————— 72
401 —————— 911
874 —————— 887
141 —————— 820
348 —————— 283

```

第一趟：增量为5

结果：72 401 874 141 283 592 911 887 820 348

72 401 874 141 283 592 911 887 820 348

72 — 874 — 283 — 911 — 820
401 — 141 — 592 — 887 — 348

第二趟：增量为2

结果：72 141 283 348 820 401 874 592 911 887

72 141 283 348 820 401 874 592 911 887

第三趟：增量为1

结果：72 141 283 348 401 592 820 874 887 911

算法分析

- 最佳情况： $T(n) = O(n \log n)$
- 最坏情况： $T(n) = O(n^2)$
- 平均情况： $T(n) = O(n \log n)$

归并排序 (Merge Sort)

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是 $O(n \log n)$ 的时间复杂度。代价是需要额外的内存空间。

算法简介

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法 (Divide and Conquer) 的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

算法描述和实现

具体算法描述如下：

1. 把长度为n的输入序列分成两个长度为n/2的子序列；
2. 对这两个子序列分别采用归并排序；
3. 将两个排序好的子序列合并成一个最终的排序序列。

Javascript代码实现：

```

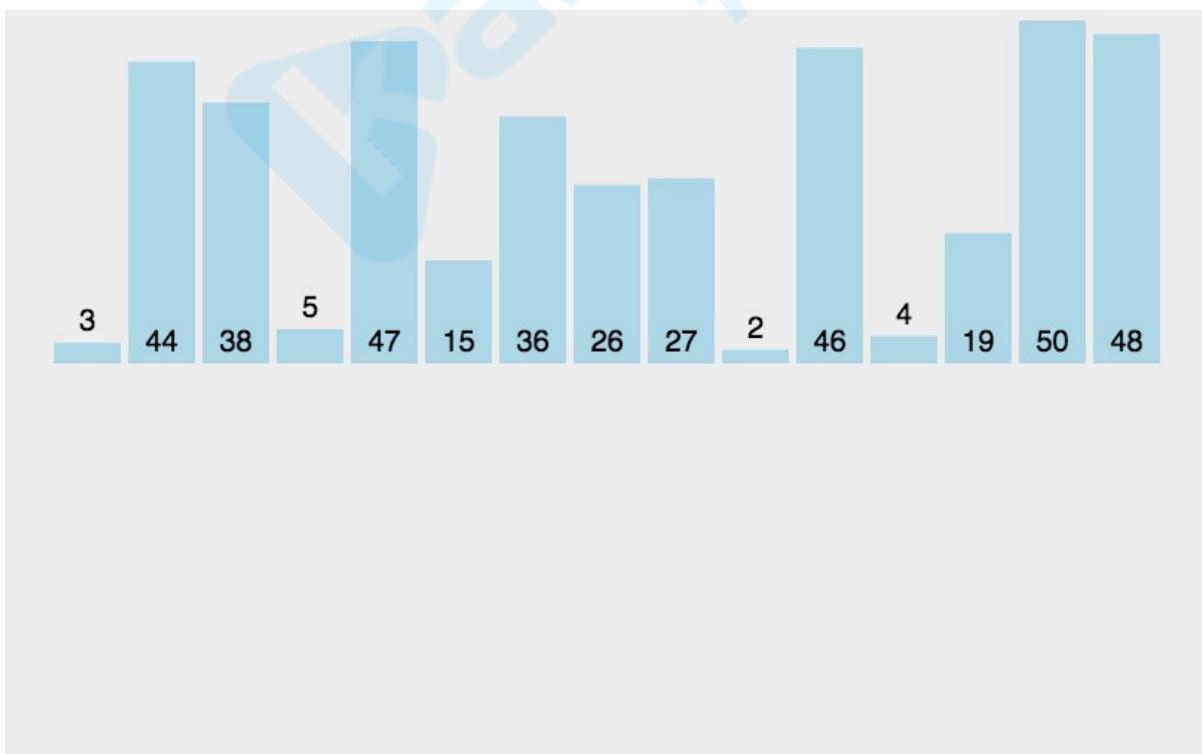
function mergeSort(arr) { //采用自上而下的递归方法
    var len = arr.length;
    if(len < 2) {
        return arr;
    }
    var middle = Math.floor(len / 2),
        left = arr.slice(0, middle),
        right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right)
{
    var result = [];
    console.time('归并排序耗时');
    while (left.length && right.length) {
        if (left[0] <= right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }
    while (left.length)
        result.push(left.shift());

    while (right.length)
        result.push(right.shift());
    console.timeEnd('归并排序耗时');
    return result;
}
var arr=[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48];
console.log(mergeSort(arr));

```

归并排序动图演示：



算法分析

- 最佳情况: $T(n) = O(n)$
- 最差情况: $T(n) = O(n \log n)$
- 平均情况: $T(n) = O(n \log n)$

快速排序 (Quick Sort)

算法简介

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

算法描述和实现

- 1.从数组中选择中间一项作为主元；
- 2.创建两个指针，左边一个指向数组的第一项，右边指向数组最后一项。移动左指针直到我们找到一个比主元大的元素，接着，移动右指针直到找到一个比主元小的元素。然后交换它们，重复这个过程，直到左指针超过了右指针。这个过程是的比主元小的值都排在了主元之前，而比主元大的值都排在了主元之后，这一步叫划分操作。
- 3.接着，算法对划分的小数组（较主元小的值组成的子数组，以及较主元大的值组成的子数组）重复之前的两个步骤，直至数组以完全排序。

```
// 快速排序
const quickSort = (function() {
    // 默认状态下的比较函数
    function compare(a, b) {
        if (a === b) {
            return 0
        }
        return a < b ? -1 : 1
    }

    function swap(array, a, b) {
        [array[a], array[b]] = [array[b], array[a]]
    }

    // 分治函数
    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

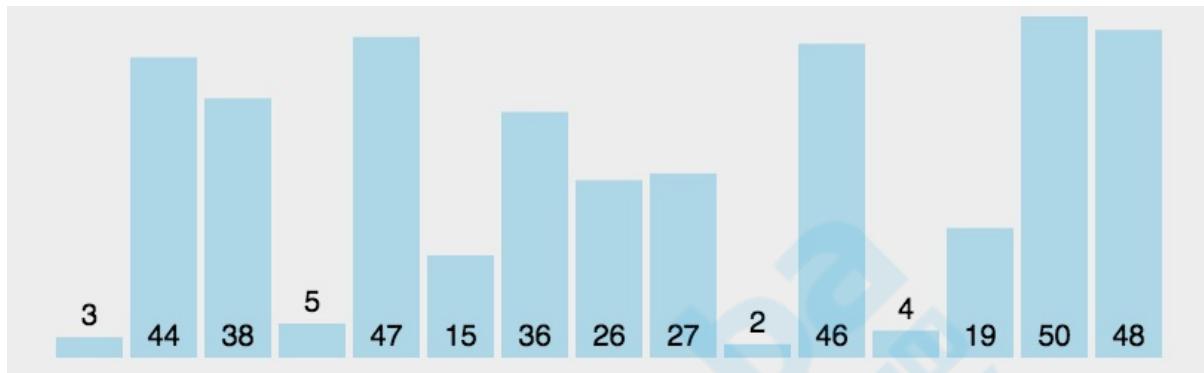
        while (i <= j) {
            while (compare(array[i], pivot) === -1) {
                i++
            }
            while (compare(array[j], pivot) === 1) {
                j--
            }
            if (i <= j) {
                swap(array, i, j)
                i++
                j--
        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

    // 快排函数
    function quick(array, left, right) {
        let index
        if (array.length > 1) {
            index = partition(array, left, right)
            quick(array, left, index - 1)
            quick(array, index + 1, right)
        }
    }
})()
```

```

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

```



算法分析

最佳情况: $T(n) = O(n \log n)$ 最差情况: $T(n) = O(n^2)$ 平均情况: $T(n) = O(n \log n)$

查找算法

二分查找法

算法简介

折半查找算法要求查找表的数据是线性结构存储，还要求查找表中的顺序是由小到大排序（由大到小排序）

算法思路及实现

- 首先设两个指针，low和height，表示最低索引和最高索引
- 然后取中间位置索引middle，判断middle处的值是否与所要查找的数相同，相同则结束查找，middle处的值比所要查找的值小就把low设为middle+1，如果middle处的值比所要查找的值大就把height设为middle-1
- 然后再新区间继续查到，直到找到或者low>height找不到所要查找的值结束查找

```

functions binarySearch(arr, target) {
    let max = arr.length - 1
    let min = 0
    while (min <= max) {
        let mid = Math.floor((max + min) / 2)
        if (target < arr[mid]) {
            max = mid - 1
        } else if (target > arr[mid]) {
            min = mid + 1
        } else {
            return mid
        }
    }
}

```

```
    }
    return -1
}
```

算法分析

最佳情况: $T(n) = O(\log n)$ 最差情况: $T(n) = O(n \log n)$ 平均情况: $T(n) = O(n \log n)$

线性查找

算法简介及实现

线性查找很简单,只需要进行简单的遍历即可.

```
const linearSearch = (arr, target) => {
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] === target) {
      return i
    }
  }
  return -1
}
```

算法分析

最佳情况: $T(n) = O(n)$ 最差情况: $T(n) = O(n)$ 平均情况: $T(n) = O(n)$

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号程序员面试官,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取





字符串类面试题

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

解析 URL Params 为对象

```
let url = 'http://www.domain.com/?user=anonymous&id=123&id=456&city=%E5%8C%97%E4%BA%AC&enabled';
parseParam(url)
/* 结果
{ user: 'anonymous',
  id: [ 123, 456 ], // 重复出现的 key 要组装成数组，能被转成数字的就转成数字类型
  city: '北京', // 中文需解码
  enabled: true, // 未指定值得 key 约定为 true
}
*/
```

```
function parseParam(url) {
  const paramsStr = /.+?(.+)$/.exec(url)[1]; // 将 ? 后面的字符串取出来
  const paramsArr = paramsStr.split('&'); // 将字符串以 & 分割后存到数组中
  let paramsObj = {};
  // 将 params 存到对象中
  paramsArr.forEach(param => {
    if (/=/ .test(param)) { // 处理有 value 的参数
      let [key, val] = param.split('!'); // 分割 key 和 value
      val = decodeURIComponent(val); // 解码
      val = /\^d+$/.test(val) ? parseFloat(val) : val; // 判断是否转为数字

      if (paramsObj.hasOwnProperty(key)) { // 如果对象有 key，则添加一个值
        paramsObj[key] = [].concat(paramsObj[key], val);
      } else { // 如果对象没有这个 key，创建 key 并设置值
        paramsObj[key] = val;
      }
    } else { // 处理没有 value 的参数
      paramsObj[param] = true;
    }
  })

  return paramsObj;
}
```

模板引擎实现

```
let template = '我是{{name}}，年龄{{age}}，性别{{sex}}';
let data = {
  name: '姓名',
  age: 18
}
render(template, data); // 我是姓名，年龄18，性别undefined
```

```
function render(template, data) {
  const reg = /\{\{(\w+)\}\}/; // 模板字符串正则
  if (reg.test(template)) { // 判断模板里是否有模板字符串
    const name = reg.exec(template)[1]; // 查找当前模板里第一个模板字符串的字段
    template = template.replace(reg, data[name]); // 将第一个模板字符串渲染
    return render(template, data); // 递归的渲染并返回渲染后的结构
  }
}
```

```

    return template; // 如果模板没有模板字符串直接返回
}

```

转化为驼峰命名

```

var s1 = "get-element-by-id"
// 转化为 getElementById

```

```

var f = function(s) {
  return s.replace(/-\w/g, function(x) {
    return x.slice(1).toUpperCase();
  })
}

```

查找字符串中出现最多的字符和个数

例: abbcccdddd -> 字符最多的是d, 出现了5次

```

let str = "abcababcbbccccc";
let num = 0;
let char = '';

// 使其按照一定的次序排列
str = str.split('').sort().join('');
// "aaabbccccc"

// 定义正则表达式
let re = /(\w)\1+/g;
str.replace(re, ($0,$1) => {
  if(num < $0.length){
    num = $0.length;
    char = $1;
  }
});
console.log(`字符最多的是${char}, 出现了${num}次`);

```

字符串查找

请使用最基本的遍历来实现判断字符串 a 是否被包含在字符串 b 中, 并返回第一次出现的位置 (找不到返回 -1)。

```

a='34';b='1234567'; // 返回 2
a='35';b='1234567'; // 返回 -1
a='355';b='12354355'; // 返回 5
isContain(a,b);

```

```

function isContain(a, b) {
  for (let i in b) {
    if (a[0] === b[i]) {
      let tmp = true;
      for (let j in a) {
        if (a[j] !== b[~~i + ~~j]) {
          tmp = false;
        }
      }
    }
  }
}

```

```

        if (tmp) {
            return i;
        }
    }
    return -1;
}

```

实现千位分隔符

```

// 保留三位小数
parseToMoney(1234.56); // return '1,234.56'
parseToMoney(123456789); // return '123,456,789'
parseToMoney(1087654.321); // return '1,087,654.321'

```

```

function parseToMoney(num) {
    num = parseFloat(num.toFixed(3));
    let [integer, decimal] = String.prototype.split.call(num, '.');
    integer = integer.replace(/\d(?=(\d{3})+$)/g, '$&');
    return integer + '.' + (decimal ? decimal : '');
}

```

正则表达式(运用了正则的前向声明和反前向声明):

```

function parseToMoney(str){
    // 仅仅对位置进行匹配
    let re = /(?=(?!\b)(\d{3})+$)/g;
    return str.replace(re, ',');
}

```

判断是否是电话号码

```

function isPhone.tel) {
    var regex = /^1[34578]\d{9}$/;
    return regex.test.tel);
}

```

验证是否是邮箱

```

function isEmail(email) {
    var regex = /^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+\.[a-zA-Z0-9_-]+$/;
    return regex.test(email);
}

```

验证是否是身份证

```

function isCardNo(number) {
    var regex = /(^\d{15}$)|(^\d{18}$)|(^\d{17}(\d|X|x)$)/;
    return regex.test(number);
}

```

参考:

- 前端面试遇到的算法题

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



JavaScript笔试部分

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

实现防抖函数（debounce）

防抖函数原理：在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。

那么与节流函数的区别直接看这个动画实现即可。



手写简化版：

```
// 防抖函数
const debounce = (fn, delay) => {
  let timer = null;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
};
```

适用场景：

- 按钮提交场景：防止多次提交按钮，只执行最后提交的一次
- 服务端验证场景：表单验证需要服务端配合，只执行一段连续的输入事件的最后一次，还有搜索联想词功能类似
生存环境请用lodash.debounce

实现节流函数（throttle）

防抖函数原理：规定在一个单位时间内，只能触发一次函数。如果这个单位时间内触发多次函数，只有一次生效。

// 手写简化版

```
// 节流函数
const throttle = (fn, delay = 500) => {
  let flag = true;
  return (...args) => {
    if (!flag) return;
    flag = false;
    setTimeout(() => {
      fn.apply(this, args);
      flag = true;
    }, delay);
  };
};
```

适用场景：

- 拖拽场景：固定时间内只执行一次，防止超高频次触发位置变动
- 缩放场景：监控浏览器resize
- 动画场景：避免短时间内多次触发动画引起性能问题

深克隆（deepclone）

简单版：

```
const newObj = JSON.parse(JSON.stringify(oldObj));
```

局限性：

- 他无法实现对函数、RegExp等特殊对象的克隆
- 会抛弃对象的constructor,所有的构造函数会指向Object
- 对象有循环引用,会报错

面试版：

```
/**
 * deep clone
 * @param {[type]} parent object 需要进行克隆的对象
 * @return {[type]} 深克隆后的对象
 */
const clone = parent => {
  // 判断类型
  const isType = (obj, type) => {
    if (typeof obj !== "object") return false;
    const typeString = Object.prototype.toString.call(obj);
    let flag;
    switch (type) {
      case "Array":
```

```

        flag = typeString === "[object Array]";
        break;
    case "Date":
        flag = typeString === "[object Date]";
        break;
    case "RegExp":
        flag = typeString === "[object RegExp]";
        break;
    default:
        flag = false;
    }
    return flag;
};

// 处理正则
const getRegExp = re => {
    var flags = "";
    if (re.global) flags += "g";
    if (re.ignoreCase) flags += "i";
    if (re.multiline) flags += "m";
    return flags;
};
// 维护两个储存循环引用的数组
const parents = [];
const children = [];

const _clone = parent => {
    if (parent === null) return null;
    if (typeof parent !== "object") return parent;

    let child, proto;

    if (isType(parent, "Array")) {
        // 对数组做特殊处理
        child = [];
    } else if (isType(parent, "RegExp")) {
        // 对正则对象做特殊处理
        child = new RegExp(parent.source, getRegExp(parent));
        if (parent.lastIndex) child.lastIndex = parent.lastIndex;
    } else if (isType(parent, "Date")) {
        // 对Date对象做特殊处理
        child = new Date(parent.getTime());
    } else {
        // 处理对象原型
        proto = Object.getPrototypeOf(parent);
        // 利用Object.create切断原型链
        child = Object.create(proto);
    }

    // 处理循环引用
    const index = parents.indexOf(parent);

    if (index != -1) {
        // 如果父数组存在本对象,说明之前已经被引用过,直接返回此对象
        return children[index];
    }
    parents.push(parent);
    children.push(child);

    for (let i in parent) {
        // 递归
        child[i] = _clone(parent[i]);
    }

    return child;
};
return _clone(parent);
};

```

局限性:

1. 一些特殊情况没有处理: 例如Buffer对象、Promise、Set、Map
2. 另外对于确保没有循环引用的对象, 我们可以省去对循环引用的特殊处理, 因为这很消耗时间

原理详解[实现深克隆](#)

实现Event(event bus)

event bus既是node中各个模块的基石, 又是前端组件通信的依赖手段之一, 同时涉及了订阅-发布设计模式, 是非常重要的基础。

简单版:

```
class EventEmeitter {
  constructor() {
    this._events = this._events || new Map(); // 储存事件/回调键值对
    this._maxListeners = this._maxListeners || 10; // 设立监听上限
  }

  // 触发名为type的事件
  EventEmeitter.prototype.emit = function(type, ...args) {
    let handler;
    // 从储存事件键值对的this._events中获取对应事件回调函数
    handler = this._events.get(type);
    if (args.length > 0) {
      handler.apply(this, args);
    } else {
      handler.call(this);
    }
    return true;
  };

  // 监听名为type的事件
  EventEmeitter.prototype.addListener = function(type, fn) {
    // 将type事件以及对应的fn函数放入this._events中储存
    if (!this._events.get(type)) {
      this._events.set(type, fn);
    }
  };
}
```

面试版:

```
class EventEmeitter {
  constructor() {
    this._events = this._events || new Map(); // 储存事件/回调键值对
    this._maxListeners = this._maxListeners || 10; // 设立监听上限
  }

  // 触发名为type的事件
  EventEmeitter.prototype.emit = function(type, ...args) {
    let handler;
    // 从储存事件键值对的this._events中获取对应事件回调函数
    handler = this._events.get(type);
    if (args.length > 0) {
      handler.apply(this, args);
    } else {
      handler.call(this);
    }
    return true;
  };

  // 监听名为type的事件
  EventEmeitter.prototype.addListener = function(type, fn) {
    // 将type事件以及对应的fn函数放入this._events中储存
    if (!this._events.get(type)) {
      this._events.set(type, fn);
    }
  };
}
```

```

};

// 监听名为type的事件
EventEmitter.prototype.addListener = function(type, fn) {
    // 将type事件以及对应的fn函数放入this._events中储存
    if (!this._events.get(type)) {
        this._events.set(type, fn);
    }
};

// 触发名为type的事件
EventEmitter.prototype.emit = function(type, ...args) {
    let handler;
    handler = this._events.get(type);
    if (Array.isArray(handler)) {
        // 如果是一个数组说明有多个监听者,需要依次此触发里面的函数
        for (let i = 0; i < handler.length; i++) {
            if (args.length > 0) {
                handler[i].apply(this, args);
            } else {
                handler[i].call(this);
            }
        }
    } else {
        // 单个函数的情况我们直接触发即可
        if (args.length > 0) {
            handler.apply(this, args);
        } else {
            handler.call(this);
        }
    }
}

return true;
};

// 监听名为type的事件
EventEmitter.prototype.addListener = function(type, fn) {
    const handler = this._events.get(type); // 获取对应事件名称的函数清单
    if (!handler) {
        this._events.set(type, fn);
    } else if (handler && typeof handler === "function") {
        // 如果handler是函数说明只有一个监听者
        this._events.set(type, [handler, fn]); // 多个监听者我们需要用数组储存
    } else {
        handler.push(fn); // 已经有多个监听者,那么直接往数组里push函数即可
    }
};

EventEmitter.prototype.removeListener = function(type, fn) {
    const handler = this._events.get(type); // 获取对应事件名称的函数清单

    // 如果是函数,说明只被监听了一次
    if (handler && typeof handler === "function") {
        this._events.delete(type, fn);
    } else {
        let position;
        // 如果handler是数组,说明被监听多次要找到对应的函数
        for (let i = 0; i < handler.length; i++) {
            if (handler[i] === fn) {
                position = i;
            } else {
                position = -1;
            }
        }
        // 如果找到匹配的函数,从数组中清除
        if (position !== -1) {
            // 找到数组对应的位置,直接清除此回调
            handler.splice(position, 1);
            // 如果清除后只有一个函数,那么取消数组,以函数形式保存
        }
    }
};

```

```

        if (handler.length === 1) {
            this._events.set(type, handler[0]);
        }
    } else {
        return this;
    }
};

}

```

实现具体过程和思路见[实现event](#)

实现instanceOf

```

// 模拟 instanceof
function instance_of(L, R) {
    // L 表示左表达式, R 表示右表达式
    var O = R.prototype; // 取 R 的显示原型
    L = L.__proto__; // 取 L 的隐式原型
    while (true) {
        if (L === null) return false;
        if (O === L)
            // 这里重点: 当 O 严格等于 L 时, 返回 true
            return true;
        L = L.__proto__;
    }
}

```

模拟new

new操作符做了这些事:

- 它创建了一个全新的对象
- 它会被执行[[Prototype]] (也就是proto) 链接
- 它使this指向新创建的对象
- 通过new创建的每个对象将最终被[[Prototype]]链接到这个函数的prototype对象上
- 如果函数没有返回对象类型Object(包含Function, Array, Date, RegExp, Error), 那么new表达式中的函数调用将返回该对象引用

```

// objectFactory(name, 'cxk', '18')
function objectFactory() {
    const obj = new Object();
    const Constructor = [].shift.call(arguments);

    obj.__proto__ = Constructor.prototype;

    const ret = Constructor.apply(obj, arguments);

    return typeof ret === "object" ? ret : obj;
}

```

实现一个call

call做了什么:

- 将函数设为对象的属性
- 执行&删除这个函数

- 指定this到函数并传入给定参数执行函数
- 如果不传入参数， 默认指向为 window

```
// 模拟 call bar.mycall(null);
//实现一个call方法:
Function.prototype.myCall = function(context) {
  //此处没有考虑context非object情况
  context.fn = this;
  let args = [];
  for (let i = 1, len = arguments.length; i < len; i++) {
    args.push(arguments[i]);
  }
  context.fn(...args);
  let result = context.fn(...args);
  delete context.fn;
  return result;
};
```

具体实现参考[JavaScript深入之call和apply的模拟实现](#)

实现apply方法

apply原理与call很相似， 不多赘述

```
// 模拟 apply
Function.prototype.myapply = function(context, arr) {
  var context = Object(context) || window;
  context.fn = this;

  var result;
  if (!arr) {
    result = context.fn();
  } else {
    var args = [];
    for (var i = 0, len = arr.length; i < len; i++) {
      args.push("arr[" + i + "]");
    }
    result = eval("context.fn(" + args + ")");
  }

  delete context.fn;
  return result;
};
```

实现bind

实现bind要做什么

- 返回一个函数， 绑定this， 传递预置参数
- bind返回的函数可以作为构造函数使用。故作为构造函数时应使得this失效， 但是传入的参数依然有效

```
// mdn的实现
if (!Function.prototype.bind) {
  Function.prototype.bind = function(oThis) {
    if (typeof this !== 'function') {
      // closest thing possible to the ECMAScript 5
      // internal IsCallable function
      throw new TypeError('Function.prototype.bind - what is trying to be bound is not callable');
    }
  }
}
```

```

var aArgs = Array.prototype.slice.call(arguments, 1),
fToBind = this,
fNOP = function() {},
fBound = function() {
    // this instanceof fBound === true时,说明返回的fBound被当做new的构造函数调用
    return fToBind.apply(this instanceof fBound
        ? this
        : oThis,
        // 获取调用时(fBound)的参.bind 返回的函数入参往往是这么传递的
        aArgs.concat(Array.prototype.slice.call(arguments)));
};

// 维护原型关系
if (this.prototype) {
    // Function.prototype doesn't have a prototype property
    fNOP.prototype = this.prototype;
}
// 下行的代码使fBound.prototype是fNOP的实例,因此
// 返回的fBound若作为new的构造函数,new生成的新对象作为this传入fBound,新对象的__proto__就是fNOP的实例
fBound.prototype = new fNOP();

return fBound;
}
}

```

详解请移步[JavaScript深入之bind的模拟实现 #12](#)

模拟Object.create

Object.create()方法创建一个新对象，使用现有的对象来提供新创建的对象的proto。

```

// 模拟 Object.create

function create(proto) {
    function F() {}
    F.prototype = proto;

    return new F();
}

```

实现类的继承

类的继承在几年前是重点内容，有n种继承方式各有优劣，es6普及后越来越不重要，那么多种写法有点『回字有四样写法』的意思，如果还想深入理解的去看红宝书即可，我们目前只实现一种最理想的继承方式。

```

function Parent(name) {
    this.parent = name
}
Parent.prototype.say = function() {
    console.log(` ${this.parent}: 你打篮球的样子像kunkun`)
}
function Child(name, parent) {
    // 将父类的构造函数绑定在子类上
    Parent.call(this, parent)
    this.child = name
}

/**
1. 这一步不用Child.prototype = Parent.prototype的原因是怕共享内存，修改父类原型对象就会影响子类
2. 不用Child.prototype = new Parent()的原因是会调用2次父类的构造方法（另一次是call），会存在一份多余的父类实例属性
3. Object.create是创建了父类原型的副本，与父类原型完全隔离
*/

```

```

/*
Child.prototype = Object.create(Parent.prototype);
Child.prototype.say = function() {
    console.log(`"${this.parent}"好，我是练习时长两年半的${this.child}`);
}

// 注意记得把子类的构造指向子类本身
Child.prototype.constructor = Child;

var parent = new Parent('father');
parent.say() // father: 你打篮球的样子像kunkun

var child = new Child('cxk', 'father');
child.say() // father好，我是练习时长两年半的cxk

```

实现JSON.parse

```

var json = '{"name":"cxk", "age":25}';
var obj = eval("(" + json + ")");

```

此方法属于黑魔法，极易容易被xss攻击，还有一种 `new Function` 大同小异。

简单的教程看这个[半小时实现一个 JSON 解析器](#)

实现Promise

我很早之前实现过一版，而且注释很多，但是居然找不到了,这是在网络上找了一版带注释的，目测没有大问题，具体过程可以看这篇[史上最易读懂的 Promise/A+ 完全实现](#)

```

var PromisePolyfill = (function () {
    // 和reject不同的是resolve需要尝试展开thenenable对象
    function tryToResolve (value) {
        if (this === value) {
            // 主要是防止下面这种情况
            // let y = new Promise(res => setTimeout(res(y)))
            throw TypeError('Chaining cycle detected for promise!')
        }

        // 根据规范2.32以及2.33 对对象或者函数尝试展开
        // 保证S6之前的 polyfill 也能和ES6的原生promise混用
        if (value !== null &&
            (typeof value === 'object' || typeof value === 'function')) {
            try {
                // 这里记录这次then的值同时要被try包裹
                // 主要原因是 then 可能是一个getter，也就是说
                // 1. value.then可能报错
                // 2. value.then可能产生副作用(例如多次执行可能结果不同)
                var then = value.then

                // 另一方面，由于无法保证 then 确实会像预期的那样只调用一个onFullfilled / onRejected
                // 所以增加了一个flag来防止resolveOrReject被多次调用
                var thenAlreadyCalledOrThrow = false
                if (typeof then === 'function') {
                    // 是thenenable 那么尝试展开
                    // 并且在该thenenable状态改变之前this对象的状态不变
                    then.bind(value)(
                        // onFullfilled
                        function (value2) {
                            if (thenAlreadyCalledOrThrow) return
                            thenAlreadyCalledOrThrow = true
                            tryToResolve.bind(this, value2)()
                        }
                    )
                }
            } catch (e) {
                if (thenAlreadyCalledOrThrow) return
                thenAlreadyCalledOrThrow = true
                reject(e)
            }
        }
    }

    // 兼容一些旧的API
    if (!Object.create) {
        Object.create = function (proto, props) {
            function F() {}
            F.prototype = proto
            return new F()
        }
    }
    if (!Object.keys) {
        Object.keys = function (obj) {
            var keys = []
            for (var key in obj) {
                if (Object.prototype.hasOwnProperty.call(obj, key)) {
                    keys.push(key)
                }
            }
            return keys
        }
    }
}

if (typeof module !== 'undefined' && module.exports) {
    module.exports = PromisePolyfill
}

```

```

        }.bind(this),

        // onRejected
        function (reason2) {
            if (thenAlreadyCalledOrThrow) return
            thenAlreadyCalledOrThrow = true
            resolveOrReject.bind(this, 'rejected', reason2)()
            }.bind(this)
        )
    } else {
        // 拥有then 但是then不是一个函数 所以也不是thenable
        resolveOrReject.bind(this, 'resolved', value)()
    }
} catch (e) {
    if (thenAlreadyCalledOrThrow) return
    thenAlreadyCalledOrThrow = true
    resolveOrReject.bind(this, 'rejected', e)()
}
} else {
    // 基本类型 直接返回
    resolveOrReject.bind(this, 'resolved', value)()
}
}

function resolveOrReject (status, data) {
    if (this.status !== 'pending') return
    this.status = status
    this.data = data
    if (status === 'resolved') {
        for (var i = 0; i < this.resolveList.length; ++i) {
            this.resolveList[i]()
        }
    } else {
        for (i = 0; i < this.rejectList.length; ++i) {
            this.rejectList[i]()
        }
    }
}

function Promise (executor) {
    if (!(this instanceof Promise)) {
        throw Error('Promise can not be called without new !')
    }

    if (typeof executor !== 'function') {
        // 非标准 但与Chrome谷歌保持一致
        throw TypeError('Promise resolver ' + executor + ' is not a function')
    }

    this.status = 'pending'
    this.resolveList = []
    this.rejectList = []

    try {
        executor(tryToResolve.bind(this), resolveOrReject.bind(this, 'rejected'))
    } catch (e) {
        resolveOrReject.bind(this, 'rejected', e)()
    }
}

Promise.prototype.then = function (onFullfilled, onRejected) {
    // 返回值穿透以及错误穿透，注意错误穿透用的是throw而不是return，否则的话
    // 这个then返回的promise状态将变成resolved即接下来的then中的onFullfilled
    // 会被调用，然而我们想要调用的是onRejected
    if (typeof onFullfilled !== 'function') {
        onFullfilled = function (data) {
            return data
        }
    }
}

```

```

if (typeof onRejected !== 'function') {
  onRejected = function (reason) {
    throw reason
  }
}

var executor = function (resolve, reject) {
  setTimeout(function () {
    try {
      // 拿到对应的handle函数处理this.data
      // 并以此为依据解析这个新的Promise
      var value = this.status === 'resolved'
        ? onFullfilled(this.data)
        : onRejected(this.data)
      resolve(value)
    } catch (e) {
      reject(e)
    }
  }.bind(this))
}

// then 接受两个函数返回一个新的Promise
// then 自身的执行永远异步与onFullfilled/onRejected的执行
if (this.status !== 'pending') {
  return new Promise(executor.bind(this))
} else {
  // pending
  return new Promise(function (resolve, reject) {
    this.resolveList.push(executor.bind(this, resolve, reject))
    this.rejectList.push(executor.bind(this, resolve, reject))
  }.bind(this))
}
}

// for promise A+ test
Promise.deferred = Promise.defer = function () {
  var dfd = {}
  dfd.promise = new Promise(function (resolve, reject) {
    dfd.resolve = resolve
    dfd.reject = reject
  })
  return dfd
}

// for promise A+ test
if (typeof module !== 'undefined') {
  module.exports = Promise
}

return Promise
})()

PromisePolyfill.all = function (promises) {
  return new Promise((resolve, reject) => {
    const result = []
    let cnt = 0
    for (let i = 0; i < promises.length; ++i) {
      promises[i].then(value => {
        cnt++
        result[i] = value
        if (cnt === promises.length) resolve(result)
      }, reject)
    }
  })
}

PromisePolyfill.race = function (promises) {
  return new Promise((resolve, reject) => {
    for (let i = 0; i < promises.length; ++i) {

```

```
        promises[i].then(resolve, reject)
    }
}
}
```

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取



如何通过HR面

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

HR通常是程序员面试的最后一面,讲道理刷人的几率不大,但是依然有人倒在了这最后一关上,我们会从HR的角度出发来分析如何应对HR面.

HR面的目的

HR面往往是把控人才质量的最后一关,与前面的技术面不同,HR面往往侧重员工风险的评估与基本的员工素质.

录用风险评估,这部分是评估候选人是否具备稳定性,是否会带来额外的管理风险,是否能马上胜任工作,比如频繁的跳槽会带了稳定性的风险,HR会慎重考虑这一点,比如在面试中候选人体现出了「杠精」潜质,HR会担心候选人在工作中会难以与他人协作或者不服从管理,带来管理风险,再比如,虽然国家明确规定在招聘中不得有性别、年龄等歧视,但是一个大龄已婚妇女会有近期产子的可能性, 可能会有长期的产假, HR也会做出评估。

员工素质评估,这部分评估候选人是否具备职场的基本素质,是否有基本的沟通能力,是否有团队精神和合作意识等等,比如一个表现极为内向的候选人,HR可能会对其沟通能力产生怀疑.

所以在与HR交流中要尽量保持踏实稳重、积极乐观的态度, 切忌暴露出夸夸其谈、负能量、浮躁等性格缺陷。

HR面的常见问题

你对未来3-5年的职业规划

目的: 这个问题就是考察候选人对未来的规划能力,主要想通过候选人的规划来嗅出候选人对工作的态度、稳定性和对技术的追求.

分析: 一定要在你的回答中体现对技术的追求、对团队的贡献、对工作的态度, 不要谈一些假大空的东西, 或者薪资、职位这些太过于功利的东西,而且最好体现出你的稳定性,如果是校招生或者工作没几年的新人最好不要涉及创业这种话题,一方面职场新人计划没几年就创业,这种很不切实际,说明候选人没法按实际出发,另一方面说明候选人的稳定性不够.

[还真有候选人因为谈创业被HR刷的](#)

建议分三部分谈:

1. 首先表示考虑过这个问题(有规划),如何谈一谈自己的现状(结合实际).
2. 接着从工作本身出发,谈谈自己会如何出色完成本职工作,如何对团队贡献、如何帮助带领团队其他成员创造更多的价值、如何帮助团队扩大影响力.
3. 最后从学习出发,谈谈自己会如何精进领域知识、如何通过提升自己专业能力,如何反哺团队.

至于想成为技术leader还是技术专家,就看自己的喜好了.

如何看待加班(996)?

目的: 考察候选人的抗压能力和责任心

分析: 这个问题几乎是必问的,虽然996ICU事件闹得沸沸扬扬,但是官方的态度很暧昧,只口头批评从没有实际行动,基本上是默许企业违反劳动法的,除了个别企在国内基本没可能找到不加班的公司,所以在这个面试题中尽量体现出自己愿意牺牲自我时间来帮助团队和企业的意愿就行了,而且要强调自己的责任心,如果真的是碰到无意义加班,好好学习怎么用vscode刷LeetCode划水是正道.

建议:

1. 把加班分为紧急加班和长期加班
2. 对于紧急加班,表示这是每个公司都会遇到的情况,自己愿意牺牲时间帮助公司和团队
3. 对于长期加班,如果是自己长期加班那么会磨练自己的技能,提高自己的效率,如果是团长期加班,自己会帮助团队找到问题,利用自动化工具或者更高效的协作流程来提高整个团队的效率,帮助大家摆脱加班

当然了,就算你提高了团队效率,还是会被安排更多的任务,加班很多时候仅仅是目的,但是你不能说出来啊,尤其是一些候选人很强硬得表示长期加班不接受,其实可以回答的更委婉,除非你是真的对这个公司没兴趣,如果以进入这个公司为第一目的,还是做个高姿态比较好.

面对大量超过自己承受能力且时间有限的工作时你会怎么办?

目的: 考察候选人时间管理和处理大量任务的能力,当然也会涉及一定的沟通能力

分析: 程序员的工作内容可能大部分时间并不在写代码上,而是要处理各种会议、需求和沟通,通常都属于工作超负荷的状态,面对上面这种问题不建议以加班的方式来解决,因为主要考察的是你的时间管理能力和沟通能力,这些要素要在回答中体现出来

建议:

1. 将大量任务分解为紧急且重要、重要但不紧急、紧急但不重要、不重要且不紧急,依次完成上述任务,在这里体现出时间管理的能力
2. 与自己的领导沟通将不重要的任务放缓执行或者砍掉,或者派给组内的新人处理,在这里体现出沟通能力

你之前在上海为什么现在来北京发展?

目的: 考察候选人的稳定性和职业选择

分析: 这个问题一般是上份工作在异地的情况下大概率出现,HR主要担心候选人异地换工作可能会不稳定,有短期内离职风险,这个时候不建议说"北京互联网公司多,机会多"这种话(合着觉得北京好跳槽?),回答最好要体现出自己的稳定性,比如"女朋友在北京,长期异地,准备来北京一起发展" "家在北京,回北京发展" 等等,潜台词就是以后会在北京发展,不会在多地之间来回摇摆.

为什么从上一家公司离职?

目的: 考察离职原因,候选人离职风险评估

分析: 这个问题经常会在跳槽的时候问到,这个时候切忌吐槽上一家公司或者自己的上一任老板,尽量从职业发展的角度来回答,凸显自己的稳定性和渴望学习上升的决心,至于一些敏感话题,比如加班太多、薪资太低这种问题也是可以谈的,毕竟你跳槽的诉求就是解决上家公司碰到的问题,但是不能触碰刚才提到的底线问题,切忌吐槽向.

建议:

1. 因为工资低、离家远、加班多、技术含量低等等原因离职
2. 因为离家远花费在路途上的时间过多,不如用来充电,因为加班多导致没有时间充电,无法提高等等

除了不要有负能量和吐槽向,这个部分可以坦诚得说出来

你还有其他公司的Offer吗?

目的: 评估候选人是否有短时间内入职其他公司的可能性

分析: 很多时候并不是候选人完美符合一个岗位的要求,HR当然想要一个技术更好、要钱更少、技术更匹配的候选人,但是候选人一般都会有这样或者那样的小问题。

比如，你的表现是可以胜任目前的岗位的，但是这个岗位不是很紧急，HR可能把你当做备胎，来找一个性价比更高的候选人。

比如，你的表现很好，履历优秀，HR不知道能不能100%拿下你。

所以如果你很希望加入这个公司，最好要做到「欲擒故纵」，既要体现自身的市场竞争力，又要给到HR一定的压力。

所以，即使你已经拿了全北京城互联网公司的offer了，也不要说自己offer多如牛毛，一副满不在乎的样子，这样会给HR造成他入职可能性不大的错觉，因为他的选择太多了。

当然，也不要跪在地上舔：“加入公司是我的梦想，我只等这一个offer”，放心吧，一定被hr放到备胎人才库中。

建议：

1. 表明自己有三四个已经确认过的offer了（没有offer也要吹，但是不要透露具体公司）
2. 但是第一意向还是本公司，如果薪资差距不大，会优先考虑本公司
3. 再透露出，有一两个offer催得比较急，希望这边快点出结果

如何与HR谈薪资？

HR与你谈论薪资经常有如下套路：

- HR：您期望的薪资是多少？
- 你：25K。

OK，你已经被HR成功套路。这个时候你的最高价就是25K了，然后HR会顺着这个价往下砍，所以你最终的薪资一般都会低于25K。等你接到offer，你的心里肯定充满了各种“悔恨”：其实当时报价26、27甚至28、29也是可以的。

正确的回答可以这样，并且还能够反套路一下HR：

- HR：您期望的薪资是多少？
- 你：就我的面试表现，贵公司最高可以给多少薪水？

哈哈，如果经验不够老道的HR可能就真会说出一个报价（如25K）来，然后，你就可以很开心地顺着这个价慢慢地往上谈了。所以这种情况下，你最终的薪资肯定是大于25K的。当然，经验老道的HR会给你一句很官方的套话：

- HR：您期望的薪资是多少？
- 你：就我的面试表现，贵公司最高可以给多少薪水？
- HR：这个暂且没法确定，要结合您几轮面试结果和用人部门的意见来综合评定。

如果HR这么回答你，我的建议是这样的：

虽然薪资很重要，但是我个人觉得这不是最重要的。我有以下建议：

- 如果你觉得你技术面试效果很好，可以报一个高一点的薪资，这样如果HR想要你，会找你商量的。
- 如果你觉得技术面试效果一般，但是你比较想进这家公司，可以报一个折中的薪资。
- 如果你觉得面试效果很好，但是你不想进这家公司，你可以适当“漫天要价”一下。
- 如果你觉得面试效果不好，但是你想进这家公司，你可以开一个稍微低一点的工资。

需要注意的是，面试求职是一个双向选择的过程。面试应该做到不卑不亢，千万不要因为面试结果不好，就低声下气地乞求工作，每个人的工作经历和经验都是不一样的，技术面试不好，知道自己的短板针对性地补缺补差就行，而不是在人事关系上动歪脑筋。

参考：

[面试技巧 | 技术岗位面试如何与HR谈薪](#)

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



面试回答问题的技巧

点击关注本[公众号](#)获取文档最新更新,并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板.

技术面试通常至少三轮:

1. 基础面试: 主要考察对岗位和简历中涉及到基础知识部分的提问, 包括一部分算法和场景设计的面试题, 这一面可能会涉及现场coding.
2. 项目面试: 主要考察简历中涉及的项目, 会涉及你项目的相关业务知识、扮演角色、技术取舍、技术攻坚等等.
3. HR面试: 这一面通常是HR把关, 主要涉及行为面试, 考察候选人是否价值观符合公司要求、工作稳定性如何、沟通协作能力如何等等.

当然, 对于初级岗或者校招生会涉及一轮笔试, 相当多的公司会在现场面之前进行一轮电话面试, 目的是最快速有效地把不符合要求的候选人筛除, 对于个别需要跨部门协作的岗位会涉及交叉面试, 比如前端候选人会被后端的面试官面试, 一些有管理需求的岗位或者重要岗位可能会涉及总监面试或者vp面.

而一个正常的技术面试流程(以项目面为例)分为大致三个部分:

1. 自我介绍
2. 项目(技术)考察
3. 向面试官提问

那么该如何准备技术面试, 如何在面试中掌握主动权呢?

自我介绍

几乎所有的面试都是从自我介绍这个环节开始的, 所以我们得搞清楚为什么自我介绍通常作为一个面试的开头.

为什么需要自我介绍

首先, 有一个很普遍的问题就是面试官很可能才刚拿到你的简历, 他需要在你自我介绍的时候快速浏览你的简历, 因为技术面试的面试官很多是一线的员工, 面试候选人只是其工作中的一小部分, 很多情况下是没有提前看过你的简历的.

其次, 自我介绍其实是一个热身, 面试官和候选人其实是陌生人, 自我介绍不管是面试还是其他情况下, 都是两个陌生人彼此交流的起点, 也是缓解候选人与面试官之间尴尬的一种热身方式.

最后, 自我介绍是展示自我、引出接下来技术面试的引子, 是你自己指定技术面试方向的一次机会.

知道了以上原因, 我们才能进行准备更好的自我介绍。

自我介绍的几个必备要素

自我介绍归根到底是一个热身运动, 因此切忌占用大量的篇幅, 上来就把自己从出生的经历到大学像流水账一样吐出来的, 往往会被没耐心的面试官打断, 而这也暴露了候选人讲话缺乏重点、沟通能力一般的缺点。

但是, 一些关键信息是必须体现的, 就我个人而言, 以下信息是必备的:

- 个人信息: 至少要体现出自己的姓名、岗位和工作年限, 应届生则必须要介绍自己的教育背景, 如果自己的前东家是个大厂 (比如BAT) 最好提及, 自己的学历是亮点 (985或者硕博或者类似于北邮这种CS强校) 最好提及, 其他的什么有没有女朋友、是不是独生子没人在意, 不要占用篇幅。这个部分重点在于「你是谁?」。
- 技术能力: 简要地介绍自己的技术栈, 切忌把自己只是简单使用过, 写过几个Demo或者看了看文档的所谓「技术栈」也说出来, 一旦后面问到算是自找尴尬。这个部分的重点在于「你会什么?」。
- 技能擅长: 重点介绍自己擅长的技术, 比如性能优化、高并发、系统架构设计或者是沟通协调能力等等, 切忌夸大其词, 要实事求是, 这是之后考察的重点。这个部分重点在于「你擅长什么?」。

自我介绍要有目的性

要重点匹配当前岗位的技术栈

你的面试简历可能包含了各种各样的技术栈,但是在自我介绍过程中需要匹配当前岗位的技术要求.

就比如你目前面试的是移动端H5前端的开发岗位,就重点在自我介绍中突出自己在移动前端的经验,而此时大篇幅得讲述自己如何用Node支撑公司的web项目就显得很不明智.

要在自我介绍中做刻意引导

如果你的自我介绍跟流水账一样,没有任何重点,其实面试官也很难办,因为他都没法往下接话...

而只要你稍作引导,绝大部分面试官就会接你的话茬,比如「你在自我介绍中重点提及了一个项目,碰到了一些难点,然后被你攻克了,效果如何如何好等等」,如果我是面试官一定会问「你的xx项目的xx难点后来是怎么解决的?」。

面试官的目的是考察候选人的能力,对候选人做出评估,因此需要知道候选人擅长什么,是否匹配岗位,面试官绝大多数情况下很乐意你这种有意无意的引导,这样双方的沟通和评估会很顺利,而不是故意刁难候选人。

如何准备自我介绍

其实最好的方法也是最笨的方法就是把自我介绍写下来,这个自我介绍一定要体现上面提到的几大必备要素,在面试前简单过几遍,能把自我介绍的内容顺利得表达出来即可,切忌跟背课文一样.

自我介绍的时间最好控制在1-3分钟之间,这些时间足够面试官把你的简历过一遍了,面试官看完简历后正好接着你的自我介绍进行提问是最舒服的节奏,别上来开始10分钟的演讲,面试官等待的时候会很尴尬,这么长的篇幅说明你的自我介绍一定是流水账式的.



技术考察

一个好的技术考察的开始,必须得有自我介绍部分好的铺垫和引导,有一种情况我们经常遇见:

候选人说了一大堆非重点的自我介绍,面试官一时语塞,完全get不到候选人的重点,也不知道候选人擅长什么、有什么亮点项目,然后就在他简历的技术栈中选了本公司也在用的技术,候选人这个时候也开始冒汗,因为这个技术栈并不是他的擅长,回答的也磕磕绊绊,面试官的引导和深入追问也没有达到很好的效果,面试就在这种尴尬的气氛中展开了,面试结束后面试官对候选人的评价是技术不熟练、没有深入理解原理,候选人的感受是,面试官专挑自己不会的问。

所以在前面的部分,一定要做好引导,把面试官的问题引到我们擅长的领域,但是这样还不够,正所谓不打无准备之仗,我们依然需要针对可能出现的问题进行准备.

那么如何准备可能的面试题?

比如你擅长前端的性能优化,在自我介绍的部分已经做好了引导,接下来面试官一定会重点考察你性能优化的能力,很可能涉及很有深度的问题,即使你擅长这方面的技术,但是如果没有准备也可能临场乱了阵脚.

多重提问

自我多重提问的意思是,当一个技术问题抛出的时候,你可能面对更深层次的追问



依旧以前端性能优化为例,面试官可能的提问:

1. 你把这个手机端的白屏时间减少了150%以上,是从哪些方面入手优化的?这个问题即使你没做过前端性能优化也能回答个七七八八,无非是组件分割、缓存、tree shaking等等,这是第一重比较浅的问题。
2. 我看你用webpack中SplitChunksPlugin这个插件进行分chunk的,你分chunk的取舍是什么?哪些库分在同一个chunk,哪些应该分开你是如何考虑的?如果你提到了SplitChunksPlugin插件可能会有类似的追问,如果没有实际操作过的候选人这个时候就难以招架了,这个过程一定是需要一定的试错和取舍的。
3. 在分chunk的过程中有没有遇到什么坑?怎么解决的?其实SplitChunksPlugin这个插件有一个暗坑,那就是chunk的id自增性导致id不固定唯一,很可能一个新依赖就导致id全部打乱,使得http缓存失效。

以上只是针对SplitChunksPlugin插件相关的优化提问,当然也可能从你的性能测试角度、代码层面进行考察,但是思路是类似的。

因此不能把自己准备的问题答案停留在一个很浅显的层面,一方面无法展示自己的技术深度,另一方面在面试官的深度体情况下容易丢分,因此在自己的答案后面多进行自我的追问,看一看能不能把问题做的更深入。

答题法则

很多面试相关的宝典都推荐使用STAR法则进行问题的应答,我们不想引入这个额外的概念,基础技术面试的部分老老实实回答面试官的问题即可,通常需要问题运用到这个法则的是项目面,比如让你介绍一下你最得意的项目,回答问题的法则有这几个要点:

- 项目背景:简要说一下项目的背景,让面试官知道这个项目是做什么的
- 个人角色:让面试官知道你在这个项目中扮演的角色
- 难点:让面试官知道你在项目开发过程中碰到的难点
- 解决方案:针对上面的难点你有哪一些解决方案,是如何结合业务进行取舍的
- 总结沉淀:在攻克上述的难点后有没有沉淀出一套通用的解决方案,有没有将自己的方案在大部门进行推广等等

重点就在于后面三条,也是最体现你个人综合素质的一部分,我是面试官的话会非常欣赏那种可以发现问题、找到多种方案、能对多种方案进行比对取舍还可以总结沉淀出通用解决方案回馈团队的人。

从上述几点可以体现出一个人的技术热情、解决问题的能力和总结提高的能力。

刻意引导

是的，在回答面试官提问的时候也可以做到刻意引导。

我们就举几个简单的例子：

- 除了Vue还用过Angular吗？这个时候很多候选人就很实诚回答「没有」，其实我们可以回答的更好，把你所知道的说出来展示自己的能力才是最重要的，你可以说「我虽然没用过，但是在学习双向绑定原理的时候了解了一下Angular脏检查的原理，在学习Nestjs的时候了解了依赖注入的原理，跟Angular也是类似的」，面试官一定会接着问你脏检查和依赖注入的问题，虽然你没有用过Angular，但是Angular的基本原理你都懂，这是很好的加分项，说明候选人有深入理解原理的意愿和触类旁通的能力
- Vue如何实现双向绑定的？很多候选人老老实实答了 `object.defineProperty` 如何如何操作，然后就没有了，其实你可以在回答完之后加上一嘴「Vue 3.0则选择了更好用的Proxy来替代`object.defineProperty`」或者「除了`object.defineProperty`这种数据劫持的方式，观察者模式和脏检查都可以实现双向绑定」，面试官大概率会问「Proxy好在哪？」或者「聊聊脏检查」等等，这样下一个问题就会依然在你的可控范围内



我们第一个例子把本来自答不上来的问题，转化为了成功展示自己能力的加分项，第二个例子让自己更多的展示了自己能力，而且始终使面试官的问题在自己的可控范围内。

向面试官提问

这个部分基本到了面试尾声了，属于做好了不影响大局，但是可能加分，如果做不好很容易踩雷的区域。

首先我们声明几个雷区：

- 切忌问结果：问了也白问，绝大部分公司规定不会透露结果的，你这样让大家很尴尬
- 切忌问工资：除了HR跟你谈工资的时候，千万别跟技术面试官谈工资，工资是所有公司的高压线，没法谈论
- 切忌问技术问题：别拿自己不会的技术难题反问面试官，完全没意义，面试官答也不是不答也不是

有几个比较好的提问可供参考：

- 如果我入职这个岗位的话，前三个月你希望我能做到些什么？
- 我的这个岗位的前任是因为什么离职的，我什么地方能做的更好？
- 你对这个职位理想人选的要求是什么？

尽量围绕你的岗位进行提问，这可以使得你更快得熟悉你的工作内容，也让面试官看到你对此岗位的兴趣和热情，重要的是这些问题对于面试官而言既可以简略回答，也可以详细的给你讲解，如果他很热情得跟你介绍此岗位相关的情况，说明你可能表现得不错，否则的话，你可能不在他的备选名单里，这个时候就需要你早做打算了。

总结

我们用大量篇幅介绍了技术面试中的一些应试技巧,但是归根到底候选人的基本功和丰富的项目经验才是硬道理.

如果你看完了整篇文章,并进行了精心的准备,他是可以让你从75分到85分的实用技巧,而不是让你从55到85的什么秘籍.

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



面试官到底想看什么样的简历？

点击关注本[公众号](#)获取文档最新更新，并可以领取配套于本指南的《前端面试手册》以及最标准的简历模板。

面试一直是程序员跳槽时期非常热门的话题，虽然现在已经过了跳槽的旺季，下一轮跳槽季需要到年底才会出现，但是当跳槽季的时候你再看这篇文章可能已经晚了，过冬的粮食永远不是冬天准备的，而是秋收的时候。

简历准备

简历是你进入面试的敲门砖，也是留给意向公司的第一印象，所以这个很重要，必须在这上面做足了文章，一份优秀的面试简历是整个面试成败的重中之重，我们会详细分析如何准备简历才能保证简历不被刷掉。

简历通常有这几部分构成：

1. 基本资料
2. 专业技能
3. 工作经历
4. 项目经历
5. 教育背景

我们会逐一进行分析。

准备简历模板

万事开头难，简历的编写如果从头开始需要浪费很多时间，其实最快速也最聪明的办法就是先找一份还不错的简历模板，之后我们只需要填写信息即可。

简历模板的选择很讲究，有些简历基本不看内容就会被刷掉，这些简历一般会对面试官进行视觉攻击，让简历给面试官的第一印象就是反感。

有两种坑爹的简历模板：

一种是经典简历模板，真是堪称『经典』，这种简历模板在我上小学的时候就有了，以现在的眼光看有点不够看了，配色也比较『魔幻』，加上表格类的简历属于low到底端的简历类型，基本上扫一眼就扔了，这种简历只需要3秒钟就能被面试官扔到垃圾堆。

个人简历

| | | | | |
|-------------|--|------|--|----|
| 姓名 | | 性别 | | 照片 |
| 生日 | | 身高 | | |
| 籍贯 | | 民族 | | |
| 政治面貌 | | 毕业院校 | | |
| 学历 | | 专业 | | |
| 联系电话 | | 电子邮件 | | |
| 邮箱 | | 家庭住址 | | |
| 个人简介 | | | | |
| 爱好特长 | | | | |
| 相关证书 | | | | |
| 社会实践 | | | | |

另一种是设计感十足的简历模板，这种简历设计感十足，这五颜六色的配色一定能亮瞎面试官的双眼，这种花里胡哨的简历同样也是3秒钟沉到垃圾堆底部的简历。

► 来自百图汇 www.5tu.cn

简历卡



个人信息

地址 上海市XXXX有限公司
电子邮件 baotuwang@163.com
电话 555-555-555
网址 www.XXXX.com

自我介绍

毕业中央美院，4年室内设计工作经验，我不是什么牛人我只是菜鸟，一直在成长中。从事网站的建设与分析，有美丽的幻想，有美好的憧憬。从学习到工作一直做我喜欢的设计，可以说现在是更为全面学习的时候，每一天我都很开心，有我喜欢的工作和生活。

教育/培训经历

● 设计主管
在环境学院团学联秘书部担任会议记录、文字整理以及日常公文起草工作具有较好的组织、管理、协调、沟通能力在通讯社采访部担任记者，并多次撰写新闻稿以及人物采访学联秘书部担任会议记录

● 设计主管
在环境学院团学联秘书部担任会议记录、文字整理以及日常公文起草工作具有较好的组织、管理、协调、沟通能力在通讯社采访部担任记者，并多次撰写新闻稿以及人物采访学联秘书部担任会议记录

工作经历

● 广州遨维互动科技有限公司
在环境学院团学联秘书部担任会议记录、文字整理以及日常公文起草工作具有较好的组织、管理、协调、沟通能力在通讯社采访部担任记者，并多次撰写新闻稿以及人物采访学联秘书部担任会议记录

● 广州遨维互动科技有限公司
在环境学院团学联秘书部担任会议记录、文字整理以及日常公文起草工作具有较好的组织、管理、协调、沟通能力在通讯社采访部担任记者，并多次撰写新闻稿以及人物采访学联秘书部担任会议记录

技术/能力

| | | |
|--|--|--|
| PHOTOSHOP | DREAMWEAVER | ILLUSTRATOR |
| <div style="width: 80%; background-color: #0072BD; height: 10px;"></div> | <div style="width: 80%; background-color: #0072BD; height: 10px;"></div> | <div style="width: 80%; background-color: #0072BD; height: 10px;"></div> |
| INDESIGN | AFTER EFFECTS | HTML&CSS3 |
| <div style="width: 60%; background-color: #0072BD; height: 10px;"></div> | <div style="width: 60%; background-color: #0072BD; height: 10px;"></div> | <div style="width: 80%; background-color: #0072BD; height: 10px;"></div> |

以上两类简历模板堪称面试官杀手，我相信只要你用了上述两类模板，绝对连让面试官看第二眼的兴趣都没有。

面试官筛简历要的是高效、清晰、内容突出，不管是HR还是技术面试官都想要在最快速的情况下看到有效信息，你眼中所谓的『视觉效果』在别人眼里就是『视觉噪音』或者『视觉垃圾』，严重影响看简历的心情和寻找有效信息的速度。

其实我发现不仅仅是在互联网技术招聘这个领域，大部分企业招聘的简历要求都很简单，清晰、简洁即可，最重要的是要内容清晰，突出主题。

就像这样，颜色不超过黑白灰三色，把强调的内容讲清楚，让面试官一眼就看到重点即可：

菜惦 (laidian)

+86 1001810018 | chuizijianli@100chui.com

广东省广州市天河区新塘街道



教育背景

菜惦简历大学

2017 年 9 月 - 2019 年 5 月

金融经济学 - 硕士

广州

- GPA: 3.65 (专业前 10%)

- 相关课程：经济金融学、金融统计学、会计与资本市场、公司金融、投资银行业务

菜惦简历大学

2013 年 9 月 - 2017 年 5 月

经济管理学院 金融经济学 - 硕士

广州

- GPA: 3.65 | 奖项荣誉：2015-2016 综合奖学金 (专业前 15%)、Resume Hack 领导力奖

实习经历

菜惦信息科技有限公司

2010 年 3 月 - 2012 年 3 月

软件工程师

广州

- 负责公司业务系统的设计及改进，参与公司网上商城系统产品功能设计及实施工作；
- 负责客户调研、客户需求分析、方案写作等工作，参与公司多个大型电子商务项目的策划工作。

菜惦信息科技有限公司

2012 年 4 月 - 2017 年 6 月

市场营销

广州

- 负责公司线上端资源的销售工作 (以开拓客户为主)，公司主要资源以广点通、360、沃门户等；
- 实时了解行业变化，跟踪客户的详细数据，为客户制定更完善的投放计划。

社团经历

菜惦范科技大学

2009 年 3 月 - 2011 年 6 月

校园大使主席

广州

- 目标带领自己的团队，辅助完成在各高校的“伏龙计划”，向全球顶尖的 AXA 金融公司推送实习生资源。
- 整体运营前期开展了相关的线上线下宣传活动，中期为进行咨询的人员提供讲解。

技能证书

- 普通话一级甲等；
- 大学英语四/六级 (CET-4/6)，良好的听说读写能力，快速浏览英语专业文件及书籍；
- 通过全国计算机二级考试，熟练运用 office 相关软件。

自我评价

- 专业能力：2 份实训经验，较好的理解专业理论知识，并运用到实践中。
- 组织能力：多年班委经验，成功组织多次各类院校活动落地，较强的组织协调能力。
- 性格品质：恪守职业道德、适应能力强、积极主动、认真细心、优秀的独立学习和工作能力。

简历模板可以去公众号『程序员面试官』后台回复『模板』二字领取。

准备个人信息

个人信息部分主要包括姓名、电话、电子邮箱、求职意向，当然这四个是必填的，其它的都是选填，填好了是加分项，否则很可能减分。

接下来才是重点：

1. **github**：如果准备一个基本没有更新的博客或者没有任何贡献的github，那么给面试官一种为了放上去而放上去的感觉，这基本上就是在跟面试官说『这个候选人平时根本没有总结提炼的习惯』，所以如果有长期维护的github或者博客一定要放上去，质量好的话会非常有用，如果没有千万别放。
2. **学历**：如果你的学历是专科、高中毕业之类的，还写在简历头部强调一遍，这就造成了你是『学渣』的印象，没有公司喜欢学渣的，这又增加了简历被刷的几率，如果是研究生以上学历可以写，突出一下学历优势，本科学历在技术面试领域基本上敲门砖级别的，没必要写。
3. **年龄**：如果你是大龄程序员，尤其是你还在求一份低端岗位的时候千万别写，一个大龄程序员在求职一个中低端岗位，说明这些年基本原地踏步，还不能加班，到这里基本上此简历就凉了一半了。
4. **照片**：形象优秀的可以贴，尤其是形象优秀的女程序员，其它的最好不要贴，如果要贴的话，最好是贴那种PS过的非常职业的证件照，那种平时搞怪的、光着膀子的生活照，基本就是自杀行为。

如果你没有特别之处，直接按下面这种最简单的个人信息填写方式即可，切勿给自己加戏：

张杰
+86 1001810018 | zhangjie@gmail.com
高级前端工程师

准备专业技能

对于程序员的专业技能其实就是技术栈，对自己的技术栈如何描述是个很难的问题，比如什么算是精通？什么算是了解？什么是熟悉？

关于对技术技能的描述有很多种，有五种的也有三种的，而且每个人对词汇的理解都不一样，我结合相关专家的理解和自己的理解来简单阐述下描述词汇的区别，我们这里只讲三种的了解、熟悉、精通。

- **了解**：使用过某一项技术，能在别人指导下完成工作，但不能胜任复杂工作，也不能独立解决问题。
- **熟悉**：大量运用过的某一项技术，能独立完成工作，且能独立完成有一定复杂度的工作，在技术的应用层面不会有太大问题，甚至理解一点原理。
- **精通**：不仅可以运用某一门技术完成复杂项目，而且理解这项技术背后的原理，可以对此技术进行二次开发，甚至本身就是技术源码的贡献者。

我们就以Vue这个框架为例，如果你可以用Vue写一些简单的页面，单独完成某几个页面的开发，但是无法脱离公司脚手架工作，也无法独立从0完成一个有一定复杂度的项目，只能称之为了解。

如果你有大量运用Vue的经验，有从0独立完成一定复杂度项目的能力，可以完全脱离脚手架进行开发，且对Vue的原理有一定的了解，可以称之为熟悉。

如果你用Vue完成过复杂度很高的项目，而且非常熟悉Vue的原理，是Vue源码的主要贡献者，亦或者根据Vue源码进行过魔改（比如mpVue），你可以称得上精通。



那么有两个坑是候选人经常犯的，『杂』和『精』，这种两个坑大量集中在应届生和刚毕业每两年的新手身上，其主要特点是『急于表现自我』、『对技术深度与广度出现无知而导致的过度自信』。

首先说说杂，比如你要应聘一个Java后端，老老实实把自己的java技术栈写好就行了，强调一下自己擅长什么即可，最好专精某领域比如『高并发』、『高可用』等等，这个时候一些简历非要给自己加戏，自己会的不会的一股脑往上堆，什么逆向、密码学、图形、驱动、AI都要体现出来，越杂越好，这种简历给人的印象就是个什么都不懂的半吊子。

再说说精，一个刚毕业的应届生，出来简历就各种精通，精通Java、精通Java虚拟机、精通spring全家桶、精通kafka等等，请放心，这种简历是不会没头没脑投过来了，这种在大学里就精通各种的天才早被他的各种学长介绍进了大厂或者外企做某某Star重点培养了，往往看到的这种也是半吊子。

再给大家一个技术栈模板：

- 前端技能：精通JavaScript、CSS、HTML，精通Vue、React、Angular等所有前端框架；
- 后端技能：精通Java、PHP、Python、Golang等所有后端语言；
- 其它技能：精通人工智能、精通驱动开发、精通逆向工程；

这样写的后果就在于让面试官一眼就看出你是个吹牛的半吊子，那些各种精通的全才在业界早就出名了，根本不可能还在投简历。

准备工作经历

工作经历本身不用花太多笔墨去写，面试官主要想看的就是每段工作经历的持续时间、在不同公司担任的职责如何、是否有大厂的工作经验等等。

那么什么简历在这里给面试官减分呢？

- 频繁跳槽：比如三年换了四家公司，每个公司呆的时长不要超过一年
- 常年初级岗：比如工作五六年之后依然在完成一些简单的项目开发
- 末流公司经历：在技术招聘届，大厂的优先级最高比如BAT、TMD甚至微软、谷歌等外企，知名度独角兽其次，比如商汤、旷视等等，一般的互联网公司排在第三，就是工作中小型的互联网公司一般大家叫不上名字，排在最后的就是外包和传统企业的经历

所以，如果你有频繁跳槽的经历怎么办？在本公司老老实实等到满一年再跳槽。

如果常年初级岗怎么办？想办法晋升或者参与一些业界知名项目，再或者写一个有一定复杂度的私人项目。

如果有末流公司经历怎么办？如果是很久以前的末流公司经验可以直接不写，也没人在乎你很早之前的工作经历，如果你现在就在末流公司，赶紧想办法跳槽，去不了大厂，去非知名的互联网公司也算是胜利大逃亡了。

不建议任何形式的简历造假，如果去一些大厂，分分钟被调出来，与其简历造假，不如现在就行动起来，比如从现在到年底跳槽季，深度参与一个知名开源项目或者做一个有一定复杂度的私人项目绰绰有余，除非996.

准备项目经历

项目经历不管对于社招还是校招都是重中之重，很多时候成败就在于项目经历这块，一个普通本科可以通过优秀的项目经历逆袭985，一个小厂的员工也可以获得大厂的面试机会。

但是必须要说一下项目经历的编写很讲究，这是为后面面试部分铺路的绝佳机会，也是直接让你的简历扑街的重点沦陷区域。

先说容易让简历扑街的几个坑位。

切忌流水账写法

项目经历流水账写法是绝大多数简历的通病，通篇下来就讲了一件事『我干了啥』。

大部分简历却是这样的：

用Vue、Vuex、Vue-router、axios等技术开发电商网站的前端部分，主要负责首页、店铺详情、商品详情、商品列表、订单详情、订单中心等相关页面的开发工作，与设计师与后端配合，可要高度还原设计稿。

这个描述有什么问题？

其实看似也没啥问题，但是这种流水账写法太多了，完全无法突出自己的优势展现自己的能力。

项目经历是考察重点，面试官想知道候选人在一次项目经历中扮演的角色、负责的模块、碰到的问题、解决的思路、达成的效果以及最后的总结与沉淀。

而上面的描述只显示了『我干了啥』，所以这种项目描述几乎是没意义的，因为对于面试官而言他看不到有效信息，没有有效信息的项目描述基本就没价值了，如果这个时候你还没有大厂经历或者名校背书，基本上也就凉了。

切忌堆积项目

堆积项目这种现象往往出现在没有什么优秀项目经历的简历身上，候选人企图以数量优势掩盖质量的劣势，其实往往适得其反，项目经历的一栏最好放2-3个项目，非常优秀的项目可能放一个就足够了，举个极端例子如果有一天尤雨溪写简历，其实只需要在项目经历那些一行『Vue.js作者』就行了，当然，他并不需要投简历。

有一些项目切忌放上去：

- demo级项目：很多简历居然还在放一些仿xx官网的demo，这是十足的减分项，有一些则是东拼西凑抄了一些框架的源码搞了个玩具项目，也没有任何价值。
- 烂大街的项目：这种以vue技术栈的为最，由于视频网站的某门课程流行，导致大量的仿饿了么、仿qq音乐、仿美团、仿去哪儿，同样Java的同学也是仿电商网站、仿大众点评等等，十份简历5份一模一样的项目，你是面试官会怎么想。
- 低质量的开源项目：一个大原则就是低star的尽量别放（除非是高质量代码的冷门项目），长期弃坑的也不要放，不要为了凑数量把低质量的项目暴露出来，好好藏着。

如果只放两个项目，最好的搭配是一个公司内部挑大梁的项目和一个社区内的开源项目，后者之所以可以占据一席之地，是因为通过你的开源项目，面试官可以通过commit完整看到你的创造过程，比如工程化建设、commit规范、代码规范、协作方式、代码能力、沟通能力等等，这甚至比面试都有用，没有比开源项目更能展示你综合素质的东西了。

切忌放虚假项目

一个项目做没做过只要是有经验的面试官一问便知，如果你真的靠假项目忽悠过了面试，那这个公司八成也有问题，人才把关不过硬，你可以想象你的队友都是什么水平，在这种公司大成长价值也不大。

好，如果说实在没项目可写了，我只能造假了，那么你应该想一下这多层追问。

比如你说你优化了一个前端项目的首屏性能，降低了白屏时间，那么面试官对这个性能优化问题会进行深挖，来考察候选人的实际水平：

1. 你的性能优化指标是怎么确定的？平均下来时间减短了多少？
2. 你的性能是如何测试的？有两种主流的性能测试方法你是怎么选的？
3. 你是根据哪些指标进行针对性优化的？
4. 除了你说的这些优化方法还有没有想过通过xx来解决？
5. 你的这个优化方法在实际操作中碰到过什么问题吗？有没有进一步做过测试？
6. 我们假设这么一种情况，比如xxxx，你会这么进行优化？

面试官多层追问的逻辑是这样的：

了解背景 -> 了解方案 -> 深挖方案 -> 模拟场景

首先得了解你性能优化的指标如何，接着需要了解你是这么测试的指标、再怎么进行针对性优化的，再接着提出一些其它解决方案考察你对优化场景的知识储备和方案决策能力，最后再模拟一个其它的业务场景，来考察你的技能迁移能力，看看是否是对某块领域有一定的了解，而不是只针对某个项目。

如果要真的在面试现场对答如流，那么一定是在某一块领域有一定知识储备的人，不是随随便便搞个项目就能蒙混过关的。

合格的项目经历如何写

合格的项目经历必须要有以下几点：

- 项目概述
- 个人职责
- 项目难点
- 工作成果

如果你不怕字太多，还可以选择性加入解决方案、选型思路等等，但是由于篇幅限制和为面试铺垫就不太建议写得太多。

项目概述的目的是让面试官理解项目，不是每个人面试官都做过你的那种项目，所以需一个简述方便面试官理解。

个人职责就是告诉面试官你在本项目中扮演的角色，是领导者？主导者？还是跟随者，你负责了哪些模块，承担了多大的工作量，以此来评估你在团队中的作用。

项目难点的目的在于让面试官看到你碰到的技术难题，方便后续面试对项目进行一系列讨论。

工作成果就很明显了，面试官需要看到你在做了上述工作到底达成了什么成绩，这个时候最好以数据说话，比如访问量、白屏时间等等。

像这种项目经历描述就比较合适：

项目经历

DataReport 系统

2018 年 3 月 - 2018 年 9 月

项目概述： DataReport 是可配置的可视化大屏系统，此系统最大的卖点是可视化搭建、无编程门槛以及丰富的行业案例，可以帮助企业快速、方便地搭建可视化大屏产品。

负责内容：

- 负责 DataReport 系统技术选型工作；
- 负责 DataReport 编辑器部分的设计与编码；
- 负责 DataReport 的前端性能优化工作；

项目难点：

- 编辑器作为主力交互区域，需要面对大量的用户事件和用户输入，再加上多图联动，逻辑与交互极其复杂；
- 大规模数据展示造成页面卡顿，部分动画掉帧严重；

项目成果：

- DataReport 项目上线一周，付费用户超过预期用户量的 20% 以上；
- 大规模数据展示的性能经过优化之后，动画无掉帧现象，性能提升超过 60%；

这个时候也切忌展开长篇大论，把技术细节一个个写上去，甚至还写了心路历程的都是大忌，一方面篇幅太大会造成视觉混乱，另一方面面试官想看到的是『简』历，不是技术总结，面试官要面对上百份简历没那么时间来看你长篇大论，长篇大论大可以在面试中展开。

最好的方法就是一行文字简单得说清楚即可，反正项目面的时候一定会问到，到时候好好把你准备的内容讲给面试官，掌握面试的主动权就是从项目经历这一栏中开始。

教育背景

应届生可以写得更详细一点，比如绩点排名怎么样，有没有突出的科目，社招就不要写太多了，简单的入学时间、学校、专业即可，而且写你的最高学历即可，没必要从初中就开始写学历流水账，没有人看的。

教育背景

合肥工业大学

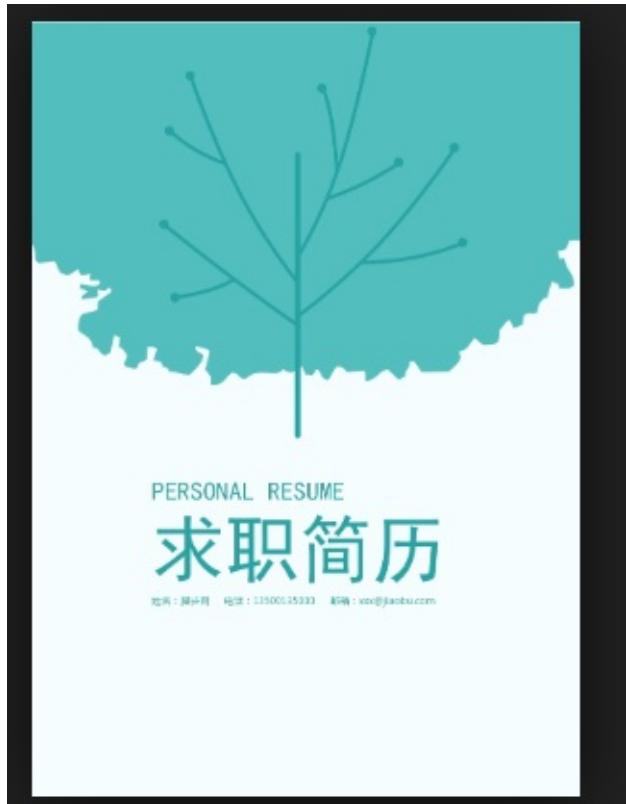
2013 年 9 月 - 2017 年 6 月

信息安全

合肥

几点注意事项

- 自我评价不建议写：技术面试几乎没人看你的自我评价，连面试技术问题都嫌『talk is cheap show me the code』，你的自我评价除了占篇幅没啥用处，充其量算是面试官的干扰信息。
- 简历封面千万别搞：这都是一些简历制作网站骗用户付费的伎俩，不仅是互联网行业，其它行业我也没见过要简历封面这种无用操作的。



- 证书不建议写：应届生可以酌情考虑弄个六级证书什么的，对于社招而言，列一堆证书甚至是减分项，国内的各种证你也懂的，是有多不自信才沦落到靠一堆证书来证明自己的价值。
- 千万别用技能图表：首先用90分、80分来评价自己的技术本身就没有说服力，也不可能这么精准，而且什么是90分、什么是80根本就没有一个公论，所以用一般的比较通用的熟悉、精通描述即可，千万别加戏，面试官或者HR没那么多闲工夫去理解你的图表，老老实实按最通用高效的方式描述自己的技术栈。



- 简历最好一页：程序员又不是设计师有时候需要作品呈现，如果你的简历超过一页那么一定是出问题了，要么项目、技术栈描述太多太杂占据大量篇幅，要么加了一堆图表或者图画来加戏，当然往往是犯前一个错误的更多。

这是我在网上找到的一个例子很能说明问题：

小白用STAR法则写出来的描述

Situation (情景)

Task (任务)

Action (行动)

Result (结果)

- “【情景】我在XX公司担任销售人员一职，负责公司的XX产品销售任务。【任务】我每月的KPI是卖掉350份XX，可以运用公司的客户数据网络，也可以自己去拓展新客户，还有另外专门的电销部门和运营部门同事负责配合。【行动】我收到任务第一天就开始布局三个不同商圈的重点客户，将600个潜在客户分为三匹进行电话+登门拜访，并且培训了两个销售助理帮助我高效完成客户的初步沟通。【结果】在开始行动三周后，我成功销售了XX产品500份，为公司带来了2万元的销售业绩，获得当月销售金奖。”

220字

当HR说“用STAR法则”的实际含义

Situation (情景)

Task (任务)

Action (行动)

Result (结果)

- “在XX公司担任销售人员一职，XX产品累计销售500份，为公司带来实际收入2万元，业绩在当月所有销售人员中排第一”

50字

简历的版面寸土寸金，别说话跟裹脚布一样，精炼的一句话即可描述你的问题。

- 不建议用任何简历制作网站或者开源的简历制作器：我之前不仅用过上述的东西，还付过费，完全是浪费时间和浪费金钱，先说简历制作网站基本上都是那种花里胡哨的简历，看起来炫但是基本是面试官最讨厌的那种形式，开源的简历制作器也是类似的，我甚至还为了自己的简历魔改过这种制作器，到头来也是浪费时间，记住简历『黑白灰』三个配色，简洁即可，切勿让简历形式喧宾夺主。

这是我整理的简历范本（项目经历可以多写一个）：

张杰

+86 1001810018 | zhangjie@gmail.com

高级前端工程师

技术技能

- 熟练掌握 JavaScript、CSS、HTML，可以脱离框架独立进行复杂项目的开发；
- 熟练掌握 Vue、React 等前端框架以及相关的全家桶，有基于全家桶构建通用前端框架的经验；
- 精通前端组件库设计，有主导设计过公司内部前端组件库的经验；
- 熟练掌握 TypeScript，有丰富的高级类型编程经验和宏编写经验；
- 熟悉前端工程化，有前端构建工具的编写经验以及独立开发前端监控系统的经验；

项目经历

DataReport 系统

2018 年 3 月 - 2018 年 9 月

项目概述：DataReport 是可配置的可视化大屏系统，此系统最大的卖点是可视化搭建、无编程门槛以及丰富的行业案例，可以帮助企业快速、方便地搭建可视化大屏产品。

负责内容：

- 负责 DataReport 系统技术选型工作；
- 负责 DataReport 编辑器部分的设计与编码；
- 负责 DataReport 的前端性能优化工作；

项目难点：

- 编辑器作为主力交互区域，需要面对大量的用户事件和用户输入，再加上多图联动，逻辑与交互极其复杂；
- 大规模数据展示造成页面卡顿，部分动画掉帧严重；

工作经历

网易

2017 年 3 月 - 2019 年 6 月

高级前端工程师

杭州

- 参与有数项目的前端开发工作，主导了 DataReport 可配置大屏系统的前端研发工作；
- 负责面向部门业务的高度定制化图表库的日常维护与开发；
- 负责新人关于可视化与图形技术的培训与指导；

教育背景

合肥工业大学

2013 年 9 月 - 2017 年 6 月

信息安全

合肥

简历范本可以去公众号『程序员面试官』后台回复『模板』二字领取。

你可能的疑问

如果你读到这里，谢谢你的耐心，可能你也会有疑问--『你这篇文章，这不让写，那不让写，我的简历填都填不满，怎么办？』。

实际上一份简历很多部分是已经固定了的，比如个人信息、教育背景、工作经历等等，其实能做文章的部分也只有技术栈和项目经历，也就是说后面两个部分是可以靠当下努力来改变的。

举个简单的例子，比如你做了3年的Java开发，公司还是用很老旧的SSM技术栈，自己其实有点沦为框架小子的意思，只能做一些增删改查这种类型的工作，虽然工作内容都能胜任，但是根本做不了更有挑战性的事情，而外面对Java工程师的要求已经越来越高了。

岗位要求：

- 1) 本科或以上学历，计算机软件或相关专业，3年以上Java开发经验
- 2) 熟悉Java/JEE，基础扎实，熟练掌握常用Java技术框架，能编写高质量简洁清晰的代码
- 3) 对于Java基础技术体系（包括JVM、类装载机制、多线程并发、IO、网络）有一定的掌握和应用经验
- 4) 良好的面向对象设计理解，熟悉面向对象设计原则，掌握设计模式及应用场景
- 5) 具有比较强的问题分析和处理能力，有比较优秀的动手能力，热衷技术，精益求精，有一定的技术癖
- 6) 熟悉底层中间件、分布式技术（包括缓存、消息系统、热部署、JMX等）
- 7) 有互联网行业高并发、高稳定可用性、高性能、大数据处理相关的开发、设计经验

我们完全可以花半年到一年的时间对某个细分领域进行专门的学习和实践，我们可以通过写私人项目、参与开源项目的方式增加自己的项目经验和项目履历，一段时间后你肯定在某个细分领域至少处于一个进阶水平，你的简历也不可能填都填不满。

对于前端工程师也是一样，如果你觉得你逐渐沦为页面仔，自己也没有拿得出手的项目，也不妨多思考之前的项目是不是有的性能部分可以优化，是不是平时的工作有很多重复性的，能不能通过node工具或者vscode插件来提高效率，又或者公司的框架用起来太繁琐，可不可以进行改造升级提高生产力。

这个时候可能有人又问，『我自己工作都多的不行，凭什么还想为公司写什么工具框架？公司会额外付钱吗？』

你写的框架和工具是你未来跳槽中的简历的重要部分，即使它现在不会变现，在你跳槽过程中一定会变现，总之这些额外工作是为你自己打工的，你的现任公司只是因此额外受益了而已。

总结

我知道现在并不是跳槽的旺季，可能很多人不会看这篇文章，但是当真正跳槽季来临的时候，往往很多人又开始为填满自己的简历而发愁，当自己的简历石沉大海，又会冒出这种言论：

- 哎呀，还是自己学历不够好，我能力没问题就是吃了学历的亏
- 自己没有大厂的履历真是吃亏，自己能力没问题，就是没大厂背书
- 所在的公司都是一些老技术栈，我的简历就太吃亏了，都怪公司

实际情况是，大厂履历、名校经历、出色项目只要有一项拿得出手，就会成为抢手货，更何况随着时间的推移，教育背景就越发不重要，更重要的还是工作履历和项目经历。

与其今后发愁如何填满简历，不如现在行动为自己的简历『打工』。

公众号

想要实时关注笔者最新的文章和最新的文档更新请关注公众号**程序员面试官**,后续的文章会优先在公众号更新.

简历模板: 关注公众号回复「模板」获取

《前端面试手册》: 配套于本指南的突击手册,关注公众号回复「fed」获取



程序员面试官

📍 扫码关注不迷路

面试真题 · 面试技巧 · 职场干货 · 技术前瞻

程序员面试官

kaikeba
开课吧