cookies、sessionStorage 和 localStorage 的区别

一、背景介绍

首先我来说一下 cookie，他的意思是小甜饼，顾名思义，cookie 确实非常小，它的大小限制为 4KB 左右，是网景公司的一位员工在 1993 年 3 月的发明。它的主要用途是保存登录信息，比如你登录某个网站可以看到“记住密码”，这通常就是通过在 Cookie 中存入一段识别用户身份的数据来实现的。

其次是 localStorage，它是 HTML5 标准中新加入的技术，它并不是什么划时代的新东西。早在 IE 6 时代，就有一个叫 userData 的东西用于本地存储，而当时考虑到浏览器的兼容性，更通用的方案是使用 Flash。而如今，localStorage 已经被大多数浏览器所支持

最后要说的是 sessionStorage，它与 localStorage 的接口类似，但保存数据的生命周期与 localStorage 不同。我们都应该知道 Session 这个词的意思，直译过来是“会话”。而 sessionStorage 是一个前端的概念，它只是可以将一部分数据在当前会话中保存下来，刷新页面数据依旧存在。但当页面关闭后，sessionStorage 中的数据就会被清空。

二、三者的区别

特性	Cookie	localStorage	sessionStorage
数据的生命期	可设置失效时间，默认是关闭浏览器后失效	除非被清除，否则永久保存	仅在当前会话下有效，关闭页面或浏览器后被清除
存放数据大小	4K左右	一般为5MB	一般为5MB
与服务器端通信	每次都会携带在HTTP头中，如果使用cookie保存过多数据会带来性能问题	仅在客户端（即浏览器）中保存，不参与和服务器的通信	仅在客户端（即浏览器）中保存，不参与和服务器的通信
易用性	需要程序员自己封装，源生的Cookie接口不友好	源生接口可以接受，亦可再次封装来对Object和Array有更好的支持	源生接口可以接受，亦可再次封装来对Object和Array有更好的支持

三、应用场景

因为考虑到每个 HTTP 请求都会带着 Cookie 的信息，所以 Cookie 当然是能精简就精简！比较常用的一个应用场景就是判断用户是否登录。针对登录过的用户，服务器端会在他登录时往 Cookie 中加入一段加密过的唯一辨识单一用户的辨识码，下次只要读取这个值就可以判断当前用户是否登录啦。曾经还使用 Cookie 来保存用户在电商网站的购物车信息，如今有了 localStorage，似乎它在这个方面便可以给 Cookie 放个假了~

而另一方面 localStorage 接替了 Cookie 管理购物车的工作，同时也能胜任其他一些工作。比如 HTML5 游戏通常会产生一些本地数据，localStorage 则是非常适合做这个工作的。如果遇到一些内容特别多的表单，为了优化用户体验，我们可能要把表单页面拆分成多个子页面，然后按步骤引导用户填写。这时候 sessionStorage 的作用就发挥出来了。

前端如何优化性能

加载时的优化

第一点：减少 HTTP 请求

一个完整的 HTTP 请求需要经历 DNS 查找，TCP 握手，浏览器发出 HTTP 请求，服务器接收请求，服务器处理请求并发回响应，浏览器接收响应等等一系列复杂的过程。当你请求较多时，直接体现在了消耗性能上面，这就是为什么要将多个小文件合并为一个大文件，从而减少 HTTP 请求次数的原因。

第二点：使用服务器端渲染

我们知道，当客户端渲染时，他是获取 HTML 文件，根据需要下载 JavaScript 文件，运行文件，生成 DOM，再渲染。这个在无形之中会拖慢我们的性能

那么服务器端渲染又是怎么一回事呢？他就是，服务端返回 HTML 文件，客户端只需解析 HTML 即可。

第三点：静态资源使用 CDN

什么是 CDN，CDN 就是，内容分发网络，它是一组分布在多个不同地理位置的 Web 服务器。我们都知道，当服务器离用户越远时，延迟越高。CDN 就是为了解决这一问题，在多个位置部署服务器，让用户离服务器更近，从而缩短请求时间。

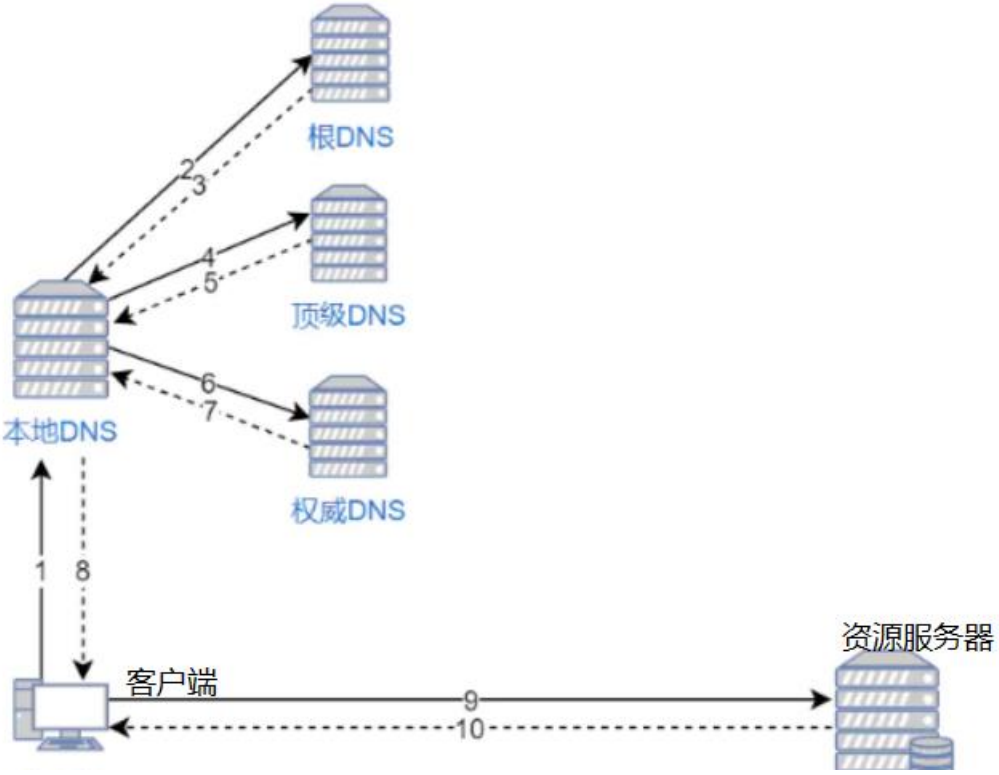
CDN 原理：

当用户访问一个网站时，如果没有 CDN，过程是这样的：

1.浏览器要将域名解析为 IP 地址，所以需要向本地 DNS 发出请求。

2.本地 DNS 依次向根服务器、顶级域名服务器、权限服务器发出请求，得到网站服务器的 IP 地址。

3.本地 DNS 将 IP 地址发回给浏览器，浏览器向网站服务器 IP 地址发出请求并得到资源。



如果用户访问的网站部署了 CDN，过程是这样的：

1.浏览器要将域名解析为 IP 地址，所以需要向本地 DNS 发出请求。

2.本地 DNS 依次向根服务器、顶级域名服务器、权限服务器发出请求，得到全局负载均衡系统（GSLB）的 IP 地址。

3.本地 DNS 再向 GSLB 发出请求，GSLB 的主要功能是根据本地 DNS 的 IP 地址判断用户的位置，筛选出距离用户较近的本地负载均衡系统（SLB），并将该 SLB 的 IP 地址作为结果返回给本地 DNS。

4.本地 DNS 将 SLB 的 IP 地址发回给浏览器，浏览器向 SLB 发出请求。

5.SLB 根据浏览器请求的资源和地址，选出最优的缓存服务器发回给浏览器。

6.浏览器再根据 SLB 发回的地址重定向到缓存服务器。

7.如果缓存服务器有浏览器需要的资源，就将资源发回给浏览器。如果没有，就向源服务器请求资源，再发给浏览器并缓存在本地。

第四点：CSS 写头部，JavaScript 写底部

所有放在 head 标签里的 CSS 和 JS 文件都会堵塞渲染。如果这些 CSS 和 JS 需要加载和解析很久的话，那么页面就空白了。所以 JS 文件要放在底部，等 HTML 解析完了再加载 JS 文件。

那为什么 CSS 文件还要放在头部呢?

因为先加载 HTML 再加载 CSS，会让用户第一时间看到的页面是没有样式的、“丑陋”的，为了避免这种情况发生，就要将 CSS 文件放在头部了。

另外，JS 文件也不是不可以放在头部，只要给 script 标签加上 defer 属性就可以了，异步下载，延迟执行。

第五点：字体图标代替图片图标

字体图标就是将图标制作成一个字体，使用时就跟字体一样，可以设置属性，例如 font-size、color 等等，非常方便。并且字体图标是矢量图，不会失真。还有一个优点是生成的文件特别小。

第六点：利用缓存不重复加载相同的资源

为了避免用户每次访问网站都得请求文件，我们可以通过添加 Expires 来控制这一行为。Expires 设置了一个时间，只要在这个时间之前，浏览器都不会请求文件，而是直接使用缓存。

第七点：图片优化

这里分为几个小点，首先是：图片延迟加载；就是在页面中，先不给图片设置路径，只有当图片出现在浏览器的可视区域时，才去加载真正的图片，这就是延迟加载。对于图片很多的网站来说，一次性加载全部图片，会对用户体验造成很大的影响，所以需要使用图片延迟加载。第二个就是：降低图片质量；图片 100% 的质量和 90% 的质量通常看不出来区别，尤其是用来当背景图的时候。我们可以在用 PS 切背景图时， 将图片切成 JPG 格式，并且将它压缩到 60% 的质量，这样基本看不出来区别。第三个就是：尽可能利用 CSS3 效果代替图片；有很多图片使用 CSS 效果（渐变、阴影等）就能画出来，这种情况选择 CSS3 效果更好。因为代码大小通常是图片大小的几分之一甚至几十分之一。最后一个就是，使用雪碧图，相信大家也都明白。

第八点：通过 webpack 按需加载代码

懒加载或者按需加载，是一种很好的优化网页或应用的方式。这种方式实际上是先把你的代码在一些逻辑断点处分离开，然后在一些代码块中完成某些

操作后，立即引用或即将引用另外一些新的代码块。这样加快了应用的初始加载速度，减轻了它的总体体积，因为某些代码块可能永远不会被加载。

运行时的优化

第一点：减少重绘重排

用 JavaScript 修改样式时，最好不要直接写样式，而是替换 class 来改变样式。再一个就是，如果要对 DOM 元素执行一系列操作，可以将 DOM 元素脱离文档流，修改完成后，再将它带回文档。推荐使用隐藏元素（display:none）或文档碎片，都能很好的实现这个方案。

第二点：使用事件委托

事件委托利用了事件冒泡，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。所有用到按钮的事件（多数鼠标事件和键盘事件）都适合采用事件委托技术， 使用事件委托可以节省内存。

第三点：if-else 对比 switch

当判断条件数量越来越多时，越倾向于使用 switch 而不是 if-else。不过，switch 只能用于 case 值为常量的分支结构，而 if-else 更加灵活。

第四点：不要覆盖原生方法

无论你的 JavaScript 代码如何优化，都比不上原生方法。因为原生方法是用低级语言写的，并且被编译成机器码，成为浏览器的一部分。当原生方法可用时，尽量使用它们，特别是数学运算和 DOM 操作。

第五点：降低 CSS 选择器的复杂性

浏览器读取选择器，遵循的原则是从选择器的右边到左边读取。所以，尽可能的降低 CSS 选择器的复杂性

第六点：使用 flexbox 布局

在早期的 CSS 布局方式中我们能对元素实行绝对定位、相对定位或浮动定位。而现在，我们有了 flexbo 布局方式，它比起早期的布局方式来说更有优势，那就是性能比较好。不过 flexbox 兼容性还是有点问题，不是所有浏览器都支持它，所以要谨慎使用。

第七点：用 transform 和 opacity 属性更改来实现动画

在 CSS 中，transforms 和 opacity 这两个属性更改不会触发重排与重绘，它们是可以由合成器单独处理的属性。

如何理解语义化

第一步，语义化的背景：

以前的 HTML 的结构，基本上就是 DIV+CSS。然而，DIV 它并没有什么任何的意义，全靠 CSS 显示页面的样式。那么近几年呢，开发者提出了 HTML 结构的语义化，所以 W3C 就制定出了语义化标签。

第二步，什么是语义化：

语义化就是构成 HTML 结构的标签要有意义，比如有这样的标签：head 表示页面头部，main 表示页面主题，footer 表示页面底部。那么这些标签构成的 HTML 结构就是有意义的，也是语义化的。如果说页面的头部、主体以及底部都用 div 来表示，那么他就不是一个语义化的 HTML 结构了。

第三步，怎么知道页面结构是否语义化了呢?

去掉 CSS，看 HTML 代码的结构，是否清晰，再看页面内容呈现，是否可以正常显示

第四步，为什么要语义化

对于编写代码的人来说，有语义化标签的 HTM 的结构，更加清晰，方便编写代码；对于团队来说，方便团队的开发与维护；对于爬虫，有利于 SEO；对于浏览器更加方便解析，最重要的是对于用户，一是假如因为网速的原因导致没有加载 CSS，页面也能呈现出良好的结构。二是某些标签属性如 alt，title 能带来良好的用户体验，还有用好 label 标签更利于用户交互；三是在特殊终端如视障阅读器中语义化的 HTML 可以呈现良好的结构。。。

第五步，怎么做才能写出语义化的 HTML 呢?

少使用或者不适用 div 和 span 标签；用 p 标签代替 div 标签；不适用样式标签，如：b 标签、font 标签；强调文本放在 strong 或者 em 标签中，不要用 b 和 i 标签；使用表格 table 时，标题要用 caption，表头要用 thead，主体部分用 tbody 包围，尾部用 tfoot 包围；表头用 th，单元格用 td。表单域用 filedset 包裹，用 lengend 标签说明表单的用途。input 标签通过 id 属性或 for 属性与 label 标签关联。html 语义化，css 类名也要语义化。等等。。。说出来这些，足够让面试官对你刮目相看了。

HTML5 都新增了哪些标签，比如：header、footer、hgroup、nav、aside、article

等等

清除浮动基本概念

块级(block)元素：总是独占一行，表现为另起一行开始，而且其后的元素也必须另起一行显示。宽度(width)、高度(height)、内边距(padding)和外边距(margin)都可控制。块级元素主要有：address、blockquote、center、dir、div、dl、fieldset、form、h1、h2、h3、h4、h5、h6、hr、isindex、menu、noframes、noscript、ol、p、pre、table、ul、li 等。

内联(inline)元素：和相邻的内联元素在同一行，宽度(width)、高度(height)、内边距的 top/bottom(padding-top/padding-bottom) 和外边距的 top/bottom(margin-top/margin-bottom) 都不可改变，就是里面文字或图片的大小。内联元素主要有：a、abbr、acronym、b、bdo、big、br、cite、code、dfn、em、font、i、img、input、kbd、label、q、s、samp、select、small、span、strike、strong、sub、sup、textarea、t、u、var 等。

可变元素：可变元素会根据上下文语境决定该元素是块级元素还是内联元素。可变元素主要有：button、del、iframe、ins、map、object、script 等。

浮动：设置浮动的元素会脱离文档流，不会影响块元素的布局，但是会影响内联元素的排列(通常是文本)。

文档流：在文档流中，块元素会单个元素独占一行。

浮动规律

如果浮动元素的上一个元素也是浮动，那么该元素会与上一个元素排列在同一行，如果行宽不够，后面的元素会被挤到下一行

如果浮动元素的上一个元素不是浮动，那么该元素仍然处于上一个元素的下方，根据浮动设定在左或者在右，而其本身也脱离文档流。后边的元素会自动往上移动到上一个文档流块元素下方为止。

高度坍塌

当没有指定高度的父元素中的子元素全部都浮动时，父元素中内部高度因为是普通流中的块元素撑起来的，所以这个时候父元素的高度会变成 0，或者会有部分浮动元素的位置会超出父元素的高度之外。这种现象，叫做"高度坍塌"。

先看下以下代码:

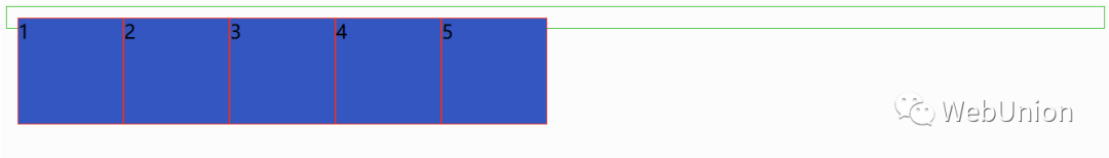
HTML

```
...
<ul className="float-list">
<li className="item">1</li>
<li className="item">2</li>
<li className="item">3</li>
<li className="item">4</li>
<li className="item">5</li>
</ul>
...
```

CSS 代码:

```
...
.float-list{
    padding: 10px;
    border: 1px solid #41c134;
    list-style: none;}
.float-list .item{
    width: 100px;
    height: 100px;
    background-color: #3456c1;
    float: left;
    border: 1px solid #e93124;}
```

以上 CSS 代码中，每个列表项元素都设置了 float:left，且列表元素没有设置具体高度值，所以就出现了"高度坍塌"，效果图如下:



清除浮动

简单来说，清除浮动的直接目的就是解决前面所说到的高度坍塌问题。清除浮动的方式主要有以下这些。

使用带 clear 属性的空元素

在浮动元素后使用一个空元素如 ，并在 CSS 中赋予.clear{clear:both;}属性即可清理浮动。亦可使用<br class="clear" />或<hr class="clear" />来进行清理。

优点：简单，代码少，浏览器兼容性好。 缺点：需要添加大量无语义的 HTML 元素，代码不够优雅，后期不容易维护。

使用这种方法，HTML 部分代码变为这样:

```
...
<ul className="float-list">
<li className="item">1</li>
<li className="item">2</li>
<li className="item">3</li>
<li className="item">4</li>
<li className="item">5</li>
<br class="clear" />
</ul>
...
```

CSS 代码:

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

效果图如下(后面讲述的其他清除浮动效果图都是一样的,故之后不会给出):

使用 CSS 的 overflow 属性

给浮动元素的容器添加 overflow:hidden;或 overflow:auto;可以清除浮动，另外在 IE6 中还需要触发 hasLayout，例如为父元素设置容器宽高或设置 zoom:1。在添加 overflow 属性后，浮动元素又回到了容器层，把容器高度撑起，达到了清理浮动的效果。

这种方法比第一种方法稍微好用点，不需要改动到 HTML 结构，只需要给容器元素添加 CSS 属性，代码如下:

```
...
.oat-list{
padding: 10px;
border: 1px solid #41c134;
```

```
list-style: none;
overflow: hidden;
*zoom: 1;}
```

给浮动的元素的容器添加浮动

给浮动元素的容器也添加上浮动属性即可清除内部浮动，但是这样会使其整体浮动，影响布局，不推荐使用。

使用 CSS 的:after 伪元素

结合:after 伪元素和 IEhack，可以完美兼容当前主流的各大浏览器，这里的 IEhack 指的是触发 hasLayout。给浮动元素的容器添加一个 clearfix 的 class，然后给这个 class 添加一个:after 伪元素实现元素末尾添加一个看不见的块元素清理浮动。这种是最推荐的清除浮动用法。

同样只需要修改 CSS 的内容:

```
...
.float-list:after{
content: "020";
display: block;
height: 0;
clear: both;
visibility: hidden;}
```

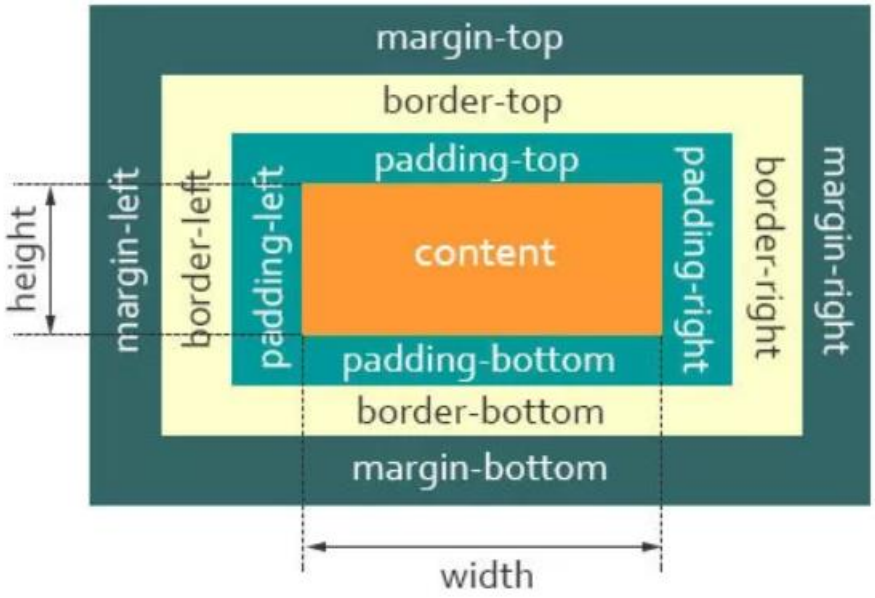
如何理解 CSS 盒模型

一、概念

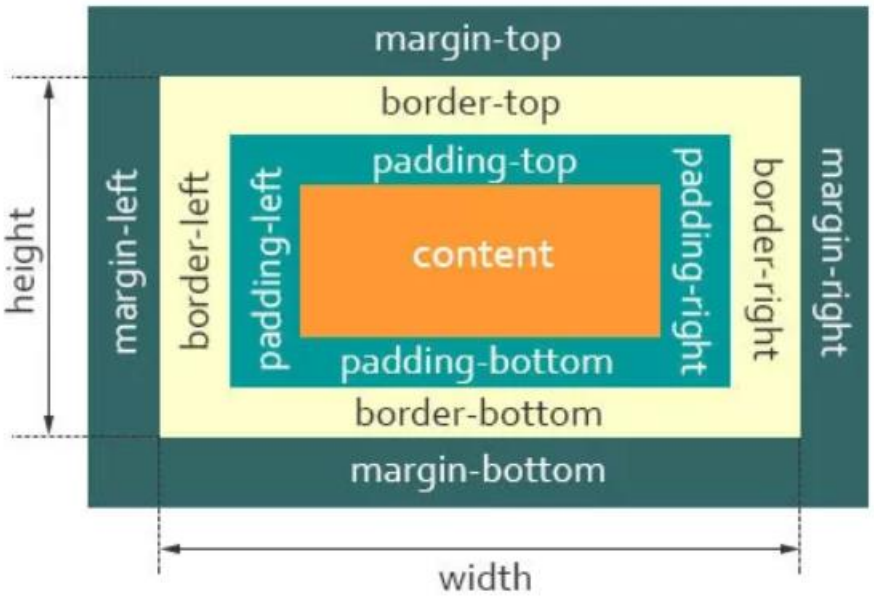
CSS 盒模型本质上是一个盒子，封装周围的 HTML 元素，它包括：外边距（margin）、边框（border）、内边距（padding）、实际内容（content）四个属性。

CSS 盒模型：标准模型 + IE 模型

1.1 W3C 盒子模型(标准盒模型)



1.2 IE 盒子模型(怪异盒模型)



二、知识点

2.1 标准模型和 IE 模型的区别

计算宽度和高度的不同。

标准盒模型：盒子总宽度/高度 = width/height + padding + border + margin。（ 即 width/height 只是内容高度，不包含 padding 和 border 值 ）

IE 盒子模型：盒子总宽度/高度 = width/height + margin = (内容区宽度/高度 + padding + border) + margin。（ 即 width/height 包含了 padding 和 border 值 ）

2.2 CSS 如何设置这两种模型

标准：box-sizing: content-box; (浏览器默认设置)

IE： box-sizing: border-box;

2.3 JS 如何获取盒模型对应的宽和高

（1）dom.style.width/height 只能取到行内样式的宽和高，style 标签中和 link 外链的样式取不到。

（2）dom.currentStyle.width/height （只有 IE 兼容）取到的是最终渲染后的宽和高

（3）window.getComputedStyle(dom).width/height 同（2）但是多浏览器支持，IE9 以上支持。

（4）dom.getBoundingClientRect().width/height 也是得到渲染后的宽和高，大多浏览器支持。IE9 以上支持，除此外还可以取到相对于视窗的上下左右的距离。

（5）dom.offsetWidth/offsetHeight 包括高度（宽度）、内边距和边框，不包括外边距。最常用，兼容性最好。

2.4 实例题（根据盒模型解释边距重叠）

例：父元素里面嵌套了一个子元素，已知子元素的高度是 100px，子元素与父元素的上边距是 10px，计算父元素的实际高度。

CSS:

```
.parents {
    width: 100px;
    background: #FF9934;
}
.child {
   width: 100%;
   height: 100px;
   background: #336667;
   margin-top: 10px;
}
```

Html:

```
<section class='parents'>
   <div class='child'><div>
</section>
```

它的父元素实际高度是 100px 或 110px 都可以。主要看怎么父元素的盒模型如何设

置。如以上代码：父元素不加 overflow: hidden，则父元素的实际高度为 100px；如加上 overflow: hidden 父元素高度为 110px，给父元素创建了 BFC，块级格式化上下文。

2.5 BFC（边距重叠解决方案）

2.5.1 BFC 基本概念

BFC: 块级格式化上下文

BFC 基本概念：BFC 是 CSS 布局的一个概念，是一块独立的渲染区域，是一个环境，里面的元素不会影响到外部的元素 。

父子元素和兄弟元素边距重叠，重叠原则取最大值。空元素的边距重叠是取 margin 与 padding 的最大值。

2.5.2 BFC 原理（渲染规则|布局规则）：

（1）内部的 Box 会在垂直方向，从顶部开始一个接着一个地放置；

（2）Box 垂直方向的距离由 margin (外边距)决定，属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠；

（3）每个元素的 margin Box 的左边， 与包含块 border Box 的左边相接触，（对于从左到右的格式化，否则相反）。即使存在浮动也是如此；

（4）BFC 在页面上是一个隔离的独立容器，外面的元素不会影响里面的元素，反之亦然。文字环绕效果，设置 float；

（5）BFC 的区域不会与 float Box 重叠（清浮动）；

（6）计算 BFC 的高度时，浮动元素也参与计算。

2.5.3 CSS 在什么情况下会创建出 BFC（即脱离文档流）

0、根元素，即 HTML 元素（最大的一个 BFC）

1、浮动（ float 的值不为 none ）

2、绝对定位元素（ position 的值为 absolute 或 fixed ）

3、行内块（ display 为 inline-block ）

4、表格单元（ display 为 table、table-cell、table-caption、inline-block 等 HTML 表格相关的属性 ）

5、弹性盒（ display 为 flex 或 inline-flex ）

6、默认值。内容不会被修剪，会呈现在元素框之外（overflow 不为 visible）

2.5.4 BFC 作用（使用场景）

1、自适应两（三）栏布局（避免多列布局由于宽度计算四舍五入而自动换行）

2、避免元素被浮动元素覆盖

3、可以让父元素的高度包含子浮动元素，清除内部浮动（原理：触发父 div 的 BFC 属性，使下面的子 div 都处在父 div 的同一个 BFC 区域之内）

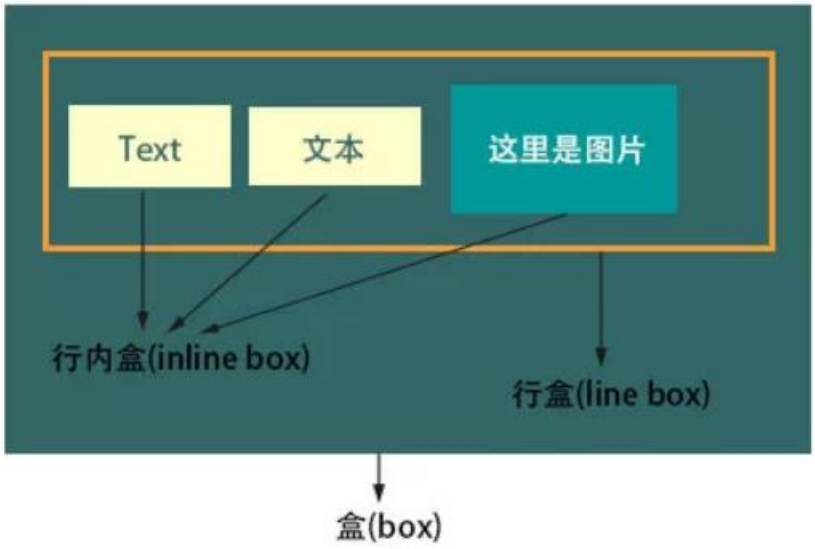
4、去除边距重叠现象，分属于不同的 BFC 时，可以阻止 margin 重叠

2.6 IFC

2.6.1 IFC 基本概念

IFC: 行内格式化上下文

IFC 基本概念：



2.6.2 IFC 原理（渲染规则|布局规则）：

（1）内部的 Box 会在水平方向，从含块的顶部开始一个接着一个地放置；

（2）这些 Box 之间的水平方向的 margin，border 和 padding 都有效；

（3）Box 垂直对齐方式：以它们的底部、顶部对齐，或以它们里面的文本的基线（baseline）对齐（默认，文本与图片对其），例：line-heigth 与 vertical-align。

第一道：就是::before 和:after 中的的双冒号和单冒号有什么区别？并解释一下这两个伪元素的作用。

储备知识：

1.单冒号（:）用于 CSS3 伪类，双冒号（::）用于 CSS3 伪元素。(伪元素由双冒号和伪元素名称组成)

2.双冒号是在当前规范中引入的，用于区分伪类和伪元素。不过浏览器需要同时支持旧的已经存在的伪元素写法，

比如:first-line、:first-letter、:before、:after 等，

3.在新的 CSS3 中引入的伪元素不允许再支持旧的单冒号是写法

想让插入的内容出现在其它内容前，使用::before，否者，使用::after；

在代码顺序上，::after 生成的内容也比::before 生成的内容靠后

回答面试官：

在 css3 中使用单冒号来表示伪类，用双冒号来表示伪元素。但是为了兼容已有的伪元素写法，在一些浏览器中也可以使用单冒号，来表示伪元素。

伪类一般匹配的是元素的一些特殊状态，如 hover、link 等，而伪元素一般匹配的是特殊位置，比如 after、before 等。

第二道，伪类和为元素的区别？

css 引入伪类和伪元素概念是为了格式化文档树以外的信息。也就是说，伪类和伪元素是用来修饰不在文档树中的部分，比如，一句话中的第一个字母，或者是列表中的第一个元素。

伪类用于当已有的元素处于某个状态时，为其添加对应的样式，这个状态是根据用户行

为而动态变化的。比如说，当用户悬停在指定的元素时，我们可以通过:hover 来描述这个元素的状态。

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。它们允许我们为元素的某些部分设置样式。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。

有时你会发现伪元素使用了两个冒号（::）而不是一个冒号（:）。这是 CSS3 的一部分，并尝试区分伪类和伪元素。大多数浏览器都支持这两个值。按照规则应该使用（::）而不是（:），从而区分伪类和伪元素。但是，由于在旧版本的 W3C 规范中并未对此进行特别区分，因此目前绝大多数的浏览器都支持使用这两种方式表示伪元素。

第三题，display 有哪些值并说明他们的作用。

block 块元素类型。默认宽度为父元素宽度，可设置宽高，并独占一行。

none 元素不显示，并从文档流中移除。

inline 行内元素类型。默认宽度为内容宽度，不可设置宽高，同行显示。

inline-block 默认宽度为内容宽度，可以设置宽高，同行显示。

补充三个:

list-item 像块元素类型一样显示，并添加样式列表标记。

table 他会让该元素作为块级表格来显示。

inherit 规定该元素从父元素继承 display 属性的值。

第四题，position 的值 relative 和 absolute 的定位原点是什么？

储备知识:

absolute

成绝对定位的元素，相对于值不为 static 的第一个父元素进行定位，也可以理解为离自己这一级元素最近的一级 position 设置为 absolute 或者 relative 的父元素的左上角为原点。

fixed（老 IE 不支持）

生成绝对定位的元素，相对于浏览器窗口进行定位。

relative

生成相对定位的元素，相对于其元素本身所在正常位置进行定位。

static

默认值。没有定位，元素出现在正常的文档流中。

回答面试官:

relative 定位的元素，是相对于元素本身的正常位置来进行定位的。

absolute 定位的元素，但是相对于它的第一个 position 值不为 static 的祖先元素来进行定位的，这句话我们可以这样来理解，我们首先需要找到绝对定位元素的一个 position 的值不为 static 的祖先元素，然后相对于这个祖先元素来定位。

第五题，就是 li 与 li 之间有看不见的空白间隔是什么原因引起的？有什么解决办法？

原因：浏览器会把 inline 元素间的空白字符（空格、换行、Tab 等）渲染成一个空格。而为了美观。我们通常是一个<li>放在一行，这导致<li>换行后产生换行字符，它变成一个空格，占用了一个字符的宽度。

解决办法:

（1）为<li>设置 float:left。不足：有些容器是不能设置浮动，如左右切换的焦点图等。

（2）将所有<li>写在同一行。不足：代码不美观。

（3）将<ul>内的字符大小直接设为 0，即 font-size:0。不足：<ul>中的其他字符大小也被设为 0, 需要额外重新设定其他字符大小, 且在 Safari 浏览器依然会出现空白间隔。

（4）消除<ul>的字符间隔 letter-spacing:-8px，不足：这也设置了<li>内的字符间隔，因此需要将<li>内的字符间隔设为默认 letter-spacing:normal。

这些解决办法虽然都有不足，但也要视情况而定，选择最合适的办法

第一题，如何让元素居中？

-水平居中：给 div 设置一个宽度，然后添加 margin:0 auto 属性

```
div {
    width: 200px;
    margin: 0 auto;
}
```

-水平居中，利用 text-align:center 实现

```
.container {
    background: rgba(0, 0, 0, 0.5);
    text-align: center;
    font-size: 0;
}

.box {
    display: inline-block;
    width: 500px;
    height: 400px;
    background-color: pink;
}
```

-让绝对定位的 div 居中

```
div {
    position: absolute;
    width: 300px;
    height: 300px;
    margin: auto;
    top: 0;
    left: 0;
    bottom: 0;
    right: 0;
    background-color: pink; /*方便看效果*/
}
```

-水平垂直居中一

```
/*确定容器的宽高宽 500 高 300 的层设置层的外边距 div{*/
position: absolute;/*绝对定位*/
width: 500px;
height: 300px;
top: 50%;
left: 50%;
```

```
margin: -150px00-250px;/*外边距为自身宽高的一半*/
background-color: pink;/*方便看效果*/
}



-水平垂直居中二
/*未知容器的宽高，利用`transform`属性*/
div {
    position: absolute; /*相对定位或绝对定位均可*/
    width: 500px;
    height: 300px;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    background-color: pink; /*方便看效果*/
}
-水平垂直居中三
/*利用 flex 布局实际使用时应考虑兼容性*/
.container {
    display: flex;
    align-items: center; /*垂直居中*/
    justify-content: center; /*水平居中*/
}
.containerdiv {
    width: 100px;
    height: 100px;
    background-color: pink; /*方便看效果*/
}
-水平垂直居中四
/*利用 text-align:center 和 vertical-align:middle 属性*/
.container {
    position: fixed;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;
    background: rgba(0, 0, 0, 0.5);
    text-align: center;
    font-size: 0;
    white-space: nowrap;
    overflow: auto;
}
```

```
.container::after {
    content: '';
    display: inline-block;
    height: 100%;
    vertical-align: middle;
}

.box {
    display: inline-block;
    width: 500px;
    height: 400px;
    background-color: pink;
    white-space: normal;
    vertical-align: middle;
}
```

这道题呢分为两种情况，一种是元素宽高固定，一种是元素宽高不固定

我们先说宽高固定这种情况，它有这么几种方法：

第一，可以利用 margin:0 auto 来实现元素的水平居中。

第二，利用绝对定位，设置四个方向的值都为 0，并将 margin 设置为 auto，由于宽高固定，因此对应方向实现平分，可以实现水平和垂直方向上的居中。

第三，利用绝对定位，先将元素的左上角通过 top:50%和 left:50%定位到页面的中心，然后再通过 margin 负值来调整元素的中心点到页面的中心。

第四，利用绝对定位，先将元素的左上角通过 top:50%和 left:50%定位到页面的中心，然后再通过 translate 来调整元素的中心点到页面的中心。

第五，使用 flex 布局，通过 align-items:center 和 justify-content:center 设置容器的垂直和水平方向上为居中对齐，然后它的子元素也可以实现垂直和水平的居中。

对于元素宽高不固定，上面的四五方法，可以实现元素的垂直和水平居中

代码我就不展示了，想具体了解代码怎么写的，可以下载本期视频的总结文档。

第二题，width:auto 和 width:100%有什么区别？

一般而言

width:100%会使元素 box 的宽度等于父元素的 content box 的宽度。

width:auto 会使元素撑满整个父元素，margin、border、padding、content 区域会自动分配水平空间。

第三题，为什么不建议使用统配符初始化 css 样式？

采用*{padding:0;margin:0;}这样的写法好处是写起来很简单，但是通配符，需要把所有的标签都遍历一遍，当网站较大时，样式比较多，这样写就大大的加强了网站运行的负载，会使网站加载的时候需要很长一段时间，因此一般大型的网站都有分层次的一套初始化样式。

出于性能的考虑，并不是所有标签都会有 padding 和 margin，因此对常见的具有默认 padding 和 margin 的元素初始化即可，并不需使用通配符*来初始化。

第四题，使用 rem 布局的优缺点？

优点：

在屏幕分辨率千差万别的时代，只要将 rem 与屏幕分辨率关联起来就可以实现页面的整体缩放，使得在设备上的展现都统一起来了。而且现在浏览器基本都已经支持 rem 了，兼容性也非常的好。

缺点：

（1）在奇葩的设备上表现效果不太好，比如一些华为的高端机型用 rem 布局会出现错乱。

（2）使用 iframe 引用也会出现问题。

（3）rem 在多屏幕尺寸适配上与当前两大平台的设计哲学不一致。即大屏的出现到底是为了看得又大又清楚，还是为了看的更多的问题。

第五题，transition 和 animation 的区别

transition 关注的是 CSS property 的变化，property 值和时间的关系是一个三次贝塞尔曲线。

animation 作用于元素本身而不是样式属性，使用关键帧的概念，应该说可以实现更自由的动画效果。

第六题，margin:auto 的填充规则？

margin 的'auto'可不是摆设，是具有强烈的计算意味的关键字，用来计算元素对应方向应该获得的剩余间距大小。但是触发 margin:auto 计算有一个前提条件，就是 width 或 height 为 auto 时，元素是具有对应方向的自动填充特性的。

（1）如果一侧定值，一侧 auto，则 auto 为剩余空间大小。

（2）如果两侧均是 auto，则平分剩余空间。

第七题，overflow 的特殊性？

（1）一个设置了 overflow:hidden 声明的元素，假设同时存在 border 属性和 padding 属性，则当子元素内容超出容器宽度高度限制的时候，剪裁的边界是 border box 的内边缘，而非 padding box 的内边缘。

（2）HTML 中有两个标签是默认可以产生滚动条的，一个是根元素<html>，另一个是文本域<textarea>。

（3）滚动条会占用容器的可用宽度或高度。

（4）元素设置了 overflow:hidden 声明，里面内容高度溢出的时候，滚动依然存在，仅仅滚动条不存在！

第八题，如果需要手动写动画，你认为最小时间间隔是多久，为什么？

多数显示器默认频率是 60Hz，即 1 秒刷新 60 次，所以理论上最小间隔为 1/60*1000ms = 16.7ms

第九题，为什么 height:100%会无效？

对于普通文档流中的元素，百分比高度值要想起作用，其父级必须有一个可以生效的高度值。

原因是如果包含块的高度没有显式指定（即高度由内容决定），并且该元素不是绝对定位，则计算值为 auto，因为解释成了 auto，所以无法参与计算。

使用绝对定位的元素会有计算值，即使祖先元素的 height 计算为 auto 也是如此。

第一题：哪些方法可以使元素隐藏？

（1）使用 display:none;隐藏元素，渲染树不会包含该渲染对象，因此该元素不会在页面中占据位置，也不会响应绑定的监听事件。

（2）使用 visibility:hidden;隐藏元素。元素在页面中仍占据空间，但是不会响应绑定的监听事件。

（3）使用 opacity:0;将元素的透明度设置为 0，以此来实现元素的隐藏。元素在页面中仍然占据空间，并且能够响应元素绑定的监听事件。

（4）通过使用绝对定位将元素移除可视区域内，以此来实现元素的隐藏。

（5）通过 z-index 负值，来使其他元素遮盖住该元素，以此来实现隐藏。

（6）通过 transform:scale(0,0)来将元素缩放为 0，以此来实现元素的隐藏。这种方法下，元素仍在页面中占据位置，但是不会响应绑定的监听事件。

第二题：如何实现两栏布局，一栏固定一栏自适应？

以左边宽度固定为 200px 为例

（1）利用浮动，将左边元素宽度设置为 200px，并且设置向左浮动。将右边元素的 margin-left 设置为 200px，宽度设置为 auto（默认为 auto，撑满整个父元素）。

（2）第二种是利用 flex 布局，将左边元素的放大和缩小比例设置为 0，基础大小设置为 200px。将右边的元素的放大比例设置为 1，缩小比例设置为 1，基础大小设置为 auto。

（3）第三种是利用绝对定位布局的方式，将父级元素设置相对定位。左边元素设置为 absolute 定位，并且宽度设置为 200px。将右边元素的 margin-left 的值设置为 200px。

（4）第四种还是利用绝对定位的方式，将父级元素设置为相对定位。左边元素宽度设置为 200px，右边元素设置为绝对定位，左边定位为 200px，其余方向定位为 0。

第三题：什么是 CSS 预处理器/后处理器？

CSS 预处理器定义了一种新的语言，其基本思想是，用一种专门的编程语言，为 CSS 增加了一些编程的特性，将 CSS 作为目标生成文件，然后开发者就只要使用这种语言进行编码工作。通俗的说，CSS 预处理器用一种专门的编程语言，进行 Web 页面样式设计，然后再编译成正常的 CSS 文件。

预处理器例如：LESS、Sass，用来预编译 Sass 或 less，增强了 css 代码的复用性，还有层级、变量、循环、函数等，具有很方便的 UI 组件模块化开发能力，极大的提高工作效率。

CSS 后处理器是对 CSS 进行处理，并最终生成 CSS 的预处理器，它属于广义上的 CSS 预处理器。我们很久以前就在用 CSS 后处理器了，最典型的例子是 CSS 压缩工具（如 clean-css），只不过以前没单独拿出来说过。

后处理器例如：PostCSS，通常被视为在完成的样式表中根据 CSS 规范处理 CSS，让其更有效；目前最常做的是给 CSS 属性添加浏览器私有前缀，实现跨浏览器兼容性的问题。

第四题：移动端的布局用过媒体查询吗？

假设你现在正用一台显示设备来阅读这篇文章，同时你也想把它投影到屏幕上，或者打印出来，而显示设备、屏幕投影和打印等这些媒介都有自己的特点，CSS 就是为文档提供在不同媒介上展示的适配方法

当媒体查询为真时，相关的样式表或样式规则会按照正常的级联规则被应用。当媒体查询返回假，标签上带有媒体查询的样式表仍将被下载（只不过不会被应用）。

包含了一个媒体类型和至少一个使用宽度、高度和颜色等媒体属性来限制样式表范围的表达式。CSS3 加入的媒体查询使得无需修改内容便可以使样式应用于某些特定的设备范围。

第五题：了解 BFC 吗?

块格式化上下文是 Web 页面的可视化 CSS 渲染的一部分，是布局过程中生成块级盒子的区域，也是浮动元素与其他元素的交互限定区域。

通俗来讲，BFC 是一个独立的布局环境，可以理解为一个容器，在这个容器中按照一定规则进行物品摆放，并且不会影响其它环境中的物品。

如果一个元素符合触发 BFC 的条件，则 BFC 中的元素布局不受外部影响。

一般来说根元素是一个 BFC 区域，浮动和绝对定位的元素也会形成 BFC，display 属性的值为 inline-block、flex 这些属性时也会创建 BFC。还有就是元素的 overflow 的值不为 visible 时都会创建 BFC。

第六题：margin 重叠问题的理解

argin 重叠指的是在垂直方向上，两个相邻元素的 margin 发生重叠的情况。

一般来说可以分为四种情形:

第一种是相邻兄弟元素的 marin-bottom 和 margin-top 的值发生重叠。这种情况下我们可以通过设置其中一个元素为 BFC 来解决。

第二种是父元素的 margin-top 和子元素的 margin-top 发生重叠。它们发生重叠是因为它们是相邻的,所以我们可以通过这一点来解决这个问题。我们可以为父元素设置 border-top、padding-top 值来分隔它们，当然我们也可以将父元素设置为 BFC 来解决。

第三种是高度为 auto 的父元素的 margin-bottom 和子元素的 margin-bottom 发生重叠。它们发生重叠一个是因为它们相邻，一个是因为父元素的高度不固定。因此我们可以为父元素设置 border-bottom、padding-bottom 来分隔它们,也可以为父元素设置一个高度，max-height 和 min-height 也能解决这个问题。当然将父元素设置为 BFC 是最简单的方法。

第四种情况，是没有内容的元素，自身的 margin-top 和 margin-bottom 发生的重叠。我们可以通过为其设置 border、padding 或者高度来解决这个问题。

在浏览器中输入 URL 并回车后都发生了什么？

一、解析 URL

URL（Universal Resource Locator）：统一资源定位符。俗称网页地址或者网址。
URL 用来表示某个资源的地址。（通过俗称就能看出来）

URL 主要由以下几个部分组成：

- a. 传输协议
- b. 服务器
- c. 域名
- d. 端口
- e. 虚拟目录

- f.文件名
- g.锚
- h.参数

现在来讨论 URL 解析，当在浏览器中输入 URL 后，浏览器首先对拿到的 URL 进行识别，抽取出域名字段。

二、DNS 解析

DNS 解析（域名解析），DNS 实际上是一个域名和 IP 对应的数据库。

IP 地址往都难以记住，但机器间互相只认 IP 地址，于是人们发明了域名，让域名与 IP 地址之间一一对应，它们之间的转换工作称为域名解析，域名解析需要由专门的域名解析服务器来完成，整个过程是自动进行的。

可以在浏览器中输入 IP 地址浏览网站，也可以输入域名查询网站，虽然得出的内容是一样的，但是调用的过程不一样，输入 IP 地址是直接从主机上调用内容，输入域名是通过域名解析服务器指向对应的主机的 IP 地址，再从主机调用网站的内容。

在进行 DNS 解析时，会经历以下步骤：

1. **查询浏览器缓存**（浏览器会缓存之前拿到的 DNS 2-30 分钟时间），如果没有找到，那么->

2. **检查系统缓存**，检查 hosts 文件，这个文件保存了一些以前访问过的网站的域名和 IP 的数据。它就像是一个本地的数据库。如果找到就可以直接获取目标主机的 IP 地址了。没有找到的话，那么->

3. **检查路由器缓存**，路由器有自己的 DNS 缓存，可能就包括了这在查询的内容；如果没有，那么->

4. **查询 ISP DNS 缓存：**ISP 服务商 DNS 缓存（**本地服务器缓存**）那里可能有相关的内容，如果还不行的话，那么->

5. **递归查询：**从根域名服务器到顶级域名服务器再到极限域名服务器依次搜索对应目标域名的 IP。

通过以上的查找，就可以获取到域名对应的 IP 了。接下来就是向该 IP 地址定位的 HTTP 服务器发起 TCP 连接。

三、浏览器与网站建立 TCP 连接（三次握手）

第一次握手：客户端向服务器端发送请求（SYN=1） 等待服务器确认；

第二次握手：服务器收到请求并确认，回复一个指令（SYN=1，ACK=1）；

第三次握手：客户端收到服务器的回复指令并返回确认（ACK=1）。

通过三次握手，建立了客户端和服务器之间的连接，现在可以请求和传输数据了。

四、请求和传输数据

比如要通过 get 请求访问“http://www.dydh.org/”，通过抓包可以看到：

请求网址（url）：http://www.dydh.org/
请求方法：GET
远程地址：IP
状态码：200 OK
Http 版本： HTTP/1.1
请求头： ...
响应头： ...

响应头中有一个：`Set-Cookie:"PHPSESSID=c882giens9f7d3oglcakhrl994; path=/"`，说明浏览器中没有关于这个网站的 cookie 信息。

当我们下一次访问相同的网站时：

可以看到，请求头中包含了这个 cookie 信息，

```
Cookie:"PHPSESSID=c882giens9f7d3oglcakhrl994; CNZZDATA1253283365=1870471808-1473694656-%7C1473694656"
```

cookie 可以用来保存一些有用的信息：Cookies 如果是首次访问，会提示服务器建立用户缓存信息，如果不是，可以利用 Cookies 对应键值，找到相应缓存，缓存里面存放着用户名，密码和一些用户设置项。

通过这种 GET 请求，和服务器的响应。可以将服务器上的目标文件传输到浏览器进行渲染。

五、浏览器渲染页面

客户端拿到服务器端传输来的文件，找到 HTML 和 MIME 文件，通过 MIME 文件，浏览器知道要用页面渲染引擎来处理 HTML 文件。

1. 浏览器会解析 html 源码，然后创建一个 DOM 树。

在 DOM 树中，每一个 HTML 标签都有一个对应的节点，并且每一个文本也都会有一个对应的文本节点。

2. 浏览器解析 CSS 代码，计算出最终的样式数据，形成 css 对象模型 CSSOM。

首先会忽略非法的 CSS 代码，之后按照浏览器默认设置——用户设置——外链样式——内联样式——HTML 中的 style 样式顺序进行渲染。

3. 利用 DOM 和 CSSOM 构建一个渲染树（rendering tree）。
渲染树和 DOM 树有点像，但是是有区别的。

DOM 树完全和 html 标签一一对应，但是渲染树会忽略掉不需要渲染的元素，比如 head、display:none 的元素等。

而且一大段文本中的每一个行在渲染树中都是独立的一个节点。
渲染树中的每一个节点都存储有对应的 css 属性。

4. 浏览器就根据渲染树直接把页面绘制到屏幕上。

CSS 优化、提高性能的方法有哪些?
加载性能:
css 压缩：将写好的 css 进行打包压缩，可以减少很多的体积。
css 单一样式:当需要下边距和左边距的时候,很多时候选择:margin:top0bottom0;但 margin-bottom:bottom;margin-left:left;执行的效率更高。
减少使用@import,而建议使用 link，因为后者在页面加载时一起加载，前者是等待页面加载完成之后再进行加载。
选择器性能:
关键选择器（keyselector)。选择器的最后面的部分为关键选择器（即用来匹配目标元素的部分)。CSS 选择符是从右到左进行匹配的。当使用后代选择器的时候，浏览器会遍历所有子元素来确定是否是指定的元素等等;
如果规则拥有 ID 选择器作为其关键选择器,则不要为规则增加标签。过滤掉无关的规则(这样样式系统就不会浪费时间去匹配它们了)。
避免使用通配规则，如*{}计算次数惊人！只对需要用到的元素进行选择。
尽量少的去对标签进行选择，而是用 class。
尽量少的去使用后代选择器，降低选择器的权重值。后代选择器的开销是最高的，尽量将选择器的深度降到最低，最高不要超过三层，更多的使用类来关联每一个标签元素。

了解哪些属性是可以通过继承而来的，然后避免对这些属性重复指定规则。
渲染性能：
慎重使用高性能属性：浮动、定位。
尽量减少页面重排、重绘。
去除空规则：{}。空规则的产生原因一般来说是为了预留样式。去除这些空规则无疑能减少 css 文档体积。
属性值为 0 时，不加单位。
属性值为浮动小数 0.**，可以省略小数点之前的 0。
标准化各种浏览器前缀：带浏览器前缀的在前。标准属性在后。
不使用@import 前缀，它会影响 css 的加载速度。
选择器优化嵌套，尽量避免层级过深。
css 雪碧图，同一页面相近部分的小图标，方便使用，减少页面的请求次数，但是同时图片本身会变大，使用时，优劣考虑清楚，再使用。
正确使用 display 的属性，由于 display 的作用，某些样式组合会无效，徒增样式体积的同时也影响解析性能。
不滥用 web 字体。对于中文网站来说 WebFonts 可能很陌生，国外却很流行。webfonts 通常体积庞大，而且一些浏览器在下载 webfonts 时会阻塞页面渲染损伤性能。
可维护性、健壮性：
将具有相同属性的样式抽离出来，整合并通过 class 在页面中进行使用，提高 css 的可维护性。
样式与内容分离：将 css 代码定义到外部 css 中。

如何优化关键渲染路径?
为尽快完成首次渲染，我们需要最大限度减小以下三种可变因素：

（1）关键资源的数量。
（2）关键路径长度。
（3）关键字节的数量。

关键资源是可能阻止网页首次渲染的资源。这些资源越少，浏览器的工作量就越小， 对 CPU 以及其他资源的占用也就越少。

同样，关键路径长度受所有关键资源与其字节大小之间依赖关系图的影响：某些资源 只能在上一资源处理完毕之后才能开始下载，并且资源越大，下载所需的往返次数就越多。最后，浏览器需要下载的关键字节越少，处理内容并让其出现在屏幕上的速度就越快。要减 少字节数，我们可以减少资源数（将它们删除或设为非关键资源），此外还要压缩和优化各 项资源，确保最大限度减小传送大小。

优化关键渲染路径的常规步骤如下：

（1）对关键路径进行分析和特性描述：资源数、字节数、长度。
（2）最大限度减少关键资源的数量：删除它们，延迟它们的下载，将它们标记为异步等。
（3）优化关键字节数以缩短下载时间（往返次数）。
（4）优化其余关键资源的加载顺序：您需要尽早下载所有关键资产，以缩短关键路径长度。

Null 与 undefined 的区别

刚接触 js 的话，可能会认为 `null` 与 `undefined` 很相似，其实不然，本文将列述 `null` 与 `undefined` 之间的不同点与相同点。

null 是什么?

`null` 有两个你必须知道的特征：

- `null` 是空或者不存在的值
- `null` 一定是被赋值的 像这样，我们给变量 `a` 赋值 `null`:

```
let a = null;

console.log(a);
// null
```

undefined 是什么?

`undefined` 通常是一个变量已经被声明，但是没有赋值，例如：

```
let b;

console.log(b);
// undefined
```

也可以给变量直接赋值为 `undefined`:

```
let c = undefined;

console.log(c);
// undefined
```

还有一种情况是从一个 `object` 里读取一个不存在的属性就返回 `undefined`:

```
var d = {};

console.log(d.fake);
// undefined
```

null 与 undefined 的相同点

在 JavaScript 里只有 **6 个否定值**：

- false
- 0(zero)
- ""(empty string)
- null
- undefined
- NaN(Not A Number)

`null` 和 `undefined` 是其中两个。其他所有的值都是真值。

另外 JavaScript 里有 6 个原始值：

- Boolean
- Null
- Undefined
- Number
- String
- Symbol

`null` 和 `undefined` 亦是其中两个。其他所有的值都是对象(objects, functions, arrays 等等)。

有趣的是，用 `typeof` 测试 `null`，会返回 `object`(译者注：这可是个大坑)：

```
1.  let a = null;
2.  let b;
3.
```

```
console.log(typeof a);
// object

console.log(typeof b);
// undefined
```

这个特征是 JavaScript 诞生时就有的，基本上被当做最早的 JavaScript 实现的一个错误。

null !== undefined

综上所述，`null` 与 `undefined` 是不同的，不过有一些共同点，所以 `null` 是严格不等于 `undefined` 的。

```
null !== undefined
```

但是，用不严格比较符的时候 `null` 是等于 `undefined` 的。

```
null == undefined
```

在 JavaScript 里，双等号判断相等时会进行隐式类型转换，所以是不严格的。

实际运用中的不同

上述所说都很好理解，那么 `null` 与 `undefined` 之间到底有啥具体不同？

我们来看下面的代码：

```
let sayHi = (str = 'hi') => {
    console.log(str);
}
```

我们创建了一个函数叫 `sayHi`，有一个参数，且设置默认值为 `hi`，调用是这样：

```
sayHi();
// hi
```

我们也可以传个参数覆盖默认值：

```
sayHi('bye');
// bye
```

但是！ `undefined` 是不会覆盖默认值的，而 `null` 则会。

```
sayHi(undefined);
// hi

sayHi(null);
// null
```

总结

- `null` 是被赋值的，表示啥都没有
- `undefined` 通常是一个变量已经被声明，但是没有赋值
- `null` 和 `undefined` 都是否定值
- `null` 和 `undefined` 都是原始值，不过奇葩的是 `typeofnull=object`
- `null!==undefined` 但是 `null==undefined`

第一题：在 js 中有几种基本数据类型?复杂数据类型?堆栈数据结构?
基本数据类型：Undefined、Null、Boolean、Number、String
值类型：数值、布尔值、null、undefined。
引用类型：对象、数组、函数。
堆栈数据结构：是一种支持后进先出(LIFO)的集合,即后被插入的数据,先被取出!
js 数组中提供了以下几个方法可以让我们很方便实现堆栈：
shift:从数组中把第一个元素删除，并返回这个元素的值。
unshift: 在数组的开头添加一个或更多元素，并返回新的长度
push:在数组的中末尾添加元素，并返回新的长度
pop:从数组中把最后一个元素删除，并返回这个元素的值。

第二题：函数声明的作用提升是什么?变量声明和函数声明的提升有什么区别?

(1) 变量声明提升：变量声明在进入执行上下文就完成了。

只要变量在代码中进行了声明，无论它在哪个位置上进行声明， js 引擎都会将它的声明放在范围作用域的顶部 qqqqqqq；

(2) 函数声明提升：执行代码之前会先读取函数声明，意味着可以把函数声明放在调用它的语句后面。

只要函数在代码中进行了声明，无论它在哪个位置上进行声明， js 引擎都会将它的声明放在范围作用域的顶部；000

(3) 变量 or 函数声明：函数声明会覆盖变量声明，但不会覆盖变量赋值。

同一个名称标识 a，即有变量声明 var a，又有函数声明 function a() {}，不管二者声明的顺序，函数声明会覆盖变量声明，也就是说，此时 a 的值是声明的函数 function a() {}。注意：如果在变量声明的同时初始化 a，或是之后对 a 进行赋值，此时 a 的值是变量的值。eg: var a; var c = 1; a = 1; function a() { return true; } console.log(a);

第三题：如何实现异步编程？

那么，我们主要有以下几个方法：

方法 1：回调函数，优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度耦合 (Coupling)，流程会很混乱，而且每个任务只能指定一个回调函数。

方法 2：事件监听，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以“去耦合”（Decoupling），有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。

方法 3：发布/订阅，性质与“事件监听”类似，但是明显优于后者。

方法 4：Promises 对象，是 CommonJS 工作组提出的一种规范，目的是为异步编程提供统一接口。

简单说，它的思想是，每一个异步任务返回一个 Promise 对象，该对象有一个 then 方法，允许指定回调函数。

第四题：什么是事件流?事件捕获？事件冒泡?

事件流:从页面中接收事件的顺序。也就是说当一个事件产生时,这个事件的传播过程，就是事件流。

IE 中的事件流叫事件冒泡；事件冒泡：事件开始时由最具体的元素接收，然后逐级向上传播到较为不具体的节点（文档）。对于 html 来说，就是当一个元素产生了一个事件，它会把这个事件传递给它的父元素，父元素接收到了之后，还要继续传递给它的上一级元素，就这样一直传播到 document 对象（亲测现在的浏览器到 window 对象，只有 IE8 及下不这样

事件捕获是不太具体的元素应该更早接受到事件,而最具体的节点应该最后接收到事件。他们的用意是在事件到达目标之前就捕获它;也就是跟冒泡的过程正好相反,以 html 的 click 事件为例，document 对象（DOM 级规范要求从 document 开始传播，但是现在的浏览器是从 window 对象开始的）最先接收到 click 事件的然后事件沿着 DOM 树依次向下传播，一直传播到事件的实际目标;

第五题：自执行函数是什么?用于什么场景？好处?

自执行函数:1、声明一个匿名函数 2、马上调用这个匿名函数。

作用：创建一个独立的作用域。

好处：防止变量弥散到全局，以免各种 js 库冲突。隔离作用域避免污染，或者截断作用域链，避免闭包造成引用变量无法释放。利用立即执行特性，返回需要的业务函数或对象，

避免每次通过条件判断来处理

场景：一般用于框架、插件等场景

最后一道题，就是：简述一下你理解的面向对象是什么？

万物皆对象，把一个对象抽象成类,具体上就是把一个对象的静态特征和动态特征抽象成属性和方法,也就是把一类事物的算法和数据结构封装在一个类之中,程序就是多个对象和互相之间的通信组成的.

面向对象具有封装性,继承性,多态性。

封装:隐蔽了对象内部不需要暴露的细节,使得内部细节的变动跟外界脱离,只依靠接口进行通信.封装性降低了编程的复杂性.

通过继承,使得新建一个类变得容易,一个类从派生类那里获得其非私有的方法和公用属性的繁琐工作交给了编译器.

而继承和实现接口以及运行时的类型绑定机制 所产生的多态,使得不同的类所产生的对象能够对相同的消息作出不同的反应,极大地提高了代码的通用性.

总之,面向对象的特性提高了大型程序的重用性和可维护性.

第一题：什么是栈和堆？

堆和栈的概念存在于数据结构中和操作系统内存中。

在数据结构中，栈中数据的存取方式为先进后出。而堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。完全二叉树是堆的一种实现方式。

在操作系统中，内存被分为栈区和堆区。

栈区内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆区内存一般由程序员分配释放，若程序员不释放，程序结束时可能由垃圾回收机制回收。

第二题：介绍 js 有哪些内置对象？

js 中的内置对象主要指的是在程序执行前存在全局作用域里的由 js 定义的一些全局属性、函数和用来实例化其他对象的构造函数对象。一般我们经常用到的如全局变量值 NaN、undefined, 全局函数如 parseInt()、parseFloat() 用来实例化对象的构造函数如 Date、Object 等，还有提供数学计算的单体内置对象如 Math 对象。

第三题：什么是原型、原型链，有什么特点？

在 js 中我们是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性值，这个属性值是一个对象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当我们使用构造函数新建一个对象后，在这个对象的内部将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说我们是不应该能够获取到这个值的，但是现在浏览器中都实现了 __proto__ 属性来让我们访问这个属性，但是我们最好不要使用这个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，我们可以通过这个方法来获取对象的原型。

当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就是我们新建的对象

为什么能够使用 toString() 等方法的原因。

特点：

JavaScript 对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

第四题： Javascript 的作用域链？

作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，我们可以访问到外层环境的变量和函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当我们查找一个变量时，如果当前执行环境中没有找到，我们可以沿着作用域链向后查找。

第五题：谈谈 This 对象的理解。

首先，我们需要知道的是：this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this 的指向可以通过四种调用模式来判断

1.第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。

2.第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。

3.第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。

4.第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

第六题：什么是闭包，为什么要用它

闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，创建的函数可以访问到当前函数的局部变量。

闭包有两个常用的用途。

闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，我们可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。

函数的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

其实闭包的本质就是作用域链的一个特殊的应用，只要了解了作用域链的创建过程，就能够理解闭包的实现原理。

第七题：Ajax 是什么？如何创建一个 Ajax？

ajax 指的是通过 JavaScript 的异步通信，从服务器获取 XML 文档从中提取数据，再更

新当前网页的对应部分，而不用刷新整个网页。

具体来说，AJAX 包括以下几个步骤。

1.创建 XMLHttpRequest 对象，也就是创建一个异步调用对象

2.创建一个新的 HTTP 请求，并指定该 HTTP 请求的方法、URL 及验证信息

3.设置响应 HTTP 请求状态变化的函数

4.发送 HTTP 请求

5.获取异步调用返回的数据

6.使用 JavaScript 和 DOM 实现局部刷新

第一题：Symbol 类型的注意点?

1.Symbol 函数前不能使用 new 命令，否则会报错。

2.Symbol 函数可以接收一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。

3.Symbol 作为属性名，该属性不会出现在 for...in、for...of 循环中，也不会被 Object.keys()、Object.getOwnPropertyNames()、JSON.stringify() 返回。

4.Object.getOwnPropertySymbols 方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。

5.Symbol.for 接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。

6.Symbol.keyFor 方法返回一个已登记的 Symbol 类型值的 key。

第二题： js 拖拽功能的实现

一个元素的拖拽过程，我们可以分为三个步骤，第一步是鼠标按下目标元素，第二步是鼠标保持按下的状态移动鼠标，第三步是鼠标抬起，拖拽过程结束。

这三步分别对应了三个事件，mousedown 事件，mousemove 事件和 mouseup 事件。只有在鼠标按下的状态移动鼠标我们才会执行拖拽事件，因此我们需要在 mousedown 事件中设置一个状态来标识鼠标已经按下，然后在 mouseup 事件中再取消这个状态。在 mousedown 事件中我们首先应该判断，目标元素是否为拖拽元素，如果是拖拽元素，我们就设置状态并且保存这个时候鼠标的位置。然后在 mousemove 事件中，我们通过判断鼠标现在的位置和以前位置的相对移动，来确定拖拽元素在移动中的坐标。最后 mouseup 事件触发后，清除状态，结束拖拽事件。

第三题：图片的懒加载和预加载

懒加载也叫延迟加载，指的是在长网页中延迟加载图片的时机，当用户需要访问时，再去加载，这样可以提高网站的首屏加载速度，提升用户的体验，并且可以减少服务器的压力。它适用于图片很多，页面很长的电商网站的场景。懒加载的实现原理是，将页面上图片的 src 属性设置为空字符串，将图片的真实路径保存在一个自定义属性中，当页面滚动的时候，进行判断，如果图片进入页面可视区域内，则从自定义属性中取出真实路径赋值给图片的 src 属性，以此来实现图片的延迟加载。

预加载指的是将所需的资源提前请求加载到本地，这样后面在需要用到时就直接从缓存取资源。通过预加载能够减少用户的等待时间，提高用户的体验。我了解的预加载的最常用的方式是使用 js 中的 image 对象，通过为 image 对象来设置 scr 属性，来实现图片的预

加载。

这两种方式都是提高网页性能的方式，两者主要区别是一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

第四题：异步编程的实现方式？

js 中的异步机制可以分为以下几种：

第一种最常见的是使用回调函数的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

第二种是 Promise 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

第三种是使用 generator 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部我们还可以将执行权转移回来。当我们遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕的时候我们再将执行权给转移回来。因此我们在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式我们需要考虑的问题是何时将函数的控制权转移回来，因此我们需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

第四种是使用 async 函数的形式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此我们可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

第五题：　js 延迟加载的方式有哪些？

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

我知道的有这么几种方式是：

第一种方式是我们一般采用的是将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。

第二种方式是给 js 脚本添加 defer 属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。

第三种方式是给 js 脚本添加 async 属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js 脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。

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

第六题：哪些操作会造成内存泄漏？

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

第二种情况是我们设置了 setInterval 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。

第三种情况是我们获取一个 DOM 元素的引用，而后面这个元素被删除，由于我们一直保留了对这个元素的引用，所以它也无法被回收。

第四种情况是不合理的使用闭包，从而导致某些变量一直被留在内存当中。

Var 和 let、const 的区别：

第一点区别就是：变量提升问题

var 声明的变量存在变量提升，而 let 与 const 声明的变量不存在变量提升，但存在暂时性死区

什么意思呢，就是在预编译阶段，js 引擎扫描代码时，遇到变量声明，会把 var 声明提到作用域的顶端，而 let 和 const 声明，则会被放在暂时性死区中。访问暂时性死区中的变量，会触发运行时错误，只有执行变量声明语句后，变量才会从暂时性死区中移除，才可正常访问。

第二点区别就是：重复定义问题

var 声明的变量可以重复定义

let 与 const 声明的变量，在同一作用域下不可重复声明，否则报错

00.

第三点区别就是：全局变量问题

也就是说，var 声明的变量可能会覆盖全局变量

而 let 与 const 声明的变量只可能会遮住全局变量，不会覆盖全局变量，也就是不会破坏全局作用域

第四点区别是：作用域的问题

在 ES6 之前，只存在全局作用域、 函数作用域：

使用 var 声明变量，可能会出现内部变量覆盖外部变量的情况，并且循环变量会泄漏为全局变量

还有在 ES5 中规定，函数声明只能在全局作用域或者函数作用域中进行，但浏览器为了兼容旧代码，不会报错

ES6 中，除了全局作用域、函数作用域，引入了 块级作用域的概念，如我们常见的{} ,if{}for(){}都会形成块级作用域：

在 严格模式下，函数声明语句的行为类似于 let，在块级作用域之外不可引用，并且会被提升至块级作用域的顶部。如果是 let 定义的函数表达式，不会被提示。

在 非严格模式下，为了兼容旧代码，最终的函数声明语句的行为类似 var，声明会被提升到外围函数或全局作用域的顶部，而不是代码块的顶部。

我们在使用 let、 const 声明变量时，循环变量不会泄漏成为全局变量。在 for 循环中，设置循环变量与循环体内部是两个单独的作用域。

还可以在块级作用域中进行 函数声明，需要注意的是，块级作用域必须存在

let 与 const 的主要区别。

主要是，const 声明的是常量，在声明后，常量指向的 内存地址中保存的数据不可改动。所以：

对于简单类型的数据，如数值、字符串等，声明后不可在修改；

对于引用类型的数据，只要保持指针不变即可。

第一题：说一下 js 的垃圾回收机制

定义：指一块被分配的内存既不能使用，又不能回收，直到浏览器进程结束。像 C 这样的编程语言，具有低级内存管理原语，开发人员使用这些原语显式地对操作系统的内存进行分配和释放。而 JavaScript 在创建对象(对象、字符串等)时会为它们分配内存，不再使用对时会“自动”释放内存，这个过程称为垃圾收集。

内存生命周期中的每一个阶段:

分配内存 — 内存是由操作系统分配的，它允许您的程序使用它。在低级语言(例如 C 语言)中，这是一个开发人员需要自己处理的显式执行的操作。然而，在高级语言中，系统会自动为你分配内在。

使用内存 — 这是程序实际使用之前分配的内存，在代码中使用分配的变量时，就会发生读和写操作。

释放内存 — 释放所有不再使用的内存,使之成为自由内存,并可以被重利用。与分配内存操作一样,这一操作在低级语言中也是需要显式地执行。

四种常见的内存泄漏：全局变量，未清除的定时器，闭包，以及 dom 的引用

全局变量 不用 var 声明的变量，相当于挂载到 window 对象上。如：b=1; 解决：使用严格模式

遗忘的定时器和回调函数

闭包

没有清理的 DOM 元素引用

第二题：声明函数作用提升？声明变量和声明函数的提升有什么区别?

变量声明提升：变量申明在进入执行上下文就完成了。只要变量在代码中进行了声明，无论它在哪个位置上进行声明，js 引擎都会将它的声明放在范围作用域的顶部

函数声明提升：执行代码之前会先读取函数声明，意味着可以把函数申明放在调用它的语句后面。只要函数在代码中进行了声明，无论它在哪个位置上进行声明，js 引擎都会将它的声明放在范围作用域的顶部

变量 or 函数声明：函数声明会覆盖变量声明，但不会覆盖变量赋值。同一个名称标识 a，即有变量声明 var a，又有函数声明 function a(){}，不管二者声明顺序，函数声明会覆盖变量声明，也就是说，此时 a 的值是声明的函数 function a() {} 注意：如果在变量声明的同时初始化 a，或是之后对 a 进行赋值，此时 a 的值变量的值

第三题：new 操作具体干了什么?

（1）创建一个空对象，并且 this 变量引用该对象，同时还继承了该函数的原型。

（2）属性和方法被加入到 this 引用的对象中。

（3）新创建的对象由 this 所引用，并且最后隐式的返回 this 。

第四题：开发中常用的几种 Content-Type ？

（1）application/x-www-form-urlencoded

浏览器的原生 form 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。该种方式提交的数据放在 body 里面，数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL

转码。

（2）multipart/form-data

该种方式也是一个常见的 POST 提交方式，通常表单上传文件时使用该种方式。

（3）application/json

告诉服务器消息主体是序列化后的 JSON 字符串。

（4）text/xml

该种方式主要用来提交 XML 格式的数据。

第一题：对于 MVVM 的理解?

MVVM 是 Model-View-ViewModel 的缩写。

Model 代表数据模型，也可以在 Model 中定义数据修改和操作的业务逻辑。

View 代表 UI 组件，它负责将数据模型转化成 UI 展现出来。

ViewModel 监听数据模型的改变和控制视图行为、处理用户交互，简单理解就是一个同步 View 和 Model 的对象，连接 Model 和 View。

在 MVVM 架构下，View 和 Model 之间并没有直接的联系，而是通过 ViewModel 进行交互，Model 和 ViewModel 之间的交互是双向的，因此 View 数据的变化会同步到 Model 中，而 Model 数据的变化也会立即反应到 View 上。

ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而 View 和 Model 之间的同步工作完全是自动的，无需人为干涉，因此开发者只需关注业务逻辑，不需要手动操作 DOM，不需要关注数据状态的同步问题，复杂的数据状态维护完全由 MVVM 来统一管理。简单来说可以参照这张图片~

第二题：VUE 的生命周期

beforeCreate（创建前） 在数据观测和初始化事件还未开始。

created（创建后） 完成数据观测，属性和方法的运算，初始化事件，$el 属性还没有显示出来。

beforeMount（载入前） 在挂载开始之前被调用，相关的 render 函数首次被调用。实例已完成以下的配置：编译模板，data 里面的数据和模板生成 html。注意此时还没有挂载 html 到页面上。

mounted（载入后） 在 el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的 html 内容替换 el 属性指向的 DOM 对象。完成模板中的 html 渲染到 html 页面中。此过程中进行 ajax 交互。

beforeUpdate（更新前） 在数据更新之前调用，发生在虚拟 DOM 重新渲染和打补丁之前。可以在该钩子中进一步地更改状态，不会触发附加的重复渲染过程。

updated（更新后） 在由于数据更改导致的虚拟 DOM 重新渲染和打补丁之后调用。调用时，组件 DOM 已经更新，所以可以执行依赖于 DOM 的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。

beforeDestroy（销毁前） 在实例销毁之前调用。实例仍然完全可用。

destroyed（销毁后） 在实例销毁之后调用。调用后，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。

1、什么是 vue 生命周期?

答：Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、编译模板、挂载 Dom→渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生命周期。

2、vue 生命周期的作用是什么?

答：它的生命周期中有多个事件钩子，让我们在控制整个 Vue 实例的过程时更容易形成好的逻辑。

3、vue 生命周期总共有几个阶段?

答：它可以总共分为 8 个阶段：创建前/后，载入前/后,更新前/后,销毁前/销毁后。

4、第一次页面加载会触发哪几个钩子?

答：会触发 下面这几个 beforeCreate, created, beforeMount, mounted 。

5、DOM 渲染在 哪个周期中就已经完成?

答：DOM 渲染在 mounted 中就已经完成了。

第三题：Vue 实现数据双向绑定的原理，这里我们暂且说的是 2.0 版本的原理：Object.defineProperty（）

vue 实现数据双向绑定主要是：采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty（）来劫持各个属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应监听回调。

当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用 Object.defineProperty 将它们转为 getter/setter。

用户看不到 getter/setter，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。

vue 的数据双向绑定 将 MVVM 作为数据绑定的入口，整合 Observer，Compile 和 Watcher 三者，通过 Observer 来监听自己的 model 的数据变化，通过 Compile 来解析编译模板指令（vue 中是用来解析 {{}}），最终利用 watcher 搭起 observer 和 Compile 之间的通信桥梁，达到数据变化 —>视图更新；视图交互变化（input）—>数据 model 变更双向绑定效果。

这里有一个简单的双向绑定代码，供大家参考

```
<body>
    <div id="app">
        <input type="text" id="txt">
        <p id="show"></p>
    </div>
</body>
<script type="text/javascript">
    var obj = {}
    Object.defineProperty(obj, 'txt', {
        get: function () {
            return obj
        },
        set: function (newValue) {
            document.getElementById('txt').value = newValue
            document.getElementById('show').innerHTML = newValue
        }
    })
    document.addEventListener('keyup', function (e) {
        obj.txt = e.target.value
```

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

第一题：说一说 VUE3.0 响应式的理解

Vue3.0 改用 Proxy 替代 Object.defineProperty。因为 Proxy 可以直接监听对象和数组的变化，并且有多达 13 种拦截方法。而且作为新标准将受到浏览器厂商重点持续的性能优化。

Proxy 只会代理对象的第一层，那么 Vue3 又是怎样处理这个问题的呢？

判断当前 Reflect.get 的返回值是否为 Object，如果是则再通过 reactive 方法做代理，这样就实现了深度观测。

监测数组的时候可能触发多次 get/set，那么如何防止触发多次呢？

我们可以判断 key 是否为当前被代理对象 target 自身属性，也可以判断旧值与新值是否相等，只有满足以上两个条件之一时，才有可能执行 trigger。

第二题：Proxy 与 Object.defineProperty 优劣对比

Proxy 的优势如下:

1）Proxy 可以直接监听对象而非属性；

2）Proxy 可以直接监听数组的变化；

3）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；

4）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；

5）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，而 Proxy 的存在浏览器兼容性问题，因此 Vue 的作者才声明需要等到下个大版本(3.0)才能用 Proxy 重写。

第三题：对 SPA 单页面的理解，优缺点是什么？

SPA(single-page application)仅在 Web 页面初始化时加载相应的 HTML、JavaScript 和 CSS。一旦页面加载完成，SPA 不会因为用户的操作而进行页面的重新加载或跳转；取而代之的是利用路由机制实现 HTML 内容的变换，UI 与用户的交互，避免页面的重新加载。

优点：

1）用户体验好、快，内容的改变不需要重新加载整个页面，避免了不必要的跳转和重复渲染；

2）SPA 相对对服务器压力小；

3）前后端职责分离，架构清晰，前端进行交互逻辑，后端负责数据处理；

缺点：

1）首屏（初次）加载慢：为实现单页 Web 应用功能及显示效果，需要在加载页面的时候将 JavaScript、CSS 统一加载，部分页面按需加载；

2）不利于 SEO：由于所有的内容都在一个页面中动态替换显示，所以在 SEO 上其有着天然的弱势。

第四题：Vue.js 的优点是什么

Vue.js 性能好，非常容易优化。

Vue.js 体积小，包含了 Vuex + Vue Router 的 Vue 项目 (gzip 之后 30kB) 。

Vue.js 更加灵活，Vue 官方提供了构建工具来协助你构建项目，但它并不限制你去如何组织你的应用代码。

Vue.js 容易上手。要学习 Vue，你只需要有良好的 HTML 和 JavaScript 基础，API 也比较少。

Vue.js 生态系统丰富，很多开源组件和 UI 框架。

第五题：Vue.use 是干什么的？原理是什么？

vue.use 是用来使用插件的，我们可以在插件中扩展全局组件、指令、原型方法等。

1、检查插件是否注册，若已注册，则直接跳出；

2、处理入参，将第一个参数之后的参数归集，并在首部塞入 this 上下文；

3、执行注册方法，调用定义好的 install 方法，传入处理的参数，若没有 install 方法并且插件本身为 function 则直接进行注册；

第一题：VUE 中 key 值的作用是什么？

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素。key 的作用主要是为了高效的更新虚拟 DOM。

第二题：keep-alive 的作用是什么？

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。include 和 exclude 都支持”,”分割字符串、数组或正则表达式，include 表示只有名称匹配的组件会被缓存。

第三题：Vue 中的组件的 data 为什么是一个函数？

每次使用组件时都会对组件进行实例化操作，并且调用 data 函数返回一个对象作为组件的数据源。这样可以保证多个组件间数据互不影响。

如果 data 是对象的话，对象属于引用类型，会影响到所有的实例。所以为了保证组件不同的实例之间 data 不冲突，data 必须是一个函数。

第四题：Vue 为什么需要虚拟 DOM？　虚拟 DOM 的优劣如何？

Virtual DOM 就是用 js 对象来描述真实 DOM，是对真实 DOM 的抽象，由于直接操作 DOM 性能低但是 js 层的操作效率高，可以将 DOM 操作转化成对象操作，最终通过 diff 算法比对差异进行更新 DOM (减少了对真实 DOM 的操作)。虚拟 DOM 不依赖真实平台环境从而也可以实现跨平台。

第五题：v-if 与 v-for 的优先级

1、v-for 优先于 v-if 被解析

2、如果同时出现，每次渲染都会先执行循环再判断条件，无论如何循环都不可避免，浪费了性能

3、要避免出现这种情况，则在外层嵌套 template，在这一层进行 v-if 判断，然后在内部进行 v-for 循环

4、如果条件出现在循环内部，可通过计算属性提前过滤掉那些不需要显示的项

第六题：v-if 与 v-show 的区别

v-if 是真正的条件渲染，直到条件第一次变为真时，才会开始渲染。

v-show 不管初始条件是什么会渲染，并且只是简单地基于 CSS 的 “display” 属性进行切换。

注意：v-if 适用于不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。

第七题：VUE 常用指令

v-model、v-class、v-for、v-if、v-show、v-on。

父组件向子组件传值

① 子组件在 props 中创建一个属性，用以接收父组件传过来的数据；

② 父组件中注册子组件。通过属性绑定（v-bind:）的形式，把需要传递给子组件的数据传递到子组件的内部，供子组件使用；

③ 在子组件标签中添加子组件 props 中创建的属性；

④ 把需要传给子组件的值赋给该属性。

我们需要注意的是：

① prop 是子组件用来接受父组件传递过来的数据的一个自定义属性；

②父组件的数据需要通过 props 把数据传给子组件，子组件需要显式地用 props 选项声明 “prop”；

③prop 是单向绑定的：当父组件的属性变化时，将传导给子组件，但是不会反过来；

④ props 中的数据都是只读的，无法进行重新赋值。

```
<!DOCTYPE html>
<html lang="en">

<head>
```

```
    <meta charset="UTF-8">
    <title>父组件向子组件传值</title>
    <script src="js/vue.js"></script>
</head>

<body>
    <div id="app">
        <!-- 父组件中注册子组件 -->
        <!-- 在子组件标签中添加子组件 props 中创建的属性，把需要传给子组件的值
赋给该属性 -->
        <mycom :parent-msg='pmsg' :content='hello'></mycom>
    </div>
    <script>
        var vm = new Vue({
            el: '#app',
            data: {
                pmsg: '父组件中的内容',
                hello: '我是父组件传过来的'
            },
            components: {
                mycom: {
                    data: function () {
                        return {
                            msg: '我是子组件本身的数据'
                        }
                    },
                    // 子组件在 props 中创建一个属性，用以接收父组件传过来的
值
                    props: ['parentMsg', 'content'],
                    template: '<h3>{{msg + "---" + parentMsg + "----
" + content}}</h3>'
                }
            }
        });
    </script>
</body>

</html>
```

子组件向父组件传值

① 子组件需要以某种方式例如点击事件的方法来触发一个自定义事件;

② 将需要传递的值作为$emit 的第二个参数，该值将作为实参传递给响应自定义事件的方法;

③ 在父组件中注册子组件，在子组件标签上监听该自定义事件，并添加一个响应该事件的处理方法。

我们需要注意的是：

① 父组件是使用 props 传递数据给子组件，但如果子组件要把数据传递回去，就需要使用自定义事件！

② 我们可以使用 v-on 绑定自定义事件，每个 Vue 实例都实现了事件接口(Events interface)，即：

使用 $on(eventName) 监听事件；

使用 $emit(eventName) 触发事件。

③ 父组件可以在使用子组件的地方直接用 v-on 来监听子组件触发的事件。

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>父组件向子组件传值</title>
    <script src="js/vue.js"></script>
</head>

<body>
    <div id="app">
        <h3>父组件</h3>
        <child @receive="handleEvent"></child>
    </div>
    <script>
        Vue.component('child', {
            template: `
                <div>
                    <h3>子组件</h3>
                    <button @click="sendMessage()">发送</button>
                </div>
            `,
            data() {
                return {
                }
            },
            methods: {
                sendMessage() { // 按钮的点击事件
                    this.$emit("receive","父组件您好，我是子组件！") // 调用父组件传递过来的方法，并且把数据传递出去
                }
            }
        })
```

```
        var vm = new Vue({
            el: '#app',
            methods: {
                // 定义在子组件中通过 this.$emit() 调用的方法
                handleEvent(val) {
                    console.log("父组件收到的消息是：",val)
                }
            }
        })
    </script>
</body>

</html>
```

在组件传值过程中，无论是父传子、还是子传父，它们都有一个共同点就是有一个中间介质。父传子的介质是 props 中的属性，子传父的介质是自定义事件。

父子组件的关系可以总结为 prop 向下传递，事件向上传递。父组件通过 prop 给子组件下发数据，子组件通过事件给父组件发送信息。

父组件通过 v-bind:绑定参数传给子组件，子组件通过 props 接受这个参数。
在组件的最底层开始写事件，由最底层组件逐步向上$emit 事件流，并携带相应参数，最后在父组件内完成总的数据处理。

兄弟组件之间如何传值

1、在 main.js 同级目录下新建 new.js 文件

2、在组件 a 中传出值

先引入 new.js 文件，再通过$emit 传值

3、在同级 b 组件中通过$on 接收

1、

```
import Vue from 'vue'
export default new Vue()
```

2、

```
<template>
  <div @click="onfocus"></div>
</template>

<script>
    import New from '@/new.js'

  export default{
    methods:{
```

```
            onfocus:function(fromid){
                New.$emit('getisshow',{
                    show:true
                })
            }
        }
    }
</script>
```

3、

```
<script>       import New from '@/new.js'

    export default{
        created(){
            New.$on('getisshow',data => {
                console.log(data)     //{show:true}
            })
        }
    }
    </script>
```

组件之间传值，不仅仅只有这些方法，想了解更多传值方法，可持续关注后期视频!

vue-router 是什么

这里的路由并不是指我们平时所说的硬件路由器，这里的路由就是 SPA（单页应用）的路径管理器。再通俗的说，vue-router 就是 WebApp 的链接路径管理系统。

vue-router 是 Vue.js 官方的路由插件，它和 vue.js 是深度集成的，适合用于构建单页面应用。vue 的单页面应用是基于路由和组件的，路由用于设定访问路径，并将路径和组件映射起来。传统的页面应用，是用一些超链接来实现页面切换和跳转的。在 vue-router 单页面应用中，则是路径之间的切换，也就是组件的切换。路由模块的本质 就是建立起 url 和页面之间的映射关系。

至于我们为啥不能用 a 标签，这是因为用 Vue 做的都是单页应用（当你的项目准备打包时，运行 npm run build 时，就会生成 dist 文件夹，这里面只有静态资源和一个 index.html 页面），所以你写的标签是不起作用的，你必须使用 vue-router 来进行管理。

vue-router 实现原理

SPA(single page application):单一页面应用程序，只有一个完整的页面；它在加载页面时，不会加载整个页面，而是只更新某个指定的容器中内容。单页面应用(SPA)的核心之一是：更新视图而不重新请求页面;vue-router 在实现单页面前端路由时，提供了两种方式：Hash 模式和 History 模式；根据 mode 参数来决定采用哪一种方式。

1、Hash 模式：

vue-router 默认 hash 模式 —— 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。hash（#）是 URL 的锚点，代表的是网页中的一个位

置，单单改变#后的部分，浏览器只会滚动到相应位置，不会重新加载网页，也就是说 hash 出现在 URL 中，但不会被包含在 http 请求中，对后端完全没有影响，因此改变 hash 不会重新加载页面；同时每一次改变#后的部分，都会在浏览器的访问历史中增加一个记录，使用"后退"按钮，就可以回到上一个位置；所以说 Hash 模式通过锚点值的改变，根据不同的值，渲染指定 DOM 位置的不同数据。hash 模式的原理是 onhashchange 事件(监测 hash 值变化)，可以在 window 对象上监听这个事件。

2、History 模式：

由于 hash 模式会在 url 中自带#，如果不想要很丑的 hash，我们可以用路由的 history 模式，只需要在配置路由规则时，加入"mode: 'history'",这种模式充分利用了 html5 history interface 中新增的 pushState() 和 replaceState() 方法。这两个方法应用于浏览器记录栈，在当前已有的 back、forward、go 基础之上，它们提供了对历史记录修改的功能。只是当它们执行修改时，虽然改变了当前的 URL ，但浏览器不会立即向后端发送请求。

```
//main.js 文件中
const router = new VueRouter({
    mode:'history',
    routes:[...]
})
```

当你使用 history 模式时，URL 就像正常的 url，例如 http://yoursite.com/user/id，比较好看!

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 http://oursite.com/user/id 就会返回 404，这就不好看了。

所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

```
export const routes = [
 {path: "/", name: "homeLink", component:Home}
 {path: "/register", name: "registerLink", component: Register},
 {path: "/login", name: "loginLink", component: Login},
 {path: "*", redirect: "/"}
]
```

此处就设置如果URL输入错误或者是URL 匹配不到任何静态资源，就自动跳到到Home页面

3、使用路由模块来实现页面跳转的方式、

方式 1：直接修改地址栏

方式 2：this.$router.push('路由地址')

方式 3：　<router-link to="路由地址"></router-link>

一、vue-router 使用方式

1:下载 `npm i vue-router-S`

2:在 main.js 中引入 `importVueRouterfrom'vue-router';`

3:安装插件 `Vue.use(VueRouter);`

4:创建路由对象并配置路由规则

```
let router = new VueRouter({routes:[{path:'/home',component:Home}]});
```

5:将其路由对象传递给 Vue 的实例，options 中加入 `router:router`

6:在 app.vue 中留坑 `<router-view></router-view>`

具体实现请看如下代码：

```
//main.js 文件中引入
import Vue from 'vue';
import VueRouter from 'vue-router';
//主体
import App from './components/app.vue';
import Home from './components/home.vue'
//安装插件
Vue.use(VueRouter); //挂载属性
//创建路由对象并配置路由规则
let router = new VueRouter({
    routes: [
        //一个个对象
        { path: '/home', component: Home }
    ]
});
//new Vue 启动
new Vue({
    el: '#app',
    //让 vue 知道我们的路由规则
    router: router, //可以简写 router
    render: c => c(App),
})
```

最后记得在在 app.vue 中“留坑”

```
//app.vue 中
<template>
    <div>
        <!-- 留坑，非常重要 -->
        <router-view></router-view>
    </div>
</template>
<script>
    export default {
        data(){
            return {}
        }
    }
</script>
```

二、vue-router 参数传递

声明式的导航 `<router-link:to="...">`和编程式的导航 `router.push(...)`都可以传参，本文主要介绍前者的传参方法，同样的规则也适用于编程式的导航。

1、用 name 传递参数

在路由文件 src/router/index.js 里配置 name 属性

```
routes: [
    {
      path: '/',
      name: 'Hello',
      component: Hello
    }
]
```

模板里(src/App.vue)用 `$route.name` 来接收 比如：

```
<p>{{ $route.name}}</p>
```

2、通过 `<router-link>` 标签中的 to 传参

这种传参方法的基本语法：

```
<router-link :to="{name:xxx,params:{key:value}}">valueString</router-link>
```

比如先在 src/App.vue 文件中

```
<router-link :to="{name:'hi1',params:{username:'jspang',id:'555'}}">Hi 页面 1</router-link>
```

然后把 src/router/index.js 文件里给 hi1 配置的路由起个 name,就叫 hi1.

```
{path:'/hi1',name:'hi1',component:Hi1}
```

最后在模板里(src/components/Hi1.vue)用 `$route.params.username` 进行接收.

```
{{$route.params.username}}-{{$route.params.id}}
```

3、利用 url 传递参数----在配置文件里以冒号的形式设置参数。

我们在/src/router/index.js 文件里配置路由

```
{
    path:'/params/:newsId/:newsTitle',
    component:Params
}
```

我们需要传递参数是新闻 ID（newsId）和新闻标题（newsTitle）.所以我们在路由配置文件里制定了这两个值。

在 src/components 目录下建立我们 params.vue 组件，也可以说是页面。我们在页面里输出了 url 传递的的新闻 ID 和新闻标题。

```
<template>
```

```
    <div>
        <h2>{{ msg }}</h2>
        <p>新闻 ID：{{ $route.params.newsId}}</p>
        <p>新闻标题：{{ $route.params.newsTitle}}</p>
    </div>
</template>
<script>
export default {
  name: 'params',
  data () {
    return {
      msg: 'params page'
    }
  }
}
</script>
```

在 App.vue 文件里加入我们的 `<router-view>`标签。这时候我们可以直接利用 url 传值了

```
<router-link to="/params/198/jspang website is very good">params</router-link>
```

4、使用 path 来匹配路由，然后通过 query 来传递参数

```
<router-link :to="{ name:'Query',query: { queryId:  status }}" >
    router-link 跳转 Query
</router-link>
```

对应路由配置：

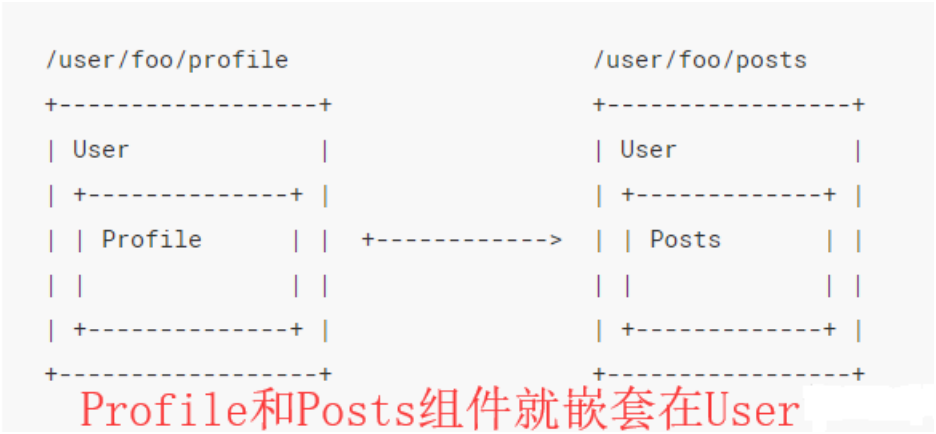
```
  {
    path: '/query',
    name: 'Query',
    component: Query
  }
```

于是我们可以获取参数：

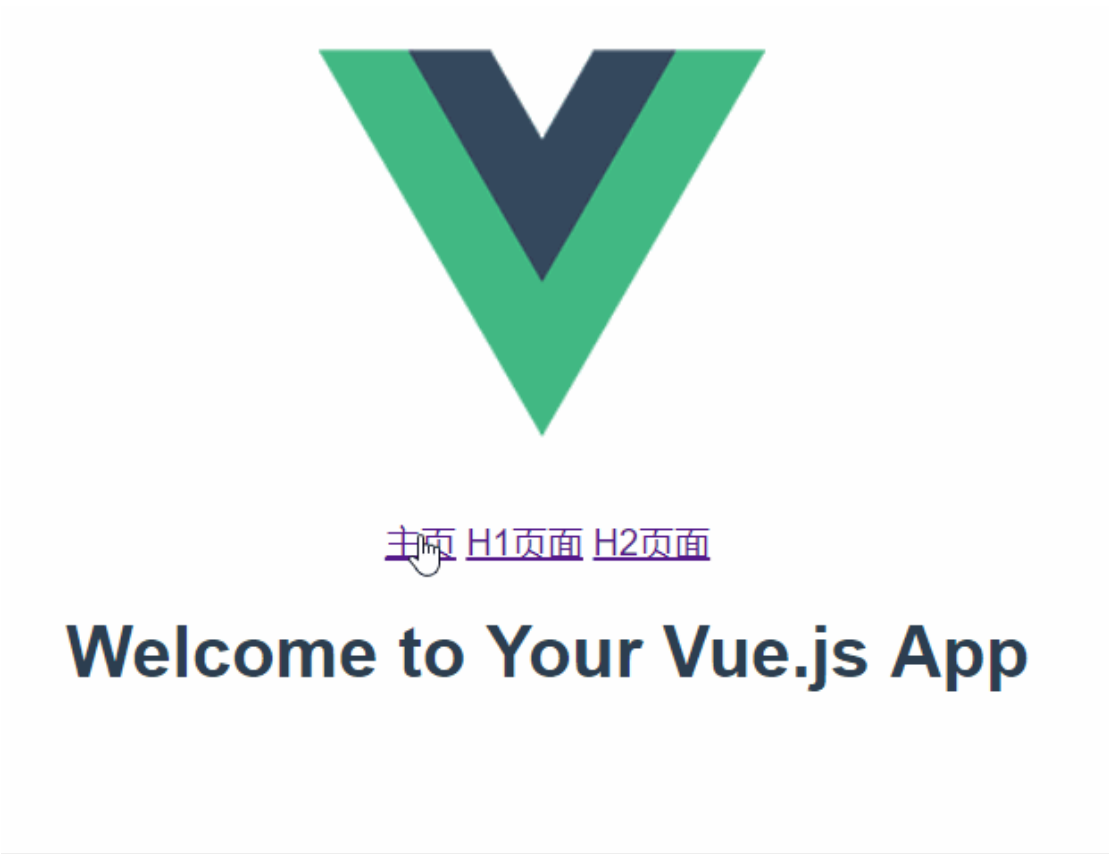
```
this.$route.query.queryId
```

一、vue-router 配置子路由(二级路由)

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各层组件，例如：



如何实现下图效果(H1 页面和 H2 页面嵌套在主页中)?



1.首先用标签增加了两个新的导航链接

```
<router-link :to="{name:'HelloWorld'}">主页</router-link>
<router-link :to="{name:'H1'}">H1 页面</router-link>
<router-link :to="{name:'H2'}">H2 页面</router-link>
```

2.在 HelloWorld.vue 加入 `<router-view>`标签，给子模板提供插入位置

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <router-view></router-view>
  </div>
</template>
```

3.在 components 目录下新建两个组件模板 H1.vue 和 H2.vue 两者内容类似，以下是 H1.vue 页面内容：

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
  </div>
</template>
<script>
  export default {
    data() {
      return {
        msg: 'I am H1 page,Welcome to H1'
      }
    }
  }
![clipboard.png](/img/bVbiDh9)

</script>
```

4.修改 router/index.js 代码，子路由的写法是在原有的路由配置下加入 children 字段。

```
  routes: [
    {
      path: '/',
      name: 'HelloWorld',
```

```
        component: HelloWorld,
        children: [{path: '/h1', name: 'H1', component: H1},//子路由的<router-view>必须在HelloWorld.vue 中出现
          {path: '/h2', name: 'H2', component: H2}
        ]
      }
  ]
```

二、单页面多路由区域操作

在一个页面里我们有 2 个以上 `<router-view>`区域，我们通过配置路由的 js 文件，来操作这些区域的内容

1.App.vue 文件，在 `<router-view>`下面新写了两行 `<router-view>`标签,并加入了些 CSS 样式

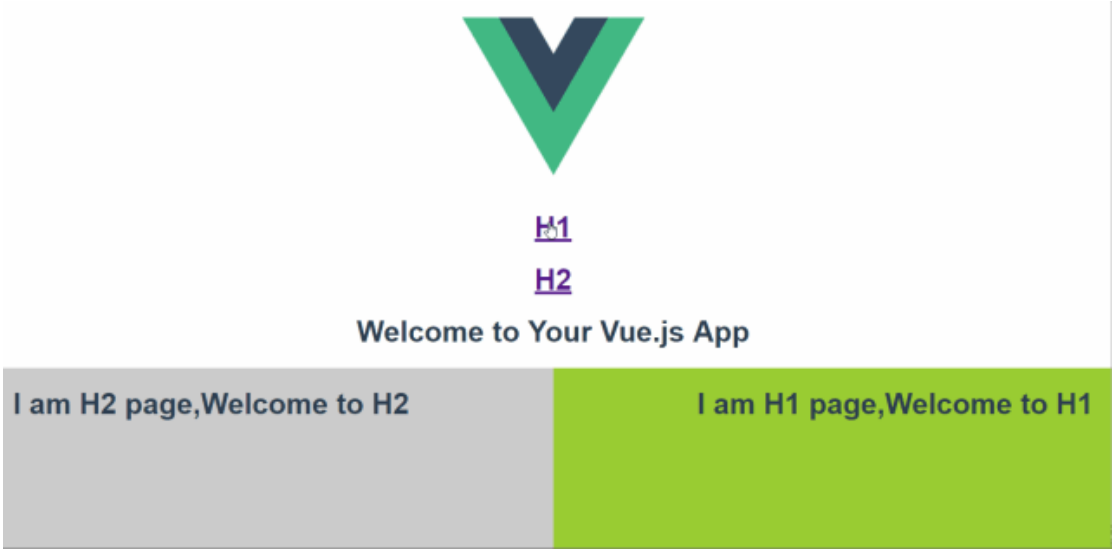
```
<template>
  <div id="app">
    <img src="./assets/logo.png">
       <router-link :to="{name:'HelloWorld'}"><h1>H1</h1></router-link>
       <router-link :to="{name:'H1'}"><h1>H2</h1></router-link>
    <router-view></router-view>
    <router-view name="left" style="float:left;width:50%;background-color:#ccc;height:300px;"/>
    <router-view name="right" style="float:right;width:50%;background-color:yellowgreen;height:300px;"/>
  </div>
</template>
```

2.需要在路由里配置这三个区域，配置主要是在 components 字段里进行

```
 export default new Router({
    routes: [
      {
        path: '/',
        name: 'HelloWorld',
        components: {default: HelloWorld,
          left:H1,//显示 H1 组件内容'I am H1 page,Welcome to H1'
          right:H2//显示 H2 组件内容'I am H2 page,Welcome to H2'
        }
      },
```

```
        {
          path: '/h1',
          name: 'H1',
          components: {default: HelloWorld,
            left:H2,//显示 H2 组件内容
            right:H1//显示 H1 组件内容
          }
        }
      ]
    })
```

上边的代码我们编写了两个路径，一个是默认的‘/’，另一个是‘/Hi’．在两个路径下的 components 里面，我们对三个区域都定义了显示内容。最后页面展示如下图：



三、Vue-router 有几种钩子函数？具体是什么及执行流程是怎样的？

钩子函数种类有：全局守卫、路由守卫、组件守卫。

完整的导航解析流程

1.导航被触发；

2.在失活的组件里调用 beforeRouteLeave 守卫；

3.调用全局 beforeEach 守卫；

4.在复用组件里调用 beforeRouteUpdate 守卫；

5.调用路由配置里的 beforeEnter 守卫；

6.解析异步路由组件；

7.在被激活的组件里调用 beforeRouteEnter 守卫；

8.调用全局 beforeResolve 守卫；

9.导航被确认；

10.调用全局的 afterEach 钩子；

11.DOM 更新；

12.用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

首先 Vuex 是什么?

Vuex 是专门为 Vue 服务，用于管理页面的数据状态、提供统一数据操作的生态系统，相当于数据库 mongoDB，MySQL 等，任何组件都可以存取仓库中的数据。现在就让我们好好了解下 Vuex 到底是个啥东西?

Vuex 采用 MVC 模式中的 Model 层，规定所有的数据必须通过 action--->mutaion--->state 这个流程进行来改变状态的。再结合 Vue 的数据视图双向绑定实现页面的更新。统一页面状态管理，可以让复杂的组件交互变的简单清晰，同时在调试时也可以通过 DEVtools 去查看状态。

在当前前端的 spa 模块化项目中不可避免的是某些变量需要在全局范围内引用，此时父子组件的传值，子父组件间的传值，兄弟组件间的传值成了我们需要解决的问题。虽然 vue 中提供了 props（父传子）commit（子传父）兄弟间也可以用 localstorage 和 sessionstorage。但是这种方式在项目开发中带来的问题比他解决的问题（难管理，难维护，代码复杂，安全性低）更多。vuex 的诞生也是为了解决这些问题，从而大大提高我们 vue 项目的开发效率。

1、vuex 有哪几种属性?

答：有五种，分别是 State、 Getter、Mutation 、Action、 Module

2、vuex 的 State 特性是?

Vuex 就是一个仓库，仓库里面放了很多对象。其中 state 就是数据源存放地，对应于与一般 Vue 对象里面的 data

state 里面存放的数据是响应式的，Vue 组件从 store 中读取数据，若是 store 中的数据发生改变，依赖这个数据的组件也会发生更新

三、它通过 mapState 把全局的 state 和 getters 映射到当前组件的 computed 计算属性中

3、vuex 的 Getter 特性是?

一、getters 可以对 State 进行计算操作，它就是 Store 的计算属性

二、虽然在组件内也可以做计算属性，但是 getters 可以在多组件之间复用

三、 如果一个状态只在一个组件内使用，是可以不用 getters

4、vuex 的 Mutation 特性是?

一、Action 类似于 mutation，不同在于:

Action 提交的是 mutation，而不是直接变更状态。

Action 可以包含任意异步操作

5、Vue.js 中 ajax 请求代码应该写在组件的 methods 中还是 vuex 的 actions 中?

一、如果请求来的数据是不是要被其他组件公用，仅仅在请求的组件内使用，就不需要放入 vuex 的 state 里。

二、如果被其他地方复用，这个很大几率上是需要的，如果需要，请将请求放入 action 里，方便复用，并包装成 promise 返回，在调用处用 async await 处理返回的数据。如果不要复用这个请求，那么直接写在 vue 文件里很方便。

6、不用 Vuex 会带来什么问题?

一、可维护性会下降，你要想修改数据，你得维护三个地方

二、可读性会下降，因为一个组件里的数据，你根本就看不出来是从哪来的

三、增加耦合，大量的上传派发，会让耦合性大大的增加，本来 Vue 用 Component 就是为了减少耦合，现在这么用，和组件化的初衷相背。

但兄弟组件有大量通信的，建议一定要用，不管大项目和小项目，因为这样会省很多事

什么是 Vuex?

Vuex 是一个专为 Vue.js 应用程序开发的状态管理插件。它采用集中式存储管理应用的所有组件的状态，而更改状态的唯一方法是提交 mutation，例 `this.$store.commit('SET_VIDEO_PAUSE', video_pause，SET_VIDEO_PAUSE` 为 mutations 属性中定义的方法 。

Vuex 解决了什么问题?

解决两个问题

- 多个组件依赖于同一状态时，对于多层嵌套的组件的传参将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。
- 来自不同组件的行为需要变更同一状态。以往采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。以上的这些模式非常脆弱，通常会导致无法维护的代码。

什么时候用 Vuex?

当项目遇到以下两种场景时

- 多个组件依赖于同一状态时。
- 来自不同组件的行为需要变更同一状态。

怎么引用 Vuex?

- 先安装依赖 `nnpm install vuex --save`
- 在项目目录 src 中建立 store 文件夹
- 在 store 文件夹下新建 index.js 文件,写入

```
import Vue from 'vue';
import Vuex from 'vuex';
Vue.use(Vuex);
//不是在生产环境 debug 为 true
const debug = process.env.NODE_ENV !== 'production';
//创建 Vuex 实例对象
const store = new Vuex.Store({
    strict:debug,//在不是生产环境下都开启严格模式
    state:{
    },
    getters:{
    },
    mutations:{
    },
    actions:{
    }
})
export default store;
```

- 然后再 main.js 文件中引入 Vuex,这么写

```
import Vue from 'vue';
import App from './App.vue';
import store from './store';
const vm = new Vue({
    store:store,
```

```
    render: h => h(App)
}).$mount('#app')
```

Vuex 的 5 个核心属性是什么？

分别是 state、getters、mutations、actions、modules 。

Vuex 中状态储存在哪里，怎么改变它？

存储在 state 中，改变 Vuex 中的状态的唯一途径就是显式地提交 (commit) mutation。

Vuex 中状态是对象时，使用时要注意什么？

因为对象是引用类型，复制后改变属性还是会影响原始数据，这样会改变 state 里面的状态，是不允许，所以先用深度克隆复制对象，再修改。

怎么在组件中批量使用 Vuex 的 state 状态？

使用 mapState 辅助函数, 利用对象展开运算符将 state 混入 computed 对象中

```
import {mapState} from 'vuex'
export default{
    computed:{
        ...mapState(['price','number'])
    }
}
```

Vuex 中要从 state 派生一些状态出来，且多个组件使用它，该怎么做，？

使用 getter 属性，相当 Vue 中的计算属性 computed，只有原状态改变派生状态才会改变。
getter 接收两个参数，第一个是 state，第二个是 getters(可以用来访问其他 getter)。

```
const store = new Vuex.Store({
    state: {
        price: 10,
        number: 10,
        discount: 0.7,
    },
    getters: {
        total: state => {
            return state.price * state.number
        },
        discountTotal: (state, getters) => {
            return state.discount * getters.total
        }
    },
});
```

然后在组件中可以用计算属性 computed 通过 `this.$store.getters.total` 这样来访问这些派生转态。

```
computed: {
    total() {
        return this.$store.getters.total
    },
    discountTotal() {
        return this.$store.getters.discountTotal
    }
}
```

怎么通过 getter 来实现在组件内可以通过特定条件来获取 state 的状态？

通过让 getter 返回一个函数，来实现给 getter 传参。然后通过参数来进行判断从而获取 state 中满足要求的状态。

```
const store = new Vuex.Store({
    state: {
        todos: [
            { id: 1, text: '...', done: true },
            { id: 2, text: '...', done: false }
        ]
    },
    getters: {
        getTodoById: (state) => (id) =>{
            return state.todos.find(todo => todo.id === id)
```

```
        }
    },
});
```

然后在组件中可以用计算属性 computed 通过 `this.$store.getters.getTodoById(2)`这样来访问这些派生转态。

```
computed: {
    getTodoById() {
        return this.$store.getters.getTodoById
    },
}
mounted(){
    console.log(this.getTodoById(2).done)//false
}
```

怎么在组件中批量使用 Vuex 的 getter 属性

使用 mapGetters 辅助函数，利用对象展开运算符将 getter 混入 computed 对象中

```
import {mapGetters} from 'vuex'
export default{
    computed:{
        ...mapGetters(['total','discountTotal'])
    }
}
```

怎么在组件中批量给 Vuex 的 getter 属性取别名并使用

使用 mapGetters 辅助函数，利用对象展开运算符将 getter 混入 computed 对象中

```
import {mapGetters} from 'vuex'
export default{
    computed:{
        ...mapGetters({
            myTotal:'total',
            myDiscountTotal:'discountTotal',
        })
    }
```

```
}
```

在 Vuex 的 state 中有个状态 number 表示货物数量，在组件怎么改变它。

首先要在 mutations 中注册一个 mutation

```
const store = new Vuex.Store({
    state: {
        number: 10,
    },
    mutations: {
        SET_NUMBER(state,data){
            state.number=data;
        }
    },
});
```

在组件中使用 `this.$store.commit` 提交 mutation，改变 number

```
this.$store.commit('SET_NUMBER',10)
```

在 Vuex 中使用 mutation 要注意什么。

mutation 必须是同步函数

在组件中多次提交同一个 mutation，怎么写使用更方便。

使用 mapMutations 辅助函数,在组件中这么使用

```
import { mapMutations } from 'vuex'
methods:{
    ...mapMutations({
        setNumber:'SET_NUMBER',
    })
}
```

然后调用 `this.setNumber(10)`相当调用 `this.$store.commit('SET_NUMBER',10)`

Vuex 中 action 和 mutation 有什么区别？

- action 提交的是 mutation，而不是直接变更状态。mutation 可以直接变更状态。
- action 可以包含任意异步操作。mutation 只能是同步操作。
- 提交方式不同，action 是用 `this.$store.dispatch('ACTION_NAME',data)`来提交。mutation 是用 `this.$store.commit('SET_NUMBER',10)`来提交。
- 接收参数不同，mutation 第一个参数是 state，而 action 第一个参数是 context，其包含了

```
{
    state,      // 等同于 `store.state`，若在模块中则为局部状态
    rootState,  // 等同于 `store.state`，只存在于模块中
    commit,     // 等同于 `store.commit`
    dispatch,   // 等同于 `store.dispatch`
    getters,    // 等同于 `store.getters`
    rootGetters // 等同于 `store.getters`，只存在于模块中
}
```

Vuex 中 action 和 mutation 有什么相同点？

第二参数都可以接收外部提交时传来的参数。

`this.$store.dispatch('ACTION_NAME',data)`和

`this.$store.commit('SET_NUMBER',10)`

在组件中多次提交同一个 action，怎么写使用更方便。

使用 mapActions 辅助函数,在组件中这么使用

```
methods:{
    ...mapActions({
        setNumber:'SET_NUMBER',
    })
}
```

然后调用 `this.setNumber(10)`相当调用

`this.$store.dispatch('SET_NUMBER',10)`

Vuex 中 action 通常是异步的，那么如何知道 action 什么时候结束呢？

在 action 函数中返回 Promise，然后再提交时候用 then 处理

```
actions:{
    SET_NUMBER_A({commit},data){
        return new Promise((resolve,reject) =>{
            setTimeout(() =>{
                commit('SET_NUMBER',10);
                resolve();
            },2000)
        })
    }
}
this.$store.dispatch('SET_NUMBER_A').then(() => {
  // ...
})
```

Vuex 中有两个 action，分别是 actionA 和 actionB，其内都是异步操作，在 actionB 要提交 actionA，需在 actionA 处理结束再处理其它操作，怎么实现？

利用 ES6 的 `async` 和 `await` 来实现。

```
actions:{
    async actionA({commit}){
        //...
    },
    async actionB({dispatch}){
        await dispatch ('actionA')//等待 actionA 完成
        // ...
    }
}
```

有用过 Vuex 模块吗，为什么要使用，怎么使用。

有，因为使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。所以将 store 分割成

模块（module）。每个模块拥有自己的 state、mutations、actions、getters，甚至是嵌套子模块，从上至下进行同样方式的分割。

在 module 文件新建 moduleA.js 和 moduleB.js 文件。在文件中写入

```
const state={
    //...
}
const getters={
    //...
}
const mutations={
    //...
}
const actions={
    //...
}
export default{
    state,
    getters,
    mutations,
    actions
}
```

然后再 index.js 引入模块

```
import Vue from 'vue';
import Vuex from 'vuex';
Vue.use(Vuex);
import moduleA from './module/moduleA'
import moduleB from './module/moduleB'
const store = new Vuex.Store({
    modules:{
        moduleA,
        moduleB
    }
})
export default store
```

在模块中，getter 和 mutation 接收的第一个参数 state，是全局的还是模块的？

第一个参数 state 是模块的 state，也就是局部的 state。

在模块中，getter 和 mutation 和 action 中怎么访问全局的 state 和 getter？

- 在 getter 中可以通过第三个参数 rootState 访问到全局的 state,可以通过第四个参数 rootGetters 访问到全局的 getter。
- 在 mutation 中不可以访问全局的 satat 和 getter，只能访问到局部的 state。
- 在 action 中第一个参数 context 中的 `context.rootState` 访问到全局的 state，`context.rootGetters` 访问到全局的 getter。

在组件中怎么访问 Vuex 模块中的 getter 和 state,怎么提交 mutation 和 action？

- 直接通过 `this.$store.getters` 和 `this.$store.state` 来访问模块中的 getter 和 state。
- 直接通过 `this.$store.commit('mutationA',data)`提交模块中的 mutation。
- 直接通过 `this.$store.dispatch('actionA,data')`提交模块中的 action。

用过 Vuex 模块的命名空间吗？为什么使用，怎么使用。

默认情况下，模块内部的 action、mutation 和 getter 是注册在全局命名空间，如果多个模块中 action、mutation 的命名是一样的，那么提交 mutation、action 时，将会触发所有模块中命名相同的 mutation、action。
这样有太多的耦合，如果要使你的模块具有更高的封装度和复用性，你可以通过添加 `namespaced: true` 的方式使其成为带命名空间的模块。

```
export default{
    namespaced: true,
    state,
    getters,
    mutations,
    actions
}
```

怎么在带命名空间的模块内提交全局的 mutation 和 action？

将 {root: true} 作为第三参数传给 dispatch 或 commit 即可。

```
this.$store.dispatch('actionA', null, { root: true })
this.$store.commit('mutationA', null, { root: true })
```

怎么在带命名空间的模块内注册全局的 action？

```
actions: {
    actionA: {
        root: true,
        handler (context, data) { ... }
    }
}
```

组件中怎么提交 modules 中的带命名空间的 moduleA 中的 mutationA？

```
this.$store.commit('moduleA/mutationA',data)
```

怎么使用 mapState，mapGetters，mapActions 和 mapMutations 这些函数来绑定带命名空间的模块？

首先使用 createNamespacedHelpers 创建基于某个命名空间辅助函数

```
import { createNamespacedHelpers } from 'vuex';
const { mapState, mapActions } = createNamespacedHelpers('moduleA');
export default {
    computed: {
        // 在 `module/moduleA` 中查找
        ...mapState({
            a: state => state.a,
            b: state => state.b
        })
    },
    methods: {
        // 在 `module/moduleA` 中查找
        ...mapActions([
            'actionA',
            'actionB'
        ])
    }
}
```

Vuex 插件有用过吗？怎么用简单介绍一下？

Vuex 插件就是一个函数，它接收 store 作为唯一参数。在 Vuex.Store 构造器选项 plugins 引入。在 store/plugin.js 文件中写入

```
export default function createPlugin(param){
    return store =>{
        //...
    }
}
```

然后在 store/index.js 文件中写入

```
import createPlugin from './plugin.js'
const myPlugin = createPlugin()
const store = new Vuex.Store({
  // ...
  plugins: [myPlugin]
})
```

在 Vuex 插件中怎么监听组件中提交 mutation 和 action？

- 用 Vuex.Store 的实例方法 `subscribe` 监听组件中提交 mutation
- 用 Vuex.Store 的实例方法 `subscribeAction` 监听组件中提交 action 在 store/plugin.js 文件中写入

```
export default function createPlugin(param) {
    return store => {
        store.subscribe((mutation, state) => {
            console.log(mutation.type)//是那个 mutation
            console.log(mutation.payload)
            console.log(state)
        })
        // store.subscribeAction((action, state) => {
        //     console.log(action.type)//是那个 action
        //     console.log(action.payload)//提交 action 的参数
        // })
        store.subscribeAction({
            before: (action, state) => {//提交 action 之前
                console.log(`before action ${action.type}`)
            },
            after: (action, state) => {//提交 action 之后
                console.log(`after action ${action.type}`)
            }
        })
```

```
    }
}
```

然后在 store/index.js 文件中写入

```
import createPlugin from './plugin.js'
const myPlugin = createPlugin()
const store = new Vuex.Store({
  // ...
  plugins: [myPlugin]
})
```

在 v-model 上怎么用 Vuex 中 state 的值？

需要通过 computed 计算属性来转换。

```
<input v-model="message">
// ...
computed: {
    message: {
        get () {
            return this.$store.state.message
        },
        set (value) {
            this.$store.commit('updateMessage', value)
        }
    }
}
```

Vuex 的严格模式是什么,有什么作用,怎么开启？

在严格模式下，无论何时发生了状态变更且不是由 mutation 函数引起的，将会抛出错误。这能保证所有的状态变更都能被调试工具跟踪到。

在 Vuex.Store 构造器选项中开启,如下

```
const store = new Vuex.Store({
    strict:true,
})
```

第一题：谈谈你对 webpack 的理解？

webpack 是一个打包模块化 js 的工具，在 webpack 里一切文件皆模块，通过 loader 转换文件，通过 plugin 注入钩子，最后输出由多个模块组合成的文件，webpack 专注构建模块化项目。WebPack 可以看做是模块的打包机器：它做的事情是，分析你的项目结构，找到 js 模块以及其它的一些浏览器不能直接运行的拓展语言，例如：Scss，TS 等，并将其打包为合适的格式以供浏览器使用。

第二题：说说 webpack 与 grunt、gulp 的不同？

三者都是前端构建工具，grunt 和 gulp 在早期比较流行，现在 webpack 相对来说比较主流，不过一些轻量化的任务还是会用 gulp 来处理，比如单独打包 CSS 文件等。

grunt 和 gulp 是基于任务和流（Task、Stream）的。类似 jQuery，找到一个（或一类）文件，对其做一系列链式操作，更新流上的数据， 整条链式操作构成了一个任务，多个任务就构成了整个 web 的构建流程。

webpack 是基于入口的。webpack 会自动地递归解析入口所需要加载的所有资源文件，然后用不同的 Loader 来处理不同的文件，用 Plugin 来扩展 webpack 功能。

所以，从构建思路来说，gulp 和 grunt 需要开发者将整个前端构建过程拆分成多个`Task`，并合理控制所有`Task`的调用关系；webpack 需要开发者找到入口，并需要清楚对于不同的资源应该使用什么 Loader 做何种解析和加工

对于知识背景来说，gulp 更像后端开发者的思路，需要对于整个流程了如指掌 webpack 更倾向于前端开发者的思路

第三题：什么是 bundle,什么是 chunk，什么是 module?

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

第四题：什么是 Loader?什么是 Plugin?

1）Loaders 是用来告诉 webpack 如何转化处理某一类型的文件，并且引入到打包出的文件中

2）Plugin 是用来自定义 webpack 打包过程的方式，一个插件是含有 apply 方法的一个对象，通过这个方法可以参与到整个 webpack 打包的各个流程(生命周期)。

第五题：有哪些常见的 Loader？他们是解决什么问题的？

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

第一题：有哪些常见的 Plugin？他们是解决什么问题的？

define-plugin：定义环境变量

commons-chunk-plugin：提取公共代码

uglifyjs-webpack-plugin：通过 UglifyES 压缩 ES6 代码

第二题：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。

第三题：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：

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；

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；

确定入口：根据配置中的 entry 找出所有的入口文件；

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；

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；

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；

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。

第四题：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 的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。

第一题：如何利用 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式。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

删除死代码（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

提取公共代码。

第二题：如何提高 webpack 的构建速度？

多入口情况下，使用 CommonsChunkPlugin 来提取公共代码

通过 externals 配置来提取常用库

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。

使用 Happypack 实现多线程加速编译

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

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

第三题：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

第四题：如何在 vue 项目中实现按需加载？

Vue UI 组件库的按需加载 为了快速开发前端项目，经常会引入现成的 UI 组件库如 ElementUI、iView 等，但是他们的体积和他们所提供的功能一样，是很庞大的。 而通常情况下，我们仅仅需要少量的几个组件就足够了，但是我们却将庞大的组件库打包到我们的源码中，造成了不必要的开销。

不过很多组件库已经提供了现成的解决方案，如 Element 出品的 babel-plugin-component 和 AntDesign 出品的 babel-plugin-import 安装以上插件后，在.babelrc 配置中或 babel-loader 的参数中进行设置，即可实现组件按需加载了。

```
{
  "presets": [["es2015", { "modules": false }]],
  "plugins": [
    [
      "component",
      {
        "libraryName": "element-ui",
        "styleLibraryName": "theme-chalk"
      }
    ]
  ]
}
```

单页应用的按需加载 现在很多前端项目都是通过单页应用的方式开发的，但是随着业务的不断扩展，会面临一个严峻的问题——首次加载的代码量会越来越多，影响用户的体验。

通过 import(*)语句来控制加载时机，webpack 内置了对于 import(*)的解析，会将 import(*)中引入的模块作为一个新的入口在生成一个 chunk。当代码执行到 import(*)语句时，会去加载 Chunk 对应生成的文件。import()会返回一个 Promise 对象，所以为了让浏览器支持，需要事先注入 Promise polyfill

第一题：let 有什么用，有了 var 为什么还要用 let?

在 ES6 之前，声明变量只能用 var，var 方式声明变量其实是很不合理的，准确的说，是因为 ES5 里面没有块级作用域是很不合理的，甚至可以说是一个语言层面的 bug。

没有块级作用域回来带很多难以理解的问题，比如 for 循环 var 变量泄露，变量覆盖等问题。

let 声明的变量拥有自己的块级作用域，且修复了 var 声明变量带来的变量提升问题。

第二题：举一些 ES6 对 String 字符串类型做的常用升级优化?

优化部分:

ES6 新增了字符串模板，在拼接大段字符串时，用反斜杠(`)取代以往的字符串相加的形式，能保留所有空格和换行，使得字符串拼接看起来更加直观，更加优雅。

升级部分:

ES6 在 String 原型上新增了 includes()方法，用于取代传统的只能用 indexOf 查找包含字符的方法(indexOf 返回-1 表示没查到不如 includes 方法返回 false 更明确，语义更清晰)此外还新增了 startsWith(), endsWith(), padStart(),padEnd(),repeat()等方法，可方便的用于查找，补全字符串。

第三题：举一些 ES6 对 Array 数组类型做的常用升级优化?

优化部分:

数组解构赋值：ES6 可以直接以 let [a,b,c] = [1,2,3]形式进行变量赋值，在声明较多变量时，不用再写很多 let(var),且映射关系清晰，且支持赋默认值。

扩展运算符：ES6 新增的扩展运算符(...)(重要),可以轻松的实现数组和松散序列的相互转

化，可以取代 arguments 对象和 apply 方法，轻松获取未知参数个数情况下的参数集合。

（尤其是在 ES5 中，arguments 并不是一个真正的数组，而是一个类数组的对象，但是扩展运算符的逆运算却可以返回一个真正的数组）。

扩展运算符还可以轻松方便的实现数组的复制和解构赋值（let a = [2,3,4]; let b = [...a]）。

升级部分:

ES6 在 Array 原型上新增了 find()方法，用于取代传统的只能用 indexOf 查找包含数组项目的方法，且修复了 indexOf 查找不到 NaN 的 bug([NaN].indexOf(NaN) === -1),

此外还新增了 copyWithin(), includes(), fill(),flat()等方法，可方便的用于字符串的查找，补全,转换等。

第四题：举一些 ES6 对 Number 数字类型做的常用升级优化?

优化部分:

ES6 在 Number 原型上新增了 isFinite(), isNaN\()方法，用来取代传统的全局 isFinite(), isNaN()方法检测数值是否有限、是否是 NaN。

ES5 的 isFinite(), isNaN()方法都会先将非数值类型的参数转化为 Number 类型再做判断这其实是不合理的

最造成 isNaN('NaN') === true 的奇怪行为--'NaN'是一个字符串，但是 isNaN 却说这就是 NaN。

而 Number.isFinite()和 Number.isNaN()则不会有此类问题(Number.isNaN('NaN') === false)。(isFinite()同上)

升级部分:

ES6 在 Math 对象上新增了 Math.cbrt()，trunc()，hypot()等等较多的科学计数法运算方法，可以更加全面的进行立方根、求和立方根等等科学计算。

举一些 ES6 对 Object 类型做的常用升级优化?(重要)

答：

优化部分：

1.对象属性变量式声明：ES6 可以直接以变量形式声明对象属性或者方法，比传统的键值对形式声明更加简洁，更加方便，语义更加清晰。

```
let [apple, orange] = ['red appe', 'yellow orange'];
let myFruits = {apple, orange};        // let myFruits = {apple: 'red appe', orange: 'yellow orange'};
```

尤其在对象解构赋值(见优化部分 2.)或者模块输出变量时，这种写法的好处体现的最为明显:

```
let {keys, values, entries} = Object;
let MyOwnMethods = {keys, values, entries}; // let MyOwnMethods = {keys: keys, values: values, entries: entries}
```

可以看到属性变量式声明属性看起来更加简洁明了。方法也可以采用简洁写法：

```
let es5Fun = {
  method: function() {}
};
let es6Fun = {
```

```
    method(){}
}
```

2.对象的解构赋值：ES6 对象也可以像数组解构赋值那样，进行变量的解构赋值：

```
let {apple, orange} = {apple: 'red appe', orange: 'yellow orange'};
```

3.对象的扩展运算符(...)：ES6 对象的扩展运算符和数组扩展运算符用法本质上差别不大，毕竟数组也就是特殊的对象。

对象的扩展运算符一个最常用也最好用的用处就在于可以轻松的取出一个目标对象内部全部或者部分的可遍历属性，从而进行对象的合并和分解。

```
let {apple, orange, ...otherFruits} = {apple: 'red apple', orange: 'yellow orange', grape: 'purple grape', peach: 'sweet peach'};
// otherFruits {grape: 'purple grape', peach: 'sweet peach'}
// 注意: 对象的扩展运算符用在解构赋值时，扩展运算符只能用在最有一个参数(otherFruits 后面不能再跟其他参数)
let moreFruits = {watermelon: 'nice watermelon'};
let allFruits = {apple, orange, ...otherFruits, ...moreFruits};
```

4.super 关键字：ES6 在 Class 类里新增了类似 this 的关键字 super。同 this 总是指向当前函数所在的对象不同，super 关键字总是指向当前函数所在对象的原型对象。

升级部分：

1.ES6 在 Object 原型上新增了 is()方法，做两个目标对象的相等比较，用来完善'==='方法。

'==='方法中 NaN === NaN //false 其实是不合理的，Object.is 修复了这个小 bug。(Object.is(NaN, NaN) // true)

2.ES6 在 Object 原型上新增了 assign()方法，用于对象新增属性或者多个对象合并。

```
const target = { a: 1 };
const source1 = { b: 2 };
const source2 = { c: 3 };
Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

注意：

assign 合并的对象 target 只能合并 source1、source2 中的自身属性。

并不会合并 source1、source2 中的继承属性，也不会合并不可枚举的属性，且无法正确复制 get 和 set 属性（会直接执行 get/set 函数，取 return 的值）。

3.ES6 在 Object 原型上新增了 getOwnPropertyDescriptors()方法，此方法增强了 ES5 中 getOwnPropertyDescriptor()方法，可以获取指定对象所有自身属性的描述对象。

结合 defineProperties()方法，可以完美复制对象，包括复制 get 和 set 属性。

4.ES6 在 Object 原型上新增了 getPrototypeOf()和 setPrototypeOf()方法，用来获取或设置当前对象的 prototype 对象。

这个方法存在的意义在于，ES5 中获取设置 prototype 对像是通过__proto__属性来实现的，

然而__proto__属性并不是 ES 规范中的明文规定的属性，只是浏览器各大产商“私自”加上去的属性，只不过因为适用范围广而被默认使用了，再非浏览器环境中并不一定就可以使用.

所以为了稳妥起见，获取或设置当前对象的 prototype 对象时，都应该采用 ES6 新增的标准用法。

5.ES6 在 Object 原型上还新增了 Object.keys()，Object.values()，Object.entries()方法，用来获取对象的所有键、所有值和所有键值对数组。

举一些 ES6 对 Function 函数类型做的常用升级优化?(重要)

答：

1.箭头函数(核心)：箭头函数是 ES6 核心的升级项之一，箭头函数里没有自己的 this,这改变了以往 JS 函数中最让人难以理解的 this 运行机制。

主要优化点：

箭头函数内的 this 指向的是函数定义时所在的对象，而不是函数执行时所在的对象。

ES5 函数里的 this 总是指向函数执行时所在的对象，这使得在很多情况下 this 的指向变得很难理解，尤其是非严格模式情况下，this 有时候会指向全局对象，这甚至也可以归结为语言层面的 bug 之一。

ES6 的箭头函数优化了这一点，它的内部没有自己的 this,这也就导致了 this 总是指向上一层的 this，如果上一层还是箭头函数，则继续向上指，直到指向到有自己 this 的函数为止，并作为自己的 this；

箭头函数不能用作构造函数，因为它没有自己的 this，无法实例化；也是**因为箭头函数没有自己的 this,所以箭头函数 内也不存在 arguments 对象。**（可以用扩展运算符代替）

2.**函数默认赋值**：ES6 之前，函数的形参是无法给默认值得，只能在函数内部通过变通方法实现。ES6 以更简洁更明确的方式进行函数默认赋值。

```
function es6Fuc (x, y = 'default') {
    console.log(x, y);
}
es6Fuc(4) // 4, default
```

升级部分：

ES6 新增了双冒号运算符，用来取代以往的 bind，call,和 apply。(浏览器暂不支持，Babel 已经支持转码)

```
foo::bar;
// 等同于
bar.bind(foo);

foo::bar(...arguments);
// 等同于
bar.apply(foo, arguments);
```

Symbol 是什么，有什么作用？

答：

Symbol 是 ES6 引入的第七种原始数据类型（说法不准确，应该是第七种数据类型，Object 不是原始数据类型之一，已更正），

所有 Symbol()生成的值都是独一无二的，可以从根本上解决对象属性太多导致属性名冲突覆盖的问题。

对象中 Symbol()属性不能被 for...in 遍历，但是也不是私有属性。

Map 是什么，有什么作用？

答：

Map 是 ES6 引入的一种类似 Object 的新的数据结构。

Map 可以理解为是 Object 的超集，**打破了以传统键值对形式定义对象，对象的 key 不再局限于字符串，也可以是 Object**。可以更加全面的描述对象的属性。

Proxy 是什么，有什么作用？

答：

Proxy 是 ES6 新增的一个构造函数，可以理解为 JS 语言的一个代理，**用来改变 JS 默认的一些语言行为**，包括拦截默认的 get/set 等底层方法，使得 JS 的使用自由度更高，可以最大限度的满足开发者的需求。

比如通过拦截对象的 get/set 方法，可以轻松地定制自己想要的 key 或者 value。

下面的例子可以看到，随便定义一个 myOwnObj 的 key,都可以变成自己想要的函数。

```
function createMyOwnObj() {
    //想把所有的 key 都变成函数，或者 Promise,或者 anything
    return new Proxy({}, {
        get(target, propKey, receiver) {
            return new Promise((resolve, reject) => {
                setTimeout(() => {
                    let randomBoolean = Math.random() > 0.5;
                    let Message;
                    if (randomBoolean) {
                        Message = `你的${propKey}运气不错，成功了`;
                        resolve(Message);
```

```
                    } else {
                        Message = `你的${propKey}运气不行，失败了`;
                        reject(Message);
                    }
                }, 1000);
            });
        }
    });
}

let myOwnObj = createMyOwnObj();

myOwnObj.hahaha.then(result => {
    console.log(result) //你的 hahaha 运气不错，成功了
}).catch(error => {
    console.log(error) //你的 hahaha 运气不行，失败了
})

myOwnObj.wuwuwu.then(result => {
    console.log(result) //你的 wuwuwu 运气不错，成功了
}).catch(error => {
    console.log(error) //你的 wuwuwu 运气不行，失败了
})
```

Reflect 是什么，有什么作用？

答：

Reflect 是 ES6 引入的一个新的对象，他的主要作用有两点：

- 一是将原生的一些零散分布在 Object、Function 或者全局函数里的方法(如 apply、delete、get、set 等等)，统一整合到 Reflect 上，这样可以更加方便更加统一的管理一些原生 API；
- 二就是因为 Proxy 可以改写默认的原生 API，如果一旦原生 API 别改写可能就找不到了，

  所以 Reflect 也可以起到备份原生 API 的作用，使得即使原生 API 被改写了之后，也可以在被改写之后的 API 用上默认的 API。

Generator 函数是什么，有什么作用？

答：

如果说 JavaScript 是 ECMAScript 标准的一种具体实现、Iterator 遍历器是 Iterator 的具体实现，那么 Generator 函数可以说是 Iterator 接口的具体实现方式。

执行 Generator 函数会返回一个遍历器对象，每一次 Generator 函数里面的 yield 都相当一次遍历器对象的 next()方法，并且可以通过 next(value)方法传入自定义的 value,来改变 Generator 函数的行为。

Generator 函数可以通过配合 Thunk 函数更轻松更优雅的实现异步编程和控制流管理。

Class、extends 是什么，有什么作用？

答：

ES6 的 class 可以看作只是一个 ES5 生成实例对象的构造函数的语法糖。

它参考了 java 语言，定义了一个类的概念，让对象原型写法更加清晰，对象实例化更像是一种面向对象编程。Class 类可以通过 extends 实现继承。

它和 ES5 构造函数的不同点：

类的内部定义的所有方法，都是不可枚举的；

```
///  ES5
function  ES5Fun  (x,  y)  {
  this.x  =  x;
  this.y  =  y;
}
ES5Fun.prototype.toString  =  function  ()  {
   return  '('  +  this.x  +  ',  '  +  this.y  +  ')';
}
var  p  =  new  ES5Fun(1,  3);
p.toString();
Object.keys(ES5Fun.prototype);  //['toString']

//ES6
class  ES6Fun  {
  constructor  (x,  y)  {
    this.x  =  x;
    this.y  =  y;
  }
  toString  ()  {
    return  '('  +  this.x  +  ',  '  +  this.y  +  ')';
  }
}
```

```
Object.keys(ES6Fun.prototype);  //[]
```

ES6 的 class 类必须用 new 命令操作，而 ES5 的构造函数不用 new 也可以执行；

ES6 的 class 类不存在变量提升，必须先定义 class 之后才能实例化，不像 ES5 中可以将构造函数写在实例化之后；

ES5 的继承，实质是先创造子类的实例对象 this，然后再将父类的方法添加到 this 上面。

ES6 的继承机制完全不同，实质是先将父类实例对象的属性和方法，加到 this 上面（所以必须先调用 super 方法），然后再用子类的构造函数修改 this。

module、export、import 是什么，有什么作用？

答：

module、export、import 是 ES6 用来统一前端模块化方案的设计思路和实现方案。

export、import 的出现统一了前端模块化的实现方案，整合规范了浏览器/服务端的模块化方法，

之后用来取代传统的 AMD/CMD、requireJS、seaJS、commondJS 等等一系列前端模块不同的实现方案，使前端模块化更加统一规范，JS 也能更加能实现大型的应用程序开发。

import 引入的模块是静态加载（编译阶段加载）而不是动态加载（运行时加载）。import 引入 export 导出的接口值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

日常前端代码开发中，有哪些值得用 ES6 去改进的编程优化或者规范？

答：

常用箭头函数来取代普通函数；

1. 常用 let 取代 var 命令；
2. 常用数组/对象的结构赋值来命名变量，结构更清晰，语义更明确，可读性更好；
3. 在长字符串多变量组合场合，用模板字符串来取代字符串累加，能取得更好地效果和阅读体验；
4. 用 Class 类取代传统的构造函数，来生成实例化对象；
5. 在大型应用开发中，要保持 module 模块化开发思维，分清模块之间的关系，常用 import、export 方法。

第一题：为什么使用箭头函数?

- **作用域安全性**：当箭头函数被一致使用时，所有东西都保证使用与根对象相同的 `thisObject`。如果一个标准函数回调与一堆箭头函数混合在一起，那么作用域就有可能变得混乱。
- **紧凑性**：箭头函数更容易读写。
- 清晰度：使用箭头函数可明确知道当前 `this` 指向。

第二题：将 Symbol 引入 ES6 的目的是什么？

`Symbol` 是一种新的、特殊的对象，可以用作对象中惟一的属性名。使用 `Symbol` 替换 `string` 可以避免不同的模块属性的冲突。还可以将 `Symbol` 设置为私有，以便尚无直接访问 `Symbol` 权限的任何人都不能访问它们的属性。

`Symbol` 是 JS 新的基本数据类型。与 `number`、`string` 和 `boolean` 原始类型一样，`Symbol` 也有一个用于创建它们的函数。与其他原始类型不同，`Symbol` 没有字面量语法。创建它们的唯一方法是使用以下方法中的 `Symbol` 构造函数

```
let symbol = Symbol();
```

第三题：解释一下原型设计模式？

原型模式会创建新的对象，而不是创建未初始化的对象，它会返回使用从原型或样本对象复制的值进行初始化的对象。原型模式也称为属性模式。

原型模式有用的一个例子是使用与数据库中的默认值匹配的值初始化业务对象。原型对象保留默认值，这些默认值将被复制到新创建的业务对象中。

传统语言很少使用原型模式，但是 JavaScript 作为一种原型语言，在构建新对象及其原型时使用这种模式。

第四题：ES6 中的临时死区是什么？

在 ES6 中，`let` 和 `const` 跟 `var`、`class` 和 `function` 一样也会被提升，只是在进入作用域和被声明之间有一段时间不能访问它们，这段时间是**临时死区（TDZ）**。

第五题：简单理解 Promise 是什么？

1：他是一个嵌套式的异步请求；

2：他有三种状态，分别是；

1:pending （初始化进行中）；

2：fulfiled（操作成功）；

3：rejected（操作失败）；

3：他的优缺点；

1：优点：将异步操作如同步操作的流程表达出来，避免一层层嵌套回调的函数

2：缺点：无法取消(也就是说在你创建一个 Promise 对象那么他就会立即执行，并且中途是不可以取消的)