基础阶段面试题

编写人：全栈全体项目经理

一、HTML与CSS面试题

1.1 H5新增特性和css3新增特性?

```
答：  1.首先 html5 为了更好的实践 web 语义化，增加了 header，footer，nav,aside,section 等语义 化标签，
    2.在表单方面，为了增强表单，为 input 增加了 color，email,data ,range 等类型，
    3.在存储方面，提供了 sessionStorage，localStorage,和离线存储，通过这些存储方式方便数 据在客户端的存储和获取，
    4.在多媒体方面规定了音频和视频元素 audio 和 vedio，另外还 有地理定位，canvas 画布，拖放，多线程编程的 web worker 和 websocket协议
    5.css3新增特性：
        CSS3 边框如 border-radius，box-shadow 等；
        CSS3 背景如 background-size，background-origin 等；
        CSS3 2D，3D 转换如 transform 等；
        CSS3 动画如 animation 等

```

1.2 BFC的理解?

```
答：1.BFC（Block Formatting Context），即块级格式化上下文，它是页面中的一块渲染区域，并且有一套属于自己的渲染规则：

- 内部的盒子会在垂直方向上一个接一个的放置
- 对于同一个BFC的俩个相邻的盒子的margin会发生重叠，与方向无关。
- 每个元素的左外边距与包含块的左边界相接触（从左到右），即使浮动元素也是如此
- BFC的区域不会与float的元素区域重叠
- 计算BFC的高度时，浮动子元素也参与计算
- BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素，反之亦然

BFC目的是形成一个相对于外界完全独立的空间，让内部的子元素不会影响到外部的元素

  2.触发BFC的条件包含不限于：

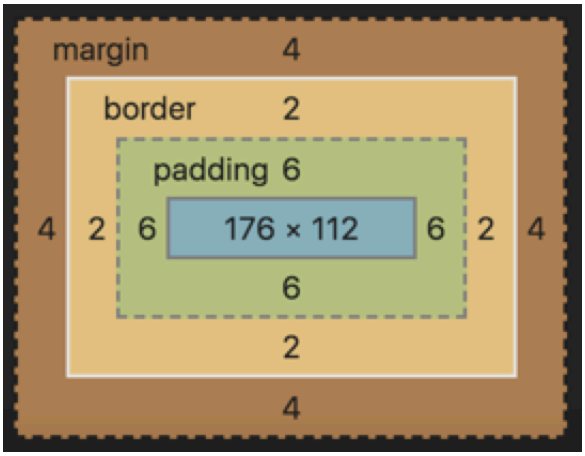
- 根元素，即HTML元素
- 浮动元素：float值为left、right
- overflow值不为 visible，为 auto、scroll、hidden
- display的值为inline-block、inltable-cell、table-caption、table、inline-table、flex、inline-flex、grid、inline-grid
- position的值为absolute或fixed
  3.利用BFC的特性，我们将`BFC`应用在以下场景：
    防止margin重叠（塌陷）
```

```
21          清除内部浮动
22          自适应多栏布局
```

1.3 说说你对盒模型的理解?

答:

- 盒模型其实就是浏览器把一个个标签都看一个矩形盒子，那每个盒子（即标签）都会有内容(width,height)，边框(border)，以及内容和边框中间的缝隙（即内间距padding），还有盒子与盒子之间的外间距（即margin）,用图表示为:



- 当然盒模型包括两种：IE盒模型和w3c标准盒模型

```
IE盒模型总宽度即就是width宽度=border+padding+内容宽度

标准盒模型总宽度＝border+padding+width
```

那如何在IE盒模型宽度和标准盒模型总宽度之间切换呢，可以通过box-sizing:border-box或设置成content-box来切换

```
其中：box-sizing：border-box  //IE盒模型
```

box-sizing：content-box //w3c盒模型

1.4 如何实现元素水平垂直居中?

答：居中是一个非常基础但又是非常重要的应用场景，实现居中的方法存在很多，可以将这些方法分成两个大类:

- 居中元素（子元素）的宽高已知
- 居中元素宽高未知
- 实现方式:
    - **利用定位+margin:auto**

先上代码:

```
<style>
    .father{
        width:500px;
        height:300px;
        border:1px solid #0a3b98;
        position: relative;
    }
    .son{
        width:100px;
        height:40px;
        background: #f0a238;
        position: absolute;
        top:0;
        left:0;
        right:0;
        bottom:0;
        margin:auto;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

父级设置为相对定位，子级绝对定位 ，并且四个定位属性的值都设置了0，那么这时候如果子级没有设置宽高，则会被拉开到和父级一样宽高

这里子元素设置了宽高，所以宽高会按照我们的设置来显示，但是实际上子级的虚拟占位已经撑满了整个父级，这时候再给它一个 `margin: auto` 它就可以上下左右都居中了

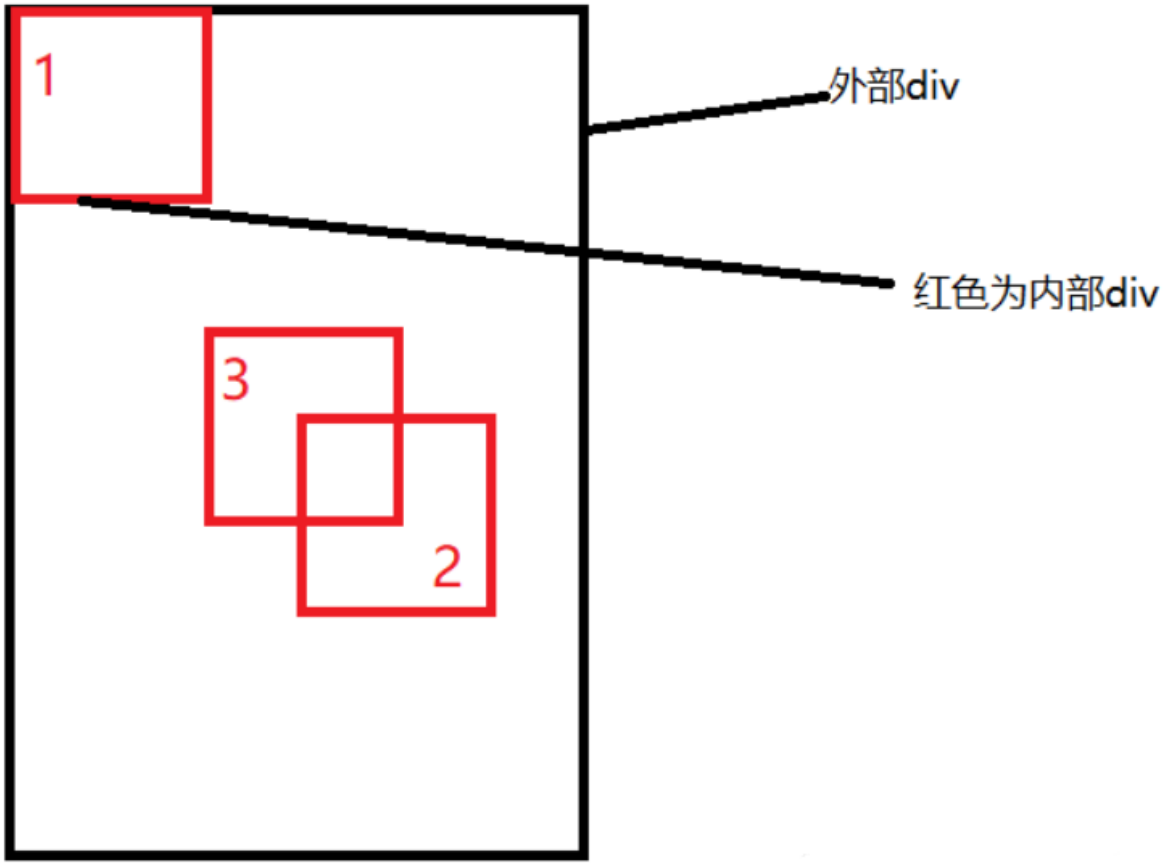
利用定位+margin:负值

绝大多数情况下，设置父元素为相对定位， 子元素移动自身50%实现水平垂直居中

```
<style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
        top: 50%;
        left: 50%;
        margin-left:-50px;
        margin-top:-50px;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
```

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

整个实现思路如下图所示：



- 初始位置为方块1的位置
- 当设置left、top为50%的时候，内部子元素为方块2的位置
- 设置margin为负数时，使内部子元素到方块3的位置，即中间位置

这种方案不要求父元素的高度，也就是即使父元素的高度变化了，仍然可以保持在父元素的垂直居中位置，水平方向上是一样的操作

但是该方案需要知道子元素自身的宽高，但是我们可以通过下面 `transform` 属性进行移动

利用定位+transform

实现代码如下：

```
<style>
    .father {
        position: relative;
        width: 200px;
        height: 200px;
        background: skyblue;
    }
    .son {
        position: absolute;
```

```
            top: 50%;
            left: 50%;
      transform: translate(-50%,-50%);
            width: 100px;
            height: 100px;
            background: red;
        }
    </style>
    <div class="father">
        <div class="son"></div>
    </div>
```

`translate(-50%, -50%)` 将会将元素位移自己宽度和高度的-50%

这种方法其实和最上面被否定掉的margin负值用法一样，可以说是 `margin` 负值的替代方案，并不需要知道自身元素的宽高

table布局

设置父元素为 `display:table-cell` ，子元素设置 `display: inline-block` 。利用 `vertical` 和 `text-align` 可以让所有的行内块级元素水平垂直居中

```
<style>
    .father {
        display: table-cell;
        width: 200px;
        height: 200px;
        background: skyblue;
        vertical-align: middle;
        text-align: center;
    }
    .son {
        display: inline-block;
        width: 100px;
        height: 100px;
        background: red;
    }
</style>
<div class="father">
    <div class="son"></div>
</div>
```

flex弹性布局

还是看看实现的整体代码：

```
<style>
    .father {
```

```
            display: flex;
            justify-content: center;
            align-items: center;
            width: 200px;
            height: 200px;
            background: skyblue;
        }
        .son {
            width: 100px;
            height: 100px;
            background: red;
        }
    </style>
    <div class="father">
        <div class="son"></div>
    </div>
```

`css3` 中了 `flex` 布局，可以非常简单实现垂直水平居中

这里可以简单看看 `flex` 布局的关键属性作用：

- display: flex时，表示该容器内部的元素将按照flex进行布局
- align-items: center表示这些元素将相对于本容器水平居中
- justify-content: center也是同样的道理垂直居中

grid网格布局

```
    <style>
        .father {
                display: grid;
                align-items:center;
                justify-content: center;
                width: 200px;
                height: 200px;
                background: skyblue;

            }
            .son {
                width: 10px;
                height: 10px;
                border: 1px solid red
            }
    </style>
    <div class="father">
        <div class="son"></div>
    </div>
```

这里看到， `gird` 网格布局和 `flex` 弹性布局都简单粗暴

小结

上述方法中，不知道元素宽高大小仍能实现水平垂直居中的方法有：

- 利用定位+margin:auto
- 利用定位+transform
- 利用定位+margin:负值
- flex布局
- grid布局

1.5 如何实现两栏布局，右侧自适应？三栏布局中间自适应呢？

答：

- 两栏布局
  - 方法一：实现思路也非常的简单：
    - 使用 float 左浮左边栏
    - 右边模块使用 margin-left 撑出内容块做内容展示
    - 为父级元素添加BFC，防止下方元素飞到上方内容

  代码如下：

```
<style>
    .box{
        overflow: hidden; 添加BFC
    }
    .left {
        float: left;
        width: 200px;
        background-color: gray;
        height: 400px;
    }
    .right {
        margin-left: 210px;
        background-color: lightgray;
        height: 200px;
    }
</style>
<div class="box">
    <div class="left">左边</div>
    <div class="right">右边</div>
</div>
```

还有一种更为简单的使用则是采取：flex弹性布局

  - 方法二：flex弹性布局

```
<style>
    .box{
        display: flex;
    }
```

```
        .left {
            width: 100px;
        }
        .right {
            flex: 1;
        }
    </style>
    <div class="box">
        <div class="left">左边</div>
        <div class="right">右边</div>
    </div>
```

`flex` 可以说是最好的方案了，代码少，使用简单

注意的是， `flex` 容器的一个默认属性值: `align-items: stretch;`

这个属性导致了列等高的效果。 为了让两个盒子高度自动，需要设置: `align-items: flex-start`

- 三栏布局
  - 实现三栏布局中间自适应的布局方式有：
    - 两边使用 float，中间使用 margin

```
需要将中间的内容放在html结构最后，否则右侧会臣在中间内容的下方

实现代码如下：

```html
<style>
    .wrap {
        background: #eee;
        overflow: hidden; <!-- 生成BFC，计算高度时考虑浮动的元素 -->
        padding: 20px;
        height: 200px;
    }
    .left {
        width: 200px;
        height: 200px;
        float: left;
        background: coral;
    }
    .right {
        width: 120px;
        height: 200px;
        float: right;
        background: lightblue;
    }
    .middle {
        margin-left: 220px;
        height: 200px;
        background: lightpink;
        margin-right: 140px;
```
```

```
    }
</style>
<div class="wrap">
    <div class="left">左侧</div>
    <div class="right">右侧</div>
    <div class="middle">中间</div>
</div>


原理如下：

- 两边固定宽度，中间宽度自适应。
- 利用中间元素的margin值控制两边的间距
- 宽度小于左右部分宽度之和时，右侧部分会被挤下去

这种实现方式存在缺陷：

- 主体内容是最后加载的。

- 右边在主体内容之前，如果是响应式设计，不能简单的换行展示
```

- 两边使用 absolute，中间使用 margin

```
基于绝对定位的三栏布局：注意绝对定位的元素脱离文档流，相对于最近的已经定位的祖先元素进行定位。无需考虑HTML中结构的顺序

<style>
  .container {
    position: relative;
  }

  .left,
  .right,
  .main {
    height: 200px;
    line-height: 200px;
    text-align: center;
  }

  .left {
    position: absolute;
    top: 0;
    left: 0;
    width: 100px;
    background: green;
  }

  .right {
    position: absolute;
    top: 0;
    right: 0;
```

```
      width: 100px;
      background: green;
    }

    .main {
      margin: 0 110px;
      background: black;
      color: white;
    }
  </style>

  <div class="container">
    <div class="left">左边固定宽度</div>
    <div class="right">右边固定宽度</div>
    <div class="main">中间自适应</div>
  </div>


  实现流程：

  - 左右两边使用绝对定位，固定在两侧。
  - 中间占满一行，但通过 margin和左右两边留出10px的间隔
```

- 两边使用 float 和负 margin

```
  <style>
    .left,
    .right,
    .main {
      height: 200px;
      line-height: 200px;
      text-align: center;
    }

    .main-wrapper {
      float: left;
      width: 100%;
    }

    .main {
      margin: 0 110px;
      background: black;
      color: white;
    }

    .left,
    .right {
      float: left;
      width: 100px;
      margin-left: -100%;
      background: green;
```

```
    }

    .right {
      margin-left: -100px; /* 同自身宽度 */
    }
  </style>

  <div class="main-wrapper">
    <div class="main">中间自适应</div>
  </div>
  <div class="left">左边固定宽度</div>
  <div class="right">右边固定宽度</div>

  实现过程：

  - 中间使用了双层标签，外层是浮动的，以便左中右能在同一行展示
  - 左边通过使用负 margin-left:-100%，相当于中间的宽度，所以向上偏移到左侧
  - 右边通过使用负 margin-left:-100px，相当于自身宽度，所以向上偏移到最右侧



  缺点：

  - 增加了 .main-wrapper 一层，结构变复杂
  - 使用负 margin，调试也相对麻烦
```

- display: table 实现
  - `<table>` 标签用于展示行列数据，不适合用于布局。但是可以使用 `display: table` 来实现布局的效果

```
  <style>
    .container {
      height: 200px;
      line-height: 200px;
      text-align: center;
      display: table;
      table-layout: fixed;
      width: 100%;
    }

    .left,
    .right,
    .main {
      display: table-cell;
    }

    .left,
    .right {
      width: 100px;
      background: green;
```

```
    }

    .main {
      background: black;
      color: white;
      width: 100%;
    }
  </style>

  <div class="container">
    <div class="left">左边固定宽度</div>
    <div class="main">中间自适应</div>
    <div class="right">右边固定宽度</div>
  </div>

  实现原理：

  - 层通过 display: table设置为表格，设置 table-layout: fixed`表示列宽自身宽度决定，而不是自动计算。
  - 内层的左中右通过 display: table-cell设置为表格单元。
  - 左右设置固定宽度，中间设置 width: 100% 填充剩下的宽度
```

- flex实现
  - 利用 `flex` 弹性布局，可以简单实现中间自适应
  - 代码如下：

```
  <style type="text/css">
      .wrap {
          display: flex;
          justify-content: space-between;
      }

      .left,
      .right,
      .middle {
          height: 100px;
      }

      .left {
          width: 200px;
          background: coral;
      }

      .right {
          width: 120px;
          background: lightblue;
      }

      .middle {
          background: #555;
```

```
            width: 100%;
            margin: 0 20px;
        }
    </style>
    <div class="wrap">
        <div class="left">左侧</div>
        <div class="middle">中间</div>
        <div class="right">右侧</div>
    </div>

    实现过程：

    - 仅需将容器设置为`display:flex;`，
    - 盒内元素两端对其，将中间元素设置为`100%`宽度，或者设为`flex:1`，即可填充空白
    - 盒内元素的高度撑开容器的高度

    优点：

    - 结构简单直观
    - 可以结合 flex的其他功能实现更多效果，例如使用 order属性调整显示顺序，让主体内容优先加载，但展示在中间
```

- grid网格布局

  代码如下:

```
  <style>
      .wrap {
          display: grid;
          width: 100%;
          grid-template-columns: 300px auto 300px;
      }

      .left,
      .right,
      .middle {
          height: 100px;
      }

      .left {
          background: coral;
      }

      .right {
          width: 300px;
          background: lightblue;
      }

      .middle {
          background: #555;
      }
```

```
    </style>
    <div class="wrap">
        <div class="left">左侧</div>
        <div class="middle">中间</div>
        <div class="right">右侧</div>
    </div>
```

1.6 CSS如何画一个三角形？原理是什么？

答：通常情况下我们会使用图片或者 `svg` 去完成三角形效果图，但如果单纯使用 `css` 如何完成一个三角形呢？

实现过程似乎也并不困难，通过边框就可完成

- 实现过程

在以前也讲过盒子模型，默认情况下是一个矩形，实现也很简单

```
<style>
    .border {
        width: 50px;
        height: 50px;
        border: 2px solid;
        border-color: #96ceb4 #ffeead #d9534f #ffad60;
    }
</style>
<div class="border"></div>
```

效果如下图所示：

将 `border` 设置 `50px` ，效果图如下所示：

白色区域则为 `width` 、 `height` ，这时候只需要你将白色区域部分宽高逐渐变小，最终变为0，则变成如下图所示：

这时候就已经能够看到4个不同颜色的三角形，如果需要下方三角形，只需要将上、左、右边框设置为0就可以得到下方的红色三角形

但这种方式，虽然视觉上是实现了三角形，但实际上，隐藏的部分任然占据部分高度，需要将上方的宽度去掉

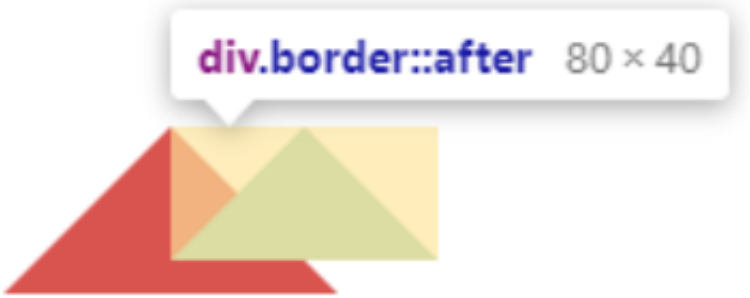
最终实现代码如下：

```
.border {
    width: 0;
    height: 0;
    border-style:solid;
    border-width: 0 50px 50px;
    border-color: transparent transparent #d9534f;
}
```

如果想要实现一个只有边框是空心的三角形，由于这里不能再使用 `border` 属性，所以最直接的方法是利用伪类新建一个小一点的三角形定位上去

```
.border {
    width: 0;
    height: 0;
    border-style:solid;
    border-width: 0 50px 50px;
    border-color: transparent transparent #d9534f;
    position: relative;
}
.border:after{
    content: '';
    border-style:solid;
    border-width: 0 40px 40px;
    border-color: transparent transparent #96ceb4;
    position: absolute;
    top: 0;
    left: 0;
}
```

效果图如下所示：



伪类元素定位参照对象的内容区域宽高都为0，则内容区域即可以理解成中心一点，所以伪元素相对中心这点定位

将元素定位进行微调以及改变颜色，就能够完成下方效果图：

最终代码如下：

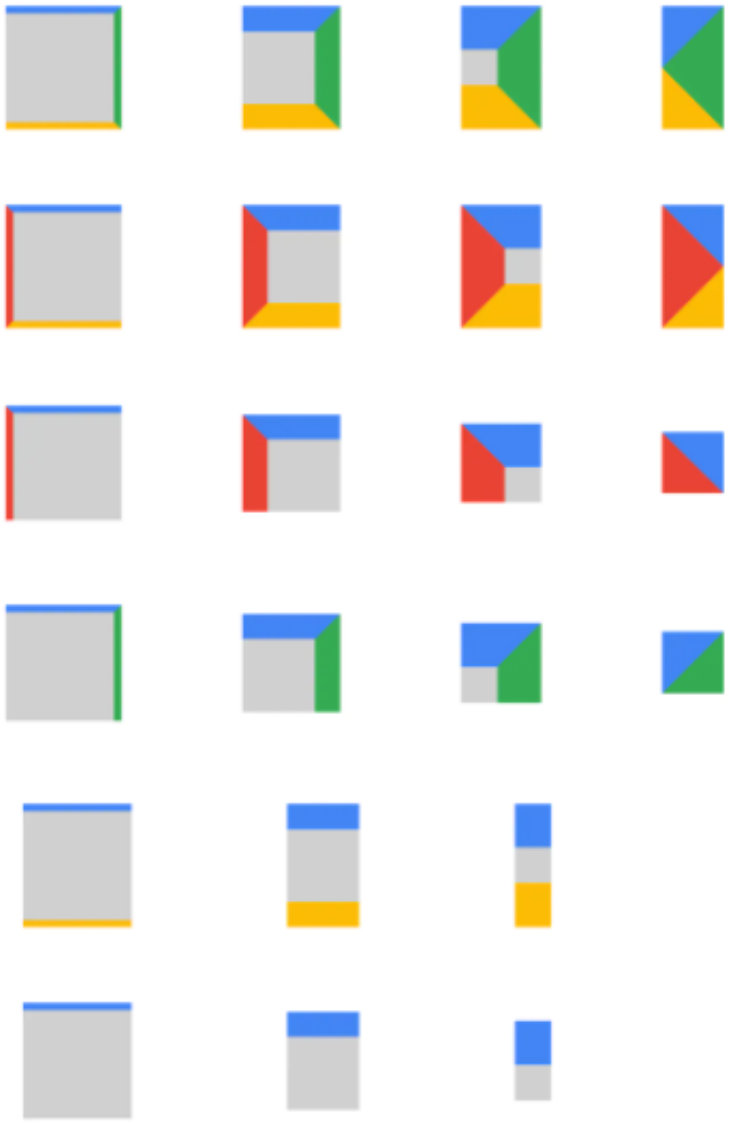
```
.border:after {
    content: '';
    border-style: solid;
    border-width: 0 40px 40px;
    border-color: transparent transparent #96ceb4;
    position: absolute;
    top: 6px;
    left: -40px;
}
```

- 原理分析

可以看到，边框是实现三角形的部分，边框实际上并不是一个直线，如果我们将四条边设置不同的颜色，将边框逐渐放大，可以得到每条边框都是一个梯形

当分别取消边框的时候，发现下面几种情况：

- 取消一条边的时候，与这条边相邻的两条边的接触部分会变成直的
- 当仅有邻边时， 两个边会变成对分的三角
- 当保留边没有其他接触时，极限情况所有东西都会消失



通过上图的变化规则，利用旋转、隐藏，以及设置内容宽高等属性，就能够实现其他类型的三角形

如设置直角三角形，如上图倒数第三行实现过程，我们就能知道整个实现原理

实现代码如下：

```
.box {
    /* 内部大小 */
    width: 0px;
    height: 0px;
    /* 边框大小 只设置两条边*/
    border-top: #4285f4 solid;
    border-right: transparent solid;
    border-width: 85px;
    /* 其他设置 */
    margin: 50px;
}
```

1.7 说说em/px/rem/vh/vw区别?

答：em/px/rem/vh/vw区别如下：

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，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

vh、vw：主要用于页面视口大小布局，在页面布局上更加方便简单

1.8 说一下 localStorage、sessionStorage和cookie的区别?(必问)

> 首选呢,再html5没有出现之前,我们浏览器存储数据的方式一般都是采用cookie,如何要使用cookie需要对cookie进行二次封装才能够更加方便的去使用,使用coookie有一下的特点,cookie具有过期时间,到达指定的时间cookie就会消失,并且一个域名下最多只能存储20条cookie,并且cookie的大小有一定的限制,最后可存储4kb
>
> 而localstorage和sessionstorage都是html5新增的两个api方法,localstroage也称之为数据持久化,当我们使用localstorage将数据存储到本地的时候,如果不在浏览器上手动清楚或者不调用clear或者removeitem方法他是不会自动清楚的.
>
> 而sessionstorage我们称之为会话存储,使用sessionstorage存储的数据再关闭当前页面之后就会消失
>
> 他们和cookie最大的区别是 cookie有过期时间,而本地存储的两个方法如果不手动清除或者关闭浏览器就会一直存储,并且本地存储的api使用起来更加简洁和方便
>
> 以上就是我对本地存储和cookie的理解

二、JavaScript面试题

2.1 Ajax原理是什么？如何实现?

答： `AJAX` 全称(Async Javascript and XML) 即异步的 `JavaScript` 和 `XML` ，是一种创建交互式网页应用的网页开发技术，可以在不重新加载整个网页的情况下，与服务器交换数据，并且更新部分网页

`Ajax` 的原理简单来说通过 `XmlHttpRequest` 对象来向服务器发异步请求，从服务器获得数据，然后用 `JavaScript` 来操作 `DOM` 而更新页面

实现 `Ajax` 异步交互需要服务器逻辑进行配合，需要完成以下步骤：

- 创建 `Ajax` 的核心对象 `XMLHttpRequest` 对象
- 通过 `XMLHttpRequest` 对象的 `open()` 方法与服务端建立连接
- 构建请求所需的数据内容，并通过 `XMLHttpRequest` 对象的 `send()` 方法发送给服务器端
- 通过 `XMLHttpRequest` 对象提供的 `onreadystatechange` 事件监听服务器端你的通信状态
- 接受并处理服务端向客户端响应的数据结果
- 将处理结果更新到 `HTML` 页面中
- 举个例子：

```
const request = new XMLHttpRequest()
request.onreadystatechange = function(e){
    if(request.readyState === 4){ // 整个请求过程完毕
        if(request.status >= 200 && request.status <= 300){
            console.log(request.responseText) // 服务端返回的结果
        }else if(request.status >=400){
            console.log("错误信息: " + request.status)
        }
    }
}
request.open('POST','http://xxxx')
request.send()
```

- ajax封装代码如下:

```
//封装一个ajax请求
function ajax(options) {
    //创建XMLHttpRequest对象
    const xhr = new XMLHttpRequest()


    //初始化参数的内容
    options = options || {}
    options.type = (options.type || 'GET').toUpperCase()
    options.dataType = options.dataType || 'json'
    const params = options.data

    //发送请求
    if (options.type === 'GET') {
        xhr.open('GET', options.url + '?' + params, true)
        xhr.send(null)
    } else if (options.type === 'POST') {
        xhr.open('POST', options.url, true)
        xhr.send(params)

    //接收请求
    xhr.onreadystatechange = function () {
        if (xhr.readyState === 4) {
            let status = xhr.status
            if (status >= 200 && status < 300) {
                options.success && options.success(xhr.responseText,
xhr.responseXML)
            } else {
                options.fail && options.fail(status)
            }
        }
    }
}
```

- 根据上面的封装后的使用方式:

```
ajax({
    type: 'post',
    dataType: 'json',
    data: {},
    url: 'https://xxxx',
    success: function(text,xml){//请求成功后的回调函数
        console.log(text)
    },
    fail: function(status){////请求失败后的回调函数
        console.log(status)
    }
})
```

2.2 数组的常用方法有哪些?

答:数组基本操作可以归纳为 增、删、改、查，需要留意的是哪些方法会对原数组产生影响，哪些方法不会

下面对数组常用的操作方法做一个归纳

- 增

下面前三种是对原数组产生影响的增添方法，第四种则不会对原数组产生影响

- push()
- unshift()
- splice()
- concat()

删

下面三种都会影响原数组，最后一项不影响原数组：

- pop()
- shift()
- splice()
- slice()

改

即修改原来数组的内容，常用 `splice`

splice()

传入三个参数，分别是开始位置，要删除元素的数量，要插入的任意多个元素，返回删除元素的数组，对原数组产生影响

```
let colors = ["red", "green", "blue"];
let removed = colors.splice(1, 1, "red", "purple"); // 插入两个值，删除一个元素
console.log(colors); // red,red,purple,blue
console.log(removed); // green，只有一个元素的数组
```

查

即查找元素，返回元素坐标或者元素值

- indexOf()
- includes()
- find()

排序方法

数组有两个方法可以用来对元素重新排序：

- reverse()
- sort()

转换方法

常见的转换方法有：

join()

join() 方法接收一个参数，即字符串分隔符，返回包含所有项的字符串

迭代方法

常用来迭代数组的方法（都不改变原数组）有如下：

- some()
- every()
- forEach()
- filter()
- map()

2.3 bind、call、apply 区别？如何实现一个bind?

答：1. `apply` 、 `call` 、 `bind` 三者的区别在于：

- 三者都可以改变函数的 `this` 对象指向
- 三者第一个参数都是 `this` 要指向的对象，如果如果没有这个参数或参数为 `undefined` 或 `null` ，则默认指向全局 `window`
- 三者都可以传参，但是 `apply` 是数组，而 `call` 是参数列表，且 `apply` 和 `call` 是一次性传入参数，而 `bind` 可以分为多次传入
- `bind` 是返回绑定this之后的函数， `apply` 、 `call` 则是立即执行

2.实现 `bind` 的步骤，我们可以分解成为三部分：

- 修改 `this` 指向
- 动态传递参数

```
// 方式一：只在bind中传递函数参数
fn.bind(obj,1,2)()

// 方式二：在bind中传递函数参数，也在返回函数中传递参数
fn.bind(obj,1)(2)
```

- 兼容 `new` 关键字

整体实现代码如下：

```
Function.prototype.myBind = function (context) {
    // 判断调用对象是否为函数
    if (typeof this !== "function") {
        throw new TypeError("Error");
    }

    // 获取参数
    const args = [...arguments].slice(1),
          fn = this;

    return function Fn() {

        // 根据调用方式，传入不同绑定值
        return fn.apply(this instanceof Fn ? new fn(...arguments) : context, args.concat(...arguments));
    }
}
```

2.4 Javascript本地存储的方式有哪些？区别及应用场景

答：`javaScript` 本地存储（也称本地缓存）的方法我们主要讲述以下四种：

- cookie
- sessionStorage
- localStorage
- indexedDB

cookie

`Cookie`，类型为「小型文本文件」，指某些网站为了辨别用户身份而储存在用户本地终端上的数据。是为了解决 `HTTP` 无状态导致的问题

作为一段一般不超过 4KB 的小型文本数据，它由一个名称（Name）、一个值（Value）和其它几个用于控制 `cookie` 有效期、安全性、使用范围的可选属性组成

但是 `cookie` 在每次请求中都会被发送，如果不使用 `HTTPS` 并对其加密，其保存的信息很容易被窃取，导致安全风险。举个例子，在一些使用 `cookie` 保持登录态的网站上，如果 `cookie` 被窃取，他人很容易利用你的 `cookie` 来假扮成你登录网站

关于 `cookie` 常用的属性如下：

- Expires 用于设置 Cookie 的过期时间

```
Expires=Wed, 21 Oct 2015 07:28:00 GMT
```

- Max-Age 用于设置在 Cookie 失效之前需要经过的秒数（优先级比 `Expires` 高）

```
Max-Age=604800
```

- `Domain` 指定了 `Cookie` 可以送达的主机名
- `Path` 指定了一个 `URL` 路径，这个路径必须出现在要请求的资源的路径中才可以发送 `Cookie` 首部

```
Path=/docs   # /docs/Web/ 下的资源会带 Cookie 首部
```

- 标记为 `Secure` 的 `Cookie` 只应通过被 `HTTPS` 协议加密过的请求发送给服务端

通过上述，我们可以看到 `cookie` 又开始的作用并不是为了缓存而设计出来，只是借用了 `cookie` 的特性实现缓存

关于 `cookie` 的使用如下：

```
document.cookie = '名字=值';
```

关于 `cookie` 的修改，首先要确定 `domain` 和 `path` 属性都是相同的才可以，其中有一个不同得时候都会创建出一个新的 `cookie`

```
Set-Cookie:name=aa; domain=aa.net; path=/  # 服务端设置
document.cookie =name=bb; domain=aa.net; path=/  # 客户端设置
```

最后 `cookie` 的删除，最常用的方法就是给 `cookie` 设置一个过期的事件，这样 `cookie` 过期后会被浏览器删除

localStorage

`HTML5` 新方法，IE8及以上浏览器都兼容

特点

- 生命周期：持久化的本地存储，除非主动删除数据，否则数据是永远不会过期的
- 存储的信息在同一域中是共享的
- 当本页操作（新增、修改、删除）了 `localStorage` 的时候，本页面不会触发 `storage` 事件,但是别的页面会触发 `storage` 事件。
- 大小：5M（跟浏览器厂商有关系）
- `localStorage` 本质上是对字符串的读取，如果存储内容多的话会消耗内存空间，会导致页面变卡
- 受同源策略的限制

下面再看看关于 `localStorage` 的使用

设置

```
localStorage.setItem('username','cfangxu');
```

获取

```
localStorage.getItem('username')
```

获取键名

```
localStorage.key(0) //获取第一个键名
```

删除

```
localStorage.removeItem('username')
```

一次性清除所有存储

```
localStorage.clear()
```

`localStorage` 也不是完美的，它有两个缺点：

- 无法像 `Cookie` 一样设置过期时间
- 只能存入字符串，无法直接存对象

```
localStorage.setItem('key', {name: 'value'});
console.log(localStorage.getItem('key')); // '[object, Object]'
```

sessionStorage

`sessionStorage` 和 `localStorage` 使用方法基本一致，唯一不同的是生命周期，一旦页面（会话）关闭， `sessionStorage` 将会删除数据

扩展的前端存储方式

`indexedDB` 是一种低级API，用于客户端存储大量结构化数据(包括, 文件/ blobs)。该API使用索引来实现对该数据的高性能搜索

虽然 `Web Storage` 对于存储较少量的数据很有用，但对于存储更大量的结构化数据来说，这种方法不太有用。 `IndexedDB` 提供了一个解决方案

优点：

- 储存量理论上没有上限
- 所有操作都是异步的，相比 `LocalStorage` 同步操作性能更高，尤其是数据量较大时
- 原生支持储存 `JS` 的对象
- 是个正经的数据库，意味着数据库能干的事它都能干

缺点：

- 操作非常繁琐
- 本身有一定门槛

关于 `indexedDB` 的使用基本使用步骤如下：

- 打开数据库并且开始一个事务
- 创建一个 `object store`
- 构建一个请求来执行一些数据库操作，像增加或提取数据等。
- 通过监听正确类型的 `DOM` 事件以等待操作完成。
- 在操作结果上进行一些操作（可以在 `request` 对象中找到）

关于使用 `indexdb` 的使用会比较繁琐，大家可以通过使用 `Godb.js` 库进行缓存，最大化的降低操作难度

区别

关于 `cookie` 、 `sessionStorage` 、 `localStorage` 三者的区别主要如下：

- 存储大小： `cookie` 数据大小不能超过 `4k` ， `sessionStorage` 和 `localStorage` 虽然也有存储大小的限制，但比 `cookie` 大得多，可以达到5M或更大
- 有效时间： `localStorage` 存储持久数据，浏览器关闭后数据不丢失除非主动删除数据； `sessionStorage` 数据在当前浏览器窗口关闭后自动删除； `cookie` 设置的 `cookie` 过期时间之前一直有效，即使窗口或浏览器关闭
- 数据与服务器之间的交互方式， `cookie` 的数据会自动的传递到服务器，服务器端也可以写 `cookie` 到客户端； `sessionStorage` 和 `localStorage` 不会自动把数据发给服务器，仅在本地保存

应用场景

在了解了上述的前端的缓存方式后，我们可以看看针对不对场景的使用选择：

- 标记用户与跟踪用户行为的情况，推荐使用 `cookie`
- 适合长期保存在本地的数据（令牌），推荐使用 `localStorage`
- 敏感账号一次性登录，推荐使用 `sessionStorage`
- 存储大量数据的情况、在线文档（富文本编辑器）保存编辑历史的情况，推荐使用 `indexedDB`

2.5 说说你对闭包的理解？闭包使用场景？

答：闭包说的通俗一点就是打通了一条在函数外部访问函数内部作用域的通道。正常情况下函数外部是访问不到函数内部作用域变量的，

表象判断是不是闭包:函数嵌套函数,内部函数被return 内部函数调用外层函数的局部变量

优点：可以隔离作用域，不造成全局污染

缺点：由于闭包长期驻留内存，则长期这样会导致内存泄露

如何解决内存泄露：将暴露全外部的闭包变量置为null

适用场景：封装组件，for循环和定时器结合使用,for循环和dom事件结合.可以在性能优化的过程中,节流防抖函数的使用,导航栏获取下标的使用

2.6 深拷贝浅拷贝的区别？如何实现一个深拷贝？

答： `JavaScript` 中存在两大数据类型：

- 基本类型
- 引用类型

基本类型数据保存在在栈内存中

引用类型数据保存在堆内存中，引用数据类型的变量是一个指向堆内存中实际对象的引用，存在栈中

浅拷贝

浅拷贝，指的是创建新的数据，这个数据有着原始数据属性值的一份精确拷贝

如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，拷贝的就是内存地址

即浅拷贝是拷贝一层，深层次的引用类型则共享内存地址

下面简单实现一个浅拷贝

```
function shallowClone(obj) {
    const newObj = {};
    for(let prop in obj) {
        if(obj.hasOwnProperty(prop)){
            newObj[prop] = obj[prop];
        }
    }
    return newObj;
}
```

在 `JavaScript` 中，存在浅拷贝的现象有：

- `Object.assign`
- `Array.prototype.slice()`，`Array.prototype.concat()`
- 使用拓展运算符实现的复制

Object.assign

```
 var obj = {
     age: 18,
     nature: ['smart', 'good'],
     names: {
         name1: 'fx',
         name2: 'xka'
     },
     love: function () {
         console.log('fx is a great girl')
     }
 }
 var newObj = Object.assign({}, fxObj);
```

slice()

```
const fxArr = ["One", "Two", "Three"]
const fxArrs = fxArr.slice(0)
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

concat()

```
const fxArr = ["One", "Two", "Three"]
const fxArrs = fxArr.concat()
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

拓展运算符

```
const fxArr = ["One", "Two", "Three"]
const fxArrs = [...fxArr]
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

深拷贝

深拷贝开辟一个新的栈，两个对象属完成相同，但是对应两个不同的地址，修改一个对象的属性，不会改变另一个对象的属性

常见的深拷贝方式有：

- _.cloneDeep()
- jQuery.extend()
- JSON.stringify()
- 手写循环递归

_.cloneDeep()

```
const _ = require('lodash');
const obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
};
const obj2 = _.cloneDeep(obj1);
console.log(obj1.b.f === obj2.b.f);// false
```

jQuery.extend()

```
const $ = require('jquery');
const obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
};
const obj2 = $.extend(true, {}, obj1);
console.log(obj1.b.f === obj2.b.f); // false
```

JSON.stringify()

```
const obj2=JSON.parse(JSON.stringify(obj1));
```

但是这种方式存在弊端，会忽略 `undefined` 、 `symbol` 和 `函数`

```
const obj = {
    name: 'A',
    name1: undefined,
    name3: function() {},
    name4:  Symbol('A')
}
const obj2 = JSON.parse(JSON.stringify(obj));
console.log(obj2); // {name: "A"}
```

循环递归

```
function deepClone(obj, hash = new WeakMap()) {
  if (obj === null) return obj; // 如果是null或者undefined我就不进行拷贝操作
  if (obj instanceof Date) return new Date(obj);
  if (obj instanceof RegExp) return new RegExp(obj);
  // 可能是对象或者普通的值  如果是函数的话是不需要深拷贝
  if (typeof obj !== "object") return obj;
  // 是对象的话就要进行深拷贝
  if (hash.get(obj)) return hash.get(obj);
  let cloneObj = new obj.constructor();
  // 找到的是所属类原型上的constructor,而原型上的 constructor指向的是当前类本身
  hash.set(obj, cloneObj);
  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      // 实现一个递归拷贝
      cloneObj[key] = deepClone(obj[key], hash);
    }
  }
  return cloneObj;
}
```

2.7 说说JavaScript中的数据类型？存储上的差别？

答：在 `JavaScript` 中，我们可以分成两种类型：

- 基本类型
- 复杂类型

两种类型的区别是：存储位置不同：

- 基本数据类型存储在栈中
- 引用类型的对象存储于堆中

基本类型

基本类型主要为以下6种：

- Number
- String
- Boolean
- Undefined
- null

- symbol

引用类型

复杂类型统称为 `Object` ，我们这里主要讲述下面三种：

- Object
- Array
- Function

2.8 什么是防抖和节流？有什么区别？如何实现？

答：**作用：本质上是优化高频率执行代码的一种手段**

如：浏览器的 `resize` 、 `scroll` 、 `keypress` 、 `mousemove` 等事件在触发时，会不断地调用绑定在事件上的回调函数，极大地浪费资源，降低前端性能

为了优化体验，需要对这类事件进行调用次数的限制，对此我们就可以采用 `throttle` （防抖）和 `debounce` （节流）的方式来减少调用频率

定义

- 节流: n 秒内只运行一次，若在 n 秒内重复触发，只有一次生效
- 防抖: n 秒后在执行该事件，若在 n 秒内被重复触发，则重新计时

相同点：

- 都可以通过使用 `setTimeout` 实现
- 目的都是，降低回调执行频率。节省计算资源

不同点：

- 函数防抖，在一段连续操作结束后，处理回调，利用 `clearTimeout` 和 `setTimeout` 实现。函数节流，在一段连续操作中，每一段时间只执行一次，频率较高的事件中使用来提高性能
- 函数防抖关注一定时间连续触发的事件，只在最后执行一次，而函数节流一段时间内只执行一次

例如，都设置时间频率为500ms，在2秒时间内，频繁触发函数，节流，每隔 500ms 就执行一次。防抖，则不管调动多少次方法，在2s后，只会执行一次

应用场景

防抖在连续的事件，只需触发一次回调的场景有：

- 搜索框搜索输入。只需用户最后一次输入完，再发送请求
- 手机号、邮箱验证输入检测
- 窗口大小 `resize` 。只需窗口调整完成后，计算窗口大小。防止重复渲染。

节流在间隔一段时间执行一次回调的场景有：

- 滚动加载，加载更多或滚到底部监听
- 搜索框，搜索联想功能

代码实现

节流

完成节流可以使用时间戳与定时器的写法

使用时间戳写法，事件会立即执行，停止触发后没有办法再次执行

```
function throttled1(fn, delay = 500) {
    let oldtime = Date.now()
    return function (...args) {
        let newtime = Date.now()
        if (newtime - oldtime >= delay) {
            fn.apply(null, args)
            oldtime = Date.now()
        }
    }
}

```

使用定时器写法， `delay` 毫秒后第一次执行，第二次事件停止触发后依然会再一次执行

```
function throttled2(fn, delay = 500) {
    let timer = null
    return function (...args) {
        if (!timer) {
            timer = setTimeout(() => {
                fn.apply(this, args)
                timer = null
            }, delay);
        }
    }
}
```

可以将时间戳写法的特性与定时器写法的特性相结合，实现一个更加精确的节流。实现如下

```
function throttled(fn, delay) {
    let timer = null
    let starttime = Date.now()
    return function () {
        let curTime = Date.now() // 当前时间
        let remaining = delay - (curTime - starttime)  // 从上一次到现在，还剩下多少多余时间
        let context = this
        let args = arguments
        clearTimeout(timer)
        if (remaining <= 0) {
            fn.apply(context, args)
            starttime = Date.now()
        } else {
            timer = setTimeout(fn, remaining);
        }
    }
}
```

防抖

简单版本的实现

```
function debounce(func, wait) {
    let timeout;

    return function () {
        let context = this; // 保存this指向
        let args = arguments; // 拿到event对象

        clearTimeout(timeout)
        timeout = setTimeout(function(){
            func.apply(context, args)
        }, wait);
    }
}
```

防抖如果需要立即执行，可加入第三个参数用于判断，实现如下：

```
function debounce(func, wait, immediate) {

    let timeout;

    return function () {
        let context = this;
        let args = arguments;

        if (timeout) clearTimeout(timeout); // timeout 不为null
        if (immediate) {
            let callNow = !timeout; // 第一次会立即执行，以后只有事件执行后才会再次触发
            timeout = setTimeout(function () {
                timeout = null;
            }, wait)
            if (callNow) {
                func.apply(context, args)
            }
        }
        else {
            timeout = setTimeout(function () {
                func.apply(context, args)
            }, wait);
        }
    }
}
```

2.9 解释下什么是事件代理？应用场景

答：事件代理（也称事件委托）事件代理，俗地来讲，就是把一个元素响应事件（`click`、`keydown`……）的函数委托到另一个元素

前面讲到，事件流的都会经过三个阶段： 捕获阶段 -> 目标阶段 -> 冒泡阶段，而事件委托就是在冒泡阶段完成

事件委托，会把一个或者一组元素的事件委托到它的父层或者更外层元素上，真正绑定事件的是外层元素，而不是目标元素

当事件响应到目标元素上时，会通过事件冒泡机制从而触发它的外层元素的绑定事件上，然后在外层元素上去执行函数

应用场景

如果我们有一个列表，列表之中有大量的列表项，我们需要在点击列表项的时候响应一个事件

```
<ul id="list">
  <li>item 1</li>
  <li>item 2</li>
  <li>item 3</li>
  ......
  <li>item n</li>
</ul>
```

如果给每个列表项一一都绑定一个函数，那对于内存消耗是非常大的

```
// 获取目标元素
const lis = document.getElementsByTagName("li")
// 循环遍历绑定事件
for (let i = 0; i < lis.length; i++) {
    lis[i].onclick = function(e){
        console.log(e.target.innerHTML)
    }
}
```

这时候就可以事件委托，把点击事件绑定在父级元素 `ul` 上面，然后执行事件的时候再去匹配目标元素

```
 // 给父层元素绑定事件
 document.getElementById('list').addEventListener('click', function (e) {
     // 兼容性处理
     var event = e || window.event;
     var target = event.target || event.srcElement;
     // 判断是否匹配目标元素
     if (target.nodeName.toLocaleLowerCase === 'li') {
         console.log('the content is: ', target.innerHTML);
     }
 });
```

还有一种场景是上述列表项并不多，我们给每个列表项都绑定了事件

但是如果用户能够随时动态的增加或者去除列表项元素，那么在每一次改变的时候都需要重新给新增的元素绑定事件，给即将删去的元素解绑事件

如果用了事件委托就没有这种麻烦了，因为事件是绑定在父层的，和目标元素的增减是没有关系的，执行到目标元素是在真正响应执行事件函数的过程中去匹配的

举个例子：

下面 `html` 结构中，点击 `input` 可以动态添加元素

```
<input type="button" name="" id="btn" value="添加" />
<ul id="ul1">
    <li>item 1</li>
    <li>item 2</li>
    <li>item 3</li>
    <li>item 4</li>
</ul>
```

使用事件委托

```
const oBtn = document.getElementById("btn");
const oUl = document.getElementById("ul1");
const num = 4;

//事件委托，添加的子元素也有事件
oUl.onclick = function (ev) {
    ev = ev || window.event;
    const target = ev.target || ev.srcElement;
    if (target.nodeName.toLowerCase() == 'li') {
        console.log('the content is: ', target.innerHTML);
    }

};

//添加新节点
oBtn.onclick = function () {
    num++;
    const oLi = document.createElement('li');
    oLi.innerHTML = `item ${num}`;
    oUl.appendChild(oLi);
};
```

可以看到，使用事件委托，在动态绑定事件的情况下是可以减少很多重复工作的

总结

适合事件委托的事件有：`click`，`mousedown`，`mouseup`，`keydown`，`keyup`，`keypress`

从上面应用场景中，我们就可以看到使用事件委托存在两大优点：

- 减少整个页面所需的内存，提升整体性能
- 动态绑定，减少重复工作

但是使用事件委托也是存在局限性：

- `focus`、`blur` 这些事件没有事件冒泡机制，所以无法进行委托绑定事件
- `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，因此也是不适合于事件委托的

如果把所有事件都用事件代理，可能会出现事件误判，即本不该被触发的事件被绑定上了事件

2.10 说说你对事件循环（Event Loop）的理解?

答：首先，`JavaScript`是一门单线程的语言，意味着同一时间内只能做一件事，但是这并不意味着单线程就是阻塞，而实现单线程非阻塞的方法就是事件循环

在`JavaScript`中，所有的任务都可以分为

- 同步任务：立即执行的任务，同步任务一般会直接进入到主线程中执行
- 异步任务：异步执行的任务，比如`ajax`网络请求，`setTimeout`定时函数等

同步任务与异步任务的运行流程图如下：

从上面我们可以看到，同步任务进入主线程，即主执行栈，异步任务进入任务队列，主线程内的任务执行完毕为空，会去任务队列读取对应的任务，推入主线程执行。上述过程的不断重复就事件循环

宏任务与微任务

如果将任务划分为同步任务和异步任务并不是那么的准确，举个例子：

```
console.log(1)

setTimeout(()=>{
    console.log(2)
}, 0)

```

```
7   new Promise((resolve, reject)=>{
8       console.log('new Promise')
9       resolve()
10  }).then(()=>{
11      console.log('then')
12  })
13
14  console.log(3)
```

如果按照上面流程图来分析代码，我们会得到下面的执行步骤：

- `console.log(1)`，同步任务，主线程中执行
- `setTimeout()`，异步任务，放到 `Event Table`，0 毫秒后 `console.log(2)` 回调推入 `Event Queue` 中
- `new Promise`，同步任务，主线程直接执行
- `.then`，异步任务，放到 `Event Table`
- `console.log(3)`，同步任务，主线程执行

所以按照分析，它的结果应该是 `1` => `'new Promise'` => `3` => `2` => `'then'`

但是实际结果是：`1` => `'new Promise'` => `3` => `'then'` => `2`

出现分歧的原因在于异步任务执行顺序，事件队列其实是一个“先进先出”的数据结构，排在前面的事件会优先被主线程读取

例子中 `setTimeout` 回调事件是先进入队列中的，按理说应该先于 `.then` 中的执行，但是结果却偏偏相反

原因在于异步任务还可以细分为微任务与宏任务

微任务

一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前

常见的微任务有：

- Promise.then
- MutaionObserver
- Object.observe（已废弃；Proxy 对象替代）
- process.nextTick（Node.js）

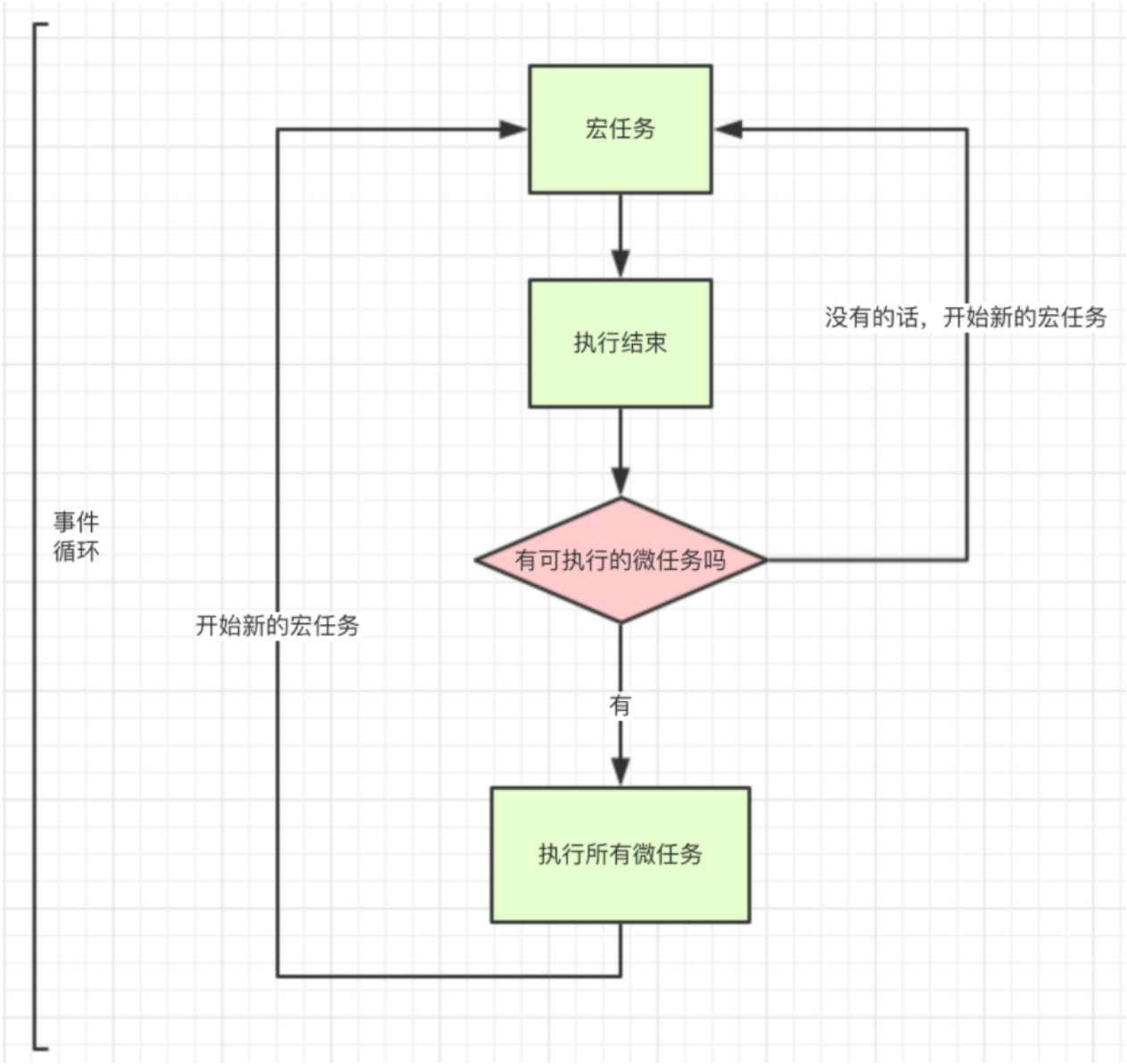
宏任务

宏任务的时间粒度比较大，执行的时间间隔是不能精确控制的，对一些高实时性的需求就不太符合

常见的宏任务有：

- script (可以理解为外层同步代码)
- setTimeout/setInterval
- UI rendering/UI事件
- postMessage、MessageChannel
- setImmediate、I/O（Node.js）

这时候，事件循环，宏任务，微任务的关系如图所示



按照这个流程，它的执行机制是：

- 执行一个宏任务，如果遇到微任务就将它放到微任务的事件队列中
- 当前宏任务执行完成后，会查看微任务的事件队列，然后将里面的所有微任务依次执行完

回到上面的题目

```
console.log(1)
setTimeout(()=>{
    console.log(2)
}, 0)
new Promise((resolve, reject)=>{
    console.log('new Promise')
    resolve()
}).then(()=>{
    console.log('then')
})
console.log(3)
```

流程如下

```
// 遇到 console.log(1) ，直接打印 1
// 遇到定时器，属于新的宏任务，留着后面执行
// 遇到 new Promise，这个是直接执行的，打印 'new Promise'
// .then 属于微任务，放入微任务队列，后面再执行
// 遇到 console.log(3) 直接打印 3
// 好了本轮宏任务执行完毕，现在去微任务列表查看是否有微任务，发现 .then 的回调，执行它，打印 'then'
// 当一次宏任务执行完，再去执行新的宏任务，这里就剩一个定时器的宏任务了，执行它，打印 2
```

async与await

`async` 是异步的意思， `await` 则可以理解为 `async wait` 。所以可以理解 `async` 就是用来声明一个异步方法，而 `await` 是用来等待异步方法执行

async

`async` 函数返回一个 `promise` 对象，下面两种方法是等效的

```
function f() {
    return Promise.resolve('TEST');
}

// asyncF is equivalent to f!
async function asyncF() {
    return 'TEST';
}
```

await

正常情况下， `await` 命令后面是一个 `Promise` 对象，返回该对象的结果。如果不是 `Promise` 对象，就直接返回对应的值

```
async function f(){
    // 等同于
    // return 123
    return await 123
}
f().then(v => console.log(v)) // 123
```

不管 `await` 后面跟着的是什么， `await` 都会阻塞后面的代码

```
async function fn1 (){
    console.log(1)
    await fn2()
    console.log(2) // 阻塞
}

async function fn2 (){
    console.log('fn2')
}

fn1()
console.log(3)
```

上面的例子中， `await` 会阻塞下面的代码（即加入微任务队列），先执行 `async` 外面的同步代码，同步代码执行完，再回到 `async` 函数中，再执行之前阻塞的代码

所以上述输出结果为： `1` ， `fn2` ， `3` ， `2`

流程分析

通过对上面的了解，我们对 `JavaScript` 对各种场景的执行顺序有了大致的了解

这里直接上代码：

```
async function async1() {
    console.log('async1 start')
    await async2()
    console.log('async1 end')
}
async function async2() {
    console.log('async2')
}
console.log('script start')
setTimeout(function () {
    console.log('settimeout')
})
async1()
new Promise(function (resolve) {
    console.log('promise1')
    resolve()
}).then(function () {
    console.log('promise2')
})
console.log('script end')
```

分析过程：

1. 执行整段代码，遇到 `console.log('script start')` 直接打印结果，输出 `script start`
2. 遇到定时器了，它是宏任务，先放着不执行
3. 遇到 `async1()` ，执行 `async1` 函数，先打印 `async1 start` ，下面遇到 `await` 怎么办？先执行 `async2` ，打印 `async2` ，然后阻塞下面代码（即加入微任务列表），跳出去执行同步代

码

4. 跳到 `new Promise` 这里，直接执行，打印 `promise1`，下面遇到 `.then()`，它是微任务，放到微任务列表等待执行
5. 最后一行直接打印 `script end`，现在同步代码执行完了，开始执行微任务，即 `await` 下面的代码，打印 `async1 end`
6. 继续执行下一个微任务，即执行 `then` 的回调，打印 `promise2`
7. 上一个宏任务所有事都做完了，开始下一个宏任务，就是定时器，打印 `settimeout`

所以最后的结果是：`scrit start`、`async1 start`、`async2`、`promise1`、`script end`、`async1 end`、`promise2`、`settimeout`

2.11 Javascript如何实现继承?

```
答：JS继承实现方式也很多，主要分ES5和ES6继承的实现
        先说一下ES5是如何实现继承的
        ES5实现继承主要是基于prototype来实现的，具体有三种方法
一是原型链继承：即 B.prototype=new A()
二是借用构造函数继承(call或者apply的方式继承)
    function B(name,age) {
      A.call(ths,name,age)
    }

三是组合继承
   组合继承是结合第一种和第二种方式

        再说一下ES6是如何实现继承的

        ES6继承是目前比较新，并且主流的继承方式，用class定义类，用extends继承类，用super()表示父类，【下面代码部分只是熟悉，不用说课】
        例如：创建A类
class  A  {
    constructor() {
        //构造器代码，new时自动执行
    }

   方法1( ) { //A类的方法 }
   方法2( ) { //A类的方法 }

}

创建B类并继承A类
class B extends A {
    constructor() {
        super()  //表示父类
    }
}

实例化B类：  var b1=new B( )
             b1.方法1( )


```

2.12 说说 JavaScript 中内存泄漏的几种情况?

答：**内存泄漏**（Memory leak）是在计算机科学中，由于疏忽或错误造成程序未能释放已经不再使用的内存

并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费

程序的运行需要内存。只要程序提出要求，操作系统或者运行时就必须供给内存

对于持续运行的服务进程，必须及时释放不再用到的内存。否则，内存占用越来越高，轻则影响系统性能，重则导致进程崩溃

大多数语言提供自动内存管理，减轻程序员的负担，这被称为"垃圾回收机制"

垃圾回收机制

Javascript 具有自动垃圾回收机制（GC：Garbage Collecation），也就是说，执行环境会负责管理代码执行过程中使用的内存

原理：垃圾收集器会定期（周期性）找出那些不在继续使用的变量，然后释放其内存

通常情况下有两种实现方式：

- 标记清除
- 引用计数

标记清除

`JavaScript` 最常用的垃圾收回机制

当变量进入执行环境是，就标记这个变量为“进入环境“。进入环境的变量所占用的内存就不能释放，当变量离开环境时，则将其标记为“离开环境“

垃圾回收程序运行的时候，会标记内存中存储的所有变量。然后，它会将所有在上下文中的变量，以及被在上下文中的变量引用的变量的标记去掉

在此之后再被加上标记的变量就是待删除的了，原因是任何在上下文中的变量都访问不到它们了

随后垃圾回收程序做一次内存清理，销毁带标记的所有值并收回它们的内存

举个例子：

```
var m = 0,n = 19 // 把 m,n,add() 标记为进入环境。
add(m, n) // 把 a, b, c标记为进入环境。
console.log(n) // a,b,c标记为离开环境，等待垃圾回收。
function add(a, b) {
  a++
  var c = a + b
  return c
}
```

引用计数

语言引擎有一张"引用表"，保存了内存里面所有的资源（通常是各种值）的引用次数。如果一个值的引用次数是 `0` ，就表示这个值不再用到了，因此可以将这块内存释放

如果一个值不再需要了，引用数却不为 `0` ，垃圾回收机制无法释放这块内存，从而导致内存泄漏

```
const arr = [1, 2, 3, 4];
console.log('hello world');
```

面代码中，数组 `[1, 2, 3, 4]` 是一个值，会占用内存。变量 `arr` 是仅有的对这个值的引用，因此引用次数为 `1` 。尽管后面的代码没有用到 `arr` ，它还是会持续占用内存

如果需要这块内存被垃圾回收机制释放，只需要设置如下：

```
arr = null
```

通过设置 `arr` 为 `null` ，就解除了对数组 `[1,2,3,4]` 的引用，引用次数变为 0，就被垃圾回收了

小结

有了垃圾回收机制，不代表不用关注内存泄露。那些很占空间的值，一旦不再用到，需要检查是否还存在对它们的引用。如果是的话，就必须手动解除引用

常见内存泄露情况

意外的全局变量

```
function foo(arg) {
    bar = "this is a hidden global variable";
}
```

另一种意外的全局变量可能由 `this` 创建：

```
function foo() {
    this.variable = "potential accidental global";
}
// foo 调用自己，this 指向了全局对象 (window)
foo();
```

上述使用严格模式，可以避免意外的全局变量

定时器也常会造成内存泄露

```
var someResource = getData();
setInterval(function() {
    var node = document.getElementById('Node');
    if(node) {
        // 处理 node 和 someResource
        node.innerHTML = JSON.stringify(someResource));
    }
}, 1000);
```

如果 `id` 为Node的元素从 `DOM` 中移除，该定时器仍会存在，同时，因为回调函数中包含对 `someResource` 的引用，定时器外面的 `someResource` 也不会被释放

包括我们之前所说的闭包，维持函数内局部变量，使其得不到释放

```
function bindEvent() {
  var obj = document.createElement('XXX');
  var unused = function () {
    console.log(obj, '闭包内引用obj obj不会被释放');
  };
  obj = null; // 解决方法
}
```

没有清理对 `DOM` 元素的引用同样造成内存泄露

```
const refA = document.getElementById('refA');
document.body.removeChild(refA); // dom删除了
console.log(refA, 'refA'); // 但是还存在引用能console出整个div 没有被回收
refA = null;
console.log(refA, 'refA'); // 解除引用
```

包括使用事件监听 `addEventListener` 监听的时候，在不监听的情况下使用 `removeEventListener` 取消对事件监听

2.13 说说new操作符具体干了什么?

答：在 `JavaScript` 中， `new` 操作符用于创建一个给定构造函数的实例对象

例子

```
function Person(name, age){
    this.name = name;
    this.age = age;
}
Person.prototype.sayName = function () {
    console.log(this.name)
}
const person1 = new Person('Tom', 20)
console.log(person1)  // Person {name: "Tom", age: 20}
t.sayName() // 'Tom'
```

从上面可以看到：

- `new` 通过构造函数 `Person` 创建出来的实例可以访问到构造函数中的属性
- `new` 通过构造函数 `Person` 创建出来的实例可以访问到构造函数原型链中的属性（即实例与构造函数通过原型链连接了起来）

现在在构建函数中显式加上返回值，并且这个返回值是一个原始类型

```
function Test(name) {
  this.name = name
  return 1
}
const t = new Test('xxx')
console.log(t.name) // 'xxx'
```

可以发现，构造函数中返回一个原始值，然而这个返回值并没有作用

下面在构造函数中返回一个对象

```
function Test(name) {
  this.name = name
  console.log(this) // Test { name: 'xxx' }
  return { age: 26 }
}
const t = new Test('xxx')
console.log(t) // { age: 26 }
console.log(t.name) // 'undefined'
```

从上面可以发现，构造函数如果返回值为一个对象，那么这个返回值会被正常使用

流程

从上面介绍中，我们可以看到 `new` 关键字主要做了以下的工作：

- 创建一个新的对象 `obj`
- 将对象与构建函数通过原型链连接起来
- 将构建函数中的 `this` 绑定到新建的对象 `obj` 上
- 根据构建函数返回类型作判断，如果是原始值则被忽略，如果是返回对象，需要正常处理

举个例子：

```
function Person(name, age){
    this.name = name;
    this.age = age;
}
const person1 = new Person('Tom', 20)
console.log(person1)  // Person {name: "Tom", age: 20}
t.sayName() // 'Tom'
```

流程图如下：

手写new操作符：

```
function mynew(Func, ...args) {
    // 1.创建一个新对象
    const obj = {}
    // 2.新对象原型指向构造函数原型对象
    obj.__proto__ = Func.prototype
    // 3.将构建函数的this指向新对象
    let result = Func.apply(obj, args)
    // 4.根据返回值判断
    return result instanceof Object ? result : obj
}
```

测试一下

```
function mynew(func, ...args) {
    const obj = {}
    obj.__proto__ = func.prototype
    let result = func.apply(obj, args)
    return result instanceof Object ? result : obj
```

```
}
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.say = function () {
    console.log(this.name)
}

let p = mynew(Person, "huihui", 123)
console.log(p) // Person {name: "huihui", age: 123}
p.say() // huihui
```

可以发现，代码虽然很短，但是能够模拟实现 `new`

2.14 说一下JavaScript原型，原型链的理解?

答：**原型**

`JavaScript` 常被描述为一种基于原型的语言——每个对象拥有一个原型对象

当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾

准确地说，这些属性和方法定义在Object的构造器函数（constructor functions）之上的 `prototype` 属性上，而非实例对象本身

下面举个例子：

函数可以有属性。 每个函数都有一个特殊的属性叫作原型 `prototype`

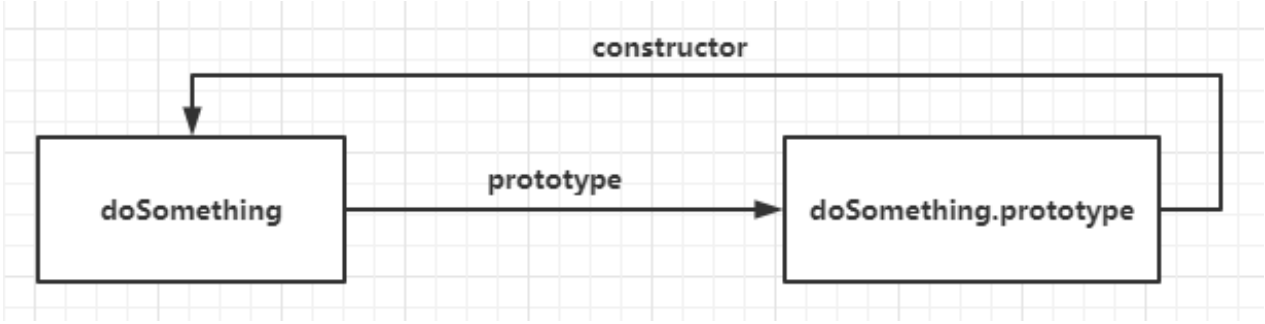
```
function doSomething(){}
console.log( doSomething.prototype );
```

控制台输出

```
{
    constructor: ƒ doSomething(),
    __proto__: {
        constructor: ƒ Object(),
        hasOwnProperty: ƒ hasOwnProperty(),
        isPrototypeOf: ƒ isPrototypeOf(),
        propertyIsEnumerable: ƒ propertyIsEnumerable(),
        toLocaleString: ƒ toLocaleString(),
        toString: ƒ toString(),
        valueOf: ƒ valueOf()
    }
}
```

上面这个对象，就是大家常说的原型对象

可以看到，原型对象有一个自有属性 `constructor` ，这个属性指向该函数，如下图关系展示



原型链

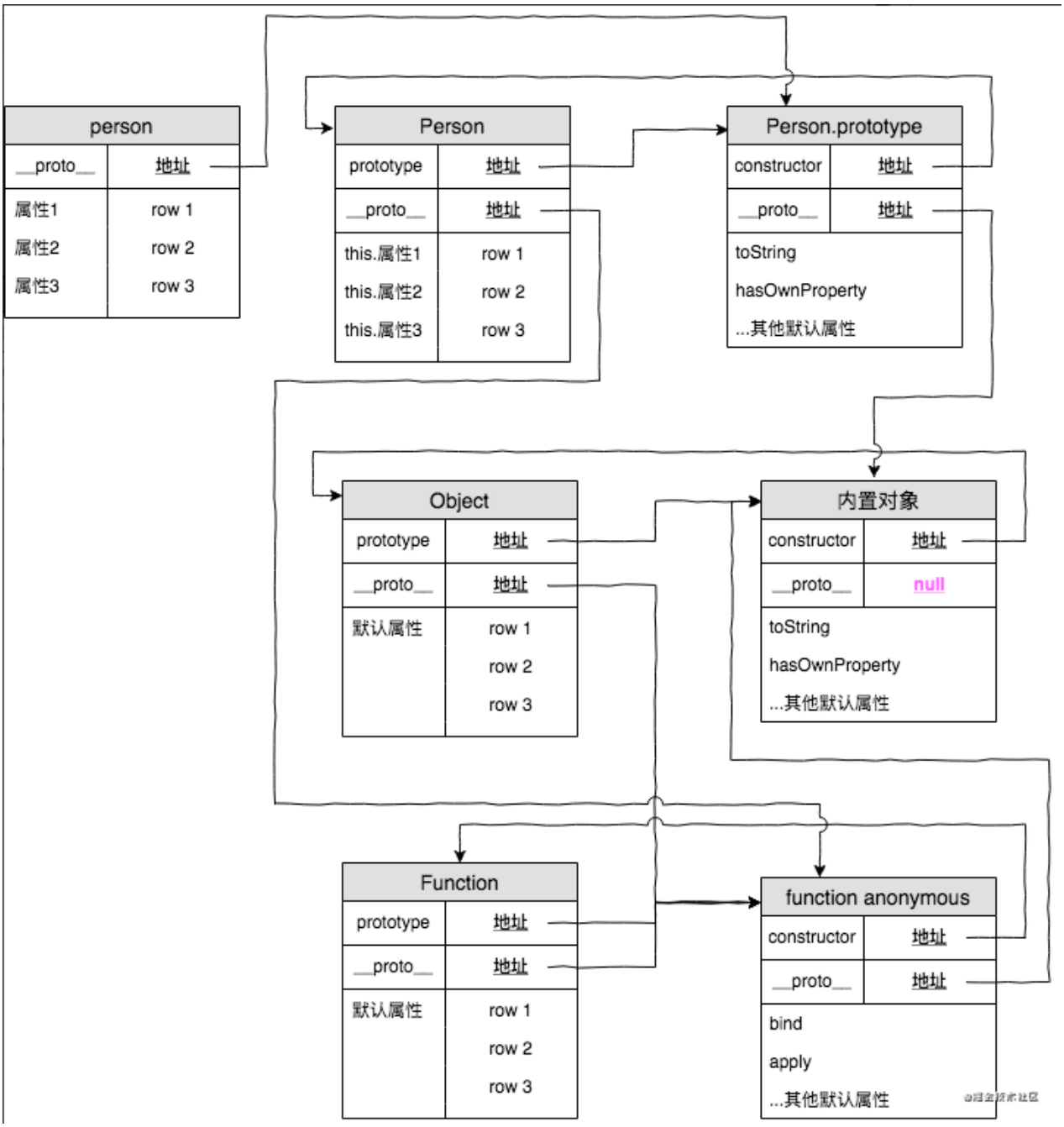
原型对象也可能拥有原型，并从中继承方法和属性，一层一层、以此类推。这种关系常被称为原型链 (prototype chain)，它解释了为何一个对象会拥有定义在其他对象中的属性和方法

在对象实例和它的构造器之间建立一个链接（它是 `__proto__` 属性，是从构造函数的 `prototype` 属性派生的），之后通过上溯原型链，在构造器中找到这些属性和方法

下面举个例子：

```
function Person(name) {
    this.name = name;
    this.age = 18;
    this.sayName = function() {
        console.log(this.name);
    }
}
// 第二步 创建实例
var person = new Person('person')
```

根据代码，我们可以得到下图



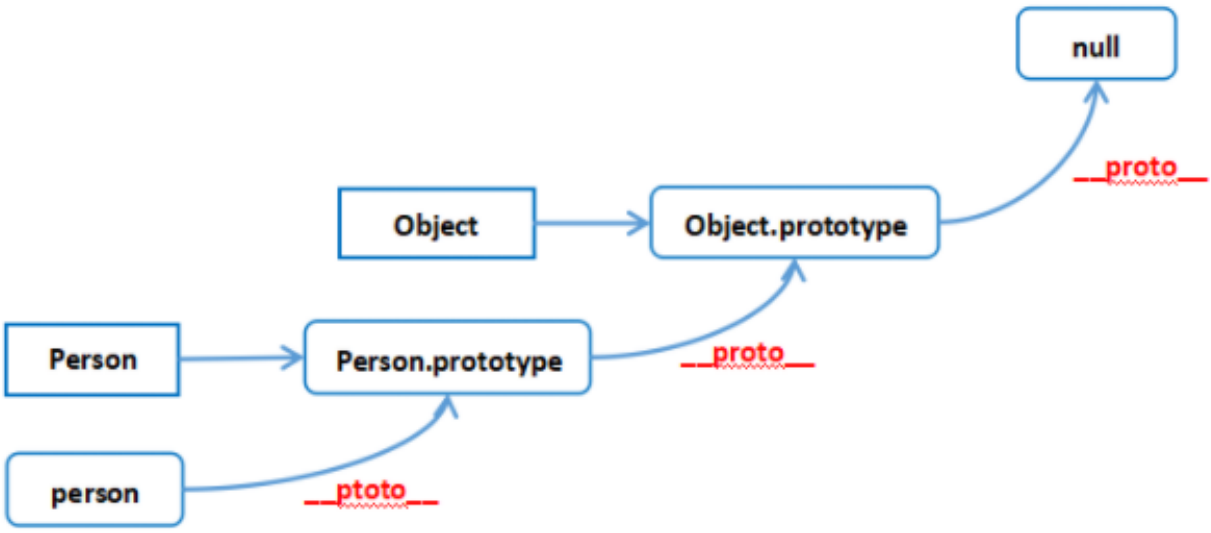
下面分析一下：

- 构造函数 `Person` 存在原型对象 `Person.prototype`
- 构造函数生成实例对象 `person` ， `person` 的 `__proto__` 指向构造函数 `Person` 原型对象
- `Person.prototype.__proto__` 指向内置对象，因为 `Person.prototype` 是个对象，默认是由 `Object` 函数作为类创建的，而 `Object.prototype` 为内置对象
- `Person.__proto__` 指向内置匿名函数 `anonymous` ，因为 Person 是个函数对象，默认由 Function 作为类创建
- `Function.prototype` 和 `Function.__proto__` 同时指向内置匿名函数 `anonymous` ，这样原型链的终点就是 `null`

总结

下面首先要看几个概念：

`__proto__` 作为不同对象之间的桥梁，用来指向创建它的构造函数的原型对象的



每个对象的 `__proto__` 都是指向它的构造函数的原型对象 `prototype` 的

```
person1.__proto__ === Person.prototype
```

构造函数是一个函数对象，是通过 `Function` 构造器产生的

```
Person.__proto__ === Function.prototype
```

原型对象本身是一个普通对象，而普通对象的构造函数都是 `Object`

```
Person.prototype.__proto__ === Object.prototype
```

刚刚上面说了，所有的构造器都是函数对象，函数对象都是 `Function` 构造产生的

```
Object.__proto__ === Function.prototype
```

`Object` 的原型对象也有 `__proto__` 属性指向 `null` ， `null` 是原型链的顶端

```
Object.prototype.__proto__ === null
```

下面作出总结：

- 一切对象都是继承自 `Object` 对象， `Object` 对象直接继承根源对象 `null`
- 一切的函数对象（包括 `Object` 对象），都是继承自 `Function` 对象
- `Object` 对象直接继承自 `Function` 对象
- `Function` 对象的 `__proto__` 会指向自己的原型对象，最终还是继承自 `Object` 对象

2.15 如何实现上拉加载，下拉刷新?

答：下拉刷新和上拉加载这两种交互方式通常出现在移动端中

本质上等同于PC网页中的分页，只是交互形式不同

开源社区也有很多优秀的解决方案，如 `iscroll` 、 `better-scroll` 、 `pulltorefresh.js` 库等等

这些第三方库使用起来非常便捷

我们通过原生的方式实现一次上拉加载，下拉刷新，有助于对第三方库有更好的理解与使用

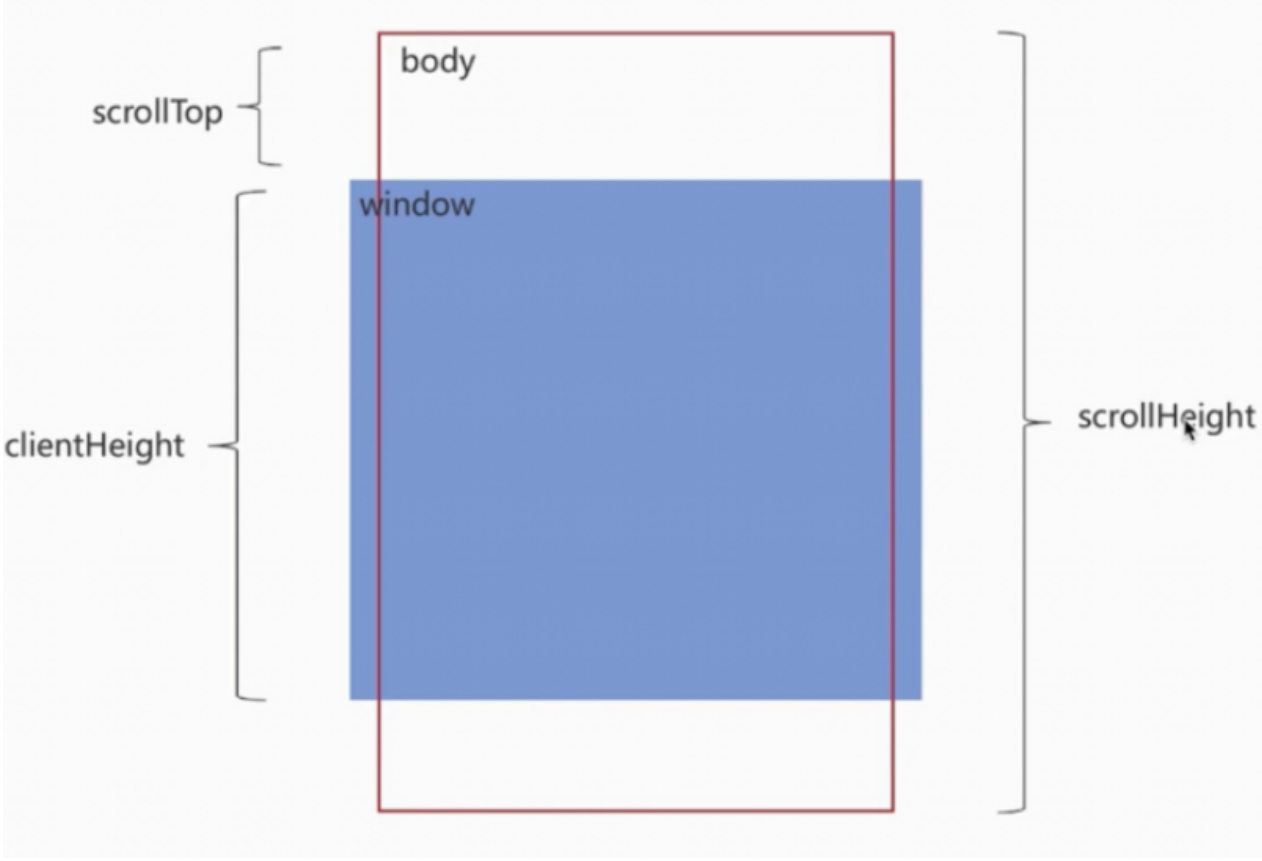
实现原理

上拉加载及下拉刷新都依赖于用户交互

最重要的是要理解在什么场景，什么时机下触发交互动作

上拉加载

首先可以看一张图



上拉加载的本质是页面触底，或者快要触底时的动作

判断页面触底我们需要先了解一下下面几个属性

- `scrollTop`：滚动视窗的高度距离`window`顶部的距离，它会随着往上滚动而不断增加，初始值是0，它是一个变化的值
- `clientHeight`:它是一个定值，表示屏幕可视区域的高度；
- `scrollHeight`：页面不能滚动时也是存在的,此时scrollHeight等于clientHeight。scrollHeight表示`body`所有元素的总长度(包括body元素自身的padding)

综上我们得出一个触底公式：

```
scrollTop + clientHeight >= scrollHeight
```

简单实现

```
let clientHeight  = document.documentElement.clientHeight; //浏览器高度
let scrollHeight = document.body.scrollHeight;
let scrollTop = document.documentElement.scrollTop;

let distance = 50;  //距离视窗还有50的时候，开始触发;

if ((scrollTop + clientHeight) >= (scrollHeight - distance)) {
    console.log("开始加载数据");
}
```

下拉刷新

下拉刷新的本质是页面本身置于顶部时，用户下拉时需要触发的动作

关于下拉刷新的原生实现，主要分成三步：

- 监听原生 `touchstart` 事件，记录其初始位置的值， `e.touches[0].pageY` ；
- 监听原生 `touchmove` 事件，记录并计算当前滑动的位置值与初始位置值的差值，大于 `0` 表示向下拉动，并借助CSS3的 `translateY` 属性使元素跟随手势向下滑动对应的差值，同时也应设置一个允许滑动的最大值；
- 监听原生 `touchend` 事件，若此时元素滑动达到最大值，则触发 `callback` ，同时将 `translateY` 重设为 `0` ，元素回到初始位置

举个例子：

`Html` 结构如下：

```
<main>
    <p class="refreshText"></p >
    <ul id="refreshContainer">
        <li>111</li>
        <li>222</li>
        <li>333</li>
        <li>444</li>
        <li>555</li>
        ...
    </ul>
</main>
```

监听 `touchstart` 事件，记录初始的值

```
var _element = document.getElementById('refreshContainer'),
    _refreshText = document.querySelector('.refreshText'),
    _startPos = 0,  // 初始的值
    _transitionHeight = 0; // 移动的距离

_element.addEventListener('touchstart', function(e) {
    _startPos = e.touches[0].pageY; // 记录初始位置
    _element.style.position = 'relative';
    _element.style.transition = 'transform 0s';
}, false);
```

监听 `touchmove` 移动事件，记录滑动差值

```
 _element.addEventListener('touchmove', function(e) {
     // e.touches[0].pageY 当前位置
     _transitionHeight = e.touches[0].pageY - _startPos; // 记录差值

     if (_transitionHeight > 0 && _transitionHeight < 60) {
         _refreshText.innerText = '下拉刷新';
         _element.style.transform = 'translateY('+_transitionHeight+'px)';

         if (_transitionHeight > 55) {
             _refreshText.innerText = '释放更新';
         }
     }
 }, false);
```

最后，就是监听 `touchend` 离开的事件

```
_element.addEventListener('touchend', function(e) {
    _element.style.transition = 'transform 0.5s ease 1s';
    _element.style.transform = 'translateY(0px)';
    _refreshText.innerText = '更新中...';
    // todo...

}, false);
```

从上面可以看到，在下拉到松手的过程中，经历了三个阶段：

- 当前手势滑动位置与初始位置差值大于零时，提示正在进行下拉刷新操作
- 下拉到一定值时，显示松手释放后的操作提示
- 下拉到达设定最大值松手时，执行回调，提示正在进行更新操作

案例

在实际开发中，我们更多的是使用第三方库，下面以 `better-scroll` 进行举例：

HTML结构

```
<div id="position-wrapper">
    <div>
        <p class="refresh">下拉刷新</p >
        <div class="position-list">
   <!--列表内容-->
        </div>
        <p class="more">查看更多</p >
    </div>
</div>
```

实例化上拉下拉插件，通过 `use` 来注册插件

```
import BScroll from "@better-scroll/core";
import PullDown from "@better-scroll/pull-down";
import PullUp from '@better-scroll/pull-up';
BScroll.use(PullDown);
BScroll.use(PullUp);
```

实例化 `BetterScroll`，并传入相关的参数

```
let pageNo = 1,pageSize = 10,dataList = [],isMore = true;
var scroll= new BScroll("#position-wrapper",{
    scrollY:true,//垂直方向滚动
    click:true,//默认会阻止浏览器的原生click事件，如果需要点击，这里要设为true
    pullUpLoad:true,//上拉加载更多
    pullDownRefresh:{
        threshold:50,//触发pullingDown事件的位置
        stop:0//下拉回弹后停留的位置
    }
});
//监听下拉刷新
scroll.on("pullingDown",pullingDownHandler);
//监测实时滚动
scroll.on("scroll",scrollHandler);
//上拉加载更多
scroll.on("pullingUp",pullingUpHandler);

async function pullingDownHandler(){
    dataList=[];
    pageNo=1;
    isMore=true;
    $(".more").text("查看更多");
    await getlist();//请求数据
    scroll.finishPullDown();//每次下拉结束后，需要执行这个操作
    scroll.refresh();//当滚动区域的dom结构有变化时，需要执行这个操作
}
async function pullingUpHandler(){
    if(!isMore){
        $(".more").text("没有更多数据了");
        scroll.finishPullUp();//每次上拉结束后，需要执行这个操作
        return;
    }
    pageNo++;
    await this.getlist();//请求数据
    scroll.finishPullUp();//每次上拉结束后，需要执行这个操作
    scroll.refresh();//当滚动区域的dom结构有变化时，需要执行这个操作
}
function scrollHandler(){
    if(this.y>50) $('.refresh').text("松手开始加载");
    else $('.refresh').text("下拉刷新");
}
function getlist(){
    //返回的数据
    let result=....;
    dataList=dataList.concat(result);
    //判断是否已加载完
    if(result.length<pageSize) isMore=false;
    //将dataList渲染到html内容中
}
```

注意点：

使用 `better-scroll` 实现下拉刷新、上拉加载时要注意以下几点：

- `wrapper` 里必须只有一个子元素
- 子元素的高度要比 `wrapper` 要高
- 使用的时候，要确定 `DOM` 元素是否已经生成，必须要等到 `DOM` 渲染完成后，再 `new BScroll()`
- 滚动区域的 `DOM` 元素结构有变化后，需要执行刷新 `refresh()`
- 上拉或者下拉，结束后，需要执行 `finishPullUp()` 或者 `finishPullDown()` ，否则将不会执行下次操作
- `better-scroll` ，默认会阻止浏览器的原生 `click` 事件，如果滚动内容区要添加点击事件，需要在实例化属性里设置 `click:true`

小结

下拉刷新、上拉加载原理本身都很简单，真正复杂的是封装过程中，要考虑的兼容性、易用性、性能等诸多细节

2.16 说说ES6新增特性都有哪些?

答：ES6新增特性常用的主要有：let/const,箭头函数，模板字符串，解构赋值，扩展操作符，模块的导入(import)和导出(export default/export),Promise,还有一些数组字符串的扩展方法,其实有很多，我平时常用的就这些

2.17 说说你对作用域链的理解?

答：**一、作用域**

作用域，即变量（变量作用域又称上下文）和函数生效（能被访问）的区域或集合

换句话说，作用域决定了代码区块中变量和其他资源的可见性

举个例子

```
function myFunction() {
    let inVariable = "函数内部变量";
}
myFunction();//要先执行这个函数，否则根本不知道里面是啥
console.log(inVariable); // Uncaught ReferenceError: inVariable is not defined
```

上述例子中，函数 `myFunction` 内部创建一个 `inVariable` 变量，当我们在全局访问这个变量的时候，系统会报错

这就说明我们在全局是无法获取到（闭包除外）函数内部的变量

我们一般将作用域分成：

- 全局作用域
- 函数作用域
- 块级作用域

全局作用域

任何不在函数中或是大括号中声明的变量，都是在全局作用域下，全局作用域下声明的变量可以在程序的任意位置访问

```
// 全局变量
var greeting = 'Hello World!';
function greet() {
  console.log(greeting);
}
// 打印 'Hello World!'
greet();
```

函数作用域

函数作用域也叫局部作用域，如果一个变量是在函数内部声明的它就在一个函数作用域下面。这些变量只能在函数内部访问，不能在函数以外去访问

```
function greet() {
  var greeting = 'Hello World!';
  console.log(greeting);
}
// 打印 'Hello World!'
greet();
// 报错：  Uncaught ReferenceError: greeting is not defined
console.log(greeting);
```

可见上述代码中在函数内部声明的变量或函数，在函数外部是无法访问的，这说明在函数内部定义的变量或者方法只是函数作用域

块级作用域

ES6引入了 `let` 和 `const` 关键字,和 `var` 关键字不同，在大括号中使用 `let` 和 `const` 声明的变量存在于块级作用域中。在大括号之外不能访问这些变量

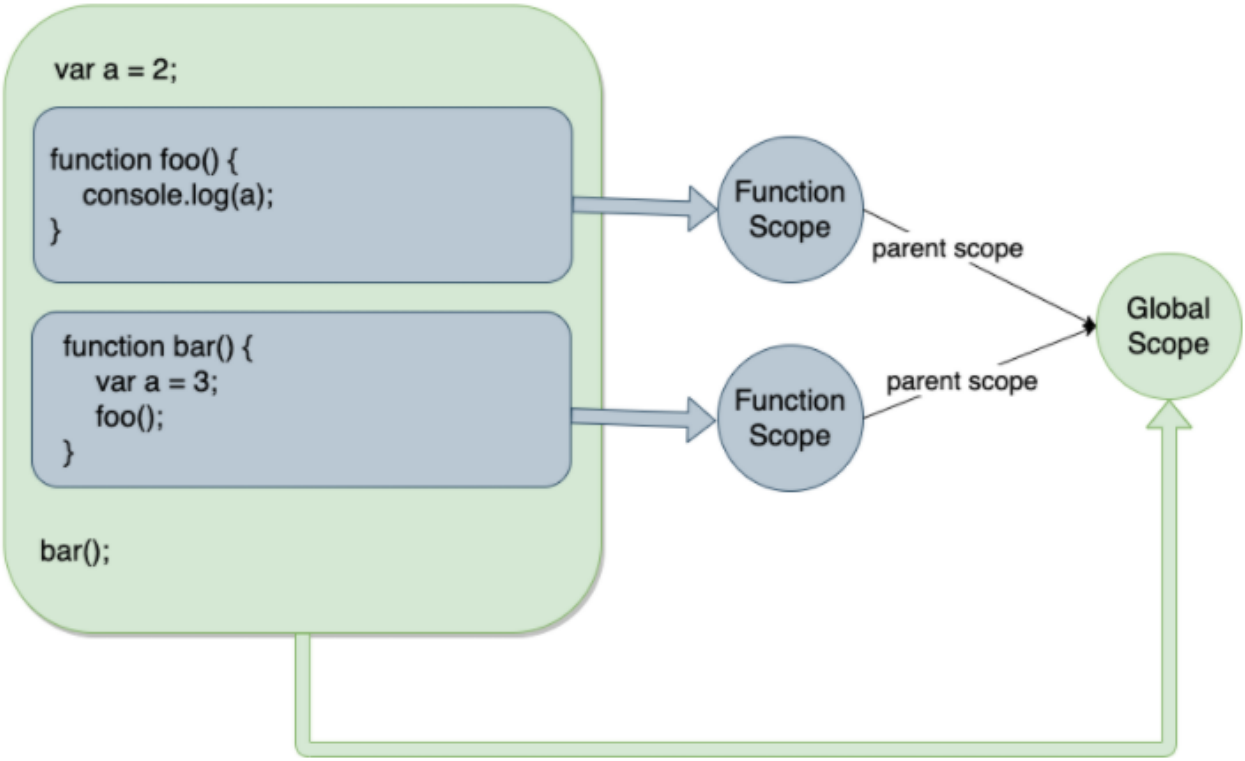
```
{
  // 块级作用域中的变量
  let greeting = 'Hello World!';
  var lang = 'English';
  console.log(greeting); // Prints 'Hello World!'
}
// 变量 'English'
console.log(lang);
// 报错：Uncaught ReferenceError: greeting is not defined
console.log(greeting);
```

词法作用域

词法作用域，又叫静态作用域，变量被创建时就确定好了，而非执行阶段确定的。也就是说我们写好代码时它的作用域就确定了， `JavaScript` 遵循的就是词法作用域

```
var a = 2;
function foo(){
    console.log(a)
}
function bar(){
    var a = 3;
    foo();
}
n()
```

上述代码改变成一张图



由于 `JavaScript` 遵循词法作用域，相同层级的 `foo` 和 `bar` 就没有办法访问到彼此块作用域中的变量，所以输出2

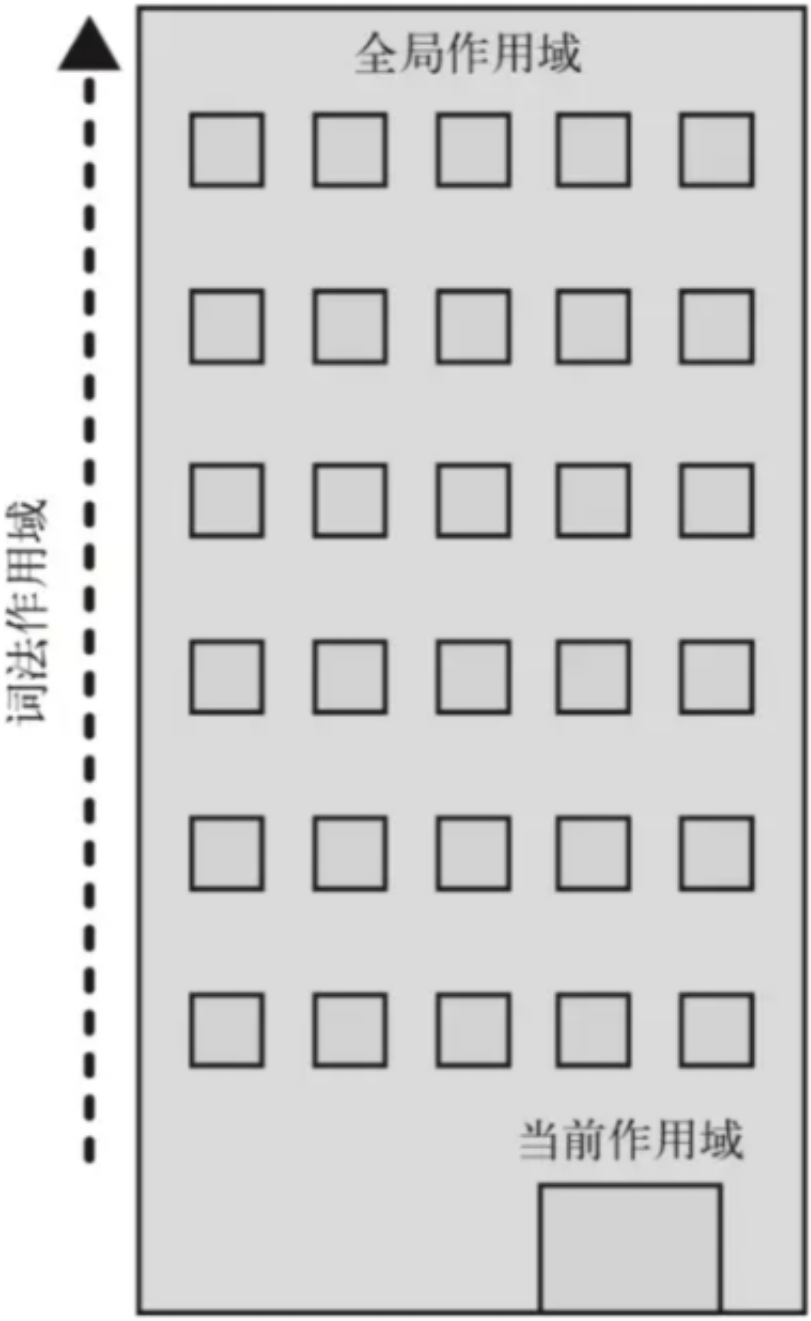
作用域链

当在 `Javascript` 中使用一个变量的时候，首先 `Javascript` 引擎会尝试在当前作用域下去寻找该变量，如果没找到，再到它的上层作用域寻找，以此类推直到找到该变量或是已经到了全局作用域

如果在全局作用域里仍然找不到该变量，它就会在全局范围内隐式声明该变量(非严格模式下)或是直接报错

这里拿《你不知道的Javascript(上)》中的一张图解释：

把作用域比喻成一个建筑，这份建筑代表程序中的嵌套作用域链，第一层代表当前的执行作用域，顶层代表全局作用域



变量的引用会顺着当前楼层进行查找，如果找不到，则会往上一层找，一旦到达顶层，查找的过程都会停止

下面代码演示下：

```
var sex = '男';
function person() {
    var name = '张三';
    function student() {
        var age = 18;
        console.log(name); // 张三
        console.log(sex); // 男
    }
    student();
    console.log(age); // Uncaught ReferenceError: age is not defined
}
person();
```

上述代码主要主要做了以下工作：

- `student` 函数内部属于最内层作用域，找不到 `name` ，向上一层作用域 `person` 函数内部找，找到了输出“张三”
- `student` 内部输出cat时找不到，向上一层作用域 `person` 函数找，还找不到继续向上一层找，即全局作用域，找到了输出“男”
- 在 `person` 函数内部输出 `age` 时找不到，向上一层作用域找，即全局作用域，还是找不到则报错

2.18 谈谈this对象的理解?

答：在函数中this到底取何值，是在函数真正被调用执行的时候确定的，函数定义的时候确定不了。因为this的取值是执行上下文环境的一部分，每次调用函数，都会产生一个新的执行上下文环境。

情况1:构造函数

所谓构造函数就是用来new对象的函数。其实严格来说，所有的函数都可以new一个对象，但是有些函数的定义是为了new一个对象，而有些函数则不是。另外注意，构造函数的函数名第一个字母大写（规则约定）。例如：Object、Array、Function等。

```
function Foo() {
  this.name='haoinli';
  this.year=1988;
  console.log(this); //Foo {name:'haojinli',year1988}

}

var f1=new Foo();
console.log(f1.name); //haoinli
console.log(f1.year); //1988
```

以上代码中，如果函数作为构造函数用，那么其中的this就代表它即将new出来的对象。

注意，以上仅限new Foo()的情况，即Foo函数作为构造函数的情况。如果直接调用Foo函数，而不是new Foo()，情况就大不一样了。

```
function Foo() {
  this.name='haoinli';
  this.year=1988;
  console.log(this); //window

}

Foo();
```

这种情况下this是window，我们后文中会说到。

情况2:函数作为对象的一个属性

如果函数作为对象的一个属性时，并且作为对象的一个属性被调用时，函数中的this指向该对象。

```
var obj = {
    x: 10,
    fn: function () {
        console.log(this);    // Object {x: 10, fn: function}
        console.log(this.x);  // 10
    }
};

obj.fn();
```

以上代码中，fn不仅作为一个对象的一个属性，而且的确是作为对象的一个属性被调用。结果this就是obj对象。

注意，如果fn函数不作为obj的一个属性被调用，会是什么结果呢?

```
var obj = {
    x: 10,
    fn: function () {
        console.log(this);    // Window {top: Window, window: Wi
        console.log(this.x);  // undefined
    }
};

var fn1 = obj.fn;
fn1();
```

如上代码，如果fn函数被赋值到了另一个变量中，并没有作为obj的一个属性被调用，那么this的值就是window，this.x为undefined。

情况3：函数用call或者apply调用

当一个函数被call和apply调用时，this的值就取传入的对象的值。至于call和apply如何使用，不会的朋友可以去查查其他资料，本系列教程不做讲解。

```
var obj = {
    x: 10
};

var fn = function () {
    console.log(this);     //Object {x: 10}
    console.log(this.x);   //10
}
fn.call(obj);
```

情况4：全局 & 调用普通函数

在全局环境下，this永远是window，这个应该没有非议。

```
console.log(this === window);  // true
```

普通函数在调用时，其中的this也都是window。

```
var x = 10;

var fn = function () {
    console.log(this);     //Window {top: Window, window: Wi
    console.log(this.x);   //10
}
fn();
```

以上代码很好理解。

不过下面的情况你需要注意一下：

```
var obj = {
    x: 10,
    fn: function () {

        function f() {
            console.log(this);    //Window {top: Window, window: Wind
            console.log(this.x); //undefined
        }
        f();

    }
};
obj.fn();
```

函数f虽然是在obj.fn内部定义的，但是它仍然是一个普通的函数，this仍然指向window

2.19 你是怎么理解ES6中 Promise的？使用场景？

答：`Promise`，译为承诺，是异步编程的一种解决方案，比传统的解决方案（回调函数）更加合理和更加强大

在以往我们如果处理多层异步操作，我们往往会像下面那样编写我们的代码

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('得到最终结果: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

阅读上面代码，是不是很难受，上述形成了经典的回调地狱

现在通过 `Promise` 的改写上面的代码

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('得到最终结果: ' + finalResult);
})
.catch(failureCallback);
```

瞬间感受到 `promise` 解决异步操作的优点：

- 链式操作减低了编码难度
- 代码可读性明显增强

下面我们正式来认识 `promise` ：

状态

`promise` 对象仅有三种状态

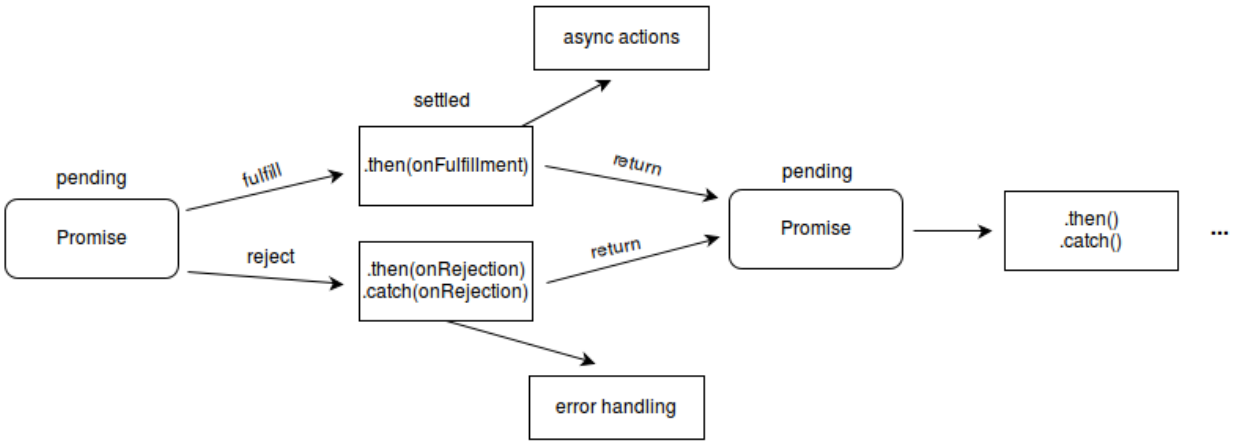
- `pending` （进行中）
- `fulfilled` （已成功）
- `rejected` （已失败）

特点

- 对象的状态不受外界影响，只有异步操作的结果，可以决定当前是哪一种状态
- 一旦状态改变（从 `pending` 变为 `fulfilled` 和从 `pending` 变为 `rejected` ），就不会再变，任何时候都可以得到这个结果

流程

认真阅读下图，我们能够轻松了解 `promise` 整个流程



用法

`Promise` 对象是一个构造函数，用来生成 `Promise` 实例

```
const promise = new Promise(function(resolve, reject) {});
```

`Promise` 构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`

- `resolve` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“成功”
- `reject` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“失败”

实例方法

`Promise` 构建出来的实例存在以下方法：

- then()
- then()
- catch()
- finally()

then()

`then` 是实例状态发生改变时的回调函数，第一个参数是 `resolved` 状态的回调函数，第二个参数是 `rejected` 状态的回调函数

`then` 方法返回的是一个新的 `Promise` 实例，也就是 `promise` 能链式书写的原因

```
getJSON("/posts.json").then(function(json) {
  return json.post;
}).then(function(post) {
  // ...
});
```

catch

`catch()` 方法是 `.then(null, rejection)` 或 `.then(undefined, rejection)` 的别名，用于指定发生错误时的回调函数

```
getJSON('/posts.json').then(function(posts) {
  // ...
}).catch(function(error) {
  // 处理 getJSON 和 前一个回调函数运行时发生的错误
  console.log('发生错误! ', error);
});
```

`Promise` 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止

```
getJSON('/post/1.json').then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});
```

一般来说，使用 `catch` 方法代替 `then()` 第二个参数

`Promise` 对象抛出的错误不会传递到外层代码，即不会有任何反应

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};
```

浏览器运行到这一行，会打印出错误提示 `ReferenceError: x is not defined` ，但是不会退出进程

`catch()` 方法之中，还能再抛出错误，通过后面 `catch` 方法捕获到

finally()

`finally()` 方法用于指定不管 Promise 对象最后状态如何，都会执行的操作

```
promise
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
```

构造函数方法

`Promise` 构造函数存在以下方法：

- all()
- race()
- allSettled()
- resolve()
- reject()
- try()

all()

`Promise.all()` 方法用于将多个 `Promise` 实例，包装成一个新的 `Promise` 实例

```
const p = Promise.all([p1, p2, p3]);
```

接受一个数组（迭代对象）作为参数，数组成员都应为 `Promise` 实例

实例 `p` 的状态由 `p1` 、 `p2` 、 `p3` 决定，分为两种：

- 只有 `p1` 、 `p2` 、 `p3` 的状态都变成 `fulfilled` ， `p` 的状态才会变成 `fulfilled` ，此时 `p1` 、 `p2` 、 `p3` 的返回值组成一个数组，传递给 `p` 的回调函数
- 只要 `p1` 、 `p2` 、 `p3` 之中有一个被 `rejected` ， `p` 的状态就变成 `rejected` ，此时第一个被 `reject` 的实例的返回值，会传递给 `p` 的回调函数

注意，如果作为参数的 `Promise` 实例，自己定义了 `catch` 方法，那么它一旦被 `rejected` ，并不会触发 `Promise.all()` 的 `catch` 方法

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result)
.catch(e => e);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result)
.catch(e => e);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// ["hello", Error: 报错了]
```

如果 `p2` 没有自己的 `catch` 方法，就会调用 `Promise.all()` 的 `catch` 方法

```
const p1 = new Promise((resolve, reject) => {
  resolve('hello');
})
.then(result => result);

const p2 = new Promise((resolve, reject) => {
  throw new Error('报错了');
})
.then(result => result);

Promise.all([p1, p2])
.then(result => console.log(result))
.catch(e => console.log(e));
// Error: 报错了
```

race()

`Promise.race()` 方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例

```
const p = Promise.race([p1, p2, p3]);
```

只要 `p1` 、 `p2` 、 `p3` 之中有一个实例率先改变状态， `p` 的状态就跟着改变

率先改变的 Promise 实例的返回值则传递给 `p` 的回调函数

```
const p = Promise.race([
  fetch('/resource-that-may-take-a-while'),
  new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error('request timeout')), 5000)
  })
]);

p
.then(console.log)
.catch(console.error);
```

allSettled()

`Promise.allSettled()` 方法接受一组 Promise 实例作为参数，包装成一个新的 Promise 实例

只有等到所有这些参数实例都返回结果，不管是 `fulfilled` 还是 `rejected` ，包装实例才会结束

```
const promises = [
  fetch('/api-1'),
  fetch('/api-2'),
  fetch('/api-3'),
];

await Promise.allSettled(promises);
removeLoadingIndicator();
```

resolve()

将现有对象转为 `Promise` 对象

```
Promise.resolve('foo')
// 等价于
new Promise(resolve => resolve('foo'))
```

参数可以分成四种情况，分别如下：

- 参数是一个 Promise 实例， `promise.resolve` 将不做任何修改、原封不动地返回这个实例
- 参数是一个 `thenable` 对象， `promise.resolve` 会将这个对象转为 `Promise` 对象，然后就立即执行 `thenable` 对象的 `then()` 方法
- 参数不是具有 `then()` 方法的对象，或根本就不是对象， `Promise.resolve()` 会返回一个新的 Promise 对象，状态为 `resolved`
- 没有参数时，直接返回一个 `resolved` 状态的 Promise 对象

reject()

`Promise.reject(reason)` 方法也会返回一个新的 Promise 实例，该实例的状态为 `rejected`

```
const p = Promise.reject('出错了');
// 等同于
const p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s) {
  console.log(s)
});
// 出错了
```

`Promise.reject()` 方法的参数，会原封不动地变成后续方法的参数

```
Promise.reject('出错了')
.catch(e => {
  console.log(e === '出错了')
})
// true
```

使用场景

将图片的加载写成一个 `Promise` ，一旦加载完成， `Promise` 的状态就发生变化

```
const preloadImage = function (path) {
  return new Promise(function (resolve, reject) {
    const image = new Image();
    image.onload  = resolve;
    image.onerror = reject;
    image.src = path;
  });
};
```

通过链式操作，将多个渲染数据分别给个 `then` ，让其各司其职。或当下个异步请求依赖上个请求结果的时候，我们也能够通过链式操作友好解决问题

```
 // 各司其职
 getInfo().then(res=>{
     let { bannerList } = res
     //渲染轮播图
     console.log(bannerList)
     return res
 }).then(res=>{

     let { storeList } = res
     //渲染店铺列表
     console.log(storeList)
     return res
 }).then(res=>{
     let { categoryList } = res
     console.log(categoryList)
     //渲染分类列表
     return res
 })
```

通过 `all()` 实现多个请求合并在一起，汇总所有请求结果，只需设置一个 `loading` 即可

```
function initLoad(){
    // loading.show() //加载loading
    Promise.all([getBannerList(),getStoreList(),getCategoryList()]).then(res=>{
        console.log(res)
        loading.hide() //关闭loading
    }).catch(err=>{
        console.log(err)
        loading.hide()//关闭loading
    })
}
//数据初始化
initLoad()
```

通过 `race` 可以设置图片请求超时

```
//请求某个图片资源
function requestImg(){
    var p = new Promise(function(resolve, reject){
        var img = new Image();
        img.onload = function(){
           resolve(img);
        }
        //img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg";
正确的
        img.src = "https://b-gold-cdn.xitu.io/v3/static/img/logo.a7995ad.svg1";
    });
    return p;
}

//延时函数，用于给请求计时
function timeout(){
    var p = new Promise(function(resolve, reject){
        setTimeout(function(){
            reject('图片请求超时');
        }, 5000);
    });
    return p;
}

Promise
.race([requestImg(), timeout()])
.then(function(results){
    console.log(results);
})
.catch(function(reason){
    console.log(reason);
});

```

2.20 说说var、let、const之间的区别?

答： 一、var

在ES5中，顶层对象的属性和全局变量是等价的，用 `var` 声明的变量既是全局变量，也是顶层变量

注意：顶层对象，在浏览器环境指的是 `window` 对象，在 `Node` 指的是 `global` 对象

```
var a = 10;
console.log(window.a) // 10
```

使用 `var` 声明的变量存在变量提升的情况

```
console.log(a) // undefined
var a = 20
```

在编译阶段，编译器会将其变成以下执行

```
var a
console.log(a)
a = 20
```

使用 `var` ，我们能够对一个变量进行多次声明，后面声明的变量会覆盖前面的变量声明

```
var a = 20
var a = 30
console.log(a) // 30
```

在函数中使用使用 `var` 声明变量时候，该变量是局部的

```
var a = 20
function change(){
    var a = 30
}
change()
console.log(a) // 20
```

而如果在函数内不使用 `var` ，该变量是全局的

```
var a = 20
function change(){
   a = 30
}
change()
console.log(a) // 30
```

二、let

`let` 是 `ES6` 新增的命令，用来声明变量

用法类似于 `var` ，但是所声明的变量，只在 `let` 命令所在的代码块内有效

```
{
    let a = 20
}
console.log(a) // ReferenceError: a is not defined.
```

不存在变量提升

```
console.log(a) // 报错ReferenceError
let a = 2
```

这表示在声明它之前，变量 `a` 是不存在的，这时如果用到它，就会抛出一个错误

只要块级作用域内存在 `let` 命令，这个区域就不再受外部影响

```
var a = 123
if (true) {
    a = 'abc' // ReferenceError
    let a;
}
```

使用 `let` 声明变量前，该变量都不可用，也就是大家常说的“暂时性死区”

最后， `let` 不允许在相同作用域中重复声明

```
let a = 20
let a = 30
// Uncaught SyntaxError: Identifier 'a' has already been declared
```

注意的是相同作用域，下面这种情况是不会报错的

```
let a = 20
{
    let a = 30
}
```

因此，我们不能在函数内部重新声明参数

```
function func(arg) {
  let arg;
}
func()
// Uncaught SyntaxError: Identifier 'arg' has already been declared
```

三、const

`const` 声明一个只读的常量，一旦声明，常量的值就不能改变

```
const a = 1
a = 3
// TypeError: Assignment to constant variable.
```

这意味着， `const` 一旦声明变量，就必须立即初始化，不能留到以后赋值

```
const a;
// SyntaxError: Missing initializer in const declaration
```

如果之前用 `var` 或 `let` 声明过变量，再用 `const` 声明同样会报错

```
var a = 20
let b = 20
const a = 30
const b = 30
// 都会报错
```

`const` 实际上保证的并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动

对于简单类型的数据，值就保存在变量指向的那个内存地址，因此等同于常量

对于复杂类型的数据，变量指向的内存地址，保存的只是一个指向实际数据的指针，`const` 只能保证这个指针是固定的，并不能确保改变量的结构不变

```
const foo = {};

// 为 foo 添加一个属性，可以成功
foo.prop = 123;
foo.prop // 123

// 将 foo 指向另一个对象，就会报错
foo = {}; // TypeError: "foo" is read-only
```

其它情况，`const` 与 `let` 一致

四、区别

`var` 、`let` 、`const` 三者区别可以围绕下面五点展开：

- 变量提升
- 暂时性死区
- 块级作用域
- 重复声明
- 修改声明的变量
- 使用

变量提升

`var` 声明的变量存在变量提升，即变量可以在声明之前调用，值为 `undefined`

`let` 和 `const` 不存在变量提升，即它们所声明的变量一定要在声明后使用，否则报错

```
// var
console.log(a)  // undefined
var a = 10

// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10

// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

暂时性死区

`var` 不存在暂时性死区

`let` 和 `const` 存在暂时性死区，只有等到声明变量的那一行代码出现，才可以获取和使用该变量

```
// var
console.log(a)  // undefined
var a = 10

// let
console.log(b)  // Cannot access 'b' before initialization
let b = 10

// const
console.log(c)  // Cannot access 'c' before initialization
const c = 10
```

块级作用域

`var` 不存在块级作用域

`let` 和 `const` 存在块级作用域

```
// var
{
    var a = 20
}
console.log(a)  // 20

// let
{
    let b = 20
}
console.log(b)  // Uncaught ReferenceError: b is not defined

// const
{
    const c = 20
}
console.log(c)  // Uncaught ReferenceError: c is not defined
```

重复声明

`var` 允许重复声明变量

`let` 和 `const` 在同一作用域不允许重复声明变量

```
// var
var a = 10
var a = 20 // 20

// let
let b = 10
let b = 20 // Identifier 'b' has already been declared

// const
const c = 10
const c = 20 // Identifier 'c' has already been declared
```

修改声明的变量

`var` 和 `let` 可以

`const` 声明一个只读的常量。一旦声明，常量的值就不能改变

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

//let
let b = 10
b = 20
console.log(b)  // 20

// const
const c = 10
c = 20
console.log(c) // Uncaught TypeError: Assignment to constant variable
```

使用

能用 `const` 的情况尽量使用 `const` ，其他情况下大多数使用 `let` ，避免使用 `var`

2.21 说说javaScript原生有几种绑定事件方式？具体如何实现？

```
答：JS原生绑定事件主要为三种：
    一是html事件处理程序
    二是DOM0级事件处理程序
三是DOM2级事件处理程序
        其中：html事件现在早已不用了，就是在html各种标签上直接添加事件，类似于css的行内样式，缺点是不好维护，因为散落在标签中,也就是耦合度太高
例如：<button onclick=”事件处理函数”>点我</button>
```

```
第二类是DOM0级事件，目前在PC端用的还是比较多的绑定事件方式，兼容性也好，主要是先获取dom元素，然后直接给dom元素添加事件
例如：var btn=document.getElementById(‘id元素’)
      btn.onclick=function() {
         //要处理的事件逻辑
      }

    DOM0事件如何移除呢？很简单：btn.onclick=null;置为空就行
   优点：兼容性好
   缺点：只支持冒泡，不支持捕获
第三类是DOM2级事件，移动端用的比较多，也有很多优点，提供了专门的绑定和移除方法
例如：  var btn=document.getElementById(‘id元素’)
        //绑定事件
     btn.addEventListener(‘click’,绑定的事件处理函数名,false)
     //移除事件
     btn.removeEventListener(‘click’,要移除的事件处理函数名，false)

    优点：支持给个元素绑定多个相同事件，支持冒泡和捕获事件机制
```

2.22 说说jQuery有哪些选择器和常用查找方法?

答：jQuery选择器有：

```
(1)基本
#id
element
.class
*
selector1,selector2,selectorN

(2)层次选择器:
ancestor descendant
parent > child
prev + next
prev ~ siblings

(3)基本过滤器选择器
:first
:last
:not
:even
:odd
:eq
:gt
:lt
:header
:animated

(4)内容过滤器选择器
:contains
```

```
  :empty
  :has
  :parent

  (5)可见性过滤器选择器
  :hidden
  :visible

  (6)属性过滤器选择器
  [attribute]
  [attribute=value]
  [attribute!=value]
  [attribute^=value]
  [attribute$=value]
  [attribute*=value]
  [attrSel1][attrSel2][attrSelN]

  (7)子元素过滤器选择器
  :nth-child
  :first-child
  :last-child
  :only-child

  (8)表单选择器
  :input
  :text
  :password
  :radio
  :checkbox
  :submit
  :image
  :reset
  :button
  :file
  :hidden

  (9)表单过滤器选择器
  :enabled
  :disabled
  :checked
  :selected
```

常用查找方法有：

```
find()
children()
filter
eq()
parent()
parents()
cloest()
....
```

2.23 判断JS数据类型的方法有哪些?

答：在 ECMAScript 规范中，共定义了 7 种数据类型，分为 基本类型 和 引用类型 两大类，如下所示：

> **基本类型**：String、Number、Boolean、Symbol、Undefined、Null
>
> **引用类型**：Object

基本类型也称为简单类型，由于其占据空间固定，是简单的数据段，为了便于提升变量查询速度，将其存储在栈中，即按值访问。

引用类型也称为复杂类型，由于其值的大小会改变，所以不能将其存放在栈中，否则会降低变量查询速度，因此，其值存储在堆(heap)中，而存储在变量处的值，是一个指针，指向存储对象的内存处，即按址访问。引用类型除 Object 外，还包括 Function 、Array、RegExp、Date 等等。

鉴于 ECMAScript 是松散类型的，因此需要有一种手段来检测给定变量的数据类型。对于这个问题，JavaScript 也提供了多种方法，但遗憾的是，不同的方法得到的结果参差不齐。

下面介绍常用的4种方法，并对各个方法存在的问题进行简单的分析。

typeof

typeof 是一个操作符，其右侧跟一个一元表达式，并返回这个表达式的数据类型。返回的结果用该类型的字符串(全小写字母)形式表示，包括以下 7 种：number、boolean、symbol、string、object、undefined、function 等。

```
typeof '';// string 有效
typeof 1;// number 有效
typeof Symbol();// symbol 有效
typeof true;//boolean 有效
typeof undefined;//undefined 有效
typeof null;//object 无效
typeof [] ;//object 无效
typeof new Function();// function 有效
typeof new Date();//object 无效
typeof new RegExp();//object 无效
```

有些时候，typeof 操作符会返回一些令人迷惑但技术上却正确的值：

- 对于基本类型，除 null 以外，均可以返回正确的结果。
- 对于引用类型，除 function 以外，一律返回 object 类型。
- 对于 null ，返回 object 类型。
- 对于 function 返回 function 类型。

其中，null 有属于自己的数据类型 Null ， 引用类型中的 数组、日期、正则 也都有属于自己的具体类型，而 typeof 对于这些类型的处理，只返回了处于其原型链最顶端的 Object 类型，没有错，但不是我们想要的结果。

instanceof

instanceof 是用来判断 A 是否为 B 的实例，表达式为：A instanceof B，如果 A 是 B 的实例，则返回 true，否则返回 false。 在这里需要特别注意的是：**instanceof 检测的是原型**，我们用一段伪代码来模拟其内部执行过程：

```
instanceof (A,B) = {
    var L = A.__proto__;
    var R = B.prototype;
    if(L === R) {
        // A的内部属性 __proto__ 指向 B 的原型对象
        return true;
    }
    return false;
}
```

从上述过程可以看出，当 A 的 **proto** 指向 B 的 prototype 时，就认为 A 就是 B 的实例，我们再来看几个例子：

```
[] instanceof Array;// true
{} instanceof Object;// true
new Date() instanceof Date;// true

function Person(){};
new Person() instanceof Person;

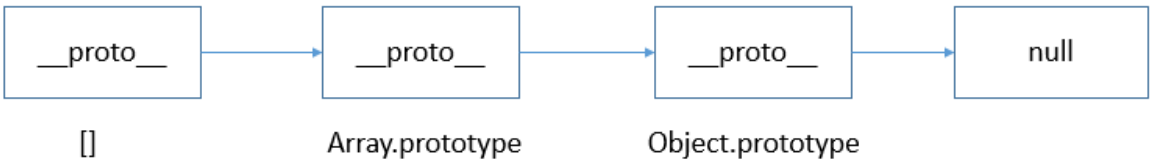
[] instanceof Object;// true
new Date() instanceof Object;// true
new Person instanceof Object;// true

```

我们发现，虽然 instanceof 能够判断出 [ ] 是Array的实例，但它认为 [ ] 也是Object的实例，为什么呢?

我们来分析一下 [ ]、Array、Object 三者之间的关系：

从 instanceof 能够判断出 [ ].**proto** 指向 Array.prototype，而 Array.prototype.**proto** 又指向了 Object.prototype，最终 Object.prototype.**proto** 指向了null，标志着原型链的结束。因此，[]、Array、Object 就在内部形成了一条原型链：



从原型链可以看出，[] 的 **proto** 直接指向Array.prototype，间接指向 Object.prototype，所以按照 instanceof 的判断规则，[] 就是Object的实例。依次类推，类似的 new Date()、new Person() 也会形成一条对应的原型链 。因此，**instanceof 只能用来判断两个对象是否属于实例关系**，而不能判断一个对象实例具体属于哪种类型。**

instanceof 操作符的问题在于，它假定只有一个全局执行环境。如果网页中包含多个框架，那实际上就存在两个以上不同的全局执行环境，从而存在两个以上不同版本的构造函数。如果你从一个框架向另一个框架传入一个数组，那么传入的数组与在第二个框架中原生创建的数组分别具有各自不同的构造函数。

```
var iframe = document.createElement('iframe');
document.body.appendChild(iframe);
xArray = window.frames[0].Array;
var arr =new xArray(1,2,3);// [1,2,3]
arr instanceof Array;// false
```

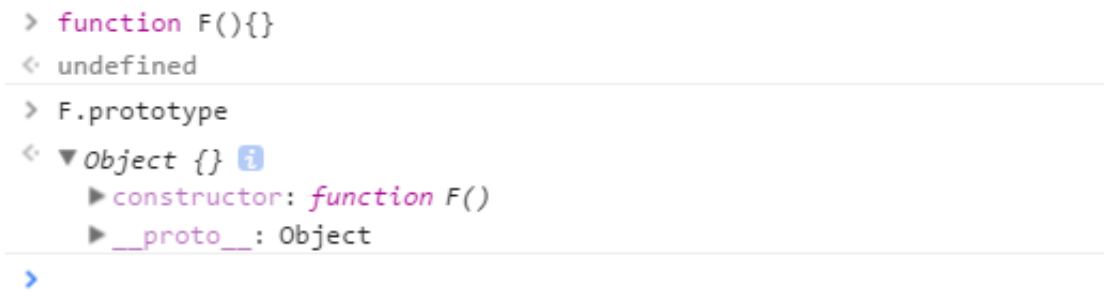
针对数组的这个问题，ES5 提供了 Array.isArray() 方法 。该方法用以确认某个对象本身是否为 Array 类型，而不区分该对象在哪个环境中创建。

```
if (Array.isArray(value)){
   //对数组执行某些操作
}
```

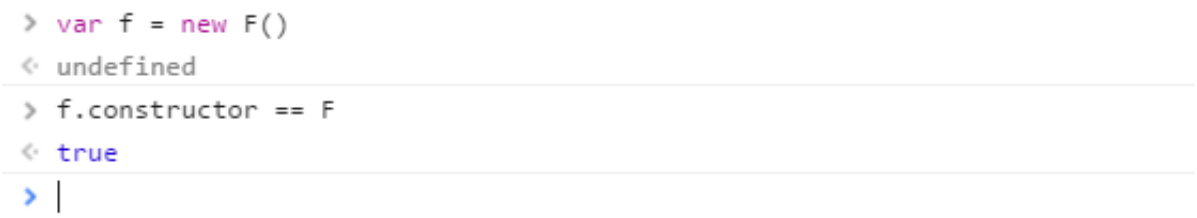
Array.isArray() 本质上检测的是对象的 [[Class]] 值，[[Class]] 是对象的一个内部属性，里面包含了对象的类型信息，其格式为 [object Xxx] ，Xxx 就是对应的具体类型 。对于数组而言，[[Class]] 的值就是 [object Array] 。

constructor

当一个函数 F被定义时，JS引擎会为F添加 prototype 原型，然后再在 prototype上添加一个 constructor 属性，并让其指向 F 的引用。如下所示：



当执行 var f = new F() 时，F 被当成了构造函数，f 是F的实例对象，此时 F 原型上的 constructor 传递到了 f 上，因此 f.constructor == F



可以看出，F 利用原型对象上的 constructor 引用了自身，当 F 作为构造函数来创建对象时，原型上的 constructor 就被遗传到了新创建的对象上， 从原型链角度讲，构造函数 F 就是新对象的类型。这样做的意义是，让新对象在诞生以后，就具有可追溯的数据类型。

同样，JavaScript 中的内置对象在内部构建时也是这样做的：

```
> ''.constructor == String
true
> new Number(1).constructor == Number
true
> true.constructor == Boolean
true
> new Function().constructor == Function
true
> new Date().constructor == Date
true
> new Error().constructor == Error
true
> [].constructor == Array
true
> document.constructor == HTMLDocument
true
> window.constructor == Window
true
>
```

细节问题：

> \1. null 和 undefined 是无效的对象，因此是不会有 constructor 存在的，这两种类型的数据需要通过其他方式来判断。
>
> \2. 函数的 constructor 是不稳定的，这个主要体现在自定义对象上，当开发者重写 prototype 后，原有的 constructor 引用会丢失，constructor 会默认为 Object

```
> function F(){}
undefined
> F.prototype = {a:'xxxx'}
Object {a: "xxxx"}
> var f = new F()
undefined
> f.constructor == F
false
> f.constructor
function Object() { [native code] }
>
```

为什么变成了 Object?

因为 prototype 被重新赋值的是一个 { }， { } 是 new Object() 的字面量，因此 new Object() 会将 Object 原型上的 constructor 传递给 { }，也就是 Object 本身。

因此，为了规范开发，在重写对象原型时一般都需要重新给 constructor 赋值，以保证对象实例的类型不被篡改。

toString

toString() 是 Object 的原型方法，调用该方法，默认返回当前对象的 [[Class]] 。这是一个内部属性，其格式为 [object Xxx] ，其中 Xxx 就是对象的类型。

对于 Object 对象，直接调用 toString() 就能返回 [object Object] 。而对于其他对象，则需要通过 call / apply 来调用才能返回正确的类型信息。

```
Object.prototype.toString.call('') ;  // [object String]
Object.prototype.toString.call(1) ;   // [object Number]
Object.prototype.toString.call(true) ;// [object Boolean]
Object.prototype.toString.call(Symbol());//[object Symbol]
Object.prototype.toString.call(undefined) ;// [object Undefined]
Object.prototype.toString.call(null) ;// [object Null]
Object.prototype.toString.call(new Function()) ;// [object Function]
Object.prototype.toString.call(new Date()) ;// [object Date]
Object.prototype.toString.call([]) ;// [object Array]
Object.prototype.toString.call(new RegExp()) ;// [object RegExp]
Object.prototype.toString.call(new Error()) ;// [object Error]
Object.prototype.toString.call(document) ;// [object HTMLDocument]
Object.prototype.toString.call(window) ;//[object global] window 是全局对象 global 的引用
```

三、ES6面试题

3.1 说一下你对promise的理解?(必问)

3.1.1 什么是promise?通过promise能够解决什么问题?

3.1.2 说一下promise的特点?也就是三种状态?

3.1.3 说一下promise怎么用?

3.1.4 在说一下promise的all方法和race方法?

3.1.5 在说一下在项目中使用promise做过什么?

> 首先promise是es6提供一种异步解决方案. 通过promise能够解决回调地狱问题. 所谓的这个回调地狱指的当我们执行完一个操作之后在接受着操作的结果只能通过回调函数的方式进行接受,使用回调函数的方式存在的弊端就是写法非常臃肿,并且后期难以维护,所谓es6给我提供了一种新的解决方案,就是promise来进行解决,promise可以通过链式调用的方式来解决层层嵌套的问题,但是写法上也不是非常好,所以我们最终的替代方案是使用async和await
>
> promise一共有三个状态,分别是进行中,成功或者失败 如何成功的话可以通过resolve方法将正确结果返回出去,通过.then的方式进行接受,失败的话可以通过reject的方式将失败的结果返回出去,通过.catch的方式进行接受,pending状态是进行中,一旦进行之后,他的状态是不可逆的
>
> 如果要使用promise,我们需要对promise进行实例化,在实例化的构造函数里面有一个回调函数,这个回调函数里面有那个参数,分别是resolve和reject,我们可以通过promise的实例化对象调用.then或者.catch方式接受结果

promise还给我们提供了.all 和 race, 其中all方法的作用是将多个请求合并成一个请求, 比如当首页要请求10个接口,我们可以promise.all进行合并,.race的作用也可以将多个请求合并成一个请求,不过是谁先请求成功就先返回谁.

我在项目中经常使用promise对api 截形封装以及如何a页面要获取到b页面的结果,我们也可以结果promise来完成

以上就是我对promise 的理解.

3.2 说一下async和await、以及他们和promise的区别?(必问)

首先async和await是解决异步的终极方案,他是generatal的语法糖. async和await一般配和使用,当我们给函数前面加上关键字async,这个时候,这个函数的返回值就是一个promise. 而await是一个同步的操作,await只能配合async只能,不然会报错,await后面可以是表达式,也可以是一个promise,在await下面的代码必须得等待await执行完之后才能在执行

他们和promise的区别就是在写法上更加的简洁.

以上就是我对async和await的理解.

3.3 说一下es6新增的特性有那些?(必问)

1. 新增了变量声明方式,也就是let和const
2. 新增了解构赋值
3. 新增了一个数组方法 字符串方法 正则表达的方法 函数的一些写法 对象的方法
4. promise
5. async await
6. class 以及 继承
7. 模块化
8. 新的数据类型
9. 大概能想到暂时只有这么多,在项目中我经常使用let和const 箭头函数。解构赋值 promise 还有 async await

3.4 说一下数组新增的方法有哪些?这些方法分别是什么意思?(必问)

Array.from方法 将类数组转化为真正的数组

Array.of方法 将数值转化为数组

copyWithin() 方法是将指定位置的成员复制到其他位置（会覆盖原有成员）

find方法和findindex方法 查找符合条件的元素。 查找符合条件元素的下表

includes方法可以查看表个数组是否包含给定的值

flat方法将多维数组转化为1维数组或者指定维度的数组

遍历的数组的方法:

forEach 类似于for循环,主要是用来遍历数组

> map方法 主要作用是映射一个新的数组,可以对数组进行遍历
>
> filter方法 主要作用返回一个符合条件新的数据,同样也有遍历数组的作用

3.5 说一下map方法、forEach方法、filter方法的作用以及他们之间的区别?(90%)

> **forEach() 方法: 循环原来的数组**
>
> **map() 方法： 循环原数组并映射一个新数组出来**
>
> **filter() 方法： 过滤不需要的数组元素**

3.6 说一下var 、let、const之间的区别?(95%)

> - `var` 存在提升，我们能在声明之前使用。 `let` 、 `const` 因为暂时性死区的原因，不能在声明前使用
> - `var` 在全局作用域下声明变量会导致变量挂载在 `window` 上，其他两者不会
> - `let` 和 `const` 作用基本一致，但是后者声明的变量不能再次赋值。

3.7 说一下箭头函数与普通函数的区别?(80%)

> 在es6中，提供了一种简洁的函数写法，我们称作“箭头函数”。
>
> 写法：函数名=(形参)=>{……} 当函数体中只有一个表达式时，{}和return可以省略，当函数体中形参只有一个时，()可以省略。
>
> 特点：箭头函数中的this始终指向箭头函数定义时的离this最近的一个函数，如果没有最近的函数就指向window。
>
> 区别:
>
> 1. 箭头函数不能用于构造函数，不能使用new** 而普通函数可以
> 2. 在普通函数中，this总是指向调用它的对象，如果用作构造函数，this指向创建的对象实例，而箭头函数指向箭头函数定义时的离this最近的一个函数，如果没有最近的函数就指向window。

3.8 说一下for in 与for of的区别?(80%)

> For in可以遍历对象 而 for of遍历对象会报错
>
> for in 遍历数组得到的数组的下表 而for of遍历得到的时候数组里面的每一个元素

3.9 说一下es6如何实现类以及如何实现类的继承?(80%)

> es6提供了类的这个概念,在es5中是没有类的这个概念,如果想在es5中实现一个类型,我们只能构造函数的方式去创建一个类,而es6给我们提供一个更方便 的方.法,那就是class,这个class理解为是构造函数的语法糖.
>
> 我们创建一个类只需要用过关键词class去声明就可以了 , 他的调用方式和构造函数的调用方式是一样的
>
> 通过es6的类还给我们提供一个extends这样的一个关键字,来实现继承
>
> 以上就是我对类的理解

3.10 说一下数组去重的方法有哪些?es6如何实现数组去重?(90%)

> indexof
>
> 双层for循环
>
> set方法

3.11 说一下如何检测对象里面有没有属性(或者如何检测一个对象是否为空)?以及如何获取对象里面所有的属性名?(70%)

> 通过object.keys方法, 返回值数组,数组里面包含的是所有属性名
>
> **Object.hasOwnProperty()**
>
> 使用for in的方式

3.12 如何将多个数组合并成为一个数组?(70%)

> es5 :
>
> concat
>
> for循环
>
> Es6:
>
> 扩展运算符
>
> map方法

3.13 说一下forEach、map、filter、reduce、some、every等方法的作用?(50%)

> reduce 遍历数据求和。
>
> some 遍历数组每一项，有一项返回true,则停止遍历，结果返回true。不改变原数组
>
> 遍历数组每一项，每一项返回true,则最终结果为true。当任何一项返回false时，停止遍历，返回false。不改变原数组
>
> #### forEach() 方法: 循环原来的数组
>
> #### map() 方法： 循环原数组并映射一个新数组出来
>
> #### filter() 方法： 过滤不需要的数组元素

四、网络安全面试题

4.1 web常见的攻击方式有哪些？如何防御

答：Web攻击（WebAttack）是针对用户上网行为或网站服务器等设备进行攻击的行为

如植入恶意代码，修改网站权限，获取网站用户隐私信息等等

Web应用程序的安全性是任何基于Web业务的重要组成部分

我们常见的Web攻击方式有

- XSS (Cross Site Scripting) 跨站脚本攻击

```
XSS 即（Cross Site Scripting）中文名称为：跨站脚本攻击。XSS的重点不在于跨站点，而在于脚本的执行。

XSS的原理是：
   恶意攻击者在web页面中会插入一些恶意的script代码。当用户浏览该页面的时候，那么嵌入到web页面中script代码会执行，因此会达到恶意攻击用户的目的。
   XSS攻击最主要有如下分类：反射型、存储型、及 DOM-based型。 反射性和DOM-baseed型可以归类为非持久性XSS攻击。存储型可以归类为持久性XSS攻击

   (1)反射型XSS的攻击步骤如下：
      1．攻击者在url后面的参数中加入恶意攻击代码。
      2．当用户打开带有恶意代码的URL的时候，网站服务端将恶意代码从URL中取出，拼接在html中并且返回给浏览器端。
      3．用户浏览器接收到响应后执行解析，其中的恶意代码也会被执行到。
      4．攻击者通过恶意代码来窃取到用户数据并发送到攻击者的网站。攻击者会获取到比如cookie等信息，然后使用该信息来冒充合法用户的行为，调用目标网站接口执行攻击等操作。
   (2)存储型XSS的攻击步骤如下：
      1．攻击者将恶意代码提交到目标网站数据库中。
```

```
    2．用户打开目标网站时，网站服务器将恶意代码从数据库中取出，然后拼接到html中返回给浏览器中。
    3．用户浏览器接收到响应后解析执行，那么其中的恶意代码也会被执行。
    4．那么恶意代码执行后，就能获取到用户数据，比如上面的cookie等信息，那么把该cookie发送到攻击者网站中，那么攻击者拿到该cookie然后会冒充该用户的行为，调用目标网站接口等违法操作。


(3)DOM型XSS的攻击步骤如下：
    1．攻击者构造出特殊的URL、在其中可能包含恶意代码。
    2．用户打开带有恶意代码的URL。
    3．用户浏览器收到响应后解析执行。前端使用js取出url中的恶意代码并执行。
    4．执行时，恶意代码窃取用户数据并发送到攻击者的网站中，那么攻击者网站拿到这些数据去冒充用户的行为操作。调用目标网站接口执行攻击者一些操作。

    DOM XSS 是基于文档对象模型的XSS。一般有如下DOM操作：
    1．使用document.write直接输出数据。
    2．使用innerHTML直接输出数据。
    3．使用location、location.href、location.replace、iframe.src、document.referer、window.name等这些。
```

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

  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

  利用受害者在被攻击网站已经获取的注册凭证，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

  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的特点

  - 攻击一般发起在第三方网站，而不是被攻击的网站。被攻击的网站无法防止攻击发生
  - 攻击利用受害者在被攻击网站的登录凭证，冒充受害者提交操作；而不是直接窃取数据
  - 整个过程攻击者并不能获取到受害者的登录凭证，仅仅是“冒用”
  - 跨站请求可以用各种方式：图片URL、超链接、CORS、Form提交等等。部分请求方式可以直接嵌入在第三方论坛、文章中，难以进行追踪
- SQL注入攻击

```
SQL注入是通过客户端的输入把SQL命令注入到一个应用的数据库中，从而执行恶意的SQL语句。
什么意思呢？我们来打个比方：我们有一个登录框，需要输入用户名和密码对吧，然后我们的密码输入 'or '123' = '123 这样的。
我们在查询用户名和密码是否正确的时候，本来执行的sql语句是：select * from user where username = '' and password = ''. 这样的sql语句，现在我们输入密码是如上这样的，然后我们会通过参数进行拼接，拼接后的sql语句就是：
select * from user where username = '' and password = ' ' or '123' = '123 '; 这样的了，那么会有一个or语句，只要这两个有一个是正确的话，就条件成立，因此 123 = 123 是成立的。因此验证就会被跳过。这只是一个简单的列子，比如还有密码比如是这样的：'; drop table user;, 这样的话，那么sql命令就变成了：
select * from user where username = '' and password = ''; drop table user;' , 那么这个时候我们会把user表直接删除了。

sql被攻击的原因是：sql语句伪造参数，然后对参数进行拼接后形成xss攻击的sql语句。最后会导致数据库被攻击了。

防范的方法：
1. 我们可以使用预编译语句(PreparedStatement，这样的话即使我们使用sql语句伪造成参数，到了服务端的时候，这个伪造sql语句的参数也只是简单的字符，并不能起到攻击的作用。
2. 数据库中密码不应明文存储的，可以对密码使用md5进行加密
```

五、网络相关面试题

5.1 说说地址栏输入 URL 敲下回车后发生了什么?

答：1.简单分析

简单的分析，从输入 `URL` 到回车后发生的行为如下：

- URL解析
- DNS 查询
- TCP 连接
- HTTP 请求
- 响应请求
- 页面渲染

二、详细分析

URL解析

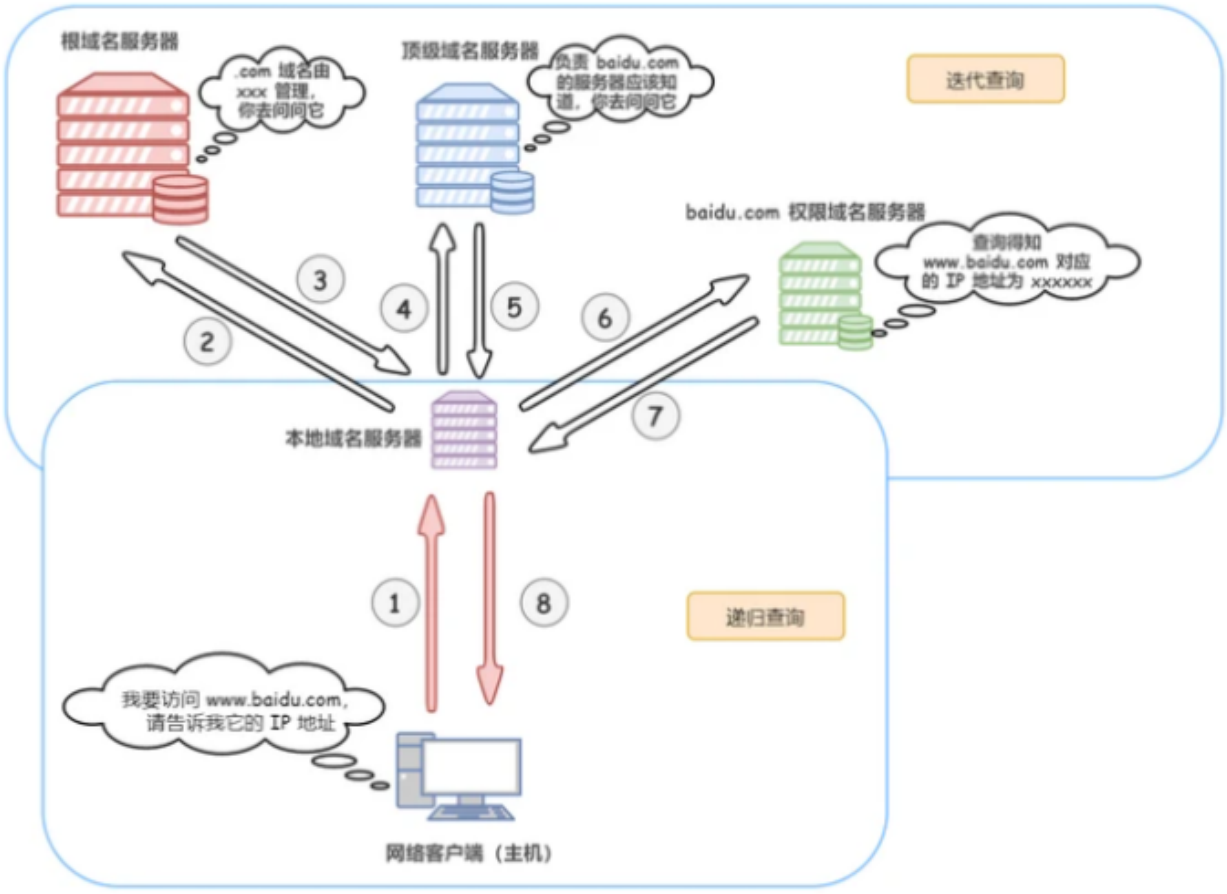
首先判断你输入的是一个合法的 `URL` 还是一个待搜索的关键词，并且根据你输入的内容进行对应操作

`URL` 的解析第过程中的第一步，一个 `url` 的结构解析如下：

DNS查询

在之前文章中讲过 `DNS` 的查询，这里就不再讲述了

整个查询过程如下图所示：

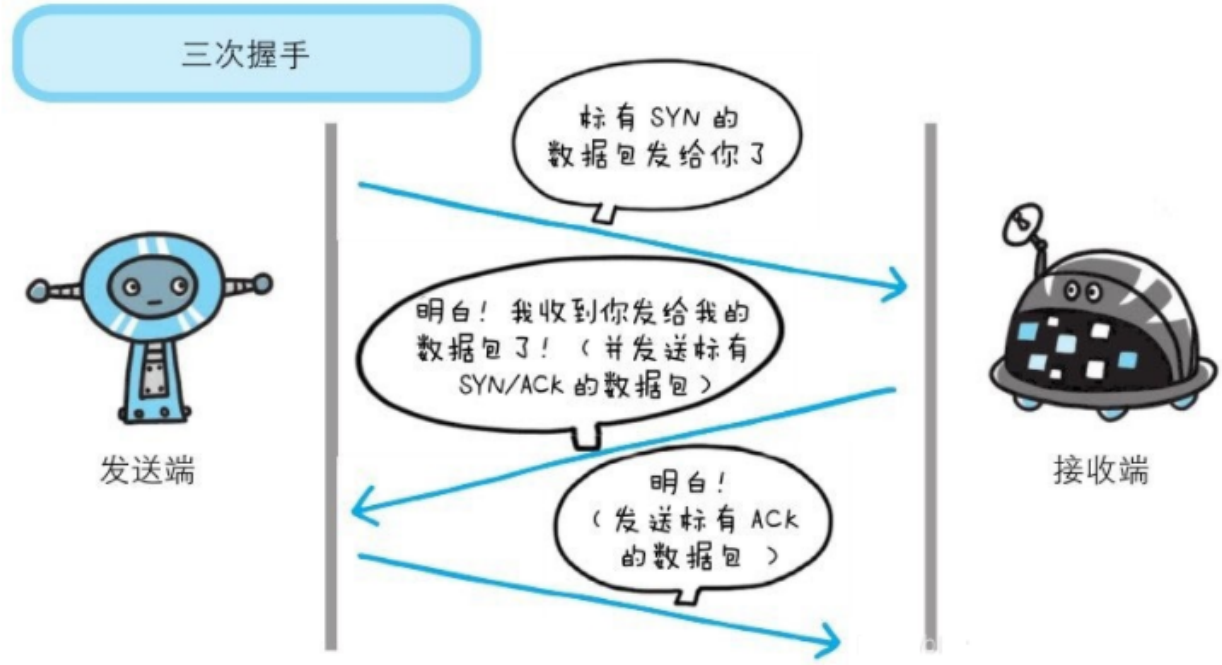


最终，获取到了域名对应的目标服务器 `IP` 地址

TCP连接

在之前文章中，了解到 `tcp` 是一种面向有连接的传输层协议

在确定目标服务器服务器的 `IP` 地址后，则经历三次握手建立 `TCP` 连接，流程如下：



发送 http 请求

当建立 `tcp` 连接之后，就可以在这基础上进行通信，浏览器发送 `http` 请求到目标服务器

请求的内容包括：

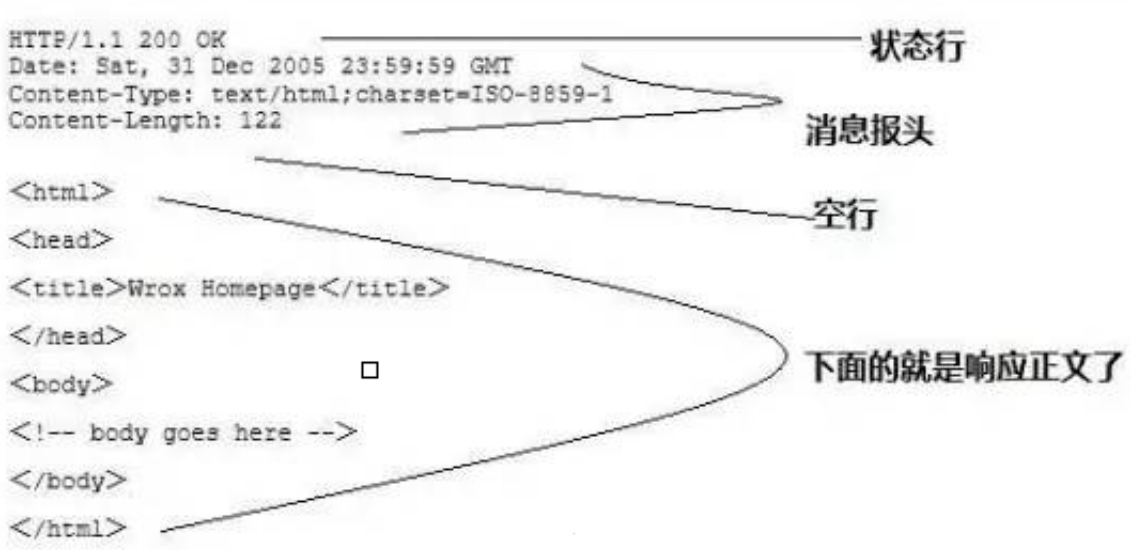
- 请求行
- 请求头
- 请求主体

响应请求

当服务器接收到浏览器的请求之后，就会进行逻辑操作，处理完成之后返回一个 `HTTP` 响应消息，包括：

- 状态行
- 响应头

- 响应正文



在服务器响应之后，由于现在 `http` 默认开始长连接 `keep-alive` ，当页面关闭之后， `tcp` 链接则会经过四次挥手完成断开

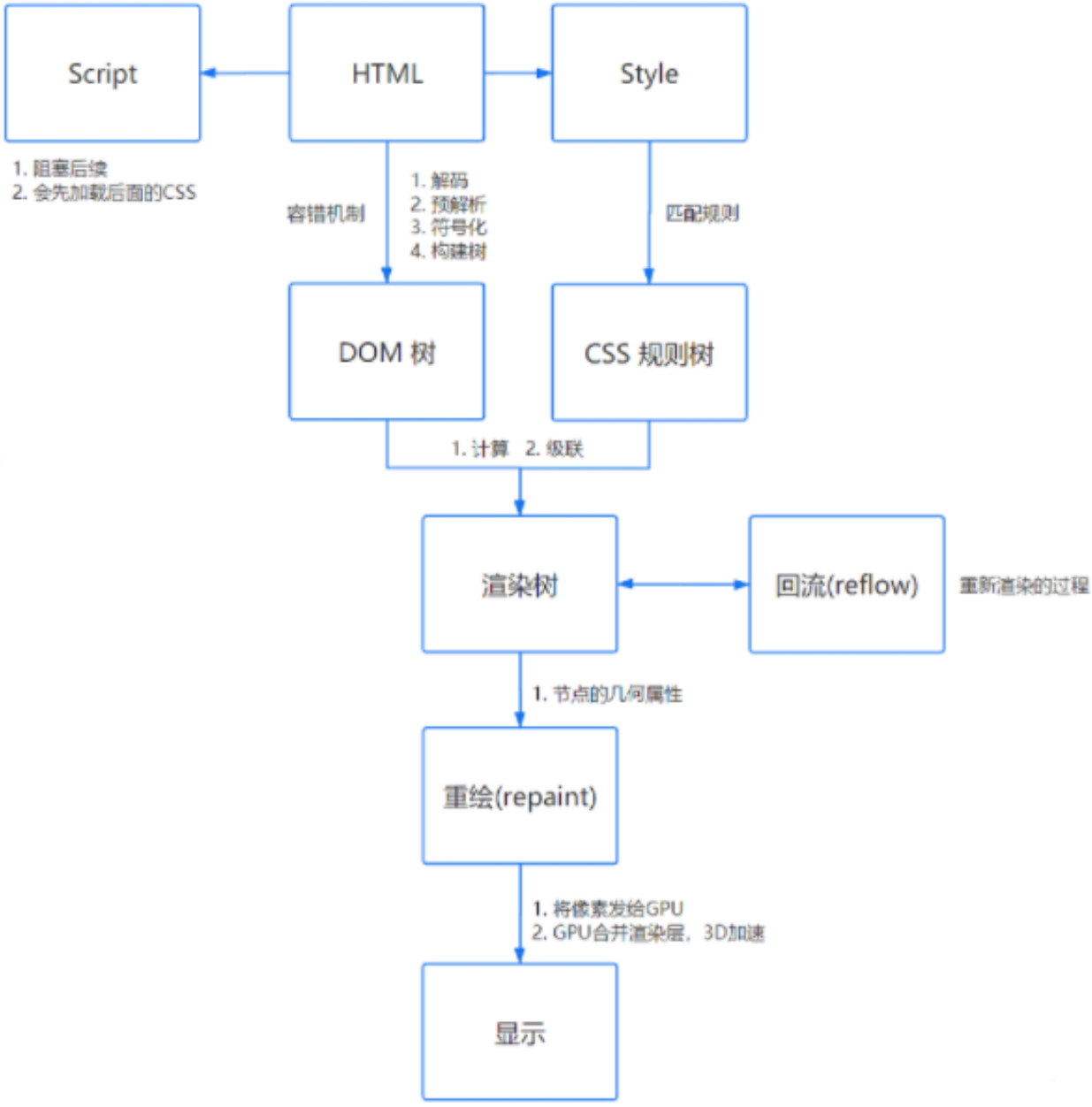
页面渲染

当浏览器接收到服务器响应的资源后，首先会对资源进行解析：

- 查看响应头的信息，根据不同的指示做对应处理，比如重定向，存储cookie，解压gzip，缓存资源等等
- 查看响应头的 Content-Type的值，根据不同的资源类型采用不同的解析方式

关于页面的渲染过程如下：

- 解析HTML，构建 DOM 树
- 解析 CSS ，生成 CSS 规则树
- 合并 DOM 树和 CSS 规则，生成 render 树
- 布局 render 树（ Layout / reflow ），负责各元素尺寸、位置的计算
- 绘制 render 树（ paint ），绘制页面像素信息
- 浏览器会将各层的信息发送给 GPU，GPU 会将各层合成（ composite ），显示在屏幕上



5.2 说一下 GET 和 POST 的区别？还有哪些提交方式?

答：**1.是什么**

`GET` 和 `POST` ，两者是 `HTTP` 协议中发送请求的方法

GET

`GET` 方法请求一个指定资源的表示形式，使用GET的请求应该只被用于获取数据

POST

`POST` 方法用于将实体提交到指定的资源，通常导致在服务器上的状态变化或**副作用**

本质上都是 `TCP` 链接，并无差别

但是由于 `HTTP` 的规定和浏览器/服务器的限制，导致他们在应用过程中会体现出一些区别

GET和POST区别

从 `w3schools` 得到的标准答案的区别如下：

- GET在浏览器回退时是无害的，而POST会再次提交请求。
- GET产生的URL地址可以被Bookmark，而POST不可以。
- GET请求会被浏览器主动cache，而POST不会，除非手动设置。
- GET请求只能进行url编码，而POST支持多种编码方式。
- GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
- GET请求在URL中传送的参数是有长度限制的，而POST没有。
- 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
- GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
- GET参数通过URL传递，POST放在Request body中

参数位置

貌似从上面看到 `GET` 与 `POST` 请求区别非常大，但两者实质并没有区别

无论 `GET` 还是 `POST` ，用的都是同一个传输层协议，所以在传输上没有区别

当不携带参数的时候，两者最大的区别为第一行方法名不同

> POST /uri HTTP/1.1 \r\n
>
> GET /uri HTTP/1.1 \r\n

当携带参数的时候，我们都知道 `GET` 请求是放在 `url` 中， `POST` 则放在 `body` 中

`GET` 方法简约版报文是这样的

```
GET /index.html?name=qiming.c&age=22 HTTP/1.1
Host: localhost
```

`POST` 方法简约版报文是这样的

```
POST /index.html HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded

name=qiming.c&age=22
```

注意：这里只是约定，并不属于 `HTTP` 规范，相反的，我们可以在 `POST` 请求中 `url` 中写入参数，或者 `GET` 请求中的 `body` 携带参数

参数长度

`HTTP` 协议没有 `Body` 和 `URL` 的长度限制，对 `URL` 限制的大多是浏览器和服务器的原因

`IE` 对 `URL` 长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持

这里限制的是整个 `URL` 长度，而不仅仅是参数值的长度

服务器处理长 `URL` 要消耗比较多的资源，为了性能和安全考虑，会给 `URL` 长度加限制

安全

`POST` 比 `GET` 安全，因为数据在地址栏上不可见

然而，从传输的角度来说，他们都是不安全的，因为 `HTTP` 在网络上是明文传输的，只要在网络节点上捉包，就能完整地获取数据报文

只有使用 `HTTPS` 才能加密安全

数据包

对于 `GET` 方式的请求，浏览器会把 `http header` 和 `data` 一并发送出去，服务器响应200（返回数据）

对于 `POST`，浏览器先发送 `header`，服务器响应100 `continue`，浏览器再发送 `data`，服务器响应200 ok

并不是所有浏览器都会在 `POST` 中发送两次包，`Firefox` 就只发送一次

提交方式除了GET和POST外，还有put和delete，简单解释一下：

- PUT请求是向服务器端发送数据的（与GET不同）从而改变信息，该请求就像数据库的update操作一样，用来修改数据的内容，但是不会增加数据的种类等，也就是说无论进行多少次PUT操作，其结果并没有不同。
- DELETE请求顾名思义，就是用来删除某一个资源的，该请求就像数据库的delete操作。

5.3 什么是HTTP? HTTP 和 HTTPS 的区别?

答：**1.HTTP (HyperText Transfer Protocol)，即超文本运输协议**，是实现网络通信的一种规范

在计算机和网络世界有，存在不同的协议，如广播协议、寻址协议、路由协议等等.....

而 `HTTP` 是一个传输协议，即将数据由A传到B或将B传输到A，并且 A 与 B 之间能够存放很多第三方，如： A<=>X<=>Y<=>Z<=>B

传输的数据并不是计算机底层中的二进制包，而是完整的、有意义的数据，如HTML 文件, 图片文件, 查询结果等超文本，能够被上层应用识别

在实际应用中，`HTTP` 常被用于在 `Web` 浏览器和网站服务器之间传递信息，以明文方式发送内容，不提供任何方式的数据加密

特点如下：

- 支持客户/服务器模式
- 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快
- 灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记

- 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间
- 无状态：HTTP协议无法根据之前的状态进行本次的请求处理

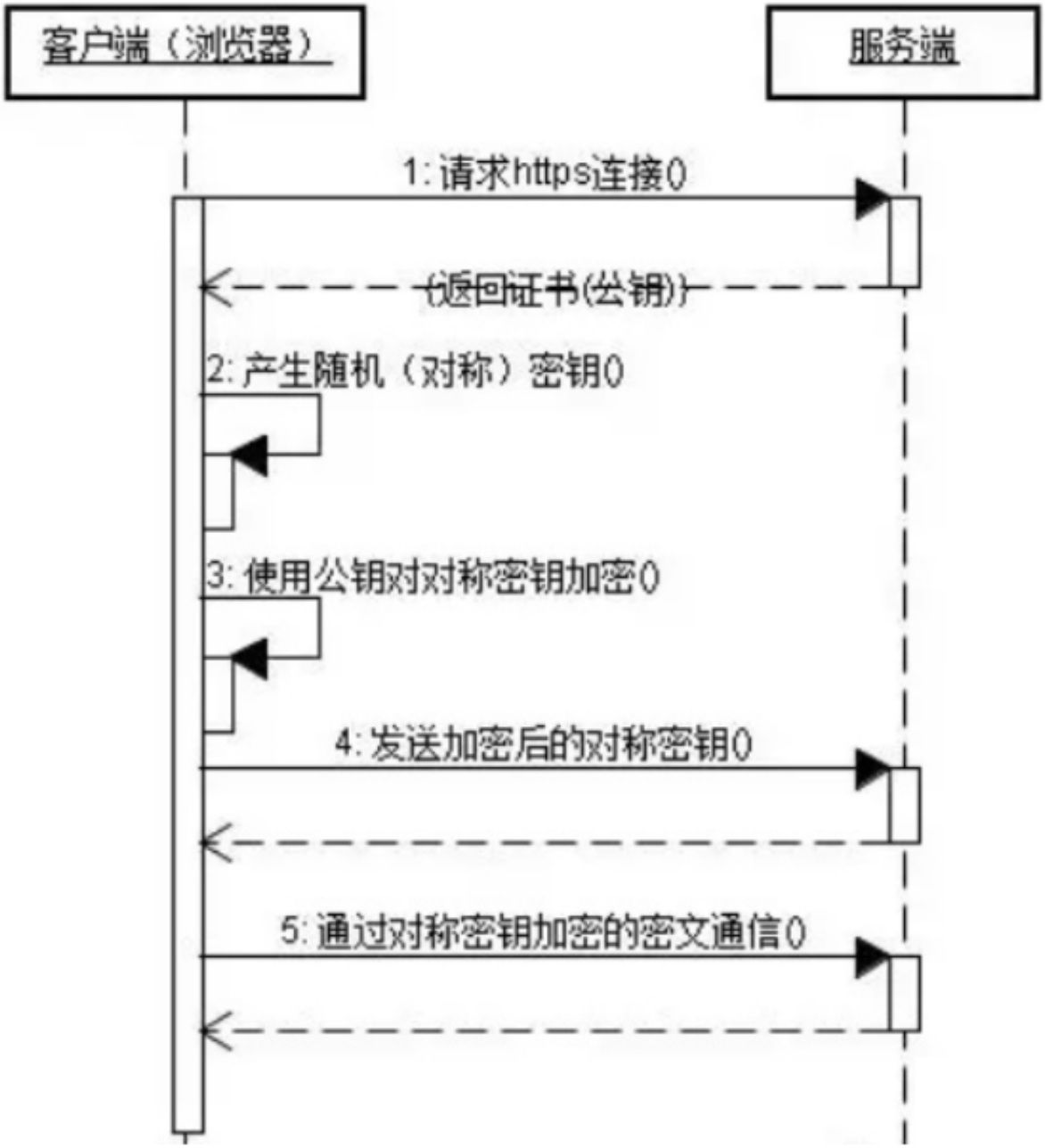
二、HTTPS

在上述介绍 `HTTP` 中，了解到 `HTTP` 传递信息是以明文的形式发送内容，这并不安全。而 `HTTPS` 出现正是为了解决 `HTTP` 不安全的特性

为了保证这些隐私数据能加密传输，让 `HTTP` 运行安全的 `SSL/TLS` 协议上，即 HTTPS = HTTP + SSL/TLS，通过 `SSL` 证书来验证服务器的身份，并为浏览器和服务器之间的通信进行加密

`SSL` 协议位于 `TCP/IP` 协议与各种应用层协议之间，浏览器和服务器在使用 `SSL` 建立连接时需要选择一组恰当的加密算法来实现安全通信，为数据通讯提供安全支持

流程图如下所示：



- 首先客户端通过URL访问服务器建立SSL连接
- 服务端收到客户端请求后，会将网站支持的证书信息（证书中包含公钥）传送一份给客户端
- 客户端的服务器开始协商SSL连接的安全等级，也就是信息加密的等级
- 客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站
- 服务器利用自己的私钥解密出会话密钥
- 服务器利用会话密钥加密与客户端之间的通信

三、区别

- HTTPS是HTTP协议的安全版本，HTTP协议的数据传输是明文的，是不安全的，HTTPS使用了SSL/TLS协议进行了加密处理，相对更安全
- HTTP 和 HTTPS 使用连接方式不同，默认端口也不一样，HTTP是80，HTTPS是443
- HTTPS 由于需要设计加密以及多次握手，性能方面不如 HTTP
- HTTPS需要SSL，SSL 证书需要钱，功能越强大的证书费用越高

5.4 http缓存中，强缓存和协商缓存的区别?

答：**浏览器缓存的作用是什么**

- 缓存可以减少冗余的数据传输。节省了网络带宽，从而更快的加载页面。
- 缓存降低了服务器的要求，从而服务器更快的响应

缓存的资源文件到什么地方去了呢?

> 那么首先来看下 **memory cache 和 disk cache** 缓存

- **memory cache:** 它是将资源文件缓存到内存中。等下次请求访问的时候不需要重新下载资源，而是直接从内存中读取数据。
- **disk cache:** 它是将资源文件缓存到硬盘中。等下次请求的时候它是直接从硬盘中读取

浏览器缓存分为2种：强制缓存和协商缓存

协商缓存原理：

- 客户端向服务器端发出请求，服务端会检测是否有对应的标识，如果没有对应的标识，服务器端会返回一个对应的标识给客户端，客户端下次再次请求的时候，把该标识带过去，然后服务器端会验证该标识，如果验证通过了，则会响应304，告诉浏览器读取缓存。如果标识没有通过，则返回请求的资源。

那么协商缓存的标识又有2种：**ETag/if-None-Match 和 Last-Modified/if-Modify-Since**

- **协商缓存Last-Modified/if-Modify-Since**

  浏览器第一次发出请求一个资源的时候，服务器会返回一个last-Modify到hearer中. Last-Modify 含义是最后的修改时间。 当浏览器再次请求的时候，request的请求头会加上 if-Modify-Since，该值为缓存之前返回的 Last-Modify. 服务器收到if-Modify-Since后，根据资源的最后修改时间(last-Modify)和该值(if-Modify-Since)进行比较，如果相等的话，则命中缓存，返回304，否则, 如果 Last-Modify > if-Modify-Since, 则会给出200响应，并且更新Last-Modify为新的值。

- **协商缓存ETag/if-None-Match**

  ETag的原理和上面的last-modified是类似的。ETag则是对当前请求的资源做一个唯一的标识。该标识可以是一个字符串，文件的size,hash等。只要能够合理标识资源的唯一性并能验证是否修改过就可以了。ETag在服务器响应请求的时候，返回当前资源的唯一标识(它是由服务器生成的)。但是只要资源有变化，ETag会重新生成的。浏览器再下一次加载的时候会向服务器发送请求，会将上一次返回的ETag值放到request header 里的 if-None-Match里面去，服务器端只要比较客户端传来的if-None-Match值是否和自己服务器上的ETag是否一致，如果一致说明资源未修改过，因此返回304，如果不一致，说明修改过，因此返回200。并且把新的Etag赋值给if-None-Match来更新该值。

last-modified 和 ETag之间对比:

- 在精度上，ETag要优先于 last-modified。
- 在性能上，Etag要逊于Last-Modified，Last-Modified需要记录时间，而Etag需要服务器通过算法来计算出一个hash值。
- 在优先级上，服务器校验优先考虑Etag

强缓存

- **基本原理：**浏览器在加载资源的时候，会先根据本地缓存资源的header中的信息(Expires 和 Cache-Control)来判断是否需要强制缓存。如果命中的话，则会直接使用缓存中的资源。否则的话，会继续向服务器发送请求。
- **Expires**

  Expires 是http1.0的规范，它的值是一个绝对时间的GMT格式的时间字符串。这个时间代表的该资源的失效时间，如果在该时间之前请求的话，则都是从缓存里面读取的。但是使用该规范时，可能会有一个缺点就是当服务器的时间和客户端的时间不一样的情况下，会导致缓存失效
- **Cache-Control**

  Cache-Control 是http1.1的规范，它是利用该字段max-age值进行判断的。该值是一个相对时间，比如 **Cache-Control: max-age=3600,** 代表该资源的有效期是3600秒。除了该字段外，我们还有如下字段可以设置：

  no-cache: 需要进行协商缓存，发送请求到服务器确认是否使用缓存。

  no-store： 禁止使用缓存，每一次都要重新请求数据。

  public： 可以被所有的用户缓存，包括终端用户和 CDN 等中间代理服务器。

  private： 只能被终端用户的浏览器缓存，不允许 CDN 等中继缓存服务器对其缓存。

  Cache-Control 与 Expires 可以在服务端配置同时启用，同时启用的时候 Cache-Control 优先级高

5.5 说说HTTP 常见的状态码有哪些，适用场景?

答：**一、是什么**

HTTP状态码（英语：HTTP Status Code），用以表示网页服务器超文本传输协议响应状态的3位数字代码

它由 RFC 2616规范定义的，并得到 `RFC 2518` 、 `RFC 2817` 、 `RFC 2295` 、 `RFC 2774` 与 `RFC 4918` 等规范扩展

简单来讲， `http` 状态码的作用是服务器告诉客户端当前请求响应的状态，通过状态码就能判断和分析服务器的运行状态

二、分类

状态码第一位数字决定了不同的响应状态，有如下：

- 1 表示消息
- 2 表示成功
- 3 表示重定向
- 4 表示请求错误
- 5 表示服务器错误

1xx

代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束

常见的有：

- 100（客户端继续发送请求，这是临时响应）：这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应

- 101：服务器根据客户端的请求切换协议，主要用于websocket或http2升级

2xx

代表请求已成功被服务器接收、理解、并接受

常见的有：

- 200（成功）： 请求已成功，请求所希望的响应头或数据体将随此响应返回
- 201（已创建）： 请求成功并且服务器创建了新的资源
- 202（已创建）： 服务器已经接收请求，但尚未处理
- 203（非授权信息）： 服务器已成功处理请求，但返回的信息可能来自另一来源
- 204（无内容）： 服务器成功处理请求，但没有返回任何内容
- 205（重置内容）： 服务器成功处理请求，但没有返回任何内容
- 206（部分内容）： 服务器成功处理了部分请求

3xx

表示要完成请求，需要进一步操作。 通常，这些状态代码用来重定向

常见的有：

- 300（多种选择）： 针对请求，服务器可执行多种操作。 服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择
- 301（永久移动）： 请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置
- 302（临时移动）： 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求
- 303（查看其他位置）： 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码
- 305 （使用代理）： 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理
- 307 （临时重定向）： 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求

4xx

代表了客户端看起来可能发生了错误，妨碍了服务器的处理

常见的有：

- 400（错误请求）： 服务器不理解请求的语法
- 401（未授权）： 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。
- 403（禁止）： 服务器拒绝请求
- 404（未找到）： 服务器找不到请求的网页
- 405（方法禁用）： 禁用请求中指定的方法
- 406（不接受）： 无法使用请求的内容特性响应请求的网页
- 407（需要代理授权）： 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理
- 408（请求超时）： 服务器等候请求时发生超时

5xx

表示服务器无法完成明显有效的请求。这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生

常见的有：

- 500（服务器内部错误）： 服务器遇到错误，无法完成请求
- 501（尚未实施）： 服务器不具备完成请求的功能。 例如，服务器无法识别请求方法时可能会返回此代码
- 502（错误网关）： 服务器作为网关或代理，从上游服务器收到无效响应
- 503（服务不可用）： 服务器目前无法使用（由于超载或停机维护）
- 504（网关超时）： 服务器作为网关或代理，但是没有及时从上游服务器收到请求
- 505（HTTP 版本不受支持）： 服务器不支持请求中所用的 HTTP 协议版本

三、适用场景

下面给出一些状态码的适用场景：

- 100：客户端在发送POST数据给服务器前，征询服务器情况，看服务器是否处理POST的数据，如果不处理，客户端则不上传POST数据，如果处理，则POST上传数据。常用于POST大数据传输
- 206：一般用来做断点续传，或者是视频文件等大文件的加载
- 301：永久重定向会缓存。新域名替换旧域名，旧的域名不再使用时，用户访问旧域名时用301就重定向到新的域名
- 302：临时重定向不会缓存，常用 于未登陆的用户访问用户中心重定向到登录页面
- 304：协商缓存，告诉客户端有缓存，直接使用缓存中的数据，返回页面的只有头部信息，是没有内容部分
- 400：参数有误，请求无法被服务器识别
- 403：告诉客户端进制访问该站点或者资源，如在外网环境下，然后访问只有内网IP才能访问的时候则返回
- 404：服务器找不到资源时，或者服务器拒绝请求又不想说明理由时
- 503：服务器停机维护时，主动用503响应请求或 nginx 设置限速，超过限速，会返回503
- 504：网关超时