HTML 部分

HTML5 新特新有哪些

1. 语义化标签（header、nav、main、article、section、aside、footer）
2. 表单控件（calender、date、time、email、url、search）
3. 音视频处理 API（audio、video）
4. Canavs/webGL
5. 拖拽释放 API
6. History API
7. 地理位置 API（geolocation）
8. websocket
9. web 存储（localStorage、sessionStorage）

CSS 部分

CSS3 新特性

1. 选择器（属性选择器、伪类选择器、伪元素选择器）
2. 盒子模型（box-sizeing: border-box）
3. 图片模糊（backdrop-filter: blur(20px)）
4. 计算盒子宽度函数（calc）
5. 过渡（transition）
6. 动画（animation）
7. 形状转换（transform: translate(xxx,xxx) / rotate(xxxdeg) / scale(xxx)）)
8. 浏览器私有前缀（-webkit- 谷歌/safari、-moz- 火狐、-o 欧朋）

```
/* 例子 */
.transition { /*渐进增强写法*/
  -webkit-transition: all .5s;
     -moz-transition: all .5s;
       -o-transition: all .5s;
          transition: all .5s;
}
.transition { /*优雅降级写法*/
          transition: all .5s;
       -o-transition: all .5s;
     -moz-transition: all .5s;
  -webkit-transition: all .5s;
}
```

9. 媒体查询

CSS 选择器优先级和权重计算

选择器:

1. ID 选择器　　#xxx
2. 类选择器　　.xxx
3. 属性选择器　　a[id=”xxx”]
4. 伪类选择器　　a:hover

5. 标签（元素）选择器　div
6. 伪元素选择器　a::after
7. 相邻选择器　h1 + p
8. 子选择器　div > a
9. 后代选择器　div a
10. 通配符选择器　*

优先级：

!important > 行内样式 > ID 选择器 > 类/属性/伪类选择器 > 标签（元素）/伪元素选择器 > 通配符选择器

权重计算：

```
/*
    每一个选择器都可以计算出来一个【权重值】，格式为：(a,b,c)
    其中：a、b、c的含义如下：
        a表示：一个选择器中【ID】选择器的个数。
        b表示：一个选择器中【类、伪类、属性】选择器的个数。
        c表示：一个选择器中【元素、伪元素】选择器的个数。
    比较时，从左到右逐位比较，谁大就优先级更高
*/
/* 第一种写法，权重值为：(0,2,3) */
div.earthy ul.list li {
    color: green;
}

/* 第二种写法，权重值为：(0,1,3)  */
div.earthy>ul>li {
    color: blue;
}

/* 第三种写法，权重值为：(0,2,2) */
.earthy ul[a="hello"]>li {
    color: orange;
}

/* 第四种写法，权重值为：(0,2,3) */
body .earthy ul[a="hello"]>li {
    color: red;
}
```

如何设置小于 12px 的字体大小

1. 利用 scale 属性
2. -webkit-text-size-adjust: none

Position 属性的值有哪些及其区别

默认定位（static）

默认值，没有定位，top, right, bottom, left 和 z-index 属性无效

相对定位（relative）

相对于元素原来位置定位，元素占据原来位置，可以设置 top, right, bottom, left 值，设置之后元素原来的位置会出现空白

绝对定位（static）

相对于最近的非 static 的已设置定位的父元素定位，元素正常脱离文档流，不再占据原来的位置，可以设置 top, right, bottom, left 值，还可以设置 margin 值，且不会与其他边距合并

固定定位（fixed）

相对于屏幕视口定位（viewport）定位，元素脱离正常文档流，不再占据原来的位置，元素的位置在屏幕滚动时不会改变，当元素祖先的 transform、persective 或 filter 属性非 none 时，定位的容器由屏幕视口改为该祖先

例如：在通过 filter 添加灰色滤镜时，如果 filter 属性添加在 body 标签上，就会导致 fixed 属性失效，所以此时一般将 filter 属性添加到 html 标签上

粘性定位（sticky）

相对定位和固定定位的混合，粘性定位可以被认为是相对定位和固定定位的混合。元素在跨越特定阈值前为相对定位，之后为固定定位。必须指定 top, right, bottom 或 left 四个阈值其中之一，才可使粘性定位生效。否则其行为与相对定位相同。

`static`

该关键字指定元素使用正常的布局行为，即元素在文档常规流中当前的布局位置。此时 `top`, `right`, `bottom`, `left` 和 `z-index` 属性无效。

`relative`

该关键字下，元素先放置在未添加定位时的位置，再在不改变页面布局的前提下调整元素位置（因此会在此元素未添加定位时所在位置留下空白）。position:relative 对 table-*-group, table-row, table-column, table-cell, table-caption 元素无效。

`absolute`

元素会被移出正常文档流，并不为元素预留空间，通过指定元素相对于最近的非 static 定位祖先元素的偏移，来确定元素位置。绝对定位的元素可以设置外边距（margins），且不会与其他边距合并。

`fixed`

元素会被移出正常文档流，并不为元素预留空间，而是通过指定元素相对于屏幕视口（viewport）的位置来指定元素位置。元素的位置在屏幕滚动时不会改变。打印时，元素会出现在的每页的固定位置。`fixed` 属性会创建新的层叠上下文。当元素祖先的 `transform`, `perspective` 或 `filter` 属性非 `none` 时，容器由视口改为该祖先。

`sticky`

元素根据正常文档流进行定位，然后相对它的*最近滚动祖先（nearest scrolling ancestor）*和 containing block (最近块级祖先 nearest block-level ancestor)，包括 table-related 元素，基于 `top`, `right`, `bottom`, 和 `left` 的值进行偏移。偏移值不会影响任何其他元素的位置。 该值总是创建一个新的层叠上下文（stacking context）。注意，一个 sticky 元素会"固定"在离它最近的一个拥有"滚动机制"的祖先上（当该祖先的 `overflow` 是 `hidden`, `scroll`, `auto`, 或 `overlay` 时），即便这个祖先不是最近的真实可滚动祖先。这有效地抑制了任何"sticky"行为（详情见Github issue on W3C CSSWG）。

box-sizeing 和盒子模型

box-sizeing：规定如何计算一个元素的总宽度和总高度，主要分为两种

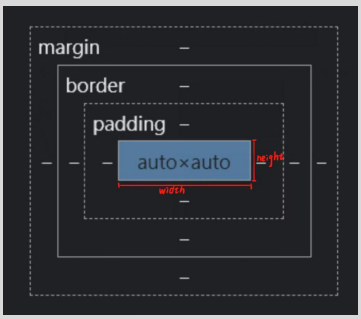
conent-box 标准盒子模型

盒子 width 和 height 只包括内容的宽和高，不包括边框（border），内边距（padding），外边距（margin）

计算公式

width = 内容的宽度

height = 内容的高度



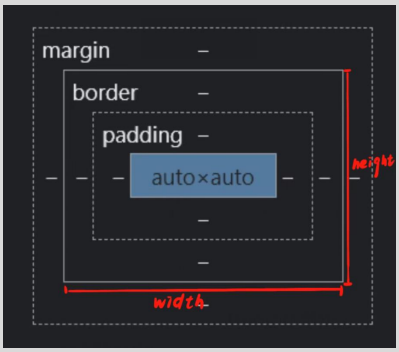
border-box 怪异盒子模型（IE 盒子模型）

盒子 width 和 height 包括内边距（padding）和边框（border），不包括外边距（margin）

计算公式

width = border + padding + 内容的宽度

height = border + padding + 内容的高度



文档流和脱离文档流

文档流：

文档流(Normal flow)也称为常规流，普通流。从直观上理解，指的是元素按照其在 HTML 中的位置顺序决定排布的过程,主要的形式是自上而下（块级元素）,一行接一行,每一行从左至右（行内元素）

脱离文档流：

元素脱离文档流之后，将不再在文档流中占据空间，而是处于浮动状态（可以理解为漂浮在文档流的上方）。脱离文档流的元素的定位基于正常的文档流，当一个元素脱离文档流后，依然在文档流中的其他元素将忽略该元素并填补其原先的空间。

外边距合并

定义：

两个 div 盒子的下边距和上边距有时会合并（折叠）为单个边距，合并的边距的大小取决于下边距和上边距的更大的值，如果两个值相等，则随便其中一个即可（注意：浮动元素（display: float）和绝对定位元素（position: absolute）不会差生外边距合并，因为他们已经正常的文档流）

产生场景：

1. 两个相邻的元素（除非后一个元素使用了清除浮动 clear: both）

```
<style>
p:nth-child(1){
  margin-bottom: 13px;
}
p:nth-child(2){
  margin-top: 87px;
}
</style>

<p>下边界范围会...</p>
<p>...会跟这个元素的上边界范围重叠。</p>
```

这个例子如果以为边界会合并的话，理所当然会猜测上下 2 个元素会合并一个 100px 的边界范围，但其实会发生边界折叠，只会挑选最大边界范围留下，所以这个例子的边界范围其实是 87px。

2. 空的块级元素（当两个盒子中间有第三个盒子，然后这个盒子没有设置任何属性，这两个盒子

还是会发生外边距合并）

解决方案：创建 BFC 或者清除浮动

BFC

BFC（块级格式化上下文），它是一个独立的渲染区域，规定了内部的盒子如何布局，并且这个区域的子元素不会影响到外面的元素，主要的规则包括内部的盒子垂直放置，计算 BFC 的高度的时候，浮动元素也会参与运算

布局规则：

1. 内部的盒子会在垂直方向一个接一个的摆放
2. 元素的垂直方向的距离由 margin 决定，属于同一个 BFC 的两个相邻的元素的 margin 会发生重叠，又称为外边距合并（可以创建两个 BFC 来解决外边距合并的问题）
3. BFC 中的元素不会和浮动的元素发生重叠
4. BFC 是一个独立的容器，它内部的子元素不会影响外面的元素
5. 计算 BFC 的高度时，浮动的元素也会参与运算

如何创建 BFC：

1. 根元素（<html></html>）
2. 浮动元素（float 值不为 none）
3. 绝对定位或固定定位元素（position 值为 absolute 或 fixed）
4. 行内块元素（display: inline-block）
5. 弹性元素或网格元素（display 属性为 flex 或 grid）

作用：

1. 去除外边距重叠
2. 清除浮动（让父元素的高度包含子浮动元素，给浮动的元素的父元素设置高度）

元素居中

水平居中：

行内元素：text-align: center;

块级元素：

已知宽度：

1. margin: 0 auto;
2. 绝对定位 + margin-left: (父盒子 width – 子盒子 width) / 2

未知宽度：

1. display: inline-block; + text-align: center;
2. 绝对定位 + margin-left: 50%; + transform: translateX(-50%)
3. flex 布局 + justify-content: center;
4. grid 布局 + justify-content: center; / justify-items: center; 或者给需要水平居中的子元素添加 jusrify-self: center;

垂直居中：

行内元素（纯文字类）：line-height = 盒子高度

块级元素：

1. 绝对定位 + margin-top: 50%; + transform: translateY(-50%)
2. flex 布局 + align-items: center;
3. grid 布局 + align-content: center; / align-items: center; 或者给需要水平居中的子元素添加 align-self: center;

水平垂直居中：

1. 绝对定位 + margin 左右各 50% + transform: translate(-50%, -50%)
2. Flex 布局 + justify-content: center + align-items: center;
3. Grid 布局

父元素：justify-content: center + align-items: center;
父元素：align-content: center + justify-items: center;
子元素：justify-self: center + align-self: center;

隐藏页面中某个元素的方法

opacity: 0

将元素设置为完全透明，不会改变页面原有布局（元素仍占据原有空间），元素绑定的事件（如 click）仍可触发

visibility: hidden

隐藏元素，不会改变页面原有布局（元素仍占据原有空间），元素绑定的事件无法触发，并且会触发重绘

display: none

隐藏元素，会改变页面原有布局（该元素和它的子元素不会被浏览器渲染），元素绑定的事件无法触发，且会触发回流和重绘

Flex 布局常见属性

父元素

1. flex-direction 设置主轴方向 row | column | row-reverse | column-reverse
2. flex-warp 设置是否换行 nowarp | warp | warp-reverse
3. flex-flow 前两项的组合属性
4. justify-content 主轴的子元素排列方式 flex-start | center | flex-end | space-around | space-between
5. align-content 侧轴的子元素排列方式（多行）flex-start | center | flex-end | space-around | space-between | stretch（高度平分父元素高度）
6. align-items 侧轴的子元素排列方式（单行）flex-start | center | flex-end | stretch（拉伸，

默认值）

子元素

1. flex　　子项所占的份数（也是 flex-grow、flex-shrink 和 flex-basis 的缩写）
2. align-self　　控制子项在侧轴的排列方式
3. order　　子项的排列顺序
4. flex-grow　　元素放大，默认不会放大（默认值 0），值为 1 则放大
5. flex-shrink　　元素缩小，默认空间不足的时候会等比缩小（默认值 1），值为 0 则不缩小
6. flex-basis　　在分配多余空间时，元素占据的主轴空间大小，默认为 auto（元素本身大小），设置为 0%后，可以结合 flex-grow: 1 + flex-shrink: 1 可以自动放大和缩小

块元素和行内元素的区别

1. 块级元素默认一个元素占一行，行内元素与之相反
2. 块元素不设置宽度则默认继承父元素宽度
3. 行内元素不可以设置 height 和上下的 margin 和 padding，可以设置左右的 margin 和 padding
4. 块级元素可以包含行内元素和块级元素。行内元素不能包含块级元素。

高度塌陷

含义：由于浮动，导致本应该在盒子内部的元素跑到了盒子的外部

解决方案：

1. 将盒子的高度和宽度固定
2. 将外层盒子也添加浮动
3. 设置 overflow 属性，触发 BFC
4. 清除浮动

清除浮动几种方式

1. 添加额外标签并设置 clear 属性

```
<div class="parent">
    //添加额外标签并且添加clear属性
    <div style="clear:both"></div>
    //也可以加一个br标签
</div>
```

2. 给父级元素添加 overflow 属性或设置高度，触发 BFC
3. 伪元素清除浮动

```
.clearfix:after{
    content:".";  /*尽量不要为空，一般写一个点*/
    height:0;     /*盒子高度为0，看不见*/
    display:block;    /*插入伪元素是行内元素，要转化为块级元素*/
    visibility:hidden;      /*content有内容，将元素隐藏*/
    clear:both;
}

.clearfix {
    *zoom: 1;   /*  *只有IE6,7识别 */
}
```

4. 双伪元素

```
.clearfix:before, .clearfix:after {
        content: "";
        display: table;
    }
    .clearfix:after {
        clear: both;
    }
    .clearfix {
        *zoom: 1;
    }
```

为什么 CSS 写在顶部，JS 写在底部

1. 浏览器可预先加载 CSS，可以不必等待 HTML 加载完成即可渲染页面（一边加载一边解析）
2. JS 存在事件处理或 DOM 操作，所以需要将 HTML 加载完成后再使用，否则可能导致报错

伪类（:first-child）和伪元素（:after）的区别

伪类:

伪类用于选择 DOM 树之外的信息，或是不能用简单选择器进行表示的信息。比如:visited，:active,first-child，:first-of-type，:target 等

伪元素:

伪元素为 DOM 树没有定义的虚拟元素。不同于其他选择器，它不以元素为最小选择单元，它选择的是元素指定内容。比如::before

区别:

1. 伪元素创建了一个文档树以外的元素并为它添加样式，这个容器不包含任何 DOM 元素但是可以包含内容。（伪元素则创建了一个文档树以外的元素）
2. 伪类使用单引号，伪元素使用双引
3. 伪类通常用于选择 DOM 元素，而伪元素通常是创建一个类 DOM 元素

link 和@import 的区别

1. link 是 HTML 标签，@import 是 CSS 提供的方式，link 还可以引入图片等资源，@import 只能加载 CSS
2. 加载顺序不同，link 在页面载入时同时加载，@import 需要网页完全载入后再加载
3. 兼容性不同，link 是 XHTML 标签，无兼容性问题，@import 式 CSS2.1 提出的，低版本浏览器不支持
4. link 支持使用 js 控制 DOM 改变样式，@import 不支持

可以被继承的元素有哪些

所有元素可以继承的属性：

1. 元素可见性：visibility、opacity
2. 光标属性：cursor

内联元素可以继承的属性：

1. 字体系列属性（font-family、font-size、font-style 等）
2. 除 text-indent、text-align 之外的文本系列属性（line-height、word-spacing 字间距、letter-spaceing 字符间距、color、text-transform 大小写、direction 书写方向）

块级元素可以继承的属性：

1. text-indent
2. text-align

CSS 中的百分比相对于谁

相对于参照物的

```
position: absolute;
top: 10%; // 相对于第一个有定位属性的元素
```

相对于自身的

```
position: relative;
top: 10%;
border-radius: 50%;
background-size: 10%;
transform: 50%;
text-indent: 10%;
line-height: 50%; // 相对自身的 font-size
```

相对于父元素

```
margin: 10%;
padding: 10%;
font-size: 10%; // 相对父元素字体大小
```

em 和 rem

em：

相对于自身的字体（font-size）大小（字体大小可以继承父元素的字体大小），如果当前文本的字体尺寸没有设置，则相对于浏览器的默认字体尺寸

例如一个 div 的字体为 18px，设置它的宽高为 10em，那么此时宽高就是 18px * 10em = 180px

rem

相对于 HTML 根元素的字体大小

例如 HTML 根元素的字体大小为 16px，那么 10rem 就等同于 10*16=160px

为什么有时候突然网页上会只有 DOM 结构而样式丢失的情况，刷新一下又好了

html 是边解析边渲染，若 css 引入方式不是放在 head 中，会使前面已经解析的 dom 先渲染出来，出现几秒的样式丢失（简而言之就是 DOM 比 CSS 样式先加载出来了）

css 和 js 是否会阻塞页面渲染

会，因为默认是从上到下，边解析便渲染，JS 可以使用 defer 等来延迟加载

1. CSS 不会阻塞 DOM 解析，但是会阻塞 DOM 渲染
2. JS 会阻塞 DOM 解析
3. CSS 会阻塞 JS 的执行（所以一般将<script>放在<link>标签前面是有道理的）
4. 浏览器遇到<script>标签且没有 defer 或 async 属性时会触发页面渲染

JavaScript 部分

JS 数据类型有哪些

基本数据类型

undefined	未定义
Null	空
Boolean	布尔
Number	数值
String	字符串
Symbol	符号，ES6 新增，表示独一无二的值，可以表示对象的唯一属性名
BigInt	任意大小的整数，ES6 新增

引用数据类型

Object	对象
Function	函数
Array	数组

我们知道 JavaScript 的引用数据类型是保存在堆内存中的，然后在栈内存中保存一个对堆内存中实际对象的引用，所以，JavaScript 中对引用数据类型的操作都是操作对象的引用而不是实际的对象。可以简单理解为，栈内存中保存了一个地址，这个地址和堆内存中的实际值是相关的

如何判断数据类型

1. typeof

   优点：可以判断所有的基本数据类型和 function

   缺点：不能判断 null、object、array（判断的返回值均为 object）

```
console.log(typeof undefined); // undefined
console.log(typeof 2); // number
console.log(typeof true); // boolean
console.log(typeof "str"); // string
console.log(typeof Symbol("foo")); // symbol
console.log(typeof 2172141653n); // bigint
console.log(typeof function () {}); // function
// 不能判别
console.log(typeof []); // object
console.log(typeof {}); // object
console.log(typeof null); // object
```

2. instanceof

   优点：可以判断 object、array 和 function，适用于判断自定义类的实例对象（判断机制为判断实例的原型是否为指定的判断类型，因为类的实例的原型为该类）

缺点：不能判断基本数据类型

```
console.log(1 instanceof Number);                    // false
console.log(true instanceof Boolean);                // false
console.log('str' instanceof String);                // false
console.log([] instanceof Array);                    // true
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
```

```
class People {}
class Student extends People {}

const vortesnail = new Student();

console.log(vortesnail instanceof People); // true
console.log(vortesnail instanceof Student); // true
```

3. Object.prototype.toString.call()
   优点：可以判断所有数据类型

```
var toString = Object.prototype.toString;
console.log(toString.call(1));                       //[object Number]
console.log(toString.call(true));                    //[object Boolean]
console.log(toString.call('mc'));                    //[object String]
console.log(toString.call([]));                      //[object Array]
console.log(toString.call({}));                      //[object Object]
console.log(toString.call(function(){}));            //[object Function]
console.log(toString.call(undefined));               //[object Undefined]
console.log(toString.call(null));                    //[object Null]
```

```
Object.prototype.toString.call(2); // "[object Number]"
Object.prototype.toString.call(""); // "[object String]"
Object.prototype.toString.call(true); // "[object Boolean]"
Object.prototype.toString.call(undefined); // "[object Undefined]"
Object.prototype.toString.call(null); // "[object Null]"
Object.prototype.toString.call(Math); // "[object Math]"
Object.prototype.toString.call({}); // "[object Object]"
Object.prototype.toString.call([]); // "[object Array]"
Object.prototype.toString.call(function () {}); // "[object Function]"
```

var、let 和 const 的区别

1. var 定义的变量可以跨块级作用域访问，不能跨函数作用于访问，let 定义的变量不能跨块级作用域和函数作用域访问
2. const 用来定义常量，声明时必须初始化赋值，对于简单数据类型定义后不支持修改，对于对象而言，const 只能保证变量所指向的内存地址不被修改，换句话说，const 只是禁止变量被重新赋值，对对象的修改没有任何影响

   例如：

   const obj = { name: “zhangsan” }

   obj = {} // 会报错，const 禁止变量被重新赋值，此操作又称修改变量

   obj.name = “lisi” // 不会报错，因为此操作是在修改对象，const 不能限制修改对象

3. var 可以在声明前使用（存在变量提升：检测到未声明的变量时，不会报错，会返回 undefined，因为早期的 js 遵循的是不报错原则），let 和 const 只能先声明后使用，否则会报错（又称暂时性死区）
4. var 可以重复声明，let 和 const 不能重复声明

undefined 和 null 的区别

undefined 表示未定义的值，表示一个变量自然的、最原始的状态

以下情况会出现 undefined

1. 声明了一个变量但是没有赋值
2. 访问对象上不存在的属性
3. 函数定义了形参但是没有传递实参，当访问函数的形参时

null 表示空值，表示一个变量被人为设置为空对象

造成 JS 内存泄露的有哪几种情况

1. 意外的全局变量
2. 过多的闭包
3. 未被清空的定时器
4. 未被销毁的事件监听

call、apply 和 bind 的区别

1. call、apply 和 bind 的作用都是修改 this 指向，其中第一个参数就是 this 的指向
2. call 和 apply 都是调用后就直接执行了，bind 不会直接执行，bind 会返回一个新函数
3. call 和 apply 都是临时修改一次 this 指向，bind 是直接锁死，不能再做修改
4. call 除第一个参数外的参数需要一个一个的声明，apply 需要声明成一个数组
5. bind 除第一个参数外的参数是为新函数设置的默认值，无法被修改

箭头函数的特点

1. 语法更加简洁
2. 没有 arguments
3. 没有自己的 this，箭头函数的 this 由创建时的外层作用域所决定
4. 箭头函数的 this 不能被 call、apply 和 bind 修改

this 指向问题

1. 全局作用域中的 this 指向 window
2. 以函数形式调用（xxx()）,this 指向 window
3. 以方法形式调用（xxx.xxx()）,this 指向调用方法的对象
4. 构造函数中的 this 指向当前实例
5. 箭头函数没有自己的 this，箭头函数的 this 由创建时的外层作用域所决定，且不能被 call、apply 和 bind 修改

new 的执行流程和实现原理

1. 创建一个普通的 JS 对象，简称新对象
2. 将构造函数的 prototype 属性设置为新对象的原型
3. 执行构造函数，将新对象设置为函数中的 this
4. 判断函数的返回值类型，如果为非原始值，则直接返回该值，如果为原始值，则返回新对象（一般

情况下，不会为构造函数指定返回值）

```
function myNew(context) {
  const obj = new Object();
  obj.__proto__ = context.prototype;
  const res = context.apply(obj, [...arguments].slice(1));
  return typeof res === "object" ? res : obj;
}
```

ES6 的新特性

1. let 和 const
2. 数组和对象的扩展运算符 …
3. Set（类似于数组，但不允许有重复值）
4. Map（存储键值对，和对象的区别在于，任何类型的值都可以作为键）
5. Promise（为了解决异步调用所产生的回调地狱的一种解决方案，有 pending、fulfilled 和 reject 三种状态）
6. Generator（异步编程的一种解决方案）
7. Proxy
8. Module（导出：export / export default 导入：import）

闭包

含义：函数中返回了一个函数，且内部函数能够访问到外部函数作用域中变量的函数
什么时候使用：当我们需要隐藏一些不希望被其他人访问的内容时可以使用闭包
形成条件：

1. 函数嵌套
2. 内部函数要引用外部函数中的变量
3. 内部函数要作为返回值返回

生命周期：

1. 闭包在外部函数调用时产生，每调用一次就会创建一个全新的闭包
2. 在内部函数丢失时销毁，当内部函数被垃圾回收了，闭包才会消失

作用：

1. 隐藏变量，避免使用全局变量造成变量污染
2. 延长局部变量的生命周期

缺点：闭包会导致函数的变量一直保存在内存中，过多的闭包可能会导致内存泄露

防抖和节流的原理和使用场景

作用：优化高频执行 JS 代码的一种手段，减低执行频率，节省资源
原理：

防抖：多次触发，只执行最后一次
节流：每隔一段时间执行一次

使用场景：

防抖：

1. 搜索框输入（用户输入完成后，再发送请求）
2. 手机号、邮箱等格式校验

3. 监听窗口大小变化

节流：

1. 滚动加载更多
2. 搜索框输入联想
3. 高频点击
4. 表单重复提交

作用域和作用域链

作用域：

1. 全局作用域
2. 函数作用域
3. 块作用域 {}

作用域链：

当我们使用一个变量的时候，JS 解析器会优先在当前作用域中寻找该变量（遵循就近原则），如果找到了就直接使用，如果没找到，就到上一层作用域中去找，以此类推，一直到全局作用域中还没找到，则报错 xxx is not defined

原型和原型链

每创建一个对象，JS 解析器就会向该对象中添加一个 prototype 属性，该属性就是原型对象

如何访问：

obj.__proto__ 或 Object.prototype

作用：

原型就相当于一个公共区域，它可以被虽有该类的实例访问，我们可以将该类实例中所有的公共属性（方法）存储到原型上，这样就只需创建一个属性，即可被所有的实例访问

每个class都有显式原型prototype

每个实例都有隐式原型**proto**

实例的隐式原型指向class的prototype

可以通过hasOwnProperty检查是否为原型链上的属性

原型链：

当访问对象的某个属性时，会优先在对象自身查找该属性，如果对象中有，则直接使用，如果对象中没有，则去对象的原型中寻找，如果原型中有，则使用，如果原型中没有，则去原型的原型中寻找，直到找到 Object 对象的原型（Object 对象的原型为 null）还没有找到该属性，则返回 undefined

slice、splice 和 split 的区别

slice：数组方法

作用：截取数组

使用：

第一个参数为起始位置（截取的数组包含该位置），第二个参数为结束位置（截取的数组不包含该位置），如果省略第二个参数，则默认截取到末尾，索引也可以是负值，如果两个参数都省略，可以对数组进行浅拷贝，返回值为截取的新数组

splice：数组方法

作用：删除、插入、替换数组中的元素

使用：

第一个参数为删除的起始位置，第二个参数为删除的数量，为 0 时则为插入，此时删除的起始位置则为插入的起始位置，第三个参数为插入的元素，返回值为被删除的元素

split：字符串方法

作用：将一个字符串按照给定的参数拆分为一个数组

字符串和数组之间如何转换

字符串转数组：split()

数组转字符串：join()

数组中哪些方法会改变原数组

会改变原数组：

1. push() 向数组末尾插入数据 返回新数组的长度
2. unshift() 向数组头部插入数据 同上
3. pop() 删除数组末尾的一个数据 返回数组的最后一个元素
4. shift() 删除数组头部的一个数据 返回数组的第一个元素
5. splice() 删除数组中的元素或向数组中插入元素 返回被删除的元素
6. reverse() 反转数组
7. sort() 对数组进行排序，默认升序，可通过回调函数指定升序还是降序，不建议对数字进行排序

不会改变原数组：

1. at() 获取指定索引的数组元素
2. conect() 连接数组
3. indexOf() 获取指定元素第一次出现的索引，不存在则返回-1
4. lastIndexOf() 获取指定元素最后一次出现的索引，不存在则返回-1
5. join() 按照指定的参数将数组拼接成字符串
6. slice() 截取数组
7. forEach() 遍历数组
8. map() 同上
9. filter() 根据指定条件查找数组中的元素，存在则返回相应数组
10. find() 同上，存在则返回第一个符合的元素，不存在则返回 undefined
11. reduce() 整合数组
12. every() 根据指定条件判断数组中的元素是否符合，全部符合返回 true，否则 false
13. some() 同上，只要有一个符合就返回 true，否则返回 false

浅拷贝和深拷贝

浅拷贝：

含义：

只复制对象的浅层（第一层），浅拷贝只会对对象本身进行复制，并不会赋值对象中的属性，对象中的属性的地址还是原来的地址，当修改的时候原来的也会受到影响（原始值一般不涉及深浅拷贝问题，因为原始值在内存中是唯一的，不允许重复存在）

实现方法：

1. arr.slice()
2. 展开运算符 …

3. Object.asign(目标对象，被复制的对象)

深拷贝：

含义：

不仅赋值对象本身，对象中的属性和元素也会被复制

实现方法：

1. JSON.parse(JSON.stringify(arr)) // stringify 转为 JSON 字符串 parse 转换为 js 对象
2. window.structuredClone()

高阶函数

含义：如果一个函数的返回值是一个函数，则这个函数就是一个高阶函数

作用：通过高阶函数，可以实现在不修改一个函数的源代码的基础上对起重新包装，添加新的功能

案例：

```
function someFn() {
  return 'hello'
}

function outer(cb) {
  return () => {
    console.log('记录日志~~~~~') // 在被包装函数的基础上新增功能
    const result = cb() // 执行被包装函数，使outer函数具有被包装函数的全部功能
    return result // 返回被包装函数的返回值
  }
}

// outer函数实现了在不修改someFn函数的同时增加了功能，
// outer函数具有someFnd的全部功能，还在其基础上增加了新功能

let result = outer(someFn)
```

Map 和 Set

Map：

作用：

用来存储键值对结构的数据，任何类型的值都可以作为数据的 key（Object 中的 key 只能是字符串或者符号，如果传递了其他类型的会被自动转换为字符串）

创建方式：new Map()

常用方法和属性：

1. size() 获取 map 中的键值对数量
2. set(key, value) 向 map 中添加键值对
3. get(key) 根据 key 获取值
4. delete(key) 删除指定数据
5. has(key) 判断 map 中是否包含指定 key
6. clear() 删除 map 中的所有键值对

Set：

作用：

用来创建一个集合，功能同数组类似，但是 Set 中不允许存在重复的值

创建方式：new Set()

常用方法和属性：

1. size 获取数量
2. add() 添加元素

3. has()　　检查元素
4. delete()　　删除元素

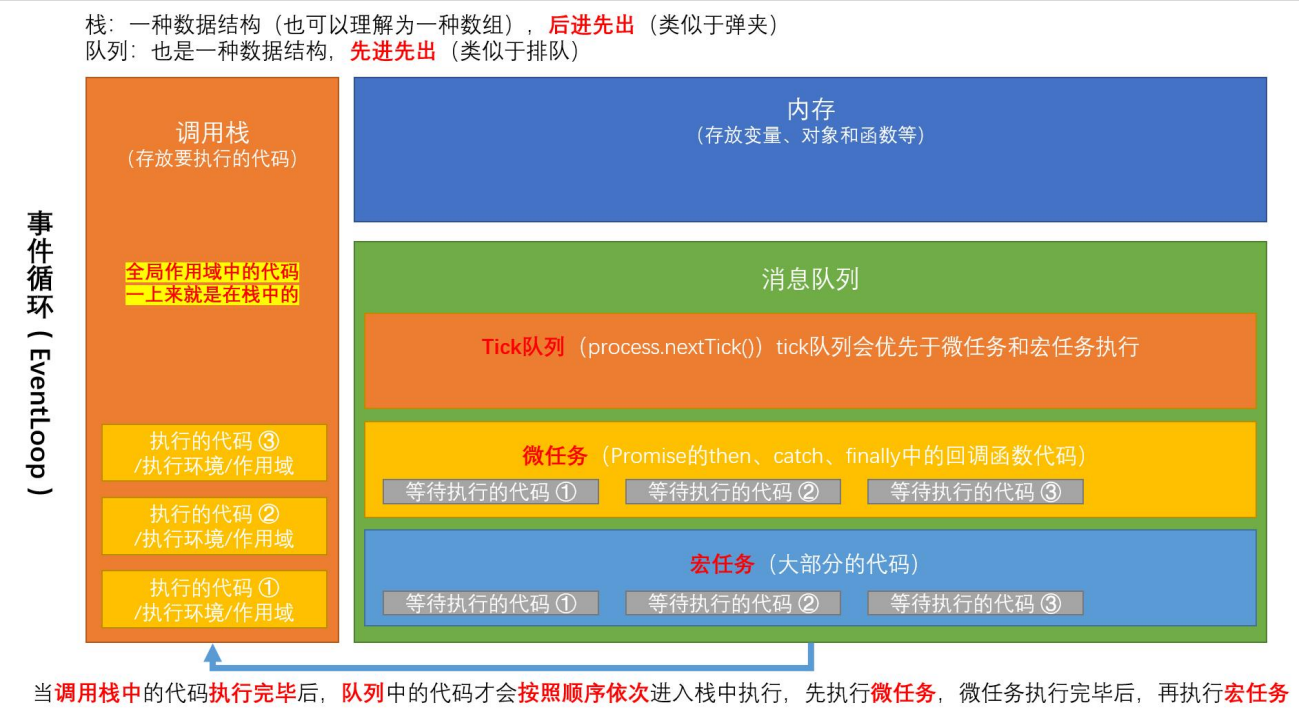
事件循环（Eventloop、JS 单线程怎么支持异步的效果）

函数在每次执行时都会产生一个执行环境，执行环境赋值存储函数执行时产生的一些数据，执行环境会存储到调用栈中，遵循后进先出的原则

当我们触发一个事件时，其响应函数不会直接就添加到调用栈中，因为调用栈中可能存在还没有执行完毕的代码，此时会将响应函数存储到消息队列中进行排队，当调用栈中的代码执行完毕之后，再根据先进先出的原则插入到调用栈中执行

消息队列中又分为宏任务队列和微任务队列，微任务队列的执行顺序优先于宏任务队列，微任务队列主要包括 Promise 的三个回调函数（.then、.catch、.finally）中的代码，其余大部分的代码（例如定时器）都是属于宏任务队列，在 Nodejs 中还存在一个 Tick 队列，Tick 队列的执行顺序优先于微任务队列

简而言之，事件循环就是先执行调用栈中的代码（例如全局作用域中的代码），然后再执行消息队列中的微任务队列（如果存在 Tick 队列就先执行 Tick 队列），然后再执行宏任务队列



Promise

在 JS 中，为了获取异步调用的结果，需要使用 return 函数来解决，然而当代码变得复杂之后，就会逐渐产生回调地狱，为了解决这一问题，于是产生了 Promise，Promise 可以解决异步中的回调问题，同时搭配 async 和 await 的使用，会使得代码更加的简洁。

当在.then 中返回值时可以在下一个.then 中获取到（每个.then 都会默认返回一个 Promise，且结果为上一个 return 的值），这就是 Promise 的链式调用。

Promise 常用函数

1. resolve()　　在执行正常时存储数据
2. reject()　　在执行错误时存储数据
3. all([…])　　同时返回多个执行结果，有一个报错就返回错误
4. allSetted([…])　　同上，不管结果成功或失败

5. race([…]) 返回执行最快的 Promise（不考虑对错，错误也参与）
6. any([…]) 同上（考虑对错，错误不参与）

Promise 包含三种状态，但是需要注意的是，Promise 的状态支支持修改一次，当修改过后就不能再被就改了，三种状态分别是

1. pending 进行中
2. fulfilled 完成，当通过 resolve 存储数据时改变
3. reject 拒绝/出错了，当出现错误或通过 reject 存储数据时改变

Promise 实现原理

queueMicrotask 用于添加一个微任务

如何将伪数组转换为数组

1. 扩展运算符（…）
2. Array.from()
3. Array.proptotype.slice.call()

async/await 的原理

async 是 Generator 函数的语法糖

严格模式（use strict）

1. 变量必须声明后再使用
2. 函数的参数不能有同名属性，否则报错
3. 不能对只读属性赋值，否则报错
4. 不能使用前缀 0 表示八进制数，否则报错
5. 不能删除不可删除的属性，否则报错
6. 禁止 this 指向全局对象
7. 增加了保留字（比如 protected、static 和 interface）

for in 和 for of 的对比

相同点：

都无法遍历 Symbol 类型的值（需要使用 Object.getOwnPropertySymbols()方法）

不同点：

1. for in 是遍历键名，for of 是遍历键值
2. for in 只遍历可枚举属性
3. for in 不仅可以遍历数字键名，还可以遍历手动添加的其他键，包括原型链上的键（for in 可以遍历原型）

for of 可以使用 break 跳出吗

可以，forEach 不可以中途跳出，return 和 break 都不可以

什么是定时器

间隔一段时间后将函数放到任务队列中

事件冒泡

开始时由最具体的元素接收，然后逐级向上传播到 DOM 最顶层节点，但是事件冒泡也有优点，例如可以利用事件冒泡实现事件委派/代理

案例：

有 A（祖盒子）、B（父盒子）、C（子盒子）三个互相包含的盒子，假设此时三个盒子都绑定了点击事件，此时如果触发 C 盒子的点击事件，那么 A、B 盒子的点击事件也会被触发

如何阻止：

e.stopPropagation()

事件委派/代理

当有多个子节点都需要绑定相同的事件时，可以将事件直接绑定到他们共同的父节点或 document 上，然后利用事件冒泡从而影响到每个子节点（子节点触发后，冒泡到父节点，然后执行父节点绑定的事件）

事件委派就是将本该绑定给对各元素的事件，统一绑定给共同的祖先元素（例如 document，注意，绑定给 document 时需要使用 e.target 判断一下是否是被触发元素，否则会导致设都可以触发事件），这样可以降低代码复杂度，提高代码可维护性（将操作多次 DOM 变成了只操作一次）

案例：

需要给 ul 下的 li 绑定点击事件，此时可以不用给每个 li 绑定点击事件，可以将点击事件绑定在 ul 或者 document 上（绑定在 document 上时，需要使用 e.target 判断触发点击事件的是否为 li，否则会导致点击任何元素都会触发事件），然后在点击 li 时，由于事件冒泡，所以 ul 或者 document 的点击事件就会被触发

如何遍历对象

1. for in 可以遍历对象的所有可枚举属性，包括对象本身的和对象继承来的属性

```
// 创建一个对象并指定其原型，bar 为原型上的属性
const obj = Object.create({
 bar: 'bar'
})

// foo 为对象自身的属性
obj.foo = 'foo'

for (let key in obj) {
 console.log(obj[key]) // foo, bar
}
```

2. Object.keys() 该方法返回对象自身属性名组成的数组，它会自动过滤掉原型链上的属性，然后可以通过数组的 forEach() 方法来遍历

```
Object.keys(obj).forEach((key) => {
 console.log(obj[key]) // foo
})
```

```
let obj = {
    name:"ned",
    like:"man"
}
Object.keys(obj) //  ['name', 'like']
```

3. Object.values() 与 Object.keys()遍历对象的特性是相同的，但是其返回的结构是以遍历的属性值构成的数组

```
let obj = {
    name:"cornd",
    age:10
}
Object.values(obj) // ['cornd', 10]
Object.defineProperty(obj, 'like', {
  value: "coding",
  enumerable: false
});
Object.values(obj) // ['cornd', 10]
```

4. Object.entries() 返回值为 Object.values()与 Object.keys()的结合，也就是说它会返回一个嵌套数组，数组内包括了属性名与属性值

```
let obj = {
    name:"cornd",
    age:10
}
Object.entries(obj) // [['name', 'cornd'],['age', 10]]
```

5. Object.getOwnPropertyNames() 其返回结果与 Object.keys()对应，但是他得特性与其相反，会返回对象得所有属性，包括了不可枚举属性

```
var arr = [];
console.log(Object.getOwnPropertyNames(arr)); // ['length']
Object.getOwnPropertyDescriptor(arr,"length").enumerable // false
```

```
// 创建一个对象并指定其原型，bar 为原型上的属性
// baz 为对象自身的属性并且不可枚举
const obj = Object.create({
 bar: 'bar'
}, {
 baz: {
  value: 'baz',
  enumerable: false
 }
})

obj.foo = 'foo'
```

// 不包括不可枚举的 baz 属性

```
Object.keys(obj).forEach((key) => {
 console.log(obj[key]) // foo
})
```

// 包括不可枚举的 baz 属性

```
Object.getOwnPropertyNames(obj).forEach((key) => {  console.log(obj[key]) // baz, foo })
```

6. Object.getOwnPropertySymbols() 返回对象自身的 Symbol 属性组成的数组，不包括字符串属性

```
Object.defineProperties(obj, {  [Symbol('baz')]: {   value: 'Symbol baz',   enumerable: false  } })
```

```
// 给对象添加一个可枚举的 Symbol 属性
obj[Symbol('foo')] = 'Symbol foo'

Object.getOwnPropertySymbols(obj).forEach((key) => {
 console.log(obj[key]) // Symbol baz, Symbol foo
})
```

7. Reflect.ownKeys() 返回的是一个大杂烩数组，即包含了对象的所有属性，无论是否可枚举还是属性是 symbol，还是继承，将所有的属性返回

```
let arr = [0,31,2]
Reflect.ownKeys(arr) // ['0', '1', '2', 'length']
```

https://blog.csdn.net/huang_wei_xiong/article/details/124226161

如何判断一个对象是不是数组

1. instanceof 用于检验构造函数的 prototype 属性是否出现在对象的原型链中的任何位置，返回一个布尔值

```
let a = [];
a instanceof Array; // true
let b = {};
b instanceof Array; // false
```

在上方代码中，instanceof运算符检测Array.prototype属性是否存在于变量a的原型链上，显然a是一个数组，拥有Array.prototype属性所以为true。

缺点：prototype 属性是可以修改的，所以并不是最初判断为 true 就一定永远为 true

2. constructor 实例的构造函数 constructor 指向构造函数

```
let a = [1, 3, 4];
a.constructor === Array; // true
```

3. Object.prototype.toString.call() 获取到对象的不同类型

```
let a = [1, 2, 3];
Object.prototype.toString.call(a) === '[object Array]'; // true
```

4. Array.isArray() 确定传递的值是否是一个数组，返回一个布尔值

```
let a = [1, 2, 3];
Array.isArray(a); // true
```

https://blog.csdn.net/weixin_50723565/article/details/126350947

大量 if 的替代方案

1. 三元表达式
2. Switch
3. 利用数组（Array.includes()）

```
function region(province){
    let result = ""
    let northProvinceArr = ["河北","黑龙江","辽宁","山东","吉林"]
    let southProvinceArr = ["广东","广西","福建","浙江","云南"]
    if(southProvinceArr.includes(province)) result = "南方";
    if(northProvinceArr.includes(province)) result = "北方"
}
```

4. 利用对象（对象数组），将判断条件设置为键

```
const obj = {
  1: "汉族",
  2: "苗族",
  3: "维吾尔族",
  4: "回族",
  5: "藏族",
  // ...等等等等
};
function formatData(a) {
  return obj[a]
}
var result = formatData(2);  // 苗族
```

```
function del(){}; // 删除操作
function add(){}; // 新增
function update(){}; // 更新
let typeFn = {
    'del': del,
    'add': add,
    'update': update
};
function process(operateType) {
    typeFn[operateType]();
};
process('del'); //删除
```

5. Map，方式同对象

```
let typeFn = new Map([
    ['del_1', function () {/*do something*/ }],
    ['add_2', function () {/*do something*/ }],
    ['update_3', function () {/*do something*/ }],
]);

function process(operateType, status) {
    typeFn.get(`${operateType}_${status}`)();
};
process('del', 1); // 删除
```

6. 策略模式（设置模式中的一种）策略模式是开发中常用的第二种设计模式，它在开发中非常常见，由两部分组成。第一部分是策略类，封装了许多具体的，相似的算法。第二部分是环境类，接受客户请求，随后将请求委托给策略类。说的通俗一点就是将相同算法的函数存放在一个包装里边，每个函数用相同的方式拿出来，就叫做策略模式。

```
function permission(role){
    const actions = {
        operations: getOperationPermission,
        admin: getAdminPermission,
        superAdmin: getSuperAdminPermission,
        user: getUserPermission,
    }
    actions[role].call()
}
```

https://blog.csdn.net/m0_37036014/article/details/106355039
https://www.qiyuandi.com/zhanzhang/zonghe/17752.html

前端如何处理数据精度丢失问题

后端 Long 数据类型导致的精度丢失（数据过长导致精度丢失）

在 js 中 number 类型有个最大值（安全值），为 9007199254740992，是 2 的 53 次方。如果超过这个值，那么 js 会出现不精确的问题

1. 使用 bingInt，bigInt 会末尾携带 n，可以使用 toString 将 n 去除
2. 在请求时加上 transformResponse，这样前端接受就不会进行转化

```
axios.post("/order/addOrder",this.order,{
    transformResponse: [
        function (data){
            return data;
        }
    ]
}).then(({data})=>{
    console.log(data)
})
}).catch(()=>{
```

3. 协商后端将数据类型转换为 String
   https://blog.csdn.net/qq_45502336/article/details/123951173

小数相加导致精度缺失问题（0.1+0.2 !== 0.3）

```
console.log( 1 - 0.8 );  //输出 0.19999999999999996
console.log( 6 * 0.7 );  //输出 4.199999999999999
console.log( 0.1 + 0.2 );  //输出 0.30000000000000004
console.log( 0.1 + 0.7 );  //输出 0.7999999999999999
console.log( 1.2 / 0.2 );  //输出 5.999999999999999
```

产生原因：计算机能读懂的是二进制，进行运算的时候，实际上是把数字转换为了二进制进行的 这个过程 丢失了精度

解决方法：

1. 通常同时乘 10 的 n，n 为小数点后的位数（以大的为准），然后结果再除以 10 的 n 次方

```
//加
function floatAdd(arg1,arg2){
    var r1,r2,m;
    try{r1=arg1.toString().split(".")[1].length}catch(e){r1=0}
    try{r2=arg2.toString().split(".")[1].length}catch(e){r2=0}
    m=Math.pow(10,Math.max(r1,r2));
    return (arg1*m+arg2*m)/m;
}

//减
function floatSub(arg1,arg2){
    var r1,r2,m,n;
    try{r1=arg1.toString().split(".")[1].length}catch(e){r1=0}
    try{r2=arg2.toString().split(".")[1].length}catch(e){r2=0}
    m=Math.pow(10,Math.max(r1,r2));
    //动态控制精度长度
    n=(r1>=r2)?r1:r2;
    return ((arg1*m-arg2*m)/m).toFixed(n);
}
```

```
//乘
function floatMul(arg1,arg2)   {
    var m=0,s1=arg1.toString(),s2=arg2.toString();
    try{m+=s1.split(".")[1].length}catch(e){}
    try{m+=s2.split(".")[1].length}catch(e){}
    return Number(s1.replace(".","")) * Number(s2.replace(".","")) /Math.pow(10,m);
}

//除
function floatDiv(arg1,arg2){
    var t1=0,t2=0,r1,r2;
    try{t1=arg1.toString().split(".")[1].length}catch(e){}
    try{t2=arg2.toString().split(".")[1].length}catch(e){}

    r1=Number(arg1.toString().replace(".",""));

    r2=Number(arg2.toString().replace(".",""));
    return (r1/r2)*Math.pow(10,t2-t1);
}
```

例如: console.log(1-0.8); 变为 console.log((1 * 10 - 0.8 * 10) / 10)

2. toFixed(小数位数) + parseFloat() toFixed 的结果是字符串

JS 运行在浏览器中是单线程还是多线程

js 运作在浏览器中,是单线程的,js 代码始终在一个线程上执行

js 是单线程，浏览器是多线程

JS 能否多线程运行

不能，JS 实现多线程主要是靠异步

正则替换用户输入里面的特殊字符

```
function replaceStr(a) {
    a = a.replace(/(<br[^>]*>| |\s*)/g, '')
        .replace(/\&/g,"&amp;")
        .replace(/\"/g,"&quot;")
        .replace(/\'/g,"&#39;")
        .replace(/\</g,"&lt;")
        .replace(/\>/g,"&gt;");
    return a;
}
```

垃圾回收机制

标记-清除法（标记就是从一组根元素开始，递归遍历这组根元素，能到达的对象称为活动对象，不能到达的对象称为垃圾对象）

https://juejin.cn/post/6981588276356317214#heading-0

什么是垃圾回收

定期找出那些不再用到的内存（变量），然后释放其内存（为什么不是实时的找出无用内存并释放呢？因为实时开销太大了）

TypeScript 部分

什么是 TS

TypeScript 是 JS 的超集（加强版），给 JS 添加了可选的静态类型和基于类的面向对象编程，拓展了 JS 的语法，TS 是纯面向对象的编程语言，包含类和接口的概念，程序员可有用它来编写面向对象的服务端或客户端程序

TS 引入了很多面向对象程序设计的特征，主要包括

1. interface 接口
2. class 类
3. enumerated type 枚举类型
4. generic 泛型
5. module 模块

注意：

1. TS 不存在兼容性问题，因为 TS 在最终是编译成 JS
2. 泛型是指在定义函数、接口或类的时候，不预先指定具体的类型，使用时再去指定类型的一种特性。可以把泛型理解为代表类型的参数

TS 和 JS 的区别是什么

1. TS 是一种面向对象语言，JS 是一种脚本语言（尽管 JS 是基于对象的）
2. TS 支持可选参数、静态类型和接口，JS 不支持这些

为什么要用 TS（TS 的好处是什么）

1. TS 可以在代码编写时就能给出编译错误，而 JS 需要在运行时才能暴露出错误
2. TS 是强类型语言，可以明确知道数据的类型，代码可读性更高，弥补了 JS 作为弱类型语言的先天性不足

3. TS 的流程趋势逐渐增强，但是 TS 是基于 JS 的，所以不能说 TS 会替代 JS 这种说法

Vue 部分

MVVM

含义：

MVVM（Model-View-ViewModel）是视图模型双向绑定，MVVM 将 MVC 中的 C（controller）演变成了 ViewModel，Model 代表数据模型，View 代表视图（UI 组件），ViewModel 是 View 和 Model 之间的桥梁，数据会绑定到 ViewModel 并自定将数据渲染到页面上，视图变化时会通知 ViewModel 更新数据，之前是通过操作 DOM 结构来实现视图更新，现在是数据驱动视图

优点：

1. 低耦合性：View（视图）可以独立于 Model（数据模型）的变化，一个 Model 可以绑定到不同的 View 上，当 View 变化的时候 Model 可以不变换，反之亦然
2. 可重用性：可以将多个 View 放在一个 Model 中，使这些 View 重用这个 Model
3. 独立开发：开发人员可以专注于业务逻辑和数据的开发（ViewModel），设计人员可以专注于页面的设计
4. 可测试性

什么是 vue

vue 是一个渐进式的 JS 框架，具有易用、灵活、高效的特征，可以将一个页面分割成多个组件，当其它页面有类似功能时，直接让封装的组件进行复用，他是构建用户界面的声明式框架，只关心图层，不关心具体是如何实现的

Vue 响应式原理

Vue2.0

原理：

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 和 getter，在数据变动的时候发布消息给订阅者，触发相应的事件回调

缺点：

1. 深度监听时，如果 data 嵌套层数过多会导致递归过多，计算量大
2. 无法监听数组变化，vm.xxx[xxx] = xxx 这种是无法进行检测的
3. 无法监听新增和删除属性，如果直接往 data 中添加或删除属性（例如：this.data.xxx = xxx），此时添加或删除的属性不是响应式的，需要使用 Vue.set、Vue.delete 或者使用 Object.asign（{}，oldObj）（完全替换一个新对象）来进行添加或删除属性

```
-  解释：Vue 不能检测到对象属性的添加或删除。由于 Vue 会在初始化实例时对属性执行
getter/setter 转化过程，所以属性必须在 data 对象上存在才能让 Vue 转换它，这样才能让它是响应的
```

Vue3.0

原理：

使用 ES6 的 Proxy 代理，利用 Proxy 对 data 返回的对象进行代理，在修改属性的时候调用 set 方法时触发所有使用该值的位置进行更新，获取属性时调用 get 方法来获取相应的值

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截

优点：

1. 支持对数组变化的监听

defineProperty 和 Proxy 的对比

1. defineProperty 是代理到静态的值级别的，Proxy 是代理在对象级别
2. defineProperty 监听不了数组的变化，Proxy 可以
3. defineProperty 监听手段比较单一，只能监听 set 和 get, Proxy 有 10 几种监听
4. defineProperty 必须得把所有的属性全部添加 defineProperty, Proxy 会对整个对象都会进行拦截

Vue2.0 和 Vue3.0 的对比

1. vue3.0 体积更小、性能更好
2. vue3.0 有 TS 支持
3. vue3.0 新增组合式 API
4. vue3.0 支持多根节点组件（碎片），vue2.0 只支持单根节点
5. vue3.0 废除了 vue2.0 的过滤器（filter）
6. vue3.0 没有 this
7. vue3.0 响应式原理是 Proxy，vue2.0 是 Object.defineProperty+消息订阅与发布

为什么 data 是一个函数

因为组件在本质上就是一个 class 类，使用的时候会进行实例化，当一个组件被复用多次，就会创建多个实例，本质上这些实例用的都是同一个构造函数，如果 data 是一个对象，因为对象是引用数据类型，则此时会影响到所有的实例（因为它们指向的都是同一个地址），为了保证各个组件之间的不同实例之间的 data 不冲突，所以 data 只能是一个函数，data 是一个函数时，每个组件实例都有自己的作用域，每个实例相互独立，不会相互影响（函数创建了函数作用域，函数作用域中的变量不被外界影响）

> 官方解析：Vue 实例的数据对象。Vue 会递归地把 data 的 property 转换为 getter/setter，从而让 data 的 property 能够响应数据变化。当一个**组件**被定义，`data` 必须声明为返回一个初始数据对象的函数，因为组件可能被用来创建多个实例。

- 当我们组件中的data写成一个**函数**时，数据是以函数返回值形式定义的，这样每复用一次data，都会返回一份新的data，拥有**自己的作用域**，不会产生数据污染。
- 当我们组件中的data写成一个**对象**时，对象是引用数据类型，它就会共用一个内存地址，在多次使用该组件时，改变其中一个组件的值会影响全部使用该组件的值。

- 在vue中一个组件可能会被其他的组件引用，为了防止多个组件实例对象之间共用一个data，产生数据污染。将data定义成一个函数，每个组件实例都有自己的作用域，每个实例相互独立，不会相互影响initData时会将其作为工厂函数都会返回全新data对象。

函数中 return 对象可以保证每个组件实例的数据是相互独立的，互不影响，如果 data 是一个对象，则每个组件实例都所共享这个对象，则会导致数据混乱（例如组件为一个计数器，此时其中一个计数器发生变化，此时所有的计数器都会发生变化）

生命周期

vue2.0

create 阶段：vue 实例被创建

beforeCreate：最初调用触发（创建前），此时 data 和 methods 中的数据还没初始化，data 和 events 不可用

created：创建完毕，data 中有值，未挂载到 DOM（不能操作 DOM），data 和 methods 中的数据以及初始化完毕，在这里可以发送请求（Ajax 请求，获取数据）

mount 阶段：vue 实例被挂载到真实的 DOM 节点

beforeMount：在模板编译后，渲染之前触发（挂载前），可以发送请求获取数据

mounted：渲染之后触发，挂载完毕，vue 实例被挂载到真实的 DOM 上，此时可以操作 DOM

update 阶段：vue 实例里的 data 数据变化时，触发组件重新渲染

beforeUpdate：更新前，数据发生修改后，模板改变前触发，此时数据发生修改，页面 UI 结构并未重新渲染完成，切勿使用此处监听数据变化

updated：更新后，数据修改之后，模板改变之后触发，能操作最新的 DOM，避免在此处修改数据（会造成更新的死循环）

destory 阶段：vue 实例被销毁

beforeDestory：实例销毁前，组件卸载前触发，可手动销毁定时器、事件绑定等

destoryed：实例销毁，组件卸载完成

vue3.0

选项式 API

beforeDestory	===>	beforeUnmount
destoryed	===>	unmounted

组合式 API

beforeCreate	===>	setup()
created	===>	setup()
beforeMount	===>	onBeforeMount()
mounted	===>	onMounted()
beforeUpdate	===>	onBeforeUpdate()
updated	===>	onUpdated()
beforeUnmount	===>	onBeforeUnmount()
unmounted	===>	onMounted()

为什么不在 beforeCreate 中发送请求获取数据?

因为此时的 data 还没初始化完成，获取到的数据无法转存

为什么 v-if 和 v-for 不建议一起使用

在 vue2.0 中，v-if 的优先级高于 v-if，所以当两者一起使用时，会导致 v-for 遍历出来的所有的元素都添加了 v-if，从而造成严重的性能浪费（此时可考虑采用 computed 先对数据进行筛选），但是在 vue3.0 中，v-if 的优先级高于 v-for

如果非要两者一起使用，可以考虑在 v-for 的外层使用 或<template></template>标签包裹（建议使用<template></template>，因为<template></template>不会被渲染成真实的 DOM 节点，可以避免多余的 DOM 节点），并将 v-if 设置在外层的 或<template></template>上

v-if 和 v-show 的原理和区别

原理

v-if：直接将元素删除或添加元素

v-show：通过给元素设置 display: none

区别：

1. ~~v-if 通过删除或添加元素来切换元素的显示和隐藏，会触发浏览器的重排和重绘，v-show 通过设置 display 样式来切换元素的显示和隐藏，只会触发重绘，~~两者都会触发重排和重绘
2. 初始化渲染时，因为 v-if 是直接删除和添加元素来控制显示和隐藏，所以 v-if 只会渲染存在的元素，而 v-show 是通过给元素设置 display 属性来控制显示和隐藏，所以尽管元素通过 v-show 隐藏了，但是还是会被渲染，v-if 的初始化渲染性能要优于 v-show（v-if 可以理解为用到的时候才渲染，v-show 切换速度快，v-if 加载速度快）
3. 如果需要频繁的切换显示和隐藏，v-if 则会多次触发重排和重绘，v-if 则只会切换 displays 属性，只会触发重绘，所以需要频繁切换显示和隐藏时 v-show 的性能更好
4. v-if 可以搭配 else 和<template></template>标签使用（因为 v-show 需要给元素设置 display 属性，但是 template 不会被渲染成真实的 DOM，所以 template 无法设置 v-show）

computed 和 watch 的原理和区别

computed：计算属性

computed 用于计算结果并返回，结果会被缓存，当计算属性中的函数所依赖的属性（响应式属性）发生变化时才会重新计算，否则调用当前函数将会从缓存中读取结果，data 不改变，computed 不更新

使用场景：适用于一个属性受多个属性影响时使用

1. 购物车结算功能
2. 处理时间格式

watch：监听器

watch 用于监听某个值的变化（已经在 data 中定义的属性或变量，变量变化时，触发 watch），然后执行相应操作

使用场景：适用于一个属性（数据）影响多个属性（数据）时使用

1. 搜索框（联想搜索）

区别：

1. 既能用 computed 又能用 watch 实现的功能，推荐使用 computed
2. computed 有缓存，watch 没有
3. computed 中的函数必须要要把结果 return 出去，watch 不用
4. computed 是计算已有的值并返回结果，watch 用于监听某个值（data 中已经定义）的变化，然后执行相应操作
5. computed 默认第一次加载的时候就开始监听，watch 默认第一次不监听（需要监听则需要配置 immediate 属性为 true）
6. computed 不支持异步，当 computed 内有异步操作时是无法监听数据变化的；watch 支持异步操作

key 的作用和原理

作用：

1. 给每个节点做一个唯一标识，便于 Diff 算法执行时更快的找到相应的节点，提高 Diff 算法执行速度，更高效的更新虚拟 DOM

2. 在数据变化时强制更新组件，避免“就地复用”带来的副作用

原理：

vue 和 react 都是采用 Diff 算法来对比新旧节点，从而更新节点，在 vue 的 Diff 算法中，会根据新节点的 key 去比对旧节点数组中的 key，从而找到对应的旧节点，如果没找到则新增一个节点，如果节点没有 key，则会采用遍历查找的方式去找到对应的旧节点，前者是 Map 映射，后者是遍历查找，显然 map 映射的速度会更快

vue 在使用 v-for 更新已经渲染过的元素列表时，默认采用就地复用策略，如果数据项的顺序被改变，vue 不会随之移动 DOM 元素的舒徐，而是就地更新每个元素，确保它们在原本指定的索引位置上渲染

组件通讯方式

父组件>子组件：props
父组件>孙组件：provide + inject
子组件>父组件：$emit + 自定义事件
兄弟组件（拥有共同父组件）：eventBus（$emit + $on）
跨级组件：vuex

nextTick 的作用

作用：

nextTick 是 vue 提供的一个全局 API，作用是在下次 DOM 更新循环结束之后执行延迟回调，在修改数据之后使用 nextTick，则可以在回调中获取更新后的 DOM

```
this.$nextTick(() => {
  this.$refs.ipt.focus()
})
```

使用场景：当某些代码需要延迟执行，延迟到 DOM 重新渲染完毕之后

组件的 $nextTick(cb) 方法，会把 cb 回调推迟到下一个 DOM 更新周期之后执行。通俗的理解是：等组件的 DOM 更新完成之后，再执行 cb 回调函数。从而能保证 cb 回调函数可以操作到最新的 DOM 元素。

Mixin

作用：

当多个组件有相同逻辑时，可以使用 mixin 进行抽离

使用场景：

新闻列表和新闻详情页的右侧栏目，可以使用 mixin

劣势：

1. 变量来源不明确，代码不利于阅读
2. 多 mixin 可能引发变量冲突
3. Mixin 和组件可能会出现多对多的关系，使得项目得复杂度变高

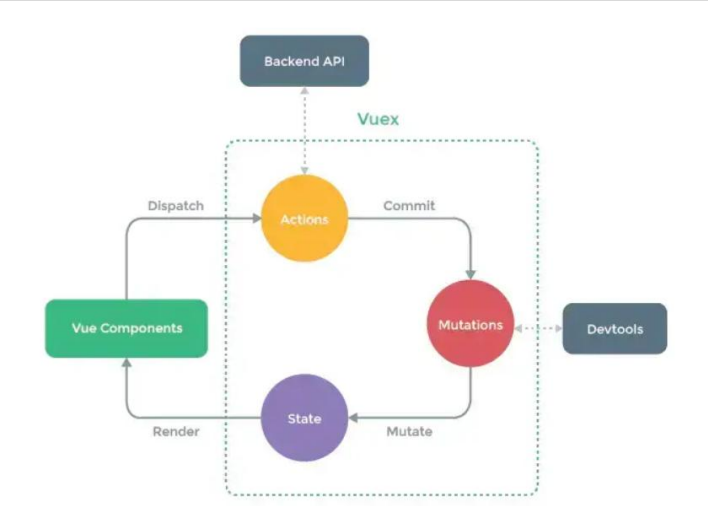
https://v2.cn.vuejs.org/v2/guide/mixins.html

Vuex

作用：状态管理工具，集中式管理应用所有组件的状态

属性：

1. state 存储数据，数据是响应式的
2. muations 处理同步操作，操作修改 state 数据
3. actions 处理异步操作，不能直接修改数据，需要借助 muations
4. getters 对数据进行加工，类似于计算属性
5. module 模块化 vuex



在 setup 中如何获取 vue 实例

setup 中没有 this，所以获取 vue 实例可以使用 getCurrentInstance

路由守卫

参数含义：

to：将要访问的路由信息对象

from：将要离开的路由对象信息

next()：

省略：不允许跳转

next()：直接放行

next('hash 地址')：强制跳转页面

next(false)：不允许跳转，停留在当前页面

全局前置守卫：

router.beforeEach((to,from,next)=>{})

```
// main.js 入口文件
    import router from './router'; // 引入路由
    router.beforeEach((to, from, next) => {
      next();
    });
    router.beforeResolve((to, from, next) => {
      next();
    });
    router.afterEach((to, from) => {
      console.log('afterEach 全局后置钩子');
    });
```

路由独享守卫：

beforeEnter:(to,form,next)=>{}

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

组件内的守卫：

```
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不！能！获取组件实例 `this`
    // 因为当守卫执行前，组件实例还没被创建
  },
  beforeRouteUpdate (to, from, next) {
    // 在当前路由改变，但是该组件被复用时调用
    // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，
    // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。
    // 可以访问组件实例 `this`
  },
  beforeRouteLeave (to, from, next) {
    // 导航离开该组件的对应路由时调用，我们用它来禁止用户离开
    // 可以访问组件实例 `this`
    // 比如还未保存草稿，或者在用户离开前，
    将setInterval销毁，防止离开之后，定时器还在调用。
  }
}
```

v-cloak

在使用插值语法（{{}}）展示或更新页面数据时：当网速比较慢时，会出现一个不好的过度现象，会让用户先看到我们的表达式，即所谓的闪烁问题

使用 v-cloak 指令，然后为其设置 css 样式 display:none 防止页面加载时出现 Vue.js 的变量名，当网络较慢，网页还在加载 Vue.js ，而导致 Vue 来不及渲染，这时页面就会显示出 Vue 源代码。我们可以使用 v-cloak 指令来解决这一问题。

keep-alive

作用：

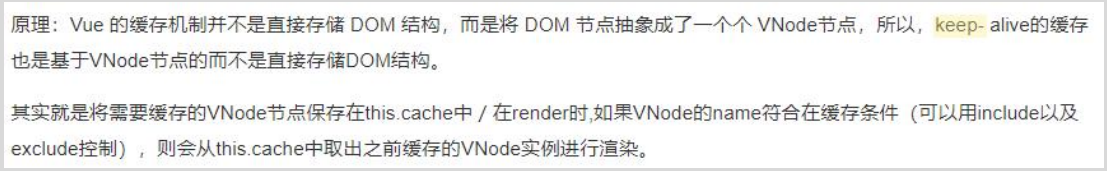
可以实现组件缓存（将不活动的组件实例保存在内存中，而不是直接将其销毁），是一个抽象组件，

不会被渲染到真实 DOM 中，也不会出现在父组件链中

可以使用 include 和 exclude 来指定要缓存和不缓存

使用 keep-alive 会触发 activated 和 deactivated 两个生命周期函数

原理：

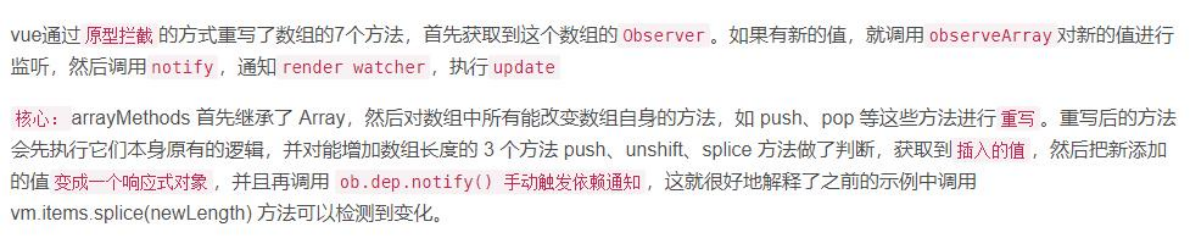


如何动态移除缓存的组件

1. keep-alive 默认不支持动态销毁已缓存的组件，所以此处给出的解决方案是通过直接操控 keep-alvie 组件里的 cahce 列表，暴力移除缓存
2. 利用 keep-alive 的 include，新打开标签时，把当前组件名加入到 include 数组里，关闭标签时从数组中删除关闭标签的组件名就可以了

https://github.com/vuejs/vue/issues/6509

Vue2.0 如何对数组方法进　行重写的



通过包裹数组更新元素的方法实现，本质就是做了两件事：

1. 调用原生对应的方法对数组进行更新
2. 重新解析模板，进而更新页面。

在 Vue 修改数组中的某个元素一定要用如下方法：

push()、pop()、shift()、unshift()、splice()、sort()、reverse()

React 部分

React 是 MVVM 框架吗

不是

React 专注的层面是 view 层，react 专注的中心是 Component，即组件，react 认为一切元素都可以抽象成组件

React 可以作为 MVVM 中的 V，即为 view，但是并不是 MVVM 框架，MVVM 的显著特点是双向数据绑定，但 react 是单向数据绑定的，react 是一个单向数据流的库，状态驱动视图，整体是函数式的思想，把组件设计成纯组件，状态和逻辑通过参数传入

class 和 hooks 的对比（类组件和函数式组件的对比）

react 提供了两套 API，一种是类（class），另外一种是函数钩子（hooks），但是 react 官方推荐使用的式函数式，因为类比较重，函数式更简洁，代码量少，比较请，而且函数式也更加符合 react 的函数式本质，hooks 是 react16.8 新增的特性

区别：

1. 类组件在大型组件时很难拆分、重构和测试，函数组件的业务代码更加聚合，事件绑定卸载也

更加的清晰，逻辑复用方便
2. 类组件引入了复杂的编程模式，如 render、props 和高阶组件
3. 类组件中涉及到 this 操作，比较容易出错
4. 函数组件的性能要比类组件好，因为类组件使用时还需要实例化，函数组件使用时直接执行函数即可
5. 在 hooks 出现之前，函数组件没有实例、生命周期、state，又称无状态组件

打包工具部分

Webpack 和 vite 的区别

1. webpack 的启动速度比 vite 慢，因为 vite 启动时不需要打包，无需分析模块依赖和编译等，当浏览器请求需要的模块时，再对模块进行编译，如此一来便缩短了编译的时间，项目越大、文件越大时的优势更加明显
2. webpack 热更新时，单个模块发生变化会对有依赖关系的所有模块进行编译，vite 改动单个模块时，只需让浏览器重新请求这个模块
3. vite 的速度要比 webpack 更快，vite 在开发时不会进行打包，而是采用 ESM 方式运行项目，在生产环境时才会对代码进行打包

loader 和 plugin 的区别

loader：

增强 webpack 处理文件的能力，会对代码进行编译

plugin：

扩展 webpack 的功能，对开发进行辅助作用

区别：

1. 两者都是为了扩展 webpack 的功能
2. loader 它只专注于转化文件（transform）这一个领域，完成压缩，打包，语言翻译；而 plugin 不仅只局限在打包，资源的加载上，还可以打包优化和压缩，重新定义环境变量等
3. loader 运行在打包文件之前（loader 为在模块加载时的预处理文件）；plugins 在整个编译周期都起作用
4. 一个 loader 的职责是单一的，只需要完成一种转换。一个 loader 其实就是一个 Node.js 模块。当需要调用多个 loader 去转换一个文件时，每个 loader 会链式的顺序执行
5. 在 webpack 运行的生命周期中会广播出许多事件，plugin 会监听这些事件，在合适的时机通过 webpack 提供的 API 改变输出结果

Webpack 常见的 loader

1. less-loader：用于将 less 编译成 css
2. css-loader：用于将 css 以 CommonJS 语法打包到 JS 中；必须配合 style-loader 共同使用，只安装 css-loader 样式不会生效。
3. style-loader：用于动态创建 style 标签，将 css 引入其中
4. sass-loader：css 预处理器
5. postcss-loader：用于补充 css 样式各种浏览器内核前缀，用于处理 css 兼容问题，需要和 postcss、postcss-preset-env 配合使用
6. babel-loader：将 Es6+ 语法转换为 Es5 语法；babel-loader 这是使 babel 和 webpack 协同工作的模块

7. ts-loader：用于配置项目 typescript
8. html-loader：想引入一个 html 页面代码片段赋值给 DOM 元素内容使用，这时就用到 html-loader
9. file-loader：用于处理文件类型资源，如 jpg，png 等图片。返回值为 publicPath 为准。
10. url-loader：处理图片类型资源，只不过它与 file-loader 有一点不同，url-loader 可以设置一个根据图片大小进行不同的操作，如果该图片大小大于指定的大小，则将图片进行打包资源，否则将图片转换为 base64 字符串合并到 js 文件里
11. vue-loader：用于编译.vue 文件
12. eslint-loader：用于检查代码是否符合规范，是否存在语法错误

Webpack 常见的 plugin

1. html-webpack-plugin：根据指定模板自动创建 html 文件，并且引入外部资源
2. mini-css-extract-plugin：将 CSS 提取为独立的文件的插件，对每个包含 css 的 js 文件都会创建一个 CSS 文件，支持按需加载 css 和 sourceMap。只能用在 webpack4 中
3. optimize-css-assets-webpack-plugin：用于压缩 css，减小 css 打包后的体积。

浏览器部分

从输入 URL 到页面加载的全过程

1. 查找缓存并比较缓存是否过期
2. DNS 域名解析域名对应 IP（DNS 服务器是基于 UDP 的，因此此处会用到 UDP 协议）
3. 根据 IP 与服务器建立 TCP 连接（三次握手）
4. 发起 HTTP 请求（三次握手的第三次发送的数据）
5. 服务器响应请求并返回结果
6. 浏览器解析数据，渲染页面（构建 DOM 树 >构建 CSS 规则树 >构建 render 树 >布局 >绘制）
7. 断开与服务器的 TCP 连接（四次挥手）

从输入 URL 到页面加载的全过程，涉及到哪些协议

1. 浏览器要将 URL 解析为 IP 地址，解析域名需要用到 DNS 协议
2. DNS 域名解析会涉及请求根域名服务器，DNS 服务器是基于 UDP 的，所以需要用到 UDP 协议
3. 建立 HTTP 连接，需要用到 HTTP 协议
4. HTTP 生成 get 请求报文，并将报文传给 TCP 层处理，需要用到 TCP 协议
5. 如果使用 HTTPS 对 HTTP 数据进行加密，需要用到 HTTPS 协议
6. TCP 的数据包发送给 IP 层，需要用到 IP 协议
7. 网段内寻址，需要用到以太网协议

为什么建立连接是三次握手，断开连接是四次挥手

服务器收到客户端的最后的报文，此时代表客户端不再向服务器发送数据，但是客户端仍可接收数据，而且此时服务器可能还有数据需要发送给客户端，所以此时服务器可以直接关闭，但是也可以发送数据给客户端后，再发送最后的保温来表示现在同意关闭连接

完整的 URL 包含什么

1. 协议：http 或 https
2. 域名：www.baidu.com 为域名，其中 baidu.com 为一级域名，www 为服务
3. 端口：不写默认为 80

4. 路径：/xx/xx
5. 查询参数：?xx=xx

https://www.baidu.com:8080/admin?type=nopass

https	协议	http或https
www.baidu.com	域名	www为服务
:8080	端口号	默认为80
/admin	路径	
?type=nopass	请求参数	

浏览器在渲染网页的时候，做了哪些事

1. 加载页面的 html 和 css
2. 将 html 转换为 DOM，将 css 转换为 CSSOM
3. 将 DOM 和 CSSOM 构建成一颗渲染树
4. 对渲染树进行构造，计算元素的位置 / 重排
5. 对网页进行绘制 / 重绘

1. 解析html代码，生成DOM tree

2. 解析css代码，生成CSSOM tree

3. 通过DOM tree 和 CSSOM tree 生成 Render tree

4. Layout（布局），计算Render tree中各个节点的位置及精确大小

5. Painting（绘制），将render tree渲染到页面上

https://www.jianshu.com/p/744edaa32fc3

重排/回流（reflow）和重绘（repaint）

重排：

当 DOM 元素的几何信息（大小和位置）发生变化，就会触发重排，此时浏览器需要重新计算元素的几何信息，重新生成布局，重新排列元素的过程就叫做重排

注意：几何信息每变化一次就会触发一次重排，重排次数过多后会严重影响页面性能

```
// 此时触发重排的次数为 3 次
box1.style.width = '300px'
box1.style.height = '400px'
box1.style.fontSize = '20px'   0.5333rem

// 此时触发重排的次数为 2 次，当display为none之后的操作不参与重排
box1.style.display = 'none'
box1.style.width = '300px'
box1.style.height = '400px'
box1.style.fontSize = '20px'
div.style.display = 'block'
```

重绘：

当 DOM 元素的外观（color、透明度）发生变化，几何信息没有发生变化，就会触发重绘，重新将元素的外观绘制出来的过程叫做重绘

注意：

1. 重绘不一定会出现重排，重排必然会出现重绘
2. 每个页面都有一次重排和重绘（页面初始化渲染的时候）
3. 使用 Vue 和 React 等框架的时候，无需担心重排和重绘问题，因为它们做了相关的优化（框架操作的都是虚拟 DOM，然后再将其转化为真实 DOM），但是需要避免在框架中修改原生 DOM 元素

如何避免：

1. 集中改变样式，避免一个个的修改元素的样式，可以采用修改元素的 class 属性来批量改变样式
2. 不要将 DOM 元素的属性值放在循环里当循环的变量

session、cookie、localStorage 和 sessionStorage 的区别

1. session 存储在服务器，cookie、localStorage 和 sessionStorage 存储在浏览器本地
2. session 要比 cookie 更加安全，cookie 可能会被盗和篡改，可以使用 read-only 设置只读
3. cookie 会在 http 请求中携带，localStorage 和 sessionStorage 只会保存在本地
4. cookie 存储的数据容量不能超过 4k，很多浏览器限制一个站点最多保存 20 个 cookie，localStorage 和 sessionStorage 存储的数据容量有 5M+
5. cookie 适合存储较小的数据，例如 session 会话标识
6. cookie 在设置的过期时间前一直有效，窗口或浏览器关闭仍然有效，sessionStorage 在当前窗口关闭前有效，localStorage 使用有效，窗口或浏览器关闭仍然有效
7. cookie 和 localStorage 在所有同源窗口中都是共享的，sessionStorage 不是

token 和 session 的区别

1. session 认证只是简单的将用户的信息存储在 session 里面，sessionID 不可预测，session 只存在于服务端
2. token 提供的是授权和认证，认证是针对用户，授权是针对 APP，目的就是让 APP 有权访问到某用户的信息，token 是唯一的
3. session 是存放在服务端，客户端只存放 sessionID，token 是存放在客户端的

跨域和同源策略

同源策略：

当协议、域名、端口中有任何一个不同，则为不同的域

解决方案：

1. JSONP（只支持 GET 请求）
   a) 创建一个 script 标签
   b) 将 script 的 src 属性设置为接口地址
   c) 接口参数，必须带一个自定义函数名
   d) 通过定义函数名去接收返回的参数

```
//动态创建 script
var script = document.createElement('script');

// 设置回调函数
function getData(data) {
    console.log(data);
}

//设置 script 的 src 属性，并设置请求地址
script.src = 'http://localhost:3000/?callback=getData';

// 让 script 生效
document.body.appendChild(script);
```

2. 配置代理（只支持开发环境）
3. 服务端设置 CROS（最终方案）
    a) 在服务器端设置 Access-Control-Allow-Origin

```
// 设置响应头
res.setHeader("Access-Control-Allow-Origin", "*")
res.setHeader("Access-Control-Allow-Methods", "GET,POST,PUT,PATCH,DELETE")
```

如何获取判断浏览器类型

window.navigator.userAgent

如何获取首屏时间

在 </head> 标签前的 <script> 标签内加入代码
new Date().getTime() - performance.timing.navigationStart

如何优化白屏时间

1. DNS 预解析
   使用 meta 标签
   <meta http-equiv="x-dns-prefetch-control" content="on" />
   使用 link 标签
   <link rel="dns-prefetch" href="https://www.baidu.com" />
1. 使用 HTTP2
   HTTP 相比于 HTTP1，解析速度更快；支持多路复用，多个请求可以共用一个 TCP 连接；提供了首部压缩功能；支持服务器推送，服务器可以在发送 HTML 页面时，主动推送其他资源，而不用等到浏览器解析到相应位置发请求再响应。
2. 减少 HTTP 请求数量，减少 HTTP 请求大小
3. 合并、压缩文件；按需加载代码，减少冗余代码
4. 采用 svg 图片或字体图标
5. 使用 defer 加载 JS（延迟加载 js）
6. 尽量将 CSS 放文件头部，JS 文件放在底部，以免堵塞渲染。JS 如果放在头部，给 script 标签加上 defer 属性，异步下载，延迟执行。
7. 服务端渲染
   客户端渲染：获取 HTML 文件，根据需要下载 JavaScript 文件并运行，生成 DOM，然后再渲染。
   服务端渲染：服务端返回 HTML 文件，客户端只需解析 HTML。
   优点：首屏渲染快，对搜索引擎优化（SEO）好。

缺点：配置麻烦，增加了服务器的计算压力。

8. 静态资源使用 内容分发网络（CDN）
   解决用户与服务器物理距离对响应时间的影响，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。
9. 资源缓存，不重复加载相同的资源
10. 图片优化（雪碧图、图片懒加载、CSS 图片懒加载）

服务端渲染和客户端渲染的区别

客户端渲染：获取 HTML 文件，根据需要下载 JavaScript 文件并运行，生成 DOM，然后再渲染。
服务端渲染：服务端返回 HTML 文件，客户端只需解析 HTML。
优点：首屏渲染快，对搜索引擎优化（SEO）好。
缺点：配置麻烦，增加了服务器的计算压力。

前后端分离（RESTAPI）和不分离

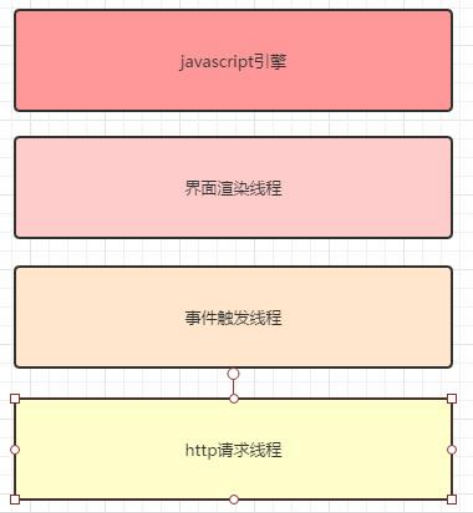
不分离：服务器成本大，一个服务器只能负责一个平台（PC、Web、移动端）的页面渲染
分离：服务器成本降低，服务器只负责数据处理，不再负责页面渲染，配合 RESTAPI 可以实现一套接口，多个平台通用

浏览器的进程有哪些

1. Browser 进程：浏览器的主进程（负责协调、主控），只有一个
   主要作用：
   负责浏览器界面显示，与用户交互。如前进，后退等
   负责各个页面的管理，创建和销毁其他进程
   将渲染（Renderer）进程得到的内存中的 Bitmap（位图），绘制到用户界面上
   网络资源的管理，下载等
2. 第三方插件进程：每种类型的插件对应一个进程，仅当使用该插件时才创建
3. GPU 进程：最多一个，用于 3D 绘制等
4. 浏览器渲染进程（即通常所说的浏览器内核）（Renderer 进程，内部是多线程的）：主要作用为页面渲染，脚本执行，事件处理等

浏览器的线程有哪些

UI 渲染线程，JS 主线程，GUI 事件触发线程，http 请求线程



网络部分

HTTP 请求/响应的步骤

1. 客户端连接到服务器
2. 发送 HTTP 请求
3. 服务器接受请求并返回 HTTP 响应
4. 释放 TCP 连接
5. 客户端（浏览器）解析 HTML 内容

HTTP 报文的组成部分

请求报文：

请求首行（请求方式 + 资源路径 + HTTP 协议版本） + 请求头 + 空行 + 请求体

请求首行：GET /index.html?username=sunwukong HTTP/1.1

响应报文：

响应首行（HTTP 协议版本 + 状态码 + 状态码描述） + 响应头 + 空行 + 响应体

响应首行：HTTP/1.1 200 OK

HTTP 状态码及常见状态码

HTTP 状态码

1xx：指示信息类，表示请求已接受，继续处理

2xx：指示成功类，表示请求已成功接受

3xx：指示重定向，表示要完成请求必须进行更近一步的操作

4xx：指示客户端错误，请求有语法错误或请求无法实现

5xx：指示服务器错误，服务器未能实现合法的请求

常见状态码

200 OK：客户端请求成功

301 Moved Permanently：永久重定向，所请求的页面已经永久重定向至新的 URL

302 Found：临时重定向，所请求的页面已经临时重定向至新的 URL

304 Not Modified 资源未修改。

403 Forbidden：对请求页面的访问被禁止

404 Not Found：请求资源不存在

500 Internal Server Error：服务器发生不可预期的错误原来缓冲的文档还可以继续使用

503 Server Unavailable：请求未完成，服务器临时过载或宕机，一段时间后可恢复正常

其余状态码

1xx（临时响应）表示临时响应并需要请求者继续执行操作的状态码

100 继续 请求者应当继续提出请求。服务器返回此代码表示已收到请求的第一部分，正在等待其余部分

101 切换协议 请求者已要求服务器切换协议，服务器已确认并准备切换

2xx（成功）表示成功处理了请求的状态码

200 成功 服务器已经成功处理了请求。通常，这表示服务器提供了请求的网页

201 已创建 请求成功并且服务器创建了新的资源

202 已接受 服务器已接受请求，但尚未处理

203 非授权信息 服务器已经成功处理了请求，但返回的信息可能来自另一来源

204 无内容 服务器成功处理了请求，但没有返回任何内容

205 重置内容 服务器成功处理了请求，但没有返回任何内容

3xx（重定向）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

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 查看其它位置 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

304 未修改 自上次请求后，请求的网页未修改过。服务器返回此响应，不会返回网页的内容

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 临时性重定向 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的位置来进行以后的请求

4xx（请求错误）这些状态码表示请求可能出错，妨碍了服务器的处理

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

410 已删除 如果请求的资源已永久删除，服务器就会返回此响应

413 请求实体过大 服务器无法处理请求，因为请求实体过大，超出了服务器的处理能力

414 请求的 URI 过长 请求的 URI（通常为网址）过长，服务器无法处理

5xx（服务器错误）这些状态码表示服务器在尝试处理请求时发生内部错误。这些错误可能是服务器本身的错误，而不是请求出错

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，从上游服务器无法收到无效响应

503 服务器不可用 服务器目前无法使用（由于超载或者停机维护）。通常，这只是暂时状态

504 网关超时 服务器作为网关代理，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

1xx 处理中

2xx 请求成功

200 请求成功

3xx 重定向

301 永久重定向（配合 location，浏览器自动处理）

302 临时重定向（配合 location，浏览器自动处理）

304 资源未被修改，浏览器存在缓存，且服务端未更新，不再向服务端发送请求

4xx 客户端错误
400 数据/格式错误
401 权限不足（身份不合格）
404 资源未找到
5xx 服务端错误
500 服务器错误
503 服务不可用（服务器超载或停机维护）
504 网关超时

HTTP 常见请求方法

GET	加载数据
POST	新建或添加数据
PUT	更新数据，全部替换
PATCH	更新数据，局部更新
DELETE	删除数据
OPTIONS	预检，浏览器自动发送，检查一些服务器的权限（是否允许跨域、支持的请求方法等）

注：
什么时候会发起 options 请求进行预检
当请求为非简单请求，且跨域时会发起 options 请求进行预检

HTTP 的缺点

1. 明文通信，安全度低
2. 不验证通信方身份
3. 无法验证报文完整性

HTTPS

HTTPS 是以安全为目标的 HTTP 通道，即 HTTP 下加入 SSL 层进行加密，作用是建立一个信息安全通道，来确保数据的传输，去报网站的真实性

HTTPS 的工作原理

1. 客户端使用 HTTPS URL 访问服务器，则要求服务器建立 SSL 链接
2. 服务器收到客户端的请求后，会将网站的证书（证书中包含了公钥）传输给客户端
3. 客户端和服务器开始协商 SSL 链接的安全等级（即为加密等级）
4. 客户端根据双方协商的安全等级，建立会话密钥，然后通过网站的公钥对会话密钥进行加密，然后传送给服务器
5. 服务器通过自己的私钥对会话密钥进行解密
6. 服务器通过会话密钥加密与客户端之间的通信

HTTPS 传输过程中为什么要使用对称加密（AES），可以用非堆成加密实现吗（RSA）

非对称加密有两个严重的问题：
1. 效率太低，会严重影响到用户打开页面的速度；
2. 服务器端只能用私钥加密，但黑客可能获取到公钥，不能保证服务器端的数据安全

对称加密：加密解密用的是同样的“钥匙”
非对称加密：加密解密用的是不同的“钥匙”

https://blog.csdn.net/Ping_Pig/article/details/90181726

HTTPS 的优缺点

1. HTTPS 要比 HTTP 协议安全，可以防止数据在传输过程中被窃取和篡改，保证数据的完整性
2. HTTPS 握手阶段比较费时，会使页面的加载时间延长 50%，增加 10%-20%的耗电
3. HTTPS 缓存不如 HTTP，会增加数据开销
4. SSL 证书需要钱，功能越强大的证书的费用越贵
5. SSL 证书需要绑定 IP，不能在同一个 IP 上绑定多个域名，IPV4 资源无法支持这种消耗

HTTP 和 HTTPS 的区别

1. HTTP 是超文本传输协议，是明文传输，安全性较低，HTTPS 测试具有安全性的 SSL 加密传输协议
2. HTTPS 需要 CA 证书，费用较高
3. HTTP 的默认端口是 80，HTTPS 的默认端口是 443
4. HTTP 的连接很简单，是无状态的

什么时候会发起 options 预检请求

当请求为非简单请求，且请求跨域时

GET 和 POST 的区别

1. get 在回退时没有副作用，post 在回退时会再次提交请求
2. get 的请求地址会被浏览器主动缓存，post 不会（除非手动设置）
3. get 的请求参数会被完整保留在浏览器历史记录里面，post 的参数不会被保留
4. get 的参数有长度显示（受 URL 长度显示，最长为 2048 字节），post 无限制
5. get 的参数通过 URL 传递，明文暴露，不安全，post 的参数时存放在请求体中的，相对更安全
6. get 适用于获取数据，post 适用于提交数据

TCP/IP 网络模型和 OSI 七层模型

链路层：负责封装和解封装 IP 报文，发送和接受 ARP/RARP 报文等。
网络层：负责路由以及把分组报文发送给目标网络或主机。
传输层：负责对报文进行分组和重组，并以 TCP 或 UDP 协议格式封装报文。
应用层：负责向用户提供应用程序，比如 HTTP、FTP、Telnet、DNS、SMTP 等

OSI七层模型	TCP/IP概念层模型	功能	TCP/IP协议族
应用层	应用层	文件传输，电子邮件，文件服务，虚拟终端	TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
表示层		数据格式化，代码转换，数据加密	没有协议
会话层		解除或建立与别的接点的联系	没有协议
传输层	传输层	提供端对端的接口	TCP，UDP
网络层	网络层	为数据包选择路由	IP，ICMP，RIP，OSPF，BGP，IGMP
数据链路层	链路层	传输有地址的帧以及错误检测功能	SLIP，CSLIP，PPP，ARP，RARP，MTU
物理层		以二进制数据形式在物理媒体上传输数据	ISO2110，IEEE802，IEEE802.2

TCP 三次握手

第一次握手　客户端 > 服务器

客户端向服务器发送连接请求（表示将要进行数据传输），等待服务器确认

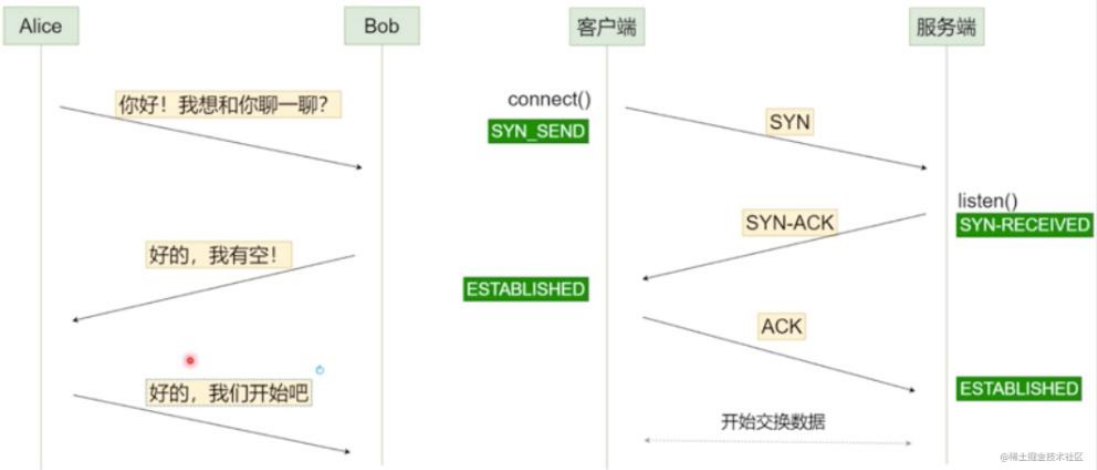
第二次握手　服务器 > 客户端

服务器收到客户端的连接请求，向客户端返回消息（表示可以进行数据传输），等待客户端确认

第三次握手　客户端 > 服务器

客户端向服务器发送同意连接的信息，三次握手结束

注意：握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据



TCP 四次挥手

第一次挥手　客户端 > 服务器

客户端向服务器发送请求，通知服务器客户端数据已经发送完毕，请求断开连接

第二次挥手　服务器 > 客户端

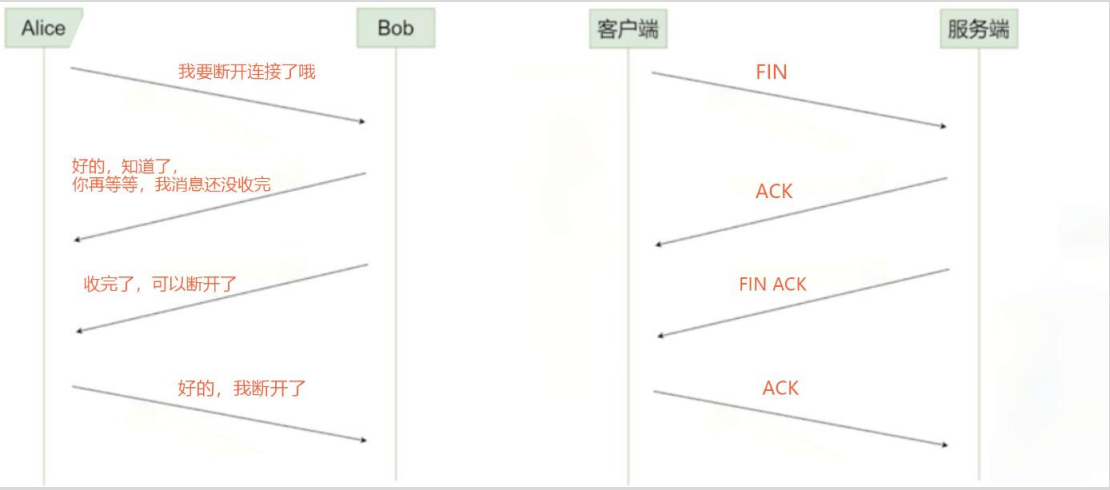
服务器收到客户端请求断开连接请求，并返回消息表示已知晓，但此时客户端还不能断开连接，因为不知道服务器是否接收完了客户端发送的信息，可能存在丢包导致消息未接收完毕

第三次挥手　服务器 > 客户端

服务器向客户端返回数据，表示客户端发送的数据已经接收完毕，可以断开连接

第四次挥手　客户端 > 服务器

客户端向服务器发送断开连接的信息，四次挥手结束



TCP 和 UDP 的区别

1. TCP 是面向连接的，而 UDP 是面向无连接的
2. TCP 仅支持单播传输，UDP 提供了单播，多播，广播的功能
3. TCP 的三次握手保证了连接的可靠性; UDP 是无连接的、不可靠的一种数据传输协议，首先不可靠性体现在无连接上，通信都不需要建立连接，对接收到的数据也不发送确认信号，发送端不知道数据是否会正确接收
4. UDP 的头部开销比 TCP 的更小，数据传输速率更高，实时性更好。

DNS 的查询过程

1. 检查浏览器缓存中是否存在该域名与 IP 地址的映射关系，如果有则解析结束，没有则继续
2. 到系统本地查找映射关系，一般在 hosts 文件中，如果有则解析结束，否则继续
3. 到本地域名服务器去查询，有则结束，否则继续
4. 本地域名服务器查询根域名服务器，该过程并不会返回映射关系，只会告诉你去下级服务器(顶级域名服务器)查询
5. 本地域名服务器查询顶级域名服务器(即 com 服务器)，同样不会返回映射关系，只会引导你去二级域名服务器查询
6. 本地域名服务器查询二级域名服务器(即 baidu.com 服务器)，引导去三级域名服务器查询
7. 本地域名服务器查询三级域名服务器(即 mail.baidu.com 服务器)，此时已经是最后一级了，如果有则返回映射关系，则本地域名服务器加入自身的映射表中，方便下次查询或其他用户查找，同时返回给该用户的计算机，没有找到则网页报错
8. 如果还有下级服务器，则依此方法进行查询，直至返回映射关系或报错
9. 进行缓存

浏览器缓存-系统本地-本地域名服务器-根域名服务器-顶级域名服务器-二级域名服务器-三级域名服务器

客户端和服务端长连接的几种方式

1. Ajax 轮询

   实现原理：ajax 轮询指客户端每间隔一段时间向服务端发起请求，保持数据的同步。

   优点：可实现基础（指间隔时间较短）的数据更新。

   缺点：这种方法也只是尽量的模拟即时传输，但并非真正意义上的即时通讯，很有可能出现客户端请求时，服务端数据并未更新。或者服务端数据已更新，但客户端未发起请求。导致多次请求资源浪费，效率低下。【数据更新不及时，效率低下】

2. Long pull 长轮询

   实现原理：long poll 指的是客户端发送请求之后，如果没有数据返回，服务端会将请求挂起放入队列（不断开连接）处理其他请求，直到有数据返回给客户端。然后客户端再次发起请求，以此轮询。在 HTTP1.0 中客户端可以设置请求头 Connection:keep-alive，服务端收到该请求头之后知道这是一个长连接，在响应报文头中也添加 Connection:keep-alive。客户端收到之后表示长连接建立完成，可以继续发送其他的请求。在 HTTP1.1 中默认使用了 Connection:keep-alive 长连接。

   优点：减少客户端的请求，降低无效的网络传输，保证每次请求都有数据返回，不会一直占用线程。

   缺点：无法处理高并发，当客户端请求量大，请求频繁时对服务器的处理能力要求较高。服务器一直保持连接会消耗资源，需要同时维护多个线程，服务器所能承载的 TCP 连接数是有上限的，这种轮询很容易把连接数顶满。每次通讯都需要客户端发起，服务端不能主动推送。【无法处理高

并发，消耗服务器资源严重，服务端不能主动推送】

3. Iframe 长连接

实现原理：在网页上嵌入一个 iframe 标签，该标签的 src 属性指向一个长连接请求。这样服务端就可以源源不断地给客户端传输信息。保障信息实时更新。

优点：消息及时传输。

缺点：消耗服务器资源。

4. Websocket

实现原理：

Websocket 实现了客户端与服务端的双向通信，只需要连接一次，就可以相互传输数据，很适合实时通讯、数据实时更新等场景。

Websocket 协议与 HTTP 协议没有关系，它是一个建立在 TCP 协议上的全新协议，为了兼容 HTTP 握手规范，在握手阶段依然使用 HTTP 协议，握手完成之后，数据通过 TCP 通道进行传输。

Websoket 数据传输是通过 frame 形式，一个消息可以分成几个片段传输。这样大数据可以分成一些小片段进行传输，不用考虑由于数据量大导致标志位不够的情况。也可以边生成数据边传递消息，提高传输效率。

优点：

双向通信。客户端和服务端双方都可以主动发起通讯。

没有同源限制。客户端可以与任意服务端通信，不存在跨域问题。

数据量轻。第一次连接时需要携带请求头，后面数据通信都不需要带请求头，减少了请求头的负荷。

传输效率高。因为只需要一次连接，所以数据传输效率高。

缺点：

长连接需要后端处理业务的代码更稳定，推送消息相对复杂；

长连接受网络限制比较大，需要处理好重连。

兼容性，WebSocket 只支持 IE10 及其以上版本。

服务器长期维护长连接需要一定的成本，各个浏览器支持程度不一；

成熟的 HTTP 生态下有大量的组件可以复用，WebSocket 则没有，遇到异常问题难以快速定位快速解决。【需要后端代码稳定，受网络限制大，兼容性差，维护成本高，生态圈小】

WebSocket

1. 实现了浏览器与服务器全双工通信
2. 基于 TCP 的，是可靠性传输协议
3. 是应用层协议
4. WebSocket 是双向通信协议，模拟 Socket 协议，可以双向发送或接受信息，HTTP 是单向的
5. WebSocket 是需要浏览器和服务器握手进行建立连接的，而 http 是浏览器发起向服务器的连接，服务器预先并不知道这个连接
6. Web 端实现即时通讯的方法

利用 Socket 建立网络连接的步骤

建立 Socket 连接至少需要一对套接字，其中一个运行于客户端，称为 ClientSocket ，另一个运行于服务器端，称为 ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

1、服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

2、客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。

为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

3、连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。

而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

非对称加密 RSA

简介：

1. 对称加密算法又称现代加密算法
2. 非对称加密是计算机通信安全的基石，保证了加密数据不会被破解
3. 非对称加密算法需要两个密钥：公开密钥(publickey) 和私有密钥(privatekey)
4. 公开密钥和私有密钥是一对

如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密

如果用私有密钥对数据进行加密，只有用对应的公开密钥才能解密

特点：

1. 算法强度复杂，安全性依赖于算法与密钥
2. 加密解密速度慢

与对称加密算法的对比：

1. 对称加密只有一种密钥，并且是非公开的，如果要解密就得让对方知道密钥
2. 非对称加密有两种密钥，其中一个是公开的

RSA 应用场景：

由于 RSA 算法的加密解密速度要比对称算法速度慢很多，在实际应用中，通常采取数据本身的加密和解密使用对称加密算法(AES)。 用 RSA 算法加密并传输对称算法所需的密钥

HTTP1、HTTP2 和 HTTP3 之间的关系

HTTP/2 相比于 HTTP/1.1，可以说是大幅度提高了网页的性能，只需要升级到该协议就可以减少很多之前需要做的性能优化工作，虽如此但 HTTP/2 并非完美的，HTTP/3 就是为了解决 HTTP/2 所存在的一些问题而被推出来的

HTTP1.1 的缺陷

1. 高延迟 — 队头阻塞(Head-Of-Line Blocking)

   队头阻塞是指当顺序发送的请求序列中的一个请求因为某种原因被阻塞时，在后面排队的所有请求也一并被阻塞，会导致客户端迟迟收不到数据。

   针对队头阻塞的解决办法:

   1. 将同一页面的资源分散到不同域名下，提升连接上限
   2. 合并小文件减少资源数`，使用精灵图
   3. 内联(Inlining)资源`是另外一种防止发送很多小图请求的技巧，将图片的原始数据嵌入在 CSS 文件里面的 URL 里，减少网络请求次数
   4. 减少请求数量，合并文件

2. 无状态特性 — 阻碍交互

无状态是指协议对于连接状态没有记忆能力`。纯净的 HTTP 是没有 cookie 等机制的，每一个连接都是一个新的连接。

Header 里携带的内容过大，在一定程度上增加了传输的成本`。且请求响应报文里有大量字段值都是重复的。

3. 明文传输 — 不安全性

HTTP/1.1 在传输数据时，所有`传输的内容都是明文`，客户端和服务器端都无法验证对方的身份，无法保证数据的安全性。

4. 不支持服务端推送

记忆口诀：队头阻塞高延迟，无状态阻交互，明文传输不安全，服务推送不支持。

HTTP1.1 的排队问题

HTTP 1.1 多个文件共用一个 TCP，这样可以减少 tcp 握手，这样 3 个文件就不用握手 9 次了，不过这样请求文件需要排队，请求和返回都需要排队， 如果第一个文件响应慢，会阻塞后面的文件，这样就产生了对头的等待问题。

有的网站可能会有很多文件，浏览器处于对机器性能的考虑，它不可能让你无限制的发请求建连接，因为建立连接需要占用资源，浏览器不想把用户的网络资源都占用了，所以浏览器最多会建立 6 个 tcp 连接；如果有上百个文件可能都需要排队，http2.0 正在解决这个问题。

HTTP2 简介

HTTP/2 是现行 HTTP 协议（HTTP/1.x）的替代，但它不是重写，HTTP/2 基于 SPDY，专注于性能，最大的一个目标是在用户和网站间只用一个连接（connection）

HTTP2 新特性

1. 二进制传输

HTTP/2 传输数据量的大幅减少,主要有两个原因:以二进制方式传输和 Header 压缩。我们先来介绍二进制传输,HTTP/2 采用二进制格式传输数据，而非 HTTP/1.x 里纯文本形式的报文 ，二进制协议解析起来更高效。HTTP/2 将请求和响应数据分割为更小的帧，并且它们采用二进制编码。

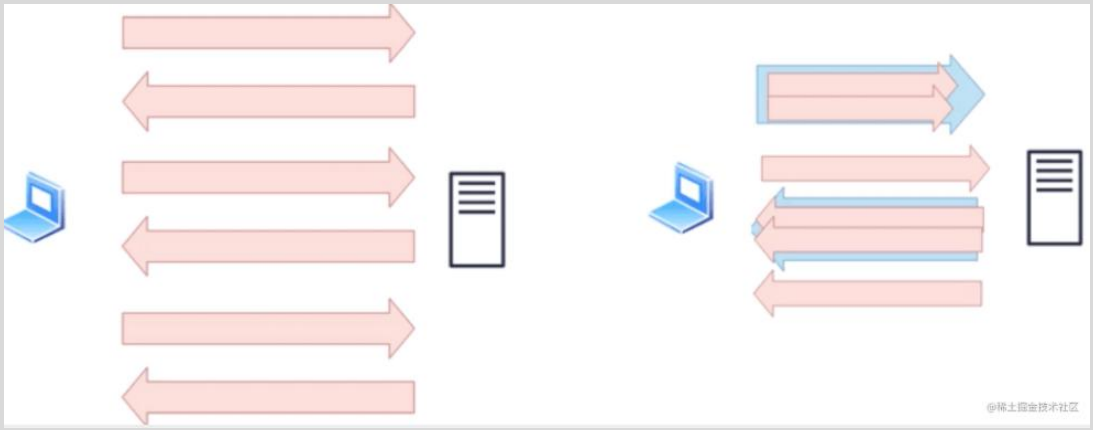
2. Header 压缩（首部压缩）

HTTP/2 并没有使用传统的压缩算法，而是开发了专门的"HPACK"算法，在客户端和服务器两端建立“字典”，用索引号表示重复的字符串，还采用哈夫曼编码来压缩整数和字符串，可以达到 50%~90% 的高压缩率。

3. 多路复用

在 HTTP/2 中引入了多路复用的技术。多路复用很好的解决了浏览器限制同一个域名下的请求数量

的问题，同时也更容易实现全速传输。



4. Server Push（服务端推送）

HTTP2 还在一定程度上改变了传统的“请求-应答”工作模式，服务器不再是完全被动地响应请求，也可以新建“流”主动向客户端发送消息。减少等待的延迟，这被称为"服务器推送"（ Server Push，也叫 Cache push）

5. 提高安全性

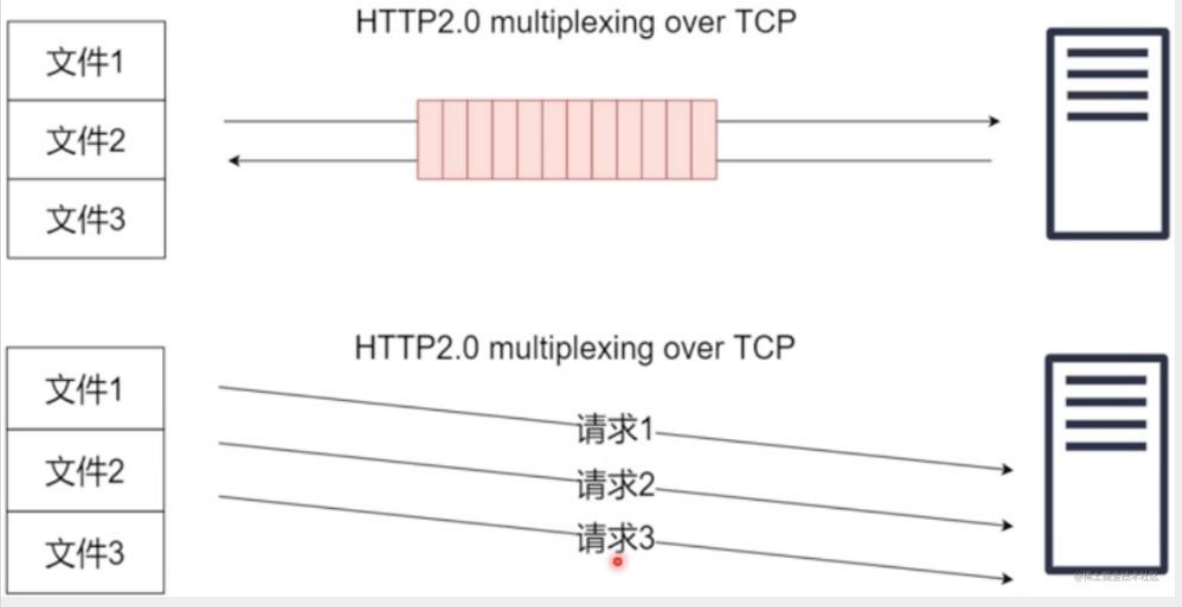
出于兼容的考虑，HTTP/2 延续了 HTTP/1 的“明文”特点，可以像以前一样使用明文传输数据，不强制使用加密通信，不过格式还是二进制，只是不需要解密。

但由于 HTTPS 已经是大势所趋，而且主流的浏览器 Chrome、Firefox 等都公开宣布只支持加密的 HTTP/2，所以“事实上”的 HTTP/2 是加密的。也就是说，互联网上通常所能见到的 HTTP/2 都是使用"https"协议名，跑在 TLS 上面。HTTP/2 协议定义了两个字符串标识符：“h2"表示加密的 HTTP/2，“h2c”表示明文的 HTTP/2。

6. 防止对头阻塞

http1.1 如果第一个文件阻塞，第二个文件也就阻塞了

http2.0 的解决，把 3 个请求打包成一个小块发送过去，即使第一个阻塞了，后面 2 个也可以回来；相当于 3 个文件同时请求，就看谁先回来谁后回来，阻塞的可能就后回来，对带宽的利用是最高的；但没有解决 TCP 的对头阻塞，如果 TCP 发过去的一个分包发丢了，他会重新发一次；http2.0 的解决了大文件的阻塞。



一个分包请求 3 个文件，即使第一个阻塞了，第二个也能返回

HTTP2 的缺陷

虽然 HTTP/2 解决了很多之前旧版本的问题，但它还是存在一个巨大的问题，主要是底层支撑的 TCP 协议造成的。HTTP/2 的缺点主要有以下几点：

1. TCP 以及 TCP+TLS 建立连接时延时
2. TCP 的队头阻塞并没有彻底解决
3. 多路复用导致服务器压力上升也容易 Timeout（超时）

HTTP3 简介

Google 在推 SPDY 的时候就搞了个基于 UDP 协议的“QUIC”协议，让 HTTP 跑在 QUIC 上而不是 TCP 上。而“HTTP over QUIC”就是 HTTP/3，真正“完美”地解决了“队头阻塞”问题。

HTTP3 新特性（QUIC 新功能）

QUIC 虽然基于 UDP，但是在原本的基础上新增了很多功能，QUIC 基于 UDP，而 UDP 是“无连接”的，根本就不需要“握手”和“挥手”，所以就比 TCP 来得快。此外 QUIC 也实现了可靠传输，保证数据一定能够抵达目的地。它还引入了类似 HTTP/2 的“流”和“多路复用”，单个“流"是有序的，可能会因为丢包而阻塞，但其他“流”不会受到影响。具体来说 QUIC 协议有以下特点：

1. 实现了类似 TCP 的流量控制、传输可靠性的功能

   虽然 UDP 不提供可靠性的传输，但 QUIC 在 UDP 的基础之上增加了一层来保证数据可靠性传输。它提供了数据包重传、拥塞控制以及其他一些 TCP 中存在的特性。
2. 实现了快速握手功能

   由于 QUIC 是基于 UDP 的，所以 QUIC 可以实现使用 0-RTT 或者 1-RTT 来建立连接，这意味着 QUIC 可以用最快的速度来发送和接收数据，这样可以大大提升首次打开页面的速度。0RTT 建连可以说是 QUIC 相比 HTTP2 最大的性能优势。
3. 集成了 TLS 加密功能
4. 多路复用，彻底解决 TCP 中队头阻塞的问题

   和 TCP 不同，QUIC 实现了在同一物理连接上可以有多个独立的逻辑数据流。实现了数据流的单独传输，就解决了 TCP 中队头阻塞的问题。

5. 连接迁移

TCP 是按照 4 要素（客户端 IP、端口，服务器 IP、端口）确定一个连接的。而 QUIC 则是让客户端生成一个 Connection ID （64 位）来区别不同连接。只要 Connection ID 不变，连接就不需要重新建立，即便是客户端的网络发生变化。由于迁移客户端继续使用相同的会话密钥来加密和解密数据包，QUIC 还提供了迁移客户端的自动加密验证。

HTTP1.0 和 HTTP1.1 的对比

长连接：HTTP1.0需要使用keep-alive参数来告知服务器建立一个长连接，而HTTP1.1默认支持长连接

节约宽带：HTTP1.1支持只发送一个header信息（不带任何body信息）

host域（设置虚拟站点，也就是说，web server上的多个虚拟站点可以共享同一个ip端口）：HTTP1.0没有host域

HTTP1、HTTP2 和 HTTP3 对比

1. HTTP/1.1 有两个主要的缺点：安全不足和性能不高
2. HTTP/2 完全兼容 HTTP/1，是“更安全的 HTTP、更快的 HTTPS”，二进制传输、头部压缩、多路复用、服务器推送等技术可以充分利用带宽，降低延迟，从而大幅度提高上网体验；
3. QUIC 基于 UDP 实现，是 HTTP/3 中的底层支撑协议，该协议基于 UDP，又取了 TCP 中的精华，实现了即快又可靠的协议

线程、进程和协程的概念

进程

进程作为拥有资源的基本单位，线程作为调度和分配的基本单位。进程就是一段程序的执行过程例如启动的某个 app。进程拥有代码和打开的文件资源、数据资源、独立的内存空间，进程拥有独立的内存空间

线程

有时被称为轻量级进程（LightWeight Process，LWP），是操作系统调度（CPU 调度）执行的最小单位。线程共享所在进程中的内存空间

协程

又称微线程，纤程。英文名 Coroutine。一句话说明什么是线程：协程是一种用户态的轻量级线程，协程的调度完全由用户控制（进程和线程都是由 cpu 内核进行调度）。

进程与进程之间完全隔离，互不干扰，一个进程崩溃不会影响其他进程，避免一个进程出错影响整个程序

线程、进程和协程的关系

1. 进程与进程之间需要传递某些数据的话，就需要通过进程通信管道 IPC 来传递
2. 一个进程中可以并发多个线程，每个线程并行执行不同的任务
3. 一个进程中的任意一个线程执行出错，会导致这个进程崩溃
4. 同一进程下的线程之间可以直接通信和共享数据
5. 当一个进程关闭之后，操作系统会回收该进程的内存空间

进程间通信的方式

管道通信：

就是操作系统在内核中开辟一段缓冲区，进程 1 可以将需要交互的数据拷贝到这个缓冲区里，进程 2 就可以读取了

消息队列通信：

消息队列就是用户可以添加和读取消息的列表，消息队列里提供了一种从一个进程向另一个进程发送数据块的方法，不过和管道通信一样每个数据块有最大长度限制

共享内存通信：

就是映射一段能被其他进程访问的内存，由一个进程创建，但多个进程都可以访问，共享进程最快的是`IPC`方式`信号量通信`：比如信号量初始值是 1，进程 1 来访问一块内存的时候，就把信号量设为 0，然后进程 2 也来访问的时候看到信号量为 0，就知道有其他进程在访问了，就不访问了

socket：

其他的都是同一台主机之间的进程通信，而在不同主机的进程通信就要用到 socket 的通信方式了，比如发起 http 请求，服务器返回数据

安全问题-SQL 注入

原理：

将 sql 代码伪装到输入参数中，然后传递给服务器进行攻击（利用服务器在执行 SQL 操作时拼接相应参数）

解决方案：

1. 对用户输入进行校验
2. 不使用动态拼接 SQL

安全问题-XSS 跨站脚本攻击（脚本注入）

原理：

攻击者利用表单填写将恶意代码混入其中，在目标网站上注入恶意代码，当用户登录网站时就会执行这些恶意代码，这些脚本可以读取 cookie，session tokens，或者其它敏感的网站信息，对用户进行钓鱼欺诈，甚至发起蠕虫攻击等

解决方案：

1. url 参数使用 encodeURIComponent 方法转义
2. 尽量不使用 innerHtml 插入 HTML 内容（避免使用 innerHTML，建议使用 textContent 或 innerText 替代，textContent 可以转义标签）
3. 使用特殊符号、标签转义符（将特殊符号进行转义，例如<和>）

安全问题-CSRF 跨站请求伪装

原理：

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的。

简而言之就是用户在 A 网站成功登录，存在 cookie 登录信息，然后坏人在 B 网站（需要诱导用户进入 B 网站），利用用户在 A 网站的登录信息（浏览器保存的 cookie）向服务器发送请求（利用请求自动携带 cookie），例如删除某个数据，服务器收到请求并验证是否存在 cookie，因为 B 网站利用的用户在 A 网站登录所保存的 cookie 信息，服务器会默认认为坏人是用户本人，所以浏览器会执行相应操作，当然 B 网站也是在用户的同一台电脑和浏览器上进行的操作，post 请求也可以实现（最简单的就是在网站中设置一个 ifream 然后将其隐藏，目前大部分浏览器可以识别这种操作）

现在大部分的浏览器的都不会在跨域的情况下自动发送 cookie，这个设计就是为了避免 csrf 的攻击

解决方案：

1. 使用 referer 头来检查请求的来源
2. 使用验证码
3. 尽量使用 post 请求（结合 token）

安全问题-DDOS

原理：

DDOS 又叫分布式拒绝服务，其原理就是利用大量的请求造成资源过载，导致服务不可用。

解决方案：

1. 限制单 IP 请求频率。
2. 防火墙等防护设置禁止 ICMP 包等
3. 检查特权端口的开放

Web2.0

Web2.0 对应的是移动互联网，用户不再只是内容接收方，可以在线阅读、点评、制造内容，成为内容的提供方，还可以与其他用户进行交流沟通。提供服务的网络平台成为中心和主导，聚集起海量网络数据。

Web3.0

假如说 web1.0 的本质是联合，那么 web2.0 的本质就是互动，它让网民更多地参与信息产品的创造、

传播和分享，而这个过程是有价值的。web2.0 的缺点是没有体现出网民劳动的价值，所以 2.0 很脆弱，缺乏商业价值。web2.0 是脆弱的，纯粹的 2.0 会在商业模式上遭遇重大挑战，需要跟具体的产业结合起来才会获得巨大的商业价值和商业成功。web3.0 是在 web2.0 的基础上发展起来的能够更好地体现网民的劳动价值，并且能够实现价值均衡分配的一种互联网方式。（运行在区块链技术上的去中心化互联网），“Web3.0”是对“Web2.0”的改进，在此环境下，用户不必在不同中心化的平台创建多种身份，而是能打造一个去中心化的通用数字身份体系，通行各个平台。

总体而言，web3.0 更多的不是仅仅一种技术上的革新。而是以统一的通讯协议，通过更加简洁的方式为用户提供更为个性化的互联网信息资讯定制的一种技术整合。将会是互联网发展中由技术创新走向用户理念创新的关键一步。

DNS 只能拿到 IP，是如何将 IP 转为 mac 地址

通过 arp 协议，arp 维护一个本地的高速缓存表，里面有 ip 到 mac 的映射，若没有则广播消息查找

Git 部分

Git 常见命令