[同步和异步 2](#_Toc1398)

[日期 2](#_Toc12175)

[随机数 、数组的随机排序 3](#_Toc28218)

[JS面向对象——封装、继承（原型链继承、借用构造函数、组合继承）、多态 4](#_Toc11976)

[从输入URL到页面加载发生了什么（包含了浏览器渲染页面的流程） 9](#_Toc7461)

[设计模式（单例模式、MVC、MVVM） 12](#_Toc4329)

[Promise 13](#_Toc19515)

[浏览器内核的理解 13](#_Toc7382)

[三列布局（圣杯布局、双飞翼布局、使用绝对定位、分别左浮动和右浮动） 13](#_Toc4091)

###### 同步和异步

**两者区别：**

1.同步会阻塞代码执行，而异步不会。

2.比如alert是同步，setTimeout是异步。

**前端使用异步的场景：（这些场景的特点都是需要等待）**

1.定时任务：setTimeout,setInterval

2.网络请求：ajax请求，动态<img>加载

3.事件绑定

**例子**：

console.log(1);

console.log(setTimeout(function(){

console.log(2);

},1000));

console.log(3);

结果：

1

2850

3

2

解释：setTimeout执行后会立马返回一个数值id，但setTimeout中定义的函数会先暂存起来，继续执行后续代码，最后再执行暂存起来的函数。

###### 日期

Date.now(); //获取当前时间毫秒数

var dt = new Date(); // 如Wed Sep 06 2017 15:39:58 GMT+0800 (中国标准时间) 一般用不到

dt.getTime(); //获取毫秒数

dt.getFullYear(); //年 ，四位数

dt.getMonth(); //月（0-11），真实的月份要加上1

dt.getDate(); //日（0-31）

dt.getHours(); //小时（0-23）

dt.getMinutes(); //分钟（0-59）

dt.getSeconds(); //秒（0-59）

**获取2017-06-10格式的日期**

function formatDate(dt){

var year = dt.getFullYear();

var month = dt.getMonth() + 1;

var date = dt.getDate();

if(month < 10){

month = '0' + month;

}

if(date < 10){

date = '0' + date;

}

return year + '-' + month + '-' + date;

}

var dt = new Date();

var formatDate = formatDate(dt);

console.log(formatDate);

###### 随机数 、数组的随机排序

Math.random() 产生一个[0，1)之间的随机数，每次执行，小数点后的位数并不是一致的

**获取随机数，要求是长度一致的字符串格式**

var random = Math.random();

var random = random + "0000000000";

var random = random.slice(0,10); //要求长度为10

console.log(random);

**数组随机排序方法：**

**法一：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

for(var i = 0;i < arr.length;i++){

var index = parseInt(Math.random() \*　arr.length);

var temp = arr[index];

arr[index] = arr[i],

arr[i] = temp;

}

return arr;

}

console.log(randomSort(arr));

**法二：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

var newArr = [];

while(arr.length > 0){

var index = parseInt(Math.random() \*　arr.length);

newArr.push(arr[index]);

arr.splice(index,1);

}

return newArr;

}

console.log(randomSort(arr));

**法三：**

var arr = [1,2,3,4,5,6,7];

function randomSort(arr){

arr.sort(function(){

return Math.random() - 0.5;

});

return arr;

}

console.log(randomSort(arr));

###### JS面向对象——封装、继承（原型链继承、借用构造函数、组合继承）、多态

1. **封装**：

我们将一类实体共同具有的属性和行为提取出来抽象为一个类，将这些属性和行为都放在这个类里。就好像人类，可以具有name,sex,age等属性，同时也具有eat(),sleep()等行为，这就可以把他们封装成一个Person类。JS这种弱类型语言不像强类型语言中通过class关键字实现类的封装（ES6另谈），而是通过声明一个函数，在函数内部通过为this变量添加属性或者方法来实现对类添加属性和方法。

如：（Book例子）

function Book(id,name){

this.id = id;

this.name = name

}

**也可以在类的原型上添加属性和方法：**

Book.prototype.display = function(){};

//或者

Book.prototype = {

display : function(){}

};

通过this添加属性和在protoytpe中添加属性的**区别**：通过类创建一个新对象时，this指向的属性和方法都会得到相应的创建，而通过prototype继承的属性或方法是每个对象通过prototype访问到，所以每次创建一个新对象时这些属性和方法不会再次创建。

1. **继承——父类的特性传给子类**

**1.借用构造函数继承**

function Parent0(){

this.name = "parent0";

this.colors = ["red","blue","yellow"];

}

function Child0(){

**Parent0.call( this ); // 或apply**

this.type = "child0";

}

第6行，在子类（Child0）中执行父类（Parent0）的构造函数，通过这种调用，把父类构造函数的this指向为子类实例化对象引用，从而导致父类执行的时候父类里面的属性都会被挂载到子类的实例上去。

new Child0().name; // Parent0

new Child0().colors; // (3) ["red", "blue", "yellow"]

但是通过这种方式，父类原型上的东西是没法继承的

Parent0.prototype.sex = "男";

Parent0.prototype.say = function() {

console.log(" Oh,My God! ");

}

**new Child0().sex; // undefined**

**// Uncaught TypeError: (intermediate value).say is not a function**

**new Child0().say();**

缺点：Child1无法继承Parent1的原型对象，并没有真正的实现继承（部分继承）

**2.原型链式继承（借用原型链实现继承）**

function Parent1(){

this.name = "parent1";

this.colors = ["red","blue","yellow"];

}

function Child1(){

this.name = "child1";

}

**Child1.prototype = new Parent1();**

这种方式能否解决借用构造函数继承的缺点呢？来看下面代码，我们依然为父类的原型添加sex属性和say方法：

Parent1.prototype.sex = "男";

Parent1.prototype.say = function() {

console.log(" Oh,My God! ");

}

**new Child1().sex; // 男**

**new Child1().say(); // Oh,My God!**

这种方式确实解决了上面借用构造函数继承方式的缺点。

但是，这种方式仍有缺点，我们来看如下代码：

var s1 = new Child1();

s1.colors.push("black");

var s2 = new Child1();

**s1.colors; // (4) ["red", "blue", "yellow", "balck"]**

**s2.colors; // (4) ["red", "blue", "yellow", "balck"]**

我们实例化了两个Child1，在实例s1中为父类的colors属性push了一个颜色，但是s2也被跟着改变了。造成这种现象的原因就是原型链上中的原型对象它俩是共用的。

这不是我们想要的，s1和s2这个两个对象应该是隔离的，这是这种继承方式的缺点。

**3.组合式继承**

这里所谓的组合是指组合借用构造函数和原型链继承两种方式。

function Parent2(){

this.name = "parent2";

this.colors = ["red","blue","yellow"];

}

function Child2(){

**Parent2.call(this);**

this.type = "child2";

}

**Child2.prototype = new Parent2()**

注意第6，9行，这种方式结合了借用构造函数继承和原型链继承的有点，能否解决上述两个实例对象没有被隔离的问题呢？

var s1 = new Child2();

s1.colors.push("black");

var s2 = new Child2();

s1.colors; // (4) ["red", "blue", "yellow", "balck"]

s2.colors; // (3) ["red", "blue", "yellow"]

可以看到，s2和s1两个实例对象已经被隔离了。

但这种方式仍有缺点。父类的构造函数被执行了两次，第一次是Child2.prototype = new Parent2()，第二次是在实例化的时候，这是没有必要的。

**3.1组合式继承优化1**

直接把父类的原型对象赋给子类的原型对象

function Parent3(){

this.name = "parent3";

this.colors = ["red","blue","yellow"];

}

Parent3.prototype.sex = "男";

Parent3.prototype.say = function(){console.log("Oh, My God！")}

function Child3(){

**Parent3.call(this);**

this.type = "child3";

}

**Child3.prototype = Parent3.prototype;**

var s1 = new Child3();

var s2 = new Child3();

console.log(s1, s2);

但是，我们来看如下代码：

console.log(s1 instanceof Child3); // true

console.log(s1 instanceof Parent3); // true

可以看到，我们无法区分实例对象s1到底是由Child3直接实例化的还是Parent3直接实例化的。用instanceof关键字来判断是否是某个对象的实例就基本无效了。

我们还可以用.constructor来观察对象是不是某个类的实例：

console.log(s1.constructor.name); // Parent3

从这里可以看到，s1的构造函数居然是父类，而不是子类Child3，这显然不是我们想要的。

**3.2组合式继承优化2**

这是继承的最完美方式

function Parent4(){

this.name = "parent4";

this.colors = ["red","blue","yellow"];

}

Parent4.prototype.sex = "男";

Parent4.prototype.say = function(){console.log("Oh, My God！")}

function Child4(){

**Parent4.call(this);**

this.type = "child4";

}

**Child4.prototype = Object.create(Parent4.prototype)；**

**Child4.prototype.constructor = Child4;**

Object.create是一种创建对象的方式，它会创建一个中间对象

var p = {name: "p"}

var obj = Object.create(p)

// Object.create({ name: "p" })

通过这种方式创建对象，新创建的对象obj的原型就是p，同时obj也拥有了属性name，这个新创建的中间对象的原型对象就是它的参数。

这种方式解决了上面的所有问题，是继承的最完美实现方式。

**4.ES6中继承**

Class 可以通过**extends**关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。

class Parent {

}

class Child1 extends Parent {

constructor(x, y, colors) {

super(x, y); // 调用父类的constructor(x, y)

this.colors = colors;

}

toString() {

return this.colors + ' ' + super.toString(); // 调用父类的toString()

}

}

上面代码中，constructor方法和toString方法之中，都出现了super关键字，它在这里表示父类的构造函数，用来新建父类的this对象。

子类必须在constructor方法中调用super方法，否则新建实例时会报错。如果子类没有定义constructor方法，这个方法会被默认添加，不管有没有显式定义，任何一个子类都有constructor方法。

ES5 的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（Parent.apply(this)）。ES6 的继承机制完全不同，实质是先创造父类的实例对象this（所以必须先调用super方法），然后再用子类的构造函数修改this。

1. **多态（多种调用方式）**

摘自设计模式：就是同一个方法多种调用方式。比如在js中对传入的参数做判断以实现多种调用方式，如定义一个add方法

function add(){

var arg = arguments,

len = arg.length;

switch(len){

case 0 : return 10;

case 1 : return + arg[0];

case 2 :return arg[0] + arg[1];

}

}

console.log(add()); //10

console.log(add(5)); //15

console.log(add(6,7)); //13

转化为类的形式：

function Add(){

function zero(){

return 10

}

function one(num){

return 10 + num;

}

function two(num1,num2){

return num1 + num2;

}

function add(){

var arg = arguments,

len = arg.length;

switch(len){

case 0 : return zero();

case 1 : return one(arg[0]);

case 2 : return two(arg[0],arg[1]);

}

}

var A = new Add();

console.log(A.add());

console.log(A.add(5));

console.log(A.add(6,7));

###### 从输入URL到页面加载发生了什么（包含了浏览器渲染页面的流程）

总体来说该过程分为以下几步:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

**1.DNS解析**

**DNS指将字符串域名映射为对应的IP地址。**请求方发起请求的前提是知道自己的请求该发向哪，而在地址栏里输入的网址其实就是对方的地址，比如<www.baidu.com>、<www.taobao.com>等，这种写法很方便用户记忆，而无需输入一长串无记忆点的IP地址，但在网络中机器之间只能互相识别IP地址，所以要将域名映射为IP地址。

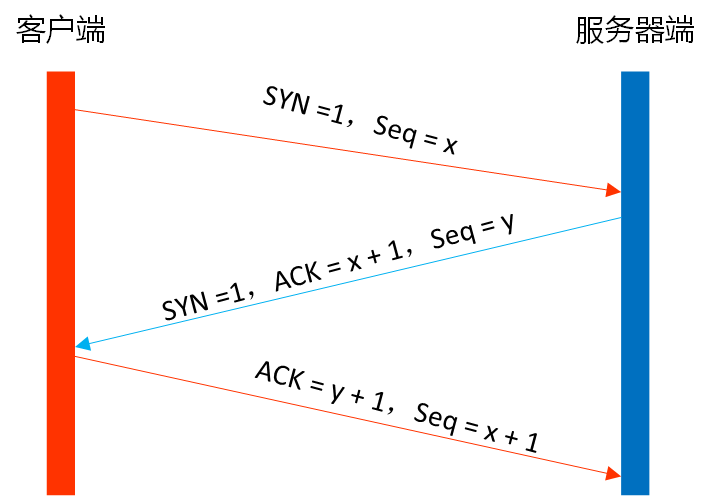
DNS是应用层协议，域名解析运行在UDP协议之上。**为什么选择UDP而非TCP？**原因如下：UDP无需连接，时效性更好，进行一次查询只需要两个DNS包。而TCP需要先用3个包建立连接，再用2个DNS包进行查询，最后用4个包断开连接，连接成本远大于查询本身，容易让DNS服务器不堪重负。

**DNS查询**

* 先检查本地hosts文件是否有这个网址映射关系，如果有就调用这个IP地址映射，完成域名解析。
* 否则，查找本地DNS解析器缓存，如果查找到则返回。
* 否则，查找本地DNS服务器，如果查找到则返回。
* 否则，按根域名服务器 -->顶级域名服务器-->次级域名服务器 -->主机名的IP地址的顺序依次查找下去，直到查找到IP地址。以输入<www.baidu.com>为例，默认所有网址尾部是有个.的，这里即<www.baidu.com>.，这个.对应的就是根域名服务器，所以网址真正的解析过程为:. -> .com-> google.com.->www.baidu.com.。

**2.TCP连接**

HTTP协议是使用TCP作为其传输层协议的,在HTTP请求发送前还要通过**三次握手**建立连接。



1. 第一次握手：建立连接。客户端发送连接请求报文段，将SYN位置为1，Sequence Number为x；然后，客户端进入SYN\_SEND状态，等待服务器的确认；
2. 第二次握手：服务器收到SYN报文段。服务器收到客户端的SYN报文段，需要对这个SYN报文段进行确认，设置Acknowledgment Number为x+1(Sequence Number+1)；同时，自己自己还要发送SYN请求信息，将SYN位置为1，Sequence Number为y；服务器端将上述所有信息放到一个报文段（即SYN+ACK报文段）中，一并发送给客户端，此时服务器进入SYN\_RECV状态；
3. 第三次握手：客户端收到服务器的SYN+ACK报文段。然后将Acknowledgment Number设置为y+1，向服务器发送ACK报文段，这个报文段发送完毕以后，客户端和服务器端都进入ESTABLISHED状态，完成TCP三次握手。

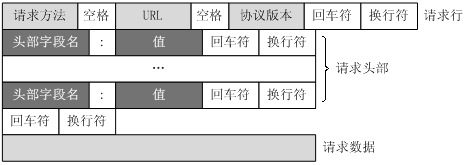
**问：为什么需要进行三次握手，而不是两次握手？**

原因是两次握手不可靠，防止已经失效的连接请求报文段突然又传送到了服务器端而产生错误。设想只有两次握手的场景，假如客户端某次发出的请求由于网络滞留原因最终到达了服务器端，但此时该请求已失效，而服务器端却会认为这是客户端发起的一次新请求，就会向客户端发出应答报文。而客户端认为自己没有发出请求，就不会对此做出响应，此时另一端的服务器通过两次握手以为建立了连接，一直等待客户端发来数据，白白浪费了资源。而三次握手的情况下，服务器端只有明确收到了客户端的应答报文，才可以确定客户端是想要建立连接的。

**3.发送HTTP请求**

其实这部分又可以称为前端工程师眼中的HTTP，它主要发生在客户端。发送HTTP请求的过程就是构建HTTP请求报文并通过TCP协议发送到服务器指定端口**(HTTP协议默认端口80, HTTPS协议默认端口443)**。

更相关的内容在[你想知道的AJAX都在这里](http://www.wuxuann-n.cn/2017/08/20/%E4%BD%A0%E6%83%B3%E7%9F%A5%E9%81%93%E7%9A%84AJAX%E9%83%BD%E5%9C%A8%E8%BF%99%E9%87%8C/)一文中已有介绍，可参考。这里就放出HTTP请求报文的直观图，并补充上文没讲的一点内容。



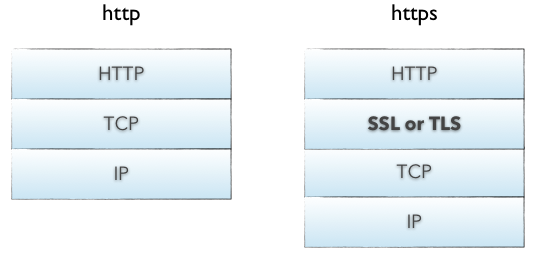
**补充：http请求方法**

HTTP/1.1 定义的请求方法有8种：GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS、TRACE。

* OPTIONS：返回服务器针对特定资源所支持的HTTP请求方法。也可以利用向Web服务器发送'\*'的请求来测试服务器的功能性。
* HEAD：向服务器索要与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。
* GET：向特定的资源发出请求。
* POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的创建和/或已有资源的修改。
* PUT：向指定资源位置上传其最新内容。
* DELETE：请求服务器删除Request-URI所标识的资源。
* TRACE：回显服务器收到的请求，主要用于测试或诊断。
* CONNECT：HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

**补充：HTTPS**

现在https用的也是越来越多，所以这边穿插一点https的知识！



HTTP报文是包裹在TCP报文中发送的，服务器端收到TCP报文时会解包提取出HTTP报文，网络请求需要很多路由器的转发，HTTP报文是明文的，所以容易在中间节点处被篡改信息。那么在HTTP报文进入TCP报文之前，先使用SSL/TLS对HTTP报文进行加密就可以解决这个问题了。从网络的层级结构看SSL/TLS位于HTTP协议与TCP协议之间。

**为什么HTTPS安全**

HTTPS更安全，是因为它利用 了SSL/TLS协议传输，SSL/TLS协议的基本思路是采用[公钥加密法](http://en.wikipedia.org/wiki/Public-key_cryptography)，也就是说，客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。虽然HTTPS相比于HTTP更安全，但是势必会带来一些时间上的损耗，如握手和加密等过程，是否使用HTTPS需要根据具体情况在安全和性能方面做出权衡。具体过程请参考经典的阮一峰先生的博客[SSL/TLS协议运行机制的概述](http://www.ruanyifeng.com/blog/2014/09/illustration-ssl.html)。

**4.服务器处理请求并返回HTTP报文**

这里只放出HTTP响应报文的图：



服务器返回给浏览器的文本信息，通常HTML, CSS, JS, 图片等文件就放在响应正文这一部分。

**5.浏览器解析渲染页面**

接下来的工作就是浏览器接收HTML,CSS,JS文件后，如何将页面呈现在我们眼前？这就是浏览器解析渲染页面的过程了！

**总体流程简单来说：**

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

**整个过程解释：**

浏览器接收到html文档，会被浏览器的HTML PARSER(HTML 解析器)解析， 通过词法分析将tag分析为相应的token，对HTML文档从上往下依次去解析token，这个过程的特点一是从上往下的，二是在此法分析的过程可以解析出link、script这样的标签，这些标签里面对应的外部资源会进一步由浏览器向网络发起请求（请求CSS和js等资源 ）。css相关的资源请求回来后会由浏览器生成相应的CSS树。将dom树和css树进行合并生成render tree渲染树，之后再进行布局layout和绘制paint，最后成功渲染出页面。

**问1：为什么要把css放在head中？**

将css样式放在head中引入，会先加载css，之后浏览器就知道这个css规则了，然后他在渲染DOM节点的时候就按照已知的css规则来渲染。

反例：如果把css内容放在DOM节点之后引入，会首先按照默认样式渲染dom内容（比如字体16px），之后加载完css发现样式的宽啊高啊都得改变，又得重新去渲染一遍，这时可能会出现屏幕的跳变，这个变化可能快可能慢，和你的电脑配、网络情况、页面规模等有关，用户体验差，所以得出结论，css要放在head中，而不要放在body中。

**问2：为什么要把js放在body最下面？**

script引入的js会阻塞页面渲染，因为script的js代码有权改变dom结构和dom内容。

遇到script会先加载和执行相应的js，该过程完毕后才会继续加载解析后续的页面。把js放在body最下面，一方面这样就不会阻塞页面的渲染了，能让页面更快地出来；另一方面放在下面，加载到它时，它已经能拿到所有标签，而放到中间，就有可能拿不到它后面的标签了，对后面标签所做的操作就失效了。

**结束连接**



**1.第一次挥手：** 主动方（可以使客户端，也可以是服务器端），设置Sequence Number，向被动方发送一个FIN报文段，告诉另一方我打算断开连接，后面不会再发送数据了，但是此时主动关闭方还是可以接收数据的。

**2.第二次挥手：**被动方收到了主动方发送的FIN报文段，向主动方回一个ACK报文段（Acknowledgment Number为Sequence Number + 1），告诉主动方，我“同意”你的关闭请求；

**3.第三次挥手：**被动方收到断开连接信息时，可能还有数据没有传完，所以等待数据全部传输结束后，再发送一条 FIN 为 1 的信息，告诉对方也做了断开连接的准备，但没有断开。

**4.第四次挥手：** 主动方收到被动方发送的FIN报文段，向被动方发送ACK报文段，被动方收到主动方的ACK报文段以后，就关闭连接；此时，主动方等待2MSL后依然没有收到回复，则证明Server端已正常关闭，那好，主动方也可以关闭连接了。

**问：为什么需要进行四次挥手？**

第一次挥手是在主动方没有数据需要再发送过去的情况下发起的，第二挥手是被动方发送ACK表示同意，当然，如果在第二次挥手时被动方也向主动方发送FIN包请求断开连接似乎也行，但考虑到它自己有可能还有数据没有发送完，所以当它真的没有数据可发了，才会进行第三次挥手发送FIN包告诉主动方，我也不再发送数据了，之后主动发再送ACK以表确认，即第四次挥手。

简而言之，一端断开连接需要两次挥手（请求和回应），两端断开连接就需要四次挥手了。

###### 设计模式（单例模式、MVC、MVVM）

###### Promise

###### 浏览器内核的理解

浏览器内核又可以分成两部分：**渲染引擎**(layout engineer 或者 Rendering Engine)和 **JS 引擎**。

**渲染引擎**负责取得网页的内容（HTML、XML、图像等等）、整理讯息（例如加入 CSS 等），以及计算网页的显示方式，然后会输出至显示器或打印机。浏览器的内核的不同对于网页的语法解释会有不同，所以渲染的效果也不相同。所有网页浏览器、电子邮件客户端以及其它需要编辑、显示网络内容的应用程序都需要内核。

**JS 引擎**则是解析和执行 javascript 语言来实现网页的动态效果。

最开始渲染引擎和 JS 引擎并没有区分的很明确，后来 JS 引擎越来越独立，**内核就倾向于只指渲染引擎**。

常见的浏览器内核可以分这四种：

Trident内核：IE,MaxThon,TT,The World,360,搜狗浏览器等。[又称MSHTML]

Gecko内核：Netscape6及以上版本，FF,MozillaSuite/SeaMonkey等

Presto内核：Opera7及以上。 [Opera内核原为：Presto，现为：Blink;]

Webkit内核：Safari,Chrome等。 [ Chrome的：Blink（WebKit的分支）]

###### 三列布局（圣杯布局、双飞翼布局、使用绝对定位、分别左浮动和右浮动）

1. **圣杯布局**

<style>

body{

min-width: 730px; /\* 2\*left + right \*/

}

.container{

**padding-left: 210px;**

**padding-right: 310px;**

}

.column{

float: left;

height: 300px;

}

.middle{

width: 100%;

background-color: rgba(255, 0, 0, .5);

}

.left{

**position: relative;**

**left: -210px;**

width: 200px;

**margin-left: -100%;**

background-color: rgba(0, 255, 0, .5);

}

.right{

**position: relative;**

**right: -310px;**

width: 300px;

**margin-left: -300px;**

background-color: rgba(0, 0, 255, .5);

}

</style>

<body>

<div class="container">

<div class="column middle"></div>

<div class="column left"></div>

<div class="column right"></div>

</div>

**2.双飞翼布局（比圣杯好理解些）**

<style>

.middle-wrapper {

float: left;

width: 100%;

}

.middle {

height: 300px;

**margin-left: 210px;**

**margin-right: 310px;**

background-color: rgba(255, 0, 0, .5);

}

.left {

float: left;

width: 200px;

height: 300px;

**margin-left: -100%;**

background-color: rgba(0, 255, 0, .5);

}

.right {

float: left;

width: 300px;

height: 300px;

**margin-left: -300px;**

background-color: rgba(0, 0, 255, .5);

}

</style>

<body>

<div class="middle-wrapper">

<div class="middle"></div>

</div>

<div class="left"></div>

<div class="right"></div>

</body>

**圣杯布局、双飞翼布局异同：**

**相同之处：**都是让三列浮动，然后通过负外边距形成三列布局，都是把主列放在文档流最前面，使主列优先加载。

**区别：**在于如何处理中间主列的位置，圣杯布局是利用父容器的左、右内边距定位；双飞翼布局是把主列嵌套在div后利用主列的左、右外边距定位。

1. **使用绝对定位实现三列布局（用的多）：中间列通过左右外边距定位，左右列用absolute定位。**

<style>

.container{

**position: relative;**

}

.middle{

margin-left: 210px;

margin-right: 310px;

height: 300px;

background-color: red;

}

.left{

position: absolute;

top: 0;

left: 0;

width: 200px;

height: 300px;

background-color: blue;

}

.right{

position: absolute;

top: 0;

right: 0;

width:300px;

height: 300px;

background-color: green;

}

</style>

<body>

<div class="container">

<div class="middle"></div>

<div class="left"></div>

<div class="right"></div>

</div>

</body>

1. **使用左浮动和右浮动：先左列左浮动，右列右浮动，然后中间列通过设置左右外边距，注意先后顺序。**

<style>

.container{

position: relative;

}

.left{

**float: left;**

width: 200px;

height: 300px;

background-color: blue;

}

.right{

**float: right;**

width:300px;

height: 300px;

background-color: green;

}

.middle{

**margin-left: 210px;**

**margin-right: 310px;**

height: 300px;

background-color: red;

}

</style>

<body>

<div class="container">

**<!-- middle写在最后 -->**

<div class="left"></div>

<div class="right"></div>

<div class="middle"></div>

</div>

</body>

###### 事件循环机制（event loop）——一些结论，涉及promise、setTimeout

1. JavaScript的一大特点就是单线程，而这个线程中拥有唯一的一个事件循环。
2. JavaScript代码的执行过程中，除了依靠**函数调用栈**来搞定函数的执行顺序外，还依靠**任务队列(task queue**)来搞定另外一些代码的执行。

3. 一个线程中，事件循环是唯一的，但是任务队列可以拥有多个。

4. 任务队列又分为**macro-task（宏任务）与micro-task（微任务）**，在最新标准中，它们被分别称为task与jobs。

5. **macro-task**大概包括：script(整体代码), setTimeout, setInterval, setImmediate, I/O, UI rendering。

6. **micro-task**大概包括: process.nextTick, Promise, Object.observe(已废弃), MutationObserver(html5新特性)

1. setTimeout/Promise等我们称之为任务源。而进入任务队列的是他们指定的具体执行任务。

来自不同任务源的任务会进入到不同的任务队列。其中setTimeout与setInterval是同源的。

8. **事件循环的顺序**，决定了JavaScript代码的执行顺序。它从script(整体代码)开始第一次循环。之后全局上下文进入函数调用栈。直到调用栈清空(只剩全局)，然后执行所有的micro-task。当所有可执行的micro-task执行完毕之后。循环再次从macro-task开始，找到其中一个任务队列执行完毕，然后再执行所有的micro-task，这样一直循环下去。

9. 其中每一个任务的执行，无论是macro-task还是micro-task，都是借助函数调用栈来完成。