

前端复习大纲

ECMAScript

基本类型

引用类型

1 数组类型

2 对象类型

3 函数类型

执行环境

闭包

this

原型, 原型链, 继承

DOM

节点

DOM事件

BOM

window对象

window.screen

window.navigator

window.location

window.history

ES6

let/const

解构赋值

字符串扩展

数值的扩展

函数的扩展

数组的扩展

对象的扩展

Symbol

Set

Map

Promise

Generator/Async

Class

修饰器Decorator

二进制数组

HTML

meta标签

H5

CSS

元素与盒

布局

移动端适配

BFC

CSS3

伪类和伪元素

HTTP

OSI七层协议

HTTP状态码

HTTP首部字段

HTTPS

web安全

跨域

http缓存

tcp三次握手四次挥手

浏览器加载/解析/渲染

原生ajax

link和@import

HTTP 1.0->1.1->2.0

前端优化

webpack

基本使用

构建速度优化

Node.js

npm

koa2

前端工程化

模块化

组件化

规范化

自动化

算法

git

bash

ECMAScript

基本类型

(1) **Undefined**: Undefined类型只有一个值，即特殊的undefined。在使用var声明变量但未对其加以初始化时，这个变量的值就是undefined。

(2) Null: Null类型是第二个只有一个值的数据类型，这个特殊的值是null。从逻辑角度来看，null值表示一个空对象指针

(3) Boolean: 该类型只有两个字面值：true和false，!!一般用来将后面的表达式强制转换为布尔类型的数据（boolean），也就是只能是true或者false；

(4) Number: 这种类型用来表示整数和浮点数值，还有一种特殊的数值，即NaN（非数值 Not a Number）。这个数值用于表示一个本来要返回数值的操作数未返回数值的情况（这样就不会抛出错误了）。

NaN本身有两个非同寻常的特点。首先，任何涉及NaN的操作（例如NaN/10）都会返回NaN，这个特点在多步计算中有可能导致问题。其次，NaN与任何值都不相等，包括NaN本身。

(5) String: String类型用于表示由零或多个16位Unicode字符组成的字符序列，即字符串。字符串可以由单引号(')或双引号(")表示。

引用类型

1 数组类型

定义：有序的数据集合，有属性length,代表数据长度，length可以设置,若小于原值，则保留设置值的长度，若大于则在数组中增加相应的undefined值

检测是否为数组：

1 [] instanceof Array instanceof检测该构造函数是否在对象的原型链上，

[].__proto__.constructor = Array

2 Array.isArray([])

3 [].constructor==Array

4 Object.prototype.toString.call([])=='[object Array]'

常用方法：

栈/队列方法：

Array.push() 在数组后面添加 返回新的数组长度

Array.pop() 尾部删除 返回移除的项

Array.shift() 头部删除 返回移出的第一项

Array.unshift() 头部添加 返回新的数组长度

位置方法：

Array.indexOf(value,起始位置) 没找到的情况返回-1，不会隐式转换

Array.lastIndexOf(value,起始位置)

转换方法：

Array.toString() 转换为字符串,以逗号隔开每一项

Array.join(连接符) 转换为字符串,默认连接符为逗号

操作方法：

Array1.concat(Array2, Array3,.....) 数组连接 返回新数组

Array.slice(起点, 终点) 起点：若为负数，从尾部开始算起，返回新数组，包含起点不包含终点

Array.splice(起点, 长度) 删除一段 改变原数组, 返回删除的项

Array.splice(起点, 0, 元素)插入一段, 没有返回值

Array.splice(起点, n, n个元素)替换,返回被替换的元素

排序方法:

```
Array.sort(function(n1,n2){    升序排列数字  
return n1-n2;  
})
```

Array.reverse() 倒置 改变原数组, 返回原数组的引用

迭代方法:

Array.every(fuction(value,index,arr){ },thisValue) 每一项都返回true的话则返回true, 否则终止运行返回false

Array.filter(fuction(value,index,arr){ },thisValue) 返回该函数方法会返回true的项组成的数组

Array.forEach(fuction(value,index,arr){ },thisValue) 数组每一项运行给定函数, 无返回值

Array.map(fuction(value,index,arr){ },thisValue) 数组每一项运行给定函数, 返回每次函数调用的结果组成的数组,

Array.some(fuction(value,index,arr){ },thisValue) 只要函数对任意一项返回true的话则返回true

归并方法:

Array.reduce(function(preV,cur,index,array){}) prev是函数上一次执行的返回值, cur是当前数组项的值, 从第二项开始循环

Array.reduceRight(function(preV,cur,index,array){})

习题: 数组去重 (index,hash,set) , 多纬数组转一维

2 对象类型

定义: 无序属性的集合, 其属性可以包含基本值, 对象或者函数

2.1 对象的属性:

(1) 数据属性: 可以在创建时直接定义

[[Configurable]]: 表示能否删除, 能否修改属性的特性, 能否修改为访问器属性, 默认为 true, 一旦修改为 false, 就不能再改回来, 除了 Writeable 之外的操作都会报错

[[Enumerable]]: 表示是否能通过 for-in、Object.keys 循环, 默认为 true

[[Writable]]: 表示是否能修改属性的值, 默认为 true, 若为 false, 赋值不会报错, 但不生效

[[Value]]: 包含这个属性的值

注: 一旦使用 Object.defineProperty 给对象添加属性, 那么如果不设置属性的特性, 那么 configurable、enumerable、writable 这些值都为默认的 false

(2) 访问器属性: 不能直接定义, 必须使用 Object.defineProperty(obj, 属性名, 描述符对象) 定义

[[Configurable]]: 同上

[[Enumerable]]: 同上

[[Get]] 读取属性时调用的函数, 默认 undefined

[[Set]] 写入属性时调用的函数, 默认 undefined

定义多个属性: **Object.defineProperties(obj, {**

```
property1: {},  
Property2: {},  
})
```

读取属性特性：Object.getPropertyDescriptor(obj, 属性名)

如果一个描述符不具有value,writable,get 和 set 任意一个关键字，那么它将被认为是一个数据描述符。如果一个描述符同时有(value或writable)和(get或set)关键字，将会产生一个异常。

2.2 创建对象的几种方式：

1 var obj=new Object(){} 构造函数和字面量

2 工厂模式：在一个函数内把对象创建好，然后把对象返回

优点：解决了创建多个相似对象的问题

缺点：不知道对象类型

3 构造函数模式：像Object和Array这样的原生构造函数，在运行时会自动存在于执行环境。此外，也可以创建自定义的构造函数，创建对象是使用new操作符，new做了什么：

1 创建对象

2 将对象的__proto__指向构造函数的原型对象->继承方法

3 构造函数.call(刚创建的对象)->继承属性

4 return 对象

优点：可以通过instanceof 判断对象的类型

缺点：构造函数内部创造的方法在每个实例中都是不同的，内存消耗大

4 原型模式：在原型对象上定义的属性和方法被所有实例共享

优点：原型中定义的方法在所有实例中相同，减少了内存的消耗

缺点：引用类型数据也共享，一个实例中的属性修改，所有实例都会受影响

5 构造函数模式+原型：

属性用构造函数模式定义，方法定义在原型上，实现了每个实例拥有自己的属性并共享同一个方法

6 Object.create (原型对象，同[Object.defineProperties\(\)](#)的第二个参数)

7 Object.assign() 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。若属性名相同则覆盖目标对象的属性

```
1 const target = { a: 1, b: 2 };  
2 const source = { b: 4, c: 5 };  
3 const returnedTarget = Object.assign(target, source); // 1,4,5
```

2.3 遍历对象的几种方式：

(1) **for in** 返回实例中(包括原型链上)所有可枚举属性,效率最低，因为要遍历访问原型链上的属性，并且还要按照 数字优先、其他的按照创建时间 的顺序遍历

(2) **Object.keys(obj)** 返回实例中所有可枚举属性的字符串数组

(3) **Object.getOwnPropertyNames(obj)** 返回实例中所有的属性包括不可枚举属性但不包括symbol 的字符串数组

(4) **Object.getOwnPropertySymbols(obj)** 返回实例中所有的symbol属性的数组

(5) Reflect.ownKeys(obj) 返回实例中所有属性包括不可枚举以及symbol

以上的 5 种方法遍历对象的键名，都遵守同样的属性遍历的次序规则。

- 首先遍历所有数值键，按照数值升序排列。
- 其次遍历所有字符串键，按照加入时间升序排列。
- 最后遍历所有 Symbol 键，按照加入时间升序排列。

2.4 包装对象（基本包装类型的实例）

为了便于操作基本类型值，ECMAScript还提供了三个特殊的引用类型：Boolean、Number和String,标准库提供了构造函数来封装布尔值、数字和字符串作为对象。这些类型与其他引用类型相似，每当读取一个基本类型值时，后台就会创建一个对应的基本包装类型的对象，从而让我们能够调用一些方法来操作这些数据。

例如操作字符串的substring方法时，后台会自动完成下列处理：

- (1) 创建String类型的一个实例。
- (2) 在实例上调用substring方法。
- (3) 销毁这个实例。

与普通的对象最大的区别：生命周期，包装对象的生存期只存在于该行代码执行的一瞬间，执行完毕后立马销毁，普通对象在执行流离开当前的执行环境之前一直存在
不建议显式的创建基本包装类型的实例，会让人分不清处理的是基本类型还是引用类型的数据。

Number类型

toFixed(整数): 按照置顶的小数位返回数值的字符串表示，如果数值本身的小数位比指定的多，则会四舍五入

String类型

属性：length:字符串长度

方法：**charAt(位置)**:返回该位置的值

charCodeAt(位置):返回该位置的字符编码

concat() 拼接，类似+，返回新字符串

slice(起始位置, 结束位置) 包含起始位置不包含结束位置，返回新字符串

substr(起始位置, 截取长度) 返回新字符串

substring(起始位置, 结束位置) 返回新字符串

indexOf(值, 起始位置) 返回从起始位置开始搜索得到首次出现该值的位置

lastIndexOf(值, 起始位置) 从后往前 ~返回从起始位置开始搜索得到首次出现该值的位置

trim() 返回一个删除前后空格之后的字符串副本

toLowerCase() 转换为小写

toUpperCase() 转换为大写

match(正则表达式或正则对象)

返回一个数组，其中存放了与它找到的匹配文本有关的信息。该数组的第 0 个元素存放的是匹配文本，其余元素存放的是与正则表达式的子表达式匹配的文本。除了这些常规的数组元素之外，返回的数组还含有两个对象属性。**index** 是匹配文本的起始字符在 stringObject 中的位置，**input** 属性声明的是对 stringObject 的引用。

如果 `regexp` 具有标志 `g`, 则 `match()` 方法将执行全局检索, 找到 `stringObject` 中的所有匹配子字符串。若没有找到任何匹配的子串, 则返回 `null`。如果找到了一个或多个匹配子串, 则返回一个数组。不过全局匹配返回的数组的内容与前者大不相同, 它的数组元素中存放的是 `stringObject` 中所有的匹配子串, 而且也没有 `index` 属性或 `input` 属性。

`search(正则表达式或正则对象)` 返回从字符串中第一个匹配项的索引

`replace(要被替换的值 (正则或者字符串), 替换的值 (字符串或者函数))` 如果第一个参数是字符串, 则只能替换第一个子字符串

若替换的值是函数则此函数的参数为

变量名	代表的值
<code>match</code>	匹配的子串。 (对应于上述的\$&。)
<code>p1, p2, ...</code>	假如 <code>replace()</code> 方法的第一个参数是串。 (对应于上述的\$1, \$2等。) <code>p1</code> 就是匹配的 <code>\a+</code> , <code>p2</code> 就是匹配 <code>\b</code> 的子串。
<code>offset</code>	匹配到的子字符串在原字符串中的位置。
<code>string</code>	被匹配的原字符串。
<code>NamedCaptureGroup</code>	命名捕获组匹配的对象

`split(分割依据 (字符串或正则表达式), 数组最大长度)`

静态方法 `String.fromCharCode(code1, code2...)` 根据字符编码返回字符串

2.5 单体内置对象

定义: 由ECMAscript实现提供的, 不依赖于宿主环境的对象

Global:所有在全局作用域中定义的属性和函数, 都是Global对象的属性。

<code>decodeURI()</code>	解码某个编码的 URI。
<code>encodeURI()</code>	把字符串编码为 URI。不会对本身属于uri的特殊字符进行编码, 如冒号、正斜杠、问号和井号
<code>eval()</code>	计算 JavaScript 字符串, 并把它作为脚本代码来执行。
<code>isFinite()</code>	检查某个值是否为有穷大的数。
<code>isNaN()</code>	检查某个值是否是数字。
<code>Number()</code>	把对象的值转换为数字。
<code>parseFloat()</code>	解析一个字符串并返回一个浮点数。 (只返回第一个数字, 允许空格, 开头不是数字返回NaN)
<code>parseInt()</code>	解析一个字符串并返回一个整数。
<code>String()</code>	把对象的值转换为字符串。

与window对象的关系：web浏览器把Global对象作为window对象的一部分加以实现

Math

min(number1,number2...) 取最小值

max(number1,number2...) 取最大值

ceil() 向上取整

floor() 向下取整

round() 四舍五入

random() 生成0-1随机数

abs() 绝对值

cos(),sin(),tan()

数组求最大值：

es5: Math.max.apply(null, [1,2,3])

es6 : Math.max(...[1,2,3])

2.6 Date

起始时间：UTC：1970年一月一日

var now=new Date(日期的毫秒数,如果传的是字符串,后台会先调用Date.parse()) 不传参的情况下,获得当前时间。

Date.parse(符合格式的字符串): 解析一个表示某个日期的字符串,并返回从1970-1-1 00:00:00 UTC 到该日期对象(该日期对象的UTC时间)的毫秒数,如果该字符串无法识别,或者一些情况下,包含了不合法的日期数值(如: 2015-02-31),则返回值为NaN。

Date.UTC(年, 月, 日, 时, 分, 秒, 毫秒) 作用同上,只是接收的参数格式不同,月份从0开始

Date.now() 方法返回自1970年1月1日 00:00:00 UTC到当前时间的毫秒数。

Date.prototype.getDate() 返回天数, 1-31

Date.prototype.getDay() 返回一周的第几天 0-6 0是星期天

Date.prototype.getFullYear() 返回年份

Date.prototype.getHours() 返回小时 0-23

Date.prototype.getMinutes() 返回分钟 0-59

Date.prototype.getMonth() 返回月份 0-11 0为一月

Date.prototype.getSeconds() 返回秒钟 0-59

Date.prototype.getTime() 返回时间毫秒

Object的几个方法：

1 **Object.freeze()** 方法可以冻结一个对象。一个被冻结的对象再也不能被修改;冻结了一个对象则不能向这个对象添加新的属性,不能删除已有属性,不能修改该对象已有属性的可枚举性、可配置性、可写性,以及不能修改已有属性的值。此外,冻结一个对象后该对象的原型也不能被修改。**freeze()** 返回和传入的参数相同的对象。

3 函数类型

内部属性：

1 **arguments**: 类数组对象，包含着传入函数的所有参数，这个对象有一个属性：

callee:指向拥有这个arguments对象的函数，例子：递归时使用arguments.callee代替使用函数名防止因为该函数名指向其他函数或者值时失效。

2 **caller**:保存着调用当前函数的函数的引用，如果是在全局中调用的当前函数，它的值为null,使用：函数名.caller

3 **this**: 引用的是函数执行的环境对象

属性和方法：

1 **length**: 表示函数希望接收的命名参数的个数

2 **apply(this值, 参数数组)**:在特定的作用域中调用函数

取数组最大值 : Math.max.apply(null,arr)

3 **call(this值, 参数...)**:在特定的作用域中调用函数

4 **bind(this值)**: 创建一个函数实例，其this值会被绑定到传入的值

注：可以传递任意数量的参数，并且可以通过arguments对象来访问这些参数，未指定返回值时为undefined，函数传参是按值传递，基本类型的值会被复制，引用类型的参数传递的是指向的内存地址

构造函数继承：

1 子类原型 = 父类原型

```
1 function Super() {}
2 Super.prototype.a=1
3 function Sub() {}
4 Sub.prototype= Super.prototype
5 let s = new Sub()
6 console.log(s.a) // 1
```

缺点：1 只能继承父类原型上的属性和方法，不能继承父类实例上的属性和方法 2 原型上引用类型属性共享的问题

2 原型链继承：Sub.prototype = new Super();

```
1 function Super() {}
2 Super.prototype.c=[1]
3 function Sub() {}
4 Sub.prototype= new Super()
5 let s1 = new Sub()
6 let s2= new Sub()
7 s1.c.push(2)
8 console.log(s2.c) // [1,2]
```

优点：父类原型和实例上的属性都能继承

缺点：同上，还是有属性共享的问题，s1上操作了原型上的c属性，s2也受到了影响

2 借用构造函数继承：在子类函数内部调用 Super.call(this)，问题：只能集成实例上的属性和方法不能继承父类原型上的属性和方法

3 组合继承：

(1) 借用构造函数+原型链继承

```
1 function Super() {
```

```

2   this.c=[1]
3 }
4 Super.prototype.a=function () {}
5 function Sub() {
6   Super.call(this)
7 }
8 Sub.prototype= new Super()
9 Sub.prototype.constructor= Sub // 重新设置constructor
10 let s = new Sub()
11 let s2= new Sub()
12 s.c.push(2)
13 console.log(s2.c) // [1],不同实例上的属性是独立的

```

优点：解决了引用类型属性共享的问题，实例属性通过借用构造函数继承，原型通过原型链继承

问题：父类构造函数要执行两遍，并且此方法借用构造函数实现了实例属性的继承，但子类原型作为父类的实例也拥有父类的实例属性，是多余的，浪费了空间

(2) 借用构造函数+子类原型 = 父类原型

```

1 function Super() {
2   this.c=[1]
3 }
4 Super.prototype.a=function () {}
5 function Sub() {
6   Super.call(this)
7 }
8 Sub.prototype= Super.prototype
9 let s = new Sub()
10 let s2= new Sub()
11 s.c.push(2)
12 console.log(s2.c)

```

优点：父类构造函数只调用一次，没有浪费多余的空间

缺点：子类实例.constructor 指向父类且无法改变

(3) 借用构造函数+Object.create()

```

1 function Super() {
2   this.c=[1]
3 }
4 Super.prototype.a=function () {}
5 function Sub() {
6   Super.call(this)
7 }
8 Sub.prototype= Object.create(Super.prototype)
9 Sub.prototype.constructor= Sub // 重新设置constructor
10 let s = new Sub()
11 let s2= new Sub()
12 s.c.push(2)
13 console.log(s2.c)

```

优点：使用Object.create创建一个__proto__指向父类原型对象的对象作为子类的原型对象，这个对象作为桥梁向下连接子类实例，向上连接父类原型对象

Object.create大致原理：

```
1 function create(o,propertiesObject){  
2 //使用临时的构造函数省去了写构造函数的时间  
3 function F(){  
4 F.prototype=o;  
5 var f=new F();  
6 //添加额外的属性  
7 Object.defineProperties(f,propertiesObject);  
8 return f;  
9 }
```

常见面试题：

1对象/数组的深拷贝：

```
1 function deepClone(target) {  
2     let type = Object.prototype.toString.call(target);  
3     let result = type === '[object Array]' ? [] : type === '[object Object]' ? {}  
4     if (result !== target) {  
5         for (let key in target) {  
6             result[key] = deepClone(target[key]);  
7         }  
8     }  
9     return result;  
10 }
```

执行环境

执行环境分为全局环境/函数环境，全局环境对象由宿主决定，浏览器中为window对象，nodejs中为global对象。

代码开始执行时，默认进入全局环境，当代码的执行流进入到一个函数时，将创建函数执行环境并推入到执行栈的顶部，若在这个函数内执行时执行流又进入了另一个函数（内部函数）。同样会将这个函数的执行环境推入到执行栈的顶部，当函数执行完毕，该函数的执行环境从环境栈弹出，把控制权交还给执行栈中的下一个执行环境（外部执行环境）

执行环境包含

(1)this 代表环境对象

(2)变量对象(variable object) 执行环境中定义的所有变量和函数都保存在变量对象中，代码无法访问，解析器在处理数据的时候会在后台使用它

(3)[[scope]]指针 指向由一个变量对象组成的带头结点的单向链表，即作用域链

执行环境创建的两个阶段：

1 创建阶段：

创建变量对象（扫描内部代码查找函数声明，将函数名及引用存入VO（VO即变量对象），若VO中存在就覆盖；查找变量声明（var），将变量名存入，初始化undefined，若VO中存在就忽略）

设置[[scope]]的值，单向链表的头部为该执行环境的VO，往下是外部的VO，直至全局VO。

最后设置this的值为执行时的环境对象

2 激活/执行阶段：设置变量的值，然后解释/执行代码

作用域链：

当代码执行，访问一个变量时，会从作用域链的头部也就是当前执行环境的VO开始找起，若找不到，就顺着作用域链往下在外部环境VO中寻找，直至全局VO，保证了变量有序的访问机制

闭包

定义：闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包的常见方式就在一个函数内部创建另一个函数，若内部函数不销毁，那么外部函数的VO会因为一直被内部函数引用而无法销毁

形成原因：内部函数的作用域链被延长，因此可以访问外部函数的变量

作用：创建私有变量/属性，这些属性不能被外部随意修改，同时可以通过指定的函数接口来操作

```
1 function User(name) {  
2     var password = 123456;  
3     this.username=name;  
4     this.getP=function () {  
5         return password  
6     };  
7     this.setP=function (newPassword) {  
8         password=newPassword  
9     }  
10 }  
11 var zb=new User('zb');  
12 console.log(zb.getP()); //123456  
13 zb.setP(654321);  
14 console.log(zb.getP()); //654321
```

副作用：由于垃圾回收机制，使用闭包会加大内存消耗，若代码书写不当，会造成内存泄漏

this

定义：this是函数的一个内部对象，引用的是函数执行时的环境对象，当一个函数被调用时，拥有它的object会作为this传入，this即为拥有这个函数的对象。

理解：任何方法调用函数其实在本质上都是以 函数名.call(thisArg,arg1,arg2.....) 的方式执行，平时使用的fn(); 的调用方法只是语法糖，实际上调用时为

fn.call(undefined)，而浏览器会在传入参数不为一个对象时设定this的默认值为window对象

改变this的三种方式：

call,apply: 以指定的this对象执行函数，除了指定this值，其他的参数在apply方法中为数组

bind: 返回一个this为指定对象的新函数，这个函数不会再受call,apply影响

几种特殊情况的this:

(1) DOM事件处理程序: 指向触发事件的元素对象

(2) 箭头函数: 在箭头函数中this就是定义时所在的对象

(3) 作为构造函数: this被绑定在正在创建的新对象

原型, 原型链, 继承

原型: 每个函数创建时会生成一个内部属性**prototype**作为函数的原型, 它是一个对象, 所以被称为原型对象, prototype最初只包含一个**constructor**属性, 指向prototype对象所在的构造函数

对象的继承: 每个对象包含一个内部指针[[prototype]], 按理来说不能被访问, 但很多浏览器实现为**_proto_**, 指针指向构造函数的原型对象, 当js读取某个对象的属性时先从对象本身搜索, 若搜不到, 就根据指向的原型对象查找

相关方法:

Object.getPrototypeOf(obj) 返回传入对象的原型对象

obj.hasOwnProperty(属性名) 检测属性是否来自实例中

Object.keys(obj) 返回实例中(包括原型链上)所有可枚举属性的字符串数组

Object.getOwnPropertyNames(obj) 返回实例中(包括原型链上)所有的属性包括不可枚举属性的字符串数组

obj.isPrototypeOf(obj) 检测此对象是否为传入对象的原型对象

原型链: prototype 作为一个对象, 也有内部指针[[prototype]], 若这个该prototype 对象是另一个构造函数的实例, 则这个原型对象作为另一个构造函数的实例内部指针指向另一个构造函数的原型对象, 直到指向 Object构造函数的 prototype (Object.getPrototypeOf(Object.prototype) === null; // true, 根据定义, null 没有原型) 并作为这个**原型链**中的最后一个环节。

注:

1字面量法设置prototype会导致constructor指针变为Object

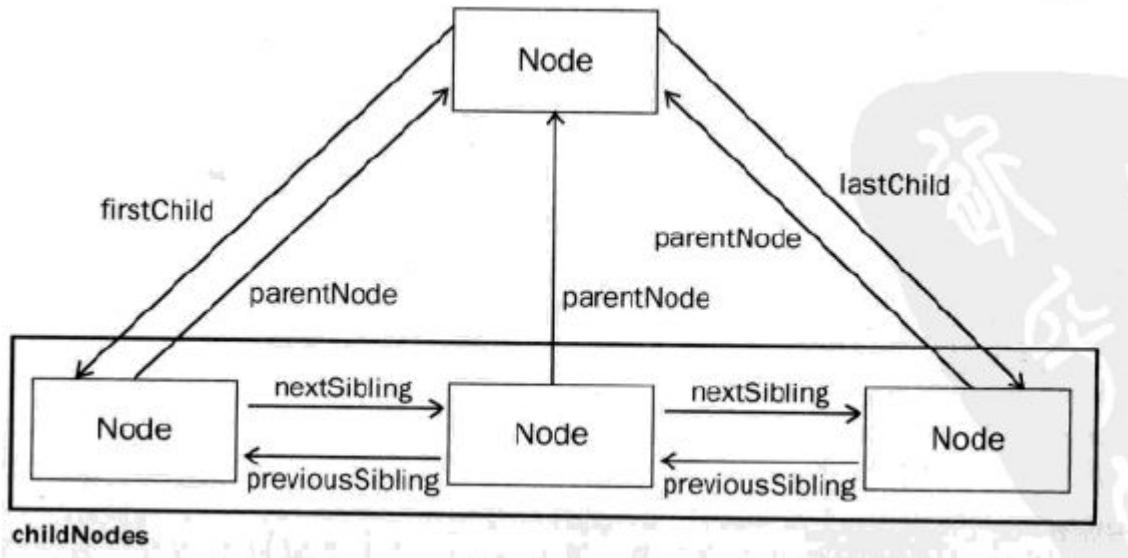
2实例与原型之间的连接是一个指针而不是一个副本, 如果在实例创建之后, 重写了整个原型对象, 那么实例无法访问重写后的原型对象中的属性和方法, (实例会永远指向旧的对象, 这个指针是在实例创建时就确定的, 而构造函数已经的prototype已经指向了新的对象)

DOM

节点

DOM 是针对HTML和XML文档的一个API, 它将任何HTML或XML文档描绘成一个由多节点构成的结构, 节点分为几种不同的类型, 每种类型分别表示文档中不同的信息或标记

文档节点是所有文档的根节点, 在html页面中, 文档节点只有一个子节点-<html>元素称之为文档元素, 每个文档只能有一个文档元素, xml中任何元素都可能成为文档元素。



childNodes 子节点集合

firstElementChild 第一个子元素节点

lastElementChild 最后一个子元素节点

hasChildNodes() 返回是否有子节点

xNode.appendChild(newNode) 末尾添加子节点

xNode.insertBefore(newNode, xNode.xChildNode) 插入到某个子节点的前面

xNode.insertBefore(newNode,null) 和appendChild 作用相同

xNode.insertBefore(newNode, xNode.firstChild) 插入到最前面

xNode.replaceChild(newNode,oldNode) 替换某个节点

xNode.removeChild(xChildNode) 删除某个节点

xNode.cloneNode(boolean) 参数为false为浅复制，没有父子节点

节点类型：

1 元素节点

提供了对元素标签名、子节点及特性的访问

元素和节点：

元素是节点的其中一种类型，nodeType 为 1 必须是含有完整信息的节点才是一个元素；一个节点不一定是一个元素，而一个元素一定是一个节点！ **div** 是一个元素，**div** 中的文本不是元素，但却是一个文本节点

nodeName 表示元素的标签名

属性：

id 在文档中的唯一标识符

lang 语言代码 很少使用 **dir** 语言的方向 很少使用 **title** 元素的附加信息

className 与元素的class特性对应，即为元素指定的CSS类

attributes 包含一个NamedNodeMap，与NodeList类似，也是一个‘动态’合集

classList 一个只读属性，返回一个元素的类属性的实时DOMTokenList 集合。使用 classList 是替代 **element.className** 作为空格分隔的字符串访问元素的类列表的一种方便的方法。

包含以下方法：

1 **add(String)** 添加指定的类值。如果这些类已经存在于元素的属性中，那么它们将被忽略。

2 **remove(String)** 删除指定的类值。

3 **item (Number)** 按集合中的索引返回类值。

4 **toggle (String)** 当只有一个参数时：切换 class value; 即如果类存在，则删除它并返回false，如果不存在，则添加它并返回true。

当存在第二个参数时：如果第二个参数的计算结果为true，则添加指定的类值，如果计算结果为false，则删除它

5 **contains(String)** 检查元素的类属性中是否存在指定的类值。

6 **replace(oldClass, newClass)** 用一个新类替换已有类。

方法：

getAttribute(特性名) 获取元素在标签上制定的特性值 (class,id,style)，若特性不存在，返回null,可以取到自定义特性，不分大小写

自定义属性只能使用getAttribute访问，而不能通过对象的属性访问。

两种特殊情况：1 style 返回css文本，而访问对象的属性返回是一个对象

2 事件，返回代码字符串，而访问对象的属性返回是一个函数

setAttribute(特性名, 要设置的值) 可以操作自定义特性

removeAttribute(特性名)

normalize() 将所有文本子节点合并为一个

getElementById(id名) ie8之后区分大小写

getElementsByTagName(元素名)

getElementsByClassName(一个或多个class名)

getElementsByTagName(name名)

querySelector(选择器字符串)

querySelectorAll(选择器字符串)

3 文本节点

9 文档节点

浏览器中的document对象是HTMLDocument的（继承自Document类型）一个实例

document对象属性：

关于子节点：

documentElement 始终指向HTML页面中的<html>元素

body 指向<body>元素

doctype 指向<!DOCTYPE>

文档信息**scrollWidth**

title <title>元素中的文本

URL 完整的url

domain 域名,可以设置为url中包含的域，一旦设置不能不能再修改

referrer 链接到当前页面的url

特殊集合

anchors 所有带name的 <a>元素 forms 所有form元素 images 所有元素 link 所有带href的 <a>元素

方法：

createElement(元素标签名)

write() 如果页面加载完之后再调用write方法会重写页面

关于浏览器各种取宽高偏移量的方法：

offsetHeight /offsetWidth: 元素本身+滚动条+内边距+边框 高度/宽度

offsetTop /offsetLeft: 元素的左/上 外边框与包含元素内边框的（包含元素的引用在offsetParent属性中）
边框的像素距离

HTMLElement.offsetParent 是一个只读属性，返回一个指向最近的（closest, 指包含层级上的最近）包含该元素的定位元素。如果没有定位的元素，则 offsetParent 为最近的 table, table cell 或根元素（标准模式下为 html; quirks 模式下为 body）。当元素的 style.display 设置为 "none" 时，offsetParent 返回 null。offsetParent 很有用，因为 [offsetTop](#) 和 [offsetLeft](#) 都是相对于其内边距边界的。

客户区大小

clientHeight/clientWidth: 元素本身+内边距的 高度/宽度

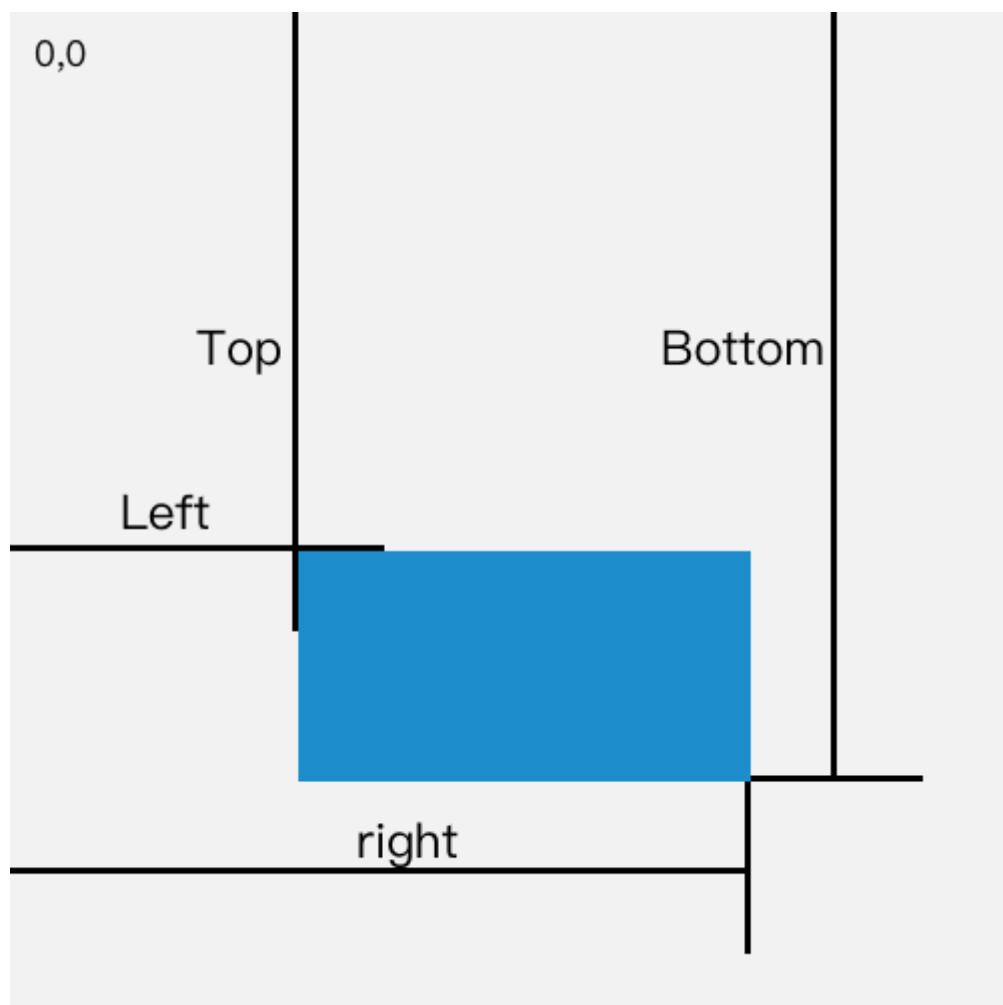
滚动大小

scrollHeight/scrollWidth 没有滚动条的情况下，元素内容(不包含边框)的总高度/宽度

scrollLeft/scrollTop 被隐藏在内容区域左侧/上方的像素数，可以设置这个属性来改变元素的滚动位置，

注： scroll相关的属性作用于生成滚动条的元素

getBoundingClientRect().left/top/bottom/right 相对于视窗的距离



DOM事件

1 事件流

事件冒泡：事件开始时由最具体的元素接收，然后逐级向上传播到较为不具体的节点

事件捕获：与冒泡相反，由document/window对象首先接收事件，最后到最具体的元素

DOM2级规定的事件流：三个阶段，捕获->处于目标阶段->冒泡

DOM2要求捕获阶段不涉及事件目标，将事件处理归为事件冒泡的一部分，但事实上浏览器在两个阶段都可以触发事件

事件是用户或者浏览器自身执行的某种动作，如click,load,mouseover是事件名，而响应事件的函数叫做事件处理程序，以on开头

2 事件处理程序

```
var btn=document.querySelector("#btn1");
```

DOM0级：

添加：

```
1 btn.onclick=function(e){  
2     console.log(this) // btn  
3 }
```

删除：

```
1 btn.onclick=null 删除点击事件
```

DOM2级：

添加：

```
1 btn.addEventListener("click", // 不需要on开头  
2 function(e){  
3     console.log(this) // btn  
4 },false) //第三个参数true表示捕获阶段调用事件处理程序，false为冒泡阶段调用
```

删除：

```
1 btn.removeEventListener('click',事件名, false) // 如果添加时使用匿名函数，则无法删除
```

优点：可以为一个元素添加多个同名事件

IE： attachEvent/detachEvent **ie8及以前** 替代addEventListener/removeEventListener

注：this是**window**,多个事情触发顺序与DOM2相反，事件名需要**on**

3 事件对象

DOM中

触发DOM上的某个事件时，会产生一个事件对象event,包含所有与事件有关的信息，包括导致事件的元素，事件类型以及其他与特定事件相关的信息，如鼠标事件产生的事件对象会包含鼠标位置的信息。

属性：

target 事件真正的目标,即点击的元素

currentTarget 例如点击按钮冒泡到了body上，触发了body的事件处理程序，那么currentTarget 为body
type 事件类型

cancelable 布尔值表示是否可以取消事件的默认行为，如果该事件的 cancelable 属性为 false, 则该事件的监听器无法阻止默认行为，调用**preventDefault()** 将产生错误.

eventPhase 表示事件正位于哪个阶段

方法：

preventDefault() 阻止默认事件，cancelable 为true时才可以使用

stopPropagation()取消捕获或者冒泡

IE中

DOM0级添加事件处理程序时event对象作为一个window属性存在。

attachEvent添加时会传递一个event对象，也可以在window中访问

属性：

srcElement 等于DOM中的target

returnValue 布尔值表示是否开启默认行为 默认为true

cancelBubble 默认为false,表示是否取消事件冒泡

type

4 事件类型

ui事件:

load : 当页面完全加载后 (包括所有js, css, 图像等外部资源) 就会触发

resize: 当浏览器窗口大小调整时就会触发

scroll : 滚动时触发

焦点事件:

focus: 元素获得焦点时触发, 不会冒泡

blur: 元素失去焦点时触发, 不会冒泡

focusin 元素获得焦点时触发, 会冒泡

focusout 元素失去焦点时触发, 会冒泡

鼠标与滚轮事件:

click: 单击 等于一个mousedown+mouseup

dblclick: 双击

mousedown: 按下任意鼠标按钮时触发 event中保存着一个button属性, 表示按钮的类别

mouseup: 释放任意鼠标按钮时触发

mousemove: 鼠标在元素内部移动时触发

mouseenter: 鼠标移入时触发, **不冒泡**。移到后代元素也不触发

mouseleave: 移出时触发, **不冒泡**

mouseover: **冒泡**

mouseout: **冒泡**

mousewheel 滚轮事件, event有一个wheelDelta值, 向上滚为120 的倍数, 向下-120的倍数

contextmenu 右键菜单事件, 阻止默认事件可以自定义菜单

鼠标事件的事件对象中保存着

鼠标指针在视图中的水平和垂直坐标**clientX**和**clientY**

鼠标指针在页面中的水平和垂直坐标**pageX**和**pageY**

鼠标指针在整个屏幕中的水平和垂直坐标**screenX**和**screenY**

键盘事件:

keydown:按下任意键触发, 按住不放会重复触发

keypress:按下任意字符键触发, 按住不放会重复触发

keyup: 释放时触发

event中 保存keyCode 键码 回车13

input: 在可编辑区域中输入字符时, 会触发这个事件, event中有一个属性data包含这个输入字符的值

事件委托:

原理: 基于事件冒泡, 指定一个事件处理程序就可以管理某一类的所有事件

优点 减少内存占用, 提升性能, 简化代码, 只添加一个事件处理程序时间快

BOM

window对象

表示浏览器实例，在浏览器中window对象包含着ECMAScript 规定的Global对象，常用方法如下

<code>alert()</code>	显示带有一段消息和一个确认按钮的警告框。
<code>clearInterval()</code>	取消由 <code>setInterval()</code> 设置的 timeout。
<code>clearTimeout()</code>	取消由 <code>setTimeout()</code> 方法设置的 timeout。
<code>close()</code>	关闭浏览器窗口。
<code>open()</code>	打开一个新的浏览器窗口或查找一个已命名的窗口。
<code>setInterval()</code>	按照指定的周期（以毫秒计）来调用函数或计算表达式。
<code>setTimeout()</code>	在指定的毫秒数后调用函数或计算表达式。

window.screen

每个 Window 对象的 screen 属性都引用一个 Screen 对象。Screen 对象中存放着有关显示浏览器屏幕的信息。

<code>availHeight</code>	返回显示屏幕的高度 (除 Windows 任务栏之外)。
<code>availWidth</code>	返回显示屏幕的宽度 (除 Windows 任务栏之外)。
<code>height</code>	返回显示屏幕的高度。
<code>width</code>	返回显示器屏幕的宽度。

window.navigator

Navigator 对象包含有关浏览器的信息

<code>appCodeName</code>	返回浏览器的代码名。
<code>appName</code>	返回浏览器的名称。一般为Netscape
<code>appVersion</code>	返回浏览器的平台和版本信息。包含真正的浏览器名称
<code>cookieEnabled</code>	返回指明浏览器中是否启用 cookie 的布尔值。
<code>platform</code>	返回运行浏览器的操作系统平台。
<code>userAgent</code>	返回由客户机发送服务器的 user-agent 头部的值。

window.location

提供了与当前窗口中加载的文档有关的信息，它即是window对象的属性 也是 document对象的属性

属性：

<code>hash</code>	设置或返回从井号 (#) 开始的 URL (锚) 。
<code>host</code>	设置或返回主机名+端口号。
<code>hostname</code>	设置或返回当前 URL 的主机名。
<code>href</code>	设置或返回完整的 URL。
<code>pathname</code>	设置或返回当前 URL 的路径部分。
<code>port</code>	设置或返回当前 URL 的端口号。
<code>protocol</code>	设置或返回当前 URL 的协议。
<code>search</code>	设置或返回从问号 (?) 开始的 URL (查询部分) 。

例： `http://localhost:63342/z.html?a=1&b=2#123`

1. `hash:"#123"`
2. `host:"localhost:63342"`
3. `hostname:"localhost"`
4. `href:"http://localhost:63342/z.html?a=1&b=2#123"`
5. `origin:"http://localhost:63342"`
6. `pathname:"/z.html"`
7. `port:"63342"`
8. `protocol:"http:"`
9. `search:"?a=1&b=2"`

方法：

`location.assign(url)` 跳转至新url并在历史记录生成一条记录

使用 `window.location` 或者 `location.href` 设置为一个url的值，也会调用assign方法

改变除了hash以外的属性，页面都会以新的url重新加载

`location.replace(url)` 方法不会在历史记录中生成记录

`location.reload()` 重新加载，传true则不使用缓存从服务器重新加载

window.history

它暴露了很多有用的方法和属性，允许你在用户浏览历史中向前和向后跳转，同时——从HTML5开始——提供了对history栈中内容的操作。

方法

方法	描述
back()	加载 history 列表中的前一个 URL。
forward()	加载 history 列表中的下一个 URL。
go()	加载 history 列表中的某个具体页面。
pushState(stateObj,title,url)	<p>增加一条历史纪录条目，浏览器显示的url会改变为设置的参数，但不会加载在某种意义上，调用 <code>pushState()</code> 与 设置 <code>window.location =</code> 页面创建并激活新的历史记录。但 <code>pushState()</code> 具有如下几条优点</p> <ul style="list-style-type: none"> ● 新的 URL 可以是与当前URL同源的任意URL。而设置 <code>window.location =</code> 时才保持同一个 <code>document</code>。 ● 如果需要，你可以不必改变URL。而设置 <code>window.location = "</code> 情况下，仅仅是新建了一个新的历史记录项。 ● 你可以为新的历史记录项关联任意数据。而基于哈希值的方式，则必须字符串里
replaceState(stateObj,title,url)	与 <code>history.pushState()</code> 非常相似，区别在于 <code>replaceState()</code> 不是新建一个。

一般pushState和replaceState配合 `window.onpopstate`事件，事件对象包含一个state属性是`history.state`的副本，`onpopstate`在使用`go,back,forward`时调用

属性

属性	描述
<code>length</code>	返回浏览器历史列表中的 URL 数量。
<code>state</code>	当使用pushState或者replaceState时会为该历史条目增加state对象

ES6

let/const

共同特性

1 块级作用域，不会定义到window上

例：循环的i值，使用var定义时，内部取i时取的是全局变量i的值，使用let时，每次循环都是一个代码块，会定义新的i，内部每次取到的i不一样

注：for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

2 不存在变量声明提升

变量声明提升存在于执行环境创建的第一阶段，只会扫描var声明的变量放入VO

3 不允许重复声明

4 暂时性死区

ES6 明确规定，如果区块中存在`let`和`const`命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域（即不会从作用域链上再查找此变量）。凡是在声明之前就使用这些变量，就会报错。

```
1 var a = 1;
```

```
2 function fn() {  
3     a = 2;  
4     let a = 3;  
5 }  
6 fn(); // a is not defined
```

const特性

1 声明时必须赋值

2 变量指向的栈内存地址所保存的数据不得改动，对于基本类型的数据，栈内存地址保存的即是数据值，所以不能改变，对于引用类型的数据，栈内存地址保存的是堆内存的地址，所以可以给对象添加属性，给数组增加项，但不能赋值为另一个对象/数组（使用Object.freeze方法冻结对象）

解构赋值

数组的解构赋值

```
1 let [a, b] = [1, 3];
```

根据位置解构，解构不成功变量的值为undefined，如果等号右边不是可遍历解构，那么会报错

可以设默认值 let [a = 1] = []；解构后的值必须 === undefined 时才会取默认值

默认值可以引用解构赋值的其他变量，但该变量必须已经声明。let [x=y, y=1] = []；// 报错，因为y还未声明

对象的解构赋值

```
1 let { a } = { a: 1 }
```

对象的属性没有次序，变量必须与属性同名，才能取到正确的值，

如果变量名和属性名不一致，例如变量名为b，属性名为a，需要写成

```
1 let { a: b } = { a : 1 };
```

实际上，最上面的例子是简写，完整的应该是 let { a : a } = { a : 1 }；对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

也可以设默认值，和数组相同

其他类型的解构赋值：字符串（有length，类似数组）、数值和布尔值（先转换为对象再解构，不能转换为对象的null和undefined都会报错）、函数

字符串扩展

ES6 为字符串添加了遍历器接口，使得字符串可以被for...of循环遍历。

新增方法：

1 includes()：返回布尔值，表示是否找到了参数字符串。

2 startsWith()：返回布尔值，表示参数字符串是否在原字符串的头部。会隐式转换

3 endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。会隐式转换

这三个方法都支持第二个参数，表示开始搜索的位置。

4 repeat(次数)：返回新字符串，将字符串重复n次，参数为小数时向下取整，负数报错，为0或NaN则返回空字符串。

5 padStart() 'x'.padStart(4, 'ab') // 'abax' 头部补全

6 padEnd() 'x'.padEnd(5, 'ab') // 'xabab' 尾部补全

模板字符串

模板字符串 (template string) 是增强版的字符串，用反引号 (`) 标识。

模板字符串中嵌入变量，需要将变量名写在 \${} 之中。

如果大括号中的值不是字符串，将按照一般的规则转为字符串。比如，大括号中是一个对象，将默认调用对象的 `toString` 方法。

正则扩展

数值的扩展

提供了二进制和八进制的新写法，分别用前缀 0b/0B 和 0o/0O 表示

Number.isFinite() 检查数值是否有限, Infinity 和非数字类型都返回 false

Number.isNaN() 检查一个值是否为 NaN

Number.isInteger() 检查一个值是否为整数

将 `parseInt` 和 `parseFloat` 移植到 `Number` 上，目的是逐步减少全局性方法，使得语言逐步模块化。

Math 对象扩展

Math.trunc() 去小数返回整数，对于空值和无法截取整数的值，返回 NaN

Math.sign() 判断一个数到底是正数、负数、还是零。

它会返回五种值。

- 参数为正数，返回 +1；
- 参数为负数，返回 -1；
- 参数为 0，返回 0；
- 参数为 -0，返回 -0；
- 其他值，返回 NaN。

函数的扩展

增加了参数默认值

指定了默认值以后，函数的 `length` 属性，将返回没有指定默认值的参数个数。也就是说，指定了默认值后，`length` 属性将失真。如果设置了默认值的参数不是尾参数，那么 `length` 属性也不再计入后面的参数了。

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个 **单独的作用域**。等到初始化结束，这个作用域就会消失。

```
1 let foo = 'outer';
2 function bar(func = () => foo) {
3   let foo = 'inner';
4   console.log(func());
5 }
6 bar(); // outer
```

应用：利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误。（`x = throwError() => {}`）

增加了 rest 参数（形式为 `...变量名`），用于获取函数的多余参数，该变量将多余的参数放到数组中

应用：替代 `arguments`, `arguments` 是类数组，使用数组的方法还需要先将其转换为数组

关于严格模式：ES2016 做了一点修改，规定只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错。

新增 `name` 属性返回函数名，`bind` 返回的函数，`name` 属性值会加上 `bound` 前缀。

箭头函数

```
1 var f = v => v
2 // 等同于
3 var f = function (v) { return v; }
```

如果没有参数用()代表参数部分，由于{}被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错。

注意点：

- 1 不能用new操作符
- 2 不可以使用yield命令，因此箭头函数不能用作 Generator 函数。
- 3 没有arguments对象
- 4 函数体内的this对象，就是定义时所在的对象，所以也不能使用bind call apply

数组的扩展

扩展运算符 (spread) 是三个点 (...)。将一个数组转为用逗号分隔的参数序列。

任何 Iterator 接口的对象（参阅 Iterator 一章），都可以用扩展运算符转为真正的数组。

```
1 console.log([... 'abc']) // ['a', 'b', 'c']
```

如果扩展运算符后面是一个空数组，则不产生任何效果。

应用：

1 替代apply:

```
1 es5: Math.max.apply(null, [1, 2, 3]);
2 es6: Math.max(...[1, 2, 3]);
```

2 复制/合并数组

```
1 const a1 = [1, 2, 3];
2 const a2 = [...a1]
```

3 与解构赋值结合

```
1 es5:
2 const a = list[0];
3 const rest = list.slice(1)
4 es6:
5 const [a, ...rest] = list
```

Array.from(arrayLike) 方法用于将两类对象转为真正的数组：类似数组的对象 (array-like object) 和可遍历 (iterable) 的对象（包括 ES6 新增的数据结构 Set 和 Map），还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

常见的类似数组的对象是 DOM 操作返回的 NodeList 集合，以及函数内部的arguments对象。

任何有length属性的对象，都可以通过Array.from方法转为数组

```
Array.from({ length: 3 }); // [ undefined, undefined, undefined ]
```

Array.of() 将一组值，转换为数组，基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的重载。它的行为非常统一。

`Array.of` 方法用于将一组值，转换为数组。

```
Array.of(3, 11, 8) // [3,11,8]
Array.of(3) // [3]
Array.of(3).length // 1
```

这个方法的主要目的，是弥补数组构造函数 `Array()` 的不足。因为参数个数的不同，会导致 `Array()` 的行为有差异。

```
Array() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

`copyWithin(target, start = 0, end = this.length)` 将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组

`[1, 2, 3, 4, 5].copyWithin(0, 3) // [4, 5, 3, 4, 5]` 将位置3到最后也就是最后两项4, 5复制到位置0

数组实例的

`find()`方法找出第一个符合条件的成员，找不到返回`undefined`

`[1, 5, 10, 15].find((value, index, arr) => value > 9) // 10`

`findIndex()` 找出第一个符合条件的index,找不到返回 -1

这两个方法都可以接受第二个参数，用来绑定回调函数的 `this` 对象。

`fill(填充值, 起始位置, 终止位置)` 使用给定值，填充一个数组。改变原数组

`entries(), keys()和values()`——用于遍历数组。它们都返回一个遍历器对象，可以用`for...of`循环进行遍历，唯一的区别是`keys()`是对键名的遍历、`values()`是对键值的遍历，`entries()`是对键值对的遍历。

`includes(值, 起始位置)` 和字符串的`includes`类似

关于数组的空位

数组的某一个位置没有任何值。比如，`Array`构造函数返回的数组都是空位，空位不是`undefined`，一个位置的值等于`undefined`，依然是有值的。空位是没有任何值

`new Array(3) // [, ,]`

ES5 对空位的处理，已经很不一致了，大多数情况下会忽略空位。

- `forEach()`, `filter()`, `reduce()`, `every()` 和`some()`都会跳过空位。
- `map()`会跳过空位，但会保留这个值
- `join()`和`toString()`会将空位视为`undefined`，而`undefined`和`null`会被处理成空字符串。

ES6 则是明确将空位转为`undefined`。

对象的扩展

属性和方法的简写（简洁写法的属性名总是字符串）

属性：`const obj = { a };` 等于 `const obj = { a: 'a' };`

方法：`const obj = { func: function() {xxx} };` 等于 `const obj = { func() {xxx} };`

属性名表达式使用[], 与简洁表示法，不能同时使用

ES6 允许字面量定义对象时，用表达式作为对象的属性名，即把表达式放在方括号内。

```
let a="abc" let obj={[a]:123}
```

属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串[object Object]

Object.is(值1, 值2) 与 === 不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

Object.assign (target,source1,source2) 方法用于对象的合并，将源对象 (source) 的所有可枚举属性，复制到目标对象 (target)。除了字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果。只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性 (enumerable: false)。

Object.setPrototypeOf(object, prototype) 用来设置一个对象的prototype对象，返回参数对象本身。

Object.getPrototypeOf(object) 取对象的原型

Object.getOwnPropertyDescriptors() 返回指定对象所有自身属性（非继承属性）的描述对象。

super 关键字 指向当前对象的原型对象。只能用在对象的方法之中

Symbol

表示独一无二的值，通过**symbol**函数生成，不能使用new，因为生成的symbol不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

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

```
1 console.log(Symbol(123)) // Symbol(123)
```

Symbol函数的参数只是表示对当前 Symbol 值的描述，因此相同参数的**Symbol**函数的返回值是不相等的。

Symbol 值不能与其他类型的值进行运算，但是Symbol 值可以显式转为字符串。另外，Symbol 值也可以转为布尔值，但是不能转为数值。

在对象的内部，使用 Symbol 值定义属性时，Symbol 值必须放在方括号之中。

Symbol 作为属性名，不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回。但是，它也不是私有属性，有一个**Object.getOwnPropertySymbols**方法，可以获取指定对象的所有 Symbol 属性名。

另一个新的API，**Reflect.ownKeys**方法可以返回所有类型的键名，包括常规键名和 Symbol 键名。

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

```
Symbol('abc') === Symbol('abc') // false
```

```
Symbol.for('abc') === Symbol.for('abc') // true
```

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

Set

它类似于数组，但是成员的值都是唯一的，没有重复的值。

[...new Set(array)] 数组去重的一种方法，向 Set 加入值的时候，**不会发生类型转换**，所以5和"5"是两个不同的值。在 Set 内部，**NaN**是相等。

属性：

size 返回Set实例的成员总数

方法：

- `add(value)`: 添加某个值，返回 Set 结构本身。
- `delete(value)`: 删除某个值，返回一个布尔值，表示删除是否成功。
- `has(value)`: 返回一个布尔值，表示该值是否为Set的成员。
- `clear()`: 清除所有成员，没有返回值。

Set 结构的实例有四个遍历方法，可以用于遍历成员。Set的遍历顺序就是**插入顺序**。

- `keys()`: 返回键名的遍历器
- `values()`: 返回键值的遍历器
- `entries()`: 返回键值对的遍历器
- `forEach()`: 使用回调函数遍历每个成员

由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以`keys()`, `values()`, `entries()`得到的结果相同

如果想在遍历操作中，同步改变原来的 Set 结构，目前没有直接的方法，但有两种变通方法。一种是利用原 Set 结构映射出一个新的结构，然后赋值给原来的 Set 结构；另一种是利用`Array.from`方法。

WeakSet 结构与 Set 类似，也是不重复的值的集合。

与 Set 的区别：

1 WeakSet 的成员只能是**对象/数组**，而不能是其他类型的值。

2 WeakSet 中的对象都是**弱引用**，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

3 没有size属性,没有clear方法，不能遍历

用法：使用`weakSet` 储存 DOM 节点，而不用担心这些节点从文档移除时，会引发内存泄漏。

Map

类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“**值—值**”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

任何具有 Iterator 接口、且每个成员都是一个**双元素的数组**的数据结构都可以当作 Map 构造函数的参数。

```
new Map([[{}, 1], [true, 123]]);
```

Map 的键实际上是跟**内存地址绑定的**，只要内存地址不一样，就视为两个键。（`undefined`和`null`也是两个不同的键）如果 Map 的键是一个简单类型的值（数字、字符串、布尔值），则只要两个值严格相等，Map 将其视为一个键

• 属性：

size 返回Map实例的成员总数

方法：

- set(key,value)：添加某个值，返回 Map 结构本身。
- get(key)：返回某个值，没有则返回undefined。
- delete(key)：删除某个值，返回一个布尔值，表示删除是否成功。
- has(key)：返回一个布尔值，表示该值是否为Map的成员。
- clear()：清除所有成员，没有返回值。

遍历，Map 的遍历顺序就是插入顺序。方法同Set

WeakMap 结构与 Map 类似，也是用于生成键值对的集合。

区别：

- 1 WeakMap只接受对象/数组作为键名（null除外），不接受其他类型的值作为键名。
- 2 WeakMap的键名所指向的对象，不计入垃圾回收机制。同WeakSet
- 3 没有size属性,没有clear方法，不能遍历

Promise

<http://www.zbzero.com/#/singleArticle/87>

Generator/Async

<http://www.zbzero.com/#/singleArticle/88>

Class

基本概念

class是构造函数的语法糖

类的数据类型就是函数，类本身就指向构造函数

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

类的所有方法都定义在类的prototype属性上面，在类的实例上面调用方法，其实就是调用原型上的方法。

类的内部所有定义的方法，都是不可枚举的（non-enumerable）。

constructor

constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

constructor方法默认返回实例对象（即this），可以指定返回另外一个对象。

类必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行。

与ES5一样，实例的属性除非显式定义在其本身（即定义在this对象上），否则都是定义在原型上（即定义在class上）。

在“类”的内部可以使用get和set关键字，对某个属性设置存值函数和取值函数，拦截该属性的存取行为。（注意：使用此方法属性定义在原型对象上）

Class可以使用表达式定义，如下

```
1 const MyClass = class Me {};
```

类名是MyClass而不是Me

注意点：

- 1 类内部默认就是严格模式
- 2 类不存在变量提升 (hoist)
- 3 在一个方法前，加上static关键字，称为“静态方法”。静态方法可以与非静态方法重名。父类的静态方法，可以被子类继承。静态方法中的this指的是类而不是实例
- 4 属性可以不写在constructor内，可以写在顶层不用加this,效果一样也是定义在(若constructor内和外定义了同名属性，以constructor为准)

ES6 为new命令引入了一个new.target属性，该属性一般用在构造函数之中，返回new命令作用于的那个构造函数。如果构造函数不是通过new命令调用的，new.target会返回undefined，因此这个属性可以用来确定构造函数是怎么调用的。

```
1 function A() {  
2     console.log(new.target)  
3 }  
4 A(); // undefined  
5 const a = new A() // Function A
```

子类继承父类时，new.target会返回子类。

继承

Class可以通过extends关键字继承

子类必须在constructor方法中调用super方法，否则新建实例时会报错。这是因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法，子类就得不到this对象。

如果子类没有定义constructor方法，这个方法会被默认添加，代码如下。也就是说，不管有没有显式定义，任何一个子类都有constructor方法。

```
1 class Sub extends Parent {}  
2 // 等同于  
3 class Sub extends Parent {  
4     constructor (...args) {  
5         super(...args);  
6     }  
7 }
```

通过Object.getPrototypeOf(Sub) === Parent 判断一个类是否继承了另一个类

super关键字：

- 1 作为函数调用时代表父类的构造函数，在constructor中super()相当于父类.prototype.constructor.call(this),作为函数时，super()只能用在子类的构造函数之中
- 2 super作为对象时

在普通方法中，指向父类的原型对象，通过super调用父类方法时，方法内部this指向当前子类实例，使用super对属性赋值时，赋值的也是子类实例的属性

在静态方法中，指向父类。在子类的静态方法中通过super调用父类的方法时，方法内部的this指向当前的子类，而不是子类的实例。

使用super的时候，必须显式指定是作为函数、还是作为对象使用，否则会报错。

```
1 console.log(super); // 报错
```

Class 作为构造函数的语法糖，同时有prototype属性和__proto__属性，因此同时存在两条继承链。

(1) 子类本身的__proto__属性，表示构造函数的继承，总是指向父类。实现了类的静态方法的继承

(2) 子类prototype属性的__proto__属性，表示方法的继承，总是指向父类的prototype属性。实现了类的实例

作为一个对象，子类（B）的原型（__proto__属性）是父类（A）；作为一个构造函数，子类（B）的原型对象（prototype属性）是父类的原型对象（prototype属性）的实例。

修饰器Decorator

类的修饰

```
1 @testable //修饰器
2 class MyTestableClass {
3 }
4 function testable(target) {
5   target.isTestable = true;
6 }
7 MyTestableClass.isTestable //true
```

@testable就是一个修饰器。它修改了MyTestableClass这个类的行为，为它加上了静态属性isTestable。testable函数的参数target是MyTestableClass类本身。

修饰器是一个对类进行处理的函数。修饰器函数的第一个参数，就是所要修饰的目标类。

方法的修饰

```
1 class Person {
2   @readonly //修饰器
3   name() { return `${this.first} ${this.last}` }
4 }
5 function readonly(target, name, descriptor){ //参数依次为类的实例，属性名，属性的描述对象
6   descriptor.writable = false;
7   return descriptor;
8 }
```

二进制数组

由三个对象组成。

ArrayBuffer对象、TypedArray对象、DataView对象是JavaScript操作二进制数据的一个接口。这些对象早就存在，属于独立的规格，ES6将它们纳入了ECMAScript规格，并且增加了新的方法。

目的：为了加快字节流数据的处理速度（js里的array功能复杂，效率低）

1 ArrayBuffer对象

代表内存之中的一段二进制数据，它不能直接读写，只能通过视图（**TypedArray**视图和**DataView**视图）来读写，视图的作用是以指定格式解读二进制数据。这意味着，可以用数组的方法操作内存。

创建：

```
1 var buf = new ArrayBuffer(length) 创建一个长度为length字节的内存区域，初始值都为0的ArrayB
```

使用**DataView**读取：

```
1 var dataView = new DataView(buf);  
2 dataView.getInt8(0) //0
```

使用**TypedArray**读取：

```
1 var typedArr = new Uint8Array(buf);  
2 console.log(typedArr[0]) //0
```

若同一段内存建立了多个视图，其中一个视图修改了内存，会影响另一个视图

从现有数据中获取：

```
1 const reader = new FileReader();  
2 reader.readAsArrayBuffer(file); //将文件读取为 ArrayBuffer
```

实例属性/方法：

1 byteLength 内存区域的字节长度

2 slice 允许将内存区域的一部分拷贝生成一个新的ArrayBuffer，除了**slice**方法，**ArrayBuffer**对象不提供任何直接读写内存的方法

静态方法：

isView(参数) 判断参数是否为**ArrayBuffer**的视图实例，即判断参数是否为**TypedArray**或**DataView**

2 TypedArray对象

用来生成内存的视图，同一段内存，不同数据有不同的解读方式，这就叫做“视图”（view），通过9个构造函数，可以生成9种数据格式的视图，比如**Uint8Array**（无符号8位整数）数组视图，**Int16Array**（16位整数）数组视图，**Float32Array**（32位浮点数）数组视图等等。这9个构造函数生成的对象，统称为**TypedArray**对象。

所有数组的方法，在类型化数组上面都能使用。与普通数组的区别：

1 **TypedArray**数组的所有成员，都是同一种类型和格式。

2 **TypedArray**数组的成员是连续的，不会有空位。

3 **TypedArray**数组成员的默认值为0

4 **TypedArray**数组只是一层视图，本身不储存数据，它的数据都储存在底层的**ArrayBuffer**对象之中，要获取底层对象必须使用**buffer**属性。

例：

```
const ab = new ArrayBuffer(10);  
const view = new Uint8Array(ab);  
view[0] = 1;
```

TypedArray的构造函数

(1) **TypedArray(buffer, byteOffset=0, length?)** 可以接收三个参数

- 第一个参数（必需）：视图对应的底层**ArrayBuffer**对象。

- 第二个参数（可选）：视图开始的字节序号，默认从0开始。
- 第三个参数（可选）：视图包含的数据个数，默认直到本段内存区域结束。

(2) **TypedArray(length)** 视图还可以不通过**ArrayBuffer**对象，直接分配内存而生成。

```
var f64a = new Float64Array(8);
```

生成一个8个成员的 `Float64Array` 数组，共64字节（每个64位浮点数占8字节）

(3) **TypedArray(typedArray)**

可以接受另一个视图实例作为参数。生成的新数组，只是复制了参数数组的值，对应的底层内存是不一样的。

(4) **TypedArray(arrayLikeObject)**

构造函数的参数也可以是一个普通数组，然后直接生成**TypedArray**实例。TypedArray视图会重新开辟内存，不会在原数组的内存上建立视图。

```
1 var f32a = new Float32Array([1,2,3]) // length: 3, byteLength:12
```

每一种构造函数都有一个`BYTES_PER_ELEMENT`属性，表示这种数据类型占据的字节数。这个属性在**TypedArray**实例上也能获取

处理溢出： 例如8位视图只能容纳八位的二进制数，若设置为256，即九位二进制数100000000，则只会保留后八位，若是无符号8位整数，赋值负数例-1，先转换为对应的正数1，二进制表示即为00000001，再进行否运算得11111110，再加一最后为255

3 DataView对象

用来生成内存的视图，可以自定义格式和字节序，比如第一个字节是**Uint8**（无符号8位整数）、第二个字节是**Int16**（16位整数）、第三个字节是**Float32**（32位浮点数）等等。

DataView实例有以下属性，含义与**TypedArray**实例的同名方法相同。

- `DataView.prototype.buffer`: 返回对应的**ArrayBuffer**对象
- `DataView.prototype.byteLength`: 返回占据的内存字节长度
- `DataView.prototype.byteOffset`: 返回当前视图从对应的**ArrayBuffer**对象的那个字节开始

DataView实例提供8个方法读取内存。

- `getInt8`: 读取1个字节，返回一个8位整数。
- `getUint8`: 读取1个字节，返回一个无符号的8位整数。
- `getInt16`: 读取2个字节，返回一个16位整数。
- `getUint16`: 读取2个字节，返回一个无符号的16位整数。
- `getInt32`: 读取4个字节，返回一个32位整数。
- `getUint32`: 读取4个字节，返回一个无符号的32位整数。
- `getFloat32`: 读取4个字节，返回一个32位浮点数。
- `getFloat64`: 读取8个字节，返回一个64位浮点数。

这一系列get方法的第一个参数都是一个字节序号（不能是负数，否则会报错），表示从哪个字节开始读取。

DataView视图提供8个方法写入内存。

- `setInt8`: 写入1个字节的8位整数。
- `setUint8`: 写入1个字节的8位无符号整数。
- `setInt16`: 写入2个字节的16位整数。
- `setUint16`: 写入2个字节的16位无符号整数。
- `setInt32`: 写入4个字节的32位整数。
- `setUint32`: 写入4个字节的32位无符号整数。
- `setFloat32`: 写入4个字节的32位浮点数。
- `setFloat64`: 写入8个字节的64位浮点数。

两个参数，第一个参数是字节序号，表示从哪个字节开始写入，第二个参数为写入的数据

简单说，ArrayBuffer对象代表原始的二进制数据，TypedArray对象代表确定类型的二进制数据，DataView对象代表不确定类型的二进制数据。它们支持的数据类型一共有9种（DataView对象支持除Uint8C以外的其他8种）。

数据类型	字节长度	含义	对应的C语言类型
Int8	1	8位带符号整数	signed char
Uint8	1	8位不带符号整数	unsigned char
Uint8C	1	8位不带符号整数 (自动过滤溢出)	unsigned char
Int16	2	16位带符号整数	short
Uint16	2	16位不带符号整数	unsigned short
Int32	4	32位带符号整数	int
Uint32	4	32位不带符号的整数	unsigned int
Float32	4	32位浮点数	float
Float64	8	64位浮点数	double

用到二进制数组的api

- File API

```
1 <input type="file" id="file1">
2 <button onclick="getArrayBuffer();">获取二进制数据，并以八位正数格式打印</button>
3 <div id="content">
4 </div>
5 <script>
6     var fileInput = document.getElementById('file1');
7     var content = document.getElementById('content');
8     // 读取文件为二进制buffer
9     function readFileBuffer(file) {
10         return new Promise(function (resolve) {
11             var reader = new FileReader();
12             reader.readAsArrayBuffer(file);
13             reader.onload = function () {
14                 resolve(reader.result)
15             }
16         })
17     }
18     // 获取buffer并打印
19     function getArrayBuffer() {
20         var file = fileInput.files[0];
21         readFileBuffer(file).then(function (buf) {
22             var i8A = new Int8Array(buf);
23             content.innerText = i8A.join(' ');
24         })
25     }
26 }
```

- ```
24 })
25 }

 - XMLHttpRequest
 - Fetch API
 - Canvas
 - WebSockets
```

参考：<http://javascript.ruanyifeng.com/stdlib/arraybuffer.html#toc16>

## 框架: vue/react/koa2...

### Vue和react

- 使用 Virtual DOM
- 提供了响应式 (Reactive) 和组件化 (Composable) 的视图组件。
- 将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库。

### 状态更新:

React当某个组件的状态发生改变时,它会以该组件为根,重新渲染整个组件子树, 为了比避免不必要的子组件渲染, 可能使用 PureComponent或者shouldComponentUpdate 方法

在 Vue 应用中, 组件的依赖是在渲染过程中自动追踪的, 所以系统能精确知晓哪个组件确实需要被重渲染。你可以理解为每一个组件都已经自动获得了 shouldComponentUpdate,

Html: React使用JSX, 灵活性高, 因为拥有所有js语言的功能, 比如临时变量和流程控制, 有个学习成本

Vue使用模板, 读起来更自然, v-if, v-for指令以及v-on的各种修饰符, 在JSX中实现需要更多的代码,vue也提供了渲染函数, 也支持jsx

Css:react 使用css in js, vue在单文件组件中style标签添加scoped为组件内的css指定作用域

## 虚拟dom

好处:

- 1 通过diff可以实现最小化的更新
- 2 使用虚拟dom 减少了dom操作次数, 从而减少了浏览器重绘 / 重排的次数
- 3 跨平台性

## HTML

### meta标签

#### 定义和用法

<meta> 元素可提供有关页面的元信息 (meta-information) 。

<meta> 标签位于文档的头部, 不包含任何内容。<meta> 标签的属性定义了与文档相关联的名称/值对

最常见的

<meta charset="UTF-8"> charset属性设置了文档的字符编码

属性：

**content**: 定义与 http-equiv 或 name 属性相关的元信息。

**http-equiv** 把content属性关联到HTTP头部

| 值               | 描述                                                | 例子                                                                                                                                                                                     |
|-----------------|---------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| content-type    | 设定页面使用的字符集                                        | <meta http-equiv="content-Type" content="text/GB2312时，代表说明网站是采用的编码是简体中文ISO-8859-1时，代表说明网站是采用的编码是英文UTF-8时，代表世界通用的语言编码；<br>PS：html5页面的做法是直接使用<meta charset='                             |
| X-UA-Compatible | IE专用标记，用来指定IE浏览器去模拟某个特定版本的IE浏览器的渲染方式，以此来解决部分兼容问题。 | <meta http-equiv="X-UA-Compatible" content="<br/>以上代码告诉IE浏览器，无论是否用DTD声明文档类型，都以IE8模式显示。<br/><meta http-equiv="X-UA-Compatible" content="IE=8" /><br/>以上代码告诉IE浏览器，IE8/9及以后的版本都会以IE8模式显示。 |
| expires         | 设定网页的过期时间。                                        | <meta http-equiv="expires" content="Fri,12Jan2010 12:00:00 GMT"><br>PS：必须使用GMT的时间格式                                                                                                    |
| refresh         | 自动刷新并指向新页面。                                       | <meta http-equiv="Refresh" content="2;URL=http://www.bing.com"><br>PS：2代表页面停留2秒后跳转到后面的网址上                                                                                              |
| set-cookie      | 如果网页过期，那么自动删除本地cookie。                            | <meta http-equiv="Set-Cookie" content="cookie_name=cookie_value; expires=Fri, 12-Jan-2010 12:00:00 GMT"><br>PS：必须使用GMT的时间格式。                                                           |
| windows-target  | 强制页面在当前窗口中以独立页面显示，可以防止自己的网页被别人当作一个frame页调用        | <meta http-equiv="Window-target" content="_top">                                                                                                                                       |
| cache-control   | 缓存机制                                              | <meta http-equiv="cache-control" content="no-store">                                                                                                                                   |

**name** 主要用于描述网页，与之对应的属性值为content，content中的内容主要是便于搜索引擎机器人查找信息和分类信息用的。

| 值                  | 描述                           | 例子                                                                                                                                                                                                                                                                  |
|--------------------|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| author             | 标注网页的作者                      | <meta name="author" content="dashen" />                                                                                                                                                                                                                             |
| <b>keywords</b>    | 页面关键词，用于被搜索引擎收录              | <meta name="keywords" content="新闻,新闻中心" />                                                                                                                                                                                                                          |
| <b>description</b> | 页面描述，用于搜索引擎收录                | <meta name="description" content="新闻中心,包括时事评论、新闻图片、新闻专题、新闻论坛、军事" />                                                                                                                                                                                                 |
| <b>viewport</b>    | 用于控制页面缩放                     | <meta name="viewport" content="width=device-width, maximum-scale=1, minimum-scale=1, user-scalable=no" /><br>详情可查看： <a href="http://www.cnblogs.com/lovesong/">http://www.cnblogs.com/lovesong/</a>                                                                 |
| <b>renderer</b>    | 指定双核浏览器默认以何种方式渲染页面。          | <meta name="renderer" content="webkit" />//默认<br><meta name="renderer" content="ie-comp" />//兼容<br><meta name="renderer" content="ie-stand" />//标准<br>PS：360浏览器支持                                                                                                   |
| revised            | 网页文档的修改时间                    | <meta name="revised" content="设计网, 6/24/2014" />                                                                                                                                                                                                                    |
| robots             | 用来告诉搜索机器人哪些页面需要索引，哪些页面不需要索引。 | <meta name="robots" content="none" /><br>取值： all none index noindex follow nofollow, 默认为 all。<br>all：文件将被检索，且页面上的链接可以被查询；<br>none：文件将不被检索，且页面上的链接不可以被索引；<br>index：文件将被检索；<br>follow：页面上的链接可以被查询；<br>noindex：文件将不被检索，但页面上的链接可以被索引；<br>nofollow：文件将不被检索，页面上的链接不可以被索引； |
| copyright          | 网站版权信息                       | <meta name="copyright" content="本页版权XXX" />                                                                                                                                                                                                                         |

H5

<!DOCTYPE>

# 新元素

## <canvas> 画布

多媒体元素

<audio> 定义声音，比如音乐或其他音频流。

目前，`<audio>` 元素支持的3种文件格式： MP3、Wav、Ogg。

**<video>** 定义视频，比如电影片段或其他视频流

目前，`<video>` 元素支持三种视频格式：MP4、WebM、Ogg。

**<source>** 标签为媒体元素（比如 **<video>** 和 **<audio>**）定义媒体资源。

**<embed>** 标签定义了一个容器，用来嵌入外部应用或者互动程序（插件）

## 表单元素

`<datalist>` 标签规定了 `<input>` 元素可能的选项列表。带下拉和自动填充的功能

## 新的语义和结构元素

<article><footer><header><nav><section>

# Web 存储

## localStorage对象

- 保存数据: localStorage.setItem(key,value);
- 读取数据: localStorage.getItem(key);
- 删除单个数据: localStorage.removeItem(key);
- 删除所有数据: localStorage.clear();
- 得到某个索引的key: localStorage.key(index);

sessionStorage对象

## 新的表单属性

**autofocus** 自动聚焦

**placeholder** 提示

## 浏览器内核

Trident : IE 360

Gecko: firefox

Webkit: Safari

Blink: chrome

## 回流 (reflow) 与重绘 (repaint)

reflow:当render树中的一部分或者全部因为大小边距等问题发生改变而需要重建的过程叫做回流

repaint:当元素的一部分属性发生变化，如外观背景色不会引起布局变化而需要重新渲染的过程叫做重绘

## 何时发生回流重绘

我们前面知道了，回流这一阶段主要是计算节点的位置和几何信息，那么当页面布局和几何信息发生变化的时候，就需要回流。比如以下情况：

- 添加或删除可见的DOM元素
- 元素的位置发生变化
- 元素的尺寸发生变化（包括外边距、内边框、边框大小、高度和宽度等）
- 内容发生变化，比如文本变化或图片被另一个不同尺寸的图片所替代。
- 页面一开始渲染的时候（这肯定避免不了）
- 浏览器的窗口尺寸变化（因为回流是根据视口的大小来计算元素的位置和大小的）

注意：回流一定会触发重绘，而重绘不一定会回流

根据改变的范围和程度，渲染树中或大或小的部分需要重新计算，有些改变会触发整个页面的重排，比如，滚动条出现的时候或者修改了根节点。

## 浏览器的优化机制

现代的浏览器都是很聪明的，由于每次重排都会造成额外的计算消耗，因此大多数浏览器都会通过队列化修改并批量执行来优化重排过程。浏览器会将修改操作放入到队列里，直到过了一段时间或者操作达到了一个阈值，才清空队列。但是！当你获取布局信息的操作的时候，会强制队列刷新，比如当你访问以下属性或者使用以下方法：

- `offsetTop`、`offsetLeft`、`offsetWidth`、`offsetHeight`
- `scrollTop`、`scrollLeft`、`scrollWidth`、`scrollHeight`
- `clientTop`、`clientLeft`、`clientWidth`、`clientHeight`
- `getComputedStyle()`
- `getBoundingClientRect`
- 具体可以访问这个网站：<https://gist.github.com/paulirish/5d52fb081b3570c81e3a>

以上属性和方法都需要返回最新的布局信息，因此浏览器不得不清空队列，触发回流重绘来返回正确的值。因此，我们在修改样式的时候，最好避免使用上面列出的属性，他们都会刷新渲染队列。如果要使用它们，最好将值缓存起来。

## 如何减少回流

### (1) 最小化重绘和重排

例如：js修改某个元素的多个内联样式 不要一个一个改，使用`style.cssText`一次改好

注：浏览器会优化，影响不大

### (2) 批量操作dom时：

- 1 使元素脱离文档流（隐藏，或将原始元素拷贝到一个脱离文档的节点中修改）
- 2 对其进行多次修改
- 3 将元素带回到文档中。

注：浏览器会优化，影响不大

### (3) 复杂动画效果，使用绝对定位让其脱离文档流

### (4) 使用位置相关的属性时最好缓存起来，每当获取布局相关的属性时，浏览器会强制回流

# CSS

## 元素与盒

在HTML中常常使用的概念是元素，而在CSS中，布局的基本单位是盒，盒总是矩形的。

元素与盒并非一一对应的关系，一个元素可能生成多个盒，CSS规则中的伪元素也可能生成盒，`display`属性为`none`的元素则不生成盒。

除了元素之外，HTML中的文本节点也可能会生成盒。

### 盒模型

一个盒包括了内容(**content**)、边框(**border**)、内边距(**padding**)、外边距(**margin**)。

### **box-sizing:**

**content-box**:默认值，标准盒子模型。`width`与`height`只包括内容的宽和高，设置元素的宽度时，为内容区的宽度，任何边距和边框都会被增加到最后绘制出来的元素宽度中

设置`width = 内容的宽度`

**border-box** 告诉浏览器你设置的边框和内边距的值是包含在`width`内的。也就是说，如果你将一个元素的`width`设为`100px`,那么这`100px`会包含其它的`border`和`padding`，内容区的实际宽度会是`width`减去`border + padding`的计算值。

设置`width = border + padding + 内容`

## 已知宽高元素的居中实现

```
.element {
 width: 600px; height: 400px;
 position: absolute; left: 50%; top: 50%;
 margin-top: -200px; /* 高度的一半 */
 margin-left: -300px; /* 宽度的一半 */
}
```

## 2 css3 兼容有问题

```
1 .element {
2 width: 600px; height: 400px;
3 position: absolute; left: 50%; top: 50%;
4 transform: translate(-50%, -50%); /* 50%为自身尺寸的一半 */
5 }
```

## 3 margin:auto

```
1 .element {
2 width: 600px; height: 400px;
3 margin: auto; /* 有了这个就自动居中了 */
4 }
```

CSS 2 规范的 10 章 可视化布局：

如果有 width 那么 ml/mr 是 auto(其他值么), 那就是0, 后面的一条 if 不生效了。

如果 width 不是 auto, 那就是有确定宽度了。 ml/mr 是 auto, 那就居中。

<https://www.zhihu.com/question/21644198/answer/42702524>

## 4 flex

### flex容器

```
1 {
2 display: flex;
3 align-items: center;
4 justify-content: center
5 }
```

## css选择器优先级

第一等级：代表 内联样式，如 style=""，权值为 **1000**

第二等级：代表 **ID选择器**，如 #id="", 权值为 **100**

第三等级：代表 **calss | 伪类 | 属性** 选择器，如 .class | :hover,:link,:target | [type], 权值 **10**

第四等级：代表 **标签 | 伪元素** 选择器，如 p | ::after, ::before, ::fist-inline, ::selection, 权值 **1**

此外，通用选择器 (\*) , 子选择器 (>) , 相邻同胞选择器 (+) 等选择器不在4等级之内，所以它们的权值都为 0；

总结：

1.先从高等级进行比较，高等级相同时，再比较低等级的，以此类推；若两者最高优先级的等级不一样，无需计算权重值，直接取优先级高的

2.完全相同的话，就采用 后者优先原则（也就是样式覆盖）；

3.css属性后面加 !important 时，无条件绝对优先（比内联样式还要优先）；

# 布局

**静态布局：**全程使用px

**流式布局：**也叫百分比布局：把元素的宽,margin,padding不再用固定数值,改用百分比,可以在不同分辨率下显示相同的版式

**响应式布局：**为了适应不同的设备，采用 CSS 的 media query 技术,可以在不同分辨率下对元素重新设置样式（不只是尺寸），在不同屏幕下可以显示不同版式

## Flex布局

Flex是Flexible Box的缩写，意为“弹性布局”

采用Flex布局的元素，称为Flex容器（flex container），简称“容器”。它的所有子元素自动成为容器成员，称为Flex项目（flex item），简称“项目”。

使用：

```
display: flex/inline-flex;
```

设为Flex布局以后，子元素的**float**、**clear**和**vertical-align**属性将失效。

main axis 容器的主轴 cross axis 交叉轴

主轴的开始位置（与边框的交叉点）叫做main start，结束位置叫做main end；交叉轴的开始位置叫做cross start，结束位置叫做cross end。

项目默认沿主轴排列。单个项目占据的主轴空间叫做main size，占据的交叉轴空间叫做cross size。

### 容器的属性

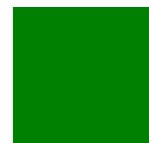
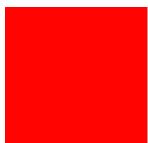
1 **flex-direction:** row | row-reverse | column | column-reverse; 容器的主轴方向即项目排列方向，

2 **flex-wrap:** 此属性定义如果一条轴线排不下如何换行, nowrap 不换行(默认) | wrap 换行 | wrap-reverse; 换行，第一行在下方

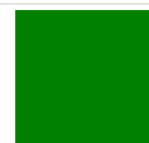
3 **flex-flow:** <flex-direction> || <flex-wrap>; 前两种属性的简写，默认值为 row nowrap

4 **justify-content** 定义项目在主轴的对齐方式

flex-start 主轴开始的地方（由flex-direction决定） | flex-end | center | space-between 两端对齐，项目之间的间隔相等



space-around 每个项目两侧的间隔相等；



5 **align-items:** flex-start | flex-end | center | baseline | stretch;

定义项目在交叉轴如何对齐，默认值为stretch,若项目没有设置高度，则高度盛满整个交叉轴，若有设置则在交叉轴起点对齐

baseline为以项目内的第一行文字为标准对齐，若项目内没有文字则在项目的后代元素中查找，若项目内没有文字，则该项目在交叉轴起点对齐

6 align-content: align-content属性定义了多根轴线(内容超过一行)的对齐方式。如果项目只有一根轴线，该属性不起作用。

- flex-start: 与交叉轴的起点对齐。
- flex-end: 与交叉轴的终点对齐。
- center: 与交叉轴的中点对齐。
- space-between: 与交叉轴两端对齐，轴线之间的间隔平均分布。
- space-around: 每根轴线两侧的间隔都相等。所以，轴线之间的间隔比轴线与边框的间隔大一倍。
- stretch (默认值) : 轴线占满整个交叉轴

align-content控制多行item在整个容器内的对齐方式(当容器交叉轴的长度大于所有行item交叉轴长度之和时)，align-items 控制该行行内的items对齐方式，

例如有两行items，首先把容器在交叉轴上分为两份

## 项目的属性：

1 order 决定项目在主轴上排序的顺序，数值越小越靠前

2 flex-grow 定义项目放大的比例，默认值为0，即有剩余空间也不放大，若不为0，则根据项目此项数值的比例来瓜分剩余空间，有两种情况：

若所有项目的flex-grow的和不大于1，则放大的值为 设置的数值 \* 剩余空间值，举一个具体的例子，若剩余空间1000px，某一个项目flex-grow为0.5，则该项目在主轴上的空间放大500px

若所有项目的flex-grow的和大于1，则放大的值为 (设置的数值/所有项目flex-grow的和) \* 剩余空间值，例：剩余空间1000px，某一个项目flex-grow为1，所有项目flex-grow的和为2，则该项目在主轴上的空间也放大500px

3 flex-shrink属性定义了项目的缩小比例，默认为1，即如果空间不足，该项目将缩小。一般在容器flex-wrap设为 nowrap不换行时，所有项目的宽度超过了容器的宽度就需要缩放

每个项目缩少的比例 为该项目flex-shrink/所有项目flex-shrink之和 \* 该项目在主轴上的空间

例如 容器宽度为2560px, 容器内有16项目每个200px的总宽度为3200px，且所有项目flex-shrink都为1，此时所有项目的宽度要缩少  $(2560/3200) * 200 = 40\text{px}$ .

如果情况稍微复杂一点，其中一个项目的flex-shrink不为1，设为5，那么此项目需要缩小  $(5/20) * 640 = 160\text{px}$ ，注意若需要缩小的值超过项目本身则按项目本身空间大小算

4 flex-basis属性定义了在分配多余空间之前，项目占据的主轴空间 (main size)。浏览器根据这个属性，计算主轴是否有多余空间。它的默认值为auto，即项目的本来大小，若设置了flex-basis，其优先级高于元素的本身的width/height

5 flex：flex属性是flex-grow, flex-shrink 和 flex-basis的简写，默认值为0 1 auto。后两个属性可选。

该属性有两个快捷值：auto (1 1 auto, 即为有剩余空间就放大，需要缩小时缩小) 和 none (0 0 auto, 无论什么情况既不放大，也不缩小)。

6 align-self: 允许单个项目有与其他项目不一样的对齐方式，可覆盖align-items属性。默认值为auto，表示继承父元素的align-items属性，如果没有父元素，则等同于stretch。

## 移动端适配

## BFC

### block formatting context 块级格式化上下文

**Formatting context** 是 W3C CSS2.1 规范中的一个概念。它是页面中的一块渲染区域，并且有一套渲染规则，它决定了其子元素将如何定位，以及和其他元素的关系和相互作用。最常见的 Formatting context 有 Block fomatting context (简称BFC)和 Inline formatting context (简称IFC)。

BFC(Block formatting context)直译为"块级格式化上下文"。它是一个独立的渲染区域，只有 Block-level box 参与，它规定了内部的Block-level Box如何布局，并且与这个区域外部毫不相干。

#### BFC布局规则：

1. 内部的Box会在垂直方向，一个接一个地放置。
2. Box垂直方向的距离由margin决定。属于同一个BFC的两个相邻Box的margin会发生重叠（让两个元素不属于同一个bfc以防止 margin 重叠）
3. 每个元素的margin box的左边，与包含块border box的左边相接触(对于从左往右的格式化，否则相反)。即使存在浮动也是如此。
4. BFC的区域不会与float box重叠。(可以以此实现三列布局)
5. 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。反之也如此。
6. 计算BFC的高度时，浮动元素也参与计算（清除浮动的原理）

#### 哪些元素会生成BFC？

1. 根元素
2. float属性不为none
3. position为absolute或fixed
4. display为inline-block, table-cell, table-caption, flex, inline-flex
5. overflow不为visible

核心概念：因为BFC内部的元素和外部的元素绝对不会互相影响，因此，当BFC外部存在浮动时，它不应该影响BFC内部Box的布局，BFC会通过变窄，而不与浮动有重叠。同样的，当BFC内部有浮动时，为了不影响外部元素的布局，BFC计算高度时会包括浮动的高度。避免margin重叠也是这样一个道理。

参考：<https://www.cnblogs.com/lhb25/p/inside-block-formatting-ontext.html>

## CSS3

### 边框：

## border-radius 圆角

- 四个值: 左上, 右上, 右下, 左下。
- 三个值: 第一个值为左上, 第二个值为右上和左下, 第三个值为右下
- 两个值: 第一个值为左上与右下, 第二个值为右上与左下
- 一个值: 四个角值相同

## border-shadow 盒阴影

|          |                          |
|----------|--------------------------|
| h-shadow | 必需。水平阴影的位置。允许负值。         |
| v-shadow | 必需。垂直阴影的位置。允许负值。         |
| blur     | 可选。模糊距离。                 |
| spread   | 可选。阴影的尺寸。                |
| color    | 可选。阴影的颜色。请参阅 CSS 颜色值。    |
| inset    | 可选。将外部阴影 (outset) 改为内部阴影 |

## 文本效果

### text-shadow 文字阴影

|          |                      |
|----------|----------------------|
| h-shadow | 必需。水平阴影的位置。允许负值。     |
| v-shadow | 必需。垂直阴影的位置。允许负值。     |
| blur     | 可选。模糊的距离。            |
| color    | 可选。阴影的颜色。参阅 CSS 颜色值。 |

**text-overflow** 属性规定当文本溢出包含元素时发生的事情。

|          |                    |
|----------|--------------------|
| clip     | 修剪文本。              |
| ellipsis | 显示省略符号来代表被修剪的文本。   |
| string   | 使用给定的字符串来代表被修剪的文本。 |

### word-wrap (overflow-wrap) 控制长度超过一行的单词是否被拆分换行

|            |            |
|------------|------------|
| break-word | 超过一行的单词会换行 |
|------------|------------|

**word-break** 规定自动换行的处理方法。

|           |                                                 |
|-----------|-------------------------------------------------|
| normal    | 使用浏览器默认的换行规则。                                   |
| break-all | 允许在单词内换行。                                       |
| keep-all  | 所有“单词”一律不拆分换行，注意，包括连续的中文字符，只能在半角空格或连字符处 (-) 换行。 |

**white-space**:控制空白字符的显示，同时还能控制是否自动换行。

|             |                               |
|-------------|-------------------------------|
| normal (默认) | 空白会被浏览器合并。换行符无效，可以换行          |
| pre         | 空白会保留。类似<pre> 标签。换行符有效，不会自动换行 |
| nowrap      | 空白合并，不会换行，直到遇到 <br> 标签为止。     |
| pre-wrap    | 空白会保留换行符有效，也会自动换行             |
| pre-line    | 空白合并，但是保留换行符。也会自动换行           |
| inherit     | 规定应该从父元素继承 white-space 属性的值。  |

## 单行文本溢出自动省略：

```
white-space: nowrap; // 不让换行
overflow:hidden; // 不让溢出
text-overfolw: ellipsis // 溢出省略号
```

## 2D 转换

```
transform:translate(x轴移动距离, y轴移动距离) 移动位置
transform:rotate(XXdeg) 旋转, 为负值则是逆时针旋转
rotateY()沿着Y轴旋转
transform:scale(2, 2) 水平和垂直方向扩大两倍
transform:skew(X轴倾斜角, Y轴倾斜角)
```

## 过渡 transition

|                                   |                         |
|-----------------------------------|-------------------------|
| <u>transition</u>                 | 简写属性, 用于在一个属性中设置四个过渡属性。 |
| <u>transition-property</u>        | 规定应用过渡的 CSS 属性的名称。      |
| <u>transition-duration</u>        | 定义过渡效果花费的时间。默认是 0。      |
| <u>transition-timing-function</u> | 规定过渡效果的时间曲线。默认是 "ease"。 |
| <u>transition-delay</u>           | 规定过渡效果何时开始。默认是 0。       |

## 动画 animation

```
1 @keyframes 关键帧的名字 // 定义关键帧
2 {
3 0% {}
4 50% {}
5 100% {}
6 }
```

|                                  |                                                                                                  |
|----------------------------------|--------------------------------------------------------------------------------------------------|
| <u>animation-name</u>            | 规定 @keyframes 动画的名称。                                                                             |
| <u>animation-duration</u>        | 规定动画完成一个周期所花费的秒或毫秒。默认是 0。                                                                        |
| <u>animation-timing-function</u> | 规定动画的速度曲线。默认是 "ease", 低速开始, 然后加快, 在结束前变慢。ease-in以低速开始越来越快; linear 匀速                             |
| <u>animation-delay</u>           | 规定动画何时开始。默认是 0, 立即开始。                                                                            |
| <u>animation-iteration-count</u> | 规定动画被播放的次数。默认是 1。值为infinite时无限次播放                                                                |
| <u>animation-direction</u>       | 规定动画是否在下一周期逆向地播放。默认是 "normal", reverse 为反向播放, alternate为奇数次正向, 偶数次反向播放, 如果动画被设置为只播放一次, 该属性将不起作用。 |
| <u>animation-play-state</u>      | 规定动画是否正在运行或暂停。默认是 "running"。paused为暂停                                                            |

## 多媒体查询 @media

**@media screen and(min-width:480px) and (max-width:960px)** 屏幕宽度大于480px 小于960px时

# img空白间隙问题：

造成原因：img是 inline-block 类型的元素，默认的垂直对齐方式vertical-align默认值是baseline(基线对齐)，即以x字母的下方为基准。



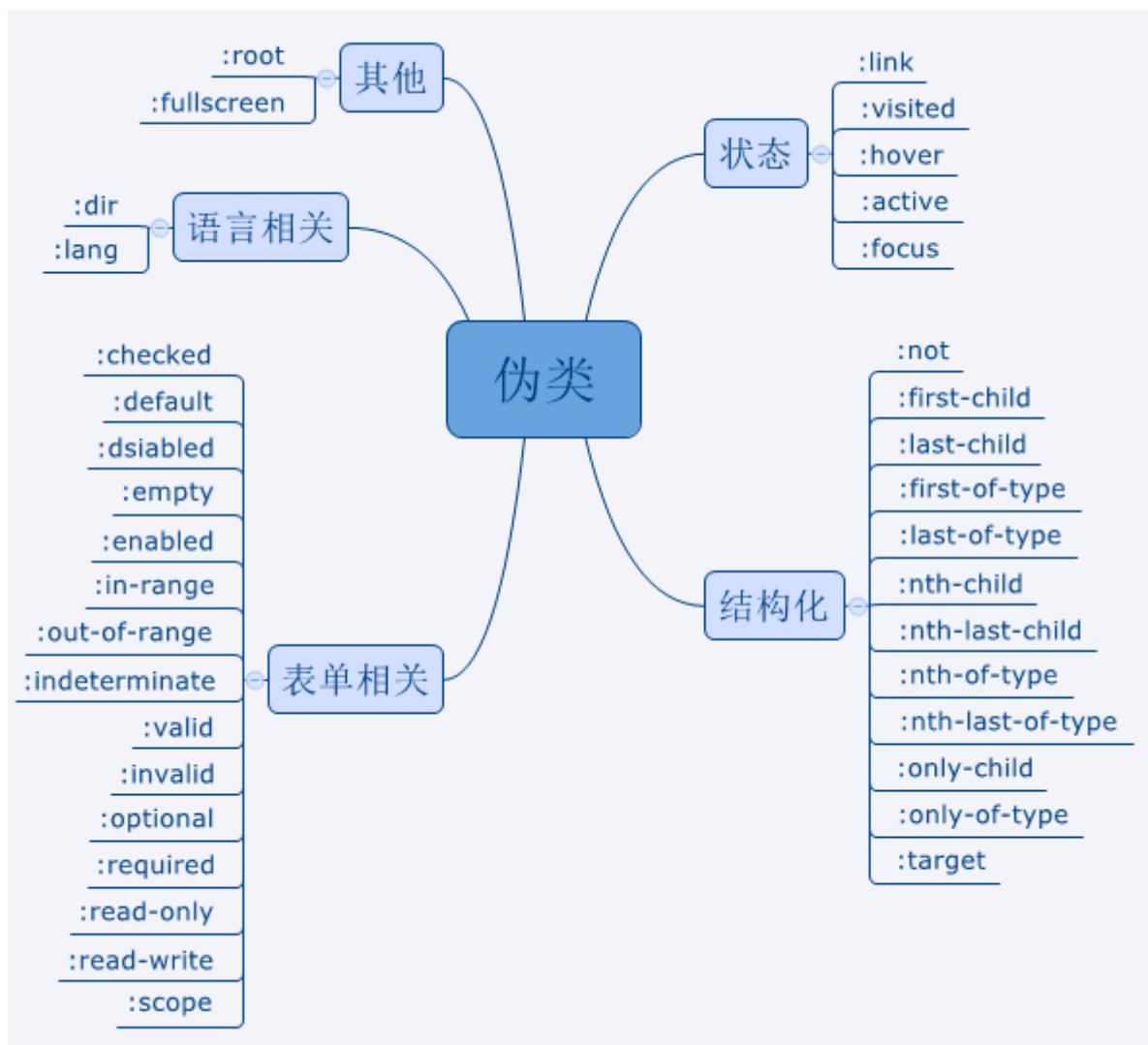
解决办法：

- 1 font-size: 0;
- 2 vertical-align: bottom;
- 3 line-height: 0;
- 4 display:block;

## 伪类和伪元素

伪类的操作对象是文档树中已有的元素，而伪元素则创建了一个文档树外的元素。因此，伪类与伪元素的区别在于：有没有创建一个文档树之外的元素。

伪类是基于元素的某种状态下（如:hover等）或者元素存在某种特性的時候（如:lang等）发挥其作用的。伪类根据动态状态来添加以实现样式的动态变化。其功能与类相似



**:nth-child(n)** :nth-child(n) 选择器匹配属于其父元素的第 N 个子元素，不论元素的类型。

例如 p:nth-child(2) 先匹配所有p元素的父元素的第2个子元素，然后判断是否是p元素，虽然定义中表明:nth-child(n)不论元素的类型，但是使用时这个伪类作用于p元素

**:nth-of-type(n)** 选择器匹配属于父元素的特定类型的第 N 个子元素的每个元素.

例如 p:nth-of-type(2) 匹配所有p元素的父元素的第二个类型为p标签的子元素，先匹配类型为p的子元素再选择第2个

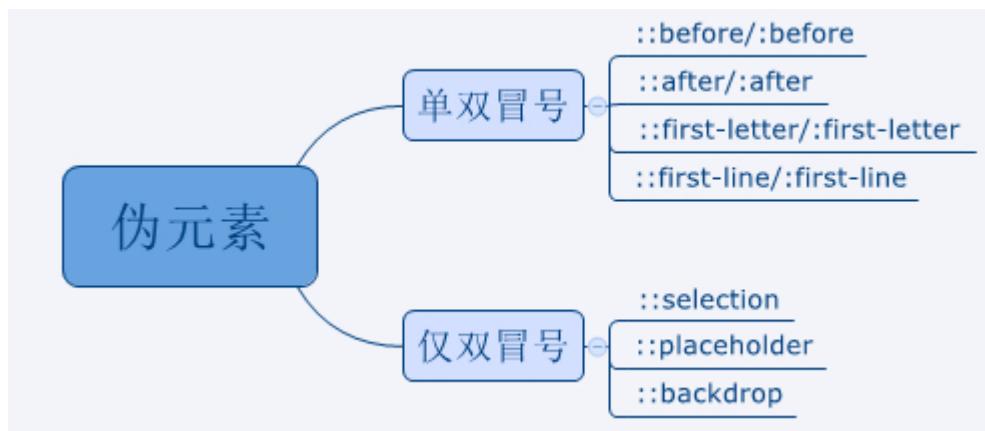
:nth-last-child(n) 同:nth-child(n) 不过从后往前匹配

例如 p:nth-last-child(2) 先匹配所有p元素的父元素的倒数第2个子元素，然后判断是否是p元素

:nth-last-of-type(n) 同:nth-child(n) 不过从后往前匹配

例如 p:nth-last-of-type(2) 匹配所有p元素的父元素的倒数第二个类型为p标签的子元素

伪元素可以对元素的特定内容进行操作（如:first-letter等）。类似像文本的第一个字母添加特殊样式这种效果，在普通选择器中是无法实现的（除非对文本中的第一个字母添加标签使其成为元素）。



# HTTP

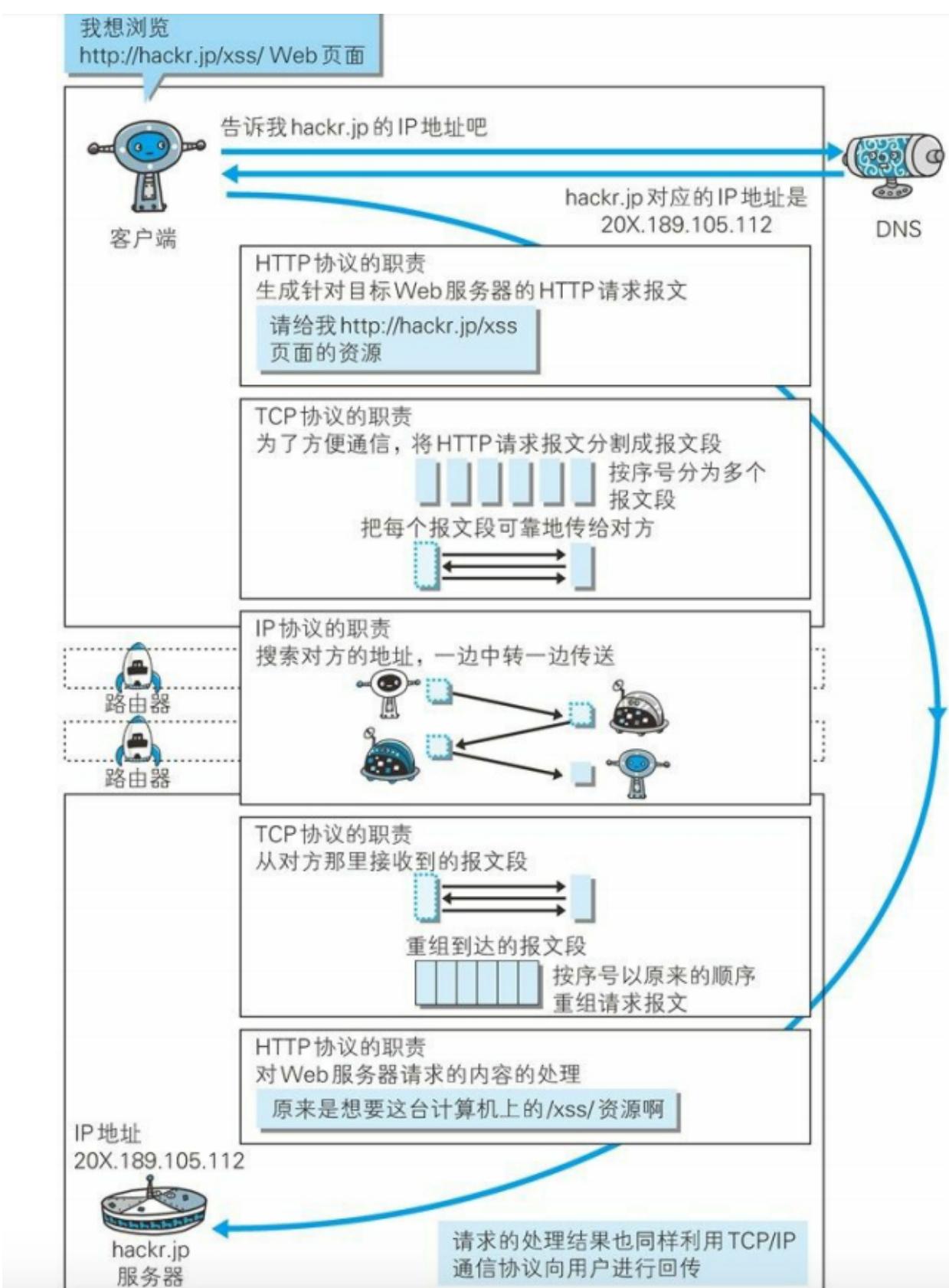
## 简介：

http的诞生于web,现已不限于web,它是tcp/ip协议的子集

TCP/IP是互联网相关协议的总称

TCP/IP分四层，分层的好处是协议变更时只需要更新单独的某层不需要更新全部

有人会把“IP”和“IP 地址”搞混，“IP”其实是一种协议的名称。



## OSI七层协议 / tcp/ip 四层协议（打🌟的）

🌟 **应用层**：决定了向用户提供应用服务时通信的活动。

TCP/IP 协议族内预存了各类通用的应用服务。比如，**FTP**（File Transfer Protocol，文件传输协议）和 **DNS**（Domain Name System，域名系统）服务就是其中两类。**HTTP** 协议也处于该层。

**DNS**（Domain Name System）提供域名到 IP 地址之间的解析服务。使用域名是因为与 IP 地址的一组纯数字相比，用字母配合数字的表示形式来指定计算机名更符合人类的记忆习惯。

### 表示层

这一层的主要功能是定义数据格式及加密。例如，FTP 允许你选择以二进制或 ASCII 格式传输。如果选择二进制，那么发送方和接收方不改变文件的内容。如果选择 ASCII 格式，发送方将把文本从发送方的字符集转换成标准的 ASCII 后发送

数据。在接收方将标准的ASCII转换成接收方计算机的字符集。示例：加密， ASCII等。

## 会话层

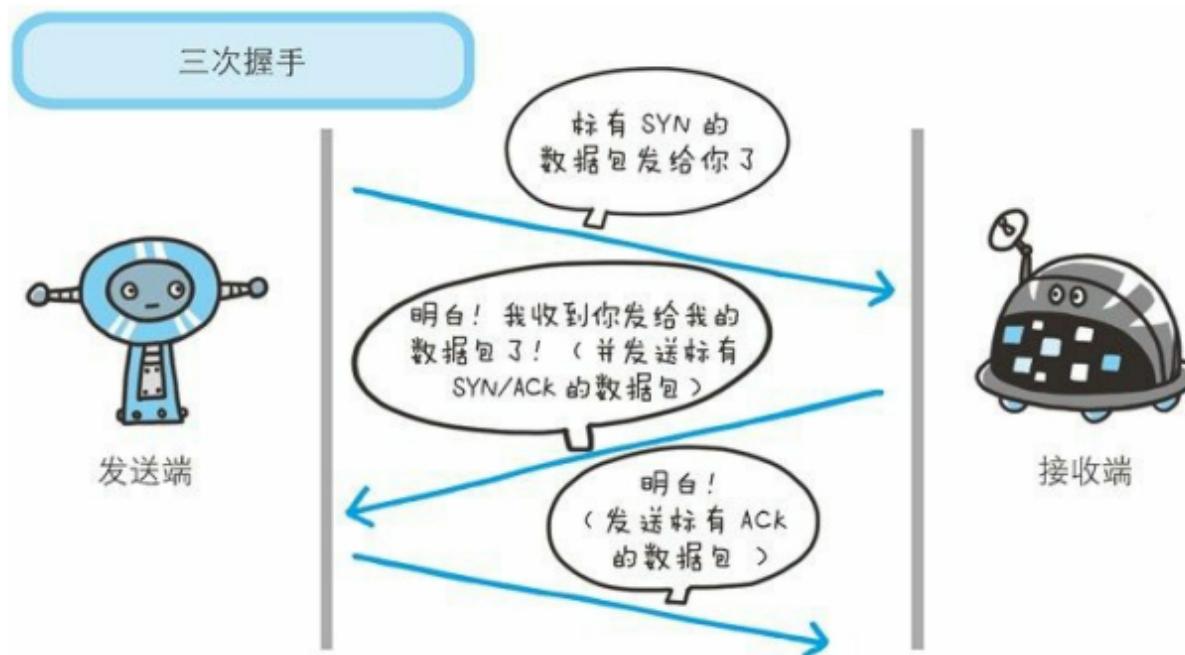
它定义了如何开始、控制和结束一个会话，包括对多个双向消息的控制和管理，以便在只完成连续消息的一部分时可以通知应用，从而使表示层看到的数据是连续的，在某些情况下，如果表示层收到了所有的[数据](#)，则用数据代表表示层。示例：RPC， SQL等。

★ **传输层** 提供处于网络连接中的两台计算机之间的数据传输。在传输层有两个性质不同的协议：

TCP (Transmission Control Protocol, 传输控制协议) 和 UDP (User Data Protocol, 用户数据报 协议) 。

TCP 位于传输层，为了方便传输，**将大块数据分割成以报文段 (segment) 为单位的数据包进行管理**。并且能确定数据是否送达，为了确保数据能够送达，TCP协议采用了**三次握手策略**：

发送端首先发送一个带 SYN 标志的数据包给对方。接收端收到后，回传一个带有 SYN/ACK 标志的数据包以示传达确认信息。最后，发送端再回传一个带 ACK 标志的数据包，代表“握手”结束。



★ **网络层** (又名网络互连层) 网络层用来处理在网络上流动的数据包。数据包是网络传输的最小数据单位。该层规定了通过怎样的路径（所谓的**传输路线**）到达对方计算机，并把数据包传送给对方。与对方计算机之间通过多台计算机或网络设备进行传输时，网络层所起的作用就是在众多的选项内选择一条**传输路线**。

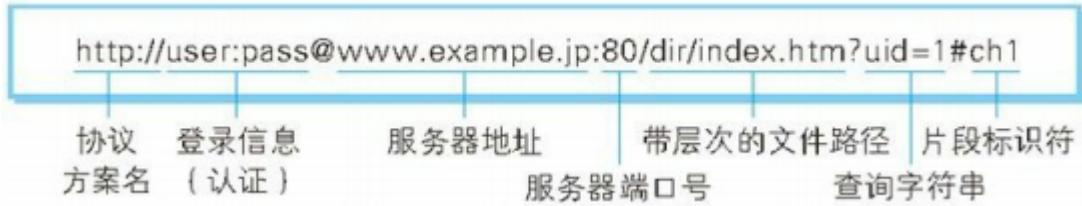
**IP** (Internet Protocol) 网际协议位于网络层。作用是把各种数据包传送给对方。而要保证确实传送到对方那里，其中两个重要的条件是 **IP 地址** 和 **MAC 地址** (Media Access Control Address) 。IP 地址指明了节点被分配到的地址，MAC 地址是指网卡所属的固定地址。IP 地址可以和 MAC 地址进行配对。IP 地址可变换，但 MAC 地址基本上不会更改。

IP 间的通信依赖 MAC 地址。在网络上，通信的双方在同一局域网 (LAN) 内的情况是很少的，通常是经过多台计算机和网络设备中转 才能连接到对方。而在进行中转时，会利用下一站中转设备的 MAC 地址来搜索下一个中转目标。这时，会采用 **ARP 协议 (Address Resolution Protocol)** 。ARP 是一种用以解析地址的协议，根据通信方的 IP 地址就可以反查出对应的 MAC 地址。在到达通信目标前的中转过程中，那些计算机和路由器等网络设备只能获悉很粗略的**传输路线**。这种机制称为**路由选择 (routing)**

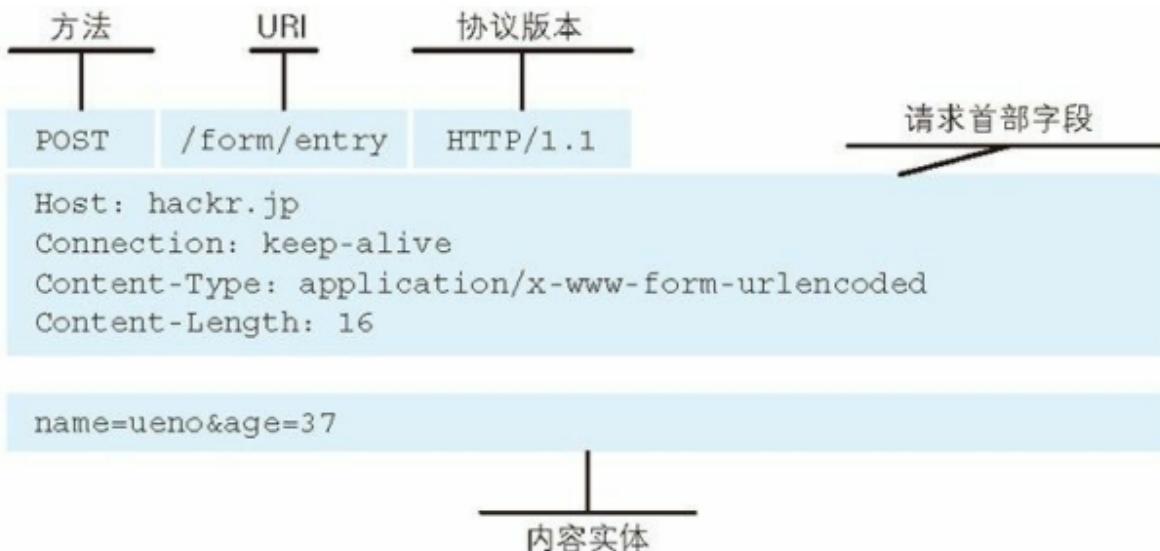
MAC 地址就像自己的 ID 号，而 IP 地址就像带着邮政编码的住址，各有各的用途。

★ **链路层** (又名数据链路层，网络接口层) 用来处理连接网络的硬件部分。包括控制操作系统、硬件的设备驱动、NIC (Network Interface Card, 网络适配器，即网卡)，及光纤等物理可见部分 (还包括连接器等一切传输媒介) 。硬件上的范畴均在链路层的作用范围之内。

URI 就是由某个协议方案表示的资源的定位标识符。协议方案是指访问资源所使用的协议类型名称。URL是 URI 的子集。



HTTP 协议规定，请求从客户端发出，最后服务器端响应该请求并返回。换句话说，肯定是先从客户端开始建立通信的，服务器端在没有接收到请求之前不会发送响应。



图：请求报文的构成



图：响应报文的构成

HTTP 是一种不保存状态，即无状态（stateless）协议。HTTP 协议自身不对请求和响应之间的通信状态进行保存。

无状态的问题：例如购物网站登陆后跳转到其他页面无法保持登陆状态

解决办法：引入 cookie

## HTTP请求方法：

GET：获取资源

POST：传输实体主体

PUT：传输文件

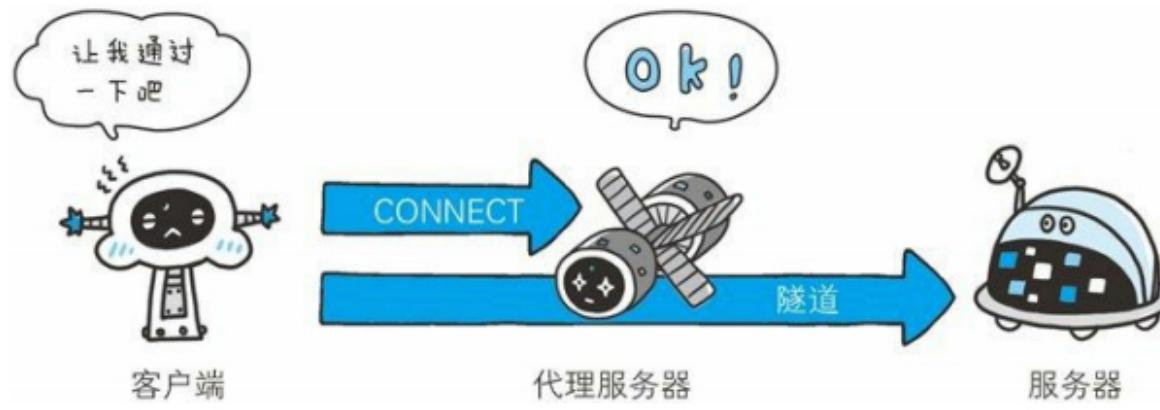
HEAD：获得报文首部 HEAD 方法和 GET 方法一样，只是不返回报文主体部分。用于确认 URI 的有效性及资源更新的日期时间等。

DELETE：删除文件

OPTIONS：询问支持的方法

TRACE：追踪路径 发送请求时，在 Max-Forwards 首部字段中填入数值，每经过一个服务器端就将该数字减 1，当数值刚好减到 0 时，就停止继续传输，最后接收到请求的服务器端则返回状态码 200 OK 的响应。客户端通过 TRACE 方法可以查询发送出去的请求是怎样被加工修改 / 篡改（例如代理）的。

CONNECT：要求用隧道协议连接代理 CONNECT 方法要求在与代理服务器通信时建立隧道，实现用隧道协议进行 TCP 通信。主要使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加密后经网络隧道传输。



在 HTTP/1.1 中，所有的连接默认都是持久连接，即保持了 TCP 连接状态

持久连接使得多数请求以管线化（pipelining）方式发送成为可能。即同时并行发送多个请求，而不需要一个接一个地等待响应了。

**cookie**：根据从服务器端发送的响应报文内的 **Set-Cookie** 的首部字段信息，通知客户端保存 Cookie。当下次客户端再往该服务器发送请求时，客户端会自动在请求报文中加入 Cookie 值后发送出去。服务器端发现客户端发送过来的 Cookie 后，会去检查究竟是从哪一个客户端发来的连接请求，然后对比服务器上的记录，最后得到之前的状态信息。

例子：

## 1. 请求报文（没有 **Cookie** 信息的状态）

```
GET /reader/ HTTP/1.1
Host: hackr.jp
*首部字段内没有Cookie的相关信息
```

## 2. 响应报文（服务器端生成 **Cookie** 信息）

```
HTTP/1.1 200 OK
Date: Thu, 12 Jul 2012 07:12:20 GMT
Server: Apache
<Set-Cookie: sid=1342077140226724; path=/; expires=Wed,
10-Oct-12 07:12:20 GMT>
Content-Type: text/plain; charset=UTF-8
```

## 3. 请求报文（自动发送保存着的 **Cookie** 信息）

```
GET /image/ HTTP/1.1
Host: hackr.jp
Cookie: sid=1342077140226724
```

## HTTP报文

用于 HTTP 协议交互的信息被称为 HTTP 报文。请求端（客户端）的 HTTP 报文叫做请求报文，响应端（服务器端）的叫做响应报文。

HTTP 报文本身是由多行（用 CR+LF 作换行符）数据构成的字符串文本。HTTP 报文大致可分为报文首部和报文主体两块。两者由最初出现的 空行（CR+LF）来划分。通常，并不一定有报文主体。

请求的报文首部第一行为请求行 包含用于请求的方法，请求 URI 和 HTTP 版本。 GET http://xxx.com HTTP/1.1

响应的报文首部第一行为状态行 包含表明响应结果的状态码，原因短语和 HTTP 版本。 HTTP/1.1 200 OK

不管是请求还是响应第一行以下的是首部字段，一般有 4 种首部，分别是：通用首部、请求首部、响应首部和实体首部。

## 发送多种数据的多部分对象集合：

multipart/form-data

```
Content-Type: multipart/form-data; boundary=AaB03x

--AaB03x
Content-Disposition: form-data; name="field1"

Joe Blow
--AaB03x
Content-Disposition: form-data; name="pics"; filename="file1.txt"
Content-Type: text/plain

... (file1.txt的数据) ...
--AaB03x--
```

使用 **boundary** 字符串来划分多部分对象集合指明的各类实体。在各个实体的起始行之前插入，例如上如插入的为 --AaB03x

多部分对象集合的每个部分类型中，都可以含有首部字段。

## 获取部分内容的范围请求：

执行范围请求时，会用到首部字段 Range 来指定资源的 byte 范围。

byte 范围的指定形式如下。

- **5001~10 000 字节**

Range: bytes=5001-10000

- **从 5001 字节之后全部的**

Range: bytes=5001-

- **从一开始到 3000 字节和 5000~7000 字节的多重范围**

Range: bytes=-3000, 5000-7000

## HTTP状态码

表示客户端 HTTP 请求的返回结果、标记服务器端的处理是否正常、通知出现的错误等工作。

204 no content 请求成功处理但没有资源可返回，一般在只需要从客户端往服务器发送信息，而对客户端不需要发送新信息内容的情况下使用。

206 表示客户端进行了范围请求，响应报文中包含由 Content-Range 指定范围的实体内容。

301 永久性重定向。该状态码表示请求的资源已被分配了新的 URI，以后应使用资源现在所指的 URI。

302 临时重定向

303 表示由于请求对应的资源存在着另一个 URI，应使用 GET 方法定向获取请求的资源。303 状态码**明确表示客户端应当采用 GET 方法获取资源**

当 301、302、303 响应状态码返回时，几乎所有的浏览器都会把 POST 改成 GET，并删除请求报文内的主体，之后请求会自动再次发送

304 表示客户端发送附带条件的请求时，服务器端允许请求访问资源，但未满足条件的情况(请求的文件未改变，命中协商缓存)。304 状态码返回时，不包含任何响应的主体部分。304 虽然被划分在 3XX 类别中，但是**和重定向没有关系**。

附带条件的请求是指采用 GET 方法的请求报文中包含 If-Match, If-Modified-Since, If-None-Match, If-Range, If-Unmodified-Since 中任一首部。

307 临时重定向。与 302 Found 有着相同的含义。尽管 302 标准禁止 POST 变换成 GET，但实际使用时大家并不遵守。307 会遵照浏览器标准，**不会从 POST 变成 GET**。但是，对于处理响应时的行为，每种浏览器有可能出现不同的情况。

400 请求报文中存在语法错误，服务器无法理解请求，一般为请求参数格式有误

401 请求需要有通过 HTTP 认证 (BASIC 认证、DIGEST 认证) 的认证信息(请求需要请求头 Authorization)。当浏览器初次接收到 401 响应，会弹出认证用的对话窗口。

403 请求资源的访问被服务器拒绝了。未获得文件系统的访问授权，访问权限出现某些问题 (从未授权的发送源 IP 地址试图访问) 等列举的情况都可能是发生 403 的原因。

404 服务器上无法找到请求的资源。

405 请求行中指定的请求方法不能被用于请求相应的资源。该响应必须返回一个 Allow 头信息用以表示出当前资源能够接受的请求方法的列表。

500 服务器端在执行请求时发生了错误。

503 服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。

504 作为网关或者代理服务器尝试执行请求时，未能及时从上游服务器（URI标识出的服务器，例如HTTP、FTP、LDAP）或者辅助服务器（例如DNS）收到响应。

## 通信数据转发程序

**代理：**是一种有转发功能的应用程序，它扮演了位于服务器和客户端“中间人”的角色，接收由客户端发送的请求并转发给服务器，同时也接收服务器返回的响应并转发给客户端。

**缓存代理：**代理转发响应时，缓存代理（Caching Proxy）会预先将资源的副本（缓存）保存在代理服务器上。当代理再次接收到对相同资源的请求时，就可以不从源服务器那里获取资源，而是将之前缓存的资源作为响应返回。

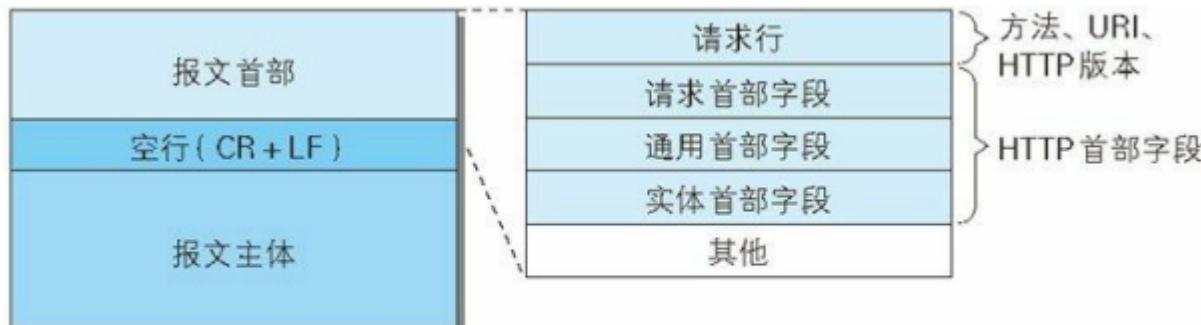
利用缓存可减少对源服务器的访问，因此也就节省了通信流量和通信时间。

**透明代理：**转发请求或响应时，不对报文做任何加工的代理类型被称为透明代理（Transparent Proxy）。反之，对报文内容进行加工的代理被称为非透明代理。

**网关：**网关区分了一个网络的内部和外部。如果一台电脑需要访问网络外的其他电脑，那么就需要配置网关来获得访问网络外部的权限。如果没有网关，电脑就无法访问局域网之外的网络部分，就像是被锁在家里一样。网关的工作机制和代理十分相似。而网关能使通信线路上的服务器提供非HTTP协议服务。

**隧道：**可按要求建立起一条与其他服务器的通信线路，届时使用SSL等加密手段进行通信。隧道的目的是确保客户端能与服务器进行安全的通信。

## HTTP报文首部



图：请求报文



图：响应报文

## HTTP首部字段

无论是请求还是响应都会使用首部字段，它能起到传递额外重要信息的作用。使用首部字段是为了给浏览器和服务器提供报文主体大小、所使用的语言、认证信息等内容。

结构： 首部字段名: 字段值

## 通用首部字段

### Cache-Control:

private: 内容只缓存到私有缓存中(仅客户端可以缓存，代理服务器不可缓存)

no-cache: 目的是为了防止从缓存中返回过期的资源。叫do-not-serve-from-cache-without-revalidation 更合适。

客户端包含此请求头则不会向服务端发送 (**if-none-match**) 以确认缓存是否有效，直接要新的文件

服务器返回的响应头若包含。则客户端之后再次访问此资源将不会直接接收缓存过的响应，即必须先与服务器确认(使用ETAG协商缓存) 返回的响应是否被更改。

no-cache=Location 由服务器返回的响应中，若报文首部字段 Cache-Control 中对 no-cache 字段名具体指定参数值，那么客户端在接收到这个被指定参数值的首部字段对应的响应报文后，就不能使用缓存。

no-store: 所有内容都不会被缓存

max-age: 如果判定缓存资源的缓存时间数值比指定时间的数值更小，那么客户端就接收缓存的资源。

当客户端指定 max-age 值为 0，则先用指纹去服务端校验一下是否有效，有效则用缓存

当服务端指定 max-age 为0，则同no-cache

### Connection:

1 可以控制不再转发给代理的首部字段 例如 connection: Upgrade，代理服务器转发时就会去掉 Upgrade首部字段

2 管理持久连接：

HTTP/1.1 版本的默认连接都是持久连接。当服务器端想明确断开连接时，则指定 Connection 首部字段的值为 Close。

**HTTP/1.1之前**默认非持久连接则需要指定 **Connection:Keep-Alive**

**Date:** 表明创建 HTTP 报文的日期和时间。1.1规定的时间格式：Date: Tue, 03 Jul 2012 04:40:59 GMT

**Pragma:** HTTP/1.1 之前版本的历史遗留字段，仅作为与 HTTP/1.0 的向后兼容而定义。在请求中，**假如 Cache-Control** 不存在的话，它的行为与 **Cache-Control: no-cache** 一致：强制要求缓存服务器在返回缓存的版本之前将请求提交到源头服务器进行验证。

**Upgrade:** 用于检测 HTTP 协议及其他协议是否可使用更高的版本进行通信，其参数值可以用来指定一个完全不同的通信协议。对于附有首部字段 Upgrade 的请求，服务器可用 **101 Switching Protocols** 状态码作为响应返回。

**Via:** 为了追踪客户端与服务器之间的请求和响应报文的传输路径。报文经过代理或网关时，会先在首部字段 Via 中附加该服务器的信息，然后再进行转发。还可避免请求回环的发生

**Warning:** 告知用户一些与缓存相关的问题的警告。

## 请求首部字段

从客户端向服务器端发送请求报文时使用的首部。补充了请求的附加 内容、客户端信息、响应内容相关优先级等信息。

**Accept:** 通知服务器，客户端能够处理的媒体类型及媒体类型的相对优先级。可使用 type/subtype 这种形式，一次指定多种媒体类型。

- 文本文件

text/html, text/plain, text/css ...

application/xhtml+xml, application/xml ...

- 图片文件

image/jpeg, image/gif, image/png ...

- 视频文件

video/mpeg, video/quicktime ...

- 应用程序使用的二进制文件

application/octet-stream, application/zip ...

若想要给显示的媒体类型增加优先级，则使用 q= 来额外表示权重值 1，用分号 (;) 进行分隔。权重值 q 的范围是 0~1（可精确到小数点后 3 位），且 1 为最大值。不指定权重 q 值时，默认权重为 q=1.0。

**Accept-Charset:** 用来通知服务器用户代理（浏览器）支持的字符集及字符集的相对优先顺序。可一次性指定多种字符集。例:Accept-Charset: iso-8859-5, unicode-1-1;q=0.8

**Accept-Encoding:** 用来告知服务器用户代理（浏览器）支持的内容编码及内容编码的优先级顺序。可一次性指定多种内容编码。可使用星号 (\*) 作为通配符，指定任意的编码格式。

- **gzip**

由文件压缩程序 gzip (GNU zip) 生成的编码格式  
(RFC1952)，采用 Lempel-Ziv 算法 (LZ77) 及 32 位循环冗余校验 (Cyclic Redundancy Check，通称 CRC)。

- **compress**

由 UNIX 文件压缩程序 compress 生成的编码格式，采用 Lempel-Ziv-Welch 算法 (LZW)。

- **deflate**

组合使用 zlib 格式 (RFC1950) 及由 deflate 压缩算法 (RFC1951) 生成的编码格式。

- **identity**

不执行压缩或不会变化的默认编码格式

**Accept-Language:** 用来告知服务器用户代理（浏览器）能够处理的语言（指中文或英文等），以及相对优先级。可一次指定多种自然语言集。

**Authorization:** 用来告知服务器用户代理（浏览器）的认证信息（证书值）。

**Expect:** 告知服务器期望出现的某种特定行为。若服务器无法理解客户端的期望作出回应而发生错误时，会返回状态码 417 Expectation Failed。HTTP/1.1 规范只定义了 100-continue（状态码 100 Continue 之意）。

**From:** 用来告知服务器用户的电子邮件地址。

**Host:** 告知服务器，请求的资源所处的互联网主机名和端口号。唯一一个必须被包含在请求内的首部字段。因为请求被发送至服务器时，请求中的主机名会用 IP 地址替换，如果此 IP 下部署运行着多个域名，那么服务器就会无法理解究竟是哪个域名对应的请求。

**If-Match:** 条件请求，在请求方法为 GET 和 HEAD 的情况下，服务器仅在请求的资源满足此首部列出的 ETag 之一时才会返回资源。而对于 PUT 或其他非安全方法来说，只有在满足条件的情况下才可以将资源上传。

**If-Modified-Since:** 会发送上一次请求 Last-Modified 的值，如果请求的资源都没有过更新，则返回状态码 304 Not Modified 的响应（协商缓存命中）。

**If-None-Match:** 作用与 If-Modified-Since 相似，不过使用 ETag 作为标识

**If-Range:** 配合 Range 使用，告知服务器若指定的 If-Range 字段值（ETag 值或者时间）和请求资源的 ETag 值或时间相一致时，则作为范围请求处理。反之，则返回全体资源。

**If-Unmodified-Since:** 条件请求，指定的请求资源只有在过期并且未发生更新的情况下，才能处理请求。如果在指定日期时间后发生了更新，则以状态码 412 Precondition Failed 作为响应返回。

**Max-Forwards:** 使用 HTTP 协议通信时，请求可能会经过代理等多台服务器。途中，如果代理服务器由于某些原因导致请求转发失败，客户端也就等不到服务器返回的响应了。对此，我们无从可知。可以灵活使用首部字段 Max-Forwards，针对以上问题产生的原因展开调查。配合请求方法 trace 使用

**Range:** 告知服务器资源的指定范围。Range: bytes=5001-10000

**Referer:** 告知服务器请求的 URI 是从哪个页面发起的。

**TE:** 告知服务器客户端能够处理响应的传输编码方式及相对优先级。它和首部字段 Accept-Encoding 的功能很相像，但是用于传输编码。只在两个节点间有效

**User-Agent:** User-Agent 会将浏览器类型及版本、操作系统及版本、浏览器内核、等信息的标识传达给服务器。

## 响应部首字段

从服务器端向客户端返回响应报文时使用的首部。补充了响应的附加内容，也会要求客户端附加额外的内容信息。

**Accept-Ranges:** 告知客户端服务器是否能处理范围请求，以指定获取服务器端某个部分的资源。可指定的字段值有两种，可处理范围请求时指定其为 **bytes**，反之则指定其为 **none**。

**Age:** 告知客户端，源服务器在多久前创建了响应。单位为秒。若创建该响应的服务器是缓存服务器，Age 值是指这个缓存向源服务器确认的时间 -> 现在的时间的差值。

**ETag:** 告知客户端实体标识。它是一种可将资源以字符串形式做唯一性标识的方式。服务器会为每份资源分配对应的 ETag 值。另外，当资源更新时，ETag 值也需要更新。生成 ETag 值时，并没有统一的算法规则，而仅仅是由服务器来分配。

强 ETag 值：不论实体发生多么细微的变化都会改变其值。

弱 Tag 值：只用于提示资源是否相同。只有资源发生了根本改变，产生差异时才会改变 ETag 值。这时，会在字段值最开始处附加 W/。

**Location:** 可以将响应接收方引导至某个与请求 URI 位置不同的资源。基本上，该字段会配合 3xx：Redirection 的响应，提供重定向的 URI。几乎所有的浏览器在接收到包含首部字段 Location 的响应

后，都会强制性地尝试对已提示的重定向资源的访问。

**Retry-After:** 告知客户端应该在多久之后再次发送请求。主要配合状态码 503 Service Unavailable 响应，或 3xx Redirect 响应一起使用。字段值可以指定为具体的日期时间（Wed, 04 Jul 2012 06: 34: 24 GMT 等格式），也可以是创建响应后的秒数。

**Server:** 告知客户端当前服务器上安装的 HTTP 服务器应用程序的信息。不单单会标出服务器上的软件应用名称，还有可能包括版本号和安装时启用的可选项。

**Vary:** 它决定了对于未来的一个请求头，应该用一个缓存的回复(response)还是向源服务器请求一个新的回复。它被服务器用来表明在内容协商算法中，应该使用哪些头部信息作为参考

例：Vary: User-Agent pc端和移动端内容不同，使用此响应头，未来客户端再请求时，若user-agent不同则内容协商算法认为需要新的回复，防止使用了错误的缓存

## Set-Cookie：详情见下文

### 实体首部字段

补充了资源内容更新时间等与实体有关的信息。

**Allow:** 用于通知客户端能够支持指定资源的所有 HTTP 方法。当服务器接收到不支持的 HTTP 方法时，会以状态码 405 Method Not Allowed 作为响应返回。与此同时，还会把所有能支持的 HTTP 方法写入首部字段 Allow 后返回。

**Content-Encoding:** 告知客户端服务器对实体的主体部分选用的内容编码方式。内容编码是指在不丢失实体信息的前提下所进行的压缩。

**Content-Language:** 告知客户端，实体主体使用的自然语言

**Content-Length:** 表明了实体主体部分的大小（单位是字节）。对实体主体进行内容编码传输时，不能再使用 Content-Length 首部字段。

**Content-Location:** 给出与报文主体部分相对应的 URI

**Content-MD5:** 对报文主体执行 MD5 算法获得的 128 位二进制数，再通过 Base64 编码后将结果写入 Content-MD5 字段值。由于 HTTP 首部无法记录二进制值，所以要通过 Base64 编码处理。为确保报文的有效性，作为接收方的客户端会对报文主体再执行一次相同的 MD5 算法。计算出的值与字段值作比较后，即可判断出报文主体的准确性。

**Content-Range:** 告知客户端作为响应返回的实体的哪个部分符合范围请求。字段值以字节为单位，表示当前发送部分及整个实体大小。

**Content-Type:** 说明了实体主体内对象的媒体类型。和首部字段 Accept 一样，字段值用 type/subtype 形式赋值。

**Expires:** 将资源失效的日期告知客户端。（绝对时间）

**Last-Modified:** 指明资源最终修改的时间（绝对时间）

## 为 Cookie 服务的首部字段

Cookie的工作机制是用户识别及状态管理。Web 网站为了管理用户的 状态会通过 Web 浏览器，把一些数据临时写入用户的计算机内。

### Set-Cookie:

| 属性           | 说明                                           |
|--------------|----------------------------------------------|
| NAME=VALUE   | 赋予 Cookie 的名称和其值（必需项）                        |
| expires=DATE | Cookie 的有效期（若不明确指定则默认为浏览器关闭前为止）              |
| path=PATH    | 将服务器上的文件目录作为Cookie的适用对象（若不指定则默认为文档所在的文件目录）   |
| domain=域名    | 作为 Cookie 适用对象的域名（若不指定则默认为创建 Cookie 的服务器的域名） |
| Secure       | 仅在 HTTPS 安全通信时才会发送 Cookie                    |
| HttpOnly     | 加以限制，使 Cookie 不能被 JavaScript 脚本访问            |

**expires:** 指定浏览器可发送 Cookie 的有效期。当省略 expires 属性时，其有效期仅限于维持浏览器会话(Session) 时间段内。这通常限于浏览器应用程序被关闭之前。

**Max-Age:** 在 cookie 失效之前需要经过的秒数。一位或多位非零 (1-9) 数字。一些老的浏览器 (ie6、ie7 和 ie8) 不支持这个属性。对于其他浏览器来说，假如二者 (指 Expires 和 Max-Age) 均存在，那么 Max-Age 优先级更高。

**path:** 可用于限制指定 Cookie 的发送范围的文件目录。字符 %x2F ("/") 可以解释为文件目录分隔符，此目录的下级目录也满足匹配的条件（例如，如果 path=/docs，那么 "/docs", "/docs/Web/" 或者 "/docs/Web/HTTP" 都满足匹配的条件）。

**domain:** 指定 cookie 可以送达的主机名。默认值为**当前文档所在的地址**

**secure:** 用于限制 Web 页面仅在 HTTPS 安全连接时，才可以发送 Cookie。

**HttpOnly:** 使 JavaScript 脚本 无法获得 Cookie。其主要目的为防止跨站脚本攻击 (Cross-site scripting, XSS) 对 Cookie 的信息窃取。

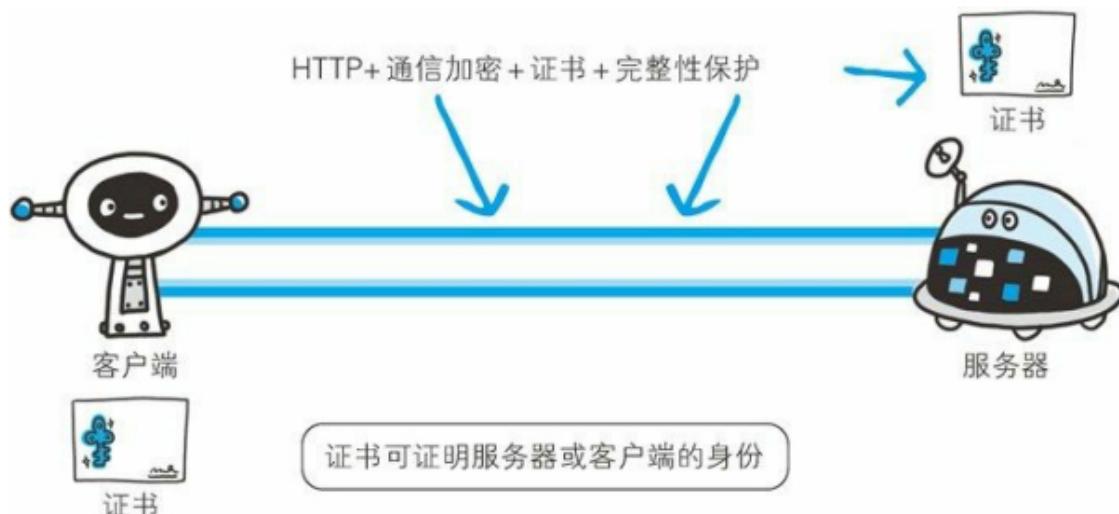
**Cookie:** 首部字段 Cookie 会告知服务器，当客户端想获得 HTTP 状态管理支持时，就会在请求中包含从服务器接收到的 Cookie。接收到多个 Cookie 时，同样可以以多个 Cookie 形式发送。

## HTTPS

**https=HTTP+ 加密 + 认证 + 完整性保护:**

http缺点：

- 1 通信使用明文（不加密），内容可能会被窃听
- 2 不验证通信方的身份，因此有可能遭遇伪装
- 3 无法证明报文的完整性，所以有可能已遭篡改



图：使用 HTTPS 通信

通常，HTTP 直接和 TCP 通信。当添加 SSL(Secure Sockets Layer 安全套接层，在应用层与传输层之间)时，则演变成先和 SSL 通信，再由 SSL 和 TCP 通信了。简言之，所谓 HTTPS，其实就是身披 SSL 协议这层外壳的 HTTP。

默认端口 http:80 https:443

**加密技术：**

1 共享密钥加密的困境：加密和解密同用一个密钥的方式称为**共享密钥加密** (Common key crypto system)，也被叫做**对称密钥加密**。

问题：无法确保安全的传送密钥，一旦密钥落入攻击者之手那么加密就毫无作用了

2 使用两把密钥的公开密钥加密：**公开密钥加密**使用一对非对称的密钥。一把叫做**私有密钥** (private key)，另一把叫做**公开密钥** (public key)。发送密文的一方使用对方的公开密钥进行加密处理，对方收到被加密的信息后，再使

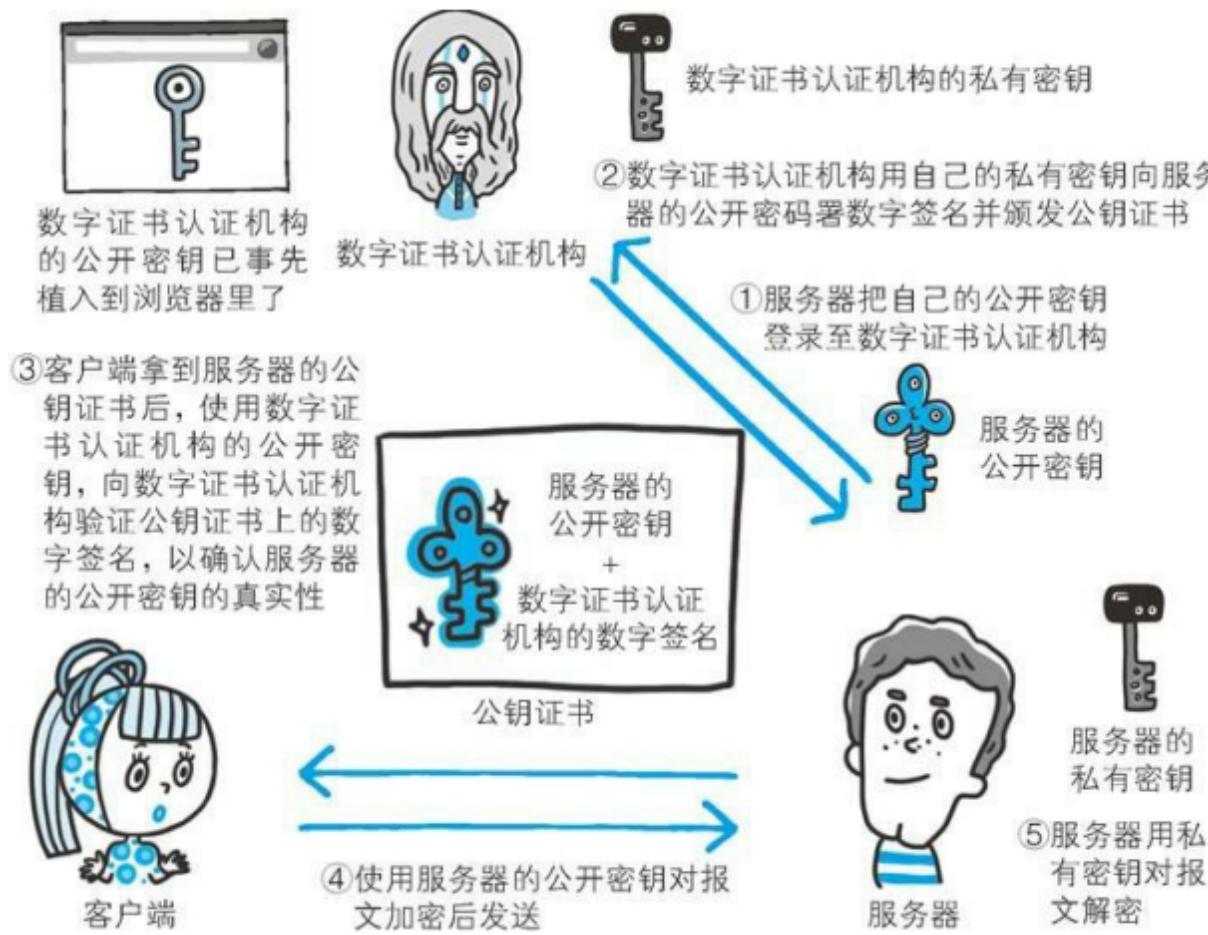
用自己的私有密钥 进行解密。

问题：与共享密钥加密相比，其处理速度要慢。

**HTTPS的加密：**使用公开密钥加密方式交换共享密钥，之后的建立通信交换报文阶段则使用共享密钥加密方式。

仍然有问题：无法证明公开密钥本身就是货真价实的公开密钥。或许在公开密钥传输途中，真正的公开密钥已经被攻击者替换掉了。

解决方法：使用由数字证书认证机构（CA, Certificate Authority）和其相关机关颁发的公开密钥证书



1 客户端请求，服务器首先会弹出一个页面提醒安装数字证书，包含着CA的公钥

2 然后服务端返回数字证书认证机构颁发的公钥证书

公钥证书包含着 1 服务器提供的非对称密钥中的公钥 2 颁发者信息 3 公钥hash算法 4 数字签名：公钥先使用hash算法得到hash再通过CA的私钥加密得到数字签名

3 客户端根据之前得到的CA公钥验证数字签名,得到hash,再使用证书内的hash算法自己算一遍看看是否相等，相等说明认证通过

4 客户端将 对称密钥 通过 验证过的公钥加密传送给服务端，服务端通过私钥解密得到共享密钥

5 客户端发送报文，报文使用对称密钥加密，服务端收到报文使用对称密钥解密

为什么不一直使用 HTTPS?

与纯文本通信相比，加密通信会消耗更多的 CPU 及内存资源。如果每次通信都加密，**1** 会消耗相当多的资源，平摊到一台计算机上时，能够处理的请求数量必定也会随之减少。仅在那些需要信息隐藏时才会加密，以节约资源。除此之外，**2** 想要节约购买证书的开销也是原因之一。

## HTTP/1.1 使用的认证方式：

**BASIC 认证（基本认证）**



客户端

## 发送请求

```
GET /private/ HTTP/1.1
Host: hackr.jp
```

- ①返回状态码 401 以告知客户端需要进行认证

```
HTTP/1.1 401 Authorization Required
Date: Mon, 19 Sep 2011 08:38:32 GMT
Server: Apache/2.2.3 (Unix)
WWW-Authenticate: Basic realm="Input Your ID and Password."
```



服务器



- ②用户 ID 和密码以 Base64 方式编码后发送  
guest:guest → Base64 → Z3Vlc3Q6Z3Vlc3Q=

```
GET /private/ HTTP/1.1
Host: hackr.jp
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
```



- ③认证成功者返回状态码 200，若认证失败则返回状态码 401

```
HTTP/1.1 200 OK
Date: Mon, 19 Sep 2011 08:38:35 GMT
Server: Apache/2.2.3 (Unix)
```

BASIC 认证虽然采用 Base64 编码方式，但这不是加密处理。不需要任何附加信息即可对其解码。换言之，由于明文解码后就是用户 ID 和密码，在 HTTP 等非加密通信的线路上进行 BASIC 认证的过程中，如果被人窃听，被盗的可能性极高。

## DIGEST 认证（摘要认证）



客户端

## 发送请求

```
GET /digest/ HTTP/1.1
Host: hackr.jp
```

- ①发送临时的质询码( 随机数， nonce )以及告知需要认证的状态码 401

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Digest realm="DIGEST",
nonce="MOSQZ0itBAA=44abb6784cc9cbfc605a5b0893d36f23de
95fcff", algorithm=MD5, qop="auth"
```



服务器



- ②发送摘要以及由质询码计算出的响应码( response )

```
GET /digest/ HTTP/1.1
Host: hackr.jp
Authorization: Digest username="guest", realm="DIGEST",
nonce="MOSQZ0itBAA=44abb6784cc9cbfc605a5b0893d36f23de95f
cff", uri="/digest/", algorithm=MD5,
response="df56389ba3f7c52e9d7551115d67472f", qop=auth,
nc=00000001, cnonce="082c875dcb2ca740"
```



- ③认证成功返回状态码 200，失败则再次发送状态码 401

```
HTTP/1.1 200 OK
Authentication-Info:
rspauth="f218e9ddb407a3d16f2f7d2c4097e900",
cnonce="082c875dcb2ca740", nc=00000001, qop=auth
```

## SSL 客户端认证

从使用用户 ID 和密码的认证方式方面来讲，只要二者的内容正确，即可认证是本人的行为。但如果用户 ID 和密码被盗，就很有可能被第三者冒充。利用 SSL 客户端认证则可以避免该情况的发生。

步骤 1：接收到需要认证资源的请求，服务器会发送 Certificate Request 报文，要求客户端提供客户端证书。

步骤 2：用户选择将发送的客户端证书后，客户端会把客户端证书信息以 Client Certificate 报文方式发送给服务器。

步骤 3：服务器验证客户端证书验证通过后方可领取证书内客户端的公开密钥，然后开始 HTTPS 加密通信。

## FormBase 认证（基于表单认证）

基于表单认证的标准规范尚未有定论，一般会使用 Cookie 来管理 Session（会话）。



## web安全

### XSS

跨站脚本 (cross site script) 为了避免与样式css混淆，所以简称为XSS。

XSS是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些代码，嵌入到web页面中去。使别的用户访问都会执行相应的嵌入代码。

从而盗取用户资料（cookie）、利用用户身份进行某种动作(XSS 是实现 CSRF 的诸多途径中的一条)

XSS攻击的分类：

#### 1 反射型

又称为非持久性跨站点脚本攻击，它是最常见的类型的XSS。漏洞产生的原因是攻击者注入的数据反映在响应中。

例：某网站会把url的参数没有转移过滤直接innerHTML显示在页面中，将参数的值改为一段script标签<script>xx</script>,然后给别人发送改后的url，一旦对方打开就会执行脚本，攻击成功

#### 2 存贮型xss攻击

又称为持久型跨站点脚本，它一般发生在XSS攻击代码存储在网站数据库，每当用户打开这个页面,攻击脚本就会执行。

例：网站留言没有过滤，把一段攻击代码作为留言，一旦用户打开这个网站的页面加载评论时脚本攻击就会执行

XSS防范方法

1 服务端以及客户端都将HTML 中的预留字符过滤替换为字符实体。(htmlEncode)

| 显示结果 | 描述             | 实体名称           |
|------|----------------|----------------|
|      | 空格             | &nbsp;         |
| <    | 小于号            | &lt;           |
| >    | 大于号            | &gt;           |
| &    | 和号             | &amp;          |
| "    | 引号             | &quot;         |
| '    | 撇号             | &apos; (IE不支持) |
| ¢    | 分 (cent)       | &cent;         |
| £    | 镑 (pound)      | &pound;        |
| ¥    | 元 (yen)        | &yen;          |
| €    | 欧元 (euro)      | &euro;         |
| §    | 小节             | &sect;         |
| ©    | 版权 (copyright) | &copy;         |
| ®    | 注册商标           | &reg;          |
| ™    | 商标             | &trade;        |
| ×    | 乘号             | &times;        |
| ÷    | 除号             | &divide;       |

2、将重要的cookie标记为http only

3、表单数据规定值的类型，例如：年龄应为只能为int、name只能为字母数字组合。。。

## CSRF (Cross Site Request Forgery, 跨站请求伪造)

攻击者盗用了你的身份（服务端仅使用cookie作为验证身份的方法），以你的名义发送恶意请求。

要完成一次CSRF攻击，受害者必须依次完成两个步骤：

登录受信任网站A，并在本地生成Cookie。

在不登出A（并且cookie有效期未过）的情况下，访问危险网站B。

1 通过 XSS 来实现 CSRF 易如反掌

2 登录受信任网站A，并在本地生成Cookie。

在不登出A的情况下，访问危险网站B。网站B通过还未失效的cookie 伪造用户发出请求（可能是img加载发送get请求，可能是通过form发送post请求），完成（转账，删除）操作

**CSRF的防御: 过滤请求的处理者。**

1 http首部字段referer: 服务端验证请求头中的referer是否在白名单中即可拦截使用自己制作的网站发起的csrf

缺点：1 某些浏览器例如ie6可以篡改referer 2 用户可以在浏览器中设置不提供referer

2 添加 token 并验证：例如登陆后返回一个token,之后提交表单时需要加上这个token参数作为校验（也可以放在自定义请求头中）

3 验证码 或者 二次验证

4 对数据库的更改操作只允许用post请求，可以防止攻击者通过生成一个img标签、点击一个链接 直接完成攻击

## sql注入原理

就是通过把SQL命令插入到Web表单递交 或 输入域名或页面请求的查询字符串，最终达到**欺骗服务器执行恶意的SQL命令**。

总的来说有以下几点：

- 1.永远不要信任用户的输入，要对用户的输入进行校验，可以通过正则表达式，或限制长度，对单引号和双"--"进行转换等。
- 2.永远不要使用动态拼装SQL，可以使用参数化的SQL或者直接使用存储过程进行数据查询存取。
- 3.永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。

## 跨域

为什么会有跨域出现？

**同源策略**

同源策略，它是由[Netscape](#)提出的一个著名的[安全策略](#)。

现在所有支持JavaScript 的浏览器都会使用这个策略。

所谓**同源是指，域名，协议，端口相同**。

同源策略限制从一个源加载的文档或脚本如何与来自另一个源的资源进行交互。

1 ajax不能跨域请求

2 iframe不能跨域获取dom

这是一个用于隔离潜在恶意文件的关键的安全机制。

禁止浏览器在服务器端不知情的情况下跨域访问！！！

## 1 JSONP

**原理：**动态添加一个<script>标签，而script标签的src属性是没有跨域的限制的。

首先在客户端注册一个callback，然后把callback的名字传递给服务器。

(1) 声明回调函数的名字：<script src="http://xxxxxx?callback=callback"></script>

服务端得到请求的数据后，将json数据直接以入参的方式，放到callback中，这样就生成了一段js语法的文档，返回给客户端。

(2) 客户端加载script标签最终得到的结果：<script src="http://xxxxxx?

**callback=callback">callback(json数据);</script>**

浏览器解析script标签，并执行返回的js，即调用客户端定义的回调函数，跨域完成

限制：只能发送get请求。

## 2.document.domain

这种方式只适合主域名相同，但子域名不同的iframe跨域。

比如主域名是<http://crossdomain.com>:9099，子域名是<http://child.crossdomain.com>:9099，这种情况下给两个页面指定一下document.domain即document.domain = crossdomain.com就可以访

## 3 代理

## 开发：webpack->devServer

## 线上：nginx：

Nginx是一款轻量级的Web 服务器/反向代理服务器,在连接高并发的情况下， Nginx是Apache服务不错的替代品。能够支持高达 50,000 个并发连接数的响应

```
server
{
 listen 8183;
 server_name admin.b2b-manager.onegot-dev.com;
 index index.html index.htm index.php;
 root /app/web/b2b-manager-web; // 前端项目路径
 include deny_file.conf;
 expires off;

 location ~ ^/api/audit/ {
 rewrite ^/api/audit/(.*)$ /report/$1 break; // 例：将/api/audit/test 替换为
/report/test
 proxy_pass http://192.168.10.114:18182; // 反向代理的URL
 proxy_set_header X-Real-IP $remote_addr;
 proxy_set_header Host $host ;
 proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
 }
}
```

## nginx 均衡负载的实现：

1 weight轮询（默认）：接收到的请求按照顺序逐一分配到不同的后端服务器，可以给不同的后端服务器设置一个权重值（weight），用于调整不同的服务器上请求的分配率；权重数据越大，被分配到请求的几率越大；

2 ip\_hash：每个请求按照发起客户端的ip的hash结果进行匹配，这样的算法下一个固定ip地址的客户端总会访问到同一个后端服务器，这也在一定程度上解决了集群部署环境下session共享的问题。

3 fair：智能调整调度算法，动态的根据后端服务器的请求处理到响应的时间进行均衡分配，响应时间短处理效率高的服务器分配到请求的概率高

4 url\_hash：按照访问的url的hash结果分配请求，每个请求的url会指向后端固定的某个服务器，可以在nginx作为静态服务器的情况下提高缓存效率。

## nginx请求403的原因：

1 全局配置中的 user 和当前用户不一致，需要改成当前用户

2 server中的root文件路径权限问题，others没有read权限

3 缺少index.html

## 4 H5-> postMessage()

## 5 cors:

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10。

CORS请求分为两种：

### 简单请求：

只要同时满足以下两大条件，就属于简单请求。

(1) 请求方法是以下三种方法之一：

- HEAD
- GET
- POST

(2) HTTP的头信息不超出以下几种字段：

Accept

Accept-Language

Content-Language

Content-Type

DPR

Downlink

Save-Data

Viewport-Width

Width

Content-Type的值不属于下列之一：

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain 无格式文本

<https://www.jianshu.com/p/b55086cbd9af>

对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个**Origin**字段。Origin字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口）

如果Origin指定的域名在许可范围内，服务器返回的响应，会多出几个头信息字段

**Access-Control-Allow-Origin:** 该字段是必须的。它的值要么是请求时Origin字段的值，要是一个\*，表示接受任意域名的请求。

**Access-Control-Allow-Credentials: true** 该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。

若想发送cookie除了服务器要指定这个字段为true之外，前端请求时也要设置  
**withCredentials**

```
var xhr = new XMLHttpRequest();
xhr.withCredentials = true;
```

如果要发送Cookie，Access-Control-Allow-Origin就不能设为星号，必须指定明确的、与请求网页一致的域名。同时，Cookie依然遵循同源政策，只有用服务器域名设置的Cookie才会上传，其他域名的Cookie并不会上传，且（跨源）原网页代码中的document.cookie也无法读取服务器域名下的Cookie。

**Access-Control-Expose-Headers:** 该字段可选。CORS请求时，XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在Access-Control-Expose-Headers里面指定。

### 非简单请求：

当cors请求的方式不为上文中的三种或者添加了自定义请求头时，就为非简单请求，非简单请求会在正式请求之前发送一次预检请求（preflight）。

预检请求的方法为**OPTIONS**，表示这个请求是用来询问的。头信息里面，关键字段是**Origin**，表示请求来自哪个源。

除了Origin字段，“预检”请求的头信息包括两个特殊字段。

#### (1) Access-Control-Request-Method

该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法

#### (2) Access-Control-Request-Headers

该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段

### 返回的头：

#### (1) Access-Control-Allow-Methods

该字段必需，它的值是逗号分隔的一个字符串，表明服务器支持的所有跨域请求的方法。注意，返回的是所有支持的方法，而不单是浏览器请求的那个方法。这是为了避免多次“预检”请求。

#### (2) Access-Control-Allow-Headers

如果浏览器请求包括Access-Control-Request-Headers字段，则Access-Control-Allow-Headers字段是必需的。它也是一个逗号分隔的字符串，表明服务器支持的所有头信息字段，不限于浏览器在“预检”中请求的字段。

#### (3) Access-Control-Allow-Credentials

该字段与简单请求时的含义相同。

#### (4) Access-Control-Max-Age

该字段可选，用来指定本次预检请求的有效期，单位为秒。上面结果中，有效期是20天（1728000秒），即允许缓存该条回应1728000秒（即20天），在此期间，不用发出另一条预检请求。

## http缓存

### 强缓存：

1. 命中了强缓存时，返回的http状态为200，在chrome的开发者工具的network里面size会显示为from cache
2. 强缓存是利用**Expires**或者**Cache-Control**这两个响应头实现的，它们都用来表示资源在客户端缓存的有效期。

**3. Expires:** 是http1.0提出的，它描述的是一个绝对时间，用GMT格式的字符串表示，如：Expires:Thu, 31 Dec 2037 23:55:55 GMT，它的缓存原理是：

浏览器第一次跟服务器请求一个资源，服务器在响应头加上Expires

浏览器在接收到这个资源后，会把这个资源连同所有响应头一起缓存下来（所以缓存命中的请求返回的header并不是来自服务器，而是来自之前缓存的header）；

浏览器再请求这个资源时，先从缓存中寻找，找到这个资源后，拿出它的Expires跟当前的请求时间比较，如果请求时间在Expires指定的时间之前，就能命中缓存。

如果缓存没有命中，浏览器直接从服务器加载资源时，Expires在重新加载的时候会被更新。

**Cache-Control:** Expires是一个绝对时间，在服务器时间与客户端时间相差较大时，缓存管理容易出现问题，比如随意修改下客户端时间，就能影响缓存命中的结果。所以在http1.1的时候，提出了一个新的header，就是**Cache-Control**，这是一个相对时间，在配置缓存的时候，以秒为单位，用数值表示，如：Cache-Control:max-age=315360000，它的缓存原理是：

原理基本同Expires一样

区别是浏览器再请求这个资源时，根据它第一次的请求时间和**Cache-Control**设定的有效期，计算出一个资源过期时间，再拿这个过期时间跟当前的请求时间比较，

Cache-Control描述的是一个相对时间，在进行缓存命中的时候，都是利用客户端时间进行判断，所以相比较Expires，Cache-Control的缓存管理更有效，安全一些。

这两个header可以只启用一个，也可以同时启用，当response header中，Expires和Cache-Control同时存在时，**Cache-Control**优先级高于**Expires**

**协商缓存：**

当浏览器对某个资源的请求没有命中强缓存，就会发一个请求到服务器，验证协商缓存是否命中，如果协商缓存命中，请求响应返回的http状态为304并且会显示一个Not Modified的字符串

**【Last-Modified, If-Modified-Since】** 的控制缓存的原理是：

浏览器第一次跟服务器请求一个资源，，在响应头加上**Last-Modified**的header，这个header表示这个资源在服务器上的最后修改时间

浏览器再次跟服务器请求这个资源时，在请求头上加上**If-Modified-Since**，这个值就是上一次请求时返回的Last-Modified的值：

服务器再次收到资源请求时，根据浏览器传过来If-Modified-Since和资源在服务器上的最后修改时间判断资源是否有变化，如果没有变化则返回304 Not Modified，但是不会返回资源内容；如果有变化，就正常返回资源内容。

浏览器收到304的响应后，就会从浏览器缓存中加载资源。

如果协商缓存没有命中，浏览器直接从服务器加载资源时，Last-Modified Header在重新加载的时候会被更新，下次请求时，If-Modified-Since会启用上次返回的Last-Modified值。

**【ETag、If-None-Match】**：【Last-Modified, If-Modified-Since】都是根据服务器时间返回的header，一般来说，在没有调整服务器时间和篡改客户端缓存的情况下，这两个header配合起来管理协商缓存是非常可靠的，但是有时候也会服务器上资源其实有变化，但是最后修改时间却没有变化的情况，而这种问题又很不容易被定位出来，而当这种情况出现的时候，就会影响协商缓存的可靠性。所以就有了另外一对header来管理协商缓存，这对header就是【ETag、If-None-Match】。它们的缓存管理的方式是：

原理基本同上，区别：

1 ETag是服务器根据当前请求的资源生成的一个唯一标识，这个唯一标识是一个字符串，只要资源有变化这个串就不同，跟最后修改时间没有关系

2 服务器每次收到请求会重新生成一个ETag用来和请求头中的If-None-Match做比较，即使命中了协商缓存，还是会返回ETag，即使这个ETag跟之前的没有变化

浏览器收到304的响应后，就会从缓存中加载资源。

## tcp三次握手四次挥手

tcp状态标志位：

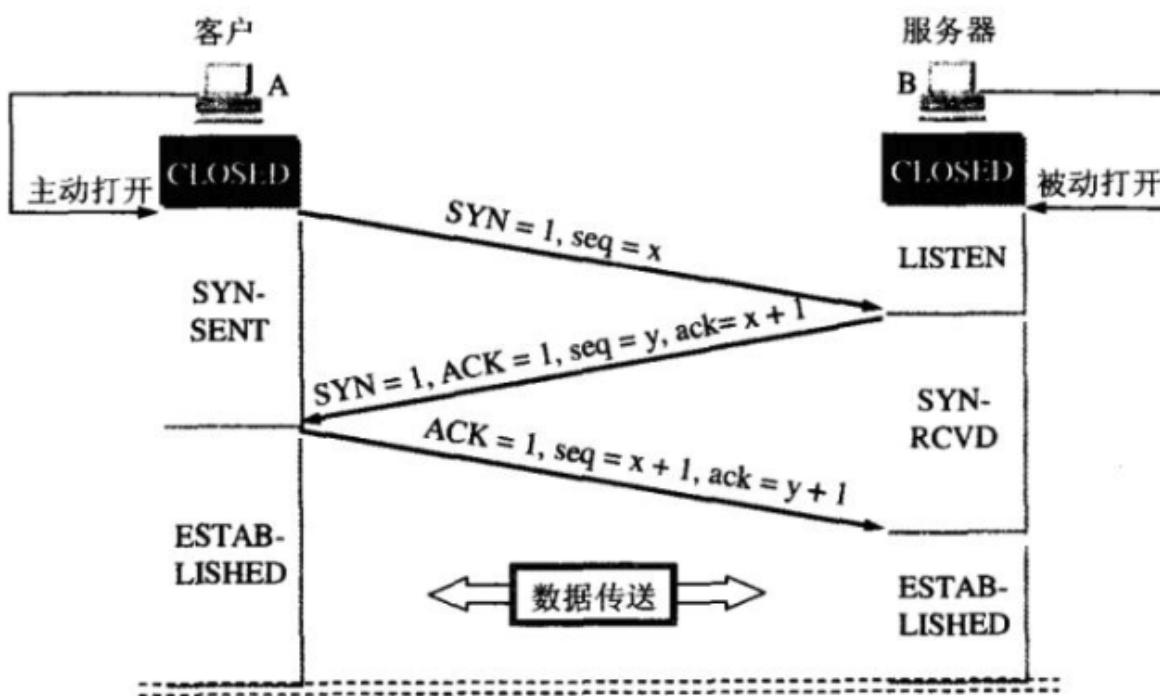
SYN(synchronous建立联机)

ACK(acknowledgement 确认)

FIN(finish结束)

seq = Sequence number(顺序号码)

ack = Acknowledge number(确认号码)



### 第一次握手

客户端向服务端发送连接请求报文段。该报文段的头部中**SYN=1, ACK=0, seq=x**。请求发送后，客户端便进入SYN-SENT状态。

- PS1: **SYN=1** 表示该报文段为连接请求报文。
- PS2: **x** 为本次TCP通信的字节流随机生成的初始序号（ISN）。

TCP规定：SYN=1的报文段不能有数据部分，但要消耗掉一个序号。

### 第二次握手

服务端收到连接请求报文段后，如果同意连接，则会发送一个应答：**SYN=1, ACK=1, seq=y, ack=x+1**。

该应答发送完成后便进入SYN-RCVD状态。

- PS1: **SYN=1, ACK=1** 表示该报文段即为连接请求，也是第一次握手的确认。
- PS2: **seq=y** 表示服务端作为发送者时，发送字节流的初始序号。
- PS3: **ack=x+1** 表示服务端希望下一个数据报发送序号从x+1开始的字节。（第一次握手消耗掉了一个序号）

### 第三次握手

当客户端收到连接同意的应答后，还要向服务端发送一个确认报文段，表示：服务端发来的连接同意应答已经成功收到。

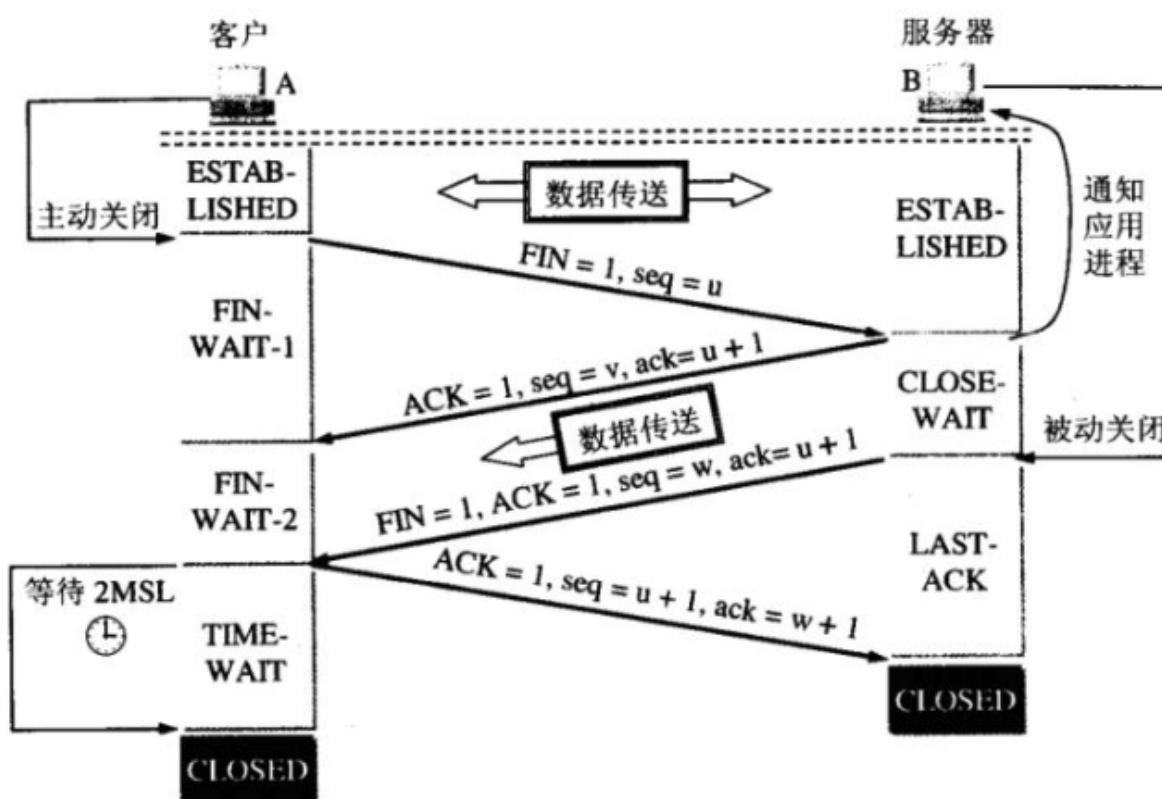
该报文段的头部为：**ACK=1, seq=x+1, ack=y+1**。

客户端发完这个报文段后便进入ESTABLISHED状态，服务端收到这个应答后也进入ESTABLISHED状态，此时连接的建立完成！

为什么连接建立需要三次握手，而不是两次握手？

防止失效的连接请求报文段被服务端接收，从而产生错误。

PS：失效的连接请求：若客户端向服务端发送的连接请求丢失，客户端等待应答超时后就会再次发送连接请求，此时，上一个连接请求就是『失效的』



作者：大闲人柴毛毛

链接：<https://www.zhihu.com/question/24853633/answer/254224088>

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

我们知道，TCP连接是双向的，因此在四次挥手中，前两次挥手用于断开一个方向的连接，后两次挥手用于断开另一个方向的连接。

### 第一次挥手

若A认为数据发送完成，则它需要向B发送连接释放请求。该请求只有报文头，头中携带的主要参数为：

**FIN=1, seq=u**。此时，A将进入FIN-WAIT-1状态。

- PS1: FIN=1表示该报文段是一个连接释放请求。
- PS2: **seq=u, u-1是A向B发送的最后一个字节的序号**。

### 第二次挥手

B收到连接释放请求后，会通知相应的应用程序，告诉它A向B这个方向的连接已经释放。此时B进入CLOSE-WAIT状态，并向A发送连接释放的应答，其报文头包含：

**ACK=1, seq=v, ack=u+1**。

- PS1: ACK=1: 除TCP连接请求报文段以外, TCP通信过程中所有数据报的ACK都为1, 表示应答。
- PS2: seq=v, v-1是B向A发送的最后一个字节的序号。
- PS3: ack=u+1表示希望收到从第u+1个字节开始的报文段, 并且已经成功接收了前u个字节。

A收到该应答, 进入FIN-WAIT-2状态, 等待B发送连接释放请求。

第二次挥手完成后, A到B方向的连接已经释放, B不会再接收数据, A也不会再发送数据。但B到A方向的连接仍然存在, B可以继续向A发送数据。

#### 第三次挥手

当B向A发完所有数据后, 向A发送连接释放请求, 请求头: FIN=1, ACK=1, seq=w, ack=u+1。B便进入LAST-ACK状态。

#### 第四次挥手

A收到释放请求后, 向B发送确认应答, 此时A进入TIME-WAIT状态。该状态会持续2MSL时间, 若该时间段内没有B的重发请求的话, 就进入CLOSED状态, 撤销TCB。当B收到确认应答后, 也便进入CLOSED状态, 撤销TCB。

**为什么A要先进入TIME-WAIT状态, 等待2MSL时间后才进入CLOSED状态?**

为了保证B能收到A的确认应答。

若A发完确认应答后直接进入CLOSED状态, 那么如果该应答丢失, B等待超时后就会重新发送连接释放请求, 但此时A已经关闭了, 不会作出任何响应, 因此B永远无法正常关闭。

<https://www.zhihu.com/question/24853633/answer/254224088>

## 为什么建立连接是三次握手, 而关闭连接却是四次挥手呢?

这是因为服务端在LISTEN状态下, 收到建立连接请求的SYN报文后, 把ACK和SYN放在一个报文里发送给客户端。而关闭连接时, 当收到对方的FIN报文时, 仅仅表示对方不再发送数据了但是还能接收数据, 己方是否现在关闭发送数据通道, 需要上层应用来决定 (此时服务端可能还在发送数据), 因此, 己方ACK和FIN一般都会分开发送。

## 浏览器加载/解析/渲染

dns解析域名为ip->向ip发请求->-tcp三次握手建立连接 -> 服务器将html文件放入响应报文实体中返回-> 浏览器解析html->

### html自上而下渲染

- 样式表在下载完成后, 将和以前下载的所有样式表一起进行解析, 解析完成后, 将对此前所有元素(含以前已经渲染的)重新进行渲染
- 遇到图片资源, 浏览器会发出请求获取图片资源. 这是异步请求, 并不会影响html文档进行加载,
- 当文档加载过程中遇到js文件, html文档会挂起渲染(加载解析渲染同步)的线程, 不仅要等待文档中js文件加载完毕, 还要等待解析执行完毕, 才可以恢复html文档的渲染线程. 即js的加载不能并行下载和解析
  - 原因: js有可能会修改DOM, 比如document.write. 这意味着, 在js执行完成前, 后续所有资源的下载可能是没有必要的, 这是js阻塞后续资源下载的根本原因.
  - 所以一般将外部引用的js文件放在</body>前
- 虽然css文件的加载不影响js文件的加载, 但是却影响js文件的执行, 即使js文件内只有一行代码, 也会造成阻塞

- 原因: 可能会有: `var width = $('#id').width()`. 这意味着, 在js代码执行前, 浏览器必须保证css文件已下载和解析完成。这也是css阻塞后续js的根本原因。
- 办法: 当js文件不需要依赖css文件时, 可以将js文件放在头部css的前面。

主要解析过程:

1. 浏览器解析html源码, **创建一棵DOM树**
2. 浏览器解析CSS代码, **创建css树**
- 3. js解析**因为文件在加载的同时也进行解析
4. 构建DOM树, 并且计算出样式数据后, 下一步就是**构建一棵渲染树(rendering tree)**
  - a. 渲染树和DOM树有区别, DOM树完全与html标签一一对应, 但是渲染树会忽略掉不需要渲染的元素, 比如head, display: none的元素等
  - b. 一大段文本中的每一行在渲染树中都是一个独立的节点
  - c. 渲染树的每一个节点都存储有对应的css属性
5. 渲染树创建好, 浏览器就可以根据渲染树直接把页面绘制到屏幕上

## 原生ajax

```

1 function ajax(url, fn) {
2 let xhr = new XMLHttpRequest();
3 xhr.open('GET', url)//初始化请求
4 xhr.send(); //发送请求
5 xhr.onreadystatechange = function () { //监听readystatechange
6 if (xhr.readyState == 4 && xhr.status == 200) {
7 fn(xhr.responseText)//请求返回的文本
8 }
9 }
10 }
```

- `readyStatus`的五个阶段
  - 0: 未初始化。尚未调用`open()`方法
  - 1: 启动。已经调用`open()`方法, 尚未调用`send()`方法
  - 2: 发送。已经调用`send()`方法, 尚未接收到响应
  - 3: 接收。已经接收部分响应数据。
  - 4: 完成。已经接收到全部响应数据, 而且已经可以在客户端使用了。【一般只需检查这个阶段】

## link和@import

就结论而言, 强烈建议使用link标签, 慎用@import方式。

区别

### 1.从属关系区别

@import是 **CSS** 提供的语法规则, 只有导入样式表的作用; link是 **HTML** 提供的标签, 不仅可以加载 CSS 文件, 还可以定义 RSS、rel属性规定当前文档与被链接文档之间的关系。) 连接属性

等。

## 2. 加载顺序区别

加载页面时，link标签引入的CSS被同时加载；@import引入的CSS将在页面加载完毕后被加载。

## 3. DOM可控性区别

可以通过JS操作DOM，插入link标签来改变样式；由于DOM方法是基于文档的，无法使用@import的方式插入样式。

# HTTP 1.0->1.1->2.0

## HTTP1.1与HTTP1.0的区别

http1.0没有**Host**头域字段(**HTTP1.0**中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名 (hostname)

而http1.1默认持久连接，在同一个连接中可以传送多个请求和响应而http1.0需要指定 **connection: keep-alive**

http1.1中引入了**ETag**头，它的值entity tag可以用来唯一的描述一个资源。请求消息中可以使用If-None-Match头域来匹配资源的entitytag是否有变化

http1.1新增了**Cache-Control**头域(消息请求和响应请求都可以使用)，它支持一个可扩展的指令子集

## HTTP2.0与HTTP1.1的区别

**1 多路复用 (Multiplexing)** 多路复用允许同时通过单一的 **HTTP/2** 连接发起多重的请求-响应消息。

http1.0：默认情况下每一次请求都需要建立一次tcp连接，包括三次握手四次挥手，很费时间

通用请求头：**Connection: keep-alive** 一定时间内，同一域名多次请求数据，只建立一次HTTP请求

http1.1：默认持久连接

keep-alive的问题：在http1.1协议中，浏览器客户端在同一时间，针对同一域名下的请求有一定数量限制。超过限制数目的请求会被阻塞，这也是为何一些站点会有多个静态资源CDN域名的原因之一

**2 二进制分帧** 在应用层(HTTP/2)和传输层(TCP or UDP)之间增加一个二进制分帧层。在二进制分帧层中，HTTP/2会将所有传输的信息分割为更小的消息和帧(frame)，并对它们采用二进制格式的编码，HTTP1.x的首部信息会被封装到 **HEADER frame**，而相应的Request Body则封装到 **DATA frame** 里面。

**3 首部压缩 (Header Compression)** HTTP/2则使用了专门为头部压缩而设计的 **HPACK** 算法。

**4 服务端推送 (Server Push)** 服务端推送是一种在客户端请求之前发送数据的机制

比如说某个页面 **index.html** 使用了 **a.css** 和 **b.js** 两个子资源，Web服务器在返回 **index.html** 的内容后表示“你可能还需要这两个文件”将 **a.css** 和 **b.js** 的内容一并发送给了客户端浏览器，于是浏览器就不需要另外去单独请求这两个文件。

实现：

```
1 nginx:
2 location ~xxx {
3 http2_push_preload on; # 加上这行
4 }
```

响应头加：

```
1 link: </index.js>; as=script; rel=preload, </index.css>; as=style; rel=preload
```

# 前端优化

## 网络层面：

### 减少http请求

- 1 合理利用http缓存 (webpack 分离第三方库的代码)
- 2 css sprite 背景图合并
- 3 webpack 打包css,js文件,
- 4 图片 data:url模式嵌入页面， 不需要发送http请求
- 5 图片懒加载

### 减少资源体积

- 1 传输内容使用gzip压缩

```
accept-encoding: gzip
content-encoding: gzip
```

- 2 js混淆、css压缩、图片压缩

- 3 按需加载

### 其他：

- 1 将静态资源放到 cdn (Content Delivery Network) , 即内容分发网络

CDN就可以理解为分布在每个县城的火车票代售点，用户在浏览网站的时候，CDN会选择一个离用户最近的CDN边缘节点来响应用户的请求，这样海南移动用户的请求就不会千里迢迢跑到北京电信机房的服务器（假设源站部署在北京电信机房）上了。

CDN的优势： (1) CDN节点解决了跨运营商和跨地域访问的问题，**访问延时大大降低**；

(2) 大部分请求在CDN边缘节点完成，CDN起到了**分流作用**，减轻了源站的负载。

2 将资源放在多个域名下，原因是http1.1浏览器对同一域名的同时请求有数量限制

## 渲染层面：

### 1 减少回流次数

- (1) 将动画需要多次重排的元素或者设置position为absolute，这样元素脱离了文档流
- (2) 布局相关属性会导致强制回流，多次使用时，最好先缓存起来**
- (3) 在内存中多次操作节点，完成后再添加到文档中去。比如一个表格，构建好整个html片段再插入到dom中而不是一行一行循环插入dom
- (4) 使用display:none技术，只引发两次回流和重绘；使用cloneNode(true or false) 和 replaceChild 技术，引发一次回流和重绘；

2 link标签放在头部，因为可以和html同时解析，script标签放在body中的最后，因为会造成阻塞，除非开启defer/async 属性

3 尽量使用css动画

## 代码层面：

1 dom操作取得的类数组**不是一个静态结果**，每次访问该集合都会重新执行查询，性能很低，所以循环使用时，最好先转换为真正的数组 (Array.from(类数组))，或者将length和成员保存在局部变量中

2 减少作用域链的查找

比如函数内一个循环，循环一万次，每次循环要访问一个全局变量，那么就要访问一万次全局变量，一万次的作用域链的查找

如果循环前，在局部变量中缓存这个全局变量，那么只要访问一次全局变量。

#### 4 使用事件代理，减少代码量，减少内存使用

6 js按需加载（例：antd中的组件）

### localStorage和sessionStorage区别

- localStorage和sessionStorage一样都是用来存储客户端临时信息的对象。
- 他们均只能存储字符串类型的对象（虽然规范中可以存储其他原生类型的对象，但是目前为止没有浏览器对其进行实现）。
- **localStorage生命周期是永久**，这意味着除非用户显示在浏览器提供的UI上清除localStorage信息，否则这些信息将永远存在。

**sessionStorage生命周期为当前窗口或标签页**，一旦窗口或标签页被永久关闭了，那么所有通过 sessionStorage存储的数据也就被清空了。

- 不同浏览器无法共享localStorage或sessionStorage中的信息。**相同浏览器的不同页面间可以共享相同的 localStorage（页面属于相同域名和端口）**，但是不同页面或标签页间无法共享 sessionStorage的信息。这里需要注意的是，页面及标签页仅指顶级窗口，如果一个标签页包含多个iframe标签且他们属于同源页面，那么他们之间是可以共享sessionStorage的。
- 存储大小大约为**5m**

## webpack

### 基本使用

<http://www.zbzero.com/#/singleArticle/91>

### 构建速度优化

<http://www.zbzero.com/#/singleArticle/92>

## Node.js

### npm

npm 由三个独立的部分组成：

- 1 网站 是开发者查找包（package）、设置参数以及管理 npm 使用体验的主要途径。
- 2 注册表 是一个巨大的数据库，保存了每个包（package）的信息。
- 3 CLI (command-line interface, 命令行界面) 通过命令行或终端运行。开发者通过 CLI 与 npm 打交道

#### 安装一个包

npm install 如果存在 package.json 文件，则会在 package.json 文件中查找针对这个包所约定的语义化版本规则，然后安装符合此规则的最新版本。

#### 更新一个包

**npm update/up/upgrade** 默认情况更新package.json中所有包，也会安装之前遗漏没有安装的包，npm 2.6.1之后，更新最顶层的包以及递归更新其所有依赖包，加 --save 会同时更改 package.json中的版本号

### 版本前缀符号的作用

- ~ 会匹配最近的小版本依赖包，比如~1.2.3会匹配所有1.2.x版本，但是不包括1.3.0
- ^ 会匹配最新的大版本依赖包，比如^1.2.3会匹配所有1.x.x的包，包括1.3.0，但是不包括2.0.0
- \* 这意味着安装最新版本的依赖包

### **npm outdated** 检查已安装的包查看是否已经过时

卸载一个包

**npm uninstall** 若想同时删除package.json中的的依赖，加 -S/-  
**package.json**

1 列出了项目所需的依赖包

2 允许指定 包的版本

3 让 项目/包 更容易分享给其他开发者使用

创建一个package.json： **npm init** ,若创建一个默认配置的加 --yes

配置：

必须：

1 name : 项目/包的名称， 小写， 无空格， 允许破折号和下划线

2 version: 版本号 格式为 x.x.x

配置项：

1 **dependencies**: 在生产环境运行时需要的包，自动添加： npm install --save/-S

2 **devDependencies** 开发程序的时候需要的模块 自动添加： npm install --save-dev/-D

对于web项目， devDependencies可能是babel， webpack， webpack-dev-server等

使用npm install 会把 dependencies 和 devDependencies内的包全部安装，若要区分,后缀+ --dev 或 --prodution

### 同版本依赖

当这个包是某个宿主包的插件时，它不需要引用宿主包，并且它很可能是设计给宿主包的某一个特定版本使用的，为了防止出现使用此插件时宿主包的版本和插件预期的宿主版本不同而无法正常使用的问题，需要在插件包的package.json内声明支持的宿主包版本，例如：一个webpack插件，支持的webpack版本为4，则添加：

```
"peerDependencies" : {
 "webpack": "4.x" //也可以写成">= xxx < xxx"
}
```

当使用者install时，若发现安装的宿主包版本不在插件内peerDependencies支持的版本范围之内，就会报错。

### **npx:**

npx 会自动查找当前依赖包中的可执行文件，如果找不到，就会去 PATH 里找。如果依然找不到，就会帮你安装

```
1 npm i webpack webpack-cli -D
2 正常在命令行里使用: ./node_modules/.bin/webpack -v
3 使用npx: npx webpack -v
```

npx还允许我们单次执行命令而不需要安装，例如

```
1 npx create-react-app my-cool-new-app
```

这条命令会临时安装 create-react-app 包，命令完成后 create-react-app 会删掉，不会出现在 global 中。下次再执行，还是会重新临时安装。

```
$ npm config set registry=https://registry.npm.taobao.org 更换为淘宝镜像
```

```
$ npm config set registry=http://registry.npmjs.org 原镜像
```

## koa2

web开发框架，核心在于实现了基于async函数的中间件机制，保证了中间件的有序执行  
实现的很简洁，开发时新建一个koa实例，通过use方法添加中间件，通过listen方法开启服务  
**中间件实现原理：**

koa导出的application类内部定义了一个数组类型属性middleware用来存放中间件

```
1 this.middleware = [];
```

use方法会将中间件push到middleware

```
1 this.middleware.push(fn);
```

当使用listen开启服务时，koa会先调用compose函数将middleware作为参数

```
1 listen(...args) {
2 const server = http.createServer(this.callback()); // 使用callback返回的函数作为收到请求的处理函数
3 return server.listen(...args);
4 }
```

```
1 callback() {
2 const fn = compose(this.middleware); // 调用compose函数返回fn
3 return (req, res) => {
4 const ctx = this.createContext(req, res); // 根据node提供的request, response对象创建上下文对象
5 return this.handleRequest(ctx, fn);
6 };
7 }
```

```
1 handleRequest(ctx, fnMiddleware) {
2 const onerror = err => ctx.onerror(err);
3 const handleResponse = () => respond(ctx);
4 return fnMiddleware(ctx).then(handleResponse).catch(onerror); // 当所有中间件执行完后调用handleResponse
5 }
```

compose函数返回一个匿名函数，这个匿名函数接收一个上下文对象context  
匿名函数内部又定义了一个dispatch函数，并执行dispatch(0)

当接收到请求时koa的处理函数handleRequest内将执行compose返回的匿名函数，即开始执行dispatch(0)

dispatch函数会将参数0作为index去middleware取第一个中间件，执行这个中间件并将 1 外层函数即闭包内的context对象 2 dispatch(1) 作为参数传入

中间件执行时若遇到 await next(); 即await dispatch(1) 则递归执行disptach： 取下一个中间件执行并将下下个中间件作为参数传入。 执行权通过await next向下传递，当最后一个中间件执行完成后，执行权返回上一层，执行完next下面的代码后，再返回上一层以此类推，这样保证了中间件的执行顺序。

```
1 function compose (middleware) {
2 return function (context) {
3 let index = -1
4 return dispatch(0)
5 function dispatch (i) {
6 if (i <= index) return Promise.reject(new Error('next() called multiple time
7 index = i
8 let fn = middleware[i] // 取中间件
9 if (!fn) return Promise.resolve() // 取不到说明执行完毕了直接返回一个状态为已完成的p
10 try {
11 return Promise.resolve(fn(context, dispatch.bind(null, i + 1))); // 返回一
12 } catch (err) {
13 return Promise.reject(err)
14 }
15 }
16 }
17 }
}
```

有一点需要注意的是dispatch内部执行中间件时使用Promise.resolve包着（11行），这么做的目的是，中间件因为是async函数，所以执行的返回值也是一个promise实例，并且只有当函数内所有的await都执行完成了这个promise实例的状态才会变为已完成，而Promise.resolve返回的promise的状态取决于内部中间件，即当所有中间件都执行完毕之后，Promise.resolve返回的外部promise的状态才会变为已完成

handleRequest内部调用compose返回的匿名函数时，利用promise的特性将处理响应的函数handleResponse作为then方法的参数保证了处理响应发生在所有中间件执行完成之后

## 前端工程化

参考自：<https://www.zhihu.com/question/24558375>

## 模块化

JS:

CommonJs :require

ES6: import export

Css:

- CSS in JS是彻底抛弃CSS，使用JS或JSON来写样式。这种方法很激进，不能利用现有的CSS技术（sass/less/postcss），而且处理伪类等问题比较困难；
- CSS Modules仍然使用CSS，只是让JS来管理依赖。它能够最大化地结合CSS生态和JS模块化能力，目前来看是最好的解决方案。Vue的scoped style也算是一种资源：

Webpack 打包

## 组件化

模块化只是在文件层面上，对代码或资源的拆分；而组件化是在设计层面上，对UI（用户界面）的拆分。

从UI拆分下来的每个包含模板(HTML)+样式(CSS)+逻辑(JS)功能完备的结构单元，我们称之为组件。

例：Vue,react等框架

## 规范化

1 目录结构制定规范

2 编码规范

3 接口规范

4 文档规范

5 git分支管理

## 自动化

1 持续集成

2 自动化构建

3 自动化部署

4 自动化测试

## 移动端适配

**window.devicePixelRatio** 获取屏幕像素比

移动端默认视窗宽度为980px,手机浏览器为了不出现滚动条会把页面自动缩放至屏幕大小

```
1 <meta name="viewport" content="width=device-width,initial-scale=1.0">
```

设置一个meta标签，name为viewport代表是一个视窗，content中设置width=device-width表示视窗的宽度为屏幕宽度，initial-scale=1.0表示不缩放

**rem**:对于iphon6, dpr=2的情况下，设计稿为750px宽，若要实现设计稿中1px的边框，则设置边框为1px，并且设置initial-scale=0.5 从而实现了0.5px边框

把html跟元素的font-size (即rem) 设置为 宽度/10，元素的宽度使用rem相对单位，从而做到了不同大小屏幕的适配

**Vw:**

- vw: 是Viewport's width的简写,1vw等于window.innerWidth的1%

- vh: 和vw类似，是Viewport's height的简写，1vh等于window.innerHeight的1%
- vmin: vmin的值是当前vw和vh中较小的值
- vmax: vmax的值是当前vw和vh中较大的值

原文: <https://www.w3cplus.com/css/vw-for-layout.html> © w3cplus.com

## 算法

### 1 冒泡排序

```

1 function bubbleSort(arr) {
2 var i,j;
3 for(i=1;i<arr.length;i++){
4 for (j=0;j<arr.length-i;j++){
5 if(arr[j]>arr[j+1]){
6 arr=[...arr.slice(0,j),arr[j+1],arr[j],...arr.slice(j+2)]
7 /* var temp=arr[j];
8 arr[j]=arr[j+1];
9 arr[j+1]=temp; 这样减少内存使用 */
10 }
11 }
12 }
13 return arr;
14 }
```

每次内部的循环会两两比较最后将最大的值放到最后面，外部循环一次后i++，使得下一次内部循环j<arr.length-i 排除上一次循环的最后一项也就是上次循环得到的最大项

### 2 选择排序

```

1 function selectSort(arr) {
2 var i,j;
3 for(i=1;i<arr.length;i++){
4 for (j=arr.length-1;j>=i;j--){
5 if(arr[j]>arr[j-1]){
6 arr=[...arr.slice(0,j-1),arr[j],arr[j-1],...arr.slice(j+1)]
7 /* var temp=arr[j];
8 arr[j]=arr[j-1];
9 arr[j-1]=temp; 这样减少内存使用 */
10 }
11 }
12 }
13 return arr;
14 }
```

每次内部的循环会两两比较最后将最小的值放到最前面，外部循环一次后i++，使得下一次内部循环j>=i 排除上一次循环的第一项也就是上次循环得到的最小项

### 3 二分法

```

1 function dich(arr,value,leftIndex=0,rightIndex=arr.length) {
2 var i=Math.floor((leftIndex+rightIndex)/2);
3 if(arr[i]==value){
4 return `在下标为${i}处找到了值${value}`;
5 }else {
6 if(arr.length==1) return '没有此值';
7 return arr[i]>value?
8 dich(arr,value,leftIndex,i-1):
9 dich(arr,value,i+1,rightIndex)
10 }
11 }

```

## 4 插入排序

结果从小到大

首先把数组第一项认为是有序的，从第二项开始，每次循环把该项插入到前面的有序列表中  
把要插入的值与前面有序的队列循环作比较，如果要插入的值比有序队列的小，则有序队列中的值向后移动 ( $arr[j+1]=arr[j]$ ) ,而第j项作为待插入的位置，内部循环结束，由于最后一次循环 $j--$ ,所以最终确定的带插入位置为 $j+1$

```

1 function insertSort(arr) {
2 for(var i=1;i<arr.length;i++){
3 var insertV=arr[i];
4 var j = i-1; //insertV是要插入的值,j是数组循环开始的位置
5 while (arr[j] > insertV){
6 arr[j+1]=arr[j];
7 j--;
8 }
9 arr[j+1]=insertV
10 }
11 return arr
12 }

```

例 [3,2,4,1]

首先把第一项3 作为有序队列，从第二项2开始，插入值insertValue 为2， 内部循环起始位置 $j = 0$ ,开始内部循环，由于 $arr[j] 3 > insertV2$

## git

创建本地仓库 **git init** 可以发现当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的

**git diff 文件名** 比较比较工作区和暂存区（最后一次add）的区别

**git diff --cache 文件名**比较暂存区和版本库的区别

**git diff HEAD -- 文件名** 比较工作区和版本库（最后一次commit）的区别

**git log**命令显示从最近到最远的提交日志 加上**--pretty=oneline**单行显示

HEAD表示当前版本 上一个版本就是 $HEAD^$ ， 上上一个版本就是 $HEAD^{^2}$  前五个版本 $HEAD~5$

**git reset --hard HEAD^** 回退到上个版本,如果要撤销，需要提前记住之前版本的commit id

--soft 会将指定版本到原HEAD版本之间所有变更退回到暂存区，工作区修改的内容不变

--hard会完全回退到指定版本，工作区的修改清空

--mixed 会将指定版本到原HEAD版本之间所有变更退回到工作区

如果没有记住commit id 也没关系，使用**git reflog** 查看之前用到的每一条命令

**工作区：**就是你在电脑里能看到的目录

**版本库：** 工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改

**撤销修改：**

(1) 还没add:

**git checkout -- file**可以丢弃工作区的修改:

若文件修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；除了撤销修改，若文件被误删时，也可以使用这个命令使文件恢复到版本库的版本，其实本质是用版本库里的版本替换工作区的版本

若已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

(2) 已经add但还没commit:

**git reset HEAD <file>**可以把暂存区的修改撤销掉（unstage），重新放回工作区，然后 git checkout 丢弃工作区修改

git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

(3) 已经commit 还没push:

上文中提到过的

**git reset --hard HEAD^**

**远程仓库**

由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以需要在github设置本地的公钥

生成ssh公钥 ssh-keygen -t rsa -C "youremail@example.com"，然后在 .ssh 目录找到 id\_rsa.pub 即为公钥

(1) 如果先有的本地仓库，再新建的远程仓库：

**git remote add origin git@github.com:xxx/xxx.git** 关联远程仓库

然后 **git push -u origin master** 把本地内容推送到远程仓库，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起

来，在以后的推送或者拉取时就可以简化命令。

(2) 如果先创建了远程仓库，那么直接 **git clone git@github.com:xxx/xxx.git** 从远程仓库克隆即可

(3) 如果远程仓库已存在，本地也有内容，不能直接clone，它会在当前目录新建一个名字为远程仓库名的子目录，此时需要使用 **git pull origin master --allow-unrelated-histories** 强行合并远程和本地仓库，然后会把本地的内容直接add->commit一次

## 分支管理

创建并切换分支dev: **git checkout -b dev**，若需要根据远程的分支创建本地分支需要 **git checkout -b dev origin/dev**

-b是创建分支的意思，等于 **git branch dev**，不加分支名列出了所有分支，**git branch -d dev** 删除dev分支，**git branch -D dev** 强制删除未合并的分支

**git checkout 分支名** 用来切换分支

**git merge**命令用于合并指定分支到当前分支，默认为快速合并

若合并分支时，发生了冲突，需要我们手动解决冲突，冲突的地方会被git标记：

```
<<<<< HEAD
var b = 123123123321321321321;
=====
var b = 123123123123123123123;
>>>>> dev
```

然后我们自行修改之后，重新 add,commit即可完成合并，若此时切换回dev分支，会发现dev分支还是上一次commit的状态并不受到master分支merge的影响

使用默认的Fast forward模式合并分支，当删除分支后，会丢掉分支信息。在log中看不出来曾经做过合并，加上参数 **--no-ff** 使用普通模式合并可以保留历史中的分支信息

使用**git stash**将你修改过的被追踪的文件（当文件add过之后就为被追踪的文件，修改过的被追踪的文件的意思就是之前add过然后又修改了还没有add）和暂存区的内容储藏起来

使用场景：在当前分支开发到一半还不能commit时，需要切换到其他分支进行其他任务，这时候需要先使用git stash将工作区储藏（注意若有新文件，需要先add到暂存区才能被储藏），然后再且切换分支，待其他任务做完后返回这个分支，使用**git stash pop** 将储藏的内容恢复到工作区或者 **git stash apply** 恢复，但是恢复后，stash内容并不删除，你需要用**git stash drop**来删除，如果你想应用更早的储藏，你可以通过名字指定它

## 标签管理

**git tag** 查看所有标签

**git tag** 标签名 默认标签是打在最新提交的commit上，标签可以通过commit id打在指定的地方

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：**git tag -a 标签名 -m 说明文字**  
通过git show 标签名可以看到说明文字

标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签。

删除本地标签 **git tag -d 标签名**，删除远程标签 **git push origin :refs/tags/标签名**

# bash

## bash和shell：

操作系统可以分成核心（kernel）和Shell（外壳）两部分，其中，Shell是操作系统与外部的主要接口，位于操作系统的外层，为用户提供与操作系统核心沟通的途径。是一个命令解释器  
bash 是一个为GNU项目编写的Unix shell，也就是linux默认用的shell。

bash是一种shell，在window中shell是cmd

## 小技巧：

1 按住左键选中的文字，会把文本放入缓冲区，按下滚轮键黏贴

2 . 当前目录， ..父目录， 一般情况下 . 可以省略

## 常用命令：

date 日期时间

cal 日历

**pwd** 显示当前目录

**file** 文件名： 判断该文件的类型

**less** 文件名： 预览文件内容

| 命令                 | 行为                           |
|--------------------|------------------------------|
| Page UP or b       | 向上翻滚一页                       |
| Page Down or space | 向下翻滚一页                       |
| UP Arrow           | 向上翻滚一行                       |
| Down Arrow         | 向下翻滚一行                       |
| G                  | 移动到最后一行                      |
| 1G or g            | 移动到开头一行                      |
| /charaters         | 向前查找指定的字符串                   |
| n                  | 向前查找下一个出现的字符串，这个字符串是之前所指定查找的 |
| q                  | 退出 less 程序                   |

**ls** 列出一个目录包含的文件及子目录

| 选项 | 长选项              | h                                                                               |
|----|------------------|---------------------------------------------------------------------------------|
| -a | --all            | 列出所有文件，甚至包括文件名以圆点开头的默认会被隐藏的隐藏文件。                                                |
| -d | --directory      | 通常，如果指定了目录名，ls 命令会列出这个目录中的内容，而不是目录本身<br>这个选项与 -l 选项结合使用，可以看到所指定目录的详细信息，而不是目录内容。 |
| -F | --classify       | 这个选项会在每个所列出的名字后面加上一个指示符。例如，如果名字是目录<br>则会加上一个'/'字符。                              |
| -h | --human-readable | 当以长格式列出时，以人们可读的格式，而不是以字节数来显示文件的大小。<br>1024字节会显示为1k                              |
| -l |                  | 以长格式显示结果。                                                                       |
| -S |                  | 命令输出结果按照文件大小来排序。                                                                |
| -t |                  | 按照修改时间来排序。                                                                      |

展示（文件索引节点）的信息。在命令中加上“-i”选项，判断是不是同一个文件

**cd** 改变当前工作目录

| 快捷键           | 运行结果                                          |
|---------------|-----------------------------------------------|
| cd            | 更改工作目录到你的家目录。                                 |
| cd -          | 更改工作目录到先前的工作目录。                               |
| cd ~user_name | 更改工作目录到用户家目录。例如, cd ~bob 会更改工作目录到用户“bob”的家目录。 |

关于文件名：

- 1 隐藏文件以 . 字符开头, ls -a 可以列出
- 2 文件名对于大小写敏感
- 3 没有文件扩展名的概念
- 4 标点符号仅限于 . - \_

## linux系统中的目录

| 目录             | 评论                                                                        |
|----------------|---------------------------------------------------------------------------|
| /              | 根目录, 万物起源。                                                                |
| /bin           | 包含系统启动和运行所必须的二进制程序。                                                       |
| /boot          | 包含 Linux 内核、初始 RAM 磁盘映像（用于启动时所需的驱动）和 启动加载程序.                              |
| /dev           | 这是一个包含设备结点的特殊目录。“一切都是文件”，也适用于设备。在这个目录里                                    |
| /etc           | 这个目录包含所有系统层面的配置文件。它也包含一系列的 shell 脚本,这个目录中的任                               |
| /home          | 通常的配置环境下，系统会在/home 下，给每个用户分配一个目录。普通用户只能 在                                 |
| /lib           | 包含核心系统程序所使用的共享库文件。这些文件与 Windows 中的动态链接库相似。                                |
| /root          | root 帐户的家目录。                                                              |
| /sbin          | 这个目录包含“系统”二进制文件。它们是完成重大系统任务的程序，通常为超级用户使                                   |
| /tmp           | 用来存储由各种程序创建的临时文件的地方。系统每次重新启动时，都会清空这个目                                     |
| /usr           | 在 Linux 系统中，/usr 目录可能是最大的一个。它包含普通用户所需要的所有程序和文                             |
| /usr/bin       | /usr/bin 目录包含系统安装的可执行程序。通常，这个目录会包含许多程序。                                   |
| /usr/lib       | 包含由/usr/bin 目录中的程序所用的共享库。                                                 |
| /usr/local     | 非系统自带程序的安装目录                                                              |
| /usr/sbin      | 包含许多系统管理程序。                                                               |
| /usr/share/doc | 大多数安装在系统中的软件包会包含一些文档。在/usr/share/doc 目录下，我们可以找                            |
| /var           | 存放的是动态文件。各种数据库，假脱机文件， 用户邮件等等，都位于在这里。                                      |
| /var/log       | 包含日志文件、各种系统活动的记录。这些文件非常重要，并且 应该时时监测它们。<br>系统安全，在一些系统中，你必须是超级用户才能查看这些日志文件。 |

## 操作文件和目录

**cp item1 item2**— 复制item1 到 item2

|                 |                                            |
|-----------------|--------------------------------------------|
| -a, --archive   | 复制文件和目录，以及它们的属性，包括所有权和权限。通常，复本具有用户所操作默认属性。 |
| -r, --recursive | 递归地复制目录及目录中的内容。当复制目录时，需要这个选项（或者-a 选项）。     |
| -u, --update    | 仅复制目标目录中不存在的文件，或者是文件内容新于目标目录中已经存在的文件。      |
| -v, --verbose   | 显示详细的命令操作信息                                |

例子：

|                     |                                                                                                       |
|---------------------|-------------------------------------------------------------------------------------------------------|
| cp file1 file2      | 复制文件 file1 内容到文件 file2。如果 file2 已经存在，file2 的内容会被 file1 的内容重存在，则会创建 file2。                             |
| cp file1 file2 dir1 | 复制文件 file1 和文件 file2 到目录 dir1。目录 dir1 必须存在。                                                           |
| cp dir1/* dir2      | 使用一个通配符，在目录 dir1 中的所有文件都被复制到目录 dir2 中。dir2 必须已经存                                                      |
| cp -r dir1 dir2     | 复制目录 dir1 中的内容到目录 dir2。如果目录 dir2 不存在，创建目录 dir2，操作完成内容和 dir1 中的一样。如果目录 dir2 存在，则目录 dir1 (和目录中的内容)将会被复制 |

## mv — 移动/重命名文件和目录, -u, -v 同上

例子：

|                     |                                                                                                   |
|---------------------|---------------------------------------------------------------------------------------------------|
| mv file1 file2      | 如果 file2 存在，它的内容会被 file1 的内容重写。如果 file2 不存在，则创建 file2。                                            |
| mv file1 file2 dir1 | 移动 file1 和 file2 到目录 dir1 中。dir1 必须已经存在。                                                          |
| mv dir1 dir2        | 如果目录 dir2 不存在，创建目录 dir2，并且移动目录 dir1 的内容到目录 dir2 中，同 dir1。如果目录 dir2 存在，移动目录 dir1 (及它的内容) 到目录 dir2。 |

## mkdir 目录名称 — 创建目录

touch 用来设置或更新文件的访问，更改，和修改时间。如果一个文件名参数是一个不存在的文件，则会创建一个空文件。

## rm — 删除文件和目录

| 选项              | 意义                                                        |
|-----------------|-----------------------------------------------------------|
| -r, --recursive | 递归地删除文件，这意味着，如果要删除一个目录，而此目录又包含子目录，那么子删除。要删除一个目录，必须指定这个选项。 |
| -f, --force     | 忽视不存在的文件，不显示提示信息。这选项覆盖了“--interactive”选项。                 |
| -v, --verbose   | 在执行 rm 命令时，显示操作信息                                         |

## ln — 创建硬链接和符号链接

**Inode:**当划分磁盘分区并格式化的时候，整个分区会被划分为两个部分，即Inode区和data block(实际数据放置在数据区域中)这个inode即是（目录、档案）文件在一个文件系统中的唯一标识，需要访问这个文件的时候必须先找到文件的 inode。 Inode 里面存储了文件的很多重要参数，其中唯一标识称作 Inumber, 其他信息还有创建时间 (ctime) 、修改时间(mtime) 、文件大小、属主、归属的用户组、读写权限、数据所在block号等信息。



操作一个文件的流程：目录inode（满足权限？） => 目录block => 档案inode（满足权限？） => 档案block

### 硬链接：

在另外的目录或本目录中增加目标文件的一个目录项，这样，一个文件就登记在多个目录中。已经存在的文件的 Inode 会被多个目录文件项使用。一个文件的硬链接数可以在目录的长列表格式的第二列中看到



硬链接只是在某个目录下新增一笔档名链接到某个inode号码的关联记录而已。如果将上图中任何一个档名删除，档案的inode与block都还存在，依然还可以通过另一个档名来读取正确的档案数据。此外，不论用哪一个档名来编辑，最终的结果都会写入相同的inode和block中，因此均能进行数据的修改。

对硬链接有如下限制：

- 不能对目录文件做硬链接。
- 不能在不同的文件系统之间做硬链接。就是说，链接文件和被链接文件必须位于同一个文件系统中。

### 符号链接：-s

符号链接也称为软链接，是将一个路径名链接到一个文件。这些文件是一种特别类型的文件。事实上，它只是一个文本文件，包含它提供链接的另一个文件的路径名，它是一个新文件



- 删除源文件或目录，只删除了数据，不会删除链接。一旦以同样文件名创建了源文件，链接将继续指向该文件的新数据。
- 在目录长列表中，符号链接作为一种特殊的文件类型显示出来，其第一个字母是l。
- 符号链接的大小是其链接文件的路径名中的字节数。

**type 命令名** 显示命令的类型

**which 程序名** 显示可执行程序的位置

**man 程序名** 显示程序手册页

**aprpos 关键字** 显示匹配关键字的手册列表

**whatis 关键字** 显示匹配关键字的手册页的名字和一行命令说明

**info GNU** 项目提供了一个命令程序手册页的替代物，称为“info”

**alias 命令名='命令'** 自定义命令，删除:unalias, 不加参数即为查看所有别名

在文件中配置：~/.bashrc

| 通配符           | 意义                |
|---------------|-------------------|
| *             | 匹配任意多个字符（包括零个或一个） |
| ?             | 匹配任意一个字符（不包括零个）   |
| [characters]  | 匹配任意一个属于字符集中的字符   |
| [!characters] | 匹配任意一个不是字符集中的字符   |
| [:class:]     | 匹配任意一个属于指定字符类中的字符 |

常用字符类

| 字符类       | 意义          |
|-----------|-------------|
| [:alnum:] | 匹配任意一个字母或数字 |
| [:alpha:] | 匹配任意一个字母    |
| [:digit:] | 匹配任意一个数字    |
| [:lower:] | 匹配任意一个小写字母  |
| [:upper:] | 匹配任意一个大写字母  |

标准的输入输出 是键盘输入，屏幕输出

0 通常是标准输入（STDIN），1是标准输出（STDOUT），2是标准错误输出（STDERR）。

标准输出重定向：命令 > 指定输出文件

小技巧：创建或者清空一个文件：> 文件名

如果命令出错不会输出到输出文件，因为这属于标准错误，而不是标准输出

错误输出重定向 2> (没有错误例如 ls > err.txt, 则err.txt为空，标准输出仍显示在屏幕)

使用”>>“操作符，将输出结果追加到文件内容之后。如果文件不存在，创建该文件

&> 来重定向标准输出和错误

**cat** 命令读取一个或多个文件，然后复制它们到标准输出

-n 显示行数 -s 禁止输出多个空白行

如果没有参数，会等待标准输出（键盘）输入文本

小技巧：创建一个短文本： cat > 文件名

使用管道操作符”|“（竖杠），一个命令的标准输出可以通过管道送至另一个命令的标准输入

例：ls -l /usr/bin | less ,把ls的输出作为less命令的输入

sort 排序，可以输入多个文件

|    |                         |                                                            |
|----|-------------------------|------------------------------------------------------------|
| -b | --ignore-leading-blanks | 默认情况下，对整行进行排序，从每行的第一个字符开始。这个选项导致程序忽略每行开头的空格，从第一个非空白字符开始排序。 |
| -f | --ignore-case           | 让排序不区分大小写。                                                 |
| -n | --numeric-sort          | 基于字符串的数值来排序。使用此选项允许根据数字值执行排序，而不是值。                         |
| -r | --reverse               | 按相反顺序排序。结果按照降序排列，而不是升序。                                    |
| -k | --key=field1[,field2]   | 对从 field1 到 field2 之间的字符排序，而不是整个文本行。看下面的讨论。                |
| -m | --merge                 | 把每个参数看作是一个预先排好序的文件。把多个文件合并成一个排好序的文件，而没有执行额外的排序。            |
| -o | --output=file           | 把排好序的输出结果发送到文件，而不是标准输出。                                    |
| -t | --field-separator=char  | 定义域分隔字符。默认情况下，域由空格或制表符分隔。                                  |

uniq 去重

|      |                                                                                |
|------|--------------------------------------------------------------------------------|
| -c   | 输出所有的重复行，并且每行开头显示重复的次数。                                                        |
| -d   | 只输出重复行，而不是特有的文本行。                                                              |
| -f n | 忽略每行开头的 n 个字段，字段之间由空格分隔，正如 sort 程序中的空格分隔符；然而，不同于 sort 程序，uniq 没有选项来设置备用的字段分隔符。 |
| -i   | 在比较文本行的时候忽略大小写。                                                                |
| -s n | 跳过（忽略）每行开头的 n 个字符。                                                             |
| -u   | 只输出独有的文本行。这是默认的。                                                               |

**cut** 用来从文本行中抽取文本，并把其输出到标准输出

cut 程序选择项

|               |                                                              |
|---------------|--------------------------------------------------------------|
| -c char_list  | 从文本行中抽取由 char_list 定义的文本。这个列表可能由一个或多个逗号分隔开的数值区间组成。           |
| -f field_list | 从文本行中抽取一个或多个由 field_list 定义的字段。这个列表可能包括一个或多个字段，或由逗号分隔开的字段区间。 |
| -d delim_char | 当指定-f 选项之后，使用 delim_char 做为字段分隔符。默认情况下，字段之间必须由单个 tab 字符分隔开。  |
| --complement  | 抽取整个文本行，除了那些由-c 和 / 或-f 选项指定的文本。                             |

**wc** 显示文件的 行数 字数 字节数

**grep** 在文本中搜索匹配行， -i 忽略大小写， -v 显示不匹配行

例： ls /bin /usr/bin | sort | uniq | grep zip

**head/tail** 打印前 / 后十行， -n 配置打印几行， tail -f 追踪文件

**tee** 程序从标准输入读入数据，并且同时复制数据到标准输出（允许数据继续随着管道线流动）一个或多个文件。

## 展开

**路径名展开**: echo D\* ,输出当前目录下D开头的的文件和目录名

**波浪线展开**: echo ~， 把波浪线展开为家目录名

**算术表达式展开**: echo \$((2+2)) 4

**花括号展开**: echo Number\_{1..5}

Number\_1 Number\_2 Number\_3 Number\_4 Number\_5

例： 创建一系列的文件和目录列表

**参数展开**: echo \$USER zb

命令替换： ls -l \$(which cp) 把 which cp 的执行结果作为一个参数传递给 ls 命令

## 引用

**双引号**： 除了 \$, \(反斜杠) , 和 ` (倒引号) , 其他特殊符号都会失效

**单引号**： 全部失效

光标移动命令

| 按键     | 行动                             |
|--------|--------------------------------|
| Ctrl-a | 移动光标到行首。                       |
| Ctrl-e | 移动光标到行尾。                       |
| Ctrl-l | 清空屏幕，移动光标到左上角。clear 命令完成同样的工作。 |

剪切和粘贴命令

| 按键     | 行动               |
|--------|------------------|
| Ctrl-k | 剪切从光标位置到行尾的文本。   |
| Ctrl-u | 剪切从光标位置到行首的文本。   |
| Ctrl-y | 把剪切环中的文本粘贴到光标位置。 |

**clear** 清屏

**history** 显示历史列表， 默认500条

历史展开命令

| 序列       | 行为                                  |
|----------|-------------------------------------|
| !!       | 重复最后一次执行的命令。可能按下上箭头按键和 enter 键更容易些。 |
| !number  | 重复历史列表中第 number 行的命令。               |
| !string  | 重复最近历史列表中，以这个字符串开头的命令。              |
| !?string | 重复最近历史列表中，包含这个字符串的命令。               |

ctrl+r 进入反向搜索模式

## 权限

ls下 第一个字符代表文件类型

| 属性 | 文件类型                                                                   |
|----|------------------------------------------------------------------------|
| -  | 一个普通文件                                                                 |
| d  | 一个目录                                                                   |
| l  | 一个符号链接。注意对于符号链接文件，剩余的文件属性总是"rwxrwxrwx"，而且都是虚拟值。真正的文件属性是指符号链接所指向的文件的属性。 |
| c  | 一个字符设备文件。这种文件类型是指按照字节流来处理数据的设备。比如说终端机或者调制解调器                           |
| b  | 一个块设备文件。这种文件类型是指按照数据块来处理数据的设备，例如一个硬盘或者 CD-ROM 盘。                       |

其余九个

| Owner | Group | World |
|-------|-------|-------|
| rwx   | rwx   | rwx   |

权限属性

|   |                                                  |                         |
|---|--------------------------------------------------|-------------------------|
| r | 允许打开并读取文件内容。                                     | 允许列出目录中的内容，前提是目录必须设置`   |
| w | 允许写入文件内容或截断文件。但是不允许对文件进行重命名或删除，重命名或删除是由目录的属性决定的。 | 允许在目录下新建、删除或重命名文件，前提是   |
| x | 允许将文件作为程序来执行，使用脚本语言编写的程序必须设置为可读才能被执行。            | 允许进入目录，例如：cd directory。 |

权限属性示例

|            |                                                                         |
|------------|-------------------------------------------------------------------------|
| -rwx----   | 一个普通文件，对文件所有者来说可读、可写、可执行。其他人无法访问。                                       |
| -rw-----   | 一个普通文件，对文件所有者来说可读可写。其他人无法访问。                                            |
| -rw-r--r-- | 一个普通文件，对文件所有者来说可读可写，文件所有者的组成员可以读该文件，其他用户可以读该文件。                         |
| -rwxr-xr-x | 一个普通文件，对文件所有者来说可读、可写、可执行。也可以被其他的任何人读取和执行。                               |
| -rw-rw---- | 一个普通文件，对文件所有者以及文件所有者的组成员来说可读可写。                                         |
| lrwxrwxrwx | 一个符号链接，符号链接的权限都是虚拟的，真实的权限应该以符号链接指向的文件为准。                                |
| drwxrwx--- | 一个目录，文件所有者以及文件所有者的组成员可以访问该目录，并且可以在该目录下新建、重命名、删除文件。                      |
| drwxr-x--- | 一个目录，文件所有者可以访问该目录，并且可以在该目录下新建、重命名、删除文件，其他用户的组成员可以访问该目录，但是不能新建、重命名、删除文件。 |

### 8进制数字权限设置

| Octal | Binary | File Mode |
|-------|--------|-----------|
| 0     | 000    | ---       |
| 1     | 001    | --x       |
| 2     | 010    | -w-       |
| 3     | 011    | -wx       |
| 4     | 100    | r--       |
| 5     | 101    | r-x       |
| 6     | 110    | rw-       |
| 7     | 111    | rwx       |

**chmod 权限 (三位0-7的数字) 文件名**

**-R 递归更改目录所有子文件**

命令符号表示法

|   |                              |
|---|------------------------------|
| u | "user"的简写，意思是文件或目录的所有者。      |
| g | 用户组。                         |
| o | "others"的简写，意思是其他所有的人。       |
| a | "all"的简写，是"u", "g"和"o"三者的联合。 |

chmod 符号表示法实例

|           |                                                    |
|-----------|----------------------------------------------------|
| u+x       | 为文件所有者添加可执行权限。                                     |
| u-x       | 删除文件所有者的可执行权限。                                     |
| +x        | 为文件所有者，用户组，和其他所有人添加可执行权限。等价于 a+x。                  |
| o-rw      | 除了文件所有者和用户组，删除其他人的读权限和写权限。                         |
| go=rw     | 给群组的主人和任意文件拥有者的人读写权限。如果群组的主人或全局之前已经有了执行的权限，他们将被移除。 |
| u+x,go=rw | 给文件拥有者执行权限并给组和其他人读和执行的权限。多种设定可以用逗号分开。              |

特殊情况：

**1 setuid(4000)**

当应用到一个可执行文件时，它把有效用户 ID 从真正的用户（实际运行程序的用户）设置成程序所有者的 ID。这种操作通常会应用到一些由超级用户所拥有的程序。当一个普通用户运行一个程

序，这个程序由根用户(root)所有，并且设置了 setuid 位，这个程序运行时具有超级用户的特权，这样程序就可以访问普通用户禁止访问的文件和目录。

**chmod u+s 程序名**

## 2 setgid(2000)

这个相似于 setuid 位，把有效用户组 ID 从真正的 用户组 ID 更改为文件所有者的组 ID。如果设置了一个目录的 setgid 位，则目录中新创建的文件 具有这个目录用户组的所有权，而不是文件创建者所属用户组的所有权。

**chmod g+s 目录**

## 默认权限 umask

|                    |                 |
|--------------------|-----------------|
| Original file mode | --- rw- rw- rw- |
| Mask               | 000 000 010 010 |
| Result             | --- rw- r-- r-- |

## chown 改变文件的所有者和用户组

|            |                                          |
|------------|------------------------------------------|
| bob        | 把文件所有者从当前属主更改为用户 bob。                    |
| bob:user:s | 把文件所有者改为用户 bob，文件用户组改为用户组 users。         |
| :admins    | 把文件用户组改为组 admins，文件所有者不变。                |
| bob:       | 文件所有者改为用户 bob，文件用户组改为用户 bob 登录系统时所属的用户组。 |

**sudo** 以另一个用户身份执行命令（通常是管理员），使用时密码使用当前用户自己的密码

## 进程

**ps** 查看进程快照 加 -x 展示所有进程的信息，不管他们由什么终端控制

其中有一列 STAT，即为state，表示进程当前的状态

|   |                                                                                                                       |
|---|-----------------------------------------------------------------------------------------------------------------------|
| R | 运行中。这意味着，进程正在运行或准备运行。                                                                                                 |
| S | 正在睡眠。进程没有运行，而是，正在等待一个事件，比如说，一个按键或者网络分组。                                                                               |
| D | 不可中断睡眠。进程正在等待 I/O，比方说，一个磁盘驱动器的 I/O。                                                                                   |
| T | 已停止。已经指示进程停止运行。稍后介绍更多。                                                                                                |
| Z | 一个死进程或“僵尸”进程。这是一个已经终止的子进程，但是它的父进程还没有清空它。（父进程没有把子进程从进程表中删除）                                                            |
| < | 一个高优先级进程。这可能会授予一个进程更多的资源，给它更多的 CPU 时间。进程的这种属性叫做 niceness。具有高优先级的进程据说是不好的（less nice），因为它占用了比较多的 CPU 时间，这样就给其它进程留下很少时间。 |
| N | 低优先级进程。一个低优先级进程（一个“好”进程）只有当其它高优先级进程被服务了之后，才会得到处理器时间。                                                                  |

还有一种做法是加 aux，会得到一些额外的列

|       |                            |
|-------|----------------------------|
| USER  | 用户 ID. 进程的所有者。             |
| %CPU  | 以百分比表示的 CPU 使用率            |
| %MEM  | 以百分比表示的内存使用率               |
| VSZ   | 虚拟内存大小                     |
| RSS   | 进程占用的物理内存的大小, 以千字节为单位。     |
| START | 进程启动的时间。若它的值超过24小时, 则用天表示。 |

**top** 程序以进程活动顺序显示连续更新的系统进程列表, 结果有两部分组成, 上面为系统概要, 下面是进程列表

把一个进程放置到后台后面加 &

**jobs** 查看当前终端启动的任务, 第一列为任务序号

**fg %任务序号** 将进程返回到前台,

**bg %任务序号** 将进程放置到后台, 若之前为停止状态则继续执行

然后ctrl + c 终止任务, 或者 ctrl + z 停止任务

**kill [-信号] pid或任务序号** 给进程发送信号。

常用信号

| 编号 | 名字   | 含义                                                                                                                            |
|----|------|-------------------------------------------------------------------------------------------------------------------------------|
| 1  | HUP  | 在当前终端运行的前台程序将会收到这个信号并终止。许多守护进程也使用这个信号来重新初始化。这意味着, 当一个守护进程收到这个信号后, 这个进程会重新启动, 且重新读取它的配置文件。Apache 网络服务器守护进程就是一个例子。              |
| 2  | INT  | 中断。实现和 Ctrl-c 一样的功能, 由终端发送。通常, 它会终止一个程序。                                                                                      |
| 9  | KILL | 杀死。这个信号很特别。KILL 信号从不被发送到目标程序。而是内核立即终止这个进程。当一个进程以这种方式终止的时候, 它没有机会去做些“清理”工作, 或者是保存工作。因为这个原因, 把 KILL 信号看作最后一招, 当其它终止信号失败后, 再使用它。 |
| 15 | TERM | 终止。这是 kill 命令发送的默认信号。如果程序仍然“活着”, 可以接受信号, 那么这它会终止。                                                                             |
| 18 | CONT | 继续。在一个停止信号后, 这个信号会恢复进程的运行。                                                                                                    |
| 19 | STOP | 停止。这个信号导致进程停止运行, 而不是终止。像 KILL 信号, 它不被发送到目标程序, 因此它不能被忽略。                                                                       |

**killall [-u user] [-signal] name...** 通过 killall 命令给多个进程发送信号

## vim

i 进入插入模式 esc 退出插入模式

:w 保存 :q 退出 :wq/ZZ 保存并退出 :q! 强制退出

A 将光标移到行尾并进入插入模式

0 将光标移到行首

o 当前行的下方插入一行

O 当前行的上方插入一行

u 撤销

## 删除 (剪切)

| 命令   | 删除的文本                |
|------|----------------------|
| x    | 当前字符                 |
| 3x   | 当前字符及其后的两个字符。        |
| dd   | 当前行。                 |
| 5dd  | 当前行及随后的四行文本。         |
| dW   | 从光标位置开始到下一个单词的开头。    |
| d\$  | 从光标位置开始到当前行的行尾。      |
| d0   | 从光标位置开始到当前行的行首。      |
| d^   | 从光标位置开始到文本行的第一个非空字符。 |
| dG   | 从当前行到文件的末尾。          |
| d20G | 从当前行到文件的第20行。        |

## 复制

|      |                      |
|------|----------------------|
| yy   | 当前行。                 |
| 5yy  | 当前行及随后的四行文本。         |
| yW   | 从当前光标位置到下一个单词的开头。    |
| y\$  | 从当前光标位置到当前行的末尾。      |
| y0   | 从当前光标位置到行首。          |
| y^   | 从当前光标位置到文本行的第一个非空字符。 |
| yG   | 从当前行到文件末尾。           |
| y20G | 从当前行到文件的第20行。        |

**p** 黏贴 **P** 到光标前 **P** 到光标后

**gg** 光标移到文件首行行首

**G** 光标移到文件末尾

**J** 连接下一行

**查找一行 (光标后) f 要搜索的字符, ; 重复搜索**

**全局查找替换**

:%s/Line/line/g

| 条目         | 含义                                                                                                                                |
|------------|-----------------------------------------------------------------------------------------------------------------------------------|
| :          | 冒号字符运行一个 ex 命令。                                                                                                                   |
| %          | 指定要操作的行数。% 是一个快捷方式, 表示从第一行到最后一行。另外, 操作范围也可以用 1,5 来代替 (因为我们的文件只有5行文本), 或者用 1,\$ 来代替, 意思是 “从第一行到文件的最后一行。” 如果省略了文本行的范围, 那么操作只对当前行生效。 |
| s          | 指定操作。在这种情况下是, 替换 (查找与替代) 。                                                                                                        |
| /Line/line | 查找类型与替代文本。                                                                                                                        |
| g          | 这是“全局”的意思, 意味着对文本行中所有匹配的字符串执行查找和替换操作。如果省略 g, 则 只替换每个文本行中第一个匹配的字符串。                                                                |

我们也可以指定一个需要用户确认的替换命令。通过添加一个“c”字符到这个命令的末尾  
替换确认按键

| 按键                | 行为                         |
|-------------------|----------------------------|
| y                 | 执行替换操作                     |
| n                 | 跳过这个匹配的实例                  |
| a                 | 对这个及随后所有匹配的字符串执行替换操作。      |
| q or esc          | 退出替换操作。                    |
| l                 | 执行这次替换并退出。l是“last”的简写。     |
| Ctrl-e,<br>Ctrl-y | 分别是向下滚动和向上滚动。用于查看建议替换的上下文。 |

编辑多个文件 vi file1 file2 ....

文件切换： 下一个n 上一个N

使用**:buffers** 命令。运行这个命令后，屏幕顶部就会显示出一个文件列表

**:buffer 编号(:buffers第一列)** 切换文件

**:e 文件名** 添加其他文件（无法用n/N 切换文件，只能用buffer）

**:r 文件名** 把整个文件插入到我们正在编辑的文件中

**scp 用户名@服务器地址:服务器文件** 安全复制服务器文件到本地

**scp 本地文件 用户名@服务器地址:服务器路径** 安全复制本地文件到服务器

**locate 路径或文件名** 执行一次快速的路径名数据库搜索，并且输出每个与给定子字符串相匹配的路径名。

## 解压

**unzip -d** 解压目录 zip文件

## 1 console.log([]==![]);

答：!比==先算,空数组是一个对象所以右边的值变成了false

`[] == false;`

当值类型的数据和引用类型的数据进行运算（算数运算，关系运算）的时候，会遵守如下的步骤。

1. 会调用引用类型数据的valueOf方法，获取返回值，尝试和值类型的数据进行运算，如果可以计算，就得出结果
2. 如果不能计算，就继续调用这个引用类型数据的toString方法，获取返回值，进行计算！

`[] .toString() // ""`

空字符串和false比较，自然就返回了true

作者：会打dota的程序猿 来源：知乎