# JavaScript

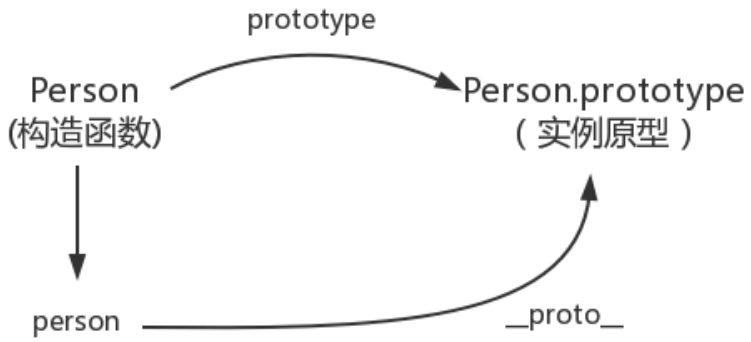
## 原型链

引用

每个函数都有一个 prototype 属性.

函数的 prototype 属性指向了一个对象，这个对象正是调用该构造函数而创建的**实例**的原型.

每个实例对象会继承原型上的属性.



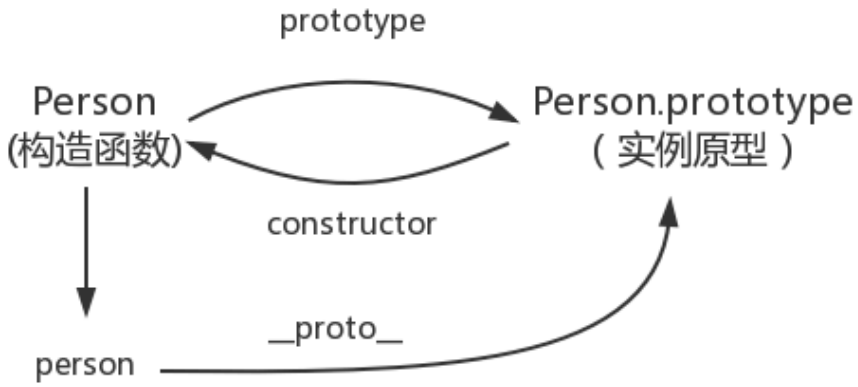

既然实例对象和构造函数都可以指向原型，那么原型是否有属性指向构造函数或者实例呢？

指向实例倒是没有，因为一个构造函数可以生成多个实例，但是原型指向构造函数倒是有的，这就要讲到第三个属性：`constructor`，每个原型都有一个 `constructor` 属性指向关联的构造函数。

```
function Person() {}

console.log(Person.prototype.constructor === Person) // true
```

更新下关系图：




```
function Person() {

}

var person = new Person();

console.log(person.__proto__ == Person.prototype) // true
console.log(Person.prototype.constructor == Person) // true
// 顺便学习一个ES5的方法,可以获得对象的原型
console.log(Object.getPrototypeOf(person) === Person.prototype) // true
```

## 实例与原型

当获取实例属性时, 如果找不到, 就会查找与对象关联的原型上的属性. 如果原型找不到, 会继续往原型的原型上找. 直到最顶层为止.

```
function Person() {

}

Person.prototype.name = 'Kevin';

var person = new Person();

person.name = 'Daisy';
console.log(person.name) // Daisy

delete person.name;
console.log(person.name) // Kevin
```

## 原型的原型

是Object.prototype.

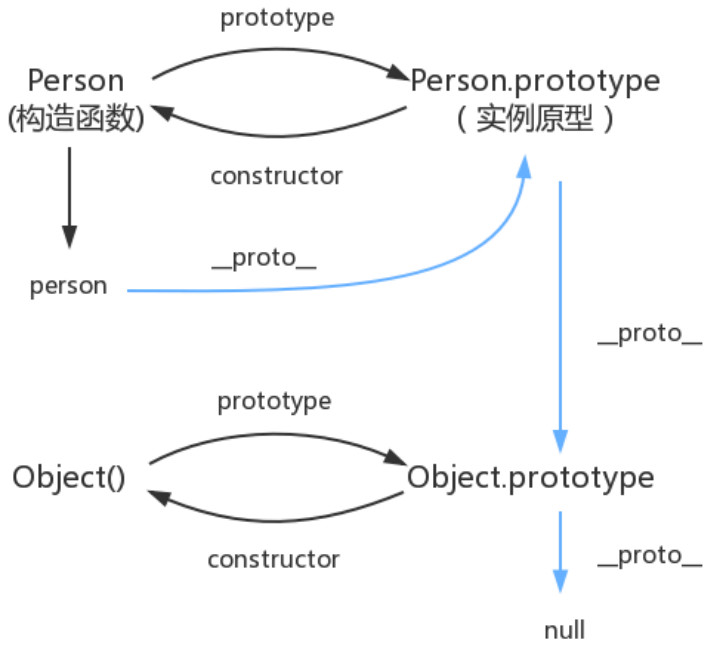
```
Person.prototype.__proto__ === Object.prototype // true
```

## 原型链

那 Object.prototype 的原型呢？

null

```
Object.prototype.__proto__ === null // true
```



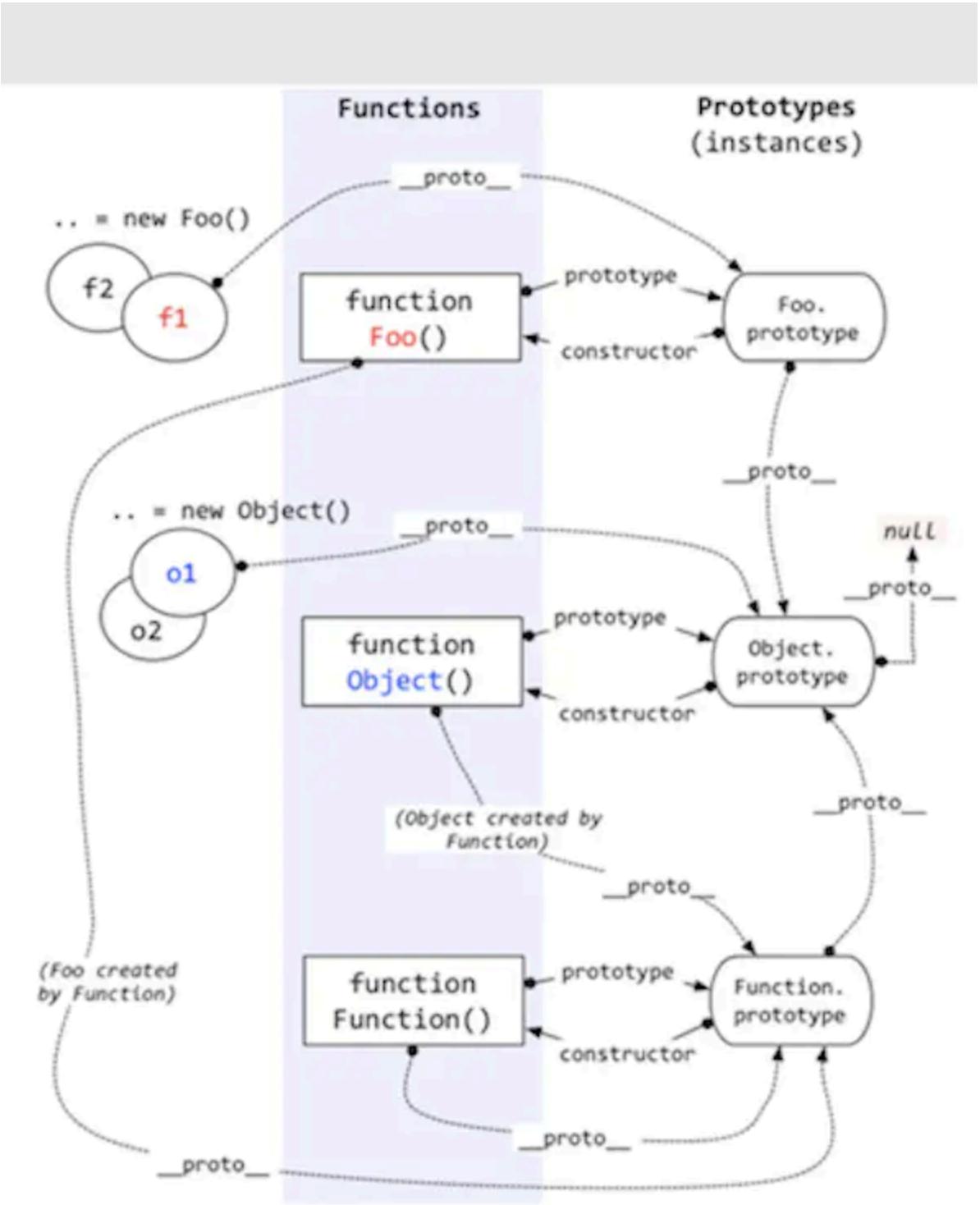

由相互关联的原型组成的链状结构就是原型链，也就是蓝色的这条线

## 补充

最后是关于继承，前面我们讲到“每一个对象都会从原型‘继承’属性”，实际上，继承是一个十分具有迷惑性的说法，引用《你不知道的JavaScript》中的话，就是：

继承意味着复制操作，然而 JavaScript 默认并不会复制对象的属性，相反，JavaScript 只是在两个对象之间创建一个关联，这样，一个对象就可以通过委托访问另一个对象的属性和函数，所以与其叫继承，委托的说法反而更准确些。

一个函数的构造函数是Function




## 总结

原型链是相互关联的原型组成的链状结构. 当获取实例属性时, 如果找不到, 就会查找与对象关联的原型上的属性. 如果原型找不到, 会继续往原型的原型上找. 直到最顶层为止. 顶层原型是null.

# 继承

引用

## 1.原型链继承

```
function Parent () {
    this.names = ['kevin', 'daisy'];
}

Parent.prototype.getName = function () {
    console.log(this.names);
}

function Child () {}

Child.prototype = new Parent();

var child1 = new Child();
console.log(child1.getName());

// 问题
// 1. 引用类型的属性被共享
var child2 = new Child();
child1.names.push('keke');
console.log(child2.names);

// 2.在创建 Child 的实例时，不能向Parent传参
```

## 2.借用构造函数(继承)

优点：

1.避免了引用类型的属性被所有实例共享

2.可以在 Child 中向 Parent 传参

```
function Parent (name) {
    this.name = name;
}

function Child (name) {
    // 构造函数
    Parent.call(this, name);
}

var child1 = new Child(['kevin']);

console.log(child1.name); // ['kevin']

var child2 = new Child(['daisy']);

console.log(child2.name); // ['daisy']
```

缺点：

方法都在构造函数中定义，每次创建实例都会创建一遍方法。

```
function Parent (name) {
    this.name = name;
    this.getName = function () { console.log(this.name);}
}
// Parent.prototype.getName = function () { console.log(this.name);}


function Child (name) {
    // 构造函数
    Parent.call(this, name);
}

var child1 = new Child(['kevin']);
```

## 3.组合继承

```
function Parent (name) {
    this.name = name;
    this.colors = ["red", "blue", "green"]
}

Parent.prototype.getName = function () { console.log(this.name) }

function Child (name, age) {
        Parent.call(this, name)
    this.age = age
}

// Child.prototype = Parent.prototype // 导致Parent.prototype.constructor修改
Child.prototype = new Parent()
Child.prototype.constructor = Child

var child1 = new Child('kevin', '18');

child1.colors.push('black');

console.log(child1.name); // kevin
console.log(child1.age); // 18
console.log(child1.colors); // ["red", "blue", "green", "black"]

var child2 = new Child('daisy', '20');

console.log(child2.name); // daisy
console.log(child2.age); // 20
console.log(child2.colors); // ["red", "blue", "green"]
```

优点：融合原型链继承和构造函数的优点，是 JavaScript 中最常用的继承模式。

缺点: 会调用两次父构造函数。

一次是设置子类型实例的原型的时候：

```
Child.prototype = new Parent();
```

一次在创建子类型实例的时候：

```
var child1 = new Child('kevin', '18');
```

升级 => 6. 寄生组合式继承 (Child.prototype 间接继承 Parent.prototype)

## 4.原型式继承

```
function createObj(o) {
    function F(){}
    F.prototype = o;
    return new F();
}
```

就是 ES5 Object.create 的模拟实现，将传入的对象作为创建的对象的原型。

缺点：

包含引用类型的属性值始终都会共享相应的值，这点跟原型链继承一样。

```
var person = {
    name: 'kevin',
    friends: ['daisy', 'kelly']
}

var person1 = createObj(person);
var person2 = createObj(person);

person1.name = 'person1';
console.log(person2.name); // kevin

person1.friends.push('taylor');
console.log(person2.friends); // ["daisy", "kelly", "taylor"]
```

注意：修改`person1.name`的值，`person2.name`的值并未发生改变，并不是因为`person1`和`person2`有独立的 name 值，而是因为`person1.name = 'person1'`，给`person1`添加了 name 值，并非修改了原型上的 name 值。

## 5. 寄生式继承

创建一个仅用于封装继承过程的函数，该函数在内部以某种形式来做增强对象，最后返回对象。

```
function createObj (o) {
    var clone = Object.create(o);
    clone.sayName = function () {
        console.log('hi');
    }
    return clone;
}
```

缺点：跟借用构造函数模式一样，每次创建对象都会创建一遍方法。

## 6. 寄生组合式继承

```
function Parent (name) {
    this.name = name;
    this.colors = ["red", "blue", "green"]
}

Parent.prototype.getName = function () { console.log(this.name) }

function Child (name, age) {
        Parent.call(this, name)
    this.age = age
}


Child.prototype = Object.create(Parent.prototype)
Child.prototype.constructor = Child

var child1 = new Child('kevin', '18');

child1.colors.push('black');

console.log(child1.name); // kevin
console.log(child1.age); // 18
console.log(child1.colors); // ["red", "blue", "green", "black"]

var child2 = new Child('daisy', '20');

console.log(child2.name); // daisy
console.log(child2.age); // 20
console.log(child2.colors); // ["red", "blue", "green"]
```

Object.create模拟和封装

```
function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child (name, age) {
    Parent.call(this, name);
    this.age = age;
}

// ES5 Object.create 的模拟实现
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}

function prototype(child, parent) {
    var prototype = object(parent.prototype);
    prototype.constructor = child;
    child.prototype = prototype;
}

// 当我们使用的时候:
prototype(Child, Parent);


var child1 = new Child('kevin', '18');

console.log(child1);
```

引用《JavaScript高级程序设计》中对寄生组合式继承的夸赞就是：

这种方式的高效率体现它只调用了一次 Parent 构造函数，并且因此避免了在 Parent.prototype 上面创建不必要的、多余的属性。与此同时，原型链还能保持不变；因此，还能够正常使用 instanceof 和 isPrototypeOf。开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式。

## 总结

原型链继承: 方法是new一个父实例赋值给子原型对象. 缺点: 1.引用类型的属性被所有实例共享. 2.不能向父构造函数传参

构造函数继承: 方法是子构造函数中执行父的构造函数并call子的this. 优点: 1.避免了引用类型的属性被所有实例共享 2.可以向父构造函数传参. 缺点: 方法都在构造函数里定义. 并且每次创建实例都会创建一遍方法.

组合继承: 结合原型链继承和构造函数继承. 子原型构造函数要配置回来. 缺点: 会执行两遍父构造函数. 一次是设置子原型对象时候. 第二次是创建子实例的时候.

寄生组合式继承: 间接的让子原型能关联到父原型. 利用es5的Object.create 传入父原型. 赋值于子原型. 优点: 1.只执行一遍父构造函数. 2.避免了在父原型上创建不必要的属性. 原型链还能保持不变, 还能使用instanceof 和 isPrototypeOf.

# 作用域

作用域是指程序源代码中定义变量的区域。

作用域规定了如何查找变量，也就是确定当前执行代码对变量的访问权限。

JavaScript 采用词法作用域(lexical scoping)，也就是静态作用域。

**作用域链是在函数定义的时候创建的**

## 静态作用域(javascript)

因为 JavaScript 采用的是词法作用域，函数的作用域在函数定义的时候就决定了。

而与词法作用域相对的是动态作用域，函数的作用域是在函数调用的时候才决定的。

让我们认真看个例子就能明白之间的区别：

```
var value = 1;

function foo() {
    console.log(value);
}

function bar() {
    var value = 2;
    foo();
}

bar();

// 结果是 ???  JavaScript采用的是静态作用域  1
```

假设JavaScript采用静态作用域，让我们分析下执行过程：

执行 foo 函数，先从 foo 函数内部查找是否有局部变量 value，如果没有，**就根据书写的位置，查找上面一层的代码**，也就是 value 等于 1，所以结果会打印 1。

假设JavaScript采用动态作用域，让我们分析下执行过程：

执行 foo 函数，依然是从 foo 函数内部查找是否有局部变量 value。如果没有，就从调用函数的作用域，也就是 bar 函数内部查找 value 变量，所以结果会打印 2。

前面我们已经说了，JavaScript采用的是静态作用域，所以这个例子的结果是 1。

## 动态作用域(bash)

也许你会好奇什么语言是动态作用域？

bash 就是动态作用域，不信的话，把下面的脚本存成例如 scope.bash，然后进入相应的目录，用命令行执行 `bash ./scope.bash`，看看打印的值是多少。

```
value=1
function foo () {
    echo $value;
}
function bar () {
    local value=2;
    foo;
}
bar
```

## 执行上下文栈

JavaScript 引擎创建了执行上下文栈（Execution context stack，ECS）来管理执行上下文

为了模拟执行上下文栈的行为，让我们定义执行上下文栈是一个数组：

```
ECStack = [];
```

试想当 JavaScript 开始要解释执行代码的时候，最先遇到的就是全局代码，所以初始化的时候首先就会向执行上下文栈压入一个全局执行上下文，我们用 globalContext 表示它，并且只有当整个应用程序结束的时候，ECStack 才会被清空，所以程序结束之前， ECStack 最底部永远有个 globalContext：

```
ECStack = [
    globalContext
];
```

现在 JavaScript 遇到下面的这段代码了：

```
function fun3() {
    console.log('fun3')
}

function fun2() {
    fun3();
}

function fun1() {
    fun2();
}

fun1();
```

当执行一个函数的时候，就会创建一个执行上下文，并且压入执行上下文栈，当函数执行完毕的时候，就会将函数的执行上下文从栈中弹出。知道了这样的工作原理，让我们来看看如何处理上面这段代码：

```
// 伪代码

// fun1()
ECStack.push(<fun1> functionContext);

// fun1中竟然调用了fun2，还要创建fun2的执行上下文
ECStack.push(<fun2> functionContext);

// 擦，fun2还调用了fun3!
ECStack.push(<fun3> functionContext);

// fun3执行完毕
ECStack.pop();

// fun2执行完毕
ECStack.pop();

// fun1执行完毕
ECStack.pop();

// javascript接着执行下面的代码，但是ECStack底层永远有个globalContext
```

**解答思考题**

好啦，现在我们已经了解了执行上下文栈是如何处理执行上下文的，所以让我们看看上篇文章《JavaScript深入之词法作用域和动态作用域》最后的问题：

```
var scope = "global scope";
function checkscope(){
    var scope = "local scope";
    function f(){
        return scope;
    }
    return f();
}
checkscope();
```

```
var scope = "global scope";
function checkscope(){
    var scope = "local scope";
    function f(){
        return scope;
    }
    return f;
}
checkscope()();
```

两段代码执行的结果一样，但是两段代码究竟有哪些不同呢？

答案就是执行上下文栈的变化不一样。

让我们模拟第一段代码：

```
ECStack.push(<checkscope> functionContext);
ECStack.push(<f> functionContext);
ECStack.pop();
ECStack.pop();
```

让我们模拟第二段代码：

```
ECStack.push(<checkscope> functionContext);
ECStack.pop();
ECStack.push(<f> functionContext);
ECStack.pop();
```

## 变量对象

变量对象是与执行上下文相关的数据作用域，存储了在上下文中定义的变量和函数声明。

因为不同执行上下文下的变量对象稍有不同，所以我们来聊聊全局上下文下的变量对象和函数上下文下的变量对象。

全局上下文中的变量对象就是全局对象.

**函数上下文**

在函数上下文中，我们用活动对象(activation object, AO)来表示变量对象。

活动对象和变量对象其实是一个东西，只是变量对象是规范上的或者说是引擎实现上的，不可在

JavaScript 环境中访问，只有到当进入一个执行上下文中，这个执行上下文的变量对象才会被激活，所以才叫 activation object 呐，而只有被激活的变量对象，也就是活动对象上的各种属性才能被访问。

活动对象是在进入函数上下文时刻被创建的，它通过函数的 arguments 属性初始化。
arguments 属性值是 Arguments 对象。

**函数执行过程**

执行上下文的代码会分成两个阶段进行处理：分析和执行，我们也可以叫做：

1. 分析阶段

> 在这个阶段中，执行上下文会分别创建变量对象，建立作用域链，以及确定this的指向。

2. 代码执行阶段

> 创建完成之后，就会开始执行代码，这个时候，会完成变量赋值，函数引用，以及执行其他代码。

**进入执行上下文**

**在进入执行上下文时，首先会处理函数声明，其次会处理变量声明**，如果如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性。

当进入执行上下文时，这时候还没有执行代码，

变量对象会包括：

1. 函数的所有形参 (如果是函数上下文)
    - 由名称和对应值组成的一个变量对象的属性被创建
    - 没有实参，属性值设为 undefined
2. 函数声明
    - 由名称和对应值（函数对象(function-object)）组成一个变量对象的属性被创建
    - 如果变量对象已经存在相同名称的属性，则完全替换这个属性
3. 变量声明
    - 由名称和对应值（undefined）组成一个变量对象的属性被创建；
    - 如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性

举个例子：

```
function foo(a) {
  var b = 2;
  function c() {}
  var d = function() {};

  b = 3;

}

foo(1);
```

在进入执行上下文后，这时候的 AO 是：

```
AO = {
    arguments: {
        0: 1,
        length: 1
    },
    a: 1,
    b: undefined,
    c: reference to function c(){},
    d: undefined
}
```

**代码执行**

在代码执行阶段，会顺序执行代码，根据代码，修改变量对象的值

还是上面的例子，当代码执行完后，这时候的 AO 是：

```
AO = {
    arguments: {
        0: 1,
        length: 1
    },
    a: 1,
    b: 3,
    c: reference to function c(){},
    d: reference to FunctionExpression "d"
}
```

## 总结

作用域在函数定义的时候已经创建.

函数执行过程:

1. 分析阶段: 执行上下文创建变量. 建立作用域链. 确定this指向.
2. 执行阶段. 变量赋值.函数引用, 执行其他代码.

到这里变量对象的创建过程就介绍完了，让我们简洁的总结我们上述所说：

1. 全局上下文的变量对象初始化是全局对象
2. 函数上下文的变量对象初始化只包括 Arguments 对象
3. 在进入执行上下文时会给变量对象添加形参、函数声明、变量声明等初始的属性值
4. 在代码执行阶段，会再次修改变量对象的属性值

# 作用域链

<<JavaScript深入之作用域链>>

当查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。这样由多个执行上下文的变量对象构成的链表就叫做作用域链。

下面，让我们以一个函数的创建和激活两个时期来讲解作用域链是如何创建和变化的。

## 函数创建

在《JavaScript深入之词法作用域和动态作用域》中讲到，函数的作用域在函数定义的时候就决定了。

这是因为函数有一个内部属性 scope，当函数创建的时候，就会保存所有父变量对象到其中，你可以理解 scope 就是所有父变量对象的层级链，但是注意：scope 并不代表完整的作用域链！

举个例子：

```
function foo() {
    function bar() {
        ...
    }
}
```

函数创建时，各自的scope为：

```
foo.[[scope]] = [
  globalContext.VO
];

bar.[[scope]] = [
    fooContext.AO,
    globalContext.VO
];
```

## 函数激活

当函数激活时，进入函数上下文，创建 VO/AO 后，就会将活动对象添加到作用链的前端。

这时候执行上下文的作用域链，我们命名为 Scope：

```
Scope = [AO].concat([[Scope]]);
```

至此，作用域链创建完毕。

## 捋一捋

以下面的例子为例，结合着之前讲的变量对象和执行上下文栈，我们来总结一下函数执行上下文中作用域链和变量对象的创建过程：

```
var scope = "global scope";
function checkscope(){
    var scope2 = 'local scope';
    return scope2;
}
checkscope();
```

1.checkscope 函数被创建，保存作用域链到 内部属性scope

```
checkscope.[[scope]] = [
    globalContext.VO
];
```

2.执行 checkscope 函数，创建 checkscope 函数执行上下文，checkscope 函数执行上下文被压入执行上下文栈

```
ECStack = [
    checkscopeContext,
    globalContext
];
```

3.checkscope 函数并不立刻执行，开始做准备工作，第一步：复制函数scope属性创建作用域链

```
checkscopeContext = {
    Scope: checkscope.[[scope]],
}
```

4.第二步：用 arguments 创建活动对象，随后初始化活动对象，加入形参、函数声明、变量声明

```
checkscopeContext = {
    AO: {
        arguments: {
            length: 0
        },
        scope2: undefined
    },
    Scope: checkscope.[[scope]],
}
```

5.第三步：将活动对象压入 checkscope 作用域链顶端

```
checkscopeContext = {
    AO: {
        arguments: {
            length: 0
        },
        scope2: undefined
    },
    Scope: [AO, [[Scope]]]
}
```

6.准备工作做完，开始执行函数，随着函数的执行，修改 AO 的属性值

```
checkscopeContext = {
    AO: {
        arguments: {
            length: 0
        },
        scope2: 'local scope'
    },
    Scope: [AO, [[Scope]]]
}
```

7.查找到 scope2 的值，返回后函数执行完毕，函数上下文从执行上下文栈中弹出

```
ECStack = [
    globalContext
];
```

checkscope函数创建的时候，保存的是根据词法所生成的作用域链，checkscope执行上下文的时候，会复制这个作用域链，作为自己作用域链的初始化，然后根据环境生成变量对象，然后将这个变量对象，添加到作用域链scope的顶端，这才完整的构建了自己的作用域链

## 总结

在源代码中当你定义（书写）一个函数的时候（并未调用），js引擎也能根据你函数书写的位置，函数嵌套的位置，基于词法作用域给你生成一个`[[scope]]`，作为该函数的属性存在（这个属性属于函数的）。即使函数不调用，所以说**基于词法作用域（静态作用域）**。

然后进入函数执行阶段，生成执行上下文，执行上下文你可以宏观的看成一个对象，（包含vo,scope,this），此时，执行上下文里的`scope`和之前属于函数的那个`[[scope]]`不是同一个，执行上下文里的scope，是在之前函数的`[[scope]]`的基础上，又新增一个`当前的AO对象`构成

的。

函数定义时候的scope和函数执行时候的scope，前者作为函数的属性，后者作为函数执行上下文的属性。

一个函数作用域链包含 [ 自身执行上下文AO(变量对象) ,所有父变量对象的层级链( `[[scope]]` ) ]

函数创建时, `[[scope]]` 保存基于词法作用域生成的父作用域链. 分析阶段(执行上下文). 复制 `[[scope]]` 创建自己作用域链. 然后根据环境生成变量对象, 并把变量对象加入作用域链顶端. 形成完整的作用域链.

# 闭包

JavaScript深入之闭包

ECMAScript中，闭包指的是：

1. 从理论角度：所有的函数。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
    1. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
    2. 在代码中引用了自由变量

```
var c = 1
function foo() {
    var a = 1
        return function() {
        console.log(c)
        a++;
        console.log(a)
    }
}
var b = foo()
```

## 分析

让我们先写个例子，例子依然是来自《JavaScript权威指南》，稍微做点改动：

```
var scope = "global scope";
function checkscope(){
    var scope = "local scope";
    function f(){
        return scope;
    }
    return f;
}

var foo = checkscope();
foo();
```

当我们了解了具体的执行过程后，我们知道 f 执行上下文维护了一个作用域链：

```
fContext = {
    Scope: [AO, checkscopeContext.AO, globalContext.VO],
}
```

对的，就是因为这个作用域链，f 函数依然可以读取到 checkscopeContext.AO 的值，说明当 f 函数引用了 checkscopeContext.AO 中的值的时候，即使 checkscopeContext 被销毁了，但是 JavaScript 依然会让 checkscopeContext.AO 活在内存中，f 函数依然可以通过 f 函数的作用域链找到它，正是因为 JavaScript 做到了这一点，从而实现了闭包这个概念。

## 必刷题

接下来，看这道刷题必刷，面试必考的闭包题：

```
var data = [];

for (var i = 0; i < 3; i++) {
  data[i] = function () {
    console.log(i);
  };
}

data[0]();
data[1]();
data[2]();
```

答案是都是 3，让我们分析一下原因：

当执行到 data[0] 函数之前，此时全局上下文的 VO 为：

```
globalContext = {
    VO: {
        data: [...],
        i: 3
    }
}
```

当执行 data[0] 函数的时候，data[0] 函数的作用域链为：

```
data[0]Context = {
    Scope: [AO, globalContext.VO]
}
```

data[0]Context 的 AO 并没有 i 值，所以会从 globalContext.VO 中查找，i 为 3，所以打印的结果就是 3。

data[1] 和 data[2] 是一样的道理。

所以让我们改成闭包看看：

```
var data = [];

for (var i = 0; i < 3; i++) {
  data[i] = (function (i) {
        return function(){
            console.log(i);
        }
  })(i);
}

data[0]();
data[1]();
data[2]();
```

当执行到 data[0] 函数之前，此时全局上下文的 VO 为：

```
globalContext = {
    VO: {
        data: [...],
        i: 3
    }
}
```

跟没改之前一模一样。

当执行 data[0] 函数的时候，data[0] 函数的作用域链发生了改变：

```
data[0]Context = {
    Scope: [AO, 匿名函数Context.AO globalContext.VO]
}
```

匿名函数执行上下文的AO为：

```
匿名函数Context = {
    AO: {
        arguments: {
            0: 0,
            length: 1
        },
        i: 0
    }
}
```

data[0]Context 的 AO 并没有 i 值，所以会沿着作用域链从匿名函数 Context.AO 中查找，这时候就会找 i 为 0，找到了就不会往 globalContext.VO 中查找了，即使 globalContext.VO 也有 i 的值(值为3)，所以打印的结果就是0。

data[1] 和 data[2] 是一样的道理。

## 总结

闭包是指那些能够访问自由变量的函数。自由变量是指在函数中使用的，但既不是函数参数也不是函数的局部变量的变量。

- 从理论角度: 所有函数都是闭包. 访问全局变量就是访问自由变量
- 从实践角度: 在创建它上下文(函数)都销毁了. 它仍然存在(比如从父函数中返回). 且引用了自由变量.

怎么获取的自由变量(父函数的)呢. 因为它作用域链保存着所有父变量对象. 从它的作用域链上读取到父变量对象.

# this指向

引用

## 从ECMAScript规范解读

如何确定this:

1. 计算 MemberExpression 的结果赋值给 ref
2. 判断 ref 是不是一个 Reference 类型
    1. ref 是 **Reference**，并且 **IsPropertyReference**(ref) 是 true,那么 this 的值为 GetBase(ref)
    2. ref 是 **Reference**, base value 值是 Environment Record, 那么this的值为 ImplicitThisValue(ref) , ImplicitThisValue(ref) 返回undefined
    3. ref 不是 Reference，那么 this 的值为 undefined, this 为 undefined，非严格模式下，this 的值为 undefined 的时候，其值会被隐式转换为全局对象。

其中MemberExpression的结果()左边的部分。

IsPropertyReference: 如果 base value 是一个**对象**(object)，就返回true

GetBase返回 base value; base value 就是属性所在的对象或者就是 EnvironmentRecord，它的值只可能是 undefined, an Object, a Boolean, a String, a Number, or an environment record 其中的一种。

**GetValue**: GetValue 返回对象属性真正的值，但是要注意：

**调用 GetValue，返回的将是具体的值，而不再是一个 Reference**.

reference类型例子:

```
var foo = 1;

// 对应的Reference是:
var fooReference = {
    base: EnvironmentRecord,
    name: 'foo',
    strict: false
};

function foo2() {
    console.log(this)
}

foo2();

var foo2Reference = {
    base: EnvironmentRecord, // EnvironmentRecord 不是object类型
    name: 'foo2',
    strict: false
};

GetValue(fooReference) // 1;
```

```
var foo = {
    bar: function () {
        return this;
    }
};

foo.bar(); // foo

// bar对应的Reference是:
var BarReference = {
    base: foo,
    propertyName: 'bar',
    strict: false
};
```

判断this:

```
var value = 1;

var foo = {
  value: 2,
  bar: function () {
    return this.value;
  }
}

//示例1  ==> foo.bar ==> reference => isPropertyReference(ref) base(foo)是对象 ==> this === foo ===
console.log(foo.bar());
//示例2  ===> () 并没有对 MemberExpression 进行计算 ===> foo.bar
console.log((foo.bar)());
//示例3  ===> 赋值操作符 ===> 使用GetValue ===> 不是reference类型 ===> 进入2-2 ===> undefined ===> 非
console.log((foo.bar = foo.bar)());
//示例4 ===> 逻辑与算法 ===> 使用GetValue
console.log((false || foo.bar)());
//示例5 ===> 逗号操作符 ===> 使用GetValue
console.log((foo.bar, foo.bar)());
```

```
function foo() {
  console.log(this)
}

var fooReference = {
    base: EnvironmentRecord, // EnvironmentRecord 不是object类型
    name: 'foo2',
    strict: false
};

foo();

// foo ===> MemberExpression 是 foo, 解析标识符 返回Reference ===> base值: EnvironmentRecord ===>  I
```

## 总结

从ECMAScript规范解读this:

如何判断this指向?

1. 先MemberExpression()计算赋值ref
2. 判断ref是不是Reference类型
    1. 如果是. 且isPropertyReference(ref) 为true, 即base值是object类型. 那

么this指向为GetBase(ref). 即base值
2. 如果是. base值为EnvironmentRecord. 那么this指向为ImplicitThisValue(ref) , ImplicitThisValue(ref) 返回undefined
3. 如果不是reference类型. this为undefined

其中MemberExpression的结果()左边的部分。

IsPropertyReference: 如果 base value 是一个对象(object)，就返回true

GetBase返回 base value; base value 就是属性所在的对象或者就是 EnvironmentRecord，它的值只可能是 undefined, an Object, a Boolean, a String, a Number, or an environment record 其中的一种。

GetValue: GetValue 返回对象属性真正的值，但是要注意：

调用 GetValue，返回的将是具体的值，而不再是一个 Reference.

MemberExpression特别例子:

带赋值操作符.逻辑运算.逗号操作符. 其中MemberExpression计算过程中会使用GetValue. MemberExpression计算结果返回值不是Reference类型. this为undefined

# 立即执行函数

在Javascript里，**圆括号不能包含声明**.

- 当圆括号出现在匿名函数的末尾想要调用函数时，它会默认将函数当成是函数声明。
- 当圆括号包裹函数时，它会默认将函数作为表达式去解析，而不是函数声明。

```
//这两种模式都可以被用来立即调用一个函数表达式，利用函数的执行来创造私有变量

(function(){/* code */}());//Crockford recommends this one，括号内的表达式代表函数立即调用表达式
(function(){/* code */})();//But this one works just as well，括号内的表达式代表函数表达式

// Because the point of the parens or coercing operators is to disambiguate
// between function expressions and function declarations, they can be
// omitted when the parser already expects an expression (but please see the
// "important note" below).

var i = function(){return 10;}();
true && function(){/*code*/}();
0,function(){}();

//如果你并不关心返回值，或者让你的代码尽可能的易读，你可以通过在你的函数前面带上一个一元操作符来存储字节

!function(){/* code */}();
~function(){/* code */}();
-function(){/* code */}();
+function(){/* code */}();

// Here's another variation, from @kuvos - I'm not sure of the performance
// implications, if any, of using the `new` keyword, but it works.
// http://twitter.com/kuvos/status/18209252090847232

new function(){ /* code */ }
new function(){ /* code */ }() // Only need parens if passing arguments
```

# instanceof原理

引用

## typeof 实现原理

`typeof` 一般被用于判断一个变量的类型，我们可以利用 `typeof` 来判断 `number`，`string`，`object`，`boolean`，`function`，`undefined`，`symbol` 这七种类型，这种判断能帮助我们搞定一些问题，比如在判断不是 object 类型的数据的时候，`typeof` 能比较清楚的告诉我们具体是哪一类的类型。但是，很遗憾的一点是，`typeof` 在判断一个 object的数据的时候只能告诉我们这个数据是 object, 而不能细致的具体到是哪一种 object

```
let s = new String('abc');
typeof s === 'object'// true
s instanceof String // true
```

要想判断一个数据具体是哪一种 object 的时候，我们需要利用 `instanceof` 这个操作符来判断

**js 在底层存储变量的时候，会在变量的机器码的低位1-3位存储其类型信息**☞

- 000：对象
- 010：浮点数
- 100：字符串
- 110：布尔
- 1：整数

but, 对于 `undefined` 和 `null` 来说，这两个值的信息存储是有点特殊的。

`null`：所有机器码均为0

`undefined`：用 −2^30 整数来表示

所以，**`typeof` 在判断 `null` 的时候就出现问题了，由于 `null` 的所有机器码均为0，因此直接被当做了对象来看待**。

然而用 `instanceof` 来判断的话☞

```
null instanceof null // TypeError: Right-hand side of 'instanceof' is not an object
复制代码
```

`null` 直接被判断为不是 object，这也是 JavaScript 的历史遗留bug，可以参考typeof。

还有一个不错的判断类型的方法，就是**Object.prototype.toString** ,我们可以利用这个方法来对一个变量的类型来进行比较准确的判断

```
Object.prototype.toString.call(1) // "[object Number]"

Object.prototype.toString.call('hi') // "[object String]"

Object.prototype.toString.call({a:'hi'}) // "[object Object]"

Object.prototype.toString.call([1,'a']) // "[object Array]"

Object.prototype.toString.call(true) // "[object Boolean]"

Object.prototype.toString.call(() => {}) // "[object Function]"

Object.prototype.toString.call(null) // "[object Null]"

Object.prototype.toString.call(undefined) // "[object Undefined]"

Object.prototype.toString.call(Symbol(1)) // "[object Symbol]"
```

## instanceof 操作符的实现原理

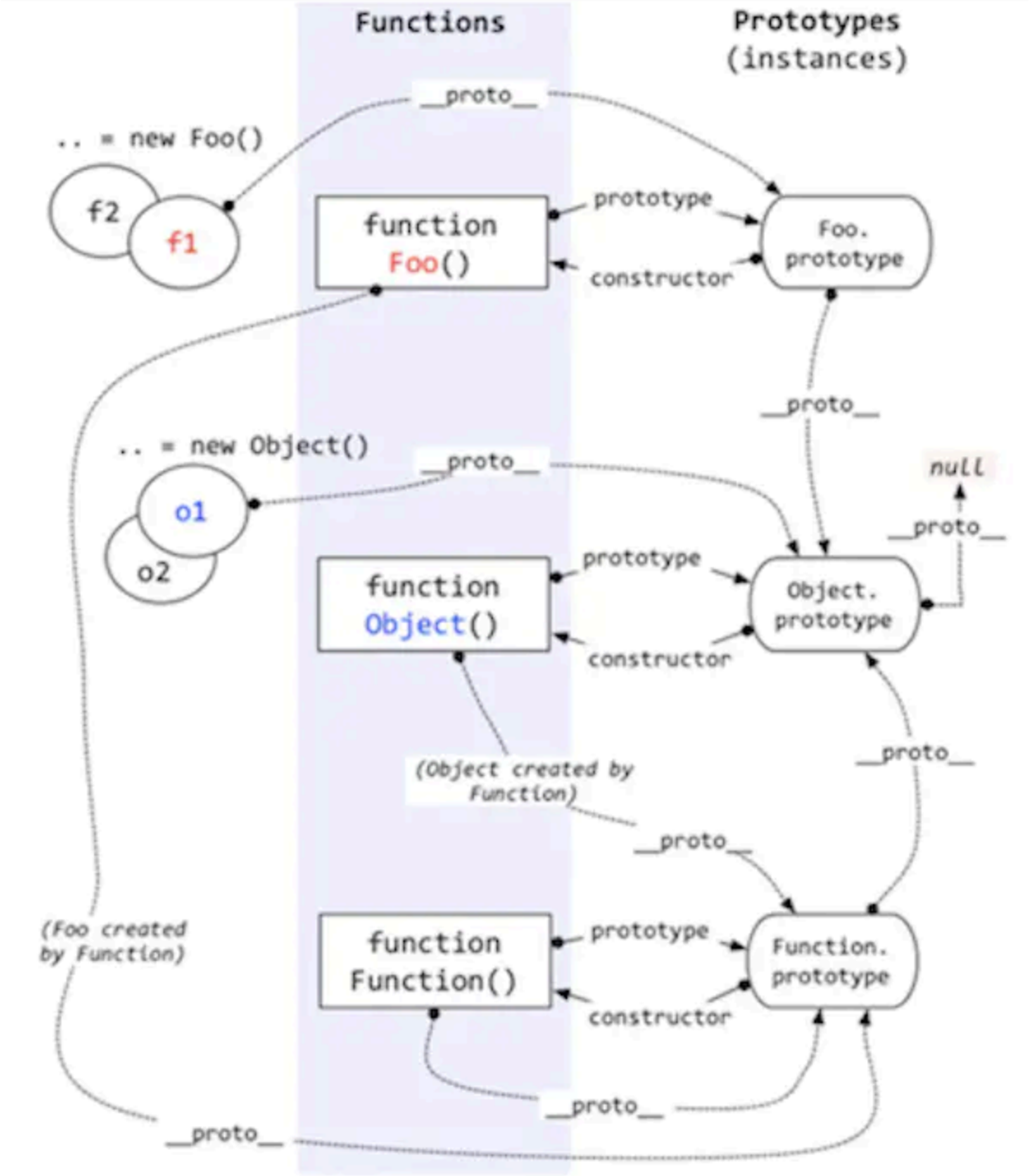
```
function new_instance_of(leftVaule, rightVaule) {
    if (typeof leftValue !== 'object' || typeof rightValue !== 'object') throw Error('Right-hand s
    let rightProto = rightVaule.prototype; // 取右表达式的 prototype 值
    leftVaule = leftVaule.__proto__; // 取左表达式的__proto__值
    while (true) {
            if (leftVaule === null) {
            return false;
        }
        if (leftVaule === rightProto) {
            return true;
        }
        leftVaule = leftVaule.__proto__
    }
}
```

例子

```
function Foo() {
}

Object instanceof Object // true
Function instanceof Function // true
Function instanceof Object // true
Foo instanceof Foo // false  Foo.__proto__ => Function.prototype != Foo.prototype
Foo instanceof Object // true
Foo instanceof Function // true
```

要想全部理解 `instanceof` 的原理，除了我们刚刚提到的实现原理，我们还需要知道 JavaScript 的原型继承原理。




```
Foo.__proto__ === Function.prototype
// true
Foo.prototype.__proto__ === Object.prototype
// true
```

## 总结

一个函数的`__proto__`是 Function.prototype, Function.prototype.`__proto__`是Object.prototype.

一般函数 instanceof 流程,`__proto__`都会找到Function.prototype, 找到Object.prototype. 最终 => null

Object, String, Number, Symbol, Function等都是函数.

```
// 一个函数的__proto__ 是 Function.prototype, Function.prototype.__proto__是Object.prototype.
leftValue = Object.__proto__ = Function.prototype;
rightValue = Object.prototype;
// 第一次判断
leftValue != rightValue
leftValue = Function.prototype.__proto__ = Object.prototype
// 第二次判断
leftValue === rightValue
// 返回 true
```

使用typeof 判断基本数据类型可以. 但对null类型使用(原理: null机器码也都为3位0)返回“object”.

使用instanceof判断对象具体类型. 判断实例原型链是否包含它在其中.

想要准确判断对象实例的类型时, 可以使用Object.prototype.toString.call(xx).

# call和apply的模拟实现

call

```
// call es3的方法 eval()
Function.prototype.call2 = function(t) {
    // 传null
    var context = t || window;
        context.fn = this;
    // var res = context.fn(...Array.from(arguments).slice(1)) // es6
    var args = [];
    for(var i = 1, len = arguments.length; i < len; i++) {
        args.push('arguments[' + i + ']');
    }
    console.log(args);
    var res = eval('context.fn(' + args +')'); // 目前浏览器不支持
    delete context.fn;
    return res;
}

Function.prototype.call2 = function(t) {
    var context = t || window; // 处理传null的情况
        context.fn = this;
    var res = context.fn(...Array.from(arguments).slice(1)) // es6
    delete context.fn;
    return res;
}

// 测试一下
var value = 2;

var obj = {
    value: 1
}

function bar(name, age) {
    console.log(this.value);
    return {
        value: this.value,
        name: name,
        age: age
    }
}

bar.call2(null); // 2
console.log(bar.call2(obj, 'kevin', 18));
```

apply

```
Function.prototype.apply2 = function(t, args = []) {
    var context = t || window; // 处理传null的情况
        context.fn = this;
    var res = context.fn(...args) // es6
    delete context.fn;
    return res;
}
bar.apply2(null); // 2
console.log(bar.apply2(obj, ['kevin', 18]));
```

## 总结

call和apply的模拟实现:

```
1. 将函数设为对象的属性
2. 将而外参数传入对象函数并执行收集返回值
3. 删除对象函数属性
```

call 和 apply 的共同点: 能够**改变函数执行时的上下文**，将一个对象的方法交给另一个对象来执行，并且是立即执行的。

# bind的模拟

`arguments`对象不是一个 `Array` 。它类似于`Array`，但除了length属性和索引元素之外没有任何`Array`属性。例如，它没有 pop 方法。但是它可以被转换为一个真正的`Array`：

```
var args = Array.prototype.slice.call(arguments);
var args = [].slice.call(arguments);

// ES2015
const args = Array.from(arguments);
const args = [...arguments];
```

## 传参的模拟实现

可以二次传参.

```
var foo = {
    value: 1
};

function bar() {
    console.log(this.value);
}

// 返回了一个函数
var bindFoo = bar.bind(foo);

bindFoo(); // 1
```

```
// 传参的模拟实现 80%
Function.prototype.bind2 = function(context) {
    context = context || window; // 处理传null的情况
        var self = this;
    var args = Array.from(arguments).slice(1) // 可以二次传参
    return function() {
        // 这个时候的arguments是指bind返回的函数传入的参数
        return self.apply(context, args.concat([...arguments]))
    };
}
```

## 构造函数效果的模拟实现

bind 还有一个特点，就是

> 一个绑定函数也能使用new操作符创建对象：这种行为就像把原函数当成构造器。提供的 this 值被忽略，同时调用时的参数被提供给模拟函数。

且

```
// 如果是构造函数，将this指向实例
Function.prototype.bind2 = function(context) {
        var self = this;
    var args = Array.from(arguments).slice(1) // 可以二次传参
    var fBound = function () {
        return self.apply(this instanceof fBound ? this : context, args.concat([...arguments]))
    };
    // 返回函数继承
    fBound.prototype = Object.create(this.prototype)
    return fBound
}
```

```
var value = 2;

var foo = {
    value: 1
};

function bar(name, age) {
    this.habit = 'shopping';
    console.log(this.value);
    console.log(name);
    console.log(age);
}

bar.prototype.friend = 'kevin';

var bindFoo = bar.bind(foo, 'daisy');

var obj = new bindFoo('18');
// undefined
// daisy
// 18
console.log(obj.habit);
console.log(obj.friend);
// shopping
// kevin
```

上面与原生bind有区别，就是new 的时候，实例的原型是指向 fBound.prototype

```
Function.prototype.bind2 = function(context) {
        var self = this;
    var args = Array.from(arguments).slice(1)
    return function F () {
        // 如果是new 直接new原函数
        return this instanceof F ? new self(...args.concat([...arguments])) : self.apply(context,
    };
}
```

## 总结

bind特点: 改变this指向,返回函数, 可以二次传参. 如果绑定函数使用new操作符创建对象.则把原函数当做构造函数, 忽略提供的this值.

# 函数柯里化

引用

在数学和计算机科学中， 柯里化是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

```
function add(a, b) {
    return a + b;
}

// 执行 add 函数，一次传入两个参数即可
// add(1, 2) // 3

// 假设有一个 curry 函数可以做到柯里化，可以传一个，多个参数
var addCurry = curry(add);
console.log(addCurry(1)(2)) // 3
```

## 用途

举个例子：

```
// 示意而已
function ajax(type, url, data) {
    var xhr = new XMLHttpRequest();
    xhr.open(type, url, true);
    xhr.send(data);
}

// 虽然 ajax 这个函数非常通用，但在重复调用的时候参数冗余
ajax('POST', 'www.test.com', "name=kevin")
ajax('POST', 'www.test2.com', "name=kevin")
ajax('POST', 'www.test3.com', "name=kevin")

// 利用 curry
var ajaxCurry = curry(ajax);

// 以 POST 类型请求数据
var post = ajaxCurry('POST');
post('www.test.com', "name=kevin");

// 以 POST 类型请求来自于 www.test.com 的数据
var postFromTest = post('www.test.com');
postFromTest("name=kevin");
```

curry 的这种用途可以理解为：参数复用。本质上是降低通用性，提高适用性

如果我们仅仅是把参数一个一个传进去，意义可能不大，但是如果我们是把柯里化后的函数传给其他函数比如 map 呢？

举个例子：

比如我们有这样一段数据：

```
var person = [{name: 'kevin'}, {name: 'daisy'}]
```

如果我们要获取所有的 name 值，我们可以这样做：

```
var name = person.map(function (item) {
    return item.name;
})
```

不过如果我们有 curry 函数：

```
var prop = curry(function (key, obj) {
    return obj[key]
});

var name = person.map(prop('name'))
```

我们为了获取 name 属性还要再编写一个 prop 函数，是不是又麻烦了些？

但是要注意，prop 函数编写一次后，以后可以多次使用，实际上代码从原本的三行精简成了一行，而且你看代码是不是更加易懂了？

`person.map(prop('name'))` 就好像直白的告诉你：person 对象遍历(map)获取(prop) name 属性。

## 版本

```
// 保存之前的参数
function sub_curry(fn, ...args) {
    return function() {
        return fn.apply(this, args.concat([].slice.call(arguments)));
    };
}

function curry(fn, len) {
    len = len || fn.length;
    return function() {
        if(arguments.length < len) {
                return curry(sub_curry.apply(this, [fn, ...arguments]), len - arguments.length)
        } else {
            return fn.apply(this, arguments)
                }
    }
}
```

## 定长curry

这样理解柯里化 ：用闭包把参数保存起来，当参数的数量足够执行函数了，就开始执行函数. 柯里化一次传一个参数

```
var curry = fn =>
    judge = (...args) =>
        args.length === fn.length
            ? fn(...args)
            : (arg) => judge(...args, arg)

// 如果传多个参数
var curry = fn =>
    judge = (...args) =>
        args.length === fn.length
            ? fn(...args)
            : (...arg) => judge(...args, ...arg)
```

上述写法 含有 `this` 的绑定问题以及 `judge` 变量的污染。

```
// example
var judge = 'judge'
var name = 'window'
var obj = {
  name: 'obj',
  say: function(a, b, c, d){ return `${this.name}'s params: ${[a, b, c, d].join(',')}` }
}
console.log(typeof judge) // => "string"
var curriedSay = curry.call(obj, obj.say) // 无法修改`this`的指向
console.log(curriedSay('a', 'b')('c')('d')) // => "window's params: a,b,c,d"
console.log(typeof judge) // => "function"
```

## 不定长 curry 的考察

如实现这样一个 add: (头条一面, 腾讯一面考了同一题)

```
add(1); // 1
add(1)(2); // 3
add(1)(2)(3); // 5
```

这样的题目其实是有问题的, 因为确实没有办法预先获取这个调用链的长度. 面试官应该传达一点, 其实给的注释并不代表 return 值...

因此, 相应的实现有:

```
const add = sum => {
    const fn = n => add(n + sum);

    fn.valueOf = () => sum;

    return fn;
}

/** Test **/
add(1); // Function
+add(1); // 1
+add(1)(2); // 3
+add(1)(2)(3); // 5
```

## 总结

函数柯里化: 将多个参数的函数转化为一系列使用一个参数的函数.

理解: 用闭包将参数保存起来, 当参数长度等于原来函数参数长度的时候开始执行. 一次只传一个参数

实现:

```
var curry = fn => judge = (...args) => args.length === fn.length ? fn(...args) : (arg) => judge(..
```

# V8引擎的垃圾回收机制

引用

## 为何需要垃圾回收

在V8引擎逐行执行JavaScript代码的过程中，当遇到函数的情况时，会为其创建一个**函数执行上下文(Context)环境并添加到调用堆栈的栈顶**，函数的作用域(handleScope)中包含了该函数中声明的所有变量，**当该函数执行完毕后，对应的执行上下文从栈顶弹出**，函数的作用域会随之销毁，**其包含的所有变量也会统一释放并被自动回收**。试想如果在这个作用域被销毁的过程中，其中的变量不被回收，即持久占用内存，那么必然会导致内存暴增，从而引发内存泄漏导致程序的性能直线下降甚至崩溃，因此内存在使用完毕之后理当归还给操作系统以保证内存的重复利用。

## 回收策略

主要是基于**`分代式垃圾回收机制`，**其根据**对象的存活时间**将内存的垃圾回收进行不同的分代，然后对不同的分代采用不同的垃圾回收算法

### 内存结构

除了`新生代`和`老生代`外，还包含其他几个部分

- **新生代**: 主要用于**存放存活时间较短的对象**, 大多数的对象开始都会被分配在这里，这个区域相对较小但是垃圾回收特别频繁，该区域被分为两半，一半用来**分配内存**(From空间)，另一半用于在垃圾回收时**将需要保留的对象复制**过来(To空间)。
- **老生代**: **新生代中的对象在存活一段时间后就会被转移到老生代内存区**，相对于新生代该内存区域的垃圾回收频率较低。老生代又分为**`老生代指针区`和`老生代数据区`，**前者包含大多数可能**存在指向其他对象的指针的对象**，后者只保存**原始数据对象**，这些对象没有指向其他对象的指针。
- `大对象区(large_object_space)`：存放体积超越其他区域大小的对象，每个对象都会有自己的内存，垃圾回收不会移动大对象区。
- `代码区(code_space)`：代码对象，会被分配在这里，唯一拥有执行权限的内存区域。
- `map区(map_space)`：存放Cell和Map，每个区域都是存放相同大小的元素，结构简单(这里没有做具体深入的了解，有清楚的小伙伴儿还麻烦解释下)。


### 新生代

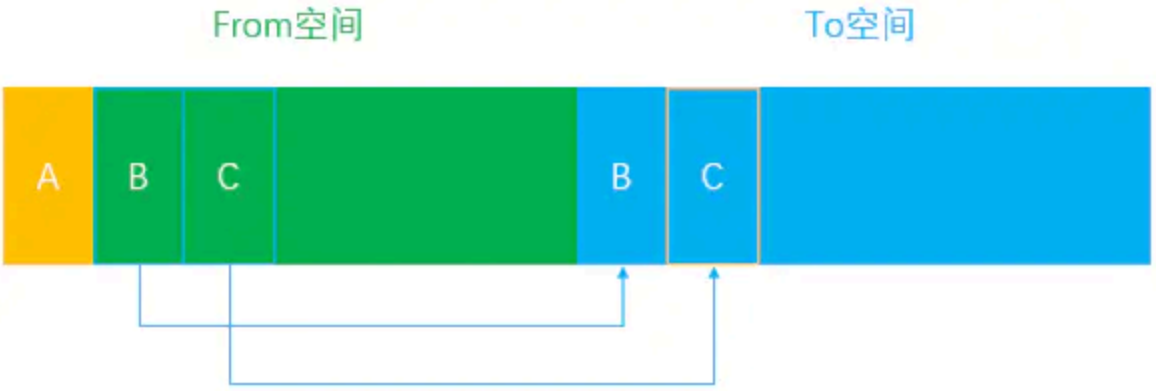
`Scavenge`算法: 是一种典型的牺牲空间换取时间的算法.

> 在 Scavenge 算法的具体实现中，主要采用了 Cheney 算法，它将新生代内存一分为二，每一个部分的空间称为 semispace，也就是我们在上图中看见的new_space中划分的两个区域，其中处于激活状态的区域我们称为 From 空间，未激活(inactive new space)的区域我们称为 To 空间。这两个空间中，始终只有一个处于使用状态，另一个处于闲置状态。我们的程序中声明的对象首先会被分配到 From 空间，当进行垃圾回收时，如果 From 空间中尚有存活对象，则会被复制到 To 空间进行保存，非存活的对象会被自动回收。当复制完成后，From 空间和 To 空间完成一次角色互换，To 空间会变为新的 From 空间，原来的 From 空间则变为 To 空间。

- 假设我们在 `From` 空间中分配了三个对象A、B、C


- 当程序主线程任务第一次执行完毕后进入垃圾回收时，发现对象A已经没有其他引用，则表示可以对其进行回收


- 对象B和对象C此时依旧处于活跃状态，因此会被复制到 `To` 空间中进行保存



- 接下来将 `From` 空间中的所有非存活对象全部清除


- 此时 `From` 空间中的内存已经清空，开始和 `To` 空间完成一次角色互换


主线程在执行第二个任务时, 再次轮回.

**对象晋升**

对象晋升的条件主要有以下两个：

- 对象是否经历过一次 `Scavenge` 算法 (根据该对象的内存地址是否发生变动来判断)
- `To` 空间的内存占比是否已经超过 `25%`

**老生代**

`Mark-Sweep(标记清除)` 和 `Mark-Compact(标记整理)` 来进行管理。

- Mark-Sweep(标记): **通过判断某个对象是否可以被访问到，从而知道该对象是否应该被回收**, 在经历过一次标记清除后，内存空间可能会出现不连续的状态，因为我们所清理的对象的内存地址可能不是连续的.
- Mark-Compact(整理): 在整理的过程中，会将活动的对象往堆内存的一端进行移动，移动完成后再清理掉边界外的全部内存

- 假设在老生代中有A、B、C、D四个对象

老生代


- 在垃圾回收的 标记 阶段，将对象A和对象C标记为活动的

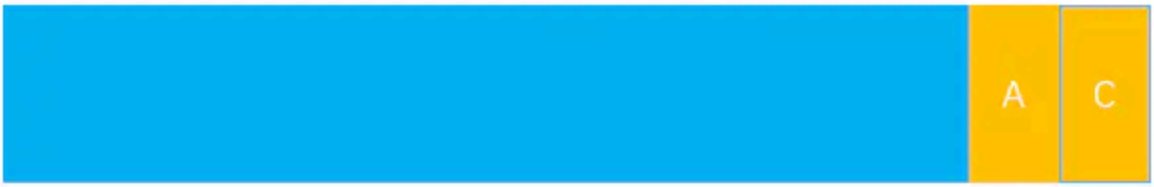
老生代-标记阶段


- 在垃圾回收的 整理 阶段，将活动的对象往堆内存的一端移动

老生代-整理阶段


- 在垃圾回收的 清除 阶段，将活动对象左侧的内存全部回收

老生代-清除阶段



> 为了减少垃圾回收带来的停顿时间，V8引擎又引入了** Incremental Marking(增量标记) **的概念，即将原本需要一次性遍历堆内存的操作改为增量标记的方式，先标记堆内存中的一部分对象，然后暂停，将执行权重新交给JS主线程，待主线程任务执行完毕后再从原来暂停标记的地方继续标记，直到标记完整个堆内存。这个理念其实有点像 React 框架中的 Fiber 架构，只有在浏览器的空闲时间才会去遍历 Fiber Tree 执行对应的任务，否则延迟执行，尽可能少地影响主线程的任务，避免应用卡顿，提升应用性能。

> 得益于增量标记的好处，V8引擎后续继续引入了 延迟清理(lazy sweeping) 和 增量式整理(incremental compaction)，让清理和整理的过程也变成增量式的。同时为了充分利用多核CPU的性能，也将引入 并行标记 和 并行清理，进一步地减少垃圾回收对主线程的影响，为应用提升更多的性能。

### 如何避免内存泄漏

- **尽可能少地创建全局变量**
- **手动清除定时器**
- **少用闭包**
- **清除DOM引用**
- **弱引用**

## 总结

v8引擎垃圾回收策略是基于**分代式垃圾回收机制**, 根据对象存活时间将内存垃圾进行不同分代,对不同分代使用不同的垃圾回收算法.

其中内存结构中 , 垃圾回收过程主要出现在**新生代**和**老生代**. 其他部分为大对象区, 代码区, map区

- 新生代: 区域分为两半.一半激活区域( 刚创建的对象开始都会在From空间)分配内存. 一半未激活区域.(To空间) 用来垃圾回收时复制需要保留的对象, 使用**scavenge算法**. 当复制完成后. 两个空间会进行一次角色互换. `To`空间会变为新的`From`空间, 原来的`From`空间则变为`To`空间.
- 老生代: 新生代中的对象存活一段时间会转移到老生代内存区. 老生代又分为指针区和数据区. 指针区包含大多可能存在指向其他对象的指针对象.数据区只保存原始数

据对象. 采用`Mark-Sweep`和`Mark-Compact`算法. 标记采用增量标记概念,这个理念其实有点像`React`框架中的`Fiber`架构

对象晋升: 新生代的对象转移到老生代的条件

- 经历过一次scavenge算法
- 在To空间时内存占比超25%

如何避免内存泄漏

- 少创建全局对象
- 手动清除定时器
- 少用闭包
- 清除DOM引用(DOM引用作为对象属性)
- 弱引用

## 浮点数精度

0.1 + 0.2 是否等于 0.3 作为一道经典的面试题，已经广外熟知，说起原因，大家能回答出这是浮点数精度问题导致，也能辩证的看待这并非是 ECMAScript 这门语言的问题，今天就是具体看一下背后的原因。

0.1根本就不是精确的0.1，而是一个有舍入误差的0.1

### 总结

- ECMAScript 采用的就是IEEE754 标准中双精确度(64位).
- 为什么0.1 + 0.2 不等于0.3。因为不能精确表示浮点数，计算时使用的是带有舍入误差的数
- 并不是所有的浮点数在计算机内部都存在舍入误差，比如0.5就没有舍入误差
- 具有舍入误差的运算结可能会符合我们的期望，原因可能是“负负得正”
- 怎么办？1个办法是使用整型代替浮点数计算；2. 是不要直接比较两个浮点数，而应该使用bignumber.js这样的浮点数运算库

## new的模拟实现

> new 运算符创建一个用户定义的对象类型的实例或具有构造函数的内置对象类型之一

new 实现了哪些功能?

- 能访问构造函数的属性
- 能访问函数原型中的属性.
- 如果构造函数带返回值. 返回类型如果是对象. 则返回这个对象. 否则还是实例对象.

### 初步实现

```
function objectFactory(Constructor, ...args) {
    var obj = {};
    obj.__proto__ = Constructor.prototype
    Constructor.apply(obj, args)
    return obj
}
```

### 如果构造函数带返回值

返回值若是对象, 则返回这个对象. 否则返回实例对象.

```
function objectFactory(Constructor, ...args) {
    var obj = {};
    obj.__proto__ = Constructor.prototype
    var res = Constructor.apply(obj, args)
    return Object.prototype.toString.call(res) === '[object Object]' ? res : obj;
}
```

### 总结

new关键字实现了

访问构造函数属性.

访问原型链的属性方法.

如果构造函数带返回值. 且类型是对象.则返回这个对象. 否则返回实例对象.

## 事件循环机制

### 浏览器环境下js引擎的事件循环机制

#### 执行栈与事件队列

当javascript代码执行的时候会将不同的变量存于内存中的不同位置：堆（heap）和栈（stack）

中来加以区分。其中，堆里存放着一些对象。而栈中则存放着一些基础类型变量以及对象的指针。但是我们这里说的执行栈和上面这个栈的意义却有些不同。

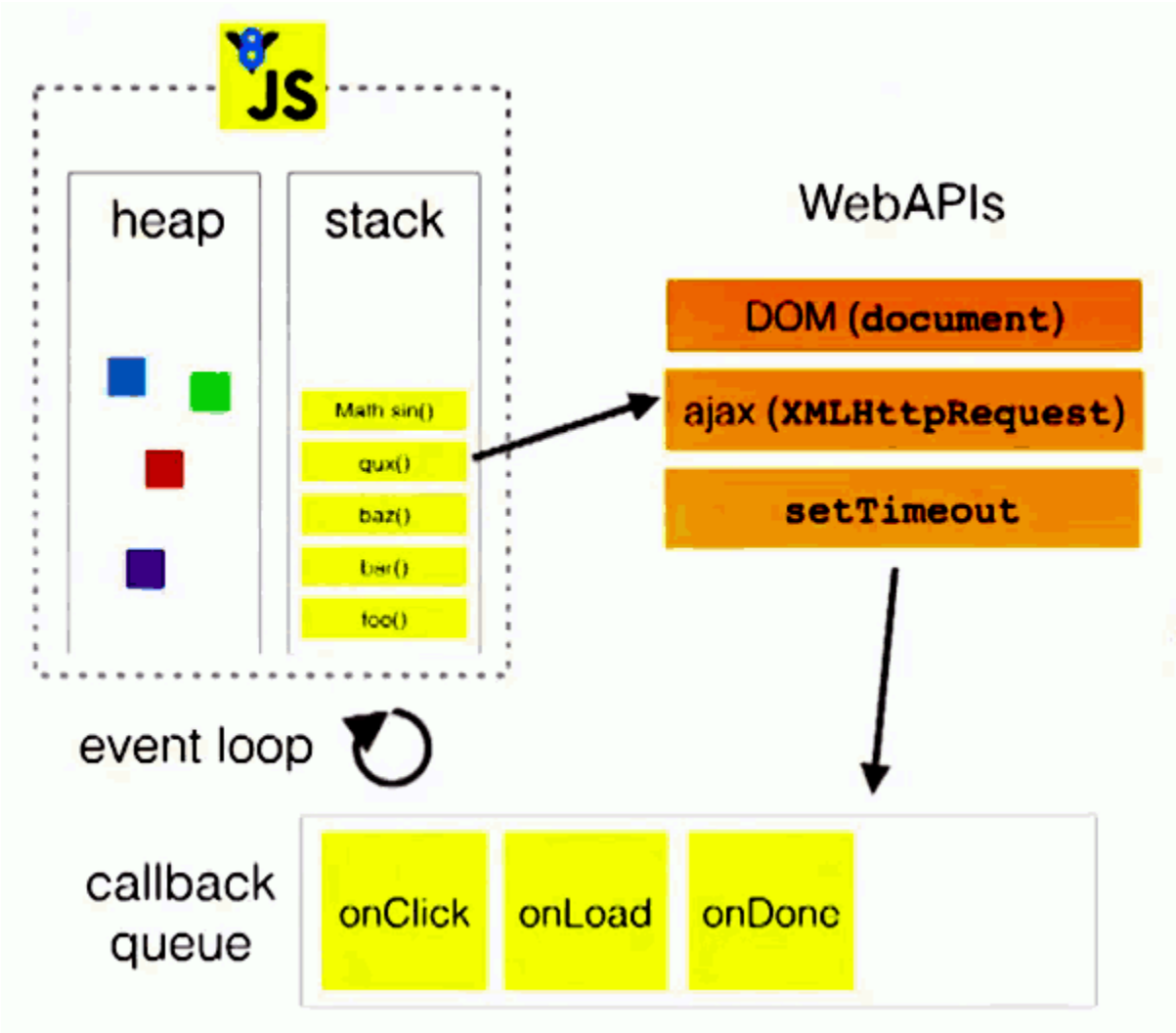
当我们调用一个方法的时候，js会生成一个与这个方法对应的执行环境（context），又叫执行上下文。这个执行环境中存在着这个方法的私有作用域，上层作用域的指向，方法的参数，这个作用域中定义的变量以及这个作用域的this对象。 而当一系列方法被依次调用的时候，因为js是单线程的，同一时间只能执行一个方法，于是这些方法被排队在一个单独的地方。这个地方被称为执行栈。

**同步代码**

- **同步代码的执行**: 当一段代码执行时, js引擎会解析这段代码. 先初始化全局执行上下文, 压入执行栈.并把同步代码按执行顺序压入执行栈.然后从头开始执行. 如果当前执行的是个方法. 那么会创建这个执行上下文，并且压入执行栈，然后执行这个方法其中的代码, 当执行完毕了, 就会将方法的执行上下文从栈中弹出. 再回到上一个方法的执行上下文. 继续执行. 直到执行栈中的代码全部执行完毕.

**异步代码**

- **异步代码**（如发送ajax请求数据）：当遇到异步事件时，**执行栈**不会等待事件结果，而是会把这事件挂起,执行其他任务. 当这个事件有结果了.会把这个事件加入**事件队列**里，不会立即执行其回调. 等到当前执行栈闲置状态时. 执行栈才会去读取事件队列里的事件. 把事件对应的回调放入执行栈中. 执行其中代码. 如此反复. 这样就形成了一个无限的循环。这就是这个过程被称为“事件循环（Event Loop）”的原因。




图中的stack表示我们所说的执行栈，web apis则是代表一些异步事件，而callback queue即事件队列。

**微任务和宏任务**

**其中因为异步任务之间并不相同**

以下事件属于宏任务：

- `setInterval()`
- `setTimeout()`

以下事件属于微任务

- `new Promise()`
- `new MutaionObserver()`

执行栈为空时, 会先查看微任务,是否有事件存在. 再去查看宏任务.

### 总结

当前执行栈遇到异步事件时.不会等其结果. 会把它挂起, 执行其他任务, 当它有结果了, 会将事件放入事件队列中. 等执行栈闲置状态时, 才会从事件队列中查看是否有任务. 若有. 将事件对应的回调放入执行栈执行.如此反复. 形成了一个无限的循环

其中因为异步任务之间并不相同, 分微任务和宏任务.

**当前执行栈执行完毕时会立刻先处理所有微任务队列中的事件，然后再去宏任务队列中取出一个事件。同一次事件循环中，微任务永远在宏任务之前执行。**

```
setTimeout(function () {
    console.log(1);
});

new Promise(function(resolve,reject){
    console.log(2)
    resolve(3)
}).then(function(val){
    console.log(val);
})
console.log(0)
// 2 0 3 1
```

## node环境下的事件循环机制

在node中，事件循环表现出的状态与浏览器中大致相同。不同的是node中有一套自己的模型。node中事件循环的实现是依靠的libuv引擎。我们知道node选择chrome v8引擎作为js解释器，v8引擎将js代码分析后去调用对应的node api，而这些api最后则由libuv引擎驱动，执行对应的任务，并把不同的事件放在不同的队列中等待主线程执行。 因此实际上node中的事件循环存在于libuv引擎中。

# promise原理

## 回顾注意点

如果调用`resolve`函数和`reject`函数时带有参数，那么它们的参数会被传递给回调函数。`reject`函数的参数通常是`Error`对象的实例，表示抛出的错误；`resolve`函数的参数除了正常的值以外，**还可能是另一个 Promise 实例**，`

```
const p1 = new Promise(function (resolve, reject) {
  // ...
});

const p2 = new Promise(function (resolve, reject) {
  // ...
  resolve(p1);
})
```

上面代码中，`p1`和`p2`都是 Promise 的实例，但是`p2`的`resolve`方法将`p1`作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时`p1`的状态就会传递给`p2`，也就是说，**`p1`的状态决定了`p2`的状态**。如果`p1`的状态是`pending`，那么`p2`的回调函数就会等待`p1`的状态改变；如果`p1`的状态已经是`resolved`或者`rejected`，那么`p2`的回调函数将会立刻执行。

---

`then`方法是定义在原型对象`Promise.prototype`上的。它的作用是为 Promise 实例添加状态改变时的回调函数。前面说过，`then`方法的第一个参数是`resolved`状态的回调函数，第二个参数是`rejected`状态的回调函数，它们都是可选的。

**`then`方法返回的是一个新的`Promise`实例**（注意，不是原来那个`Promise`实例）。因此可以采用**链式写法**，即`then`方法后面再调用另一个`then`方法

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

上面的代码使用`then`方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结

果作为参数，传入第二个回调函数。

### Promise.prototype.finally()

`finally()`方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

### Promise.all()

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

```
const p = Promise.all([p1, p2, p3]);
// p: [Result1, Result2, Result3]
```

上面代码中，`Promise.all()`方法接受一个数组作为参数，`p1`、`p2`、`p3`都是 Promise 实例，如果不是，就会先调用下面讲到的`Promise.resolve`方法，将参数转为 Promise 实例，再进一步处理。另外，`Promise.all()`方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例

`p`的状态由`p1`、`p2`、`p3`决定，分成两种情况。

（1）只有`p1`、`p2`、`p3`的状态都变成`fulfilled`，`p`的状态才会变成`fulfilled`，此时`p1`、`p2`、`p3`的返回值组成一个数组，传递给`p`的回调函数。

（2）只要`p1`、`p2`、`p3`之中有一个被`rejected`，`p`的状态就变成`rejected`，此时第一个被`reject`的实例的返回值，会传递给`p`的回调函数。

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

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

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

如果**参数有自己的catch方法**. 会导致`Promise.all()`方法参数里面的实例 resolved，因此会调用`then`方法指定的回调函数，而不会调用`catch`方法指定的回调函数。

### Promise.race()

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

适用于没有返回值的函数.

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

上面代码中，只要`p1`、`p2`、`p3`之中有一个实例率先改变状态，`p`的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给`p`的回调函数。

### Promise.allSettled()

`romise.allSettled()`方法接受一组 Promise 实例作为参数，包装成一个新的 Promise 实例。只有等到所有这些参数实例都返回结果，**不管是`fulfilled`还是`rejected`，包装实例才会结束。**

`fulfilled`时，对象有`value`属性，`rejected`时有`reason`属性，对应两种状态的返回值

```
const resolved = Promise.resolve(42);
const rejected = Promise.reject(-1);

const allSettledPromise = Promise.allSettled([resolved, rejected]);

allSettledPromise.then(function (results) {
  console.log(results);
});
// [
//    { status: 'fulfilled', value: 42 },
//    { status: 'rejected', reason: -1 }
// ]
```

有时候，我们**不关心异步操作的结果，只关心这些操作有没有结束**。这时，`Promise.allSettled()`方法就很有用。如果没有这个方法，想要确保所有操作都结束，就很麻烦。`Promise.all()`方法无法做到这一点。

### Promise.any()

该方法接受一组 Promise 实例作为参数，包装成一个新的 Promise 实例返回。只要参数实例有一个变成`fulfilled`状态，包装实例就会变成`fulfilled`状态；如果所有参数实例都变成`rejected`状态，包装实例就会变成`rejected`状态。

**`Promise.any()`跟`Promise.race()`方法很像，只有一点不同，就是不会因为某个 Promise 变成`rejected`状态而结束。**

### Promise.resolve()

有时需要将现有对象转为 Promise 对象，`Promise.resolve()`方法就起到这个作用.

```
setTimeout(function () {
  console.log('three');
}, 0);

Promise.resolve().then(function () {
  console.log('two');
});

console.log('one');

// one
// two
// three
```

**Promise.reject()**

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

# generator

## Iterator（迭代器）的概念

Iterator 的作用有三个：一是为各种数据结构，提供一个统一的、简便的访问接口；二是使得数据结构的成员能够按某种次序排列；三是 ES6 创造了一种新的遍历命令 `for...of` 循环，Iterator 接口主要供 `for...of` 消费。

迭代过程

（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。

（2）第一次调用指针对象的 `next` 方法，可以将指针指向数据结构的第一个成员。

（3）第二次调用指针对象的 `next` 方法，指针就指向数据结构的第二个成员。

（4）不断调用指针对象的 `next` 方法，直到它指向数据结构的结束位置。

**每一次调用 `next` 方法，都会返回数据结构的当前成员的信息**。具体来说，就是返回一个**包含 `value` 和 `done` 两个属性的对象**。其中，**`value` 属性是当前成员的值，`done` 属性是一个布尔值，表示遍历是否结束。**

下面是一个模拟 `next` 方法返回值的例子。

```
var it = makeIterator(['a', 'b']);

it.next() // { value: "a", done: false }
it.next() // { value: "b", done: false }
it.next() // { value: undefined, done: true }

function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {value: array[nextIndex++], done: false} :
        {value: undefined, done: true};
    }
  };
}
```

原生具备 Iterator 接口的数据结构如下。

- Array
- Map
- Set
- String
- TypedArray
- 函数的 arguments 对象
- NodeList 对象

下面的例子是数组的`Symbol.iterator`属性。

```
let arr = ['a', 'b', 'c'];
let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }
iter.next() // { value: 'b', done: false }
iter.next() // { value: 'c', done: false }
iter.next() // { value: undefined, done: true }
```

## Generator 函数

Generator 函数有多种理解角度。语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

形式上，Generator 函数是一个普通函数，但是有两个特征。一是，`function`关键字与函数名之间有一个星号；二是，函数体内部使用`yield`表达式，定义不同的内部状态（`yield`在英语里的意思就是“产出”），返回的**遍历器对象**(iterator对象)

```
function* foo() {
        yield 'hi';
    yield 'world'
    return "ending";
}
var h = foo();
h.next()
// {value: "hi", done: false}
h.next()
// {value: "world", done: false}
h.next()
// {value: "ending", done: true}
h.next()
// {value: undefined, done: true}


function* f2() {
        yield 'hi';
    yield 'world'
}
var h2 = f2();
h2.next()
// {value: "hi", done: false}
h2.next()
// {value: "world", done: false}
h2.next()
// {value: undefined, done: true}

function* f3() {
    return 'no yield'
}
var h3 = f3();
h3.next();
// {value: "no yield", done: true}
h3.next();
// {value: undefined, done: true}

function* f4() {
    yield 'no return'
}
var h4 = f4();
h4.next();
// {value: "no return", done: false}
h4.next();
// {value: undefined, done: true}
```

```
function* f5() {
    console.log('no yield')
}
var h5 = f5();
h5.next();
// {value: undefined, done: true}
```

### next 方法的参数

如果`next`方法没有参数，`yield`表达式的返回值是`undefined`。当`next`方法带一个参数，该参数就会被当作上一个`yield`表达式的返回值。

```
function* foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z);
}

var a = foo(5);
a.next() // Object{value:6, done:false}
a.next() // Object{value:NaN, done:false}
a.next() // Object{value:NaN, done:true}

var b = foo(5);
b.next() // { value:6, done:false }
b.next(12) // { value:8, done:false }
b.next(13) // { value:42, done:true }
```

注意，由于`next`方法的参数表示上一个`yield`表达式的返回值，所以在第一次使用`next`方法时，传递参数是无效的。V8 引擎直接忽略第一次使用`next`方法时的参数.

如果想要第一次调用`next`方法时，就能够输入值，可以在 Generator 函数外面再包一层。

```
function wrapper(generatorFunction) {
  return function (...args) {
    let generatorObject = generatorFunction(...args);
    generatorObject.next();
    return generatorObject;
  };
}

const wrapped = wrapper(function* () {
  console.log(`First input: ${yield}`);
  return 'DONE';
});

wrapped().next('hello!')
// First input: hello!
```

**总结**

调用 Generator 函数，返回一个遍历器对象(iterator对象). 第一次执行next方法, 如果遇到`yield`关键字 . 会停止. 返回一个对象. value`属性的值就是当前` yield `表达式的值`, done值为false. 表示遍历还没有结束.

第二次调用，Generator 函数从上次`yield`表达式停下的地方. 一直执行到下一个`yield`表达式, 如果没有, 则`done`值为`true`, `value值`为如果有return. 为`return值`. 没有则`undefined`.

如果`next`方法没有参数, `yield`表达式的返回值是`undefined` . 当`next`方法带一个参数, 该参数就会被当作上一个`yield`表达式的返回值。

ES6 提供了`yield*`表达式，用来在一个 Generator 函数里面执行另一个 Generator 函数。

# CSS

## 盒模型

### 认识

1. 基本概念：标准模型+IE模型。 包括margin,border,padding,content
2. 标准模型和IE模型的区别
3. css如何设置获取这两种模型的宽和高
4. js如何设置获取盒模型对应的宽和高

5. 根据盒模型解释边距重叠
6. BFC（边距重叠解决方案，还有IFC）解决边距重叠

## 基本概念

**什么是盒模型：**盒模型又称框模型（Box Model）,包含了元素内容（content）、内边距（padding）、边框（border）、外边距（margin）几个要素

- 标准模型: width = **content** ,高度计算相同.
- IE模型: width = **content+padding+border**

## css如何设置获取这两种模型的宽和高

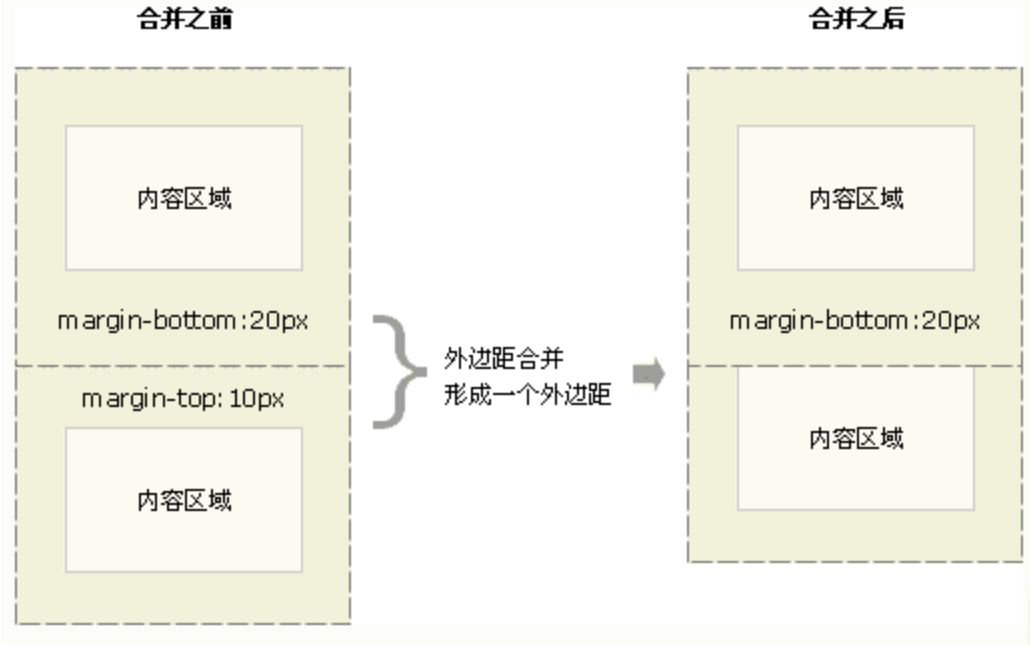
```
.standrad {
    box-sizing: content-box;
}
.ie {
    box-sizing: border-box;
}
```

## js如何设置获取盒模型对应的宽和高

- `dom.style.width/height` 只能取到行内样式的宽和高，style标签中和link外链的样式取不到。
- 取到的是最终渲染后的宽和高
- `dom.currentStyle.width/height` 只有IE支持此属性
- `getCurrentStyle(dom).width` 多浏览器支持, IE9以上支持
- `dom.getBoundingClientRect().width` 多浏览器支持, IE9以上支持. 还可以获得相对视窗的距离

## 外边距重叠

当两个垂直外边距相遇时，他们将形成一个外边距，合并后的外边距高度等于两个发生合并的外边距的高度中的较大者。**注意**：只有普通文档流中块框的垂直外边距才会发生外边距合并，行内框、浮动框或绝对定位之间的外边距不会合并。




## BFC

BFC: 块级格式化上下文.

BFC决定了元素如何对其内容进行定位，以及与其他元素的关系和相互作用。当设计到可视化布局的时候，BFC提供了一个环境，HTML元素在这个环境中按照一定的规则进行布局。**一个环境中的元素不会影响到其他环境中的布局**。

**BFC的原理（渲染规则）**

1. BFC元素垂直方向的边距会发生重叠。**属于不同BFC外边距不会发生重叠**
2. BFC的区域**不会与浮动元素的布局重叠**。
3. BFC元素是一**个独立的容器，外面的元素不会影响里面的元素。里面的元素也不会影响外面的元素**。
4. **计算BFC高度的时候，浮动元素也会参与计算**(清除浮动)

**如何创建BFC**

1. **overflow不为visible**;
2. float的值不为none；
3. position的值不为static或relative；
4. **display属性为inline-blocks,table,table-cell,table-caption,flex,inline-flex**;

使用情况

1. BFC内的外边距不与外部的外边距发生重叠。
2. BFC元素不会与浮动元素发生重叠(在没BFC时, 浮动元素重叠, 但是文本信息不会被浮动元素所覆盖)。**BFC的话. 兄弟元素浮动不重叠**
3. 子元素都浮动. **BFC的话. 计算父元素高度的时候，浮动元素也会参与计算. (清除浮动)**

## 总结

盒模型包裹 content, padding, border,margin几个元素,

两种模型: 标准盒模型 / IE盒模型 , 其中**标准盒模型width等于content**. 高度同理. **IE盒模型width等content+padding+border**.高度同理.

css设置模型的属性是`box-sizing`, 其中标准为`content-box`. IE为`border-box`;

js获取元素渲染后的宽高: **getCurrentStyle(DOM) .DOM.getBoundingClientRect()** IE9以上

BFC: 块级格式化上下文.

**具有 BFC 特性的元素可以看作是隔离了的独立容器，容器里面的元素不会在布局上影响到外面的元素，并且 BFC 具有普通容器所没有的一些特性。**

其中普通文档流. 当两个垂直外边距相遇时. 外边距会合并. 值为大的一方. 所以有了**BFC(块级格式化上下文)**的说法. 其中BFC提供了一个环境. **内部元素和外部元素互不影响.所以外边距不会合并.另外BFC的话. 有兄弟元素浮动不重叠. 计算父元素高度的时候. 浮动元素也会参与计算.(清除浮动)**

创建BFC的方法有. 1.overflow不为visible. 2.成为浮动元素. 3.不是文档流.即position不为static. relative; 4.display的值为inline-block. table. flex...

# CSS 选择器

**一个行内样式+1000，一个id选择器+100，一个属性选择器、class或者伪类+10，一个元素选择器，或者伪元素+1，通配符+0。**

- `简单选择器 (Simple selectors)`：通过元素类型、class 或 id 匹配一个或多个元素。
- `属性选择器 (Attribute selectors)`：通过 属性 / 属性值 匹配一个或多个元素。
- `伪类 (Pseudo-classes)`：匹配处于确定状态的一个或多个元素，比如被鼠标指针悬

停的元素，或当前被选中或未选中的复选框，或元素是 DOM 树中一父节点的第一个子节点。

- `伪元素（Pseudo-elements）`:匹配处于相关的确定位置的一个或多个元素，例如每个段落的第一个字，或者某个元素之前生成的内容。
- `组合器（Combinators）`：这里不仅仅是选择器本身，还有以有效的方式组合两个或更多的选择器用于非常特定的选择的方法。例如，你可以只选择 divs 的直系子节点的段落，或者直接跟在 headings 后面的段落。
- `多用选择器（Multiple selectors）`：这些也不是单独的选择器；这个思路是将以逗号分隔开的多个选择器放在一个 CSS 规则下面， 以将一组声明应用于由这些选择器选择的所有元素。

## 简单选择器

- 类型选择器 ( 元素选择器)
- 类选择器 （Class selectors）
- ID 选择器
- 通用选择器 (*)
- 组合器 eg: A,B / A B / A > B / A + B / A ~ B

## 属性选择器

分**存在和值属性选择器** 和 **子串值属性选择器**。

存在和值属性选择器

- [attr]：该选择器选择包含 attr 属性的所有元素，不论 attr 的值为何。
- [attr=val]：该选择器仅选择 attr 属性被赋值为 val 的所有元素。
- [attr**~=**val]**：**该选择器仅选择 attr 属性的值（以空格间隔出多个值）中有**包含 val 值**的所有元素，比如位于被空格分隔的多个类（class）中的一个类。

子串值属性选择器

- [attr |= val] : 选择attr属性的值以val（包括val）或val-开头的元素（-用来处理语言编码）。
- [attr ^= val] : 选择attr属性的值以val开头（包括val）的元素。
- [attr $= val] : 选择attr属性的值以val结尾（包括val）的元素。
- [attr *= val] : 选择attr属性的值中包含字符串val的元素。

### 伪类和伪元素

伪类: 一个 CSS 伪类（pseudo-class） 是一个以冒号(:)作为前缀的关键字

```
:hover {
}...
```

伪元素: 前缀是两个冒号 (:😊

### 总结

css选择器有简单选择器(id, classs, 元素), 属性选择器. 组合选择器. 伪类选择器(**:**). 伪元素选择器(**::**).

优先级: **一个行内样式+1000，一个id选择器+100，一个属性选择器、class或者伪类+10，一个元素选择器，或者伪元素+1，通配符+0。**

## position

- static
- relative
- absolute
- fixed
- sticky: 是相对位置和固定位置的混合体，它允许被定位的元素表现得像相对定位一样，直到它滚动到某个阈值点（例如，从视口顶部起10像素）为止，此后它就变得固定了

## flex布局

- flex-direction 主轴方向
    - 横轴 row / row-reverse
    - 竖轴 column / column-reverse
- flex-wrap 是否换行. 默认nowrap
- 两个属性 `flex-direction` 和 `flex-wrap` 组合为简写属性 `flex-flow`。第一个指定的值为 `flex-direction`， 第二个指定的值为 `flex-wrap`.

## flex内的元素的属性

这几个 flex 属性的作用其实就是**改变了 flex 容器中的可用空间( available space )的行为**。同时，可用空间对于 flex 元素的对齐行为也是很重要的

- `flex-grow` 按比例分配空间. 若被赋值为一个正整数， flex 元素会以 `flex-basis` 为基础，沿主轴方向增长尺寸。这会使该元素延展，并占据此方向轴上的可用空间（available space）。如果有其他元素也被允许延展，那么他们会各自占据可用空间的一部分。
- `flex-shrink` 处理flex元素收缩的问题, **如果我们的容器中没有足够排列flex元素的空间，那么可以把flex元素 `flex-shrink` 属性设置为正整数来缩小它所占空间到 `flex-basis` 以下**。与 `flex-grow`属性一样，可以赋予不同的值来控制flex元素收缩的程度 —— 给 `flex-shrink` 属性赋予更大的数值可以比赋予小数值的同级元素收缩程度更大
- `flex-basic` 定义了该元素的**空间大小（the size of that item in terms of the space）**，flex容器里除了元素所占的空间以外的富余空间就是**可用空间** available space。 该属性的默认值是 `auto` 。**此时，浏览器会检测这个元素是否具有确定的尺寸。 在上面的例子中, 所有元素都设定了宽度（width）为100px，所以 `flex-basis` 的值为100px。**

```
.box {
    display: flex;
}

.temp {
    /* 简写 */
        flex: 1 1 100px;
}
```

- `flex: initial` === `flex: 0 1 auto` 不会拉伸. 即不会超过它们 `flex-basis` 的尺寸. 可以收缩.
- `flex: auto` === `flex: 1 1 auto` 在需要的时候既可以拉伸也可以收缩
- `flex: none` === `flex: 0 0 auto` 既不能拉伸或者收缩，但是元素会按具有 `flex-basis: auto` 属性的flexbox进行布局。
- `flex: <positive-number>` === flex: x x 0; 元素可以在 `flex-basis` 为0的基础上伸缩。

`order` 属性定义项目的排列顺序。数值越小，排列越靠前，默认为0。

`align-self`属性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

## 元素间的对齐和空间分配

- `align-items`属性可以使元素在交叉轴方向对齐。
    - `stretch`默认被拉伸到最高元素的高度
    - `flex-start`
    - `flex-end`
    - `center`
- `justify-content`属性用来使元素在主轴方向上对齐
    - `stretch`
    - `flex-start`
    - `flex-end`
    - `center`
    - `space-around`：均匀排列每个元素
    - `space-between`：均匀排列每个元素，首个元素放置于起点，末尾元素放置于终点

## 总结

Flex 是 Flexible Box 的缩写，意为"弹性布局"，用来为盒状模型提供最大的灵活性。

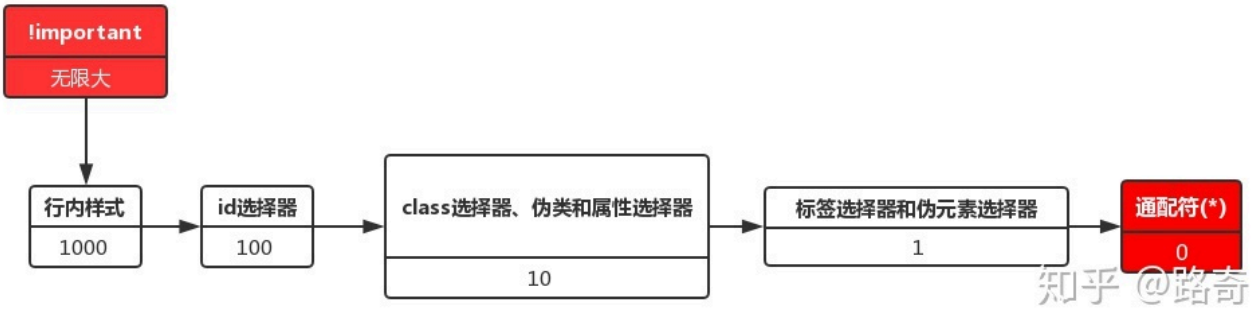
- flex-direction
- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content

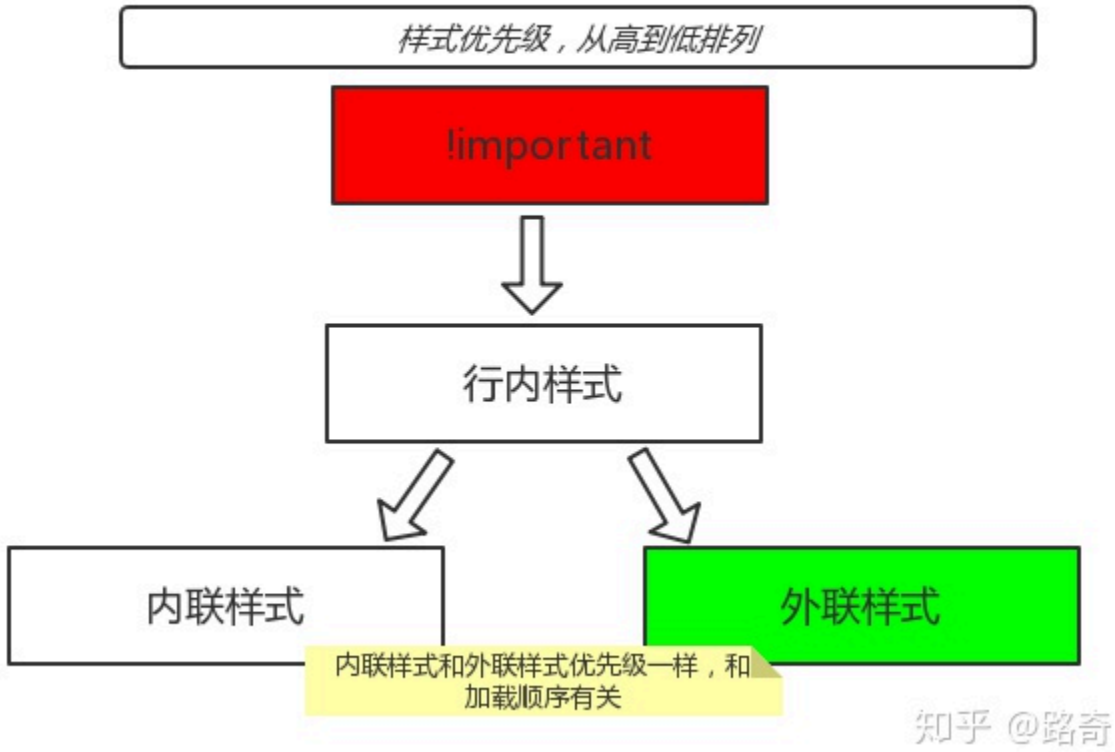
flex上的元素属性:

> - `order`
> - `flex-grow`
> - `flex-shrink`
> - `flex-basis`
> - `flex`
> - `align-self`

# css样式权重和优先级

**权重记忆口诀**：*从0开始，一个行内样式+1000，一个id选择器+100，一个属性选择器、class或者伪类+10，一个元素选择器，或者伪元素+1，通配符+0。*




## 总结

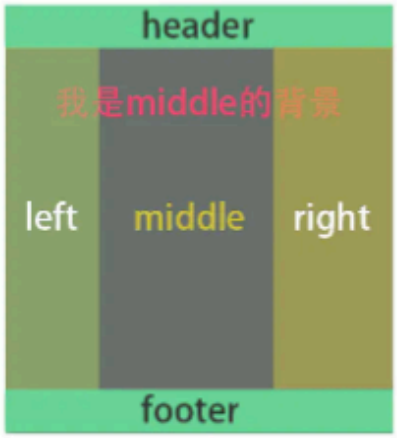
1. 常用选择器权重优先级：**!important > id > class > tag / ***
2. !important可以提升样式优先级，但不建议使用。如果!important被用于一个简写的样式属性，那么这条简写的样式属性所代表的子属性都会被应用上!important。 例

如：*background: blue !important;*

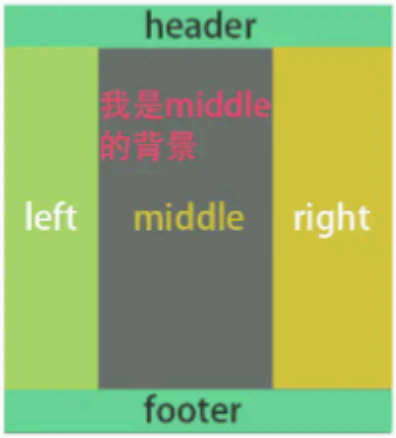
3. 如果两条样式都使用!important，则权重值高的优先级更高
4. 在css样式表中，同一个CSS样式你写了两次，后面的会覆盖前面的
5. 样式指向同一元素，权重规则生效，权重大的被应用
6. 样式指向同一元素，权重规则生效，权重相同时，顺序原则，后面定义的被应用
7. 样式不指向同一元素时，权重规则失效，就近原则生效，离目标元素最近的样式被应用

# 双飞翼布局(三栏布局)

**左右两栏固定宽度，中间部分自适应的三栏布局**




圣杯




双飞翼

- 双飞翼布局:

首先把middle、left、right都放出来, middle中增加inner

给它们三个设置上float: left, 脱离文档流；

一定记得给container设置上overflow: hidden; 可以形成BFC撑开文档

left、right设置上各自的宽度

middle设置width: 100%;

left设置 margin-left: -100%, right设置 right: -rightWidth;

container设置padding: 0, rightWidth, 0, leftWidth;

```
<div class="header">header</div>
 <div class="container">
   <div class="middle">
     <div class="inner">
       <h4>middle</h4>
       <p>
         middlemiddlemiddlemiddlemiddlemiddlemiddlemiddle
         middlemiddlemiddlemiddlemiddlemiddlemiddlemiddle
         middlemiddlemiddlemiddlemiddlemiddlemiddlemiddle
         middlemiddlemiddlemiddlemiddlemiddlemiddlemiddle
         middlemiddlemiddlemiddlemiddlemiddlemiddlemiddle
         middlemiddlemiddlemiddlemiddle
       </p>
     </div>
   </div>
   <div class="left">
     <h4>left</h4>
     <p>
       leftleftleftleftleftleftleftleftleftleftleftleft
       leftleftleftleftleftleftleftleftleftleftleftleft
       leftleftleftleftleftleftleftleftleftleftleftleft
     </p>
   </div>
   <div class="right">
     <h4>right</h4>
     <p>
       rightrightrightrightrightrightrightrightrightright
       rightrightrightrightrightrightrightrightrightright
       rightrightrightrightrightrightright
     </p>
   </div>
 </div>
 <div class="footer">footer</div>
<style>
   * {
     margin: 0;
     padding: 0;
   }

   .header,
   .footer {
     height: 100px;
     line-height: 100px;
     background-color: green;
     text-align: center;
     font-size: 30px;
```

```
    font-weight: bolder;
  }

  .footer {
    background-color: goldenrod;
  }

  .container {
    overflow: hidden;
  }

  .left,
  .middle,
  .right {
    float: left;
    min-height: 130px;
    word-break: break-all;
  }

  .left {
    margin-left: -100%;
    width: 200px;
    background-color: red;
  }

  .right {
    margin-left: -220px;
    width: 220px;
    background-color: green;
  }

  .middle {
    width: 100%;
    height: 100%;
    background-color: blue;
  }

  .inner {
    margin: 0 220px 0 200px;
    min-height: 130px;
    background: blue;
    word-break: break-all;
    overflow: hidden;
  }
</style>
```

CSS 属性 `word-break` 指定了怎样在单词内断行。

```
normal
```

使用默认的断行规则。

```
break-all
```

对于non-CJK (CJK 指中文/日文/韩文) 文本，可在任意字符间断行。

```
keep-all
```

CJK 文本不断行。 Non-CJK 文本表现同 `normal` 。

```
break-word
```

他的效果是`word-break: normal` 和 `overflow-wrap: anywhere` 的合，不论 `overflow-wrap`的值是多少。

## 总结

三栏布局的几种方法:

1. 浮动定位 (兼容性好) , 父容器创建BFC. 清除浮动
2. 绝对定位 (脱离文档流)
3. flex布局 (高等级浏览器/移动端,优先于1,2)
4. 表格布局 (兼容性好,ie8,高度一致性) table > table-cell
5. 网格布局 (css3,高等级浏览器/移动端)

## 几种解决inline-block间隙的方案

几种解决inline-block间隙的方案 (**inline-blcok块之间的不可见符号会被保留父层字体的1/3大小的空间**)

1. 改变书写结构

```
<li>item1</li><li>item2</li><li>item3</li><li>item4</li><li>item5</li>
```

2. 打包工具
3. css hack
    - 将父容器的字体大小设置为0
    - letter-spacing值为0

### img图片有间隙是因为基线的原因

```
img {
    display: block;
    vertical-align: bottom;
    line-height: 0;
}
/*指定父元素 font-size: 0; */
/*或者hack margin-top: -1px;*/
```

### 文本超出省略

```
/* 单行 */
.text_ellipsis {
    overflow: hidden;
    text-overflow: ellipsis;
    white-wrap: nowrap;
}

/* 多行 */
.text_more_ellipsis {
    display: -webkit-box;
    overflow: hidden;
    text-overflow: ellipsis;
    -webkit-line-clamp: 2;
    -webkit-box-orient: vertical;
    /* 高度=行高*行数 兼容非webkit内核*/
    line-height:30px;
    height:60px;
}
```

## CSS3

- 过渡 transition
- 动画 animation, @keyframes
- 形状转换 transform

- 选择器
  - 

| 选择器 | 例子 | 例子描述 |
| --- | --- | --- |
| element1~element2 | p~ul | 选择前面有 <p> 元素的每个 <ul> 元素。 |
| [attribute^=value] | a[src^="https"] | 选择其 src 属性值以 "https" 开头的每个 <a> 元素。 |
| [attribute$=value] | a[src$=".pdf"] | 选择其 src 属性以 ".pdf" 结尾的所有 <a> 元素。 |
| [attribute*=value] | a[src*="abc"] | 选择其 src 属性中包含 "abc" 子串的每个 <a> 元素。 |
| :first-of-type | p:first-of-type | 选择属于其父元素的首个 <p> 元素的每个 <p> 元素。 |
| :last-of-type | p:last-of-type | 选择属于其父元素的最后 <p> 元素的每个 <p> 元素。 |
| :only-of-type | p:only-of-type | 选择属于其父元素唯一的 <p> 元素的每个 <p> 元素。 |
| :only-child | p:only-child | 选择属于其父元素的唯一子元素的每个 <p> 元素。 |
| :nth-child(n) | p:nth-child(2) | 选择属于其父元素的第二个子元素的每个 <p> 元素。 |
| :nth-last-child(n) | p:nth-last-child(2) | 同上，从最后一个子元素开始计数。 |
| :nth-of-type(n) | p:nth-of-type(2) | 选择属于其父元素第二个 <p> 元素的每个 <p> 元素。 |
| :nth-last-of-type(n) | p:nth-last-of-type(2) | 同上，但是从最后一个子元素开始计数。 |
| :last-child | p:last-child | 选择属于其父元素最后一个子元素每个 <p> 元素。 |
| :root | :root | 选择文档的根元素。 |
| :empty | p:empty | 选择没有子元素的每个 <p> 元素（包括文本节点）。 |
| :target | #news:target | 选择当前活动的 #news 元素。 |
| :enabled | input:enabled | 选择每个启用的 <input> 元素。 |
| :disabled | input:disabled | 选择每个禁用的 <input> 元素 |
| :checked | input:checked | 选择每个被选中的 <input> 元素。 |
| :not(selector) | :not(p) | 选择非 <p> 元素的每个元素。 |
| ::selection | ::selection | 选择被用户选取的元素部分。 |

- 媒体查询 @media screen

# CSS样式隔离

# CSS性能优化

> 避免使用**@import**，外部的css文件中使用@import会使得页面在加载时增加额外的延迟。

在 link 标签中去 @import 其他 css，页面会等到所有资源加载完成后，才开始解析 link 标签中 @import 的 css。

使用导入样式的缺点:

- 导入样式，只能放在 style 标签的第一行，放其他行则会无效。
- @import 声明的样式表不能充分利用浏览器并发请求资源的行为，其加载行为往往会延后触发或被其他资源加载挂起。

- 由于 @import 样式表的延后加载，可能会导致页面样式闪烁。

> 避免过分重排

- **浏览器重新计算元素位置和几何结构的过程为reflow**
- **任何一个节点触发来reflow，会导致他的子节点和祖先节点重新渲染**

其中导致reflow的情况:

1. 改变窗口的大小
2. 改变文字的大小
3. 添加 删除样式表
4. 内容的改变 输入框输入内容也会
5. 伪类的激活
6. 操作class属性
7. 脚本操作dom js改变css类
8. 计算offsetWidth和offsetHeight
9. 设置style属性
10. 改变元素的内外边距

减少reflow对性能的影响的建议:

1. 使用classs
2. 避免使用计算表达式
3. 不要table布局

> 避免过分repaint

1. 当一个元素的外观被改变，但是布局没有改变的情况
2. 当元素改变的时候，不影响元素在页面中的位置，浏览器仅仅会用新的样式重绘此元素

> CSS动画启用GPU加速, 3d加速

> 文件压缩

# 总结

- 避免使用**@import**
- 避免过分重排(**浏览器重新计算元素位置和几何结构的过程为reflow**)

- CSS动画启用GPU加速, 3d加速
- 文件压缩
- 减少css嵌套, 减少通配符
- 使用cssSprite

# div居中的几种方法

## 垂直居中

1. flex 布局实现 （元素已知宽度）
2. position
3. margin （元素已知宽度）
4. transform （元素未知宽度）
5. table-cell 布局实现 : table 实现垂直居中， 子集元素可以是块元素， 也可以不是块元素

```
.box {
        display: table-cell;
    vertical-align: middle;
}
```

## table-cell 应用

内容居中

```
.box{
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
```

**元素两端垂直对齐**

```
<style>
.box{
     display: table;
     width: 90%;
     margin: 10px  auto;
     padding: 10px;
     border: 1px solid green;
     height: 100px;
}
.left,.right{
    display: table-cell;
    width: 20%;
    border: 1px solid red;
}
.center{
    /* padding-top: 10px; */
    height: 100px;
    background-color: green;
}
p {
        margin: 0
}
</style>

<div class="box">
    <div class="left">
        <p>我是左边</p>
    </div>
    <div class="center">
        <p>我是中间</p>
    </div>
    <div class="right">
        <p>我是右边</p>
    </div>
</div>
```

## 总结

1. flex 布局实现 （元素已知宽度）
2. position
3. margin （元素已知宽度）
4. transform （元素未知宽度）

5. table-cell 布局实现 : table 实现垂直居中，子集元素可以是块元素，也可以不是块元素

# HTML&&浏览器

## 行内元素. 块级元素

行内元素有：
a, span, label, strong, em, br, img, input, select, textarea, cite,
块级元素：
div, h1~h6, p, form, ul, li, ol, dl, address, hr, menu, table, fieldset

块级元素的特点:

1. 占整行
2. 可以控制宽高
3. 可以容纳其他元素

行内元素的特点:

1. 不占一行.
2. 宽度与内容有关
3. 只能容纳文本或者其他行内元素

## 跨标签页通信

在浏览器中，我们可以同时打开多个Tab页，每个Tab页可以粗略理解为一个“独立”的运行环境，即使是全局对象也不会在多个Tab间共享。然而有些时候，我们希望能在这些“独立”的Tab页面之间同步页面的数据、信息或状态。

### 同源页面间的跨页面通信

#### 1. BroadCast Channel (广播通道)

下面的方式就可以创建一个标识为`AlienZHOU`的频道：

```
const bc = new BroadcastChannel('AlienZHOU');
```
复制代码

各个页面可以通过`onmessage`来监听被广播的消息：

```
bc.onmessage = function (e) {
    const data = e.data;
    const text = '[receive] ' + data.msg + ' —— tab ' + data.from;
    console.log('[BroadcastChannel] receive message:', text);
};
复制代码
```

要发送消息时只需要调用实例上的`postMessage`方法即可：

```
bc.postMessage(mydata);
```

### 2. Service Worker

### 3. LocalStorage

当 LocalStorage 变化时，会触发`storage`事件。利用这个特性，我们可以在发送消息时，把消息写入到某个 LocalStorage 中；然后在各个页面内，通过监听`storage`事件即可收到通知。

```
window.addEventListener('storage', function (e) {
    if (e.key === 'ctc-msg') {
        const data = JSON.parse(e.newValue);
        const text = '[receive] ' + data.msg + ' —— tab ' + data.from;
        console.log('[Storage I] receive message:', text);
    }
});
```

注意，这里有一个细节：我们在mydata上添加了一个取当前毫秒时间戳的`.st`属性。这是因为，`storage`事件只有在值真正改变时才会触发。举个例子：

```
window.localStorage.setItem('test', '123');
window.localStorage.setItem('test', '123');
```

由于第二次的值`'123'`与第一次的值相同，所以以上的代码只会在第一次`setItem`时触发`storage`事件。因此我们通过设置`st`来保证每次调用时一定会触发`storage`事件。

### 4. Shared Worker

### 5. IndexedDB

其思路很简单：与 Shared Worker 方案类似，消息发送方将消息存至 IndexedDB 中；接收方（例如所有页面）则通过轮询去获取最新的信息。在这之前，我们先简单封装几个 IndexedDB 的工具方法。

- 打开数据库连接：

```
function openStore() {
    const storeName = 'ctc_aleinzhou';
    return new Promise(function (resolve, reject) {
        if (!('indexedDB' in window)) {
            return reject('don\'t support indexedDB');
        }
        const request = indexedDB.open('CTC_DB', 1);
        request.onerror = reject;
        request.onsuccess =  e => resolve(e.target.result);
        request.onupgradeneeded = function (e) {
            const db = e.srcElement.result;
            if (e.oldVersion === 0 && !db.objectStoreNames.contains(storeName)) {
                const store = db.createObjectStore(storeName, {keyPath: 'tag'});
                store.createIndex(storeName + 'Index', 'tag', {unique: false});
            }
        }
    });
}
复制代码
```

- 存储数据

```
function saveData(db, data) {
    return new Promise(function (resolve, reject) {
        const STORE_NAME = 'ctc_aleinzhou';
        const tx = db.transaction(STORE_NAME, 'readwrite');
        const store = tx.objectStore(STORE_NAME);
        const request = store.put({tag: 'ctc_data', data});
        request.onsuccess = () => resolve(db);
        request.onerror = reject;
    });
}
复制代码
```

- 查询/读取数据

```
function query(db) {
    const STORE_NAME = 'ctc_aleinzhou';
    return new Promise(function (resolve, reject) {
        try {
            const tx = db.transaction(STORE_NAME, 'readonly');
            const store = tx.objectStore(STORE_NAME);
            const dbRequest = store.get('ctc_data');
            dbRequest.onsuccess = e => resolve(e.target.result);
            dbRequest.onerror = reject;
        }
        catch (err) {
            reject(err);
        }
    });
}
复制代码
```

剩下的工作就非常简单了。首先打开数据连接，并初始化数据：

```
openStore().then(db => saveData(db, null))
复制代码
```

对于消息读取，可以在连接与初始化后轮询：

```
openStore().then(db => saveData(db, null)).then(function (db) {
    setInterval(function () {
        query(db).then(function (res) {
            if (!res || !res.data) {
                return;
            }
            const data = res.data;
            const text = '[receive] ' + data.msg + ' —— tab ' + data.from;
            console.log('[Storage I] receive message:', text);
        });
    }, 1000);
});
复制代码
```

最后，要发送消息时，只需向 IndexedDB 存储数据即可：

```
openStore().then(db => saveData(db, null)).then(function (db) {
    // …… 省略上面的轮询代码
    // 触发 saveData 的方法可以放在用户操作的事件监听内
    saveData(db, mydata);
});
```

## 非同源页面之间的通信

### iframe

页面与 iframe 通信非常简单，首先需要在页面中监听 iframe 发来的消息，做相应的业务处理：

```
/* 业务页面代码 */
window.addEventListener('message', function (e) {
    // …… do something
});
复制代码
```

然后，当页面要与其他的同源或非同源页面通信时，会先给 iframe 发送消息：

```
/* 业务页面代码 */
window.frames[0].window.postMessage(mydata, '*');
复制代码
```

其中为了简便此处将`postMessage`的第二个参数设为了`'*'`，你也可以设为 iframe 的 URL。iframe 收到消息后，会使用某种跨页面消息通信技术在所有 iframe 间同步消息，例如下面使用的 Broadcast Channel：

```
/* iframe 内代码 */
const bc = new BroadcastChannel('AlienZHOU');
// 收到来自页面的消息后，在 iframe 间进行广播
window.addEventListener('message', function (e) {
    bc.postMessage(e.data);
});
复制代码
```

其他 iframe 收到通知后，则会将该消息同步给所属的页面：

```
/* iframe 内代码 */
// 对于收到的 (iframe) 广播消息，通知给所属的业务页面
bc.onmessage = function (e) {
    window.parent.postMessage(e.data, '*');
};
```

下图就是使用 iframe 作为“桥”的非同源页面间通信模式图。

img

## 总结

同源页面之间通信

- broadcast channel
- service worker
- LocalStorage
- indexedDB
- shared worker

非同源页面通信

- iframe

# history和hash两种路由

- hash模式：监听浏览器地址hash值变化，执行相应的js切换网页；
- history模式：利用history API实现url地址改变，网页内容改变；

## history

> history.pushState(object, title, url) 向历史添加一条记录

`pushState()` 方法不会触发页面刷新，只是导致 History 对象发生变化，地址栏会有变化。

- object：是一个对象，通过 pushState 方法可以将该对象内容传递到新页面中。如果不需要这个对象，此处可以填 null。
- title：指标题，几乎没有浏览器支持该参数，传一个空字符串比较安全。
- url：新的网址，必须与当前页面处在同一个域。不指定的话则为当前的路径，如果设置了一个跨域网址，则会报错。

**注意**：如果 pushState 的 URL 参数设置了一个新的锚点值（即 hash），并不会触发 hashchange 事件。反过来，如果 URL 的锚点值变了，则会在 History 对象创建一条浏览记录。

> history.replaceState(object, title, url) 替换当前地址.

用法与 pushState() 方法一样。

> popstate 事件

每当 history 对象出现变化时，就会触发 popstate 事件。

注意：

- 仅仅调用`pushState()`方法或`replaceState()`方法，并不会触发该事件;
- **只有用户点击浏览器倒退按钮和前进按钮，或者使用 JavaScript 调用`History.back()`、`History.forward()`、`History.go()`方法时才会触发。**
- 另外，该事件只针对同一个文档，如果浏览历史的切换，导致加载不同的文档，该事件也不会触发。
- 页面第一次加载的时候，浏览器不会触发popstate事件。

使用的时候，可以为popstate事件指定回调函数，回调函数的参数是一个 event 事件对象，它的 state 属性指向当前的 state 对象。

```
window.addEventListener('popstate', function(e) {
        //e.state 相当于 history.state
        console.log('state: ' + JSON.stringify(e.state));
        console.log(history.state);
})
```

## 总结

前端路由方式有两种. hash模式和history模式

- hash模式使用onhashchange监听浏览器地址的hash值变化 . 即#xx
- history模式 利用history api实现地址改变. 使用popstate事件监听history 对象变化

`pushState(), replaceState`方法不会触发页面刷新，只是导致 History 对象发生变化，地址栏会有变化。

# DOM树

- 在解析过程中遇到 JavaScript 脚本，DOM 解析器是如何处理的？
- DOM 解析器是如何处理跨站点资源的?

## 什么是 DOM

从网络传给渲染引擎的 HTML 文件字节流是无法直接被渲染引擎理解的，所以要将其转化为渲染引擎能够理解的内部结构，这个结构就是 DOM。DOM 提供了对 HTML 文档结构化的表述.

在渲染引擎中，DOM 有三个层面的作用

- 从页面的视角来看，DOM 是生成页面的基础数据结构。
- 从 JavaScript 脚本视角来看，DOM 提供给 JavaScript 脚本操作的接口，通过这套接口，JavaScript 可以对 DOM 结构进行访问，从而改变文档的结构、样式和内容。
- 从安全视角来看，DOM 是一道安全防护线，一些不安全的内容在 DOM 解析阶段就被拒之门外了。

**DOM 是表述 HTML 的内部数据结构，它会将 Web 页面和 JavaScript 脚本连接起来，并过滤一些不安全的内容。**

## DOM 树如何生成

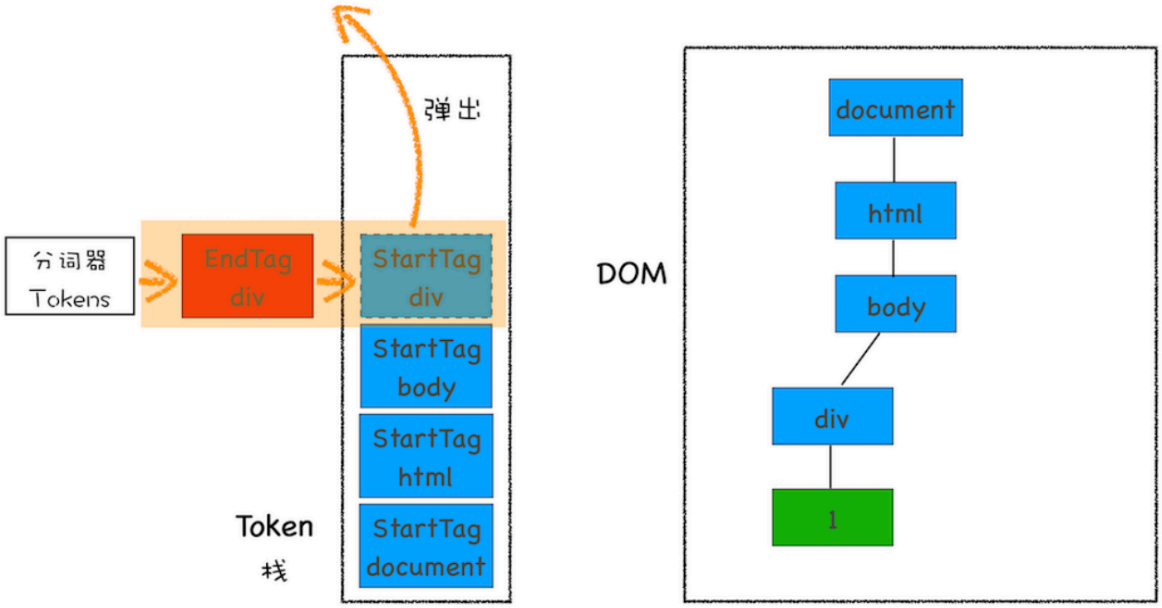
在渲染引擎内部，有一个叫HTML 解析器（**HTMLParser**）的模块，它的职责就是负责将 HTML 字节流转换为 DOM 结构。

网络进程加载了多少数据，HTML 解析器便解析多少数据。

那详细的流程是怎样的呢？网络进程接收到响应头之后，会根据响应头中的 content-type 字段来判断文件的类型，比如 content-type 的值是“text/html”，那么浏览器就会判断这是一个 HTML 类型的文件，然后**为该请求选择或者创建一个渲染进程**。渲染进程准备好之后，**网络进程和渲染进程之间会建立一个共享数据的管道**，网络进程接收到数据后就往这个管道里面放，而**渲染进程则从管道的另外一端不断地读取数据，并同时将读取的数据“喂”给 HTML 解析器**。你可以把这个管道想象成一个“水管”，网络进程接收到的字节流像水一样倒进这个“水管”，而“水管”的另外一端是渲染进程的 HTML 解析器，它会动态接收字节流，并将其解析为 DOM。

> HTML 解析器维护了一个Token 栈结构，该 Token 栈主要用来计算节点之间的父子关系，在第一个阶段中生成的 Token 会被按照顺序压到这个栈中。具体的处理规则如下所示：

- 如果压入到栈中的是StartTag Token，HTML 解析器会为该 Token 创建一个 DOM 节点，然后将该节点加入到 DOM 树中，它的父节点就是栈中相邻的那个元素生成的节点。
- 如果分词器解析出来是文本 Token，那么会生成一个文本节点，然后将该节点加入到 DOM 树中，文本 Token 是不需要压入到栈中，它的父节点就是当前栈顶 Token 所对应的 DOM 节点。
- 如果分词器解析出来的是EndTag 标签，比如是 EndTag div，HTML 解析器会查看 Token 栈顶的元素是否是 StarTag div，如果是，就将 StartTag div 从栈中弹出，表示该 div 元素解析完成。




元素弹出 Token 栈示意图

## JavaScript 是如何影响 DOM 生成的

当解析到 script 脚本标签时.

```
<html>
<body>
    <div>1</div>
    <script>
    let div1 = document.getElementsByTagName('div')[0]
    div1.innerText = 'time.geekbang'
    </script>
    <div>test</div>
</body>
</html>
```

这时候 HTML 解析器暂停工作，JavaScript 引擎介入，并执行 script 标签中的这段脚本.

当在页面中引入 JavaScript 文件时.

```
<html>
<body>
    <div>1</div>
    <script type="text/javascript" src='foo.js'></script>
    <div>test</div>
</body>
</html>
```

其整个执行流程还是一样的，执行到 JavaScript 标签时，暂停整个 DOM 的解析，执行 JavaScript 代码，不过这里执行 JavaScript 时，需要先**下载这段 JavaScript 代码**。这里需要重点关注下载环境，**因为JavaScript 文件的下载过程会阻塞 DOM 解析**，而通常下载又是非常耗时的，会受到网络环境、JavaScript 文件大小等因素的影响。

其中浏览器主要优化是**预解析操作**. 一些相关的策略来规避:

- CDN加速, 压缩js文件
- 没有操作DOM. 设为异步加载 async, defer
    - async 脚本文件一旦加载完成，会立即执行
    - defer 需要在完全加载和解析完成之后执行

## 总结

DOM是表述HTML的内容数据结构, 它将web页面与javascript脚本连接起来,并过滤一些不安全内容.

DOM树由HTML解析器, 把HTML字节流转换成DOM结构.

HTML 解析器并不是等整个文档加载完成之后再解析的，而是网络进程加载了多少数据，HTML 解析器便解析多少数据。

DOM树建立流程:

网络进程判断这个请求是html类型的文件 ===> 创建或选择一个渲染进程 => 网络进程与这个渲染进程建立一个共享数据管道, 网络进程接收到数据, 则往管道放=> 渲染进程从管道读取数据 ===> 最后HTML解析器解析成DOM结构

当解析流程遇到javascript脚本时. 会暂停DOM解析. 执行javascript代码. 若是外链js文件. 则需要先下载这js文件. 可能会阻塞 DOM 解析.

优化方案:

- CDN加速js文件下载. js文件压缩
- 若没有DOM操作, 启用异步加载. 添加async / defer 属性
    - async: 文件加载完成, 立即执行
    - defer: 在DOM完全加载和解析完成之后执行.

# 事件模型

## EventTarget 接口

DOM 的事件操作（监听和触发），都定义在`EventTarget`接口。所有节点对象都部署了这个接口，其他一些需要事件通信的浏览器内置对象（比如，`XMLHttpRequest`、`AudioNode`、`AudioContext`）也部署了这个接口。

该接口主要提供三个实例方法。

- `addEventListener`：绑定事件的监听函数
- `removeEventListener`：移除事件的监听函数
- `dispatchEvent`：触发事件

### addEventListener

关于参数，有两个地方需要注意。

首先，第二个参数除了监听函数，还可以是一个具有`handleEvent`方法的对象。

```
document.addEventListener('click', {
  handleEvent: function (event) {
    console.log('click');
  }
}, {
  capture: false, once: true, passive: true
})
```

上面代码中，`addEventListener`方法的第二个参数，就是一个具有`handleEvent`方法的对象。

其次，第三个参数除了布尔值`useCapture`，还可以是一个属性配置对象。该对象有以下属性。

> - `capture`：布尔值，表示该事件是否在 捕获阶段 触发监听函数。
> - `once`：布尔值，表示监听函数是否只触发一次，然后就自动移除。
> - `passive`：布尔值，表示监听函数不会调用事件的 `preventDefault` 方法。如果监听函数调用了，浏览器将忽略这个要求，并在监控台输出一行警告。

`addEventListener`方法可以为针对当前对象的同一个事件，添加多个不同的监听函数。这些函数按照添加顺序触发，即先添加先触发。**如果为同一个事件多次添加同一个监听函数，该函数只会执行一次，多余的添加将自动被去除**（不必使用`removeEventListener`方法手动去除）。

如果希望向监听函数传递参数，可以用匿名函数包装一下监听函数。

```
function print(x) {
    console.log(x)
}
var a = '123'
document.addEventListener('click', function(){ print(a) }, false);
```

### removeEventListener

参数必须与`addEventListener`方法完全一致. 否则失效. 移除的监听函数，必须是`addEventListener`方法添加的那个监听函数，而且必须在同一个元素节点，否则无效。

### dispatchEvent

`EventTarget.dispatchEvent`方法在当前节点上触发指定事件，从而触发监听函数的执行。该方法返回一个布尔值，只要有一个监听函数(**阻止了默认事件**)调用了`Event.preventDefault()`，则返回值为`false`，否则为`true`。

`dispatchEvent`方法的参数是一个`Event`对象的实例

```
document.addEventListener('click', function(){console.log('hello')}, false);
var event = new Event('click');
document.dispatchEvent(event);
```

## 监听函数

- html 的 on-属性 冒泡事件
- 元素的事件属性 el.onXXX
- dom.addEventListener

> html 的 on-属性

```
<body onload="doSomething()">
// 等同于
document.body.setAttribute('onclick', 'doSomething()');
```

## 事件的传播

一个事件发生后，会在子元素和父元素之间传播（propagation）。这种传播分成三个阶段。

- **第一阶段**：从`window`对象传导到目标节点（上层传到底层），称为“捕获阶段”（capture phase）。
- **第二阶段**：在目标节点上触发，称为“目标阶段”（target phase）。
- **第三阶段**：从目标节点传导回`window`对象（从底层传回上层），称为“冒泡阶段”（bubbling phase）。

捕获阶段(window到目标) -> 目标阶段(触发) -> 冒泡阶段(目标到window).

## 事件的代理

由父节点监听函数统一处理多个子节点的事件. 叫做事件代理.

```
var ul = document.querySelector('ul');

// li
ul.addEventListener('click', function (event) {
  if(event.target.tagName.toLowerCase() === 'li'){
     // some code
  }
});
```

如果希望事件不再传播，使用 `event.stopPropagation()`；

```
// 事件传播到 document 元素后，就不再向下传播了
p.addEventListener('click', function (event) {
  event.stopPropagation();
}, true);

// 事件冒泡到 document 元素后，就不再向上冒泡了
p.addEventListener('click', function (event) {
  event.stopPropagation();
}, false);
```

但是，`stopPropagation`方法只会阻止这个事件的传播.不会阻止该事件触发`<p>`节点的其他`click`事件的监听函数。也就是说，不是彻底取消`click`事件。

**如果想要彻底阻止这个事件的传播，不再触发后面所有`click`的监听函数，可以使用`stopImmediatePropagation`方法。**

stopImmediatePropagation() 只能阻止后面绑定的事件. 之前绑定过的不能阻止.

```
document.addEventListener('click', function (event) {
  event.stopImmediatePropagation();
  console.log(1);
});

document.addEventListener('click', function(event) {
  // 不会被触发
  console.log(2);
});
```

## Event 对象

浏览器原生提供一个`Event`对象，所有的事件都是这个对象的实例，或者说继承

了`Event.prototype`对象。

`Event`对象本身就是一个构造函数，可以用来生成新的实例。

```
event = new Event(type, options);
```

`Event`构造函数接受两个参数。第一个参数`type`是字符串，表示事件的名称；第二个参数`options`是一个对象，表示事件对象的配置。该对象主要有下面两个属性。

- `bubbles`：布尔值，可选，默认为`false`，表示事件对象是否冒泡。
- `cancelable`：布尔值，可选，默认为`false`，表示事件是否可以被取消，即能否用`Event.preventDefault()`取消这个事件。一旦事件被取消，就好像从来没有发生过，不会触发浏览器对该事件的默认行为。

## Event 对象的属性

Event.currentTarget，Event.target

`Event.currentTarget`属性返回事件**当前所在的节点，即正在执行的监听函数所绑定的那个节点**。

`Event.target`属性返回**原始触发事件的那个节点，即事件最初发生的节点**。事件传播过程中，不同节点的监听函数内部的`Event.target`与`Event.currentTarget`属性的值是不一样的，前者总是不变的，后者则是指向监听函数所在的那个节点对象。

```
// HTML代码为
// <p id="para">Hello <em>World</em></p>
function hide(e) {
  console.log(this === e.currentTarget);  // 总是 true
  console.log(this === e.target);  // 有可能不是 true
  e.target.style.visibility = 'hidden';
}

para.addEventListener('click', hide, false);
```

上面代码中，如果在`para`节点的`<em>`子节点上面点击，则`e.target`指向`<em>`子节点，导致`<em>`子节点（即 World 部分）会不可见。如果点击 Hello 部分，则整个`para`都将不可见。

## Event 对象的实例方法

- `Event.preventDefault`方法取消浏览器对当前事件的默认行为。比如点击链接后，浏览器默认会跳转到另一个页面，使用这个方法以后，就不会跳转了
- `Event.stopPropagation`方法阻止事件在 DOM 中继续传播，防止再触发定义在别的节点上的监听函数，但是不包括在当前节点上其他的事件监听函数。
- `Event.stopImmediatePropagation`方法阻止同一个事件的其他监听函数被调用，不管监听函数定义在当前节点还是其他节点。也就是说，该方法阻止事件的传播，比`Event.stopPropagation()`更彻底。

## CustomEvent 接口

```
var event = new CustomEvent('build', { 'detail': 'hello' });

function eventHandler(e) {
  console.log(e.detail);
}

document.body.addEventListener('build', function (e) {
  console.log(e.detail);
});

document.body.dispatchEvent(event);
```

## 总结

addEventListener接收三个参数, 事件名, 事件函数, 是否捕获触发. 第二参数可以是 对象. 第三参数也可以是对象

一个具有`handleEvent`方法的对象。

其次，第三个参数除了布尔值`useCapture`，还可以是一个属性配置对象。该对象有以下属性。

> - `capture` ：布尔值，表示该事件是否在 捕获阶段 触发监听函数。
> - `once` ：布尔值，表示监听函数是否只触发一次，然后就自动移除。
> - `passive` ：布尔值，表示监听函数不会调用事件的 `preventDefault` 方法。如果监听函数调用了，浏览器将忽略这个要求，并在监控台输出一行警告。

事件传播三阶段: 捕获阶段(顶层到目标) -> 目标阶段 -> 冒泡阶段(从目标冒泡到顶层)

dispatchEvent接收event对象. 并触发事件函数.

监听函数有三种方法:

- html的on-属性
- 元素的事件属性
- addEventListener函数

事件代理: 由父节点监听函数处理多个子节点的事件.

不想让事件传播可以使用. event.stopPropagation() 阻止该事件传播.
event.stopImmediatePropagation() 可以阻止, 其后面绑定的所有相同事件.

event.preventDefault() 可以取消浏览器对该事件的默认行为.

- event.currentTaget: **指事件当前所在的节点, 即正在执行的监听函数所绑定的那个节点**。
- event.target: **指触发事件的节点, 即事件最初发生的节点**

# 浏览器缓存机制

## 介绍http报文

- 请求报文: 请求行 + HTTP头 + 请求主体(post请求才有)
- 响应报文: 状态行 + HTTP头 + 响应主体
- HTTP头包含通用信息头, 请求头/响应头, 实体头

HTTP请求(Request)报文，报文格式为：**请求行 – HTTP头(通用信息头，请求头，实体头) – 请求报文主体(只有POST才有报文主体)**，如下图

img

img

HTTP响应(Response)报文，报文格式为：**状态行 – HTTP头(通用信息头，响应头，实体头) – 响应报文主体**，如下图

img

img

注：通用信息头指的是请求和响应报文都支持的头域，分别为Cache-Control、Connection、Date、Pragma、Transfer-Encoding、Upgrade、Via；实体头则是实体信息的实体头域，分别为Allow、Content-Base、Content-Encoding、Content-Language、Content-Length、Content-Location、Content-MD5、Content-Range、Content-Type、Etag、Expires、Last-Modified、extension-header。这里只是为了方便理解，将通用信息头，响应头/请求头，实体头都归为了HTTP头。

## 缓存过程分析

浏览器与服务器通信的方式为应答模式，即是：**浏览器发起HTTP请求 – 服务器响应该请求。那么浏览器第一次向服务器发起该请求后拿到请求结果，会根据响应报文中HTTP头的缓存标识，决定是否缓存结果，是则将请求结果和缓存标识存入浏览器缓存中**，简单的过程如下图：

img

由上图我们可以知道：

- 浏览器每次发起请求，都会先在浏览器缓存中查找该请求的结果以及缓存标识
- 浏览器每次拿到返回的请求结果都会将该结果和缓存标识存入浏览器缓存中

这里我们根据是否需要向服务器重新发起HTTP请求将缓存过程分为两个部分，分别是强制缓存和协商缓存。

浏览器http请求 -> 浏览器缓存 (是否有缓存) -> (没有)请求服务器,返回结果给浏览器 -> 浏览器将请求结果存入缓存(当前进程缓存/硬盘缓存)中.

## 强缓存

强制缓存的情况主要有三种

1. 没有缓存结果和标识, 则直接向服务器发起请求
2. **存在该缓存结果和缓存标识，但该结果已失效, 带缓存标识向浏览器请求(使用协商缓存)**
3. **存在该缓存结果和缓存标识. 结果有效，直接返回该结果**

强缓存规则是 **当浏览器向服务器发起请求时，服务器会将缓存规则放入HTTP响应报文的HTTP头中和请求结果一起返回给浏览器. 控制字段是Expires和Cache-Control. 其中Cache-Control优先**.

**Expires(HTTP/1.0)**

存储的值是**服务器返回该缓存结果的到期时间**.

被cache-control替代的原因是客户端和服务端可能因为时区的不同发生误差.

现在浏览器默认使用的是HTTP/1.1

**Cache-Control(HTTP/1.1)**

主要取值为：

- public：所有内容都将被缓存（客户端和代理服务器都可缓存）
- private：所有内容只有客户端可以缓存，Cache-Control的默认取值
- no-cache：客户端缓存内容，但是是否使用缓存则需要经过协商缓存来验证决定
- no-store：所有内容都不会被缓存，即不使用强制缓存，也不使用协商缓存
- max-age=xxx (xxx is numeric)：**缓存内容将在xxx秒后失效**

内存缓存(from memory cache)和硬盘缓存(from disk cache)，如下:

- 内存缓存(from memory cache)：内存缓存具有两个特点，分别是快速读取和时效性：
    - 快速读取：内存缓存会将编译解析后的文件，直接存入该进程的内存中，占据该进程一定的内存资源，以方便下次运行使用时的快速读取。
    - 时效性：一旦该进程关闭，则该进程的内存则会清空。
- 硬盘缓存(from disk cache)：硬盘缓存则是直接将缓存写入硬盘文件中，读取缓存需要对该缓存存放的硬盘文件进行I/O操作，然后重新解析该缓存内容，读取复杂，速度比内存缓存慢。

在浏览器中，**浏览器会在js和图片等文件解析执行后直接存入内存缓存中**，那么当刷新页面时只需直接从内存缓存中读取(**from memory cache**)；**而css文件则会存入硬盘文件中，所以每次渲染页面都需要从硬盘读取缓存(from disk cache)。**

访问https://heyingye.github.io/

img

关闭博客的标签页

重新打开https://heyingye.github.io/

img

刷新

img

## 协商缓存

协商缓存就是在强缓存结果失效后, 带着缓存标识向服务器发起请求, 由服务器根据缓存标识决定是否使用缓存的过程. 有两种情况.

1. 协商缓存生效, 返回304, 缓存没有更新.
2. 协商缓存失效. 返回200和新的请求结果.

控制字段: `Last-Modified` / `If-Modified-Since` 和 `Etag` / `If-None-Match`; 其中 `Etag` / `If-None-Match` 优先级比较高

**`Last-Modified` / `If-Modified-Since`**

`Last-Modified` 是服务器响应请求时，返回该资源文件在服务器最后被修改的时间

`If-Modified-Since` 是客户端再次发起该请求时，携带上次请求返回的 `Last-Modified` 值

**服务器根据 `If-Modified-Since` 的值与该资源最后修改时间做对比**, 有更新则返回200和新结果. 没有返回304.继续使用缓存.

**`Etag` / `If-None-Match`**

`Etag` 是服务器响应请求时，返回当前资源文件的一个唯一标识(由服务器生成)

`If-None-Match` 是客户端再次发起该请求时，携带上次请求返回的唯一标识 `Etag` 值

**服务器根据 `If-None-Match` 的值与该资源存在服务器的Etag值做对比**, 一致则返回304，代表资源无更新，继续使用缓存文件；不一致则重新返回资源文件，状态码为200

## 总结

http报文由 请求报文和响应报文.

构成是 请求行/状态行 + HTTP头(通用信息头 + 请求头/响应头 + 实体头 ) + 请求主体(post请求才有)/响应主体

缓存过程: 浏览器发去HTTP请求 -> 浏览器缓存是否有缓存标识和缓存结果且是否有效 -> 发送给服务器返回请求结果(有标识无效/没标识-> 200+结果)(有标识且有效 -> 304 继续使用缓存) -> 浏览器存储请求结果到缓存中.

缓存分为: 当前进程缓存和硬盘缓存. 当标签页存在时使用进程缓存. 被关闭重新打开使用硬盘缓存. css时效性不高刷新时使用硬盘缓存. js使用进程缓存.

Expires(HTTP/1.0)值为服务器该缓存结果到期时间. 优先级低于Cache-Control(HTTP/1.1) . 原因是Expires可能存在时区误差.

If-Modified-Since值为Last-Modified(该资源最后修改时间), 客户端再次发送请求时携带该值与服务器该资源最后修改时间做对比. 优先级低于If-None-Match值为Etag.

If-None-Match值为Etag(服务器生成该资源的唯一标识), 客户端再次发送请求时携带该值与服务器Etag做对比.

强制缓存优先于协商缓存进行. 若强制缓存(Expires和**Cache-Control**)生效则直接使用缓存，若不生效则进行协商缓存(Last-Modified / If-Modified-Since和**Etag / If-None-Match**)，协商缓存由服务器决定是否使用缓存，若协商缓存失效，那么代表该请求的缓存失效，重新获取请求结果，再存入浏览器缓存中；生效则返回304，继续使用缓存.

img

## 浏览器架构

从一次常见的访问入手，逐步了解浏览器是如何展示页面的。

1. 输入处理 ===> UI线程会先判断我们输入的内容是要搜索的内容还是要访问一个站点，因为地址栏同时也是一个搜索框。
2. 访问开始 ===> UI线程将借助网络线程访问站点资源. 浏览器页签的标题上会出现加载中的图标，同时网络线程会根据适当的网络协议，例如DNS lookup和TLS为这次请求建立连接。
3. 处理响应数据 ===> 根据 数据的类型（**Content-Type**）. 作不同处理. 是HTML时，会将数据传递给渲染进程做进一步的渲染工作。但是如果数据类型是zip文件或者其他文件格式时，会将数据传递给下载管理器做进一步的文件预览或者下载工作。 在开始渲染之前，网络线程要先检查数据的安全性.
4. 渲染过程 ===> 当所有的检查结束后，网络线程确信浏览器可以访问站点时，网络线程通知UI线程数据已经准备好了。UI线程会根据当前的站点找到一个渲染进程(通

知网络进程这个地址时同步进行)完成接下来的渲染工作。当然，如果出现重定向的请求时，提前初始化的渲染进程可能就不会被使用了

5. 提交访问 ===> 经历前面的步骤，数据和渲染进程都已经准备好了。浏览器进程会通过IPC向渲染进程提交这次访问，同时也会保证渲染进程可以通过网络线程继续获取数据。一旦浏览器进程收到来自渲染进程的确认完毕的消息，就意味着访问的过程结束了，文档渲染的过程就开始了。这时，地址栏显示出表明安全的图标，同时显示出站点的信息。访问历史中也会加入当前的站点信息。为了能恢复访问历史信息，当页签或窗口被关闭时，访问历史的信息会被存储在硬盘中。
6. 加载完毕 ===> 当访问被提交给渲染进程，渲染进程会继续加载页面资源并且渲染页面。当渲染进程"结束"渲染工作，会给浏览器进程发送消息，这个消息会在页面中所有子页面（frame）结束加载后发出，也就是onLoad事件触发后发送。当收到"结束"消息后，UI线程会隐藏页签标题上的加载状态图标，表明页面加载完毕。但这里"结束"并不意味着所有的加载工作都结束了，因为可能还有JavaScript在加载额外的资源或者渲染新的视图。

### 总结

- 浏览器进程做为最重要的进程负责大多数页签外部的工作，包括地址栏显示、网络请求、页签状态管理等。
- 不同的渲染进程负责不同的站点渲染工作，渲染进程间彼此独立。
- 渲染进程在渲染页面的过程中会通过浏览器进程获取站点资源，只有安全的资源才会被渲染进程接收到。
- 渲染进程中主线程负责除了图像生成外绝大多数工作，如何减少主线程上代码的运行是交互性能优化的关键。
- 渲染进程中的合成线程和栅格线程负责图像生成，利用分层技术可以优化图像生成的效率。
- 当用户与页面发生交互时，事件的传播途径从浏览器进程到渲染进程的合成线程再根据事件监听的区域决定是否要传递给渲染进程的主线程处理。

## 浏览器内存泄露

### 内存生命周期

内存也是有**生命周期**的，不管什么程序语言，一般可以按顺序分为三个周期：

- 分配期：分配所需要的内存
- 使用期：使用分配到的内存（读、写）

- 释放期：不需要时将其释放和归还

内存分配 -> 内存使用 -> 内存释放。

## 什么是内存泄漏？

> 在计算机科学中，**内存泄漏**指由于疏忽或错误造成程序未能释放已经不再使用的内存。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

无用的内存还在占用，得不到释放和归还。比较严重时，无用的内存会持续递增，从而导致整个系统卡顿，甚至崩溃。

## JavaScript 内存管理机制

### 引用计数垃圾收集

> 这是最初级的垃圾收集算法。此算法把“对象是否不再需要”简化定义为“对象有没有其他对象引用到它”。如果没有引用指向该对象（零引用），对象将被垃圾回收机制回收。

看下下面的例子，“这个对象”的内存被回收了吗？

```
// “这个对象”分配给 a 变量
var a = {
  a: 1,
  b: 2,
}
// b 引用“这个对象”
var b = a;
// 现在，“这个对象”的原始引用 a 被 b 替换了
a = 1;
```

当前执行环境中，“这个对象”内存还没有被回收的，需要手动释放“这个对象”的内存（当然是还没离开执行环境的情况下），例如：

```
b = null;
// 或者 b = 1，反正替换“这个对象”就行了
```

这样引用的"这个对象"的内存就被回收了。

ES6 把**引用**有区分为**强引用**和**弱引用**，这个目前只有再 Set 和 Map 中才有。

**强引用**才会有**引用计数**叠加，只有引用计数为 0 的对象的内存才会被回收，所以一般需要手动回收内存（手动回收的前提在于**标记清除法**还没执行，还处于当前执行环境）。

而**弱引用**没有触发**引用计数**叠加，只要引用计数为 0，弱引用就会自动消失，无需手动回收内存。

**限制：循环引用**

该算法有个限制：无法处理循环引用的事例。在下面的例子中，两个对象被创建，并互相引用，形成了一个循环。它们被调用之后会离开函数作用域，所以它们已经没有用了，可以被回收了。然而，引用计数算法考虑到它们互相都有至少一次引用，所以它们不会被回收。

**标记清除法**

从2012年起，所有现代浏览器都使用了标记-清除垃圾回收算法.

这个算法把“对象是否不再需要”简化定义为“对象是否可以获得”。

> 当变量进入执行环境时标记为“进入环境”，当变量离开执行环境时则标记为“离开环境”，被标记为“进入环境”的变量是不能被回收的，因为它们正在被使用，而标记为“离开环境”的变量则可以被回收

环境可以理解为我们的作用域，但是全局作用域的变量只会在页面关闭才会销毁。

```
// 假设这里是全局变量
// b 被标记进入环境
var b = 2;
function test() {
  var a = 1;
  // 函数执行时，a 被标记进入环境
  return a + b;
}
// 函数执行结束，a 被标记离开环境，被回收
// 但是 b 就没有被标记离开环境
test();
```

## JavaScript 内存泄漏的一些场景

- 意外的全局变量
- 被遗忘的计时器(组件销毁时没清除)
- 被遗忘的事件监听(订阅)器(组件销毁时没清除)

- Set, Map的引用. (推荐WeakSet. WeakMap)
- dom的引用
- 闭包

## 总结

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Memory_Management

内存泄漏: 由于疏忽或错误造成程序未能释放已经不再使用的内存.

**标记清除法**

从2012年起，所有现代浏览器都使用了标记-清除垃圾回收算法.

这个算法把“对象是否不再需要”简化定义为“对象是否可以获得”。

这个算法假定设置一个叫做根（root）的对象（在Javascript里，根是全局对象）。垃圾回收器将定期从根开始，找所有从根开始引用的对象，然后找这些对象引用的对象……从根开始，垃圾回收器将找到所有可以获得的对象和收集所有不能获得的对象。

产生内存泄漏的场景:

- 意外的全局变量
- 被遗忘的计时器(组件销毁时没清除)
- 被遗忘的事件监听(订阅)器(组件销毁时没清除)
- Set, Map的引用. (推荐WeakSet. WeakMap)
- dom的引用
- 闭包

# 性能

## 前端性能优化指标RAIL

RAIL是一个旅程，为了提升用户在网站的交互体验而不断探索。你需要去理解用户如何感知你的站点，这样才能设置最佳的性能目标

- 聚焦用户
- 100ms内响应用户的输入
- 10ms内产生1帧，在滚动或者动画执行时

- 最大化主线程的空闲时间
- 5s内让网页变得可交互

## 前端性能优化手段