

1. 函数的 3 种定义方法

1.1 函数声明

```
//ES5
function getSum() {}
function () {} //匿名函数
//ES6()=>{}
//如果 {} 内容只有一行 {} 和 return 关键字可省,
```

1.2 函数表达式(函数字面量)

```
//ES5
var sum=function() {}
//ES6
let sum=()=>{} //如果 {} 内容只有一行 {} 和 return 关键字可省,
```

1.3 构造函数

```
var sum=new GetSum(num1, num2)
```

1.4 三种方法的对比

- 1.函数声明有预解析,而且函数声明的优先级高于变量;
- 2.使用 Function 构造函数定义函数的方式是一个函数表达式,这种方式会导致解析两次代码,影响性能。第一次解析常规的 JavaScript 代码,第二次解析传入构造函数的字符串

2.ES5 中函数的 4 种调用

在 ES5 中函数内容的 this 指向和调用方法有关

2.1 函数调用模式

包括函数名()和匿名函数调用,this 指向 window

```
function getSum() {
    console.log(this) //window
}
getSum()

(function() {
    console.log(this) //window
})();

var getSum=function() {
    console.log(this) //window
}
getSum()
```

2.2 方法调用

对象.方法名(),this 指向对象

```
var objList = {
    name: 'methods',
    getSum: function() {
        console.log(this) //objList 对象
    }
}
objList.getSum()
```

2.3 构造器调用

new 构造函数名(),this 指向构造函数

```
function Person() {
    console.log(this); //指向构造函数 Person
}
var personOne = new Person();
```

2.4 间接调用

利用 call 和 apply 来实现,this 就是 call 和 apply 对应的第一个参数,如果不

传值或者第一个值为 null,undefined 时 this 指向 window

```
function foo() {  
    console.log(this);  
}  
foo.apply('我是 apply 改变的 this 值');//我是 apply 改变的 this 值  
foo.call('我是 call 改变的 this 值');//我是 call 改变的 this 值
```

3.ES6 中函数的调用

箭头函数不可以当作构造函数使用，也就是不能用 new 命令实例化一个对象，否则会抛出一个错误。

箭头函数的 this 是和定义时有关和调用无关。

调用就是函数调用模式。

```
((() => {  
    console.log(this)//window  
}))()  
  
let arrowFun = () => {  
    console.log(this)//window}  
arrowFun()  
  
let arrowObj = {  
    arrFun: function() {  
        (() => {  
            console.log(this)//arrowObj  
        })()  
    }  
}  
arrowObj.arrFun();
```

4.call,apply 和 bind

1. IE5 之前不支持 call 和 apply, bind 是 ES5 出来的;

2. call 和 apply 可以调用函数, 改变 this, 实现继承和借用别的方法;

4.1 call 和 apply 定义

调用方法, 用一个对象替换掉另一个对象(this)

对象.call(新 this 对象, 实参 1, 实参 2, 实参 3.....)

对象.apply(新 this 对象, [实参 1, 实参 2, 实参 3.....])

4.2 call 和 apply 用法

1. 间接调用函数, 改变作用域的 this 值

2. 劫持其他对象的方法

```
var foo = {  
  name: "张三",  
  logName: function() {  
    console.log(this.name);  
  }  
}  
  
var bar = {  
  name: "李四"  
};
```

```
foo.logName.call(bar); // 李四
```

实质是 call 改变了 foo 的 this 指向为 bar, 并调用该函数

3. 两个函数实现继承

```
function Animal(name) {  
  this.name = name;  
  this.showName = function() {  
    console.log(this.name);  
  }  
}  
  
function Cat(name) {
```

```

        Animal.call(this, name);
    }
    var cat = new Cat("Black Cat");
    cat.showName(); //Black Cat

```

4.为类数组(arguments 和 nodeList)添加数组方法 push,pop

```

(function() {
    Array.prototype.push.call(arguments, '王五');
    console.log(arguments); //['张三', '李四', '王五']
})('张三', '李四')

```

5.合并数组

```

let arr1=[1,2,3];
let arr2=[4,5,6];
Array.prototype.push.apply(arr1,arr2); //将 arr2 合并到了 arr1 中

```

6.求数组最大值

```

Math.max.apply(null, arr)

```

7.判断字符类型

```

Object.prototype.toString.call({})

```

4.3 bind

bind 是 function 的一个函数扩展方法，bind 以后代码重新绑定了 func 内部的 this 指向,不会调用方法,不兼容 IE8。

```

var name = '李四'
var foo = {
    name: "张三",
    logName: function(age) {
        console.log(this.name, age);
    }
}

```

```
}  
  
var fooNew = foo.logName;  
var fooNewBind = foo.logName.bind(foo);  
fooNew(10)//李四,10  
fooNewBind(11)//张三,11    因为 bind 改变了 fooNewBind 里面的 this 指向
```

5. JS 常见的四种设计模式

5.1 工厂模式

简单的工厂模式可以理解为解决多个相似的问题。

```
function CreatePerson(name, age, sex) {  
    var obj = new Object();  
    obj.name = name;  
    obj.age = age;  
    obj.sex = sex;  
    obj.sayName = function() {  
        return this.name;  
    }  
    return obj;  
}  
  
var p1 = new CreatePerson("longen", '28', '男');  
var p2 = new CreatePerson("tughenhua", '27', '女');  
console.log(p1.name); // longen  
console.log(p1.age);   // 28  
console.log(p1.sex);   // 男  
console.log(p1.sayName()); // longen  
  
console.log(p2.name);   // tughenhua  
console.log(p2.age);    // 27  
console.log(p2.sex);    // 女  
console.log(p2.sayName()); // tughenhua
```

5.2 单例模式

只能被实例化(构造函数给实例添加属性与方法)一次

```
// 单体模式
var Singleton = function(name) {
    this.name = name;
};
Singleton.prototype.getName = function() {
    return this.name;
}
// 获取实例对象
var getInstance = (function() {
    var instance = null;
    return function(name) {
        if(!instance) { //相当于一个一次性阀门, 只能实例化一次
            instance = new Singleton(name);
        }
        return instance;
    }
})();
// 测试单体模式的实例, 所以 a==b
var a = getInstance("aa");
var b = getInstance("bb");
```

5.3 沙箱模式

将一些函数放到自执行函数里面,但要用闭包暴露接口,用变量接收暴露的接

口,再调用里面的值,否则无法使用里面的值。

```
let sandboxModel=(function() {
    function sayName() {};
    function sayAge() {};
    return {
        sayName:sayName,
        sayAge:sayAge
    }
})();
```

5.4 发布者订阅模式

就例如如我们关注了某一个公众号,然后他对应的有新的消息就会给你推送

//发布者与订阅模式

```
var shoeObj = {}; // 定义发布者
shoeObj.list = []; // 缓存列表 存放订阅者回调函数

// 增加订阅者
shoeObj.listen = function(fn) {
    shoeObj.list.push(fn); // 订阅消息添加到缓存列表
}

// 发布消息
shoeObj.trigger = function() {
    for (var i = 0, fn; fn = this.list[i++];) {
        fn.apply(this, arguments); // 第一个参数只是改变 fn 的
    }
}

// 小红订阅如下消息
shoeObj.listen(function(color, size) {
    console.log("颜色是: " + color);
    console.log("尺码是: " + size);
});

// 小花订阅如下消息
shoeObj.listen(function(color, size) {
    console.log("再次打印颜色是: " + color);
    console.log("再次打印尺码是: " + size);
});

shoeObj.trigger("红色", 40);
shoeObj.trigger("黑色", 42);
```

代码实现逻辑是用数组存贮订阅者,发布者回调函数里面通知的方式是遍历

订阅者数组,并将发布者内容传入订阅者数组。

6.原型链

6.1 定义

对象继承属性的一个链条

6.2 构造函数, 实例与原型对象的关系

```
var Person = function (name) { this.name = name; } //person 是构造函数
var o3personTwo = new Person('personTwo') //personTwo 是实例
```

原型对象都有一个默认的 constructor 属性指向构造函数

6.3 创建实例的方法

1.字面量

```
let obj={ 'name': '张三' }
```

2.Object 构造函数创建

```
let Obj=new Object()
Obj.name='张三'
```

3.使用工厂模式创建对象

```
function createPerson(name) {
  var o = new Object();
  o.name = name;
};
return o;
}
var person1 = createPerson('张三');
```

4.使用构造函数创建对象

```
function Person(name) {  
    this.name = name;  
}  
  
var person1 = new Person('张三');
```

6.4 new 运算符

1.创了一个新对象;

2.this 指向构造函数;

3.构造函数有返回,会替换 new 出来的对象,如果没有就是 new 出来的对象

4.手动封装一个 new 运算符

```
var new2 = function (func) {  
    var o = Object.create(func.prototype); //创建对象  
    var k = func.call(o);                  //改变 this 指向，把结果付给  
k  
    if (typeof k === 'object') {           //判断 k 的类型是不是对象  
        return k;                          //是，返回 k  
    } else {  
        return o;                          //不是返回返回构造函数  
的  
        执行结果  
    }  
}
```

6.5 对象的原型链

7.继承的方式

JS 是一门弱类型动态语言,封装和继承是他的两大特性

7.1 原型链继承

将父类的实例作为子类的原型

1.代码实现

定义父类:

```
// 定义一个动物类
function Animal (name) {
  // 属性
  this.name = name || 'Animal';
  // 实例方法
  this.sleep = function() {
    console.log(this.name + '正在睡觉!');
  }
}
// 原型方法
Animal.prototype.eat = function(food) {
  console.log(this.name + '正在吃:' + food);
};
```

子类:

```
function Cat() {
}
Cat.prototype = new Animal();
Cat.prototype.name = 'cat';

// Test Code
var cat = new Cat();
console.log(cat.name); // cat
console.log(cat.eat('fish')); // cat 正在吃: fish   undefined
console.log(cat.sleep()); // cat 正在睡觉!   undefined
```

```
console.log(cat instanceof Animal); //true
console.log(cat instanceof Cat); //true
```

2.优缺点

简单易于实现,但是要想为子类新增属性和方法,必须要在 new Animal()这样的语句之后执行,无法实现多继承

7.2 构造继承

实质是利用 call 来改变 Cat 中的 this 指向

1.代码实现

子类:

```
function Cat(name) {
  Animal.call(this);
  this.name = name || 'Tom';
}
```

2.优缺点

可以实现多继承,不能继承原型属性/方法

7.3 实例继承

为父类实例添加新特性,作为子类实例返回

1.代码实现

子类

```
function Cat(name) {
  var instance = new Animal();
  instance.name = name || 'Tom';
  return instance;
}
```

2.优缺点

不限制调用方式,但不能实现多继承

7.4 拷贝继承

将父类的属性和方法拷贝一份到子类中

1.子类:

```
function Cat(name) {  
    var animal = new Animal();  
    for(var p in animal) {  
        Cat.prototype[p] = animal[p];  
    }  
    Cat.prototype.name = name || 'Tom';  
}
```

2.优缺点

支持多继承,但是效率低占用内存

7.5 组合继承

通过调用父类构造,继承父类的属性并保留传参的优点,然后通过将父类实例作为子类原型,实现函数复用

1.子类:

```
function Cat(name) {  
    Animal.call(this);  
    this.name = name || 'Tom';  
}  
  
Cat.prototype = new Animal();  
Cat.prototype.constructor = Cat;
```

7.6 寄生组合继承

```
function Cat(name) {
```

```

    Animal.call(this);
    this.name = name || 'Tom';
}
(function() {
    // 创建一个没有实例方法的类
    var Super = function() {};
    Super.prototype = Animal.prototype;
    //将实例作为子类的原型
    Cat.prototype = new Super();
})();

```

7.7 ES6 的 extends 继承

ES6 的继承机制是先创造父类的实例对象 `this`（所以必须先调用 `super` 方法），然后再用子类的构造函数修改 `this`，

```

class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y); // 调用父类的 constructor(x, y)
        this.color = color;
    }

    toString() {
        return this.color + ' ' + super.toString(); // 调用父类的 toString()
    }
}

```

作者：火狼

原文链接：<https://segmentfault.com/a/1190000014405410>

8 JS 继承的实现方式

既然要实现继承，那么首先我们得有一个父类，代码如下：

```
// 定义一个动物类
```

```
function Animal (name) {

    // 属性

    this.name = name || 'Animal';

    // 实例方法

    this.sleep = function(){

        console.log(this.name + '正在睡觉! ');

    }

}

// 原型方法

Animal.prototype.eat = function(food) {

    console.log(this.name + '正在吃: ' + food);

};
```

1、原型链继承

核心： 将父类的实例作为子类的原型

```
function Cat(){

}

Cat.prototype = new Animal();

Cat.prototype.name = 'cat';

// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.eat('fish'));

console.log(cat.sleep());

console.log(cat instanceof Animal); //true
```

```
console.log(cat instanceof Cat); //true
```

特点:

1. 非常纯粹的继承关系，实例是子类的实例，也是父类的实例
2. 父类新增原型方法/原型属性，子类都能访问到
3. 简单，易于实现

缺点:

1. 要想为子类新增属性和方法，必须要在 `new Animal()` 这样的语句之后执行，不能放到构造器中
2. 无法实现多继承
3. 来自原型对象的引用属性是所有实例共享的（详细请看附录代码：[示例 1](#)）
4. 创建子类实例时，无法向父类构造函数传参

推荐指数：★★（3、4 两大致命缺陷）

2017-8-17 10:21:43 补充：感谢 MMHS 指出。缺点 1 中描述有误：可以在 Cat 构造函数中，为 Cat 实例增加实例属性。如果要新增原型属性和方法，则必须放在 `new Animal()` 这样的语句之后执行。

2、构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

```
function Cat(name){  
    Animal.call(this);  
    this.name = name || 'Tom';  
}  
  
// Test Code  
  
var cat = new Cat();  
  
console.log(cat.name);  
  
console.log(cat.sleep());  
  
console.log(cat instanceof Animal); // false  
  
console.log(cat instanceof Cat); // true
```


特点:

1. 解决了 1 中，子类实例共享父类引用属性的问题
2. 创建子类实例时，可以向父类传递参数
3. 可以实现多继承（call 多个父类对象）

缺点:

1. 实例并不是父类的实例，只是子类的实例
2. 只能继承父类的实例属性和方法，不能继承原型属性/方法
3. 无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

推荐指数：★★（缺点 3）

3、实例继承

核心：为父类实例添加新特性，作为子类实例返回

```
function Cat(name){  
  
    var instance = new Animal();  
  
    instance.name = name || 'Tom';  
  
    return instance;  
  
}  
  
// Test Code  
  
var cat = new Cat();  
  
console.log(cat.name);  
  
console.log(cat.sleep());  
  
console.log(cat instanceof Animal); // true  
  
console.log(cat instanceof Cat); // false
```

特点:

1. 不限制调用方式，不管是 `new 子类()` 还是 `子类()`, 返回的对象具有相同的效果

缺点:

1. 实例是父类的实例，不是子类的实例

2. 不支持多继承

推荐指数：★★

4、拷贝继承

```
function Cat(name){

    var animal = new Animal();

    for(var p in animal){

        Cat.prototype[p] = animal[p];

    }

    Cat.prototype.name = name || 'Tom';

}


// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // false

console.log(cat instanceof Cat); // true
```

特点：

1. 支持多继承

缺点：

1. 效率较低，内存占用高（因为要拷贝父类的属性）
2. 无法获取父类不可枚举的方法（不可枚举方法，不能使用 `for in` 访问到）

推荐指数：★（缺点 1）

5、组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后将父类实例作为子类原型，实现函数复用

```
function Cat(name){
```

```

    Animal.call(this);

    this.name = name || 'Tom';
}

Cat.prototype = new Animal();

// 感谢 @学无止境c 的提醒，组合继承也是需要修复构造函数指向的。

Cat.prototype.constructor = Cat;

// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); // true

```

特点：

1. 弥补了方式 2 的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法
2. 既是子类的实例，也是父类的实例
3. 不存在引用属性共享问题
4. 可传参
5. 函数可复用

缺点：

1. 调用了两次父类构造函数，生成了两份实例（子类实例将子类原型上的那份屏蔽了）

推荐指数：★★★★（仅仅多消耗了一点内存）

6、寄生组合继承

核心：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点

```

function Cat(name){

    Animal.call(this);

```

```

    this.name = name || 'Tom';
}

(function(){

    // 创建一个没有实例方法的类

    var Super = function({});

    Super.prototype = Animal.prototype;

    //将实例作为子类的原型

    Cat.prototype = new Super();

})();

// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); //true

感谢 @bluedrink 提醒，该实现没有修复 constructor。

Cat.prototype.constructor = Cat; // 需要修复下构造函数

```

特点：

1. 堪称完美

缺点：

1. 实现较为复杂

推荐指数：★★★★（实现复杂，扣掉一颗星）

附录代码：

示例一：

```
function Animal (name) {

    // 属性

    this.name = name || 'Animal';

    // 实例方法

    this.sleep = function(){

        console.log(this.name + '正在睡觉! ');

    }

    //实例引用属性

    this.features = [];

}

function Cat(name){

}

Cat.prototype = new Animal();

var tom = new Cat('Tom');

var kissy = new Cat('Kissy');


console.log(tom.name); // "Animal"

console.log(kissy.name); // "Animal"

console.log(tom.features); // []

console.log(kissy.features); // []

tom.name = 'Tom-New Name';

tom.features.push('eat');


//针对父类实例值类型成员的更改，不影响

console.log(tom.name); // "Tom-New Name"
```

```
console.log(kissy.name); // "Animal"
```

//针对父类实例引用类型成员的更改，会通过影响其他子类实例

```
console.log(tom.features); // ['eat']
```

```
console.log(kissy.features); // ['eat']
```

原因分析：

关键点：属性查找过程

执行 `tom.features.push`，首先找 `tom` 对象的实例属性（找不到），

那么去原型对象中找，也就是 `Animal` 的实例。发现有，那么就直接在这个对象的

`features` 属性中插入值。

在 `console.log(kissy.features)`；的时候。同上，`kissy` 实例上没有，那么去原型上找。

刚好原型上有，就直接返回，但是注意，这个原型对象中 `features` 属性值已经变化了。

从浏览器多进程到 JS 单线程

大纲

- 区分进程和线程

-

浏览器是多进程的

-

- 浏览器都包含哪些进程？
- 浏览器多进程的优势
- 重点是浏览器内核（渲染进程）
- Browser 进程和浏览器内核（Renderer 进程）的通信过程

-

梳理浏览器内核中线程之间的关系

-

- GUI 渲染线程与 JS 引擎线程互斥
- JS 阻塞页面加载
- WebWorker，JS 的多线程？
- WebWorker 与 SharedWorker

-

简单梳理下浏览器渲染流程

- - load 事件与 DOMContentLoaded 事件的先后
 - css 加载是否会阻塞 dom 树渲染?
 - 普通图层和复合图层

从 Event Loop 谈 JS 的运行机制

- - 事件循环机制进一步补充
 - 单独说说定时器
 - setTimeout 而不是 setInterval
- 事件循环进阶: macrotask 与 microtask
- 写在最后的话

区分进程和线程

线程和进程区分不清,是很多新手都会犯的错误,没有关系。这很正常。先看看下面这个形象的比喻:

- 进程是一个工厂,工厂有它的独立资源
- 工厂之间相互独立
- 线程是工厂中的工人,多个工人协作完成任务
- 工厂内有一个或多个工人
- 工人之间共享空间

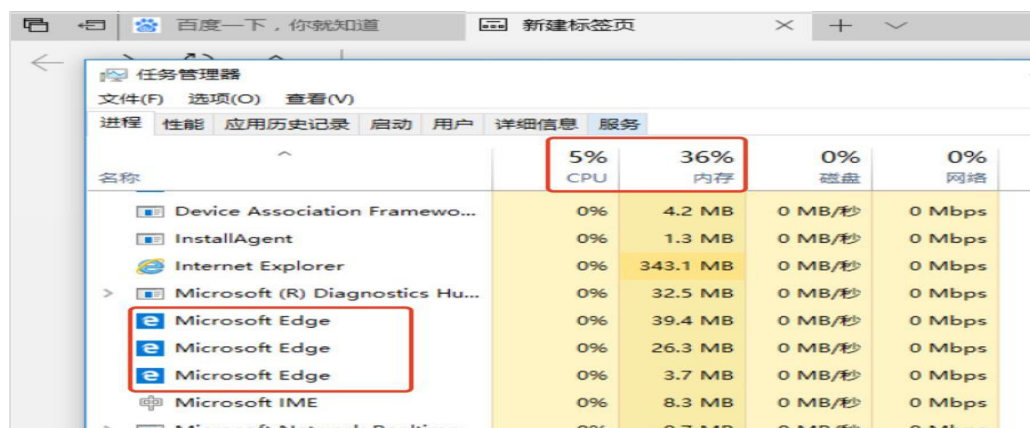
再完善完善概念:

- 工厂的资源 -> 系统分配的内存(独立的一块内存)
- 工厂之间的相互独立 -> 进程之间相互独立
- 多个工人协作完成任务 -> 多个线程在进程中协作完成任务
- 工厂内有一个或多个工人 -> 一个进程由一个或多个线程组成

- 工人之间共享空间 -> 同一进程下的各个线程之间共享程序的内存空间（包括代码段、数据集、堆等）

然后再巩固下：

如果是 windows 电脑中，可以打开任务管理器，可以看到有一个后台进程列表。对，那里就是查看进程的地方，而且可以看到每个进程的内存资源信息以及 cpu 占有率。



名称	5% CPU	36% 内存	0% 磁盘	0% 网络
Device Association Framework...	0%	4.2 MB	0 MB/秒	0 Mbps
InstallAgent	0%	1.3 MB	0 MB/秒	0 Mbps
Internet Explorer	0%	343.1 MB	0 MB/秒	0 Mbps
Microsoft (R) Diagnostics Hu...	0%	32.5 MB	0 MB/秒	0 Mbps
Microsoft Edge	0%	39.4 MB	0 MB/秒	0 Mbps
Microsoft Edge	0%	26.3 MB	0 MB/秒	0 Mbps
Microsoft Edge	0%	3.7 MB	0 MB/秒	0 Mbps
Microsoft IME	0%	8.3 MB	0 MB/秒	0 Mbps
Microsoft Network Realtime...	0%	0.7 MB	0 MB/秒	0 Mbps

所以，应该更容易理解了：**进程是 cpu 资源分配的最小单位**（系统会给它分配内存）

最后，再用较为官方的术语描述一遍：

- 进程是 cpu 资源分配的最小单位（是能拥有资源和独立运行的最小单位）
- 线程是 cpu 调度的最小单位（线程是建立在进程的基础上的一次程序运行单位，一个进程中可以有多个线程）

tips

- 不同进程之间也可以通信，不过代价较大
- 现在，一般通用的叫法：**单线程与多线程**，都是指在一个进程内的单和多。（所以核心还是得属于一个进程才行）

浏览器是多进程的

理解了进程与线程了区别后，接下来对浏览器进行一定程度上的认识：（先看下简化理解）

- 浏览器是多进程的
- 浏览器之所以能够运行，是因为系统给它的进程分配了资源（cpu、内存）
- 简单点理解，每打开一个 Tab 页，就相当于创建了一个独立的浏览器进程。

关于以上几点的验证，请再第一张图：



任务	内存	CPU	网络	进程 ID
标签页: 浏览器多线程和js单	244 MB	0.1	0	1164
标签页: [redacted]	278 MB	0.1	0	1124
标签页: 浏览器UI多线程及	229 MB	0.1	0	1184
标签页: 理解WebKit和Chrc	256 MB	0.1	0	1103
GPU 进程	770 MB	0.1	0	391
标签页: 百度翻译	234 MB	0.2	0	1090
标签页: [redacted]	322 MB	0.4	0	1182
标签页: DMQ/mvvm: 剖析v	290 MB	1.0	0	1024
标签页: [redacted]	277 MB	2.1	0	1122
浏览器	762 MB	31.9	0	302

结束进程

图中打开了 **Chrome** 浏览器的多个标签页，然后可以在 **Chrome 的任务管理器**中看到有多个进程（分别是每一个 **Tab** 页面有一个独立的进程，以及一个主进程）。
感兴趣的可以自行尝试下，如果再多打开一个 **Tab** 页，进程正常会+1 以上

注意：在这里浏览器应该也有自己的优化机制，有时候打开多个 **tab** 页后，可以在 **Chrome** 任务管理器中看到，有些进程被合并了
（所以每一个 **Tab** 标签对应一个进程并不一定是绝对的）

浏览器都包含哪些进程？

知道了浏览器是多进程后，再来看看它到底包含哪些进程：（为了简化管理，仅列举主要进程）

1. **Browser 进程：**浏览器的进程（负责协调、主控），只有一个。作用有
2.
 - 负责浏览器界面显示，与用户交互。如前进，后退等
 - 负责各个页面的管理，创建和销毁其他进程
 - 将 **Renderer** 进程得到的内存中的 **Bitmap**，绘制到用户界面上
 - 网络资源的管理，下载等
3. **第三方插件进程：**每种类型的插件对应一个进程，仅当使用该插件时才创建
4. **GPU 进程：**最多一个，用于 3D 绘制等
5. **浏览器渲染进程（浏览器内核）（Renderer 进程，内部是多线程的）：**默认每个 **Tab** 页面一个进程，互不影响。主要作用为
- 6.

- 页面渲染，脚本执行，事件处理等

强化记忆：在浏览器中打开一个网页相当于新起了一个进程（进程内有自己的多线程）

当然，浏览器有时会将多个进程合并（譬如打开多个空白标签页后，会发现多个空白标签页被合并成了一个进程），如图



任务	内存	CPU	网络	进程 ID
标签页: [redacted]	190 MB	0.1	0	19705
标签页: medium.com	162 MB	0.0	0	19850
标签页: 新标签页	285 MB	0.0	0	25966
标签页: 新标签页				
标签页: 新标签页				
标签页: 新标签页				
标签页: 新标签页				
扩展程序: Octotree	159 MB	0.0	0	25967
MIME 处理程序: [redacted]	159 MB	0.0	0	17267
插件: Chrome PDF Plugin	145 MB	0.0	0	17147

另外，可以通过 Chrome 的[更多工具](#) -> [任务管理器](#)自行验证

浏览器多进程的优势

相比于单进程浏览器，多进程有如下优点：

- 避免单个 page crash 影响整个浏览器
- 避免第三方插件 crash 影响整个浏览器
- 多进程充分利用多核优势
- 方便使用沙盒模型隔离插件等进程，提高浏览器稳定性

简单点理解：如果浏览器是单进程，那么某个 Tab 页崩溃了，就影响了整个浏览器，体验有多差；同理如果是单进程，插件崩溃了也会影响整个浏览器；而且多进程还有其它的诸多优势。。。

当然，内存等资源消耗也会更大，有点空间换时间的意思。

重点是浏览器内核（渲染进程）

重点来了，我们可以看到，上面提到了这么多的进程，那么，对于普通的前端操作来说，最终要的是什么呢？答案是**渲染进程**

可以这样理解，页面的渲染，JS 的执行，事件的循环，都在这个进程内进行。接下来重点分析这个进程

请牢记，浏览器的渲染进程是多线程的（这点如果不理解，请回头看进程和线程的区分）

终于到了线程这个概念了？，好亲切。那么接下来看看它都包含了哪些线程（列举一些主要常驻线程）：

1.

GUI 渲染线程

2.

- 负责渲染浏览器界面，解析 HTML，CSS，构建 DOM 树和 RenderObject 树，布局和绘制等。
- 当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时，该线程就会执行
- 注意，**GUI 渲染线程与 JS 引擎线程是互斥的**，当 JS 引擎执行时 GUI 线程会被挂起（相当于被冻结了），GUI 更新会被保存在一个队列中**等到 JS 引擎空闲时立即被执行**。

3.

JS 引擎线程

4.

- 也称为 JS 内核，负责处理 Javascript 脚本程序。（例如 V8 引擎）
- JS 引擎线程负责解析 Javascript 脚本，运行代码。
- JS 引擎一直等待着任务队列中任务的到来，然后加以处理，一个 Tab 页（renderer 进程）中无论什么时候都只有一个 JS 线程在运行 JS 程序
- 同样注意，**GUI 渲染线程与 JS 引擎线程是互斥的**，所以如果 JS 执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞。

5.

事件触发线程

6.

- 归属于浏览器而不是 JS 引擎，用来控制事件循环（可以理解，JS 引擎自己都忙不过来，需要浏览器另开线程协助）
- 当 JS 引擎执行代码块如 `setTimeout` 时（也可来自浏览器内核的其他线程，如鼠标点击、AJAX 异步请求等），会将对应任务添加到事件线程中
- 当对应的事件符合触发条件被触发时，该线程会把事件添加到待处理队列的队尾，等待 JS 引擎的处理

○

注意，由于 JS 的单线程关系，所以这些待处理队列中的事件都得排队等待 JS 引擎处理（当 JS 引擎空闲时才会去执行）

○

7.

定时触发器线程

8.

- 传说中的 `setInterval` 与 `setTimeout` 所在线程
- 浏览器定时计数器并不是由 JavaScript 引擎计数的, (因为 JavaScript 引擎是单线程的, 如果处于阻塞线程状态就会影响记计时的准确)
- 因此通过单独线程来计时并触发定时 (计时完毕后, 添加到事件队列中, 等待 JS 引擎空闲后执行)
- 注意, W3C 在 HTML 标准中规定, 规定要求 `setTimeout` 中低于 4ms 的时间间隔算为 4ms。

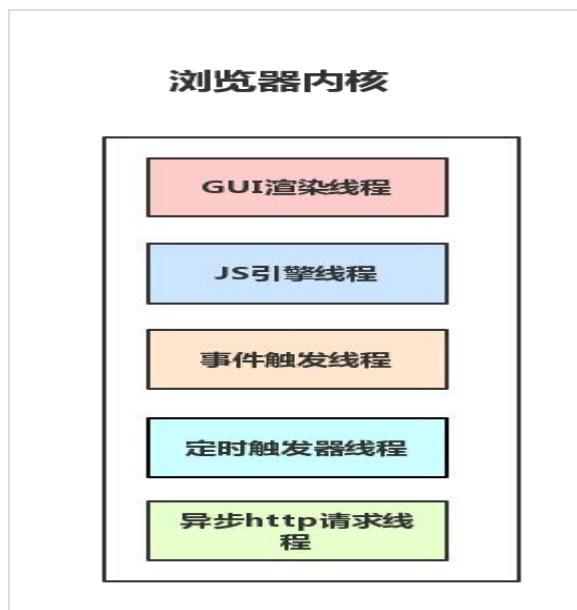
9.

异步 http 请求线程

10.

- 在 XMLHttpRequest 在连接后是通过浏览器新开一个线程请求
- 将检测到状态变更时, 如果设置有回调函数, 异步线程就产生状态变更事件, 将这个回调再放入事件队列中。再由 JavaScript 引擎执行。

看到这里, 如果觉得累了, 可以先休息下, 这些概念需要被消化, 毕竟后续将提到的事件循环机制就是基于事件触发线程的, 所以如果仅仅是看某个碎片化知识, 可能会有一种似懂非懂的感觉。要完成的梳理一遍才能快速沉淀, 不易遗忘。放张图巩固下吧:



再说一点, 为什么 JS 引擎是单线程的? 额, 这个问题其实应该没有标准答案, 譬如, 可能仅仅是因为由于多线程的复杂性, 譬如多线程操作一般要加锁, 因此最初设计时选择了单线程。。。

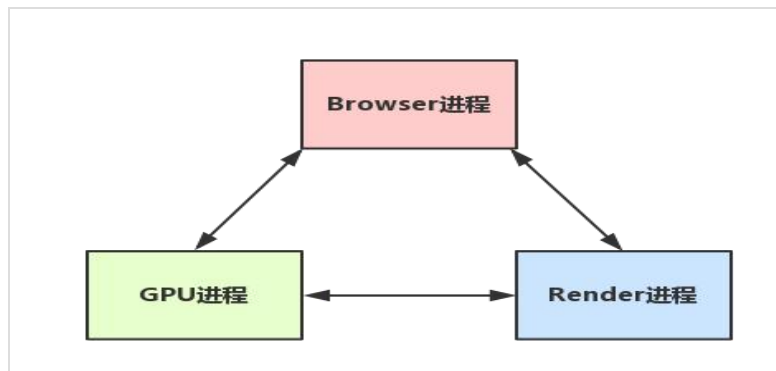
Browser 进程和浏览器内核 (Renderer 进程) 的通信过程

看到这里，首先，应该对浏览器内的进程和线程都有一定理解了，那么接下来，再谈谈浏览器的 **Browser** 进程（控制进程）是如何和内核通信的，这点也理解后，就可以将这部分的知识串联起来，从头到尾有一个完整的概念。

如果自己打开任务管理器，然后打开一个浏览器，就可以看到：**任务管理器中出现了两个进程**（一个是**主控进程**，一个则是**打开 Tab 页的渲染进程**），然后在这前提下，看下整个的过程：(简化了很多)

- **Browser** 进程收到用户请求，首先需要获取页面内容（譬如通过网络下载资源），随后将该任务通过 **RendererHost** 接口传递给 **Render** 进程
- **Render** 进程的 **Renderer** 接口收到消息，简单解释后，交给渲染线程，然后开始渲染
- - 渲染线程接收请求，加载网页并渲染网页，这其中可能需要 **Browser** 进程获取资源和需要 **GPU** 进程来帮助渲染
 - 当然可能会有 **JS** 线程操作 **DOM**（这样可能会造成回流并重绘）
 - 最后 **Render** 进程将结果传递给 **Browser** 进程
- **Browser** 进程接收到结果并将结果绘制出来

这里绘一张简单的图：（很简化）



看完这一整套流程，应该对浏览器的运作有了一定理解了，这样有了知识架构的基础后，后续就方便往上填充内容。

这块再往深处讲的话就涉及到浏览器内核源码解析了，不属于本文范围。

如果这一块要深挖，建议去读一些浏览器内核源码解析文章，或者可以先看看参考下来源中的第一篇文章，写的不错

梳理浏览器内核中线程之间的关系

到了这里，已经对浏览器的运行有了一个整体的概念，接下来，先简单梳理一些概念

GUI 渲染线程与 JS 引擎线程互斥

由于 JavaScript 是可操纵 DOM 的，如果在修改这些元素属性同时渲染界面（即 JS 线程和 UI 线程同时运行），那么渲染线程前后获得的元素数据就可能不一致了。

因此为了防止渲染出现不可预期的结果，浏览器设置 GUI 渲染线程与 JS 引擎为互斥的关系，当 JS 引擎执行时 GUI 线程会被挂起，GUI 更新则会被保存在一个队列中等到 JS 引擎线程空闲时立即被执行。

JS 阻塞页面加载

从上述的互斥关系，可以推导出，JS 如果执行时间过长就会阻塞页面。

譬如，假设 JS 引擎正在进行巨量的计算，此时就算 GUI 有更新，也会被保存到队列中，等待 JS 引擎空闲后执行。

然后，由于巨量计算，所以 JS 引擎很可能很久很久后才能空闲，自然会感觉到巨卡无比。

所以，要尽量避免 JS 执行时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

WebWorker，JS 的多线程？

前文中有提到 JS 引擎是单线程的，而且 JS 执行时间过长会阻塞页面，那么 JS 就真的对 cpu 密集型计算无能为力么？

所以，后来 HTML5 中支持了 **Web Worker**。

MDN 的官方解释是：

Web Worker 为 Web 内容在后台线程中运行脚本提供了一种简单的方法。线程可以执行任务而不干扰用户界面

一个 worker 是使用一个构造函数创建的一个对象(e.g. `Worker()`) 运行一个命名的 JavaScript 文件

这个文件包含将在工作线程中运行的代码；`workers` 运行在另一个全局上下文中，不同于当前的 `window`

因此，使用 `window` 快捷方式获取当前全局的范围（而不是 `self`）在一个 `Worker` 内将返回错误

这样理解下：

- 创建 `Worker` 时，JS 引擎向浏览器申请开一个子线程（子线程是浏览器开的，完全受主线程控制，而且不能操作 `DOM`）
- JS 引擎线程与 `worker` 线程间通过特定的方式通信（`postMessage API`，需要通过序列化对象来与线程交互特定的数据）

所以，如果有非常耗时的工作，请单独开一个 `Worker` 线程，这样里面不管如何翻天覆地都不会影响 JS 引擎主线程，只待计算出结果后，将结果通信给主线程即可，**perfect!**

而且注意下，**JS 引擎是单线程的**，这一点的本质仍然未改变，`Worker` 可以理解是浏览器给 JS 引擎开的外挂，专门用来解决那些大量计算问题。

其它，关于 `Worker` 的详解就不是本文的范畴了，因此不再赘述。

WebWorker 与 SharedWorker

既然都到了这里，就再提一下 `SharedWorker`（避免后续将这两个概念搞混）

- `WebWorker` 只属于某个页面，不会和其他页面的 `Render` 进程（浏览器内核进程）共享
- - 所以 `Chrome` 在 `Render` 进程中（每一个 `Tab` 页就是一个 `render` 进程）创建一个新的线程来运行 `Worker` 中的 `JavaScript` 程序。
- `SharedWorker` 是浏览器所有页面共享的，不能采用与 `Worker` 同样的方式实现，因为它不隶属于某个 `Render` 进程，可以为多个 `Render` 进程共享使用
- - 所以 `Chrome` 浏览器为 `SharedWorker` 单独创建一个进程来运行 `JavaScript` 程序，在浏览器中每个相同的 `JavaScript` 只存在一个 `SharedWorker` 进程，不管它被创建多少次。

看到这里，应该就很容易明白了，本质上就是进程和线程的区别。**SharedWorker** 由独立的进程管理，**WebWorker** 只是属于 **render** 进程下的一个线程

简单梳理下浏览器渲染流程

本来是直接计划开始谈 **JS** 运行机制的，但想了想，既然上述都一直在谈浏览器，直接跳到 **JS** 可能再突兀，因此，中间再补充下浏览器的渲染流程（简单版本）

为了简化管理，前期工作直接省略成：（要展开的或完全可以写另一篇超长文）

- 浏览器输入 **url**，浏览器主进程接管，开一个下载线程，

然后进行 **http** 请求（略去 **DNS** 查询，**IP** 寻址等等操作），然后等待响应，获取内容，

随后将内容通过 **RendererHost** 接口转交给 **Renderer** 进程

- 浏览器渲染流程开始

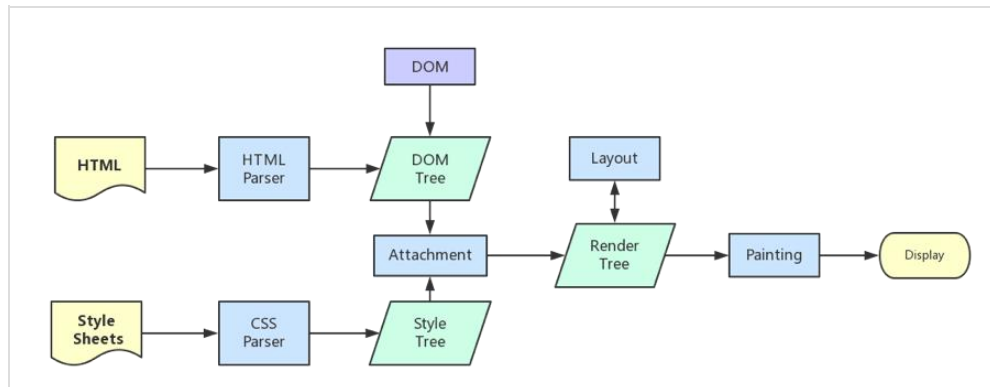
浏览器内核拿到内容后，渲染大概可以划分成以下几个步骤：

1. 解析 **html** 建立 **dom** 树
2. 解析 **css** 构建 **render** 树（将 **CSS** 代码解析成树形的数据结构，然后结合 **DOM** 合并成 **render** 树）
3. 布局 **render** 树（**Layout/reflow**），负责各元素尺寸、位置的计算
4. 绘制 **render** 树（**paint**），绘制页面像素信息
5. 浏览器会将各层的信息发送给 **GPU**，**GPU** 会将各层合成（**composite**），显示在屏幕上。

所有详细步骤都已经略去，渲染完毕后就是 **load** 事件了，之后就是自己的 **JS** 逻辑处理了

既然略去了一些详细的步骤，那么就提一些可能需要注意的细节把。

这里重绘参考来源中的一张图：（参考来源第一篇）



load 事件与 DOMContentLoaded 事件的先后

上面提到，渲染完毕后会触发 `load` 事件，那么你能分清楚 `load` 事件与 `DOMContentLoaded` 事件的先后么？

很简单，知道它们的定义就可以了：

- 当 `DOMContentLoaded` 事件触发时，仅当 DOM 加载完成，不包括样式表，图片。

(譬如如果有 `async` 加载的脚本就不一定完成)

- 当 `onload` 事件触发时，页面上所有的 DOM，样式表，脚本，图片都已经加载完成了。

(渲染完毕了)

所以，顺序是：`DOMContentLoaded -> load`

css 加载是否会阻塞 dom 树渲染？

这里说的是头部引入 `css` 的情况

首先，我们都知道：`css` 是由单独的下载线程异步下载的。

然后再说下几个现象：

- `css` 加载不会阻塞 DOM 树解析（异步加载时 DOM 照常构建）
- 但会阻塞 render 树渲染（渲染时需等 `css` 加载完毕，因为 render 树需要 `css` 信息）

这可能也是浏览器的一种优化机制。

因为你加载 **css** 的时候，可能会修改下面 **DOM** 节点的样式，如果 **css** 加载不阻塞 **render** 树渲染的话，那么当 **css** 加载完之后，**render** 树可能又得重新重绘或者回流了，这就造成了一些没有必要的损耗。所以干脆就先把 **DOM** 树的结构先解析完，把可以做完的工作做完，然后等你 **css** 加载完之后，在根据最终的样式来渲染 **render** 树，这种做法性能方面确实会比较好一点。

普通图层和复合图层

渲染步骤中就提到了 **composite** 概念。

可以简单的这样理解，浏览器渲染的图层一般包含两大类：**普通图层**以及**复合图层**

首先，普通文档流内可以理解为一个复合图层（这里称为**默认复合层**，里面不管添加多少元素，其实都是在同一个复合图层中）

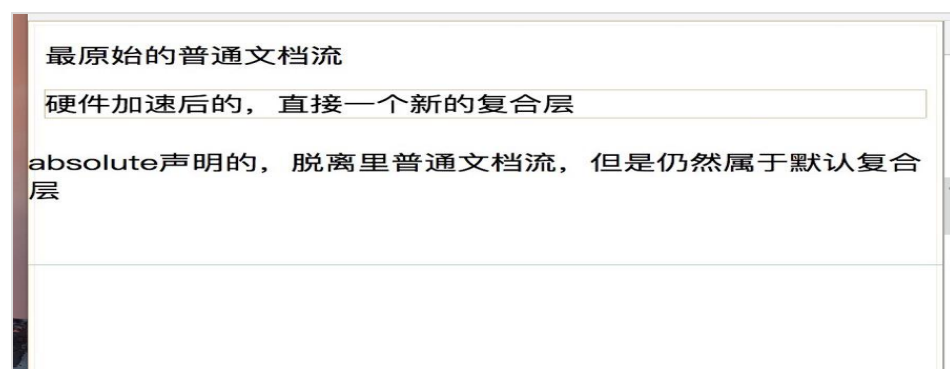
其次，**absolute** 布局（**fixed** 也一样），虽然可以脱离普通文档流，但它仍然属于**默认复合层**。

然后，可以通过**硬件加速**的方式，声明一个**新的复合图层**，它会单独分配资源（当然也会脱离普通文档流，这样一来，不管这个复合图层中怎么变化，也不会影响**默认复合层**里的回流重绘）

可以简单理解下：**GPU** 中，各个复合图层是单独绘制的，所以互不影响，这也是为什么某些场景硬件加速效果一级棒

可以 **Chrome 源码调试 -> More Tools -> Rendering -> Layer borders** 中看到，黄色的就是复合图层信息

如下图。可以验证上述的说法



如何变成复合图层（硬件加速）

将该元素变成一个复合图层，就是传说中的硬件加速技术

- 最常用的方式：`translate3d`、`translateZ`
- `opacity` 属性/过渡动画（需要动画执行的过程中才会创建合成层，动画没有开始或结束后元素还会回到之前的状态）
- `will-change` 属性（这个比较偏僻），一般配合 `opacity` 与 `translate` 使用（而且经测试，除了上述可以引发硬件加速的属性外，其它属性并不会变成复合层），

作用是提前告诉浏览器要变化，这样浏览器会开始做一些优化工作（这个最好用完后就释放）

- `<video><iframe><canvas><webgl>` 等元素
- 其它，譬如以前的 flash 插件

absolute 和硬件加速的区别

可以看到，`absolute` 虽然可以脱离普通文档流，但是无法脱离默认复合层。

所以，就算 `absolute` 中信息改变时不会改变普通文档流中 `render` 树，但是，浏览器最终绘制时，是整个复合层绘制的，所以 `absolute` 中信息的改变，仍然会影响整个复合层的绘制。

（浏览器会重绘它，如果复合层中内容多，`absolute` 带来的绘制信息变化过大，资源消耗是非常严重的）

而硬件加速直接就是在另一个复合层了（另起炉灶），所以它的信息改变不会影响默认复合层

（当然了，内部肯定会影响属于自己的复合层），仅仅是引发最后的合成（输出视图）

复合图层的作用？

一般一个元素开启硬件加速后会变成复合图层，可以独立于普通文档流中，改动后可以避免整个页面重绘，提升性能

但是尽量不要大量使用复合图层，否则由于资源消耗过度，页面反而会变的更卡

硬件加速时请使用 `index`

使用硬件加速时，尽可能的使用 `index`，防止浏览器默认给后续的元素创建复合层渲染

具体的原理是这样的：

****webkit CSS3 中，如果这个元素添加了硬件加速，并且 `index` 层级比较低，那么在这个元素的后面其它元素（层级比这个元素高的，或者相同的，并且 `relative` 或 `absolute` 属性相同的），会默认变为复合层渲染，如果处理不当会极大的影响性能****

简单点理解，其实可以认为是一个隐式合成的概念：如果 **a** 是一个复合图层，而且 **b** 在 **a** 上面，那么 **b** 也会被隐式转为一个复合图层，这点需要特别注意

另外，这个问题可以在这个地址看到重现（原作者分析的挺到位的，直接上链接）：

<http://web.jobbole.com/83575/>

从 Event Loop 谈 JS 的运行机制

到此时，已经是属于浏览器页面初次渲染完毕后的事情，JS 引擎的一些运行机制分析。

注意，这里不谈可执行上下文，VO，scop chain 等概念（这些完全可以整理成另一篇文章了），这里主要是结合 Event Loop 来谈 JS 代码是如何执行的。

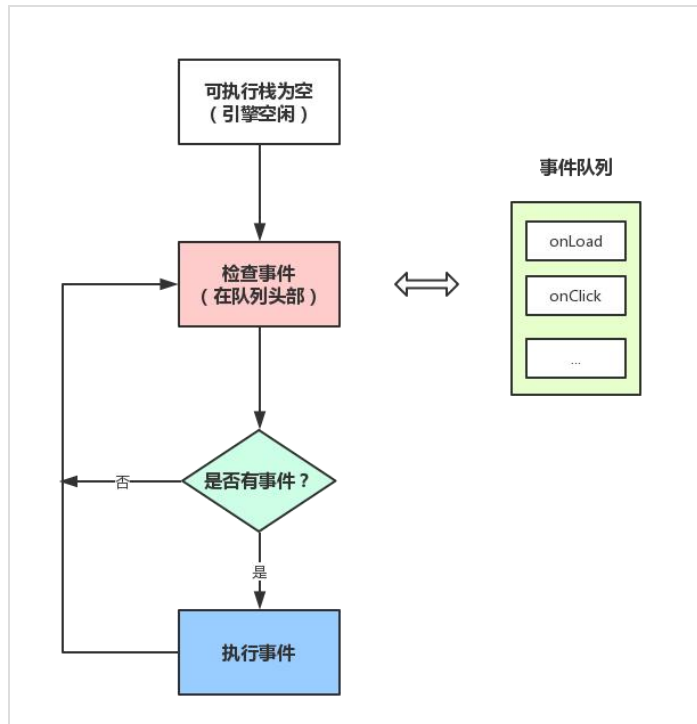
读这部分的前提是已经知道了 JS 引擎是单线程，而且这里会用到上文中的几个概念：（如果不是很理解，可以回头温习）

- JS 引擎线程
- 事件触发线程
- 定时触发器线程

然后再理解一个概念：

- JS 分为同步任务和异步任务
- 同步任务都在主线程上执行，形成一个执行栈
- 主线程之外，事件触发线程管理着一个任务队列，只要异步任务有了运行结果，就在任务队列之中放置一个事件。
- 一旦执行栈中的所有同步任务执行完毕（此时 JS 引擎空闲），系统就会读取任务队列，将可运行的异步任务添加到可执行栈中，开始执行。

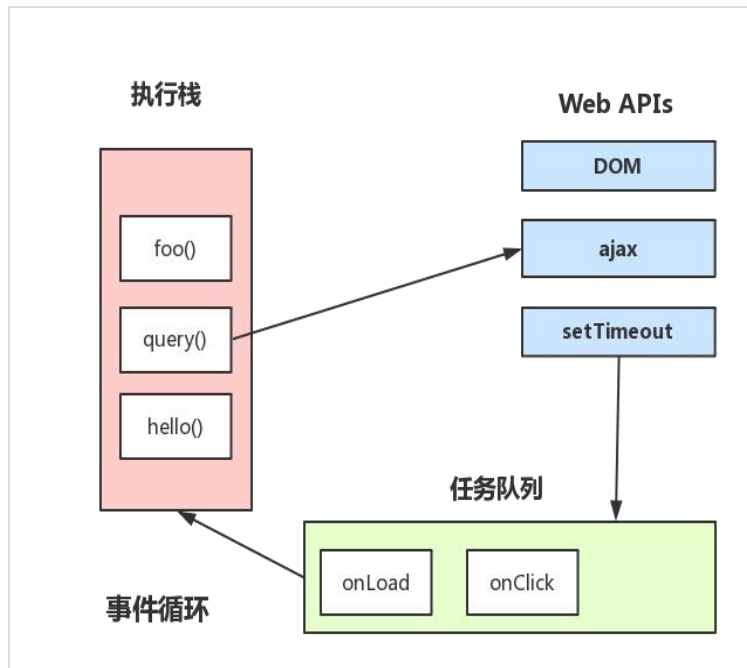
看图：



看到这里，应该就可以理解了：为什么有时候 `setTimeout` 推入的事件不能准时执行？因为可能在它推入到事件列表时，主线程还不空闲，正在执行其它代码，所以自然有误差。

事件循环机制进一步补充

这里就直接引用一张图片来协助理解：（参考自 Philip Roberts 的演讲《[Help, I'm stuck in an event-loop](#)》）



上图大致描述就是：

- 主线程运行时会产生执行栈，

栈中的代码调用某些 **api** 时，它们会在事件队列中添加各种事件（当满足触发条件后，如 **ajax** 请求完毕）

- 而栈中的代码执行完毕，就会读取事件队列中的事件，去执行那些回调
- 如此循环
- 注意，总是要等待栈中的代码执行完毕后才去读取事件队列中的事件

单独说说定时器

上述事件循环机制的核心是：**JS 引擎线程**和**事件触发线程**

但事件上，里面还有一些隐藏细节，譬如调用 **setTimeout** 后，是如何等待特定时间后才添加到事件队列中的？

是 **JS 引擎** 检测的么？当然不是了。它是由**定时器线程**控制（因为 **JS 引擎** 自己都忙不过来，根本无暇分身）

为什么要单独的定时器线程？因为 **JavaScript 引擎** 是单线程的，如果处于阻塞线程状态就会影响记计时的准确，因此很有必要单独开一个线程用来计时。

什么时候会用到定时器线程？当使用 `setTimeout` 或 `setInterval` 时，它需要定时器线程计时，计时完成后就会将特定的事件推入事件队列中。

譬如：

```
setTimeout(function(){  
  
    console.log('hello!');  
  
}, 1000);
```

这段代码的作用是当 1000 毫秒计时完毕后（由定时器线程计时），将回调函数推入事件队列中，等待主线程执行

```
setTimeout(function(){  
  
    console.log('hello!');  
  
}, 0);  
  
console.log('begin');
```

这段代码的效果是最快的时间内将回调函数推入事件队列中，等待主线程执行

注意：

- 执行结果是：先 `begin` 后 `hello!`
- 虽然代码的本意是 0 毫秒后就推入事件队列，但是 W3C 在 HTML 标准中规定，规定要求 `setTimeout` 中低于 4ms 的时间间隔算为 4ms。

(不过也有一说是不同浏览器有不同的最小时间设定)

- 就算不等待 4ms，就算假设 0 毫秒就推入事件队列，也会先执行 `begin`（因为只有可执行栈内空了后才会主动读取事件队列）

setTimeout 而不是 setInterval

用 `setTimeout` 模拟定期计时和直接用 `setInterval` 是有区别的。

因为每次 `setTimeout` 计时到后就会去执行，然后执行一段时间后会继续 `setTimeout`，中间就多了误差

（误差多少与代码执行时间有关）

而 `setInterval` 则是每次都精确的隔一段时间推入一个事件

（但是，事件的实际执行时间不一定就准确，还有可能是这个事件还没执行完毕，下一个事件就来了）

而且 `setInterval` 有一些比较致命的问题就是：

- 累计效应（上面提到的），如果 `setInterval` 代码在（`setInterval`）再次添加到队列之前还没有完成执行，

就会导致定时器代码连续运行好几次，而之间没有间隔。

就算正常间隔执行，多个 `setInterval` 的代码执行时间可能会比预期小（因为代码执行需要一定时间）

- （这一块后续有补充，`setInterval` 自带的优化，不会重复添加回调）
- 而且把浏览器最小化显示等操作时，`setInterval` 并不是不执行程序，

它会把 `setInterval` 的回调函数放在队列中，等浏览器窗口再次打开时，一瞬间全部执行时

所以，鉴于这么多问题，目前一般认为的最佳方案是：用 `setTimeout` 模拟 `setInterval`，或者特殊场合直接用 `requestAnimationFrame`

补充：JS 高程中有提到，JS 引擎会对 `setInterval` 进行优化，如果当前事件队列中有 `setInterval` 的回调，不会重复添加。不过，仍然是有很多问题。。。

事件循环进阶：macrotask 与 microtask

这段参考了参考来源中的第 2 篇文章（英文版的），（加了下自己的理解重新描述了下），强烈推荐有英文基础的同学直接观看原文，作者描述的很清晰，示例也很不错，如下：

<https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/>

上文中将 JS 事件循环机制梳理了一遍，在 ES5 的情况是够用了，但是在 ES6 盛行的现在，仍然会遇到一些问题，譬如下面这题：

```
console.log('script start');

setTimeout(function() {

    console.log('setTimeout');

}, 0);
```



```
Promise.resolve().then(function() {  
  
    console.log('promise1');  
  
}).then(function() {  
  
    console.log('promise2');  
  
});  
  
console.log('script end');
```

嗯哼，它的正确执行顺序是这样子的：

script start

script end

promise1

promise2

setTimeout

为什么呢？因为 Promise 里有了一个全新的概念：microtask

或者，进一步，JS 中分为两种任务类型：macrotask 和 microtask，在 ECMAScript 中，microtask 称为 jobs，macrotask 可称为 task

它们的定义？区别？简单点可以按如下理解：

macrotask（又称之为宏任务），可以理解是每次执行栈执行的代码就是一个宏任务（包括每次从事件队列中获取一个事件回调并放到执行栈中执行）

- - 每一个 task 会从头到尾将这个任务执行完毕，不会执行其它
 - 浏览器为了能够使得 JS 内部 task 与 DOM 任务能够有序的执行，会在一个 task 执行结束后，在下一个 task 执行开始前，对页面进行重新渲染

（`task->渲染->task->...`）

- microtask（又称为微任务），可以理解是在当前 task 执行结束后立即执行的任务

-

- 也就是说，在当前 **task** 任务后，下一个 **task** 之前，在渲染之前
- 所以它的响应速度相比 **setTimeout** (**setTimeout** 是 **task**) 会更快，因为无需等渲染
- 也就是说，在某一个 **macrotask** 执行完后，就会将在它执行期间产生的所有 **microtask** 都执行完毕（在渲染前）

分别很么样的场景会形成 **macrotask** 和 **microtask** 呢？

- **macrotask**: 主代码块，**setTimeout**，**setInterval** 等（可以看到，事件队列中的每一个事件都是一个 **macrotask**）
- **microtask**: **Promise**，**process.nextTick** 等

__补充：在 **node** 环境下，**process.nextTick** 的优先级高于 **Promise**__，也就是可以简单理解为：在宏任务结束后会先执行微任务队列中的 **nextTickQueue** 部分，然后才会执行微任务中的 **Promise** 部分。

参考：<https://segmentfault.com/q/1010000011914016>

再根据线程来理解下：

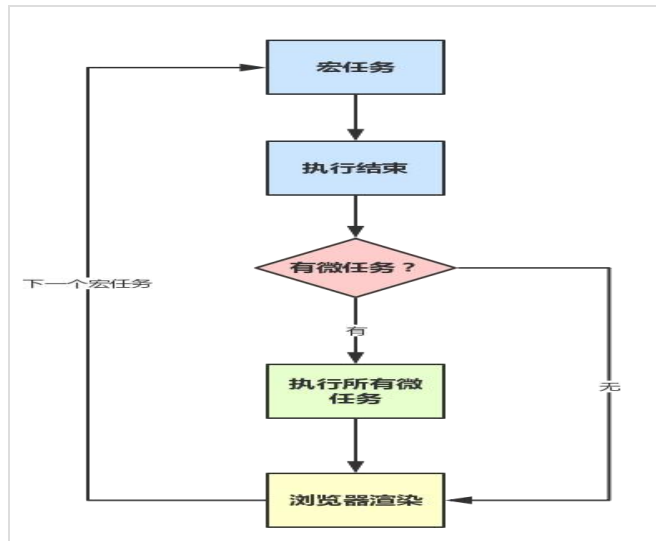
- **macrotask** 中的事件都是放在一个事件队列中的，而这个队列由**事件触发线程**维护
- **microtask** 中的所有微任务都是添加到微任务队列（**Job Queues**）中，等待当前 **macrotask** 执行完毕后执行，而这个队列由 **JS 引擎线程**维护

（这点由自己理解+推测得出，因为它是在主线程下无缝执行的）

所以，总结下运行机制：

- 执行一个宏任务（栈中没有就从事件队列中获取）
- 执行过程中如果遇到微任务，就将它添加到微任务的队列中
- 宏任务执行完毕后，立即执行当前微任务队列中的所有微任务（依次执行）
- 当前宏任务执行完毕，开始检查渲染，然后 **GUI** 线程接管渲染
- 渲染完毕后，**JS** 线程继续接管，开始下一个宏任务（从事件队列中获取）

如图：



另外，请注意下 **Promise** 的 **polyfill** 与官方版本的区别：

- 官方版本中，是标准的 **microtask** 形式
- **polyfill**，一般都是通过 **setTimeout** 模拟的，所以是 **macrotask** 形式
- 请特别注意这两点区别

注意，有一些浏览器执行结果不一样（因为它们可能把 **microtask** 当成 **macrotask** 来执行了），但是为了简单，这里不描述一些不标准的浏览器下的场景（但记住，有些浏览器可能并不标准）

20180126 补充：使用 **MutationObserver** 实现 **microtask**

MutationObserver 可以用来实现 **microtask**
（它属于 **microtask**，优先级小于 **Promise**，一般是 **Promise** 不支持时才会这样做）

它是 **HTML5** 中的新特性，作用是：监听一个 **DOM** 变动，当 **DOM** 对象树发生任何变动时，**Mutation Observer** 会得到通知

像以前的 **Vue** 源码中就是利用它来模拟 **nextTick** 的，具体原理是，创建一个 **TextNode** 并监听内容变化，然后要 **nextTick** 的时候去改一下这个节点的文本内容，如下：（**Vue** 的源码，未修改）

```
var counter = 1var observer = new MutationObserver(nextTickHandler)var textNode = document.createTextNode(String(counter))
```

```
observer.observe(textNode, {  
  
  characterData: true  
  
})  
  
timerFunc = () => {  
  
  counter = (counter + 1) % 2  
  
  textNode.data = String(counter)  
  
}
```

对应 Vue 源码链接

不过，现在的 Vue（2.5+）的 nextTick 实现移除了 MutationObserver 的方式（据说是兼容性原因），

取而代之的是使用 MessageChannel

（当然，默认情况仍然是 Promise，不支持才兼容的）。

MessageChannel 属于宏任务，优先级是：MessageChannel->setTimeout，
所以 Vue（2.5+）内部的 nextTick 与 2.4 及之前的实现是不一样的，需要注意下。

这里不展开，可以看下

<https://juejin.im/post/5a1af88f5188254a701ec230>

JS 异步精讲

问题点

-
- 什么是单线程，和异步有什么关系
 - 什么是 event-loop
 - jQuery 的 Deferred
 - Promise 的基本使用和原理
 - async/await（和 Promise 的区别、联系）

一、什么是单线程，和异步有什么关系

-
- 单线程- 只有一个线程，只能做一件事

- 原因-避免 DOM 渲染的冲突
- 解决方案-异步

1) 单线程- 只有一个线程，只能做一件事

基础事例

// 循环运行期间,JS 执行和 DOM 渲染暂时卡顿

```
var i, sum = 0;
```

```
for (i = 0; i<100000000000; i++) {
```

```
    sum += i;
```

```
}
```

```
console.log(sum);
```

// alert 不处理,JS 执行和 DOM 渲染暂时卡顿

```
console.log(1);
```

```
alert('hello');
```

```
console.log(2)
```

2) 原因 - 避免 DOM 渲染的冲突

1. 浏览器需要渲染 DOM
2. JS 可以修改 DOM 结构
3. 所以 JS 执行的时候，浏览器 DOM 渲染会暂停
4. 两段 JS 也不能同时执行(都修改 DOM 就冲突了)
5. webworker 支持多线程，但是不能修改访问 DOM

3) 解决方案 - 异步

基础事例

```
console.log(100)
```

```
setTimeout(function () {
```

```
    console.log(200); // 反正 1000ms 之后执行
```

```
}, 1000);           // 先不管他,先让其它 JS 代码运行
```

```
console.log(300);
```

```
console.log(400);
```

4) 异步 - 存在的问题

问题一：没有按照书写方式执行，可读性差

问题二：callback 中不容易模块化

二、什么 event-loop

文字解释

1. 事件轮询，JS 实现异步的具体解决方案
2. 同步代码，直接执行
3. 异步函数先放在异步队列中
4. 待同步函数执行完毕，轮询执行异步队列函数

例子分析

例子一：

```
1 // 主进程
2
3 console.log(3)

setTimeout(function () {
    console.log(1)
}, 100)
setTimeout(function () {
    console.log(2)
})
console.log(3)

1 // 异步队列
2
3 // 立刻被放入
4 function () {
5     console.log
6 }
7
8 // 100ms 之后被放
9 function () {
10     console.log
11 }
```

先看左下角，是我们要执行的代码，第一个是延迟 100 毫秒打印 1，第二个是延迟 0，打印 2，按照上面讲的，同步代码先执行，异步代码放在异步队列中，最后执行完同步在异步队列的函数，执行第一个函数的时候，`setTimeout` 是个异步函数，那我们是不是应该立即放在右侧的队列中呢？答案是否定的，因为它有一个延时，我们需要 100 毫秒之后才能放进去，执行第二个 `setTimeout` 函数，由于没有延时，所以会被放进异步队列中，最后一个直接在主进程，所以直接执行打印，等同步执行完之后，立刻看异步队列中，这时候只有第二个 `setTimeout` 执行完之后，在去看异步队列有没有，因为 100 这毫秒对于计算机来说，是一个相当长的时间，所以 js 引擎会一直轮询异步队列中有没有可执行函数，直接 100 毫秒之后第一个 `setTimeout` 被放进异步队列中，然后才执行。

例子一：

```
$.ajax({
  url: 'xxxxx',
  success: function (result) {
    console.log('a')
  }
})
setTimeout(function () {
  console.log('b')
}, 100)
setTimeout(function () {
  console.log('c')
})
console.log('d')
```

上图流程跟例子一一样，这边我们疑惑的是，当我们执行 `ajax` 的时候，里面的 `success` 是异步函数，它要什么时候被放进异步队列中呢？

当然是请求成功的时候被放进异步队列中，但我们不知道是 大于 100 毫秒还是小于，所以打印结果有两种情况，先打印 `d c b a`，或者打印 `d c a b`。

三、是否用过 jQuery 的 Deferred

jQuery 1.5 的变化 -1.5 之前


```

var ajax = $.ajax({
  url: 'data.json',
  success: function () {
    console.log('success1')
    console.log('success2')
    console.log('success3')
  },
  error: function () {
    console.log('error')
  }
})
console.log(ajax) // 返回一个 XHR 对象

```

jQuery 1.5 的变化 - 1.5 之后

```

// 很像 Promise 的写法
var ajax = $.ajax('data.json')
ajax.then(function () {
  console.log('success 1')
}, function () {
  console.log('error 1')
})
.ajax.then(function () {
  console.log('success 2')
}, function () {
  console.log('error 2')
})

```

jQuery 1.5 的变化

无法改变 JS 异步和单线程的本质
只能从写法上杜绝 **callback** 这种形式
它是一种语法糖形式，但是解耦了代码
很好的体现:开放封装原则

1) 什么是 deferred 对象？

开发网站的过程中，我们经常遇到某些耗时很长的 **javascript** 操作。其中，既有异步的操作（比如 **ajax** 读取服务器数据），也有同步的操作（比如遍历一个大型数组），它们都不是立即能得到结果的。

通常的做法是，为它们指定回调函数（**callback**）。即事先规定，一旦它们运行结束，应该调用哪些函数。

但是，在回调函数方面，jQuery 的功能非常弱。为了改变这一点，jQuery 开发团队就设计了 **deferred** 对象。

简单说，**deferred** 对象就是 jQuery 的回调函数解决方案。

2) deferred 应用

我们来看一个具体的例子。假定有一个很耗时的操作 **wait**:

```
var wait = function() {  
  
    var tasks = function() {  
  
        alert('执行完毕');  
  
    };  
  
    setTimeout(tasks,5000);  
  
}
```

我们为它指定回调函数，应该怎么做呢？

你可能会使用\$when()

```
$.when(wait())

.done(function(){

    alert('成功!')

})

.fail(function() {

    alert('出错!');

})
```

但是，这样写的话，done()方法会立即执行，起不到回调函数的作用。原因在于\$.when()的参数只能是 deferred 对象，所以必须对 wait()进行改写：

```
var dtd = $.Deferred(); //新建一个 deferred
```

```
var wait = function (dtd) {

    var tasks = function () {

        console.log('执行完毕');

        dtd.resolve(); //改变 deferred 对象执行状态

    };

    setTimeout(tasks, 5000);
```

```
    return dtd;

}
```

现在，`wait()`函数返回的是 `deferred` 对象，这就可以加上链式操作了。

```
$.when(wait(dtd))

    .done(function(){ alert("成功！"); })

    .fail(function(){ alert("出错！"); });
```

更多内容可参考[这里](#)

四、Promise 的基本使用和原理

1) 什么是 Promise

一个 **Promise** 对象代表一个目前还不可用，但是在未来的某个时间点可以被解析的值。它允许你以一种同步的方式编写异步代码。**Promise** 的出现，原本是为了解决回调地狱的问题。所有人在讲解 **Promise** 时，都会以一个 **ajax** 请求为例，此处我们也用一个简单的 **ajax** 的例子来带大家看一下 **Promise** 是如何使用的。

ajax 请求的传统写法：

```
getData(method, url, successFun, failFun){

    var xmlHttp = new XMLHttpRequest();

    xmlHttp.open(method, url);

    xmlHttp.send();

    xmlHttp.onload = function () {

        if (this.status == 200 ) {

            successFun(this.response);
```

```

    } else {

        failFun(this.statusText);

    }

};

xmlHttp.onerror = function () {

    failFun(this.statusText);

};

}

```

改为 promise 后的写法:

```

getData(method, url){

    var promise = new Promise(function(resolve, reject){

        var xmlHttp = new XMLHttpRequest();

        xmlHttp.open(method, url);

        xmlHttp.send();

        xmlHttp.onload = function () {

            if (this.status == 200 ) {

                resolve(this.response);

            } else {

                reject(this.statusText);

            }

        };

    });
}

```

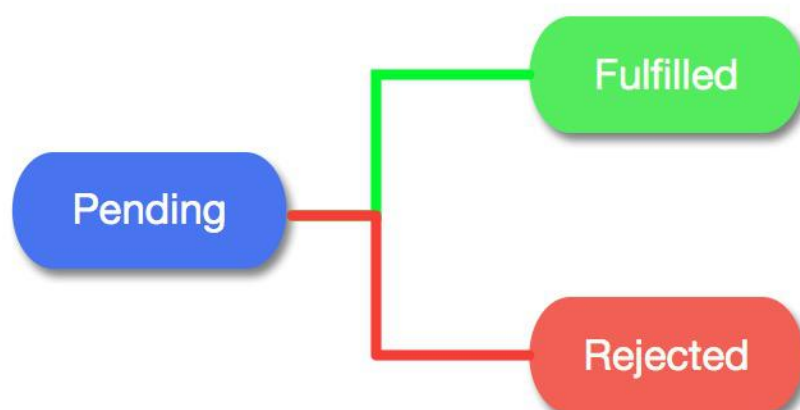
```
xmlHttpRequest.onerror = function () {  
  
    reject(this.statusText);  
  
};  
  
})  
  
return promise;  
  
}  
  
getData('get', 'www.xxx.com').then(successFun, failFun)
```

很显然，我们把异步中使用回调函数的场景改为了`.then()`、`.catch()`等函数链式调用的方式。基于 `promise` 我们可以把复杂的异步回调处理方式进行模块化。

2) Promise 的原理分析

其实 `promise` 原理说起来并不难，它内部有三个状态，分别是 `pending`, `fulfilled` 和 `rejected`。

`pending` 是对象创建后的初始状态，当对象 `fulfill`(成功)时变为 `fulfilled`，当对象 `reject`(失败)时变为 `rejected`。且只能从 `pending` 变为 `fulfilled` 或 `rejected`，而不能逆向或从 `fulfilled` 变为 `rejected`，从 `rejected` 变为 `fulfilled`。如图所示：



使用 webpack 构建多页面应用

前言

之前使用 vue2.x + webpack3.x 撸了一个 vue 单页脚手架

vue 版 spa 脚手架

有兴趣的同学可以看下,内附详细注释,适合刚学习 webpack 的童鞋.

react 版 spa 脚手架

但在一些场景下,单页应用显然无法满足我们的需求,于是便有了

mulXc-cli

好了,废话不多说,进入正题!!!!

文件结构

— build	构建服务和 webpack 配置
— build.js	构建全量压缩包 (打包项目)
— setting.js	多页面入口配置
— style.js	页面 1 对 1 抽取生成 css 文件
— utils.js	工具类
— webpack.base.conf.js	webpack 通用配置
— webpack.dev.conf.js	webpack 开发环境配置
— webpack.prod.conf.js	webpack 生产环境配置
— config	webpack 开发/生产环境部分配置

—— dist	项目打包目录
—— package.json	项目配置文件
—— src	项目目录
—— common	多页面公用方法与 css
—— about	about 页面
—— home	home 页面

思路

多页面应用,顾名思义:就是有多个页面(废话!!!)

从 **webpack** 的角度来看:

- 1.多个入口(entry),每个页面对应一个入口,理解为 js 资源.
- 2.多个 html 实例,webpack 使用 **html-webpack-plugin** 插件来生成 html 页面.
- 3.每个页面需要对应的 css 文件.webpack 使用 **extract-text-webpack-plugin** 抽取 css.

这样我们一个多页面应用该有的东西都具备了,go,开撸!!!

入口配置与 html 页面生成

通过以上文件结构,我们可以找到我们在 **build/setting.js** 进行了多页面入口配置与 html 页面生成。

setting.js

//node 文件操作模块

```
const fs = require('fs');
```



```
//node 路径模块

const path = require('path');

//使用 node.js 的文件操作模块来获取 src 文件夹下的文件夹名称 ->[about,common,home]

const entryFiles = fs.readdirSync(path.resolve(__dirname, '../src'));

//生成 html 文件插件

const HtmlWebpackPlugin = require('html-webpack-plugin');

//工具类提取_resolve 方法

const { _resolve } = require('../utils');

//入口文件过滤 common 文件夹(因为 common 文件夹我们用来存放多页面之间公用的方法与 css,所以不放入入口进行构建!)

const rFiles = entryFiles.filter(v => v !== 'common');

module.exports = {

  //构建 webpack 入口

  entryList: () => {

    const entryList = {};

    rFiles.map(v => {

      entryList[v] = _resolve(`../src/${v}/index.js`);

    });

    return entryList;

  },

  //src 文件夹下的文件夹名称 ->[about,common,home]
```

```
entryFiles: entryFiles,

//使用 html-webpack-plugin 生成多个 html 页面.=>[home.html,about.html]

pageList: () => {

  const pageList = [];

  rFiles.map(v => {

    pageList.push(

      new HtmlWebpackPlugin({

        template: _resolve(`../src/${v}/index.html`),

        filename: _resolve(`../dist/${v}.html`),

        chunks: ['common', v],

        //压缩配置

        minify: {

          //删除 Html 注释

          removeComments: true,

          //去除空格

          collapseWhitespace: true,

          //去除属性引号

          removeAttributeQuotes: true

        },

        chunksSortMode: 'dependency'

      })

    )

  })

}
```

```
    );

  });

  return pageList;

}

};
```

webpack.base.conf.js

//引入 setting.js 入口配置方法,与 html 生成配置

```
const { entryList, pageList } = require('./setting.js');

const baseConf = {

  entry: entryList(),

  plugins: [...pageList()]

};
```

CSS 文件生成

通过以上文件结构,我们可以找到我们在 build/style.js 进行了 css 文件生成。

style.js

```
const path = require('path');

//抽取 css 文件插件

const ExtractTextPlugin = require('extract-text-webpack-plugin');

//引入入口配置
```

```
const { entryList, entryFiles } = require('./setting.js');
```

```
//多个 ExtractTextPlugin 实例
```

```
const plugins = [];
```

```
entryFiles.map(v => {
```

```
  plugins.push(
```

```
    new ExtractTextPlugin({
```

```
      filename: `css/${v}.[contenthash].css`,
```

```
      allChunks: false
```

```
    })
```

```
  );
```

```
});
```

```
module.exports = {
```

```
  //使用正则匹配到每个页面对应 style 文件夹下的 css/less 文件,并配置 loader  
  来进行解析.从而实现 html<->css 1 对 1
```

```
  rulesList: () => {
```

```
    const rules = [];
```

```
    entryFiles.map((v, k) => {
```

```
      rules.push({
```

```
        test: new RegExp(`src(\\\\\\\\|\\\\/){v}(\\\\\\\\\\\\\\\\|\\\\/\\\\)style(\\\\\\\\\\\\\\\\|\\\\/\\\\).*\\\\.  
(css|less)$`),
```

```
        use: plugins[k].extract({
```

```

        fallback: 'style-loader',

        use: ['css-loader', 'postcss-loader', 'less-loader']

    })

  });

});

return rules;

},

// 插件实例

stylePlugins: plugins

};

```

webpack.prod.conf.js

// 引入方法

```

const { rulesList, stylePlugins } = require('./style.js');

const prodConf = {

  module: {

    rules: [...rulesList()]

  },

  plugins: [...stylePlugins]

};

```

总结

ok,一个简易版的多页面应用脚手架就这样搞定啦,是不是很简单!!!

[gayhub](#) 地址

喜欢的童鞋给个 **star** 哈.您的支持就是我最大的动力!!!