JavaScript高级

目录

[JavaScript高级 1](#_Toc494819115)

[day01 3](#_Toc494819116)

[01-介绍阶段目标.avi 3](#_Toc494819117)

[02-回顾-JavaScript特点.avi 4](#_Toc494819118)

[03-回顾-JavaScript组成.avi 4](#_Toc494819119)

[04-浏览器的组成.avi 4](#_Toc494819120)

[05-对象介绍.avi 4](#_Toc494819121)

[06-面向对象介绍.avi 4](#_Toc494819122)

[07-演示面向对象的开发过程.avi 4](#_Toc494819123)

[08-创建对象.avi 4](#_Toc494819124)

[09-构造函数.avi 4](#_Toc494819125)

[10-静态成员和实例成员.avi 4](#_Toc494819126)

[11-构造函数的原型.avi 4](#_Toc494819127)

[12-对象的原型.avi 4](#_Toc494819128)

[13-三角关系.avi 5](#_Toc494819129)

[14-原型链.avi 5](#_Toc494819130)

[15-属性查找规则.avi 5](#_Toc494819131)

[16-注意点.avi 5](#_Toc494819132)

[17-扩展内置对象.avi 5](#_Toc494819133)

[18-随机方块-工具对象.avi 5](#_Toc494819134)

[19-随机方块-方块对象.avi 5](#_Toc494819135)

[20-随机方块-随机生成坐标.avi 5](#_Toc494819136)

[21-随机方块-调用.avi 5](#_Toc494819137)

[day02 5](#_Toc494819138)

[01-项目介绍.avi 5](#_Toc494819139)

[02-创建食物对象.avi 6](#_Toc494819140)

[03-随机生成食物.avi 6](#_Toc494819141)

[04-删除食物.avi 6](#_Toc494819142)

[05-自调用函数，避免命名冲突.avi 6](#_Toc494819143)

[06-蛇对象.avi 6](#_Toc494819144)

[07-游戏对象.avi 6](#_Toc494819145)

[08-蛇的move方法.avi 6](#_Toc494819146)

[09-删除蛇.avi 6](#_Toc494819147)

[10-蛇移动.avi 6](#_Toc494819148)

[11-键盘控制蛇的移动.avi 6](#_Toc494819149)

[12-蛇吃食物.avi 6](#_Toc494819150)

[13-main执行代码.avi 7](#_Toc494819151)

[14-bind.avi 7](#_Toc494819152)

[15-自调用函数的问题.avi 7](#_Toc494819153)

[16-压缩代码.avi 7](#_Toc494819154)

[17-自调用函数的参数.avi 7](#_Toc494819155)

[day03 7](#_Toc494819156)

[01-预解析-预解析的概念01-02.avi 7](#_Toc494819157)

[02-预解析-js中预解析的特点-03-变量名提升.avi 7](#_Toc494819158)

[03-预解析-js中预解析的特点-04-变量名提升-练习.avi 7](#_Toc494819159)

[04-预解析-js中预解析的特点-05-函数声明.avi 8](#_Toc494819160)

[05-预解析-js中预解析的特点-06-声明式函数的特点.avi 8](#_Toc494819161)

[06-预解析-js中预解析的特点-07-各种函数定义形式的异同.avi 8](#_Toc494819162)

[07-预解析-js中预解析的特点-08-函数声明与变量声明在一起的时候的注意点.avi 8](#_Toc494819163)

[08-词法作用域-作用域的概念\_词法作用域以及块级作用域.avi 8](#_Toc494819164)

[09-词法作用域-js中词法作用域的概念细节-01.avi 8](#_Toc494819165)

[10-词法作用域-js中词法作用域的概念细节-02.avi 9](#_Toc494819166)

[11-词法作用域-代码分析练习-01-问题说明.avi 9](#_Toc494819167)

[12-词法作用域-代码分析练习-02-分析步骤.avi 9](#_Toc494819168)

[13-词法作用域-代码分析练习-03-参数与变量声明问题.avi 9](#_Toc494819169)

[14-词法作用域-代码分析案例-04-参数与变量和函数声明的问题.avi 9](#_Toc494819170)

[15-作用域链-作用域链的概念.avi 9](#_Toc494819171)

[day04 9](#_Toc494819172)

[01-闭包-闭包的概念.avi 10](#_Toc494819173)

[02-闭包-闭包要解决的问题-01-提出问题.avi 10](#_Toc494819174)

[03-闭包-闭包要解决的问题-02-利用return的办法.avi 10](#_Toc494819175)

[04-闭包-闭包要解决的问题-03-利用返回函数实现间接访问闭包中的数据.avi 10](#_Toc494819176)

[05-函数是对象-将函数作为一般数据使用-01-赋值操作.avi 10](#_Toc494819177)

[06-函数是对象-将函数作为一般数据使用-02-将函数作为参数传递.avi 10](#_Toc494819178)

[07-函数是对象-将函数作为一般数据使用-03-函数作为参数的使用方法就是调用.avi 11](#_Toc494819179)

[08-函数是对象-将函数作为一般数据使用-04-将函数作为返回值使用.avi 11](#_Toc494819180)

[09-闭包的应用-操作函数中多个变量的方法.avi 11](#_Toc494819181)

[10-闭包的应用-实现含有私有数据的对象.avi 11](#_Toc494819182)

[11-闭包的应用-闭包的核心技术.avi 11](#_Toc494819183)

[12-闭包的应用-沙箱模式.avi 11](#_Toc494819184)

[day05 12](#_Toc494819185)

[01-函数的四种调用模式-分类介绍与函数调用模式.avi 12](#_Toc494819186)

[02-函数的四种调用模式-方法调用模式.avi 12](#_Toc494819187)

[03-函数的四种调用模式-面试题.avi 12](#_Toc494819188)

[04-函数的四种调用模式-构造器模式-01-区分与执行步骤.avi 12](#_Toc494819189)

[05-函数的四种调用模式-构造器模式-02-返回数据.avi 12](#_Toc494819190)

[06-函数的四种调用模式-面试题.avi 12](#_Toc494819191)

[07-关于构造函数结合性的一个小结.avi 13](#_Toc494819192)

[08-函数的四种调用模式-上下文调用模式-01-概念与语法形式介绍.avi 13](#_Toc494819193)

[09-函数的四种调用模式-上下文调用模式-02-使用方法.avi 13](#_Toc494819194)

[10-函数的四种调用模式-上下文调用模式-03-带有参数的调用.avi 13](#_Toc494819195)

[11-函数的四种调用模式-上下文调用模式-实际应用.avi 13](#_Toc494819196)

[12-函数的四种调用模式-call调用语法.avi 13](#_Toc494819197)

[13-函数的四种调用模式-借用构造函数继承.avi 14](#_Toc494819198)

day01

01-介绍阶段目标.avi

1. 理解面向对象开发思想
2. 掌握 JavaScript 面向对象开发相关模式
3. 掌握在 JavaScript 中使用正则表达式

02-回顾-JavaScript特点.avi

**JavaScript 是什么**

解析执行：轻量级解释型的

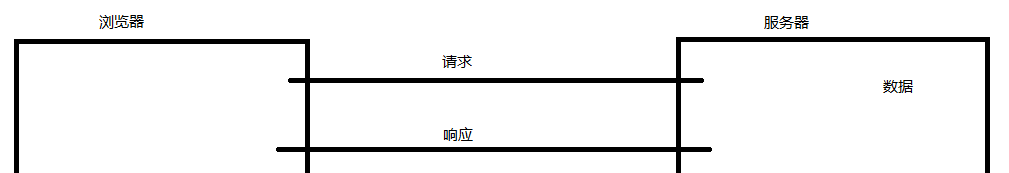
语言特点：动态，头等函数 (First-class Function)

又称函数是 JavaScript 中的一等公民

执行环境：在宿主环境（host environment）下运行，浏览器是最常见的 JavaScript 宿主环境

但是在很多非浏览器环境中也使用 JavaScript ，例如 node.js

**为什么要有JS呢?**



**JS最初的目的是用来对客户端的输入信息预检测**

由于以前的网速慢,如果一旦用户输入的信息格式错误,那来回太消耗时间和资源了

所以可以在客户端将一些格式给规范好

比如:

长度应为6~18个字符

用户名必须以英文字母开头

该用户名已被注册

如果不满足以上条件,直接不发送给服务器

**JS的发展**

技术在发展,时代在进步

JS也在慢慢的进化

第一阶段:

在JS的发展过程中,有一段时间一度被用于小广告,当然这个阶段很快就过去了



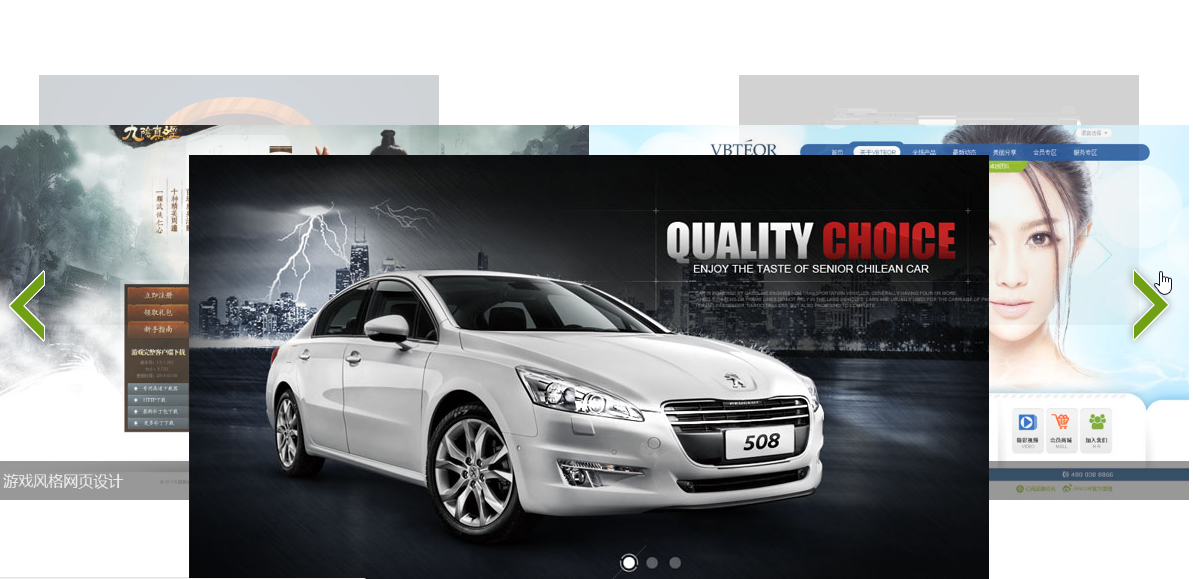
第二阶段:

Google将JS用来做异步框架,实现搜索框实时提示



第三阶段:

人们都是视觉动物,都会追求更美好的界面,所以在这个阶段开始,JS被大量用于做一些界面特效



第四阶段:

人类的本性是堕落的,是慵懒的,特别是世界上最懒的一个人种:**程序猿**

他们认为JS仅仅只能做网页前端一点都不酷,有一颗一统天下的雄心

所以在这个阶段,JS可以做一些移动端的应用了,也就是**WebApp**

第五阶段:

人类的野心也是很强大的,咱们发现JS全都是在前端技术领域非常牛逼,但是后台依旧要依赖服务器应用,所以这时候又出现了一个很厉害的技术:**NodeJS**

最后阶段:

JS(PHP)已经是世界上最好的语言了...

03-回顾-JavaScript组成.avi

**JavaScript 的组成**

1. ECMAScript - 语法规范

变量、数据类型、类型转换、操作符

流程控制语句：判断、循环语句

数组、函数、作用域、预解析

对象、属性、方法、简单类型和复杂类型的区别

内置对象：Math、Date、Array，基本包装类型String、Number、Boolean

1. Web APIs

BOM

onload页面加载事件，window顶级对象

定时器

location、history

DOM

获取页面元素，注册事件

属性操作，样式操作

节点属性，节点层级

动态创建元素

事件：注册事件的方式、事件的三个阶段、事件对象

**JavaScript 可以做什么**

阿特伍德定律：

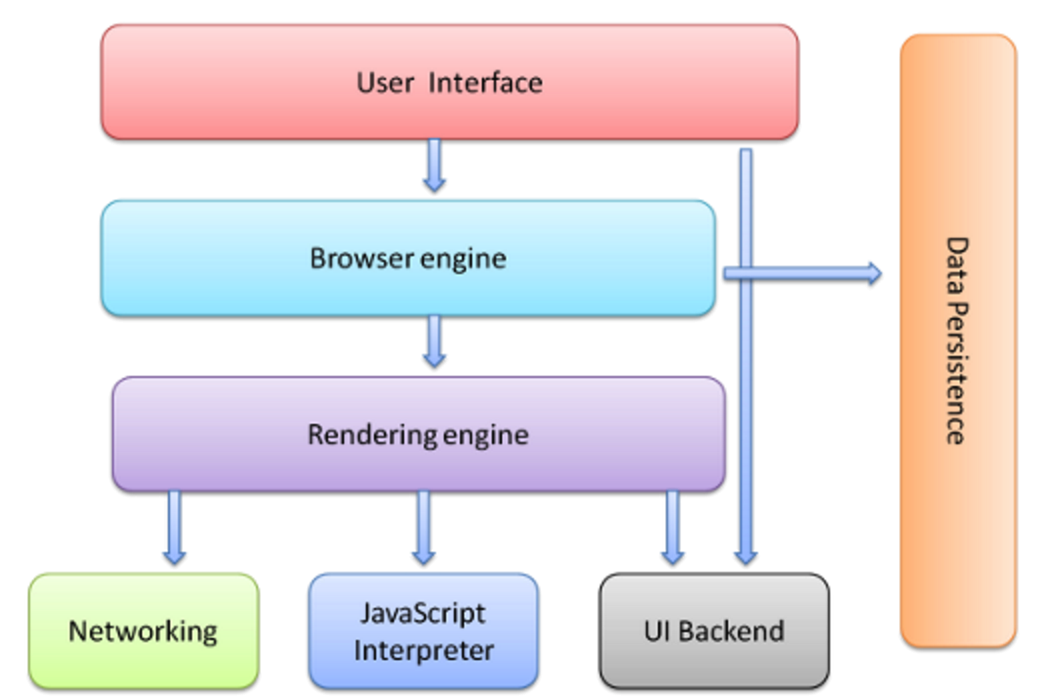
Any application that can be written in JavaScript, will eventually be written in JavaScript.

任何可以用JavaScript来写的应用，最终都将用JavaScript来写

阿特伍德 stackoverflow的创始人之一

一句话总结: **JavaScript是万能的!** JavaScript是世界上最好的语言!

04-浏览器的组成.avi



用户界面: 位于最上层,也就是我们用户看到的页面

浏览器引擎: 用户界面显示的东西都是由浏览器引擎提供的,浏览器引擎可以对下层的渲染引擎进行操作,提取数据显示给用户界面.还会用到数据持久化,说人话也就是存储一些数据到本地

渲染引擎: 用于解析代码,提供数据给上层浏览器引擎,渲染引擎需要依赖网络模块,JS的解析器,UI后端(组合框,弹框等)

通过以上三层,用户才能看到最终的页面,三层都是各司其职的

05-对象介绍.avi

**(1) 对象是单个事物的抽象。**

一本书、一辆汽车、一个人都可以是对象，一个数据库、一张网页、一个与远程服务器的连接也可以是对象。当实物被抽象成对象，实物之间的关系就变成了对象之间的关系，从而就可以模拟现实情况，针对对象进行编程。

**(2) 对象是一个容器，封装了属性（property）和方法（method）。**

属性是对象的状态，方法是对象的行为（完成某种任务）。比如，我们可以把动物抽象为animal对象，使用“属性”记录具体是那一种动物，使用“方法”表示动物的某种行为（奔跑、捕猎、休息等等）。

在实际开发中，对象是一个抽象的概念，可以将其简单理解为：数据集或功能集。

ECMAScript-262 把对象定义为：无序属性的集合，其属性可以包含基本值、对象或者函数。 严格来讲，这就相当于说对象是一组没有特定顺序的值。对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。

提示：每个对象都是基于一个引用类型创建的，这些类型可以是系统内置的原生类型，也可以是开发人员自定义的类型。

通过前面的讲解我们知道类是可以存放方法的，所以，我们就考虑使用类封装来这多个方法，将来再做数组的操作时，不用去找具体的方法，先找到这个类，然后使用这个类中的方法(调用类中的方法；类名.方法名)。这就是面向对象思想的编程方式。

**面向对象与面向过程的区别**

面向过程：

完成一个需求的步骤：首先是搞清楚我们要做什么，然后在分析怎么做，最后我们再代码体现。一步一步去实现，而具体的每一步都需要我们去实现和操作。这些步骤相互调用和协作，完成我们的需求。(在上面的每一个具体步骤中我们都是参与者)（注释：需求——分析——实现）

那么什么是面向过程开发呢? 面向过程开发，其实就是面向着具体的每一个步骤和过程，把每一个步骤和过程完成，然后由这些功能方法相互调用，完成需求。

面向过程的代表语言：C语言

当需求单一，或者简单时，我们一步一步去操作没问题，并且效率也挺高。可随着需求的更改（项目开发中更改需求），功能的增多，发现需要面对每一个步骤很麻烦了，这时就开始思索，能不能把这些步骤和功能在进行封装，封装时根据不同的功能，进行不同的封装（不同的方法），这样结构就清晰了很多。

用的时候，找到对应的类就可以了，这就是面向对象的思想。

**世界是由对象组成的: 万物皆对象。**

06-面向对象介绍.avi

面向对象是基于面向过程的编程思想（只是把面向过程的东西进行了封装）。

特点：是一种更符合我们思想习惯的思想（遇到问题，让对象去解决）；

可以将复杂的事情简单化（条理清晰，后期好维护）

将我们从执行者变成了指挥者（直接指挥对象调用方法就可）

面向过程强调的是：每一个步骤的功能（更关注过程）；而面向对象强调的是：对象，然后有对象调用相应的功能（更关注结果）。



面向对象编程 —— Object Oriented Programming，简称 OOP ，是一种编程开发思想。 它将真实世界各种复杂的关系，抽象为一个个对象，然后由对象之间的分工与合作，完成对真实世界的模拟。

在面向对象程序开发思想中，每一个对象都是功能中心，具有明确分工，可以完成接受信息、处理数据、发出信息等任务。 因此，面向对象编程具有灵活、代码可复用、高度模块化等特点，容易维护和开发，比起由一系列函数或指令组成的传统的过程式编程（procedural programming），更适合多人合作的大型软件项目。

面向对象与面向过程：

面向过程就是亲力亲为，事无巨细，面面俱到，步步紧跟，有条不紊

面向对象就是找一个对象，指挥得结果

面向对象将执行者转变成指挥者

面向对象不是面向过程的替代，而是面向过程的封装

面向对象的特性：

封装性

继承性

[多态性]抽象

07-演示面向对象的开发过程.avi

通过一个小案例演示面向过程和面向对象的区别

打印成绩表

面向过程的方式: 使用对象记录成绩数据

*// 打印学生的成绩表  
//   
//   
// 1 面向过程的方式  
// // 1.1 记录学生的成绩***var *stu1*** = {**name**: **'zs'**, **subject**: **'语文'**, **score**: 90};  
**var *stu2*** = {**name**: **'ls'**, **subject**: **'语文'**, **score**: 80};  
*// // 1.2 打印学生的成绩***console**.log(***stu1***.**name**, ***stu1***.**subject**, ***stu1***.**score**);  
**console**.log(***stu2***.**name**, ***stu2***.**subject**, ***stu2***.**score**);

面向对象的方式: 一个构造函数作为创建对象的模板,创建的对象都是独立的个体,并且都具有打印自身成绩的方法

*//2 面向对象的方式  
// 创建一个模板，用于创建对象（实例instance）  
// 在JavaScript中创建对象的模板是构造函数  
// 而在其他语言中创建对象的模板是类***function** *Student*(name, subject, score) {  
 **this**.**name** = name;  
 **this**.**subject** = subject;  
 **this**.**score** = score;  
  
 **this**.printScore = **function** () {  
 **console**.log(**this**.**name**, **this**.**subject**, **this**.**score**);  
 }  
}  
  
**var *stu1*** = **new** *Student*(**'zs'**, **'语文'**, 90);  
**var *stu2*** = **new** *Student*(**'ls'**, **'语文'**, 80);  
  
***stu1***.printScore();  
***stu2***.printScore();

08-创建对象.avi

创建对象的第一种方式: new Object() 创建一个空对象,然后可以添加属性

*// 1 new Object()***var *hero*** = **new** Object(); *// 空对象****hero***.**blood** = 100;  
***hero***.**name** = **'刘备'**;  
***hero***.**weapon** = **'剑'**;  
  
***hero***.attack = **function** () {  
 **console**.log(**this**.**weapon** + **' 攻击敌人'**);  
}

创建对象的第二种方式: 字面量 {}, 也是创建一个空对象,然后可以添加属性,当然也可以在大括号里使用键值对的方式添加属性

**var *hero*** = {}; *// 空对象***var *hero1*** = {  
 **blood**: 100,  
 **name**: **'刘备'**,  
 **weapon**: **'剑'**,  
 attack: **function** () {  
 **console**.log(**this**.**weapon** + **' 攻击敌人'**);  
 }  
}

创建对象的第三种方式: 工厂函数,由于上面两种方式创建多个对象不太方便,如果需要创建20个对象需要把代码复制20遍,而每个对象只有属性不一样

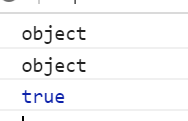
所以才诞生了工厂函数创建对象的方式,可以批量创建多个对象,只需要把每个对象的属性传给函数即可返回一个赋有属性的新对象

**function** *createHero*(name, blood, weapon) {  
 **var** o = **new** Object();  
 o.**name** = name;  
 o.**blood** = blood;  
 o.**weapon** = weapon;  
 o.attack = **function** () {  
 **console**.log(**this**.**weapon** + **' 攻击敌人'**);  
 }  
 **return** o;  
}  
  
**var *hero*** = *createHero*(**'刘备'**, 100, **'剑'**);  
**var *hero1*** = *createHero*(**'关羽'**, 100, **'刀'**);  
*// console.log(typeof hero);*

需要注意的是: typeof是查看数据类型,而在js中数据类型分为两类,1.基本数据类型,2.引用数据类型,而对象就是引用数据类型都是object表示,所以所有的对象都是object类型

**var *arr*** = [];  
**console**.log(**typeof *hero***);  
**console**.log(**typeof *arr***);  
**console**.log(***arr* instanceof** Array);

效果如下:



而instanceof关键字可以查看对象是否是通过该构造函数创建的,例如想看arr是否是通过Array这个构造函数创建出来的,就可以使用arr instanceof Array

09-构造函数.avi

综上所述,如果通过以上三种方式创建对象,其实全部都是使用的Object的构造函数,那么所有的对象其实都是通过Object创建出来的

这样会有个问题,在开发过程中如果能知道这个对象的具体类型,那就方便太多了,例如: 学生这一类型的对象就用Student来表示,老师就用Teacher来表示,数组就用Array来表示

就不会出现管理混乱的情况了,因为如果全部都是Object类型,那将来用对象将会很麻烦

所以就诞生了创建对象的最重要的一种方式: **构造函数**

**之前创建的对象打印在控制台都是object类型**,其实使用该对象时并不能准确的知道它到底是一个什么东西,可能想描述学生对象,可能是想描述狗狗对象,所以需要给之前**创建的对象指定一种自定义的类型**,这时候就需要用到**构造函数**了

构造函数: 是一种比较特殊的函数,在JS中可以用于初始化对象,它和普通函数的区别在于它默认返回值为this,**这个函数不是直接调用的,而是使用new关键字创建对象的**

构造函数的特点:

1.构造函数首字母大写

2.构造函数并没有显示返回任何东西,new 关键字会自动创建给定的类型并返回他们,当调用构造函数时,new会自动创建this对象,且**类型就是构造函数类型**

3.也可以在构造函数中显示调用return.如果返回的值是一个对象，它会代替新创建的对象返回。如果返回的值是一个基本类型，它会被忽略，新创建的对象会被返回.

构造函数也是函数,只是我们习惯将函数名首字母大写,与普通函数进行区分

**只有在函数名()前面加了new关键字,该函数才会被当成构造函数使用**

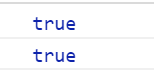
*// 构造函数 -- 构造对象的函数***function** *Hero*(name, blood, weapon) {  
 **this**.**name** = name;  
 **this**.**blood** = blood;  
 **this**.**weapon** = weapon;  
  
 **this**.attack = **function** () {  
 **console**.log(**this**.**weapon** + **' 攻击敌人'**);  
 }  
}   
  
*// 1 会在内存中创建一个空对象  
// 2 设置构造函数的this，让this指向刚刚创建好的对象  
// 3 执行构造函数中的代码  
// 4 返回对象***var *hero*** = **new** *Hero*(**'刘备'**, 100, **'剑'**);

判断构造函数的具体类型需要使用instanceof关键字:

语法: A instanceof B

作用: 判断A是否是通过B创建出来的对象

**console**.log(***hero* instanceof** *Hero*);  
**var *arr*** = [];  
**console**.log(***arr* instanceof** Array);

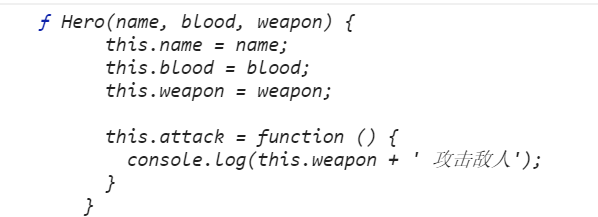


返回值为true表示该对象是通过该构造方法创建出来的

通过对象也可以反过来获取构造函数,每个对象身上都有一个属性: constructor

该属性就是创建对象的构造函数

**console**.log(***hero***.**constructor**);



10-静态成员和实例成员.avi

在对象中有属性和方法

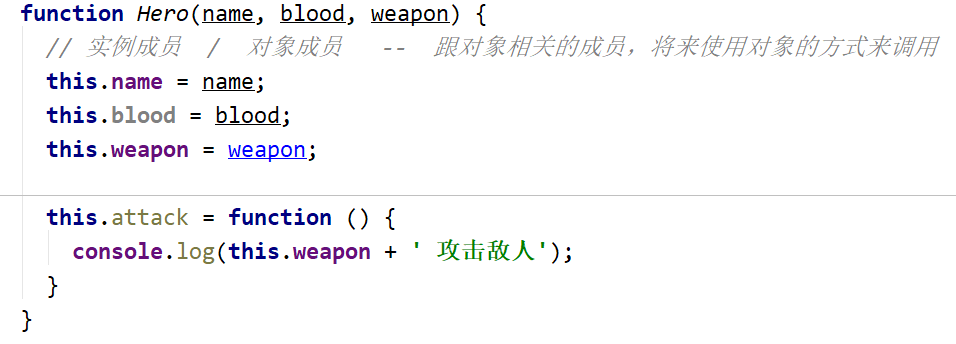
属性: 特征,名词,例如:姓名,性别,年龄等

方法: 行为,动词,例如:说话,洗衣服,做饭,暖床

这些属性和方法都被成为对象的成员

成员分为静态成员和实例(对象)成员

构造函数中用this.赋值的都是对象成员



对象成员可以直接通过对象.的方式访问,例如调用方法:

**var *hero*** = **new** *Hero*(**'刘备'**, 100, **'剑'**);  
***hero***.attack();  
  
**var *hero1*** = **new** *Hero*(**'关羽'**, 100, **'刀'**);  
***hero1***.attack();

效果如下:



添加静态成员: 直接给构造函数来添加成员,构造函数.成员名 = 值

*// 静态成员 -- 直接给构造函数添加的成员  
Hero*.**version** = **'1.0'**;

访问静态成员: 对象无法访问,只能通过构造函数来访问

*// 静态成员不能使用对象的方式来调用***console**.log(***hero***.**version**);  
  
*// 静态成员使用构造函数来调用***console**.log(*Hero*.**version**);



简单点总结就是,静态成员是跟着构造函数走的,对象成员是跟着对象走的

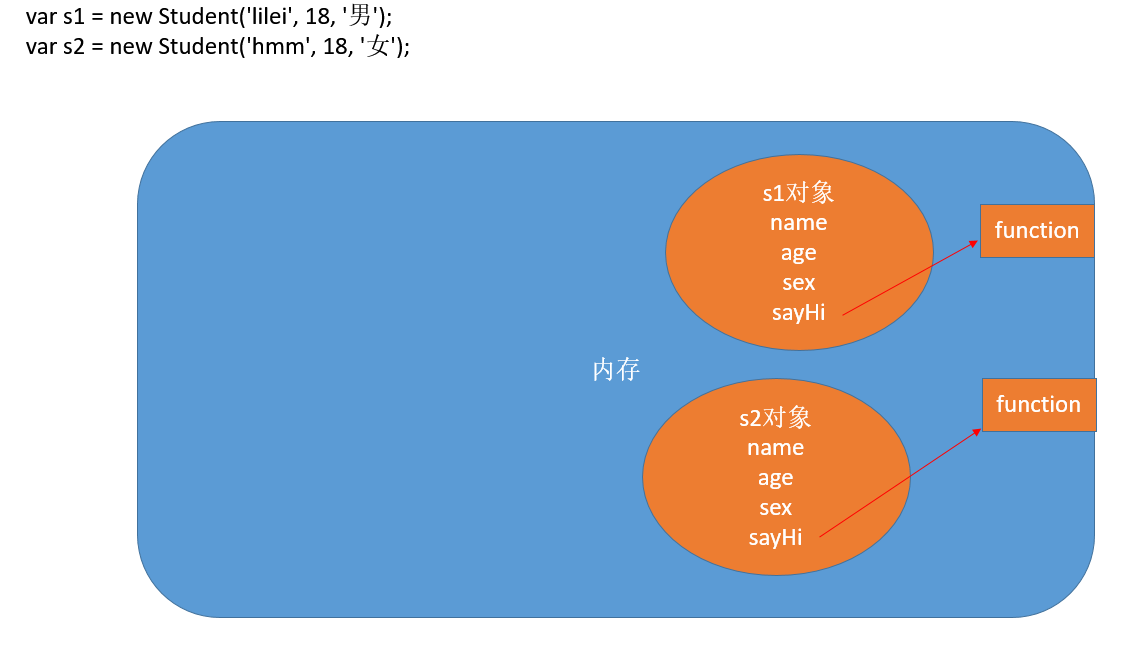
11-构造函数的原型.avi

写一个学生的构造函数并创建多个对象:

*// 构造函数* **function** *Student*(name, age, sex) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 **this**.**sex** = sex;  
  
 *// 多个对象，会存储多个sayHi方法* **this**.sayHi = **function** () {  
 **console**.log(**'大家好，我是'** + **this**.**name**);  
 }  
 }   
  
 **var *s1*** = **new** *Student*(**'lilei'**, 18, **'男'**);  
 **var *s2*** = **new** *Student*(**'hmm'**, 18, **'女'**);  
  
 ***s1***.sayHi();  
 ***s2***.sayHi();  
  
 **console**.log(***s1***.sayHi === ***s2***.sayHi);

这样做会有一个问题,多个对象都有sayHi()方法,就等于是在内存中创建了多次sayHi()方法

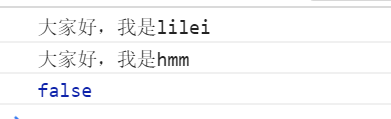
如图所示:



如何判断是否创建了多个呢?

用第一个学生的sayHi与第二个学生的sayHi进行等值判断,如果为true则表示在内存中的地址值相同,也就是只有一个函数,如果为false则表示创建了多个sayHi,注意不能加括号,否则表示调用函数

控制台打印结果为false,证实无误



其实这个问题在循环绑定事件时也遇到过,原理相同

解决方法也相同

可以先在全局位置定义一个有名字的函数,然后在构造中赋值即可

*// 解决sayHi存储多份的问题***function** *Student*(name, age, sex) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 **this**.**sex** = sex;  
  
 *// 多个对象，会存储多个sayHi方法* **this**.sayHi = *sayHi*;  
}  
  
**function** *sayHi*() {  
 **console**.log(**'大家好，我是'** + **this**.***name***);  
}  
  
**var *s1*** = **new** *Student*(**'lilei'**, 18, **'男'**);  
**var *s2*** = **new** *Student*(**'hmm'**, 18, **'女'**);  
  
***s1***.sayHi();  
***s2***.sayHi();  
  
**console**.log(***s1***.sayHi === ***s2***.sayHi);

效果如下:



虽然这个问题解决了

但是这种解决方法并不好,因为一个对象可能会具有很多行为,也就是说在构造函数中会具备很多方法,那这样一来会导致工作量大大增加,而且还极有可能会和其他函数重名产生冲突,对全局作用域污染也很严重

所以需要更好的解决方法: **构造函数的原型**

所有的构造函数都有一个原型对象: prototype

可以通过原型给构造函数添加方法,而且这个方法不管创建多少对象,在内存中都是唯一的

只要是通过该构造函数创建的对象,都可以直接使用通过原型添加的成员

*// 每一个构造函数都有一个属性 原型 / 原型对象* **function** *Student*(name, age, sex) {  
 **this**.**name** = name;  
 **this**.**age** = age;  
 **this**.**sex** = sex;  
 }   
  
 *// console.dir(Student.prototype);  
 Student*.**prototype**.sayHi = **function** () {  
 **console**.log(**'大家好，我是'** + **this**.**name**);  
 }  
  
 *// 通过Student构造函数，创建的对象，可以访问Student.prototype中的成员* **var *s1*** = **new** *Student*(**'lilei'**, 18, **'男'**);  
 **var *s2*** = **new** *Student*(**'hmm'**, 18, **'女'**);  
  
 ***s1***.sayHi();  
 ***s2***.sayHi();  
  
 **console**.log(***s1***.sayHi === ***s2***.sayHi);

效果如下:

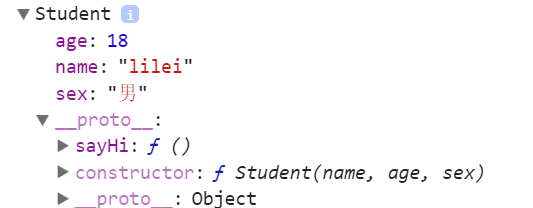


通过原型的方式添加方法,就可以完美的避免了在全局位置创建有名字的函数的问题

12-对象的原型.avi

prototype属性是构造函数中的一个属性

那在创建出来的对象中是否也有这个属性呢?



打印s1对象之后发现并没有prototype属性

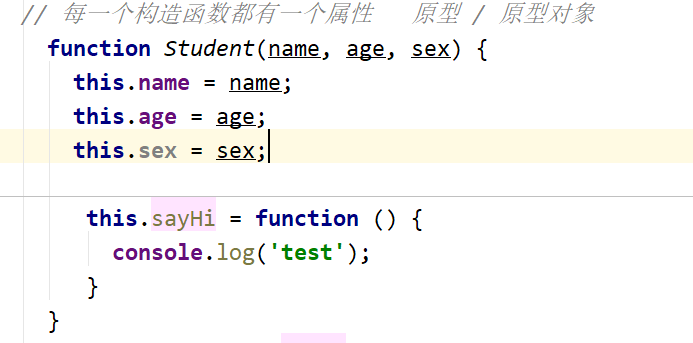
但是展开后看到了一个\_\_proto\_\_属性,该属性也是个对象,再展开后发现里面有sayHi()方法

因此得出结论: prototype只有构造函数才有,对象中的原型属性是\_\_proto\_\_,该属性指向了构造函数中的prototype,他俩一毛一样,只不过一个是构造函数中使用的,一个是对象中使用的

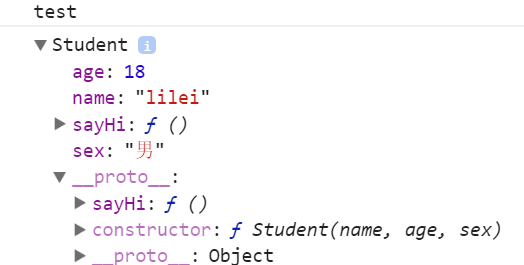
在日志中可以看到,sayHi方法并没有直接存储在s1对象中,而是存储在原型对象中了

如果在s1对象中本身就有sayHi方法结果会是怎样呢?

给构造函数添加一个sayHi方法

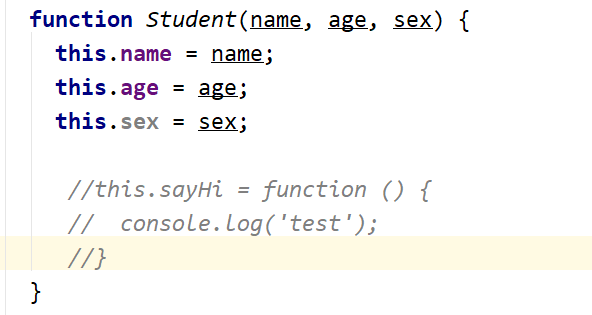


调用sayHi方法:

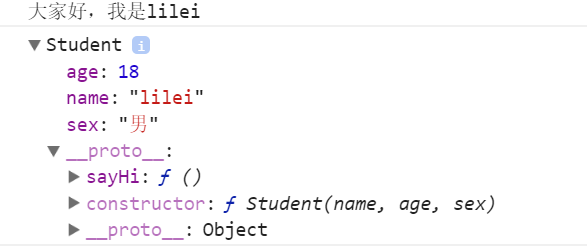


发现打印出来的是test,也就是说调用的是s1对象自身的sayHi方法,而没有调用原型对象中的sayHi方法

将构造函数中的方法注释掉:



效果如下:

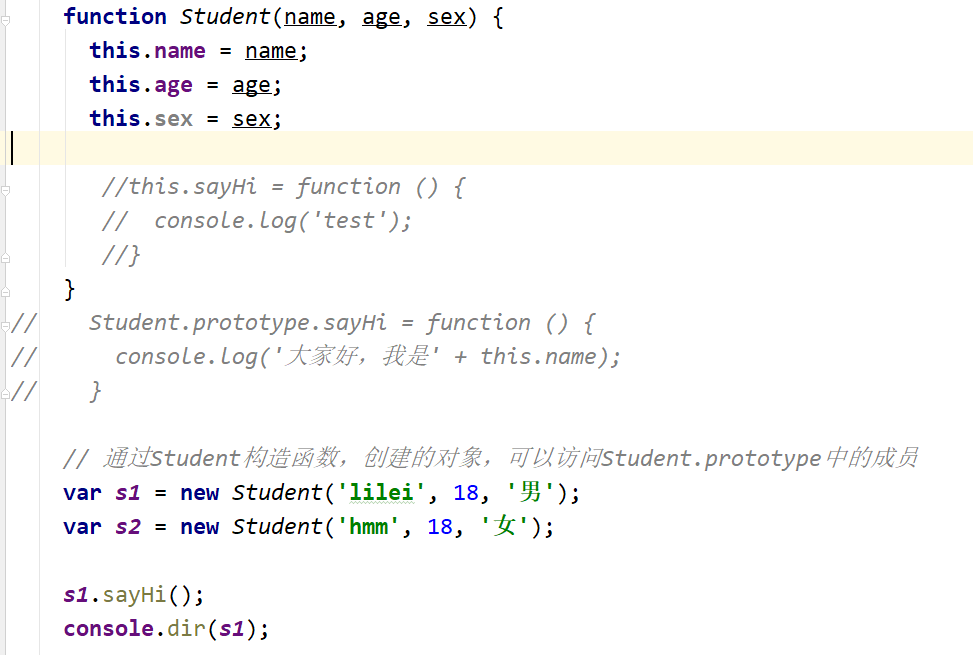


依旧使用了原型的方法

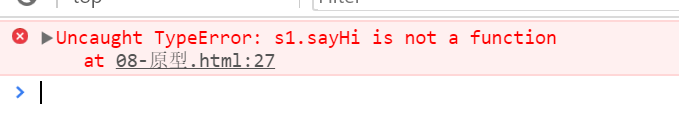
所以,在对象和原型中都有同名方法时,优先使用对象自身的方法(就近原则)

如果该对象没有这个方法,那么就会使用原型中的方法

如果对象中和原型中都没有的方法,此时就会报错



效果如下:



就是因为这个特性,所以在对象中才会有\_\_proto\_\_属性用来访问构造函数的原型对象

但是该属性是非标准属性,所以在生产环境中不要使用

打印日志证明两者是否相等:

console.log(***s1***.\_\_proto\_\_ === *Student*.**prototype**);

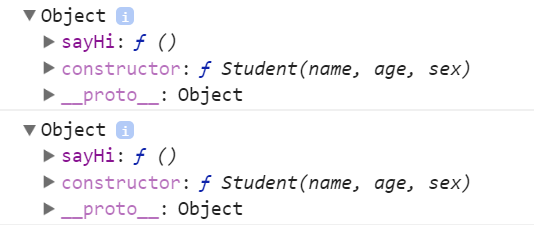
效果如下:



通过dir的方式查看两者的详细信息:

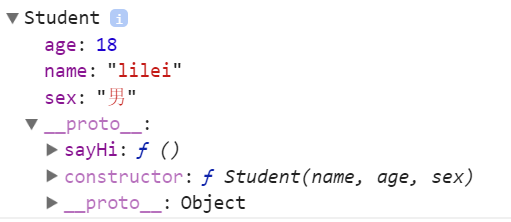
**console**.dir(***s1***.\_\_proto\_\_);  
**console**.dir(*Student*.**prototype**);

效果如下:



发现确实完全一样,但是原型中存着一个constructor属性,保存的就是Student构造函数

在原型中所有的属性对象都可以访问,所以这也是为什么可以通过对象直接访问constructor属性的原因 (constructor属性在原型里而不在对象里)



为什么需要这个属性呢?

这个属性其实就是存储了创建该对象的构造函数

所以通过它就可以判断对象是否使用该构造函数创建的

**console**.log(***s1***.**constructor** === *Student*);  
  
**var *arr*** = [];  
*// Array.prototype.constructor***console**.log(***arr***.**constructor** === Array);

只要是对象都会有constructor属性,该属性存储在原型中,而对象可以直接使用原型中的属性



13-三角关系.avi

构造函数 原型对象 对象 三者之间的关系

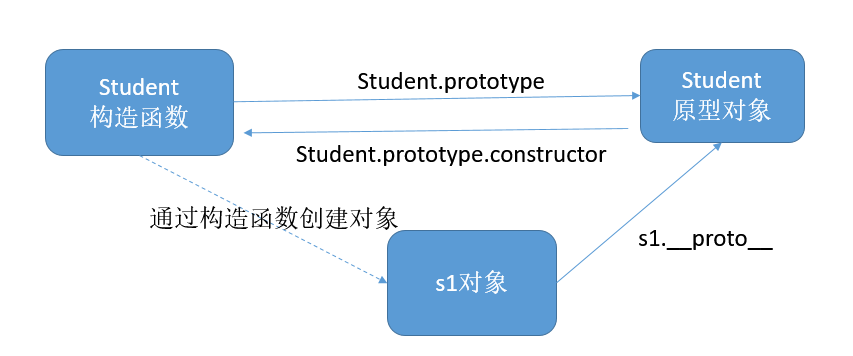
对象是由new关键字结合构造函数创建出来的

构造函数中都有一个原型 prototype

对象中有一个\_\_proto\_\_属性指向了构造函数的原型prototype

原型中有一个属性constructor指向了创建对象的构造函数

三者关系如图所示:



14-原型链.avi

上图是一个最基本的简单三角关系

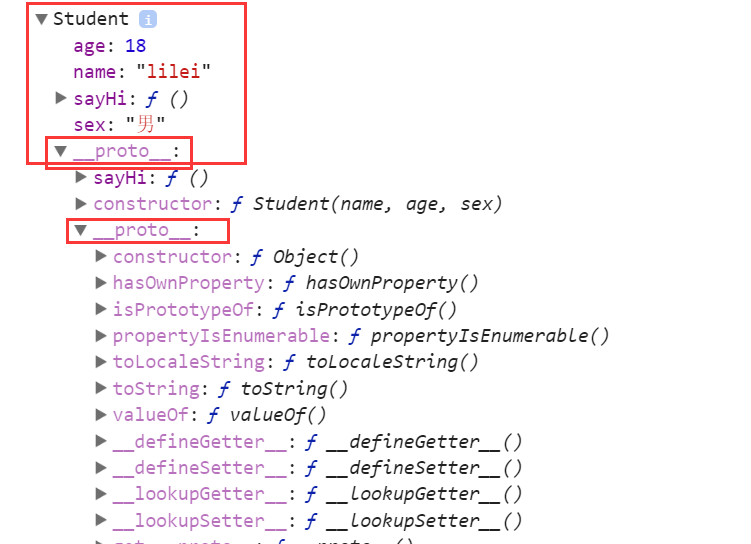
其中有一个细节需要注意,原型对象prototype也是对象

是对象就一定有\_\_proto\_\_属性

那原型对象的\_\_proto\_\_属性指向的是谁呢?

打印s1对象观察

**console**.log(***s1***);

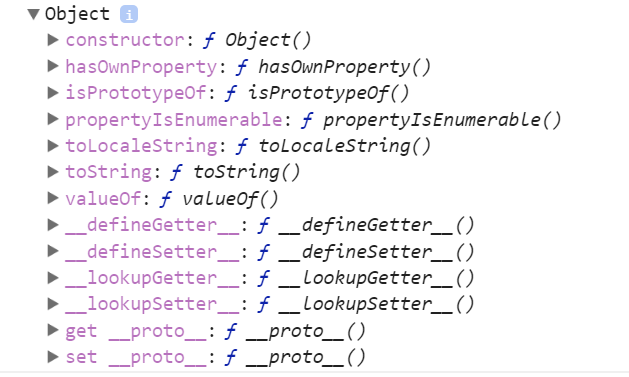


可以看到s1中的\_\_proto\_\_属性是Student的原型

这个原型对象中的\_\_proto\_\_其实是object对象

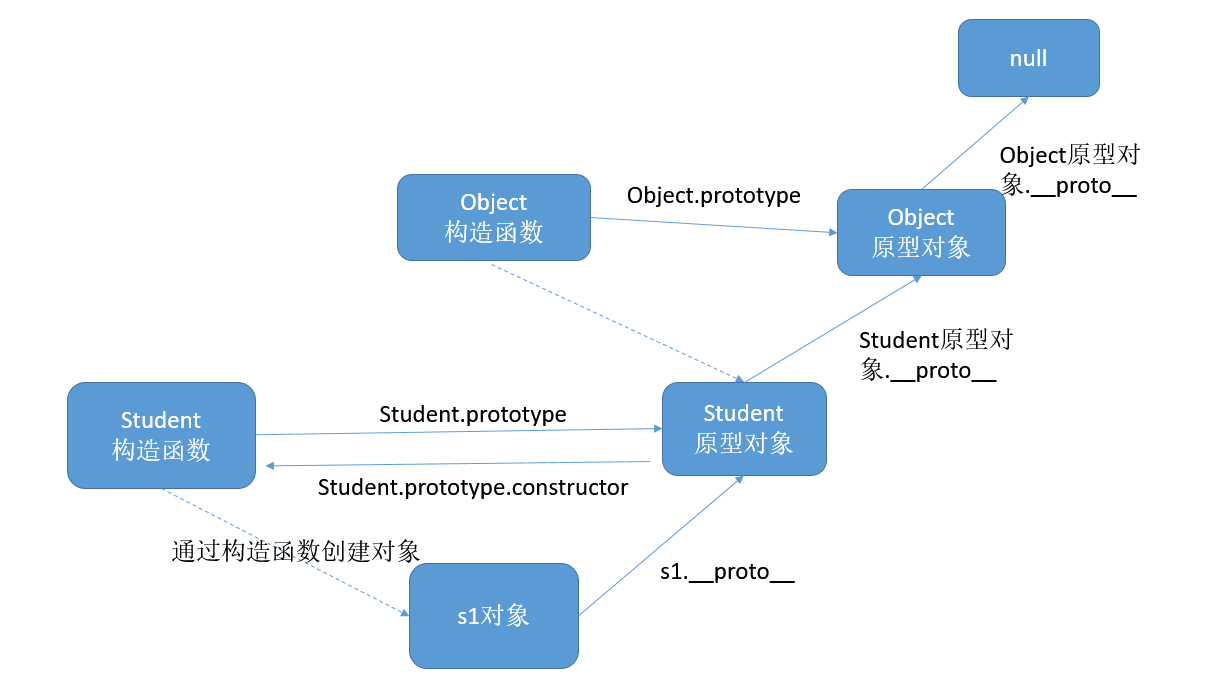
或者可以这样打印

**var *o*** = ***s1***.\_\_proto\_\_;  
**console**.dir(***o***.\_\_proto\_\_);



效果一样

他们的关系如下图:



Student的原型对象其实就是一个由Object构造函数创建的一个对象,这个对象中也有一个\_\_proto\_\_指向了Object构造函数的prototype

再往上找,继续看Object的原型对象中的\_\_proto\_\_原型对象

*// o.\_\_proto\_\_ Object的原型对象* **var *o1*** = ***o***.\_\_proto\_\_;  
 **console**.dir(***o1***.\_\_proto\_\_);



object的原型对象中的原型对象就是null

也就是最顶层就是null

看图可以发现,由s1开始一层一层向上形成一个链状结构

这就是**原型链**

原型链有什么意义呢?

在s1调用方法的时候会先找s1自身有没有,如果没有就会向上找原型,如果没有就会继续找原型对象的原型,逐层往上找,这就是原型链存在的意义

15-属性查找规则.avi

根据分析原型链可以得出结论

s1对象的原型对象的原型对象就是Object构造函数的原型对象prototype

可以打日志查看结果

**console**.log(***s1***.\_\_proto\_\_.\_\_proto\_\_ === Object.**prototype**);



在之前的学习过程中提到过,所有的对象都有toString方法

这是为什么呢?

其实就是原型链顶端Object的原型实现的

s1调用toString没问题

**console**.log(***s1***.toString());



s1对象能调用toString说明s1对象本身有该方法

或者是它的原型链上层有该方法

所以可以将它原型链上面所有的原型对象都打印出来找找

**console**.dir(***s1***);  
**console**.dir(***s1***.\_\_proto\_\_);  
**console**.dir(***s1***.\_\_proto\_\_.\_\_proto\_\_);



最终在s1的原型对象中的原型对象里找到了toString方法

实际上也就是Object构造函数的原型对象里有该方法

而该方法内部打印出来的就是[object Object],所以所有对象打印出来的结果都是这样

当然,了解了原型链的特性后,其实toString方法我们也可以自己重写,那将来调用toString方法就会使用就近原则了

了解了原型链后,接下来需要知道访问属性的原理了

在构造函数的原型中添加一个属性test,赋值为abc,然后直接使用对象去访问属性

*// 构造函数的原型对象增加一个属性  
Student*.**prototype**.**test** = **'abc'**;

*// 读取属性 先在对象本身查找test属性，如果没有找到的话，会去原型链上查找* **console**.log(***s1***.**name**);  
 **console**.log(***s1***.**test**);

效果如下:



说明会现在s1对象上找test属性,如果没找到就会去原型链上继续寻找,找到Student的原型中有test所以就输出了

接下来通过s1对象分别设置name属性和test属性,然后打印s1和s2的name以及test属性,观察会发生什么变化

*// 设置属性****s1***.**name** = **'xxx'**;  
*// test属性在原型对象上，而在设置属性的值的时候，不会搜索原型链  
// 而是直接给对象新增一个test属性****s1***.**test** = **'123xxx'**;  
  
**console**.log(***s1***.**name**);  
**console**.log(***s1***.**test**);  
  
**console**.log(***s2***.**name**);  
**console**.log(***s2***.**test**);

效果如下:



之前说原型属性在内存中是唯一的,就算多个对象也是会共享同一个属性,为何使用s1对象修改了test属性后,s2却没有跟着修改呢?

原因是因为,test属性最开始设置在原型对象中

而这里设置s1.test是设置s1对象中的属性(设置属性时不会先去原型链中找,如果当前对象没有该属性就添加该属性)

所以原型上的属性test压根没有改变,最后输出的时候由于就近原则,s1对象中有test属性了,所以就直接输出了s1对象身上的test属性值

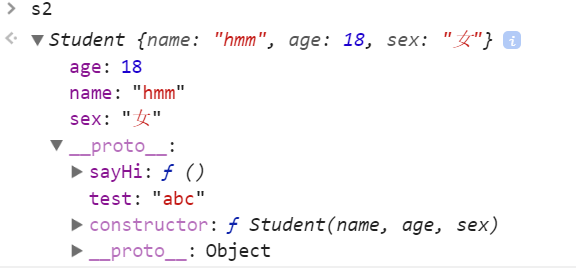
最后打印s2对象的test属性,发现s2对象身上并没有test属性,所以还是在原型链上去找到test属性并输出

所以,s1.test = ‘123xxx’根本没有改变原型对象的test属性,只是给s1对象加了个属性而已

查看日志证实:



确实在s1对象中有一个test属性,s1的原型对象中也有一个test属性,并且原型对象中的test属性压根没有改变



反观s2对象压根没有test属性,所以会去原型链上找到test属性并输出

总结:

获取对象属性时,如果对象自身没有该属性,就会沿着原型链向上找,直到最顶端还没有就报错

而设置对象属性时是直接在对象自身找该属性,如果有就修改,如果没有就添加该属性,所以设置属性时是不会去查找原型链的

16-注意点.avi

接下来介绍使用原型时的注意事项

还是写一个Student构造函数

**function** *Student*(name, age) {  
 *// 一般情况下，对象的属性在构造函数中来设置* **this**.**name** = name;  
 **this**.**age** = age;  
}

属性一般在构造函数中来赋值

方法一般通过原型来添加

*// 一般情况下，对象的方法在构造函数的原型对象中来设置  
 Student*.**prototype**.sayHi = **function** () {  
 **console**.log(**'sayHi'**);  
 }  
 *Student*.**prototype**.eat = **function** () {  
 **console**.log(**'eat'**);  
 }

现在只有两个方法

如果一个对象有很多方法,这样添加还是有些麻烦,因为Student.prototype每次添加方法都要写

不过我们知道prototype其实就是一个对象而已

既然是对象,那就可以使用对象字面量的方式重新赋值,代码如下

Student.**prototype** = {  
 sayHi: **function** () {  
 **console**.log(**'sayHi'**);  
 },  
 eat: **function** () {  
 **console**.log(**'eat'**);  
 }  
}

这样一来就算添加多个方法也不需要重复的写Student.prototype了

但是,这样写有个小问题

之前学习原型时知道原型对象中有一个属性是constructor,其实就是构造函数

如果通过上面的方式来添加方法,就等于是给原型对象重新赋值了,那么constructor属性自然就没了

这个属性没了的后果就是

**var *s1*** = **new** *Student*(**'zs'**, 18);

**console**.log(***s1***.**constructor**);

输出结果:

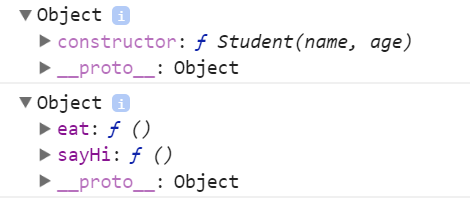


本应该是Student构造函数,结果却是Object构造函数

在给prototype赋值前后分别打印prototype观察

**console**.dir(*Student*.**prototype**);  
*// Student.prototype.constructor  
Student*.**prototype** = {  
 sayHi: **function** () {  
 **console**.log(**'sayHi'**);  
 },  
 eat: **function** () {  
 **console**.log(**'eat'**);  
 }  
}  
**console**.dir(*Student*.**prototype**);

效果如下:



给prototype重新赋值后,constructor属性就没了

那为什么打印s1对象的constructor输出结果是Object构造函数呢?

原因很简单,因为Student构造函数的原型对象中没有constructor,那就会沿着原型链继续找到Object的原型对象,Object的原型对象中有constructor就是Object构造函数

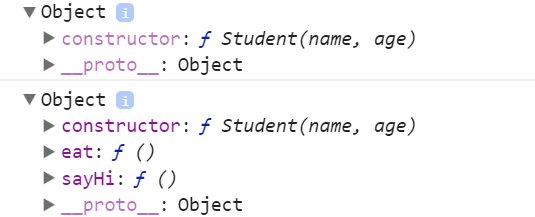
知道了这个问题如何产生的,接下来就是解决问题了

解决问题的方法非常简单,既然给prototype重新赋值constructor属性就没有了

那在重新赋值的时候,手动设置一个constructor属性就可以了,而这个属性的值就是Student构造函数

console.dir(*Student*.**prototype**);  
*// Student.prototype.constructor  
// 可以重新改变原型对象的prototype属性，设置为一个新的对象  
//   
// 当改变构造函数的prototype的时候，需要重新设置constructor属性  
Student*.**prototype** = {  
 **constructor**: *Student*,  
 sayHi: **function** () {  
 console.log(**'sayHi'**);  
 },  
 eat: **function** () {  
 console.log(**'eat'**);  
 }  
}  
 console.dir(*Student*.**prototype**);

继续对比输出结果:



这样一来就完美的将方法一次添加到原型中了,并且也没有破坏原型原有的constructor属性

分别在给原型赋值之前和之后创建对象并使用sayHi方法

原型赋值之后创建对象:

Student.**prototype** = {  
 **constructor**: Student,  
 sayHi: **function** () {  
 **console**.log(**'sayHi'**);  
 },  
 eat: **function** () {  
 **console**.log(**'eat'**);  
 }  
}  
*// console.dir(Student.prototype);***var *s1*** = **new** Student(**'zs'**, 18);  
***s1***.sayHi();

效果如下:

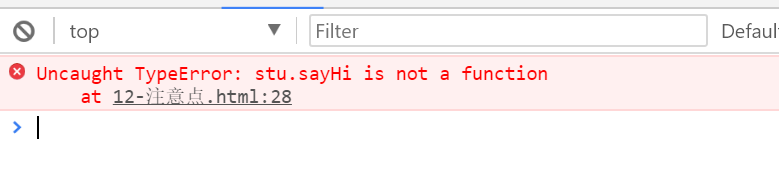


在给原型赋值之后创建对象并调用方法没问题

在原型赋值之前创建对象并使用sayHi方法:

**var *stu*** = **new** *Student*(**'ls'**, 18);  
***stu***.sayHi();  
*// console.dir(Student.prototype);  
// Student.prototype.constructor  
// 可以重新改变原型对象的prototype属性，设置为一个新的对象  
//   
// 当我们改变构造函数的prototype的时候，需要重新设置constructor属性  
Student*.**prototype** = {  
 **constructor**: *Student*,  
 sayHi: **function** () {  
 **console**.log(**'sayHi'**);  
 },  
 eat: **function** () {  
 **console**.log(**'eat'**);  
 }  
}

效果如下:

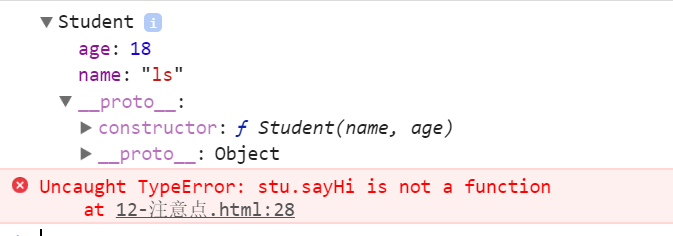


报错了

可以在报错之前打印出stu对象观察

**var *stu*** = **new** *Student*(**'ls'**, 18);  
console.dir(***stu***);  
***stu***.sayHi();

效果如下:



发现对象自身没有sayHi方法,原型中也没有sayHi方法

也就是说在prototype被赋值之前,Student的原型对象还是最初默认的原型对象,这时候创建Student对象,里面存储的都是最初的原型对象

所以**想使用原型对象中的方法,一定要先设置原型属性后再创建对象**

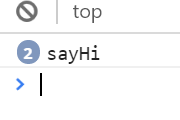
**先去设置原型属性,再创建对象,才可以访问原型对象中的成员**

如果不是给prototype重新赋值,只是给prototype添加成员,则不需要考虑创建对象的先后顺序,只要在添加成员后去使用即可

例如:

**function** *Student*(name, age) {  
 *// 一般情况下，对象的属性在构造函数中来设置* **this**.**name** = name;  
 **this**.**age** = age;  
}  
**var *stu*** = **new** *Student*(**'ls'**, 18);  
  
*Student*.**prototype**.sayHi = **function** () {  
 **console**.log(**"sayHi"**);  
}  
  
***stu***.sayHi();  
**var *s1*** = **new** *Student*(**'zs'**, 18);  
***s1***.sayHi();

效果如下:



可以看到没有报错,并且都输出了

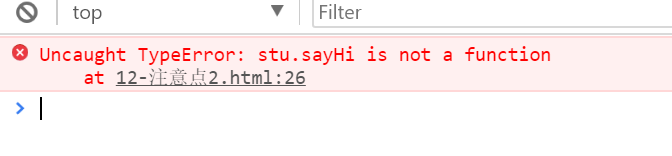
只要是通过prototype.sayHi这种方式添加的方法,就算对象在上面创建的,也没关系,只要在添加方法后使用即可

**而重新给prototype赋值则不行,必须等赋值完后才可使用里面的方法**

同样的代码,只是将sayHi方法添加到原型的方式改变了一下

**function** *Student*(name, age) {  
 *// 一般情况下，对象的属性在构造函数中来设置* **this**.**name** = name;  
 **this**.**age** = age;  
}  
**var *stu*** = **new** *Student*(**'ls'**, 18);  
  
*Student*.**prototype** = {  
 **constructor**: *Student*,  
 sayHi: **function** () {  
 **console**.log(**'sayHi'**);  
 }  
}  
  
***stu***.sayHi();  
**var *s1*** = **new** *Student*(**'zs'**, 18);  
***s1***.sayHi();

效果如下:



17-扩展内置对象.avi

之前的概念全部都是讲解原型的特性和注意事项, 接下来讲解原型的具体应用

原型的作用可以扩展内置对象中的方法

JS提供的内置对象有很多方法,但是在具体的业务中,提供的那些方法可能远不够我们使用

而有些功能可能JS没提供但是业务中经常要使用,所以可以根据具体的业务需求,自己封装一些方法然后添加到内置对象的原型中,从而实现扩展功能

例如: 获取数组中所有偶数的和

这种功能如果在项目中经常用到,而Array又没有提供方法,这时候就可以自己封装一个方法,然后添加到Array的原型中,后面要用就可以直接通过数组对象来使用了,非常方便

代码如下:

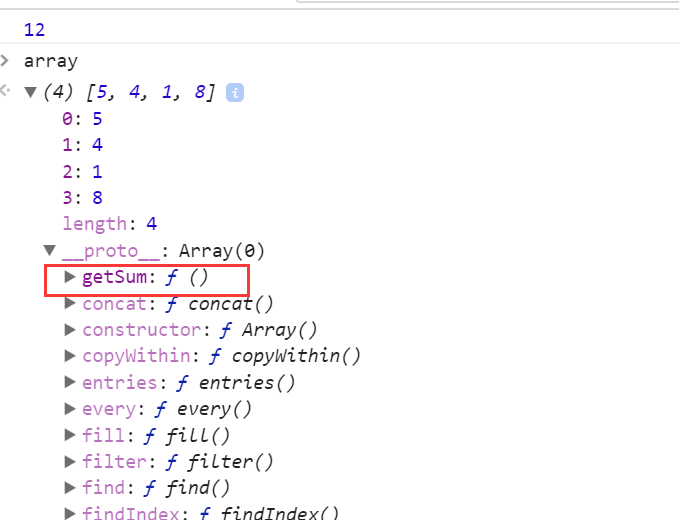
**var *array*** = [5, 4, 1, 8];  
  
Array.**prototype**.getSum = **function** () {  
 *// 求数组中所有偶数的和* **var** sum = 0;  
 **for** (**var** i = 0; i < **this**.**length**; i++) {  
 **if** (**this**[i] % 2 === 0) {  
 sum += **this**[i];  
 }  
 }  
 **return** sum;  
}

**console**.log(***array***.getSum());

效果如下:



打印array,可以看到自己添加的getSum方法在原型对象中



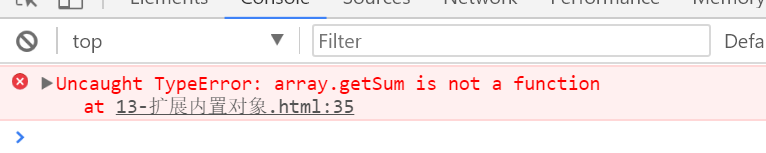
可以看到array对象是在添加原型方法之前创建的,但是由于并没有给prototype重新赋值,所以随时添加随时用

如果给prototype重新赋值

Array.**prototype** = {  
 getSum: **function** () {  
 *// 求数组中所有偶数的和* **var** sum = 0;  
 **for** (**var** i = 0; i < **this**.**length**; i++) {  
 **if** (**this**[i] % 2 === 0) {  
 sum += **this**[i];  
 }  
 }  
 **return** sum;  
 }  
 }  
  
**console**.log(***array***.getSum());

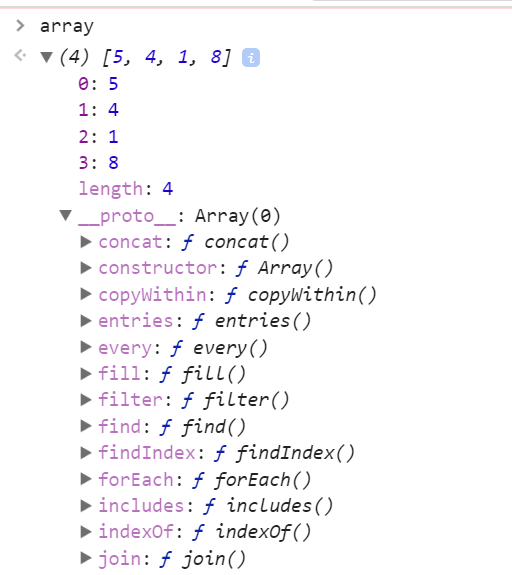
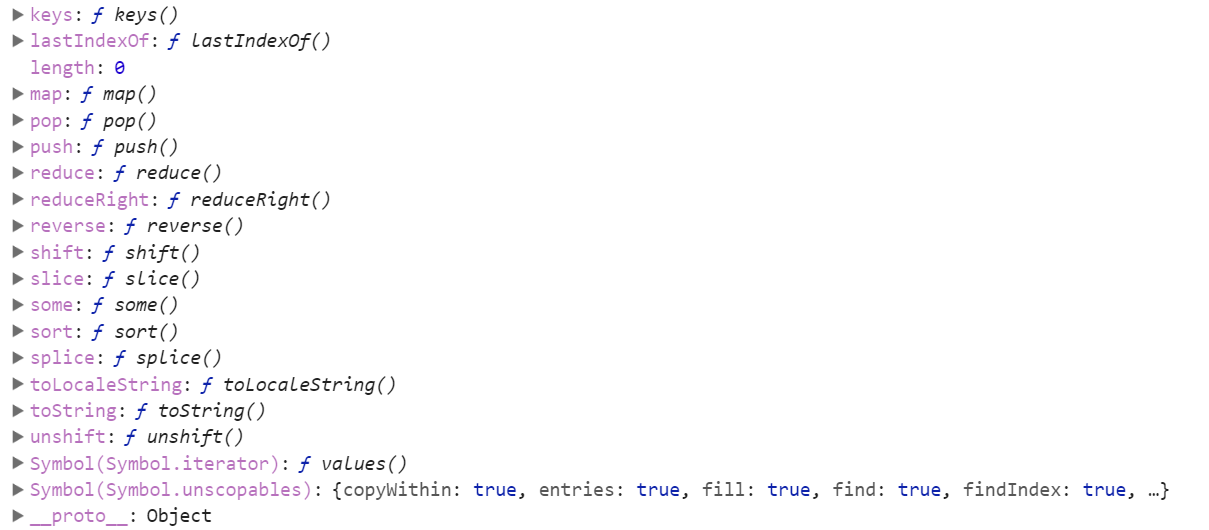
内置对象的原型本身就有很多方法,这样重新赋值会导致所有的方法丢失吗?

效果如下:



报错了

打印array

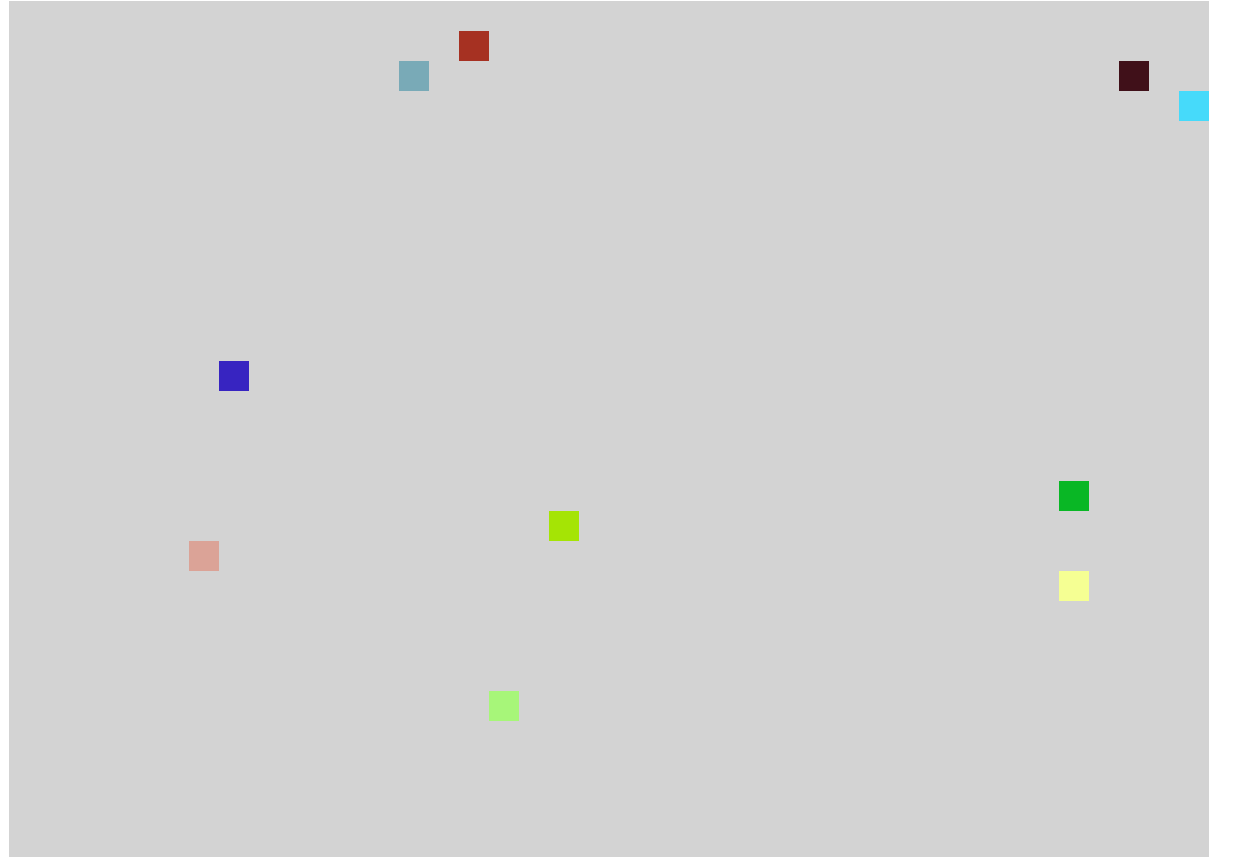
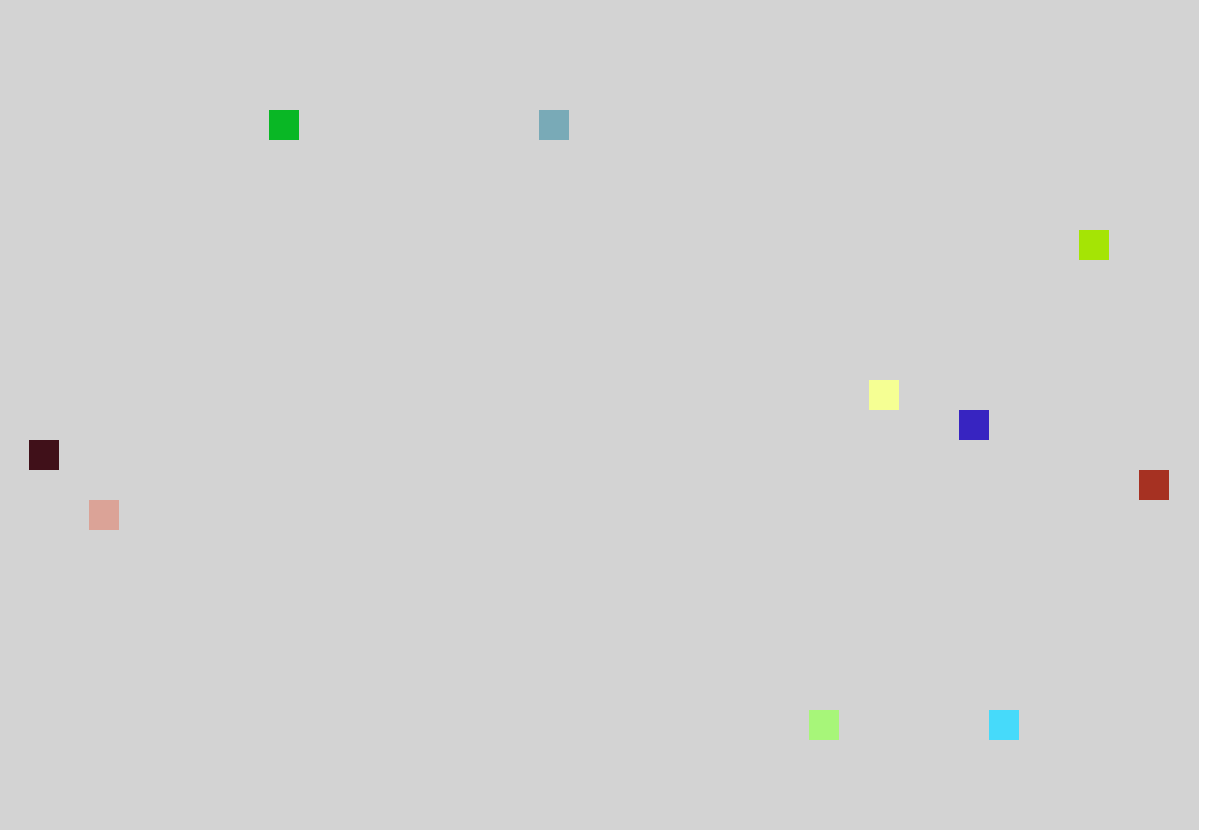
发现不仅内置对象本身的方法没有丢失

而且getSum方法也没添加进去

**切记: JS内置对象的原型不能整体覆盖,只能单独地动态地为原型添加属性或方法,只能通过prototype.方法名 的方式进行添加或修改!不能直接将prototype重新赋值覆盖掉!**

18-随机方块-工具对象.avi

现在通过面向对象的思维方式来完成一个随机生成小方块的案例:

每隔500毫秒随机生成一些方块

首先分析需求

需要在一个大盒子里生成若干个小方块,这些小方块的位置都是随机的

按照以前的面向过程的思维方式,就要开始写生成随机数的代码,然后创建小方块,最后按照随机位置去摆放

但是现在用的是高端的面向对象的思维方式

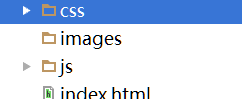
比如生成随机数是一个功能,可以封装成一个工具对象,该工具对象具备生成随机数的功能

比如小方块就是一个对象

在面向对象的思考方式中,所有的东西都应该被抽象成一个对象,这样写代码后期管理起来会非常轻松

在这个项目中功能比较简单,所以只需要两个对象即可,一个用于生产随机数的工具对象,一个就是方块对象

写代码的时候同样使用项目的方式来管理,三层分离



结构代码:

<**div id="container"**>  
   
</**div**>

只需要一个盒子即可

给盒子添加样式,样式代码如下:

**#container** {  
 **width**: 800**px**;  
 **height**: 600**px**;  
 **background-color**: **lightgray**;  
 **position**: **relative**;  
   
}

需要将css引入HTML文件

<**link rel="stylesheet" href="css/style.css"**>

效果如下;

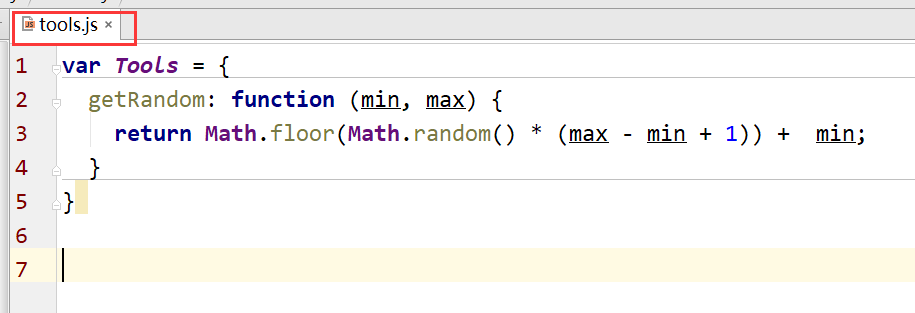


接下来开始写js代码需要注意,不同的对象放到不同的js文件中,方便将来维护,如果某个功能出现问题了,只需要找到相应的对象来解决即可,而分文件来存放不同功能模块的对象最大的好处就是可以快速且准确的定位到相关模块

先做一个生成随机数的对象,提供生成指定范围的随机数方法,工具对象一般习惯首字母大写

**var *Tools*** = {  
 getRandom: **function** (min, max) {  
 **return Math**.floor(**Math**.random() \* (max - min + 1)) + min;  
 }  
}

由于该对象在整个项目中只有一个,所以没必要写构造函数,直接用Object对象即可



**Math**.floor(**Math**.random() \* (max - min + 1)) + min;

该公式可以获取到从min到max的随机数,min可以取到,max也可以取到

最后测试一下随机数是否有问题

*// 测试生成随机数* **console**.log(***Tools***.getRandom(1, 10));









19-随机方块-方块对象.avi

工具对象已经写完了,继续来写第二个对象

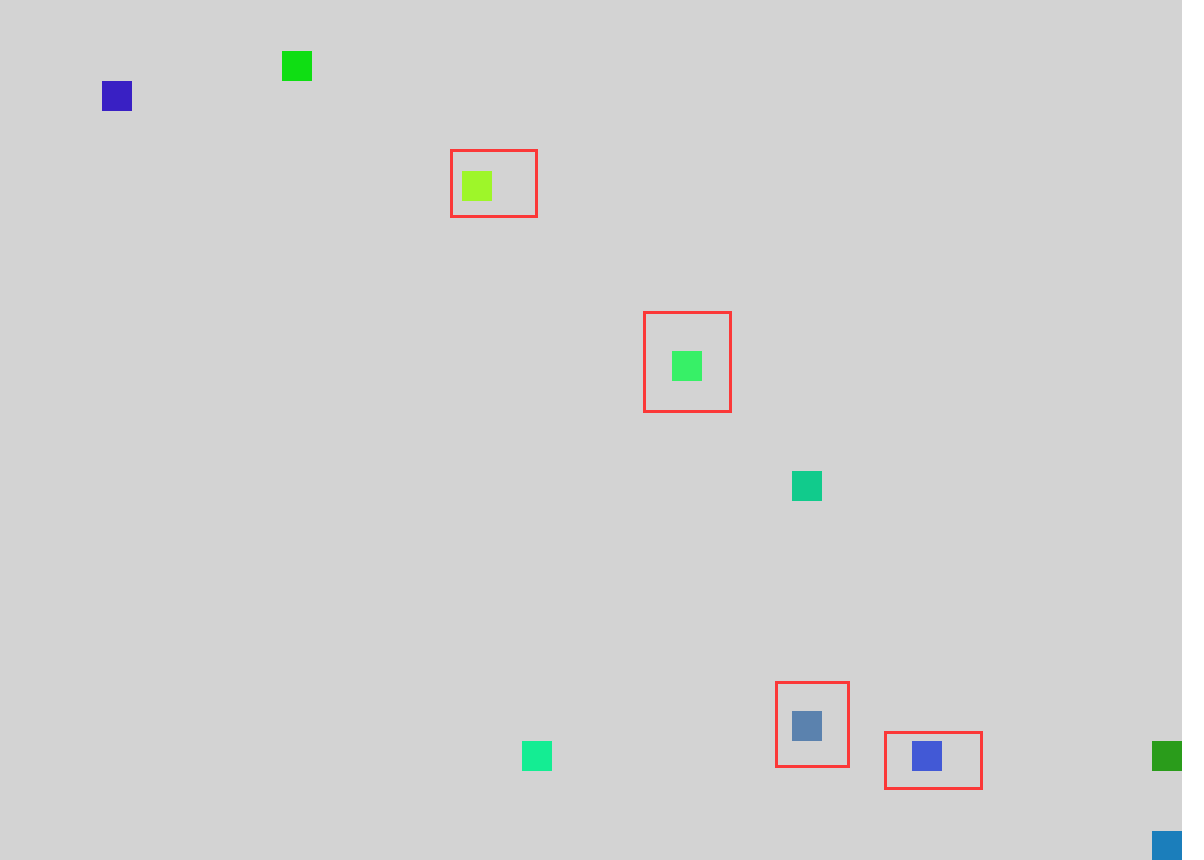
方块对象

页面中要生成很多方块,所以会有很多对象,这个对象就需要用构造函数来创建了

决定写构造函数了之后就要来确定这个对象需要哪些属性和方法

也就是要分析对象的行为和特征

看图分析:



不同的方块背景色不同

方块有大小

方块都有自己的坐标

每个方块都会自己跑到对应的位置去,这是一种行为,也就是一个随机生成坐标并跑过去的方法

分析完了属性和方法,就动手开始写构造函数

先写属性:

**function** *Box*(options) {  
 options = options || {};  
 *// 设置对象的属性* **this**.**backgroundColor** = options.**backgroundColor** || **'red'**;  
 **this**.**width** = options.**width** || 20;  
 **this**.**height** = options.**height** || 20;  
 **this**.**x** = options.**x** || 0;  
 **this**.**y** = options.**y** || 0;  
}

由于分析出来的属性比较多,所以没必要定义很多形参,直接使用一个对象来接收这些属性参数即可

这个对象可能其他人也会用到,所以为了避免出现别人忘了传参数的情况,可以做一个容错处理 options = options || {}

这样做的好处就是即使别人没传参数,我们自己创建一个空对象,至少不会出现报错

继续给对象进行属性赋值,如果传入了这些属性就赋值,没传入也会给一个默认值,提高代码的健壮性,让代码不会轻易报错

属性赋值完了之后需要知道,这个方块最终是会被添加到页面中显示的

也就是说最后会创建元素并给元素设置方块对象的这些属性

在构造函数里创建一个div,创建完毕后就添加到页面中

既然要添加到页面中,所以需要把页面中的container容器传进来,使用一个形参接收

最后还需要给新建的div设置属性,由于构造函数写过多的代码不合适

所以给构造函数再添加一个原型方法init,在该方法中进行属性的设置

在创建完div之后一定要记得调用init方法

**function** *Box*(parent, options) {  
 options = options || {};  
 *// 设置对象的属性* **this**.**backgroundColor** = options.**backgroundColor** || **'red'**;  
 **this**.**width** = options.**width** || 20;  
 **this**.**height** = options.**height** || 20;  
 **this**.**x** = options.**x** || 0;  
 **this**.**y** = options.**y** || 0;  
  
 *// 创建对应的div* **this**.**div** = **document**.createElement(**'div'**);  
 parent.appendChild(**this**.**div**);  
 **this**.**parent** = parent;  
  
 *// 设置div的样式* **this**.init();  
}  
  
*// 初始化div (方块)的样式  
Box*.**prototype**.init = **function** () {  
 **var** div = **this**.**div**;  
 div.**style**.**backgroundColor** = **this**.**backgroundColor**;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**left** = **this**.**x** + **'px'**;  
 div.**style**.**top** = **this**.**y** + **'px'**;  
 *// 脱离文档流* div.**style**.**position** = **'absolute'**}

最后进行测试:

**var *parent*** = **document**.getElementById(**'container'**);  
**var *box1*** = **new** *Box*(***parent***);

测试的时候就直接创建对象,不传入options了,因为做了默认值的处理,不传入就表示使用默认值

效果如下:



20-随机方块-随机生成坐标.avi

方块对象的属性都搞定了,而且也初始化成功了

最后就差方块的行为了,也就是随机生成方块的坐标并设置位置

继续给Box构造函数的原型对象添加方法,随机生成位置

随机生成的位置范围是多少呢?

可以使用容器的宽度除以方块的宽度,算出横向最多可以放多少个盒子

然后生成 0 到 最多可放的盒子数 的随机数

最后乘以方块的宽度即可随机生成方块的横向坐标

纵坐标同理,把宽度换成高度即可

最后设置方块的left和top值

需要注意,想获取容器的宽度,需要使用到parent对象,所以在构造函数中将parent对象保存到对象的属性中 this.parent = parent即可

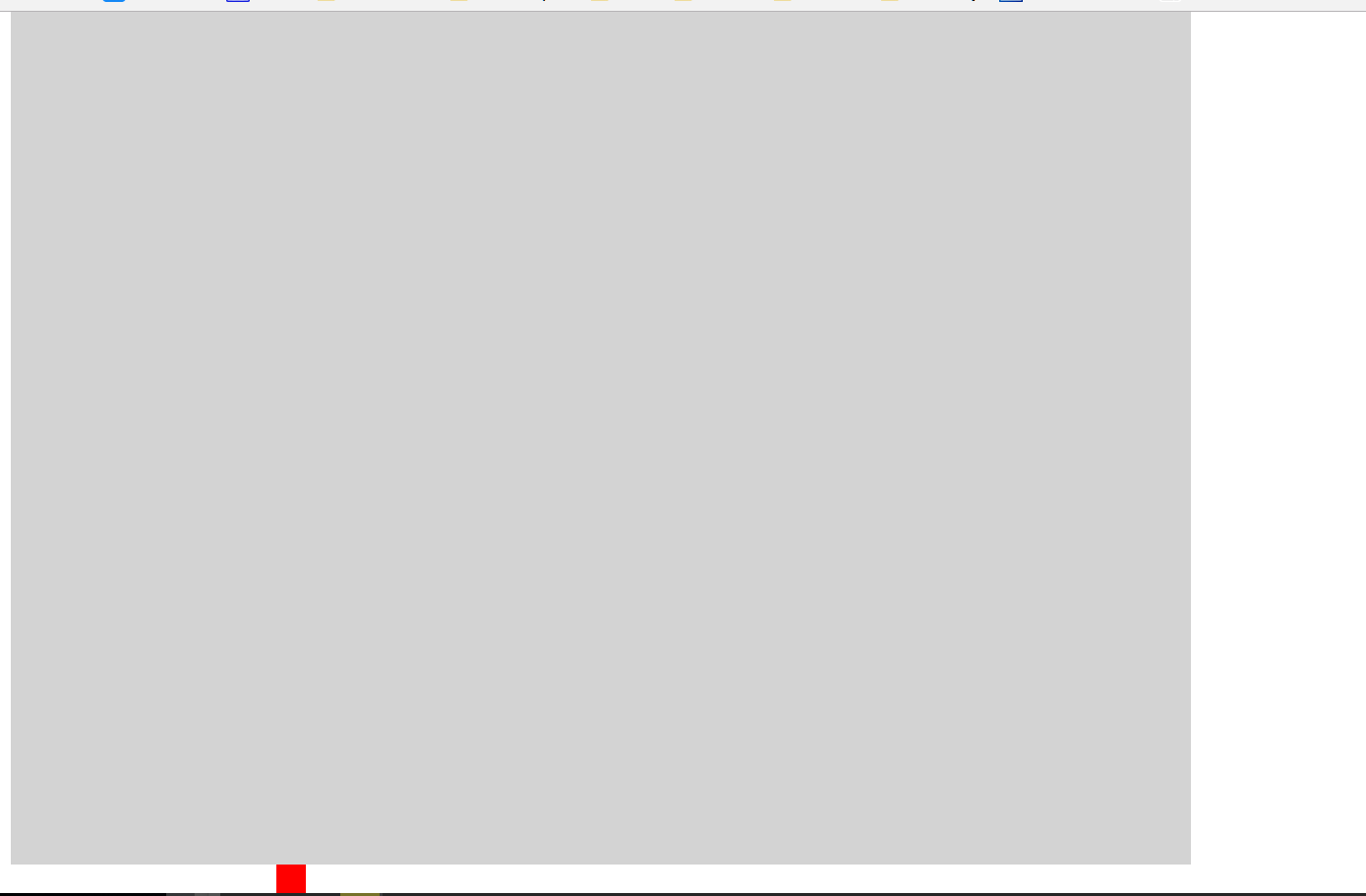
代码如下:

**function** *Box*(parent, options) {  
 options = options || {};  
 *// 设置对象的属性* **this**.**backgroundColor** = options.**backgroundColor** || **'red'**;  
 **this**.**width** = options.**width** || 20;  
 **this**.**height** = options.**height** || 20;  
 **this**.**x** = options.**x** || 0;  
 **this**.**y** = options.**y** || 0;  
  
 *// 创建对应的div* **this**.**div** = **document**.createElement(**'div'**);  
 parent.appendChild(**this**.**div**);  
 **this**.**parent** = parent;  
  
 *// 设置div的样式* **this**.init();  
}  
  
*// 初始化div (方块)的样式  
Box*.**prototype**.init = **function** () {  
 **var** div = **this**.**div**;  
 div.**style**.**backgroundColor** = **this**.**backgroundColor**;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**left** = **this**.**x** + **'px'**;  
 div.**style**.**top** = **this**.**y** + **'px'**;  
 *// 脱离文档流* div.**style**.**position** = **'absolute'**}  
  
*// 随机生成方块的位置  
Box*.**prototype**.random = **function** () {  
 *// 父容器的宽度/方块的宽度 总共能放多少个方块* **var** x = ***Tools***.getRandom(0, **this**.**parent**.**offsetWidth** / **this**.**width**) \* **this**.**width**;  
 **var** y = ***Tools***.getRandom(0, **this**.**parent**.**offsetHeight** / **this**.**height**) \* **this**.**height**;  
  
 **this**.**div**.**style**.**left** = x + **'px'**;  
 **this**.**div**.**style**.**top** = y + **'px'**;  
}

测试随机生成方块的位置:

**var *parent*** = **document**.getElementById(**'container'**);  
 **var *box1*** = **new** *Box*(***parent***);  
*// // 随机生成方块的坐标* ***box1***.random();

效果如下:



发现有时候随机生成的会超出一格,解决方法很简单,将随机数范围的max-1即可

*Box*.**prototype**.random = **function** () {  
 *//container的宽度 / 放块的宽度 = 最多可以放多少个方块* **var** left = ***Tools***.getRandom(0, ***container***.**offsetWidth** / **this**.**width** - 1) \* **this**.**width**;  
 **var** top = ***Tools***.getRandom(0, ***container***.**offsetHeight** / **this**.**height** - 1) \* **this**.**height**;  
 **this**.**div**.**style**.**left** = left + **"px"**;  
 **this**.**div**.**style**.**top** = top + **"px"**;  
}

21-随机方块-调用.avi

通过上面写代码的过程,已经把所有要用的对象都写完了,准备工作也全做完了

万事俱备,只需要最后按照顺序调用对象的方法即可!

按照项目文件管理,JS都不应该写在html中,之前的测试代码仅供测试,上线之前都会删掉

最后实现随机生成方块的功能还需要再写一个js文件

这个js文件就是程序的入口,起名为main.js



js文件的引用是有先后顺序的,一定要注意,测试代码都注释掉

接下来开始完成最后的功能

生成10个方块并随机生成颜色

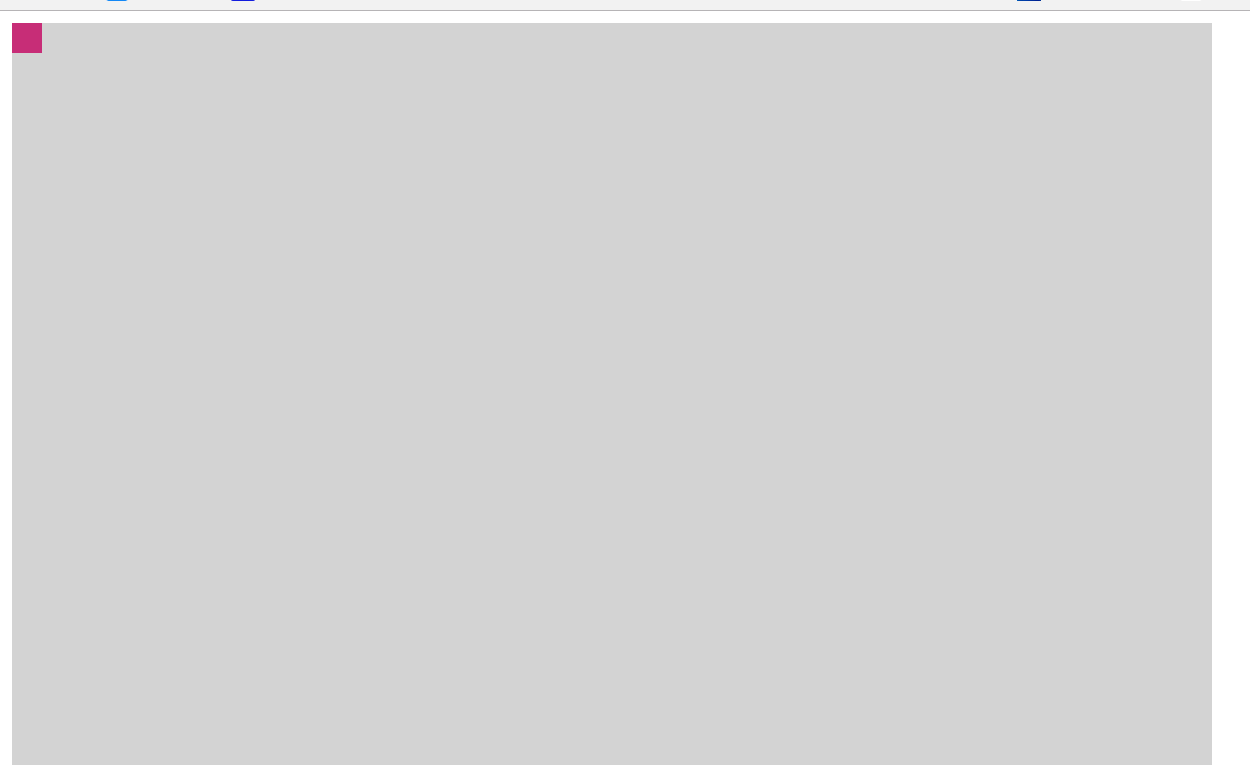
可以通过循环生成10个方块,创建Box对象时需要传入一个容器,所以还需要获取container元素

随机生成颜色rgb值,设置属性时直接使用rgb()的方式拼接字符串即可

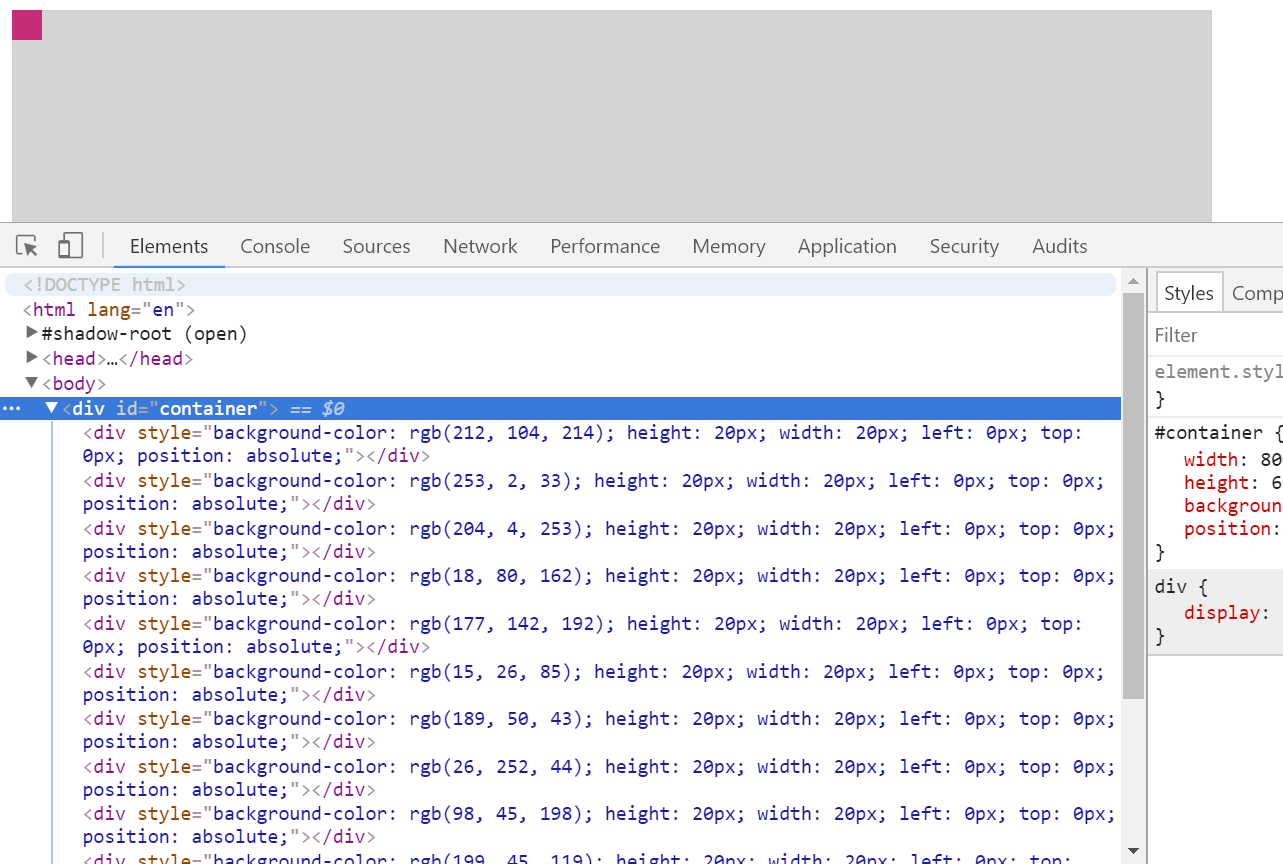
main.js代码如下:

*// 生成10个方块，随机生成颜色  
  
// 获取容器***var *container*** = **document**.getElementById(**'container'**);  
  
**for** (**var *i*** = 0; ***i*** < 10; ***i***++) {  
 **var *r*** = ***Tools***.getRandom(0, 255);  
 **var *g*** = ***Tools***.getRandom(0, 255);  
 **var *b*** = ***Tools***.getRandom(0, 255);  
  
 **var *box*** = **new** *Box*(***container***, {  
 **backgroundColor**: **'rgb('**+ ***r*** +**','**+ ***g*** +**','**+ ***b*** +**')'** });  
}

写完后先测试一下:



虽然只看到了一个方块,其实是10个层叠在一起了,初始位置都在0,0



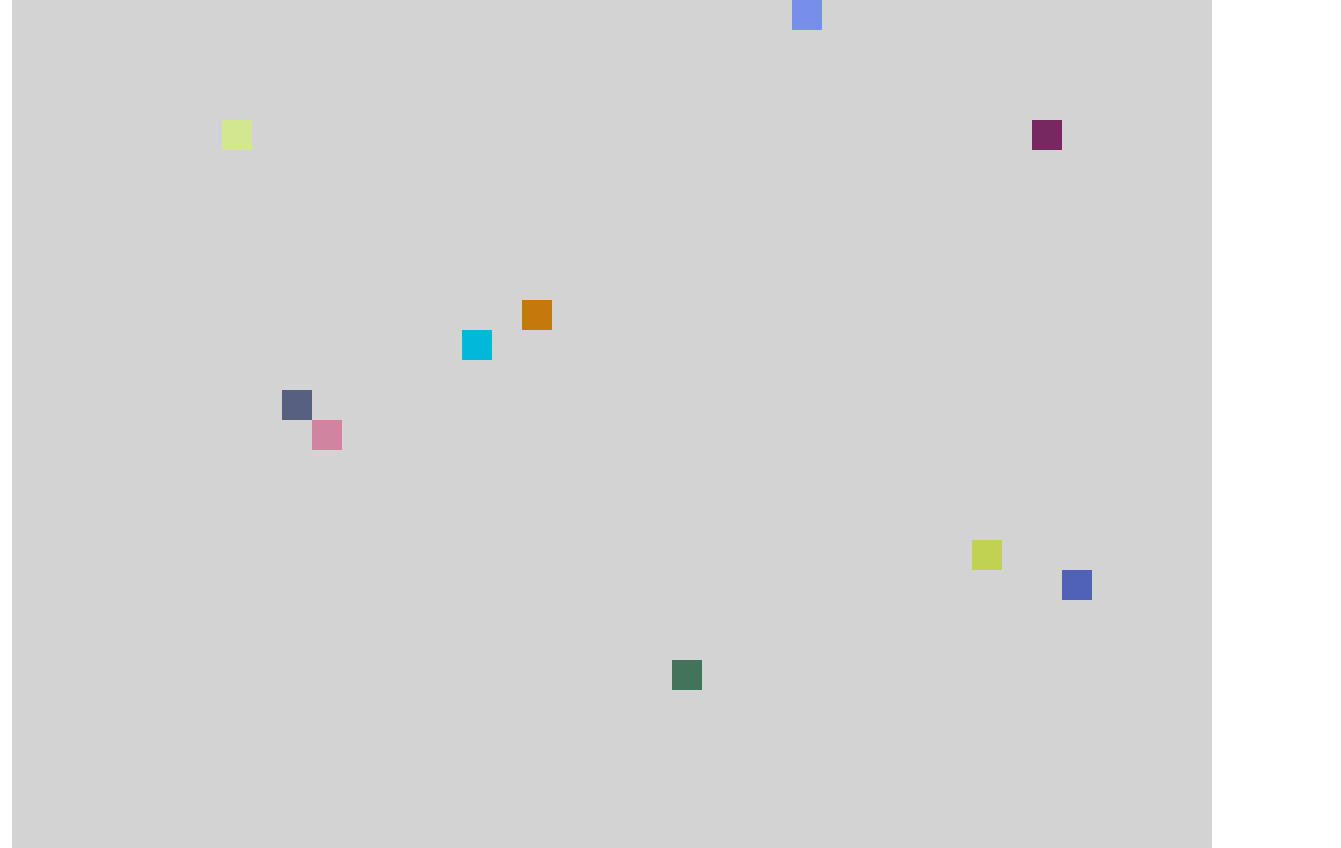
最后就是使用定时器,每500毫秒让所有的方块对象调用一下random方法

需要在定时器中获取所有的方块对象,所以可以在循环创建方块对象时将它们都放入数组中存储,方便管理

代码如下:

*// 生成10个方块，随机生成颜色  
  
// 获取容器***var *container*** = **document**.getElementById(**'container'**);  
  
*// 数组，存储创建的方块对象***var *array*** = [];  
**for** (**var *i*** = 0; ***i*** < 10; ***i***++) {  
 **var *r*** = ***Tools***.getRandom(0, 255);  
 **var *g*** = ***Tools***.getRandom(0, 255);  
 **var *b*** = ***Tools***.getRandom(0, 255);  
  
 **var *box*** = **new** *Box*(***container***, {  
 **backgroundColor**: **'rgb('**+ ***r*** +**','**+ ***g*** +**','**+ ***b*** +**')'** });  
 *// 把创建好的方块对象，添加到数组中* ***array***.push(***box***);  
}  
  
  
*// 设置随机位置，开启定时器*setInterval(*randomBox*, 500);  
  
*// 页面加载完成，先设置随机位置  
randomBox*();  
  
**function** *randomBox*() {  
 *// 随机生成方块的坐标* **for** (**var** i = 0; i < ***array***.**length**; i++) {  
 **var** box = ***array***[i];  
 box.random();  
 }  
}

效果如下:

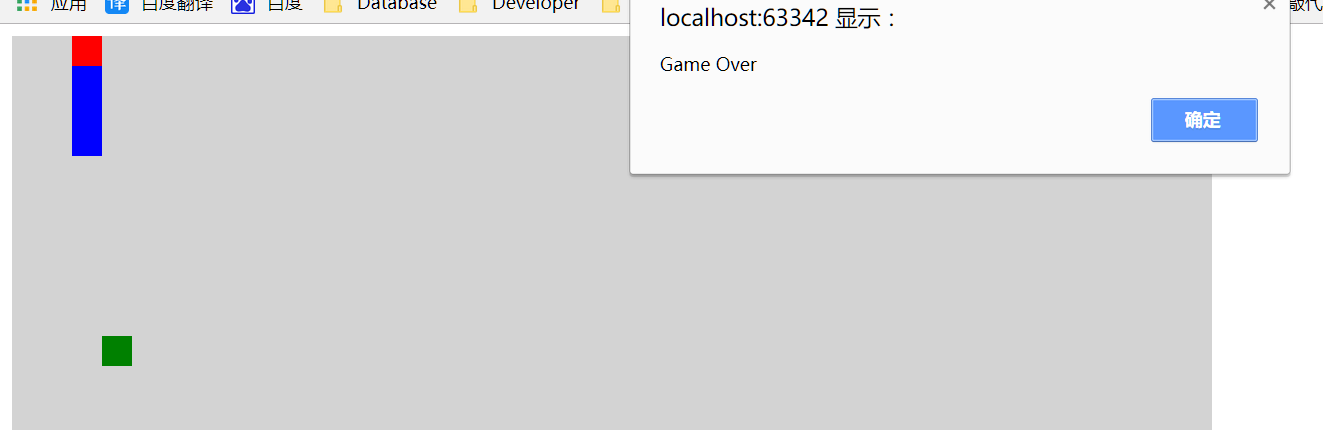


day02

01-项目介绍.avi



撞墙之后:

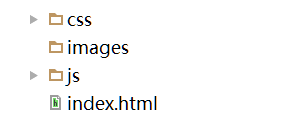


案例目标:

游戏的目的是用来体会JavaScript高级语法的使用,不需要具备抽象对象的能力,使用面向对象的方式分析问题,需要一个漫长的过程,需要看各位的造化,得靠悟性

有的人三两天,有的人三五载,切莫急躁,慢慢体会这个过程就好

开始做之前同样需要搭建结构,这是一个比较大的项目,同样也是三层分离



第一步先写页面结构代码,引入外联样式

<!DOCTYPE **html**>  
<**html lang="en"**>  
<**head**>  
 <**meta charset="UTF-8"**>  
 <**title**>Document</**title**>  
 <**link rel="stylesheet" href="css/style.css"**>  
</**head**>  
<**body**>   
 <**div id="map"**></**div**>  
  
</**body**>  
</**html**>

style.css代码如下:

**#map** {  
 **width**: 800**px**;  
 **height**: 600**px**;  
 **background-color**: **lightgray**;  
 **position**: **relative**;  
}

效果如下:



02-创建食物对象.avi

开始做项目了就得开始分析,哪些东西可以抽象成对象?

蛇

食物

游戏对象

前两个很好理解,一眼就能看到

游戏对象是啥呢?

因为在做游戏的时候,很多业务逻辑需要进行处理,这些业务逻辑如果不用对象来管理就会很混乱

例如,判断蛇吃到了食物,判断蛇撞墙了,判断蛇的方向等等等等,这些都被称为业务逻辑

分析到现在为止,这个案例中,需要用到三个对象

那开始做就肯定挑软柿子捏,从最简单的食物对象入手

分析食物对象的属性:

大小

坐标

背景色

食物的行为:

把自己摆在页面上一个随机坐标 (渲染)

分析完后就开始写代码了,与随机方块案例一样,构造函数中需要接受很多参数,所以使用一个对象批量传进来,并且对对象进行非空处理以及默认值的处理,保证代码的健壮性

**function** *Food*(options) {  
 options = options || {};  
 **this**.**x** = options.**x** || 0;  
 **this**.**y** = options.**y** || 0;  
  
 **this**.**width** = options.**width** || 20;  
 **this**.**height** = options.**height** || 20;  
  
 **this**.**color** = options.**color** || **'green'**;  
}

食物的构造函数设置了属性之后,食物对象还需要具备一个方法,将自己放到地图上去

再给食物的原型对象添加一个渲染方法 render

该方法的作用就是将食物放到地图上,所以需要传入一个地图对象,指定将食物放到某个地图上

*// 渲染  
Food*.**prototype**.render = **function** (map) {  
  
 *// 动态创建div 页面上显示的食物* **var** div = **document**.createElement(**'div'**);  
 map.appendChild(div);  
  
 *// 设置div的样式* div.**style**.**position** = **"absolute"**;  
 div.**style**.**left** = **this**.**x** + **'px'**;  
 div.**style**.**top** = **this**.**y** + **'px'**;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**backgroundColor** = **this**.**color**;  
}

最后写测试代码来测试一下:

*// 测试***var *map*** = **document**.getElementById(**'map'**);  
**var *food*** = **new Food**();  
***food***.render(***map***);

效果如下:



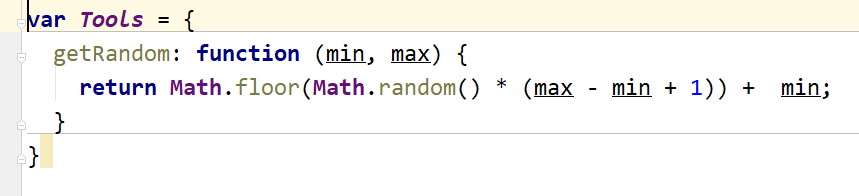
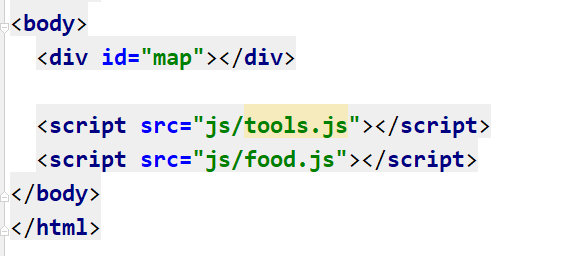
03-随机生成食物.avi

食物已经能够放到页面中了

接下来就是随机生成食物的位置

随机生成食物的位置又要用到随机数了

之前写过这样的一个工具对象,所以可以直接拿过来用了



能够将食物渲染到页面后,需要在渲染之前随机生成坐标x/y

随机生成坐标的算法和之前随机方块案例的算法一毛一样

使用容器的宽度除以食物的宽度算出横向最多可以容纳多少个食物

然后生成 0到最大食物数 的随机数

最后乘以食物的宽度即可随机生成食物的横坐标

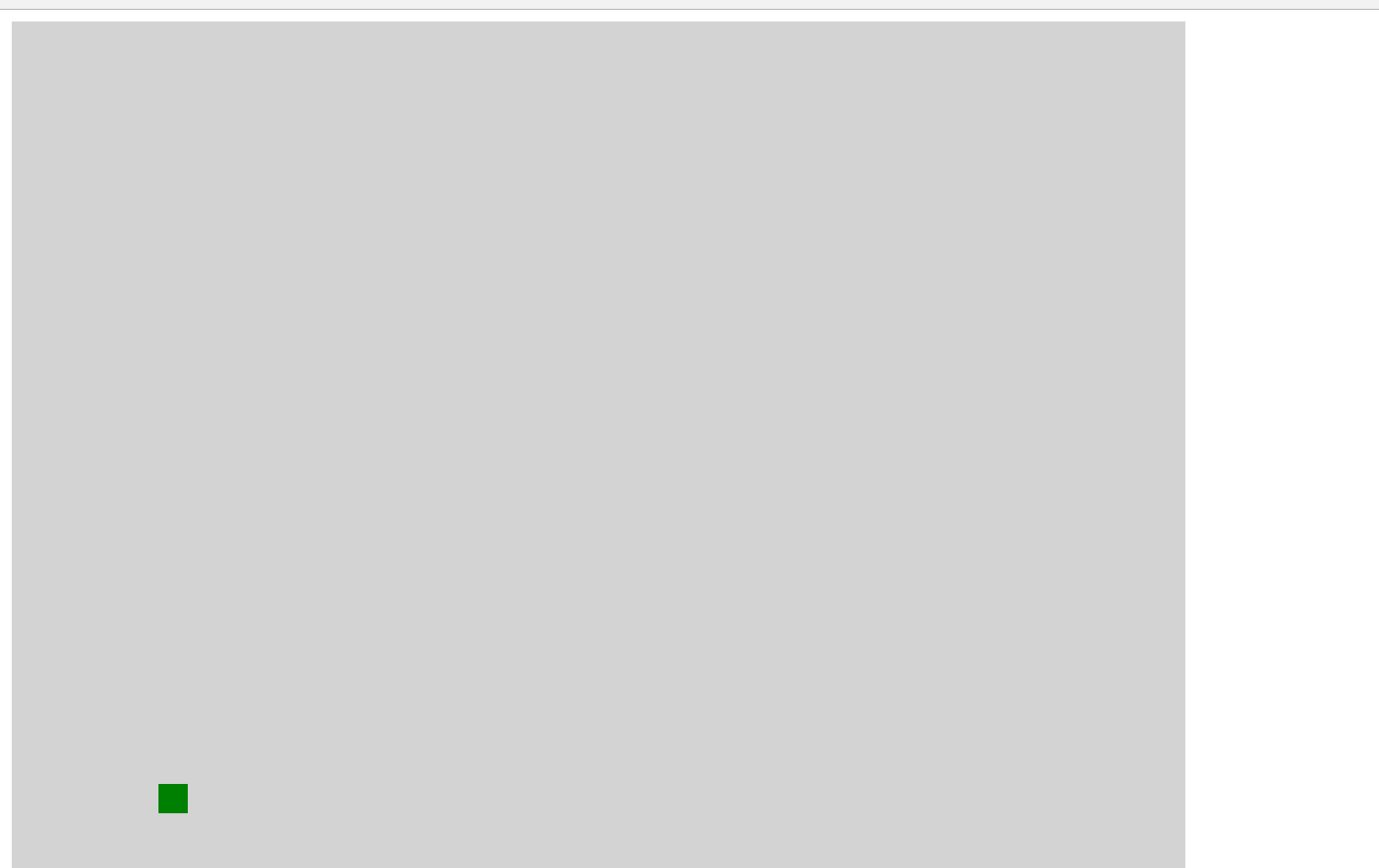
纵坐标同理

为了避免食物超出盒子,还需要将生成的随机数减一,因为工具对象生成的随机数是包括最大值的

代码如下:

*// 渲染  
Food*.**prototype**.render = **function** (map) {  
*// 随机设置x和y的值* **this**.**x** = ***Tools***.getRandom(0, map.**offsetWidth**/**this**.**width** - 1) \* **this**.**width**;  
 **this**.**y** = ***Tools***.getRandom(0, map.**offsetHeight**/**this**.**height** - 1) \* **this**.**height**;  
  
 *// 动态创建div 页面上显示的食物* **var** div = **document**.createElement(**'div'**);  
 map.appendChild(div);  
 *// 设置div的样式* div.**style**.**position** = **"absolute"**;  
 div.**style**.**left** = **this**.**x** + **'px'**;  
 div.**style**.**top** = **this**.**y** + **'px'**;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**backgroundColor** = **this**.**color**;  
}

效果如下:



04-删除食物.avi

食物对象已经完成的差不多了

再看完整版的游戏,在游戏运行的过程中还有哪些和食物相关的东西呢?

蛇吃掉食物时,蛇会变长,而且食物会被删除,并且还会随机生成一个新的食物

所以这个删掉食物并且生成新的食物的操作也是需要有食物对象来封装的

而且分析这个操作会发现,其实就是渲染食物之前,删除之前的食物即可完成删除之前的食物并且生成新的食物

所以确定了代码写在render方法的最前面

接下来的问题就是如何删除页面上的食物呢?

页面上蛇也是盒子,食物也是盒子,如何分辨谁是食物呢?

通过代码确实是不好分辨的,不过可以通过一个简单的方式解决这个问题

在每次创建食物盒子的时候,把这个对象记录起来

每次调用render方法生成新食物的时候,先把之前记录的食物给删掉就可以了

删掉食物这个操作不属于渲染食物,所以再写一个函数来调用

代码如下:

*//定义一个变量 用于记录上一次食物元素***var *lastFood***;  
  
*// 渲染  
Food*.**prototype**.render = **function** (map) {  
 *// 删除之前创建的食物  
 remove*();  
  
*// 随机设置x和y的值* **this**.**x** = ***Tools***.getRandom(0, map.**offsetWidth** / **this**.**width** - 1) \* **this**.**width**;  
 **this**.**y** = ***Tools***.getRandom(0, map.**offsetHeight** / **this**.**height** - 1) \* **this**.**height**;  
  
 *// 动态创建div 页面上显示的食物* **var** div = **document**.createElement(**'div'**);  
 map.appendChild(div);  
  
 *//创建完食物就记录下来 下次创建食物时方便删掉* ***lastFood*** = div;  
  
 *// 设置div的样式* div.**style**.**position** = **"absolute"**;  
 div.**style**.**left** = **this**.**x** + **'px'**;  
 div.**style**.**top** = **this**.**y** + **'px'**;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**backgroundColor** = **this**.**color**;  
}  
  
**function** *remove*() {  
 *//第一次进来时lastFood还是undefined 所以要进行非空校验 如果有就移除掉* **if** (***lastFood***) {  
 ***lastFood***.parentNode.removeChild(***lastFood***);  
 }  
}

remove方法需要注意,第一次调用时lastFood是undefined,所以需要进行非空校验

第二次调用时就会把第一个食物给删掉,并且生成一个新的随机食物在页面上

现在的测试效果和之前没啥区别,但是将来蛇吃食物的时候就有大用处了



05-自调用函数，避免命名冲突.avi

代码写到现在,一直是使用面向对象的思想在写代码

不同模块有不同的js文件

但是所有的js文件都引用到了html中执行

代码都写在全局作用域,这样很容出现命名冲突

例如在food.js中用了一个变量叫index,在tools.js中也有一个变量叫index,这时候后执行的就会把先执行的覆盖掉

不同模块之间很容易出现干扰,虽然分模块来写代码可维护性强

但是将来肯定也不希望出现变量名都不知道怎么起的情况

所以需要避免这个问题,尽量不要在全局作用域写代码

既然不在全局作用域写代码,那就只能在函数作用域写代码了

*// 所有的js文件中书写代码，都是全局作用域***function** *fn*() {  
 *// 局部作用域* **var** n = 1;  
}

*fn*();

定义一个fn函数,把代码全部放到fn函数中,就不会影响到全局作用域了

但是也有问题,如果其他地方也有叫fn的函数就又会冲突了

所以只能用匿名函数:

**function**() {  
 **var** n = 1;  
}

可是匿名函数没法调用

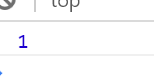
不过js提供了一种匿名函数自调用的方式

调用函数就是在函数体后面加小括号()

把整个匿名函数用小括号括起来就是一个函数体,再到后面加一个小括号即可调用

(**function**() {  
 **var** n = 1;  
 **console**.log(n);  
})();

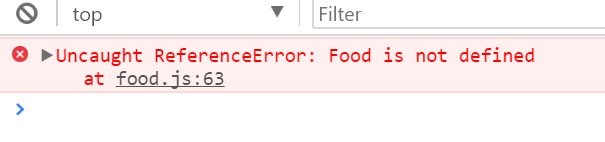
效果如下:



这样一来,将所有的代码全部放到这个匿名函数中即可完美的解决全局作用域命名冲突的问题!

代码如下:

*// 所有的js文件中书写代码，都是全局作用域  
  
//function fn() {  
// // 局部作用域  
// var n = 1;  
//}  
// fn();  
//   
//  
// 自调用函数 -- 开启一个新的作用域，避免命名冲突*(**function** () {  
 *// 局部作用域* **var** position = **'absolute'**;  
 *// 记录上一次创建的食物，为删除做准备* **var** lastFood;  
  
 **function** *Food*(options) {  
 options = options || {};  
 **this**.**x** = options.**x** || 0;  
 **this**.**y** = options.**y** || 0;  
  
 **this**.**width** = options.**width** || 20;  
 **this**.**height** = options.**height** || 20;  
  
 **this**.**color** = options.**color** || **'green'**;  
 }  
  
 *// 渲染  
 Food*.**prototype**.render = **function** (map) {  
 *// 删除之前创建的食物  
 remove*();  
  
 *// 随机设置x和y的值* **this**.**x** = ***Tools***.getRandom(0, map.**offsetWidth** / **this**.**width** - 1) \* **this**.**width**;  
 **this**.**y** = ***Tools***.getRandom(0, map.**offsetHeight** / **this**.**height** - 1) \* **this**.**height**;  
  
 *// 动态创建div 页面上显示的食物* **var** div = **document**.createElement(**'div'**);  
 map.appendChild(div);  
  
 lastFood = div;  
  
 *// 设置div的样式* div.**style**.**position** = position;  
 div.**style**.**left** = **this**.**x** + **'px'**;  
 div.**style**.**top** = **this**.**y** + **'px'**;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**backgroundColor** = **this**.**color**;  
 };  
  
 **function** *remove*() {  
 **if** (lastFood) {  
 lastFood.parentNode.removeChild(lastFood);  
 }  
 }  
  
})();  
  
  
*// 测试***var *map*** = **document**.getElementById(**'map'**);  
**var *food*** = **new** *Food*();  
***food***.render(***map***);  
  
  
看测试结果:



第63行报错了!

Food未定义

说明找不到Food这个构造函数了

因为使用了自调用函数,把构造函数也放到局部作用域了

所以只能在内部访问,外面无法访问了

怎么办呢?

把需要对外暴露的东西单独对外暴露即可

在自调用函数的末尾加上一句

*//把Food构造函数对外暴露 让外部可以访问***window**.**Food** = *Food*;

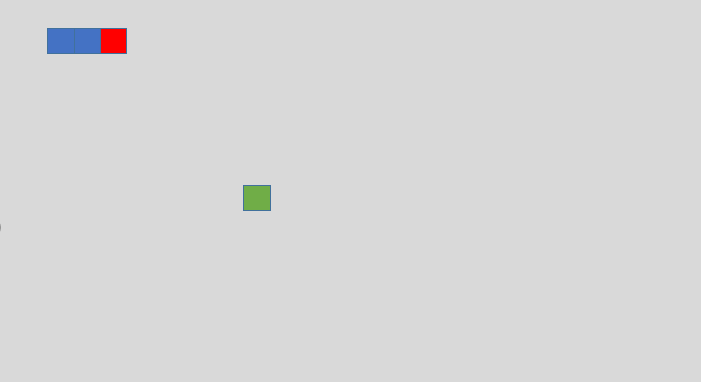
因为window对象是顶级对象,在使用window对象身上的属性或方法时可以省略window

所以这样做其他地方就都可以直接访问Food了



06-蛇对象.avi

接下来要分析蛇对象所具备的特征和行为



蛇有身体,也就是说有身体属性,而身体是由好多个方块组成,最前面红色的方块就是蛇头,后面蓝色的方块就是蛇节

每个蛇节有大小

蛇有移动的方向,所以方向也是一个属性

所以总结如下:

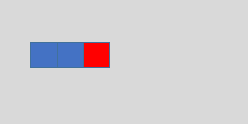
- width 蛇节的宽度 默认20

- height 蛇节的高度 默认20

- body 数组，蛇的头部和身体，第一个位置是蛇头

- direction 蛇运动的方向 默认right 可以是 left top bottom

其中body数组的每个蛇节还需要记录自己所在的位置,例如下图,最后一个蛇节的位置就是距离左侧一个蛇节的宽度,只需要记录为1即可,距离顶部就是两个蛇节的宽度,只需要记录为2即可,按照这种记录方式记录每个蛇节在盒子中的位置即可 将来要使用位置时只用拿该坐标乘一个蛇节的宽度即可



以上是蛇的属性,蛇和食物一样,最终也要被渲染到页面上显示,所以也需要一个渲染方法render

分析完毕后开始写构造函数,之前学习了自调用函数,可以避免全局作用域下变量名冲突的问题

所以现在也一样在自调用函数中写构造函数,最后将构造函数暴露到全局

(**function** () {  
 **function** *Snake*(options) {  
 options = options || {};  
 *// 蛇节 的大小* **this**.**width** = options.**width** || 20;  
 **this**.**height** = options.**height** || 20;  
 *// 蛇移动的方向* **this**.**direction** = options.**direction** || **'right'**;  
 *// 蛇的身体(蛇节) 第一个元素是蛇头* **this**.**body** = [  
 {**x**: 3, **y**: 2, **color**: **'red'**},  
 {**x**: 2, **y**: 2, **color**: **'blue'**},  
 {**x**: 1, **y**: 2, **color**: **'blue'**}  
 ];  
 }  
 *// 暴露构造函数给外部* **window**.**Snake** = *Snake*;  
})()

接下来完成渲染方法render

完成渲染操作其实就是将蛇的每个蛇节显示到页面中,也就是需要遍历body属性

取出每一个蛇节

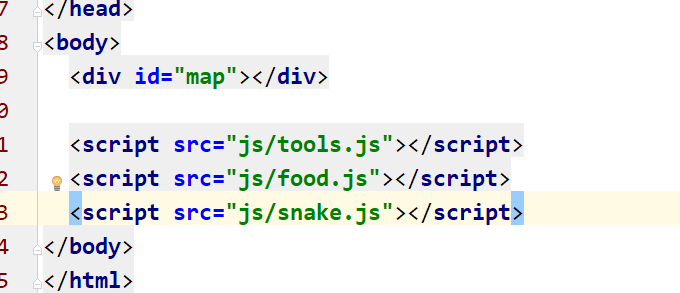
创建div后将其添加到页面中

还需要给div设置样式(每个蛇节在什么位置,什么大小,什么颜色)

Snake.**prototype**.render = **function** (map) {  
 *// 把每一个蛇节渲染到地图上* **for** (**var** i = 0, len = **this**.**body**.**length**; i < len; i++) {  
 *// 蛇节* **var** object = **this**.**body**[i];  
 *//* **var** div = **document**.createElement(**'div'**);  
 map.appendChild(div);  
 *// 设置样式* div.**style**.**position** = **'absolute'**;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**left** = object.**x** \* **this**.**width** + **'px'**;  
 div.**style**.**top** = object.**y** \* **this**.**height** + **'px'**;  
 div.**style**.**backgroundColor** = object.**color**;  
 }  
}

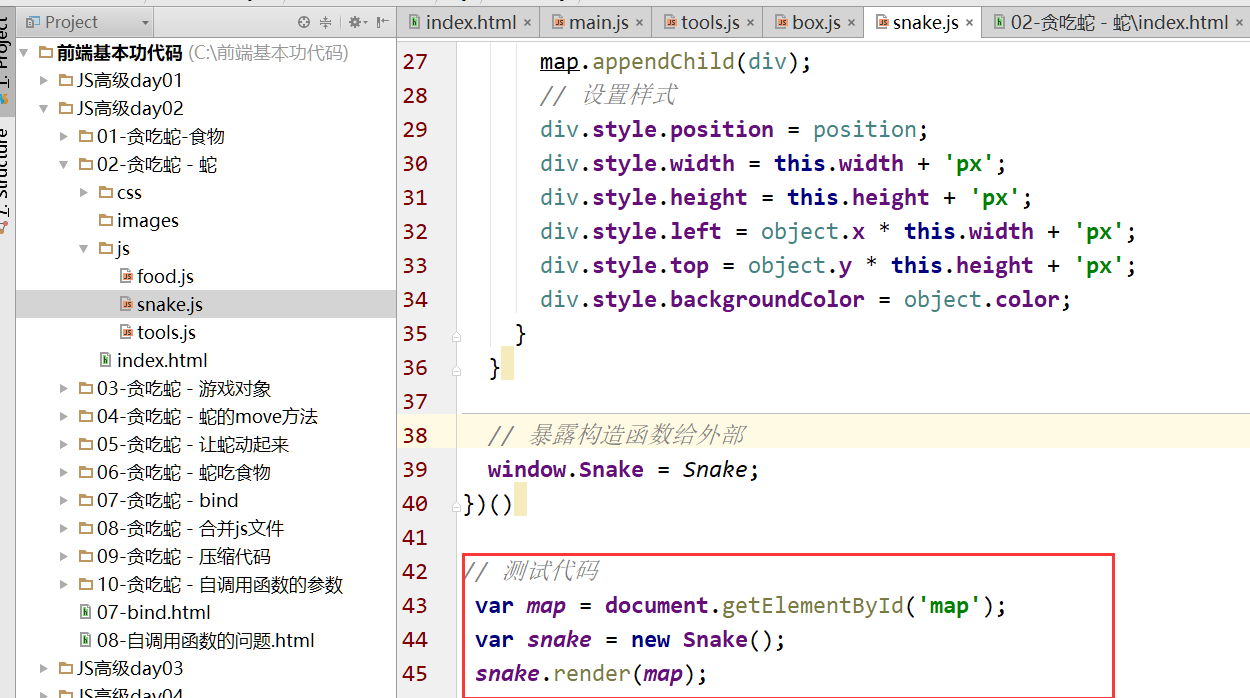
写完之后一定要进行测试:

引入snake.js

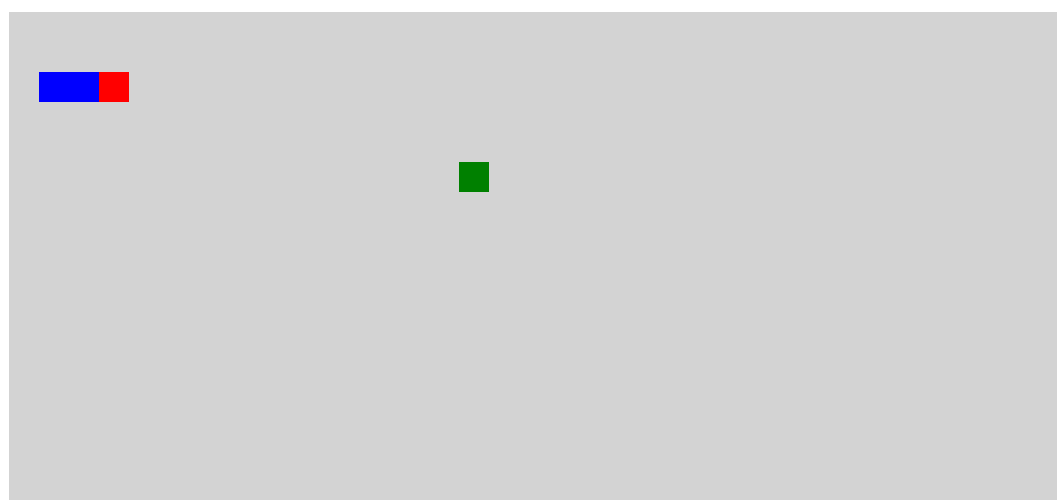


在snake.js中写测试代码测试自己:

**var *map*** = document.getElementById(**'map'**);  
**var *snake*** = **new Snake**();  
***snake***.render(***map***);



效果如下:



07-游戏对象.avi

任何事或物都可以抽象成对象,最后再来一个控制逻辑的游戏对象

游戏对象的功能就是用于处理整个游戏的所有业务逻辑(例如判断蛇吃食物,蛇的移动,蛇方向的改变等等)

先从最基本的逻辑开始,就是测试代码做的事情,将蛇和食物创建出来,然后渲染到页面上

其实就是在游戏对象的构造函数中来创建蛇和食物对象,然后提供一个开始游戏的方法

开始游戏时,使用各自对象的渲染方法将页面渲染出来

游戏对象的构造函数:

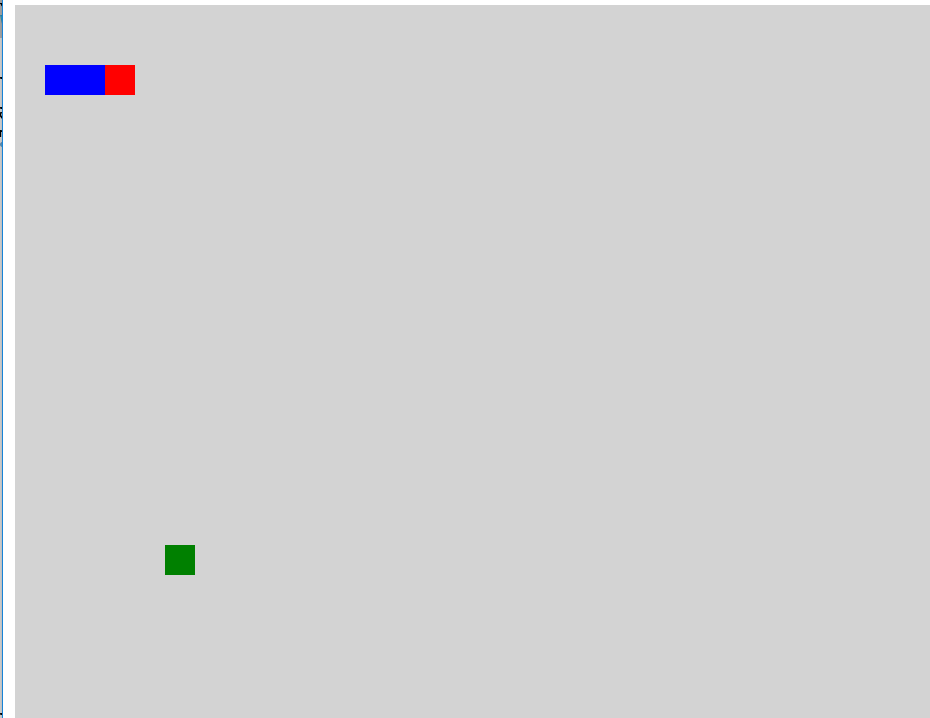
(**function** () {  
 **function** *Game*(map) {  
 **this**.**food** = **new** *Food*();  
 **this**.**snake** = **new Snake**();  
 **this**.**map** = map;  
 }  
  
 *Game*.**prototype**.start = **function** () {  
 *// 1 把蛇和食物对象，渲染到地图上* **this**.**food**.render(**this**.**map**);  
 **this**.**snake**.render(**this**.**map**);  
 *// 2 开始游戏的逻辑* }  
  
 *// 暴露构造函数给外部* **window**.**Game** = *Game*;  
})();

现在基本功能写完了,后面就是继续写开始游戏后的业务逻辑

先测试

*// 测试***var *map*** = **document**.getElementById(**'map'**);  
**var *game*** = **new Game**(***map***);  
***game***.start();

效果如下:和之前一样,也没问题,但是现在就把之前的测试代码全部封装起来了,将来只用创建一个Game对象,调用start方法就可以开始了!



08-蛇的move方法.avi

Game对象现在已经可以将蛇和食物渲染到页面上了

下一步就是让蛇动起来

按方向键改变蛇的方向

蛇吃到食物后让蛇变长

最后碰到墙壁后GAMEOVER

先分析让蛇移动的功能:



让蛇移动其实就是,蛇头向右走一个蛇节的宽度,然后第一个蛇节移动到蛇头之前的位置,第二个蛇节移动到第一个蛇节之前的位置

知道这个原理后就开始继续加移动的功能

注意: 让蛇移动的功能应该添加到蛇对象中!

倒序遍历body属性,获取所有蛇节(不要蛇头,for条件表达式>0)

将当前蛇节的x赋值为前一个蛇节的x

当前蛇节的y赋值为前一个蛇节的y

下面再来控制蛇头的移动,蛇头的移动需要分析,按方向键会使蛇头移动的方向发生改变,那么需不需要绑定键盘按下事件呢?

答案是不需要在蛇对象中做这些事情,因为绑定事件控制方向这个操作应该由game对象来完成

蛇对象只需要知道,根据自身的direction属性来判断蛇往哪个方向走即可

将来在game对象中绑定键盘按下事件,按下上下左右时也只需要修改蛇对象的direction属性即可

分析完后代码实现非常简单

取出蛇头,判断蛇移动的方向

如果往右走就是x+=1

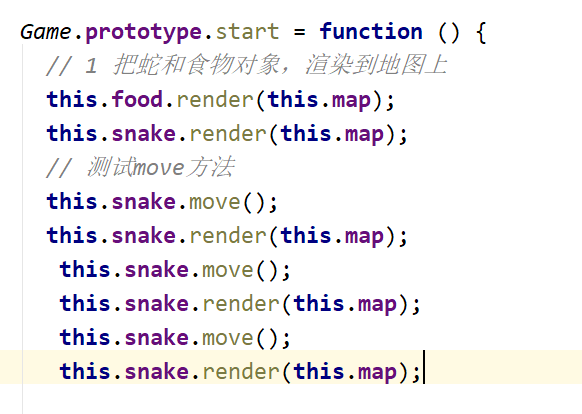
往左走就是x-=1

往上走就是y-=1

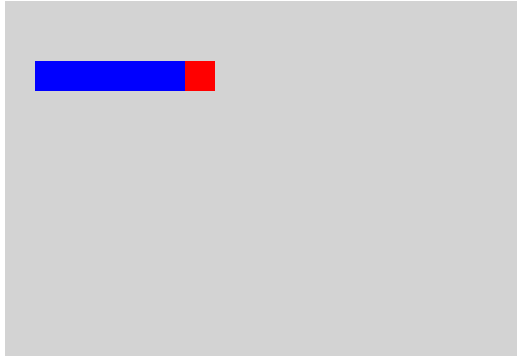
往下走就是y+=1

*// 控制蛇移动的方法  
Snake*.**prototype**.move = **function** () {  
  
 *// 控制蛇的身体移动（当前蛇节 到 上一个蛇节的位置）* **for** (**var** i = **this**.**body**.**length** - 1; i > 0; i--) {  
 **this**.**body**[i].**x** = **this**.**body**[i - 1].**x**;  
 **this**.**body**[i].**y** = **this**.**body**[i - 1].**y**;  
 }  
 *// 控制蛇头的移动(必须先移动蛇节 再移动头，不然会出现问题 画图思考就知道了)  
 // 判断蛇移动的方向* **var** head = **this**.**body**[0];  
 **switch**(**this**.**direction**) {  
 **case 'right'**:   
 head.**x** += 1;  
 **break**;  
 **case 'left'**:   
 head.**x** -= 1;  
 **break**;  
 **case 'top'**:  
 head.**y** -= 1;  
 **break**;  
 **case 'bottom'**:  
 head.**y** += 1;  
 **break**;  
 }  
  
}

进行测试: 注意,由于move方法只是改变了每个蛇节的位置,但并不会对页面造成改变,所以需要再调用一次render方法让页面重新渲染(重绘)



效果如下:



经过三次移动后蛇竟然变长了!

原因也很简单,因为每次move方法都是将蛇往前挪一格,然后调用了render方法按照新的位置重新创建蛇节并渲染到页面,但是! 没有将前一次渲染的蛇删除,所以并不是变成了,而是每次移动一格,并且没有删掉之前的蛇,导致三条蛇叠在一起!

所以就和食物一样,每次渲染之前一定要先把之前的蛇删掉

其实每一次移动都是将蛇对象的位置改变,然后根据新的位置重新创建DOM对象并渲染

09-删除蛇.avi

接下来就是再来创建一个删除蛇的方法

既然要删除’蛇’,这里的蛇并非销毁蛇对象,而是把页面上的蛇节全部从页面上移除掉! 也就是说是将页面上的标签移除掉!

那么就需要在创建每个蛇节的时候将每一节都存储起来,多个对象所以需要用数组来存储

在render函数渲染时会创建蛇节对象,所以创建一个div就存储一个div

*// 记录之前创建的蛇***var** elements = [];

*Snake*.**prototype**.render = **function** (map) {  
 *// 删除之前创建的蛇  
 remove*();  
 *// 把每一个蛇节渲染到地图上* **for** (**var** i = 0, len = **this**.**body**.**length**; i < len; i++) {  
 *// 蛇节* **var** object = **this**.**body**[i];  
 *//* **var** div = **document**.createElement(**'div'**);  
 map.appendChild(div);  
  
 *// 记录当前蛇* elements.push(div);  
  
 *// 设置样式* div.**style**.**position** = position;  
 div.**style**.**width** = **this**.**width** + **'px'**;  
 div.**style**.**height** = **this**.**height** + **'px'**;  
 div.**style**.**left** = object.**x** \* **this**.**width** + **'px'**;  
 div.**style**.**top** = object.**y** \* **this**.**height** + **'px'**;  
 div.**style**.**backgroundColor** = object.**color**;  
 }  
}

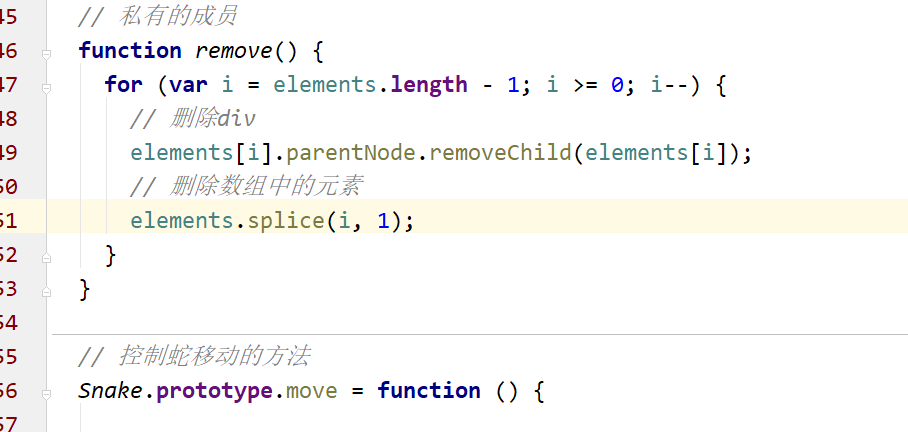
接下来就要开始写remove方法了

但是这个方法不希望被外界访问,因为这个方法威力太大了,一旦误调用将会直接导致游戏无法进行下去

所以这样的方法应该尽可能的私有化

就是将方法定义在snake.js的自调用函数内部,只要不设置在Snake构造函数的原型中都可以

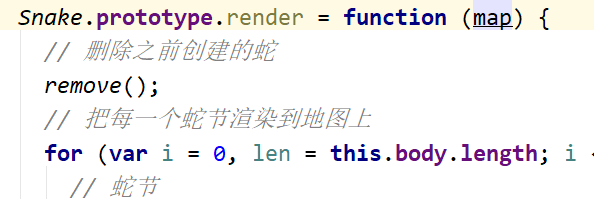
放到自调用函数中,写一个普通的函数



代码如下: 遍历elements并全部从页面中移除掉

*// 私有的成员***function** *remove*() {  
 **for** (**var** i = elements.**length** - 1; i >= 0; i--) {  
 *// 删除div* elements[i].parentNode.removeChild(elements[i]);  
 *// 删除数组中的元素* elements.splice(i, 1);  
 }  
}

封装好remove函数后再render顶部调用即可,每次都先删除蛇再添加



测试:



调用一下move再调用render即可让蛇移动一格



仅仅只是测试一下,不局限于形式

10-蛇移动.avi

基本工作准备完成后,接下来重点就是做game对象中的游戏逻辑了

首先要让蛇动起来

通过键盘控制蛇的方向

蛇遇到食物后需要变长并且重新生成食物

蛇碰到边界后GAMEOVER

这就是整个游戏大概的逻辑,当然完成后还可以加一些其他的判断逻辑(例如蛇吃到自己也是游戏结束)

这里只做以上功能

第一步: 让蛇动起来

蛇对象中已经有了move方法,该方法调用一次蛇会动一小格

所以让蛇不停的动就是来个定时器,在指定时间间隔后不断的调用move方法即可

在game对象的start方法中来完成这些逻辑,但是把这些代码全写在一个start方法中就比较乱了,所以分工明确后就再封装成一个个的私有函数在start方法中调用

*Game*.**prototype**.start = **function** () {  
 *// 1 把蛇和食物对象，渲染到地图上* **this**.**food**.render(**this**.**map**);  
 **this**.**snake**.render(**this**.**map**);  
  
 *// 2 开始游戏的逻辑  
 // 2.1 让蛇移动起来  
 // 2.2 当蛇遇到边界游戏结束  
 runSnake*();  
 *// 2.3 通过键盘控制蛇移动的方向*  
 *// 2.4 当蛇遇到食物*}

接下来就是封装一个私有函数runSnake即可

注意: 在runSnake函数中需要使用定时器,在定时器的回调函数中this指向的是window对象,所以无法获取到game对象,也就无法获取到snake

想获取当前的game对象该怎么办呢?

用一种傻瓜的方式,在构造函数中,将this赋值给一个全局变量,例如that

将来想在私有函数中使用this时就可以直接用that即可

**var** that; *// 记录游戏对象***function** *Game*(map) {  
 **this**.**food** = **new** *Food*();  
 **this**.**snake** = **new Snake**();  
 **this**.**map** = map;  
 that = **this**;  
}

在runSnake函数中需要隔一段时间调用一次move后再执行render渲染

还需要进行判断是否到达边界,一旦到达边界就要清理定时器并提示结束游戏

判断到达边界的条件就是x和y不得小于0

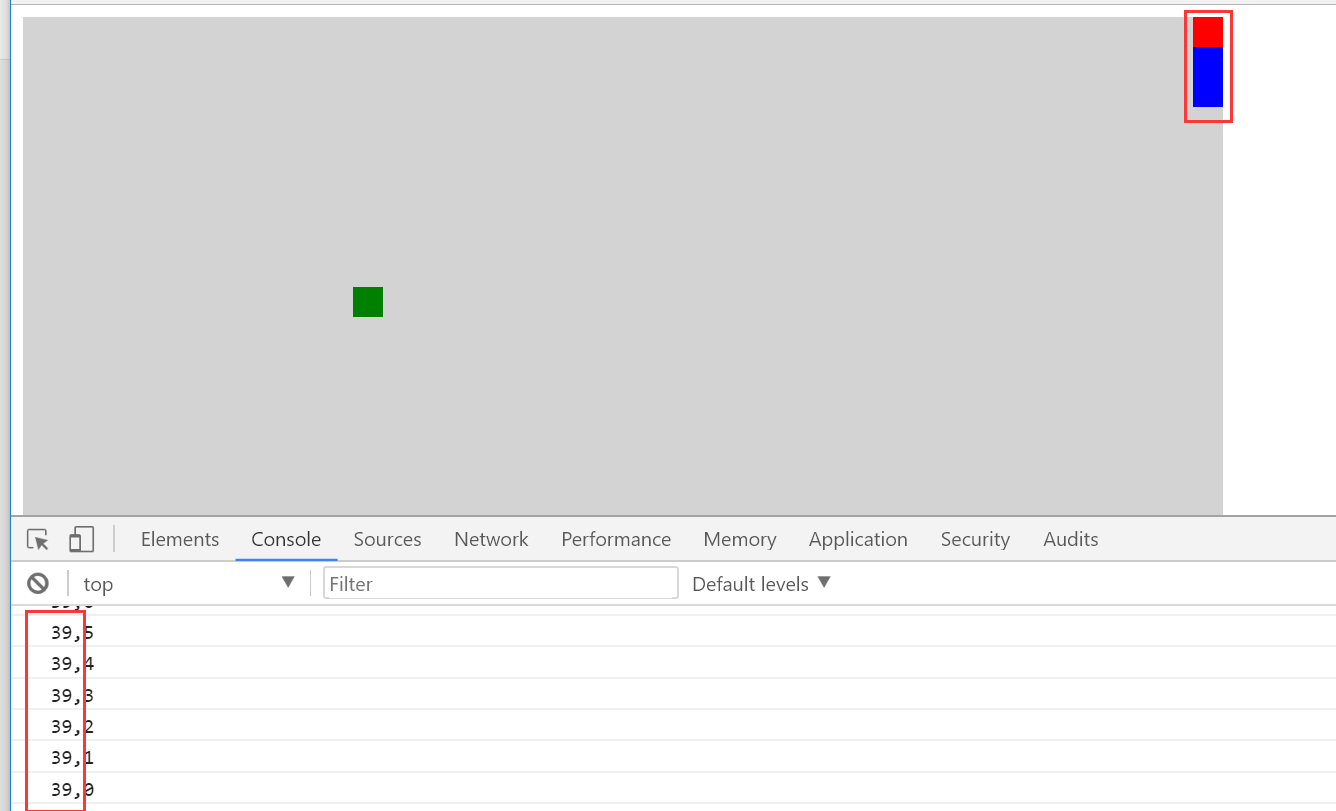
x和y最大不能大于容器横向或纵向最后一个方块的位置

案例的容器宽800,每个方块宽20,最大能容纳800/20 = 40个方块

那么最后那个方块所在的位置应该是39\*20 = 780,也就是39

因为坐标看的是left值而非right值

其实最简单的办法就是在蛇的move方法中把x和y的值打印出来,蛇走到每个位置都观察一下即可



有了判断条件之后就需要知道判断条件代码写在哪儿

注意: 需要将判断条件放到move方法和render方法的调用之间!

因为如果蛇已经到边界了,那最后一次执行还会move并render,导致蛇会移除整个容器,这一点与alert阻止渲染没有任何关系!

如果把判断条件放到move和render之间是什么意思呢?

因为move方法并不会将页面上的蛇移动,只是会把蛇的位置改变,也就是下一步要去的地方

所以调用完该方法之后马上要判断这个值是否合法(是否移出容器)

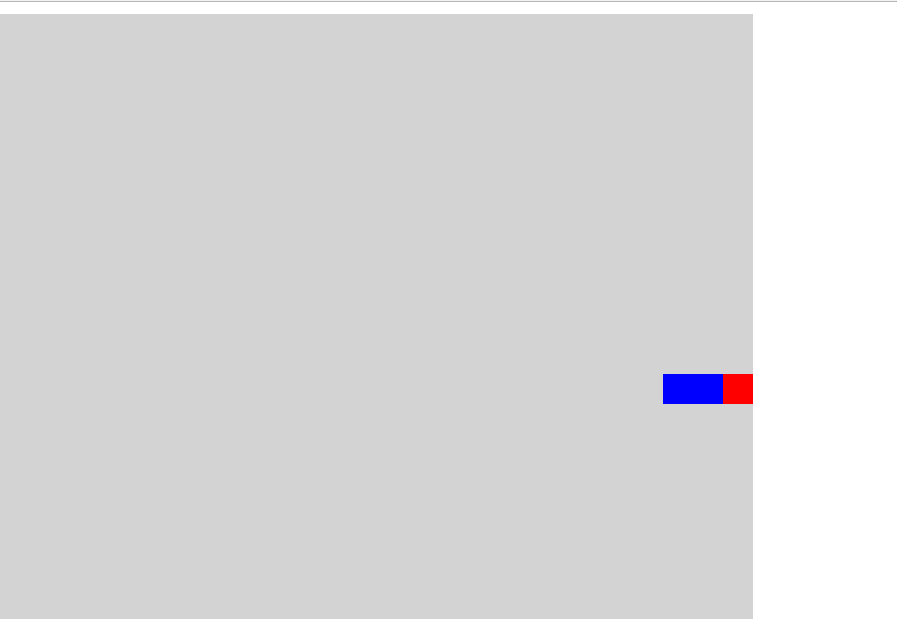
如果不合法就立马清除定时器,并且弹窗提醒用户,然后一定要return!

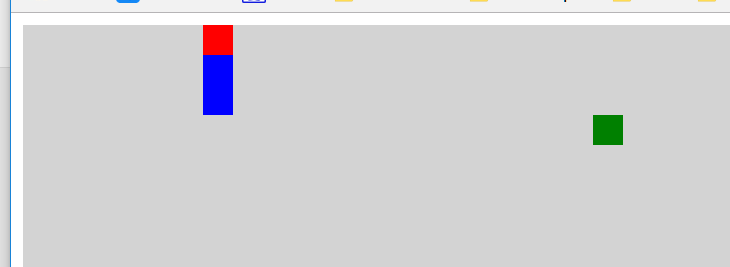
阻止代码继续往后执行,否则执行了render还是白搭

这样就算GAMEOVER,蛇头也不会移出容器,用户体验大大提升!

*// 私有的函数 让蛇移动***function** *runSnake*() {  
 **var** timerId = setInterval(**function** () {  
 *// 让蛇走一格  
 // 在定时器的function中this是指向window对象的  
 // this.snake  
 // 要获取游戏对象中的蛇属性* that.**snake**.move();  
  
 *// 2.2 当蛇遇到边界游戏结束  
 // 获取蛇头的坐标* **var** maxX = that.map.**offsetWidth** / that.**snake**.**width**;  
 *//var maxX = that.map.offsetWidth / that.snake.width-1;//（减一或者下面判断使用>=，注意边界 7个格子 left应该是6个格子的距离）* **var** maxY = that.map.**offsetHeight** / that.**snake**.**height**;  
 **var** headX = that.**snake**.**body**[0].**x**;  
 **var** headY = that.**snake**.**body**[0].**y**;  
 **if** (headX < 0 || headX > maxX || headY < 0 || headY >= maxY) {  
  
 clearInterval(timerId);  
 alert(**'Game Over'**);*//会阻止浏览器渲染，注释掉就可以或者自己写个弹出的div模拟。* **return**;  
 }  
  
 that.**snake**.render(that.map);  
 }, 150);  
}

效果如下:





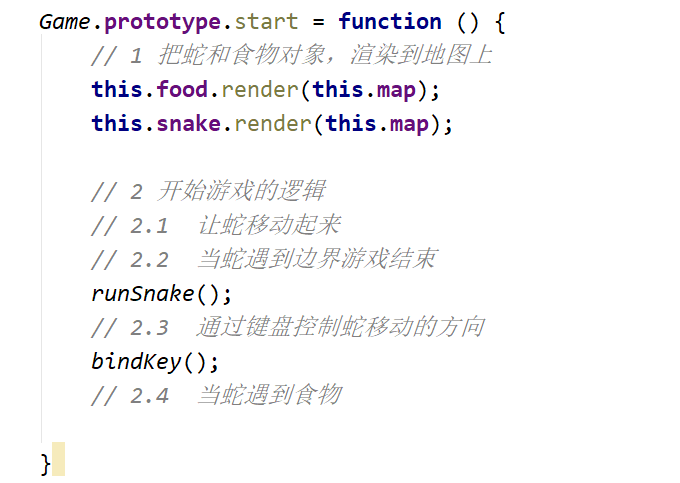
都非常完美,完全不会超出一点!

11-键盘控制蛇的移动.avi

蛇已经会跑了,也会死了

但是蛇还不会改变方向,下面就是通过方向键改变蛇的方向

同样也是在start方法中调用一个方法来完成通过按键控制蛇移动的方向



由于在snake对象的move方法中已经做了判断,根据snake.direction属性来判断往哪个方向走

所以现在只需要捕获用户按下上下左右键,分别修改对应的direction属性即可让蛇自己跑

在bindKey中绑定键盘按下事件

打印观察上下左右的keycode是多少,然后使用switch进行判断

如果按上键,将snake对象的direction属性改为top

其他几个键同理

*// 通过键盘控制蛇移动的方向***function** *bindKey*() {  
 *// document.onkeydown = function () {};* **document**.addEventListener(**'keydown'**, **function** (e) {  
 *// console.log(e.keyCode);  
 // 37 - left  
 // 38 - top  
 // 39 - right  
 // 40 - bottom* **switch** (e.**keyCode**) {  
 **case** 37:  
 that.**snake**.**direction** = **'left'**;  
 **break**;  
 **case** 38:  
 that.**snake**.**direction** = **'top'**;  
 **break**;  
 **case** 39:  
 that.**snake**.**direction** = **'right'**;  
 **break**;  
 **case** 40:  
 that.**snake**.**direction** = **'bottom'**;  
 **break**;  
 }  
 }, **false**);  
}

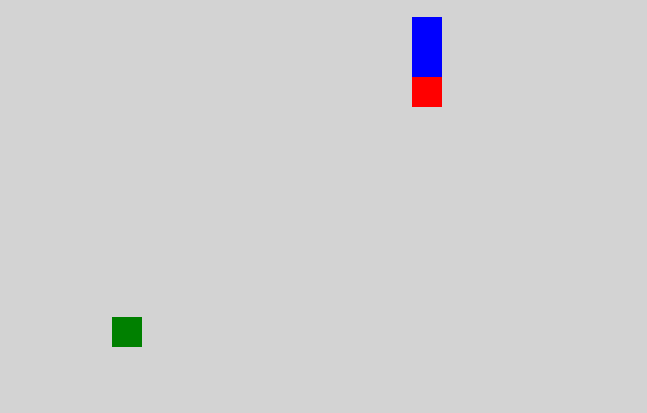
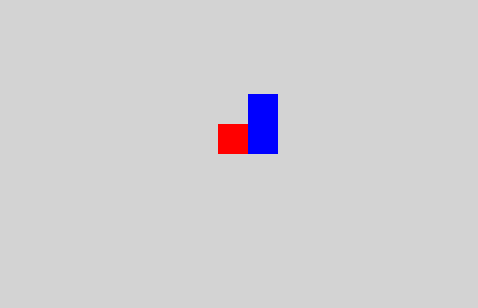
最后按上下左右键会改变snake对象的direction值

由于调用过runSnake函数,会有个定时器每隔150ms让蛇移动

让蛇移动时就会判断方向从而实现让蛇改变方向的功能

Soeasy!

效果如下:



12-蛇吃食物.avi

现在蛇的基本功能大致完成的差不多了,最后还有个功能

蛇头碰到食物时要将食物从页面删除并生产新食物

蛇身长度+1

分析问题:

当蛇头和食物的位置重合时即可判定为吃到食物了

而且蛇是在移动的过程中才能吃到食物,也就是说吃食物的判断应该在snake对象的move方法中完成

但是要在move中判断蛇头和食物的位置,move中可以获取到snake对象,但是无法获取到food对象,所以需要在参数中将food传入进来

而且在蛇吃完食物后,食物需要重新生成,也就是说food对象还需要重新调用render方法,而render方法需要传入一个map对象指定将食物渲染到哪个地方

所以move方法中除了要传入food,还要传入map两个参数

给move方法添加参数后就一定要注意,在任何调用move的地方都需要传入实际参数,否则会报错

在move方法中要判断蛇头和食物的位置是否重合,需要获取两者的坐标

需要注意,蛇头的xy存储的是距离左边有几个蛇节宽度,食物的xy存储的是在地图上真实的位置,所以需要将蛇头的xy乘以一个蛇节的宽高转为蛇头的真实坐标

再判断蛇头的xy和食物的xy是否相等

如果相等表示蛇头和食物重合了,需要做两件事情:

1. 将蛇添加一节
2. 重新生成新食物

生成新食物很简单,直接调用food对象的render方法即可

将蛇添加一节怎么实现呢?

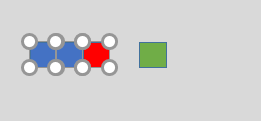
其实就是在snake对象的body数组末尾再添加一个新对象,该对象的xy和颜色与当前最后一个蛇节完全一样,那么此时添加进body数组中并不会在页面上显示

但是由于蛇一直在动,一直在渲染页面,下一次调用move方法时,添加进来的蛇节就会被渲染出来

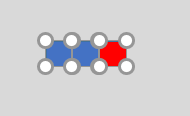
下一次调用move方法正好往前走一步,那么前面每一节都会往前走一步,但由于最后一节和前一节一样,所以最后一节还滞留在原处,所以就形成了蛇变长的效果

例如:

吃食物前



吃掉食物的瞬间,这时候会添加一个蛇节在末尾,与现在的最后一个蛇节相同位置,但没有显示出来,因为还没有渲染



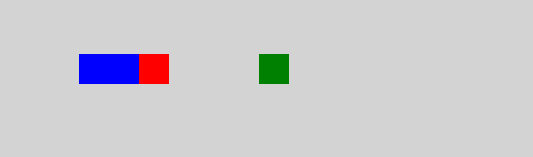
下一次move就会渲染,第三个蛇节会往前走一步,第二个蛇节也会往前走一步,第一个蛇节也会往前走一步,由于新加进来的和第三个蛇节位置一样,所以出现在了原来第三节出现的位置,所以给用户的感觉就是增长了一节



代码如下:

*// 控制蛇移动的方法  
Snake*.**prototype**.move = **function** (food, map) {  
 *// 控制蛇的身体移动（当前蛇节 到 上一个蛇节的位置）* **for** (**var** i = **this**.**body**.**length** - 1; i > 0; i--) {  
 **this**.**body**[i].**x** = **this**.**body**[i - 1].**x**;  
 **this**.**body**[i].**y** = **this**.**body**[i - 1].**y**;  
 }  
 *// 控制蛇头的移动  
 // 判断蛇移动的方向* **var** head = **this**.**body**[0];  
 **switch**(**this**.**direction**) {  
 **case 'right'**:   
 head.**x** += 1;  
 **break**;  
 **case 'left'**:   
 head.**x** -= 1;  
 **break**;  
 **case 'top'**:  
 head.**y** -= 1;  
 **break**;  
 **case 'bottom'**:  
 head.**y** += 1;  
 **break**;  
 }  
  
 *// 2.4 判断蛇头是否和食物的坐标重合* **var** headX = head.**x** \* **this**.**width**;  
 **var** headY = head.**y** \* **this**.**height**;  
 **if** (headX === food.**x** && headY === food.**y**) {  
 *// 让蛇增加一节  
 // 获取蛇的最后一节* **var** last = **this**.**body**[**this**.**body**.**length** - 1];  
 **this**.**body**.push({  
 **x**: last.**x**,  
 **y**: last.**y**,  
 **color**: last.**color** })  
  
 *// 随机在地图上重新生成食物* food.render(map);  
 }  
  
}

效果如下:





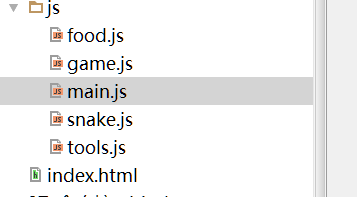
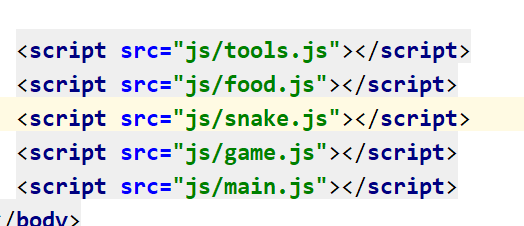
13-main执行代码.avi

之前写代码都是测试阶段,将所有的测试代码全部放到game.js中了

这样做是不对的,每个js就是一个对象的构造函数而已

执行代码的地方应该单独定义一个程序的入口

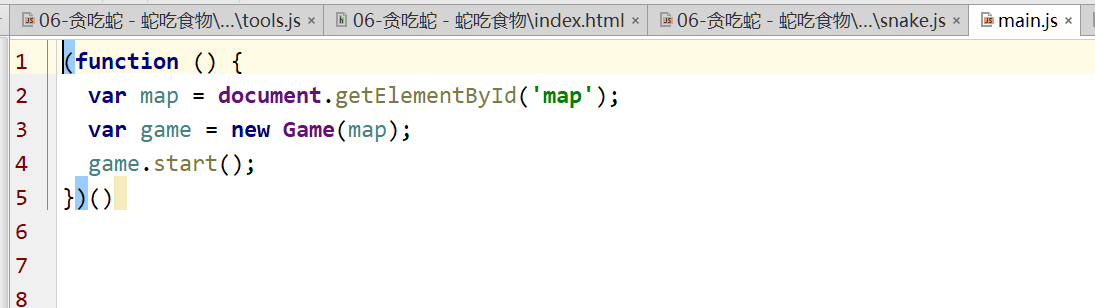
不要直接写在HTML中,可以再来一个js并引用

将之前的测试代码放到main.js中即可

最终的效果其实就只是创建game对象,调用start方法即可

非常方便!



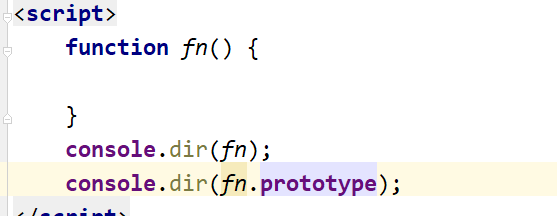
14-bind.avi

bind方法是ES5中添加的新方法

主要作用在于改变函数this的指向

bind方法也是函数的一个方法

注意: 函数也是对象!

简单的例子先演示一下效果:

**var *a*** = 123;  
**function** *fn*() {  
 **console**.log(**this**.***a***);  
}  
*fn*();

现在全局作用域有一个变量a,值为123

fn函数中输出this.a

最后调用fn函数

直接调用fn函数就相当于是window对象调用的fn函数

也就是说this指向的就是window对象,所以输出的结果就是123



这时创建一个新的对象o,该对象中也存一个属性a,值为abc

*// 新建一个方法，bind中的第一个参数可以改变函数中的this的指向***var *o*** = {**a**: **'abc'**};

调用fn的bind方法,将o传入,bind方法的返回值就是一个新函数

使用fn1变量接收,并调用fn1

**var *fn1*** = *fn*.bind(***o***);   
***fn1***(); *// 相当于 o.fn()*

这时候调用fn1会执行fn函数,并且fn函数中用到的所有的this都会指向o

也就等同于使用o.fn()

效果如下:



注意: 调用bind方法只会返回一个新函数,并不会执行这个函数,还需要自行手动调用

完整代码如下:

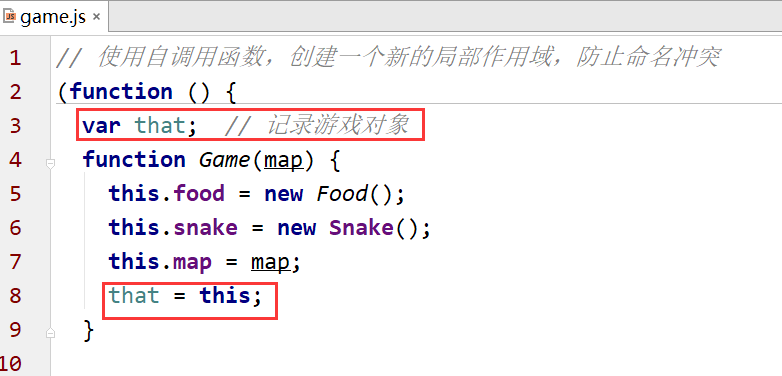
**var *a*** = 123;  
 **function** *fn*() {  
 **console**.log(**this**.***a***);  
 }  
 *fn*();  
  
 *// fn.prototype  
   
 // bind 方法 ES5中新增的方法  
 // 新建一个方法，bind中的第一个参数可以改变函数中的this的指向  
 // bind并没有调用方法***var *o*** = {**a**: **'abc'**};  
**var *fn1*** = *fn*.bind(***o***);   
***fn1***(); *// 相当于 o.fn()*

学习bind方法的目的是啥呢?

在贪吃蛇的game对象的runSnake方法中,设置了定时器,而在定时器的回调函数中需要使用snake对象

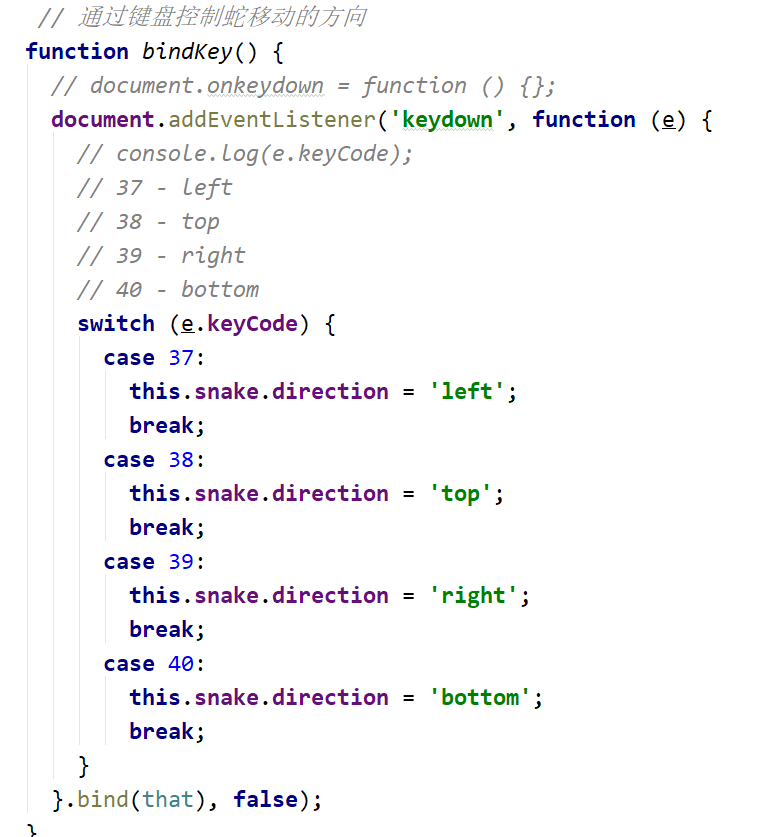
由于是在定时器的回调函数中,所以无法直接访问this,只能在创建对象时将this指向另一个变量that然后使用

如果想在定时器的回调函数中使用this指向game对象,可以调用回调函数的bind方法,传入that即可





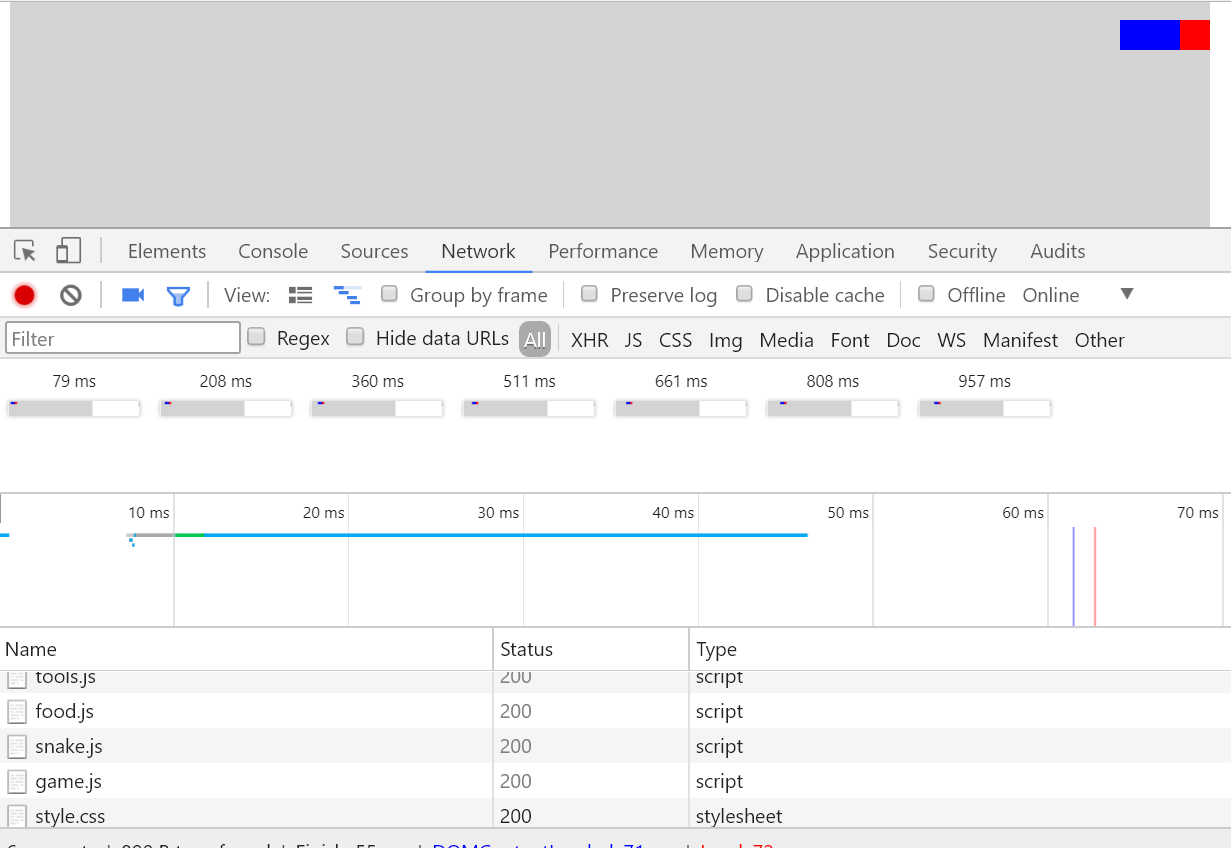
bindkey方法同理



15-自调用函数的问题.avi

现在项目开发完毕了,但是有一个问题

在我们访问网页的时候,打开浏览器的调试工具network一栏可以看到:

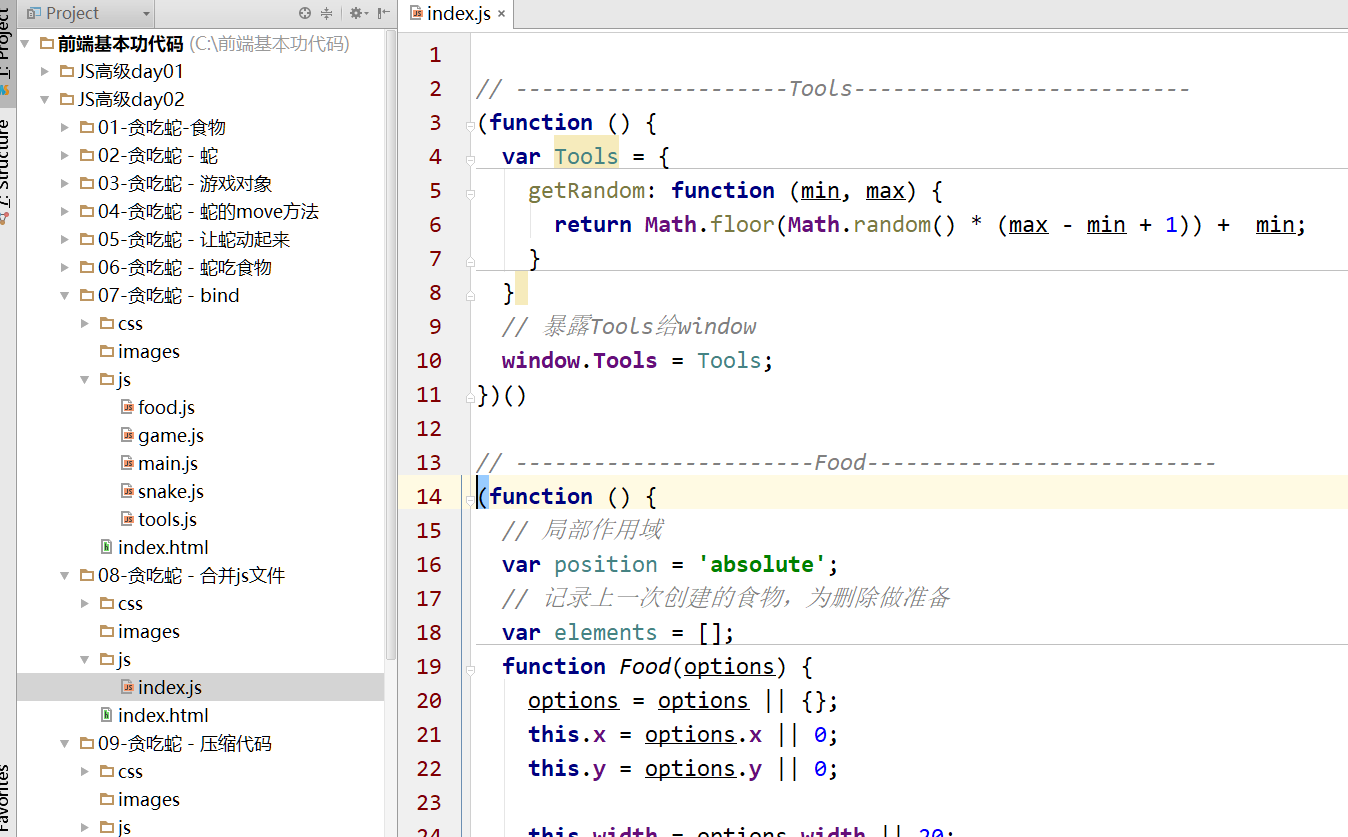


开一个网页一次加载了好多文件

虽然开发的时候把所有的模块拆分是方便开发和维护,但是最终项目上线了,用户去访问的时候如果有很多很多js要下载,要很多次请求服务器,也是很消耗性能的

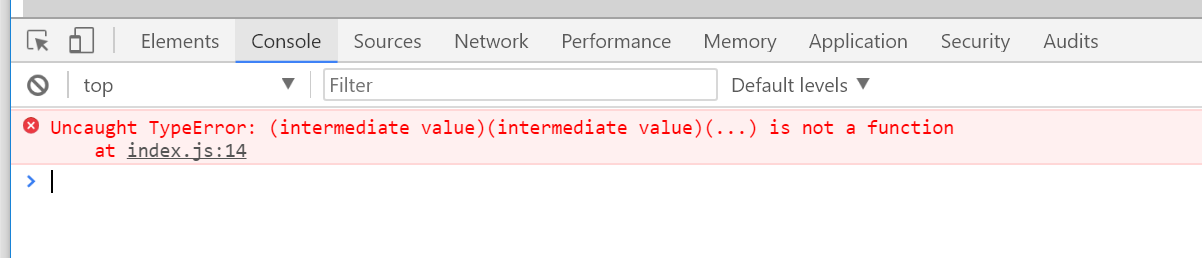
所以,**将来项目上线的时候会把所有的js合并,开发的时候还是分开来开发**

接下来手工合并一下,将来会有一些工具自动生成合并后的代码



其他js全部删掉,一定要按照引入顺序依次拷贝代码过来

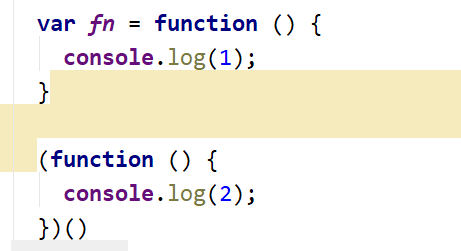
效果如下:



报错了!

任何代码都没改变,只是拷贝到了一个js文件中竟然会报错!

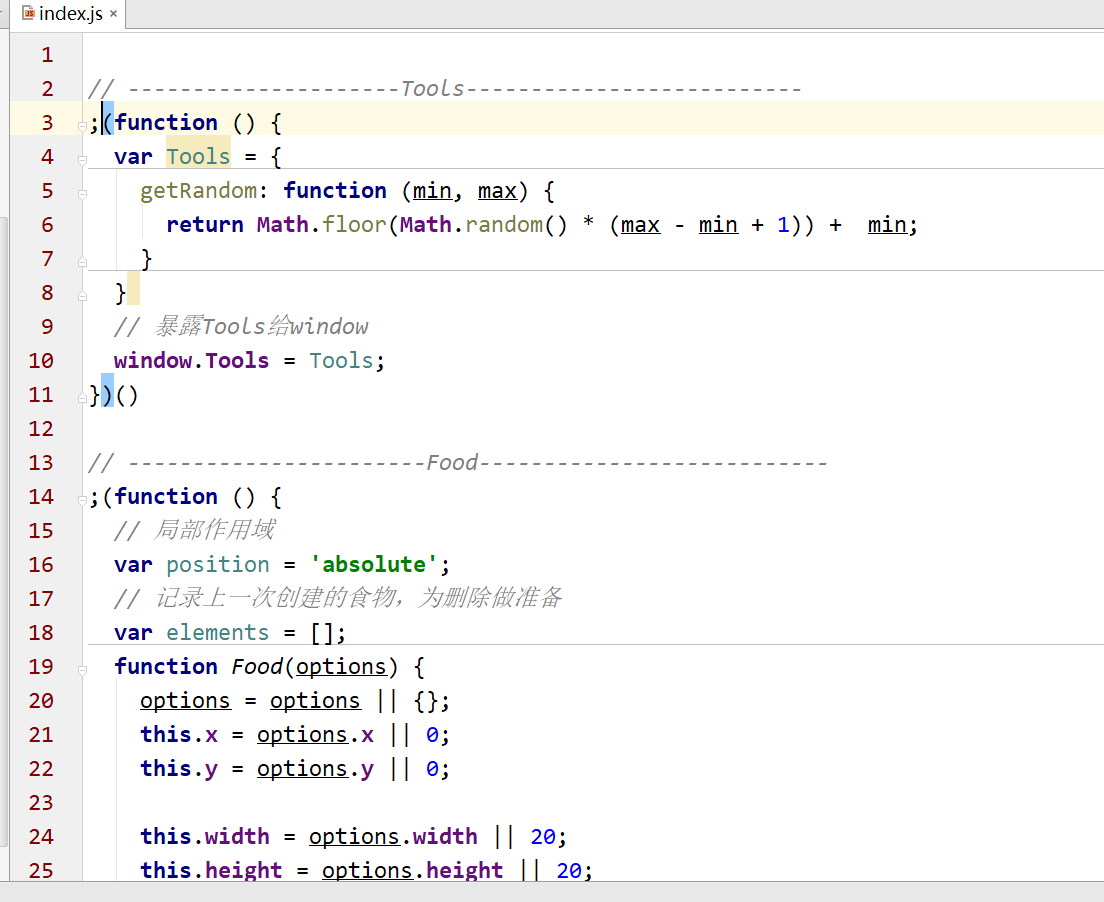
举例说明:

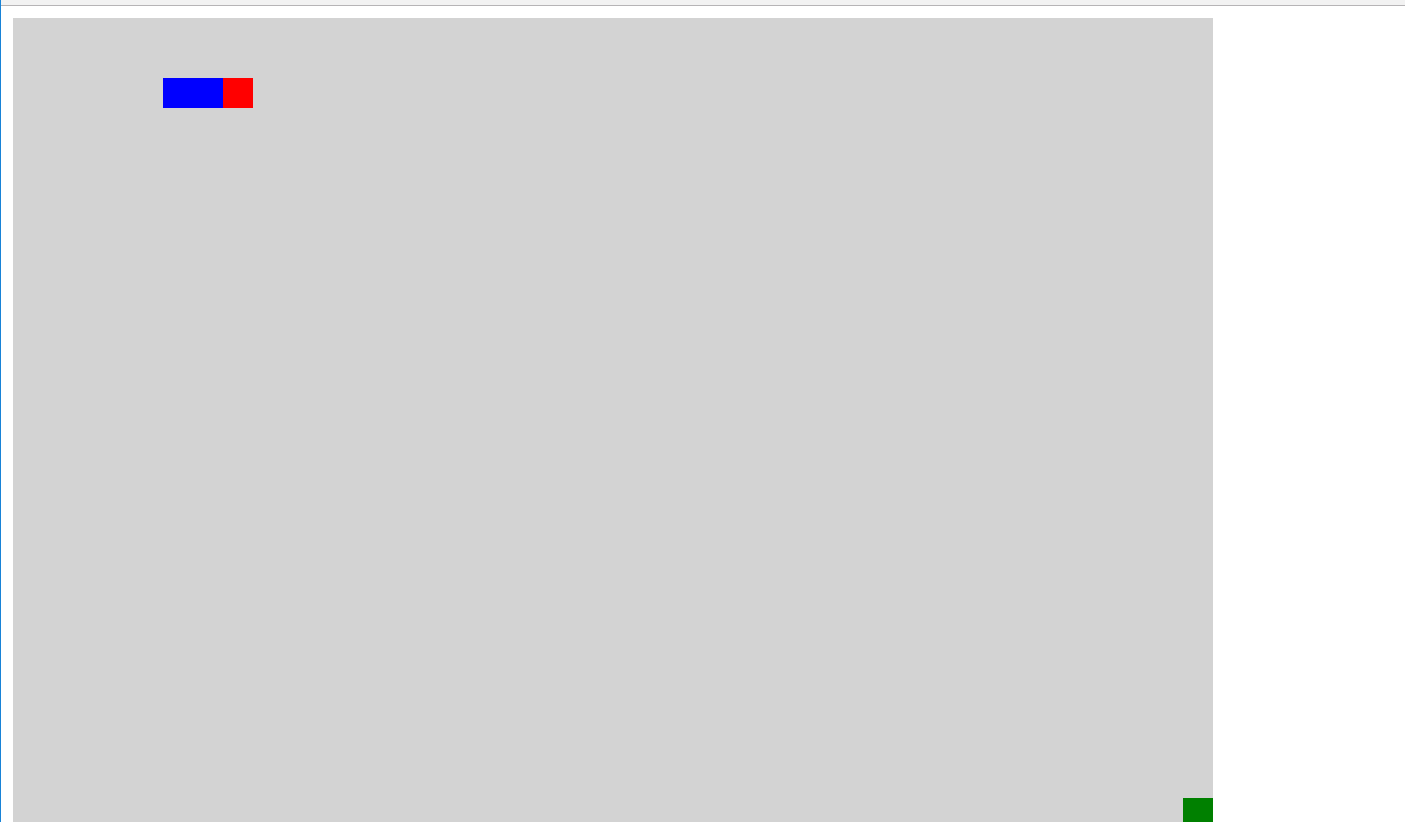


如果自调用函数前面有代码,代码解析器会去掉中间所有的空格,赋值语句完成后就是fn后面紧接着自调用函数,那么语法就会出错

所以以后使用多个自调用函数时一定要在自调用函数前面加分号,加在前面的目的是为了防止别人坑你

效果如下:





16-压缩代码.avi

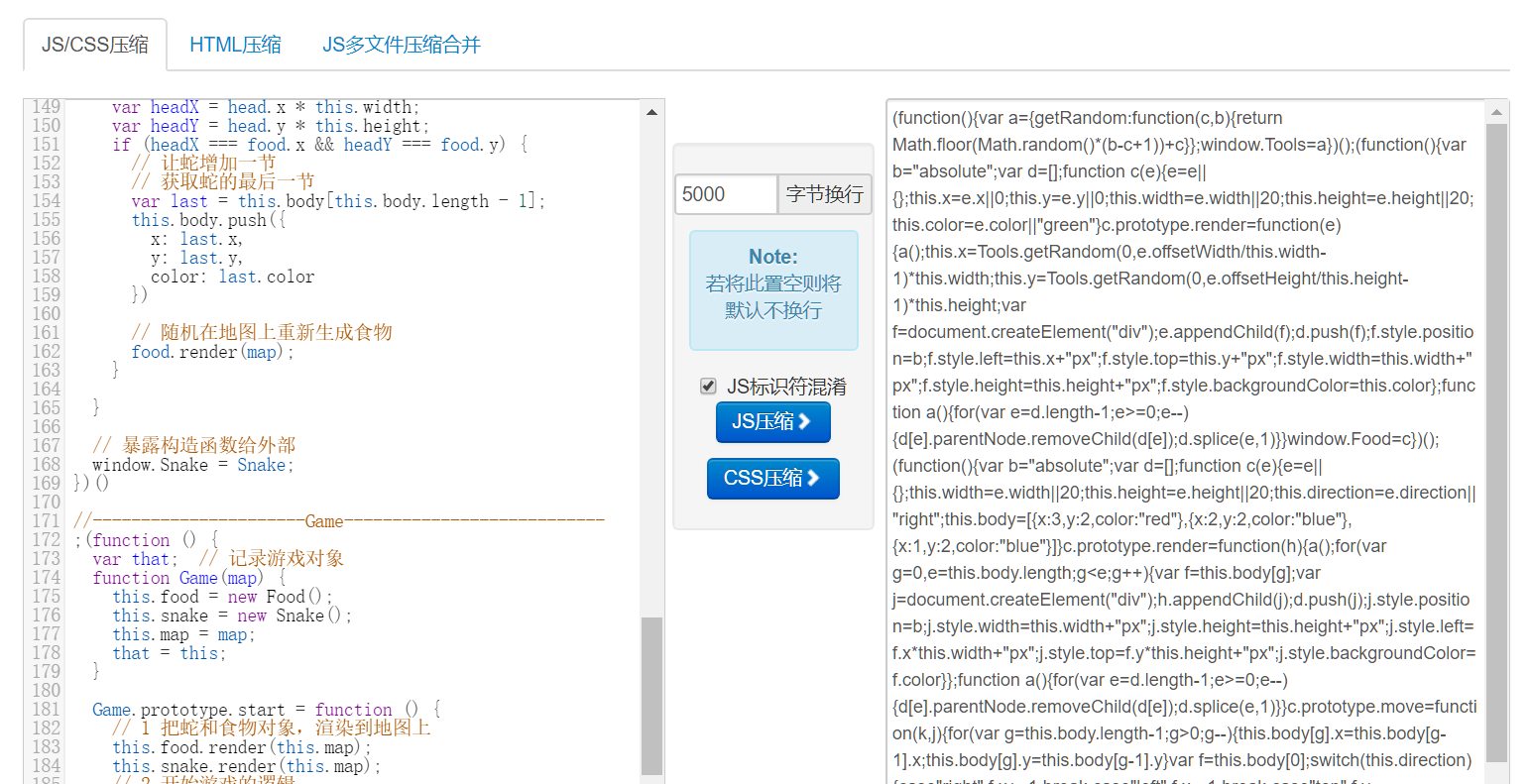
现在已经合并了代码,但是代码体积还是不小的,所以还可以进一步压缩,大大提高用户体验,节省用户宝贵的流量!

在搜索引擎中搜索js压缩,会有各种各样在线工具可供使用

这里使用OSChina的工具: <http://tool.oschina.net/jscompress>



效果如下:



记得勾选混淆哦,可以将自己起的变量名变得很短!

短小精悍!自然更省流量!

最后将生成的压缩后的代码拷贝回项目即可

17-自调用函数的参数.avi

最后还有一个小细节,一般在写自调用函数时,会给自调用函数定义两个参数:

window和undefined

在调用时传入两个参数

这两个参数的目的是为了进一步的压缩代码,因为在函数内部要使用window对象,那代码压缩时不能把window字符串给压缩

如果把window作为参数传递到函数中,那么形参是可以被压缩的,又可以省几个字符

当然,传入的实参肯定不能被压缩,但是只用传入一次,而函数中可能会多次用到window

undefined同理



效果如下:

