# 数据类型

## 基本类型（值）

·String

- 任意字符串

·Number

- 任意数字

·Boolean

- true

- false

·undefined

·null

## 引用类型（对象）

·Object

- Function

-- 可以执行的对象

- Array

-- 有数值下标、内部数据是有序的对象

## 判断

·typeof

- 判断数据类型

- 返回数据类型的字符串表达

- 可以判断undefined

**typeof** a === "undefined"

- 可以判断number

**typeof** a === "number"

- 可以判断string

**typeof** a === "string"

- 可以判断boolean

**typeof** a === "boolean"

- 可以判断Function

**typeof** b1.b3 === "function"

- **不能判断null与object、object与array**

**var** b1 = {

b2:[1,"abc",console.log],

b3:**function** () {

console.log("b3");

**return function** () {

**return** "我在哪？";

}

}

};

·instanceof

- 实例，判断对象的具体类型

- 可以判断Object

b1 **instanceof** Object

- 可以判断Array

b1.b2 **instanceof** Array

- 可以判断Function

b1.b3 **instanceof** Function

·===

- 判断两值是否完全相等

- 可以判断undefined

a === **undefined**

实例：实例对象

类型：类型对象

**function** Person(name, age) {//构造函数 类型

**this**.name = name;

**this**.age = age;

}

**var** p = **new** Person("tom",12);//根据类型创建的实例

// Person("jack",12);//语法没问题，但一般不这么用

## undefined与null的区别

·undefined代表定义未赋值

·null代表定义并赋值了，只是值为null

## 什么时候给变量赋值为null呢？

·初始赋值，表明将要赋值为对象

·结束时赋值，让对象成为垃圾对象（被垃圾回收器回收，赋值为null并不是立即释放）

## 严格区别变量类型与数据类型

### 数据的类型

·基本类型

·对象类型

### 变量的类型

变量内存值的类型

·基本类型：保存的就是基本类型的数据

·引用类型：保存的是地址值

## 堆和栈的区别

一、  堆(heap)和栈(stack)

栈(stack)会自动分配内存空间，会自动释放。堆(heap)动态分配的内存，大小不定也不会自动释放。

二、  基本类型和引用类型

基本类型：简单的**数据段存放在栈**内存中，**占据固定大小**的空间。

引用类型：指那些可能由多个值构成的**对象，保存在堆**内存中,包含引用类型的变量**实际上保存的不是变量本身，而是指向该对象的指针**。

1. 传值和传址

从一个向另一个变量**复制引用类型的值**，复制的其实是指针，因此两个变量最终指向同一个对象。即**复制的是栈中的地址而不是堆中的对象**。

从一个变量复向另一个变量**复制基本类型的值，会创建这个值的副本**。

# 数据、变量、内存

## 什么是数据？

·存储在内存中代表特定信息的东西，本质上是二进制编码

·数据的特点：

- 可传递

- 可运算

·一切皆数据

·内存中所有操作的目标：数据

- 算术运算

- 逻辑运算

- 赋值

- 运行函数

## 什么是内存？

·内存条通电后产生的可存储数据的空间（临时的）

·内存产生和死亡：

- 内存条（电路板）--> 通电 --> 产生内存空间 --> 存储数据 --> 处理数据 --> 断电 --> 内存空间和数据都消失

·一块小内存的2个数据

- 内部存储的数据

- 地址值

·内存分类

- 栈：全局变量/局部变量

- 堆：对象

## 什么是变量？

·可变化的量，由变量名的变量值组成

·每个变量都对应一块小内存，变量名用来查找对应的内存，变量值就是内存中保存的数值数据。

## 内存、数据和变量三者之间的关系

·内存用来存储数据的空间

·变量是内存的标识

## 问题：var a = xxx，a内存中到底保存的是什么？

·xxx是基本数据，保存的就是这个数据

·xxx是对象，保存的是对象的地址值

·xxx是一个变量，保存的是xxx的内存内容（可能是基本数据，也可能是地址值）

## 关于引用变量赋值问题

·多个引用变量指向同一个对象，通过一个变量修改对象内部数据，其他所有变量看到的是修改之后的数据。

·多个引用变量指向同一个对象，让其中一个引用变量指向另一个对象，其他所有变量依然指向前一个变量。

**var** xx = {age:12};

**function** fn2(obj) {

obj = {age:15};//不会修改传入的参数值

}

fn2(xx);

console.log(xx.age);//12

## 在JS调用函数时传递变量参数，是值传递还是引用传递

**var** a = 3;

**function** fn3(a) {

a = a + 1;

}

fn3(a);

console.log(a);//3

·理解1：都是值（基本/地址）传递，

·理解2：可能是值传递，也可能是引用传递（地址值）

## JS引擎如何管理内存？

·内存生命周期

- 分配小内存空间，得到它的使用权

- 存储数据，可以往后进行操作

- 释放小内存空间

·释放内存

- 局部变量：函数执行完自动释放

- 对象：成为垃圾对象 --> 垃圾回收器回收

# 对象

**var** p = {

name : "Tom",

age : 12,

setName : **function** (name) {

**this**.name = name;

},

setAge : **function** (age) {

**this**.age = age;

}

};

## 什么是对象？

·多个数据的封装体

·保存多个数据的容器

·一个对象代表现实中的一个事物

## 为什么要用对象？

·统一管理多个数据

## 对象的组成

·属性

- 由属性名（字符串）和属性值（任意类型）组成

·方法

- 一种特别的属性（属性值为函数）

如何访问对象内部数据？

·.属性名：编码简单，有时不能用

·[“属性名”]：编码麻烦，能通用

## 什么时候必须用[“属性名”]的方式？

·属性名包含特殊字符

例如：- 空格

**var** p ={};

//给p对象添加一个属性：content-type: text/json

// p.content-type = "text/json";//报错

p["content-type"] = "text/json";

console.log(p["content-type"]);//text/json

·属性名不确定

//变量名不确定

**var** p ={};

**var** propName = "myAge";

**var** value = 19;

//给p添加myAge属性，属性值为value的值

// p.propName = vaalue;//不能用

p[propName] = value;

console.log(p.myAge);

# 函数

## 什么是函数？

·具有特定功能的n条语句的封装体

·只有函数是可以执行的，其他类型的数据不能执行

## 为什么要用函数？

·提高代码复用

·便于阅读交流

## 如何定义函数？

·函数声明

**function** fn() {//函数声明

console.log("fn");

}

·表达式

**var** fn2 = **function** () {

console.log("fn2");

}

## 如何调用（执行）函数

·函数名();直接调用

·对象名.函数名():通过对象调用

·new 函数名();new 调用

·函数名.call(obj)

- 可以传入一个对象，函数中的this则代表该对象

- call()方法可以将实参在对象之后依次传递

·函数名.apply(obj)

- 可以传入一个对象，函数中的this则代表该对象

- apply()方法需要将参数封装到一个数组中传递

**var** obj = {name:"李四",age:12};

**var** fn2 = **function** (a,b) {

**this**.name = "张三";

**this**.age = a+b;

};

fn2.call(obj,10,7);

console.log(obj);

fn2.apply(obj,[10,8]);

console.log(obj);

## 回调函数

### 什么是回调函数？

·你定义的

·你没有调用

·但最终它执行了

### 常见的回调函数

·dom事件回调函数

document.getElementById("btn").onclick = **function** () {

console.log("这是dom事件回调函数");

};

·定时器回调函数

setTimeout(**function** () {

console.log("这是定时器回调函数");

},3000);

·ajax请求回调函数

·生命周期回调函数

# IIFE

## 全称

·Immediately-Invoked Function Expression(即使调用函数表达式)，匿名函数自调用

(**function** () {

**var** a = 3;

console.log(a + 3);

})();

## 作用

·隐藏实现

·不会污染外部（一般是全局）命名空间

(**function** () {

**var** a = 1;

**function** test() {

console.log(++a);

}

window.$ = **function** () {//$是全局变量的函数，返回一个对象，里面有test属性，值为上面的test函数

**return**{

test : test

};

}

})();

$().test();//运行上面匿名函数test函数

# 函数中的this

## this是什么？

·所有函数内部都有一个变量this

·它的值是调用函数的当前对象。

·任何函数本质上都是通过某个对象来调用的，如果没有直接指定就是window

## 如何确定this的值？

·函数名()

- 直接调用，this的值是window

·对象名.函数名()

- this的值是调用函数的对象

·new 函数名()

- this的值是新创建的对象

·函数名.call(对象名)

- this的值是传入的对象

·函数名.apply(对象名)

- this的值是传入的对象

# 分号

·JS一条语句的后面可以不加分号

·是否加分号是编码风格问题，没有应不应该，只有喜不喜欢

·在下面2种情况不加分号会有问题

- 小括号开头的前一条语句

- 中方括号开头的前一条语句

·解决办法

- 在行首加分号

·强有力的例子：vue.js库

# 函数的prototype

## 函数的prototype属性

·每个函数都有一个prototype属性，它默认指向一个Object空对象（即称为原型对象）

·原型对象中有一个属性constructor,它指向构造函数对象

## 该原型对象添加属性(一般都是方法)

·作用

- 函数的所有实例对象自动拥有原型中的属性（方法）

console.log(Date.prototype,**typeof** Date.prototype);//Date函数的原型对象添加了很多方法，如获取时间的

**function** Fun() {

}

console.log(Fun.prototype);

console.log(Date.prototype.constructor === Date);//原型对象中有一个属性constructor,它指向函数对象

console.log(Fun.prototype.constructor === fun);

Fun.prototype.test = **function** () {

console.log("test()");

};//为Fun函数的原型对象添加test方法

**var** fun = **new** Fun();//创建Fun的实例

fun.test();//通过实例调用为Fun函数的原型对象添加的test方法

## 显示原型与隐式原型

·每个函数function都有一个prototype，即显示原型属性，默认指向一个空的Object对象

·每个实例对象都有一个\_\_proto\_\_，称为隐式原型

·对象的隐式原型的值为其对应构造函数的显示原型的值

·函数的prototype属性：在函数定义时自动添加，默认值是一个空Object对象

·对象的\_\_proto\_\_属性:创建对象时自动添加的，默认值为构造函数的prototype属性值

·程序员能直接操作显示原型，但不能直接操作隐式原型（ES6之前）

**function** Fn() {//内部语句:this.prototype = {}

}

//每个函数function都有一个prototype，即显示原型

console.log(Fn.prototype);

//每个实例对象都有一个\_\_proto\_\_，称为隐式原型

**var** fn = **new** Fn();//内部语句：this.\_\_proto\_\_ = Fn.prototype

console.log(fn.\_\_proto\_\_);

//对象的隐式原型的值为其对应构造函数的显示原型的值

console.log(Fn.prototype === fn.\_\_proto\_\_);//true

//给原型添加方法

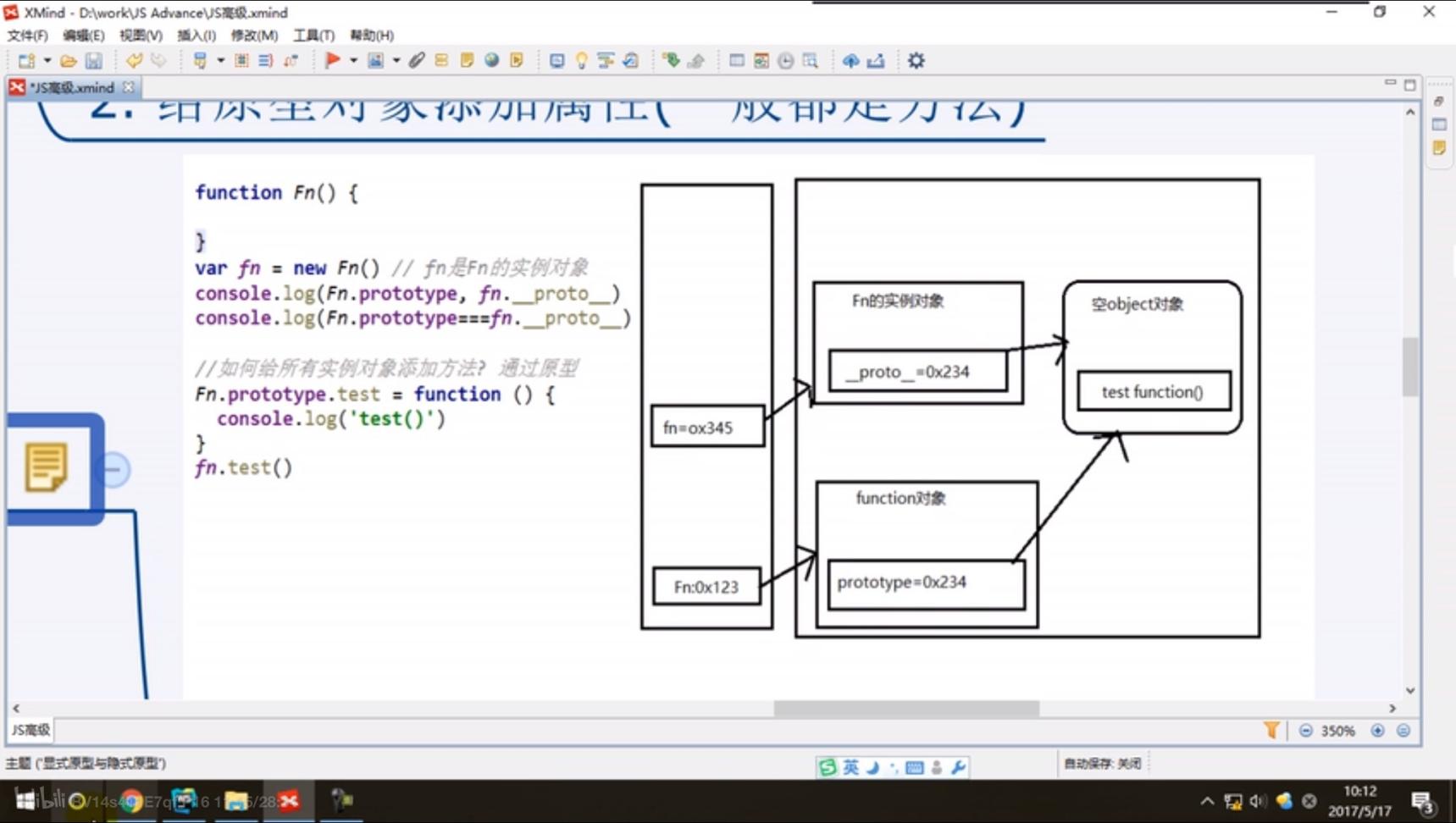
Fn.prototype.test = **function** () {

console.log("test()");

};

//通过实例调用原型的方法

fn.test();



## 原型链

·访问一个对象的属性时

- 先在自身属性查找，找到返回

- 如果没找到，再沿着\_\_proto\_\_这条链向上查找，找到返回

- 一直找到Object的原型对象，如果没找到，则返回undefined

·别名

- 隐式原型链

·作用

- 查找对象的属性（方法）

**function** Fn() {

**this**.test1 = **function** () {

console.log("test1()");

};

}

console.log(Fn.prototype);//变量会提前声明，这时候就有test2属性了，只是没有对应的方法

Fn.prototype.test2 = **function** () {

console.log("test2()");

};

console.log(Fn.prototype);//这时候test2属性会赋值对应的方法

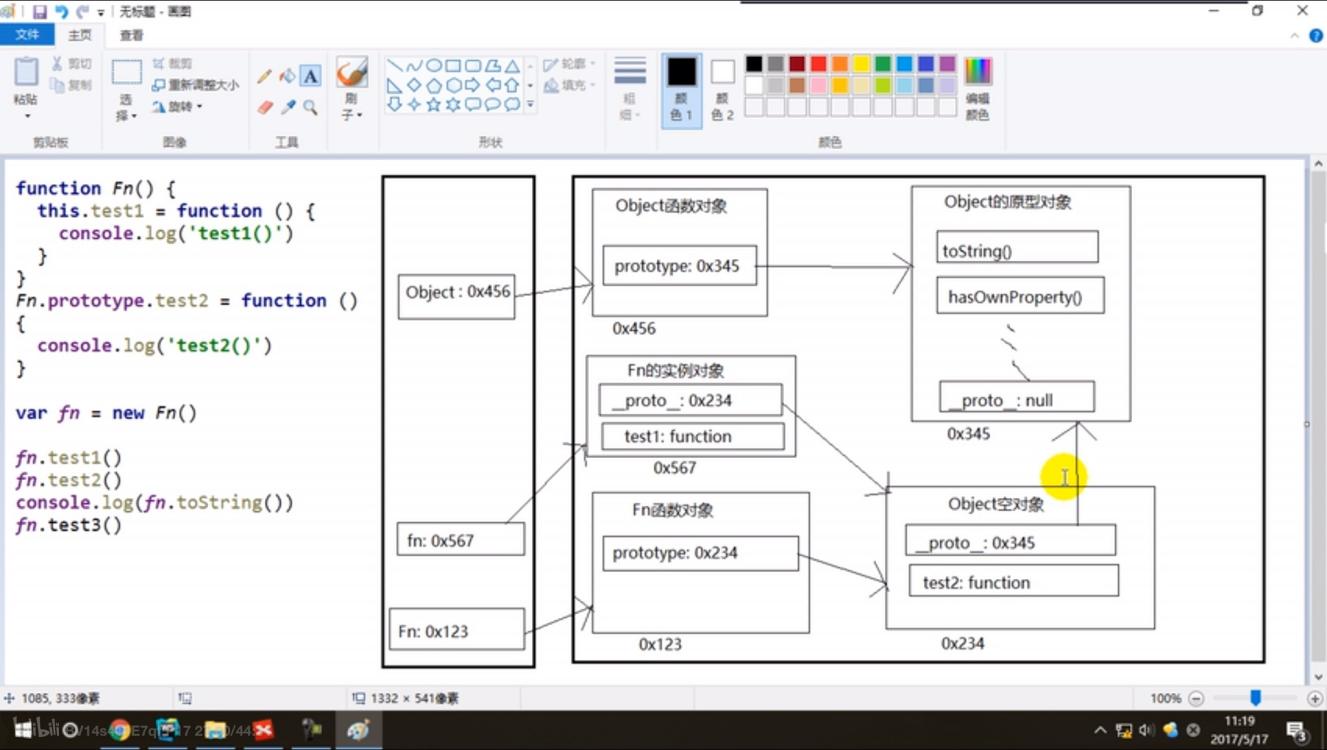
**var** fn = **new** Fn();

fn.test1();

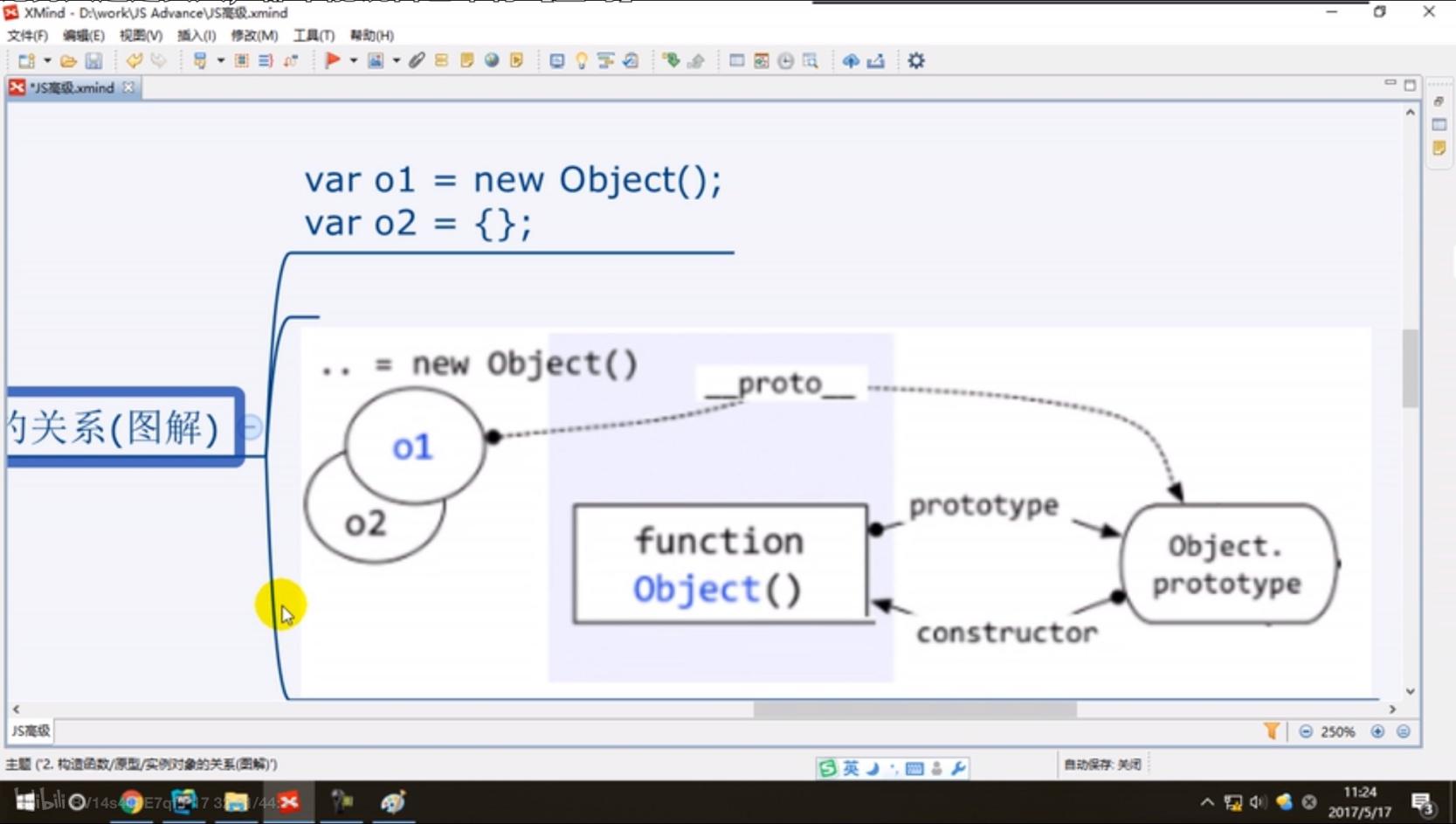
fn.test2();

console.log(fn.toString());

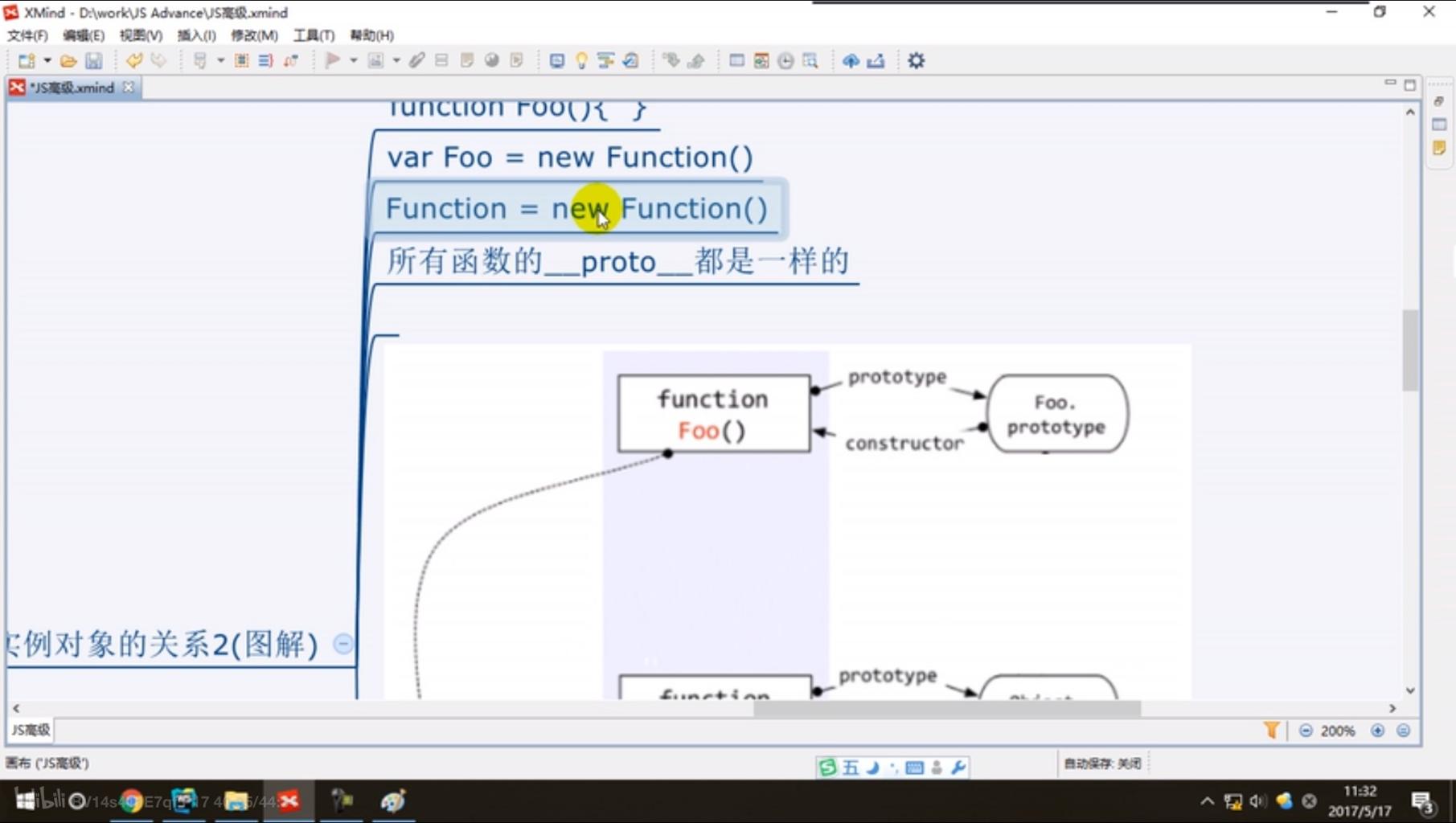
// fn.test3();//报错

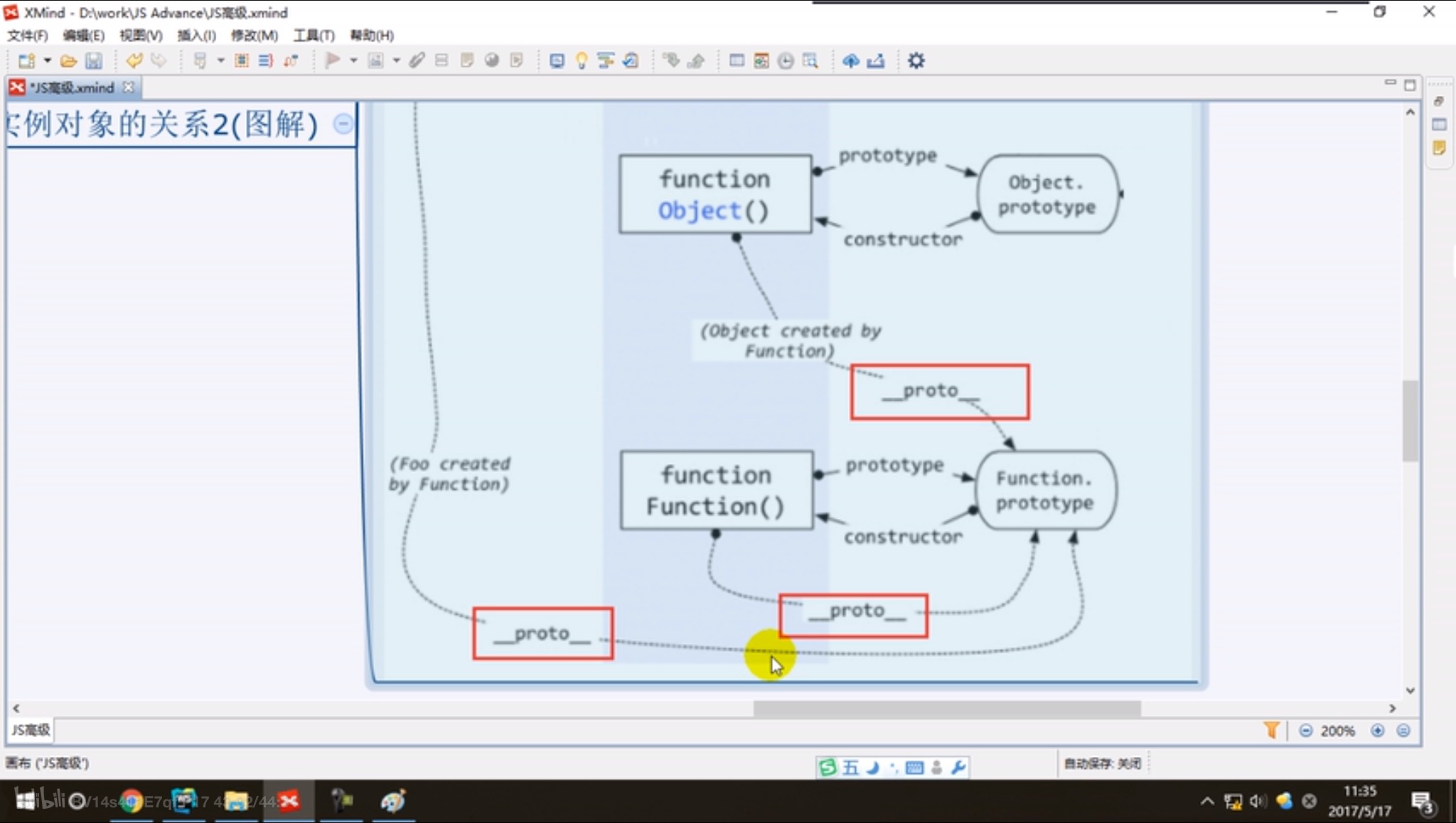


### 构造函数/原型/实例对象的关系



### 构造函数/原型/实例对象的关系2





·函数的显示原型指向的对象默认是空Object实例对象（但Object不满足）

console.log(**typeof** Object.prototype);//Object

console.log(Object.prototype **instanceof** Object);//false

·所有函数都是Function的实例（包含Function本身）

console.log(Function.\_\_proto\_\_ === Function.prototype);//true

·Object的原型对象是原型链的尽头

console.log(Object.prototype.\_\_proto\_\_);//null

# 属性

·读取对象的属性值时，会自动到原型链中查找

·设置对象的属性值时，不会查找原型链，如果当前对象中没有此属性，直接添加此属性并设置其值

**function** Fn() {

}

Fn.prototype.a = "xxx";

**var** fn1 = **new** Fn();

console.log(fn1.a);//xxx

**var** fn2 = **new** Fn();

fn2.a = "yyy";//修改的只是fn2中的属性，不会修改原型的属性

console.log(fn1.a,fn2.a);//xxx，yyy fn2.a先读取fn2中的属性

·方法一般定义在原型中，属性一般通过构造函数定义在对象本身

**function** Person(name, age) {

**this**.name = name;

**this**.age = age;

}

Person.prototype.setName = **function** (name) {//在原型方法中修改对象的属性值

**this**.name = name;

};

**var** p1 = **new** Person("Tom",12);

p1.setName("Bob");

console.log(p1.name,p1.age);//Bob 12

**var** p2 = **new** Person("Tom",12);

p2.setName("张三");

console.log(p2.name,p2.age);//张三 12

console.log(p1.\_\_proto\_\_ === p2.\_\_proto\_\_);//true

# 探索instanceof

## instanceof是如何判断的？

·表达式

- A instanceof B

-- 左边是实例对象

-- 右边是构造函数

- 如果B函数的显示原型对象在A对象的原型链上，返回true,否则返回false

·Function是通过new自己产生的实例

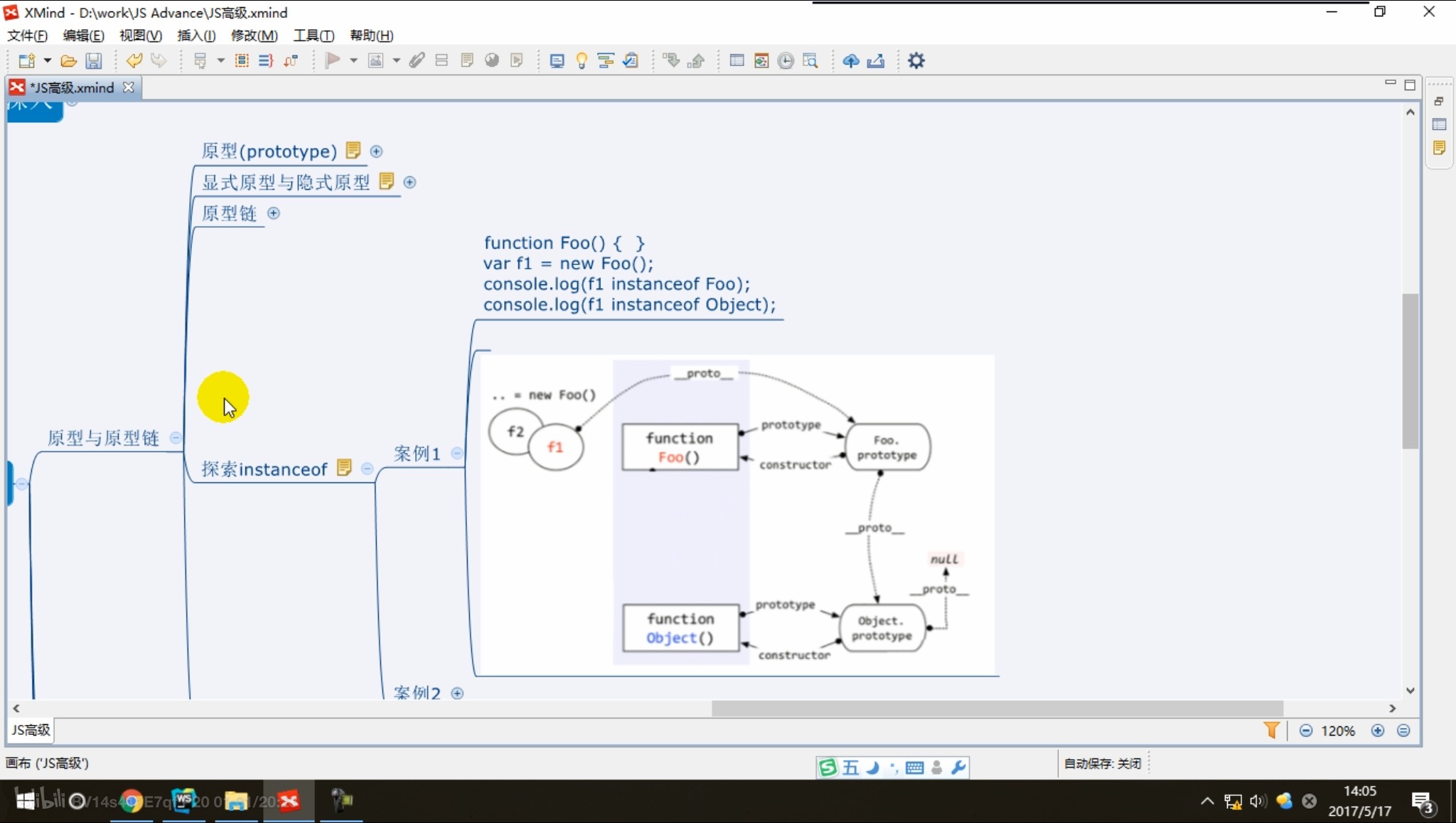
**function** Foo() {

}

**var** f1 = **new** Foo();

console.log(f1 **instanceof** Foo);//true

console.log(f1 **instanceof** Object);//true

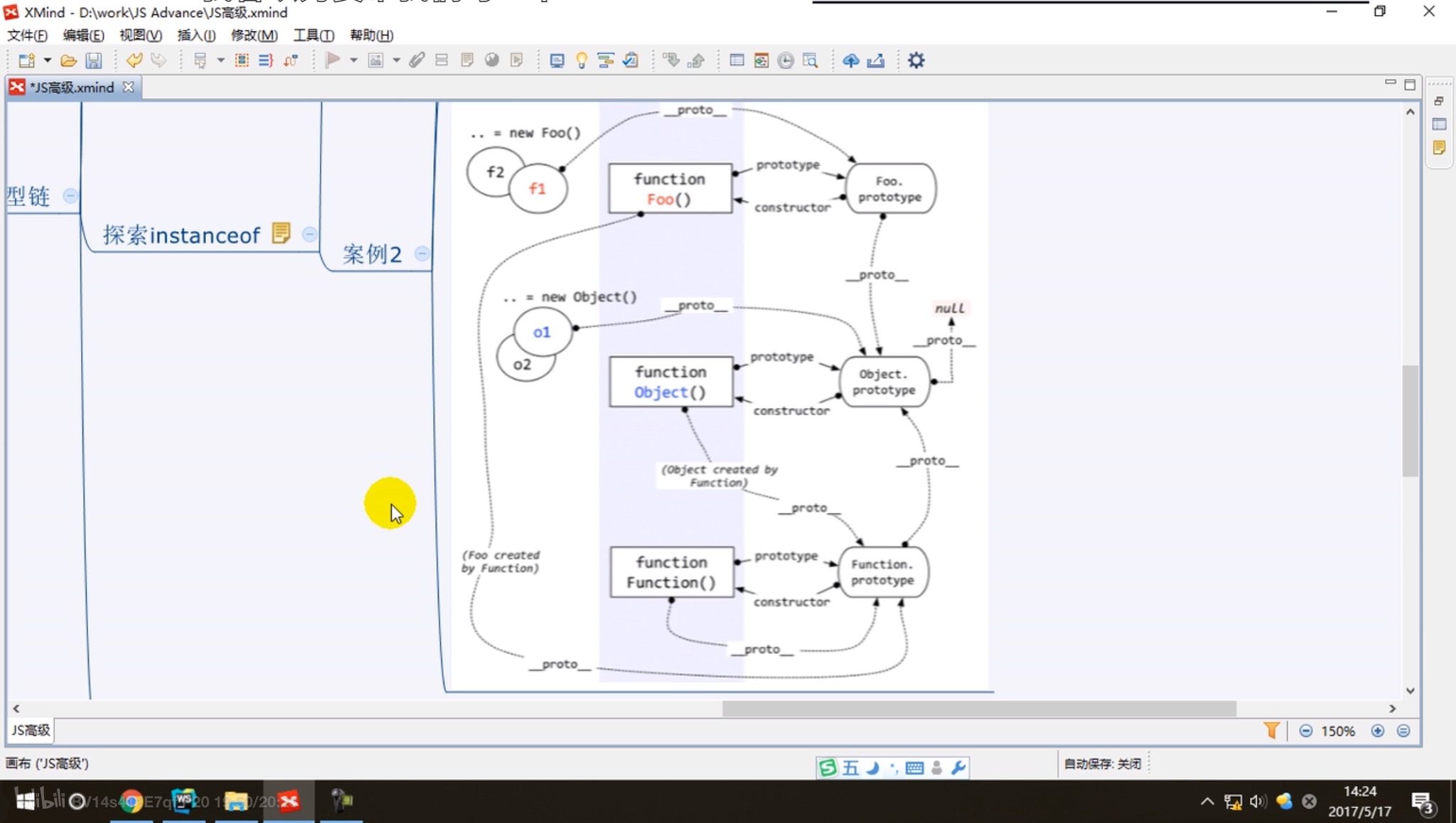


console.log(Object **instanceof** Function);//true

console.log(Object **instanceof** Object);//true

console.log(Function **instanceof** Function);//true

console.log(Function **instanceof** Object);//true



# 原型面试题

**function** A() {

}

A.prototype.n = 1;

**var** b = **new** A();

A.prototype = {

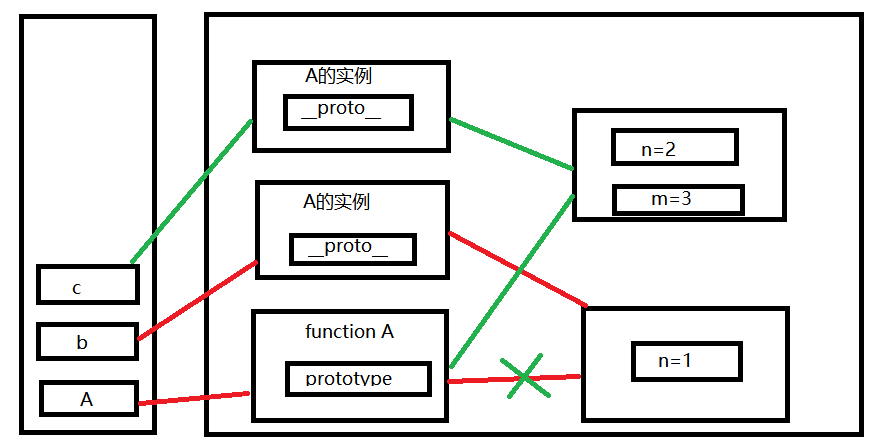
n:2,

m:3

};

**var** c = **new** A();

console.log(b.n , b.m , c.n, c.m);//1 undefined 2 3



**function** F() {

}

Object.prototype.a = **function** () {

console.log("a()")

};

Function.prototype.b = **function** () {

console.log("b()");

};

**var** f = **new** F();

f.a();//a() F() --> F.prototype --> Object.prototype

// f.b();//报错,f的原型链不能找到Function函数的原型，F() --> F.prototype --> Object.prototype --> null

F.a();//a() F() --> Function.prototype --> Object.prototype

F.b();//b() F() --> Function.prototype

# 变量提升与函数提升

## 变量声明提升

·通过var定义（声明）的变量，在定义语句之前就可以访问到

·值：undefined

**var** a = 3;

**function** fn() {

console.log(a);

**var** a = 4;

}

fn();//输出undefined，因为函数里面先声明var a，然后在执行输出语句，再a = 4

## 函数声明提升

·通过function声明的函数，在之前就可以直接调用

·值：函数定义（对象）

# 执行上下文

## 代码分类（位置）

·全局代码

·函数（局部）代码

## 全局执行上下文

·在执行全局代码前将window确定为全局执行上下文

·对全局数据进行预处理

- var定义的全局变量 --> undefined，添加为window的属性

- function声明的全局函数 --> 赋值(fun)，添加为window的方法

- this --> 赋值(window)

·开始执行全局代码

## 函数执行上下文

·在调用函数时，准备执行函数体之前，创建对应的函数执行上下文对象

·对局部数据进行预处理

- 形参变量 --> 赋值（实参）--> 添加为执行上下文的属性

- arguments --> 赋值（实参列表），添加为执行上下文的属性

- var定义的局部变量 --> undefined，添加为执行上下文的属性

- function声明的函数 --> 赋值(fun)，添加为执行上下文的方法

- this --> 赋值（调用函数的对象）

·开始执行函数体代码

## 执行上下文栈

·在全局代码执行前，JS引擎就会创建一个栈来存储管理所有执行的上下文对象

·在全局执行上下文（window）确定后，将其添加到栈中（压栈）

·在函数执行上下文创建后，将其添加到栈中（压栈）

·在当前函数执行完后，将栈顶的对象移除（出栈）

·当所有的代码执行完后，栈中只剩下window

## 面试题

console.log("全局开始"+ i);

**var** i = 1;

foo(i);

**function** foo(i) {

**if** (i == 4) {

**return**;

}

console.log("foo开始" + i);

foo(i + 1);

console.log("foo结束"+ i);

}

console.log("全局结束"+i)

依次输出：



整个过程中产生了5个执行上下文

Window 1 2 3 4

**function** a() {

}

**var** a;

console.log(**typeof** a);//function 先变量提升，然后函数提升

**if** (!(b **in** window)) {//b in window返回true，!之后不执行if语句

**var** b = 1;

}

console.log(b);//undefined

**var** c = 1;

**function** c(c) {

console.log(c);

}

c(2);//报错，先变量提升var c，然后函数提升c，然后把1赋值为c，此时c不是函数

# 作用域与作用域链

## 理解

·就是一块“地盘”，一个代码段所在的区域

·它是静态的（相对于上下文对象），在编写代码时就确定了

## 分类

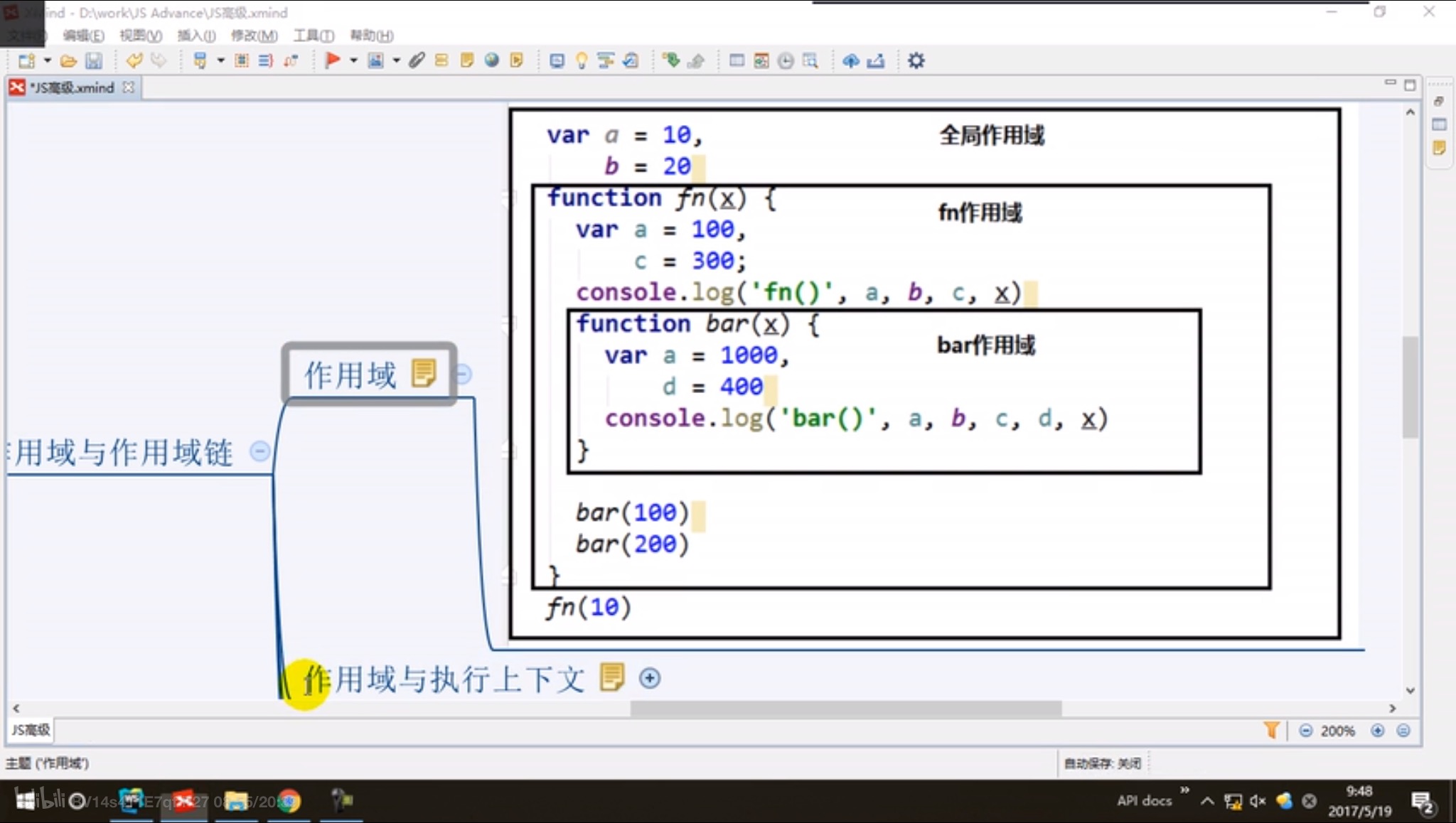
·全局作用域

·函数作用域

·没有块作用域（ES6以上才有）

## 作用

·隔离变量，不同作用域下同名变量不会有冲突



# 作用域与执行上下文

## 区别1

·全局作用域之外，每个函数都会创建自己的作用域，作用域在函数定义时就已经确定了，而不是在函数调用时。

·全局执行上下文环境是在全局作用域确定之后，JS代码马上执行之前创建

·函数执行上下文环境是在调用函数时，函数体代码执行之前创建

## 区别2

·作用域是静态的，只要函数定义好了就一直存在，且不会再变化

·执行上下文是动态的，调用函数时创建，函数调用结束时上下文环境就会自动释放

## 联系

·执行上下文（对象）是从属于所在的作用域

·全局上下文环境 --> 全局作用域

·函数上下文环境 --> 对应的函数作用域

# 作用域与作用域链

## 理解

·多个上下级关系的作用域形成的链，它的方向是由内向外的

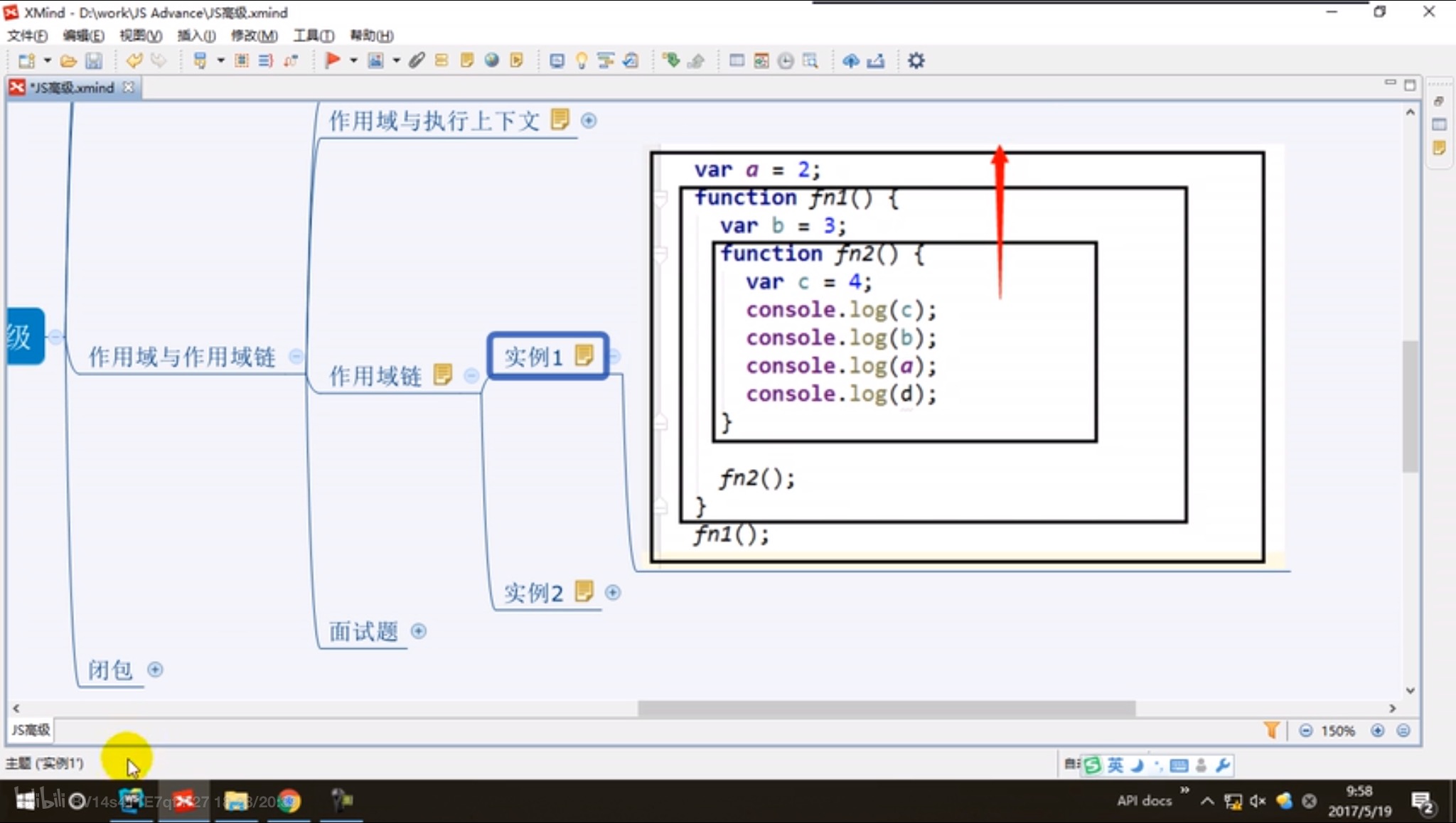
·查找变量时就是沿着作用域链来查找的

## 查找一个变量的查找规则

·在当前作用域下的执行上下文中查找对应的属性，如果有直接返回，否则进入上一级作用域

·在上一级作用域的执行上下文中查找对应的属性，如果有直接返回，否则进入上一级作用域

·重复执行以上操作，直到全局作用域，如果还找不到就抛出异常



## 面试题

**var** x = 10;

**function** fn() {

console.log(x);

}

**function** show(f) {

**var** x = 20;

f();

}

show(fn);//10 ，作用域是静态的，只要函数定义好了就一直存在，且不会再变化

**var** fn = **function** () {

console.log(fn);

};

fn();

输出：



**var** obj = {

fn2:**function** () {

console.log(fn2);//从本函数作用域中找fn2，未找到就跳出本作用域，到全局作用域中找fn2

console.log(**this**.fn2);//直接从调用函数的对象中找fn2

}

};

obj.fn2();//报错

# 闭包

//点击某个按钮，提示点击的是哪个按钮

//写法1

**var** btns = document.getElementsByTagName("button");

//遍历加监听

**for** (**var** i = 0,length = btns.length; i < length; i++) {//这样写提高性能,这个for循环里声明的i是全局变量

**var** btn = btns[i];

btn.index = i;//将i存储到每个btn的index属性

**var** b = "试试";//这里声明的b也是全局变量

btn.onclick = **function** () {

alert("按钮"+(**this**.index+1));

}

}

console.log(i,b);

//写法2

//利用闭包

**for** (**var** i = 0,length = btns.length; i < length; i++) {//这样写提高性能,这个for循环里声明的i是全局变量

(**function** (i) {//这里的i是局部变量

**var** btn = btns[i];

btn.onclick = **function** () {

alert("按钮"+(i+1));

}

})(i)//这里的i是全局变量

}

上面的闭包在执行立即执行函数时产生，并且会一直保存引用的参数i，

## 如何产生闭包？

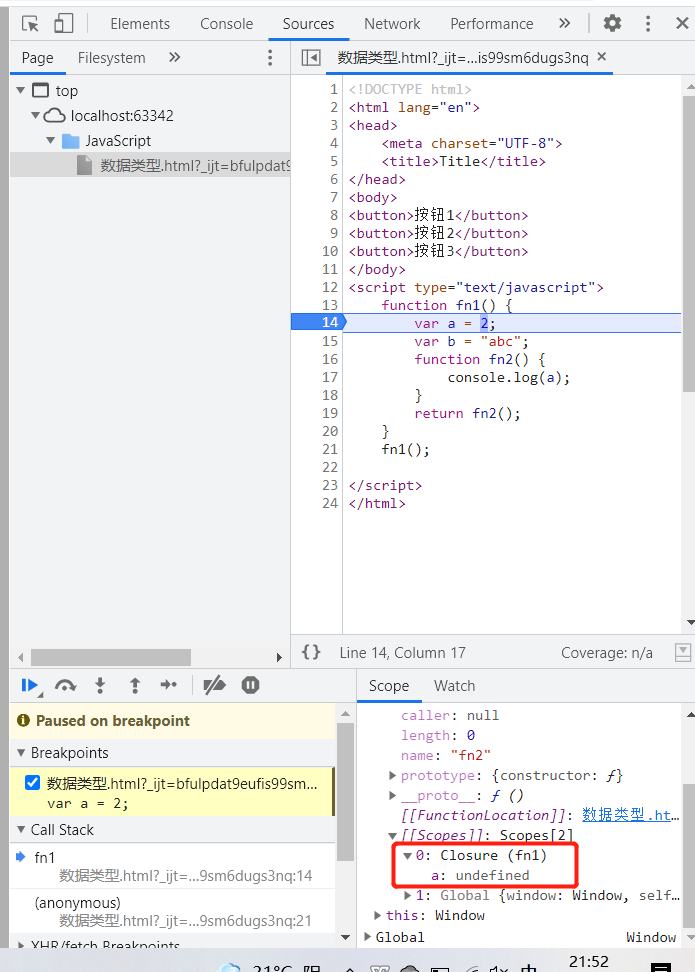
·当一个嵌套的内部（子）函数引用了嵌套的外部（父）函数的变量（或函数）时，执行外部函数时会产生闭包

## 什么是闭包？

·理解一：闭包是嵌套的内部函数

·理解二：包含被引用变量（或函数）的对象

·注意：闭包存在于嵌套的内部函数中



## 产生闭包的条件

·函数嵌套

·内部函数引用了外部函数的数据（变量/函数）

**function** fn1() {

**var** a = 2;

**var** b = "abc";

**function** fn2() {//函数提升后就会产生闭包（不用调用函数）

console.log(a);

}

**return** fn2();

}

fn1();

## 常见的闭包

### 将函数作为另外一个函数的返回值

**function** fn1() {

**var** a = 2;

**function** fn2() {

a++;

console.log(a);

}

**return** fn2;

}

**var** f = fn1();//产生了闭包，但没有执行fn2

f();//3

f();//4

**var** f1 = fn1();//产生了闭包，但没有执行fn2

f1();//3

f1();//4

f();//5

f1();//5

### 将函数作为实参传递给另一个函数调用

//延迟显示

**function** showDelay(msg,time) {//传入延迟显示的信息和时间

setTimeout(**function** () {

alert(msg);

},time);

}

showDelay("你好",2000);

## 闭包的作用

·使函数内部的变量在函数执行完后，仍然存活在内存中（延长了局部变量的生命周期）

·让函数外部可以操作（读写）到函数内部的数据（变量/函数）

## 函数执行完后，函数内部声明的局部变量是否还存在？

·一般情况下不会存在

·如果有内嵌函数引用局部变量并且函数提升后，形成闭包，则会存在在闭包里面

## 在函数外部能直接访问函数内部的局部变量吗?

·不能

·但是我们可以通过闭包让外部操作它

## 闭包的生命周期

·产生

- 在嵌套内部函数定义执行完时就产生了（不是在调用）

·死亡

- 在嵌套的内部函数成为垃圾对象时

**function** fn1() {

//此时闭包就已经产生了（函数提升，内部函数对象已经创建）

**var** a = 2;

**function** fn2() {

a++;

console.log(a);

}

**return** fn2;

}

**var** f = fn1();//产生了闭包，但没有执行fn2

f();//3

f();//4

f = **null**;//没有变量引用，成为垃圾对象，闭包死亡

## 闭包应用

### 定义JS模块

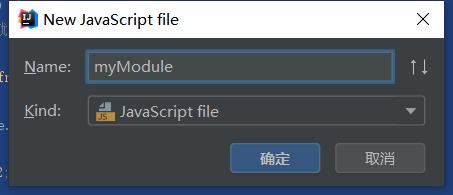
·具有特定功能的JS文件

·将所有的数据和功能都封装在一个函数内部（私有的）

·只向外暴露一个包信n个方法的对象或函数

·模块的使用者，只需要通过模块暴露的对象调用方法来实现对应的功能

1.创建JS文件



里面写模块代码

**function** myModule() {

//私有数据

**var** msg = "Hello World";

**function** doSomething() {

console.log("doSomething()" + msg.toUpperCase());//全部变成大写

}

**function** doOtherthing() {

console.log("doOtherthing()" + msg.toLowerCase());//全部变成小写

}

//向外暴露的对象（给外部使用的方法）

**return** {//将可以调用的函数返回

doSomething: doSomething,

doOtherthing: doOtherthing

}

}

(**function** myModule2() {

//私有数据

**var** msg = "Hello World";

**function** doSomething() {

console.log("doSomething()" + msg.toUpperCase());//全部变成大写

}

**function** doOtherthing() {

console.log("doOtherthing()" + msg.toLowerCase());//全部变成小写

}

//向外暴露的对象（给外部使用的方法）

window.myModule2 = {//将可以调用的函数赋值为window的属性

doSomething: doSomething,

doOtherthing: doOtherthing

}

})();

//一般用这种写法，好处：代码压缩时

(**function** myModule3(window) {

//私有数据

**var** msg = "Hello World";

**function** doSomething() {

console.log("doSomething()" + msg.toUpperCase());//全部变成大写

}

**function** doOtherthing() {

console.log("doOtherthing()" + msg.toLowerCase());//全部变成小写

}

//向外暴露的对象（给外部使用的方法）

window.myModule3 = {//将可以调用的函数赋值为window的属性

doSomething: doSomething,

doOtherthing: doOtherthing

}

})(window);

使用者调用

<script type="text/javascript">

**var** module = myModule();

module.doSomething();

module.doOtherthing();

//直接用

myModule2.doSomething();

myModule2.doOtherthing();

//直接用

myModule3.doSomething();

myModule3.doOtherthing();

</script>

## 闭包的缺点及解决

### 缺点

·函数执行完后，函数内的局部变量没有释放，占用内存时间会变长

·容易造成内存泄露

### 解决

·能不用闭包就不用

·及时释放（让内部函数成为垃圾对象 --> 回收闭包）

## 内存溢出与内存泄露

### 内存溢出

·一种程序运行出现的错误

·当程序运行需要的内存超过剩余内存时，就会抛出内存溢出错误

//内存溢出

**var** obj = {};

**for** (**var** i = 0; i < 10000; i++) {

obj[i] = **new** Array(100000);

}



### 内存泄露

·占用的内存没有及时释放

·内存泄露积累多了就容易导致内存溢出

·常见的内存溢出

- 意外的全局变量

//意外的全局变量

**function** fn() {

a = 3;//实际上使用的是全局变量中的a，需要var声明才是局部变量

console.log(a);//3

}

fn();

console.log(a);//3

- 没有及时清理计时器或回调函数

//没有及时清理计时器

**var** timer = setInterval(**function** () {//启动循环定时器后不清理

console.log("\_\_\_\_\_\_")

},1000);

//不用的时候关闭定时器防止内存泄露

clearInterval(timer);

- 闭包

## 面试题

**var** name = "The window";

**var** object = {

name : "My Object",

getNameFunc : **function** () {

**return function** () {

**return this**.name;

};

}

};

alert(object.getNameFunc()());//The window 没有闭包，此时this是window

**var** name2 = "The window";

**var** object2 = {

name2 : "My Object",

getNameFunc : **function** () {

**var** that = **this**;

**return function** () {

**return** that.name2;

};

}

};

alert(object2.getNameFunc()());//My Object 有闭包，此时this是object2

**function** fun(n,o) {

console.log(o);

**return** {

fun:**function** (m) {

**return** fun(m,n);

}

};

}

**var** a = fun(0);//undefined 只传了n，没有传o，输出undefined，此时a为一个对象，里面有fun属性，是一个函数，返回并调用fun(m,n)函数，并产生闭包，闭包的属性为n,值为0,没有释放

a.fun(1);//0 a调用fun(1)，然后又调用了fun(1,0)方法，输出0,并产生新的闭包，闭包的属性为n,值为1,后面没有继续调用，闭包释放

a.fun(2);//0 a调用fun(2)，然后又调用了fun(2,0)方法，输出0,并产生新的闭包，闭包的属性为n,值为1,后面没有继续调用，闭包释放

a.fun(3);//0 a调用fun(3)，然后又调用了fun(3,0)方法，输出0,并产生新的闭包，闭包的属性为n,值为1,后面没有继续调用，闭包释放

console.log("-------------");

**var** b = fun(0).fun(1).fun(2).fun(3);//undefined 0 1 2 调用fun(0)，输出undefined，返回值一个对象，里面有fun属性，是一个函数，返回并调用fun(m,n)函数，并产生闭包，闭包的属性为n,值为0,没有释放

// 然后调用fun(1)，再调用fun(1,0)方法,输出0，并产生新的闭包，闭包的属性为n,值为1,没有释放

// 然后继续调用fun(2)，再调用fun(2,1)方法，输出1，并产生新的闭包，闭包的属性为n,值为2,没有释放

//然后继续调用fun(3)，再调用fun(3,2)方法，输出2，并产生新的闭包，闭包的属性为n,值为3,没有释放

console.log("-------------");

**var** c = fun(0).fun(1);//undefined 0 用fun(0)，输出undefined，返回值一个对象，里面有fun属性，是一个函数，返回并调用fun(m,n)函数，并产生闭包，闭包的属性为n,值为0,没有释放

// 然后调用fun(1)，再调用fun(1,0)方法,输出0，并产生新的闭包，闭包的属性为n,值为1,没有释放

c.fun(2);//1 调用fun(2)，再调用fun(2,1)方法，输出1，并产生新的闭包，闭包的属性为n,值为2,被释放

c.fun(3);//1 调用fun(3)，再调用fun(3,1)方法，输出1，并产生新的闭包，闭包的属性为n,值为3,被释放

# 对象创建模式

## Object构造函数模式

·套路：先创建Object对象，再动态添加属性/方法

·适用场景：起始时不确定对象内部数据

·缺点：语句太多

**var** p = **new** Object();

p.name = "Tom";

p.age = 12;

p.setName = **function** (name) {

**this**.name = name;

};

p.setName("jack");

console.log(p);

## 对象字面量模式

·套路：使用{}创建对象，同时指定属性/方法

·适用场景：起始时对象内部数据是确定的

·缺点：如果创建多个对象，有重复代码

**var** p = {

name : "Tom",

age : 12,

setName : **function** (name) {

**this**.name = name;

}

};

p.setName("jack");

console.log(p);

## 工厂模式

·返回一个对象的函数 --> 工厂函数

·套路：通过工厂函数动态创建对象并返回

·使用场景：需要创建多个对象

·缺点：对象没有一个具体的类型，都是Object类型

**function** createPerson(name, age) {

**var** obj = {

name : name,

age : age,

setName : **function** (name) {

**this**.name = name;

}

};

**return** obj;

}

**var** p1 = createPerson("张三",17);

**var** p2 = createPerson("李四",19);

p1.setName("jack");

console.log(p1,p2);

## 自定义构造函数模式

·套路：自定义构造函数，通过new创建对象

·适用场景：需要创建多个类型确定的对象

·缺点：每个对象都有相同的数据，浪费内存

**function** Person(name, age) {

**this**.name = name;

**this**.age = age;

**this**.setName = **function** (name) {

**this**.name = name;

};

}

**var** p1 = **new** Person("张三",17);

**var** p2 = **new** Person("李四",19);

p1.setName("jack");

console.log(p1,p2);

## 构造函数+原型的组合模式

·套路：自定义构造函数，属性在函数中初始化，方法添加到原型上

·使用场景:需要创建多个类型确定的对象

**function** Person(name, age) {

**this**.name = name;

**this**.age = age;

}

Person.prototype.setName = **function** (name) {

**this**.name = name;

};

**var** p1 = **new** Person("张三",17);

**var** p2 = **new** Person("李四",19);

p1.setName("jack");

console.log(p1,p2);

# 继承模式

## 原型链继承

子类型的原型为父类型的一个实例对象

### 方法

·定义父类型构造函数

·给父类型的原型添加方法

·定义子类型的构造函数

·创建父类型的对象赋值给子类型的原型

·将子类型原型的构造属性设置为子类型,修正constructor属性

·给子类型原型添加方法

·创建子类型的对象：可以调用父类型的方法

//父类型

**function** Super1() {

**this**.supProp = "Super property";

}

//给父类型的原型添加方法

Super1.prototype.showSuperProp = **function** () {

console.log(**this**.supProp);

};

//子类型

**function** Sub() {

**this**.subProp = "sub property";

}

//创建父类型的对象赋值给子类型的原型

Sub.prototype = **new** Super1();

//将子类型原型的构造属性设置为子类型

Sub.prototype.constructor = Sub;

//给子类型的原型添加方法

Sub.prototype.showSubProp = **function** () {

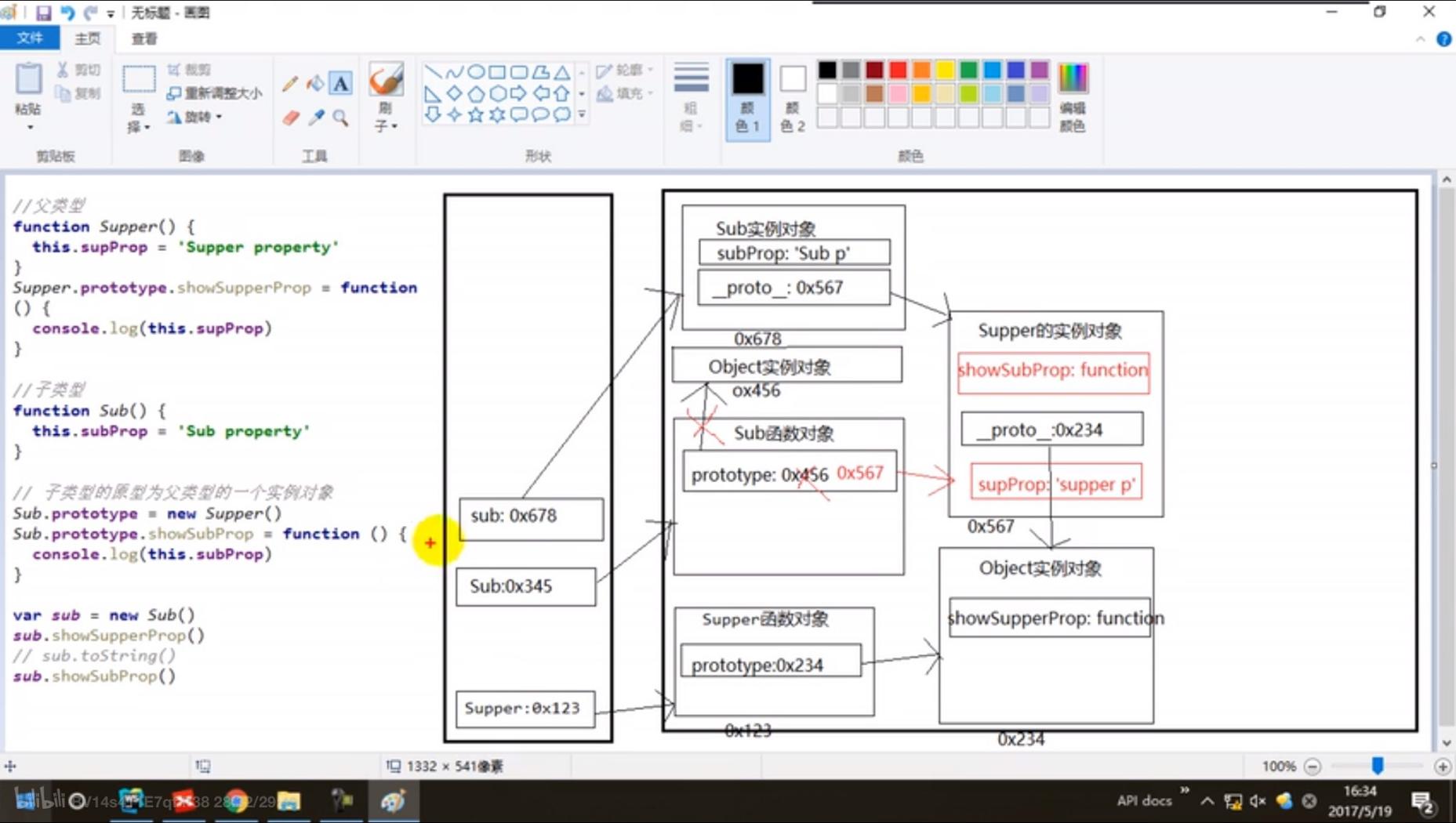
console.log(**this**.subProp);

};

**var** sub = **new** Sub();//创建子类型的对象

sub.showSuperProp();//可以调用父类型的方法

sub.showSubProp();//也可以调用自己的方法



## 借用构造函数继承（假的继承）

在子类型构造函数中通用super()调用父类型构造函数

### 方法

·定义父类型构造函数

·定义子类型构造函数

·在子类型构造函数中调用父类型构造

//父类型

**function** Person(name, age) {

**this**.name = name;

**this**.age = age;

}

//子类型

**function** Student(name, age, sex) {

//在子类型构造函数中调用父类型构造

Person.call(**this**,name,age);

//上面语句相当于this.Person(name,age)，与下两条注释语句相同

// this.name = name;

// this.age = age;

**this**.sex = sex;

}

**var** s = **new** Student("张三",20,"男");

console.log(s.name,s.age,s.sex);

## 组合继承

·利用原型链实现对父类型对象的方法继承

·利用super()借用父类型构建函数初始化相同属性

//父类型

**function** Person(name, age) {

**this**.name = name;

**this**.age = age;

}

//给父类型的原型添加方法

Person.prototype.setName = **function** (name) {

**this**.name = name;

};

//子类型

**function** Student(name, age, sex) {

//在子类型构造函数中调用父类型构造，为了得到属性

Person.call(**this**,name,age);

//上面语句相当于this.Person(name,age)，与下两条注释语句相同

// this.name = name;

// this.age = age;

**this**.sex = sex;

}

//子类型的原型等于父类型的实例,不用传参，为了看到父类型的方法

Student.prototype = **new** Person();

//将子类型原型的构造属性设置为子类型,修正constructor属性

Student.prototype.constructor = Student;

//给子类型的原型添加方法

Student.prototype.setSex = **function** (sex) {

**this**.sex = sex;

};

**var** s = **new** Student("张三",20,"男");

s.setSex("女");

s.setName("李四");

console.log(s.name,s.age,s.sex);

# 进程与线程

## 进程

·程序的一次执行，它占用一片独有的内存空间

·可以通过windows任务管理器查看进程

·有的程序多进程，有的程序单进程

## 线程

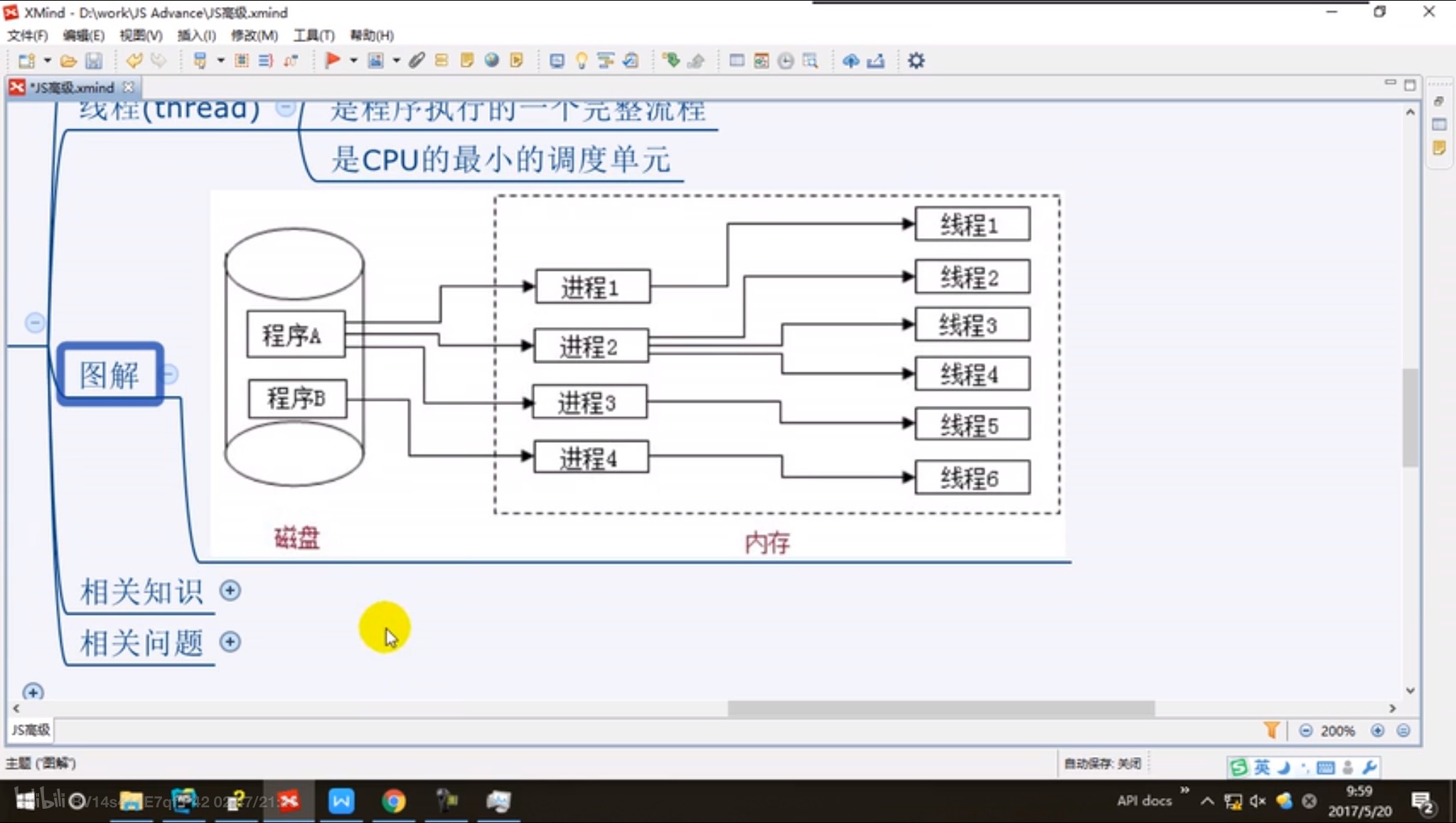
·是进程内的一个独立执行单元

·是程序执行的一个完整流程

·是CPU的最小调度单元

·一个进程可以有多个线程

## 图解



## 相关知识

·应用程序必须运行在某个进程的某个线程上

·一个进程中至少有一个运行线程：主线程，进程启动后自动创建

·一个进程中也可以使用运行多个线程，我们会说程序是多线程运行的

·一个进程内的数据可以供其中的多个线程直接共享

·多个进程之间的数据是不能直接共享的

·线程池（thread pool）：保存多个线程对象的容器，实现线程对象的反复利用

## 相关问题

### 什么是多进程与多线程

·多进程运行

- 一应用程序可以同时启动多个实例运行

·多线程

- 在一个进程内，同时有多个线程运行

### 单线程与多线程的比较

·多线程

- 优点

-- 有效提升CPU的利用率

- 缺点

-- 创建多线程开销

-- 线程间切换开销

-- 死锁与状态同步问题

·单线程

- 优点

-- 顺序编程简单易懂

-- 缺点

-- 效率低

### JS是单线程还是多线程

·JS是单线程运行的

·但使用H5中的Web Workers可以多线程运行

### 浏览器运行是单线程还是多线程

·都是多线程运行的

### 浏览器运行是单进程还是多进程

·有的是单进程

·有的是多进程

# 浏览器内核

·支撑浏览器运行的最核心的程序

·不同的浏览器可能不一样

- Chrome,Safari:

-- webkit

- firefox

-- Gecko

- IE

-- Trident

- 360，搜狗等国内浏览器

-- Trident + webkit

·内核由很多模块组成

**主线程**

- JS引擎模块

-- 负责JS程序的编译与运行

- html,css文档解析模块

-- 负责页面文本的解析

- DOM/CSS模块

-- 负责dom/css在内存中的相关处理

- 布局和渲染模块

-- 负责页面的布局和效果的绘制（内存中的对象）

等......

**分线程**

- 定时器模块

-- 负责定时器的管理

- 事件响应模块

-- 负责事件的管理

- 网络请求模块

-- 负责ajax请求

# 定时器

## 定时器真是定时执行吗？

·定时器并不能保证真正的定时执行

·一般会延迟一点（可以接受）

**var** start = Date.now();

setTimeout(**function** () {

console.log("定时器执行了",Date.now()-start);//200多

},200);

·也有可能延迟很长时间（不能接受）

**var** start = Date.now();

//做一个长时间的工作

**for** (**var** i = 0; i < 100000000; i++) {

}

setTimeout(**function** () {

console.log("定时器执行了",Date.now()-start);//400左右

},200);

## 定时器回调函数是在分线程执行的吗？

·在主线程执行的，JS是单线程的

## 定时器是如何实现的？

·事件循环模型

# JS是单线程执行的

## 如何证明JS执行是单线程的？

·setTimeout()的回调函数是在主线程执行的

·定时器回调函数只有在运行栈中的代码全部执行完后才有可能执行

setTimeout(**function** () {

console.log("定时器执行10000");

},10000);

setTimeout(**function** () {

console.log("定时器执行1000");

},1000);

setTimeout(**function** () {

console.log("定时器执行0");

},0);

**function** fn() {

console.log("fn()");

}

fn();

console.log("alter之前");

alert("-------");//暂停当前主线程的执行，同时暂停计时（我测试的并没有暂停计时，我的问题？），点击确定后，恢复程序执行和计时

console.log("alter之后");



## 为什么JS要用单线程模式

·作为浏览器脚本语言，JS的主要用途是与用户互动，以及操作DOM

·这绝对了它只能是单线程，否则会带来很复杂的同步问题

## 代码的分类

·初始化代码（同步代码）

- 包含绑定dom事件监听，设置定时器，发生ajax请求的代码

·回调代码（异步代码）

- 处理回调逻辑

## JS引擎执行代码的基本流程

·先执行初始化代码：包含一些特别的代码

·设置定时器

·绑定监听

·发送ajax请求

·后面在某个时刻才会执行回调代码 回调函数（异步执行）

·初始化代码 --> 回调代码

# 事件循环模型

## 模型的2个重要组成部分

·事件（定时器/DOM事件/Ajax）管理模块

·回调队列

## 模型的运转流程

·执行初始化代码，将事件回调函数交给对应模块管理

·当事件发生时，管理模块会将回调函数及其数据添加到回调列队中

·只有初始化代码执行完后（可能要一定时间），才会遍历读取回调队列中的回调函数执行

## 相关重要概念

·执行栈(execution stack)

- 所有的代码都是在此空间中执行的

·浏览器内核(browser core)

- JS引擎模块（在主线程处理）

- 其它模块（在主/分线程处理）

·callback queue

- 任务队列(task queue)

- 消息队列(message queue)

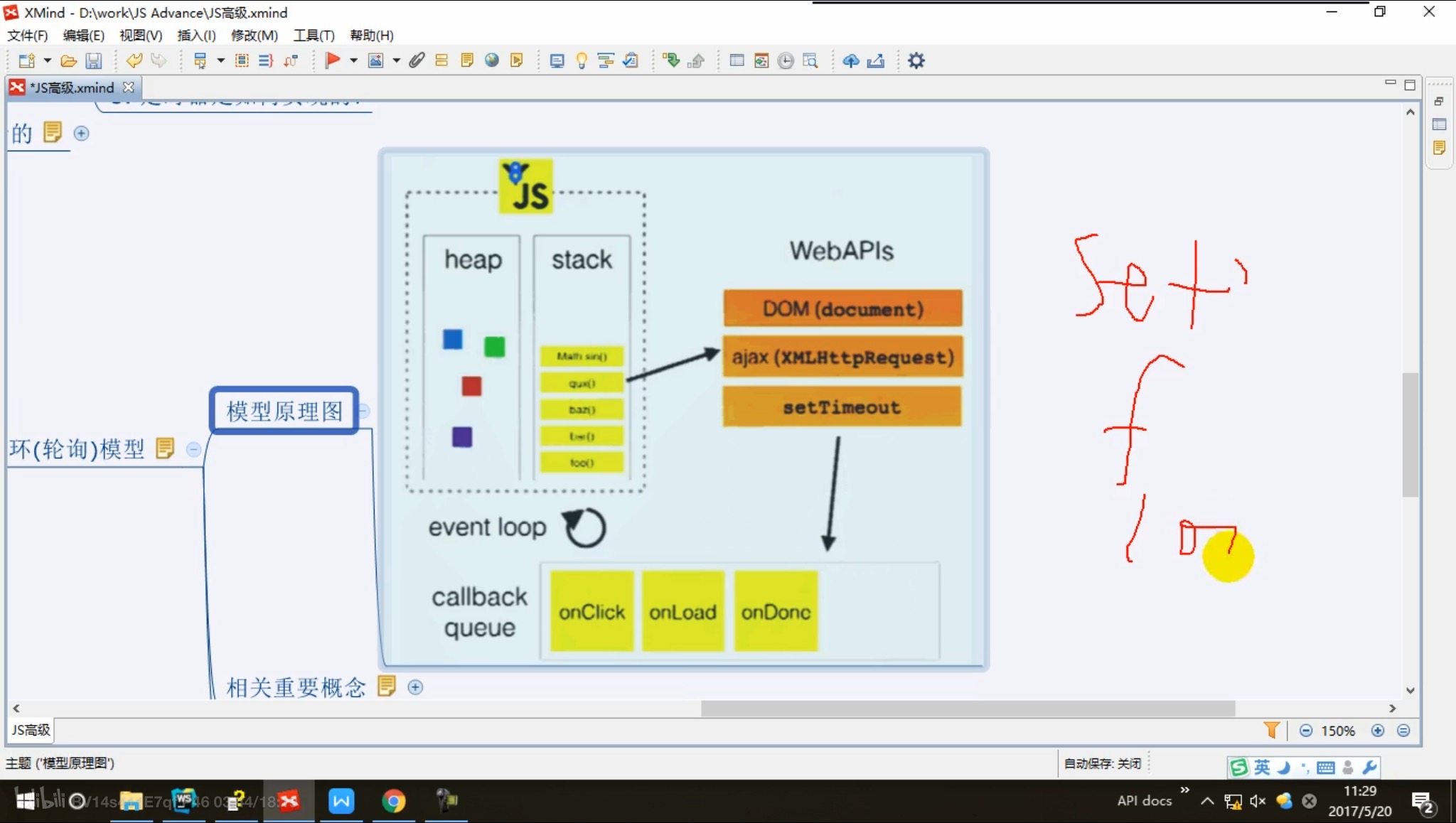
- 事件队列(event queue)

·事件轮询(event loop)

- 从任务队列中循环取出回调函数放入执行栈中处理（一个接一个）

·事件驱动模型

·请求响应模型



# Web Workers多线程

·H5规范提供了JS分线程的实现，取名为：Web Workers，我们可以将大计算量的代码交由web Worker运行而不冻结用户界面

·相关API

- Worker

-- 构造函数。加载分线程执行的JS文件

- Worker.prototype.onmessage

-- 用于接受另一个线程的回调函数

- Worker.prototype.postMessage

-- 向另一个线程发生信息

·缺点

- 慢

- 子线程完全受主线程控制

- Worker内代码不能操作DOM（更新UI）

- 不能跨域加载JS

- 并没有改变JS单线程的本质

- 不是每个浏览器都支持这个新特性

·使用

- 创建在分线程执行的JS文件

- 在主线程中的JS中发消息并设置回调

## 斐波那契数列（单线程）

F(0) = 0,F(1) = 1,...F(n) = F(n-1)+F(n-2)

**function** fibonacci(n) {//递归调用消耗内存，效率低，计算慢，计算过程不能操作其他东西

**if** (n === 0) {

**return** 0;

}

**return** n<=2 ? 1 : fibonacci(n-1) + fibonacci(n-2);

}

**var** input = document.getElementById("number");

**var** btn = document.getElementById("btn");

btn.onclick = **function** () {

**var** number = input.value;

**var** result = fibonacci(number);

alert(result);

}

## 斐波那契数列（多线程）

### 主线程的JS

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Title</title>

</head>

<body>

<input type="text" placeholder="数值" id="number">

<button id="btn">计算</button>

</body>

<script type="text/javascript">

**var** input = document.getElementById("number");

**var** btn = document.getElementById("btn");

btn.onclick = **function** () {

**var** number = input.value;

//创建一个worker对象，并将js文件传入

**var** worker = **new** Worker("worker.js");

//绑定接收消息的监听

worker.onmessage = **function** (ev) {

console.log("主线程接收分线程返回的数据" + ev.data);

alert(ev.data);

};

//向分线程发送消息

worker.postMessage(number);

console.log("主线程向分线程发生数据：" + number);

}

</script>

</html>

### 分线程的JS

**var** onmessage = **function** (ev) {

**var** number = ev.data;

console.log("分线程接收到主线程发送的数据：" + number);

//计算

**var** result = fibonacci(number);

postMessage(result);

console.log("分线程向主线程返回数据：" + result);

alert(result);//报错，alert是window的方法，在分线程不能调用

};

**function** fibonacci(n) {//递归调用消耗内存，效率低，计算慢，计算过程不能操作其他东西

**if** (n === 0) {

**return** 0;

}

**return** n<=2 ? 1 : fibonacci(n-1) + fibonacci(n-2);

}

console.log(**this**);//返回的对象是DedicatedWorkerGlobalScope

//分线程中的全局对象不再是window，所以在分线程中不可能更新界面

