张东18301092802

准备三样东西:

1. 犀牛书——JavaScript权威指南

2. 微信公众号: 前端大全

3. 打印上一届笔记: 预习，上课跟着看

# 正课:

1. 正则表达式:

2. String的正则API:

## 1. 正则表达式: 见ftp中regexp.pdf

## 2. String的正则API

### 1.查找: 4个场景

1、查找一个固定的敏感词出现的位置:

回顾: **str.indexOf()**

var i=str.indexOf("敏感词",start)

在str中，从start位置开始，查找下一个“敏感词”的位置

省略第二个参数start，默认从0位置开始查找

返回值: 如果找到，返回敏感词第一个字的位置

如果找不到，返回-1

问题: 只能查找一种固定的敏感词

2、模糊查找符合正则表达式要求的敏感词:

var i=**str.search**(/正则/);

在str中查找第一个符合正则要求的敏感词的位置

返回值: 如果找到，返回敏感词第一个字的位置

如果找不到，返回-1

问题1: 正则默认都是区分大小写的

解决: 在第二个/后加后缀i, ignore

问题2: 只能返回位置i，无法返回敏感词的内容

3、 查询敏感词的内容: 2种情况:

1. 只查看第一个敏感词的内容和位置:

var arr=**str.match**(/正则/i);

在str中查找第一个符合正则要求的敏感词的内容和位置

返回值: 如果找到: arr:[ 0:"敏感词", index: 位置i ]

补充: js中其实一切都是关联数组

如果找不到: 返回null

强调: 如果一个函数有可能返回null，都要先验证不是null，再使用。

问题: 正则表达式默认只找到第一个就退出

解决: 在第二个/后加后缀g, global(全部)

2. 查找所有敏感词的内容:

var arr=str.match(/正则/ig)

在str中查找所有符合正则要求的敏感词的内容

返回值: 如果找到: arr: [ 敏感词1, 敏感词2,... ]

如果没找到，返回null

问题: 只能获得内容，无法获得位置

4、查找每个敏感词的内容和位置: reg.exec()

### 2.替换: 2种:

1. 简单替换: 将所有敏感词都替换为统一的新值:

str=str.**replace**(/正则/ig,"新值")

查找str中符合正则要求的所有敏感词，替换为统一的新值

强调: 所有字符串API是无权直接修改原字符串，总是返回新字符串作为处 理结果，所以必须用变量接住返回的新字符串。

2. 高级替换: 可根据每次找到的敏感词不同，动态生成替换的新值

str=str.replace(/正则/ig,function(kword){

return 新值

});

查找str中每个符合正则要求的敏感词，每找到一个敏感词，就自动调用一 次回调函数。每次调用时，都自动传入本次的敏感词作为参数kword，再将 返回的新值，替换回字符串中。

衍生操作: 删除找到的敏感词，其实就是替换为""

### 3.切割

切割: 将一个字符串按指定的分隔符切割为多段子字符串:

2种:

1. 简单切割: 切割符是固定的:

回顾: var arr=str.split("切割符")

2. 复杂切割: 切割符不是固定的

var arr=**str.split**(/正则/)

# 正课:

1. RegExp对象:

2. Function:

3. OOP:

## 1. RegExp对象: 正则表达式(Regular Expression)

什么是: 在程序中专门保存一条正则表达式，并提供用正则表达式执行验证和查找功能的API 的对象

何时: 只要在程序中保存和使用正则表达式时，都要先创建正则表达式对象

如何:

创建正则表达式对象: 2种

1. 用//直接量方式创建: var reg=/正则/ig;

何时: 如果正则表达式不需要动态生成时

2. 用new创建: var reg=**new RegExp**("正则"[,"ig"])

何时: 如果正则表达式需要用程序动态生成时

### API:

### 1. 验证:

var bool=**reg.test(str)**

用正则表达式reg检查一个字符串str的格式是否符合要求。

返回值: true/false

问题: 正则表达式，默认只要找到符合规则的内容，就返回true，不要求从头到尾完整匹配——所以，也有人用reg.test()做查找关键词

解决: 今后凡是验证必须前加^后加$，表示从头到尾完整匹配

### 2. **查找**敏感词:

**reg.exec(str)**

什么是: 查找str中每个符合reg要求的敏感词的内容和位置

何时: 只要既查找每个关键词的内容，又查找每个关键词的位置

如何:

var arr=reg.exec(str);

执行过程:

每次只能查找一个敏感词，

返回: arr: [ 0: "敏感词", index: 位置i ] ——同match不加g的情况

反复调用reg.exec可自动向后找！

找不到，返回null

## 2. Function:

什么是: 专门保存一段可重用的代码段的程序结构，再起一个名字

何时: 只要一段代码可能被反复使用，都要定义为函数

### 1.创建**函数**: 3种:

参考2\_function\_1.pdf

**1. 声明方式:**

function 函数名(形参列表){

函数体;

return 返回值;

}

问题: 会被声明提前

解决: 改声明方式为赋值方式

**2. 赋值方式:** var函数名=function (形参列表){

函数体;

return 返回值;

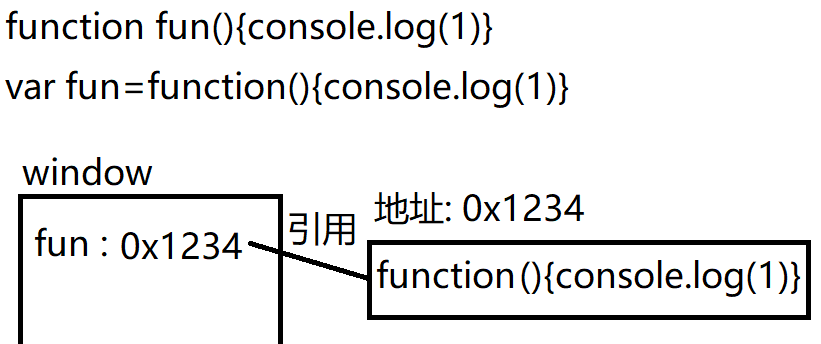
}

优点: 因为是赋值方式，不会被声明提前

揭示了js中函数的本质:

1. 函数其实也是一个保存一段代码的对象

2. 函数名其实只是一个普通的变量而已。函数名变量通过函数对象的地 址引用着函数对象。所以使用函数名变量等效于使用函数对象本身。



**3. 用new Function**构造函数创建函数对象:

var 函数名=new Function("形参1","形参2",...,"函数体")

### 2.重载:

什么是: 相同函数名，不同形参列表的多个函数，在调用时，可根据传入实参的不同，自动选择匹配的函数执行。

为什么: 减少API的个数，减轻调用者的负担

何时: 一件事，可能根据传入参数的不同，执行不同的操作时

比如: arr.splice():

删除元素: arr.splice(i,n)

插入元素: arr.splice(i,0,新值1,...)

替换元素: arr.splice(i,n,新值1,...)

如何:

问题: js语法默认不支持重载

因为js中不允许多个同名函数同时存在，如果多个同名函数同时存在，则只有最后一个生效

**解决: 借助arguments**

arguments: 每个函数中自带的，自动接收所有传入函数的参数值的类数组对象

类数组对象: 长得像数组的对象

vs 数组: 相同: 1. 下标, 2. length, 3. 遍历

不同: 类型不同！API不通用！

如何: 2步:

1. 仅定义一个函数，且不要定义形参

2. 在函数内判断arguments的length或内容，根据不同的参数，执行不同的操作

### 3.匿名函数:

什么是: 定义函数时，不指定函数名的函数

为什么: 节约内存

何时: 如果一个函数只使用一次，不会重用时，都要用匿名函数

如何: 2种；

1. 回调函数: 自己定义的函数，但是不是自己亲自调用，而是给别的函数去调用。

比如: arr.sort(function(a,b){return a-b})

setInterval/setTimeout(function(){...},ms)

str.replace(/正则/ig,function(kword){return 新值})

2. 匿名函数自调:

什么是: 定义函数后，立刻执行！

为什么: 避免使用全局变量，造成全局污染

何时: 几乎所有的js代码都要装在匿名函数自调中

如何: (function(){ ... })()

### 4.作用域(scope)/作用域链:

参考2\_function\_4.pdf

**闭包:** 参考2\_function\_4.ppt

## 3. OOP:

### 1. OOP: 面向对象编程

什么是对象: 程序中描述现实中一个具体事物的属性和功能的程序结构

什么是面向对象编程: 程序都是先将数据和功能定义在对象中，再按需使用对象的数据和功能

为什么: 便于大量数据和功能的维护和使用

如何: 三大特点:**封装，继承，多态**

### 封装:

什么是: 创建对象集中保存一个事物的属性和功能

为什么: 便于大量数据和功能的管理和使用

何时: 要面向对象编程，都要先封装对象，再按需使用对象的属性和功能

如何: 3种:

#### 1. 用{}直接量:

var 对象名={

属性名:属性值,

... : ... ,

方法名:function(){

... this.成员 ...

}

}

访问对象的成员: (属性和方法)

对象.属性名

对象.方法名()

问题: 对象的方法要访问对象自己的属性或方法，无法直接使用

为什么: 所有对象的属性和方法默认都不在作用域链当中。任何程序无权 自动进入对象中获取成员。

不好的解决: 在方法中: 对象名.成员名

为什么: 紧耦合，如果对象名发生变化，方法的内容必须同时修改才行。

**好的解决办法: this**

什么是this: 在函数调用时，**自动指向正在调用当前函数·前的对象**

何时: 只要对象自己的方法，要使用自己的成员，都要加this.

强调: 笔试时，this与定义时无关，只和调用时有关

全局函数调用，没有· this默认指window

#### 2. 用new创建: 2步:

1. 先创建空对象: var obj=new Object();

2. 向空对象中添加新属性: obj.新成员=值/function(){...}

js中对象的特点:

1. 对象创建完成后，依然可继续添加新成员

2. 访问对象中不存在的属性，不报错！而是返回undefined

3. 访问对象的成员，既可用.，又可用["成员名"]

选择: 如果成员名是变化的，就必须用[变量]

如果成员名可以写死，就可用.简化

揭示: 其实js底层，所有对象都是关联数组！

所以，对象也可用for in遍历其中的属性

前两种方式的问题: 一次只能创建一个对象

如果反复创建多个相同结构的对象时，代码重复很多

解决: 用构造函数创建对象

#### 3. 用构造函数反复创建统一类型的多个对象:

构造函数: 描述一类对象统一结构的函数

为什么: 重用创建对象的代码！

何时: 要反复创建多个相同结构的对象时

如何: 2步:

1. 定义构造函数:

function 类型名(形参1，形参2...){

//this->new->{}

this.属性名=值; //将来的对象中一定要有指定的"属性"

this.xxx = 值;

this.方法名=function(){//将来的对象中必须有指定的"方法"

... ...

}

}

2. 调用构造函数反复创建多个对象:

var obj=new Student(属性值,...)

new:

1. 创建一个新的空对象:{}

2. 自动设置本次创建的子对象，继承构造函数的原型对象。

3. 调用构造函数:

自动将构造函数中的this->new->{}

构造函数中的每一句话，都通过强行赋值的方式，为新对象{}添加 规定的新属性

4. 返回新对象地址

问题: 放在构造函数中的方法定义，会在每个子对象中重复创建函数对象，浪费内存！

解决:

### 继承:

什么是: 父对象的成员，子对象无需重复创建，就可直接使用

为什么: 代码重用，节约内存

何时: 只要多个子对象需要相同的成员时

如何:

原型对象: 专门保存所有子对象共有成员的父对象

不用创建，定义构造函数时，附赠的

构造函数.prototype -> 当前构造函数的原型对象

如何继承: new的第2步，自动设置本次创建的子对象，继承构造函数的原型对象。

每个子对象都有\_ \_proto\_ \_属性，指向自己的父级对象

**向原型对象中添加共有成员:**

构造函数.prototype.成员=值/function(){...}

自有属性和共有属性:

获取属性值, 没有任何差别

差别在: 修改时: 自有属性，可用子对象修改

共有属性，不能用某一个子对象修改

只能用原型对象修改

**内置对象的构造函数和原型对象:**

内置对象: ES标准中规定的，浏览器已经定义好的可直接使用的对象

包括: 11个

String Number Boolean Array RegExp Date Math Error

Function Object global

除了Math和global外，其余都是一种类型:都包含2部分组成:

1. 构造函数: 创建该类型的子对象

2. 原型对象: 保存该类型的所有子对象共有的成员

原型链: prototype chain

什么是: 由多级父对象逐级继承，形成的链式结构

2大作用:

1. 保存了当前对象可用的所有属性:

2. 控制着属性的使用顺序

### **多态:**

什么是: 一个函数在不同情况下表现出不同的状态

包括: 2种:

1. 重载: overload

2. 重写:override

什么是: 在子对象中定义和父对象同名的成员

何时: 只要从父对象继承来的成员不好用

如何:

# 正课:

1. 自定义继承:

2. 静态方法:

3. ES5:

## 1. 自定义继承:

见PDF: 1\_自定义继承.pdf

## 2. 静态方法:

什么是: 不需要创建子对象，就可用构造函数直接调用的方法

问题: 原型对象中的方法，只能被当前类型的子对象使用

何时: 2种:

1. 如果一个函数，不确定将来使用它的对象的类型时

2. 如果一个函数在调用时，不需要提前创建子对象

如何: 构造函数.静态方法=function(){ ... }

调用时: 构造函数.静态方法()

强调: 调用静态方法之前不需要创建子对象

定义实例方法：将方法添加到原型对象中

定义静态方法：将方法添加到构造函数对象上

## 3. ES5:

### 1. 严格模式:

什么是: 比普通js运行机制要求更严格的模式

为什么: js语言有广受诟病的很多缺陷

何时: 今后所有js，都要运行在严格模式下.

如何: 在当前代码段的顶部: **"use strict";**

严格模式要求:

1. 禁止给未声明的变量赋值:

2. 静默失败升级为错误:

静默失败: 执行不成功，也不报错

3. 禁用了arguments.callee:

arguments.callee专门在调用函数时，获得当前函数本身

何时: 专门用于递归调用中获得当前函数

为什么禁用: 不再建议使用递归

因为: 递归重复计算量太大！效率极低

解决: 几乎所有的递归都可用循环代替

4. 普通函数和匿名函数自调用中的this不再指向window

而是undefined

### 2. 保护对象:

问题: JS中对象毫无自保能力

解决: ES5提供了一系列措施保护对象

如何:

**1. 保护对象的单个属性**

ES5将对象的属性，分为2大类:

1. 命名属性: 可用.直接访问的属性

又分为: 2类:

1. 数据属性: 实际存储属性值的属性

2. 访问器属性: 不实际存储属性值，专门提供对另一个数据属性的自定义保护！

2. 内部属性: 实际存在，但不能用.访问到的属性

保护数据属性:

ES5将一个数据属性，又定义为一个缩微的小对象

属性: {

value: 实际存储属性值,

writable: true/false, //控制是否可修改

**enumerable: true/false**, //控制是否可被for in遍历

//只能防住for in，但是用.依然可访问

**configurable: true/false** //控制是否可删除该属性

//还控制是否可修改另外两个特性

//一旦改为false，不可逆！

//所以，修改其它两个开关时都会同时修改 configurable:false,作为双保险。

}

默认：三个开关都是true

删除属性：delete 对象.属性

获取一个属性的四大特性: **了解**

var miniObj=

Object.**getOwnPropertyDescriptor**(obj, "属性名");

修改一个属性的四大特性:

Object.defineProperty(obj,"属性名",{

开关: true/false,

... : ...

})

同时修改多个属性的四大特性:

**Object.defineProperties**(obj,{

属性名:{

开关:true/false,

... : ...

},

... : {

}

})

问题: 开关不能用自定义规则保护属性

解决: 用访问器属性

**访问器属性:**

何时: 只要用自定义规则保护对象属性时

如何:

前提: 先有一个隐藏的数据属性实际存储数据

再添加一个访问器属性来保护隐藏的数据属性:

Object.defineProperty(obj,"访问器属性名",{

get:function(){ return this.受保护的数据属性 },

set:function(value){

//value可接住将来要赋的新值

//如果value符合条件

//才this.受保护的数据属性=value

//否则

//报错

},

enumerable:true, //让访问器属性代替数据属性抛头露面

configurable:false

})

访问器属性何时起作用:

访问器属性的用法和普通属性是一样的

只不过:

1. 当试图获取访问器属性的值时，自动调用get()

2. 当试图修改访问器属性的值时，自动调用set()，并传入新值作为参数value

### **3. 保护对象的结构**

保护对象结构: 3个级别:

1. 防扩展: 禁止给当前对象添加新属性

Object.preventExtensions(obj)

原理：每个对象中都有一个内部属性：extensible:true，preventExtensitions()将对象的true改为false

2. 密封: 在兼具防扩展的同时，又禁止删除现有属性

Object.seal(obj)

禁止向obj中添加新属性

禁止删除obj中现有属性

原理: 1. extensible:true自动改为false

2. 自动将所有属性的configurable都设置为false

3. 冻结: 在兼具密封的基础上，又禁止修改属性值

Object.freeze(obj)

原理: 1. Object.seal(obj)

2. 又自动将所有属性的writable都设置为false

禁止向obj中添加新属性

禁止删除obj中现有属性

禁止修改属性值

**Object.create()**: 仅有父对象，也可创建子对象

何时: 如果没有构造函数也想创建子对象时

如何: var child=Object.create(father,{

//为child添加新自有属性

//语法相当于defineProperties

属性名:{

value:属性值,

开关: true/false,

... : ...

}

})

原理: 1. 创建一个空对象

2. 让空对象继承father

3. 为新对象添加自有属性

### 替换this:

何时: 如果一个函数调用时，其中的this不是想要的，就可替换为想要的对象.

如何: 3种:

1. 临时调用函数，并临时替换一次this:

call(): 要求传入函数的参数，是分别传入

apply(): 要求传入函数的参数，必须放在一个数组中整体传入。

何时: 如果参数值是放在一个数组中的

原理: 先打散数组为单个值，再注射进函数中

2. 基于原函数，创建一个一模一样的新函数，并永久绑定this:和部分参数值

var 新函数=原函数.bind(替换this的对象, 其他参数...)

1. 创建一个和原函数一模一样的新函数

2. 永久替换函数中的this为指定对象

何时: 替换回调函数中的this

### 数组API

**1. 查找元素**: 查找一个指定元素在数组中的位置

var i=arr.indexOf(elem[, starti ])

用法同str.indexOf()

**2. 判断:** 判断数组中的元素是否符合要求:

1. 判断数组中的元素是否都符合要求:

var bool=arr.every(function(val, i, arr){

//val->当前正在遍历到的元素值

//i->当前正在遍历的位置

//arr->当前正在遍历的数组对象

return 判断条件;

})

依次判断arr中每个元素，是否都符合“判断条件”的要求

执行过程: every自动拿着回调函数，在每个元素上执行一次

每次执行时，都自动传入: 当前元素值，当前位置, 当前数组对象

每次执行回调函数后，都返回对当前元素的判断结果。

只有所有元素经过判断都返回true时，every才返回true

2. 判断数组中是否包含符合要求的元素:

var bool=arr.some(function(elem,i,arr){

return 判断条件

})

**3. 遍历:** 依次对数组中每个元素执行相同的操作

1. 对原数组中的元素执行相同的操作：

arr.forEach(function(elem,i,arr){

操作

})

会自动拿着回调函数，在每个元素上执行一次！

2. 复制出原数组中的元素，执行相同操作后，放入新数组中，原数组保持不变。

var 新数组=arr.map(function(elem, i, arr){

return 新元素值;

})

map: 3件事:

1. 创建一个新的空数组

2. 自动在每个元素上调用一次回调函数，将返回的新值，放入新数组中相同位置

3. 返回新数组

**4. 过滤和汇总:**

1. 过滤: 复制出原数组中符合条件的元素，组成新数组返回

var 新数组=arr.filter(function(elem,i,arr){

return 判断条件;

})

执行过程: filter会自动拿着回调函数在每个元素上执行一次

每次执行时，如果当前元素符合“判断条件”要求，就复制到新数组中。

2. 汇总: 将数组中的所有元素，经过统计，得出一个最终结论

var result=arr.reduce(function(prev,elem,i,arr){

return 将当前元素的内容汇总到prev中

},start);

//start: 开始的基础值

//prev: 截止目前位置，之前的元素的临时汇总值

# 正课:

## 1. ES6

1. ES6: 不改变原理的基础上，尽量简化代码

let: 代替var，用来声明变量:

为什么: js有两个广受诟病的缺陷:

1. 声明提前

2. 没有块级作用域

### 参数增强:

1. **默认参数**: 即使不提供实参，形参也有备用的参数值

如何: function fun(形参1, ..., 形参n=默认值)

执行效果: fun(实参1,.../\*没有提供实参n\*/)

fun中的形参n的值就采用默认值

强调: 带默认值的参数必须定义在参数列表的末尾

2. **剩余参数（rest）**: 代替arguments，接收不确定个数的参数值

为什么: arguments的问题:

1. 不是纯正的数组类型，无法使用数组所有的API

2. 只能获得全部参数值，无法有选择的获得部分

如何: 定义函数时: function fun(形参1,...数组名){ ... }

执行时: ...可自动获得除**形参1**之外的剩余参数,保存在一个数组中。

优点: 1. ...后的数组名获得的是一个纯正的数组，可以使用数组API

2. 可有选择的获得部分参数值

3. **散播（spread）**: 代替apply，专门执行打散数组的操作:

为什么: apply的本职是替换this，顺便打散数组

如何: 调用函数时: fun(...数组)

执行时: ...会先将数组打散为单个元素，再分别传入fun中

### 箭头函数:

用于简化回调函数和匿名函数的

1. 去function，改为=>

2. 如果函数体只有一句话，则可省略{}

如果仅有的一句话还是return，则必须省略return

3. 如果只有一个形参，则可省略()

箭头函数特点: 一旦改为箭头函数后，**内外this一致**

今后如果不希望内外this相同时，就不能用箭头函数

### 解构:

从一个大的对象中，提取出想要的成员，单独使用

3种:

**1. 数组解构:** 将一个大的数组中的元素提取出来单独使用:

如何: 下标对下标:

var [变量1, 变量2, ...]=[元素1, 元素2, ...]

0 1 0 1

结果: 变量1=元素1;

变量2=元素2;

**2. 对象解构:** 从一个对象中提取出想要的成员单独使用

如何: 属性对属性:

var {属性1:变量1, 属性2:变量2, ...}=

{属性1:值1, 属性2:值2, ...}

结果: 变量1=值1; 变量2=值2;

**3. 参数解构:** 其实是对象解构在函数传参时的应用

何时: 多个参数都可选时

如何: 2步:

1. 定义时: 将参数列表定义为对象语法:

//function fun({属性1:形参1,属性2:形参2,...})

function fun({形参1,形参2,...}){ ... }

2. 调用时: 将传入的参数放在一个对象中整体传入

fun({

属性1: 值1, ...

})

执行: fun将整个实参对象传给形参对象，形参对象通过解构，从实参对象中抽取对应的参数值。

如果找不到对应的，则形参值默认为undefined

**for...of...**

最简化的遍历数组和类数组对象的方法

1. for循环:

for(var i=0;i<arr.length;i++){

var elem=arr[i];

}

最灵活的，万能的

2. forEach:

arr.forEach((elem,i,arr)=>{ elem })

局限: 无法控制循环的方向和步调，只能顺序依次遍历每个元素

3. for of:

for(var elem of arr){

//of会依次获得arr中每个元素的值

}

局限: 只能获得元素值，无法获得当前位置

vs for in: for(var key in obj/关联数组)——自定义下标名

for(var elem of arr/类数组对象)——数字下标

### ES6对整个面向对象语法的简化: class

1. 对象直接量的简化:

如果属性名和变量名一致时，可以简化

var sage=11;

var lilei={

sname, //sname: sname,

sage, //sage: sage,

intr(){ ... } //:function(){ ... }

}

2. 对创建一种类型的简化:

//1. 用class{}包裹构造函数和原型对象

class Student{

//2. 构造函数名提升为类型名，构造函数统一更名为 **constructor**

constructor(sname,sage){

this.sname=sname;

this.sage=sage;

}

//3. 所有原型对象方法可直接写在class中与构造函数平级（不要逗号）

intr(){

console.log(`I'm ${this.sname}, I'm ${this.sage}`);

}

}

3. 继承:

//1.让子类型继承父类型: Plane extends Enemy

//不再需要Object.setPrototypeOf

class Plane extends Enemy{

constructor(fname,speed,score){

//2.用super(fname,speed)调用父类型构造

//不再需要call，不再需要传this

super(fname,speed);

this.score=score;

}

4. 静态方法:

class 类名{

constructor(){

}

方法(){ ... }

static 静态方法(){ ... }

}

5. 访问器属性:

class Emp{

constructor(...){ ... }

get age(){return this.\_age}

set age(value){

if(value>=18&&value<=65)

this.\_age=value;

else throw Error("年龄超范围");

}

}

### Promise:

问题: 多个异步调用的函数必须依次排队执行

错误的做法: 仅顺序编写:

task1();

task2();

传统的做法: 利用回调函数: 2步:

1. 在定义异步调用的函数时，就要定义回调函数参数，并决定何时调用回调函数。

function task1(callback){//callback准备提前接收下一个任务，但暂不执行

异步调用: 定时器、ajax

在异步调用的程序最后一句话之后，调用callback()

}

2. 调用前一个任务task1时，应提前将下一个任务封装在一个函数中，传入task1，相当于交给callback

task1(

function(){

task2()

}

)

问题: 回调地狱 callback hell

多级函数调用的嵌套

根源: 回调函数要求，必须在前一个任务开始时，就要提前将后一个任务传入。

**解决: Promise:**

如何: 2步:

1. 在定义异步调用的函数时:

1. 不要callback参数

2. 用return new Promise(function(open){

包裹原函数的**所有函数体**

})

3. 决定在原异步函数最后执行的一句话之后，打开开关:open()

2. 在调用函数时:

task1().then(task2)

如果task2也支持Promise

.then(task3)