基础常识

代码注释

- 注释的功能：描述架构、记录函数的参数和用法、解释思路和想法、讲解特性和使用方法
- **不能任意添加注释**
  - 当代码本身就能表示要做什么的时候不需要添加注释
  - 一个原则：如果代码不够清晰，需要的不是一个注释，而是对代码进行修改
    - 例如：在某个for循环内部存在一个新循环用来进行判断，那么，可以用一个函数调用代替其中的内容，而不是使用注释进行说明(要善于封装函数、善于使用高级的、可读性高的属性)

语句分号问题

- 是否加分号时编码风格的问题，没有应该不应该，只有喜欢不喜欢，但推荐加
- 实际只有以下情况才有必要加分号：
  - `() [] 正则 \\ + -` 开头的语句前加分号
  - 小括号开头的前一条语句（立即执行函数）
  - 中方括号的前一条语句
  - 例：Vue.js不使用分号

JS 数据类型

基本类型：字符串、数字、布尔、空(Null)、未定义(Undefined)、Symbol、BigInt

对象类型：对象(Object)、函数(function)、数组(Array)

> 引用数据类型赋值给新的变量，只是将索引值传递给它，并不复制产生新的内容，会随原内容修改而修改；
>
> 变量类型(变量值的类型)：基本类型 和 引用类型(保存的是地址)

- Number 数字型 0、1、2、3.1415...
  - JS中 Number类型 被定义为 64为双精度浮点型数据
  - 在其他语言中，一般具有int、float、double等类型
  - js中尽可能不要使用小数的运算，存在精度问题。
    - 可以对小数进行取整操作
- BigInt 长整型
  - BigInt 类型 用于表示任意精度格式的整数，在数字后尾使用n 标识(1234567890n)
  - 特点：
    - Number类型的数字取值范围：-(2^53-1) ~ (2^53-1)
    - BigInt 类型，用于表示超出这个范围的数值
- String 字符串型

    - String类型的数据必须用引号括起来，浏览器的 prompt() 获取到的数据默认为字符串类型。
- Boolean 布尔数据类型：true false
- Null：
    - 声明且定义为null
    - 表示一个不存在的值 或 无效的object 或 无效的地址应用
- Undefined
    - 声明但未定义的变量的初始值
    - 没有实际参数的形式参数的值(形参)
- Object 对象类型
    - 数组 数据结构是对象的一种
    - 用于处理数据的指令或数据结构

> 关于引用变量赋值问题：
>
> - 多个引用变量指向同一个对象，当通过一个变量修改对象内部的数据时，另一个变量中看到的数据会随之修改
>
> ```
> var obj1 = {name:one }
> var obj2 = obj1
> obj1.name=two;
> console.log(obj2.name);  //two
>
> var a = { age : 12 }
> var b = a;
> // 在这一步a的索引发生改变
> a ={ name:tom , age:13}
> b.age = 14;
> console.log(b.age,a.age,a.name)  //14,13,tom
>
> function fn(obj){
>    // 在这一步a的索引又发生改变
>    obj = {age:15}
> }
> fn(a)
> console.log(a.age);   //15
> ```

- Symbol 数据类型(没学！)
    - 《 Node.js入门教程 》57页

类型的判断

- 方法1：`console.log(typeof 数据)` //打印数据的数据类型
- 方法2：type 数据 === 'number'
- 方法3：instanceof 原型判断法

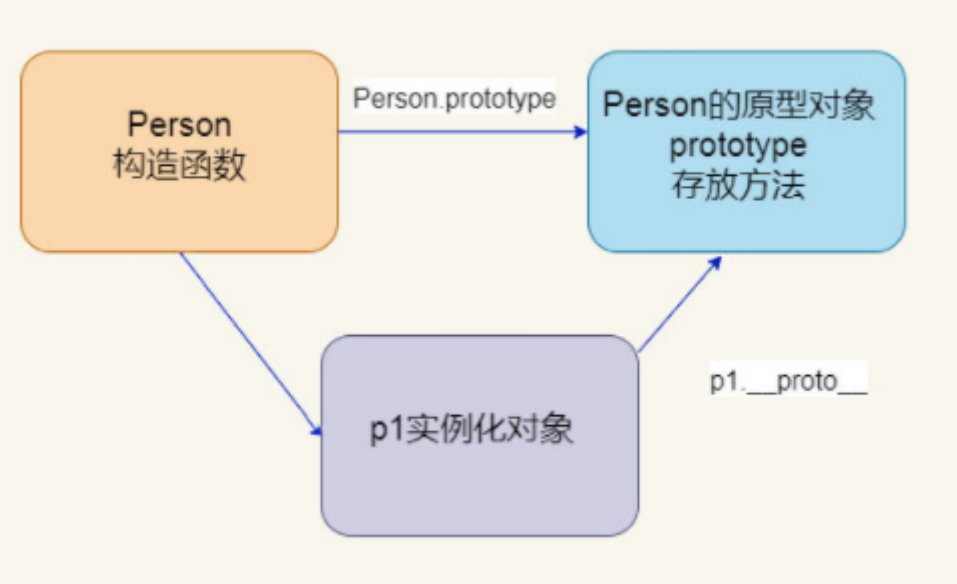
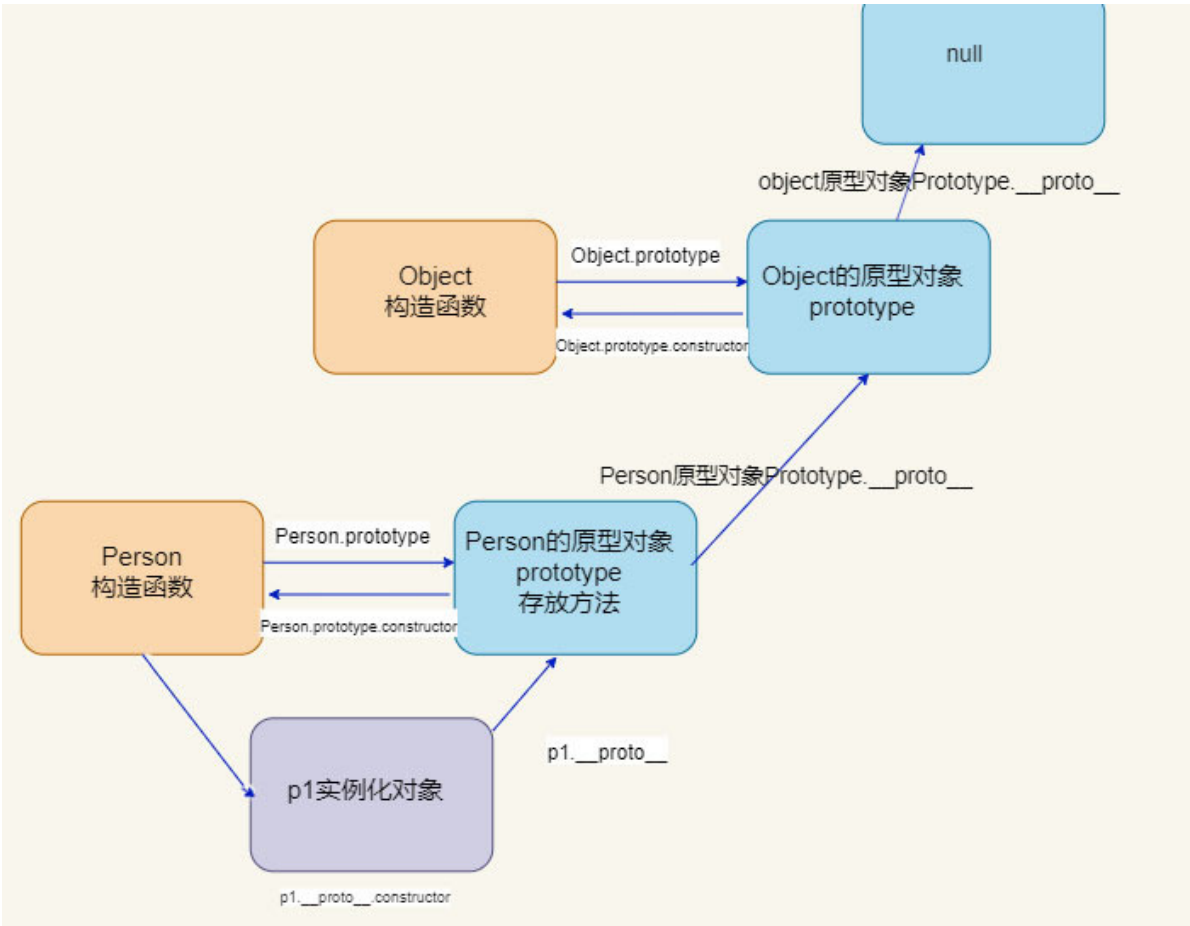
- ```
  {} instanceof Object; //true
  [] instanceof Array;  //true
  [] instanceof Object; //true
  "123" instanceof String; //falsenew String(123) instanceof String; //true
  ```

函数与变量提升

- 只有var声明的变量才会变量提升，不适用var就不会提升。
- 特征一：
  - 函数声明提前，在函数定义以后，代码执行时，会将函数的所有声明全部提前定义，可以提前调用执行
  - ```
    showValue("我爱你");
    function showValue(one){
       console.log(one);
    }
    //这里的函数定义会被提前，可以成功调用
    ```
- 不能执行的代码
  - ```
    showVlue("不爱你")
    var showValue = function(value){
       console.log(value)
    }
    // 这里是将定义好的函数 存储在一个变量，不涉及函数的定义，所以不存在函数定义提前，无法执行
    ```
- 当两种方式都存在时，会以第二中方式执行，无论二者的书写顺序
  - 原因：var 声明变量提前，但对应的函数还没执行，function定义的函数提前，
  - 最后执行到该段代码时，var 定义的函数会替换掉原本定义的function函数

> 面试题：
>
> ```
> var c=1;
> function c(c){
> console.log(c);
> }
> c(2)    //结果：报错
>
> // 分析：
> // 变量提升与命名提升后，var c ==> function c(){} ==> c=1 ==> c()就未定义
>
>
> //面试题2
> var x=10;
> function fn(){
> console.log(x);
> }
> function show(f){
> var x=20;
> f();
> ```
    ```

```
}
show(fn);      //输出10
//分析：函数的上下文作用域，在代码运行前就已经决定，在fn中只能读取到全局作用域中的x=10

//面试题3
var obj={
fn2：function(){
   console.log(fn2);
}
}
obj.fu2();    //报错，在全局中未定义fun2，若想输出自身的fn2需要使用this.fn2
```

JS高级

Arguments 对象

- 每个函数都会有一个Arguments对象实例arguments，它引用着函数的实参，可以用数组下标的方式"[]"引用arguments的元素。
- `arguments.length` 为函数实参个数
- `arguments.callee` 引用函数自身
- 箭头函数没有 Arguments对象，使用 rest替代

构造函数

- 构造函数定义时： **首字母大写** (规范)
- 构造函数本事身 也是一个普通函数
- 构造函数的调用必须使用`new关键字`,用来创建建实例对象
- 构造函数的内部执行流程
  - 立刻在堆内存中创建一个新的对象
  - 将新建的对象设置为函数中的this
  - 逐个执行函数中的代码
  - 将新建的对象作为返回值
- 普通函数如果不`return`,则没有返回值；构造函数默认返回值是对象
- 定义：通过 new 函数名, 来实例化对象的函数叫构造函数。任何的函数都可以作为构造函数存在。之所以有构造函数与普通函数之分，主要从功能上进行区别的，构造函数的主要功能为初始化对象，特点是和new 一起使用。new就是在创建对象，从无到有，构造函数就是在为初始化的对象添加属性和方法。

静态成员与实例成员

- 静态成员：在构造函数本身上添加的成员，只能由构造函数本身来访问

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.run = function () {
        console.log(this.name+'在奔跑');
    }
}
var p2 = new Person('李四', 24);  // 创建实例化对象
Person.sex = '男';  // 创建静态成员
console.log(Person.sex);  // '男'
console.log(p2.sex);  // undefined
```

- 实例成员：在构造函数内部或原型上添加的成员，只能由实例化的对象来访问

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.run = function () {
        console.log(this.name+'在奔跑');
    }
}

var p1 = new Person('张三', 20);  // 创建实例化对象
console.log(p1.name);  // 用实例化对象访问name属性
p1.run();  // 用实例化对象访问run方法
console.log(Person.name);  // 无法访问
```

prototype原型对象

- 每一个构造函数都有一个prototype 属性，该构造函数创建的对象会默认链接到该属性上
- 可以通过prototype来添加新的属性和方法，此时所有该构造函数创建的实例对象都会具有这些属性和方法
- 作用：共享方法,节约内存
- 语法：
    - `构造函数.prototype.属性名 = 值;`
    - `构造函数.prototype.方法名 = function(){定义方法体};`

原型对象

- 隐式原型对象 与 显示原型属性 是同一个东西
- 可以通过 显示原型对象 追加属性，在再隐式对象上使用

```
//定义一个构造函数
function Demo(){
    this.a=1;
    this.b=2;
}
//创建Demo实例
const d = new Demo();
console.log(Demo.prototype);      //显示原型属性
console.log(d.__proto__);         //隐式原型对象
```

对象原型

- 每一个对象都会有一个属性 **proto** 指向构造函数的 prototype 原型对象，从而可以使用构造函数 prototype 原型对象的属性和方法
- `__proto__对象原型` = `原型对象 prototype` 。
- `__proto__对象原型` 的**意义**就在于为对象的查找机制提供一个方向，或者说一条路线，但是它是一个非标准属性，因此实际开发中，不可以使用这个属性，它只是内部指向原型对象prototype。



原型链

- **当访问一个对象的某个属性或方法时，会先在这个对象本身属性上查找，**
    - 如果没有找到，则会去它的`__proto__`上查找，即它的构造函数的prototype，
    - 如果还没有找到就会再在构造函数的prototype的`__proto__`中查找，
    - 这样一层一层向上查找就会形成一个链式结构，我们称为原型链。



constructor属性

- 用来保存数据的部分，一般省略，内部会自动创建（详情看pink）
- 构造函数的原型对象的constructor属性指向了构造函数,实例对象的原型的constructor属性也指向了构造函数

原型链和成员的查找机制

> 当访问一个对象的属性（包括方法）时，首先查找这个对象自身有没有该属性。 如果没有就查找它的原型（也就是 **proto**指向的 prototype 原型对象）。 如果还没有就查找原型对象的原型（Object的原型对象）。 依此类推一直找到 Object 为止（null）。 **proto**对象原型的意义就在于为对象成员查找机制提供一个方向，或者说一条路线。

原型对象this指向

构造函数中的this和原型对象的this,都指向我们new出来的实例对象

```
function Person(name) {
          this.mName = name;
};
var that;
Person.prototype.sayName = function() {
     console.log('我是大咖');
     that = this;
}
var p1 = new Person('张三');
```

```
// 1. 在构造函数中,里面this指向的是实例对象
p1.sayName();
console.log(that);
console.log(p1);
console.log(that === p1); //true
// 2.原型对象函数里面的this指向的是实例对象
```

通过原型为数组扩展内置方法

```
Array.prototype.sum = function() {
  var sum = 0;
  for (var i = 0; i < this.length; i++) {
  sum += this[i];
  }
  return sum;
};
//此时数组对象中已经存在sum()方法了  可以使用数组.sum()进行数据的求和
```

其他方法

- `Object.prototype.hasOwnProperty(prop)` 方法
  - 可以判断一个属性定义在对象本身而不是继承原型链的方法，主要用于判断某个对象中是否有某个属性，返回值为布尔值。
- Object.prototype.isPrototypeOf(Object)方法
  - 接收一个对象，用来判断当前对象是否在传入的参数对象的原型链上，返回一个布尔值。

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    // 自身
    // this.sayHi = function() {
    //     console.log('自身的sayHi');
    // }
};
Person.prototype.sayHi = function() {
    console.log('prototype的sayHi');
};
var p1 = new Person('乔碧萝', 43);
// 可以判断一个属性定义在对象本身而不是继承原型链的方法
console.log(p1.hasOwnProperty('name'));
console.log(p1.hasOwnProperty('sayHi'));
// isPrototypeOf方法接收一个对象，用来判断当前对象是否在传入的参数对象的原型链上，返回一个布尔值
console.log(Person.prototype.isPrototypeOf(p1)); //true
console.log(Object.prototype.isPrototypeOf(p1)); //true
var date = new Date();
console.log(Person.prototype.isPrototypeOf(date)); //false
```

> 测试题：

```
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
```

理解 Object

- 创建对象的三种方法
  - 字面量法 var one={...}
  - 构造函数法 function two(uname){ this.uname=uname; ...}
  - 利用new Object() var three=new Object();
- 遍历对象的方法
  - for(i **in** obj_name) {...}
    - i 是每一项的索引名
  - Object.keys(obj_name);

```
//参数：要遍历的对象的名字，结果以数组形式输出
//返回值为对象的所有属性名组成的 可枚举属性 的数组
console.log(Object.keys(person));
```

  - Object.getOwnPropertyNames(obj_name);
    - 返回对象的所有属性名组成的数组(包含不可枚举属性，但不包含Symbol值作为名称的属性)
- **以上内容详见笔记本**

检查属性是否存在

- `if( a in obj1 ){ }`

```
var obj1={
   uname:"小刘",
   money:0
}

// 不可靠的检测方法：直接使用
if(obj.money){
   //存在该属性
}

// 可靠的检测方法 if...in
```

```
if(obj.money in obj){
   console.log("存在");
}
```

删除属性

- 使用`delete`操作符将属性从对象中完全删除。

```
obj.name = undefined;
此时name属性仍存在obj对象中，怎样才能在obj中移除name属性呢？
//删除属性
delete obj.name;
```

Object属性

- 普通属性：通过赋值操作定义的属性
- 数据属性(存储数据)：需要设置value值，但不能使用 set和 get
  - 不包含值，而是定义了一个`get`和`set`函数，
  - 当读取属性时，调用`get`函数；当写属性时，调用`set`函数。

```
let person = {
    name:'吴老板',
    age:18
}

//为person对象追加新的属性
//参数1：要添加属性的对象的名字
//参数2：要添加的属性的名字
//参数3：添加的属性的配置项（值）
Object.defineProperty(person,'school',{
    value:18,              //该属性对应的值，默认为undefined,可以是任意有效js值(数值，对象，函数等)
    enumerable:true,       //控制属性是否可以能否通过 for-in 循环，进行枚举。
    writable:true,         //控制属性值是否可以被修改
    configurable:true      //控制属性是否可以通过 delete 删除属性从而重新定义属性，能否修改属性的特性，能否把属性修改为访问器属性。
})

默认：
// 1.使用该方法追加的属性，默认情况：不可枚举，无法遍历得到    enumerable:false,
// 2.默认情况下不能被修改。      writable:false,
// 3.默认情况下不可以被删除     configurable:false
```

- 访问器属性：不能设置value值，使用get 和set 函数

```
var obj = {
    age: 12,
   //访问器属性
```

```
    get add(){
        return this.age + 2;
    },
    set add(value){
        this.myname = value;
    }
};

console.log(obj.add);    //  14 这里执行了add的get函数
obj.add.value=10;
console.log(obj.age);    //  12 上一句赋值操作触发了add的set函数
```

```
//访问器属性
Object.defineProperty(obj, 'age', {
    configurable: true,
    enumerable: true,
    get: function() {
       // 读取 age 属性时，会执行这个函数。返回值就是 age 的值。
        return this._age;
    },
    set: function(newValue) {
       // 写入 age  属性时，会执行这个函数。
        if (newValue > 5) {
            this._age += newValue;
        } else {
            this._age = newValue;
        }
    }
});
```

- 注意：
  - 通过赋值操作定义对象，添加的普通属性，**默认都是可修改、可枚举、可配置的(删除和添加)。**
  - 使用 `Object.defineProperty()` 添加的属性**默认都是不可修改，不可枚举，不可配置(删除和添加)。**
    - 配置的默认值为false，可设置为true

Object.defineproperty 绑定数据

```
let number = 10;
let person = {
    age:'number'
}

Object.defineProperty(person,'age'{
    //需求：当number的值更改后，person的age值就跟着发生更改
    //当读取person的age属性时，get函数就被调用，且返回值就是age的值
    //原理：只要读取get的值时，就调用get函数，重新为获取number的值更新到age
     get(){
        return number
     }

    //需求：当person的age值修改后nubmber跟着修改
```

```
    //当修改person的age值时，就调用set函数，且会收到修改的具体的值
    //原理：修改person的age数据后，将age的值赋给number
     set(value){
         number = value;
     }
})

//注意：使用get() 或者set() 时，不能给追加的属性再写value值
```

定义多个Object属性

- 定义单个属性的内部特性使用`Object.defineProperty()`
- 定义多个属性使用的是`Object.defineProperties()`
    - 参数1：属性所属的对象
    - 参数2：包含被定义属性的对象

```
var obj = {
    age: 2
};
Object.defineProperties(obj, {
    name: {
        configurable: false,
        writable: true,
        value: '李小谷'
    },
    age: {
        get: function () {
            return this._age;
        },
        set: function (newValue) {
            if (newValue > 5) {
                this._age += newValue;
            } else {
                this._age = newValue;
            }
        }
    }
});
```

获取Object属性特征

- 该方法返回指定对象上一个自有属性对应的属性描述符
- 自有属性指的是直接赋予该对象的属性，不需要从原型链上进行查找的属性
- `Object.getOwnPropertyDescriptor(obj, "prop")`
    - `obj`：目标所在的对象。
    - `prop`：要获取特性的属性。**注意属性要带引号！**
    - 返回值：如果给定的属性存在于对象上，则返回属性描述符对象。否则返回 undefined
- `Object.getOwnPropertyDescriptors(obj)`

- `obj`：要获取的目标对象。
- 返回值：所指定的对象的所有自身属性的特性描述符，如果没有任何自身属性则返回空对象。

```
// 构造函数
function Star(name, age, sex) {
    // 实例成员
    // 属性
    this.name = name;
    this.age = age;
    this.sex = sex;
    // 会唱歌
    // this.singing = function() {
    //     console.log('会唱歌');
    // }
};
// 添加的方法
Star.prototype.singing = function() {
    console.log('会唱歌pro');
};
var s1 = new Star('王一博', 24, '男');
// 指定对象上一个自有属性对应的属性描述符。
console.log(Object.getOwnPropertyDescriptor(s1, 'name')); // 返回实例对象name的配置
console.log(Object.getOwnPropertyDescriptor(s1, 'singing')); //undefined
```

数据代理

- 数据代理：通过一个对象代理对另一个对象中属性的操作（读/写）。

```
let obj = { x:100 }
let obj2 = { y:200 }

//通过obj2中的x对obj中的x进行读写操作
Object.defineProperty( obj2,'x'{
   get(){
       return obj.x
   },
   set(value){
       obj.x = value
   }
})
```

IIFE 匿名函数自调用

- 就是 立即执行函数
- 作用：
    - 隐藏实现，不会污染外部（全局）命名空间
    - 用它来编码`js模块`

- ```
  (function(){
     var a=3
     console.log(a+3)   //6
  })()
  var a=4
  console.log(a);       //4 ,
  (function(){
     var a=1
     function test(){
        console.log(++a)
     }
     // 向外暴漏一个全局函数
     windows.$=function(){
        return {
           test:test
        }
     }
  })()

  $().test();   $()返回{text: text}        // 2
  ```

闭包Closure

- 如何产生闭包：当一个嵌套的内部(子)函数引用了外部(父)函数的变量(函数)时，就产生了闭包。
- 闭包到底是什么?
  - 理解1：闭包是嵌套的内部函数 (绝大多数)
  - 理解2：包含被引用变量(函数)的对象 (少数理解)
  - 闭包存在于嵌套的内部函数中
- 产生闭包的三个条件：
  - 函数嵌套
  - 内部函数引用了外部函数的数据（变量/函数）
  - 执行了外部函数，但不一定需要执行内部函数(执行函数定义就会产生闭包)

常见的闭包

- 将函数作为另一个函数的返回值

```
function fn1(){
   var a=2
   function fn2(){
      a++;
      console.log(a);
   }
   return fn2
}
var f=fn1()
f()  // 3
f()  // 4
fn1()
```
```

```
// 分析：这个程序一共产生了两个闭包，没次执行一下外部函数，就产生一个闭包
// 在两次f()的过程中，都只执行了fn2，并且使用了f保留下来的a值
```

- 函数作为实参传递给另一个函数

```
function showTime(msg,time){
   serTimeout(function(){
      // 内部函数使用了外部函数的数据 msg
      console.log(msg)
   },time)
}
showTime("123",1000);
```

闭包的作用

- 使函数内部的局部变量在函数执行后，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)函数内部的数据
- 问题：
  - 函数执行完后，局部变量一般不会继续存在，但存在于闭包中的变量可能继续存在（被引用时）
  - 在函数外部不能直接访问函数内部的变量，但通过闭包可以返回一个内部函数，操作函数内部的数据

```
// 假设需求：让一个变量在外部只能读但不能修改
// 使用闭包，在外部函数中返回一个内部函数，内部函数执行返回数据的值(或限制对数据的操作)
function fn1(){
   var a=2
   function fn2(){
      a++;
      return a
   }
   function fn3(){
      a++;
      return a
   }
   return fn2
}
var f=fn1()
// 例如这个函数只能对a执行++ 操作并读取a的值
// 在函数执行结束后，fn3、fn2都被当做垃圾对象回收，但因为f仍然引用着fn3的函数，所以a不会被回收
```

闭包的生命周期

- 产生：在嵌套函数的外部定义执行完成就产生，(不是调用时才产生)
- 死亡：在嵌套的内部函数成为垃圾对象时

闭包的应用

- 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的内部（私有）
  - 只向外暴露一个包含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

```
// 方式1： 具有明显优势，带参数window可避免代码压缩产生的问题
// - 外部引入该函数，可以直接使用myfunction对象提供的方法
(function(window){
   var msg='myname'
   function fn1(){
      console.log('msg'+'fn1')
   }
   function fn2(){
      console.log('msg'+'fn2')
   }
   window.myfunction={
      fn1:fn1,
      fn2:fn2
   }
})(window);
fn1();

// 方式2：
function myfunction2(){
   var msg='myname'
   function fn1(){
      console.log('msg'+'fn1')
   }
   function fn2(){
      console.log('msg'+'fn2')
   }
   return {
      fn1:fn1,
      fn2:fn2
   }
}
// 需要先执行一次，才能使用它提供的方法
var myfunctions = myfunction2();
myfunctions.fn1()
```

闭包的缺点

- 函数执行结束后，函数内的局部变量没有释放，长时间占用内存，容易导致内存泄漏
- 解决方法：
  - 能不用闭包，就不用
  - 及时手动释放，赋值为null，让对应函数成为垃圾对象，回收闭包

内存溢出与泄露

- 内存溢出
  - 一种程序运行出现的错误，程序无法运行
  - 当程序执行需要的内存超过了剩余内存时，就提示内存不足的错误！
- 内存泄露
  - 占用的内存**没有及时释放**，占用着内存
  - 内存泄露累计多了就容易导致**内存溢出**
  - 常见的内存泄露：
    - 意外的全局变量，例：在局部变量中没有声明的变量，成为了全局变量
    - 没有及时清理的计时器或回调函数，再后台一直运行
    - 闭包

> 面试题：
>
> ```
> // 代码片段1：
> var name = 'The window';
> var object ={
> name:'My Object',
> getNamefun:function({
>    return function(){
>       return this.name;
>    }
> })
> };
> alert(object.getNamefun()());  // the window
> // 分析：object.getNamefun() 返回值是一个函数，函数内部的this默认指向windows
>
> // 代码片段2：
> var name2 = 'The window';
> var object2 ={
> name2:'My Object',
> getNamefun:function({
>    var that=this;
>    return function(){
>       return that.name;
>    }
> })
> };
> alert(object2.getNamefun()());  // My Object
> // 分析：that 在执行时保存了this指向的Object2
> ```

Image 对象

- 让浏览器缓存一张照片

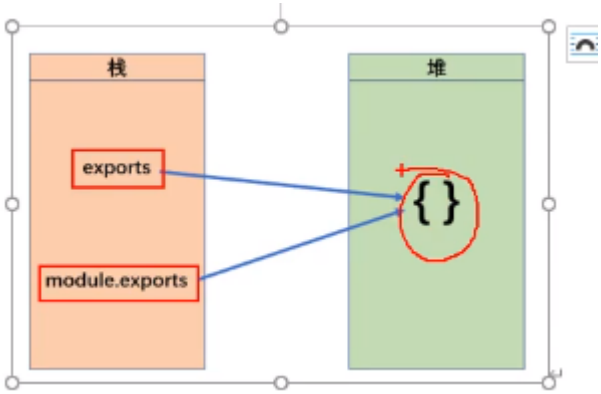
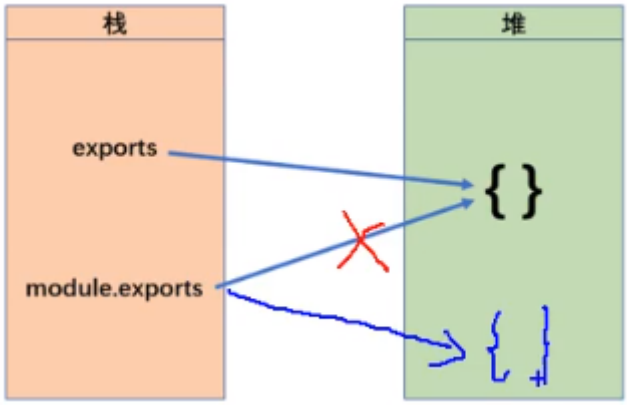
```
// js创建Image对象,等价于document.createElement('img');
// 宽、高可以省略：var a = new Image();
var a = new Image(100,100);
// 定义Image 对象
a.src = "./xxx.gif"
// 将定义的Image 内容放在页面中
document.body.appendChild(a);

//相当于在body中定义了如下内容
<img width="100" height="200" src="picture.jpg">
```

- **src 属性一定要写到 onload 的后面，否则程序在 IE 中会出错**
- 使用图片的js操作，要等图片加载完成在执行
    - ```
      var a = new Image(100,100);
      a.src = "./xxx.gif"
      a.onload = function(){
          // 此时再操作图片，避免因图片还未加载而出错
      }
      ```
- 考虑到浏览器的兼容性和网页的加载时间，尽量不要在 Image 对象里放置过多的图片
- Image对象的 **complete属性**
    - 当图像处于装载过程中，该属性值false,
    - 当发生了onload、onerror、onabort中任何一个事件后，则表示图像装载过程结束（不管成没成功），此时complete属性为true）

正则表达式 RegExp

在线正则表达式测试 (oschina.net)

- 用于定义一些字符串规则，计算机可以根据正则表达式进行字符串的检测
- 正则表达式是一个对象
    - 创建正则表达式的两种方法
        - 使用构造函数，更加灵活 `var 变量 = new RegExp('正则表达式','匹配模式');`
        - 使用字面量，更简便 `var 变量 = /正则表达式/匹配模式`
- 转义字符`\`反斜杠
    - `/\./`表示 `.`
    - `/\\\\/` 表示`\\`
      ```

```
var str =  'abc';
// 使用正则表达式步骤:
// 1.创建正则表达式对象 var 变量 = new RegExp('正则表达式','匹配模式');
var one = new RegExp('a');
// 使用字面量创建正则表达式   var 变量 = /正则表达式/匹配模式
var one = /a/i;

// 2. text() 方法检测字符串是否符合规则; 符合true;不符合false
var result = one.test(str);
console.log(result);   //返回值 true   说明str符合one
```

- **匹配模式**

符号	描述	实例
i	忽略大小写	`new RegExp('正则表达式','i');`
g	全局匹配模式	`new RegExp('正则表达式','g');`
gi	全局匹配模式且忽略大小写	

- **正则表达式规则**

符号	含义	实例
|	或(有一个就行)	/a|b/ （存在a或b）
[ ]	或(有一个就行)	`/[A-z]/` (存在A-z之间的任意字母） `/[ab]/` (存在ab)
[^]	除了^里面内的都可以	`/[^a-z]/` (除了a-z之件的字母都可以)
{ }	连续出现的次数	`/ba{3}/` 连续出现3个a； `/(ab){3}/`连续出现3个ab；`/b{1,3}c/` 连续出现1-3次b； `/b{3,}/`出现3次及以上次数b
n+	至少包含一个	`/a|b+/` 至少出现一个a或b
n*	包含0个或多个，相当于没写	
n?	包含0个或1个	
n$	以n结尾	
^n	以n开头	

手机号的正则：`/^1[3-9][0-9]{9}$/`

- 元字符

符号	含义	实例
`.`	查找任意字符，除了换行和行结束符	
`\w`	任意字母、数字、_	
`\W`	与\w相反，即`[^A-z0-9_]`	
\d	任意数字	
\D	除了数字	
\s	空格	
\S	除了空格	
\b	单词边界	`/\bchild\b/` 检测有没有单独的单词`child`
\B	除了单词边界	

去除字符串的所有空格：`str = str.replace('/\s/g')`

去除字符串前后的空格：`/^\s*|\s*$/g`

支持正则表达式的Sting方法

方法	描述	实例
`split()`	将一个字符串根据'拆分符'拆分为一个数组	`str.split(/[a-z]/)` 遇到a-z之间的符号进行拆分
`search()`	检索字符串中是否含有指定内容,返回'索引值'或-1	`str.search(/a[be]c/)` 是否含有abc或aec
`match()`	默认将第一个符合条件的内容从字符串中提取出来；采用正则表达式的全局匹配模式，返回所有符合结果组成的数组	`str.match(/[A-a]/ig)`
`replace()`	将符合规则的字符串进行替换，参数1：被替换的内容，参数2：新的内容; 默认只替换第一个;可以使用全局匹配模式	`str.preplace('/a/g','@')`

ES6

函数中this指向

调用方式	this指向
普通函数调用	window
构造函数调用	实例对象 原型对象里面的方法也指向实例对象
对象方法调用	该方法所属对象
事件绑定方法	绑定事件对象
定时器函数	window
立即执行函数	window

严格模式

- JS提供正常模式和严格模式两种，目的：为未来版本做铺垫
- 严格模式只在IE10以上浏览器版本才支持，旧版本中会被忽略
- ES5严格模式是具有限制性的JS变体，更改内容：
    1. 消除了JavaScript语法的一些不合理、不严谨，减少怪异行为
    2. 提高编译器效率，增加运行速度。
    3. 禁用了在 ECMAScript 的未来版本中可能会定义的一些语法，为未来新版本的 Javascript 做好铺垫。比如一些保留字如：class, enum, export, extends, import, super 不能做变量名

脚本开启严格模式

- 在代码前使用`"use strict";`

```
"use strict";
console.log("这里之后的代码都是严格模式");
```

- 使用立即执行函数包裹（同时存在严格模式和非严格模式时使用）

```
(function(){
   "use strict";
    console.log("这里的代码都是严格模式");
})()
console.log("这里是非严格模式");
```

函数开启严格模式

- 把“use strict”; (或 'use strict';) 声明放在函数体所有语句之前

```
function fn(){
   "use strict";
   return "这是严格模式。";
}
```

严格模式的规范

- this指向

全局作用域	window对象
全局作用域中函数	undefined
对象、事件等	谁调用指向谁

1. 变量规定

① 在正常模式中，如果一个变量没有声明就赋值，默认是全局变量。严格模式禁止这种用法，变量都必须先用 var 命令声明，然后再使用。

② 严禁删除已经声明变量。例如，delete x; 语法是错误的。

2. 严格模式下 this 指向问题

① 以前在全局作用域函数中的 this 指向 window 对象。

② 严格模式下全局作用域中函数中的 this 是 undefined。

③ 以前构造函数时不加 new也可以 调用,当普通函数，this 指向全局对象

④ 严格模式下,如果 构造函数不加new调用, this 指向的是undefined 如果给他赋值则 会报错

⑤ new 实例化的构造函数指向创建的对象实例。

⑥ 定时器 this 还是指向 window 。

⑦ 事件、对象还是指向调用者。

3. 函数变化

① 函数不能有重名的参数。

② 函数必须声明在顶层.新版本的 JavaScript 会引入“块级作用域”（ES6 中已引入）。为了与新版本接轨，不允许在非函数的代码块内声明函数。

更多严格模式要求参考：https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Strict_mode

模块化

- 理解：将复杂的程序依据一定规则**拆**分为单个文件，使用时再组**合**在一起
- 特点：模块的内部数据是私有的，仅向外暴露一定的属性和方法
- 模块化的优势：
  - 降低复杂度，提高解耦性（高耦合低内聚）
  - 避免命名冲突

    - 更好的分离功能代码，实现按需加载
    - 更高的可复用性、高维护性
- 引入模块的查询机制：逐级向上级文件查询依赖的包

CommonJS

- 规范：
    - 每个文件都是一个模块
    - CommonJS模块化的代码，既可以在服务器端运行、也可以在浏览器运行
    - 服务器端：模块化的代码可以直接运行 Node.js环境
    - 浏览器端：模块化的代码需要经过**Browserify编译**
- 基本语法：
    - 暴露语法：
        - 第一种(统一暴露)：`module.exports = value` //value代表要暴露的内容，可以是一个对象
        - 第二种(分别暴露)：`exports.xxx = value;`
    - 引入语法：
        - 引入第三方模块：`conse xxx = require('xxx')` //xxx为模块名
        - 引入自定义模块：`require('xxx')` //xxx为模块文件路径
        - 一般取引入模块名为别名，引入什么取决于暴露的内容
- 内置关系：
    - `exports == module.exports` 两者默认是同一个东西
    - 若出现对`module.exports`的再赋值，则以`module.exports`为准，`exports`不再生效
    - 暴漏的本质：向空对象中加东西
    - 

服务器端Node：

- node下直接使用如下方法运行即可
- exports是module.exports的引用，只有文件中才存在exports

```
//暴露showDate ，并取别名为a
module.exports.a = function showData(){}
module.exports.add = function(a,b){ console.log(a+b); }
-----------------------------------------------------------------------
// 引入modeule自定义模块    ./表示当前文件夹路径
const module1 = require('./module1');
//调用module1暴露的方法
module1.a();
```

```
let data ='atguigu';
//暴露一个对象，对象中包含着需要暴露的方法和属性
//改变了原有空对象的指向，exports失效，以module.exports为准，
//且再次赋值时覆盖原暴露的内容
module.exports = {
    name:'zy',
    showData(){
        console.log(data);
    }
}
-----------------------------------------------------------------------
//引入 引入的只是导出的内容
const module = require('./module');
//调用模块的方法
console.log(module.name);
module.showData();
```




```
//使用exports暴露 sub方法
exports.sub = function(a,b){
    console.log(a-b);
}

//引入sub方法
const module = require('./module');
module.sub();
```

```
//引入第三方模块
const http = require('http');
```

浏览器端：

- 浏览器端没有`require`等node中的语法
- 需要经过**Browserify编译**（全局安装）
  - 全局安装Browserify包`npm i browserify -g`
- 加工代码
  - 只需加工汇总代码的文件即可
  - 命令行切换到需要编译的文件目录
  - `browserify 引入汇总的文件名 -o 输出编译后文件的路径`
  - 例：`browserify ./app.js -o ./build.js`
  - html页面引入build.js

ES6模块化规范

- 每个文件都是一个模块，
- 浏览器中可以直接使用；Node中需要进行编译
- 要借助Babel 和Browserify依次编译
  - Bable解决es6模块化的兼容性问题：编译为node的模块化和es5
  - browserify 再将node的模块化转化为es5

准备相关依赖包

- 全局安装`Browserify`和`babel-cli`(babel的命令工具)
  - `browserify ./app.js -o ./build.js` //Browserify
  - `npm install babel-cli -g` //babel-cli
- 局部安装babel-preset-es2015
  - 只需要在项目中安装即可
  - `npm install babel-presrt-es2015`
- 定义`.baelrc`文件 运行时控制文件
  - 与要解析的文件平级
  - ```
    {
        "presets":['es2015']
    }
    ```
- 直接加工要解析的文件夹（会自动忽略不需要加工的文件）
  - `babel 要加工的文件夹路径 -d 加工后的文件夹路径` //会自动创建生成后的文件夹
    - `babel ./src d ./bulid` 把根目录的文件加工后放在build文件夹中
- index.html 引入需要加载后的文件夹
    ```

基本语法:

- 暴露模块
  - 分别暴露：export 暴露内容 //直接在要暴露的内容前加`export`
  - 统一暴露：export default { ...,...}
  - 默认暴露：`export default ... = xxx` //...是组件名，xxx是函数或对象或其他
- 引入模块
  - 方法1：
  - 方法2：

```
//分别暴露
const data = 'atguigu';
export const name = 'zyy';
export function showName(){
    console.log(data);
}
```

```
//引入 适用于分别暴露和统一暴露
import { myName, myAge, myfn, myClass } from "./test.js";
```

第六集

- 暴露
- bable编译
- 再使用**Browserify编译**
- 就可以使用了

ES6面向对象

面向过程pop

- 思想：分析解决问题的步骤，根据步骤依次实现，使用时再调用，以事件的步骤划分。
- 优点：性能高，适合跟硬件联系紧密的东西，例：单片机采用面向过程。

面向对象

- 思想：把事物分解为一个个对象，由对象之间的分工合作,以对象的功能划分。
  - 抽取(抽象)`对象共用的属性和行为`组织(封装) 成一个`类`(模板)
  - 对类进行`实例化`，获取类的`对象`
- 优点：具有灵活、代码可复用、易于维护和开发，适合多人合作的大型项目。
- 缺点：性能比面向过程低。
- 特性：封装性 继承性 多态性

类：抽象了对象的公共部分，泛指一大类 class。

对象:在js中，对象是一组无序的相关属性和方法的集合，所有的事物都是对象。

- 对象特指某一个，通过类的实例化为一个具体的对象
- 属性：事物的特征，在对象中用属性表示（常用名词）
- 方法：事物的行为，在对象中用方法表示（常用动词）

创建类

- 通过`class 类名`创建类，类名习惯首字母大写
- `constructor()`方法：是类的构造函数(默认方法)，用于传递参数，返回实例对象。
  - 通过new命令生成对象实例时，自动调用该方法。
  - 如果没有定义，类内部会自动生成`constructor()`
- 注意：
  - 定义类不加小括号 创建实例必须用new并用小括号
  - 类中的函数不需要加`function()`
  - 类的多个函数方法之间不需要逗号分隔
  - 自有属性是对象实例中的属性，不会出现在原型上

```
// 创建类
class Star{
   constructor(uname,age){
      this.uname=uname;
   }
   say(){
      console.log("我是say方法")
   }
   song(song){
      console.log(this.name + song)
   }
}

// 创建实例   必须使用new实例化对象
var ldh = new Star("刘德华",18);
ldh.song("冰心");      //控制台会输出 刘德华冰心


// 使用类表达式
// 这种模式创建类的单例，并且 不会在作用域中暴露类的引用
```

类的继承 ( )

```
class Son extends Father {
   //没看完
}

var son = new Son();
```

。。。。。

cookie,Storage,token区别

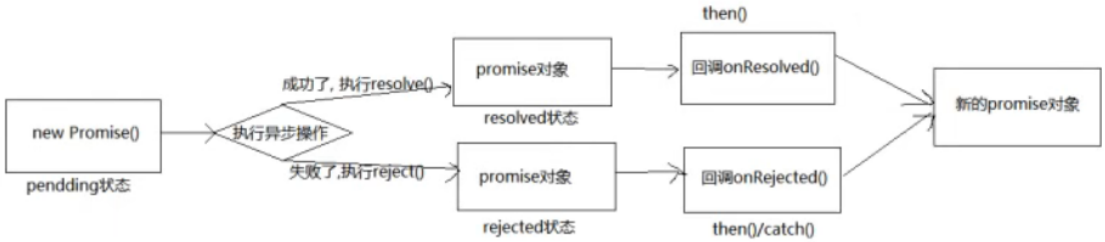
1. 生命周期
    1. cookie(具有maxAge属性,也就是最大超时时间)
        1. 如果有设置maxAge属性,那么在不超过最大超时时间的前提下,只要用户和开发者不主动删除,那么cookie中的数据将永久存在
        2. **如果没有设置maxAge属性,那么只要关闭当前标签页或者关闭浏览器,那么cookie中的数据就会被销毁**
    2. localStorage(持久化存储)
        1. 只要用户和开发者不主动删除,那么localStorage中的数据将永久存在
    3. sessionStorage(临时存储)
        1. 只要关闭当前标签页或者关闭浏览器,那么sessionStorage中的数据就会被销毁
2. 存储位置
    1. cookie
        1. 如果有设置maxAge属性,存储于硬盘中
        2. 如果没有设置maxAge属性,存储于内存中
    2. localStorage
        1. 存储于硬盘中
    3. sessionStorage
        1. 存储于内存中
3. 存储大小
    1. cookie->4KB
    2. localStorage->大部分浏览器都是5MB,IE只能存储3MB左右
    3. sessionStorage->大部分浏览器都是5MB,IE只能存储3MB左右
4. 作用范围
    1. cookie
        1. cookie的作用范围受到两个属性的约束
            1. domain属性->代表当前cookie只能在该网址及其衍生出来的子网址中使用
                1. 也就是说高级网址不能访问低级网址的cookie,低级网址可以访问高级网址的cookie
            2. path属性->代表当前cookie能够被该路由以及子路由进行使用
                1. 例如:path="/a",那么"/a/b"路由就可以访问他的cookie,如果是"/c"就无法访问他的cookie
    2. localStorage
        1. 该存储方式,可以实现跨标签传输,主要和域名有关
    3. sessionStorage
        1. 该存储方式,只能是用于当前标签页,无法跨标签传输
            1. 只有在当前标签不关闭而且域名相同的情况下才能访问到存储的sessionStorage
5. 与服务器之间的关系
    1. cookie("被借用"的本地存储)
        1. **cookie是服务器创建,浏览器存储**
        2. 发送请求的时候,浏览器会自动发送对应的cookie

        3. 服务器通过给**响应头**设置**set-Cookie**属性,来向浏览器返回cookie数据
        4. 浏览器通过给**请求头**设置**Cookie**属性,来向服务器发送cookie数据
        5. **牛逼的其实是浏览器,浏览器会自动识别响应头中的cookie,并自动保存,还会自动发送**
    2. localStorage
        1. 与服务器不熟,没有关系
    3. sessionStorage
        1. 与服务器不熟,没有关系
6. 三者之间的区别
    1. cookie
        1. 发送请求会自动携带cookie
        2. cookie安全性不太好
        3. cookie太小了,体积才4KB
        4. cookie可以被用户禁用
        5. cookie的读写受到服务器端的控制
    2. localStorage
        1. 体积较大,可存储5MB的数据
        2. localStorage不能被用户禁用
        3. 不会自动发送数据,如果有需要,前端人员可以自己书写发送的业务逻辑
    3. sessionStorage
        1. 不能实现跨标签页传输
        2. 存储时间较短
        3. 性能较高!!!
7. 请问token与上述三者有什么区别?
    1. token是一串数据,他是服务器返回的用户的唯一标识,而上述三者他们不是数据,他们是用于存储数据的手段
    2. 前端开发中一般会将token存储于上述三者中
    3. token的本质是字符串
    4. token是服务器根据用户的唯一标识进行base64等加密手段处理之后得到的结果

Promise

> 简单说就是一个类 容器，里面保存着某个未来才会结束的事件，通常是一个异步操作的结果。支持链式调用,以解决回调地狱问题：将异步从外观上同步化

- 异步编程的解决方案
    - 异步编程：fs文件操作、数据库操作、AJAX、定时器...
- 语法上：Promise是 一个构造函数
- 功能上：Promise对象用来封装一个异步操作并可以获取成功 / 失败的结果
- 流程：启动异步任务 => 返回Promise对象 => 给Promise对象绑定回调函数(可以在异步任务结束后指定多个)
-



- Promise的状态（实例对象中的一个属性【PromiseState】）
  - pending ==> resolved / fulfilled 成功
  - pending ==> rejected 失败
  - 说明：只有这两种可能，且一个promise对象只能改变一次
    - 无论成功还是失败，都有一个结果数据
- Promise 对象的值（实例对象中的另一个属性【PromiseResult】）
  - 保存着异步任务 成功 / 失败 的结果
  - reslove
  - reject
- then链式调用的特点：
  - 第一个then执行完毕后(无论成功还是失败)，都会执行第二个then里函数的代码
  - then函数中可以有返回值，让下一个then的形参接受
  - 如果返回值是一个promise对象，下一个then的形参接收到的不是promise对象，而是promise对象内部的resolve函数的实际参数

```
// 创建Promise对象
// 结果成功，调用reslove     结果失败，调用reject
const p = new Promise((reslove,reject)=>{
    reslove('abc');  // 执行 p.then中的第一个函数
})

// 成功调用第一个回调函数，失败调用第二个回调函数
p.then(value=>{
  // 成功的回调 value为'abc'
},err=>{
    // 失败的回调
})

// 链式编程，多个then
p.then(data1=>{
    console.log(data1);
    // 开发中常返回一个promise对象
    return a;
}).then(data2=>{
    // 这里的data2是上一次then回调 return的值 a
    console.log(data2);
})

// 封装promise函数
function text(time){
    // 返回promise对象
    return new Promise(function(resolve,reject){
        // 这里执行异步操作，借助promise特性 将异步变成同步
        resolve(time);
```

```
    })
}
```

catch() finally()

```
// 一般，我们会将以下代码：
p.then(data=>{
    console.log(data)
},err=>{
    console.log(err)
})

// 写成：
p.then(data=>{          // 在成功时执行
    console.log(data)
}).catch(err=>{         // 在失败时执行
    console.log(err)
}).finally(()=>{
    console.log("无论成功或者失败，都会执行")
})
```

all()

- 参数：是一个数组，数组元素都是Promise实例对象
- 只有数组中的Promise都成功后，才会执行 then的第一个回调，且只执行一次

```
// util 是node内置模块，util.promisify方法将原生方法转为promise形式
let readFilePromise= util.promisify(fs.readFile);
let p1= readFilePromise(filePath1,"utf-8");
let p2= readFilePromise(filePath2,"utf-8");
let p3= readFilePromise(filePath3,"utf-8");
Promise.all([p1,p2,p3]).then((data)=>{
    // data是一个数组，数组的每一项分别对应Promise对象成功的值
    console.log(data);
}).catch(err=>{
    console.log(err);
})
```

race()

- 参数：是一个数组，数组元素都是Promise实例对象
- 只要有一个promise执行成功，就执行then中的代码，且只执行一次!
- ```
  Promise.race([p1,p2,p3]).then(data=>{
      console.log(data);    // ???
  }).catch((err)=>{
      console.log(err);
  })
  ```
  ```

async\await

> 异步编程的最终解决方案!

- *async* 使函数返回 Promise

  await 使函数等待 Promise

- `await` 关键字只能在 `async` 函数中使用。
- 如果await后是一个 Promise，会把 resolve的值返回
- async函数里面的 await 是同步执行的
- ```
  // async 函数中才能写 await

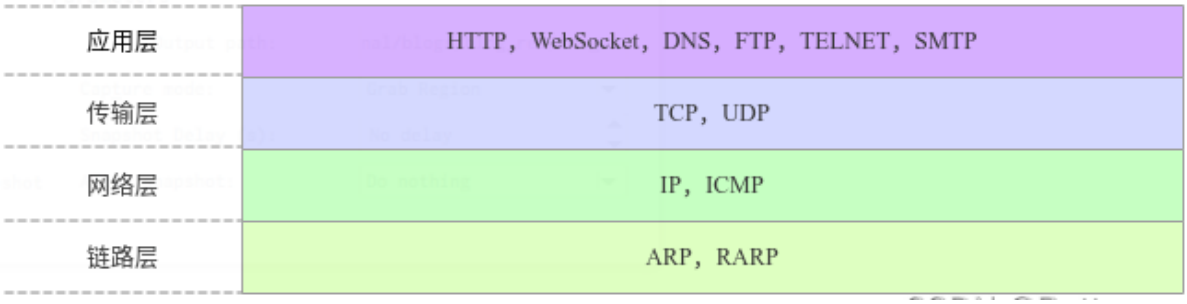
  async function myDisplay() {
    let myPromise = new Promise(function(myResolve, myReject) {
      setTimeout(function() { myResolve("I love You !!"); }, 3000);
    });
     // 在async中 await可以将异步的内容转为同步的方式执行
    document.getElementById("demo").innerHTML = await myPromise;
  }

  myDisplay();
  ```

- 在 await后写一个基本数据类型，会对这个基本数据类型进行包装，包装为promise对象
- ```
  async function func(){
     let data1= await 123;
     //1.await后只写一个基本数据类型，会将这个基本数据类型包装为 promise对象
     // 即 data1相当于：new Promise((resolve,reject)=>{resolve(123)})
     console.log(data1);   // 123
     return data1;         // return await data1
  }

  let a = func();
  a.then((data)=>{
     console.log(data);    // 123   data为上面返回值 Promise对象的执行结果
  });
  ```

JSON数据

- `JSON.stringify( value )` //将 JavaScript 格式的数据转换为json格式的数据
- `JSON.parse( value )` //将json格式的数据进行解析为JavaScript 对象，可以是数组、对象或其他;
- 特点:
  - 大括号 **{}** 保存对象
  - 中括号 **[]** 保存数组，数组可以包含多个对象
  - JSON 布尔值可以是 true 或者 false
  - 文件类型为`.json`后缀

- 可以通过`js数组/对象`的形式操作json中的`数组/对象`
- ```
  var JSONObject= {
      "name":"菜鸟教程",
      "url":"www.runoob.com",
      "slogan":"学的不仅是技术，更是梦想！"
  };

  var JSONObject = {
      "sites": [
          { "name":"菜鸟教程" , "url":"www.runoob.com" },
          { "name":"google" , "url":"www.google.com" },
          { "name":"微博" , "url":"www.weibo.com" }
      ]
  }


  var JSONObject = { "runoob":null }
  ```

- 注意点：
  - JSON 不能存储 Date 对象，会将所有日期转换为字符串。
  - JSON 不允许包含函数，JSON.stringify() 会删除 JavaScript 对象的函数，包括 key 和 value
  - 可以在执行 JSON.stringify() 函数前将函数转换为字符串来避免以上问题的发生
  - 但不建议在 JSON 中使用函数。

WebWorker

> web worker 是运行在后台的 JavaScript，独立于其他脚本，不会影响页面的性能。您可以继续做任何愿意做的事情：点击、选取内容等等，而此时 web worker 在后台运行。实现js的主线程为单线程，扩展新的线程在后台运行。可用于前端性能优化。

WebSocket

- 基于TCP协议的应用层协议，实现了浏览器与服务器之间全双工通信
- 与服务器建立长连接，允许服务器主动向客户端发送信息，真正的双向平等对话
- 虽然WebSocket协议在建立连接时会使用HTTP协议，但这并不意味着WebSocket协议是基于HTTP协议实现的。
- WebSocket协议完全可以取代Ajax方法，用来向服务器端发送文本和二进制数据，而且还没有“同域限制”。
- 协议标识符是ws (加密为wss)，服务器网址就是 URL，如`ws:localhost:8080/msg`
- 使用场景：在线聊天、实时数据刷新
  ```



客户端(原生)

- H5中提供了相关的API，可以直接使用
- WebSocket实例：connet
  - readyState属性，表示目前的状态
    - 0：正在连接
    - 1：连接成功
    - 2：正在关闭
    - 3：连接关闭
- websocket事件

事件	描述
open	连接建立时触发
message	客户端收到服务器发来的数据时触发
error	通信发生错误时触发
close	连接关闭时触发

- websocket方法

方法	描述
实例.send()	客户端向服务器发送数据
实例.close()	关闭连接

```
// 判断浏览器是否支持 WebSocket
if(window.WebSocket != undefined) {
   // 1.建立WebSocket 连接，ws协议  localhost服务器地址  :1740 端口号
   var connection = new WebSocket('ws://localhost:1740');
}

// 2.连接成功后，触发实例对象身上的open事件
connection.addEventListener('open', function(event) {
    console.log("建立连接成功");
})
// 3.客户端通过send方法向服务器端发送数据。
connection.send('我是数据，可以是字符串或二进制数据Blob');
// 4.客户端收到服务器的数据时，触发message事件
connection.addEventListener('message', function(event) {
    //event.data 包含了服务器返回的数据，可能时二进制数据或字符串
    if(event.data instanceof ArrayBuffer){
```

```
        console.log('收到的是二进制数据');
    }
   if(typeof event.data === String)
    console.log(event.data);
})
// 5.当出现错误时，触发error事件
connection.addEventListener('error', function(err) {
    console.log(err);
})

// 6.手动断开连接
connection.close();
// 7.断开连接时触发close事件
connection.addEventListener('close', function() {
    console.log('连接断开');
})
```

除了动态判断收到的数据类型，也可以使用binaryType属性，显式指定收到的二进制数据类型

```
// 收到的是 blob 数据
ws.binaryType = "blob";
ws.onmessage = function(e) {
  console.log(e.data.size);
};

// 收到的是 ArrayBuffer 数据
ws.binaryType = "arraybuffer";
ws.onmessage = function(e) {
  console.log(e.data.byteLength);
};
```

6. webSocket.send（）

send方法用于向服务器发送数据

- 发送文本

```
ws.send("Hello WebSockets!");
```

- 发送Blob数据

```
var file = document
  .querySelector('input[type="file"]')
  .files[0];
ws.send(file);
```

- 发送ArrayBuffer

```
var img = canvas_context.getImageData(0, 0, 400, 320);
var binary = new Uint8Array(img.data.length);
for (var i = 0; i < img.data.length; i++) {
  binary[i] = img.data[i];
}
ws.send(binary.buffer);
```

服务器端(包)

- Node.js中不能直接使用Websocket，需要使用第三方包
- 推荐1：nodejs-websocket
  - 安装依赖包`npm i nodejs-websocket`

```
// 导入nodejs-websocket包
const ws = require('nodejs-websocket');
const POST = 8003;

// 创建一个server服务
// 当用户连接，该函数就会被执行，为每一个用户都创建一个connect对象
const server = ws.createServer(connect => {
    console.log('有个用户连接了,connect中保存着它的一些信息');
    // 每当接收到用户传递过来的数据，就会触发text事件
    connect.on('text', data => {
        console.log('接收到用户传递的数据：' + data);
        // send(),方法向客户端响应数据，只能给当前用户发消息
        connect.send(data);
    });
    // 当有用户websocket连接断开时，触发close事件
    // 注：注册close事件，就必须注册error事件
    connect.on('close', () => {
        console.log("连接断开了");
    });
    // 注册error事件，处理用户出错的信息
    connect.on('error', (err) => {
        console.log('用户连接异常');
    });
});

// 监听POST端口
server.listen(POST, () => {
    console.log('websocket服务启动成功,监听了的端口:' + POST);
})


// 给所有的用户发消息，广播事件
function broadcast(msg){
   // server.connections:是一个数组，包含了所有的用户
```

```
    server.connections.forEach(item=>{
        item.send(msg);
    })
}
```

socket.io

- 建立在WebSocket协议之上，并提供额外的保证，支持退到HTTP长轮询或自动重新连接
- 在浏览器端、服务器端都可以使用
- 注意：
    - Socket.IO **不是** WebSocket 实现
    - WebSocket 客户端将无法成功连接到 Socket.IO 服务器
    - Socket.IO 客户端也无法连接到普通的 WebSocket 服务器
    - Socket.IO 不应在移动应用程序的后台服务中使用，高耗电

1. 服务器端安装 socket.io `npm install socket.io`
2. 客户端
    - Vue：安装 socket.io `npm install socket.io`
    - 原生，直接引入对应js文件即可，或者使用CDN

服务器

- 依赖http模块的服务实例（提供 WebSocket 连接无法实现，它将回退到 HTTP 长轮询。）
- 注意：
    - `const io = require('socket.io')(app);`必须写在http服务端口创建与监听之后，否则报错
    - 因为使用到http服务，所以存在跨域问题；而WebSocket本身不存在跨域问题

```
const http = require('http');
// 创建一个http服务
const app = http.createServer();
// 监听http的request请求
app.on('request', (req, res) => {
    fs.readFile(__dirname + './index.html', function(err, data) {
        if (err) {
            res.writeHead(500);
            return res.end('error loading index.html')
        }
        res.writeHead(200);
        res.end(data);
    })
})
app.listen(8006, function() {
    console.log('8006端口监听中');
})

// 1.引入io模块并传入app实例
const io = require('socket.io')(app);
// 2.监听用户连接事件
```

```
io.on('connection', socket => {
    console.log('有新的用户连接了');
    // socket.emit()方法   触发某个事件,可用于给浏览器发数据
    //  参数1：事件的名字       参数2：传递的数据
    socket.emit('send', { name: '小吴' });
    // socket.on()方法，接受服务器发送来的数据
    //  参数1：事件名;        参数2：事件回调，接受参数，表示服务器发送来的数据
    socket.on('send', (data) => {
       console.log(data);
    });
   // io.emit() 广播事件，给所有已经连接的客户端发送消息
   io.emit('adduser',data);
   // socket.on('disconnect', () => {})  监听用户离开事件
   socket.on('disconnect', () => {
       //处理程序
   })
})
```

客户端

```
// 前提：正确引入了socket.io.js文件
// 连接socket服务 参数为后台创建的http服务地址和端口
var socket = io('http://127.0.0.1:8006');
// socket.on()方法，接受服务器发送来的数据
socket.on('send', (data) => {
    console.log(data);
});
// scoket.emit()方法，触发事件，向服务器发送数据
socket.emit('go',{password:'123'});
```

进程与线程

进程

-