目录

- 栈和堆，基本数据类型 和 引用数据类型
- js基本数据类型：7种
- 扩展运算符(`...变量`)和 `rest` 运算符(`变量...`)
- 构造函数
  - 构造函数的 `prototype` 和实例对象的 `__proto__` 都指向构造函数原型对象
  - `instanceof`

    ```
    console.log( son instanceof Person );
    ```

  - 继承

    ```
    Worker.prototype = new Person();
    Worker.prototype.constructor = Worker;
    ```

- ECMA6 `class` 语法
  - **构造函数创建**

    ```
    class Person{
        //class属性添加
      constructor(name, sex, age){
        this.name = name;
        this.sex = sex;
        this.age = age;
      }
      //class方法的添加
      showSelf(){
        console.log(`我叫${this.name},是一位${this.sex}性。`);
      }
    }
    var p1 = new Person(参数);
    ```

  - **构造函数的继承** (`extends`)

    ```
    class Worker extends Person{
      //属性的继承
      constructor(name, sex, age, job){
        super(name, sex, age);
        this.job = job;
      }
      【注】class extends会自动继承方法。

      //添加自己的方法
      showJob(){
        console.log(this.job);
      }
    }
    ```

- 容易混乱 `this` 指向时，使用 `this` 的方法
  - 声明变量存储 `this`

    ```
    let _this = this;
    ```

  - `bind()` 改变 `this` 指向

    ```
    setTimeout(this.show.bind(this), 4000);
    ```

1.栈和堆，基本数据类型 和 引用数据类型

- 堆(heap):
  1. 堆是没有结构的，数据可以任意存放。堆用于复杂数据类型（引用类型）分配空间，例如数组对象、object对象。
  2. 堆是动态分配内存，内存大小不一，不会自动释放
- 栈(stack):
  1. 栈是有结构的，每个区块按照一定次序存放（后进先出），栈中主要存放一些基本类型的 变量和对象 的引用，存在栈中的 数据大小 与 生存期 必须是确定的。可以明确知道每个区块的大小，因此，栈 的寻址速度要快于 堆。
  2. 栈是自动分配相对**固定大小**的内存空间，并由系统自动释放

- js中基本数据类型
  - `Number`
  - `Boolean`
  - `String`
  - `null`
  - `undefined`
  - `Object`（`Array`、`Data`...）
  - `Symbol`

- JS中 `Object` (引用/复合) 类型
  - 对象、数组、字符串、Set集合/Map集合、函数、日期对象、Math对象、history历史记录、location地址栏....

  - 引用数据类型(`Object`) 都是通过 栈的指针，来访问堆中的数据

2.扩展运算符 `...变量` (和rest：`变量...` 相反)

- 扩展运算符(`...`)用于**取出** 参数**对象**中的所有 **可遍历属性**，**拷贝到当前对象之中**.
  - 取出可遍历的属性或元素
- 对象 中使用扩展运算符
  - 对象的扩展运算符 基本只用在拷贝对象，和 `Object.assign()` 一样是浅拷贝

```
let bar = { a: 1, b: 2 };
let baz = { ...bar }; // { a: 1, b: 2 }
// 扩展运算符会改变 第一层变量指针，因为是浅拷贝

// 上述方法实际上等价于：
let bar = { a: 1, b: 2 };
let baz = Object.assign({}, bar); // { a: 1, b: 2 }
```

- 数组 中使用扩展运算符
  1. 复制数组 (浅拷贝)，类似于 `arr.concat()`

     ```
     var arr1 = [10,20,30];
     var arr2 = [...arr1];
     console.log(arr2);  //10,20,30
     ```

  2. 拼接数组，类似于 `arr.concat()` (`concat` 也属于浅拷贝，拷贝不了二维数组)

     ```
     var arr1 = [10,20];
     var arr2 = [30,40];
     var arr3 = [...arr1,...arr2];
     console.log(arr3);  //[10,20,30,40]
     ```

  3. 将伪数组转为真数组

     ```
     var arr = [...伪数组];    //类似于Array.from(伪数组);
     ```

  4. 使用 `push` 和 `...` 拷贝数组，浅拷贝

     ```
     var arr = [];
     var arr2 = [10,20,30];
     arr.push(...arr2);
     alert(arr); //10,20,30
     ```z
     ```

- 字符串 中使用扩展运算符
  - 将字符串变为数组，类似于： `str.split("")` 空字符串分割

    ```
    var str = "hello";
    var arr = [...str];
    console.log(arr);  //"h" "e" "l" "l" "o"
    ```

- Math方法 使用扩展运算符 (可以传入多个参数)
  - 类似于 `Math.max.apply(arr);`

    ```
    var arr = [10,20,30,40,50];
    console.log(Math.max(...arr)); //50
    ```

- Set集合 中使用扩展运算符
  - 扩展运算符遍历取值 `Set` 时,取值的是 `value` .
    ```

- Set集合变数组

```
var arr = [...set];
```

- Map集合 中使用扩展运算符
  - 扩展运算符遍历取值`Map`时,取值的是`[key, value]`这样的一个个数组

3.关于引用数据类型的指针

- 两个原则
  1. 变量只能存储 **引用数据类型的地址，地址指向堆中的引用数据类型内存。或基本数据类型**
  2. 拥有指针的变量 被重新赋值，就会丢失指针。

```
var a = { a: 1};
var c = a;  //c 复制了一份 a的指针
a = 2;  //这时 a 的不再存储{a:1} 的地址;但是c依旧拥有{a:1}的地址。
```

4.rest运算符：`...变量名` 将数据合并成数组，和扩展运算符相反

- `rest`运算符可以与解构赋值结合起来，用于生成数组

```
var [first, ...last] = [10,20,30,40];
console.log(first); //10
console.log(last);  //[20, 30, 40]
```

- `rest`运算符常常用在函数形参中
  - `rest`运算符只能放在末尾

```
function show(a,b,...c){}
```

- **扩展**：判断数据类型：`Object.prototype.toString.call(要判断的数据);`

5.复习面向对象

- 面向过程编程思想：只考虑数学逻辑
- 面向对象编程思想：直接将生活逻辑映射到我们的程序
  1. 分析有哪些实体
  2. 设计实体的属性功能
  3. 实体间的相互作用
- 有一辆车，时速60km/h，一条路1000km，问题：如果让这辆车跑完这条路，需要多长时间
  - 面向过程编程思想：只考虑数学逻辑

```
var hours = 1000 / 60;
alert(hours);
```

  - 面向对象编程思想：

```
var cars = {
  speed: 60,
  run: function(road){
    return road.length / this.speed;
  }
}

var kuahai = {
  length: 1000
}

var hours = cars.run(kuahai);
alert(hours);
```

6.构造封装函数

- **工厂模式**封装创建对象的函数
  - 工厂模式：1.采集原料；2.车间加工；3.出厂
    - 凡是满足上述三个步骤创建对象的函数，我们把它叫做工厂方法

```
function createPerson(name, sex){
  //1.原料
  var obj = new Object();

  //2.加工
  obj.name = name;
  obj.sex = sex;
  obj.showName = function(){
    return `我的名字叫做 :${this.name},  ` ;
  }
  obj.showSex = function(){
    return `我是一位${this.sex}性 `;
  }

  //3.出厂
  return obj;
}

var p1 = createPerson("小明","男");  //小明 , 男
var p2 = createPerson("小丽","女");  //小丽 , 女
```

7.官方函数创建对象

```
typeof Array    //function
var arr = new Array(); //创建一个数组对象，Array是一个函数
```

- 官方函数创建对象和自己工厂模式的区别
  - 工厂模式：1.没有`new`；2.每一个新创建出来的对象拥有自己的一套函数（方法）

8.让工厂模式有`new`

- 如果，我们某一个函数，使用`new`运算符去调用
  1. 当前函数中的`this`**指向新创建的对象**
  2. 自动完成 1.原料操作 和 3.出厂操作(`return`)
- 通过`new`调用的函数，**叫做构造函数**，构造函数可以构造对象
  - 构造函数为了和普通函数区分，一般情况下，**首字母大写。**

```
function NewPerson(name){
  //1.自动原料操作
  // this = new Obejct();

  // 2.加工
  this.name = name;

  //3.自动return （如果手动return一个对象，就不会返回this）
  //return this
}
```

  - new在执行时会做四件事情：
    1. 在内存中创建一个新的空对象。
    2. 让`this`指向这个新的对象。
    3. 执行构造函数里面的代码，给这个对象添加属性和方法。
    4. 返回这个新对象（所以构造函数里面不需要return，但是如果显性返回一个对象，就不会返回这个对象）

9.添加数组方法

```
// 给数组添加求和方法
var arr = [1,2,3,4]
arr.sum = function(){
  return this.reduce((prev,item) => prev+item , 0)
}
console.log(arr.sum()) //10
```

10.prototype原型对象

- `prototype`原型对象
  - 概念：每一个函数上都有一个`prototype`原型对象
- 用在构造函数上，我们可以给构造函数的原型`prototype`，添加方法
  - 如果我们**将方法添加到构造函数的原型`prototype`上**，构造函数**构造出来的对象共享原型上的所有方法。**

通过new调用的函数，叫做构造函数，构造函数可以构造对象
new Array() new String() new Date()

11.混合法-构造函数

```
//函数的属性写在构造函数内
function Person(name, sex){
  // 省略 1.创建 和 3.返回步骤
  //2.加工
  this.name = name;
  this.sex = sex;
}
//Preson构造函数添加方法,添加在构造函数原型上，prototype上,构建出来的对象可以访问这些方法
Person.prototype.showName = function(){
  return `我的名字叫做 :${this.name},  ` ;
}
Person.prototype.showSex = function(){
  return `我是一位${this.sex}性 `;
}

//不能忘记new
var p1 = new Person("小明","男");  //小明 , 男
var p2 = new Person("小丽","女");  //小丽 , 女
alert(p1.showName());
alert(p2.showName());

alert(p1.showName === p2.showName);//true
```

- 构造函数注意点
  1. 函数的属性写在构造函数内
  2. 函数的方法写在构造函数prototype原型上
  3. 调用构造函数时，要在构造函数前加new.

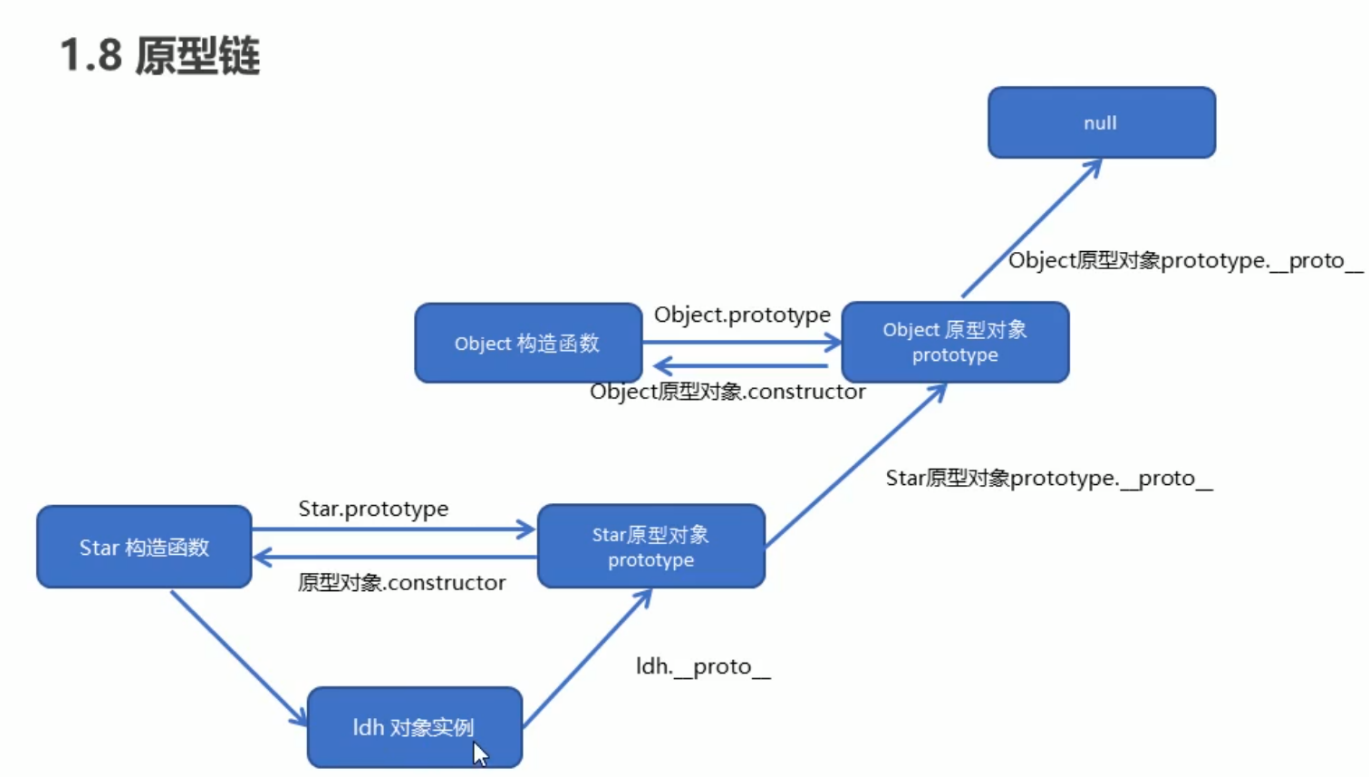
12. `__proto__` 和 `prototype` 原型对象的关系 和 `instanceof` 关键字

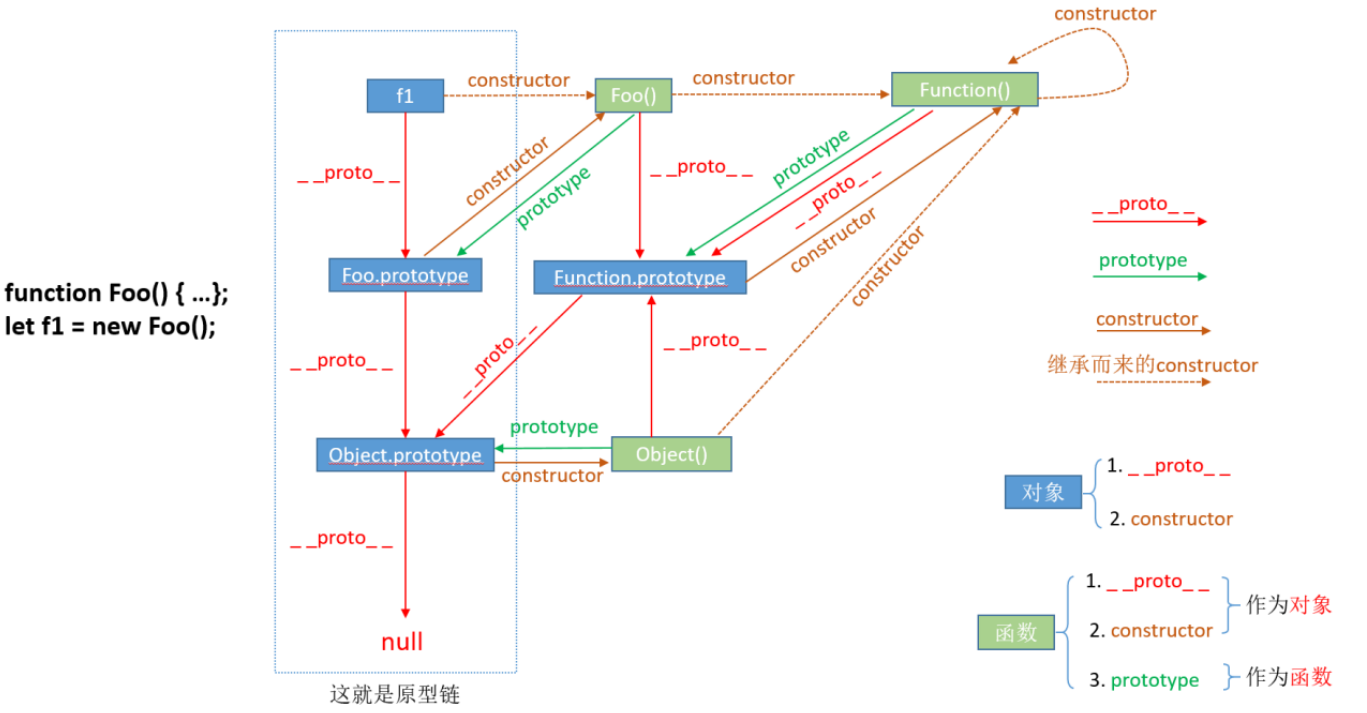
- `实例对象.__proto__` 指向 构造函数原型对象
- `构造函数.prototype` 指向 构造函数原型对象
  - `实例对象.__proto__ === 构造函数.prototype`

- `instanceof` 关键字
  - 功能：判断类的 `prototype` 对象是否存在于实例对象的原型链上。是则返回true。
    - 实例对象的原型链是否和这个 类的 `prototype` 相等
  - 格式： `A instanceof B;`

    ```
    console.log( son instanceof Person );
    ```

  - 分析： `instanceof` 运算符的第一个变量是一个对象，暂时称为 `A`；第二个变量一般是一个函数，暂时称为 `B`。 `instanceof` 的判断规则是：沿着 `A` 的 `__proto__` 这条线来找，同时沿着 `B` 的 `prototype` 这条线来找，如果两条线能找到同一个引用，即同一个对象，那么就返回 `true`。如果找到终点还未重合，则返回 `false`。
- 原型链





13.面向对象的语法

- 面向对象：继承、封装(封装构造函数)、多态
- 面向对象是一个编程思想，支撑面向对象编程思想的语法是 类(ECMA6之前没有类这个概念)和对象
  - ECMA6之前构造函数充当类的角色，构造函数和对象 实现面向对象程序的时候，就会体现出继承、封装、多态的特点

14.继承

- 继承本质上是让写代码更加省时，减少冗余度。
- **继承** 构造函数**属性**:

```
function Teddy(name, type, age, color){
  Dog.call(this, name, type, age);
  //让Dog的this指向Teddy的this

  this.color = color;
}
```

- **继承**构造函数**方法**
  1. 通过`for...in`遍历继承：利用`for...in`会遍历原型链,

     ```
     for(var attr in Person){
       Worker.prototype[attr] = Person.prototype[attr];
     }
     ```

     - 缺点：后期给`Person`原型对象添加方法，也要给`Worker`再次添加方法，**不是真正的继承**
  2. 通过调用构造函数（常用）

     ```
     Worker.prototype = new Person();
     Worker.prototype.constructor = Worker;
     ```

  3. `Object.create()`

     ```
     Worker.prototype = Object.create(Person);
     ```

15.对象的拷贝

- 通过`for...in`循环来进行拷贝

```
for(var attr in Dog.prototype){
  Teddy.prototype[attr] = Dog.prototype[attr]
}
```

  - 缺点：后期给`Dog`原型对象添加方法，也要给`Teddy`再次添加方法，**不是真正的继承**

16.多态

- 继承 和 多态 同一件事情的两种完全不同的侧重
  - 继承：侧重是从父一级构造函数，继承到的属性和方法。
    - 从父级构造函数继承的属性和方法
  - 多态：侧重是，子一级，自己**重写**的和新增的属性和方法。
    - 自己新增的属性和方法。或修改从父构造函数得到的属性和方法

17.ECMA6 `class`语法

- ECMA5 构造函数方法继承
  1. 通过for...in遍历继承
  2. 通过调用构造函数
  3. Object.create()

- ECMA6 `class` 语法
  - **构造函数创建**

```
class Person{
    //class属性添加
  constructor(name, sex, age){
    this.name = name;
    this.sex = sex;
    this.age = age;
  }

  //class方法的添加
  showSelf(){
    console.log(`我叫${this.name},是一位${this.sex}性。`);
  }
}
var p1 = new Person(参数);
```

  - **构造函数的继承**（ `extends` ）

```
class Worker extends Person{
  //属性的继承
  constructor(name, sex, age, job){
    super(name, sex, age);
    this.job = job;
  }
  【注】class extends会自动继承方法。

  //添加自己的方法
  showJob(){
    console.log(this.job);
  }
}
```

    - `super` 代表 `Person` 父类
    - 基本上，ES6 的 `class` 可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的 `class` 写法只是让对象原型的写法更加清晰、更像面向对象编程的语法。

18.面向对象实战和this指向问题

- 拖拽改造
  1. 不能有函数嵌套
  2. 可以有全局变量
- 面向过程 => 面向对象

1. `window.onload` => 构造函数
2. 全局的变量 => 构造函数的属性
3. 全局函数 => 构造函数的方法
   - 遇到`this`指向问题

19.this指向谁

- 总结this容易混乱的部分
  1. 事件绑定：指向绑定的节点
  2. 定时器： 指向`window`
- 事件绑定
  1. 在外面取`this`，里面使用 外面`this`指向的主人

```
oA.onclick = function(){
  var _this = this;   //取外面this值
  //赋值
  oBtn.onclick = function(){
      _this.show();  //在里面使用
  };
}
```

  2. 通过`bind()`改变`this`指向

```
oBtn.onclick = this.show.bind(this);
```

- 定时器
  1. 在外面取`this`，里面使用 外面`this`指向的主人

```
oA.onclick = function(){
  let _this = this;
  setTimeout(function(){
    _this.show();
  }, 4000);
}
```

  2. 通过`bind()`改变`this`指向

```
setTimeout(this.show.bind(this), 4000);
```

20.拖拽继承版和选项改造

1. 继承拖拽-实现限制出界
2. 选项卡-面向对象改造