Javascript基础 - 面向对象编程/原型链/继承

#前端知识自检

第一章 - 面向对象编程

什么是面向对象编程?

面向对象是一种编程思想，经常被拿来和面向过程比较。

其实说的简单点，

面向过程关注的重点是动词，是分析出解决问题需要的步骤，然后编写函数实现每个步骤，最后依次调用函数。

而面向对象关注的重点是主谓，是把构成问题的事物拆解为各个对象，而拆解出对象的目的也不是为了实现某个步骤，而是为了描述这个事物在当前问题中的各种行为。

面向对象的特点是什么？是
封装：让使用对象的人不考虑内部实现，只考虑功能使用 把内部的代码保护起来，只留出一些 api 接口供用户使用
继承：就是为了代码的复用，从父类上继承出一些方法和属性，子类也有自己的一些属性
多态：是不同对象作用于同一操作产生不同的效果。多态的思想实际上是把“想做什么”和“谁去做”分开

比如下棋的过程,

面向过程是这样的：开局 ➔ 白方下棋 ➔ 棋盘展示 ➔ 检查胜负 ➔ 黑方下棋 ➔ 棋盘展示 ➔ 检查胜负 ➔ 循环
用代码表示可能是一连串函数的调用

```
init();
whitePlay(); // 里面实现一遍下棋的操作
repaint(); // 棋盘展示
check();
blackPlay(); // 再单独实现一遍下棋的操作
repaint(); // 棋盘展示
check();
```

面向对象是这样的：棋盘.开局 ➜ 选手.下棋 ➜ 棋盘.重新展示 ➜ 棋盘.检查胜负 ➜ 选手.下棋 ➜ 棋盘.重新展示 ➜ 棋盘.检查胜负
用代码表示可能是这样的

```
const checkerBoard = new CheckerBoard(); // CheckerBoard 类内部封账了棋盘的操作，比如初始化棋盘，检查胜负关系等
const whitePlayer = new Player('white'); // Player 类内部封装了各种玩家的操作，比如等待，落棋，悔棋
const blackPlayer = new Player('black');

whitePlayer.start(); // start 方法的结束，内部封装了或者通过事件发布触发 checkerBoard.repaint(), checkerBoard.check()的调用
blackPlayer.start();
```

你只需要调用 new 一个 player, 然后调用 start 方法，也就是说我们只需要关注行为，而不需要知道内部到底做了什么。

而且如果要加一些新功能，比如悔棋，比如再加一个玩家，面向对象都很好扩展。

在上面的例子中，面向对象的特性是怎么表现出来的呢?

封装：Player, CheckerBoard 类，使用的时候并不需要知道内部实现了什么，只需要考虑暴露出的 api 的使用
继承：whitePlayer 和 blackPlayer 都继承自 Player，都可以直接使用 Player 的各种方法和属性
多态：whitePlayer.start() 和 blackPlayer.start() 下棋的颜色分别是白色和黑色

什么时候适合使用面向对象

可以看出来，在比较复杂的问题面前，或者参与方较多的时候，面向对象的编程思想可以很好的简化问题，并且能够更好的扩展和维护。

而在比较简单的问题面前，面向对象和面向过程其实差异并不明显，也可以一步一步地按照步骤来调用。

Js 中的面向对象

对象包含什么

方法
属性

一些内置对象

Object Array Date Function RegExp

创建对象

1. 普通方式

每一个新对象都要重新写一遍 color 和 start 的赋值

```
const Player = new Object();
Player.color = "white";
Player.start = function () {
  console.log("white下棋");
};
```

或者工厂模式，这两种方式都无法识别对象类型，比如 Player 的类型只是 Object

```
function createObject() {
  const Player = new Object();
  Player.color = "white";
  Player.start = function () {
    console.log("white下棋");
  };
  return Player;
}
```

2. 构造函数/实例

通过 this 添加的属性和方法总是指向当前对象的，所以在实例化的时候，通过 this 添加的属性和方法都会在内存中复制一份，这样就会造成内存的浪费。

但是这样创建的好处是即使改变了某一个对象的属性或方法，不会影响其他的对象（因为每一个对象都是复制的一份）

```
function Player(color) {
  this.color = color;
  this.start = function () {
    console.log(color + "下棋");
  };
}

const whitePlayer = new Player("white");
const blackPlayer = new Player("black");
```

Tips. 怎么看函数是不是在内存中创建了多次呢?

比如 2. 构造函数中，我们可以看到 whitePlayer.start === blackPlayer.start // 输出 false

3. 原型

通过原型继承的方法并不是自身的，我们要在原型链上一层一层的查找，这样创建的好处是只在内存中创建一次，实例化的对象都会指向这个 prototype 对象。

```
function Player(color) {
  this.color = color;
}

Player.prototype.start = function () {
  console.log(color + "下棋");
};

const whitePlayer = new Player("white");
const blackPlayer = new Player("black");
```

4. 静态属性

是绑定在构造函数上的属性方法，需要通过构造函数访问

比如我们想看一下一共创建了多少个玩家的实例

```
function Player(color) {
  this.color = color;
  if (!Player.total) {
    Player.total = 0;
  }
  Player.total++;
}

let p1 = new Player("white");
console.log(Player.total); // 1
let p2 = new Player("black");
console.log(Player.total); // 2
```

第二章 - 原型及原型链

在原型上添加属性或者方法有什么好处?

刚才已经说过了，如果不通过原型的方式，每生成一个新对象，都会在内存中新开辟一块存储空间，当对象变多之后，性能会变得很差。

但是通过

```
Player.prototype.xx = function () {};
Player.prototype.xx = function () {};
Player.prototype.xx = function () {};
```

这种方式向原型对象添加属性或者方法的话，又显得非常麻烦。所以我们可以这样写

```
Player.prototype = {
  start: function () {
    console.log("下棋");
  },
  revert: function () {
    console.log("悔棋");
  },
};
```

怎么找到 Player 的原型对象?

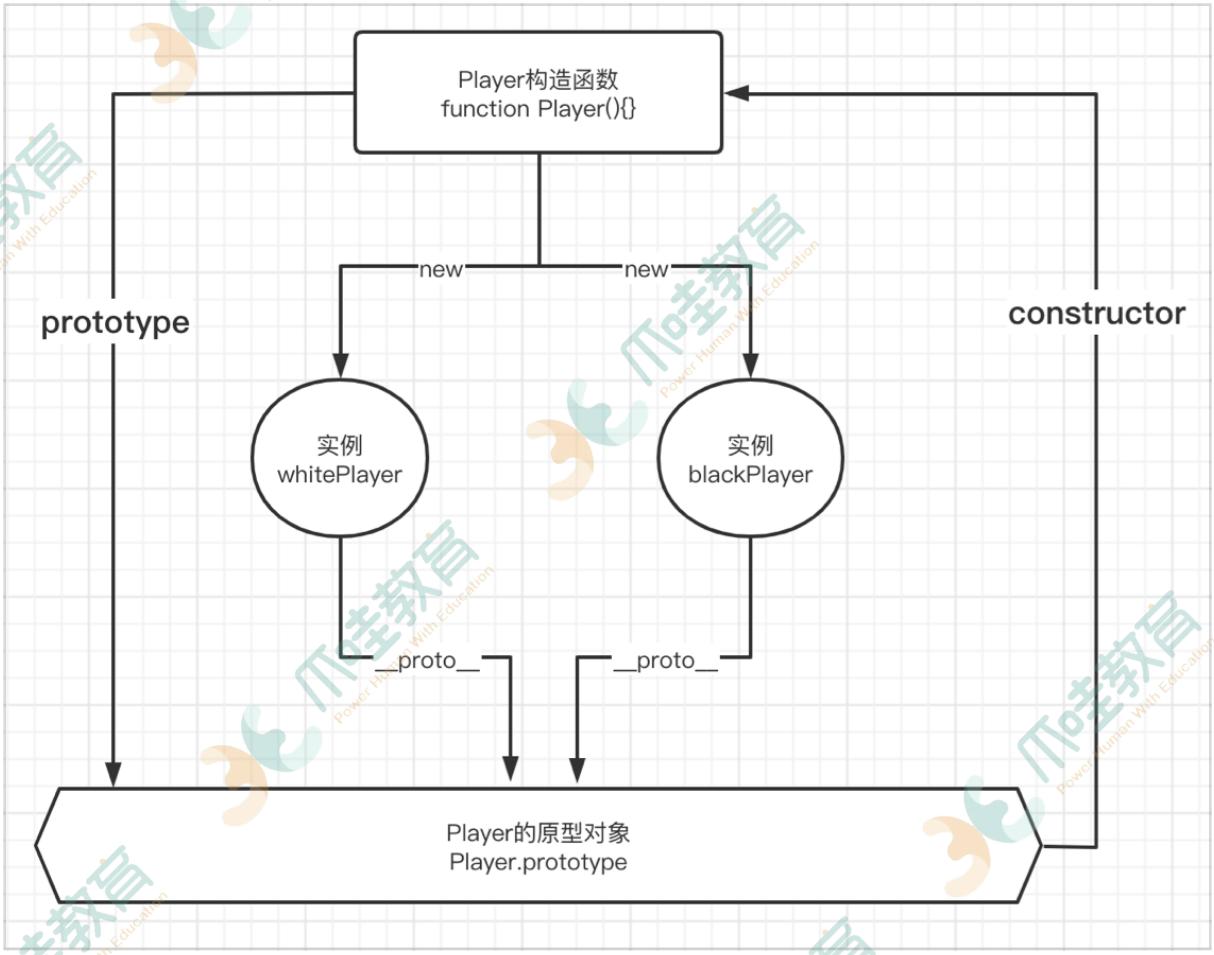
```
function Player(color) {
  this.color = color;
}

Player.prototype.start = function () {
  console.log(color + "下棋");
};

const whitePlayer = new Player("white");
const blackPlayer = new Player("black");

console.log(blackPlayer.__proto__); // Player {}
console.log(Object.getPrototypeOf(blackPlayer)); // Player {}, 可以通过
Object.getPrototypeOf来获取__proto__
console.log(Player.prototype); // Player {}
console.log(Player.__proto__); // [Function]
```

可以看一下 prototype.png 原型的流程图



那么 new 关键字到底做了什么?

1. 一个继承自 Player.prototype 的新对象 whitePlayer 被创建
2. whitePlayer.**proto** 指向 Player.prototype，即 whitePlayer.**proto** = Player.prototype
3. 将 this 指向新创建的对象 whitePlayer
4. 返回新对象

   4.1 如果构造函数没有显式返回值，则返回 this

   4.2 如果构造函数有显式返回值，是基本类型，比如 number,string,boolean, 那么还是返回 this

   4.3 如果构造函数有显式返回值，是对象类型，比如{ a: 1 }, 则返回这个对象{ a: 1 }

后面看一下怎么手写实现 new 函数

```
// 1. 用new Object() 的方式新建了一个对象 obj
// 2. 取出第一个参数，就是我们要传入的构造函数。此外因为 shift 会修改原数组，所以
arguments 会被去除第一个参数
```

```
// 3. 将 obj 的原型指向构造函数，这样 obj 就可以访问到构造函数原型中的属性
// 4. 使用 apply，改变构造函数 this 的指向到新建的对象，这样 obj 就可以访问到构造函数中的
属性
// 5. 返回 obj
function objectFactory() {
  let obj = new Object();
  let Constructor = [].shift.call(arguments);
  obj.__proto__ = Constructor.prototype;
  let ret = Constructor.apply(obj, arguments);
  return typeof ret === "object" ? ret : obj;
}
```

原型链又是什么呢?

我们都知道当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还查不到，就去找原型的原型，一直找到最顶层为止。

举个例子

```
function Player() {}

Player.prototype.name = "Kevin";

var p1 = new Player();

p1.name = "Daisy";
// 查找p1对象中的name属性，因为上面添加了name，所以会输出“Daisy”
console.log(p1.name); // Daisy

delete p1.name;
// 删除了p1.name，然后查找p1发现没有name属性，就会从p1的原型p1.__proto__中去找，也就是
Player.prototype，然后找到了name，输出"Kevin"
console.log(p1.name); // Kevin
```

那如果我们在 Player.prototype 中也找不到 name 属性呢,那么就会去 Player.prototype.**proto**

中去寻找，也就是{}。

```
Object.prototype.name = "root";

function Player() {}

Player.prototype.name = "Kevin";

var p1 = new Player();

p1.name = "Daisy";
// 查找p1对象中的name属性，因为上面添加了name，所以会输出“Daisy”
console.log(p1.name); // Daisy

delete p1.name;
// 删除了p1.name，然后查找p1发现没有name属性，就会从p1的原型p1.__proto__中去找，也就是
Player.prototype，然后找到了name，输出"Kevin"
console.log(p1.name); // Kevin

delete Player.prototype.name;

console.log(p1.name);
```

这样一条通过**proto**和 prototype 去连接的对象的链条，就是原型链

第三章 - 继承

原型链继承

实现

```
function Parent() {
  this.name = "parentName";
```

```
}

Parent.prototype.getName = function () {
  console.log(this.name);
};

function Child() {}

// Parent的实例同时包含实例属性方法和原型属性方法，所以把new Parent()赋值给Child.prototype。
// 如果仅仅Child.prototype = Parent.prototype，那么Child只能调用getName，无法调用.name
// 当Child.prototype = new Parent()后， 如果new Child()得到一个实例对象child，那么
// child.__proto__ === Child.prototype;
// Child.prototype.__proto__ === Parent.prototype
// 也就意味着在访问child对象的属性时，如果在child上找不到，就会去Child.prototype去找，如果还找不到，就会去Parent.prototype中去找，从而实现了继承。
Child.prototype = new Parent();
// 因为constructor属性是包含在prototype里的，上面重新赋值了prototype，所以会导致Child的constructor指向[Function: Parent]，有的时候使用child1.constructor判断类型的时候就会出问题
// 为了保证类型正确，我们需要将Child.prototype.constructor 指向他原本的构造函数Child
Child.prototype.constructor = Child;

var child1 = new Child();

child1.getName(); // parentName
```

隐含的问题

1. 如果有属性是引用类型的，一旦某个实例修改了这个属性，所有实例都会受到影响

```
function Parent() {
  this.actions = ["eat", "run"];
}
function Child() {}
```

```
Child.prototype = new Parent();
Child.prototype.constructor = Child;

const child1 = new Child();
const child2 = new Child();

child1.actions.pop();

console.log(child1.actions); // ['eat']
console.log(child2.actions); // ['eat']
```

2. 创建 Child 实例的时候，不能传参

构造函数继承

看到上面的问题 1，我们想一下该怎么解决呢?

能不能想办法把 Parent 上的属性方法，添加到 Child 上呢？而不是都存在原型对象上，防止被所有实例共享。

实现

针对问题 1. 我们可以使用 call 来复制一遍 Parent 上的操作

```
function Parent() {
  this.actions = ["eat", "run"];
  this.name = "parentName";
}

function Child() {
  Parent.call(this);
}

const child1 = new Child();
const child2 = new Child();
```

```
child1.actions.pop();

console.log(child1.actions); // ['eat']
console.log(child1.actions); // ['eat', 'run']
```

针对问题 2. 我们应该怎么传参呢?

```
function Parent(name, actions) {
  this.actions = actions;
  this.name = name;
}

function Child(id, name, actions) {
  Parent.call(this, name); // 如果想直接传多个参数, 可以Parent.apply(this,
Array.from(arguments).slice(1));
  this.id = id;
}

const child1 = new Child(1, "c1", ["eat"]);
const child2 = new Child(2, "c2", ["sing", "jump", "rap"]);

console.log(child1.name); // { actions: [ 'eat' ], name: 'c1', id: 1 }
console.log(child2.name); // { actions: [ 'sing', 'jump', 'rap' ], name: 'c2',
id: 2 }
```

隐含的问题

属性或者方法想被继承的话，只能在构造函数中定义。而如果方法在构造函数内定义了，那么每次创建实例都会创建一遍方法，多占一块内存。

```
function Parent(name, actions) {
  this.actions = actions;
  this.name = name;
  this.eat = function () {
    console.log(`${name} - eat`);
```

```
  };
}

function Child(id) {
  Parent.apply(this, Array.prototype.slice.call(arguments, 1));
  this.id = id;
}

const child1 = new Child(1, "c1", ["eat"]);
const child2 = new Child(2, "c2", ["sing", "jump", "rap"]);

console.log(child1.eat === child2.eat); // false
```

组合继承

通过原型链继承我们实现了基本的继承，方法存在 prototype 上，子类可以直接调用。但是引用类型的属性会被所有实例共享，并且不能传参。

通过构造函数继承，我们解决了上面的两个问题：使用 call 在子构造函数内重复一遍属性和方法创建的操作，并且可以传参了。

但是构造函数同样带来了一个问题，就是构造函数内重复创建方法，导致内存占用过多。

是不是突然发现原型链继承是可以解决方法重复创建的问题？ 所以我们将这两种方式结合起来，这就叫做组合继承

实现

```
function Parent(name, actions) {
  this.name = name;
  this.actions = actions;
}

Parent.prototype.eat = function () {
  console.log(`${this.name} - eat`);
};
```

```
function Child(id) {
  Parent.apply(this, Array.from(arguments).slice(1));
  this.id = id;
}

Child.prototype = new Parent();
Child.prototype.constructor = Child;

const child1 = new Child(1, "c1", ["hahahahahhah"]);
const child2 = new Child(2, "c2", ["xixixixixixx"]);

child1.eat(); // c1 - eat
child2.eat(); // c2 - eat

console.log(child1.eat === child2.eat); // true
```

隐含的问题

调用了两次构造函数，做了重复的操作

1. Parent.apply(this, Array.from(arguments).slice(1));
2. Child.prototype = new Parent();

寄生组合式继承

上面重复调用了 2 次构造函数，想一下，我们可以精简掉哪一步?

我们可以考虑让 Child.prototype 间接访问到 Parent.prototype

实现

```
function Parent(name, actions) {
  this.name = name;
  this.actions = actions;
}
```

```
Parent.prototype.eat = function () {
  console.log(`${this.name} - eat`);
};

function Child(id) {
  Parent.apply(this, Array.from(arguments).slice(1));
  this.id = id;
}

// 模拟Object.create的效果
// 如果直接使用Object.create的话，可以写成Child.prototype =
Object.create(Parent.prototype);
let TempFunction = function () {};
TempFunction.prototype = Parent.prototype;
Child.prototype = new TempFunction();

Child.prototype.constructor = Child;

const child1 = new Child(1, "c1", ["hahahahahhah"]);
const child2 = new Child(2, "c2", ["xixixixixixx"]);
```

也许有的同学会问，为什么一定要通过桥梁的方式让 Child.prototype 访问到 Parent.prototype?
直接 Child.prototype = Parent.prototype 不行吗?
答：不行！！

咱们可以来看一下

```
function Parent(name, actions) {
  this.name = name;
  this.actions = actions;
}

Parent.prototype.eat = function () {
  console.log(`${this.name} - eat`);
};
```

```
function Child(id) {
  Parent.apply(this, Array.from(arguments).slice(1));
  this.id = id;
}

Child.prototype = Parent.prototype;

Child.prototype.constructor = Child;

console.log(Parent.prototype); // Child { eat: [Function], childEat:
[Function] }

Child.prototype.childEat = function () {
  console.log(`childEat - ${this.name}`);
};

const child1 = new Child(1, "c1", ["hahahahahhah"]);

console.log(Parent.prototype); // Child { eat: [Function], childEat:
[Function] }
```

可以看到，在给 Child.prototype 添加新的属性或者方法后，Parent.prototype 也会随之改变，这可不是我们想看到的。