**02-OOP与继承-20231202**

**自我介绍**

麓一。

* 不用预习；
* 知识体系；
* 50%；

授之以渔。

学习：

* **器**：工具， Javascript, Vue, React, Angular...Webpack, node.
* **术**：用前端做了什么项目。React 设计了什么，针对什么问题，解决了什么。
* **道**：知识体系。 -> **以道驭术**。

*邓宁克鲁格效应。*

**课程大纲**

*JS 面向对象编程*

*原型与原型链*

*继承*

**课程目标**

* 搞清楚OOP
* 搞清楚原型链
* 知道组合寄生继承，知道 class 继承
* 知道怎么创建对象， function / class.

**什么是面向对象**

Object-oriented Programming，- Java 和设计模式。

Java - 一切皆对象。

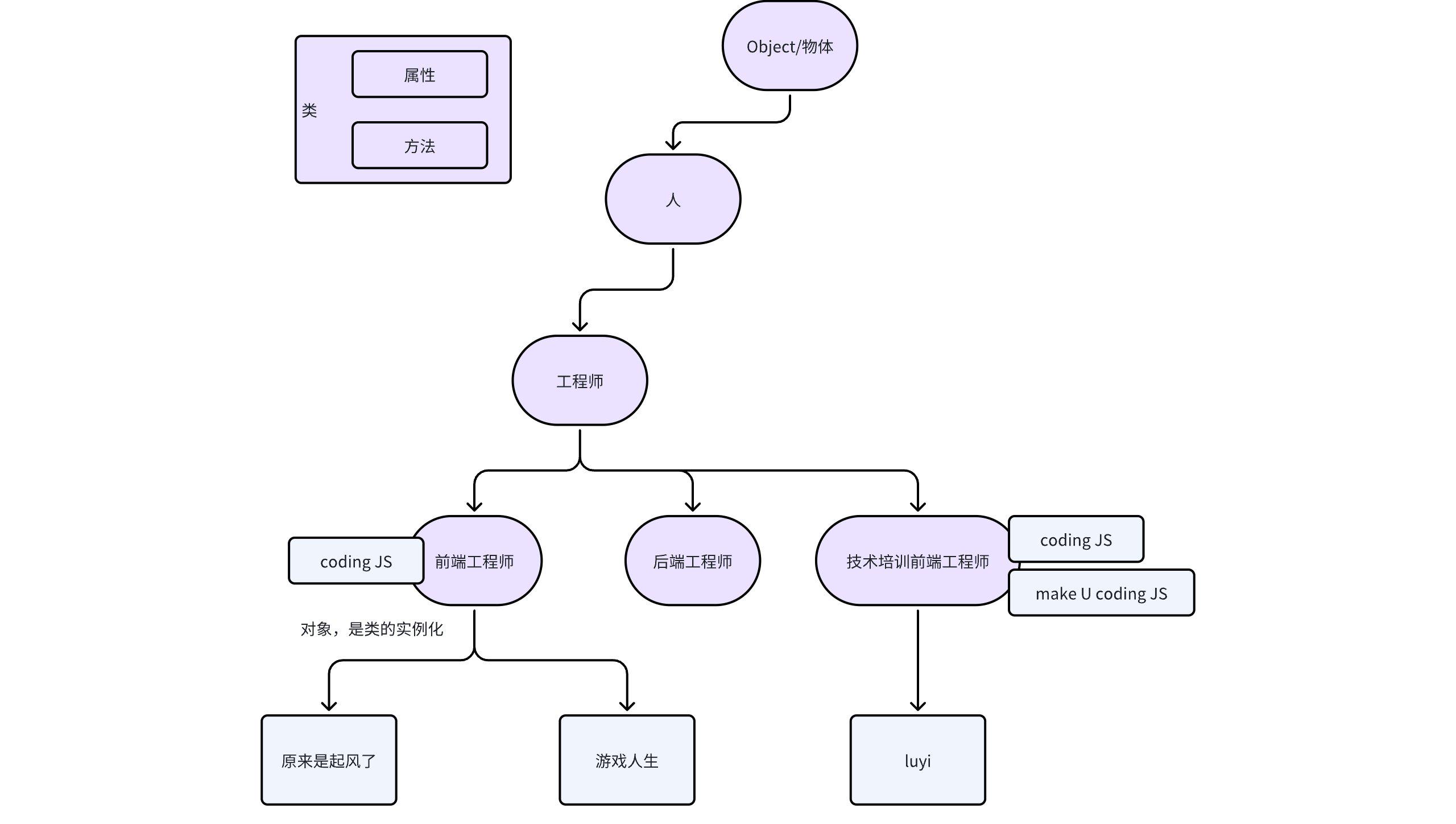
**对比**

* 面向过程 - C

|  |
| --- |
| JavaScript // utils/index.js   {  namelist: [  { text: 'lu' }, {text: 'yi'}  ] }  export function getUserName(obj) {  // 只是一个数据的 pipeline   return obj.namelist.reduce((total, item) => `${total.text}.${item.text}`) }; export function getUserSex(obj) {  // 数据的 pipeline   if(obj.hasSth) {  return "MALE"  } else {  return "FEMALE"  } }  getUserName(obj) |

* 面向对象 - Java

|  |
| --- |
| JavaScript // libs/user class User {  constructor(obj) {  this.obj = obj  };  getName() {  this.obj.namelist.reduce((total, item) => `${total.text}.${item.text}`)  };  getSex() {  if(this.obj.hasSth) {  return "MALE"  } else {  return "FEMALE"  }  } };  const **user** = new User(obj); user.getName(); |



* 面向切面 - Spring

**举例**

下棋

**面向过程：**

|  |
| --- |
| JavaScript init();  whitePlay(); repaint();  check();  blackPlay(); repaint();  check();  whitePlay(); repaint();  check();  blackPlay(); repaint();  check(); |

**面向对象：**

|  |
| --- |
| JavaScript // 创建一个棋盘，CheckerBoard， 内部封装了各种棋盘的操作。 const checkerBoard = new CheckerBoard();  const whitePlayer = new Player('white'); const blackPlayer = new Player('black');   whitePlayer.start(); checkerBoard.repaint(); checkerBoard.check();  blackPlayer.start();  whitePlayer.rollback(); |

**React**

|  |
| --- |
| JavaScript // class, render ? function no render ?  class Button extends Component {  render() {  return <div></div>  } };  new Button().render();   function Card () {    // useMemo, useCallback???  const handleClick = () => {    }   return <div onClick={handleClick}></div> }  Card(); |

**JS 中的面向对象**

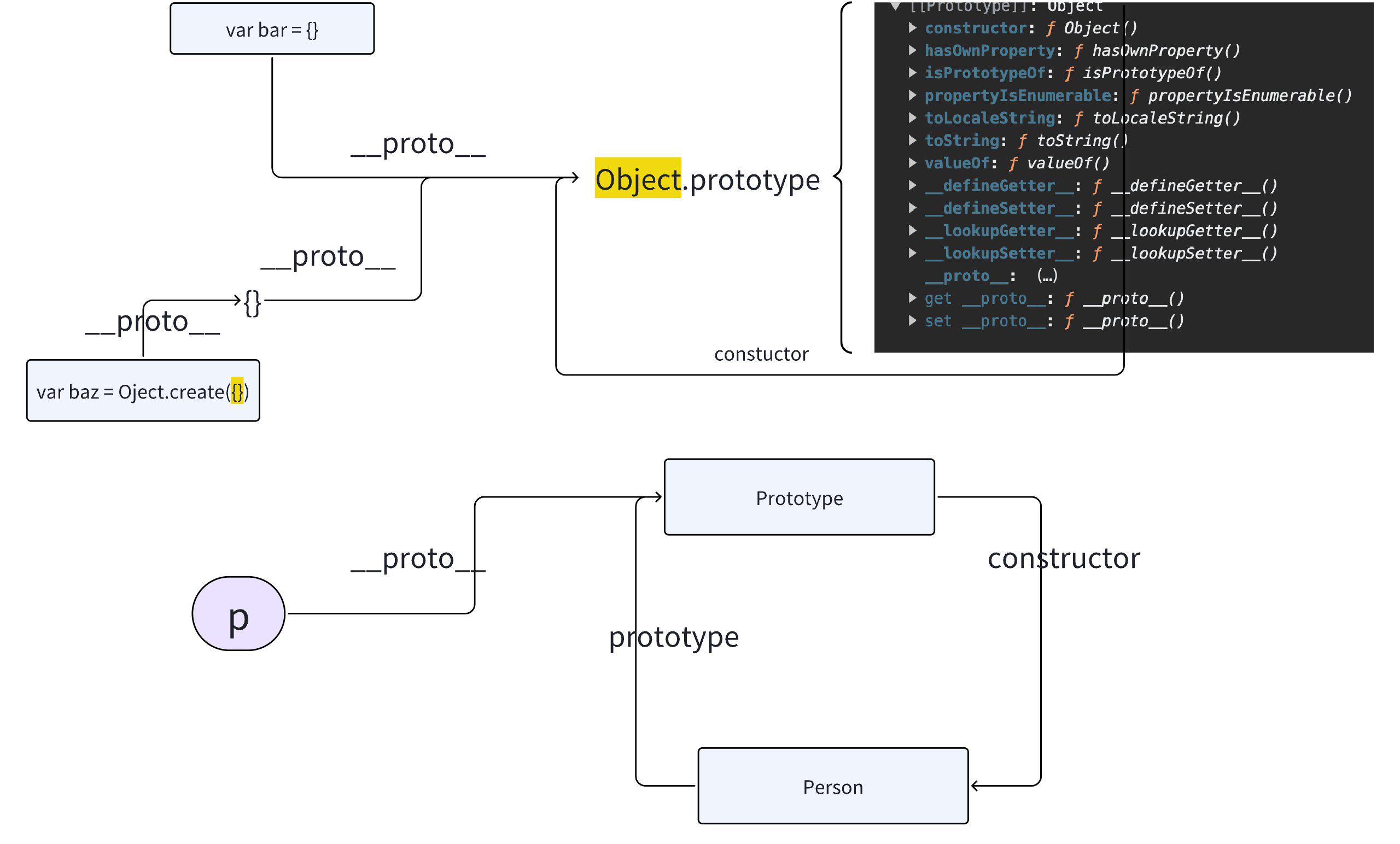
*JS 中创建一个对象，有哪些方法？*

*类和对象之间的关系*

|  |
| --- |
| JavaScript Object.create() var bar = {}; var p = new Person(); |

**Object.create / {}**

var bar = {} 和 Object.create({});



*如果用 Object.create 创建一个对象，类似于 var bar = {} 一样的效果？*

**\_\_proto\_\_**

本质是原型链关系，JavaScript这门语言，本质设计的时候，就是原型链关系。

A.\_\_proto\_\_ === B 的时候。

A.xxx， 如果 A上面没有xxx, 我可以去 B上面去找，如果B上面有，我就用 B 的。

**new 关键字**

|  |
| --- |
| JavaScript function Person(name, age) {  this.name = name;  this.age = age;  // this.e = function() {}   };  Person.prototype.eat = function() {  };  Person.prototype.walk = function() {  };  const p = new Person("麓一", 35);  **// 三件事** **// 1. new 创建了一个对象，这个对象的原型链，指向了构造函数的原型。** **p.\_\_proto\_\_ === Person.prototype;**  **// 2. Person 有个原型，原型上有个constructor， 是 Person 自己** **Person.prototype.constructor === Person;**  **// 3. p 这个对象，是构造函数构造的，p 的构造是谁呢？很显然，是 Person.** **p.constructor === Person;**   // p 要有能力 p.eat(); p.walk(); |

**new 关键字，到底干了什么？**

* 创建了一个对象；
* 这个对象的原型，指向了这个 Person / Function 的prototype
* 该对象实现了 person 的方法；
* 根据一些特定情况返回对象
* 如果这个构造函数没有返回值，或者返回一个非对象类型，则new 最后返回创建的这个对象（p）；
* 如果这个构造函数明确返回了一个对象，则返回这个对象；

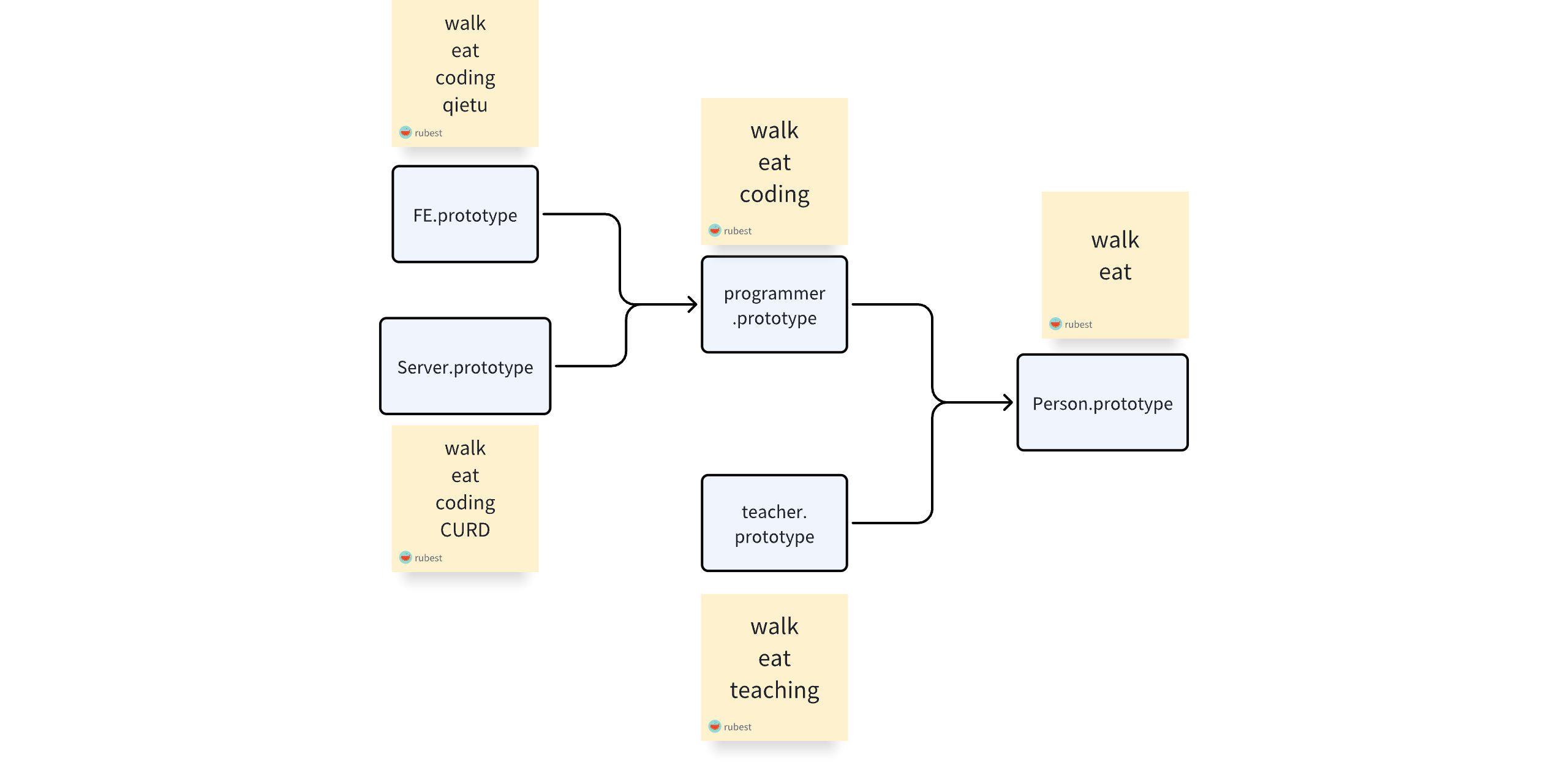
|  |
| --- |
| JavaScript  function newFunc(Person, ...rest) {  if(typeof Person !== "function") {  throw new Error('new operator function the frist param must be a function');  }    var obj = Object.create(Person.prototype);  var result = Person.apply(obj, rest);  return result && typeof result === 'object' ? result : obj; } |

反过来，我们怎么实现一个 Object.create

|  |
| --- |
| JavaScript // 犀牛书，例 6.1  function **inherit**(p) {  if(p === null) throw TypeError();  if(Object.create) {  return Object.create(p)  };    if( typeof p !== "object" && typeof p !== "function") throw TypeError();    function f() {};  f.prototype = p;  return new f(); } |

**继承**

*继承，是描述类和类之间的关系*



**原型继承**

|  |
| --- |
| JavaScript function Parent(name) {  this.name = name;  this.c = "parent" }  Parent.prototype.getName = function() {  console.log(this.name); };  function Child() {};  Child.prototype = new Parent(); Child.prototype.constructor = Child;  // new Child()  // 1. child 不能传参 // 2. 如果 Parent 有属性是引用类型，一旦修改了，所有都受影响。 function Parent(name) {  this.name = name;  this.c = "parent"  this.namelist = ['lu', 'yi'] }  Parent.prototype.getName = function() {  console.log(this.name); };  function Child(sex) {  this.sex = sex; };  Child.prototype = new Parent(); Child.prototype.constructor = Child;  const c1 = new Child(); const c2 = new Child();  c1.namelist.pop();  console.log(c2.namelist) |

**构造函数继承**

|  |
| --- |
| JavaScript function Parent(name) {  this.name = name; }  Parent.prototype.getName = function() {  console.log(this.name); };  function Child(name) {  Parent.call(this, name); } |

**组合继承**

|  |
| --- |
| JavaScript function Parent(name) {  this.name = name; }  Parent.prototype.getName = function() {  console.log(this.name); };  function Child(name) {  Parent.call(this, name); }  // 问题：我只想构建一个 原型链的关系。 Child.prototype = new Parent();  Child.prototype.constructor = Child; |

**组合寄生式继承**

|  |
| --- |
| JavaScript function Parent(name) {  this.name = name; }  Parent.prototype.getName = function() {  console.log(this.name); };  function Child(name) {  Parent.call(this, name); }  // 问题：我只想构建一个 原型链的关系。 // Child.prototype = Object.create(Parent.prototype); Child.prototype = inherit(Parent.prototype) Child.prototype.constructor = Child; |

**Q：**

**组合寄生和 class 的区别？**

*loose 模式，babel 进行降级编译的时候，考虑到体积等，只实现核心功能*

Loose 模式应该差不多，主要是这两个区别：

Class 继承，会继承静态属性；

子类中，必须在 constructor中调用 super, 因为子类自己的 this 对象，必须先通过父类的构造函数完成。