类

es6 引入的 class 类实质上是 JavaScript 基于原型继承的语法糖。

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.sayHi = () => {
  return `Hello ${this.name}`;
};

// 等同于

class Animal {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return `Hello ${this.name}`;
  }
}
```

类由两部分组成：类声明，类表达式

1. 类声明

```
class Animal {
  constructor(name) {
    this.name = name;
  }
}
```

类实际上是个特殊的函数，普通函数声明和类函数声明有一个重要的区别就是函数 声明会提升，而类声明不会。如果先访问，后声明就会抛出类似于下面的错误。

```
let animal = new Animal();
// Uncaught ReferenceError: Cannot access 'Animal' before initialization

class Animal {}
```

2. 类表达式

类表达式可以是被命名的或匿名的，(ps: 类表达式也同样受到类声明中提到的提升问题的困扰。)

```
// 匿名类
let Animal = class {
  constructor(name) {
    this.name = name;
  }
};

// 命名类
let Animal = class Cat {
  constructor(name) {
    this.name = name;
  }

  getClassName() {
    return Cat.name;
  }
};
```

此时类名字Cat只能在 class 内部使用，指代当前类，在类的外部只能用Animal。

3. 构造函数 constructor方法是类的默认方法，通过new创建对象实例时，自动会调用该方法， 一个类必须拥有constructor方法，如果没有写，JavaScript 引擎会默认加上空的constructor方法。

```
class Animal {}

// 等同于

class Animal {
  constructor() {}
}
```

constructor方法默认返回实例对象(既this),完全可以指定返回另外一个对象

```
class Animal {
  constructor() {
    return Object.create(null);
  }
}

new Animal() instanceof Animal; // false;
```

上面代码中，constructor函数返回一个全新的对象，结果导致实例对象不是Animal类的实例。

4. 严格模式 类和模块的内部，默认就是严格模式，比如，构造函数，静态方法，原型方法，getter 和 setter 都在严格模式下执行。

5. 类的实例 类的实例，通过 new 创建, 创建时会自动调用构造函数

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return "My name is " + this.name;
  }
}

let animal = new Animal("rudy");
animal.sayHi(); // My name is rudy
```

6. 存取器 与 ES5 一样，在类的内部可以使用get和set关键字，对某个属性设置存取 函数和取值函数，拦截该属性的存取行为。

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  get name() {
    return "rudy";
  }

  set name(value) {
    console.log("setter, " + this.value);
  }
}

let animal = new Animal("rudy");
animal.name = "Tom"; // setter, Tom
console.log(a.name); // rudy
```

7. 静态方法 使用static修饰符修饰的方法称为静态，它们不需要实例化，直接通过类来调用。

```
class Animal {
  static sayHi(name) {
    console.log("i am " + name);
  }
}

let animal = new Animal();
Animal.sayHi("rudy"); // i am rudy
animal.sayHi("rudy"); // Uncaught TypeError: animal.sayHi is not a function
```

8. 实例属性，静态属性 ES6 中的实例属性只能通过构造函数中的`this.xxx`来定义，但最近 ES7 中可以直接在类里面定义：

```
class Animal {
  name = "rudy";
  static value = 11;
  sayHi() {
    console.log(`hello, ${this.name}`);
  }
}

let animal = new Animal();
animal.sayHi(); // hello, rudy
Animal.value; // 11
animal.value; // undefiend
```

9. 类的继承 使用`extends`关键字实现继承，子类中使用`super`关键字来调用父类的构造函数和方法。

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return "this is " + this.name;
  }
}

class Cat extends Animal {
  constructor(name, value) {
    super(name); // 调用父类的 constructor(name)
    this.value = value;
  }

  sayHi() {
    return `omg, ${super.sayHi()} it is ${this.value}`;
  }
}

let cat = new Cat("Tom", 11);
cat.sayHi(); // omg, this is Tom it is 11;
```

10. super 关键字 `super`这个关键字，既可以当着函数使用，也可以当着对象使用。两种情况下，用法完全不同。

第一种情况，`super`作为函数调用时，代表父类的构造函数。ES6 要求，字类的构造函数必须执行一次`super`函数。

```
class Animal {}

class Cat extends Animal {
  constructor() {
    super();
  }
}
```

上面代码中，子类Cat的构造函数中的super()，代表调用父类的构造函数，这是必须的，否在 JavaScript 引擎会报错。 注意，super虽然代表了父类Animal的构造函数，但是返回的是字类Cat的实例，既super内部的this指的是Cat的实例，因此super()在这里相当于 Animal.prototype.constructor.call(this)。

```
class Animal {
  constructor() {
    console.log(new.target.name); // new.target指向当前正在执行的函数
  }
}

class Cat extends Animal {
  constructor() {
    super();
  }
}

new Animal(); // Animal;
new Cat(); // Cat;
```

可以看出，在super()执行时，它指向的是子类Cat的构造函数，而不是父类Animal的构造函数，也就是说super内部的this指向是Cat。

作为函数时，super()只能用在子类的构造函数之中，用在其他地方就会报错。

```
class Animal {}

class Cat extends Animal {
  hi() {
    super(); // Uncaught SyntaxError: 'super' keyword unexpected here
  }
}
```

第二种情况，super作为对象时:

> 1. 在普通方法中，指向父类的原型对象，
> 2. 在静态方法中，指向父类。

```

```

```
class Animal {
  getName() {
    return "rudy";
  }
}

class Cat extends Animal {
  constructor() {
    super();
    console.log(super.getName());
  }
}

let cat = new Cat(); // rudy;
```

上面代码中，子类Cat中的super.getName(),就是将super当作一个对象使用，这时，super在普通方法中，指向的是Animal.prototype，super.getName()相当于Animal.prototype.getName()。

这里需要注意，由于super指向的是父类原型对象，所以定义在父类实例上的方法和属性，是无法通过super获取到的。

```
class Animal {
  constructor() {
    this.name = "rudy";
  }
}

class Cat extends Animal {
  constructor() {
    super();
  }
  getName() {
    return super.name;
  }
}

let cat = new Cat();
cat.getName(); // undefined;
```

上面代码中，name是父类实例的属性，而不是父类原型对象的属性，所以super.name引用不到它。

用在静态方法中，super将指向父类，而不是父类的原型对象。

```
class Animal {
  static getName(name) {
    console.log("static", name);
  }
```

```
  getName(name) {
    console.log("instance", name);
  }
}

class Cat extends Animal {
  constructor() {
    super();
  }

  static getName(name) {
    super.getName(name);
  }

  getName(name) {
    super.getName(name);
  }
}

Cat.getName("rudy"); // static rudy;

let cat = new Cat();
cat.getName("tom"); // instance tom;
```

在上面代码中，super在静态方法中指向父类，在普通方法中指向父类的原型对象。

另外，在字类的静态方法中通过super调用父类的方法时，方法内部的this指向当前的子类，而不是子类实例。

```
class Animal {
  constructor() {
    this.name = "rudy";
  }

  static print() {
    console.log(this.name);
  }
}

class Cat extends Animal {
  constructor() {
    super();
    this.name = 2;
  }

  static print() {
    super.print();
  }
}
```

```
Cat.name = "Tom";
Cat.print(); // Tom;
```

参考资料

1. 主要参考的是司徒正美老师的文章，ps文章已经很老了，有些问题但并为修复
2. 主要引用的是后面的实例问题，但答案是我自己写的
3. 对正则API的引用