对象、类与面向对象编程

类

ES6新引入class关键字具有正式定义类的能力。
class是ES中新的基础性语法糖结构
ES6表面上支持正式的面向对象编程，实际上话是用的原型与构造函数的概念

类定义

两种方式：类声明和类表达式

```
// 类声明
class Person {}
// 类表达式
const Animal = class {};
```

函数声明可以提升，但是类声明不能提升

```
console.log(a); // [Function: a]
function a() {}
console.log(a); // [Function: a]


console.log(Person); // ReferenceError: Person is not defined
class Person {}
console.log(Person); //[class Person]
```

函数受函数作用域限制，类受块作用域限制

```
{
 function FunctionDeclaration() {}
 class ClassDeclaration {}
}
console.log(FunctionDeclaration); // FunctionDeclaration() {}
console.log(ClassDeclaration); // ReferenceError: ClassDeclaration is not defined
```

类表达式名称是可选的，只能类内部访问

```
let Person = class PersonName {
 identify() {
 console.log(Person.name, PersonName.name);
 }
}
let p = new Person();
```

```
p.identify(); // PersonName PersonName
console.log(Person.name); // PersonName
console.log(PersonName); // ReferenceError: PersonName is not defined
```

类的构成

类可以包含构造函数，实例方法，获取函数，设置函数和静态类方法
类名首字母大写

```
// 空类定义，有效
class Foo {}
// 有构造函数的类，有效
class Bar {
 constructor() {}
}
// 有获取函数的类，有效
class Baz {
 get myBaz() {}
}
// 有静态方法的类，有效
class Qux {
 static myQux() {}
}
```

类构造函数

constructor 关键字用于在类定义块内部创建类的构造函数
constructor 会告诉解释器使用new操作符创建类的新实例时调用这个函数
构造函数定义非必需，不定义则自动定义为空函数

实例化

new操作符在创建类的实例时干了什么？(与调用构造函数一样)

1. 内存中创建一个新对象
2. 新对象内部[[prototype]]指向构造函数的prototype
3. 构造函数内部this指向新对象
4. 执行构造函数（给新对象添加属性）
5. 如果构造函数返回非空对象，则返回该对象；否则返回新对象

new后面调用构造函数可以传参，不传参可以不写括号

```
class Person {
    constructor(name) {
        this.name = name
    }
}

let person1 = new Person('iiTzYaox')
let person2 = new Person()
let person3 = new Person
console.log(person1.name)    // iiTzYaox
```

```
console.log(person2.name)    // undefined
console.log(person3.name)    // undefined
```

如果类构造函数没有返回新对象而是别的对象，这个对象就不能通过instanceof操作符检测与类的关联。因为原型没有修改

```
class Person {
    constructor(override) {
        this.foo = 'foo';
        if (override) {
            return {
                bar: 'bar'
            };
        }
    }
}
let p1 = new Person(),
    p2 = new Person(true);
console.log(p1); // Person{ foo: 'foo' }
console.log(p1 instanceof Person); // true
console.log(p2); // { bar: 'bar' }
console.log(p2 instanceof Person); // false
```

类构造函数必须用new调用

把类当成特殊函数

ES类是一种特殊的函数

- 能用typeof操作符检测
- 有prototype属性，且constructor指向自身
- 可以用instanceof 构造函数原型是否存在 实例的原型链上
- 函数是js一等公民，所以类也是。
- 函数能立即调用，类也能立即实例化

```
// 因为是一个类表达式，所以类名是可选的
let p = new class Foo {
    constructor(x) {
        console.log(x);
    }
}('bar'); // bar
console.log(p); // Foo {}
```

实例、原型和类成员

实例成员

其实就是new操作符调用构造函数是会执行构造函数然后给新对象添加属性。
实例成员就是构造函数内部的属性 `this.属性名`
也可以直接写在类的最顶层
因为是实例的成员，所以不会访问到原型，所以不存在引用值问题

```
class Person {
    constructor() {
        this.friend = ['a', 'b']
    }
}
let person1 = new Person()
let person2 = new Person()
person1.friend.push('c')
console.log(person2.friend) // [ 'a', 'b' ]

class foo {
  bar = 'hello';
  baz = 'world';

  constructor() {
    // ...
  }
}
```

原型方法和访问器

- 类块中的方法就是原型方法
- 类块只能添加方法
- 访问器也是方法，所以可以用访问器来设置对象/原始值属性
- 可以用字符串，符号，计算属性值作为键

在实例间共享方法，类定义语法把在类块中定义的方法作为原型方法。

```
class Person {
    constructor() {
        this.friend = ['a', 'b']
        this.fn1 = function () { }
    }
    fn2() { }
}
let person1 = new Person()
let person2 = new Person()
person1.friend.push('c')
console.log(person2.friend)    // [ 'a', 'b' ]
console.log(person1.fn1==person2.fn1)    // false
// 原型方法
console.log(person1.fn2==person2.fn2)    // true
```

类块中只能给原型添加方法，不能直接添加对象/原始值属性；但是可以通过获取和设置访问器添加

```
class Person {
 name: 'Jake'
}
// Uncaught SyntaxError: Unexpected token
```

类方法等同于对象属性，可以使用字符串、符号或计算的值作为键

```
const symbolKey = Symbol('symbolKey');
class Person {
 stringKey() {
 console.log('invoked stringKey');
 }
 [symbolKey]() {
 console.log('invoked symbolKey');
 }
 ['computed' + 'Key']() {
 console.log('invoked computedKey');
 }
}
let p = new Person();
p.stringKey(); // invoked stringKey
p[symbolKey](); // invoked symbolKey
p.computedKey(); // invoked computedKey
```

静态类方法

类自己的方法，不会被继承
一般用来做实例工厂：

```
class Person {
 constructor(age) {
 this.age_ = age;
 }
 sayAge() {
 console.log(this.age_);
 }
 static create() {
 // 使用随机年龄创建并返回一个 Person 实例
 return new Person(Math.floor(Math.random()*100));
 }
}
console.log(Person.create()); // Person { age_: ... }
```

非函数原型和类成员

可以在类外面添加数据

```
class Person {
 sayName() {
 console.log(`${Person.greeting} ${this.name}`);
 }
}
// 在类上定义数据成员
Person.greeting = 'My name is';
// 在原型上定义数据成员
Person.prototype.name = 'Jake';
let p = new Person();
p.sayName(); // My name is Jake
```

如果是给类自己定义成员，也可以直接用静态属性

```
class Person {
    sayName() {
        console.log(`${Person.greeting} ${this.name}`);
    }
    // 静态属性添加类成员
    static greeting = 'My name is'
}
// 在类上定义数据成员
// Person.greeting = 'My name is'; 等效
```

迭代器与生成器方法

类定义语法支持在原型和类本身上定义生成器方法

```
class Person {
 // 在原型上定义生成器方法
 *createNicknameIterator() {
 yield 'Jack';
 yield 'Jake';
 yield 'J-Dog';
 }
 // 在类上定义生成器方法
 static *createJobIterator() {
 yield 'Butcher';
 yield 'Baker';
 yield 'Candlestick maker';
 }
}
let jobIter = Person.createJobIterator();
console.log(jobIter.next().value); // Butcher
console.log(jobIter.next().value); // Baker
console.log(jobIter.next().value); // Candlestick maker
let p = new Person();
let nicknameIter = p.createNicknameIterator();
console.log(nicknameIter.next().value); // Jack
console.log(nicknameIter.next().value); // Jake
console.log(nicknameIter.next().value); // J-Dog
```

通过添加默认迭代器，把类实例变成可迭代对象

```
class Person {
 constructor() {
 this.nicknames = ['Jack', 'Jake', 'J-Dog'];
 }
 *[Symbol.iterator]() {
 yield *this.nicknames.entries();
 }
}
let p = new Person();
for (let [idx, nickname] of p) {
 console.log(nickname);
}
```

也可以返回迭代器实例

```
class Person {
 constructor() {
 this.nicknames = ['Jack', 'Jake', 'J-Dog'];
 }
 [Symbol.iterator]() {
 return this.nicknames.entries();
 }
}
let p = new Person();
for (let [idx, nickname] of p) {
 console.log(nickname);
}
// Jack
// Jake
// J-Dog
```

继承

ES6新语法，背后依然是原型链

继承基础

ES6类支持单继承，使用extends关键字，就可以继承任何拥有[[Construct]]和原型的对象

构造函数，HomeObject和super()

派生类可以用super引用原型

- 类构造函数中使用 super 可以调用父类构造函数
- 在静态方法中可以通过 super 调用继承的类上定义的静态方法
- super只能在派生类狗在函数和静态方法中使用
- 不饿能单独引用super关键词，要么调用构造函数，要么调用静态方法
- 派生类没定义构造函数则自动调用super()
- 派生类构造函数中，不能再调用super()之前引用this
- 派生类构造函数中要么调用super要么返回一个对象

```
class Vehicle {
    constructor() {
        this.hasEngine = true;
    }
    static identify() {
        console.log('vehicle');
    }
}
class Bus extends Vehicle {
    constructor() {
        // 不要在调用 super()之前引用 this，否则会抛出 ReferenceError
        super(); // 相当于 super.constructor()
        console.log(this instanceof Vehicle); // true
        console.log(this); // Bus { hasEngine: true }
    }
```

```
    static identify() {
        super.identify();
    }
}
new Bus();
```

抽象基类

可以供其他类继承，但本身不会被实例化
使用new.target实现
new.target检测函数或构造方法是否是通过new运算符被调用的。
new调用类构造函数，new.target指向构造函数

```
// 抽象基类
class Vehicle {
    constructor() {
        console.log(new.target);
        if (new.target === Vehicle) {
            throw new Error('Vehicle cannot be directly instantiated');
        }
    }
}
// 派生类
class Bus extends Vehicle { }
new Bus(); // class Bus {}
new Vehicle(); // class Vehicle {}
   // Error: Vehicle cannot be directly instantiated
```

继承内置类型

ES6 类为继承内置引用类型提供了顺畅的机制
有些内置类型的方法会返回新实例，默认情况下，实例的类型与原始实例一致。若要覆盖这个默认行为则可以覆盖 Symbol.species 访问器

```
class SuperArray extends Array { }
let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x % 2))
console.log(a1); // [1, 2, 3, 4, 5]
console.log(a2); // [1, 3, 5]
console.log(a1 instanceof SuperArray); // true
console.log(a2 instanceof SuperArray); // true

// 覆盖Symbol.species 访问器
class SuperArray extends Array {
    static get [Symbol.species]() {
        return Array;
    }
}
let a1 = new SuperArray(1, 2, 3, 4, 5);
let a2 = a1.filter(x => !!(x % 2))
console.log(a1); // [1, 2, 3, 4, 5]
console.log(a2); // [1, 3, 5]
console.log(a1 instanceof SuperArray); // true
console.log(a2 instanceof SuperArray); // false
```

类混入

ES6没有显式的多类继承，但是可以模拟实现

```
class Vehicle { }
let FooMixin = (Superclass) => class extends Superclass {
    foo() {
        console.log('foo');
    }
};
let BarMixin = (Superclass) => class extends Superclass {
    bar() {
        console.log('bar');
    }
};
let BazMixin = (Superclass) => class extends Superclass {
    baz() {
        console.log('baz');
    }
};
function mix(BaseClass, ...Mixins) {
    return Mixins.reduce((accumulator, current) => current(accumulator), BaseClass);
}
class Bus extends mix(Vehicle, FooMixin, BarMixin, BazMixin) { }
let b = new Bus();
b.foo(); // foo
b.bar(); // bar
b.baz(); // baz
```

总结

下面的模式适用于创建对象。

- 工厂模式就是一个简单的函数，这个函数可以创建对象，为它添加属性和方法，然后返回这个对象。这个模式在构造函数模式出现后就很少用了。
- 使用构造函数模式可以自定义引用类型，可以使用 new 关键字像创建内置类型实例一样创建自定义类型的实例。不过，构造函数模式也有不足，主要是其成员无法重用，包括函数。考虑到函数本身是松散的、弱类型的，没有理由让函数不能在多个对象实例间共享。
- 原型模式解决了成员共享的问题，只要是添加到构造函数 prototype 上的属性和方法就可以共享。而组合构造函数和原型模式通过构造函数定义实例属性，通过原型定义共享的属性和方法。

JavaScript 的继承主要通过原型链来实现。原型链涉及把构造函数的原型赋值为另一个类型的实例。这样一来，子类就可以访问父类的所有属性和方法，就像基于类的继承那样。原型链的问题是所有继承的属性和方法都会在对象实例间共享，无法做到实例私有。盗用构造函数模式通过在子类构造函数中调用父类构造函数，可以避免这个问题。这样可以让每个实例继承的属性都是私有的，但要求类型只能通过构造函数模式来定义（因为子类不能访问父类原型上的方法）。目前最流行的继承模式是组合继承，即通过原型链继承共享的属性和方法，通过盗用构造函数继承实例属性。除上述模式之外，还有以下几种继承模式。

- 原型式继承可以无须明确定义构造函数而实现继承，本质上是对给定对象执行浅复制。这种操作的结果之后还可以再进一步增强。

- 与原型式继承紧密相关的是寄生式继承，即先基于一个对象创建一个新对象，然后再增强这个新对象，最后返回新对象。这个模式也被用在组合继承中，用于避免重复调用父类构造函数导致的浪费。
- 寄生组合继承被认为是实现基于类型继承的最有效方式。

ECMAScript 6新增的类很大程度上是基于既有原型机制的语法糖。类的语法让开发者可以优雅地定义向后兼容的类，既可以继承内置类型，也可以继承自定义类型。类有效地跨越了对象实例、对象原型和对象类之间的鸿沟。