对象、类与面向对象编程

ECMA-262将对象定义为一组属性的无序集合
可以把对象想象成一张散列表，内容是key/value对，value可以是数据，可以是函数

理解对象

早期用 new Object()创建实例
现在流行 对象字面量

```
let person = {
 name: "Nicholas",
 age: 29,
 job: "Software Engineer",
 sayName() {
 console.log(this.name);
 }
};
```

属性的类型

ECMA-262使用内部特性来描述属性的特征。这些特性是由为JavaScript实现引擎的规范定义的。所以开发者不能在js中直接访问这些特性
两个中括号把特性的名称括起来，标识为内部特性。如[[Enumerable]]
属性分为：数据属性和访问器属性

数据属性

数据属性包含保存一个数据值的位置。值从这个位置读取/写入。
数据属性有4个

- [[Configurable]]：表示属性是否可以通过 delete 删除并重新定义，是否可以修改它的特性，以及是否可以把它改为访问器属性。默认情况下，所有直接定义在对象上的属性的这个特性都是 true，如前面的例子所示
- [[Enumerable]]：表示属性是否可以通过 for-in 循环返回。默认情况下，所有直接定义在对象上的属性的这个特性都是 true，如前面的例子所示。
- [[Writable]]：表示属性的值是否可以被修改。默认情况下，所有直接定义在对象上的属性的这个特性都是 true，如前面的例子所示。
- [[Value]]：包含属性实际的值。这就是前面提到的那个读取和写入属性值的位置。这个特性的默认值为 undefined。

属性是否可配置(删属性，修改配置)
属性是否可遍历
值 是否可修改
实际值

一旦一个属性配置成 不可配置后，只能修改writable属性，别的都不能改

访问器属性

包含一个获取(getter)函数和一个设置(setter)函数
非必须
读取访问器属性，调用获取函数 getter
写入访问器属性，调用设置函数 setter

访问器属性有4个:

- [[Configurable]]：表示属性是否可以通过 delete 删除并重新定义，是否可以修改它的特性，以及是否可以把它改为数据属性。默认情况下，所有直接定义在对象上的属性的这个特性都是 true。
- [[Enumerable]]：表示属性是否可以通过 for-in 循环返回。默认情况下，所有直接定义在对象上的属性的这个特性都是 true。
- [[Get]]：获取函数，在读取属性时调用。默认值为 undefined。
- [[Set]]：设置函数，在写入属性时调用。默认值为 undefined。

属性是否可配置
属性是否可遍历
获取函数 undefined
设置函数 undefined

访问器属性必须由Object.defineProperty()定义

如果属性是显式声明，特性默认值都是true,通过DefineProperty定义，特性默认全是false

定义多个属性

Object.defineProperties()
多个描述符，一次性定义多个属性

```
Object.defineProperty(book, "year", {
 get() {
 return this.year_;
 },
 set(newValue) {
 if (newValue > 2017) {
 this.year_ = newValue;
 this.edition += newValue - 2017;
 }
 }
});

let book = {};
Object.defineProperties(book, {
 year_: {
 value: 2017
 },
 edition: {
 value: 1
 },
 year: {
```

```
  get() {
  return this.year_;
  },
   set(newValue) {
  if (newValue > 2017) {
  this.year_ = newValue;
  this.edition += newValue - 2017;
  }
  }
  }
});
```

读取属性的特性

Object.getOwnPropertyDescriptor()方法可以取得指定属性的属性描述符
ECMAScript 2017 （ES8）新增了 Object.getOwnPropertyDescriptors()静态方法

JS 1.js

```
let testObj = {
    name: 'TESTOBJ'
}

Object.defineProperty(testObj, 'QAQ',
    {
        get() { return 10 },
        set(value) {
            if (value > 10) {
                console.log(value)
            }
        },
        configurable: true
    })

console.log(Object.getOwnPropertyDescriptor(testObj, 'QAQ'))
console.log(Object.getOwnPropertyDescriptors(testObj))
```

问题 输出 调试控制台 终端

```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

尝试新的跨平台 PowerShell https://aka.ms/pscore6

PS C:\Users\Aleen\Desktop> node .\1.js
{
  get: [Function: get],
  set: [Function: set],
  enumerable: false,
  configurable: true
}
{
  name: {
    value: 'TESTOBJ',
    writable: true,
    enumerable: true,
    configurable: true
  },
  QAQ: {
    get: [Function: get],
    set: [Function: set],
    enumerable: false,
    configurable: true
  }
}
```

合并对象

merge 合并
mixin 混入 （读 mix in）

ES6合并对象，
Object.assign(toObj,fromObj)
把fromObj中 可枚举，自有 的属性复制到toObj 浅拷贝，返回toObj
相同属性，后者优先

```
let dest, src, result;
/**
 * 简单复制
 */
dest = {};
src = { id: 'src' };
result = Object.assign(dest, src);
// Object.assign 修改目标对象
// 也会返回修改后的目标对象
console.log(dest === result); // true
console.log(dest !== src); // true
console.log(result); // { id: src }
console.log(dest); // { id: src }
/**
 * 多个源对象
 */
dest = {};
result = Object.assign(dest, { a: 'foo' }, { b: 'bar' });
console.log(result); // { a: foo, b: bar }
/**
 * 获取函数与设置函数
 */
dest = {
 set a(val) {
 console.log(`Invoked dest setter with param ${val}`);
 }
};
src = {
 get a() {
 console.log('Invoked src getter');
 return 'foo';
 }
};
Object.assign(dest, src);
// 调用 src 的获取方法
// 调用 dest 的设置方法并传入参数"foo"
// 因为这里的设置函数不执行赋值操作
// 所以实际上并没有把值转移过来
console.log(dest); // { set a(val) {...} }
```

对象标示以及相等判断

Object.is()

```
console.log(Object.is(true, 1)); // false
console.log(Object.is({}, {})); // false
console.log(Object.is("2", 2)); // false
```

```
// 正确的 0、-0、+0 相等/不等判定
console.log(Object.is(+0, -0)); // false
console.log(Object.is(+0, 0)); // true
console.log(Object.is(-0, 0)); // false
// 正确的 NaN 相等判定
console.log(Object.is(NaN, NaN)); // true
```

增强的对象语法

ES6为定义和操作对象新增很多语法糖

- 属性值简写
  会自动查找同名变量，如果没有同名变量则报错
  代码压缩程序会在不同作用域间保留属性名，以防止找不到引用

```
let name = 'Matt';
let person = {
 name: name
};
console.log(person); // { name: 'Matt' }

let name = 'Matt';
let person = {
 name
};
console.log(person); // { name: 'Matt' }

// 代码压缩也会保留name标识符
function makePerson(name) {
 return {
 name
 };
}
let person = makePerson('Matt');
console.log(person.name); // Matt

function makePerson(a) {
 return {
 name: a
 };
}
var person = makePerson("Matt");
console.log(person.name); // Matt
```

- 可计算属性
  可以直接在对象字面量中实现动态赋值

```
// 之前
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';
let person = {};
person[nameKey] = 'Matt';
person[ageKey] = 27;
person[jobKey] = 'Software engineer';
```

```
console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }

// 之后
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';
let person = {
 [nameKey]: 'Matt',
 [ageKey]: 27,
 [jobKey]: 'Software engineer'
};
console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }

// 可以更加复杂
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';
let uniqueToken = 0;
function getUniqueKey(key) {
 return `${key}_${uniqueToken++}`;
}
let person = {
 [getUniqueKey(nameKey)]: 'Matt',
 [getUniqueKey(ageKey)]: 27,
 [getUniqueKey(jobKey)]: 'Software engineer'
};
console.log(person); // { name_0: 'Matt', age_1: 27, job_2: 'Software engineer' }
```

- 简写方法名（很常用，vue methods）

```
// 之前
let person = {
 sayName: function(name) {
 console.log(`My name is ${name}`);
 }
};
person.sayName('Matt'); // My name is Matt

// 之后
let person = {
 sayName(name) {
 console.log(`My name is ${name}`);
 }
};
person.sayName('Matt'); // My name is Matt
person.name = 'Matt';
person.sayName(); // My name is Matt

// 简写可以与可计算属性互相兼容
const methodKey = 'sayName';
let person = {
 [methodKey](name) {
 console.log(`My name is ${name}`);
 }
}
person.sayName('Matt'); // My name is Matt
```

对象结构

ES6新增了对象结构语法

```
// 不使用对象解构
let person = {
 name: 'Matt',
 age: 27
};
let personName = person.name,
 personAge = person.age;
console.log(personName); // Matt
console.log(personAge); // 27

// 使用对象解构
let person = {
 name: 'Matt',
 age: 27
};
let { name: personName, age: personAge } = person;
console.log(personName); // Matt
console.log(personAge); // 27

// 可以设置新的属性，可以设置默认值
let person = {
 name: 'Matt',
 age: 27
};
let { name, job='Software engineer' } = person;
console.log(name); // Matt
console.log(job); // Software engineer
```

解构在内部使用ToObject(),会把原始值先转换成对象，然后解构
null和undefined不能解构

```
let { length } = 'foobar';
console.log(length); // 6
let { constructor: c } = 4;
console.log(c === Number); // true
let { _ } = null; // TypeError
let { _ } = undefined; // TypeError
```

如果是事先声明的变量需要包含在一对括号内

```
let personName, personAge;
let person = {
 name: 'Matt',
 age: 27
};
({name: personName, age: personAge} = person);
console.log(personName, personAge); // Matt, 27
```

1. 嵌套解构

```
let person = {
 name: 'Matt',
 age: 27,
 job: {
 title: 'Software engineer'
 }
};
let personCopy = {};
({
 name: personCopy.name,
 age: personCopy.age,
 job: personCopy.job
} = person);
// 因为一个对象的引用被赋值给 personCopy，所以修改
// person.job 对象的属性也会影响 personCopy
person.job.title = 'Hacker'
console.log(person);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }
console.log(personCopy);
// { name: 'Matt', age: 27, job: { title: 'Hacker' } }

// 嵌套解构
let person = {
 name: 'Matt',
 age: 27,
 job: {
 title: 'Software engineer'
 }
};
// 声明 title 变量并将 person.job.title 的值赋给它
let { job: { title } } = person;
console.log(title); // Software engineer
```

2. 部分解构

解构到一半报错，前面的照样成功

```
let person = {
 name: 'Matt',
 age: 27
};
let personName, personBar, personAge;
try {
 // person.foo 是 undefined，因此会抛出错误
 ({name: personName, foo: { bar: personBar }, age: personAge} = person);
} catch(e) {}
console.log(personName, personBar, personAge);
// Matt, undefined, undefined
```

3. 参数上下文匹配

函数参数列表也可以解构，不与影响arguments对象

```
let person = {
 name: 'Matt',
 age: 27
};
```

```
function printPerson(foo, {name, age}, bar) {
 console.log(arguments);
 console.log(name, age);
}
function printPerson2(foo, {name: personName, age: personAge}, bar) {
 console.log(arguments);
 console.log(personName, personAge);
}
printPerson('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27
printPerson2('1st', person, '2nd');
// ['1st', { name: 'Matt', age: 27 }, '2nd']
// 'Matt', 27
```