Class基本属性：

类的内部所有定义的方法，都是不可枚举的。

方法之间不需要逗号分隔，加了会报错。

类的内部定义的方法都在原型对象上面。

constructor方法是类的默认方法，通过new命令生成对象实例时，自动调用该方法。一个类必须有constructor方法，如果没有显式定义，一个空的constructor方法会被默认添加。

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

类必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不用new也可以执行

阮一峰简介class表达式的那里。

类和模块的内部，默认就是严格模式，所以不需要使用use strict指定运行模式。只要你的代码写在类或模块之中，就只有严格模式可用。考虑到未来所有的代码，其实都是运行在模块之中，所以 ES6 实际上把整个语言升级到了严格模式。

类不存在变量提升（hoist），这一点与 ES5 完全不同。上面代码中，Foo类使用在前，定义在后，这样会报错，因为 ES6 不会把类的声明提升到代码头部。这种规定的原因与下文要提到的继承有关，必须保证子类在父类之后定义。

阮一峰简介注意点this的指向的那里。

类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。注意，如果静态方法包含this关键字，这个this指的是类，而不是实例。父类的静态方法，可以被子类继承。

实例属性除了定义在constructor()方法里面的this上面，也可以定义在类的最顶层。

阮一峰静态属性的那里。

Class的继承：

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，子类的构造函数必须执行一次super函数。注意，super虽然代表了父类A的构造函数，但是返回的是子类B的实例，即super内部的this指的是B的实例，因此super()在这里相当于A.prototype.constructor.call(this)。上面代码中，new.target指向当前正在执行的函数。可以看到，在super()执行时，它指向的是子类B的构造函数，而不是父类A的构造函数。也就是说，super()内部的this指向的是B。

第二种情况，super作为对象时，在普通方法中，指向父类的原型对象；在静态方法中，指向父类。

ES6 规定，在子类普通方法中通过super调用父类的方法时，方法内部的this指向当前的子类实例

如果super作为对象，用在静态方法之中，这时super将指向父类，而不是父类的原型对象。另外，在子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，而不是子类的实例。

（1）子类的\_\_proto\_\_属性，表示构造函数的继承，总是指向父类。

1. 子类prototype属性的\_\_proto\_\_属性，表示方法的继承，总是指向父类的prototype属性。

上面代码的A，只要是一个有prototype属性的函数，就能被B继承。由于函数都有prototype属性（除了Function.prototype函数），因此A可以是任意函数。

Let、const

for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

暂时性死区：只要块级作用域内存在let命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

不允许重复声明：let不允许在相同作用域内，重复声明同一个变量。

为什么需要块级作用域：第一种场景，内层变量可能会覆盖外层变量。第二种场景，用来计数的循环变量泄露为全局变量。

块级作用域与函数声明。能不能在块级作用域内声明函数。

const声明的变量不得改变值，这意味着，const一旦声明变量，就必须立即初始化，不能留到以后赋值。声明不赋值就会报错。

const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，const只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心。

ES6 为了改变这一点，一方面规定，为了保持兼容性，var命令和function命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于顶层对象的属性。也就是说，从 ES6 开始，全局变量将逐步与顶层对象的属性脱钩。

解构赋值

数组解构赋值实际上是内部调用的iterator接口，对象不具有iterator接口但是也可以解构赋值？

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

如果变量名与属性名不一致，必须写成下面这样。

let { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz // "aaa"

let obj = { first: 'hello', last: 'world' };

let { first: f, last: l } = obj;

f // 'hello'

l // 'world'

这实际上说明，对象的解构赋值是下面形式的简写（参见《对象的扩展》一章）。

let { foo: foo, bar: bar } = { foo: 'aaa', bar: 'bbb' };

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者。

let { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz // "aaa"

foo // error: foo is not defined

上面代码中，foo是匹配的模式，baz才是变量。真正被赋值的是变量baz，而不是模式foo。

函数参数也可以进行解构赋值。

解构赋值用途：交换变量的值；从函数返回多个值；函数参数的定义；遍历map结构；

Symbol原始数据

注意，Symbol函数前不能使用new命令，否则会报错。这是因为生成的 Symbol 是一个原始类型的值，不是对象。也就是说，由于 Symbol 值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

Symbol 值作为属性名时，该属性还是公开属性，不是私有属性。

Symbol 作为属性名，遍历对象的时候，该属性不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回。但是，它也不是私有属性，有一个Object.getOwnPropertySymbols()方法，可以获取指定对象的所有 Symbol 属性名。该方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。

有时，我们希望重新使用同一个 Symbol 值，Symbol.for()方法可以做到这一点。它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建一个以该字符串为名称的 Symbol 值，并将其注册到全局。

注意，Symbol.for()为 Symbol 值登记的名字，是全局环境的，不管有没有在全局环境运行。

Set和map数据结构

上面代码也展示了一种去除数组重复成员的方法。

// 去除数组的重复成员

[...new Set(array)]

上面的方法也可以用于，去除字符串里面的重复字符。

[...new Set('ababbc')].join('')

向 Set 加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值。Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于精确相等运算符（===），主要的区别是向 Set 加入值时认为NaN等于自身，而精确相等运算符认为NaN不等于自身。

Set.prototype.add(value)：添加某个值，返回 Set 结构本身。

Set.prototype.delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

Set.prototype.has(value)：返回一个布尔值，表示该值是否为Set的成员。

Set.prototype.clear()：清除所有成员，没有返回值。

Set.prototype.keys()：返回键名的遍历器

Set.prototype.values()：返回键值的遍历器

Set.prototype.entries()：返回键值对的遍历器

Set.prototype.forEach()：使用回调函数遍历每个成员

需要特别指出的是，Set的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用 Set 保存一个回调函数列表，调用时就能保证按照添加顺序调用。

entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。

Set 结构的实例默认可遍历，它的默认遍历器生成函数就是它的values方法。

使用 Set 可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。

作为构造函数，Map 也可以接受一个数组作为参数。该数组的成员是一个个表示键值对的数组。

const map = new Map([

['name', '张三'],

['title', 'Author']

]);

map.size // 2

map.has('name') // true

map.get('name') // "张三"

map.has('title') // true

map.get('title') // "Author"

上面代码在新建 Map 实例时，就指定了两个键name和title。

事实上，不仅仅是数组，任何具有 Iterator 接口、且每个成员都是一个双元素的数组的数据结构（详见《Iterator》一章）都可以当作Map构造函数的参数。这就是说，Set和Map都可以用来生成新的 Map。

const set = new Set([

['foo', 1],

['bar', 2]

]);

const m1 = new Map(set);

m1.get('foo') // 1

const m2 = new Map([['baz', 3]]);

const m3 = new Map(m2);

m3.get('baz') // 3

Map结构的实例的属性和操作方法：size，get，set，has，delete，clear。

Map.prototype.keys()：返回键名的遍历器。

Map.prototype.values()：返回键值的遍历器。

Map.prototype.entries()：返回所有成员的遍历器。

Map.prototype.forEach()：遍历 Map 的所有成员。

需要特别注意的是，Map 的遍历顺序就是插入顺序。

Map 结构的默认遍历器接口（Symbol.iterator属性），就是entries方法。

Iterator接口

ES6 规定，默认的 Iterator 接口部署在数据结构的Symbol.iterator属性，或者说，一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历的”（iterable）。Symbol.iterator属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器对象。该对象的根本特征就是具有next方法。每次调用next方法，都会返回一个代表当前成员的信息对象，具有value和done两个属性。

对于原生部署 Iterator 接口的数据结构，不用自己写遍历器生成函数，for...of循环会自动遍历它们。除此之外，其他数据结构（主要是对象）的 Iterator 接口，都需要自己在Symbol.iterator属性上面部署，这样才会被for...of循环遍历。

调用iterator接口的场合：解构赋值、扩展运算符、yeild\*、for.....of、Array.from()。

字符串、数组、类数组、set、map原生具有iterator接口。

遍历器对象除了具有next方法，还可以具有return方法和throw方法。如果你自己写遍历器对象生成函数，那么next方法是必须部署的，return方法和throw方法是否部署是可选的。

有些数据结构是在现有数据结构的基础上，计算生成的。比如，ES6 的数组、Set、Map 都部署了以下三个方法，调用后都返回遍历器对象。entries() 返回一个遍历器对象，用来遍历[键名, 键值]组成的数组。对于数组，键名就是索引值；对于 Set，键名与键值相同。Map 结构的 Iterator 接口，默认就是调用entries方法。keys() 返回一个遍历器对象，用来遍历所有的键名。values() 返回一个遍历器对象，用来遍历所有的键值。这三个方法调用后生成的遍历器对象，所遍历的都是计算生成的数据结构。

forEach这种写法的问题在于，无法中途跳出forEach循环，break命令或return命令都不能奏效。

Generator函数

调用 Generator 函数，返回一个遍历器对象，代表 Generator 函数的内部指针。以后，每次调用遍历器对象的next方法，就会返回一个有着value和done两个属性的对象。value属性表示当前的内部状态的值，是yield表达式后面那个表达式的值；done属性是一个布尔值，表示是否遍历结束。最后一个是return的话，value值就是return后面的值。

需要注意的是，yield表达式后面的表达式，只有当调用next方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

另外，yield表达式如果用在另一个表达式之中，必须放在圆括号里面。

yield表达式用作函数参数或放在赋值表达式的右边，可以不加括号。

Generator 函数执行后，返回一个遍历器对象。该对象本身也具有Symbol.iterator属性，执行后返回自身。它的Symbol.iterator属性，也是一个遍历器对象生成函数，执行后返回它自己。

上面代码使用for...of循环，依次显示 5 个yield表达式的值。这里需要注意，一旦next方法的返回对象的done属性为true，for...of循环就会中止，且不包含该返回对象，所以上面代码的return语句返回的6，不包括在for...of循环之中。

Generator 函数返回的遍历器对象，都有一个throw方法，可以在函数体外抛出错误，然后在 Generator 函数体内捕获。如果 Generator 函数内部没有部署try...catch代码块，那么throw方法抛出的错误，将被外部try...catch代码块捕获。throw方法被捕获以后，会附带执行下一条yield表达式。也就是说，会附带执行一次next方法。

Generator 函数返回的遍历器对象，还有一个return方法，可以返回给定的值，并且终结遍历 Generator 函数。如果return方法调用时，不提供参数，则返回值的value属性为undefined。如果 Generator 函数内部有try...finally代码块，且正在执行try代码块，那么return方法会导致立刻进入finally代码块，执行完以后，整个函数才会结束。

Yield\*后面跟的是遍历器对象。实际上，任何数据结构只要有 Iterator 接口，就可以被yield\*遍历。

如果被代理的 Generator 函数有return语句，那么yield\*表达式的返回值就是被代理的generator函数的return语句的返回值。

Generator 函数总是返回一个遍历器，ES6 规定这个遍历器是 Generator 函数的实例，也继承了 Generator 函数的prototype对象上的方法。

Generator状态机：上面的 Generator 实现与 ES5 实现对比，可以看到少了用来保存状态的外部变量ticking，这样就更简洁，更安全（状态不会被非法篡改）、更符合函数式编程的思想，在写法上也更优雅。Generator 之所以可以不用外部变量保存状态，是因为它本身就包含了一个状态信息，即目前是否处于暂停态。

Async函数

async函数对 Generator 函数的改进，体现在以下四点。

（1）内置执行器。

Generator 函数的执行必须靠执行器，所以才有了co模块，而async函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，只要一行。

asyncReadFile();

上面的代码调用了asyncReadFile函数，然后它就会自动执行，输出最后结果。这完全不像 Generator 函数，需要调用next方法，或者用co模块，才能真正执行，得到最后结果。

（2）更好的语义。

async和await，比起星号和yield，语义更清楚了。async表示函数里有异步操作，await表示紧跟在后面的表达式需要等待结果。

（3）更广的适用性。

co模块约定，yield命令后面只能是 Thunk 函数或 Promise 对象，而async函数的await命令后面，可以是 Promise 对象和原始类型的值（数值、字符串和布尔值，但这时会自动转成立即 resolved 的 Promise 对象）。

（4）返回值是 Promise。

async函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方法指定下一步的操作。进一步说，async函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而await命令就是内部then命令的语法糖。

async函数返回一个 Promise 对象，可以使用then方法添加回调函数。当函数执行的时候，一旦遇到await就会先返回，等到异步操作完成，再接着执行函数体内后面的语句。

async函数内部return语句返回的值，会成为then方法回调函数的参数。async函数内部抛出错误，会导致返回的 Promise 对象变为reject状态。抛出的错误对象会被catch方法回调函数接收到。

async函数返回的 Promise 对象，必须等到内部所有await命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数。

正常情况下，await命令后面是一个 Promise 对象，返回该对象的结果。如果不是 Promise 对象，就直接返回对应的值。另一种情况是，await命令后面是一个thenable对象（即定义then方法的对象），那么await会将其等同于 Promise 对象。await命令后面的 Promise 对象如果变为reject状态，则reject的参数会被catch方法的回调函数接收到。注意，上面代码中，await语句前面没有return，但是reject方法的参数依然传入了catch方法的回调函数。这里如果在await前面加上return，效果是一样的。任何一个await语句后面的 Promise 对象变为reject状态，那么整个async函数都会中断执行。

多个await命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

let foo = await getFoo();

let bar = await getBar();

上面代码中，getFoo和getBar是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有getFoo完成以后，才会执行getBar，完全可以让它们同时触发。

let [foo, bar] = await Promise.all([getFoo(), getBar()]);

Promise

const p1 = new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('fail')), 3000)})

const p2 = new Promise(function (resolve, reject) {

setTimeout(() => resolve(p1), 1000)})

p2

.then(result => console.log(result))

.catch(error => console.log(error))

// Error: fail

上面代码中，p1是一个 Promise，3 秒之后变为rejected。p2的状态在 1 秒之后改变，resolve方法返回的是p1。由于p2返回的是另一个 Promise，导致p2自己的状态无效了，由p1的状态决定p2的状态。所以，后面的then语句都变成针对后者（p1）。又过了 2 秒，p1变为rejected，导致触发catch方法指定的回调函数。

-------------------------------------------------------------------------------

new Promise((resolve, reject) => {

resolve(1);

console.log(2);}).then(r => {

console.log(r);});

// 2// 1

上面代码中，调用resolve(1)以后，后面的console.log(2)还是会执行，并且会首先打印出来。这是因为立即 resolved 的 Promise 是在本轮事件循环的末尾执行，总是晚于本轮循环的同步任务。

一般来说，调用resolve或reject以后，Promise 的使命就完成了，后继操作应该放到then方法里面，而不应该直接写在resolve或reject的后面。所以，最好在它们前面加上return语句，这样就不会有意外。

new Promise((resolve, reject) => {

return resolve(1);

// 后面的语句不会执行 console.log(2);})