Base

let 命令

let命令所声明的对象只在let命令所在的代码块生效

```
class LetTest {
  //输出let和var 的区别
  consoleValLet(number) {
    var a = [],
      b = [];
    for (var i = 0; i < 10; i++) {
      a[i] = function () {
        console.log(i);
      };
    }
    for (let j = 0; j < 10; j++) {
      b[j] = () => {
        console.log(j);
      }
    }
    a[number]();
    b[number]();
  }
}
var letTest = new LetTest();
```

字符串拓展

```
class StringES6 {
  constructor(value) {
    this.str = value;
  }

  // 鉴别一个字符由2个字节还是4个字节组成的最简单方法
  is32Bit(c) {
    return c.codePointAt(0) > 0xFFFF;
  };

  // 字符串遍历
  forEachStr(str) {
    for (let codePoint of str) {
      console.log(codePoint);
    }
  }
  testNewWay(include, startsWith, endsWith) {
```

```
    // 字符串新方法 includes,startsWith,endsWith
    // 支持第二个参数，表示搜索开始位置。eg 3
    testStr.startsWith(startsWith);
    testStr.endsWith(endsWith);
    testStr.includes(include, 0);
  }
  repeatStr() {
    // repeat 重复字符串指定次数
    // 参数如果是小数会取整，如果是负数或者Infinity 则会报错
    // NAN 等同于 0 ，字符串则会转成数字
    this.str.repeat(3);
  }
  padString(value) {
    // 字符串补全长度， padStart,padEnd
    // 2个参数，第一个参数用来指定字符串的最小长度，第二个参数是用来补全的字符串。
    testStr.padStart(value, 10); // 由开头补全
    testStr.padEnd(value, 4); // 由末尾开始补全
  }
```

模版字符串 `

```
  // 模板字符串 ` 字符
  // 当做普通字符串使用
  // 如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。
  templateStringTest() {
    let strPlus = ``;
    // 单行字符串
    strPlus = `this is a plusStr`;
    // 多行字符串
    strPlus += `add a
multiline str
in it
`;
    // 字符串中嵌入变量
    let str1 = '变量1';
    let str2 = '变量2';
    strPlus += `add the
one params ${str1} , two params ${str2}`;
    console.log(strPlus);

    // 大括号内部可以放入任意的JavaScript表达式，可以进行运算，以及引用对象属性。
    let str3 = `${str1}+${str2}=${str1 + str2}`;
    console.log(str3);

    // 模板字符串之中还能调用函数。
    function strTest() {
      return 'hello world';
    }
    console.log(`foo ${strTest()} bar`); // foo hello world bar
```

```
    // 如果大括号中的值不是字符串，将按照一般的规则转为字符串。比如，大括号中是一个对象，
将默认调用对象的toString方法。

    // 标签模板
    // alert `123` === alert(123); // true
    {
      let a = 5;
      let b = 10;

      function tag(stringArr, value1, value2) {
        // ....
      }
      tag `Hello ${a + b} world ${a * b}`;
      // 等同于
      tag(['Hello', 'world', ''], 15, 10);
    }
    // “标签模板”的一个重要应用，就是过滤HTML字符串，防止用户输入恶意内容。
  }
  rawTest() {

    // String.raw 方法，往往用来充当模板字符串的处理函数，返回一个斜杠都被转译的字符
串。
    let a = String.raw `Hi\n${2 + 3}`;
    // String.raw方法也可以作为正常的函数使用。这时，它的第一个参数，应该是一个具有raw
属性的对象，且raw属性的值应该是一个数组。
    let b = String.raw({
      raw: 'test'
    }, 0, 1, 2); // 't0e1s2t'
    console.log(a, b);
  }
}
var stringExtension = new StringES6(`hello world!`);
```

正则拓展

ES6 如果RegExp构造函数第一个参数是一个正则对象，那么可以使用第二个参数指定修饰符。而且，返回的正则表达式会忽略原有的正则表达式的修饰符，只使用新指定的修饰符。
type 为修饰符 新增 u 含义为“Unicode”模式，用来正确处理大于\uFFFF的Unicode字符 y 修饰符，与g修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始。不同之处在于，g修饰符只要剩余位置中存在匹配就可，而y修饰符确保匹配必须从剩余的第一个位置开始，这也就是“粘连”的涵义。

点字符

点（.）字符在正则表达式中，含义是除了换行符以外的任意单个字符。对于码点大于0xFFFF的Unicode字符，点字符不能识别，必须加上u修饰符。
ES6新增了使用大括号表示Unicode字符，这种表示法在正则表达式中必须加上u修饰符，才能识别。

y修饰符

y修饰符的作用与g修饰符类似，也是全局匹配，后一次匹配都从上一次匹配成功的下一个位置开始。不同之处

在于，g修饰符只要剩余位置中存在匹配就可，而y修饰符确保匹配必须从剩余的第一个位置开始，这也就是"粘连"的涵义。
// sticky 属性 // 与y修饰符相匹配，表示是否设置了y修饰符 // flags 属性 返回正则表达式的修饰符

```
class RegExpES6 {
  // 正则拓展
  constructor(regExp, type) {
    this.reg = new RegExp(regExp, type);
  }
  //new RegExp(/abc/g, 'i').toString() == /abc/i; // true
  testU() {
    /^\uD83D/u.test('\uD83D\uDC2A'); // false
    /^\uD83D/.test('\uD83D\uDC2A'); // true
  }
  testY() {
    let s = 'aaa_aa_a';
    let reg1 = /a+/g;
    let reg2 = /a+/y;

    reg1.exec(s);
    reg2.exec(s);
    // y修饰符的设计本意，就是让头部匹配的标志^在全局匹配中都有效。
    /b/y.exec('aba'); // null
  }
  consoleProperties() {
    console.log(`sticky: ${this.reg.sticky}`);
    console.log(`flags:${this.reg.flags}`);
  }
}
var regEx = new RegExpES6('\d', 'ym');
```

数值的拓展

isFinite() 判断数值是否是有限的

isNaN() 判断是不是NAN

** ES6将全局方法parseInt()和parseFloat()，移植到Number对象上面，行为完全保持不变。**

Number.EPSILON Number.EPSILON的实质是一个可以接受的误差范围。

安全整数和Number.isSafeInteger()

JavaScript能够准确表示的整数范围在-2^53到2^53之间（不含两个端点），超过这个范围，无法精确表示这个值。 ES6引入了Number.MAX_SAFE_INTEGER和Number.MIN_SAFE_INTEGER这两个常量，用来表示这个范围的上下限。

```
class NumberES6 {
```

```
  // 数值的扩展
  testInFinte() {
    // Number.isFinite
    Number.isFinite(15); // true
    Number.isFinite(Infinity); // false
  }
  testIsNaN() {
    // Number.isNaN
    Number.isNaN(12); // false
    Number.isNaN('15'); // false
    Number.isNaN(true); // false
    Number.isNaN('true'); // false
    Number.isNaN('trsads' / 0); // true
  }

  testIsInteger() {
    Number.isInteger(3.0) === Number.isInteger(3); // true
    Number.isInteger(3.1); // false
  }
  withInErrorMargin(left, right) {
    return Math.abs(left - right) < Number.EPSILON;
  }
  testSavaInteger() {
    Number.MAX_SAFE_INTEGER === Math.pow(2, 53) - 1;
    Number.MIN_SAFE_INTEGER === -Number.MAX_SAFE_INTEGER;
    Number.isSafeInteger(Number.MAX_SAFE_INTEGER); // true
    Number.isSafeInteger(null); // false
    // 实际使用这个函数时，需要注意。验证运算结果是否落在安全整数的范围内，不要只验证运算结果，而要同时验证参与运算的每个值。
  }
  others() {
    // Math对象的扩展
    // 去除小数部分
    Math.trunc(4.1); // 4
    // 参数为正数，返回+1；参数为负数，返回-1；参数为0，返回0；参数为-0，返回-0;其他值，返回NaN。
    Math.sign(-5); // -1
    // 取立方根
    Math.cbrt('8'); // 2
    // 整数使用32位二进制形式表示
    Math.clz32(0); // 0
  }
}
```

数组的扩展

- Array.from() 将两类对象转为真正的数组：类似数组的对象（array-like object）和可遍历（iterable）的对象（包括ES6新增的数据结构Set和Map）。

```
let arrayLike = {
  '0': 'a',
  '1': 'b',
  '2': 'c',
  length: 3
};
let arr = Array.from(arrayLike);
Array.from({length: 3}) // [undefined, undefined, undefined]
```

- Array.from还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。

```
Array.from(arrayLike,x => x+x);
// 等同于
Array.from(arrayLike).map(x => x+x);
```

- Array.from()的另一个应用是，将字符串转为数组，然后返回字符串的长度。因为它能正确处理各种Unicode字符，可以避免JavaScript将大于\uFFFF的Unicode字符，算作两个字符的bug。

Array.of()

- Array.of方法用于将一组值，转换为数组。这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。

```
Array.of(3, 11, 8) // [3,11,8]
Array.of(3) // [3]
Array.of(3).length // 1
Array() // []
Array(3) // [, , ,]
Array(3, 11, 8) // [3, 11, 8]
```

- 数组实例的copyWithin()

它接受三个参数。 target（必需）：从该位置开始替换数据。 start（可选）：从该位置开始读取数据，默认为0。如果为负值，表示倒数。 end（可选）：到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

```
Array.prototype.copyWithin(target, start = 0, end = this.length)
[1, 2, 3, 4, 5].copyWithin(0, 3) // [4, 5, 3, 4, 5]
```

Symbol

1. Symbol 作为属性名，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。
   Reflect.ownKeys() 可以返回所有类型的键名
2. Symbol.for() 与 Symbol()
   for() 会检测key是否已经存在，如果不存在才会新建一个值 Symbol.keyFor方法返回一个已登记的 Symbol 类型值的key。

Set

Set 类似于数组，但是成员的值都是唯一的。 Set 本身是一个构造函数。

属性

- prototype 默认就是 Set
- size 返回实例成员的总数（访问器，不可写）

方法

- add(value) 加入重复值会失败,加入时不会发生 类型转换，所以 5 和 '5' 为不同值，判断标准类似与 === , 但是**NaN 等于自身**
- delete(value) 删除某个值，返回一个布尔值表示是否成功
- has (value) 表示某个值是否是Set成员，返回布尔值
- clear() 清楚所有成员
- 遍历操作 keys(),values(), entries(), forEach() Set没有键名，只有键值，所以keys()和 values()行为完全一致

代码示例

Set构造

Set构造可以接受一个数组或者类似一个数组的对象(*element 节点*)作为参数

```
function initSet(type, arrayLike) {
  switch (type) {
    case 1:
      arrayLike = [1, 2, 4, 2];
      break; //1,2,4
    case 2:
      arrayLike = document.getElementsByTagName('div');
      break;
    default:
      throw new Error('init Set by use default way need a type !')
  }
  return new Set(arrayLike);
}
```

Set 数组去重， 推荐用way 2

```
// way 1
let mySet = new Set();
[1, 1, 2, 5, 6, 2, '5', 1, NaN, NaN].forEach(x => mySet.add(x));
// way 2
function dedupe(array) {
  return Array.from(new Set(array));
}
```

利用Set实现并集，交集，差集

```
function UIDtest(set1, set2) {
  return {
    union: new Set([...set1, ...set2]),
    intersect: new Set([...set1].filter(x => set2.has(x))),
    difference: new Set([...set1].filter(x => !set2.has(x)))
  }
}
```

WeakSet

与Set类似，也是不重复的值的集合
用于储存DOM节点，而不用担心这些节点从文档移除时，引发内存泄漏

1. WeakSet的成员只能是对象，而不能是其他类型的值
2. WeakSet中的对象都是弱引用，如果其他对象都不再引用该对象，那么对象会被回收，即时WeakSet 仍然存在与WeakSet中

WeakSet 构造

WeakSet作为构造函数，可以接受所有具有Iterable接口的对象

方法

- add
- delete
- value *没有size，不能遍历*

```
function testWeakSet(arrayLike) {
  let ws = new WeakSet(arrayLike);
  const a = [
    [1, 2],
    [3, 4]
  ];
  ws = new WeakSet(a); // {[1,2],[3,4]}
```

```
const b = [1, 3];
ws = new WeakSet(b); // error b数组的成员对象不是对象。
```

Map

创建目的： 为了解决JS对象 只能用字符串当作键。
Map 键 不限于字符串，各种类型的值（包括对象）都可以当作键。是一种更完善的Hash结构实现。如果对一个键多次复制，后面的值会覆盖之前的值。
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，事实上，任何具有Iterator接口的数据结构都可以当作Map构造函数的参数

属性

- size

方法

- set(key,value) get(key) has(key) delete(key) clear()
- 遍历方法: keys(),values(), entries(), forEach()

```
function initMap(keyArray, valueArray) {
  let myMap = new Map();
  if (keyArray.length !== valueArray.length) {
    throw new Error('The params length need to be equals!');
  }
  for (let i = 0, len = keyArray.length; i < len; i++) {
    myMap.set(keyArray[i], valueArray[i]);
  }
  return myMap;
}
```

示例

检测键是否存在于map

```
function testMap(map, key) {
  if (map.has(key)) {
    console.log(map.get(key));
    map.delete(key);
  } else {
    console.log('this key is not exist in the map !')
  }
}
```

WeakMap

键名只接受对象，设计目的在于，有时我们想在某个对象上面存放一些数据，但这会行程对于这个对象的引用，一旦我们不再需要这个对象，就必须手动删除这个引用

Iterator 和 for...of 循环

Iterator 是一种接口，为各种不同的数据结构提供统一的访问机制

作用

1. 为各种数据结构，提供一个统一的、简便的访问接口
2. 使得数据结构的成员能够按某种次序排序
3. ES6创造了一种新的遍历命令for...of 循环

遍历过程

（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。

（2）第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。

（3）第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。

（4）不断调用指针对象的next方法，直到它指向数据结构的结束位置。

调用Iterator接口的场合

1. 解构复制
2. 扩展运算符...
3. yield*后面跟一个可遍历结构
4. 其他场合，任何接受数组作为参数的场合，其实都调用

杂项介绍

了遍历器接口 for..of Array.from() Map()... Promise.all() 字符串是一个类似数组的对象，也原生具有Iterator接口 for...of 可正确识别32位UTF-16字符 for...of 的优点:

1. 对比es5 forEach() 他可以与break、continue 和 return 配合使用
2. 对比for ...in 他不会手动添加其他键，而且循环顺序是指定的。

Reflect

##设计目的：

1. 将Obejct对象的一些明显属于语言内部的方法（比如Object.defineProperty），放到Reflect对象上

2. 修改某些Object方法的返回结果，让其变得更加合理。
3. 让Object操作都变成函数行为， name in obj ==> Reflect.has(obj,name)
4. Reflect对象的方法与Proxy对象的方法一一对应。这就让Proxy对象可以方便的调用对应的Reflect方法，完成默认行为
5. 有了Reflect会让很多操作更易读 Function.prototype.apply.call(Math.floor, undefined, [1.75]); ==》 Reflect.apply(Math.floor, undefined, [1.75]) // 1

静态方法

Reflect.apply(target,thisArg,args) Reflect.construct(target,args) Reflect.get(target,name,receiver)
Reflect.set(target,name,value,receiver) Reflect.defineProperty(target,name,desc)
Reflect.deleteProperty(target,name) Reflect.has(target,name) Reflect.ownKeys(target)
Reflect.isExtensible(target) Reflect.preventExtensions(target)
Reflect.getOwnPropertyDescriptor(target, name) Reflect.getPrototypeOf(target)
Reflect.setPrototypeOf(target, prototype)

```
// 实例 使用Proxy实现观察者模式
const queuedObservers = new Set();

const observe = fn => queuedObservers.add(fn);
const observable = obj => new Proxy(obj, {
  set
});

function set(target, key, value, receiver) {
  const result = Reflect.set(target, key, value, receiver);
  queuedObservers.forEach(observer => observer(target));
  return result;
}
const person = observable({
  name: '张三',
  age: 20
});

function print(target) {
  console.log(`${target.name}, ${target.age}`)
}

function isYoung(target) {
  console.log(`this person is too  ${ target.age > 18 ? 'old' : 'young'  }`);
}
observe(print);
observe(isYoung);
```

Promise

Promise 是异步编程的一种解决方案，比传统的解决方案- 回调函数和事件 -更合理和更强大。 所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）从语法上说，Promise是一个对象，从他可以获取异步操作的消息。

特点

1. 对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态： Pending(进行中) ，Resolved(已完成，又称Fulfilled)和 Rejected(已失败)
2. 一旦状态改变，就不会再变，任何适合都可以得到这个结果
3. 缺点：
    1. 无法取消Promise
    2. 如果不设置回调函数，Promise内部抛出的错误不会反应到外部。3.当处于Pending状态时，无法得知进展到哪一个阶段
4. Promise 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个catch语句捕获。

构造

Promise对象是一个构造函数，用来生成Promise实例

then()

可以用该方法来分别指定Resolved状态和Reject状态的回调函数

catch()

该方法可以捕获错误，是.then(null,rejection)的别名，用于指定发生错误时的回调函数

all()、race()

all()用于将多个Promise实例，包装成一个新的Promise实例，每个resolved，总的状态才会变成resolved，只要有一个rejected,总的状态就会变成rejected
race() 和all() 类似，但是只要有一个状态改变，总的状态就会改变

```
var p = Promise.all([p1,p2,p3] 可以不是数组，但必须具有Iterator接口)
```

resolve()

将现有对象转换为Promise对象。 1.参数是一个Promise实例 2.参数是一个thenable对象，即具有then方法，resolve()会将这个对象转为Promise对象，并立即执行then方法 3.参数不具有then方法，或者根本不是一个对象，则会返回一个新的Promise对象，状态为resolve。 4.不带有任何参数，直接返回一个Resolved状态的对象

reject()

返回一个新的Promise实例，状态为rejected

done()

总是处于回调链的尾端，保证抛出任何可能出现的错误

finally()

不管怎么样都会执行。与done（）的区别是该函数不管怎样都会必须执行

示例

Promise 构造举例

```
let myPromise = new Promise(
  function (resolve, reject) {
    if ( /* 异步操作成功*/ true) {
      let value = 'haha';
      resolve(value);
    } else {
      reject(error);
    }
  }
)
myPromise.then(function (value) {
  console.log(`Resolved ! value is :${value}`);
}, function (error) {
  console.log(`Rejected ! error is ${error}`);
});
```

Promise对象实现的Ajax操作的例子

```
var getJSON = function (url) {
  var promise = new Promise(function (resolve, reject) {
    var client = new XMLHttpRequest();
    client.open("GET", url);
    client.onreadystatechange = handler;
    client.responseType = "json";
    client.setRequestHeader("Accept", "application/json");
    client.send();

    function handler() {
      if (this.readyState !== 4) {
        return;
      }
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
  });
```

```
  return promise;
};

// getJSON("/posts.json").then(function(json) {
//   console.log('Contents: ' + json);
// }, function(error) {
//   console.error('出错了', error);
// });
```

done() 实现

```
Promise.prototype.done = function (onFulfilled, onRejected) {
  this.then(onFulfilled, onRejected)
    .catch(function (reason) {
      // 抛出一个全局错误
      setTimeout(() => {
        throw reason
      }, 0);
    });
};
```

finally() 实现

```
Promise.prototype.finally = function (callback) {
  let P = this.constructor;
  return this.then(
    value => P.resolve(callback()).then(() => value),
    reason => P.resolve(callback()).then(() => {
      throw reason
    })
  );
};
```

Generator

形式上，Generator 函数是一个普通函数，但是有两个特征。

1. function关键字与函数名之间有一个星号；
2. 函数体内部使用yield语句，定义不同的内部状态（yield在英语里的意思就是"产出"）。

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语句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。 Generator函数可以不用yield语句，这时

就变成了一个单纯的暂缓执行函数。 **注意，由于next方法的参数表示上一个yield语句的返回值，所以第一次使用next方法时，不能带有参数。**V8引擎直接忽略第一次使用next方法时的参数，只有从第二次使用next方法开始，参数才是有效的。从语义上讲，第一个next方法用来启动遍历器对象，所以不用带有参数。
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。 throw方法可以接受一个参数，该参数会被catch语句接收，建议抛出Error对象的实例 throw方法被捕获以后，会附带执行下一条yield语句。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。

JavaScript语言的Thunk函数

JavaScript 语言是传值调用，它的 Thunk 函数含义有所不同。在 JavaScript 语言中，Thunk 函数替换的不是表达式，而是多参数函数，将其替换成一个只接受回调函数作为参数的单参数函数。 例子：// 正常版本的readFile（多参数版本）

```
fs.readFile(fileName, callback);

// Thunk版本的readFile（单参数版本）
var Thunk = function (fileName) {
  return function (callback) {
    return fs.readFile(fileName, callback);
  };
};

var readFileThunk = Thunk(fileName);
readFileThunk(callback);
```

async 函数

ES2017 标准引入了 async 函数，使得异步操作变得更加方便。 async 函数是什么？一句话，它就是 Generator 函数的语法糖。对Generator函数的改进

优点

1. 内置执行器，不需要co模块支持
2. 更好的语义
3. 更广的适用性 await命令后面可以是Promise对象和原始类型的值
4. 返回值是Promise对象，比Iterator对象好操作，可以用then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。
   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。如果不是，会被转成一个立即resolve的 Promise 对象。 只要一个await语句后面的 Promise 变为reject，那么整个async函数都会中断执行。 我们希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个await放在try...catch结构里面，这样不管这个异步操作是否成功，第二个await都会执行。

使用注意点

1. await命令后面的Promise对象，运行结果可能是rejected，所以最好把await命令放在try...catch代码块中。
2. 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。
3. await命令只能用在async函数之中，如果用在普通函数，就会报错。如果确实希望多个请求并发执行，可以使用Promise.all方法。

class

class也是直接对类使用new命令，跟构造函数的用法完全一致。 构造函数的prototype属性，在ES6的"类"上面继续存在。事实上，**类的所有方法都定义在类的prototype属性上面。 类的内部所有定义的方法，都是不可枚举的（non-enumerable）。**

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。默认返回实例对象（即this），完全可以指定返回另一个对象 **class 与传统构造函数相比，不存在变量提升**

私有方法

ES6不提供，但有变通方法可模拟实现

1. 在命名上加以区别，_functionName
2. 将私有方法移出模块

```
class Widget {
  foo (baz) {
    bar.call(this, baz);
  }

  // ...
}
```

3. 利用Symbol值的唯一性，将私有方法的名字命名为一个Symbol值

```
const bar = Symbol('bar');
const snaf = Symbol('snaf');

export default class myClass{

  // 公有方法
  foo(baz) {
    this[bar](baz);
  }

  // 私有方法
```

```
  [bar](baz) {
    return this[snaf] = baz;
  }

  // ...
};
```

this 的指向

解决this指向问题的方法：

1. 在构造方法中绑定this
2. 使用箭头函数
3. 使用Proxy

name 属性

总是返回紧跟在class 关键字后面的类名

new.target属性

用于返回构造函数通过什么调用的，用途看下面

```
const MyClass = class Me {
  constructor(name) {
    if (new.target === Me || new.target === undefined) {
      throw new Error(`本类必须通过子类构建实例化且必须使用new关键字！`)
    }
    this.name = name;
    this.getName = this.getName.bind(this);
  }
  getName() {
    return this.name;
  }
}
```

class的继承

extends super关键字表示父类的构造函数，用来新建父类的this对象，子类必须在constructor方法中调用super方法。因为子类没有自己的this对象，必须继承父类的this对象，然后对其加工 extends的继承目标

1. 子类继承Object类，子类就是构造函数Object的复制
2. 不存在继承，则直接继承Function.prototype
3. 继承null，Function.prototype Object.getPrototypeOf() 从子类上获取父类

```
//@addChildVariable
```

```
class MyChildClass extends MyClass {
  constructor(name, age) {
    super(name);
    this.age = age;
  }
  // message: '这样定义静态属性无效！'
  get myName() {
    return this.name;
  }
  set myName(value) {
    this.name = value;
    console.log(value);
  }
  static greeting() {
    console.log(`this is MyChildClass's static function !`);
  }
  getInfo() {
    return `name is ${super.getName()} age is ${this.age}`;
  }
  // ES7提案的属性定义方法 Babel转码器支持
  // message = 'ES7属性等式定义！';
  // static message = 'ES7静态属性定义！';
  // 提案： 私有属性 在属性名之前用#表示
}
```

原生构造函数的继承

因为ES6是先新建父类的实例对象this，然后再用子类的构造函数修饰this，使得父类的所有行为都可以继承。而ES5是先生成子类实例，再讲父类的属性添加到子类上 //Class的取值函数和存值函数 getter ,setter //Class的静态方法 static 静态方法通过类调用，可被子类继承，也可被super对象调用 //Class的静态属性和实例属性 */

Mixin模式的实现

多个类的接口“混入”（mix in）另一个类

```
function mix(...mixins) {
  class Mix {}

  for (let mixin of mixins) {
    copyProperties(Mix, mixin);
    copyProperties(Mix.prototype, mixin.prototype);
  }

  return Mix;
}

function copyProperties(target, source) {
  for (let key of Reflect.ownKeys(source)) {
```

```
    if (key !== "constructor" &&
      key !== "prototype" &&
      key !== "name"
    ) {
      let desc = Object.getOwnPropertyDescriptor(source, key);
      Object.defineProperty(target, key, desc);
    }
  }
}
```

Decorator 修饰器

是一个函数，用来修改类的行为ES7的提案

```
function addChildVariable(target) {
  target.decorator = addChildVariable;
}
```

模块化Module

CommonJS 模块

```
let { stat, exists, readFile } = require('fs');

// 等同于
let _fs = require('fs');
let stat = _fs.stat;
let exists = _fs.exists;
let readfile = _fs.readfile;
```

上面的代码实质是整体加载fs模块（即加载fs的所有方法），生成一个对象(_fs)，然后再从这个对象上读取3个方法，这种加载称为**运行时加载**，因为只有运行时才能得到这个对象，导致完全没办法在编译时做**静态优化**。
ES6模块不是对象，而是通过export命令显式指定输出的代码，再通过import命令输入。

```
import { stat, exists, readFile } from 'fs';
```

上面代码的实质是从fs模块加载3个方法，其他方法不加载。这种加载成为**编译时加载**或者**静态加载**，即ES6可以在编译时就在完成模块加载，效率比CommonJS模块的加载方式高。当然，这也导致了没法引用ES6模块本身，因为他不是对象。

优点

1. 不再需要UMD模块格式了，将来服务器和浏览器都会支持 ES6 模块格式。目前，通过各种工具库，其实已经做到了这一点。
2. 将来浏览器的新 API 就能用模块格式提供，不再必须做成全局变量或者navigator对象的属性。
3. 不再需要对象作为命名空间（比如Math对象），未来这些功能可以通过模块提供。

export命令

命令用于规定模块的对外接口

1. 写法1

```
export var firstName = 'Michael';
export var lastName = 'Jackson';
export var year = 1958;
```

2. 写法2

```
var firstName = 'Michael';
var lastName = 'Jackson';
var year = 1958;

export {firstName, lastName, year};
```

export除了输出变量还可以输出函数或者类（class） export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以渠道模块内部实时的值

```
export var foo = 'bar';
setTimeout(() => foo = 'baz', 500);
// 输出变量foo,值为bar,500毫秒后变成baz
```

export命令可以出现在模块的任何位置，只要处于模块顶层就可以。下面介绍的import也是如此

import 命令

- import命令接受一对大括号，里面指定要从其他模块导入的变量名。大括号里面的变量名必须与被导入模块对外接口的名称相同。
- 如果想为输入的变量重新取一个名字，import命令要使用as关键字，将输入的变量重命名，如下面代码的newName
- import后面的from指定模块文件的位置，可以是相对路径或者绝对路径，也可以是模块名（必须有配置文件）
- import 命令具有提升效果
- 由于import是静态执行，所以不能使用表达式和变量
- import语句会执行所加载的模块

- 重复执行同一句import模块只会执行一次

```
import {myMethod} from 'util';
import {firstName as newName, lastName, year} from './profile';

function setName(element) {
  element.textContent = newName + ' ' + lastName;
}
```

模块的整体加载

除了指定加载某个输出值，还可以使用整体加载，即用星号（*）指定一个对象，所有输出值都加载在这个对象上面。

```
// circle.js
export function area(radius) {
  return Math.PI * radius * radius;
}

export function circumference(radius) {
  return 2 * Math.PI * radius;
}
// 普通加载方式
// main.js

import { area, circumference } from './circle';

console.log('圆面积：' + area(4));
console.log('圆周长：' + circumference(14));
// 整体加载方式
import * as circle from './circle';

console.log('圆面积：' + circle.area(4));
console.log('圆周长：' + circle.circumference(14));
```

模块整体加载所在的那个对象比如上例的 circle ，应该是可以静态分析的，所以不允许运行时改变。

export default 命令

由于import命令的时候，用户需要知道所要加载的变量名或者函数名，否则无法加载。
为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。

```
// export-default.js
export default function () {
```

```
  console.log('foo');
}
// import-default.js 注意，这里import命令后面不需要大括号
import customName from './export-default';
customName(); // 'foo'
```

如果想在一条import语句中，同时输入默认方法和其他接口，可以写成下面这样。

```
import _, { each, each as forEach } from 'lodash';
```

export default 方法也可以用来输出类

```
// MyClass.js
export default class { ... }

// main.js
import MyClass from 'MyClass';
let o = new MyClass();
```

模块的继承

模块之间也可以继承。

假设有一个circleplus模块，继承了circle模块。

```
// circleplus.js

export * from 'circle';
export var e = 2.71828182846;
export default function(x) {
  return Math.exp(x);
}
```

上面代码中的export *，表示再输出circle模块的所有属性和方法。注意，export *命令会忽略circle模块的default方法。然后，上面代码又输出了自定义的e变量和默认方法。

这时，也可以将circle的属性或方法，改名后再输出。

```
// circleplus.js

export { area as circleArea } from 'circle';
```

上面代码表示，只输出circle模块的area方法，且将其改名为circleArea。

加载上面模块的写法如下。

```
// main.js

import * as math from 'circleplus';
import exp from 'circleplus';
console.log(exp(math.e));
```

上面代码中的import exp表示，将circleplus模块的默认方法加载为exp方法。

跨模块常量

本书介绍const命令的时候说过，const声明的常量只在当前代码块有效。如果想设置跨模块的常量（即跨多个文件），或者说一个值要被多个模块共享，可以采用下面的写法。

```
// constants.js 模块
export const A = 1;
export const B = 3;
export const C = 4;

// test1.js 模块
import * as constants from './constants';
console.log(constants.A); // 1
console.log(constants.B); // 3

// test2.js 模块
import {A, B} from './constants';
console.log(A); // 1
console.log(B); // 3
```

如果要使用的常量非常多的情况

可以建一个专门的constants目录，将各种常量写在不同的文件里面，保存在该目录下。然后，将这些文件输出的常量，合并在index.js里面。使用的时候，直接加载index.js就可以了。

```
// constants/db.js
export const db = {
  url: 'http://my.couchdbserver.local:5984',
  admin_username: 'admin',
  admin_password: 'admin password'
};

// constants/user.js
export const users = ['root', 'admin', 'staff', 'ceo', 'chief', 'moderator'];
```

```
// constants/index.js
export {db} from './db';
export {users} from './users';
// script.js
import {db, users} from './index';
```

import（）

问题：import命令会被 JavaScript 引擎静态分析，先于模块内的其他模块执行（叫做"连接"更合适）。所以，下面的代码会报错。

```
if (x === 2) {
  import MyModual from './myModual';
}
```

这样的设计，固然有利于编译器提高效率，但也导致无法在运行时加载模块。在语法上，条件加载就不可能实现。如果import命令要取代 Node 的require方法，这就形成了一个障碍。因为require是运行时加载模块，import命令无法取代require的动态加载功能。

```
const path = './' + fileName;
const myModual = require(path);
```

提案：引入import()函数，完成动态加载。

```
import(specifier)
```

上面代码中，import函数的参数specifier，指定所要加载的模块的位置。import命令能够接受什么参数，import()函数就能接受什么参数，两者区别主要是后者为动态加载。
import()返回一个 Promise 对象。 import()函数可以用在任何地方，不仅仅是模块，非模块的脚本也可以使用。它是运行时执行，也就是说，什么时候运行到这一句，也会加载指定的模块。另外，import()函数与所加载的模块没有静态连接关系，这点也是与import语句不相同。

import()类似于 Node 的require方法，区别主要是前者是异步加载，后者是同步加载。

适用场合

1. 按需加载
2. 条件加载
3. 动态的模块路径

注意点

import()加载模块成功以后，这个模块会作为一个对象，当作then方法的参数。因此，可以使用对象解构赋值的语法，获取输出接口。

```
import('./myModule.js')
.then(({export1, export2}) => {
  // ...·
});
// 如果模块有default 输出接口，可以用参数直接获得
import('./myModule.js')
.then(myModule => {
  console.log(myModule.default);
});
```

如果想同时加载多个模块，可以采用下面的写法。

```
Promise.all([
  import('./module1.js'),
  import('./module2.js'),
  import('./module3.js'),
])
.then(([module1, module2, module3]) => {
   ...
});
```

import()也可以用在 async 函数之中。

```
async function main() {
  const myModule = await import('./myModule.js');
  const {export1, export2} = await import('./myModule.js');
  const [module1, module2, module3] =
    await Promise.all([
      import('./module1.js'),
      import('./module2.js'),
      import('./module3.js'),
    ]);
}
main();
```