1：函数默认值的指定

ES6 之前，不能直接为函数的参数指定默认值，只能采用变通的方法。

```
function log(x, y) {
  y = y || 'World';
  console.log(x, y);
}

log('Hello') // Hello World
log('Hello', 'China') // Hello China
log('Hello', '') // Hello World
```

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

```
function log(x, y = 'World') {
  console.log(x, y);
}

log('Hello') // Hello World
log('Hello', 'China') // Hello China
log('Hello', '') // Hello
```

另外，一个容易忽略的地方是，参数默认值不是传值的，而是每次都重新计算默认值表达式的值。也就是说，参数默认值是惰性求值的。

```
let x = 99;
function foo(p = x + 1) {
  console.log(p);
}

foo() // 100

x = 100;
foo() // 101
```

上面代码中，参数`p`的默认值是`x + 1`。这时，每次调用函数`foo`，都会重新计算`x + 1`，而不是默认`p`等于 100。

作为练习，请问下面两种写法有什么差别？

```
// 写法一
function m1({x = 0, y = 0} = {}) {
  return [x, y];
}

// 写法二
function m2({x, y} = { x: 0, y: 0 }) {
  return [x, y];
}
```

上面两种写法都对函数的参数设定了默认值，区别是写法一函数参数的默认值是空对象，但是设置了对象解构赋值的默认值；写法二函数参数的默认值是一个有具体属性的对象，但是没有设置对象解构

上一章 下一章

作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域

作用域

一旦设置了参数的默认值，函数进行声明初始化时，参数会形成一个单独的作用域（context）。等到初始化结束，这个作用域就会消失。这种语法行为，在不设置参数默认值时，是不会出现的。

```
var x = 1;

function f(x, y = x) {
  console.log(y);
}

f(2) // 2
```

上面代码中，参数`y`的默认值等于变量`x`。调用函数`f`时，参数形成一个单独的作用域。在这个作用域里面，默认值变量`x`指向第一个参数`x`，而不是全局变量`x`，所以输出是`2`。

再看下面的例子。

```
let x = 1;

function f(y = x) {
  let x = 2;
  console.log(y);
}

f() // 1
```

上面代码中，函数`f`调用时，参数`y = x`形成一个单独的作用域。这个作用域里面，变量`x`本身没有定义，所以指向外层的全局变量`x`。函数调用时，函数体内部的局部变量`x`影响不到默认值变量`x`。

如果此时，全局变量`x`不存在，就会报错。

```
function f(y = x) {
  let x = 2;
  console.log(y);
}

f() // ReferenceError: x is not defined
```

下面这样写，也会报错。

```
var x = 1;

function foo(x = x) {
  // ...
}

foo() // ReferenceError: x is not defined
```

上面代码中，参数`x = x`形成一个单独作用域。实际执行的是`let x = x`，由于暂时性死区的原因，这行代码会报错”x 未定义“。

如果参数的默认值是一个函数，该函数的作用域也遵守这个规则。请看下面的例子。

```
let foo = 'outer';

function bar(func = () => foo) {
  let foo = 'inner';
  console.log(func());
}

bar(); // outer
```

上面代码中，函数`bar`的参数`func`的默认值是一个匿名函数，返回值为变量`foo`。函数参数形成的单独作用域里面，并没有定义变量`foo`，所以`foo`指向外层的全局变量`foo`，因此输出`outer`。

如果写成下面这样，就会报错。

```
function bar(func = () => foo) {
  let foo = 'inner';
  console.log(func());
}

bar() // ReferenceError: foo is not defined
```

上面代码中，匿名函数里面的`foo`指向函数外层，但是函数外层并没有声明变量`foo`，所以就报错了。

```
  var x = 1;  //1:全局作用域
      //2:foo2内部形成一个单独的作用域，
function foo2(x,yy = function(){ x=2;
console.log(x)}){
    var x = 3;  // 3:函数内部作用域

    yy();  // 2
    console.log(x) ;
}
console.log(x+"===")  //1
foo2();  //3
```

这里不理解为什么！！

如果将 `var x = 3` 的 `var` 去除，函数 `foo` 的内部变量 `x` 就指向第一个参数 `x`，与匿名函数内部的 `x` 是一致的，所以最后输出的就是 `2`，而外层的全局变量 `x` 依然不受影响。

```
var x = 1;
function foo(x, y = function() { x = 2; }) {
  x = 3;
  y();
  console.log(x);
}

foo() // 2
x // 1
```

应用

edit

利用参数默认值，可以指定某一个参数不得省略，如果省略就抛出一个错误。

```
function throwIfMissing() {
  throw new Error('Missing parameter');
}

function foo(mustBeProvided = throwIfMissing()) {
  return mustBeProvided;
}

foo()
// Error: Missing parameter
```

上面代码的 `foo` 函数，如果调用的时候没有参数，就会调用默认值 `throwIfMissing` 函数，从而抛出一个错误。

从上面代码还可以看到，参数 `mustBeProvided` 的默认值等于 `throwIfMissing` 函数的运行结果（注意函数名 `throwIfMissing` 之后有一对圆括号），这表明参数的默认值不是在定义时执行，而是在运行时执行。如果参数已经赋值，默认值中的函数就不会运行。

另外，可以将参数默认值设为 `undefined`，表明这个参数是可以省略的。

```
function foo(optional = undefined) { ··· }
```

2：reset参数

引入了es6 reset参数，形式 ...变量名