函数的扩展

1. 函数参数默认值

1.1. 基本用法

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, y 无值是 undefined 会取 'World'
log('Hello', 'China') // Hello China
log('Hello', '') // Hello World, y 有值是空字符串，还是会取 'World'
```

如果参数 `y` 赋值了 `falsely`，则该赋值不起作用，结果被改为默认值。

为了避免这个问题，通常需要先判断一下参数 `y` 是否被赋值，如果没有，再等于默认值。

```
if (typeof y === 'undefined') {
  y = '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
```

ES6 的写法除了简洁，还有两个好处：

> - 首先，阅读代码的人，可以立刻意识到哪些参数是可以省略的，不用查看函数体或文档；
> - 其次，有利于将来的代码优化，即使未来的版本在对外接口中，彻底拿掉这个参数，也不会导致以前的代码无法运行。

默认值的参数变量是默认声明的，所以不能用 `let` 或 `const` 再次声明。

```
function foo(x = 5) {
  let x = 1; // Identifier 'x' has already been declared
```

```
  const x = 2; // Identifier 'x' has already been declared
}
```

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);
}
// p = x + 1; x 值改变后，p 会重新求值
foo() // 100，x 是 99 时，p 是 100

x = 100;
foo() // 101，x 是 100 时，p 是 101
```

每次调用函数 `foo()`，都会重新计算 `x + 1`。

1.2. 函数的 length 属性

指定了默认值以后，函数的 `length` 属性，将返回没有指定默认值的参数个数。这是因为 `length` 属性的含义是，该函数预期传入的参数个数。某个参数指定默认值以后，预期传入的参数个数就不包括这个参数了。

```
(function (a) {}).length // 1
(function (a = 5) {}).length // 0
(function (a, b, c = 5) {}).length // 2
```

如果设置了默认值的参数不是尾参数，那么length属性也不再计入后面的参数了。

```
(function (a = 0, b, c) {}).length // 0
(function (a, b = 1, c) {}).length // 1
```

1.3. 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。

```
let x = 1;
function f(x, y = x) {
  console.log(y);
}
f(2) // 2
```

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。

```
let x = 1;
function f(y = x) {
  let x = 2; // 此时定义的 x 不影响默认变量 x
  console.log(y);
}
f() // 1
```

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
```

2. rest 参数

rest 参数（形式为 ...变量名），用于获取函数的多余参数，这样就不需要使用 arguments 对象了。**rest 参数搭配的变量是一个数组**，该变量将多余的参数放入数组中。

```
function add(...values) {
  let sum = 0;
  for (let val of values) {
    sum += val;
  }
  return sum;
}
add(2, 5, 3) // 10
```

使用 rest 参数代替 arguments 变量的例子。

```
// arguments 变量的写法
function sortNumbers() {
  // arguments 是类似数组的对象，需要先转换为数值才能使用 sort() 方法
  return Array.from(arguments).sort();
}

// rest 参数的写法
const sortNumbers = (...numbers) => numbers.sort();
```

rest 参数之后不能再有其他参数（即只能是最后一个参数），否则会报错。

```
function f(a, ...b, c) {
  // ...
}
// Rest parameter must be last formal parameter
```

函数的 length 属性，不包括 rest 参数。

```
(function(a) {}).length  // 1
(function(...a) {}).length  // 0
(function(a, ...b) {}).length  // 1
```

3. name 属性

函数的 name 属性，返回该函数的函数名，ES6 将其写入了标准。

```
function foo() {}
foo.name // "foo"
```

如果将一个匿名函数赋值给一个变量，ES6 的 name 属性会返回变量名作为实际的函数名。

```
var f = function () {};
f.name // "f"
```

如果将一个具名函数赋值给一个变量，ES6 的name属性都返回这个具名函数原本的名字。

```
const bar = function baz() {};
bar.name // "baz"
```

Function 构造函数返回的函数实例，name属性的值为anonymous。

```
(new Function).name // "anonymous"（匿名的，不知名的）
```

bind 返回的函数，name 属性值会加上 bound 前缀。

```
function foo() {};
foo.bind({}).name // "bound foo"
```

```
(function(){}).bind({}).name // "bound "
```

4. Function.prototype.toString()

ES2019 对函数实例的 `toString()` 方法做出了修改。修改后的 `toString()` 方法，明确要求返回一模一样的原始代码。

```
function /* foo comment */ foo () {}
foo.toString() // "function /* foo comment */ foo () {}"

function a() {/**/}
a.toString() // 'function a() {/**/}'
```

5. catch 命令的参数省略

以前明确要求 `try...catch` 结构中，`catch` 命令后面必须跟参数，接受 `try` 代码块抛出的错误对象。

```
try {
  // ...
} catch (err) {
  // 处理错误
}
```

很多时候，`catch` 代码块可能用不到这个参数。但为了保证语法正确，还是必须写。ES2019 做出了改变，允许 `catch` 语句省略参数。

```
try {
  // ...
} catch {
  // ...
}
```