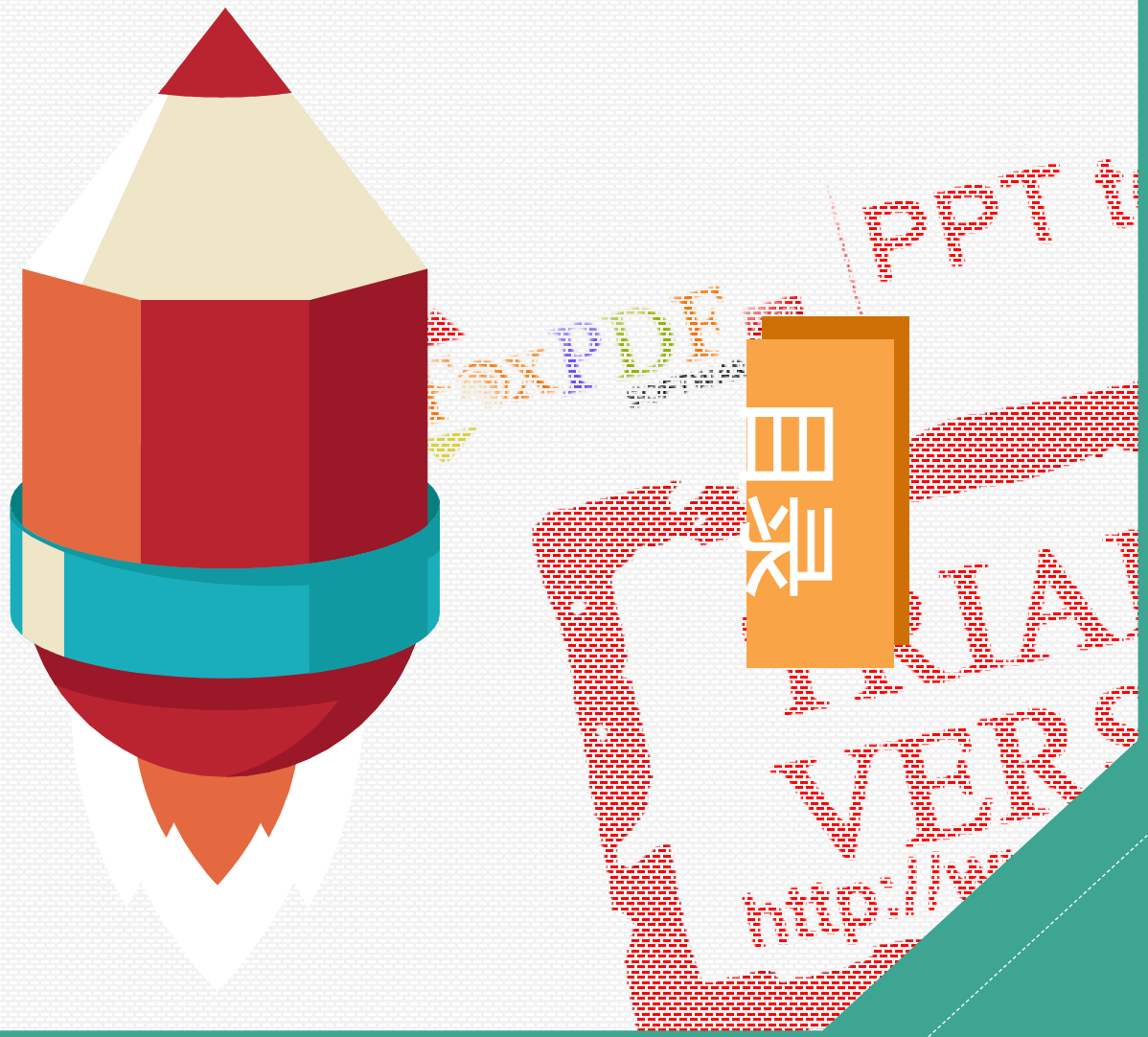
# 第28课：ES6进阶之函数类型拓展

主讲老师：万章



# 函数的参数设置

## 箭头函数(最重要更新之一)

# 函数的参数设置

# 函数的默认参数设置

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
```

上面代码检查函数log的参数y有没有赋值，如果没有，则指定默认值为World。

这种写法的缺点在于，如果参数y赋值了，但是对应的布尔值为false，则该赋值不起作用。就像上面代码的最后一行，参数y等于空字符，结果被改为默认值

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

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

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。

# 函数的默认参数设置

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
```

等价

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

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

```
function Point(x = 0, y = 0) {
    this.x = x;
    this.y = y;
}

let p = new Point();
p;// { x: 0, y: 0 }
let o=new Point("hello","world");
o;//{x:"hello",y:"world"}
```

用在构造函数上

除了简洁，ES6 的写法还有两个好处：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。

# 函数的默认参数设置

```
function foo(x = 5) {
  let x = 1; // error
  const x = 2; // error
}
```

参数变量是默认声明的，所以不能用let或const再次声明。

参数变量x是默认声明的，在函数体中，不能用let或const再次声明，否则会报错。

```
// 不报错
function foo(x, x, y) {
  // ...
}

// 报错
function foo(x, x, y = 1) {
  // ...
}
// SyntaxError: Duplicate parameter name not allowed in this context
```

使用参数默认值时，函数不能有同名参数

```
let x = 99;

function foo(p = x++) {
    console.log(p);
}

foo() // 99
foo() // 100
foo() // 101
```

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。

左侧代码中，参数p的默认值是x++。这时，每次调用函数foo，都会重新计算x++，而不是默认p等于 99。

# 函数的默认参数设置

```
function foo({x, y = 5}) {
  console.log(x, y);
}

foo({}) // undefined 5
foo({x: 1}) // 1 5
foo({x: 1, y: 2}) // 1 2
foo() // TypeError: Cannot read property 'x' of undefined
```

参数默认值可以与解构赋值的默认值，结合起来使用。

上面代码只使用了对象的解构赋值默认值，没有使用函数参数的默认值。只有当函数foo的参数是一个对象时，变量x和y才会通过解构赋值生成。如果函数foo调用时没提供参数，变量x和y就不会生成，从而报错。通过提供函数参数的默认值，就可以避免这种情况。

```
function foo({x, y = 5} = {}) {
  console.log(x, y);
}

foo() // undefined 5
```

# 函数的默认参数设置

通常情况下，定义了默认值的参数，应该是函数的尾参数。因为这样比较容易看出来，到底省略了哪些参数。

如果非尾部的参数设置默认值，实际上这个参数是没法省略的。

```
// 例一
function f(x = 1, y) {
  return [x, y];
}

f() // [1, undefined]
f(2) // [2, undefined])
f(, 1) // 报错
f(undefined, 1) // [1, 1]

// 例二
function f(x, y = 5, z) {
  return [x, y, z];
}

f() // [undefined, 5, undefined]
f(1) // [1, 5, undefined]
f(1, ,2) // 报错
f(1, undefined, 2) // [1, 5, 2]
```

有默认值的参数都不是尾参数。这时，无法只省略该参数，而不省略它后面的参数，**除非显式输入undefined**。

如果传入undefined，将触发该参数等于默认值，null则没有这个效果。

```
function foo(x = 5, y = 6) {
  console.log(x, y);
}

foo(undefined, null)
// 5 null
```

# 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。也就是说，指定了默认值后，length属性将失真。

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

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

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。

# 函数的参数作用域

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
```

左侧代码中，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。

```
let x = 1;

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

f() // 1
```

左侧代码中，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，如右侧代码。

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

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

# 函数的默认参数设置

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

foo() // 3
x // 1
```

左侧代码中，函数foo的参数形成一个单独作用域。这个作用域里面，首先声明了变量x，然后声明了变量y，y的默认值是一个匿名函数。这个匿名函数内部的变量x，指向同一个作用域的第一个参数x。

函数foo内部又声明了一个内部变量x，该变量与第一个参数x由于不是同一个作用域，所以不是同一个变量，因此执行y后，内部变量x和外部全局变量x的值都没变。

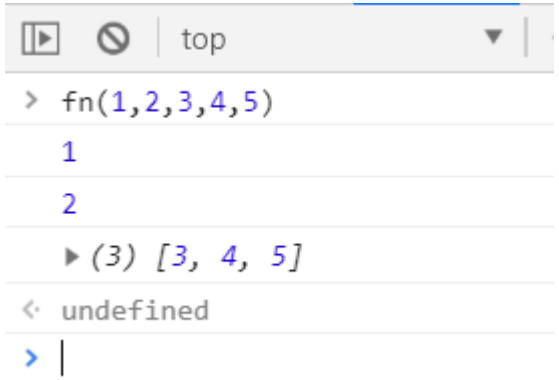
```
var x = 1;
function foo(x, y = function() { x = 2; }) {
  x = 3;
  y();
  console.log(x);
}

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

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。

# 函数的rest参数设置

ES6 引入 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(是真实数组不是arguments那种的类数组)，该变量将多余的参数放入数组中。

```
function fn(x,y,...other){
    console.log(x);
    console.log(y);
    console.log(other);
}
```




```
function push(array, ...items) {
  items.forEach(function(item) {
    array.push(item);
    console.log(item);
  });
}

var a = [];
push(a, 1, 2, 3)
```

用 rest 参数改写数组push方法

注意， rest 参数它就是一个真正的数组，数组特有的方法都可以使用。

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。

```
// 报错
function f(a, ...b, c) {
  // ...
}
```

# 函数的严格模式

```
function doSomething(a, b) {
  'use strict';
  // code
}
```

```
// 报错
function doSomething(a, b = a) {
  'use strict';
  // code
}

// 报错
const doSomething = function ({a, b}) {
  'use strict';
  // code
};
```

从 ES5 开始，函数内部可以设定为严格模式。

ES2016 做了一点修改，规定只要函数参数使用了默认值、解构赋值、或者扩展运算符，那么函数内部就不能显式设定为严格模式，否则会报错。

这样规定的原因是，函数内部的严格模式，同时适用于函数体和函数参数。但是，函数执行的时候，先执行函数参数，然后再执行函数体。

这样就有一个不合理的地方，只有从函数体之中，才能知道参数是否应该以严格模式执行，但是参数却应该先于函数体执行。

所以在ES6中，只要参数使用了默认值、解构赋值、或者扩展运算符，就不能显式指定严格模式。

# 函数的name属性

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

函数的name属性，返回该函数的函数名。

这个属性早就被浏览器广泛支持，但是直到 ES6，才将其写入了标准。

```
var f = function () {};

// ES5
f.name // ""

// ES6
f.name // "f"
```

需要注意的是，ES6 对这个属性的行为做出了一些修改。如果将一个匿名函数赋值给一个变量，ES5 的name属性，会返回空字符串，而 ES6 的name属性会返回实际的函数名。

```
const bar = function baz() {};

// ES5
bar.name // "baz"

// ES6
bar.name // "baz"
```

如果将一个具名函数赋值给一个变量，则 ES5 和 ES6 的name属性都返回这个具名函数原本的名字。

# 箭头函数

# 箭头函数(ES6最重要更新之一)

```
function fn(name){
    console.log(name);
}
```

等价

```
fn = name => name;
```

```
> fn("万章")
< "万章"
>
```

```
fn = (name,age) => name+age;
```

等价

```
function fn(name,age){
    console.log(name+age);
}
```

```
fn = () => console.log("万章");
```

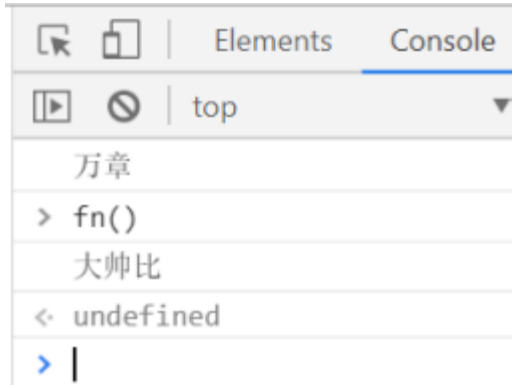
等价

```
function fn(){
    console.log("万章");
}
```

如果箭头函数不需要参数或需要多个参数，就使用一个圆括号代表参数部分。

# 箭头函数(ES6最重要更新之一)

```
fn = () => console.log("大帅比");console.log("万章");
```




```
function fn(){
    console.log("大帅比");
    console.log("万章");
}
```

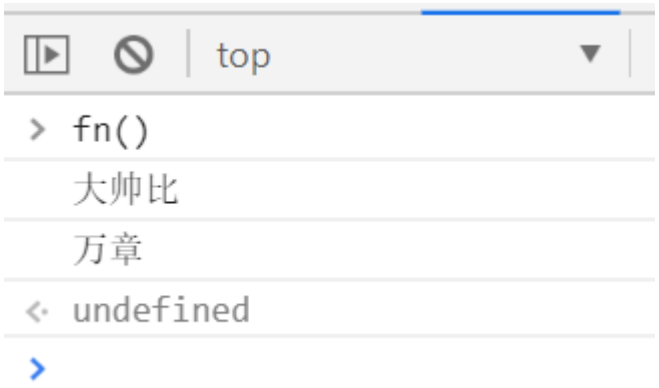
```
fn = () => console.log("大帅比");console.log("万章");
```

```
fn = () => console.log("大帅比");

console.log("万章");
```

# 箭头函数(ES6最重要更新之一)

```
fn = () => {console.log("大帅比");console.log("万章")};
```

```
function fn(){
    console.log("大帅比");
    console.log("万章");
}
```




如果箭头函数的代码块部分**多于一条语句**，就要使用大括号将它们括起来

# 箭头函数(ES6最重要更新之一)

```
function fn(){
    return {x:1,y:2};
}
```

```
fn = () => {x:1,y:2};
```

```
fn = () => ({x:1,y:2});
```

由于大括号被解释为代码块，所以如果箭头函数直接返回一个对象，必须在对象外面加上括号，否则会报错

# 箭头函数(ES6最重要更新之一)

```
const full = ({ first, last }) => first + ' ' + last;

// 等同于
function full(person) {
  return person.first + ' ' + person.last;
}
```

箭头函数可以与变量解构结合使用。

```
const numbers = (...nums) => nums;

numbers(1, 2, 3, 4, 5)
// [1,2,3,4,5]

const headAndTail = (head, ...tail) => [head, tail];

headAndTail(1, 2, 3, 4, 5)
// [1,[2,3,4,5]]
```

rest 参数与箭头函数结合的例子。

```
// 正常函数写法
[1,2,3].map(function (x) {
  return x * x;
});

// 箭头函数写法
[1,2,3].map(x => x * x);
```

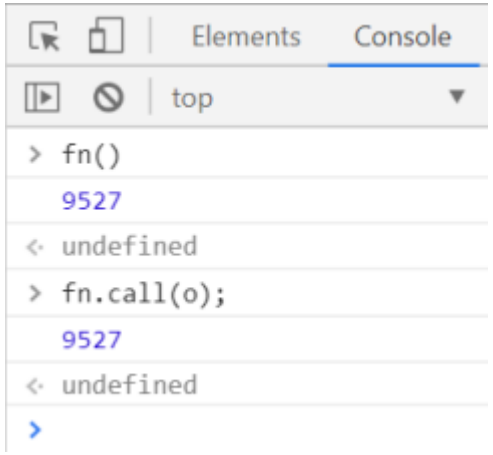
```
// 正常函数写法
var result = values.sort(function (a, b) {
  return a - b;
});

// 箭头函数写法
var result = values.sort((a, b) => a - b);
```

简化回调函数

# 箭头函数的关键注意点

（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。

```
fn = () => console.log(this.x);

var x=9527;

let o={x:1,y:2};
```




WTF!!!!!!!　　call方法里面的this竟然指向了window对象

# 箭头函数的关键注意点

**（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。**

```
var str = 'window';

const obj = {
    str: 'obj',
    fn: () => {
        console.log(this.str);
    },
    fn2: function () {
        console.log(this.str, '当前词法作用域中的this')
        return {
            str: 'newObj',
            fn: () => {
                console.log(this.str);
            }
        }
    }
}

obj.newFn = () => {
    console.log(this.str);
}
```

```
> obj.fn();
  window
< undefined
> obj.newFn();
  window
< undefined
> var newObj = obj.fn2();
  obj 当前词法作用域中的this
< undefined
> newObj.fn();
  obj
< undefined
>
```

这里已经不难看出来了，当我们创建对象的时候，是在全局作用域下创建的，而对象中的方法也是这时候创建的（参照obj.newFn），所以这时候的this是指向全局的，而我们在fn2里面创建的对象，这个对象的方法的this就指向他被创建时的词法作用域obj了

# 箭头函数的关键注意点

**（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。**

```
var str = 'window';

const obj = {
    str: 'obj',
    fn: function () {
        console.log(this.str, '当前词法作用域中的this');
        return () => {
            console.log(this.str);
        }
    }
}
const obj2 = {
    str: 'obj2'
}
```


这时候就可以发现， 无论我们怎么改变箭头函数arrowFn的调用方式，都不会改变this的指向，this始终指向它被创建时所处的词法作用域中的this，新的函数arrowFn在被创建的时候,词法作用域

# 箭头函数的关键注意点

**（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。**

```
var a = 11

function test1() {
    this.a = 22;
    let b = function () {
        console.log(this.a);
    };
    b();//输出11
}
var x = new test1();
```

普通函数中的this指向的是调用它的对象，如果没有直接调用对象，会指向undefined或者window，一般都会指向window，在严格模式下才会指向undefined

```
var a = 11;

function test2() {
    this.a = 22;
    let b = () => {
        console.log(this.a)
    }
    b(); //输出22
}
var x = new test2();
```

在使用new操作符新生成一个对象x中，拥有一个属性a,值为22,同时也创了一个新的箭头函数,这个箭头函数是在新对象x中建立的，所以函数内的this指向的就是对象x

# 箭头函数的关键注意点

**（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。**

```
function foo() {
  setTimeout(() => {
    console.log('id:', this.id);
  }, 100);
}

var id = 21;

foo.call({ id: 42 });
// id: 42
```

setTimeout的参数是一个箭头函数，这个箭头函数的定义生效是在foo函数生成时，而它的真正执行要等到 100 毫秒后。如果是普通函数，执行时this应该指向全局对象window，这时应该输出21。

但是，箭头函数导致this总是指向函数定义生效时所在的对象（本例是{id: 42}），所以输出的是42。因为是在{id:42}这个对象创造的一个延时器，所以延时器里面的箭头函数指向的就是{id:42}

# 箭头函数的关键注意点

**（1）函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。**

```
var handler = {
  id: '123456',

  init: function() {
    document.addEventListener('click',
      event => this.doSomething(event.type), false);
  },

  doSomething: function(type) {
    console.log('Handling ' + type  + ' for ' + this.id);
  }
};
```

上面代码的init方法中，使用了箭头函数，这导致这个箭头函数里面的this，总是指向handler对象。（这个监听事件在init函数被执行的时候才会创造出来，所以监听事件里面的箭头函数也是在handler对象的init方法被执行时才会创立，那么自然这个箭头函数里面的this指向的就是handler对象）

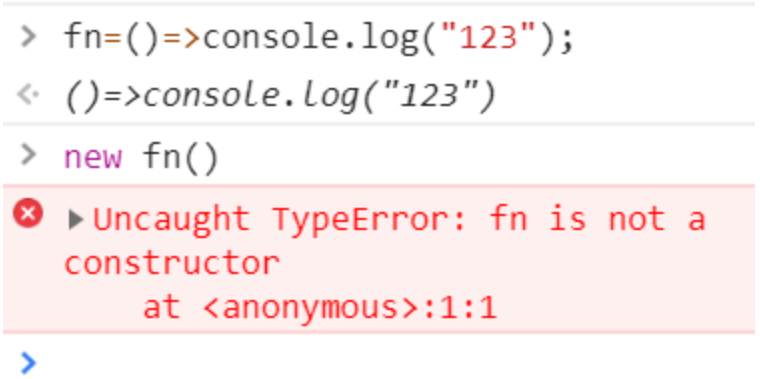
```
// ES6
function foo() {
    setTimeout(() => {
        console.log('id:', this.id);
    }, 100);
}
```

```
// ES5
function foo() {
  var _this = this;

  setTimeout(function () {
    console.log('id:', _this.id);
  }, 100);
}
```

this指向的固定化，并不是因为箭头函数内部有绑定this的机制，实际原因是箭头函数根本没有自己的this，导致内部的this就是外层代码块的this。正是因为它没有this，所以也就不能用作构造函数。

# 箭头函数的关键注意点

（2）不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。



# 箭头函数的关键注意点

（3）不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

```
function fn1(){
    console.log(arguments);
}

fn2=()=>console.log(arguments);
```

```
> fn1(1,2,3,4,5)
                                                    demo.html:177
  ▶ Arguments(5) [1, 2, 3, 4, 5, callee: ƒ, Symbol(Symbol.iterator): ƒ]
< undefined
> fn2(1,2,3,4,5)
  ▶ Uncaught ReferenceError:                        demo.html:180
  arguments is not defined
      at fn2 (file:///E:/%E5%B7%A5%E4%BD%9C%E6%96%87%E6%A1%A3/VIP%E8%AF%BE%E7%A8%8B/%E4%B8%87%E7%AB%A0%E8%AF%BE%E4%BB%B6/%E8%AF%BE%E7%A8%8B%E4%BA%8C(JavaScript%E8%AF%BE%E4%BB%B6)/%E7%AC%AC28%E8%AF%BE%EF%BC%9AES6%E8%BF%9B%E9%98%B6%E4%B9%8B%E5%87%BD%E6%95%B0%E7%B1%BB%E5%9E%8B%E6%8B%93%E5%B1%95/demo/demo.html:180:29)
      at <anonymous>:1:1
>
```

# 箭头函数的不适用场合

由于箭头函数使得this从“动态”变成“静态”，下面两个场合不应该使用箭头函数。

```
const cat = {
  lives: 9,
  jumps: () => {
    this.lives--;
  }
}
```

第一个场合是定义对象的方法，且该方法内部包括this。

cat.jumps()方法是一个箭头函数，这是错误的。调用cat.jumps()时，如果是普通函数，该方法内部的this指向cat；如果写成上面那样的箭头函数，使得this指向全局对象，因此不会得到预期结果。这是因为对象不构成单独的作用域，导致jumps箭头函数定义时的作用域就是全局作用域。

```
var button = document.getElementById('press');
button.addEventListener('click', () => {
  this.classList.toggle('on');
});
```

第二个场合是需要动态this的时候，也不应使用箭头函数。

上面代码运行时，点击按钮会报错，因为button的监听函数是一个箭头函数，导致里面的this就是全局对象。如果改成普通函数，this就会动态指向被点击的按钮对象。

# 箭头函数的嵌套使用(个人不建议，看起来还没传统模式可读性强ε=(´o｀*))))

```
var insert = function insert(value) {
    return {
        into: function into(array) {
            return {
                after: function after(afterValue) {
                    array.splice(array.indexOf(afterValue) + 1, 0, value);
                    return array;
                }
            };
        }
    };
};
```

```
let insert = (value) => ({
    into: (array) => ({
        after: (afterValue) => {
            array.splice(array.indexOf(afterValue) + 1, 0, value);
            return array;
        }
    })
});
```