function factorial(n) {

if (n === 1) return 1;

return n \* factorial(n - 1);

}

1. 需要保存n个调用记录的n的阶层， 复杂度O(n)；
2. “stack overflow”

尾调用优化

function factorial(n, total) {

if (n === 1) return total;

return factorial(n - 1, n \* total);

}

factorial(5, 1)

1. 只要保存一个调用记录，复杂度O(1);
2. 尾递归不需要保存内部变量、调用位置等信息，只需要保留内层函数的调用记录，每次执行调用记录只有一项；
3. 节省内存，无‘stack overflow’；
4. 注意： 只在严格模式下尾调用才生效，ES6才实现该机制。

柯里化（currying）

function currying(fn, n) {

return function (m) {

return fn.call(this, m, n);

};

}

function tailFactorial(n, total) {

if (n === 1) return total;

return tailFactorial(n - 1, n \* total);

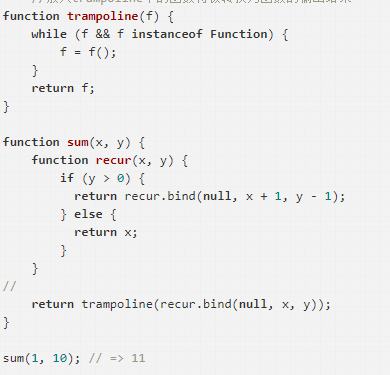
}

const factorial = currying(tailFactorial, 1);

多参数函数改为单参数函数。

https://stackoverflow.com/questions/25228871/how-to-understand-trampoline-in-javascript

<https://segmentfault.com/a/1190000004018047>



function tco(f) {

var value;

var active = false;

var accumulated = [];

return function accumulator() {

accumulated.push(arguments);

if (!active) {

active = true;

while (accumulated.length) {

value = f.apply(this, accumulated.shift());

}

active = false;

return value;

}

}

}

//这种方式确实有点奇怪，但的确没有改动很多源码，只是以直接量的形式使用tco函数包裹源码

var sum = tco(function(x, y) {

if (y > 0) {

return sum(x + 1, y - 1)

}

else {

return x

}

});

sum(1, 10) // => 11

sum(1, 100000) // => 100001 没有造成栈溢出