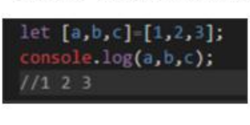
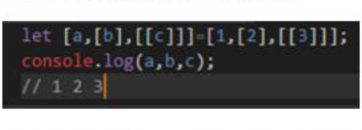
数组的解构赋值

1.简单的赋值方式



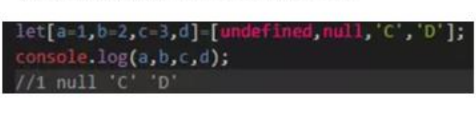
```
let [a,b,c]=[1,2,3];
console.log(a,b,c);
//1 2 3
```

2.多维数组解构赋值



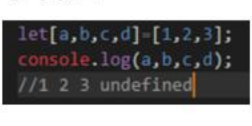
```
let [a,[b],[[c]]]=[1,[2],[[3]]];
console.log(a,b,c);
// 1 2 3
```

3.默认值，只有当右边对应位置为undefined时候才会选择默认（null不属于undefined）



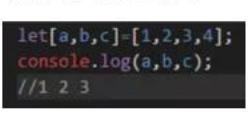
```
let[a=1,b=2,c=3,d]=[undefined,null,'C','D'];
console.log(a,b,c,d);
//1 null 'C' 'D'
```

4.左右不对等,会相应的对号入座，没有的以undefined赋值左边多于右边



```
let[a,b,c,d]=[1,2,3];
console.log(a,b,c,d);
//1 2 3 undefined
```

右边多余左边



```
let[a,b,c]=[1,2,3,4];
console.log(a,b,c);
//1 2 3
```

对象赋值

1.普通赋值，对象右边的顺序可以打乱

```
let{foo,bar}={foo:'zhangSan',bar:'liSi'};
console.log(foo,bar);
//zhangSan liSi
let{name,job}={job:'science',name:'xiaoMing'};
console.log(name,job);
//xiaoMing science
```

2.默认值赋值，同数组

```
let{foo=2,bar=1}={bar:'liSi'};
console.log(foo,bar);
//2 liSi
```

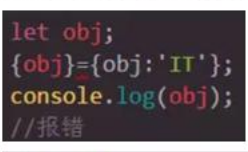
3.变量名和属性名不一致

```
let{obj:name}={obj:'xiaoWang'};
console.log(name);
//xiaoWang
let ob={name:"sanMao",job:'science'};
let {name:N,job:J}=ob;
console.log(N,J);
//sanMao science
```

也就是说，对象的解构赋值的内部机制，是先找到同名属性，然后再赋给对应的变量。真正被赋值的是后者，而不是前者

4.圆括号的用法

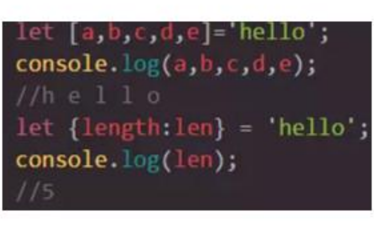
如果在解构之前就已经定义了对象，解构需要加圆括号



```
let obj;
{obj}={obj:'IT'};
console.log(obj);
//报错
```

```
let obj;
({obj}={obj:'IT'});
console.log(obj);
//IT
```

字符串的解构



```
let [a,b,c,d,e]='hello';
console.log(a,b,c,d,e);
//h e l l o
let {length:len} = 'hello';
console.log(len);
//5
```

为了程序的易读性，我们会使用 ES6 的解构赋值：

function f({a,b}){}

f({a:1,b:2});复制代码

这个例子的函数调用中，会真的产生一个对象吗？如果会，那大量的函数调用会白白生成很多有待 GC 释放的临时对象，那么就意味着在函数参数少时，还是需要尽量避免采用解构传参，而使用传统的：

function f(a,b){}

f(1,2);

上面的描述其实同时提了好几个问题：

1. 会不会产生一个对象？
2. 参数少时，是否需要尽量避免采用解构传参？
3. 对性能(CPU/内存)的影响多大？

1. 从 V8 字节码分析两者的性能表现

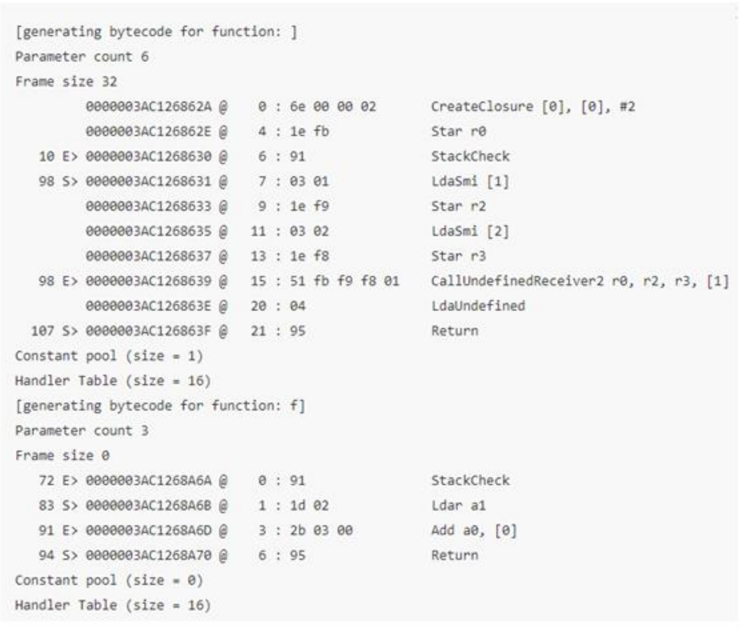
首先从上面给的代码例子中，确实会产生一个对象。但是在实际项目中，有很大的概率是不需要产生这个临时对象的。那么我们就分析一下示例代码。

function f(a,b){ return a+b; } const d = f(1, 2);

鉴于很多人没有 V8，因此我们使用 node.js 代替。运行：(以下内容仅作为辅助理解当前示例)

node --print-bytecode add.js

其中的 --print-bytecode 可以查看 V8 引擎生成的字节码。在输出结果中查找 [generating bytecode for function: f]：



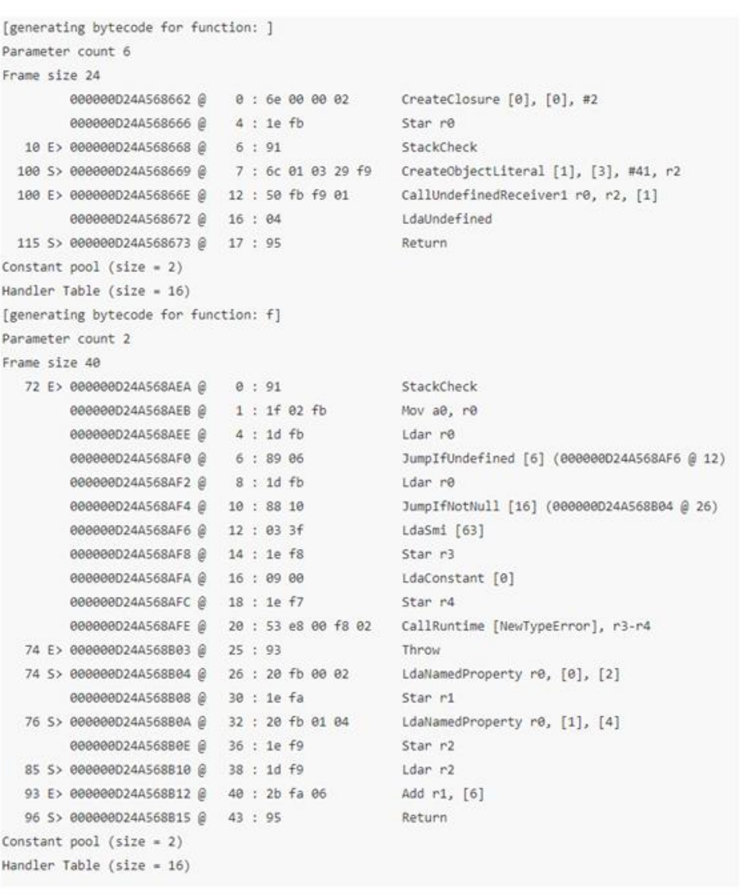
```
[generating bytecode for function: ]
Parameter count 6
Frame size 32
         0000003AC126862A @    0 : 6e 00 00 02       CreateClosure [0], [0], #2
         0000003AC126862E @    4 : 1e fb             Star r0
   10 E> 0000003AC1268630 @    6 : 91                StackCheck
   98 S> 0000003AC1268631 @    7 : 03 01             LdaSmi [1]
         0000003AC1268633 @    9 : 1e f9             Star r2
         0000003AC1268635 @   11 : 03 02             LdaSmi [2]
         0000003AC1268637 @   13 : 1e f8             Star r3
   98 E> 0000003AC1268639 @   15 : 51 fb f9 f8 01    CallUndefinedReceiver2 r0, r2, r3, [1]
         0000003AC126863E @   20 : 04                LdaUndefined
  107 S> 0000003AC126863F @   21 : 95                Return
Constant pool (size = 1)
Handler Table (size = 16)
[generating bytecode for function: f]
Parameter count 3
Frame size 0
   72 E> 0000003AC1268A6A @    0 : 91                StackCheck
   83 S> 0000003AC1268A6B @    1 : 1d 02             Ldar a1
   91 E> 0000003AC1268A6D @    3 : 2b 03 00          Add a0, [0]
   94 S> 0000003AC1268A70 @    6 : 95                Return
Constant pool (size = 0)
Handler Table (size = 16)
```

Star r0 将当前在累加器中的值存储在寄存器 r0 中。

LdaSmi [1] 将小整数（Smi）1 加载到累加器寄存器中。

而函数体只有两行代码：Ldar a1 和 Add a0, [0]。

当我们使用解构赋值后：

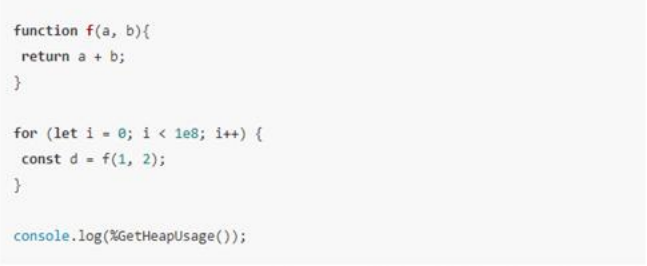


```
[generating bytecode for function: ]
Parameter count 6
Frame size 24
         000000D24A568662 @    0 : 6e 00 00 02       CreateClosure [0], [0], #2
         000000D24A568666 @    4 : 1e fb             Star r0
   10 E> 000000D24A568668 @    6 : 91                StackCheck
  100 S> 000000D24A568669 @    7 : 6c 01 03 29 f9    CreateObjectLiteral [1], [3], #41, r2
  100 E> 000000D24A56866E @   12 : 50 fb f9 01       CallUndefinedReceiver1 r0, r2, [1]
         000000D24A568672 @   16 : 04                LdaUndefined
  115 S> 000000D24A568673 @   17 : 95                Return
Constant pool (size = 2)
Handler Table (size = 16)
[generating bytecode for function: f]
Parameter count 2
Frame size 40
   72 E> 000000D24A568AEA @    0 : 91                StackCheck
         000000D24A568AEB @    1 : 1f 02 fb          Mov a0, r0
         000000D24A568AEE @    4 : 1d fb             Ldar r0
         000000D24A568AF0 @    6 : 89 06             JumpIfUndefined [6] (000000D24A568AF6 @ 12)
         000000D24A568AF2 @    8 : 1d fb             Ldar r0
         000000D24A568AF4 @   10 : 88 10             JumpIfNotNull [16] (000000D24A568B04 @ 26)
         000000D24A568AF6 @   12 : 03 3f             LdaSmi [63]
         000000D24A568AF8 @   14 : 1e f8             Star r3
         000000D24A568AFA @   16 : 09 00             LdaConstant [0]
         000000D24A568AFC @   18 : 1e f7             Star r4
         000000D24A568AFE @   20 : 53 e8 00 f8 02    CallRuntime [NewTypeError], r3-r4
   74 E> 000000D24A568B03 @   25 : 93                Throw
   74 S> 000000D24A568B04 @   26 : 20 fb 00 02       LdaNamedProperty r0, [0], [2]
         000000D24A568B08 @   30 : 1e fa             Star r1
   76 S> 000000D24A568B0A @   32 : 20 fb 01 04       LdaNamedProperty r0, [1], [4]
         000000D24A568B0E @   36 : 1e f9             Star r2
   85 S> 000000D24A568B10 @   38 : 1d f9             Ldar r2
   93 E> 000000D24A568B12 @   40 : 2b fa 06          Add r1, [6]
   96 S> 000000D24A568B15 @   43 : 95                Return
Constant pool (size = 2)
Handler Table (size = 16)
```

我们可以看到，代码明显增加了很多，CreateObjectLiteral 创建了一个对象。本来只有 2 条核心指令的函数突然增加到了近 20 条。其中不乏有 JumpIfUndefined、CallRuntime 、Throw 这种指令。

2. 使用 --trace-gc 参数查看内存

由于这个内存占用很小，因此我们加一个循环。

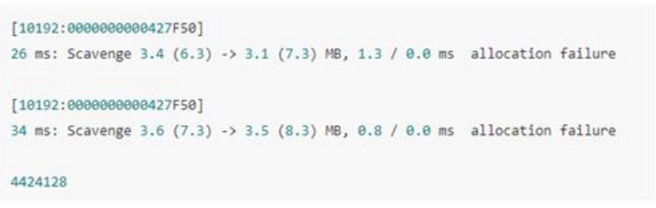


```
function f(a, b){
 return a + b;
}

for (let i = 0; i < 1e8; i++) {
 const d = f(1, 2);
}

console.log(%GetHeapUsage());
```

%GetHeapUsage() 函数有些特殊，以百分号(%)开头，这个是 V8 引擎内部调试使用的函数，我们可以通过命令行参数 --allow-natives-syntax 来使用这些函数。

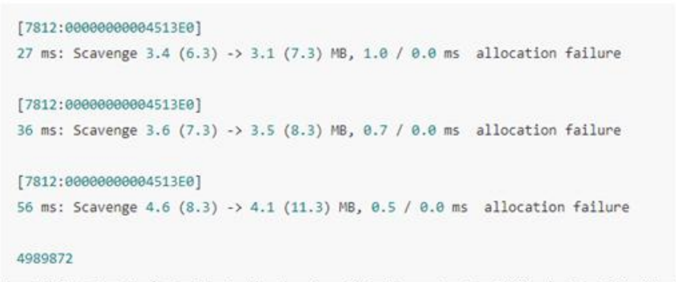
node --trace-gc --allow-natives-syntax add.js

得到结果（为了便于阅读，我调整了输出格式）：



```
[10192:0000000000427F50]
26 ms: Scavenge 3.4 (6.3) -> 3.1 (7.3) MB, 1.3 / 0.0 ms  allocation failure

[10192:0000000000427F50]
34 ms: Scavenge 3.6 (7.3) -> 3.5 (8.3) MB, 0.8 / 0.0 ms  allocation failure

4424128
```

当使用解构赋值后：



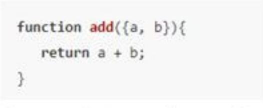
```
[7812:00000000004513E0]
27 ms: Scavenge 3.4 (6.3) -> 3.1 (7.3) MB, 1.0 / 0.0 ms  allocation failure

[7812:00000000004513E0]
36 ms: Scavenge 3.6 (7.3) -> 3.5 (8.3) MB, 0.7 / 0.0 ms  allocation failure

[7812:00000000004513E0]
56 ms: Scavenge 4.6 (8.3) -> 4.1 (11.3) MB, 0.5 / 0.0 ms  allocation failure

4989872
```

可以看到多了因此内存分配，而且堆空间的使用也比之前多了。使用 --trace_gc_verbose 参数可以查看 gc 更详细的信息，还可以看到这些内存都是新生代，清理起来的开销还是比较小的。

3. Escape Analysis 逃逸分析

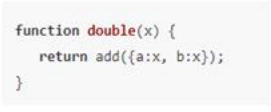
通过逃逸分析，V8 引擎可以把临时对象去除。

还考虑之前的函数：



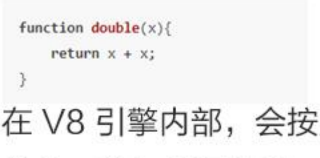
```
function add({a, b}){
   return a + b;
}
```

如果我们还有一个函数，double，用于给一个数字加倍。



```
function double(x) {
   return add({a:x, b:x});
}
```

而这个 double 函数最终会被编译为



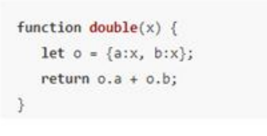
```
function double(x){
    return x + x;
}
```

在 V8 引擎内部，会按照如下步骤进行逃逸分析处理：

首先，增加中间变量：

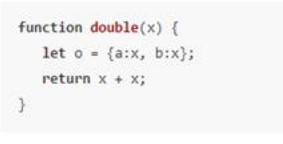
```
function add(o){
 return o.a + o.b;
}

function double(x) {
   let o = {a:x, b:x};
   return add(o);
}
```

把对函数 add 的调用进行内联展开，变成：



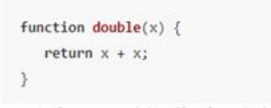
```
function double(x) {
   let o = {a:x, b:x};
   return o.a + o.b;
}
```

替换对字段的访问操作：



```
function double(x) {
   let o = {a:x, b:x};
   return x + x;
}
```

删除没有使用到的内存分配：



```
function double(x) {
   return x + x;
}
```

通过 V8 的逃逸分析，把本来分配到堆上的对象去除了。

4. 结论

不要做这种语法层面的微优化，引擎会去优化的，业务代码还是更加关注可读性和可维护性。如果你写的是库代码，可以尝试这种优化，把参数展开后直接传递，到底能带来多少性能收益还得看最终的基准测试。

举个例子就是 Chrome 49 开始支持 Proxy，直到一年之后的 Chrome 62 才改进了 Proxy 的性能，使 Proxy 的整体性能提升了 24% ~ 546%。