Kyle Owen

ES6 Tail Call Optimization Explained

19 Jul 2015

From Wikipedia: “A tail call is a subroutine call performed as the final action of a procedure.”

```
return foo();
```

A tail call optimization may occur when the last thing to evaluate before a function returns is a function invocation. In certain circumstances, the interpreter can the reuse current stack frame for the function call instead of creating a new one. Below I will explain what circumstances are necessary and why this is an optimization.

Interpreter

Before we begin, it’s important to understand that the ES6 tail call optimization is an optimization implemented by the interpreter. ES6 does not specify new syntax for denoting tail call optimization, so don’t continue reading expecting to see any new JS syntax. Instead, pay attention to how the code is structured.

To talk about the tail call optimization, it’ll be useful to discuss different ways to calculate fibonacci numbers, and the difference between a recursive process and a recursive procedure. Although tail call optimizations can occur in non-recursive functions, the fibonacci examples below are useful for understanding some situations where the optimization will occur. You may have seen some of this

before, but I promise - it'll be worth the read.

Fibonacci

Let's first calculate fibonacci numbers using a recursive process:

```
function fib(n) {
  if (n <= 1){
    return n;
  } else {
    return fib(n-1) + fib(n - 2);
  }
}
```

Below is a diagram of how this process will develop - each node in this tree represents a function call to fib.

```
/*
                              fib(5)
                             /      \
                            /        \
                       fib(4)          fib(3)
                      /     \           /   \
                     /       \         /     \
                fib(3)         fib(2)    .........
                /  \            /  \
               /    \          /    \
          fib(2)  fib(1)  fib(1)  fib(0)
          /   \
         /     \
    fib(1)  fib(0)

*/
```

Notice, none of these nodes has any clue that they are part of a huge process to calculate fib(5). For example, the calls to fib(2) have no clue that they are part of a process to calculate the value of fib(5).

Also, none of the nodes on this tree have enough information to encapsulate the entire state of the process. What I mean is, looking at the local variables inside any one of the environments these calls are running in, it'd be

impossible to determine what point in the process for calculating fib(5) you were at.

So, what is keeping track of this information? What keeps track of how each of these nodes should be combined? The call stack.

The call stack keeps track of how each call to fib must be combined after their calculations complete. The call stack is the one remembering to combine the calls of fib(3) and fib(2) to get fib(4), and the calls of fib(4) and fib(3) to get fib(5).

If we were to stop the interpreter partway through the process and resume execution in a new environment with a clean call stack, there would be no way to complete the process, since all of the information about how to combine recursive calls would have been lost. **Think of a recursive process as a series of deferred operations, where there is information hidden to each recursive call - that hidden information is in the call stack.**

When executing this in a browser, there are a bunch of stack frames which are created, with new environments for each call. You can see this in your browser's debugger.

Fibonacci another way

Besides the fact that the algorithm above repeats a lot of calculations unnecessarily (fib(1) is calculated many times), the algorithm has O(n) memory complexity for the call stack. Looking at the bottom of left branch of the fib tree above, when calculating fib(1) the callstack has 5 frames on it - the calls to fib 5 to 1 each remembering how to combine with the other calls. When fib(1) completes, that stack frame is popped off the call stack (so fib(2) would be at the top of the stack), and then another stack frame gets added to calculated fib(0). So for the fib implementation above, the call stack will have at most n stack frames at any given time. O(n) memory complexity.

Now, here is an iterative way to calculate fibonacci sequence:

```
function fibIter(n){
  var a = 1, b = 0, temp;

  while (n > 0){
    temp = a;
    a = a + b;
    b = temp;
    n--;
  }

  return b;
}

/*

To picture this, imagine a,b moving
along the sequence as such:

b,a
0,1,1,2,3,5 ....

  b,a
0,1,1,2,3,5 ....

    b,a
0,1,1,2,3,5 ....

etc...

*/
```

This function will only use one stack frame on the call stack since there are no other function calls. Now, what if we define this same process recursively?

```
function fibIterRecursive(n, a, b){
  if (n === 0) {
    return b;
  } else {
    return fibIterRecursive(n-1, a + b, a);
  }
};

function fib(n){
  return fibIterRecursive(n, 1, 0);
}
```

Calculating fib(5), here is what the calls will look like

```
fib(5)
fibIterRecursive(5, 1, 0)
fibIterRecursive(4, 1, 1)
fibIterRecursive(3, 2, 1)
fibIterRecursive(2, 3, 2)
fibIterRecursive(1, 5, 3)
fibIterRecursive(0, 8, 5)
```

In this implementation, the entire state of the process **is** encapsulated in each function call. Notice, each call has enough information to complete the process for calculating fib(5). If we stopped that process at fibIterRecursive(2, 3, 2) and then resumed it in a different environment with a clean call stack, we would still get the correct number. This is in contrast to the recursive process' nodes from above, which had no clue how they were being combined (the interpreter kept track of that).

We can call the fibIterRecursive implementation a recursive procedure - a function that calls itself, but does not have hidden information which the interpreter needs to keep track of.

The Optimization

The fibIterRecursive function mirrors the fibIter function from above, so one would hope that the number of stack frames in each process would be the same. Ideally, when calculating fibIterRecursive, the call stack would not bother remembering a bunch of information in stack frames, since that would be unnecessary. But, it still does in ES5 - for each call to fibIterRecursive, a new stack frame is created. That means that in ES5, we still have O(n) memory complexity. **In ES6, new stack frames will not be created thus allowing for a O(1) memory complexity - and that is the optimization**. Let's see how that works...

In ES6, using the same exact code as fibIterRecursive, the interpreter will notice a few things:

- the last thing that needs to evaluate before the return statement is a function call (see next point)
- there is no action after the function call that the interpreter needs to remember to do (like combining fib(n-1) and fib(n-2) in the first example)
- the next recursive call does not need access to any of the local variables in the current stack frame

So, instead of creating a new stack frame for each call, the current stack frame will be cleared and reused. This is great because, in es5, calling fibIterRecursive(30000) will give you a stack overflow due to the O(n) memory complexity, but in es6 with the optimization, the stack frame will be reused and it won't cause a stack overflow (however, without a bigInt library, fibIterRecursive(30000) would return Infinity because of JS number limits).

To summarize, if the last thing that happens before the return statement is the invocation of a function which does not need to access any of the current local variables, the interpreter specified by ES6 will optimize that call by reusing the stack frame.
Something important to notice: this optimization will not apply to **recursive processes** as defined above (since they are characterized by interpreters keeping track of hidden information). The optimization will apply to **recursive procedures** as defined above, and other non-recursive processes. Here are some other examples:

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

// In sumTwoNumbers, the call to add will be opti
mized:

function sumTwoNumbers(a,b){
  return add(a,b);
```

```
}

// But in sumFourNumbers:

function sumFourNumbers(a,b,c,d){
  return add(a,b) + add(c,d);
}

// the optimization won't happen because the
// interpreter needs to remember to add together
// the values returned by add(1,2) and add(3,4).
// The last thing to happen before the return
// is not a function invocation.

// here's another example where the optimization
won't happen:

function addOne(a){
  var one = 1;
  function inner(b){
    return b + one;
  }
  return inner(a);
}

// the optimization won't happen because addToOne
needs to
// access the variable 'one' from addOne's enviro
ment
```

It's an interpreter thing. Note: you must be in strict mode for this optimization to work. Here is a link to the ES6 compatibility to see where the optimization has been implemented. Other resources for understanding tail calls that I've found helpful include: this and that. Also, my favorite resource for understanding recursion: SICP

Note: there is a creative way to get around this in es5. It's not used often. If you're interested check out this great blog post.

https://kangax.github.io/compat-table/es6/

http://www.2ality.com/2015/06/tail-call-optimization.html

http://raganwald.com/2015/02/07/tail-calls-defult-arguments-recycling.html

https://mitpress.mit.edu/sicp/

http://raganwald.com/2013/03/28/trampolines-in-javascript.html

Related Posts