

Use JavaScript closures efficiently

Avoid memory leaks in your Node.js apps

Michael Dawson

Software Developer, IBM
IBM Ottawa Lab

02 May 2016

Gireesh Punathil

Software Engineer, IBM
IBM

Understand the characteristics of JavaScript closures and their lifecycles so that you can prevent memory leaks in your Node.js applications.

Closures are used in Node.js pervasively in various forms to support Node's asynchronous and event-driven programming model. By having a good understanding of closures, you can ensure the functional correctness, stability, and scalability of the applications that you develop.

Closures are a natural way of associating data with the code that acts on the data, with *continuation passing* as the main semantic style. When you use closures, data elements that you define within an enclosing scope become accessible to a function that's created in the scope, even when the enclosing scope is logically exited. In languages (such as JavaScript) in which functions are first-class variables, this behavior is of great importance, because the lifecycle of functions determines the lifecycle of the data elements that are visible to the functions. In this environment, it's dangerously easy to inadvertently retain much more data in memory than you expect.

This tutorial examines three of the main use cases in which closures are used in Node:

- Completion handler
- Intermediary function
- Listener function

For each use case, we provide sample code and indicate both the closure's expected lifespan and the memory that's retained during that lifespan. This information can help you design your JavaScript applications with insight into how these uses cases affect memory usage, so that you can avoid memory leaks in your apps.

READ: [Understand memory leaks in JavaScript applications](#)

READ: [Memory leak patterns in JavaScript](#)

READ: [Core dump debugging for the IBM SDK for Node.js](#)

Closures and asynchronous programming

If you are familiar with traditional sequential programming, you might ask the following questions when first trying to understand the asynchronous model:

- If a function is called asynchronously, how do you make sure that the code following it (or around it) can work with the data that's available in the scope when the call is made? Or in other words, how do you implement the rest of the code, which depends on the results and side effects of the asynchronous call?
- After the asynchronous call is made, and the program continues to execute code that's unrelated to the asynchronous call, how do you later return to the original calling scope to carry on after the asynchronous call is complete?

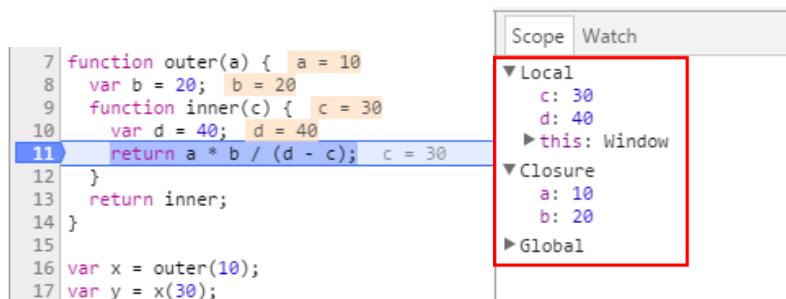
Closures and callbacks are the answer to these questions. In the most common and simplest use case, an asynchronous method takes a callback method (which has an associated closure) as one of its arguments. This function is often defined in-line at the call site for the asynchronous method, and the function has access to the data elements (local variables and parameters) of the scope that encloses the call site.

As an example, take the following JavaScript code:

```
function outer(a) {
  var b = 20;
  function inner(c) {
    var d = 40;
    return a * b / (d - c);
  }
  return inner;
}

var x = outer(10);
var y = x(30);
```

Here's a snapshot of the same code in a live debugging session:



The `inner` function is called at line 17 (line 11 in the preceding listing) and executes at line 11 (the listing's line 5). At line 16 (the listing's line 10), the `outer` function — which returns the `inner`

function — is called. As shown in the screenshot, when the `inner` function is called at line 17, and executes at line 11, it has access to both of its local variables (`c` and `d`) **and** the variables defined in the `outer` function (`a` and `b`) — even though the scope for the `outer` function was exited when the call to the `outer` function completed at line 16.

“ To avoid memory leaks, it's important to understand when and for how long the callback method remains reachable. ”

The callback method is in a state in which it can be invoked (that is, is reachable from a garbage-collection point of view), so it keeps alive all of the data elements to which it has access. To avoid memory leaks, it's important to understand when and for how long the callback method remains in that state.

At a high level, closures commonly come into play in at least three kinds of use cases. In all three, the fundamental premise is the same: the ability for a small piece of reusable code (a callable function) to work with, and optionally retain, a context.

Use case 1: Completion handler

In the completion handler pattern, a function (`c1`) is passed as an argument to a method (`m1`), and `c1` is invoked as a completion handler when `m1` is complete. As part of the pattern, the implementation for `m1` ensures that the reference that it retains to `c1` is cleared after it's no longer needed. `c1` often needs one or more of the data elements in the scope where `m1` was called. The closure that provides access to this scope is defined when `c1` is created. One common approach is to use an anonymous method defined in-line where the `m1` is called. The result is a closure for `c1` that provides access to all of the variables and parameters that are available to `m1`.

An example is the `setTimeout()` method. When the timer expires, the completion function is invoked, and the reference to the completion function (`c1`) that's retained for the timer is cleared:

```
function CustomObject() {  
}  
  
function run() {  
    var data = new CustomObject()  
    setTimeout(function() {  
        data.i = 10  
    }, 100)  
}  
run()
```

The completion function uses the `data` variable from the context in which the `setTimeout` method was called. Even after the `run()` method has completed, `CustomObject` can be referenced by the closure that's created for the completion handler and isn't garbage-collected.

Memory retention

The closure context is created when the completion function (`c1`) is defined, and the context consists of the variables and the parameters that are accessible in the scope where `c1` is created. The closure for `c1` is retained until both:

- The completion method is invoked and finishes running **or** the timer is cleared.
- No other references to `c1` occur. (In the case of anonymous functions, no other references occur if the preceding condition in this list is met.)

By using the [Chrome developer tools](#), we can see that the `Timeout` object that represents our timer has a reference to the completion function (the anonymous method passed to `setTimeout`) through the `_onTimeout` field:

Summary	Class filter	All objects	Distance	Object...	Shallow Size	Retained S...
Constructor			5	0 %	210	0 %
Module			4	0 %	210	0 %
Timeout			4	72	0 %	72
Timeout @24779			4	72	0 %	72
_onTimeout :: <i>function ()</i> @24775			5	36	0 %	36
▶ shared :: (shared function info) @14299			6	104	0 %	104
▶ data :: CustomObject @24765			6	52	0 %	52
▶ __proto__ :: <i>function Empty()</i> @179			3	36	0 %	36
▶ _idleNext :: Timer @23563			3	16	0 %	16
▶ _idlePrev :: Timer @23563			3	16	0 %	16
▶ _idleStart :: number @24781			5	12	0 %	12
▶ __proto__ :: @23599			4	12	0 %	12
▶ Timeout @24843			4	72	0 %	72
InternalArray			3	0 %	96	0 %
Agent			7	0 %	64	0 %

While the timer is in effect, the `Timeout` object, the `_onTimeout` field, and the closure function are all held in the heap through one single reference to them — the timeout event that's pending in the system. When the timer fires, and the subsequent callback completes, the pending event in the event loop is removed. All three objects are no longer accessible and become eligible for collection in the subsequent garbage-collection cycle.

When the timer is cleared (through the `clearTimeout` method), the completion function is removed from the `_onTimeout` field and — provided that no other references to the function occur — it can be collected in the subsequent garbage-collection cycle, even though the `Timeout` object is retained because the main program holds a reference to it.

In this screenshot, [heap dumps](#) taken before and after the timer fires are compared:

Comparison	CustomObject	heapprof-548192848.501405					
Constructor		# New	# Deleted	# Delta	Alloc. Size	Freed Size	Size Delta
CustomObject		0	1	-1	0	12	-12
▶ CustomObject @23029			.			12	

The `#New` column shows new objects that were added between the dumps, and the `#Deleted` column shows objects that were collected between the dumps. The highlighted section shows that the `CustomObject` was present in the first dump but is collected and isn't available in the second dump, freeing 12 bytes of memory.

In this pattern, the natural flow of execution is such that memory is retained only until the completion handler (`c1`) finishes its job of "completing" the method (`M1`). The result is that — provided that the completion of the methods called by the application occurs in a timely manner — you don't need to take special care to avoid memory leaks.

When you design functions that implement this pattern, ensure that any references to callback functions are cleared when the callback is fired. In this way, you ensure that in terms of memory retention, expectations of applications that use your functions are met.

Use case 2: Intermediary function

In some cases, you need to be able to work with data in a more repetitive, iterative, and out-of-bounds manner, whether that data is created in an asynchronous or synchronous manner. For such cases, you can return an *intermediary function*, which can be called one or more times to either access the required data or complete the required computation. As with the completion handlers, you create the closure when you define the function, and the closure provides access to all of the variables and parameters that are available in the scope where the function was defined.

One example of this pattern is data streaming, wherein a server returns a large chunk of data and the client's data receiver callback is invoked for every chunk of data that arrives. Because the data flow is asynchronous, the operation (such as accumulation of data) must be iterative and occur in an out-of-bounds manner. The following program illustrates this scenario:

```
function readData() {
  var buf = new Buffer(1024 * 1024 * 100)
  var index = 0
  buf.fill('g') //simulates real read

  return function() {
    index++
    if (index < buf.length) {
      return buf[index-1]
    } else {
      return ''
    }
  }
}

var data = readData()
var next = data()
while (next !== '') {
  // process data()
  next = data()
}
```

In this case, `buf` is retained for as long as the `data` variable is still in scope. The size of the `buf` buffer causes a large amount of memory to be retained, even though this might not be obvious to the application developer. We can see this effect by using the Chrome developer tools, as shown in a snapshot taken after the `while` loop is completed: The large buffer is being retained, even though it will no longer be used.

Retainers	Object	Distance	Shallow Size	Retained Size	
▼native in Buffer @843		-	72	0 %	84 0 %
▼buf in system / Context @26275		-	32	0 %	32 0 %
▼context in <i>function</i> () @569		-	36	0 %	68 0 %
▼[31] in (Isolate) @17		-	0	0 %	1 612 0 %
[4] in (GC roots) @3		-	0	0 %	290 132 0 %
▼[1] in smalloc @108329684		-	104 857 600	98 %	104 857 600 98 %
▼native in Buffer @843		-	72	0 %	84 0 %
▼buf in system / Context @26275		-	32	0 %	32 0 %
▼context in <i>function</i> () @569		-	36	0 %	68 0 %
▼[31] in (Isolate) @17		-	0	0 %	1 612 0 %
[4] in (GC roots) @3		-	0	0 %	290 132 0 %

Memory retention

Even after the application finishes using the intermediary function, the reference to the function keeps the associated closure alive. To allow the data to be collected, the application must overwrite this reference — for example, by setting the reference to the intermediary function as follows:

```
// Manual cleanup
data = null;
```

This code allows the closure context to be garbage-collected. The following screenshot from a heap dump, taken after data was set to null, shows that the manual nullification allows garbage collection of the retained data:

Comparison ▼	Buffer	heapprof-550138911.396123 ▼					
Constructor	# N...	# D...	# Delta	Alloc. Size ▼	Freed Size	Size Delta	
► SlowBuffer	3	4	-1	44	60	-16	
► ArrayBuffer	1	1	0	12	12	0	
► Buffer	1	3	-2	12	104 857 708	-104 857 696	
► writeBuffer	1	0	+1	12	0	+12	

The highlighted line indicates that the buffer has been collected and its associated memory freed.

It's often possible to construct intermediary functions so that they limit the potential memory leaks. For example, an intermediary that allows incremental reading of a large data set could remove references to the portions of the data that were returned. But in these cases, it's important that this approach not pose a concern for other parts of the application that might have access to that data in a manner other than through the intermediary function.

When creating APIs that implement the intermediary pattern, be careful to document the memory-retention characteristics so that consumers understand the need to ensure that all references are nullified. Even better, if possible implement your API such that the data retained can be released when it's no longer required within the intermediary function.

For example, the function from the previous example in this section could be rewritten as:

```
return function() {
  index++;
  if (index < buf.length) {
    return buf[index-1]
  } else {
    buf = null
    return
  }
}
```

This version ensures that after the large buffer is no longer required, it can be collected.

Use case 3: Listener functions

A common pattern is to register functions that listen for the occurrence of a particular event. Though extremely handy in asynchronous programming, this registration causes the listener function to "escape" into the event emitter's internal cache along with any associated closure context. This is fine as long as the events are being generated and processed, and the event interceptor module knows when to stop event emission and clear the listeners. The risk arises when the lifecycle for a listener function becomes indefinite or unknown by the application. As such, listener functions are most at risk of causing a memory leak.

“Listener functions are most at risk of causing a memory leak.”

Most streaming/buffering scenarios use this mechanism to cache or accumulate instantaneous data that's defined in an outer method, while the access is done in a closure function. Consider this example:

```
var EventEmitter = require('events').EventEmitter
var ev = new EventEmitter()

function run() {
  var buf = new Buffer(1024 * 1024 * 100)
  var index = 0
  buf.fill('g')
  ev.on('readNext', function nextReader() {
    var byte = buf[index]
    index++
    process(byte)
  });
}
```

Memory retention

The following screenshot, taken after the call to the `run()` method, shows how the memory is retained for the large buffer, `buf`. As you can see through the dominator tree, the large buffer is kept alive because of its association with the event:

▼[1] in Buffer @828792428	9	104 857 600	99 %	104 857 600	99 %
▼native in SlowBuffer @20651	8	16	0 %	16	0 %
▼parent in Buffer @20649	7	96	0 %	96	0 %
▼buf in function nextReader() @20635	6	36	0 %	36	0 %
▼readNext in @20627	5	12	0 %	12	0 %
▼_events in EventEmitter @20507	4	60	0 %	60	0 %
▼ev in function () @20967	3	36	0 %	36	0 %
▼[24] in (Isolate) @15	2	0	0 %	204	0 %
[4] in (GC roots) @3	1	0	0 %	154 976	0 %

The data retained by a callback function (listener) is held alive, even after all of the data is read, not to mention the fact that outer function has long returned. Because the function (nextReader) accesses the buffer, a closure context is created and maintained in the runtime, which will be cleared only when the function is detached from the event and no longer referenced.

```
// Once the purpose of 'readNext' event is met,
// remove the event from the Emitter
ev.removeListener('readNext', nextReader);
```

The risk arises when you install listeners whose lifecycle is:

1. Neither in your control nor fully known.
2. Under your supervision and control, but the closure is anonymous, making it infeasible to reference and remove.

While the Node core libraries implement a number of event emitters (such as streams) and their consumers (such as http), their lifecycle definitions are well known. Similarly, when designing applications and library modules with event emitters and event consumers, you should take the time to understand the behaviors described above, and ensure that the lifecycle of the events is well defined and that they can be de-registered when no longer needed.

“When registering functions as listeners, make sure that they will be de-registered at an appropriate lifecycle phase of the application.”

A well-known example of this use case is a typical HTTP server implementation:

```
var http = require('http');

function runServer() {
  /* data local to runServer, but also accessible to
   * the closure context retained for the anonymous
   * callback function by virtue of the lexical scope
   * in the outer enclosure.
   */
  var buf = new Buffer(1024 * 1024 * 100);
  buf.fill('g');

  http.createServer(function (req, res) {
    res.end(buf);
  }).listen(8080);

}
```

Although this example shows a handy way to use the inner functions, note that the callback function — and hence the buffer object — are alive for as long as the server object is alive. Only when the server closes does the object become eligible to be collected. You can see in the following screenshot that the buffer is kept alive because of its use by the server request listener:

Constructor	Distan...	Object...	Shallow Size	Retained Size	▼
▶ <code>function () @14359</code>	4		36	0 %	36
▶ <code>shared :: (shared function info) @14295</code>	5		104	0 %	104
▶ <code>buf :: Buffer @24745</code>	5		96	0 %	96
▶ <code>parent :: SlowBuffer @24747</code>	6		16	0 %	16
▶ <code>native :: Buffer @631786236</code>	7		104 857 600	99 %	104 857 600
▶ <code>properties :: (object properties)[] @24751</code>	7		20	0 %	20
▶ <code>elements :: (object elements)[] @24753</code>	7		12	0 %	12
▶ <code>__proto__ :: SlowBuffer @7409</code>	4		12	0 %	12
▶ <code>elements :: (object elements)[] @24749</code>	6		12	0 %	12
▶ <code>__proto__ :: Buffer @19019</code>	3		12	0 %	12
▶ <code>__proto__ :: function Empty() @179</code>	3		36	0 %	36
▶ <code>function () @17023</code>	5		36	0 %	36
▶ <code>function () @17025</code>	5		36	0 %	36

The lesson is that for any listener that retain a substantial amount of data, you need to understand and document the required lifespan for the listener and ensure that the listener is deregistered when it's no longer necessary. It's also advisable to ensure that listeners retain the minimum amount of data possible across invocations, because of their typically long lifetimes.

Conclusion

Closures are powerful programming constructs for achieving binding between code and data in a more flexible and out-of-bounds manner. However, the scoping semantics can be unfamiliar to programmers who are used to older languages such as Java or C++. To avoid memory leaks, it's important to understand the characteristics of closures and their lifecycles.

RELATED TOPICS: [Web development](#) [Node.js](#)

© Copyright IBM Corporation 2016

(www.ibm.com/legal/copytrade.shtml)

[Trademarks](#)

(www.ibm.com/developerworks/ibm/trademarks/)