
build unknown npm package 0.1.13

TAGG: Threads à gogo for Node.js

Threads à gogo (*) is a native module for Node.js that provides an asynchronous, evented and/or continuation passing style API for moving blocking/longish CPU-bound tasks out of Node's event loop to JavaScript threads that run in parallel in the background and that use all the available CPU cores automatically; all from within **a single Node** process.

Installing the module

With npm:

```
1 npm install threads_a_gogo
```

From source:

```
1 git clone http://github.com/xk/node-threads-a-gogo.git
2 cd node-threads-a-gogo
3 # One of
4 node-gyp rebuild
5 node test/all.js
6 # or
7 npm install
8 node test/all.js
9 # or
10 node-waf configure build install test
11 # Depending of what version of node you've got.
12 #
13 # THREADS_A_GOGO CURRENTLY (v0.1.13 / 2017)
14 # RUNS ON NODES v0.5.1 TO v6.9.2
```

Basic functionality test:

```
1 cd node-threads-a-gogo
2 node test/all.js
3 # If all goes well should output something like this:
4 0.OK.1.OK.2.OK.3.TAGG OBJECT OK
5 4.OK.5.OK.6.OK.7.OK.8.OK.9.OK.10.OK.11.OK.12.THREAD OBJECT OK
6 13.OK.WAITING FOR LOAD CB
7 14.OK.15.LOAD CALLBACK OK
8 16.OK.WAITING FOR EVAL CB
```

```
9 17.OK.18.OK.19.EVAL CALLBACK OK
10 20.OK.WAITING FOR EVENT LISTENER CB
11 21.OK.22.OK.23.OK.24.OK.25.OK.26.EVENT LISTENER CB.OK
12 27.OK.WAITING FOR DESTROY CB
13 28.OK.29.DESTROY CB OK
14 END
15 THREADS_A_GOGO v0.1.13 BASIC FUNCTIONALITY TEST: OK, IT WORKS!
```

To include the module in your project:

```
1 var tagg= require('threads_a_gogo');
```

You need a node with a v8 >= 3.2.4 to run this module. Any node >= 0.5.1 comes with a v8 >= 3.2.4.

The module **runs fine, though, in any node >= 0.1.13** as long as you build it with a v8 >= 3.2.4, see here. To do that you simply have to replace /node/deps/v8 with a newer version of v8 and re-compile it (recompile node). To get any version of node goto <http://nodejs.org/dist/>, and for v8 goto <http://github.com/v8/v8>, click on “branch”, select the proper tag (>= 3.2.4), and download the .zip.

Intro

After the initialization phase of a Node program, whose purpose is to setup listeners and callbacks to be executed in response to events, the next phase, the proper execution of the program, is orchestrated by the event loop whose duty is to juggle events, listeners and callbacks quickly and without any hiccups nor interruptions that would ruin its performance

Both the event loop and said listeners and callbacks run sequentially in a single thread of execution, Node’s main thread. If any of them ever blocks, nothing else will happen for the duration of the block: no more events will be handled, no more callbacks nor listeners nor timeouts nor nextTick()ed functions will have the chance to run and do their job, because they won’t be called by the blocked event loop, and the program will turn sluggish at best, or appear to be frozen and dead at worst.

A.- Here’s a program that makes Node’s event loop spin freely and as fast as possible: it simply prints a dot to the console in each turn:

```
1 cat examples/quickIntro_loop.js
2 node examples/quickIntro_loop.js
```

```
1 (function spinForever () {
2   process.stdout.write(".");
3   setImmediate(spinForever);
4 })();
```

B.- Here's another program that adds to the one above a fibonacci(35) call in each turn, a CPU-bound task that takes quite a while to complete and that blocks the event loop making it spin slowly and clumsily. The point is simply to show that you can't put a job like that in the event loop because Node will stop performing properly when its event loop can't spin fast and freely due to a callback/listener/nextTick()ed function that's blocking.

```
1 cat examples/quickIntro_blocking.js
2 node examples/quickIntro_blocking.js
```

```
1 function fibo (n) {
2   return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1;
3 }
4
5 (function fiboLoop () {
6   process.stdout.write(fibo(35).toString());
7   setImmediate(fiboLoop);
8 })();
9
10 (function spinForever () {
11   process.stdout.write(".");
12   setImmediate(spinForever);
13 })();
```

C.- The program below uses `threads_a_gogo` to run the fibonacci(35) calls in a background thread, so Node's event loop isn't blocked at all and can spin freely again at full speed:

```
1 cat examples/quickIntro_oneThread.js
2 node examples/quickIntro_oneThread.js
```

```
1 function fibo (n) {
2   return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1;
3 }
4
5 function cb (err, data) {
6   process.stdout.write(data);
7   this.eval('fibo(35)', cb);
8 }
9
10 var thread= require('threads_a_gogo').create();
11
12 thread.eval(fibo).eval('fibo(35)', cb);
13
14 (function spinForever () {
15   process.stdout.write(".");
16   setImmediate(spinForever);
17 })();
```

D.- This example is almost identical to the one above, only that it creates 5 threads instead of one,

each running a fibonacci(35) in parallel and in parallel too with Node's event loop that keeps spinning happily at full speed in its own thread:

```
1 cat examples/quickIntro_fiveThreads.js
2 node examples/quickIntro_fiveThreads.js
```

```
1 function fibo (n) {
2   return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1;
3 }
4
5 function cb (err, data) {
6   process.stdout.write(" ["+ this.id+ "]" + data);
7   this.eval('fibo(35)', cb);
8 }
9
10 var tagg= require('threads_a_gogo');
11
12 tagg.create().eval(fibo).eval('fibo(35)', cb);
13 tagg.create().eval(fibo).eval('fibo(35)', cb);
14 tagg.create().eval(fibo).eval('fibo(35)', cb);
15 tagg.create().eval(fibo).eval('fibo(35)', cb);
16 tagg.create().eval(fibo).eval('fibo(35)', cb);
17
18 (function spinForever () {
19   process.stdout.write(".");
20   setImmediate(spinForever);
21 })();
```

E.- The next one asks `threads_a_gogo` to create a pool of 10 background threads, instead of creating them manually one by one:

```
1 cat examples/quickIntro_multiThread.js
2 node examples/quickIntro_multiThread.js
```

```
1 function fibo (n) {
2   return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1;
3 }
4
5 var numThreads= 10;
6 var threadPool= require('threads_a_gogo').createPool(numThreads).all.
   eval(fibo);
7
8 threadPool.all.eval('fibo(35)', function cb (err, data) {
9   process.stdout.write(" ["+ this.id+ "]" + data);
10   this.eval('fibo(35)', cb);
11 });
12
13 (function spinForever () {
14   process.stdout.write(".");
15   setImmediate(spinForever);
```

```
16 })();
```

F.- This is a demo of the `threads_a_gogo` EventEmitter API, using one thread:

```
1 cat examples/quickIntro_oneThreadEvented.js
2 node examples/quickIntro_oneThreadEvented.js
```

```
1 var thread= require('threads_a_gogo').create();
2 thread.load(__dirname + '/quickIntro_evented_childThreadCode.js');
3
4 /*
5   This is the code that's .load()ed into the child/background thread:
6
7   function fibo (n) {
8     return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1;
9   }
10
11   thread.on('giveMeTheFibo', function onGiveMeTheFibo (data) {
12     this.emit('theFiboIs', fibo(+data)); //Emits 'theFiboIs' in the
13     parent/main thread.
14   });
15 */
16
17 //Emit 'giveMeTheFibo' in the child/background thread.
18 thread.emit('giveMeTheFibo', 35);
19
20 //Listener for the 'theFiboIs' events emitted by the child/background
21 thread.
22 thread.on('theFiboIs', function cb (data) {
23   process.stdout.write(data);
24   this.emit('giveMeTheFibo', 35);
25 });
26
27 (function spinForever () {
28   process.stdout.write(".");
29   setImmediate(spinForever);
30 })();
```

G.- This is a demo of the `threads_a_gogo` EventEmitter API, using a pool of threads:

```
1 cat examples/quickIntro_multiThreadEvented.js
2 node examples/quickIntro_multiThreadEvented.js
```

```
1 var numThreads= 10;
2 var threadPool= require('threads_a_gogo').createPool(numThreads);
3 threadPool.load(__dirname + '/quickIntro_evented_childThreadCode.js');
4
5 /*
6   This is the code that's .load()ed into the child/background threads:
```

```

7
8   function fibo (n) {
9       return n > 1 ? fibo(n - 1) + fibo(n - 2) : 1;
10  }
11
12  thread.on('giveMeTheFibo', function onGiveMeTheFibo (data) {
13      this.emit('theFiboIs', fibo(+data)); //Emits 'theFiboIs' in the
        parent/main thread.
14  });
15
16  */
17
18  //Emit 'giveMeTheFibo' in all the child/background threads.
19  threadPool.all.emit('giveMeTheFibo', 35);
20
21  //Listener for the 'theFiboIs' events emitted by the child/background
        threads.
22  threadPool.on('theFiboIs', function cb (data) {
23      process.stdout.write(" ["+ this.id+ "]" + data);
24      this.emit('giveMeTheFibo', 35);
25  });
26
27  (function spinForever () {
28      process.stdout.write(".");
29      setImmediate(spinForever);
30  })();

```

More examples

The [examples](#) directory contains a few more examples:

- ex01_basic: Running a simple function in a thread.
- ex02_events: Sending events from a worker thread.
- ex03_ping_pong: Sending events both ways between the main thread and a worker thread.
- ex04_main: Loading the worker code from a file.
- ex05_pool: Using the thread pool.
- ex06_jason: Passing complex objects to threads.

Module API

TAGG Object API

```

1  tagg= require('threads_a_gogo') -> tagg object
2
3  { create: [Function],

```

```
4   createPool: [Function: createPool],
5   version: '0.1.13' }
```

.create()

`tagg.create(/*no arguments */)` -> thread object

.createPool(numThreads)

`tagg.createPool(numberOfThreads)` -> threadPool object

.version

`tagg.version` -> A string with the threads_a_gogo version number.

Thread Object API (this is the thread object you get in node's main thread)

```
1  thread= tagg.create() -> thread object
2
3  { load: [Function: load],
4    eval: [Function: eval],
5    emit: [Function: emit],
6    destroy: [Function: destroy],
7    id: 0,
8    version: '0.1.13',
9    on: [Function: on],
10   once: [Function: once],
11   _on: {},
12   removeAllListeners: [Function: removeAllListeners] }
```

.id

`thread.id` -> A sequential thread serial number.

.version

`thread.version` -> A string with the threads_a_gogo version number.

.load(path [, cb])

`thread.load(path [, cb]) -> thread`

Loads the file at `path` and `thread.eval(fileContents, cb)`.

NOTE that most methods return the `thread` object so that calls can be chained like this: `thread.load("boot.js").eval("boot()").emit("go").on("event", cb) ...`

.eval(program [, cb])

`thread.eval(program [, cb]) -> thread`

Converts `program.toString()` and `eval()`s (which is the equivalent of loading a script in a browser) it in the thread's global context, and (if provided) returns to a callback the completion value: `cb(err, completionValue)`. Some examples you can try in node's console, but first please declare a `function cb (a,b){ console.log(a,b)}`, that will be handy, then type `thread.eval("Math.random()", cb)` the thread will run that (`Math.random()`) and the `cb(err, result)` will get (and display) a `null` (`random number`). A `thread.eval('function hello (){ puts("Hello!"); return "world!" }', cb)` will eval that in the global context of the thread and thus create a global function `hello` in the thread's js global context and your `cb(err, result)` will get `null undefined` in the result, `null` because there were no errors and `undefined` because the completion value of a function declaration is `undefined`. On the plus side, now you have injected for the first time in your life some js code of yours in a TAGG's thread and you can tell it to run that, do a `thread.eval('hello()', cb)` and the thread will print `Hello!` in the console and the `cb(err, result)` will receive `world!` into result. How cool is that?

.on(eventType, listener)

`thread.on(eventType, listener) -> thread`

Registers the listener `listener(data [, data2 ...])` for any events of `eventType` that the thread `thread` may emit. For example, declare a `function cb (a,b,c){ console.log(a,b,c)}` and then do `thread.on("event", cb)`. Now whenever the thread emits an event of the type `event` (which by the way can be any arbitrary name/string you choose), `cb` will be triggered. Let's do that with a `thread.eval('i=5; while (i--)thread.emit("event", "How", "cool", "is that?")')` and the console will display `How cool is that?` five times, huh, unbelievable.

.once(eventType, listener)

`thread.once(eventType, listener) -> thread`

Like `thread.on()`, but the listener will only be triggered once.

.removeAllListeners([eventType])

`thread.removeAllListeners([eventType]) -> thread`

Deletes all listeners for all eventTypes unless `eventType` is provided, in which case it deletes the listeners only of the event type `eventType`.

.emit(eventType, eventData [, eventData ...])

`thread.emit(eventType, eventData [, eventData ...]) -> thread`

Emit an event of `eventType` with `eventData` in the thread `thread`. All its arguments are `.toString()`ed.

.destroy([rudely])

`thread.destroy([0 (nicely) | 1 (rudely)] [, cb]) -> undefined`

Destroys the thread. If the first parameter is not provided or falsy or 0 (the default) the thread will keep running until both its nextTick/setImmediate queue and its pending `.eval()` jobs queue are empty. If it's truthy or 1 (rudely) the thread's event loop will exit as soon as possible, regardless. If a callback `cb` (optional) is provided, it will be called when the thread has been killed and completely destroyed, the `cb` will receive no arguments and the receiver (its 'this') points to the global object. If the thread is stuck in a while (1) ; or similar it won't end and currently tagg has no way around that.

_on

Ignore, don't touch that. The `_on` object holds the event listeners.

Thread object API (this is the thread object that exists not in node but as a global in every thread)

Inside every thread `.create()`d by `threads_a_gogo`, there's a global `thread` object with these properties:

```
1 thread (a global) ->
2
3 { id: 0,
4   version: '0.1.13',
5   on: [Function: on],
6   once: [Function: once],
7   emit: [Function: emit],
8   removeAllListeners: [Function: removeAllListeners],
9   nextTick: [Function: nextTick],
10  _on: {},
11  _ntq: [] }
```

.id

`thread.id` -> the serial number of this thread.

.version

`thread.version` -> A string with the `threads_a_gogo` version number.

.on(eventType, listener)

`thread.on(eventType, listener)` -> thread

Just like `thread.on()` above. But in this case it will receive events emitted by node's main thread, because, remember, all this exists in the thread's own js context which is totally independent of node's js context.

.once(eventType, listener)

`thread.once(eventType, listener)` -> thread

Just like `thread.once()` above.

.emit(eventType, eventData [, eventData ...])

`thread.emit(eventType, eventData [, eventData ...]) -> thread`

Just like `thread.emit()` above. What this emits will trigger a listener (if set) in node's main thread.

.removeAllListeners([eventType])

`thread.removeAllListeners([eventType]) -> thread`

Just like `thread.removeAllListeners()` above.

.nextTick(function)

`thread.nextTick(function) -> undefined`

Like `setImmediate()`, but twice as fast. Every thread runs its own event loop. First it looks for any `.eval()`s that node may have sent, if there are any it runs the first one, after that it looks in the `nextTick/setImmediate` queue, if there are any functions there to run, it runs them all (but only in chunks of up to 8192 so that `nextTick/setImmediate` can never totally block a thread), and then looks again to see if there are any more `.eval()` events, if there are runs the first one, and repeats this loop forever until there are no more `.eval()` events nor `nextTick/setImmediate` functions at which point the thread goes to sleep until any further `.eval()`s or `.emit()`s sent from node awake it.

_on

Ignore, don't touch that. The `_on` object holds the event listeners.

_ntq

Ignore, don't touch that. The `_ntq` array holds the `nextTick()`ed and/or `setImmediate()`d functions.

Globals in the threads' js contexts

Inside every thread `.create()`d by `threads_a_gogo`, on top of the usual javascript ones there's these other globals:

puts(arg1 [, arg2 ...])

`puts(arg1 [, arg2 ...])` -> undefined

`.toString()`s and `fprintf(stdout)`s and `fflush(stdout)`es its arguments to (you guessed it) `stdout`.

setImmediate(function)

`setImmediate(function)` -> undefined

Just an alias for `thread.nextTick(function)`.

thread

The thread object described above is also a global.

Thread pool API

```
1 pool= tagg.createPool( numbreOfThreads ) ->
2
3 { load: [Function: load],
4   on: [Function: on],
5   any:
6     { eval: [Function: evalAny],
7       emit: [Function: emitAny] },
8   all:
9     { eval: [Function: evalAll],
10      emit: [Function: emitAll] },
11   totalThreads: [Function: getTotalThreads],
12   idleThreads: [Function: getIdleThreads],
13   pendingJobs: [Function: getPendingJobs],
14   destroy: [Function: destroy],
15   pool:
16     [ { load: [Function: load],
17        eval: [Function: eval],
18        emit: [Function: emit],
19        destroy: [Function: destroy],
20        id: 0,
21        version: '0.1.13',
22        on: [Function: on],
23        once: [Function: once],
24        _on: {},
25        removeAllListeners: [Function: removeAllListeners] } ] }
```

.load(path [, cb])

`pool.load(path [, cb]) -> pool`

`thread.load(path [, cb])` in every one of the pool's threads. The cb will be called as many times as threads there are.

.on(eventType, listener)

`pool.on(eventType, listener) -> pool`

Registers a listener for events of eventType. Any events of eventType emitted by any thread of the pool will trigger that cb/listener. For example, if you set a `pool.on("event", cb)` and any thread does a `thread.emit("event", "Hi", "there")`, a `cb(a, b)` will receive "Hi" in a and "There" in b.

.any.eval(program, cb)

`pool.any.eval(program, cb) -> pool`

Like `thread.eval()`, but in any of the pool's threads.

.any.emit(eventType, eventData [, eventData ...])

`pool.any.emit(eventType, eventData [, eventData ...]) -> pool`

Like `thread.emit()` but in any of the pool's threads.

.all.eval(program, cb)

`pool.all.eval(program, cb) -> pool`

Like `thread.eval()`, but in all the pool's threads.

.all.emit(eventType, eventData [, eventData ...])

`pool.all.emit(eventType, eventData [, eventData ...]) -> pool`

Like `thread.emit()` but in all the pool's threads.

.totalThreads()

`pool.totalThreads()` -> returns the number of threads in this pool: as supplied in `.createPool(number)`. It's also the same as `pool.pool.length`.

.idleThreads()

`pool.idleThreads()` -> returns the number of threads in this pool that are currently idle. Does not work very well currently. Better don't use it.

.pendingJobs()

`threadPool.pendingJobs()` -> returns the number of jobs pending. Does not work very well currently. Better don't use it.

.destroy([rudely])

`pool.destroy([rudely])` -> undefined

If rudely is 0 or falsy the thread will exit when there aren't any more events in its events queue. If rudely it will quit regardless of that. If stuck in a while (1) ; or similar it won't and currently tagg has no way around that. At least not yet. Pull requests are very much welcomed, just so you know.

.pool

`pool.pool` is an array that contains all the threads in the thread pool for your tinkering pleasure.

Rationale

Node.js is the most awesome, cute and super-sexy piece of free, open source software.

Its event loop can spin as fast and smooth as a turbo, and roughly speaking, **the faster it spins, the more power it delivers**. That's why @ryah took great care to ensure that no -possibly slow- I/O operations could ever block it: a pool of background threads (thanks to Marc Lehmann's libeio library) handle any blocking I/O calls in the background, in parallel.

In Node it's verboten to write a server like this:

```
1 http.createServer(function (req,res) {
2   res.end( fs.readFileSync(path) );
3 }).listen(port);
```

Because synchronous I/O calls **block the turbo**, and without proper boost, Node.js begins to stutter and behaves clumsily. To avoid it there's the asynchronous version of `.readFile()`, in continuation passing style, that takes a callback:

```
1 fs.readFile(path, function cb (err, data) { /* ... */ });
```

It's cool, we love it (*), and there's hundreds of ad hoc built-in functions like this in Node to help us deal with almost any variety of possibly slow, blocking I/O.

But what's with longish, CPU-bound tasks?

How do you avoid blocking the event loop, when the task at hand isn't I/O bound, and lasts more than a few fractions of a millisecond?

```
1 http.createServer(function cb (req,res) {
2   res.end( fibonacci(40) );
3 }).listen(port);
```

You simply can't, because there's no way... well, there wasn't before `threads_a_gogo`.

What is Threads A GoGo for Node.js

`threads_a_gogo` provides the asynchronous API for CPU-bound tasks that's missing in Node.js. Both in continuation passing style (callbacks), and in event emitter style (event listeners).

The same API Node uses to delegate a longish I/O task to a background (libeio) thread:

```
asyncIOTask(what, cb);
```

`threads_a_gogo` uses to delegate a longish CPU task to a background (JavaScript) thread:

```
thread.eval(program, cb);
```

So with `threads_a_gogo` you can write:

```
1 http.createServer(function (req,res) {
2   thread.eval('fibonacci(40)', function cb (err, data) {
3     res.end(data);
4   });
5 }).listen(port);
```

And it won't block the event loop because the `fibonacci(40)` will run in parallel in a separate background thread.

Why Threads

Threads (kernel threads) are very interesting creatures. They provide:

- 1.- Parallelism: All the threads run in parallel. On a single core processor, the CPU is switched rapidly back and forth among the threads providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one. With 10 compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with 1/10th the speed of the real CPU. On a multi-core processor, threads are truly running in parallel, and get time-sliced when the number of threads exceed the number of cores. So with 12 compute bound threads on a quad-core processor each thread will appear to run at 1/3rd of the nominal core speed.
- 2.- Fairness: No thread is more important than another, cores and CPU slices are fairly distributed among threads by the OS scheduler.
- 3.- Threads fully exploit all the available CPU resources in your system. On a loaded system running many tasks in many threads, the more cores there are, the faster the threads will complete. Automatically.
- 4.- The threads of a process share exactly the same address space, that of the process they belong to. Every thread can access every memory address within the process' address space. This is a very appropriate setup when the threads are actually part of the same job and are actively and closely cooperating with each other. Passing a reference to a chunk of data via a pointer is many orders of magnitude faster than transferring a copy of the data via IPC.

Why not multiple processes.

The "can't block the event loop" problem is inherent to Node's evented model. No matter how many Node processes you have running as a Node-cluster, it won't solve its issues with CPU-bound tasks.

Launch a cluster of N Nodes running the example B (`quickIntro_blocking.js`) above, and all you'll get is N -instead of one- Nodes with their event loops blocked and showing a sluggish performance.