Скрипт на Nodejs должен принимать на вход два параметра:

- memorylimit - число, limit памяти.
- file - имя файла.

Прочитывать указанный файл, и отправлять его содержимое на работу в виртуальную машину (http://nodejs.org/api/vm.html) в отдельном процессе.

Проще говоря, это песочница, надо передавать название файла для запуска в песочнице. Отдельный процесс нужен для контроля памяти, за который отвечает memorylimit параметер. Для этого при запуске нового процесса с виртуальной машиной надо указать:

node --max-old-space-size={memorylimit} vmscript.js

Ознакомление с vm модулем (для себя впервые)

On Aug 6, 3:36 am, Steve M <sm49...@gmail.com> wrote:
> I was hoping to avoid having to spawn individual nodejs processes to
> avoid the overhead. It would be great if it was possible somehow to
> have a 'pool' of nodejs processes running that scale up and down
> automatically based on load, but I haven't seen any way to do this
> within nodejs or with an external tool.

You're going to want to run them in a separate process, since you are running arbitrary javascript. For example, someone could write a script that locks up the process forever, which means your main script would stop serving requests if you were running it within the same process.

For the lazy, here's the code:

```
var vm = require('vm'),
    code = 'var square = n * n;',
    fn = new Function('n', code),
    script = vm.createScript(code),
    sandbox;
 n = 5;
 sandbox = { n: n };
 benchmark = function(title, funk) {
   var end, i, start;
   start = new Date;
   for (i = 0; i < 5000; i++) {
     funk();
   }
   end = new Date;
   console.log(title + ': ' + (end - start) + 'ms');
```

```
}
var ctx = vm.createContext(sandbox);
benchmark('vm.runInThisContext', function() { vm.runInThisContext(code); });
benchmark('vm.runInNewContext', function() { vm.runInNewContext(code, sandbox); }
benchmark('script.runInThisContext', function() { script.runInThisContext(); });
benchmark('script.runInNewContext', function() { script.runInNewContext(sandbox);
benchmark('script.runInContext', function() { script.runInContext(ctx); });
benchmark('fn', function() { fn(n); });
```

This is a pretty simple benchmark script – there are some fundamental issues with it but it gives enough of a view that we can gauge a general sense of relative performance of various methods of executing the script. The script.* functions will use the pre-compiled script whereas the first two will compile at time of execution. The last item is a reference point. Executed on my machine, this gives me the following result:

```
vm.runInThisContext: 127ms
vm.runInNewContext: 1288ms
script.runInThisContext: 3ms
script.runInNewContext: 1110ms
script.runInContext: 23ms
fn: 0ms
```

Sandboxes#

The `sandbox` argument to `vm.runInNewContext` and `vm.createContext`, along with the `initSandbox` argument to`vm.createContext`, do not behave as one might normally expect and their behavior varies between different versions of Node.

The key issue to be aware of is that V8 provides no way to directly control the global object used within a context. As a result, while properties of your `sandbox` object will be available in the context, any properties from the `prototype`s of the `sandbox` may not be available. Furthermore, the `this` expression within the global scope of the context evaluates to the empty object (`{}`) instead of to your sandbox.

Your sandbox's properties are also not shared directly with the script. Instead, the properties of the sandbox are copied into the context at the beginning of execution, and then after execution, the properties are copied back out in an attempt to propagate any changes.

Globals#

Properties of the global object, like `Array` and `String`, have different values inside of a context. This means that common expressions like `[] instanceof Array` or `Object.getPrototypeOf([]) === Array.prototype` may not produce expected results when used inside of scripts evaluated via the `vm` module.

Some of these problems have known workarounds listed in the issues for `vm` on GitHub. for example,`Array.isArray` works around the example problem with `Array`.

vm.runInNewContext(code, [sandbox], [filename])#

`vm.runInNewContext` compiles `code`, then runs it in `sandbox` and returns the result. Running code does not have access to local scope. The object `sandbox` will be used as the global object for `code`. `sandbox` and `filename` are optional, `filename` is only used in stack traces.
Example: compile and execute code that increments a global variable and sets a new one. These globals are contained in the sandbox.

```
var util = require('util'),
    vm = require('vm'),
    sandbox = {
      animal: 'cat',
      count: 2
    };

vm.runInNewContext('count += 1; name = "kitty"', sandbox, 'myfile.vm');
console.log(util.inspect(sandbox));


// { animal: 'cat', count: 3, name: 'kitty' }
```

Note that running untrusted code is a tricky business requiring great care. To prevent accidental global variable leakage, `vm.runInNewContext` is quite useful, but safely running untrusted code requires a separate process.
In case of syntax error in `code`, `vm.runInNewContext` emits the syntax error to stderr and throws an exception.
process.execPath

node --v8-options
  --max_new_space_size (max size of the new generation (in kBytes))
      type: int default: 0
  --max_old_space_size (max size of the old generation (in Mbytes))
      type: int default: 0
  --max_executable_size (max size of executable memory (in Mbytes))
      type: int default: 0