openresty / lua-nginx-module

Star 1,203 Fork 266

Embed the Power of Lua into NGINX http://openresty.org

2,671 commits **6** branches **272** releases **21** contributors

branch: **master** lua-nginx-module / +

doc: bumped version to 0.9.12.

agentzh authored 3 days ago latest commit 73884588f6

deps	add missing module info for unit-test; remove unused code; update ndk…	4 years ago
doc	doc: bumped version to 0.9.12.	3 days ago
dtrace	feature: added new dtrace static probe http-lua-user-thread-wait.	2 years ago
misc/recv-until-pm	cosocket: did a minor optimization for receiveuntil patterns no longe…	3 years ago
src	bugfix: added allocation failure check for ngx_array_init(). thanks T…	15 days ago
t	tests: fixed a small bug in a test case.	7 days ago
tapset	feature: initial support for dtrace static probes. it requires nginx-…	2 years ago
util	test: now we require the ngx_lua_upstream module for the test suite.	7 months ago
.gitignore	feature: added new configuration directive, init_worker_by_lua, to ru…	8 months ago
.gitmodules	updated ndk upstream location	4 years ago
Changes	fixed the date in the Changes file.	3 years ago
README.markdown	doc: bumped version to 0.9.12.	3 days ago
config	config: now we also explicitly check the Lua ABI/language version in …	22 days ago
valgrind.suppress	suppressed a false positive in libdl.	5 months ago

Code
Issues 23
Pull Requests 12
Wiki
Pulse
Graphs

HTTPS clone URL

`https://github.com`

You can clone with HTTPS or Subversion.

Clone in Desktop

Download ZIP

README.markdown

Name

ngx_lua - Embed the power of Lua into Nginx

This module is not distributed with the Nginx source. See the installation instructions.

Table of Contents

Status

This module is under active development and is production ready.

Version

This document describes ngx_lua v0.9.12 released on 2 September 2014.

Synopsis

```
# set search paths for pure Lua external libraries (';;' is the default path):
lua_package_path '/foo/bar/?.lua;/blah/?.lua;;';

# set search paths for Lua external libraries written in C (can also use ';;'):
lua_package_cpath '/bar/baz/?.so;/blah/blah/?.so;;';

server {
    location /inline_concat {
        # MIME type determined by default_type:
        default_type 'text/plain';

        set $a "hello";
        set $b "world";
        # inline Lua script
        set_by_lua $res "return ngx.arg[1]..ngx.arg[2]" $a $b;
        echo $res;
    }

    location /rel_file_concat {
        set $a "foo";
        set $b "bar";
        # script path relative to nginx prefix
        # $ngx_prefix/conf/concat.lua contents:
        #
        #    return ngx.arg[1]..ngx.arg[2]
        #
        set_by_lua_file $res conf/concat.lua $a $b;
        echo $res;
    }

    location /abs_file_concat {
        set $a "fee";
        set $b "baz";
        # absolute script path not modified
        set_by_lua_file $res /usr/nginx/conf/concat.lua $a $b;
        echo $res;
    }

    location /lua_content {
        # MIME type determined by default_type:
        default_type 'text/plain';

        content_by_lua "ngx.say('Hello,world!')";
    }

     location /nginx_var {
        # MIME type determined by default_type:
        default_type 'text/plain';

        # try access /nginx_var?a=hello,world
        content_by_lua "ngx.print(ngx.var['arg_a'], '\\n')";
    }

    location /request_body {
         # force reading request body (default off)
         lua_need_request_body on;
         client_max_body_size 50k;
         client_body_buffer_size 50k;

         content_by_lua 'ngx.print(ngx.var.request_body)';
    }

    # transparent non-blocking I/O in Lua via subrequests
    location /lua {
        # MIME type determined by default_type:
        default_type 'text/plain';

        content_by_lua '
            local res = ngx.location.capture("/some_other_location")
            if res.status == 200 then
                ngx.print(res.body)
            end';
    }

    # GET /recur?num=5
    location /recur {
        # MIME type determined by default_type:
        default_type 'text/plain';
        content_by_lua '
           local num = tonumber(ngx.var.arg_num) or 0

           if num > 50 then
               ngx.say("num too big")
               return
           end

           ngx.say("num is: ", num)

           if num > 0 then
               res = ngx.location.capture("/recur?num=" .. tostring(num - 1))
               ngx.print("status=", res.status, " ")
               ngx.print("body=", res.body)
           else
               ngx.say("end")
           end
           ';
    }

    location /foo {
        rewrite_by_lua '
            res = ngx.location.capture("/memc",
                { args = { cmd = "incr", key = ngx.var.uri } }
            )
        ';

        proxy_pass http://blah.blah.com;
    }

    location /blah {
        access_by_lua '
            local res = ngx.location.capture("/auth")

            if res.status == ngx.HTTP_OK then
                return
            end

            if res.status == ngx.HTTP_FORBIDDEN then
                ngx.exit(res.status)
            end

            ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
        ';

        # proxy_pass/fastcgi_pass/postgres_pass/...
    }

    location /mixed {
        rewrite_by_lua_file /path/to/rewrite.lua;
        access_by_lua_file /path/to/access.lua;
        content_by_lua_file /path/to/content.lua;
    }

    # use nginx var in code path
    # WARN: contents in nginx var must be carefully filtered,
    # otherwise there'll be great security risk!
    location ~ ^/app/(.+) {
            content_by_lua_file /path/to/lua/app/root/$1.lua;
    }

    location / {
       lua_need_request_body on;

       client_max_body_size 100k;
```

```
        client_body_buffer_size 100k;

        access_by_lua '
            -- check the client IP address is in our black list
            if ngx.var.remote_addr == "132.5.72.3" then
                ngx.exit(ngx.HTTP_FORBIDDEN)
            end

            -- check if the request body contains bad words
            if ngx.var.request_body and
                     string.match(ngx.var.request_body, "fsck")
            then
                return ngx.redirect("/terms_of_use.html")
            end

            -- tests passed
        ';

        # proxy_pass/fastcgi_pass/etc settings
    }
}
```

Back to TOC

Description

This module embeds Lua, via the standard Lua 5.1 interpreter or LuaJIT 2.0/2.1, into Nginx and by leveraging Nginx's subrequests, allows the integration of the powerful Lua threads (Lua coroutines) into the Nginx event model.

Unlike Apache's mod_lua and Lighttpd's mod_magnet, Lua code executed using this module can be *100% non-blocking* on network traffic as long as the Nginx API for Lua provided by this module is used to handle requests to upstream services such as MySQL, PostgreSQL, Memcached, Redis, or upstream HTTP web services.

At least the following Lua libraries and Nginx modules can be used with this ngx_lua module:

- lua-resty-memcached
- lua-resty-mysql
- lua-resty-redis
- lua-resty-dns
- lua-resty-upload
- lua-resty-websocket
- lua-resty-lock
- lua-resty-string
- ngx_memc
- ngx_postgres
- ngx_redis2
- ngx_redis
- ngx_proxy
- ngx_fastcgi

Almost all the Nginx modules can be used with this ngx_lua module by means of ngx.location.capture or ngx.location.capture_multi but it is recommended to use those `lua-resty-*` libraries instead of creating subrequests to access the Nginx upstream modules because the former is usually much more flexible and memory-efficient.

The Lua interpreter or LuaJIT instance is shared across all the requests in a single nginx worker process but request contexts are segregated using lightweight Lua coroutines.

Loaded Lua modules persist in the nginx worker process level resulting in a small memory footprint in Lua even when under heavy loads.

Back to TOC

Typical Uses

Just to name a few:

- Mashup'ing and processing outputs of various nginx upstream outputs (proxy, drizzle, postgres, redis, memcached, and etc) in Lua,
- doing arbitrarily complex access control and security checks in Lua before requests actually reach the upstream backends,
- manipulating response headers in an arbitrary way (by Lua)
- fetching backend information from external storage backends (like redis, memcached, mysql, postgresql) and use that information to choose which upstream backend to access on-the-fly,
- coding up arbitrarily complex web applications in a content handler using synchronous but still non-blocking access to the database backends and other storage,
- doing very complex URL dispatch in Lua at rewrite phase,
- using Lua to implement advanced caching mechanism for Nginx's subrequests and arbitrary locations.

The possibilities are unlimited as the module allows bringing together various elements within Nginx as well as exposing the power of the Lua language to the user. The module provides the full flexibility of scripting while offering performance levels comparable with native C language programs both in terms of CPU time as well as memory footprint. This is particularly the case when LuaJIT 2.x is enabled.

Other scripting language implementations typically struggle to match this performance level.

The Lua state (Lua VM instance) is shared across all the requests handled by a single nginx worker process to minimize memory use.

Back to TOC

Nginx Compatibility

The latest module is compatible with the following versions of Nginx:

- 1.7.x (last tested: 1.7.4)
- 1.6.x
- 1.5.x (last tested: 1.5.12)
- 1.4.x (last tested: 1.4.4)
- 1.3.x (last tested: 1.3.11)
- 1.2.x (last tested: 1.2.9)
- 1.1.x (last tested: 1.1.5)
- 1.0.x (last tested: 1.0.15)
- 0.9.x (last tested: 0.9.4)
- 0.8.x >= 0.8.54 (last tested: 0.8.54)

Back to TOC

Installation

The ngx_openresty bundle can be used to install Nginx, ngx_lua, either one of the standard Lua 5.1 interpreter or LuaJIT 2.0/2.1, as well as a package of powerful companion Nginx modules. The basic installation step is a simple `./configure --with-luajit && make && make install`.

Alternatively, ngx_lua can be manually compiled into Nginx:

1. Install LuaJIT 2.0 or 2.1 (recommended) or Lua 5.1 (Lua 5.2 is *not* supported yet). LuaJIT can be downloaded from the the LuaJIT project website and Lua 5.1, from the Lua project website. Some distribution package managers also distribute LuajIT and/or Lua.
2. Download the latest version of the ngx_devel_kit (NDK) module HERE.
3. Download the latest version of ngx_lua HERE.
4. Download the latest version of Nginx HERE (See Nginx Compatibility)

Build the source with this module:

```
wget 'http://nginx.org/download/nginx-1.7.4.tar.gz'
tar -xzvf nginx-1.7.4.tar.gz
cd nginx-1.7.4/

# tell nginx's build system where to find LuaJIT 2.0:
export LUAJIT_LIB=/path/to/luajit/lib
export LUAJIT_INC=/path/to/luajit/include/luajit-2.0

# tell nginx's build system where to find LuaJIT 2.1:
export LUAJIT_LIB=/path/to/luajit/lib
export LUAJIT_INC=/path/to/luajit/include/luajit-2.1

# or tell where to find Lua if using Lua instead:
#export LUA_LIB=/path/to/lua/lib
#export LUA_INC=/path/to/lua/include

# Here we assume Nginx is to be installed under /opt/nginx/.
./configure --prefix=/opt/nginx \
        --add-module=/path/to/ngx_devel_kit \
        --add-module=/path/to/lua-nginx-module

make -j2
make install
```

Back to TOC

C Macro Configurations

While building this module either via OpenResty or with the NGINX core, you can define the following C macros via the C compiler options:

- `NGX_LUA_USE_ASSERT` When defined, will enable assertions in the ngx_lua C code base. Recommended for debugging or testing builds. It can introduce some (small) runtime overhead when enabled. This macro was first introduced in the `v0.9.10` release.
- `NGX_LUA_ABORT_AT_PANIC` When the Lua/LuaJIT VM panics, ngx_lua will instruct the current nginx worker process to quit gracefully by default. By specifying this C macro, ngx_lua will abort the current nginx worker process (which usually result in a core dump file) immediately. This

option is useful for debugging VM panics. This option was first introduced in the `v0.9.8` release.
- `NGX_LUA_NO_FFI_API` Excludes pure C API functions for FFI-based Lua API for NGINX (as required by lua-resty-core, for example). Enabling this macro can make the resulting binary code size smaller.

To enable one or more of these macros, just pass extra C compiler options to the `./configure` script of either NGINX or OpenResty. For instance,

```
./configure --with-cc-opt="-DNGX_LUA_USE_ASSERT -DNGX_LUA_ABORT_AT_PANIC"
```

Back to TOC

Installation on Ubuntu 11.10

Note that it is recommended to use LuaJIT 2.0 or LuaJIT 2.1 instead of the standard Lua 5.1 interpreter wherever possible.

If the standard Lua 5.1 interpreter is required however, run the following command to install it from the Ubuntu repository:

```
apt-get install -y lua5.1 liblua5.1-0 liblua5.1-0-dev
```

Everything should be installed correctly, except for one small tweak.

Library name `liblua.so` has been changed in liblua5.1 package, it only comes with `liblua5.1.so`, which needs to be symlinked to `/usr/lib` so it could be found during the configuration process.

```
ln -s /usr/lib/x86_64-linux-gnu/liblua5.1.so /usr/lib/liblua.so
```

Back to TOC

Community

Back to TOC

English Mailing List

The openresty-en mailing list is for English speakers.

Back to TOC

Chinese Mailing List

The openresty mailing list is for Chinese speakers.

Back to TOC

Code Repository

The code repository of this project is hosted on github at openresty/lua-nginx-module.

Back to TOC

Bugs and Patches

Please submit bug reports, wishlists, or patches by

1. creating a ticket on the GitHub Issue Tracker,
2. or posting to the OpenResty community.

Back to TOC

Lua/LuaJIT bytecode support

As from the `v0.5.0rc32` release, all `*_by_lua_file` configure directives (such as content_by_lua_file) support loading Lua 5.1 and LuaJIT 2.0/2.1 raw bytecode files directly.

Please note that the bytecode format used by LuaJIT 2.0/2.1 is not compatible with that used by the standard Lua 5.1 interpreter. So if using LuaJIT 2.0/2.1 with ngx_lua, LuaJIT compatible bytecode files must be generated as shown:

```
/path/to/luajit/bin/luajit -b /path/to/input_file.lua /path/to/output_file.luac
```

The `-bg` option can be used to include debug information in the LuaJIT bytecode file:

```
/path/to/luajit/bin/luajit -bg /path/to/input_file.lua /path/to/output_file.luac
```

Please refer to the official LuaJIT documentation on the `-b` option for more details:

http://luajit.org/running.html#opt_b

Also, the bytecode files generated by LuaJIT 2.1 is *not* compatible with LuaJIT 2.0, and vice versa. The support for LuaJIT 2.1 bytecode was first added in ngx_lua v0.9.3.

Similarly, if using the standard Lua 5.1 interpreter with ngx_lua, Lua compatible bytecode files must be generated using the `luac` commandline utility as shown:

```
luac -o /path/to/output_file.luac /path/to/input_file.lua
```

Unlike as with LuaJIT, debug information is included in standard Lua 5.1 bytecode files by default. This can be striped out by specifying the `-s` option as shown:

```
luac -s -o /path/to/output_file.luac /path/to/input_file.lua
```

Attempts to load standard Lua 5.1 bytecode files into ngx_lua instances linked to LuaJIT 2.0/2.1 or vice versa, will result in an error message, such as that below, being logged into the Nginx `error.log` file:

```
[error] 13909#0: *1 failed to load Lua inlined code: bad byte-code header in /path/to/t
```

Loading bytecode files via the Lua primitives like `require` and `dofile` should always work as expected.

Back to TOC

System Environment Variable Support

If you want to access the system environment variable, say, `foo`, in Lua via the standard Lua API os.getenv, then you should also list this environment variable name in your `nginx.conf` file via the env directive. For example,

```
env foo;
```

Back to TOC

HTTP 1.0 support

The HTTP 1.0 protocol does not support chunked output and requires an explicit `Content-Length` header when the response body is not empty in order to support the HTTP 1.0 keep-alive. So when a HTTP 1.0 request is made and the lua_http10_buffering directive is turned `on`, ngx_lua will buffer the output of ngx.say and ngx.print calls and also postpone sending response headers until all the response body output is received. At that time ngx_lua can calculate the total length of the body and construct a proper `Content-Length` header to return to the HTTP 1.0 client. If the `Content-Length` response header is set in the running Lua code, however, this buffering will be disabled even if the lua_http10_buffering directive is turned `on`.

For large streaming output responses, it is important to disable the lua_http10_buffering directive to minimise memory usage.

Note that common HTTP benchmark tools such as `ab` and `http_load` issue HTTP 1.0 requests by default. To force `curl` to send HTTP 1.0 requests, use the `-0` option.

Back to TOC

Statically Linking Pure Lua Modules

When LuaJIT 2.x is used, it is possible to statically link the bytecode of pure Lua modules into the Nginx executable.

Basically you use the `luajit` executable to compile `.lua` Lua module files to `.o` object files containing the exported bytecode data, and then link the `.o` files directly in your Nginx build.

Below is a trivial example to demonstrate this. Consider that we have the following `.lua` file named `foo.lua`:

```
-- foo.lua
local _M = {}

function _M.go()
    print("Hello from foo")
end
```

```
return _M
```

And then we compile this `.lua` file to `foo.o` file:

```
/path/to/luajit/bin/luajit -bg foo.lua foo.o
```

What matters here is the name of the `.lua` file, which determines how you use this module later on the Lua land. The file name `foo.o` does not matter at all except the `.o` file extension (which tells `luajit` what output format is used). If you want to strip the Lua debug information from the resulting bytecode, you can just specify the `-b` option above instead of `-bg` .

Then when building Nginx or OpenResty, pass the `--with-ld-opt="foo.o"` option to the `./configure` script:

```
./configure --with-ld-opt="/path/to/foo.o" ...
```

Finally, you can just do the following in any Lua code run by ngx_lua:

```
local foo = require "foo"
foo.go()
```

And this piece of code no longer depends on the external `foo.lua` file any more because it has already been compiled into the `nginx` executable.

If you want to use dot in the Lua module name when calling `require` , as in

```
local foo = require "resty.foo"
```

then you need to rename the `foo.lua` file to `resty_foo.lua` before compiling it down to a `.o` file with the `luajit` command-line utility.

It is important to use exactly the same version of LuaJIT when compiling `.lua` files to `.o` files as building nginx + ngx_lua. This is because the LuaJIT bytecode format may be incompatible between different LuaJIT versions. When the bytecode format is incompatible, you will see a Lua runtime error saying that the Lua module is not found.

When you have multiple `.lua` files to compile and link, then just specify their `.o` files at the same time in the value of the `--with-ld-opt` option. For instance,

```
./configure --with-ld-opt="/path/to/foo.o /path/to/bar.o" ...
```

If you have just too many `.o` files, then it might not be feasible to name them all in a single command. In this case, you can build a static library (or archive) for your `.o` files, as in

```
ar rcus libmyluafiles.a *.o
```

then you can link the `myluafiles` archive as a whole to your nginx executable:

```
./configure \
    --with-ld-opt="-L/path/to/lib -Wl,--whole-archive -lmyluafiles -Wl,--no-whole-archi
```

where `/path/to/lib` is the path of the directory containing the `libmyluafiles.a` file. It should be noted that the linker option `--whole-archive` is required here because otherwise our archive will be skipped because no symbols in our archive are mentioned in the main parts of the nginx executable.

Back to TOC

Data Sharing within an Nginx Worker

To globally share data among all the requests handled by the same nginx worker process, encapsulate the shared data into a Lua module, use the Lua `require` builtin to import the module, and then manipulate the shared data in Lua. This works because required Lua modules are loaded only once and all coroutines will share the same copy of the module (both its code and data). Note however that Lua global variables (note, not module-level variables) WILL NOT persist between requests because of the one-coroutine-per-request isolation design.

Here is a complete small example:

```
-- mydata.lua
local _M = {}

local data = {
    dog = 3,
    cat = 4,
    pig = 5,
}

function _M.get_age(name)
    return data[name]
end

return _M
```

and then accessing it from `nginx.conf` :

```
location /lua {
    content_by_lua '
        local mydata = require "mydata"
        ngx.say(mydata.get_age("dog"))
    ';
}
```

The `mydata` module in this example will only be loaded and run on the first request to the location `/lua` , and all subsequent requests to the same nginx worker process will use the reloaded instance of the module as well as the same copy of the data in it, until a `HUP` signal is sent to the Nginx master process to force a reload. This data sharing technique is essential for high performance Lua applications based on this module.

Note that this data sharing is on a *per-worker* basis and not on a *per-server* basis. That is, when there are multiple nginx worker processes under an Nginx master, data sharing cannot cross the process boundary between these workers.

It is usually recommended to share read-only data this way. You can also share changeable data among all the concurrent requests of each nginx worker process as long as there is *no* nonblocking I/O operations (including ngx.sleep) in the middle of your calculations. As long as you do not give the control back to the nginx event loop and ngx_lua's light thread scheduler (even implicitly), there can never be any race conditions in between. For this reason, always be very careful when you want to share changeable data on the worker level. Buggy optimizations can easily lead to hard-to-debug race conditions under load.

If server-wide data sharing is required, then use one or more of the following approaches:

1. Use the ngx.shared.DICT API provided by this module.
2. Use only a single nginx worker and a single server (this is however not recommended when there is a multi core CPU or multiple CPUs in a single machine).
3. Use data storage mechanisms such as `memcached`, `redis`, `MySQL` or `PostgreSQL`. The ngx_openresty bundle associated with this module comes with a set of companion Nginx modules and Lua libraries that provide interfaces with these data storage mechanisms.

Back to TOC

Known Issues

Back to TOC

TCP socket connect operation issues

The tcpsock:connect method may indicate `success` despite connection failures such as with `Connection Refused` errors.

However, later attempts to manipulate the cosocket object will fail and return the actual error status message generated by the failed connect operation.

This issue is due to limitations in the Nginx event model and only appears to affect Mac OS X.

Back to TOC

Lua Coroutine Yielding/Resuming

- Because Lua's `dofile` and `require` builtins are currently implemented as C functions in both Lua 5.1 and LuaJIT 2.0/2.1, if the Lua file being loaded by `dofile` or `require` invokes ngx.location.capture*, ngx.exec, ngx.exit, or other API functions requiring yielding in the *top-level* scope of the Lua file, then the Lua error "attempt to yield across C-call boundary" will be raised. To avoid this, put these calls requiring yielding into your own Lua functions in the Lua file instead of the top-level scope of the file.
- As the standard Lua 5.1 interpreter's VM is not fully resumable, the methods ngx.location.capture, ngx.location.capture_multi, ngx.redirect, ngx.exec, and ngx.exit cannot be used within the context of a Lua pcall() or xpcall() or even the first line of the `for ... in ...` statement when the standard Lua 5.1 interpreter is used and the `attempt to yield across metamethod/C-call boundary` error will be produced. Please use LuaJIT 2.x, which supports a fully resumable VM, to avoid this.

Back to TOC

Lua Variable Scope

Care must be taken when importing modules and this form should be used:

```
local xxx = require('xxx')
```

instead of the old deprecated form:

```
require('xxx')
```

Here is the reason: by design, the global environment has exactly the same lifetime as the Nginx request handler associated with it. Each request handler has its own set of Lua global variables and that is the idea of request isolation. The Lua module is actually loaded by the first Nginx request handler and is cached by the `require()` built-in in the `package.loaded` table for later reference, and the `module()` builtin used by some Lua modules has the side effect of setting a global variable to the loaded module table. But this global variable will be cleared at the end of the request handler, and every subsequent request handler all has its own (clean) global environment. So one will get Lua exception for accessing the `nil` value.

Generally, use of Lua global variables is a really really bad idea in the context of ngx_lua because

1. misuse of Lua globals has very bad side effects for concurrent requests when these variables are actually supposed to be local only,
2. Lua global variables require Lua table look-up in the global environment (which is just a Lua table), which is kinda expensive, and
3. some Lua global variable references are just typos, which are hard to debug.

It's *highly* recommended to always declare them via "local" in the scope that is reasonable.

To find out all the uses of Lua global variables in your Lua code, you can run the lua-releng tool across all your .lua source files:

```
$ lua-releng
Checking use of Lua global variables in file lib/foo/bar.lua ...
        1       [1489]  SETGLOBAL       7 -1    ; contains
        55      [1506]  GETGLOBAL       7 -3    ; setvar
        3       [1545]  GETGLOBAL       3 -4    ; varexpand
```

The output says that the line 1489 of file `lib/foo/bar.lua` writes to a global variable named `contains`, the line 1506 reads from the global variable `setvar`, and line 1545 reads the global `varexpand`.

This tool will guarantee that local variables in the Lua module functions are all declared with the `local` keyword, otherwise a runtime exception will be thrown. It prevents undesirable race conditions while accessing such variables. See Data Sharing within an Nginx Worker for the reasons behind this.

Back to TOC

Locations Configured by Subrequest Directives of Other Modules

The ngx.location.capture and ngx.location.capture_multi directives cannot capture locations that include the echo_location, echo_location_async, echo_subrequest, or echo_subrequest_async directives.

```
location /foo {
    content_by_lua '
        res = ngx.location.capture("/bar")
    ';
}
```

```
location /bar {
    echo_location /blah;
}
location /blah {
    echo "Success!";
}
```

```
$ curl -i http://example.com/foo
```

will not work as expected.

Back to TOC

Special PCRE Sequences

PCRE sequences such as `\d` , `\s` , or `\w` , require special attention because in string literals, the backslash character, `\` , is stripped out by both the Lua language parser and by the Nginx config file parser before processing. So the following snippet will not work as expected:

```
# nginx.conf
? location /test {
?     content_by_lua '
?         local regex = "\d+"  -- THIS IS WRONG!!
?         local m = ngx.re.match("hello, 1234", regex)
?         if m then ngx.say(m[0]) else ngx.say("not matched!") end
?     ';
? }
# evaluates to "not matched!"
```

To avoid this, *double* escape the backslash:

```
# nginx.conf
location /test {
    content_by_lua '
        local regex = "\\\\d+"
        local m = ngx.re.match("hello, 1234", regex)
        if m then ngx.say(m[0]) else ngx.say("not matched!") end
    ';
}
# evaluates to "1234"
```

Here, `\\\\d+` is stripped down to `\\d+` by the Nginx config file parser and this is further stripped down to `\d+` by the Lua language parser before running.

Alternatively, the regex pattern can be presented as a long-bracketed Lua string literal by encasing it in "long brackets", `[[...]]` , in which case backslashes have to only be escaped once for the Nginx config file parser.

```
# nginx.conf
location /test {
    content_by_lua '
        local regex = [[\\d+]]
        local m = ngx.re.match("hello, 1234", regex)
        if m then ngx.say(m[0]) else ngx.say("not matched!") end
    ';
}
# evaluates to "1234"
```

Here, `[[\\d+]]` is stripped down to `[[\d+]]` by the Nginx config file parser and this is processed correctly.

Note that a longer from of the long bracket, `[=[...]=]`, may be required if the regex pattern contains `[...]` sequences. The `[=[...]=]` form may be used as the default form if desired.

```
# nginx.conf
location /test {
    content_by_lua '
        local regex = [=[[0-9]+]=]
        local m = ngx.re.match("hello, 1234", regex)
        if m then ngx.say(m[0]) else ngx.say("not matched!") end
    ';
}
# evaluates to "1234"
```

An alternative approach to escaping PCRE sequences is to ensure that Lua code is placed in external script files and executed using the various `*_by_lua_file` directives. With this approach, the backslashes are only stripped by the Lua language parser and therefore only need to be escaped once each.

```
-- test.lua
local regex = "\\d+"
local m = ngx.re.match("hello, 1234", regex)
if m then ngx.say(m[0]) else ngx.say("not matched!") end
-- evaluates to "1234"
```

Within external script files, PCRE sequences presented as long-bracketed Lua string literals do not require modification.

```
-- test.lua
local regex = [[\d+]]
local m = ngx.re.match("hello, 1234", regex)
if m then ngx.say(m[0]) else ngx.say("not matched!") end
-- evaluates to "1234"
```

Back to TOC

Mixing with SSI Not Supported

Mixing SSI with ngx_lua in the same Nginx request is not supported at all. Just use ngx_lua exclusively. Everything you can do with SSI can be done atop ngx_lua anyway and it can be more efficient when using ngx_lua.

Back to TOC

SPDY Mode Not Fully Supported

Certain Lua APIs provided by ngx_lua do not work in Nginx's SPDY mode yet: ngx.location.capture, ngx.location.capture_multi, and ngx.req.socket.

Back to TOC

TODO

Back to TOC

Short Term

- review and apply Jader H. Silva's patch for `ngx.re.split()` .
- review and apply vadim-pavlov's patch for ngx.location.capture's `extra_headers` option
- use `ngx_hash_t` to optimize the built-in header look-up process for ngx.req.set_header, ngx.header.HEADER, and etc.
- add configure options for different strategies of handling the cosocket connection exceeding in the pools.
- add directives to run Lua codes when nginx stops.
- add `ignore_resp_headers` , `ignore_resp_body` , and `ignore_resp` options to ngx.location.capture and ngx.location.capture_multi methods, to allow micro performance tuning on the user side.

Back to TOC

Longer Term

- add automatic Lua code time slicing support by yielding and resuming the Lua VM actively via Lua's debug hooks.
- add `stat` mode similar to mod_lua.

Back to TOC

Changes

The changes of every release of this module can be obtained from the ngx_openresty bundle's change logs:

http://openresty.org/#Changes

Back to TOC

Test Suite

The following dependencies are required to run the test suite:

- Nginx version >= 1.4.2

- Perl modules:

  - Test::Nginx: http://github.com/openresty/test-nginx

- Nginx modules:

  - ngx_devel_kit
  - ngx_set_misc
  - ngx_auth_request (this is not needed if you're using Nginx 1.5.4+.

    - ngx_echo
    - ngx_memc
    - ngx_srcache
    - ngx_lua (i.e., this module)
    - ngx_lua_upstream
    - ngx_headers_more
    - ngx_drizzle
    - ngx_rds_json
    - ngx_coolkit
    - ngx_redis2

The order in which these modules are added during configuration is important because the position of any filter module in the filtering chain determines the final output, for example. The correct adding order is shown above.

- 3rd-party Lua libraries:
    - lua-cjson
- Applications:
    - mysql: create database 'ngx_test', grant all privileges to user 'ngx_test', password is 'ngx_test'
    - memcached: listening on the default port, 11211.
    - redis: listening on the default port, 6379.

See also the developer build script for more details on setting up the testing environment.

To run the whole test suite in the default testing mode:

```
cd /path/to/lua-nginx-module
export PATH=/path/to/your/nginx/sbin:$PATH
prove -I/path/to/test-nginx/lib -r t
```

To run specific test files:

```
cd /path/to/lua-nginx-module
export PATH=/path/to/your/nginx/sbin:$PATH
prove -I/path/to/test-nginx/lib t/002-content.t t/003-errors.t
```

To run a specific test block in a particular test file, add the line `--- ONLY` to the test block you want to run, and then use the `prove` utility to run that `.t` file.

There are also various testing modes based on mockeagain, valgrind, and etc. Refer to the Test::Nginx documentation for more details for various advanced testing modes. See also the test reports for the Nginx test cluster running on Amazon EC2: http://qa.openresty.org.

Back to TOC

Copyright and License

This module is licensed under the BSD license.

Copyright (C) 2009-2014, by Xiaozhe Wang (chaoslawful) chaoslawful@gmail.com.

Back to TOC

See Also

- lua-resty-memcached library based on ngx_lua cosocket.
- lua-resty-redis library based on ngx_lua cosocket.
- lua-resty-mysql library based on ngx_lua cosocket.
- lua-resty-upload library based on ngx_lua cosocket.
- lua-resty-dns library based on ngx_lua cosocket.
- lua-resty-websocket library for both WebSocket server and client, based on ngx_lua cosocket.
- lua-resty-string library based on LuaJIT FFI.
- lua-resty-lock library for a nonblocking simple lock API.
- Routing requests to different MySQL queries based on URI arguments
- Dynamic Routing Based on Redis and Lua
- Using LuaRocks with ngx_lua
- Introduction to ngx_lua
- ngx_devel_kit
- echo-nginx-module
- drizzle-nginx-module
- postgres-nginx-module
- memc-nginx-module
- The ngx_openresty bundle
- Nginx Systemtap Toolkit

Back to TOC

Directives

Back to TOC

lua_use_default_type

syntax: *lua_use_default_type on | off*

default: *lua_use_default_type on*

context: *http, server, location, location if*

Specifies whether to use the MIME type specified by the default_type directive for the default value of the `Content-Type` response header. If you do not want a default `Content-Type` response header for your Lua request handlers, then turn this directive off.

This directive is turned on by default.

This directive was first introduced in the `v0.9.1` release.

Back to TOC

lua_code_cache

syntax: *lua_code_cache on* | *off*

default: *lua_code_cache on*

context: *http, server, location, location if*

Enables or disables the Lua code cache for Lua code in `*_by_lua_file` directives (like set_by_lua_file and content_by_lua_file) and Lua modules.

When turning off, every request served by ngx_lua will run in a separate Lua VM instance, starting from the `0.9.3` release. So the Lua files referenced in set_by_lua_file, content_by_lua_file, access_by_lua_file, and etc will not be cached and all Lua modules used will be loaded from scratch. With this in place, developers can adopt an edit-and-refresh approach.

Please note however, that Lua code written inlined within nginx.conf such as those specified by set_by_lua, content_by_lua, access_by_lua, and rewrite_by_lua will not be updated when you edit the inlined Lua code in your `nginx.conf` file because only the Nginx config file parser can correctly parse the `nginx.conf` file and the only way is to reload the config file by sending a `HUP` signal or just to restart Nginx.

Even when the code cache is enabled, Lua files which are loaded by `dofile` or `loadfile` in *_by_lua_file cannot be cached (unless you cache the results yourself). Usually you can either use the init_by_lua or init_by_lua_file directives to load all such files or just make these Lua files true Lua modules and load them via `require` .

The ngx_lua module does not support the `stat` mode available with the Apache `mod_lua` module (yet).

Disabling the Lua code cache is strongly discouraged for production use and should only be used during development as it has a significant negative impact on overall performance. For example, the performance a "hello world" Lua example can drop by an order of magnitude after disabling the Lua code cache.

Back to TOC

lua_regex_cache_max_entries

syntax: *lua_regex_cache_max_entries <num>*

default: *lua_regex_cache_max_entries 1024*

context: *http*

Specifies the maximum number of entries allowed in the worker process level compiled regex cache.

The regular expressions used in ngx.re.match, ngx.re.gmatch, ngx.re.sub, and ngx.re.gsub will be cached within this cache if the regex option `o` (i.e., compile-once flag) is specified.

The default number of entries allowed is 1024 and when this limit is reached, new regular expressions will not be cached (as if the `o` option was not specified) and there will be one, and only one, warning in the `error.log` file:

```
2011/08/27 23:18:26 [warn] 31997#0: *1 lua exceeding regex cache max entries (1024), ..
```

Do not activate the `o` option for regular expressions (and/or `replace` string arguments for ngx.re.sub and ngx.re.gsub) that are generated *on the fly* and give rise to infinite variations to avoid hitting the specified limit.

Back to TOC

lua_regex_match_limit

syntax: *lua_regex_match_limit <num>*

default: *lua_regex_match_limit 0*

context: *http*

Specifies the "match limit" used by the PCRE library when executing the ngx.re API. To quote the PCRE manpage, "the limit ... has the effect of limiting the amount of backtracking that can take place."

When the limit is hit, the error string "pcre_exec() failed: -8" will be returned by the ngx.re API functions on the Lua land.

When setting the limit to 0, the default "match limit" when compiling the PCRE library is used. And this is the default value of this directive.

This directive was first introduced in the `v0.8.5` release.

Back to TOC

lua_package_path

syntax: *lua_package_path <lua-style-path-str>*

default: *The content of LUA_PATH environ variable or Lua's compiled-in defaults.*

context: *http*

Sets the Lua module search path used by scripts specified by set_by_lua, content_by_lua and others. The path string is in standard Lua path form, and `;;` can be used to stand for the original search paths.

As from the `v0.5.0rc29` release, the special notation `$prefix` or `${prefix}` can be used in the search path string to indicate the path of the `server prefix` usually determined by the `-p PATH` command-line option while starting the Nginx server.

Back to TOC

lua_package_cpath

syntax: *lua_package_cpath <lua-style-cpath-str>*

default: *The content of LUA_CPATH environment variable or Lua's compiled-in defaults.*

context: *http*

Sets the Lua C-module search path used by scripts specified by set_by_lua, content_by_lua and others. The cpath string is in standard Lua cpath form, and `;;` can be used to stand for the original cpath.

As from the `v0.5.0rc29` release, the special notation `$prefix` or `${prefix}` can be used in the search path string to indicate the path of the `server prefix` usually determined by the `-p PATH` command-line option while starting the Nginx server.

Back to TOC

init_by_lua

syntax: *init_by_lua <lua-script-str>*

context: *http*

phase: *loading-config*

Runs the Lua code specified by the argument `<lua-script-str>` on the global Lua VM level when the Nginx master process (if any) is loading the Nginx config file.

When Nginx receives the `HUP` signal and starts reloading the config file, the Lua VM will also be re-created and `init_by_lua` will run again on the new Lua VM. In case that the lua_code_cache directive is turned off (default on), the `init_by_lua` handler will run upon every request because in this special mode a standalone Lua VM is always created for each request.

Usually you can register (true) Lua global variables or pre-load Lua modules at server start-up by means of this hook. Here is an example for pre-loading Lua modules:

```
init_by_lua 'cjson = require "cjson"';

server {
    location = /api {
        content_by_lua '
            ngx.say(cjson.encode({dog = 5, cat = 6}))
        ';
    }
}
```

You can also initialize the lua_shared_dict shm storage at this phase. Here is an example for this:

```
lua_shared_dict dogs 1m;

init_by_lua '
    local dogs = ngx.shared.dogs;
    dogs:set("Tom", 56)
';

server {
    location = /api {
        content_by_lua '
            local dogs = ngx.shared.dogs;
            ngx.say(dogs:get("Tom"))
        ';
    }
}
```

But note that, the lua_shared_dict's shm storage will not be cleared through a config reload (via the `HUP` signal, for example). So if you do *not* want to re-initialize the shm storage in your `init_by_lua` code in this case, then you just need to set a custom flag in the shm storage and always check the flag in your `init_by_lua` code.

Because the Lua code in this context runs before Nginx forks its worker processes (if any), data or code loaded here will enjoy the Copy-on-write (COW) feature provided by many operating systems among all the worker processes, thus saving a lot of memory.

Do *not* initialize your own Lua global variables in this context because use of Lua global variables have performance penalties and can lead to global namespace pollution (see the Lua Variable Scope section for more details). The recommended way is to use proper Lua module files (but do not use the standard Lua function module() to define Lua modules because it pollutes the global namespace as well) and call require() to load your own module files in `init_by_lua` or other contexts (require() does cache the loaded Lua modules in the global `package.loaded` table in the Lua registry so your modules will only loaded once for the whole Lua VM instance).

Only a small set of the Nginx API for Lua is supported in this context:

- Logging APIs: ngx.log and print,
- Shared Dictionary API: ngx.shared.DICT.

More Nginx APIs for Lua may be supported in this context upon future user requests.

Basically you can safely use Lua libraries that do blocking I/O in this very context because blocking the master process during server start-up is completely okay. Even the Nginx core does blocking I/O (at least on resolving upstream's host names) at the configure-loading phase.

You should be very careful about potential security vulnerabilities in your Lua code registered in this context because the Nginx master process is often run under the `root` account.

This directive was first introduced in the `v0.5.5` release.

Back to TOC

init_by_lua_file

syntax: *init_by_lua_file <path-to-lua-script-file>*

context: *http*

phase: *loading-config*

Equivalent to init_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code or Lua/LuaJIT bytecode to be executed.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

This directive was first introduced in the `v0.5.5` release.

Back to TOC

init_worker_by_lua

syntax: *init_worker_by_lua <lua-script-str>*

context: *http*

phase: *starting-worker*

Runs the specified Lua code upon every Nginx worker process's startup when the master process is enabled. When the master process is disabled, this hook will just run after init_by_lua*.

This hook is often used to create per-worker reoccurring timers (via the ngx.timer.at Lua API), either for backend healthcheck or other timed routine work. Below is an example,

```
init_worker_by_lua '
    local delay = 3  -- in seconds
    local new_timer = ngx.timer.at
    local log = ngx.log
    local ERR = ngx.ERR
    local check

    check = function(premature)
        if not premature then
            -- do the health check or other routine work
            local ok, err = new_timer(delay, check)
            if not ok then
                log(ERR, "failed to create timer: ", err)
                return
            end
        end
    end

    local ok, err = new_timer(delay, check)
    if not ok then
        log(ERR, "failed to create timer: ", err)
        return
    end
';
```

This directive was first introduced in the `v0.9.5` release.

Back to TOC

init_worker_by_lua_file

syntax: *init_worker_by_lua_file <lua-file-path>*

context: *http*

phase: *starting-worker*

Similar to init_worker_by_lua, but accepts the file path to a Lua source file or Lua bytecode file.

This directive was first introduced in the `v0.9.5` release.

Back to TOC

set_by_lua

syntax: *set_by_lua $res <lua-script-str> [$arg1 $arg2 ...]*

context: *server, server if, location, location if*

phase: *rewrite*

Executes code specified in `<lua-script-str>` with optional input arguments `$arg1 $arg2 ...`, and returns string output to `$res`. The code in `<lua-script-str>` can make API calls and can retrieve input arguments from the `ngx.arg` table (index starts from `1` and increases sequentially).

This directive is designed to execute short, fast running code blocks as the Nginx event loop is blocked during code execution. Time consuming code sequences should therefore be avoided.

This directive is implemented by injecting custom commands into the standard ngx_http_rewrite_module's command list. Because ngx_http_rewrite_module does not support nonblocking I/O in its commands, Lua APIs requiring yielding the current Lua "light thread" cannot work in this directive.

At least the following API functions are currently disabled within the context of `set_by_lua`:

- Output API functions (e.g., ngx.say and ngx.send_headers)
- Control API functions (e.g., ngx.exit)
- Subrequest API functions (e.g., ngx.location.capture and ngx.location.capture_multi)
- Cosocket API functions (e.g., ngx.socket.tcp and ngx.req.socket).
- Sleeping API function ngx.sleep.

In addition, note that this directive can only write out a value to a single Nginx variable at a time. However, a workaround is possible using the ngx.var.VARIABLE interface.

```
location /foo {
    set $diff ''; # we have to predefine the $diff variable here

    set_by_lua $sum '
        local a = 32
        local b = 56

        ngx.var.diff = a - b;  -- write to $diff directly
        return a + b;          -- return the $sum value normally
    ';

    echo "sum = $sum, diff = $diff";
}
```

This directive can be freely mixed with all directives of the ngx_http_rewrite_module, set-misc-nginx-module, and array-var-nginx-module modules. All of these directives will run in the same order as they appear in the config file.

```
set $foo 32;
set_by_lua $bar 'tonumber(ngx.var.foo) + 1';
set $baz "bar: $bar";  # $baz == "bar: 33"
```

As from the `v0.5.0rc29` release, Nginx variable interpolation is disabled in the `<lua-script-str>` argument of this directive and therefore, the dollar sign character (`$`) can be used directly.

This directive requires the ngx_devel_kit module.

Back to TOC

set_by_lua_file

syntax: *set_by_lua_file $res <path-to-lua-script-file> [$arg1 $arg2 ...]*

context: *server, server if, location, location if*

phase: *rewrite*

Equivalent to set_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code, or, as from the `v0.5.0rc32` release, the Lua/LuaJIT bytecode to be executed.

Nginx variable interpolation is supported in the `<path-to-lua-script-file>` argument string of this directive. But special care must be taken for injection attacks.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

When the Lua code cache is turned on (by default), the user code is loaded once at the first request and cached and the Nginx config must be reloaded each time the Lua source file is modified. The Lua code cache can be temporarily disabled during development by switching lua_code_cache `off` in `nginx.conf` to avoid reloading Nginx.

This directive requires the ngx_devel_kit module.

Back to TOC

content_by_lua

syntax: *content_by_lua <lua-script-str>*

context: *location, location if*

phase: *content*

Acts as a "content handler" and executes Lua code string specified in `<lua-script-str>` for every request. The Lua code may make API calls and is executed as a new spawned coroutine in an independent global environment (i.e. a sandbox).

Do not use this directive and other content handler directives in the same location. For example, this directive and the proxy_pass directive should not be used in the same location.

Back to TOC

content_by_lua_file

syntax: *content_by_lua_file <path-to-lua-script-file>*

context: *location, location if*

phase: *content*

Equivalent to content_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code, or, as from the `v0.5.0rc32` release, the Lua/LuaJIT bytecode to be executed.

Nginx variables can be used in the `<path-to-lua-script-file>` string to provide flexibility. This however carries some risks and is not ordinarily recommended.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

When the Lua code cache is turned on (by default), the user code is loaded once at the first request and cached and the Nginx config must be reloaded each time the Lua source file is modified. The Lua code cache can be temporarily disabled during development by switching lua_code_cache `off` in `nginx.conf` to avoid reloading Nginx.

Back to TOC

rewrite_by_lua

syntax: *rewrite_by_lua <lua-script-str>*

context: *http, server, location, location if*

phase: *rewrite tail*

Acts as a rewrite phase handler and executes Lua code string specified in `<lua-script-str>` for every request. The Lua code may make API calls and is executed as a new spawned coroutine in an independent global environment (i.e. a sandbox).

Note that this handler always runs *after* the standard ngx_http_rewrite_module. So the following will work as expected:

```
location /foo {
    set $a 12; # create and initialize $a
    set $b ""; # create and initialize $b
    rewrite_by_lua 'ngx.var.b = tonumber(ngx.var.a) + 1';
    echo "res = $b";
}
```

because `set $a 12` and `set $b ""` run *before* rewrite_by_lua.

On the other hand, the following will not work as expected:

```
?  location /foo {
?      set $a 12; # create and initialize $a
?      set $b ''; # create and initialize $b
?      rewrite_by_lua 'ngx.var.b = tonumber(ngx.var.a) + 1';
?      if ($b = '13') {
?         rewrite ^ /bar redirect;
?         break;
?      }
?
?      echo "res = $b";
?  }
```

because `if` runs *before* rewrite_by_lua even if it is placed after rewrite_by_lua in the config.

The right way of doing this is as follows:

```
location /foo {
    set $a 12; # create and initialize $a
```

```
        set $b ''; # create and initialize $b
        rewrite_by_lua '
            ngx.var.b = tonumber(ngx.var.a) + 1
            if tonumber(ngx.var.b) == 13 then
                return ngx.redirect("/bar");
            end
        ';

        echo "res = $b";
    }
```

Note that the ngx_eval module can be approximated by using rewrite_by_lua. For example,

```
    location / {
        eval $res {
            proxy_pass http://foo.com/check-spam;
        }

        if ($res = 'spam') {
            rewrite ^ /terms-of-use.html redirect;
        }

        fastcgi_pass ...;
    }
```

can be implemented in ngx_lua as:

```
    location = /check-spam {
        internal;
        proxy_pass http://foo.com/check-spam;
    }

    location / {
        rewrite_by_lua '
            local res = ngx.location.capture("/check-spam")
            if res.body == "spam" then
                return ngx.redirect("/terms-of-use.html")
            end
        ';

        fastcgi_pass ...;
    }
```

Just as any other rewrite phase handlers, rewrite_by_lua also runs in subrequests.

Note that when calling `ngx.exit(ngx.OK)` within a rewrite_by_lua handler, the nginx request processing control flow will still continue to the content handler. To terminate the current request from within a rewrite_by_lua handler, calling ngx.exit with status >= 200 (`ngx.HTTP_OK`) and status < 300 (`ngx.HTTP_SPECIAL_RESPONSE`) for successful quits and `ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)` (or its friends) for failures.

If the ngx_http_rewrite_module's rewrite directive is used to change the URI and initiate location re-lookups (internal redirections), then any rewrite_by_lua or rewrite_by_lua_file code sequences within the current location will not be executed. For example,

```
    location /foo {
        rewrite ^ /bar;
        rewrite_by_lua 'ngx.exit(503)';
    }
    location /bar {
        ...
    }
```

Here the Lua code `ngx.exit(503)` will never run. This will be the case if `rewrite ^ /bar last` is used as this will similarly initiate an internal redirection. If the `break` modifier is used instead, there will be no internal redirection and the `rewrite_by_lua` code will be executed.

The `rewrite_by_lua` code will always run at the end of the `rewrite` request-processing phase unless rewrite_by_lua_no_postpone is turned on.

Back to TOC

rewrite_by_lua_file

syntax: *rewrite_by_lua_file <path-to-lua-script-file>*

context: *http, server, location, location if*

phase: *rewrite tail*

Equivalent to rewrite_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code, or, as from the `v0.5.0rc32` release, the Lua/LuaJIT bytecode to be executed.

Nginx variables can be used in the `<path-to-lua-script-file>` string to provide flexibility. This however carries some risks and is not ordinarily recommended.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

When the Lua code cache is turned on (by default), the user code is loaded once at the first request and cached and the Nginx config must be reloaded each time the Lua source file is modified. The Lua code cache can be temporarily disabled during development by switching lua_code_cache `off` in `nginx.conf` to avoid reloading Nginx.

The `rewrite_by_lua_file` code will always run at the end of the `rewrite` request-processing phase unless rewrite_by_lua_no_postpone is turned on.

Back to TOC

access_by_lua

syntax: *access_by_lua <lua-script-str>*

context: *http, server, location, location if*

phase: *access tail*

Acts as an access phase handler and executes Lua code string specified in `<lua-script-str>` for every request. The Lua code may make API calls and is executed as a new spawned coroutine in an independent global environment (i.e. a sandbox).

Note that this handler always runs *after* the standard ngx_http_access_module. So the following will work as expected:

```
location / {
    deny    192.168.1.1;
```

```
        allow   192.168.1.0/24;
        allow   10.1.1.0/16;
        deny    all;

        access_by_lua '
            local res = ngx.location.capture("/mysql", { ... })
            ...
        ';

        # proxy_pass/fastcgi_pass/...
    }
```

That is, if a client IP address is in the blacklist, it will be denied before the MySQL query for more complex authentication is executed by access_by_lua.

Note that the ngx_auth_request module can be approximated by using access_by_lua:

```
    location / {
        auth_request /auth;

        # proxy_pass/fastcgi_pass/postgres_pass/...
    }
```

can be implemented in ngx_lua as:

```
    location / {
        access_by_lua '
            local res = ngx.location.capture("/auth")

            if res.status == ngx.HTTP_OK then
                return
            end

            if res.status == ngx.HTTP_FORBIDDEN then
                ngx.exit(res.status)
            end

            ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)
        ';

        # proxy_pass/fastcgi_pass/postgres_pass/...
    }
```

As with other access phase handlers, access_by_lua will *not* run in subrequests.

Note that when calling `ngx.exit(ngx.OK)` within a access_by_lua handler, the nginx request processing control flow will still continue to the content handler. To terminate the current request from within a access_by_lua handler, calling ngx.exit with status >= 200 (`ngx.HTTP_OK`) and status < 300 (`ngx.HTTP_SPECIAL_RESPONSE`) for successful quits and `ngx.exit(ngx.HTTP_INTERNAL_SERVER_ERROR)` (or its friends) for failures.

Back to TOC

access_by_lua_file

syntax: *access_by_lua_file <path-to-lua-script-file>*

context: *http, server, location, location if*

phase: *access tail*

Equivalent to access_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code, or, as from the `v0.5.0rc32` release, the Lua/LuaJIT bytecode to be executed.

Nginx variables can be used in the `<path-to-lua-script-file>` string to provide flexibility. This however carries some risks and is not ordinarily recommended.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

When the Lua code cache is turned on (by default), the user code is loaded once at the first request and cached and the Nginx config must be reloaded each time the Lua source file is modified. The Lua code cache can be temporarily disabled during development by switching lua_code_cache `off` in `nginx.conf` to avoid repeatedly reloading Nginx.

Back to TOC

header_filter_by_lua

syntax: *header_filter_by_lua <lua-script-str>*

context: *http, server, location, location if*

phase: *output-header-filter*

Uses Lua code specified in `<lua-script-str>` to define an output header filter.

Note that the following API functions are currently disabled within this context:

- Output API functions (e.g., ngx.say and ngx.send_headers)
- Control API functions (e.g., ngx.exit and ngx.exec)
- Subrequest API functions (e.g., ngx.location.capture and ngx.location.capture_multi)
- Cosocket API functions (e.g., ngx.socket.tcp and ngx.req.socket).

Here is an example of overriding a response header (or adding one if absent) in our Lua header filter:

```
location / {
    proxy_pass http://mybackend;
    header_filter_by_lua 'ngx.header.Foo = "blah"';
}
```

This directive was first introduced in the `v0.2.1rc20` release.

Back to TOC

header_filter_by_lua_file

syntax: *header_filter_by_lua_file <path-to-lua-script-file>*

context: *http, server, location, location if*

phase: *output-header-filter*

Equivalent to header_filter_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code, or as from the `v0.5.0rc32` release, the Lua/LuaJIT bytecode to be executed.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

This directive was first introduced in the `v0.2.1rc20` release.

Back to TOC

body_filter_by_lua

syntax: *body_filter_by_lua <lua-script-str>*

context: *http, server, location, location if*

phase: *output-body-filter*

Uses Lua code specified in `<lua-script-str>` to define an output body filter.

The input data chunk is passed via ngx.arg1 and the "eof" flag indicating the end of the response body data stream is passed via ngx.arg2.

Behind the scene, the "eof" flag is just the `last_buf` (for main requests) or `last_in_chain` (for subrequests) flag of the Nginx chain link buffers. (Before the `v0.7.14` release, the "eof" flag does not work at all in subrequests.)

The output data stream can be aborted immediately by running the following Lua statement:

```
return ngx.ERROR
```

This will truncate the response body and usually result in incomplete and also invalid responses.

The Lua code can pass its own modified version of the input data chunk to the downstream Nginx output body filters by overriding ngx.arg[1] with a Lua string or a Lua table of strings. For example, to transform all the lowercase letters in the response body, we can just write:

```
location / {
    proxy_pass http://mybackend;
    body_filter_by_lua 'ngx.arg[1] = string.upper(ngx.arg[1])';
}
```

When setting `nil` or an empty Lua string value to `ngx.arg[1]`, no data chunk will be passed to the downstream Nginx output filters at all.

Likewise, new "eof" flag can also be specified by setting a boolean value to ngx.arg[2]. For example,

```
location /t {
    echo hello world;
    echo hiya globe;

    body_filter_by_lua '
        local chunk = ngx.arg[1]
        if string.match(chunk, "hello") then
            ngx.arg[2] = true  -- new eof
            return
        end

        -- just throw away any remaining chunk data
        ngx.arg[1] = nil
    ';
}
```

Then `GET /t` will just return the output

```
hello world
```

That is, when the body filter sees a chunk containing the word "hello", then it will set the "eof" flag to true immediately, resulting in truncated but still valid responses.

When the Lua code may change the length of the response body, then it is required to always clear out the `Content-Length` response header (if any) in a header filter to enforce streaming output, as in

```
location /foo {
    # fastcgi_pass/proxy_pass/...

    header_filter_by_lua 'ngx.header.content_length = nil';
    body_filter_by_lua 'ngx.arg[1] = string.len(ngx.arg[1]) .. "\\n"';
}
```

Note that the following API functions are currently disabled within this context due to the limitations in NGINX output filter's current implementation:

- Output API functions (e.g., ngx.say and ngx.send_headers)
- Control API functions (e.g., ngx.exit and ngx.exec)
- Subrequest API functions (e.g., ngx.location.capture and ngx.location.capture_multi)
- Cosocket API functions (e.g., ngx.socket.tcp and ngx.req.socket).

Nginx output filters may be called multiple times for a single request because response body may be delivered in chunks. Thus, the Lua code specified by in this directive may also run multiple times in the lifetime of a single HTTP request.

This directive was first introduced in the `v0.5.0rc32` release.

Back to TOC

body_filter_by_lua_file

syntax: *body_filter_by_lua_file <path-to-lua-script-file>*

context: *http, server, location, location if*

phase: *output-body-filter*

Equivalent to body_filter_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code, or, as from the `v0.5.0rc32` release, the Lua/LuaJIT bytecode to be executed.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

This directive was first introduced in the `v0.5.0rc32` release.

Back to TOC

log_by_lua

syntax: *log_by_lua <lua-script-str>*

context: *http, server, location, location if*

phase: *log*

Run the Lua source code inlined as the `<lua-script-str>` at the `log` request processing phase. This does not replace the current access logs, but runs after.

Note that the following API functions are currently disabled within this context:

- Output API functions (e.g., ngx.say and ngx.send_headers)
- Control API functions (e.g., ngx.exit)
- Subrequest API functions (e.g., ngx.location.capture and ngx.location.capture_multi)
- Cosocket API functions (e.g., ngx.socket.tcp and ngx.req.socket).

Here is an example of gathering average data for $upstream_response_time:

```
lua_shared_dict log_dict 5M;

server {
    location / {
        proxy_pass http://mybackend;

        log_by_lua '
            local log_dict = ngx.shared.log_dict
            local upstream_time = tonumber(ngx.var.upstream_response_time)

            local sum = log_dict:get("upstream_time-sum") or 0
            sum = sum + upstream_time
            log_dict:set("upstream_time-sum", sum)

            local newval, err = log_dict:incr("upstream_time-nb", 1)
            if not newval and err == "not found" then
                log_dict:add("upstream_time-nb", 0)
                log_dict:incr("upstream_time-nb", 1)
            end
        ';
    }

    location = /status {
        content_by_lua '
            local log_dict = ngx.shared.log_dict
            local sum = log_dict:get("upstream_time-sum")
            local nb = log_dict:get("upstream_time-nb")

            if nb and sum then
                ngx.say("average upstream response time: ", sum / nb,
                        " (", nb, " reqs)")
            else
                ngx.say("no data yet")
            end
        ';
    }
}
```

This directive was first introduced in the `v0.5.0rc31` release.

Back to TOC

log_by_lua_file

syntax: *log_by_lua_file <path-to-lua-script-file>*

context: *http, server, location, location if*

phase: *log*

Equivalent to log_by_lua, except that the file specified by `<path-to-lua-script-file>` contains the Lua code, or, as from the `v0.5.0rc32` release, the Lua/LuaJIT bytecode to be executed.

When a relative path like `foo/bar.lua` is given, they will be turned into the absolute path relative to the `server prefix` path determined by the `-p PATH` command-line option while starting the Nginx server.

This directive was first introduced in the `v0.5.0rc31` release.

Back to TOC

lua_need_request_body

syntax: *lua_need_request_body <on|off>*

default: *off*

context: *main | server | location*

phase: *depends on usage*

Determines whether to force the request body data to be read before running rewrite/access/access_by_lua* or not. The Nginx core does not read the client request body by default and if request body data is required, then this directive should be turned `on` or the ngx.req.read_body function should be called within the Lua code.

To read the request body data within the $request_body variable, client_body_buffer_size must have the same value as client_max_body_size. Because when the content length exceeds client_body_buffer_size but less than client_max_body_size, Nginx will buffer the data into a temporary file on the disk, which will lead to empty value in the $request_body variable.

If the current location includes rewrite_by_lua or rewrite_by_lua_file directives, then the request body will be read just before the rewrite_by_lua or rewrite_by_lua_file code is run (and also at the `rewrite` phase). Similarly, if only content_by_lua is specified, the request body will not be read until the content handler's Lua code is about to run (i.e., the request body will be read during the content phase).

It is recommended however, to use the ngx.req.read_body and ngx.req.discard_body functions for finer control over the request body reading process instead.

This also applies to access_by_lua and access_by_lua_file.

Back to TOC

lua_shared_dict

syntax: *lua_shared_dict <name> <size>*

default: *no*

context: *http*

phase: *depends on usage*

Declares a shared memory zone, `<name>` , to serve as storage for the shm based Lua dictionary `ngx.shared.<name>` .

Shared memory zones are always shared by all the nginx worker processes in the current nginx server instance.

The `<size>` argument accepts size units such as `k` and `m` :

```
http {
    lua_shared_dict dogs 10m;
    ...
}
```

See ngx.shared.DICT for details.

This directive was first introduced in the `v0.3.1rc22` release.

Back to TOC

lua_socket_connect_timeout

syntax: *lua_socket_connect_timeout <time>*

default: *lua_socket_connect_timeout 60s*

context: *http, server, location*

This directive controls the default timeout value used in TCP/unix-domain socket object's connect method and can be overridden by the settimeout method.

The `<time>` argument can be an integer, with an optional time unit, like `s` (second), `ms` (millisecond), `m` (minute). The default time unit is `s` , i.e., "second". The default setting is `60s` .

This directive was first introduced in the `v0.5.0rc1` release.

Back to TOC

lua_socket_send_timeout

syntax: *lua_socket_send_timeout <time>*

default: *lua_socket_send_timeout 60s*

context: *http, server, location*

Controls the default timeout value used in TCP/unix-domain socket object's send method and can be overridden by the settimeout method.

The `<time>` argument can be an integer, with an optional time unit, like `s` (second), `ms` (millisecond), `m` (minute). The default time unit is `s` , i.e., "second". The default setting is `60s` .

This directive was first introduced in the `v0.5.0rc1` release.

Back to TOC

lua_socket_send_lowat

syntax: *lua_socket_send_lowat <size>*

default: *lua_socket_send_lowat 0*

context: *http, server, location*

Controls the `lowat` (low water) value for the cosocket send buffer.

Back to TOC

lua_socket_read_timeout

syntax: *lua_socket_read_timeout <time>*

default: *lua_socket_read_timeout 60s*

context: *http, server, location*

phase: *depends on usage*

This directive controls the default timeout value used in TCP/unix-domain socket object's receive method and iterator functions returned by the receiveuntil method. This setting can be overridden by the settimeout method.

The `<time>` argument can be an integer, with an optional time unit, like `s` (second), `ms` (millisecond), `m` (minute). The default time unit is `s` , i.e., "second". The default setting is `60s` .

This directive was first introduced in the `v0.5.0rc1` release.

Back to TOC

lua_socket_buffer_size

syntax: *lua_socket_buffer_size <size>*

default: *lua_socket_buffer_size 4k/8k*

context: *http, server, location*

Specifies the buffer size used by cosocket reading operations.

This buffer does not have to be that big to hold everything at the same time because cosocket supports 100% non-buffered reading and parsing. So even `1` byte buffer size should still work everywhere but the performance could be terrible.

This directive was first introduced in the `v0.5.0rc1` release.

Back to TOC

lua_socket_pool_size

syntax: *lua_socket_pool_size <size>*

default: *lua_socket_pool_size 30*

context: *http, server, location*

Specifies the size limit (in terms of connection count) for every cosocket connection pool associated with every remote server (i.e., identified by either the host-port pair or the unix domain socket file path).

Default to 30 connections for every pool.

When the connection pool exceeds the available size limit, the least recently used (idle) connection already in the pool will be closed to make room for the current connection.

Note that the cosocket connection pool is per nginx worker process rather than per nginx server instance, so size limit specified here also applies to every single nginx worker process.

This directive was first introduced in the `v0.5.0rc1` release.

Back to TOC

lua_socket_keepalive_timeout

syntax: *lua_socket_keepalive_timeout <time>*

default: *lua_socket_keepalive_timeout 60s*

context: *http, server, location*

This directive controls the default maximal idle time of the connections in the cosocket built-in connection pool. When this timeout reaches, idle connections will be closed and removed from the pool. This setting can be overridden by cosocket objects' setkeepalive method.

The `<time>` argument can be an integer, with an optional time unit, like `s` (second), `ms` (millisecond), `m` (minute). The default time unit is `s`, i.e., "second". The default setting is `60s`.

This directive was first introduced in the `v0.5.0rc1` release.

Back to TOC

lua_socket_log_errors

syntax: *lua_socket_log_errors on|off*

default: *lua_socket_log_errors on*

context: *http, server, location*

This directive can be used to toggle error logging when a failure occurs for the TCP or UDP cosockets. If you are already doing proper error handling and logging in your Lua code, then it is recommended to turn this directive off to prevent data flushing in your nginx error log files (which is usually rather expensive).

This directive was first introduced in the `v0.5.13` release.

Back to TOC

lua_ssl_ciphers

syntax: *lua_ssl_ciphers <ciphers>*

default: *lua_ssl_ciphers DEFAULT*

context: *http, server, location*

Specifies the enabled ciphers for requests to a SSL/TLS server in the tcpsock:sslhandshake method. The ciphers are specified in the format understood by the OpenSSL library.

The full list can be viewed using the “openssl ciphers” command.

This directive was first introduced in the `v0.9.11` release.

Back to TOC

lua_ssl_crl

syntax: *lua_ssl_crl <file>*

default: *no*

context: *http, server, location*

Specifies a file with revoked certificates (CRL) in the PEM format used to verify the certificate of the SSL/TLS server in the tcpsock:sslhandshake method.

This directive was first introduced in the `v0.9.11` release.

Back to TOC

lua_ssl_protocols

syntax: *lua_ssl_protocols [SSLv2] [SSLv3] [TLSv1] [TLSv1.1] [TLSv1.2]*

default: *lua_ssl_protocols SSLv3 TLSv1 TLSv1.1 TLSv1.2*

context: *http, server, location*

Enables the specified protocols for requests to a SSL/TLS server in the tcpsock:sslhandshake method.

This directive was first introduced in the `v0.9.11` release.

Back to TOC

lua_ssl_trusted_certificate

syntax: *lua_ssl_trusted_certificate <file>*

default: *no*

context: *http, server, location*

Specifies a file path with trusted CA certificates in the PEM format used to verify the certificate of the SSL/TLS server in the tcpsock:sslhandshake method.

This directive was first introduced in the `v0.9.11` release.

See also lua_ssl_verify_depth.

Back to TOC

lua_ssl_verify_depth

syntax: *lua_ssl_verify_depth <number>*

default: *lua_ssl_verify_depth 1*

context: *http, server, location*

Sets the verification depth in the server certificates chain.

This directive was first introduced in the `v0.9.11` release.

See also lua_ssl_trusted_certificate.

Back to TOC

lua_http10_buffering

syntax: *lua_http10_buffering on|off*

default: *lua_http10_buffering on*

context: *http, server, location, location-if*

Enables or disables automatic response buffering for HTTP 1.0 (or older) requests. This buffering mechanism is mainly used for HTTP 1.0 keep-alive which replies on a proper `Content-Length` response header.

If the Lua code explicitly sets a `Content-Length` response header before sending the headers (either explicitly via ngx.send_headers or implicitly via the first ngx.say or ngx.print call), then the HTTP 1.0 response buffering will be disabled even when this directive is turned on.

To output very large response data in a streaming fashion (via the ngx.flush call, for example), this directive MUST be turned off to minimize memory usage.

This directive is turned `on` by default.

This directive was first introduced in the `v0.5.0rc19` release.

Back to TOC

rewrite_by_lua_no_postpone

syntax: *rewrite_by_lua_no_postpone on|off*

default: *rewrite_by_lua_no_postpone off*

context: *http*

Controls whether or not to disable postponing rewrite_by_lua and rewrite_by_lua_file directives to run at the end of the `rewrite` request-processing phase. By default, this directive is turned off and the Lua code is postponed to run at the end of the `rewrite` phase.

This directive was first introduced in the `v0.5.0rc29` release.

Back to TOC

lua_transform_underscores_in_response_headers

syntax: *lua_transform_underscores_in_response_headers on|off*

default: *lua_transform_underscores_in_response_headers on*

context: *http, server, location, location-if*

Controls whether to transform underscores (`_`) in the response header names specified in the ngx.header.HEADER API to hypens (`-`).

This directive was first introduced in the `v0.5.0rc32` release.

Back to TOC

lua_check_client_abort

syntax: *lua_check_client_abort on|off*

default: *lua_check_client_abort off*

context: *http, server, location, location-if*

This directive controls whether to check for premature client connection abortion.

When this directive is turned on, the ngx_lua module will monitor the premature connection close event on the downstream connections. And when there is such an event, it will call the user Lua function callback (registered by ngx.on_abort) or just stop and clean up all the Lua "light threads" running in the current request's request handler when there is no user callback function registered.

According to the current implementation, however, if the client closes the connection before the Lua code finishes reading the request body data via ngx.req.socket, then ngx_lua will neither stop all the running "light threads" nor call the user callback (if ngx.on_abort has been called). Instead, the reading operation on ngx.req.socket will just return the error message "client aborted" as the second return value (the first return value is surely `nil`).

When TCP keepalive is disabled, it is relying on the client side to close the socket gracefully (by sending a `FIN` packet or something like that). For (soft) real-time web applications, it is highly recommended to configure the TCP keepalive support in your system's TCP stack implementation in order to detect "half-open" TCP connections in time.

For example, on Linux, you can configure the standard listen directive in your `nginx.conf` file like this:

```
listen 80 so_keepalive=2s:2s:8;
```

On FreeBSD, you can only tune the system-wide configuration for TCP keepalive, for example:

```
# sysctl net.inet.tcp.keepintvl=2000
# sysctl net.inet.tcp.keepidle=2000
```

This directive was first introduced in the `v0.7.4` release.

See also ngx.on_abort.

Back to TOC

lua_max_pending_timers

syntax: *lua_max_pending_timers <count>*

default: *lua_max_pending_timers 1024*

context: *http*

Controls the maximum number of pending timers allowed.

Pending timers are those timers that have not expired yet.

When exceeding this limit, the ngx.timer.at call will immediately return `nil` and the error string "too many pending timers".

This directive was first introduced in the `v0.8.0` release.

Back to TOC

lua_max_running_timers

syntax: *lua_max_running_timers <count>*

default: *lua_max_running_timers 256*

context: *http*

Controls the maximum number of "running timers" allowed.

Running timers are those timers whose user callback functions are still running.

When exceeding this limit, Nginx will stop running the callbacks of newly expired timers and log an error message "N lua_max_running_timers are not enough" where "N" is the current value of this directive.

This directive was first introduced in the `v0.8.0` release.

Back to TOC

Nginx API for Lua

Back to TOC

Introduction

The various `*_by_lua` and `*_by_lua_file` configuration directives serve as gateways to the Lua API within the `nginx.conf` file. The Nginx Lua API described below can only be called within the user Lua code run in the context of these configuration directives.

The API is exposed to Lua in the form of two standard packages `ngx` and `ndk`. These packages are in the default global scope within ngx_lua and are always available within ngx_lua directives.

The packages can be introduced into external Lua modules like this:

```
local say = ngx.say

local _M = {}

function _M.foo(a)
    say(a)
end

return _M
```

Use of the package.seeall flag is strongly discouraged due to its various bad side-effects.

It is also possible to directly require the packages in external Lua modules:

```
local ngx = require "ngx"
local ndk = require "ndk"
```

The ability to require these packages was introduced in the `v0.2.1rc19` release.

Network I/O operations in user code should only be done through the Nginx Lua API calls as the Nginx event loop may be blocked and performance drop off dramatically otherwise. Disk operations with relatively small amount of data can be done using the standard Lua `io` library but huge file reading and writing should be avoided wherever possible as they may block the Nginx process significantly. Delegating all network and disk I/O operations to Nginx's subrequests (via the ngx.location.capture method and similar) is strongly recommended for maximum performance.

Back to TOC

ngx.arg

syntax: *val = ngx.arg[index]*

context: *set_by_lua*, body_filter_by_lua**

When this is used in the context of the set_by_lua or set_by_lua_file directives, this table is read-only and holds the input arguments to the config directives:

```
value = ngx.arg[n]
```

Here is an example

```
location /foo {
    set $a 32;
    set $b 56;

    set_by_lua $res
        'return tonumber(ngx.arg[1]) + tonumber(ngx.arg[2])'
        $a $b;

    echo $sum;
}
```

that writes out `88`, the sum of `32` and `56`.

When this table is used in the context of body_filter_by_lua or body_filter_by_lua_file, the first element holds the input data chunk to the output filter code and the second element holds the boolean flag for the "eof" flag indicating the end of the whole output data stream.

The data chunk and "eof" flag passed to the downstream Nginx output filters can also be overridden by assigning values directly to the corresponding table elements. When setting `nil` or an empty Lua string value to `ngx.arg[1]`, no data chunk will be passed to the downstream Nginx output filters at all.

Back to TOC

ngx.var.VARIABLE

syntax: *ngx.var.VAR_NAME*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua**

Read and write Nginx variable values.

```
value = ngx.var.some_nginx_variable_name
ngx.var.some_nginx_variable_name = value
```

Note that only already defined nginx variables can be written to. For example:

```
location /foo {
    set $my_var ''; # this line is required to create $my_var at config time
    content_by_lua '
        ngx.var.my_var = 123;
        ...
    ';
}
```

That is, nginx variables cannot be created on-the-fly.

Some special nginx variables like `$args` and `$limit_rate` can be assigned a value, some are not, like `$arg_PARAMETER`.

Nginx regex group capturing variables `$1` , `$2` , `$3` , and etc, can be read by this interface as well, by writing `ngx.var[1]` , `ngx.var[2]` , `ngx.var[3]` , and etc.

Setting `ngx.var.Foo` to a `nil` value will unset the `$Foo` Nginx variable.

```
ngx.var.args = nil
```

WARNING When reading from an Nginx variable, Nginx will allocate memory in the per-request memory pool which is freed only at request termination. So when you need to read from an Nginx variable repeatedly in your Lua code, cache the Nginx variable value to your own Lua variable, for example,

```
local val = ngx.var.some_var
--- use the val repeatedly later
```

to prevent (temporary) memory leaking within the current request's lifetime. Another way of caching the result is to use the ngx.ctx table.

This API requires a relatively expensive metamethod call and it is recommended to avoid using it on hot code paths.

Back to TOC

Core constants

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, *log_by_lua*, ngx.timer.**

```
    ngx.OK (0)
    ngx.ERROR (-1)
    ngx.AGAIN (-2)
    ngx.DONE (-4)
    ngx.DECLINED (-5)
```

Note that only three of these constants are utilized by the Nginx API for Lua (i.e., ngx.exit accepts `NGX_OK` , `NGX_ERROR` , and `NGX_DECLINED` as input).

```
    ngx.null
```

The `ngx.null` constant is a `NULL` light userdata usually used to represent nil values in Lua tables etc and is similar to the lua-cjson library's `cjson.null` constant. This constant was first introduced in the `v0.5.0rc5` release.

The `ngx.DECLINED` constant was first introduced in the `v0.5.0rc19` release.

Back to TOC

HTTP method constants

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua*, ngx.timer.**

```
    ngx.HTTP_GET
```

```
ngx.HTTP_HEAD
ngx.HTTP_PUT
ngx.HTTP_POST
ngx.HTTP_DELETE
ngx.HTTP_OPTIONS   (added in the v0.5.0rc24 release)
ngx.HTTP_MKCOL     (added in the v0.8.2 release)
ngx.HTTP_COPY      (added in the v0.8.2 release)
ngx.HTTP_MOVE      (added in the v0.8.2 release)
ngx.HTTP_PROPFIND  (added in the v0.8.2 release)
ngx.HTTP_PROPPATCH (added in the v0.8.2 release)
ngx.HTTP_LOCK      (added in the v0.8.2 release)
ngx.HTTP_UNLOCK    (added in the v0.8.2 release)
ngx.HTTP_PATCH     (added in the v0.8.2 release)
ngx.HTTP_TRACE     (added in the v0.8.2 release)
```

These constants are usually used in ngx.location.capture and ngx.location.capture_multi method calls.

Back to TOC

HTTP status constants

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua*, ngx.timer.**

```
value = ngx.HTTP_OK (200)
value = ngx.HTTP_CREATED (201)
value = ngx.HTTP_SPECIAL_RESPONSE (300)
value = ngx.HTTP_MOVED_PERMANENTLY (301)
value = ngx.HTTP_MOVED_TEMPORARILY (302)
value = ngx.HTTP_SEE_OTHER (303)
value = ngx.HTTP_NOT_MODIFIED (304)
value = ngx.HTTP_BAD_REQUEST (400)
value = ngx.HTTP_UNAUTHORIZED (401)
value = ngx.HTTP_FORBIDDEN (403)
value = ngx.HTTP_NOT_FOUND (404)
value = ngx.HTTP_NOT_ALLOWED (405)
value = ngx.HTTP_GONE (410)
value = ngx.HTTP_INTERNAL_SERVER_ERROR (500)
value = ngx.HTTP_METHOD_NOT_IMPLEMENTED (501)
value = ngx.HTTP_SERVICE_UNAVAILABLE (503)
value = ngx.HTTP_GATEWAY_TIMEOUT (504) (first added in the v0.3.1rc38 release)
```

Back to TOC

Nginx log level constants

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua*, ngx.timer.**

```
ngx.STDERR
ngx.EMERG
ngx.ALERT
ngx.CRIT
ngx.ERR
ngx.WARN
ngx.NOTICE
ngx.INFO
ngx.DEBUG
```

These constants are usually used by the ngx.log method.

Back to TOC

print

syntax: *print(...)*

context: *init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua*, ngx.timer.**

Writes argument values into the nginx `error.log` file with the `ngx.NOTICE` log level.

It is equivalent to

```
ngx.log(ngx.NOTICE, ...)
```

Lua `nil` arguments are accepted and result in literal `"nil"` strings while Lua booleans result in literal `"true"` or `"false"` strings. And the `ngx.null` constant will yield the `"null"` string output.

There is a hard coded `2048` byte limitation on error message lengths in the Nginx core. This limit includes trailing newlines and leading time stamps. If the message size exceeds this limit, Nginx will truncate the message text accordingly. This limit can be manually modified by editing the `NGX_MAX_ERROR_STR` macro definition in the `src/core/ngx_log.h` file in the Nginx source tree.

Back to TOC

ngx.ctx

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua*, ngx.timer.**

This table can be used to store per-request Lua context data and has a life time identical to the current request (as with the Nginx variables).

Consider the following example,

```
location /test {
    rewrite_by_lua '
        ngx.say("foo = ", ngx.ctx.foo)
        ngx.ctx.foo = 76
    ';
    access_by_lua '
        ngx.ctx.foo = ngx.ctx.foo + 3
    ';
    content_by_lua '
        ngx.say(ngx.ctx.foo)
    ';
}
```

Then `GET /test` will yield the output

```
foo = nil
79
```

That is, the `ngx.ctx.foo` entry persists across the rewrite, access, and content phases of a request.

Every request, including subrequests, has its own copy of the table. For example:

```
location /sub {
    content_by_lua '
        ngx.say("sub pre: ", ngx.ctx.blah)
        ngx.ctx.blah = 32
        ngx.say("sub post: ", ngx.ctx.blah)
    ';
}

location /main {
    content_by_lua '
        ngx.ctx.blah = 73
        ngx.say("main pre: ", ngx.ctx.blah)
        local res = ngx.location.capture("/sub")
        ngx.print(res.body)
        ngx.say("main post: ", ngx.ctx.blah)
    ';
}
```

Then `GET /main` will give the output

```
main pre: 73
sub pre: nil
sub post: 32
main post: 73
```

Here, modification of the `ngx.ctx.blah` entry in the subrequest does not affect the one in the parent request. This is because they have two separate versions of `ngx.ctx.blah` .

Internal redirection will destroy the original request `ngx.ctx` data (if any) and the new request will have an empty `ngx.ctx` table. For instance,

```
location /new {
    content_by_lua '
        ngx.say(ngx.ctx.foo)
    ';
}

location /orig {
    content_by_lua '
        ngx.ctx.foo = "hello"
        ngx.exec("/new")
    ';
}
```

Then `GET /orig` will give

```
nil
```

rather than the original `"hello"` value.

Arbitrary data values, including Lua closures and nested tables, can be inserted into this "magic" table. It also allows the registration of custom meta methods.

Overriding `ngx.ctx` with a new Lua table is also supported, for example,

```
ngx.ctx = { foo = 32, bar = 54 }
```

When being used in the context of init_worker_by_lua*, this table just has the same lifetime of the current Lua handler.

The `ngx.ctx` lookup requires relatively expensive metamethod calls and it is much slower than explicitly passing per-request data along by your own function arguments. So do not abuse this API for saving your own function arguments because it usually has quite some performance impact. And because of the metamethod magic, never "local" the `ngx.ctx` table outside your function scope.

Back to TOC

ngx.location.capture

syntax: *res = ngx.location.capture(uri, options?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Issue a synchronous but still non-blocking *Nginx Subrequest* using `uri`.

Nginx's subrequests provide a powerful way to make non-blocking internal requests to other locations configured with disk file directory or *any* other nginx C modules like `ngx_proxy`, `ngx_fastcgi`, `ngx_memc`, `ngx_postgres`, `ngx_drizzle`, and even ngx_lua itself and etc etc etc.

Also note that subrequests just mimic the HTTP interface but there is *no* extra HTTP/TCP traffic *nor* IPC involved. Everything works internally, efficiently, on the C level.

Subrequests are completely different from HTTP 301/302 redirection (via ngx.redirect) and internal redirection (via ngx.exec).

You should always read the request body (by either calling ngx.req.read_body or configuring lua_need_request_body on) before initiating a subrequest.

Here is a basic example:

```
res = ngx.location.capture(uri)
```

Returns a Lua table with three slots (`res.status`, `res.header`, `res.body`, and `res.truncated`).

`res.status` holds the response status code for the subrequest response.

`res.header` holds all the response headers of the subrequest and it is a normal Lua table. For multi-value response headers, the value is a Lua (array) table that holds all the values in the order that they appear. For instance, if the subrequest response headers contain the following lines:

```
Set-Cookie: a=3
Set-Cookie: foo=bar
Set-Cookie: baz=blah
```

Then `res.header["Set-Cookie"]` will be evaluated to the table value `{"a=3", "foo=bar", "baz=blah"}`.

`res.body` holds the subrequest's response body data, which might be truncated. You always need to check the `res.truncated` boolean flag to see if `res.body` contains truncated data. The data truncation here can only be caused by those unrecoverable errors in your subrequests like the cases that the remote end aborts the connection prematurely in the middle of the response body data stream or a read timeout happens when your subrequest is receiving the response body data from the remote.

URI query strings can be concatenated to URI itself, for instance,

```
res = ngx.location.capture('/foo/bar?a=3&b=4')
```

Named locations like `@foo` are not allowed due to a limitation in the nginx core. Use normal locations combined with the `internal` directive to prepare internal-only locations.

An optional option table can be fed as the second argument, which supports the options:

- `method` specify the subrequest's request method, which only accepts constants like `ngx.HTTP_POST`.
- `body` specify the subrequest's request body (string value only).
- `args` specify the subrequest's URI query arguments (both string value and Lua tables are accepted)
- `ctx` specify a Lua table to be the ngx.ctx table for the subrequest. It can be the current request's ngx.ctx table, which effectively makes the parent and its subrequest to share exactly the same context table. This option was first introduced in the `v0.3.1rc25` release.
- `vars` take a Lua table which holds the values to set the specified Nginx variables in the subrequest as this option's value. This option was first introduced in the `v0.3.1rc31` release.
- `copy_all_vars` specify whether to copy over all the Nginx variable values of the current request to the subrequest in question. modifications of the nginx variables in the subrequest will not affect the current (parent) request. This option was first introduced in the `v0.3.1rc31` release.
- `share_all_vars` specify whether to share all the Nginx variables of the subrequest with the current (parent) request. modifications of the Nginx variables in the subrequest will affect the current (parent) request.
- `always_forward_body` when set to true, the current (parent) request's request body will always be forwarded to the subrequest being created if the `body` option is not specified. The request body read by either ngx.req.read_body() or lua_need_request_body on will be directly forwarded to the subrequest without copying the whole request body data when creating the subrequest (no matter the request body data is buffered in memory buffers or temporary files). By default, this option is `false` and when the `body` option is not specified, the request body of the current (parent) request is only forwarded when the subrequest takes the `PUT` or `POST` request method.

Issuing a POST subrequest, for example, can be done as follows

```
res = ngx.location.capture(
    '/foo/bar',
    { method = ngx.HTTP_POST, body = 'hello, world' }
)
```

See HTTP method constants methods other than POST. The `method` option is `ngx.HTTP_GET` by default.

The `args` option can specify extra URI arguments, for instance,

```
ngx.location.capture('/foo?a=1',
    { args = { b = 3, c = ':' } }
)
```

is equivalent to

```
ngx.location.capture('/foo?a=1&b=3&c=%3a')
```

that is, this method will escape argument keys and values according to URI rules and concatenate them together into a complete query string. The format for the Lua table passed as the `args` argument is identical to the format used in the ngx.encode_args method.

The `args` option can also take plain query strings:

```
ngx.location.capture('/foo?a=1',
    { args = 'b=3&c=%3a' } }
)
```

This is functionally identical to the previous examples.

The `share_all_vars` option controls whether to share nginx variables among the current request and its subrequests. If this option is set to `true`, then the current request and associated subrequests will share the same Nginx variable scope. Hence, changes to Nginx variables made by a subrequest will affect the current request.

Care should be taken in using this option as variable scope sharing can have unexpected side effects. The `args`, `vars`, or `copy_all_vars` options are generally preferable instead.

This option is set to `false` by default

```
location /other {
    set $dog "$dog world";
    echo "$uri dog: $dog";
}

location /lua {
    set $dog 'hello';
    content_by_lua '
        res = ngx.location.capture("/other",
            { share_all_vars = true });

        ngx.print(res.body)
        ngx.say(ngx.var.uri, ": ", ngx.var.dog)
    ';
}
```

Accessing location `/lua` gives

```
/other dog: hello world
/lua: hello world
```

The `copy_all_vars` option provides a copy of the parent request's Nginx variables to subrequests when such subrequests are issued. Changes made to these variables by such subrequests will not affect the parent request or any other subrequests sharing the parent request's variables.

```
location /other {
    set $dog "$dog world";
    echo "$uri dog: $dog";
}

location /lua {
    set $dog 'hello';
    content_by_lua '
```

```
            res = ngx.location.capture("/other",
                { copy_all_vars = true });

            ngx.print(res.body)
            ngx.say(ngx.var.uri, ": ", ngx.var.dog)
        ';
}
```

Request `GET /lua` will give the output

```
/other dog: hello world
/lua: hello
```

Note that if both `share_all_vars` and `copy_all_vars` are set to true, then `share_all_vars` takes precedence.

In addition to the two settings above, it is possible to specify values for variables in the subrequest using the `vars` option. These variables are set after the sharing or copying of variables has been evaluated, and provides a more efficient method of passing specific values to a subrequest over encoding them as URL arguments and unescaping them in the Nginx config file.

```
location /other {
    content_by_lua '
        ngx.say("dog = ", ngx.var.dog)
        ngx.say("cat = ", ngx.var.cat)
    ';
}

location /lua {
    set $dog '';
    set $cat '';
    content_by_lua '
        res = ngx.location.capture("/other",
            { vars = { dog = "hello", cat = 32 }});

        ngx.print(res.body)
    ';
}
```

Accessing `/lua` will yield the output

```
dog = hello
cat = 32
```

The `ctx` option can be used to specify a custom Lua table to serve as the ngx.ctx table for the subrequest.

```
location /sub {
    content_by_lua '
        ngx.ctx.foo = "bar";
    ';
}
location /lua {
    content_by_lua '
        local ctx = {}
        res = ngx.location.capture("/sub", { ctx = ctx })

        ngx.say(ctx.foo);
        ngx.say(ngx.ctx.foo);
    ';
}
```

Then request `GET /lua` gives

```
bar
nil
```

It is also possible to use this `ctx` option to share the same ngx.ctx table between the current (parent) request and the subrequest:

```
location /sub {
    content_by_lua '
        ngx.ctx.foo = "bar";
    ';
}
location /lua {
    content_by_lua '
        res = ngx.location.capture("/sub", { ctx = ngx.ctx })
        ngx.say(ngx.ctx.foo);
    ';
}
```

Request `GET /lua` yields the output

```
bar
```

Note that subrequests issued by ngx.location.capture inherit all the request headers of the current request by default and that this may have unexpected side effects on the subrequest responses. For example, when using the standard `ngx_proxy` module to serve subrequests, an "Accept-Encoding: gzip" header in the main request may result in gzipped responses that cannot be handled properly in Lua code. Original request headers should be ignored by setting proxy_pass_request_headers to `off` in subrequest locations.

When the `body` option is not specified and the `always_forward_body` option is false (the default value), the `POST` and `PUT` subrequests will inherit the request bodies of the parent request (if any).

There is a hard-coded upper limit on the number of concurrent subrequests possible for every main request. In older versions of Nginx, the limit was `50` concurrent subrequests and in more recent versions, Nginx `1.1.x` onwards, this was increased to `200` concurrent subrequests. When this limit is exceeded, the following error message is added to the `error.log` file:

```
[error] 13983#0: *1 subrequests cycle while processing "/uri"
```

The limit can be manually modified if required by editing the definition of the `NGX_HTTP_MAX_SUBREQUESTS` macro in the `nginx/src/http/ngx_http_request.h` file in the Nginx source tree.

Please also refer to restrictions on capturing locations configured by subrequest directives of other modules.

Back to TOC

ngx.location.capture_multi

syntax: *res1, res2, ... = ngx.location.capture_multi({ {uri, options?}, {uri, options?}, ... })*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Just like ngx.location.capture, but supports multiple subrequests running in parallel.

This function issues several parallel subrequests specified by the input table and returns their results in the same order. For example,

```
res1, res2, res3 = ngx.location.capture_multi{
    { "/foo", { args = "a=3&b=4" } },
    { "/bar" },
    { "/baz", { method = ngx.HTTP_POST, body = "hello" } },
}

if res1.status == ngx.HTTP_OK then
    ...
end

if res2.body == "BLAH" then
    ...
end
```

This function will not return until all the subrequests terminate. The total latency is the longest latency of the individual subrequests rather than the sum.

Lua tables can be used for both requests and responses when the number of subrequests to be issued is not known in advance:

```
-- construct the requests table
local reqs = {}
table.insert(reqs, { "/mysql" })
table.insert(reqs, { "/postgres" })
table.insert(reqs, { "/redis" })
table.insert(reqs, { "/memcached" })

-- issue all the requests at once and wait until they all return
local resps = { ngx.location.capture_multi(reqs) }

-- loop over the responses table
for i, resp in ipairs(resps) do
    -- process the response table "resp"
end
```

The ngx.location.capture function is just a special form of this function. Logically speaking, the ngx.location.capture can be implemented like this

```
ngx.location.capture =
    function (uri, args)
        return ngx.location.capture_multi({ {uri, args} })
    end
```

Please also refer to restrictions on capturing locations configured by subrequest directives of other modules.

Back to TOC

ngx.status

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua**

Read and write the current request's response status. This should be called before sending out the response headers.

```
ngx.status = ngx.HTTP_CREATED
status = ngx.status
```

Setting `ngx.status` after the response header is sent out has no effect but leaving an error message in your nginx's error log file:

```
attempt to set ngx.status after sending out response headers
```

Back to TOC

ngx.header.HEADER

syntax: *ngx.header.HEADER = VALUE*

syntax: *value = ngx.header.HEADER*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua**

Set, add to, or clear the current request's `HEADER` response header that is to be sent.

Underscores (_) in the header names will be replaced by hyphens (-) by default. This transformation can be turned off via the lua_transform_underscores_in_response_headers directive.

The header names are matched case-insensitively.

```
-- equivalent to ngx.header["Content-Type"] = 'text/plain'
ngx.header.content_type = 'text/plain';

ngx.header["X-My-Header"] = 'blah blah';
```

Multi-value headers can be set this way:

```
ngx.header['Set-Cookie'] = {'a=32; path=/', 'b=4; path=/'}
```

will yield

```
Set-Cookie: a=32; path=/
Set-Cookie: b=4; path=/
```

in the response headers.

Only Lua tables are accepted (Only the last element in the table will take effect for standard headers such as `Content-Type` that only accept a single value).

```
ngx.header.content_type = {'a', 'b'}
```

is equivalent to

```
ngx.header.content_type = 'b'
```

Setting a slot to `nil` effectively removes it from the response headers:

```
ngx.header["X-My-Header"] = nil;
```

The same applies to assigning an empty table:

```
ngx.header["X-My-Header"] = {};
```

Setting `ngx.header.HEADER` after sending out response headers (either explicitly with ngx.send_headers or implicitly with ngx.print and similar) will throw out a Lua exception.

Reading `ngx.header.HEADER` will return the value of the response header named `HEADER`.

Underscores (`_`) in the header names will also be replaced by dashes (`-`) and the header names will be matched case-insensitively. If the response header is not present at all, `nil` will be returned.

This is particularly useful in the context of header_filter_by_lua and header_filter_by_lua_file, for example,

```
location /test {
    set $footer '';

    proxy_pass http://some-backend;

    header_filter_by_lua '
        if ngx.header["X-My-Header"] == "blah" then
            ngx.var.footer = "some value"
        end
    ';

    echo_after_body $footer;
}
```

For multi-value headers, all of the values of header will be collected in order and returned as a Lua table. For example, response headers

```
Foo: bar
Foo: baz
```

will result in

```
{"bar", "baz"}
```

to be returned when reading `ngx.header.Foo`.

Note that `ngx.header` is not a normal Lua table and as such, it is not possible to iterate through it using the Lua `ipairs` function.

For reading *request* headers, use the ngx.req.get_headers function instead.

Back to TOC

ngx.resp.get_headers

syntax: *headers = ngx.resp.get_headers(max_headers?, raw?)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua**

Returns a Lua table holding all the current response headers for the current request.

```
local h = ngx.resp.get_headers()
for k, v in pairs(h) do
    ...
end
```

This function has the same signature as ngx.req.get_headers except getting response headers instead of request headers.

This API was first introduced in the `v0.9.5` release.

Back to TOC

ngx.req.start_time

syntax: *secs = ngx.req.start_time()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua**

Returns a floating-point number representing the timestamp (including milliseconds as the decimal part) when the current request was created.

The following example emulates the `$request_time` variable value (provided by ngx_http_log_module) in pure Lua:

```
local request_time = ngx.now() - ngx.req.start_time()
```

This function was first introduced in the `v0.7.7` release.

See also ngx.now and ngx.update_time.

Back to TOC

ngx.req.http_version

syntax: *num = ngx.req.http_version()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua**

Returns the HTTP version number for the current request as a Lua number.

Current possible values are 1.0, 1.1, and 0.9. Returns `nil` for unrecognized values.

This method was first introduced in the `v0.7.17` release.

Back to TOC

ngx.req.raw_header

syntax: *str = ngx.req.raw_header(no_request_line?)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua**

Returns the original raw HTTP protocol header received by the Nginx server.

By default, the request line and trailing `CR LF` terminator will also be included. For example,

```
ngx.print(ngx.req.raw_header())
```

gives something like this:

```
GET /t HTTP/1.1
Host: localhost
Connection: close
Foo: bar
```

You can specify the optional `no_request_line` argument as a `true` value to exclude the request line from the result. For example,

```
ngx.print(ngx.req.raw_header(true))
```

outputs something like this:

```
Host: localhost
Connection: close
Foo: bar
```

This method was first introduced in the `v0.7.17` release.

Back to TOC

ngx.req.get_method

syntax: *method_name = ngx.req.get_method()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua**

Retrieves the current request's request method name. Strings like `"GET"` and `"POST"` are returned instead of numerical method constants.

If the current request is an Nginx subrequest, then the subrequest's method name will be returned.

This method was first introduced in the `v0.5.6` release.

See also ngx.req.set_method.

Back to TOC

ngx.req.set_method

syntax: *ngx.req.set_method(method_id)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua**

Overrides the current request's request method with the `request_id` argument. Currently only numerical method constants are supported, like `ngx.HTTP_POST` and `ngx.HTTP_GET`.

If the current request is an Nginx subrequest, then the subrequest's method will be overridden.

This method was first introduced in the `v0.5.6` release.

See also ngx.req.get_method.

Back to TOC

ngx.req.set_uri

syntax: *ngx.req.set_uri(uri, jump?)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua**

Rewrite the current request's (parsed) URI by the `uri` argument. The `uri` argument must be a Lua string and cannot be of zero length, or a Lua exception will be thrown.

The optional boolean `jump` argument can trigger location rematch (or location jump) as ngx_http_rewrite_module's rewrite directive, that is, when `jump` is `true` (default to `false`), this function will never return and it will tell Nginx to try re-searching locations with the new URI value at the later `post-rewrite` phase and jumping to the new location.

Location jump will not be triggered otherwise, and only the current request's URI will be modified, which is also the default behavior. This function will return but with no returned values when the `jump` argument is `false` or absent altogether.

For example, the following nginx config snippet

```
rewrite ^ /foo last;
```

can be coded in Lua like this:

```
ngx.req.set_uri("/foo", true)
```

Similarly, Nginx config

```
rewrite ^ /foo break;
```

can be coded in Lua as

```
ngx.req.set_uri("/foo", false)
```

or equivalently,

```
ngx.req.set_uri("/foo")
```

The `jump` can only be set to `true` in rewrite_by_lua and rewrite_by_lua_file. Use of jump in other contexts is prohibited and will throw out a Lua exception.

A more sophisticated example involving regex substitutions is as follows

```
location /test {
    rewrite_by_lua '
        local uri = ngx.re.sub(ngx.var.uri, "^/test/(.*)", "$1", "o")
        ngx.req.set_uri(uri)
    ';
    proxy_pass http://my_backend;
}
```

which is functionally equivalent to

```
location /test {
    rewrite ^/test/(.*) /$1 break;
    proxy_pass http://my_backend;
}
```

Note that it is not possible to use this interface to rewrite URI arguments and that ngx.req.set_uri_args should be used for this instead. For instance, Nginx config

```
rewrite ^ /foo?a=3? last;
```

can be coded as

```
ngx.req.set_uri_args("a=3")
ngx.req.set_uri("/foo", true)
```

or

```
ngx.req.set_uri_args({a = 3})
ngx.req.set_uri("/foo", true)
```

This interface was first introduced in the `v0.3.1rc14` release.

Back to TOC

ngx.req.set_uri_args

syntax: *ngx.req.set_uri_args(args)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua**

Rewrite the current request's URI query arguments by the `args` argument. The `args` argument can be either a Lua string, as in

```
ngx.req.set_uri_args("a=3&b=hello%20world")
```

or a Lua table holding the query arguments' key-value pairs, as in

```
ngx.req.set_uri_args({ a = 3, b = "hello world" })
```

where in the latter case, this method will escape argument keys and values according to the URI escaping rule.

Multi-value arguments are also supported:

```
ngx.req.set_uri_args({ a = 3, b = {5, 6} })
```

which will result in a query string like `a=3&b=5&b=6` .

This interface was first introduced in the `v0.3.1rc13` release.

See also ngx.req.set_uri.

Back to TOC

ngx.req.get_uri_args

syntax: *args = ngx.req.get_uri_args(max_args?)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua***

Returns a Lua table holding all the current request URL query arguments.

```
location = /test {
    content_by_lua '
        local args = ngx.req.get_uri_args()
        for key, val in pairs(args) do
            if type(val) == "table" then
                ngx.say(key, ": ", table.concat(val, ", "))
            else
                ngx.say(key, ": ", val)
            end
        end
    ';
}
```

Then `GET /test?foo=bar&bar=baz&bar=blah` will yield the response body

```
foo: bar
bar: baz, blah
```

Multiple occurrences of an argument key will result in a table value holding all the values for that key in order.

Keys and values are unescaped according to URI escaping rules. In the settings above, `GET /test?a%20b=1%61+2` will yield:

```
a b: 1a 2
```

Arguments without the `=<value>` parts are treated as boolean arguments. `GET /test?foo&bar` will yield:

```
foo: true
bar: true
```

That is, they will take Lua boolean values `true`. However, they are different from arguments taking empty string values. `GET /test?foo=&bar=` will give something like

```
foo:
bar:
```

Empty key arguments are discarded. `GET /test?=hello&=world` will yield an empty output for instance.

Updating query arguments via the nginx variable `$args` (or `ngx.var.args` in Lua) at runtime is also supported:

```
ngx.var.args = "a=3&b=42"
local args = ngx.req.get_uri_args()
```

Here the `args` table will always look like

```
{a = 3, b = 42}
```

regardless of the actual request query string.

Note that a maximum of 100 request arguments are parsed by default (including those with the same name) and that additional request arguments are silently discarded to guard against potential denial of service attacks.

However, the optional `max_args` function argument can be used to override this limit:

```
local args = ngx.req.get_uri_args(10)
```

This argument can be set to zero to remove the limit and to process all request arguments received:

```
local args = ngx.req.get_uri_args(0)
```

Removing the `max_args` cap is strongly discouraged.

Back to TOC

ngx.req.get_post_args

syntax: *args, err = ngx.req.get_post_args(max_args?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua**

Returns a Lua table holding all the current request POST query arguments (of the MIME type `application/x-www-form-urlencoded`). Call ngx.req.read_body to read the request body first or turn on the lua_need_request_body directive to avoid errors.

```
location = /test {
    content_by_lua '
        ngx.req.read_body()
        local args, err = ngx.req.get_post_args()
        if not args then
            ngx.say("failed to get post args: ", err)
```

```
                return
            end
            for key, val in pairs(args) do
                if type(val) == "table" then
                    ngx.say(key, ": ", table.concat(val, ", "))
                else
                    ngx.say(key, ": ", val)
                end
            end
        ';
    }
```

Then

```
# Post request with the body 'foo=bar&bar=baz&bar=blah'
$ curl --data 'foo=bar&bar=baz&bar=blah' localhost/test
```

will yield the response body like

```
foo: bar
bar: baz, blah
```

Multiple occurrences of an argument key will result in a table value holding all of the values for that key in order.

Keys and values will be unescaped according to URI escaping rules.

With the settings above,

```
# POST request with body 'a%20b=1%61+2'
$ curl -d 'a%20b=1%61+2' localhost/test
```

will yield:

```
a b: 1a 2
```

Arguments without the `=<value>` parts are treated as boolean arguments. `GET /test?foo&bar` will yield:

```
foo: true
bar: true
```

That is, they will take Lua boolean values `true`. However, they are different from arguments taking empty string values. `POST /test` with request body `foo=&bar=` will return something like

```
foo:
bar:
```

Empty key arguments are discarded. `POST /test` with body `=hello&=world` will yield empty outputs for instance.

Note that a maximum of 100 request arguments are parsed by default (including those with the same name) and that additional request arguments are silently discarded to guard against potential denial of service attacks.

However, the optional `max_args` function argument can be used to override this limit:

```
local args = ngx.req.get_post_args(10)
```

This argument can be set to zero to remove the limit and to process all request arguments received:

```
local args = ngx.req.get_post_args(0)
```

Removing the `max_args` cap is strongly discouraged.

Back to TOC

ngx.req.get_headers

syntax: *headers = ngx.req.get_headers(max_headers?, raw?)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua, log_by_lua**

Returns a Lua table holding all the current request headers.

```
local h = ngx.req.get_headers()
for k, v in pairs(h) do
    ...
end
```

To read an individual header:

```
ngx.say("Host: ", ngx.req.get_headers()["Host"])
```

Note that the ngx.var.HEADER API call, which uses core $http_HEADER variables, may be more preferable for reading individual request headers.

For multiple instances of request headers such as:

```
Foo: foo
Foo: bar
Foo: baz
```

the value of `ngx.req.get_headers()["Foo"]` will be a Lua (array) table such as:

```
{"foo", "bar", "baz"}
```

Note that a maximum of 100 request headers are parsed by default (including those with the same name) and that additional request headers are silently discarded to guard against potential denial of service attacks.

However, the optional `max_headers` function argument can be used to override this limit:

```
local args = ngx.req.get_headers(10)
```

This argument can be set to zero to remove the limit and to process all request headers received:

```
local args = ngx.req.get_headers(0)
```

Removing the `max_headers` cap is strongly discouraged.

Since the `0.6.9` release, all the header names in the Lua table returned are converted to the pure lower-case form by default, unless the `raw` argument is set to `true` (default to `false`).

Also, by default, an `__index` metamethod is added to the resulting Lua table and will normalize the keys to a pure lowercase form with all underscores converted to dashes in case of a lookup miss. For example, if a request header `My-Foo-Header` is present, then the following invocations will all pick up the value of this header correctly:

```
ngx.say(headers.my_foo_header)
ngx.say(headers["My-Foo-Header"])
ngx.say(headers["my-foo-header"])
```

The `__index` metamethod will not be added when the `raw` argument is set to `true` .

Back to TOC

ngx.req.set_header

syntax: *ngx.req.set_header(header_name, header_value)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*

Set the current request's request header named `header_name` to value `header_value` , overriding any existing ones.

By default, all the subrequests subsequently initiated by ngx.location.capture and ngx.location.capture_multi will inherit the new header.

Here is an example of setting the `Content-Length` header:

```
ngx.req.set_header("Content-Type", "text/css")
```

The `header_value` can take an array list of values, for example,

```
ngx.req.set_header("Foo", {"a", "abc"})
```

will produce two new request headers:

```
Foo: a
Foo: abc
```

and old `Foo` headers will be overridden if there is any.

When the `header_value` argument is `nil` , the request header will be removed. So

```
ngx.req.set_header("X-Foo", nil)
```

is equivalent to

```
ngx.req.clear_header("X-Foo")
```

Back to TOC

ngx.req.clear_header

syntax: *ngx.req.clear_header(header_name)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua**

Clears the current request's request header named `header_name`. None of the current request's existing subrequests will be affected but subsequently initiated subrequests will inherit the change by default.

Back to TOC

ngx.req.read_body

syntax: *ngx.req.read_body()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Reads the client request body synchronously without blocking the Nginx event loop.

```
ngx.req.read_body()
local args = ngx.req.get_post_args()
```

If the request body is already read previously by turning on lua_need_request_body or by using other modules, then this function does not run and returns immediately.

If the request body has already been explicitly discarded, either by the ngx.req.discard_body function or other modules, this function does not run and returns immediately.

In case of errors, such as connection errors while reading the data, this method will throw out a Lua exception *or* terminate the current request with a 500 status code immediately.

The request body data read using this function can be retrieved later via ngx.req.get_body_data or, alternatively, the temporary file name for the body data cached to disk using ngx.req.get_body_file. This depends on

1. whether the current request body is already larger than the client_body_buffer_size,
2. and whether client_body_in_file_only has been switched on.

In cases where current request may have a request body and the request body data is not required, The ngx.req.discard_body function must be used to explicitly discard the request body to avoid breaking things under HTTP 1.1 keepalive or HTTP 1.1 pipelining.

This function was first introduced in the `v0.3.1rc17` release.

Back to TOC

ngx.req.discard_body

syntax: *ngx.req.discard_body()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Explicitly discard the request body, i.e., read the data on the connection and throw it away immediately. Please note that ignoring request body is not the right way to discard it, and that this function must be called to avoid breaking things under HTTP 1.1 keepalive or HTTP 1.1 pipelining.

This function is an asynchronous call and returns immediately.

If the request body has already been read, this function does nothing and returns immediately.

This function was first introduced in the `v0.3.1rc17` release.

See also ngx.req.read_body.

Back to TOC

ngx.req.get_body_data

syntax: *data = ngx.req.get_body_data()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Retrieves in-memory request body data. It returns a Lua string rather than a Lua table holding all the parsed query arguments. Use the ngx.req.get_post_args function instead if a Lua table is required.

This function returns `nil` if

1. the request body has not been read,
2. the request body has been read into disk temporary files,
3. or the request body has zero size.

If the request body has not been read yet, call ngx.req.read_body first (or turned on lua_need_request_body to force this module to read the request body. This is not recommended however).

If the request body has been read into disk files, try calling the ngx.req.get_body_file function instead.

To force in-memory request bodies, try setting client_body_buffer_size to the same size value in client_max_body_size.

Note that calling this function instead of using `ngx.var.request_body` or `ngx.var.echo_request_body` is more efficient because it can save one dynamic memory allocation and one data copy.

This function was first introduced in the `v0.3.1rc17` release.

See also ngx.req.get_body_file.

Back to TOC

ngx.req.get_body_file

syntax: *file_name = ngx.req.get_body_file()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Retrieves the file name for the in-file request body data. Returns `nil` if the request body has not been read or has been read into memory.

The returned file is read only and is usually cleaned up by Nginx's memory pool. It should not be manually modified, renamed, or removed in Lua code.

If the request body has not been read yet, call ngx.req.read_body first (or turned on lua_need_request_body to force this module to read the request body. This is not recommended however).

If the request body has been read into memory, try calling the ngx.req.get_body_data function instead.

To force in-file request bodies, try turning on client_body_in_file_only.

This function was first introduced in the `v0.3.1rc17` release.

See also ngx.req.get_body_data.

Back to TOC

ngx.req.set_body_data

syntax: *ngx.req.set_body_data(data)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Set the current request's request body using the in-memory data specified by the `data` argument.

If the current request's request body has not been read, then it will be properly discarded. When the current request's request body has been read into memory or buffered into a disk file, then the old request body's memory will be freed or the disk file will be cleaned up immediately, respectively.

This function was first introduced in the `v0.3.1rc18` release.

See also ngx.req.set_body_file.

Back to TOC

ngx.req.set_body_file

syntax: *ngx.req.set_body_file(file_name, auto_clean?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Set the current request's request body using the in-file data specified by the `file_name` argument.

If the optional `auto_clean` argument is given a `true` value, then this file will be removed at request completion or the next time this function or ngx.req.set_body_data are called in the same request. The `auto_clean` is default to `false` .

Please ensure that the file specified by the `file_name` argument exists and is readable by an Nginx worker process by setting its permission properly to avoid Lua exception errors.

If the current request's request body has not been read, then it will be properly discarded. When the current request's request body has been read into memory or buffered into a disk file, then the old request body's memory will be freed or the disk file will be cleaned up immediately, respectively.

This function was first introduced in the `v0.3.1rc18` release.

See also ngx.req.set_body_data.

Back to TOC

ngx.req.init_body

syntax: *ngx.req.init_body(buffer_size?)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua**

Creates a new blank request body for the current request and inializes the buffer for later request body data writing via the ngx.req.append_body and ngx.req.finish_body APIs.

If the `buffer_size` argument is specified, then its value will be used for the size of the memory buffer for body writing with ngx.req.append_body. If the argument is omitted, then the value specified by the standard client_body_buffer_size directive will be used instead.

When the data can no longer be hold in the memory buffer for the request body, then the data will be flushed onto a temporary file just like the standard request body reader in the Nginx core.

It is important to always call the ngx.req.finish_body after all the data has been appended onto the current request body. Also, when this function is used together with ngx.req.socket, it is required to call ngx.req.socket *before* this function, or you will get the "request body already exists" error message.

The usage of this function is often like this:

```
ngx.req.init_body(128 * 1024)  -- buffer is 128KB
for chunk in next_data_chunk() do
    ngx.req.append_body(chunk) -- each chunk can be 4KB
end
ngx.req.finish_body()
```

This function can be used with ngx.req.append_body, ngx.req.finish_body, and ngx.req.socket to implement efficient input filters in pure Lua (in the context of rewrite_by_lua* or access_by_lua*), which can be used with other Nginx content handler or upstream modules like ngx_http_proxy_module and ngx_http_fastcgi_module.

This function was first introduced in the `v0.5.11` release.

Back to TOC

ngx.req.append_body

syntax: *ngx.req.append_body(data_chunk)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua**

Append new data chunk specified by the `data_chunk` argument onto the existing request body created by the ngx.req.init_body call.

When the data can no longer be hold in the memory buffer for the request body, then the data will be flushed onto a temporary file just like the standard request body reader in the Nginx core.

It is important to always call the ngx.req.finish_body after all the data has been appended onto the current request body.

This function can be used with ngx.req.init_body, ngx.req.finish_body, and ngx.req.socket to implement efficient input filters in pure Lua (in the context of rewrite_by_lua* or access_by_lua*), which can be used with other Nginx content handler or upstream modules like ngx_http_proxy_module and ngx_http_fastcgi_module.

This function was first introduced in the `v0.5.11` release.

See also ngx.req.init_body.

Back to TOC

ngx.req.finish_body

syntax: *ngx.req.finish_body()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua**

Completes the construction process of the new request body created by the ngx.req.init_body and ngx.req.append_body calls.

This function can be used with ngx.req.init_body, ngx.req.append_body, and ngx.req.socket to implement efficient input filters in pure Lua (in the context of rewrite_by_lua* or access_by_lua*), which can be used with other Nginx content handler or upstream modules like ngx_http_proxy_module and ngx_http_fastcgi_module.

This function was first introduced in the `v0.5.11` release.

See also ngx.req.init_body.

Back to TOC

ngx.req.socket

syntax: *tcpsock, err = ngx.req.socket()*

syntax: *tcpsock, err = ngx.req.socket(raw)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Returns a read-only cosocket object that wraps the downstream connection. Only receive and receiveuntil methods are supported on this object.

In case of error, `nil` will be returned as well as a string describing the error.

The socket object returned by this method is usually used to read the current request's body in a streaming fashion. Do not turn on the lua_need_request_body directive, and do not mix this call with ngx.req.read_body and ngx.req.discard_body.

If any request body data has been pre-read into the Nginx core request header buffer, the resulting cosocket object will take care of this to avoid potential data loss resulting from such pre-reading. Chunked request bodies are not yet supported in this API.

Since the `v0.9.0` release, this function accepts an optional boolean `raw` argument. When this argument is `true`, this function returns a full-duplex cosocket object wrapping around the raw downstream connection socket, upon which you can call the receive, receiveuntil, and send methods.

When the `raw` argument is `true`, it is required that no pending data from any previous ngx.say, ngx.print, or ngx.send_headers calls exists. So if you have these downstream output calls previously, you should call ngx.flush(true) before calling `ngx.req.socket(true)` to ensure that there is no pending output data. If the request body has not been read yet, then this "raw socket" can also be used to read the request body.

You can use the "raw request socket" returned by `ngx.req.socket(true)` to implement fancy protocols like WebSocket, or just emit your own raw HTTP response header or body data. You can refer to the lua-resty-websocket library for a real world example.

This function was first introduced in the `v0.5.0rc1` release.

Back to TOC

ngx.exec

syntax: *ngx.exec(uri, args?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Does an internal redirect to `uri` with `args`.

```
ngx.exec('/some-location');
ngx.exec('/some-location', 'a=3&b=5&c=6');
ngx.exec('/some-location?a=3&b=5', 'c=6');
```

Named locations are also supported, but query strings are ignored. For example,

```
location /foo {
    content_by_lua '
        ngx.exec("@bar");
    ';
}

location @bar {
    ...
}
```

The optional second `args` can be used to specify extra URI query arguments, for example:

```
ngx.exec("/foo", "a=3&b=hello%20world")
```

Alternatively, a Lua table can be passed for the `args` argument for ngx_lua to carry out URI escaping and string concatenation.

```
ngx.exec("/foo", { a = 3, b = "hello world" })
```

The result is exactly the same as the previous example. The format for the Lua table passed as the `args` argument is identical to the format used in the ngx.encode_args method.

Note that this is very different from ngx.redirect in that it is just an internal redirect and no new HTTP traffic is involved.

This method never returns.

This method *must* be called before ngx.send_headers or explicit response body outputs by either ngx.print or ngx.say.

It is strongly recommended to combine the `return` statement with this call, i.e., `return ngx.exec(...)`.

This method is similar to the echo_exec directive of the echo-nginx-module.

Back to TOC

ngx.redirect

syntax: *ngx.redirect(uri, status?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Issue an `HTTP 301` or `302` redirection to `uri`.

The optional `status` parameter specifies whether `301` or `302` to be used. It is `302` (`ngx.HTTP_MOVED_TEMPORARILY`) by default.

Here is an example assuming the current server name is `localhost` and that it is listening on Port 1984:

```
return ngx.redirect("/foo")
```

which is equivalent to

```
return ngx.redirect("http://localhost:1984/foo", ngx.HTTP_MOVED_TEMPORARILY)
```

Redirecting arbitrary external URLs is also supported, for example:

```
return ngx.redirect("http://www.google.com")
```

We can also use the numerical code directly as the second `status` argument:

```
return ngx.redirect("/foo", 301)
```

This method *must* be called before ngx.send_headers or explicit response body outputs by either ngx.print or ngx.say.

This method is very much like the rewrite directive with the `redirect` modifier in the standard [[HttpRewriteModule]], for example, this `nginx.conf` snippet

```
rewrite ^ /foo? redirect;  # nginx config
```

is equivalent to the following Lua code

```
return ngx.redirect('/foo');  -- Lua code
```

while

```
rewrite ^ /foo? permanent;  # nginx config
```

is equivalent to

```
return ngx.redirect('/foo', ngx.HTTP_MOVED_PERMANENTLY)  -- Lua code
```

URI arguments can be specified as well, for example:

```
return ngx.redirect('/foo?a=3&b=4')
```

This method call terminates the current request's processing and never returns. It is recommended to combine the `return` statement with this call, i.e., `return ngx.redirect(...)`, so as to be more explicit.

Back to TOC

ngx.send_headers

syntax: *ok, err = ngx.send_headers()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Explicitly send out the response headers.

Since `v0.8.3` this function returns `1` on success, or returns `nil` and a string describing the error otherwise.

Note that there is normally no need to manually send out response headers as ngx_lua will automatically send headers out before content is output with ngx.say or ngx.print or when content_by_lua exits normally.

Back to TOC

ngx.headers_sent

syntax: *value = ngx.headers_sent*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua**

Returns `true` if the response headers have been sent (by ngx_lua), and `false` otherwise.

This API was first introduced in ngx_lua v0.3.1rc6.

Back to TOC

ngx.print

syntax: *ok, err = ngx.print(...)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Emits arguments concatenated to the HTTP client (as response body). If response headers have not been sent, this function will send headers out first and then output body data.

Since `v0.8.3` this function returns `1` on success, or returns `nil` and a string describing the error otherwise.

Lua `nil` values will output `"nil"` strings and Lua boolean values will output `"true"` and `"false"` literal strings respectively.

Nested arrays of strings are permitted and the elements in the arrays will be sent one by one:

```
local table = {
    "hello, ",
    {"world: ", true, " or ", false,
        {": ", nil}}
}
ngx.print(table)
```

will yield the output

```
hello, world: true or false: nil
```

Non-array table arguments will cause a Lua exception to be thrown.

The `ngx.null` constant will yield the `"null"` string output.

This is an asynchronous call and will return immediately without waiting for all the data to be written into the system send buffer. To run in synchronous mode, call `ngx.flush(true)` after calling `ngx.print`. This can be particularly useful for streaming output. See ngx.flush for more details.

Please note that both `ngx.print` and ngx.say will always invoke the whole Nginx output body filter chain, which is an expensive operation. So be careful when calling either of these two in a tight loop; buffer the data yourself in Lua and save the calls.

Back to TOC

ngx.say

syntax: *ok, err = ngx.say(...)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Just as ngx.print but also emit a trailing newline.

Back to TOC

ngx.log

syntax: *ngx.log(log_level, ...)*

context: *init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Log arguments concatenated to error.log with the given logging level.

Lua `nil` arguments are accepted and result in literal `"nil"` string while Lua booleans result in literal `"true"` or `"false"` string outputs. And the `ngx.null` constant will yield the `"null"` string output.

The `log_level` argument can take constants like `ngx.ERR` and `ngx.WARN`. Check out Nginx log level constants for details.

There is a hard coded `2048` byte limitation on error message lengths in the Nginx core. This limit includes trailing newlines and leading time stamps. If the message size exceeds this limit, Nginx will truncate the message text accordingly. This limit can be manually modified by editing the `NGX_MAX_ERROR_STR` macro definition in the `src/core/ngx_log.h` file in the Nginx source tree.

Back to TOC

ngx.flush

syntax: *ok, err = ngx.flush(wait?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Flushes response output to the client.

`ngx.flush` accepts an optional boolean `wait` argument (Default: `false`) first introduced in the `v0.3.1rc34` release. When called with the default argument, it issues an asynchronous call (Returns immediately without waiting for output data to be written into the system send buffer). Calling the function with the `wait` argument set to `true` switches to synchronous mode.

In synchronous mode, the function will not return until all output data has been written into the system send buffer or until the send_timeout setting has expired. Note that using the Lua coroutine mechanism means that this function does not block the Nginx event loop even in the synchronous mode.

When `ngx.flush(true)` is called immediately after ngx.print or ngx.say, it causes the latter functions to run in synchronous mode. This can be particularly useful for streaming output.

Note that `ngx.flush` is not functional when in the HTTP 1.0 output buffering mode. See HTTP 1.0 support.

Since `v0.8.3` this function returns `1` on success, or returns `nil` and a string describing the error otherwise.

Back to TOC

ngx.exit

syntax: *ngx.exit(status)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua**

When `status >= 200` (i.e., `ngx.HTTP_OK` and above), it will interrupt the execution of the current request and return status code to nginx.

When `status == 0` (i.e., `ngx.OK`), it will only quit the current phase handler (or the content handler if the content_by_lua directive is used) and continue to run later phases (if any) for the current request.

The `status` argument can be `ngx.OK` , `ngx.ERROR` , `ngx.HTTP_NOT_FOUND` , `ngx.HTTP_MOVED_TEMPORARILY` , or other HTTP status constants.

To return an error page with custom contents, use code snippets like this:

```
ngx.status = ngx.HTTP_GONE
ngx.say("This is our own content")
-- to cause quit the whole request rather than the current phase handler
ngx.exit(ngx.HTTP_OK)
```

The effect in action:

```
$ curl -i http://localhost/test
```

```
HTTP/1.1 410 Gone
Server: nginx/1.0.6
Date: Thu, 15 Sep 2011 00:51:48 GMT
Content-Type: text/plain
Transfer-Encoding: chunked
Connection: keep-alive

This is our own content
```

Number literals can be used directly as the argument, for instance,

```
ngx.exit(501)
```

Note that while this method accepts all HTTP status constants as input, it only accepts `NGX_OK` and `NGX_ERROR` of the core constants.

It is recommended, though not necessary (for contexts other than header_filter_by_lua), to combine the `return` statement with this call, i.e., `return ngx.exit(...)`, to give a visual hint to others reading the code.

When being used in the context of header_filter_by_lua, `ngx.exit()` is an asynchronous operation and will return immediately. This behavior might change in the future. So always use `return` at the same time, as suggested above.

Back to TOC

ngx.eof

syntax: *ok, err = ngx.eof()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Explicitly specify the end of the response output stream. In the case of HTTP 1.1 chunked encoded output, it will just trigger the Nginx core to send out the "last chunk".

When you disable the HTTP 1.1 keep-alive feature for your downstream connections, you can rely on descent HTTP clients to close the connection actively for you when you call this method. This trick can be used do back-ground jobs without letting the HTTP clients to wait on the connection, as in the following example:

```
location = /async {
    keepalive_timeout 0;
    content_by_lua '
        ngx.say("got the task!")
        ngx.eof()  -- a descent HTTP client will close the connection at this point
        -- access MySQL, PostgreSQL, Redis, Memcached, and etc here...
    ';
}
```

But if you create subrequests to access other locations configured by Nginx upstream modules, then you should configure those upstream modules to ignore client connection abortions if they are not by default. For example, by default the standard ngx_http_proxy_module will terminate both the subrequest and the main request as soon as the client closes the connection, so it is important to turn on the proxy_ignore_client_abort directive in your location block configured by ngx_http_proxy_module:

```
proxy_ignore_client_abort on;
```

A better way to do background jobs is to use the ngx.timer.at API.

Since `v0.8.3` this function returns `1` on success, or returns `nil` and a string describing the error otherwise.

Back to TOC

ngx.sleep

syntax: *ngx.sleep(seconds)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Sleeps for the specified seconds without blocking. One can specify time resolution up to 0.001 seconds (i.e., one milliseconds).

Behind the scene, this method makes use of the Nginx timers.

Since the `0.7.20` release, The `0` time argument can also be specified.

This method was introduced in the `0.5.0rc30` release.

Back to TOC

ngx.escape_uri

syntax: *newstr = ngx.escape_uri(str)*

context: *init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Escape `str` as a URI component.

Back to TOC

ngx.unescape_uri

syntax: *newstr = ngx.unescape_uri(str)*

context: *init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Unescape `str` as an escaped URI component.

For example,

```
ngx.say(ngx.unescape_uri("b%20r56+7"))
```

gives the output

```
b r56 7
```

Back to TOC

ngx.encode_args

syntax: *str = ngx.encode_args(table)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Encode the Lua table to a query args string according to the URI encoded rules.

For example,

```
ngx.encode_args({foo = 3, ["b r"] = "hello world"})
```

yields

```
foo=3&b%20r=hello%20world
```

The table keys must be Lua strings.

Multi-value query args are also supported. Just use a Lua table for the argument's value, for example:

```
ngx.encode_args({baz = {32, "hello"}})
```

gives

```
baz=32&baz=hello
```

If the value table is empty and the effect is equivalent to the `nil` value.

Boolean argument values are also supported, for instance,

```
ngx.encode_args({a = true, b = 1})
```

yields

```
a&b=1
```

If the argument value is `false`, then the effect is equivalent to the `nil` value.

This method was first introduced in the `v0.3.1rc27` release.

Back to TOC

ngx.decode_args

syntax: *table = ngx.decode_args(str, max_args?)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Decodes a URI encoded query-string into a Lua table. This is the inverse function of ngx.encode_args.

The optional `max_args` argument can be used to specify the maximum number of arguments parsed from the `str` argument. By default, a maximum of 100 request arguments are parsed (including those with the same name) and that additional URI arguments are silently discarded to guard against potential denial of service attacks.

This argument can be set to zero to remove the limit and to process all request arguments received:

```
local args = ngx.decode_args(str, 0)
```

Removing the `max_args` cap is strongly discouraged.

This method was introduced in the `v0.5.0rc29` .

Back to TOC

ngx.encode_base64

syntax: *newstr = ngx.encode_base64(str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Encode `str` to a base64 digest.

Back to TOC

ngx.decode_base64

syntax: *newstr = ngx.decode_base64(str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Decodes the `str` argument as a base64 digest to the raw form. Returns `nil` if `str` is not well formed.

Back to TOC

ngx.crc32_short

syntax: *intval = ngx.crc32_short(str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Calculates the CRC-32 (Cyclic Redundancy Code) digest for the `str` argument.

This method performs better on relatively short `str` inputs (i.e., less than 30 ~ 60 bytes), as compared to ngx.crc32_long. The result is exactly the same as ngx.crc32_long.

Behind the scene, it is just a thin wrapper around the `ngx_crc32_short` function defined in the Nginx core.

This API was first introduced in the `v0.3.1rc8` release.

Back to TOC

ngx.crc32_long

syntax: *intval = ngx.crc32_long(str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Calculates the CRC-32 (Cyclic Redundancy Code) digest for the `str` argument.

This method performs better on relatively long `str` inputs (i.e., longer than 30 ~ 60 bytes), as compared to ngx.crc32_short. The result is exactly the same as ngx.crc32_short.

Behind the scene, it is just a thin wrapper around the `ngx_crc32_long` function defined in the Nginx core.

This API was first introduced in the `v0.3.1rc8` release.

Back to TOC

ngx.hmac_sha1

syntax: *digest = ngx.hmac_sha1(secret_key, str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Computes the HMAC-SHA1 digest of the argument `str` and turns the result using the secret key `<secret_key>`.

The raw binary form of the `HMAC-SHA1` digest will be generated, use ngx.encode_base64, for example, to encode the result to a textual representation if desired.

For example,

```
local key = "thisisverysecretstuff"
local src = "some string we want to sign"
local digest = ngx.hmac_sha1(key, src)
ngx.say(ngx.encode_base64(digest))
```

yields the output

```
R/pvxzHC4NLtj7S+kXFg/NePTmk=
```

This API requires the OpenSSL library enabled in the Nginx build (usually by passing the `--with-http_ssl_module` option to the `./configure` script).

This function was first introduced in the `v0.3.1rc29` release.

Back to TOC

ngx.md5

syntax: *digest = ngx.md5(str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns the hexadecimal representation of the MD5 digest of the `str` argument.

For example,

```
location = /md5 {
    content_by_lua 'ngx.say(ngx.md5("hello"))';
}
```

yields the output

```
5d41402abc4b2a76b9719d911017c592
```

See ngx.md5_bin if the raw binary MD5 digest is required.

Back to TOC

ngx.md5_bin

syntax: *digest = ngx.md5_bin(str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns the binary form of the MD5 digest of the `str` argument.

See ngx.md5 if the hexadecimal form of the MD5 digest is required.

Back to TOC

ngx.sha1_bin

syntax: *digest = ngx.sha1_bin(str)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns the binary form of the SHA-1 digest of the `str` argument.

This function requires SHA-1 support in the Nginx build. (This usually just means OpenSSL should be installed while building Nginx).

This function was first introduced in the `v0.5.0rc6` .

Back to TOC

ngx.quote_sql_str

syntax: *quoted_value = ngx.quote_sql_str(raw_value)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns a quoted SQL string literal according to the MySQL quoting rules.

Back to TOC

ngx.today

syntax: *str = ngx.today()*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns current date (in the format `yyyy-mm-dd`) from the nginx cached time (no syscall involved unlike Lua's date library).

This is the local time.

Back to TOC

ngx.time

syntax: *secs = ngx.time()*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns the elapsed seconds from the epoch for the current time stamp from the nginx cached time (no syscall involved unlike Lua's date library).

Updates of the Nginx time cache an be forced by calling ngx.update_time first.

Back to TOC

ngx.now

syntax: *secs = ngx.now()*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns a floating-point number for the elapsed time in seconds (including milliseconds as the decimal part) from the epoch for the current time stamp from the nginx cached time (no syscall involved unlike Lua's date library).

You can forcibly update the Nginx time cache by calling ngx.update_time first.

This API was first introduced in `v0.3.1rc32`.

Back to TOC

ngx.update_time

syntax: *ngx.update_time()*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Forcibly updates the Nginx current time cache. This call involves a syscall and thus has some overhead, so do not abuse it.

This API was first introduced in `v0.3.1rc32`.

Back to TOC

ngx.localtime

syntax: *str = ngx.localtime()*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns the current time stamp (in the format `yyyy-mm-dd hh:mm:ss`) of the nginx cached time (no syscall involved unlike Lua's os.date function).

This is the local time.

Back to TOC

ngx.utctime

syntax: *str = ngx.utctime()*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns the current time stamp (in the format `yyyy-mm-dd hh:mm:ss`) of the nginx cached time (no syscall involved unlike Lua's os.date function).

This is the UTC time.

Back to TOC

ngx.cookie_time

syntax: *str = ngx.cookie_time(sec)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns a formatted string can be used as the cookie expiration time. The parameter `sec` is the time stamp in seconds (like those returned from ngx.time).

```
ngx.say(ngx.cookie_time(1290079655))
    -- yields "Thu, 18-Nov-10 11:27:35 GMT"
```

Back to TOC

ngx.http_time

syntax: *str = ngx.http_time(sec)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Returns a formated string can be used as the http header time (for example, being used in `Last-`

`Modified` header). The parameter `sec` is the time stamp in seconds (like those returned from ngx.time).

```
ngx.say(ngx.http_time(1290079655))
    -- yields "Thu, 18 Nov 2010 11:27:35 GMT"
```

Back to TOC

ngx.parse_http_time

syntax: *sec = ngx.parse_http_time(str)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Parse the http time string (as returned by ngx.http_time) into seconds. Returns the seconds or `nil` if the input string is in bad forms.

```
local time = ngx.parse_http_time("Thu, 18 Nov 2010 11:27:35 GMT")
if time == nil then
    ...
end
```

Back to TOC

ngx.is_subrequest

syntax: *value = ngx.is_subrequest*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua**

Returns `true` if the current request is an nginx subrequest, or `false` otherwise.

Back to TOC

ngx.re.match

syntax: *captures, err = ngx.re.match(subject, regex, options?, ctx?, res_table?)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Matches the `subject` string using the Perl compatible regular expression `regex` with the optional `options`.

Only the first occurrence of the match is returned, or `nil` if no match is found. In case of errors, like seeing a bad regular expression or exceeding the PCRE stack limit, `nil` and a string describing the error will be returned.

When a match is found, a Lua table `captures` is returned, where `captures[0]` holds the whole substring being matched, and `captures[1]` holds the first parenthesized sub-pattern's capturing, `captures[2]` the second, and so on.

```
local m, err = ngx.re.match("hello, 1234", "[0-9]+")
if m then
    -- m[0] == "1234"

else
    if err then
        ngx.log(ngx.ERR, "error: ", err)
        return
    end

    ngx.say("match not found")
end
```

```
local m, err = ngx.re.match("hello, 1234", "([0-9])[0-9]+")
-- m[0] == "1234"
-- m[1] == "1"
```

Named captures are also supported since the `v0.7.14` release and are returned in the same Lua table as key-value pairs as the numbered captures.

```
local m, err = ngx.re.match("hello, 1234", "([0-9])(?<remaining>[0-9]+)")
-- m[0] == "1234"
-- m[1] == "1"
-- m[2] == "234"
-- m["remaining"] == "234"
```

Unmatched subpatterns will have `nil` values in their `captures` table fields.

```
local m, err = ngx.re.match("hello, world", "(world)|(hello)|(?<named>howdy)")
-- m[0] == "hello"
-- m[1] == nil
-- m[2] == "hello"
-- m[3] == nil
-- m["named"] == nil
```

Specify `options` to control how the match operation will be performed. The following option characters are supported:

```
a             anchored mode (only match from the beginning)

d             enable the DFA mode (or the longest token match semantics).
              this requires PCRE 6.0+ or else a Lua exception will be thrown.
              first introduced in ngx_lua v0.3.1rc30.

D             enable duplicate named pattern support. This allows named
              subpattern names to be repeated, returning the captures in
              an array-like Lua table. for example,
                local m = ngx.re.match("hello, world",
                                       "(?<named>\w+), (?<named>\w+)",
                                       "D")
                -- m["named"] == {"hello", "world"}
              this option was first introduced in the v0.7.14 release.
              this option requires at least PCRE 8.12.

i             case insensitive mode (similar to Perl's /i modifier)

j             enable PCRE JIT compilation, this requires PCRE 8.21+ which
              must be built with the --enable-jit option. for optimum performance,
              this option should always be used together with the 'o' option.
              first introduced in ngx_lua v0.3.1rc30.

J             enable the PCRE Javascript compatible mode. this option was
              first introduced in the v0.7.14 release. this option requires
              at least PCRE 8.12.

m             multi-line mode (similar to Perl's /m modifier)

o             compile-once mode (similar to Perl's /o modifier),
              to enable the worker-process-level compiled-regex cache

s             single-line mode (similar to Perl's /s modifier)

u             UTF-8 mode. this requires PCRE to be built with
              the --enable-utf8 option or else a Lua exception will be thrown.

U             similar to "u" but disables PCRE's UTF-8 validity check on
              the subject string. first introduced in ngx_lua v0.8.1.

x             extended mode (similar to Perl's /x modifier)
```

These options can be combined:

```
local m, err = ngx.re.match("hello, world", "HEL LO", "ix")
-- m[0] == "hello"
```

```
local m, err = ngx.re.match("hello, 美好生活", "HELLO, (.{2})", "iu")
-- m[0] == "hello, 美好"
-- m[1] == "美好"
```

The `o` option is useful for performance tuning, because the regex pattern in question will only be compiled once, cached in the worker-process level, and shared among all requests in the current Nginx worker process. The upper limit of the regex cache can be tuned via the lua_regex_cache_max_entries directive.

The optional fourth argument, `ctx`, can be a Lua table holding an optional `pos` field. When the `pos` field in the `ctx` table argument is specified, `ngx.re.match` will start matching from that offset (starting from 1). Regardless of the presence of the `pos` field in the `ctx` table, `ngx.re.match` will always set this `pos` field to the position *after* the substring matched by the whole pattern in case of a successful match. When match fails, the `ctx` table will be left intact.

```
local ctx = {}
local m, err = ngx.re.match("1234, hello", "[0-9]+", "", ctx)
     -- m[0] = "1234"
     -- ctx.pos == 5
```

```
local ctx = { pos = 2 }
local m, err = ngx.re.match("1234, hello", "[0-9]+", "", ctx)
     -- m[0] = "34"
     -- ctx.pos == 5
```

The `ctx` table argument combined with the `a` regex modifier can be used to construct a lexer atop `ngx.re.match`.

Note that, the `options` argument is not optional when the `ctx` argument is specified and that the empty Lua string (`""`) must be used as placeholder for `options` if no meaningful regex options are required.

This method requires the PCRE library enabled in Nginx. (Known Issue With Special PCRE Sequences).

To confirm that PCRE JIT is enabled, activate the Nginx debug log by adding the `--with-debug` option to Nginx or ngx_openresty's `./configure` script. Then, enable the "debug" error log level in `error_log` directive. The following message will be generated if PCRE JIT is enabled:

```
pcre JIT compiling result: 1
```

Starting from the `0.9.4` release, this function also accepts a 5th argument, `res_table`, for letting the caller supply the Lua table used to hold all the capturing results. Starting from `0.9.6`, it is the caller's responsibility to ensure this table is empty. This is very useful for recycling Lua tables and saving GC and table allocation overhead.

This feature was introduced in the `v0.2.1rc11` release.

Back to TOC

ngx.re.find

syntax: *from, to, err = ngx.re.find(subject, regex, options?, ctx?, nth?)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Similar to ngx.re.match but only returns the begining index (`from`) and end index (`to`) of the matched substring. The returned indexes are 1-based and can be fed directly into the string.sub API function to obtain the matched substring.

In case of errors (like bad regexes or any PCRE runtime errors), this API function returns two `nil` values followed by a string describing the error.

If no match is found, this function just returns a `nil` value.

Below is an example:

```
local s = "hello, 1234"
local from, to, err = ngx.re.find(s, "([0-9]+)", "jo")
if from then
    ngx.say("from: ", from)
    ngx.say("to: ", to)
    ngx.say("matched: ", string.sub(s, from, to))
else
    if err then
        ngx.say("error: ", err)
        return
    end
    ngx.say("not matched!")
end
```

This example produces the output

```
from: 8
to: 11
matched: 1234
```

Because this API function does not create new Lua strings nor new Lua tables, it is much faster than ngx.re.match. It should be used wherever possible.

Since the `0.9.3` release, an optional 5th argument, `nth`, is supported to specify which (submatch) capture's indexes to return. When `nth` is 0 (which is the default), the indexes for the whole matched substring is returned; when `nth` is 1, then the 1st submatch capture's indexes are returned; when `nth` is 2, then the 2nd submatch capture is returned, and so on. When the specified submatch does not have a match, then two `nil` values will be returned. Below is an example for this:

```
local str = "hello, 1234"
local from, to = ngx.re.find(str, "([0-9])([0-9]+)", "jo", nil, 2)
if from then
    ngx.say("matched 2nd submatch: ", string.sub(str, from, to))  -- yields "234"
end
```

This API function was first introduced in the `v0.9.2` release.

Back to TOC

ngx.re.gmatch

syntax: *iterator, err = ngx.re.gmatch(subject, regex, options?)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Similar to ngx.re.match, but returns a Lua iterator instead, so as to let the user programmer iterate all the matches over the `<subject>` string argument with the PCRE `regex`.

In case of errors, like seeing an ill-formed regular expression, `nil` and a string describing the error will be returned.

Here is a small example to demonstrate its basic usage:

```
local iterator, err = ngx.re.gmatch("hello, world!", "([a-z]+)", "i")
if not iterator then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

local m
m, err = iterator()    -- m[0] == m[1] == "hello"
if err then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

m, err = iterator()    -- m[0] == m[1] == "world"
if err then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

m, err = iterator()    -- m == nil
if err then
    ngx.log(ngx.ERR, "error: ", err)
    return
end
```

More often we just put it into a Lua loop:

```lua
local it, err = ngx.re.gmatch("hello, world!", "([a-z]+)", "i")
if not it then
    ngx.log(ngx.ERR, "error: ", err)
    return
end

while true do
    local m, err = it()
    if err then
        ngx.log(ngx.ERR, "error: ", err)
        return
    end

    if not m then
        -- no match found (any more)
        break
    end

    -- found a match
    ngx.say(m[0])
    ngx.say(m[1])
end
```

The optional `options` argument takes exactly the same semantics as the ngx.re.match method.

The current implementation requires that the iterator returned should only be used in a single request. That is, one should *not* assign it to a variable belonging to persistent namespace like a Lua package.

This method requires the PCRE library enabled in Nginx. (Known Issue With Special PCRE Sequences).

This feature was first introduced in the `v0.2.1rc12` release.

Back to TOC

ngx.re.sub

syntax: *newstr, n, err = ngx.re.sub(subject, regex, replace, options?)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Substitutes the first match of the Perl compatible regular expression `regex` on the `subject` argument string with the string or function argument `replace`. The optional `options` argument has exactly the same meaning as in ngx.re.match.

This method returns the resulting new string as well as the number of successful substitutions. In case of failures, like syntax errors in the regular expressions or the `<replace>` string argument, it will return `nil` and a string describing the error.

When the `replace` is a string, then it is treated as a special template for string replacement. For example,

```lua
local newstr, n, err = ngx.re.sub("hello, 1234", "([0-9])[0-9]", "[$0][$1]")
if newstr then
    -- newstr == "hello, [12][1]34"
    -- n == 1
else
    ngx.log(ngx.ERR, "error: ", err)
    return
end
```

where `$0` referring to the whole substring matched by the pattern and `$1` referring to the first parenthesized capturing substring.

Curly braces can also be used to disambiguate variable names from the background string literals:

```
local newstr, n, err = ngx.re.sub("hello, 1234", "[0-9]", "${0}00")
    -- newstr == "hello, 10034"
    -- n == 1
```

Literal dollar sign characters (`$`) in the `replace` string argument can be escaped by another dollar sign, for instance,

```
local newstr, n, err = ngx.re.sub("hello, 1234", "[0-9]", "$$")
    -- newstr == "hello, $234"
    -- n == 1
```

Do not use backlashes to escape dollar signs; it will not work as expected.

When the `replace` argument is of type "function", then it will be invoked with the "match table" as the argument to generate the replace string literal for substitution. The "match table" fed into the `replace` function is exactly the same as the return value of ngx.re.match. Here is an example:

```
local func = function (m)
    return "[" .. m[0] .. "][" .. m[1] .. "]"
end
local newstr, n, err = ngx.re.sub("hello, 1234", "( [0-9] ) [0-9]", func, "x")
    -- newstr == "hello, [12][1]34"
    -- n == 1
```

The dollar sign characters in the return value of the `replace` function argument are not special at all.

This method requires the PCRE library enabled in Nginx. (Known Issue With Special PCRE Sequences).

This feature was first introduced in the `v0.2.1rc13` release.

Back to TOC

ngx.re.gsub

syntax: *newstr, n, err = ngx.re.gsub(subject, regex, replace, options?)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Just like ngx.re.sub, but does global substitution.

Here is some examples:

```
local newstr, n, err = ngx.re.gsub("hello, world", "([a-z])[a-z]+", "[$0,$1]", "i")
if newstr then
    -- newstr == "[hello,h], [world,w]"
    -- n == 2
```

```
else
    ngx.log(ngx.ERR, "error: ", err)
    return
end


local func = function (m)
    return "[" .. m[0] .. "," .. m[1] .. "]"
end
local newstr, n, err = ngx.re.gsub("hello, world", "([a-z])[a-z]+", func, "i")
    -- newstr == "[hello,h], [world,w]"
    -- n == 2
```

This method requires the PCRE library enabled in Nginx. (Known Issue With Special PCRE Sequences).

This feature was first introduced in the `v0.2.1rc15` release.

Back to TOC

ngx.shared.DICT

syntax: *dict = ngx.shared.DICT*

syntax: *dict = ngx.shared[name_var]*

context: *init_by_lua*, init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Fetching the shm-based Lua dictionary object for the shared memory zone named `DICT` defined by the lua_shared_dict directive.

Shared memory zones are always shared by all the nginx worker processes in the current nginx server instance.

The resulting object `dict` has the following methods:

- get
- get_stale
- set
- safe_set
- add
- safe_add
- replace
- delete
- incr
- flush_all
- flush_expired
- get_keys

Here is an example:

```
http {
    lua_shared_dict dogs 10m;
    server {
        location /set {
            content_by_lua '
                local dogs = ngx.shared.dogs
```

```
                    dogs:set("Jim", 8)
                    ngx.say("STORED")
                ';
            }
            location /get {
                content_by_lua '
                    local dogs = ngx.shared.dogs
                    ngx.say(dogs:get("Jim"))
                ';
            }
        }
    }
```

Let us test it:

```
$ curl localhost/set
STORED

$ curl localhost/get
8

$ curl localhost/get
8
```

The number `8` will be consistently output when accessing `/get` regardless of how many Nginx workers there are because the `dogs` dictionary resides in the shared memory and visible to *all* of the worker processes.

The shared dictionary will retain its contents through a server config reload (either by sending the `HUP` signal to the Nginx process or by using the `-s reload` command-line option).

The contents in the dictionary storage will be lost, however, when the Nginx server quits.

This feature was first introduced in the `v0.3.1rc22` release.

Back to TOC

ngx.shared.DICT.get

syntax: *value, flags = ngx.shared.DICT:get(key)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Retrieving the value in the dictionary ngx.shared.DICT for the key `key`. If the key does not exist or has been expired, then `nil` will be returned.

In case of errors, `nil` and a string describing the error will be returned.

The value returned will have the original data type when they were inserted into the dictionary, for example, Lua booleans, numbers, or strings.

The first argument to this method must be the dictionary object itself, for example,

```
local cats = ngx.shared.cats
local value, flags = cats.get(cats, "Marry")
```

or use Lua's syntactic sugar for method calls:

```
local cats = ngx.shared.cats
local value, flags = cats:get("Marry")
```

These two forms are fundamentally equivalent.

If the user flags is `0` (the default), then no flags value will be returned.

This feature was first introduced in the `v0.3.1rc22` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.get_stale

syntax: *value, flags, stale = ngx.shared.DICT:get_stale(key)*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Similar to the get method but returns the value even if the key has already expired.

Returns a 3rd value, `stale`, indicating whether the key has expired or not.

Note that the value of an expired key is not guaranteed to be available so one should never rely on the availability of expired items.

This method was first introduced in the `0.8.6` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.set

syntax: *success, err, forcible = ngx.shared.DICT:set(key, value, exptime?, flags?)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Unconditionally sets a key-value pair into the shm-based dictionary ngx.shared.DICT. Returns three values:

- `success`: boolean value to indicate whether the key-value pair is stored or not.
- `err`: textual error message, can be `"no memory"`.
- `forcible`: a boolean value to indicate whether other valid items have been removed forcibly when out of storage in the shared memory zone.

The `value` argument inserted can be Lua booleans, numbers, strings, or `nil`. Their value type will also be stored into the dictionary and the same data type can be retrieved later via the get method.

The optional `exptime` argument specifies expiration time (in seconds) for the inserted key-value pair. The time resolution is `0.001` seconds. If the `exptime` takes the value `0` (which is the default), then the item will never be expired.

The optional `flags` argument specifies a user flags value associated with the entry to be stored. It can also be retrieved later with the value. The user flags is stored as an unsigned 32-bit integer internally. Defaults to `0` . The user flags argument was first introduced in the `v0.5.0rc2` release.

When it fails to allocate memory for the current key-value item, then `set` will try removing existing items in the storage according to the Least-Recently Used (LRU) algorithm. Note that, LRU takes priority over expiration time here. If up to tens of existing items have been removed and the storage left is still insufficient (either due to the total capacity limit specified by lua_shared_dict or memory segmentation), then the `err` return value will be `no memory` and `success` will be `false` .

If this method succeeds in storing the current item by forcibly removing other not-yet-expired items in the dictionary via LRU, the `forcible` return value will be `true` . If it stores the item without forcibly removing other valid items, then the return value `forcible` will be `false` .

The first argument to this method must be the dictionary object itself, for example,

```
local cats = ngx.shared.cats
local succ, err, forcible = cats.set(cats, "Marry", "it is a nice cat!")
```

or use Lua's syntactic sugar for method calls:

```
local cats = ngx.shared.cats
local succ, err, forcible = cats:set("Marry", "it is a nice cat!")
```

These two forms are fundamentally equivalent.

This feature was first introduced in the `v0.3.1rc22` release.

Please note that while internally the key-value pair is set atomically, the atomicity does not go across the method call boundary.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.safe_set

syntax: *ok, err = ngx.shared.DICT:safe_set(key, value, exptime?, flags?)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Similar to the set method, but never overrides the (least recently used) unexpired items in the store when running out of storage in the shared memory zone. In this case, it will immediately return `nil` and the string "no memory".

This feature was first introduced in the `v0.7.18` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.add

syntax: *success, err, forcible = ngx.shared.DICT:add(key, value, exptime?, flags?)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Just like the set method, but only stores the key-value pair into the dictionary ngx.shared.DICT if the key does *not* exist.

If the `key` argument already exists in the dictionary (and not expired for sure), the `success` return value will be `false` and the `err` return value will be `"exists"`.

This feature was first introduced in the `v0.3.1rc22` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.safe_add

syntax: *ok, err = ngx.shared.DICT:safe_add(key, value, exptime?, flags?)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Similar to the add method, but never overrides the (least recently used) unexpired items in the store when running out of storage in the shared memory zone. In this case, it will immediately return `nil` and the string "no memory".

This feature was first introduced in the `v0.7.18` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.replace

syntax: *success, err, forcible = ngx.shared.DICT:replace(key, value, exptime?, flags?)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Just like the set method, but only stores the key-value pair into the dictionary ngx.shared.DICT if the key *does* exist.

If the `key` argument does *not* exist in the dictionary (or expired already), the `success` return value will be `false` and the `err` return value will be `"not found"`.

This feature was first introduced in the `v0.3.1rc22` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.delete

syntax: *ngx.shared.DICT:delete(key)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Unconditionally removes the key-value pair from the shm-based dictionary ngx.shared.DICT.

It is equivalent to `ngx.shared.DICT:set(key, nil)`.

This feature was first introduced in the `v0.3.1rc22` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.incr

syntax: *newval, err = ngx.shared.DICT:incr(key, value)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Increments the (numerical) value for `key` in the shm-based dictionary ngx.shared.DICT by the step value `value`. Returns the new resulting number if the operation is successfully completed or `nil` and an error message otherwise.

The key must already exist in the dictionary, otherwise it will return `nil` and `"not found"`.

If the original value is not a valid Lua number in the dictionary, it will return `nil` and `"not a number"`.

The `value` argument can be any valid Lua numbers, like negative numbers or floating-point numbers.

This feature was first introduced in the `v0.3.1rc22` release.

See also ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.flush_all

syntax: *ngx.shared.DICT:flush_all()*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Flushes out all the items in the dictionary. This method does not actuall free up all the memory blocks in the dictionary but just marks all the existing items as expired.

This feature was first introduced in the `v0.5.0rc17` release.

See also ngx.shared.DICT.flush_expired and ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.flush_expired

syntax: *flushed = ngx.shared.DICT:flush_expired(max_count?)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Flushes out the expired items in the dictionary, up to the maximal number specified by the optional `max_count` argument. When the `max_count` argument is given `0` or not given at all, then it means unlimited. Returns the number of items that have actually been flushed.

Unlike the flush_all method, this method actually free up the memory used by the expired items.

This feature was first introduced in the `v0.6.3` release.

See also ngx.shared.DICT.flush_all and ngx.shared.DICT.

Back to TOC

ngx.shared.DICT.get_keys

syntax: *keys = ngx.shared.DICT:get_keys(max_count?)*

context: *init_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Fetch a list of the keys from the dictionary, up to `<max_count>`.

By default, only the first 1024 keys (if any) are returned. When the `<max_count>` argument is given the value `0`, then all the keys will be returned even there is more than 1024 keys in the dictionary.

WARNING Be careful when calling this method on dictionaries with a really huge number of keys. This method may lock the dictionary for quite a while and block all the nginx worker processes that are trying to access the dictionary.

This feature was first introduced in the `v0.7.3` release.

Back to TOC

ngx.socket.udp

syntax: *udpsock = ngx.socket.udp()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Creates and returns a UDP or datagram-oriented unix domain socket object (also known as one type of the "cosocket" objects). The following methods are supported on this object:

- setpeername
- send
- receive
- close
- settimeout

It is intended to be compatible with the UDP API of the LuaSocket library but is 100% nonblocking out of the box.

This feature was first introduced in the `v0.5.7` release.

See also ngx.socket.tcp.

Back to TOC

udpsock:setpeername

syntax: *ok, err = udpsock:setpeername(host, port)*

syntax: *ok, err = udpsock:setpeername("unix:/path/to/unix-domain.socket")*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Attempts to connect a UDP socket object to a remote server or to a datagram unix domain socket file. Because the datagram protocol is actually connection-less, this method does not really establish a "connection", but only just set the name of the remote peer for subsequent read/write operations.

Both IP addresses and domain names can be specified as the `host` argument. In case of domain names, this method will use Nginx core's dynamic resolver to parse the domain name without blocking and it is required to configure the resolver directive in the `nginx.conf` file like this:

```
resolver 8.8.8.8;  # use Google's public DNS nameserver
```

If the nameserver returns multiple IP addresses for the host name, this method will pick up one randomly.

In case of error, the method returns `nil` followed by a string describing the error. In case of success, the method returns `1`.

Here is an example for connecting to a UDP (memcached) server:

```
location /test {
    resolver 8.8.8.8;

    content_by_lua '
        local sock = ngx.socket.udp()
        local ok, err = sock:setpeername("my.memcached.server.domain", 11211)
        if not ok then
            ngx.say("failed to connect to memcached: ", err)
            return
        end
        ngx.say("successfully connected to memcached!")
        sock:close()
    ';
}
```

Since the `v0.7.18` release, connecting to a datagram unix domain socket file is also possible on Linux:

```
local sock = ngx.socket.udp()
local ok, err = sock:setpeername("unix:/tmp/some-datagram-service.sock")
if not ok then
    ngx.say("failed to connect to the datagram unix domain socket: ", err)
    return
end
```

assuming the datagram service is listening on the unix domain socket file `/tmp/some-datagram-service.sock` and the client socket will use the "autobind" feature on Linux.

Calling this method on an already connected socket object will cause the original connection to be closed first.

This method was first introduced in the `v0.5.7` release.

Back to TOC

udpsock:send

syntax: *ok, err = udpsock:send(data)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Sends data on the current UDP or datagram unix domain socket object.

In case of success, it returns `1`. Otherwise, it returns `nil` and a string describing the error.

The input argument `data` can either be a Lua string or a (nested) Lua table holding string fragments. In case of table arguments, this method will copy all the string elements piece by piece to the underlying Nginx socket send buffers, which is usually optimal than doing string concatenation operations on the Lua land.

This feature was first introduced in the `v0.5.7` release.

Back to TOC

udpsock:receive

syntax: *data, err = udpsock:receive(size?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Receives data from the UDP or datagram unix domain socket object with an optional receive buffer size argument, `size`.

This method is a synchronous operation and is 100% nonblocking.

In case of success, it returns the data received; in case of error, it returns `nil` with a string describing the error.

If the `size` argument is specified, then this method will use this size as the receive buffer size. But when this size is greater than `8192`, then `8192` will be used instead.

If no argument is specified, then the maximal buffer size, `8192` is assumed.

Timeout for the reading operation is controlled by the lua_socket_read_timeout config directive and the settimeout method. And the latter takes priority. For example:

```
sock:settimeout(1000)  -- one second timeout
local data, err = sock:receive()
if not data then
    ngx.say("failed to read a packet: ", data)
    return
end
ngx.say("successfully read a packet: ", data)
```

It is important here to call the settimeout method *before* calling this method.

This feature was first introduced in the `v0.5.7` release.

Back to TOC

udpsock:close

syntax: *ok, err = udpsock:close()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Closes the current UDP or datagram unix domain socket. It returns the `1` in case of success and returns `nil` with a string describing the error otherwise.

Socket objects that have not invoked this method (and associated connections) will be closed when the socket object is released by the Lua GC (Garbage Collector) or the current client HTTP request finishes processing.

This feature was first introduced in the `v0.5.7` release.

Back to TOC

udpsock:settimeout

syntax: *udpsock:settimeout(time)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Set the timeout value in milliseconds for subsequent socket operations (like receive).

Settings done by this method takes priority over those config directives, like lua_socket_read_timeout.

This feature was first introduced in the `v0.5.7` release.

Back to TOC

ngx.socket.tcp

syntax: *tcpsock = ngx.socket.tcp()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Creates and returns a TCP or stream-oriented unix domain socket object (also known as one type of the "cosocket" objects). The following methods are supported on this object:

- connect
- sslhandshake
- send
- receive
- close
- settimeout
- setoption
- receiveuntil
- setkeepalive
- getreusedtimes

It is intended to be compatible with the TCP API of the LuaSocket library but is 100% nonblocking out of the box. Also, we introduce some new APIs to provide more functionalities.

The cosocket object created by this API function has exactly the same lifetime as the Lua handler creating it. So never pass the cosocket object to any other Lua handler (including ngx.timer callback functions) and never share the cosocket object between different NGINX requests.

For every cosocket object's underlying connection, if you do not explicitly close it (via close) or put it back to the connection pool (via setkeepalive), then it is automatically closed when one of the following two events happens:

- the current request handler completes, or
- the Lua cosocket object value gets collected by the Lua GC.

Fatal errors in cosocket operations always automatically close the current connection (note that, read timeout error is the only error that is not fatal), and if you call close on a closed connection, you will get the "closed" error.

Starting from the `0.9.9` release, the cosocket object here is full-duplex, that is, a reader "light thread" and a writer "light thread" can operate on a single cosocket object simultaneously (both "light threads" must belong to the same Lua handler though, see reasons above). But you cannot have two "light threads" both reading (or writing or connecting) the same cosocket, otherwise you might get an error like "socket busy reading" when calling the methods of the cosocket object.

This feature was first introduced in the `v0.5.0rc1` release.

See also ngx.socket.udp.

Back to TOC

tcpsock:connect

syntax: *ok, err = tcpsock:connect(host, port, options_table?)*

syntax: *ok, err = tcpsock:connect("unix:/path/to/unix-domain.socket", options_table?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Attempts to connect a TCP socket object to a remote server or to a stream unix domain socket file without blocking.

Before actually resolving the host name and connecting to the remote backend, this method will always look up the connection pool for matched idle connections created by previous calls of this method (or the ngx.socket.connect function).

Both IP addresses and domain names can be specified as the `host` argument. In case of domain names, this method will use Nginx core's dynamic resolver to parse the domain name without blocking and it is required to configure the resolver directive in the `nginx.conf` file like this:

```
resolver 8.8.8.8;  # use Google's public DNS nameserver
```

If the nameserver returns multiple IP addresses for the host name, this method will pick up one randomly.

In case of error, the method returns `nil` followed by a string describing the error. In case of success, the method returns `1`.

Here is an example for connecting to a TCP server:

```
location /test {
    resolver 8.8.8.8;

    content_by_lua '
        local sock = ngx.socket.tcp()
```

```
            local ok, err = sock:connect("www.google.com", 80)
            if not ok then
                ngx.say("failed to connect to google: ", err)
                return
            end
            ngx.say("successfully connected to google!")
            sock:close()
        ';
    }
```

Connecting to a Unix Domain Socket file is also possible:

```
local sock = ngx.socket.tcp()
local ok, err = sock:connect("unix:/tmp/memcached.sock")
if not ok then
    ngx.say("failed to connect to the memcached unix domain socket: ", err)
    return
end
```

assuming memcached (or something else) is listening on the unix domain socket file `/tmp/memcached.sock`.

Timeout for the connecting operation is controlled by the lua_socket_connect_timeout config directive and the settimeout method. And the latter takes priority. For example:

```
local sock = ngx.socket.tcp()
sock:settimeout(1000)  -- one second timeout
local ok, err = sock:connect(host, port)
```

It is important here to call the settimeout method *before* calling this method.

Calling this method on an already connected socket object will cause the original connection to be closed first.

An optional Lua table can be specified as the last argument to this method to specify various connect options:

- `pool` specify a custom name for the connection pool being used. If omitted, then the connection pool name will be generated from the string template `"<host>:<port>"` or `"<unix-socket-path>"`.

The support for the options table argument was first introduced in the `v0.5.7` release.

This method was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:sslhandshake

syntax: *session, err = tcpsock:sslhandshake(reused_session?, server_name?, ssl_verify?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Does SSL/TLS handshake on the currently established connection.

The optional `reused_session` argument can take a former SSL session userdata returned by a previous `sslhandshake` call for exactly the same target. For short-lived connections, reusing SSL sessions can usually speed up the handshake by one order by magnitude but it is not so useful if the connection pool is enabled. This argument defaults to `nil`. If this argument takes the boolean `false` value, no SSL session userdata would return by this call and only a Lua boolean will be returned as the first return value; otherwise the current SSL session will always be returned as the first argument in case of successes.

The optional `server_name` argument is used to specify the server name for the new TLS extension Server Name Indication (SNI). Use of SNI can make different servers share the same IP address on the server side. Also, when SSL verification is enabled, this `server_name` argument is also used to validate the server name specified in the server certificate sent from the remote.

The optional `ssl_verify` argument takes a Lua boolean value to control whether to perform SSL verification. When set to `true`, the server certificate will be verified according to the CA certificates specified by the lua_ssl_trusted_certificate directive. You may also need to adjust the lua_ssl_verify_depth directive to control how deep we should follow along the certificate chain. Also, when the `ssl_verify` argument is true and the `server_name` argument is also specified, the latter will be used to validate the server name in the server certificate.

For connections that have already done SSL/TLS handshake, this method returns immediately.

This method was first introduced in the `v0.9.11` release.

Back to TOC

tcpsock:send

syntax: *bytes, err = tcpsock:send(data)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Sends data without blocking on the current TCP or Unix Domain Socket connection.

This method is a synchronous operation that will not return until *all* the data has been flushed into the system socket send buffer or an error occurs.

In case of success, it returns the total number of bytes that have been sent. Otherwise, it returns `nil` and a string describing the error.

The input argument `data` can either be a Lua string or a (nested) Lua table holding string fragments. In case of table arguments, this method will copy all the string elements piece by piece to the underlying Nginx socket send buffers, which is usually optimal than doing string concatenation operations on the Lua land.

Timeout for the sending operation is controlled by the lua_socket_send_timeout config directive and the settimeout method. And the latter takes priority. For example:

```
sock:settimeout(1000)  -- one second timeout
local bytes, err = sock:send(request)
```

It is important here to call the settimeout method *before* calling this method.

In case of any connection errors, this method always automatically closes the current connection.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:receive

syntax: *data, err, partial = tcpsock:receive(size)*

syntax: *data, err, partial = tcpsock:receive(pattern?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Receives data from the connected socket according to the reading pattern or size.

This method is a synchronous operation just like the send method and is 100% nonblocking.

In case of success, it returns the data received; in case of error, it returns `nil` with a string describing the error and the partial data received so far.

If a number-like argument is specified (including strings that look like numbers), then it is interpreted as a size. This method will not return until it reads exactly this size of data or an error occurs.

If a non-number-like string argument is specified, then it is interpreted as a "pattern". The following patterns are supported:

- `'*a'` : reads from the socket until the connection is closed. No end-of-line translation is performed;
- `'*l'` : reads a line of text from the socket. The line is terminated by a `Line Feed` (LF) character (ASCII 10), optionally preceded by a `Carriage Return` (CR) character (ASCII 13). The CR and LF characters are not included in the returned line. In fact, all CR characters are ignored by the pattern.

If no argument is specified, then it is assumed to be the pattern `'*l'` , that is, the line reading pattern.

Timeout for the reading operation is controlled by the lua_socket_read_timeout config directive and the settimeout method. And the latter takes priority. For example:

```
sock:settimeout(1000)  -- one second timeout
local line, err, partial = sock:receive()
if not line then
    ngx.say("failed to read a line: ", err)
    return
end
ngx.say("successfully read a line: ", line)
```

It is important here to call the settimeout method *before* calling this method.

Since the `v0.8.8` release, this method no longer automatically closes the current connection when the read timeout error happens. For other connection errors, this method always automatically closes the connection.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:receiveuntil

syntax: *iterator = tcpsock:receiveuntil(pattern, options?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

This method returns an iterator Lua function that can be called to read the data stream until it sees the specified pattern or an error occurs.

Here is an example for using this method to read a data stream with the boundary sequence `--abcedhb` :

```
local reader = sock:receiveuntil("\r\n--abcedhb")
local data, err, partial = reader()
if not data then
    ngx.say("failed to read the data stream: ", err)
end
ngx.say("read the data stream: ", data)
```

When called without any argument, the iterator function returns the received data right *before* the specified pattern string in the incoming data stream. So for the example above, if the incoming data stream is `'hello, world! -agentzh\r\n--abcedhb blah blah'` , then the string `'hello, world! -agentzh'` will be returned.

In case of error, the iterator function will return `nil` along with a string describing the error and the partial data bytes that have been read so far.

The iterator function can be called multiple times and can be mixed safely with other cosocket method calls or other iterator function calls.

The iterator function behaves differently (i.e., like a real iterator) when it is called with a `size` argument. That is, it will read that `size` of data on each invocation and will return `nil` at the last invocation (either sees the boundary pattern or meets an error). For the last successful invocation of the iterator function, the `err` return value will be `nil` too. The iterator function will be reset after the last successful invocation that returns `nil` data and `nil` error. Consider the following example:

```
local reader = sock:receiveuntil("\r\n--abcedhb")

while true do
    local data, err, partial = reader(4)
    if not data then
        if err then
            ngx.say("failed to read the data stream: ", err)
            break
        end

        ngx.say("read done")
        break
    end
    ngx.say("read chunk: [", data, "]")
end
```

Then for the incoming data stream `'hello, world! -agentzh\r\n--abcedhb blah blah'` , we shall get the following output from the sample code above:

```
read chunk: [hell]
read chunk: [o, w]
read chunk: [orld]
read chunk: [! -a]
read chunk: [gent]
read chunk: [zh]
read done
```

Note that, the actual data returned *might* be a little longer than the size limit specified by the `size` argument when the boundary pattern has ambiguity for streaming parsing. Near the boundary of the data stream, the data string actually returned could also be shorter than the size limit.

Timeout for the iterator function's reading operation is controlled by the lua_socket_read_timeout config directive and the settimeout method. And the latter takes priority. For example:

```
local readline = sock:receiveuntil("\r\n")

sock:settimeout(1000)  -- one second timeout
line, err, partial = readline()
if not line then
    ngx.say("failed to read a line: ", err)
    return
end
ngx.say("successfully read a line: ", line)
```

It is important here to call the settimeout method *before* calling the iterator function (note that the `receiveuntil` call is irrelevant here).

As from the `v0.5.1` release, this method also takes an optional `options` table argument to control the behavior. The following options are supported:

- `inclusive`

The `inclusive` takes a boolean value to control whether to include the pattern string in the returned data string. Default to `false`. For example,

```
local reader = tcpsock:receiveuntil("_END_", { inclusive = true })
local data = reader()
ngx.say(data)
```

Then for the input data stream `"hello world _END_ blah blah blah"`, then the example above will output `hello world _END_`, including the pattern string `_END_` itself.

Since the `v0.8.8` release, this method no longer automatically closes the current connection when the read timeout error happens. For other connection errors, this method always automatically closes the connection.

This method was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:close

syntax: *ok, err = tcpsock:close()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Closes the current TCP or stream unix domain socket. It returns the `1` in case of success and returns `nil` with a string describing the error otherwise.

Note that there is no need to call this method on socket objects that have invoked the setkeepalive method because the socket object is already closed (and the current connection is saved into the built-in connection pool).

Socket objects that have not invoked this method (and associated connections) will be closed when the socket object is released by the Lua GC (Garbage Collector) or the current client HTTP request finishes processing.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:settimeout

syntax: *tcpsock:settimeout(time)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Set the timeout value in milliseconds for subsequent socket operations (connect, receive, and iterators returned from receiveuntil).

Settings done by this method takes priority over those config directives, i.e., lua_socket_connect_timeout, lua_socket_send_timeout, and lua_socket_read_timeout.

Note that this method does *not* affect the lua_socket_keepalive_timeout setting; the `timeout` argument to the setkeepalive method should be used for this purpose instead.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:setoption

syntax: *tcpsock:setoption(option, value?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

This function is added for LuaSocket API compatibility and does nothing for now. Its functionality will be implemented in future.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:setkeepalive

syntax: *ok, err = tcpsock:setkeepalive(timeout?, size?)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Puts the current socket's connection immediately into the cosocket built-in connection pool and keep it alive until other connect method calls request it or the associated maximal idle timeout is expired.

The first optional argument, `timeout` , can be used to specify the maximal idle timeout (in milliseconds) for the current connection. If omitted, the default setting in the lua_socket_keepalive_timeout config directive will be used. If the `0` value is given, then the timeout interval is unlimited.

The second optional argument, `size` , can be used to specify the maximal number of connections allowed in the connection pool for the current server (i.e., the current host-port pair or the unix domain socket file path). Note that the size of the connection pool cannot be changed once the pool is created. When this argument is omitted, the default setting in the lua_socket_pool_size config directive will be used.

When the connection pool exceeds the available size limit, the least recently used (idle) connection already in the pool will be closed to make room for the current connection.

Note that the cosocket connection pool is per Nginx worker process rather than per Nginx server instance, so the size limit specified here also applies to every single Nginx worker process.

Idle connections in the pool will be monitored for any exceptional events like connection abortion or unexpected incoming data on the line, in which cases the connection in question will be closed and removed from the pool.

In case of success, this method returns `1` ; otherwise, it returns `nil` and a string describing the error.

This method also makes the current cosocket object enter the "closed" state, so there is no need to manually call the close method on it afterwards.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

tcpsock:getreusedtimes

syntax: *count, err = tcpsock:getreusedtimes()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

This method returns the (successfully) reused times for the current connection. In case of error, it returns `nil` and a string describing the error.

If the current connection does not come from the built-in connection pool, then this method always returns `0` , that is, the connection has never been reused (yet). If the connection comes from the connection pool, then the return value is always non-zero. So this method can also be used to determine if the current connection comes from the pool.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

ngx.socket.connect

syntax: *tcpsock, err = ngx.socket.connect(host, port)*

syntax: *tcpsock, err = ngx.socket.connect("unix:/path/to/unix-domain.socket")*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

This function is a shortcut for combining ngx.socket.tcp() and the connect() method call in a single operation. It is actually implemented like this:

```
local sock = ngx.socket.tcp()
local ok, err = sock:connect(...)
if not ok then
    return nil, err
end
return sock
```

There is no way to use the settimeout method to specify connecting timeout for this method and the lua_socket_connect_timeout directive must be set at configure time instead.

This feature was first introduced in the `v0.5.0rc1` release.

Back to TOC

ngx.get_phase

syntax: *str = ngx.get_phase()*

context: *init_by_lua, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Retrieves the current running phase name. Possible return values are

- `init` for the context of init_by_lua or init_by_lua_file.
- `set` for the context of set_by_lua or set_by_lua_file.
- `rewrite` for the context of rewrite_by_lua or rewrite_by_lua_file.
- `access` for the context of access_by_lua or access_by_lua_file.
- `content` for the context of content_by_lua or content_by_lua_file.
- `header_filter` for the context of header_filter_by_lua or header_filter_by_lua_file.
- `body_filter` for the context of body_filter_by_lua or body_filter_by_lua_file.
- `log` for the context of log_by_lua or log_by_lua_file.
- `timer` for the context of user callback functions for ngx.timer.*.

This API was first introduced in the `v0.5.10` release.

Back to TOC

ngx.thread.spawn

syntax: *co = ngx.thread.spawn(func, arg1, arg2, ...)*

context: *rewrite_by_lua, access_by_lua*, content_by_lua*, ngx.timer.**

Spawns a new user "light thread" with the Lua function `func` as well as those optional arguments `arg1`, `arg2`, and etc. Returns a Lua thread (or Lua coroutine) object represents this "light thread".

"Light threads" are just a special kind of Lua coroutines that are scheduled by the ngx_lua module.

Before `ngx.thread.spawn` returns, the `func` will be called with those optional arguments until it returns, aborts with an error, or gets yielded due to I/O operations via the Nginx API for Lua (like tcpsock:receive).

After `ngx.thread.spawn` returns, the newly-created "light thread" will keep running asynchronously usually at various I/O events.

All the Lua code chunks running by rewrite_by_lua, access_by_lua, and content_by_lua are in a boilerplate "light thread" created automatically by ngx_lua. Such boilerplate "light thread" are also called "entry threads".

By default, the corresponding Nginx handler (e.g., rewrite_by_lua handler) will not terminate until

1. both the "entry thread" and all the user "light threads" terminates,
2. a "light thread" (either the "entry thread" or a user "light thread" aborts by calling ngx.exit, ngx.exec, ngx.redirect, or ngx.req.set_uri(uri, true), or
3. the "entry thread" terminates with a Lua error.

When the user "light thread" terminates with a Lua error, however, it will not abort other running "light threads" like the "entry thread" does.

Due to the limitation in the Nginx subrequest model, it is not allowed to abort a running Nginx subrequest in general. So it is also prohibited to abort a running "light thread" that is pending on one ore more Nginx subrequests. You must call ngx.thread.wait to wait for those "light thread" to terminate before quitting the "world". A notable exception here is that you can abort pending subrequests by calling ngx.exit with and only with the status code `ngx.ERROR` (-1), `408` , `444` , or `499` .

The "light threads" are not scheduled in a pre-emptive way. In other words, no time-slicing is performed automatically. A "light thread" will keep running exclusively on the CPU until

1. a (nonblocking) I/O operation cannot be completed in a single run,
2. it calls coroutine.yield to actively give up execution, or
3. it is aborted by a Lua error or an invocation of ngx.exit, ngx.exec, ngx.redirect, or ngx.req.set_uri(uri, true).

For the first two cases, the "light thread" will usually be resumed later by the ngx_lua scheduler unless a "stop-the-world" event happens.

User "light threads" can create "light threads" themselves. And normal user coroutines created by coroutine.create can also create "light threads". The coroutine (be it a normal Lua coroutine or a "light thread") that directly spawns the "light thread" is called the "parent coroutine" for the "light thread" newly spawned.

The "parent coroutine" can call ngx.thread.wait to wait on the termination of its child "light thread".

You can call coroutine.status() and coroutine.yield() on the "light thread" coroutines.

The status of the "light thread" coroutine can be "zombie" if

1. the current "light thread" already terminates (either successfully or with an error),
2. its parent coroutine is still alive, and
3. its parent coroutine is not waiting on it with ngx.thread.wait.

The following example demonstrates the use of coroutine.yield() in the "light thread" coroutines to do manual time-slicing:

```
local yield = coroutine.yield

function f()
    local self = coroutine.running()
    ngx.say("f 1")
    yield(self)
    ngx.say("f 2")
    yield(self)
    ngx.say("f 3")
end

local self = coroutine.running()
ngx.say("0")
yield(self)

ngx.say("1")
ngx.thread.spawn(f)

ngx.say("2")
yield(self)
```

```
ngx.say("3")
yield(self)

ngx.say("4")
```

Then it will generate the output

```
0
1
f 1
2
f 2
3
f 3
4
```

"Light threads" are mostly useful for doing concurrent upstream requests in a single Nginx request handler, kinda like a generalized version of ngx.location.capture_multi that can work with all the Nginx API for Lua. The following example demonstrates parallel requests to MySQL, Memcached, and upstream HTTP services in a single Lua handler, and outputting the results in the order that they actually return (very much like the Facebook BigPipe model):

```
-- query mysql, memcached, and a remote http service at the same time,
-- output the results in the order that they
-- actually return the results.

local mysql = require "resty.mysql"
local memcached = require "resty.memcached"

local function query_mysql()
    local db = mysql:new()
    db:connect{
                host = "127.0.0.1",
                port = 3306,
                database = "test",
                user = "monty",
                password = "mypass"
              }
    local res, err, errno, sqlstate =
            db:query("select * from cats order by id asc")
    db:set_keepalive(0, 100)
    ngx.say("mysql done: ", cjson.encode(res))
end

local function query_memcached()
    local memc = memcached:new()
    memc:connect("127.0.0.1", 11211)
    local res, err = memc:get("some_key")
    ngx.say("memcached done: ", res)
end

local function query_http()
    local res = ngx.location.capture("/my-http-proxy")
    ngx.say("http done: ", res.body)
end

ngx.thread.spawn(query_mysql)      -- create thread 1
ngx.thread.spawn(query_memcached)  -- create thread 2
ngx.thread.spawn(query_http)       -- create thread 3
```

This API was first enabled in the `v0.7.0` release.

Back to TOC

ngx.thread.wait

syntax: *ok, res1, res2, ... = ngx.thread.wait(thread1, thread2, ...)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Waits on one or more child "light threads" and returns the results of the first "light thread" that terminates (either successfully or with an error).

The arguments `thread1`, `thread2`, and etc are the Lua thread objects returned by earlier calls of ngx.thread.spawn.

The return values have exactly the same meaning as coroutine.resume, that is, the first value returned is a boolean value indicating whether the "light thread" terminates successfully or not, and subsequent values returned are the return values of the user Lua function that was used to spawn the "light thread" (in case of success) or the error object (in case of failure).

Only the direct "parent coroutine" can wait on its child "light thread", otherwise a Lua exception will be raised.

The following example demonstrates the use of `ngx.thread.wait` and ngx.location.capture to emulate ngx.location.capture_multi:

```lua
local capture = ngx.location.capture
local spawn = ngx.thread.spawn
local wait = ngx.thread.wait
local say = ngx.say

local function fetch(uri)
    return capture(uri)
end

local threads = {
    spawn(fetch, "/foo"),
    spawn(fetch, "/bar"),
    spawn(fetch, "/baz")
}

for i = 1, #threads do
    local ok, res = wait(threads[i])
    if not ok then
        say(i, ": failed to run: ", res)
    else
        say(i, ": status: ", res.status)
        say(i, ": body: ", res.body)
    end
end
```

Here it essentially implements the "wait all" model.

And below is an example demonstrating the "wait any" model:

```lua
function f()
    ngx.sleep(0.2)
    ngx.say("f: hello")
    return "f done"
end
```

```
function g()
    ngx.sleep(0.1)
    ngx.say("g: hello")
    return "g done"
end

local tf, err = ngx.thread.spawn(f)
if not tf then
    ngx.say("failed to spawn thread f: ", err)
    return
end

ngx.say("f thread created: ", coroutine.status(tf))

local tg, err = ngx.thread.spawn(g)
if not tg then
    ngx.say("failed to spawn thread g: ", err)
    return
end

ngx.say("g thread created: ", coroutine.status(tg))

ok, res = ngx.thread.wait(tf, tg)
if not ok then
    ngx.say("failed to wait: ", res)
    return
end

ngx.say("res: ", res)

-- stop the "world", aborting other running threads
ngx.exit(ngx.OK)
```

And it will generate the following output:

```
f thread created: running
g thread created: running
g: hello
res: g done
```

This API was first enabled in the `v0.7.0` release.

Back to TOC

ngx.thread.kill

syntax: *ok, err = ngx.thread.kill(thread)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, ngx.timer.**

Kills a running "light thread" created by ngx.thread.spawn. Returns a true value when successful or `nil` and a string describing the error otherwise.

According to the current implementation, only the parent coroutine (or "light thread") can kill a thread. Also, a running "light thread" with pending NGINX subrequests (initiated by ngx.location.capture for example) cannot be killed due to a limitation in the NGINX core.

This API was first enabled in the `v0.9.9` release.

Back to TOC

ngx.on_abort

syntax: *ok, err = ngx.on_abort(callback)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua**

Registers a user Lua function as the callback which gets called automatically when the client closes the (downstream) connection prematurely.

Returns `1` if the callback is registered successfully or returns `nil` and a string describing the error otherwise.

All the Nginx API for Lua can be used in the callback function because the function is run in a special "light thread", just as those "light threads" created by ngx.thread.spawn.

The callback function can decide what to do with the client abortion event all by itself. For example, it can simply ignore the event by doing nothing and the current Lua request handler will continue executing without interruptions. And the callback function can also decide to terminate everything by calling ngx.exit, for example,

```
local function my_cleanup()
    -- custom cleanup work goes here, like cancelling a pending DB transaction

    -- now abort all the "light threads" running in the current request handler
    ngx.exit(499)
end

local ok, err = ngx.on_abort(my_cleanup)
if not ok then
    ngx.log(ngx.ERR, "failed to register the on_abort callback: ", err)
    ngx.exit(500)
end
```

When lua_check_client_abort is set to `off` (which is the default), then this function call will always return the error message "lua_check_client_abort is off".

According to the current implementation, this function can only be called once in a single request handler; subsequent calls will return the error message "duplicate call".

This API was first introduced in the `v0.7.4` release.

See also lua_check_client_abort.

Back to TOC

ngx.timer.at

syntax: *ok, err = ngx.timer.at(delay, callback, user_arg1, user_arg2, ...)*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

Creates an Nginx timer with a user callback function as well as optional user arguments.

The first argument, `delay`, specifies the delay for the timer, in seconds. One can specify fractional seconds like `0.001` to mean 1 millisecond here. `0` delay can also be specified, in which case the timer will immediately expire when the current handler yields execution.

The second argument, `callback`, can be any Lua function, which will be invoked later in a background "light thread" after the delay specified. The user callback will be called automatically by the Nginx core with the arguments `premature`, `user_arg1`, `user_arg2`, and etc, where the `premature` argument takes a boolean value indicating whether it is a premature timer expiration or not, and `user_arg1`, `user_arg2`, and etc, are those (extra) user arguments specified when calling `ngx.timer.at` as the remaining arguments.

Premature timer expiration happens when the Nginx worker process is trying to shut down, as in an Nginx configuration reload triggered by the `HUP` signal or in an Nginx server shutdown. When the Nginx worker is trying to shut down, one can no longer call `ngx.timer.at` to create new timers with nonzero delays and in that case `ngx.timer.at` will return `nil` and a string describing the error, that is, "process exiting".

Starting from the `v0.9.3` release, it is allowed to create zero-delay timers even when the Nginx worker process starts shutting down.

When a timer expires, the user Lua code in the timer callback is running in a "light thread" detached completely from the original request creating the timer. So objects with the same lifetime as the request creating them, like cosockets, cannot be shared between the original request and the timer user callback function.

Here is a simple example:

```
location / {
    ...
    log_by_lua '
        local function push_data(premature, uri, args, status)
            -- push the data uri, args, and status to the remote
            -- via ngx.socket.tcp or ngx.socket.udp
            -- (one may want to buffer the data in Lua a bit to
            -- save I/O operations)
        end
        local ok, err = ngx.timer.at(0, push_data,
                                     ngx.var.uri, ngx.var.args, ngx.header.status)
        if not ok then
            ngx.log(ngx.ERR, "failed to create timer: ", err)
            return
        end
    ';
}
```

One can also create infinite re-occuring timers, for instance, a timer getting triggered every `5` seconds, by calling `ngx.timer.at` recursively in the timer callback function. Here is such an example,

```
local delay = 5
local handler
handler = function (premature)
    -- do some routine job in Lua just like a cron job
    if premature then
        return
    end
    local ok, err = ngx.timer.at(delay, handler)
    if not ok then
        ngx.log(ngx.ERR, "failed to create the timer: ", err)
        return
    end
end

local ok, err = ngx.timer.at(delay, handler)
```

```
    if not ok then
        ngx.log(ngx.ERR, "failed to create the timer: ", err)
        return
    end
```

Because timer callbacks run in the background and their running time will not add to any client request's response time, they can easily accumulate in the server and exhaust system resources due to either Lua programming mistakes or just too much client traffic. To prevent extreme consequences like crashing the Nginx server, there are built-in limitations on both the number of "pending timers" and the number of "running timers" in an Nginx worker process. The "pending timers" here mean timers that have not yet been expired and "running timers" are those whose user callbacks are currently running.

The maximal number of pending timers allowed in an Nginx worker is controlled by the lua_max_pending_timers directive. The maximal number of running timers is controlled by the lua_max_running_timers directive.

According to the current implementation, each "running timer" will take one (fake) connection record from the global connection record list configured by the standard worker_connections directive in `nginx.conf`. So ensure that the worker_connections directive is set to a large enough value that takes into account both the real connections and fake connections required by timer callbacks (as limited by the lua_max_running_timers directive).

A lot of the Lua APIs for Nginx are enabled in the context of the timer callbacks, like stream/datagram cosockets (ngx.socket.tcp and ngx.socket.udp), shared memory dictionaries (ngx.shared.DICT), user coroutines (coroutine.*), user "light threads" (ngx.thread.*), ngx.exit, ngx.now/ngx.time, ngx.md5/ngx.sha1_bin, are all allowed. But the subrequest API (like ngx.location.capture), the ngx.req.* API, the downstream output API (like ngx.say, ngx.print, and ngx.flush) are explicitly disabled in this context.

You can pass most of the standard Lua values (nils, booleans, numbers, strings, tables, closures, file handles, and etc) into the timer callback, either explicitly as user arguments or implicitly as upvalues for the callback closure. There are several exceptions, however: you *cannot* pass any thread objects returned by coroutine.create and ngx.thread.spawn or any cosocket objects returned by ngx.socket.tcp, ngx.socket.udp, and ngx.req.socket because these objects' lifetime is bound to the request context creating them while the timer callback is detached from the creating request's context (by design) and runs in its own (fake) request context. If you try to share the thread or cosocket objects across the boundary of the creating request, then you will get the "no co ctx found" error (for threads) or "bad request" (for cosockets). It is fine, however, to create all these objects inside your timer callback.

This API was first introduced in the `v0.8.0` release.

Back to TOC

ngx.config.debug

syntax: *debug = ngx.config.debug*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, *init_by_lua*, init_worker_by_lua**

This boolean field indicates whether the current Nginx is a debug build, i.e., being built by the `./configure` option `--with-debug`.

This field was first introduced in the `0.8.7`.

Back to TOC

ngx.config.prefix

syntax: *prefix = ngx.config.prefix()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, *init_by_lua*, init_worker_by_lua**

Returns the Nginx server "prefix" path, as determined by the `-p` command-line option when running the nginx executable, or the path specified by the `--prefix` command-line option when building Nginx with the `./configure` script.

This function was first introduced in the `0.9.2`.

Back to TOC

ngx.config.nginx_version

syntax: *ver = ngx.config.nginx_version*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, *init_by_lua*, init_worker_by_lua**

This field take an integral value indicating the version number of the current Nginx core being used. For example, the version number `1.4.3` results in the Lua number 1004003.

This API was first introduced in the `0.9.3` release.

Back to TOC

ngx.config.nginx_configure

syntax: *str = ngx.config.nginx_configure()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, *init_by_lua**

This function returns a string for the NGINX `./configure` command's arguments string.

This API was first introduced in the `0.9.5` release.

Back to TOC

ngx.config.ngx_lua_version

syntax: *ver = ngx.config.ngx_lua_version*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, *init_by_lua**

This field take an integral value indicating the version number of the current `ngx_lua` module being used. For example, the version number `0.9.3` results in the Lua number 9003.

This API was first introduced in the `0.9.3` release.

Back to TOC

ngx.worker.exiting

syntax: *exiting = ngx.worker.exiting()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua*, init_worker_by_lua**

This function returns a boolean value indicating whether the current Nginx worker process already starts exiting. Nginx worker process exiting happens on Nginx server quit or configuration reload (aka HUP reload).

This API was first introduced in the `0.9.3` release.

Back to TOC

ngx.worker.pid

syntax: *pid = ngx.worker.pid()*

context: *set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.*, init_by_lua*, init_worker_by_lua**

This function returns a Lua number for the process ID (PID) of the current Nginx worker process. This API is more efficient than `ngx.var.pid` and can be used in contexts where the ngx.var.VARIABLE API cannot be used (like init_worker_by_lua).

This API was first introduced in the `0.9.5` release.

Back to TOC

ndk.set_var.DIRECTIVE

syntax: *res = ndk.set_var.DIRECTIVE_NAME*

context: *init_worker_by_lua*, set_by_lua*, rewrite_by_lua*, access_by_lua*, content_by_lua*, header_filter_by_lua*, body_filter_by_lua*, log_by_lua*, ngx.timer.**

This mechanism allows calling other nginx C modules' directives that are implemented by Nginx Devel Kit (NDK)'s set_var submodule's `ndk_set_var_value` .

For example, the following set-misc-nginx-module directives can be invoked this way:

- set_quote_sql_str
- set_quote_pgsql_str
- set_quote_json_str
- set_unescape_uri
- set_escape_uri
- set_encode_base32
- set_decode_base32
- set_encode_base64
- set_decode_base64
- set_encode_hex
- set_decode_hex

- set_sha1
- set_md5

For instance,

```
local res = ndk.set_var.set_escape_uri('a/b');
-- now res == 'a%2fb'
```

Similarly, the following directives provided by encrypted-session-nginx-module can be invoked from within Lua too:

- set_encrypt_session
- set_decrypt_session

This feature requires the ngx_devel_kit module.

Back to TOC

coroutine.create

syntax: *co = coroutine.create(f)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, init_by_lua*, ngx.timer.*, *header_filter_by_lua*, body_filter_by_lua**

Creates a user Lua coroutines with a Lua function, and returns a coroutine object.

Similar to the standard Lua coroutine.create API, but works in the context of the Lua coroutines created by ngx_lua.

This API was first usable in the context of init_by_lua* since the `0.9.2`.

This API was first introduced in the `v0.6.0` release.

Back to TOC

coroutine.resume

syntax: *ok, ... = coroutine.resume(co, ...)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, init_by_lua*, ngx.timer.*, *header_filter_by_lua*, body_filter_by_lua**

Resumes the executation of a user Lua coroutine object previously yielded or just created.

Similar to the standard Lua coroutine.resume API, but works in the context of the Lua coroutines created by ngx_lua.

This API was first usable in the context of init_by_lua* since the `0.9.2`.

This API was first introduced in the `v0.6.0` release.

Back to TOC

coroutine.yield

syntax: *... = coroutine.yield(...)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, init_by_lua*, ngx.timer.*, *header_filter_by_lua*, body_filter_by_lua**

Yields the executation of the current user Lua coroutine.

Similar to the standard Lua coroutine.yield API, but works in the context of the Lua coroutines created by ngx_lua.

This API was first usable in the context of init_by_lua* since the `0.9.2`.

This API was first introduced in the `v0.6.0` release.

Back to TOC

coroutine.wrap

syntax: *co = coroutine.wrap(f)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, init_by_lua*, ngx.timer.*, *header_filter_by_lua*, body_filter_by_lua**

Similar to the standard Lua coroutine.wrap API, but works in the context of the Lua coroutines created by ngx_lua.

This API was first usable in the context of init_by_lua* since the `0.9.2`.

This API was first introduced in the `v0.6.0` release.

Back to TOC

coroutine.running

syntax: *co = coroutine.running()*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, init_by_lua*, ngx.timer.*, *header_filter_by_lua*, body_filter_by_lua**

Identical to the standard Lua coroutine.running API.

This API was first usable in the context of init_by_lua* since the `0.9.2`.

This API was first enabled in the `v0.6.0` release.

Back to TOC

coroutine.status

syntax: *status = coroutine.status(co)*

context: *rewrite_by_lua*, access_by_lua*, content_by_lua*, init_by_lua*, ngx.timer.*, *header_filter_by_lua*, body_filter_by_lua**

Identical to the standard Lua coroutine.status API.

This API was first usable in the context of init_by_lua* since the `0.9.2`.

This API was first enabled in the `v0.6.0` release.

Back to TOC