openresty / **lua-resty-websocket**

WebSocket support for the ngx_lua module (and OpenResty)

64 commits 1 branch 4 releases 2 contributors

branch: master **lua-resty-websocket** / +

doc: mentioned the bitop library dependency when using the standard L… ···

agentzh authored 9 days ago latest commit 505f577434

lib/resty/websocket	bumped version to 0.04.	22 days ago
t	tests: fixed test cases that could fail incorrectly in the "HUP reloa…	2 months ago
.gitignore	updated .gitignore a bit.	a year ago
Makefile	added a Makefile.	a year ago
README.markdown	doc: mentioned the bitop library dependency when using the standard L…	9 days ago
valgrind.suppress	suppressed a false positive in libdl.	5 months ago

README.markdown

Name

lua-resty-websocket - Lua WebSocket implementation for the ngx_lua module

Table of Contents

Status

This library is considered production ready.

Description

This Lua library implements a WebSocket server and client libraries based on the ngx_lua module.

This Lua library takes advantage of ngx_lua's cosocket API, which ensures 100% nonblocking behavior.

Note that only RFC 6455 is supported. Earlier protocol revisions like "hybi-10", "hybi-07", and "hybi-00" are not and will not be considered.

Synopsis

```
local server = require "resty.websocket.server"

local wb, err = server:new{
    timeout = 5000,  -- in milliseconds
    max_payload_len = 65535,
}
if not wb then
    ngx.log(ngx.ERR, "failed to new websocket: ", err)
    return ngx.exit(444)
end
```

```
local data, typ, err = wb:recv_frame()

if not data then
    ngx.log(ngx.ERR, "failed to receive a frame: ", err)
    return ngx.exit(444)
end

if typ == "close" then
    -- send a close frame back:

    local bytes, err = wb:send_close(1000, "enough, enough!")
    if not bytes then
        ngx.log(ngx.ERR, "failed to send the close frame: ", err)
        return
    end
    local code = err
    ngx.log(ngx.INFO, "closing with status code ", code, " and message ", data)
    return
end

if typ == "ping" then
    -- send a pong frame back:

    local bytes, err = wb:send_pong(data)
    if not bytes then
        ngx.log(ngx.ERR, "failed to send frame: ", err)
        return
    end
elseif typ == "pong" then
    -- just discard the incoming pong frame

else
    ngx.log(ngx.INFO, "received a frame of type ", typ, " and payload ", data)
end

wb:set_timeout(1000)  -- change the network timeout to 1 second

bytes, err = wb:send_text("Hello world")
if not bytes then
    ngx.log(ngx.ERR, "failed to send a text frame: ", err)
    return ngx.exit(444)
end

bytes, err = wb:send_binary("blah blah blah...")
if not bytes then
    ngx.log(ngx.ERR, "failed to send a binary frame: ", err)
    return ngx.exit(444)
end

local bytes, err = wb:send_close(1000, "enough, enough!")
if not bytes then
    ngx.log(ngx.ERR, "failed to send the close frame: ", err)
    return
end
```

Back to TOC

Modules

Back to TOC

resty.websocket.server

To load this module, just do this

```
local server = require "resty.websocket.server"
```

Back to TOC

Methods

Back to TOC

new

```
syntax: wb, err = server:new()
```

```
syntax: wb, err = server:new(opts)
```

Performs the websocket handshake process on the server side and returns a WebSocket server object.

In case of error, it returns `nil` and a string describing the error.

An optional options table can be specified. The following options are as follows:

- `max_payload_len`

  Specifies the maximal length of payload allowed when sending and receiving WebSocket frames.

- `send_masked`

  Specifies whether to send out masked WebSocket frames. When it is `true`, masked frames are always sent. Default to `false`.

- `timeout`

  Specifies the network timeout threshold in milliseconds. You can change this setting later via the `set_timeout` method call. Note that this timeout setting does not affect the HTTP response header sending process for the websocket handshake; you need to configure the send_timeout directive at the same time.

Back to TOC

set_timeout

```
syntax: wb:set_timeout(ms)
```

Sets the timeout delay (in milliseconds) for the network-related operations.

Back to TOC

send_text

```
syntax: bytes, err = wb:send_text(text)
```

Sends the `text` argument out as an unfragmented data frame of the `text` type. Returns the number of bytes that have actually been sent on the TCP level.

In case of errors, returns `nil` and a string describing the error.

Back to TOC

send_binary

```
syntax: bytes, err = wb:send_binary(data)
```

Sends the `data` argument out as an unfragmented data frame of the `binary` type. Returns the number of bytes that have actually been sent on the TCP level.

In case of errors, returns `nil` and a string describing the error.

Back to TOC

send_ping

```
syntax: bytes, err = wb:send_ping()

syntax: bytes, err = wb:send_ping(msg)
```

Sends out a `ping` frame with an optional message specified by the `msg` argument. Returns the number of bytes that have actually been sent on the TCP level.

In case of errors, returns `nil` and a string describing the error.

Note that this method does not wait for a pong frame from the remote end.

Back to TOC

send_pong

```
syntax: bytes, err = wb:send_pong()

syntax: bytes, err = wb:send_pong(msg)
```

Sends out a `pong` frame with an optional message specified by the `msg` argument. Returns the number of bytes that have actually been sent on the TCP level.

In case of errors, returns `nil` and a string describing the error.

Back to TOC

send_close

```
syntax: bytes, err = wb:send_close()

syntax: bytes, err = wb:send_close(code, msg)
```

Sends out a `close` frame with an optional status code and a message.

In case of errors, returns `nil` and a string describing the error.

For a list of valid status code, see the following document:

http://tools.ietf.org/html/rfc6455#section-7.4.1

Note that this method does not wait for a `close` frame from the remote end.

Back to TOC

send_frame

```
syntax: bytes, err = wb:send_frame(fin, opcode, payload)
```

Sends out a raw websocket frame by specifying the `fin` field (boolean value), the opcode, and the payload.

For a list of valid opcode, see

http://tools.ietf.org/html/rfc6455#section-5.2

In case of errors, returns `nil` and a string describing the error.

To control the maximal payload length allowed, you can pass the `max_payload_len` option to the `new` constructor.

To control whether to send masked frames, you can pass `true` to the `send_masked` option in the `new` constructor method. By default, unmasked frames are sent.

Back to TOC

recv_frame

```
syntax: data, typ, err = wb:recv_frame()
```

Receives a WebSocket frame from the wire.

In case of an error, returns two `nil` values and a string describing the error.

The second return value is always the frame type, which could be one of `continuation`, `text`, `binary`, `close`, `ping`, `pong`, or `nil` (for unknown types).

For `close` frames, returns 3 values: the extra status message (which could be an empty string), the string "close", and a Lua number for the status code (if any). For possible closing status codes, see

http://tools.ietf.org/html/rfc6455#section-7.4.1

For other types of frames, just returns the payload and the type.

For fragmented frames, the `err` return value is the Lua string "again".

Back to TOC

resty.websocket.client

To load this module, just do this

```
local client = require "resty.websocket.client"
```

A simple example to demonstrate the usage:

```
local client = require "resty.websocket.client"
local wb, err = client:new()
local uri = "ws://127.0.0.1:" .. ngx.var.server_port .. "/s"
local ok, err = wb:connect(uri)
if not ok then
    ngx.say("failed to connect: " .. err)
    return
end

local data, typ, err = wb:recv_frame()
if not data then
    ngx.say("failed to receive the frame: ", err)
    return
end
```

```
    ngx.say("received: ", data, " (", typ, "): ", err)

    local bytes, err = wb:send_text("copy: " .. data)
    if not bytes then
        ngx.say("failed to send frame: ", err)
        return
    end

    local bytes, err = wb:send_close()
    if not bytes then
        ngx.say("failed to send frame: ", err)
        return
    end
```

Back to TOC

Methods

Back to TOC

client:new

```
syntax: wb, err = client:new()

syntax: wb, err = client:new(opts)
```

Instantiates a WebSocket client object.

In case of error, it returns `nil` and a string describing the error.

An optional options table can be specified. The following options are as follows:

- `max_payload_len`

  Specifies the maximal length of payload allowed when sending and receiving WebSocket frames.

- `send_unmasked`

  Specifies whether to send out an unmasked WebSocket frames. When it is `true`, unmasked frames are always sent. Default to `false`. RFC 6455 requires, however, that the client MUST send masked frames to the server, so never set this option to `true` unless you know what you are doing.

- `timeout`

  Specifies the default network timeout threshold in milliseconds. You can change this setting later via the `set_timeout` method call.

Back to TOC

client:connect

```
syntax: ok, err = wb:connect("ws://<host>:<port>/<path>")

syntax: ok, err = wb:connect("ws://<host>:<port>/<path>", options)
```

Connects to the remote WebSocket service port and performs the websocket handshake process on the client side.

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.

An optional Lua table can be specified as the last argument to this method to specify various connect options:

- `protocols`

  Specifies all the subprotocols used for the current WebSocket session. It could be a Lua table holding all the subprotocol names or just a single Lua string.

- `origin`

  Specifies the value of the `Origin` request header.

- `pool`

  Specifies 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>` .

Back to TOC

client:close

```
syntax: ok, err = wb:close()
```

Closes the current WebSocket connection. If no `close` frame is sent yet, then the `close` frame will be automatically sent.

Back to TOC

client:set_keepalive

```
syntax: ok, err = wb:set_keepalive(max_idle_timeout, pool_size)
```

Puts the current Redis connection immediately into the ngx_lua cosocket connection pool.

You can specify the max idle timeout (in ms) when the connection is in the pool and the maximal size of the pool every nginx worker process.

In case of success, returns `1` . In case of errors, returns `nil` with a string describing the error.

Only call this method in the place you would have called the `close` method instead. Calling this method will immediately turn the current redis object into the `closed` state. Any subsequent operations other than `connect()` on the current objet will return the `closed` error.

Back to TOC

client:set_timeout

```
syntax: wb:set_timeout(ms)
```

Identical to the `set_timeout` method of the `resty.websocket.server` objects.

Back to TOC

client:send_text

```
syntax: bytes, err = wb:send_text(text)
```

Identical to the send_text method of the `resty.websocket.server` objects.

Back to TOC

client:send_binary

```
syntax: bytes, err = wb:send_binary(data)
```

Identical to the send_binary method of the `resty.websocket.server` objects.

Back to TOC

client:send_ping

```
syntax: bytes, err = wb:send_ping()

syntax: bytes, err = wb:send_ping(msg)
```

Identical to the send_ping method of the `resty.websocket.server` objects.

Back to TOC

client:send_pong

```
syntax: bytes, err = wb:send_pong()

syntax: bytes, err = wb:send_pong(msg)
```

Identical to the send_pong method of the `resty.websocket.server` objects.

Back to TOC

client:send_close

```
syntax: bytes, err = wb:send_close()

syntax: bytes, err = wb:send_close(code, msg)
```

Identical to the send_close method of the `resty.websocket.server` objects.

Back to TOC

client:send_frame

```
syntax: bytes, err = wb:send_frame(fin, opcode, payload)
```

Identical to the send_frame method of the `resty.websocket.server` objects.

To control whether to send unmasked frames, you can pass `true` to the `send_unmasked` option in the `new` constructor method. By default, masked frames are sent.

Back to TOC

client:recv_frame

```
syntax: data, typ, err = wb:recv_frame()
```

Identical to the recv_frame method of the `resty.websocket.server` objects.

Back to TOC

resty.websocket.protocol

To load this module, just do this

```
local protocol = require "resty.websocket.protocol"
```

Back to TOC

Methods

Back to TOC

recv_frame

```
syntax: data, typ, err = protocol.recv_frame(socket, max_payload_len, force_masking)
```

Receives a WebSocket frame from the wire.

Back to TOC

build_frame

```
syntax: frame = protocol.build_frame(fin, opcode, payload_len, payload, masking)
```

Builds a raw WebSocket frame.

Back to TOC

send_frame

```
syntax: bytes, err = protocol.send_frame(socket, fin, opcode, payload, max_payload_len, masking)
```

Sends a raw WebSocket frame.

Back to TOC

Automatic Error Logging

By default the underlying ngx_lua module does error logging when socket errors happen. If you are already doing proper error handling in your own Lua code, then you are recommended to disable this automatic error logging by turning off ngx_lua's lua_socket_log_errors directive, that is,

```
lua_socket_log_errors off;
```

Back to TOC

Limitations

- This library cannot be used in code contexts like init_by_lua*, set_by_lua*, log_by_lua*, and header_filter_by_lua* where the ngx_lua cosocket API is not available.
- The `resty.websocket` object instance cannot be stored in a Lua variable at the Lua module level, because it will then be shared by all the concurrent requests handled by the same nginx worker process (see http://wiki.nginx.org/HttpLuaModule#Data_Sharing_within_an_Nginx_Worker) and result in bad race conditions when concurrent requests are trying to use the same `resty.websocket` instance. You should always initiate `resty.websocket` objects in function local variables or in

the `ngx.ctx` table. These places all have their own data copies for each request.

Back to TOC

Installation

It is recommended to use the latest ngx_openresty bundle directly where this library is bundled and enabled by default. At least ngx_openresty 1.4.2.9 is required. And you need to enable LuaJIT when building your ngx_openresty bundle by passing the `--with-luajit` option to its `./configure` script. No extra Nginx configuration is required.

If you want to use this library with your own Nginx build (with ngx_lua), then you need to ensure you are using at least ngx_lua 0.9.0 (and lua-bitop library if you are not using LuaJIT). Also, You need to configure the lua_package_path directive to add the path of your lua-resty-websocket source tree to ngx_lua's Lua module search path, as in

```
# nginx.conf
http {
    lua_package_path "/path/to/lua-resty-websocket/lib/?.lua;;";
    ...
}
```

and then load the library in Lua:

```
local server = require "resty.websocket.server"
```

Back to TOC

TODO

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

Bugs and Patches

Please report bugs or submit patches by

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

Back to TOC

Author

Yichun "agentzh" Zhang (章亦春) agentzh@gmail.com, CloudFlare Inc.

Back to TOC

Copyright and License

Back to TOC

See Also

- Blog post WebSockets with OpenResty by Aapo Talvensaari.
- the ngx_lua module: http://wiki.nginx.org/HttpLuaModule
- the websocket protocol: http://tools.ietf.org/html/rfc6455
- the lua-resty-upload library
- the lua-resty-redis library
- the lua-resty-memcached library
- the lua-resty-mysql library

Back to TOC