如何编写正确且高效的OpenResty应用

自我介绍

GitHub: https://github.com/spacewander

具有多年 OpenResty 开发经验

OpenResty Inc. 员工

第一部分
OpenResty

init_by_lua* 何时运行

- nginx -T / nginx -s
- nginx.pid

ngx.worker.id() 是否唯一

- reload
- binary upgrade

LRU 和 shdict:safe_op

LRU version	safe version
add	safe_add
set	safe_set

勿忘 update time

Nginx 只在开始处理事件时才会更新缓存的时间

所以：

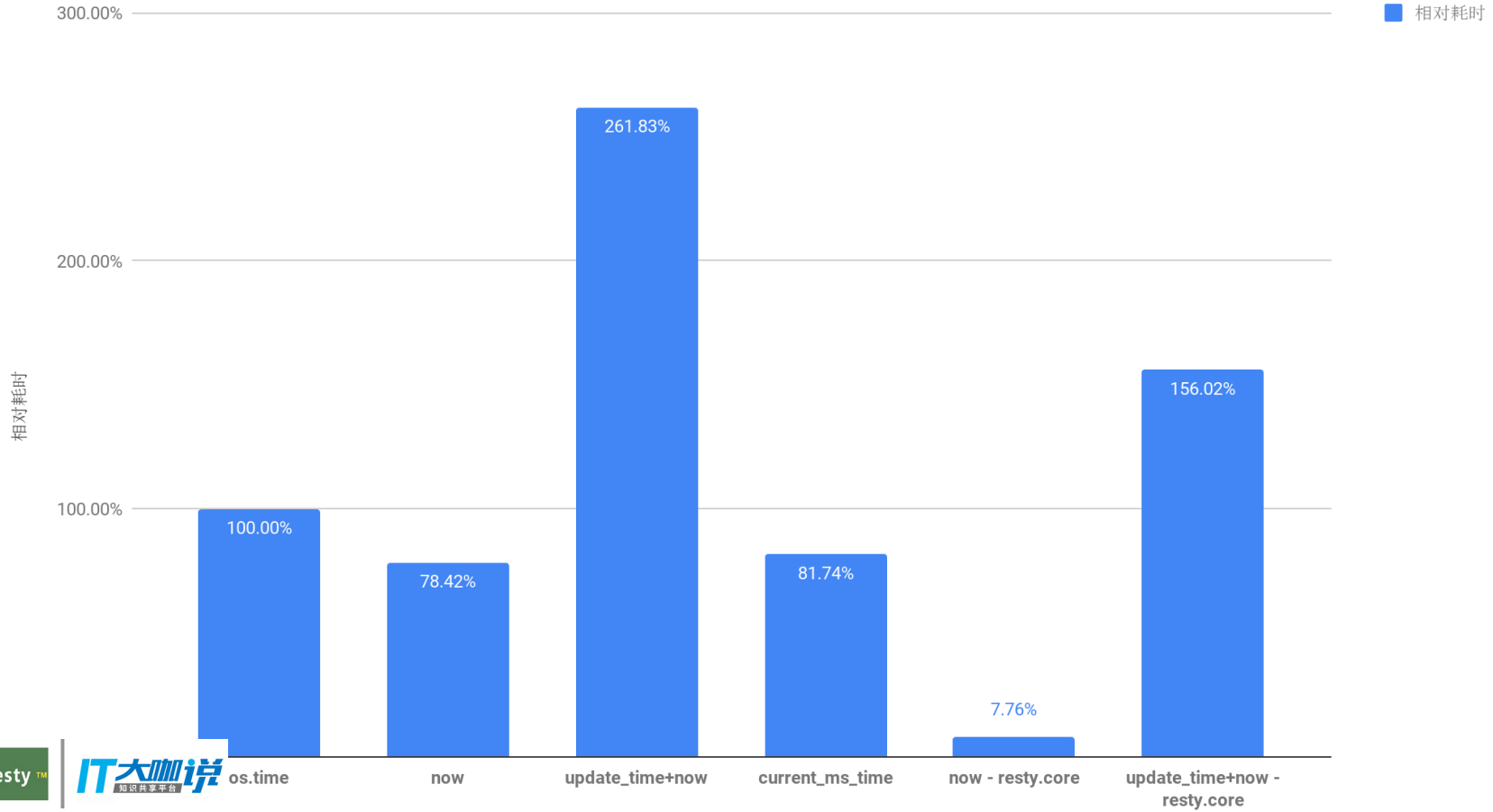
1. 尽量避免 blocking 操作
2. 如果需要耗时的操作，完成后调用 ngx.update_time
   需要准确的时间戳，不要直接调用 ngx.now()

获取准确时间的操作 - 性能比较

- os.time()
- ngx.now()
- ngx.update() + ngx.now()
- current_ms_time
- ngx.now() (resty.core.time version)
- ngx.update() + ngx.now() (resty.core.time version)

```
local function current_ms_time()
        local rc =
ffi.C.gettimeofday(tv[0], nil)
        if rc ~= 0 then
            error(errno())
        end
        return
tonumber(tv[0].tv_sec) +
tonumber(tv[0].tv_usec) / 1e6
end
```

测试在支持 vDSO 的 Linux 上进行



有限的 timer

ngx.timer 的总量受限于三个参数：

1. worker_connections
2. lua_max_pending_timers
3. lua_max_running_timers

确保它们足够大。定期执行的 timer，用 ngx.timer.every 而不是 ngx.timer.at

节省 ngx.timer 可以改善性能

批处理/队列/时间轮?

复用 timer / 长连接的挑战

r->pool?

复用 timer / 长连接的挑战

ctx->user_co_ctx?

第二部分
LuaJIT

不可变的 string

1. O(1) 的比较
2. 更小的内存占用

1. 无法把字符串当作 byte[] 使用
2. 操作字符串时开销大
3. 昂贵的 lj_str_new

昂贵且不安全的 lj_str_new

默认 hash 函数有碰撞的风险：https://github.com/LuaJIT/LuaJIT/issues/168

OpenResty 自带的 LuaJIT 改用硬件加速的 CRC32 实现了相关的 hash 函数：https://github.com/openresty/luajit2/commit/7923c634

这个 hash 函数只有在 x64 和支持 SSE4.2 的环境里才会开启。

依旧昂贵的 lj_str_new

1. 使用 table.concat 和 string.format
2. 调用 cosocket:send 和 ngx.say/ngx.log 时传递 table 而不是 string
3. 避免无谓的字符串创建，比如用 string.byte 代替 string.sub

谁可以代替 byte[]?

1. table
2. FFI buffer

把 table 当 buffer 用

table.new (创建）

table.clear （清空）

table.clone （浅复制）

table.remove (移除）

table.insert / t[#t + 1] （插入）

table.clone

https://github.com/openresty/luajit2/commit/617f118

直接调用 LuaJIT 内部的 lj_tab_dup

buffer 复用

1. 共享变量
2. https://github.com/openresty/lua-tablepool

共享变量

```
local do_sth
do
        local global_ctx = table.new(32, 0)
        function do_sth()
                local ctx = global_ctx
                -- should not yield
        end
```

tablepool

```
local tablepool = require "tablepool"

local tablepool_fetch = tablepool.fetch

local tablepool_release = tablepool.release

local function do_sth()
        local ctx = tablepool_fetch("my_pool", 32, 0)

        -- yield

        tablepool_release("my_pool", ctx)
```

避免在 table 中间存储 nil

```
local s = {0, 1, nil, 2, 3, nil}

print(#s) -- get 5
```

避免在 table 中间存储 nil

1. nil 对 #table 的返回结果的影响
2. nil 对 unpack 的影响
3. for i in ipairs() 和 for i = 1, #table 对 nil 的不同处理
4. 应该使用 ngx.null 作为占位符

有上限的 unpack

```
#define LUAI_MAXCSTACK    8000    /* Max. # of stack slots for a C func (<10K). */
```

FFI buffer 作为 byte[]

内存占用更低，但是周边 API 支持较差

复用的例子：https://github.com/openresty/lua-resty-core/blob/master/lib/resty/core/base.lua get_string_buf

LUAJIT_NUMMODE

x86/64 架构下，当我们在编译时设置 LUAJIT_NUMMODE 为 2 时，LuaJIT 会尽可能把数值类型在内部存储成 32 位 int 的形式。（不影响 for i = 1, #t 这样的字节码）

JIT tracing - live demo

http://wiki.luajit.org/Bytecode-2.0

http://wiki.luajit.org/SSA-IR-2.0

```
local total = #s
local crc = 0xffffffff
for i = 1, total do
        local byte = str_byte(s, i)
        crc = bxor(CRC32[band(bxor(crc, byte), 0xff) + 1], rshift(crc, 8))
end
return bxor(crc, 0xffffffff)
```

```
uint32_t  crc;
crc = 0xffffffff;
while (len--) {
        crc = ngx_crc32_table256[(crc ^ *p++) & 0xff] ^ (crc >> 8);
}
return crc ^ 0xffffffff;
```

纯 Lua 实现性能只有 C (O0) 版本的一半!

改动两行代码

+local crc_buf = ffi.new("unsigned int[256]", CRC32)

-crc = bxor(CRC32[band(bxor(crc, byte), 0xff) + 1], rshift(crc, 8))

+crc = bxor(crc_buf[band(bxor(crc, byte), 0xff)], rshift(crc, 8))

移除边界检查，让纯 Lua 实现的速度 x2.5

虽然还是比 C -O2 版本慢……编解码运算，C 才是王道

Why lua-resty-core is faster?

以 ngx.re.find 为例

```
--require "resty.core"
local ls = ("01234"):rep(100)
for i = 1, 1e7 do
        ngx.re.find(ls, "a", "jo")
end
```

由于 JIT 时可以优化掉 FFI 调用的数据交换过程，所以当一个 API 在数据交换上耗费的比重越多，改写成 FFI 时带来的性能提升越大。

比如 ngx.re.find (数据交换复杂)

比如 ngx.time （C 部分的逻辑简单，大部分耗时在数据交换上）

反之，如果一个 API 耗费在数据交换的比重小，则 FFI 化带来的提升就小，比如 ngx.md5。

FFI 改造还能减少 stitch，这方面的提升需要结合具体上下文分析。