链滴

OpenResty 实战之缓存应用

作者：fc13240

原文链接：https://ld246.com/article/1544090296613

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

原文链接：OpenResty实战之缓存应用

本文给大家介绍一下OpenResty的缓存应用，主要涉及Shared_Dict，LruCache，多级缓存mlcache。

缓存原则

缓存是一个大型系统中非常重要的一个组成部分。一个生产环境的缓存，需要根据自己的业务瓶颈制合理的缓存方案。一般来说缓存有两个原则：

1. 缓存越靠近用户的请求越好：能在用户本地的，就不要再发，能使用cdn的就不要回源服务器。
2. 尽量使用本机和本进程的缓存解决，因为跨机器或者跨机房会造成带宽压力和延迟。

shared_dict

OpenResty自带的缓存机制，是提前创建了一块固定大小的共享内存，所有的worker都可以（加锁使用。内部使用LRU算法，过期的数据将被清除。

完整示例：

```
worker_processes  1;
error_log logs/error.log info;

events {
    worker_connections 512;
}

http {

    log_format myformat '$remote_addr $status $time_local';
    access_log logs/access.log myformat;

    lua_shared_dict mycache 8M;

    server {
        listen 8080;
        charset utf-8;

        location /redis {
            content_by_lua_file lua/redis_exp.lua;
        }

        location /mixed {
            content_by_lua_block {
                function get_cache(key)
                    local cache = ngx.shared.mycache
                    local value = cache:get(key)
                    return value
                end

                function set_cache(key, value, expire)
                    if not expire then
                        expire = 0
```

```
            end

            local cache = ngx.shared.mycache
            local succ, err, forcible = cache:set(key, value, expire)
            return succ
        end

        local sum = get_cache("sum")
        if not sum then
            set_cache("sum", 1)
            sum = 1
        end
        sum = sum + 1
        set_cache("sum", sum)
        ngx.say("sum ", sum)
      }
    }
  }
}
```

Lua LRU cache

这个引入了一个Lua的外部库，在单个worker中各个请求之间进行数据共享，这里尤其是要注意库的用方式，因为不同的调用方式会导致缓存失效。

我们先给出正确的版本：

```
worker_processes  1;
error_log logs/error.log info;

events {
    worker_connections 512;
}

http {

    log_format myformat '$remote_addr $status $time_local';
    access_log logs/access.log myformat;

    lua_package_path "/home/zhoulihai/Desktop/work/lua/?.lua;;";

    server {
        listen 8080;
        charset utf-8;

        location /redis {
            content_by_lua_file lua/redis_exp.lua;
        }

        location /mixed {
            content_by_lua '
                require("lru").go()
            ';
```

```
        }
    }
}
```

这里主要注意两点：lua_package_path "/home/zhoulihai/Desktop/work/lua/?.lua;;";和content_by_lua

第一个参数是指定lua脚本的位置，作用是为第二句话打基础，直接使用content_by_lua_file来调用lu脚本使用的位置变量和content_by_lua脚本的位置变量是不同的。这里需要声明一下位置。

第二个地方应该是这样使用content_by_lua，因为这个缓存不能每次都创建一个新的，利用的是OpenResty的只加载一次lua代码的特性，使创建缓存的代码只运行一次。还有一种方法是另创建一个lua本来调用这句话。

lru.lua文件：

```
local _M = {}

local lrucache = require "resty.lrucache"

local c, err = lrucache.new(200)
if not c then
    return error("error on new lrucache: ", err or "unknow")
end

function _M:go()
    local sum = c:get("sum")
    if not sum then
        c:set("sum", 0)
        sum = 0
    end
    sum = sum + 1
    c:set("sum", sum)
    ngx.say("sum ", sum)
end

return _M
```

这样每访问一次都会增加一个计数。

另一个演示

```
location /mixed {
    content_by_lua_file lua/lru_t.lua;
}
```

lru_t.lua文件：

```
require("lru").go()
```

“LRU cache的目的”

做一个东西一定是有它的目的的，不能为了炫技术而去做一点东西。首先，为什么做缓存，答案是节时间。为了实现一个缓存，你可能会这样做：

```
cache = []

### set

cache[key] = value

### get

return cache[key]
```

但是久而久之，如果key足够多，占用的内存会越来越大。你会想着删除一点key，但是这里遇到问题，删哪些部分呢？

于是一般人都会想，很久不用的key就可以删掉了，最近使用过的key一定是要保留的。这就是lrucach的想法了：key个数可以定义最多数目，超过数目了，删除最老的数据，保留最新的。

“实现原理”

为了实现LRUCache，一定要有一个cache[key]=value这种kv的存储结构，但是怎么**维护顺序**呢？resy采用的是双向链表维护顺序。

队列和节点

lrucache有两个节点队列，一个是free_queue，另一个是cache_queue。set操作为从free_queue中出节点放到cache队列中；get操作为从cache队列中取node。

因为lua和c语言可以借助ffi很方便地互相嵌入，节点的定义是使用c代码定义的：

```
typedef struct lrucache_queue_s  lrucache_queue_t;

struct lrucache_queue_s {

    double             expire;  /* in seconds */

    lrucache_queue_t  *prev;

    lrucache_queue_t  *next;

};
```

一开始初始化queue。

```
local mt = { __index = _M }

function _M.new(size)
```

```
    if size < 1 then
        return nil, "size too small"
    end
    local self = {
        hasht = {},
        free_queue = queue_init(size),
        cache_queue = queue_init(),
        key2node = {},
        node2key = {},
    }
    return setmetatable(self, mt)
end
```

set操作

如果free_queue是有node的，set一个key的步骤如下：

- 从free队列中取一个node
- 把node和key加入到node2key和key2node中

```
node2key[ptr2num(node)] = key
key2node[key] = node
```

- 把node从free队列中删除， queue_remove(node)，删除队列node代码如下：

```
local function queue_remove(x)
    local prev = x.prev
    local next = x.next
    next.prev = prev
    prev.next = next
    -- for debugging purpose only:
```

```
    x.prev = NULL
    x.next = NULL
end
```

- 把node加入到cache队列中， queue_insert_head(self.cache_queue, node)

```
-- (bt) h[0] as a header node without any data, so queue length is size+1
local function queue_insert_head(h, x)
    x.next = h[0].next
    x.next.prev = x
    x.prev = h
    h[0].next = x
end
```

如果free_queue没有空闲的node了，set一个key的步骤如下：

与上面第一步的"从free队列中取一个node"不同，这里取node的方法为从cache队列的最后摘一个ode下来。

```
local free_queue = self.free_queue
local node2key = self.node2key
-- (bt) if free_queue is empty, delete the last node.
if queue_is_empty(free_queue) then
    -- evict the least recently used key
    -- assert(not queue_is_empty(self.cache_queue))
    node = queue_last(self.cache_queue)
    local oldkey = node2key[ptr2num(node)]
    -- print(key, ": evicting oldkey: ", oldkey, ", oldnode: ",
    --         tostring(node))
    if oldkey then
        hasht[oldkey] = nil
        key2node[oldkey] = nil
```

```
    end
else
    -- take a free queue node
    node = queue_head(free_queue)
    -- print(key, ": get a new free node: ", tostring(node))
end
```

这里说下queue_last函数O(1)的实现：

```
local function queue_last(h)
    return h[0].prev
end
```

除了去node不一致之外，其他步骤与free queue有空闲node是一样的。

get操作

和set相比，get的操作比较简单。如果有key存在，则把node移到cache队列的最前面。

```
function _M.get(self, key)
    local hasht = self.hasht
    local val = hasht[key]
    if not val then
        return nil
    end
    local node = self.key2node[key]
    -- print(key, ": moving node ", tostring(node), " to cache queue head")
    -- (bt) remove it from queue and insert to head later.
    local cache_queue = self.cache_queue
    queue_remove(node)
    queue_insert_head(cache_queue, node)
```

```
    if node.expire >= 0 and node.expire < ngx_now() then
        -- (bt) expired
        -- print("expired: ", node.expire, " > ", ngx_now())
        return nil, val
    end
    return val
end
```

delete操作

如果删除某一个key后，需要回收cache_queue中的node到free_queue中。

```
function _M.delete(self, key)
    self.hasht[key] = nil
    local key2node = self.key2node
    local node = key2node[key]
    if not node then
        return false
    end
    key2node[key] = nil
    self.node2key[ptr2num(node)] = nil
    -- (bt) 把node回收到free_queue中
    queue_remove(node)
    queue_insert_tail(self.free_queue, node)
    return true
end
```

队列操作

lru-cache中使用链表操作，remove, insert_head等，耗时都是O(1)。

```
-- (bt) queue is a double-pointer-list.
local function queue_init(size)
    if not size then
        size = 0
    end
    local q = ffi_new(queue_arr_type, size + 1)
    ffi_fill(q, ffi_sizeof(queue_type, size + 1), 0)
    if size == 0 then
        q[0].prev = q
        q[0].next = q
    else
        local prev = q[0]
        for i = 1, size do
          local e = q[i]
          prev.next = e
          e.prev = prev
          prev = e
        end
        -- (bt) it is a loop
        local last = q[size]
        last.next = q
        q[0].prev = last
    end
    return q
end
-- (bt) if queue is empty, header is tail.
local function queue_is_empty(q)
```

```
    -- print("q: ", tostring(q), "q.prev: ", tostring(q), ": ", q == q.prev)
    return q == q[0].prev
end
local function queue_remove(x)
    local prev = x.prev
    local next = x.next
    next.prev = prev
    prev.next = next
    -- for debugging purpose only:
    x.prev = NULL
    x.next = NULL
end
-- (bt) h[0] as a header node without any data, so queue length is size+1
local function queue_insert_head(h, x)
    x.next = h[0].next
    x.next.prev = x
    x.prev = h
    h[0].next = x
end
local function queue_last(h)
    return h[0].prev
end
local function queue_head(h)
    return h[0].next
end
```

新的多级缓存库lua-resty-mlcache

lua-resty-mlcache用于 OpenResty 的快速自动分层缓存。

这个库可以作为键 / 值存储缓存的标量 Lua 类型和表来操作，结合了 Lua 共享的 dict API 和 Lua-resty-lucache 的威力，从而得到一个非常高性能和灵活的缓存解决方案。

特点:

使用 TTLs 进行缓存和负缓存。

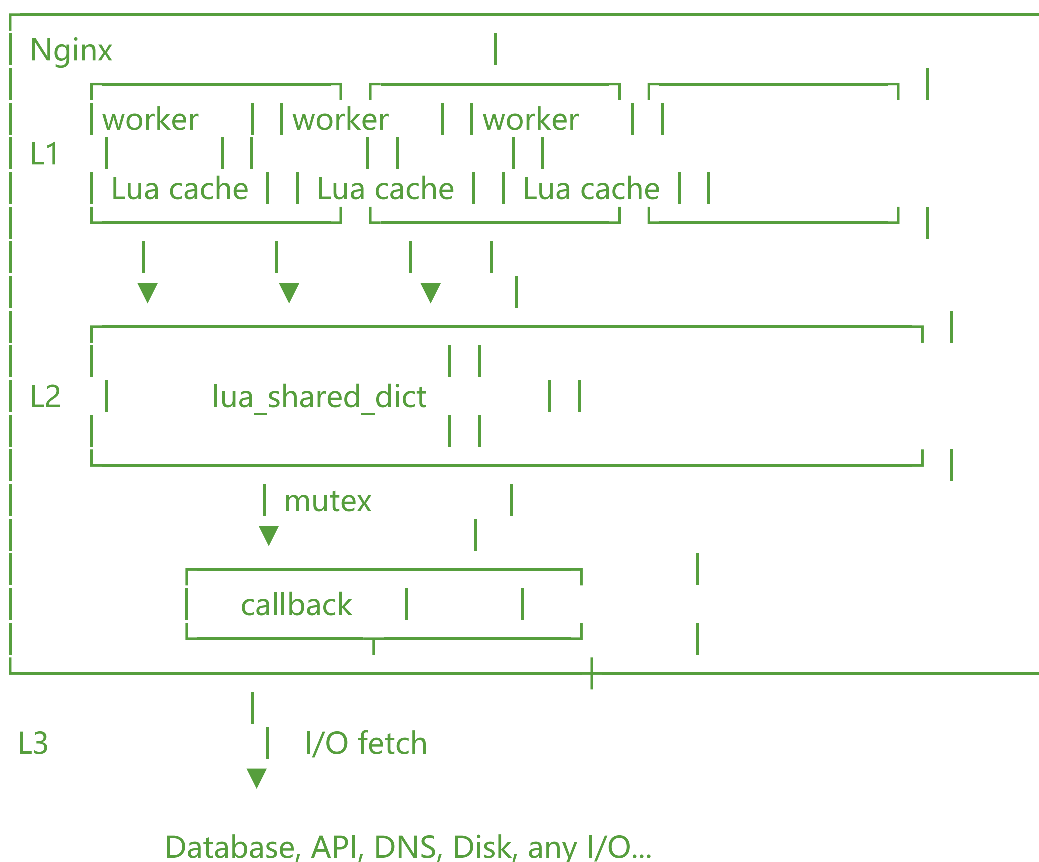
通过 lua-resty-lock 内置 mutex，可以防止数据库 / 后端在缓存未命中时对数据库 / 后端的狗堆效
。

内置的工作人员之间的通信，以宣传高速缓存无效，并允许工作人员更新他们的 L1(lua-resty-lucache 缓存的更改(set ()， delete ())。

支持分离命中和未命中的缓存队列。

可以创建多个独立的实例来保存各种类型的数据，同时依赖于相同的共享型 lua L2高速缓存。

这个库中各种缓存级别的说明:



缓存级别的层次结构是:

L1: 最近最少使用的 Lua VM 缓存，使用了 Lua-resty-lucache。 如果填充，则提供最快的查找，并避免耗尽工作人员的 Lua VM 内存。

L2: 所有工作者共享的共享存储区。 只有当 L1未命中时才能访问这个级别，并防止 worker 请求 L3 存。

L3: 一个自定义函数，它只能由一个工作者来运行，以避免数据库 / 后端上的狗堆效应(通过 lua-resty-

ock)。 通过 l 3获取的值将被设置为 l 2缓存，以便其他工作者进行检索。

参考：

OpenResty缓存

resty_lrucache解读