跟我学Nginx+Lua开发

极客学院出版

前言

天下武功，为快不破。Nginx 的看家本领就是速度，Lua 的拿手好戏亦是速度，这两者的结合在速度上无疑有基因上的优势。最先将 Nginx，Lua 组合到一起的是 OpenResty，它有一个 ngx_lua 模块，将 Lua 嵌入到了 Nginx 里面。本教程从环境搭建到实战讲解，逐步向读者展示如何使用 Nginx+Lua 框架进行开发。

适用人群

开发高速高并发网站的程序员。

学习前提

学习本教程前，你需要了解 Java、Lua 等编程语言。

鸣谢：http://jinnianshilongnian.iteye.com/category/333854

目录

1

安装 Nginx+Lua 开发环境

首先我们选择使用 OpenResty，其是由 Nginx 核心加很多第三方模块组成，其最大的亮点是默认集成了 Lua 开发环境，使得 Nginx 可以作为一个 Web Server 使用。借助于 Nginx 的事件驱动模型和非阻塞 IO，可以实现高性能的 Web 应用程序。而且 OpenResty 提供了大量组件如 Mysql、Redis、Memcached 等等，使在 Nginx 上开发Web 应用更方便更简单。目前在京东如实时价格、秒杀、动态服务、单品页、列表页等都在使用Nginx+Lua 架构，其他公司如淘宝、去哪儿网等。

安装环境

安装步骤可以参考 http://openresty.org/#Installation。

创建目录 /usr/servers，以后我们把所有软件安装在此目录

Java 代码 收藏代码

```
mkdir -p /usr/servers
cd /usr/servers/
```

安装依赖（我的环境是 ubuntu，可以使用如下命令安装，其他的可以参考 openresty 安装步骤）

Java 代码 收藏代码

```
apt-get install libreadline-dev libncurses5-dev libpcre3-dev libssl-dev perl
```

下载 ngx_openresty-1.7.7.2.tar.gz 并解压

Java 代码 收藏代码

```
wget http://openresty.org/download/ngx_openresty-1.7.7.2.tar.gz
tar -xzvf ngx_openresty-1.7.7.2.tar.gz
```

ngx_openresty-1.7.7.2/bundle 目录里存放着 nginx 核心和很多第三方模块，比如有我们需要的 Lua 和 LuaJIT。

安装 LuaJIT

Java 代码 收藏代码

```
cd bundle/LuaJIT-2.1-20150120/
make clean && make && make install
ln -sf luajit-2.1.0-alpha /usr/local/bin/luajit
```

下载 ngx_cache_purge 模块，该模块用于清理 nginx 缓存

Java 代码 收藏代码

```
cd /usr/servers/ngx_openresty-1.7.7.2/bundle
wget https://github.com/FRiCKLE/ngx_cache_purge/archive/2.3.tar.gz
tar -xvf 2.3.tar.gz
```

下载 nginx_upstream_check_module 模块，该模块用于 ustream 健康检查

Java 代码 收藏代码

```
cd /usr/servers/ngx_openresty-1.7.7.2/bundle
wget https://github.com/yaoweibin/nginx_upstream_check_module/archive/v0.3.0.tar.gz
tar -xvf v0.3.0.tar.gz
```

安装 ngx_openresty

Java 代码 收藏代码

```
cd /usr/servers/ngx_openresty-1.7.7.2
./configure --prefix=/usr/servers --with-http_realip_module  --with-pcre  --with-luajit --add-module=./bundle/ngx_ca
make && make install
```

--with*** 安装一些内置/集成的模块

--with-http_realip_module 取用户真实 ip 模块

-with-pcre Perl 兼容的达式模块

--with-luajit 集成 luajit 模块

--add-module 添加自定义的第三方模块，如此次的 ngx_che_purge

到 /usr/servers 目录下

Java 代码 收藏代码

```
cd /usr/servers/
ll
```

会发现多出来了如下目录，说明安装成功

/usr/servers/luajit：luajit 环境，luajit 类似于 java 的 jit，即即时编译，lua 是一种解释语言，通过 luajit 可以即时编译 lua 代码到机器代码，得到很好的性能；

/usr/servers/lualib：要使用的 lua 库，里边提供了一些默认的 lua 库，如 redis，json 库等，也可以把一些自己开发的或第三方的放在这；

/usr/servers/nginx：安装的 nginx；

通过 /usr/servers/nginx/sbin/nginx -V 查看 nginx 版本和安装的模块

启动 nginx

/usr/servers/nginx/sbin/nginx

接下来该配置 nginx+lua 开发环境了

配置环境

配置及 Nginx HttpLuaModule 文档在可以查看 http://openresty.org/#Installation。

编辑 nginx.conf 配置文件

Java 代码 收藏代码

```
vim /usr/servers/nginx/conf/nginx.conf
```

在 http 部分添加如下配置

Java 代码 收藏代码

```
\#lua模块路径，多个之间”;”分隔，其中”;;”表示默认搜索路径，默认到/usr/servers/nginx下找
lua_package_path "/usr/servers/lualib/?.lua;;"; #lua 模块
lua_package_cpath "/usr/servers/lualib/?.so;;"; #c模块
```

为了方便开发我们在 /usr/servers/nginx/conf 目录下创建一个 lua.conf

Java 代码 收藏代码

```
\#lua.conf
server {
    listen       80;
    server_name  _;
}
```

在 nginx.conf 中的 http 部分添加 include lua.conf 包含此文件片段

Java 代码 收藏代码

```
include lua.conf;
```

测试是否正常

Java 代码 收藏代码

/usr/servers/nginx/sbin/nginx -t

如果显示如下内容说明配置成功

- nginx: the configuration file /usr/servers/nginx/conf/nginx.conf syntax is ok
- nginx: configuration file /usr/servers/nginx/conf/nginx.conf test is successful

HelloWorld

在 lua.conf 中 server 部分添加如下配置

Java 代码 收藏代码

```
location /lua {
    default_type 'text/html';
        content_by_lua 'ngx.say("hello world")';
}
```

测试配置是否正确

Java 代码 收藏代码

```
/usr/servers/nginx/sbin/nginx -t
```

重启 nginx

Java 代码 收藏代码

```
/usr/servers/nginx/sbin/nginx -s reload
```

访问如 http://192.168.1.6/lua（自己的机器根据实际情况换 ip），可以看到如下内容

```
hello world
```

lua 代码文件

我们把 lua 代码放在 nginx 配置中会随着 lua 的代码的增加导致配置文件太长不好维护，因此我们应该把 lua 代码移到外部文件中存储。

Java 代码 收藏代码

```
vim /usr/servers/nginx/conf/lua/test.lua
```

Java 代码 收藏代码

```
\#添加如下内容
ngx.say("hello world");
```

然后 lua.conf 修改为

Java 代码 收藏代码

```
location /lua {
    default_type 'text/html';
    content_by_lua_file conf/lua/test.lua; #相对于nginx安装目录
}
```

此处 conf/lua/test.lua 也可以使用绝对路径 /usr/servers/nginx/conf/lua/test.lua。

lua_code_cache

默认情况下 lua_code_cache 是开启的，即缓存 lua 代码，即每次 lua 代码变更必须reload nginx 才生效，如果在开发阶段可以通过 lua_code_cache off;关闭缓存，这样调试时每次修改 lua 代码不需要 reload nginx；但是正式环境一定记得开启缓存。

Java 代码 收藏代码

```
    location /lua {
        default_type 'text/html';
        lua_code_cache off;
        content_by_lua_file conf/lua/test.lua;
}
```

开启后 reload nginx 会看到如下报警

nginx: [alert] lua_code_cache is off; this will hurt performance in /usr/servers/nginx/conf/lua.conf:8

错误日志

如果运行过程中出现错误，请不要忘记查看错误日志。

Java 代码 收藏代码

```
tail -f /usr/servers/nginx/logs/error.log
```

到此我们的基本环境搭建完毕。

nginx+lua 项目构建

以后我们的 nginx lua 开发文件会越来越多，我们应该把其项目化，已方便开发。项目目录结构如下所示：

example

```
example.conf    ---该项目的nginx 配置文件
lua             ---我们自己的lua代码
  test.lua
lualib          ---lua依赖库/第三方依赖
  *.lua
  *.so
```

其中我们把 lualib 也放到项目中的好处就是以后部署的时候可以一起部署，防止有的服务器忘记复制依赖而造成缺少依赖的情况。

我们将项目放到到 /usr/example 目录下。

/usr/servers/nginx/conf/nginx.conf 配置文件如下(此处我们最小化了配置文件)

Java 代码 收藏代码

```
\#user  nobody;
worker_processes  2;
error_log  logs/error.log;
events {
    worker_connections  1024;
}
http {
    include       mime.types;
    default_type  text/html;

    #lua模块路径，其中”;;”表示默认搜索路径，默认到/usr/servers/nginx下找
    lua_package_path "/usr/example/lualib/?.lua;;";  #lua 模块
    lua_package_cpath "/usr/example/lualib/?.so;;";  #c模块
    include /usr/example/example.conf;
}
```

通过绝对路径包含我们的 lua 依赖库和 nginx 项目配置文件。

/usr/example/example.conf 配置文件如下

Java 代码 收藏代码

```
server {
    listen       80;
    server_name  _;

    location /lua {
        default_type 'text/html';
        lua_code_cache off;
        content_by_lua_file /usr/example/lua/test.lua;
    }
}
```

lua 文件我们使用绝对路径 /usr/example/lua/test.lua。

到此我们就可以把 example 扔 svn 上了。

2

Nginx+Lua 开发入门

Nginx 入门

本文目的是学习 Nginx+Lua 开发，对于 Nginx 基本知识可以参考如下文章：

nginx 启动、关闭、重启

http://www.cnblogs.com/derekchen/archive/2011/02/17/1957209.html

agentzh 的 Nginx 教程

http://openresty.org/download/agentzh-nginx-tutorials-zhcn.html

Nginx+Lua 入门

http://17173ops.com/2013/11/01/17173-ngx-lua-manual.shtml

nginx 配置指令的执行顺序

http://zhongfox.github.io/blog/server/2013/05/15/nginx-exec-order/

nginx 与 lua 的执行顺序和步骤说明

http://www.mrhaoting.com/?p=157

Nginx 配置文件 nginx.conf 中文详解

http://www.ha97.com/5194.html

Tengine 的 Nginx 开发从入门到精通

http://tengine.taobao.org/book/

官方文档

http://wiki.nginx.org/Configuration

第 2 章 Lua 入门

本文目的是学习 Nginx+Lua 开发，对于 Lua 基本知识可以参考如下文章：

Lua 简明教程

http://coolshell.cn/articles/10739.html

lua 在线 lua 学习教程

http://book.luaer.cn/

Lua 5.1 参考手册

http://www.codingnow.com/2000/download/lua_manual.html

Lua 5.3 参考手册

http://cloudwu.github.io/lua53doc/

Nginx Lua API

和一般的 Web Server 类似，我们需要接收请求、处理并输出响应。而对于请求我们需要获取如请求参数、请求头、Body 体等信息；而对于处理就是调用相应的 Lua 代码即可；输出响应需要进行响应状态码、响应头和响应内容体的输出。因此我们从如上几个点出发即可。

接收请求

example.conf 配置文件

Java 代码 收藏代码

```
location ~ /lua_request/(\d+)/(\d+) {
    #设置nginx变量
    set $a $1;
    set $b $host;
    default_type "text/html";
    #nginx内容处理
    content_by_lua_file /usr/example/lua/test_request.lua;
    #内容体处理完成后调用
    echo_after_body "ngx.var.b $b";
}
```

test_request.lua

Java 代码 收藏代码

```
--nginx变量
local var = ngx.var
ngx.say("ngx.var.a : ", var.a, "<br/>")
ngx.say("ngx.var.b : ", var.b, "<br/>")
ngx.say("ngx.var[2] : ", var[2], "<br/>")
ngx.var.b = 2;

ngx.say("<br/>")

--请求头
local headers = ngx.req.get_headers()
ngx.say("headers begin", "<br/>")
ngx.say("Host : ", headers["Host"], "<br/>")
```

```
ngx.say("user-agent : ", headers["user-agent"], "<br/>")
ngx.say("user-agent : ", headers.user_agent, "<br/>")
for k,v in pairs(headers) do
    if type(v) == "table" then
        ngx.say(k, " : ", table.concat(v, ","), "<br/>")
    else
        ngx.say(k, " : ", v, "<br/>")
    end
end
ngx.say("headers end", "<br/>")
ngx.say("<br/>")

--get请求uri参数
ngx.say("uri args begin", "<br/>")
local uri_args = ngx.req.get_uri_args()
for k, v in pairs(uri_args) do
    if type(v) == "table" then
        ngx.say(k, " : ", table.concat(v, ", "), "<br/>")
    else
        ngx.say(k, ": ", v, "<br/>")
    end
end
ngx.say("uri args end", "<br/>")
ngx.say("<br/>")

--post请求参数
ngx.req.read_body()
ngx.say("post args begin", "<br/>")
local post_args = ngx.req.get_post_args()
for k, v in pairs(post_args) do
    if type(v) == "table" then
        ngx.say(k, " : ", table.concat(v, ", "), "<br/>")
    else
        ngx.say(k, ": ", v, "<br/>")
    end
end
ngx.say("post args end", "<br/>")
ngx.say("<br/>")
--请求的http协议版本
ngx.say("ngx.req.http_version : ", ngx.req.http_version(), "<br/>")
--请求方法
ngx.say("ngx.req.get_method : ", ngx.req.get_method(), "<br/>")
--原始的请求头内容
ngx.say("ngx.req.raw_header : ",  ngx.req.raw_header(), "<br/>")
--请求的body内容体
```

```
ngx.say("ngx.req.get_body_data() : ", ngx.req.get_body_data(), "<br/>")
ngx.say("<br/>")
```

ngx.var： nginx 变量，如果要赋值如 ngx.var.b = 2，此变量必须提前声明；另外对于 nginx location 中使用正则捕获的捕获组可以使用 ngx.var [捕获组数字]获取；

ngx.req.get_headers：获取请求头，默认只获取前100，如果想要获取所以可以调用ngx.req.get_headers(0)；获取带中划线的请求头时请使用如 headers.user_agent 这种方式；如果一个请求头有多个值，则返回的是 table；

ngx.req.get_uri_args：获取 url 请求参数，其用法和 get_headers 类似；

ngx.req.get_post_args：获取 post 请求内容体，其用法和 get_headers 类似，但是必须提前调用 ngx.req.read_body() 来读取 body 体（也可以选择在 nginx 配置文件使用lua_need_request_body on;开启读取 body 体，但是官方不推荐）；

ngx.req.raw_header：未解析的请求头字符串；

ngx.req.get_body_data：为解析的请求 body 体内容字符串。

如上方法处理一般的请求基本够用了。另外在读取 post 内容体时根据实际情况设置 client_body_buffer_size 和 client_max_body_size 来保证内容在内存而不是在文件中。

使用如下脚本测试

Java **代码** 收藏代码

```
wget --post-data 'a=1&b=2' 'http://127.0.0.1/lua_request/1/2?a=3&b=4' -O -
```

输出响应

example.conf 配置文件

Java **代码** 收藏代码

```
location /lua_response_1 {
    default_type "text/html";
    content_by_lua_file /usr/example/lua/test_response_1.lua;
}
```

test_response_1.lua

Java 代码 收藏代码

```
--写响应头
ngx.header.a = "1"
--多个响应头可以使用table
ngx.header.b = {"2", "3"}
--输出响应
ngx.say("a", "b", "<br/>")
ngx.print("c", "d", "<br/>")
--200状态码退出
return ngx.exit(200)
ngx.header：输出响应头；
ngx.print：输出响应内容体；
ngx.say：通ngx.print，但是会最后输出一个换行符；
ngx.exit：指定状态码退出。
```

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_response_2 {
    default_type "text/html";
    content_by_lua_file /usr/example/lua/test_response_2.lua;
}
```

test_response_2.lua

Java 代码 收藏代码

```
ngx.redirect("http://jd.com", 302)
```

ngx.redirect：重定向；

ngx.status= 状态码，设置响应的状态码；ngx.resp.get_headers() 获取设置的响应状态码；ngx.send_headers() 发送响应状态码，当调用 ngx.say/ngx.print 时自动发送响应状态码；可以通过 ngx.headers_sent=true 判断是否发送了响应状态码。

其他 API

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_other {
    default_type "text/html";
    content_by_lua_file /usr/example/lua/test_other.lua;
}
```

test_other.lua

Java 代码 收藏代码

```
--未经解码的请求uri
local request_uri = ngx.var.request_uri;
ngx.say("request_uri : ", request_uri, "<br/>");
--解码
ngx.say("decode request_uri : ", ngx.unescape_uri(request_uri), "<br/>");
--MD5
ngx.say("ngx.md5 : ", ngx.md5("123"), "<br/>")
--http time
ngx.say("ngx.http_time : ", ngx.http_time(ngx.time()), "<br/>")
```

- ngx.escape_uri/ngx.unescape_uri： uri 编码解码；
- ngx.encode_args/ngx.decode_args：参数编码解码；
- ngx.encode_base64/ngx.decode_base64：BASE64 编码解码；
- ngx.re.match：nginx 正则表达式匹配；

更多 Nginx Lua API 请参考 http://wiki.nginx.org/HttpLuaModule#Nginx_API_for_Lua。

Nginx 全局内存

使用过如 Java 的朋友可能知道如 Ehcache 等这种进程内本地缓存，Nginx 是一个 Master 进程多个 Worker 进程的工作方式，因此我们可能需要在多个 Worker 进程中共享数据，那么此时就可以使用 ngx.shared.DICT 来实现全局内存共享。

首先在 nginx.conf 的 http 部分分配内存大小

Java 代码 收藏代码

```
\#共享全局变量，在所有worker间共享
lua_shared_dict shared_data 1m;
```

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_shared_dict {
    default_type "text/html";
    content_by_lua_file /usr/example/lua/test_lua_shared_dict.lua;
}
```

test_lua_shared_dict.lua

Java 代码 收藏代码

```
--1、获取全局共享内存变量
local shared_data = ngx.shared.shared_data

--2、获取字典值
local i = shared_data:get("i")
if not i then
    i = 1
    --3、惰性赋值
    shared_data:set("i", i)
    ngx.say("lazy set i ", i, "<br/>")
end
--递增
i = shared_data:incr("i", 1)
ngx.say("i=", i, "<br/>")
```

更多 API 请参考 http://wiki.nginx.org/HttpLuaModule#ngx.shared.DICT。

到此基本的 Nginx Lua API 就学完了，对于请求处理和输出响应如上介绍的 API 完全够用了，更多 API 请参考官方文档。

Nginx Lua 模块指令

Nginx 共11个处理阶段，而相应的处理阶段是可以做插入式处理，即可插拔式架构；另外指令可以在 http、server、server if、location、location if 几个范围进行配置：

指令	所处处理阶段	使用范围	解释
init_by_lua init_by_lua_file	loading-config	http	nginx Master进程加载配置时执行； 通常用于初始化全局配置/预加载Lua模块
init_worker_by_lua init_worker_by_lua_file	starting-worker	http	每个Nginx Worker进程启动时调用的计时器，如果Master进程不允许则只会在init_by_lua之后调用； 通常用于定时拉取配置/数据，或者后端服务的健康检查
set_by_lua set_by_lua_file	rewrite	server,server if,location,location if	设置nginx变量，可以实现复杂的赋值逻辑；此处是阻塞的，Lua代码要做到非常快；
rewrite_by_lua rewrite_by_lua_file	rewrite tail	http,server,location,location if	rrewrite阶段处理，可以实现复杂的转发/重定向逻辑；
access_by_lua	access tail	http,server,location,location if	请求访问阶段处理，用于访问控制

access_by_lua_file			
content_by_lua content_by_lua_file	content	location，location if	内容处理器，接收请求处理并输出响应
header_filter_by_lua header_filter_by_lua_file	output-header-filter	http，server，location，location if	设置header和cookie
body_filter_by_lua body_filter_by_lua_file	output-body-filter	http，server，location，location if	对响应数据进行过滤，比如截断、替换。
log_by_lua log_by_lua_file	log	http，server，location，location if	log阶段处理，比如记录访问量/统计平均响应时间

更详细的解释请参考 http://wiki.nginx.org/HttpLuaModule#Directives。如上指令很多并不常用，因此我们只拿其中的一部分做演示。

init_by_lua

每次 Nginx 重新加载配置时执行，可以用它来完成一些耗时模块的加载，或者初始化一些全局配置；在 Master 进程创建 Worker 进程时，此指令中加载的全局变量会进行 Copy-OnWrite，即会复制到所有全局变量到 Worker 进程。

nginx.conf 配置文件中的 http 部分添加如下代码

Java 代码 收藏代码

```
\#共享全局变量，在所有worker间共享
lua_shared_dict shared_data 1m;

init_by_lua_file /usr/example/lua/init.lua;
```

init.lua

Java 代码 收藏代码

```
--初始化耗时的模块
local redis = require 'resty.redis'
local cjson = require 'cjson'

--全局变量，不推荐
count = 1

--共享全局内存
local shared_data = ngx.shared.shared_data
shared_data:set("count", 1)
```

test.lua

Java 代码 收藏代码

```
count = count + 1
ngx.say("global variable : ", count)
local shared_data = ngx.shared.shared_data
ngx.say(", shared memory : ", shared_data:get("count"))
shared_data:incr("count", 1)
ngx.say("hello world")
```

访问如 http://192.168.1.2/lua 会发现全局变量一直不变，而共享内存一直递增

global variable : 2 , shared memory : 8 hello world

另外注意一定在生产环境开启 lua_code_cache，否则每个请求都会创建 Lua VM 实例。

init_worker_by_lua

用于启动一些定时任务，比如心跳检查，定时拉取服务器配置等等；此处的任务是跟 Worker 进程数量有关系的，比如有2个 Worker 进程那么就会启动两个完全一样的定时任务。

nginx.conf 配置文件中的 http 部分添加如下代码

Java **代码** 收藏代码

```
init_worker_by_lua_file /usr/example/lua/init_worker.lua;
```

init_worker.lua

Java **代码** 收藏代码

```
local count = 0
local delayInSeconds = 3
local heartbeatCheck = nil

heartbeatCheck = function(args)
  count = count + 1
  ngx.log(ngx.ERR, "do check ", count)

  local ok, err = ngx.timer.at(delayInSeconds, heartbeatCheck)

  if not ok then
    ngx.log(ngx.ERR, "failed to startup heartbeart worker...", err)
  end
end

heartbeatCheck()
```

ngx.timer.at：延时调用相应的回调方法；ngx.timer.at(秒单位延时，回调函数，回调函数的参数列表)；可以将延时设置为0即得到一个立即执行的任务，任务不会在当前请求中执行不会阻塞当前请求，而是在一个轻量级线程中执行。

另外根据实际情况设置如下指令

- lua_max_pending_timers 1024; #最大等待任务数
- lua_max_running_timers 256; #最大同时运行任务数

set_by_lua

设置 nginx 变量，我们用的 set 指令即使配合 if 指令也很难实现负责的赋值逻辑；

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_set_1 {
    default_type "text/html";
    set_by_lua_file $num /usr/example/lua/test_set_1.lua;
    echo $num;
}
```

set_by_lua_file：语法 set_by_lua_file $var lua_file arg1 arg2...; 在 lua代码中可以实现所有复杂的逻辑，但是要执行速度很快，不要阻塞；

test_set_1.lua

Java 代码 收藏代码

```
local uri_args = ngx.req.get_uri_args()
local i = uri_args["i"] or 0
local j = uri_args["j"] or 0

return i + j
```

得到请求参数进行相加然后返回。

访问如 http://192.168.1.2/lua_set_1?i=1&j=10 进行测试。 如果我们用纯 set 指令是无法实现的。

再举个实际例子，我们实际工作时经常涉及到网站改版，有时候需要新老并存，或者切一部分流量到新版

首先在 example.conf 中使用 map 指令来映射 host 到指定 nginx 变量，方便我们测试

Java 代码 收藏代码

```
 ############ 测试时使用的动态请求

map $host $item_dynamic {
    default                    "0";
    item2014.jd.com            "1";
}
```

如绑定 hosts

- 192.168.1.2 item.jd.com ;
- 192.168.1.2 item2014.jd.com ;

此时我们想访问 item2014.jd.com 时访问新版，那么我们可以简单的使用如

Java 代码 收藏代码

```
if ($item_dynamic = "1") {
   proxy_pass http://new;
}
proxy_pass http://old;
```

但是我们想把商品编号为 8 位(比如品类为图书的)没有改版完成，需要按照相应规则跳转到老版，但是其他的到新版；虽然使用 if 指令能实现，但是比较麻烦，基本需要这样

Java 代码 收藏代码

```
set jump "0";
if($item_dynamic = "1") {
    set $jump "1";
}
if(uri ~ "^/6[0-9]{7}.html") {
   set $jump "${jump}2";
}
\#非强制访问新版，且访问指定范围的商品
if (jump == "02") {
   proxy_pass http://old;
}
proxy_pass http://new;
```

以上规则还是比较简单的，如果涉及到更复杂的多重 if/else 或嵌套 if/else 实现起来就更痛苦了，可能需要到后端去做了；此时我们就可以借助 lua 了：

Java 代码 收藏代码

```
set_by_lua $to_book '
    local ngx_match = ngx.re.match
    local var = ngx.var
    local skuId = var.skuId
    local r = var.item_dynamic ~= "1" and ngx.re.match(skuId, "^[0-9]{8}$")
    if r then return "1" else return "0" end;
';
set_by_lua $to_mvd '
    local ngx_match = ngx.re.match
    local var = ngx.var
    local skuId = var.skuId
    local r = var.item_dynamic ~= "1" and ngx.re.match(skuId, "^[0-9]{9}$")
    if r then return "1" else return "0" end;
';
\#自营图书
if ($to_book) {
   proxy_pass http://127.0.0.1/old_book/$skuId.html;
}
\#自营音像
if ($to_mvd) {
   proxy_pass http://127.0.0.1/old_mvd/$skuId.html;
}
\#默认
proxy_pass http://127.0.0.1/proxy/$skuId.html;
```

rewrite_by_lua

执行内部 URL 重写或者外部重定向，典型的如伪静态化的 URL 重写。其默认执行在 rewrite 处理阶段的最后。

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_rewrite_1 {
   default_type "text/html";
   rewrite_by_lua_file /usr/example/lua/test_rewrite_1.lua;
```

```
    echo "no rewrite";
}
```

test_rewrite_1.lua

Java 代码 收藏代码

```
if ngx.req.get_uri_args()["jump"] == "1" then
  return ngx.redirect("http://www.jd.com?jump=1", 302)
end
```

当我们请求 `http://192.168.1.2/lua_rewrite_1` 时发现没有跳转，而请求 `http://192.168.1.2/lua_rewrite_1?jump=1` 时发现跳转到京东首页了。 此处需要301/302跳转根据自己需求定义。

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_rewrite_2 {
    default_type "text/html";
    rewrite_by_lua_file /usr/example/lua/test_rewrite_2.lua;
    echo "rewrite2 uri : $uri, a : $arg_a";
}
```

test_rewrite_2.lua

Java 代码 收藏代码

```
if ngx.req.get_uri_args()["jump"] == "1" then
  ngx.req.set_uri("/lua_rewrite_3", false);
  ngx.req.set_uri("/lua_rewrite_4", false);
  ngx.req.set_uri_args({a = 1, b = 2});
end
```

ngx.req.set_uri(uri, false)：可以内部重写 uri（可以带参数），等价于 rewrite ^ /lua_rewrite_3；通过配合 if/else 可以实现 rewrite ^ /lua_rewrite_3 break；这种功能；此处两者都是 location 内部 url 重写，不会重新发起新的 location 匹配；

ngx.req.set_uri_args：重写请求参数，可以是字符串(a=1&b=2)也可以是 table；

访问如 `http://192.168.1.2/lua_rewrite_2?jump=0` 时得到响应 rewrite2 uri : /lua_rewrite_2, a :

访问如 http://192.168.1.2/lua_rewrite_2?jump=1 时得到响应 rewrite2 uri : /lua_rewrite_4, a : 1

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_rewrite_3 {
    default_type "text/html";
    rewrite_by_lua_file /usr/example/lua/test_rewrite_3.lua;
    echo "rewrite3 uri : $uri";
}
```

c test_rewrite_3.lua

Java 代码 收藏代码

```
if ngx.req.get_uri_args()["jump"] == "1" then
   ngx.req.set_uri("/lua_rewrite_4", true);
   ngx.log(ngx.ERR, "==========")
   ngx.req.set_uri_args({a = 1, b = 2});
end
```

ngx.req.set_uri(uri, true)：可以内部重写 uri，即会发起新的匹配 location 请求，等价于 rewrite ^ /lua_rewrite_4 last；此处看 error log 是看不到我们记录的log。

所以请求如 http://192.168.1.2/lua_rewrite_3?jump=1 会到新的 location 中得到响应，此处没有 /lua_rewrite_4，所以匹配到 /lua 请求，得到类似如下的响应 global variable : 2 , shared memory : 1 hello world

即

```
rewrite ^ /lua_rewrite_3;              等价于  ngx.req.set_uri("/lua_rewrite_3", false);
rewrite ^ /lua_rewrite_3 break;      等价于  ngx.req.set_uri("/lua_rewrite_3", false); 加 if/else判断/break/return
rewrite ^ /lua_rewrite_4 last;         等价于  ngx.req.set_uri("/lua_rewrite_4", true);
```

注意，在使用 rewrite_by_lua 时，开启 rewrite_log on;后也看不到相应的 rewrite log。

access_by_lua

用于访问控制，比如我们只允许内网 ip 访问，可以使用如下形式

Java 代码 收藏代码

```
allow    127.0.0.1;
allow    10.0.0.0/8;
allow    192.168.0.0/16;
allow    172.16.0.0/12;
deny     all;
```

example.conf 配置文件

Java 代码 收藏代码

```
location /lua_access {
    default_type "text/html";
    access_by_lua_file /usr/example/lua/test_access.lua;
    echo "access";
}
```

test_access.lua

Java 代码 收藏代码

```
if ngx.req.get_uri_args()["token"] ~= "123" then
  return ngx.exit(403)
end
```

即如果访问如 `http://192.168.1.2/lua_access?token=234` 将得到 403 Forbidden 的响应。这样我们可以根据如 cookie/ 用户 token 来决定是否有访问权限。

content_by_lua

此指令之前已经用过了，此处就不讲解了。

另外在使用 PCRE 进行正则匹配时需要注意正则的写法，具体规则请参考 http://wiki.nginx.org/HttpLuaModule中的Special PCRE Sequences部 分。还有其他的注意事项也请阅读官方文档。

3

Redis/SSDB+Twemproxy 安装与使用

目前对于互联网公司不使用 Redis 的很少，Redis 不仅仅可以作为 key-value 缓存，而且提供了丰富的数据结果如 set、list、map 等，可以实现很多复杂的功能；但是 Redis 本身主要用作内存缓存，不适合做持久化存储，因此目前有如 SSDB、ARDB 等，还有如京东的 JIMDB，它们都支持 Redis 协议，可以支持 Redis 客户端直接访问；而这些持久化存储大多数使用了如LevelDB、RocksDB、LMDB 持久化引擎来实现数据的持久化存储；京东的 JIMDB 主要分为两个版本：LevelDB 和 LMDB，而我们看到的京东商品详情页就是使用 LMDB 引擎作为存储的，可以实现海量KV存储；当然 SSDB 在京东内部也有些部门在使用；另外调研过得如豆瓣的 beansDB 也是很不错的。具体这些持久化引擎之间的区别可以自行查找资料学习。

Twemproxy 是一个 Redis/Memcached 代理中间件，可以实现诸如分片逻辑、HashTag、减少连接数等功能；尤其在有大量应用服务器的场景下 Twemproxy 的角色就凸显了，能有效减少连接数。

Redis 安装与使用

下载 redis 并安装

Java 代码

```
cd /usr/servers/
wget https://github.com/antirez/redis/archive/2.8.19.tar.gz
tar -xvf 2.8.19.tar.gz
cd redis-2.8.19/
make
```

通过如上步骤构建完毕。

后台启动 Redis 服务器

Java 代码

```
nohup /usr/servers/redis-2.8.19/src/redis-server  /usr/servers/redis-2.8.19/redis.conf &
```

查看是否启动成功

Java 代码

```
ps -aux | grep redis
```

进入客户端

Java 代码

```
/usr/servers/redis-2.8.19/src/redis-cli  -p 6379
```

执行如下命令

Java 代码

```
127.0.0.1:6379> set i 1
OK
127.0.0.1:6379> get i
"1"
```

通过如上命令可以看到我们的 Redis 安装成功。更多细节请参考 http://redis.io/。

SSDB 安装与使用

下载 SSDB 并安装

Java 代码

```
\#首先确保安装了g++，如果没有安装，如ubuntu可以使用如下命令安装
apt-get install g++
cd /usr/servers
wget https://github.com/ideawu/ssdb/archive/1.8.0.tar.gz
tar -xvf 1.8.0.tar.gz
make
```

后台启动 SSDB 服务器

Java 代码

```
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/servers/ssdb-1.8.0/ssdb.conf &
```

查看是否启动成功

Java 代码

```
ps -aux | grep ssdb
```

进入客户端

Java 代码

```
/usr/servers/ssdb-1.8.0/tools/ssdb-cli -p 8888
/usr/servers/redis-2.8.19/src/redis-cli  -p 888
```

因为 SSDB 支持 Redis 协议，所以用 Redis 客户端也可以访问

执行如下命令

Java 代码

```
127.0.0.1:8888> set i 1
OK
127.0.0.1:8888> get i
"1"
```

安装过程中遇到错误请参考 http://ssdb.io/docs/zh_cn/install.html；对于 SSDB 的配置请参考官方文档 https://github.com/ideawu/ssdbhttp://ssdb.io/docs/zh_cn/install.html。

Twemproxy 安装与使用

首先需要安装 autoconf、automake、libtool 工具，比如 ubuntu 可以使用如下命令安装

Java 代码

```
apt-get install autoconf automake
apt-get install libtool
```

下载 Twemproxy 并安装

Java 代码

```
cd /usr/servers
wget https://github.com/twitter/twemproxy/archive/v0.4.0.tar.gz
tar -xvf v0.4.0.tar.gz
cd twemproxy-0.4.0/
autoreconf -fvi
./configure && make
```

此处根据要注意，如上安装方式在有些服务器上可能在大量如mset时可能导致 Twemproxy 崩溃，需要使用如 CFLAGS="-O1" ./configure && make 或 CFLAGS="-O3 -fno-strict-aliasing" ./configure && make 安装。

配置

Java 代码

```
vim /usr/servers/twemproxy-0.4.0/conf/nutcracker.yml
```

Java 代码

```
server1:
  listen: 127.0.0.1:1111
  hash: fnv1a_64
  distribution: ketama
  redis: true
  servers:
   - 127.0.0.1:6379:1
```

启动 Twemproxy 代理

Java 代码

```
/usr/servers/twemproxy-0.4.0/src/nutcracker  -d -c /usr/servers/twemproxy-0.4.0/conf/nutcracker.yml
```

-d 指定后台启动 -c 指定配置文件；此处我们指定了代理端口为 1111，其他配置的含义后续介绍。

查看是否启动成功

Java 代码

```
ps -aux | grep nutcracker
```

进入客户端

Java 代码

```
/usr/servers/redis-2.8.19/src/redis-cli  -p 1111
```

执行如下命令

Java 代码

```
127.0.0.1:1111> set i 1
OK
127.0.0.1:1111> get i
"1"
```

Twemproxy 文档请参考 https://github.com/twitter/twemproxy。

到此基本的安装就完成了。接下来做一些介绍。

Redis 设置

基本设置

Java 代码

```
\#端口设置，默认6379
port 6379
\#日志文件，默认/dev/null
logfile ""
```

Redis 内存

Java 代码

```
内存大小对应关系
\# 1k => 1000 bytes
\# 1kb => 1024 bytes
\# 1m => 1000000 bytes
\# 1mb => 1024*1024 bytes
\# 1g => 1000000000 bytes
\# 1gb => 1024*1024*1024 bytes

\#设置Redis占用100mb的大小
maxmemory  100mb

\#如果内存满了就需要按照如相应算法进行删除过期的/最老的
\#volatile-lru 根据LRU算法移除设置了过期的key
\#allkeys-lru  根据LRU算法移除任何key(包含那些未设置过期时间的key)
\#volatile-random/allkeys->random 使用随机算法而不是LRU进行删除
\#volatile-ttl 根据Time-To-Live移除即将过期的key
\#noeviction   永不过期，而是报错
maxmemory-policy volatile-lru

\#Redis并不是真正的LRU/TTL，而是基于采样进行移除的，即如采样10个数据移除其中最老的/即将过期的
maxmemory-samples 10
```

而如 Memcached 是真正的 LRU，此处要根据实际情况设置缓存策略，如缓存用户数据时可能带上了过期时间，此时采用 volatile-lru 即可；而假设我们的数据未设置过期时间，此时可以考虑使用 allkeys-lru/allkeys->random；假设我们的数据不允许从内存删除那就使用noeviction。

内存大小尽量在系统内存的 60%~80% 之间，因为如客户端、主从时复制时都需要缓存区的，这些也是耗费系统内存的。

Redis 本身是单线程的，因此我们可以设置每个实例在 6-8GB 之间，通过启动更多的实例提高吞吐量。如 128 GB 的我们可以开启 `8GB * 10` 个实例，充分利用多核 CPU。

Redis 主从

实际项目时，为了提高吞吐量，我们使用主从策略，即数据写到主 Redis，读的时候从从 Redis上读，这样可以通过挂载更多的从来提高吞吐量。而且可以通过主从机制，在叶子节点开启持久化方式防止数据丢失。

Java 代码

```
\#在配置文件中挂载主从，不推荐这种方式，我们实际应用时Redis可能是会宕机的
slaveof masterIP masterPort
\#从是否只读，默认yes
slave-read-only yes
\#当从失去与主的连接或者复制正在进行时，从是响应客户端（可能返回过期的数据）还是返回“SYNC with master in progress”
slave-serve-stale-data yes
\#从库按照默认10s的周期向主库发送PING测试连通性
repl-ping-slave-period 10
\#设置复制超时时间（SYNC期间批量I/O传输、PING的超时时间），确保此值大于repl-ping-slave-period
\#repl-timeout 60
\#当从断开与主的连接时的复制缓存区，仅当第一个从断开时创建一个，缓存区越大从断开的时间可以持续越长
\# repl-backlog-size 1mb
\#当从与主断开持续多久时清空复制缓存区，此时从就需要全量复制了，如果设置为0将永不清空
\# repl-backlog-ttl 3600
\#slave客户端缓存区，如果缓存区超过256mb将直接断开与从的连接，如果持续60秒超过64mb也会断开与从的连接
client-output-buffer-limit slave 256mb 64mb 60
此处需要根据实际情况设置client-output-buffer-limit slave和 repl-backlog-size；比如如果网络环境不好，从与主经常断开，而每
```

主从示例

Java 代码

```
cd /usr/servers/redis-2.8.19
cp redis.conf redis_6660.conf
cp redis.conf redis_6661.conf
vim redis_6660.conf
vim redis_6661.conf
```

将端口分别改为 port 6660 和 port 6661，然后启动

Java 代码

```
nohup /usr/servers/redis-2.8.19/src/redis-server  /usr/servers/redis-2.8.19/redis_6660.conf &
nohup /usr/servers/redis-2.8.19/src/redis-server  /usr/servers/redis-2.8.19/redis_6661.conf &
```

查看是否启动

Java 代码

```
ps -aux | grep redis
```

进入从客户端，挂主

Java 代码

```
/usr/servers/redis-2.8.19/src/redis-cli  -p 6661
```

Java 代码

```
127.0.0.1:6661> slaveof 127.0.0.1 6660
OK
127.0.0.1:6661> info replication
\# Replication
role:slave
master_host:127.0.0.1
master_port:6660
master_link_status:up
master_last_io_seconds_ago:3
master_sync_in_progress:0
slave_repl_offset:57
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

进入主

Java 代码

```
/usr/servers/redis-2.8.19# /usr/servers/redis-2.8.19/src/redis-cli  -p 6660
```

Java 代码

```
127.0.0.1:6660> info replication
\# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6661,state=online,offset=85,lag=1
master_repl_offset:85
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:84
127.0.0.1:6660> set i 1
OK
```

进入从

Java 代码

```
/usr/servers/redis-2.8.19/src/redis-cli  -p 6661
```

Java 代码

```
127.0.0.1:6661> get i
"1"
```

此时可以看到主从挂载成功，可以进行主从复制了。使用 slaveof no one 断开主从。

Redis 持久化

Redis 虽然不适合做持久化存储，但是为了防止数据丢失有时需要进行持久化存储，此时可以挂载一个从（叶子节点）只进行持久化存储工作，这样假设其他服务器挂了，我们可以通过这个节点进行数据恢复。

Redis 持久化有 RDB 快照模式和 AOF 追加模式，根据自己需求进行选择。

RDB 持久化

Java 代码

```
\#格式save seconds changes 即N秒变更N次则保存，从如下默认配置可以看到丢失数据的周期很长，通过save “” 配置可以完全
save 900 1
save 300 10
save 60 10000
\#RDB是否进行压缩，压缩耗CPU但是可以减少存储大小
rdbcompression yes
```

```
\#RDB保存的位置，默认当前位置
dir ./
\#RDB保存的数据库名称
dbfilename dump.rdb
\#不使用AOF模式，即RDB模式
appendonly no
```

可以通过 set 一个数据，然后很快的 kill 掉 redis 进程然后再启动会发现数据丢失了。

AOF 持久化

AOF（append only file）即文件追加模式，即把每一个用户操作的命令保存下来，这样就会存在好多重复的命令导致恢复时间过长，那么可以通过相应的配置定期进行 AOF 重写来减少重复。

Java 代码

```
\#开启AOF
appendonly yes
\#AOF保存的位置，默认当前位置
dir ./
\#AOF保存的数据库名称
appendfilename appendonly.aof
\#持久化策略，默认每秒fsync一次，也可以选择always即每次操作都进行持久化，或者no表示不进行持久化而是借助操作系统的同步
appendfsync everysec

\#AOF重写策略（同时满足如下两个策略进行重写）
\#当AOF文件大小占到初始文件大小的多少百分比时进行重写
auto-aof-rewrite-percentage 100
\#触发重写的最小文件大小
auto-aof-rewrite-min-size 64mb

\#为减少磁盘操作，暂缓重写阶段的磁盘同步
no-appendfsync-on-rewrite no
```

此处的 appendfsync everysec 可以认为是 RDB 和 AOF 的一个折中方案。

#当 bgsave 出错时停止写（MISCONF Redis is configured to save RDB snapshots, but is currently not able to persist on disk.），遇到该错误可以暂时改为 no，当写成功后再改回 yes stop-writes-on-bgsave-error yes

更多 Redis 持久化请参考 http://redis.readthedocs.org/en/latest/topic/persistence.html。

Redis 动态调整配置

获取 maxmemory(10mb)

Java 代码

```
127.0.0.1:6660> config get maxmemory
1) "maxmemory"
2) "10485760"
```

设置新的 maxmemory(20mb)

Java 代码

```
127.0.0.1:6660> config set maxmemory 20971520
OK
```

但是此时重启 redis 后该配置会丢失，可以执行如下命令重写配置文件

Java 代码

```
127.0.0.1:6660> config rewrite
OK
```

注意：此时所以配置包括主从配置都会重写。

Redis 执行 Lua 脚本

Redis 客户端支持解析和处理 lua 脚本，因为 Redis 的单线程机制，我们可以借助 Lua 脚本实现一些原子操作，如扣减库存/红包之类的。此处不建议使用 EVAL 直接发送 lua 脚本到客户端，因为其每次都会进行 Lua 脚本的解析，而是使用 SCRIPT LOAD+ EVALSHA 进行操作。未来不知道是否会用 luajit 来代替 lua，让redis lua 脚本性能更强。

到此基本的 Redis 知识就讲完了。

Twemproxy 设置

一旦涉及到一台物理机无法存储的情况就需要考虑使用分片机制将数据存储到多台服务器，可以说是 Redis 集群；如果客户端都是如 Java 没什么问题，但是如果有多种类型客户端（如 PHP、C）等也要使用那么需要保证它们的分片逻辑是一样的；另外随着客户端的增加，连接数也会随之增多，发展到一定地步肯定会出现连接数不够用的；此时 Twemproxy 就可以上场了。主要作用：分片、减少连接数。另外还提供了 Hash Tag 机制来帮助我们将相似的数据存储到同一个分片。另外也可以参考豌豆荚的 https://github.com/wandoulabs/codis。

基本配置

其使用 YML 语法，如

Java 代码

```
server1:
  listen: 127.0.0.1:1111
  hash: fnv1a_64
  distribution: ketama
  timeout:1000
  redis: true
  servers:
   - 127.0.0.1:6660:1
   - 127.0.0.1:6661:1
```

- server1：是给当前分片配置起的名字，一个配置文件可以有多个分片配置；
- listen： 监听的 ip 和端口；
- hash：散列算法；
- distribution：分片算法，比如一致性 Hash/ 取模；
- timeout：连接后端 Redis 或接收响应的超时时间；
- redis：是否是 redis 代理，如果是 false 则是 memcached 代理；
- servers：代理的服务器列表，该列表会使用 distribution 配置的分片算法进行分片；

分片算法

hash 算法：

```
one_at_a_time
md5
crc16
crc32 (crc32 implementation compatible with libmemcached)
crc32a (correct crc32 implementation as per the spec)
fnv1_64
fnv1a_64
fnv1_32
fnv1a_32
hsieh
murmur
jenkins
```

分片算法：

```
ketama(一致性 Hash 算法)
modula(取模)
random(随机算法)
```

服务器列表

servers:

- ip:port:weight alias

如

servers:

- 127.0.0.1:6660:1
- 127.0.0.1:6661:1

或者

servers:

- 127.0.0.1:6660:1 server1
- 127.0.0.1:6661:1 server2

推荐使用后一种方式，默认情况下使用 ip:port:weight 进行散列并分片，这样假设服务器宕机换上新的服务器，那么此时得到的散列值就不一样了，因此建议给每个配置起一个别名来保证映射到自己想要的服务器。即如果不使用一致性 Hash 算法来作缓存服务器，而是作持久化存储服务器时就更有必要了（即不存在服务器下线的情况，即使服务器 ip:port 不一样但仍然要得到一样的分片结果）。

HashTag

比如一个商品有：商品基本信息(p:id:)、商品介绍(d:id:)、颜色尺码(c:id:)等，假设我们存储时不采用 HashTag 将会导致这些数据不会存储到一个分片，而是分散到多个分片，这样获取时将需要从多个分片获取数据进行合并，无法进行 mget；那么如果有了 HashTag，那么可以使用“::”中间的数据做分片逻辑，这样 id 一样的将会分到一个分片。

nutcracker.yml配置如下

Java 代码

```
server1:
  listen: 127.0.0.1:1111
  hash: fnv1a_64
  distribution: ketama
  redis: true
  hash_tag: "::"
  servers:
   - 127.0.0.1:6660:1 server1
   - 127.0.0.1:6661:1 server2
```

连接 Twemproxy

Java 代码

```
/usr/servers/redis-2.8.19/src/redis-cli  -p 1111
```

Java 代码

```
127.0.0.1:1111> set p:12: 1
OK
127.0.0.1:1111> set d:12: 1
OK
127.0.0.1:1111> set c:12: 1
OK
```

在我的服务器上可以连接 6660 端口

Java 代码

```
/usr/servers/redis-2.8.19/src/redis-cli  -p 6660
127.0.0.1:6660> get p:12:
"1"
127.0.0.1:6660> get d:12:
"1"
127.0.0.1:6660> get c:12:
"1"
```

一致性 Hash 与服务器宕机

如果我们把 Redis 服务器作为缓存服务器并使用一致性 Hash 进行分片，当有服务器宕机时需要自动从一致性 Hash 环上摘掉，或者其上线后自动加上，此时就需要如下配置：

#是否在节点故障无法响应时自动摘除该节点，如果作为存储需要设置为为false auto_eject_hosts: true #重试时间（毫秒），重新连接一个临时摘掉的故障节点的间隔，如果判断节点正常会自动加到一致性Hash环上 server_retry_timeout: 30000 #节点故障无法响应多少次从一致性Hash环临时摘掉它，默认是2 server_failure_limit: 2

支持的 Redis 命令

不是所有 Redis 命令都支持，请参考 [https://github.com/twitter/twemproxy/blob/master/notes/redis.mdhref="https://github.com/twitter/twemproxy/blob/master/notes/redis.md")。

因为我们所有的 Twemproxy 配置文件规则都是一样的，因此我们应该将其移到我们项目中。

Java 代码

```
cp /usr/servers/twemproxy-0.4.0/conf/nutcracker.yml  /usr/example/
```

另外 Twemproxy 提供了启动/重启/停止脚本方便操作，但是需要修改配置文件位置为 /usr/example/nutcracker.yml。

Java 代码

```
chmod +x /usr/servers/twemproxy-0.4.0/scripts/nutcracker.init
vim /usr/servers/twemproxy-0.4.0/scripts/nutcracker.init
```

将 OPTIONS 改为

OPTIONS="-d -c /usr/example/nutcracker.yml"

另外注释掉. /etc/rc.d/init.d/functions；将 daemon --user ${USER} ${prog} $OPTIONS 改为 ${prog} $OPTIONS；将 killproc 改为 killall。

这样就可以使用如下脚本进行启动、重启、停止了。

```
/usr/servers/twemproxy-0.4.0/scripts/nutcracker.init {start|stop|status|restart|reload|condrestart}
```

对于扩容最简单的办法是：

1. 创建新的集群；
2. 双写两个集群；
3. 把数据从老集群迁移到新集群（不存在才设置值，防止覆盖新的值）；
4. 复制速度要根据实际情况调整，不能影响老集群的性能；
5. 切换到新集群即可，如果使用Twemproxy代理层的话，可以做到迁移对读的应用透明。

4

Lua 模块开发

在实际开发中，不可能把所有代码写到一个大而全的 lua 文件中，需要进行分模块开发；而且模块化是高性能 Lua 应用的关键。使用 require 第一次导入模块后，所有 Nginx 进程全局共享模块的数据和代码，每个 Worker 进程需要时会得到此模块的一个副本（Copy-On-Write），即模块可以认为是每 Worker 进程共享而不是每 Nginx Server 共享；另外注意之前我们使用 init_by_lua 中初始化的全局变量是每请求复制一个；如果想在多个 Worker 进程间共享数据可以使用 ngx.shared.DICT 或如 Redis 之类的存储。

在 /usr/example/lualib 中已经提供了大量第三方开发库如 cjson、redis 客户端、mysql客户端：

```
cjson.so
resty/
  aes.lua
  core.lua
  dns/
  lock.lua
  lrucache/
  lrucache.lua
  md5.lua
  memcached.lua
  mysql.lua
  random.lua
  redis.lua
  ......
```

需要注意在使用前需要将库在 nginx.conf 中导入：

Java 代码

```
\#lua模块路径，其中”;;”表示默认搜索路径，默认到/usr/servers/nginx下找
lua_package_path "/usr/example/lualib/?.lua;;"; #lua 模块
lua_package_cpath "/usr/example/lualib/?.so;;"; #c模块
```

使用方式是在lua中通过如下方式引入

Java 代码

```
local cjson = require(“cjson”)
local redis = require(“resty.redis”)
```

接下来我们来开发一个简单的 lua 模块。

Java 代码

```
vim /usr/example/lualib/module1.lua
```

Java 代码

```
local count = 0
local function hello()
  count = count + 1
  ngx.say("count : ", count)
end

local _M = {
  hello = hello
}

return _M
```

开发时将所有数据做成局部变量/局部函数；通过 _M 导出要暴露的函数，实现模块化封装。

接下来创建 test_module_1.lua

Java 代码

```
vim /usr/example/lua/test_module_1.lua
```

Java 代码

```
local module1 = require("module1")

module1.hello()
```

使用 local var = require ("模块名")，该模块会到 lua_package_path 和lua_package_cpath 声明的的位置查找我们的模块，对于多级目录的使用 require ("目录1.目录2.模块名")加载。

example.conf 配置

Java 代码

```
location /lua_module_1 {
   default_type 'text/html';
   lua_code_cache on;
   content_by_lua_file /usr/example/lua/test_module_1.lua;
}
```

访问如 `http://192.168.1.2/lua_module_1` 进行测试，会得到类似如下的数据，count 会递增

```
count : 1
count： 2
.....
count： N
```

此时可能发现 count 一直递增，假设我们的 worker_processes 2，我们可以通过 kill -9 nginx worker process 杀死其中一个 Worker 进程得到 count 数据变化。

假设我们创建了 vim/usr/example/lualib/test/module2.lua 模块，可以通过 local module2 = require("test.module2") 加载模块

基本的模块开发就完成了，如果是只读数据可以通过模块中声明 local 变量存储；如果想在每 Worker 进程共享，请考虑竞争；如果要在多个 Worker 进程间共享请考虑使用 ngx.shared.DICT 或如 Redis 存储。

5

常用 Lua 开发库 1-redis、mysql、http 客户端

对于开发来说需要有好的生态开发库来辅助我们快速开发，而 Lua 中也有大多数我们需要的第三方开发库如 Redis、Memcached、Mysql、Http 客户端、JSON、模板引擎等。一些常见的 Lua 库可以在 github 上搜索，https://github.com/search?utf8=%E2%9C%93&q=lua+resty。

Redis 客户端

lua-resty-redis 是为基于 cosocket API 的 ngx_lua 提供的 Lua redis 客户端，通过它可以完成 Redis 的操作。默认安装 OpenResty 时已经自带了该模块，使用文档可参考https://github.com/openresty/lua-resty-redis。

在测试之前请启动 Redis 实例：

```
nohup /usr/servers/redis-2.8.19/src/redis-server/usr/servers/redis-2.8.19/redis_6660.conf &
```

基本操作

编辑 test_redis_baisc.lua

Java 代码

```
local function close_redis(red)
    if not red then
        return
    end
    local ok, err = red:close()
    if not ok then
        ngx.say("close redis error : ", err)
    end
end

local redis = require("resty.redis")

--创建实例
local red = redis:new()
--设置超时（毫秒）
red:set_timeout(1000)
--建立连接
local ip = "127.0.0.1"
local port = 6660
local ok, err = red:connect(ip, port)
if not ok then
    ngx.say("connect to redis error : ", err)
    return close_redis(red)
end
--调用API进行处理
```

```
ok, err = red:set("msg", "hello world")
if not ok then
    ngx.say("set msg error : ", err)
    return close_redis(red)
end

--调用API获取数据
local resp, err = red:get("msg")
if not resp then
    ngx.say("get msg error : ", err)
    return close_reedis(red)
end
--得到的数据为空处理
if resp == ngx.null then
    resp = "  --比如默认值
end
ngx.say("msg : ", resp)

close_redis(red)
```

基本逻辑很简单，要注意此处判断是否为 nil，需要跟 ngx.null 比较。

example.conf 配置文件

Java 代码

```
 location /lua_redis_basic {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_redis_basic.lua;
}
```

访问如 `http://192.168.1.2/lua_redis_basic` 进行测试，正常情况得到如下信息 msg : hello world

连接池

建立 TCP 连接需要三次握手而释放 TCP 连接需要四次握手，而这些往返时延仅需要一次，以后应该复用 TCP 连接，此时就可以考虑使用连接池，即连接池可以复用连接。

我们只需要将之前的 close_redis 函数改造为如下即可：

Java 代码

```
local function close_redis(red)
    if not red then
        return
    end
    --释放连接(连接池实现)
    local pool_max_idle_time = 10000 --毫秒
    local pool_size = 100 --连接池大小
    local ok, err = red:set_keepalive(pool_max_idle_time, pool_size)
    if not ok then
        ngx.say("set keepalive error : ", err)
    end
end
```

即设置空闲连接超时时间防止连接一直占用不释放；设置连接池大小来复用连接。

此处假设调用 red:set_keepalive()，连接池大小通过 nginx.conf 中 http 部分的如下指令定义：

#默认连接池大小，默认 30 lua_socket_pool_size 30; #默认超时时间,默认 60s lua_socket_keepalive_timeout 60s;

注意：

1. 连接池是每 Worker 进程的，而不是每 Server 的；
2. 当连接超过最大连接池大小时，会按照 LRU 算法回收空闲连接为新连接使用；
3. 连接池中的空闲连接出现异常时会自动被移除；
4. 连接池是通过 ip 和 port 标识的，即相同的 ip 和 port 会使用同一个连接池（即使是不同类型的客户端如 Redis、Memcached）；
5. 连接池第一次 set_keepalive 时连接池大小就确定下了，不会再变更；
6. cosocket 的连接池 http://wiki.nginx.org/HttpLuaModule#tcpsock:setkeepalive。

pipeline

pipeline 即管道，可以理解为把多个命令打包然后一起发送；MTU（Maxitum Transmission Unit 最大传输单元）为二层包大小，一般为 1500 字节；而 MSS（Maximum Segment Size 最大报文分段大小）为四层包大小，其一般是 1500-20（IP 报头）-20（TCP 报头）=1460 字节；因此假设我们执行的多个 Redis 命令能在一个报文中传输的话，可以减少网络往返来提高速度。因此可以根据实际情况来选择走 pipeline 模式将多个命令打包到一个报文发送然后接受响应，而 Redis 协议也能很简单的识别和解决粘包。

修改之前的代码片段

Java 代码

```
red:init_pipeline()
red:set("msg1", "hello1")
red:set("msg2", "hello2")
red:get("msg1")
red:get("msg2")
local respTable, err = red:commit_pipeline()

--得到的数据为空处理
if respTable == ngx.null then
    respTable = {}  --比如默认值
end

--结果是按照执行顺序返回的一个table
for i, v in ipairs(respTable) do
   ngx.say("msg : ", v, "<br/>")
end
```

通过 init_pipeline() 初始化，然后通过 commit_pipieline() 打包提交 init_pipeline() 之后的Redis命令；返回结果是一个 lua table，可以通过 ipairs 循环获取结果；

配置相应 location，测试得到的结果

```
msg : OK
msg : OK
msg : hello1
msg : hello2
```

Redis Lua 脚本

利用 Redis 单线程特性，可以通过在 Redis 中执行 Lua 脚本实现一些原子操作。如之前的 red:get("msg") 可以通过如下两种方式实现：

1、直接 eval：

Java 代码

```
local resp, err = red:eval("return redis.call('get', KEYS[1])", 1, "msg");
```

2、script load 然后 evalsha SHA1 校验和，这样可以节省脚本本身的服务器带宽： Java 代码

```
local sha1, err = red:script("load",  "return redis.call('get', KEYS[1])");
if not sha1 then
   ngx.say("load script error : ", err)
```

```
  return close_redis(red)
end
ngx.say("sha1 : ", sha1, "<br/>")
local resp, err = red:evalsha(sha1, 1, "msg");
```

首先通过 script load 导入脚本并得到一个 sha1 校验和（仅需第一次导入即可），然后通过evalsha 执行 sha1 校验和即可，这样如果脚本很长通过这种方式可以减少带宽的消耗。

此处仅介绍了最简单的 redis lua 脚本，更复杂的请参考官方文档学习使用。

另外 Redis 集群分片算法该客户端没有提供需要自己实现，当然可以考虑直接使用类似于Twemproxy 这种中间件实现。

Memcached 客户端使用方式和本文类似，本文就不介绍了。

Mysql 客户端

lua-resty-mysql 是为基于 cosocket API 的 ngx_lua 提供的 Lua Mysql 客户端，通过它可以完成 Mysql 的操作。默认安装 OpenResty 时已经自带了该模块，使用文档可参考https://github.com/openresty/lua-resty-mysql。

编辑 test_mysql.lua

Java 代码

```
local function close_db(db)
    if not db then
        return
    end
    db:close()
end

local mysql = require("resty.mysql")
--创建实例
local db, err = mysql:new()
if not db then
    ngx.say("new mysql error : ", err)
    return
end
--设置超时时间(毫秒)
db:set_timeout(1000)

local props = {
    host = "127.0.0.1",
    port = 3306,
    database = "mysql",
    user = "root",
    password = "123456"
}

local res, err, errno, sqlstate = db:connect(props)

if not res then
   ngx.say("connect to mysql error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
   return close_db(db)
end
```

```
--删除表
local drop_table_sql = "drop table if exists test"
res, err, errno, sqlstate = db:query(drop_table_sql)
if not res then
  ngx.say("drop table error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
  return close_db(db)
end

--创建表
local create_table_sql = "create table test(id int primary key auto_increment, ch varchar(100))"
res, err, errno, sqlstate = db:query(create_table_sql)
if not res then
  ngx.say("create table error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
  return close_db(db)
end

--插入
local insert_sql = "insert into test (ch) values('hello')"
res, err, errno, sqlstate = db:query(insert_sql)
if not res then
  ngx.say("insert error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
  return close_db(db)
end

res, err, errno, sqlstate = db:query(insert_sql)

ngx.say("insert rows : ", res.affected_rows, " , id : ", res.insert_id, "<br/>")

--更新
local update_sql = "update test set ch = 'hello2' where id =" .. res.insert_id
res, err, errno, sqlstate = db:query(update_sql)
if not res then
  ngx.say("update error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
  return close_db(db)
end

ngx.say("update rows : ", res.affected_rows, "<br/>")
--查询
local select_sql = "select id, ch from test"
res, err, errno, sqlstate = db:query(select_sql)
if not res then
  ngx.say("select error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
  return close_db(db)
end
```

```
for i, row in ipairs(res) do
  for name, value in pairs(row) do
    ngx.say("select row ", i, " : ", name, " = ", value, "<br/>")
  end
end

ngx.say("<br/>")
--防止sql注入
local ch_param = ngx.req.get_uri_args()["ch"] or ''
--使用ngx.quote_sql_str防止sql注入
local query_sql = "select id, ch from test where ch = " .. ngx.quote_sql_str(ch_param)
res, err, errno, sqlstate = db:query(query_sql)
if not res then
  ngx.say("select error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
  return close_db(db)
end

for i, row in ipairs(res) do
  for name, value in pairs(row) do
    ngx.say("select row ", i, " : ", name, " = ", value, "<br/>")
  end
end

--删除
local delete_sql = "delete from test"
res, err, errno, sqlstate = db:query(delete_sql)
if not res then
  ngx.say("delete error : ", err, " , errno : ", errno, " , sqlstate : ", sqlstate)
  return close_db(db)
end

ngx.say("delete rows : ", res.affected_rows, "<br/>")


close_db(db)
```

对于新增/修改/删除会返回如下格式的响应：

Java 代码

```
{
    insert_id = 0,
    server_status = 2,
    warning_count = 1,
```

```
    affected_rows = 32,
    message = nil
}
```

affected_rows 表示操作影响的行数，insert_id 是在使用自增序列时产生的 id。

对于查询会返回如下格式的响应：

Java 代码

```
{
    { id= 1, ch= "hello"},
    { id= 2, ch= "hello2"}
}
```

null 将返回 ngx.null。

example.conf 配置文件

Java 代码

```
location /lua_mysql {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_mysql.lua;
}
```

访问如 http://192.168.1.2/lua_mysql?ch=hello 进行测试，得到如下结果

Java 代码

```
insert rows : 1 , id : 2
update rows : 1
select row 1 : ch = hello
select row 1 : id = 1
select row 2 : ch = hello2
select row 2 : id = 2
select row 1 : ch = hello
select row 1 : id = 1
delete rows : 2
```

客户端目前还没有提供预编译 SQL 支持（即占位符替换位置变量），这样在入参时记得使用 ngx.quote_sql_str 进行字符串转义，防止 sql 注入；连接池和之前 Redis 客户端完全一样就不介绍了。

对于 Mysql 客户端的介绍基本够用了，更多请参考 https://github.com/openresty/lua-resty-mysql。

其他如 MongoDB 等数据库的客户端可以从 github 上查找使用。

Http 客户端

OpenResty 默认没有提供 Http 客户端，需要使用第三方提供；当然我们可以通过 ngx.location.capture 去方式实现，但是有一些限制，后边我们再做介绍。

我们可以从 github 上搜索相应的客户端，比如 https://github.com/pintsized/lua-resty-http。

lua-resty-http

下载 lua-resty-http 客户端到 lualib

Java 代码

```
cd /usr/example/lualib/resty/
wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http_headers.lua
wget https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http.lua
```

test_http_1.lua

Java 代码

```
local http = require("resty.http")
--创建http客户端实例
local httpc = http.new()

local resp, err = httpc:request_uri("http://s.taobao.com", {
    method = "GET",
    path = "/search?q=hello",
    headers = {
        ["User-Agent"] = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.22
    }
})

if not resp then
    ngx.say("request error :", err)
    return
end

--获取状态码
ngx.status = resp.status
```

```
--获取响应头
for k, v in pairs(resp.headers) do
    if k ~= "Transfer-Encoding" and k ~= "Connection" then
        ngx.header[k] = v
    end
end
--响应体
ngx.say(resp.body)

httpc:close()
```

响应头中的 Transfer-Encoding 和 Connection 可以忽略，因为这个数据是当前 server 输出的。

example.conf 配置文件

Java 代码

```
location /lua_http_1 {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_http_1.lua;
}
```

在 nginx.conf 中的 http 部分添加如下指令来做 DNS 解析

Java 代码

```
resolver 8.8.8.8;
```

记得要配置 DNS 解析器 resolver 8.8.8.8，否则域名是无法解析的。

访问如 `http://192.168.1.2/lua_http_1` 会看到淘宝的搜索界面。

使用方式比较简单，如超时和连接池设置和之前 Redis 客户端一样，不再阐述。更多客户端使用规则请参考 https://github.com/pintsized/lua-resty-http。

ngx.location.capture

ngx.location.capture 也可以用来完成 http 请求，但是它只能请求到相对于当前 nginx 服务器的路径，不能使用之前的绝对路径进行访问，但是我们可以配合 nginx upstream 实现我们想要的功能。

在 nginx.cong中 的 http 部分添加如下 upstream 配置

Java 代码

```
upstream backend {
    server s.taobao.com;
    keepalive 100;
}
```

即我们将请求 upstream 到 backend；另外记得一定要添加之前的 DNS 解析器。

在 example.conf 配置如下 location

Java 代码

```
location ~ /proxy/(.*) {
   internal;
   proxy_pass http://backend/$1$is_args$args;
}
```

internal 表示只能内部访问，即外部无法通过 url 访问进来； 并通过 proxy_pass 将请求转发到 upstream。

test_http_2.lua

Java 代码

```
local resp = ngx.location.capture("/proxy/search", {
    method = ngx.HTTP_GET,
    args = {q = "hello"}

})
if not resp then
    ngx.say("request error :", err)
    return
end
```

```
ngx.log(ngx.ERR, tostring(resp.status))

--获取状态码
ngx.status = resp.status

--获取响应头
for k, v in pairs(resp.header) do
    if k ~= "Transfer-Encoding" and k ~= "Connection" then
        ngx.header[k] = v
    end
end
--响应体
if resp.body then
    ngx.say(resp.body)
end
```

通过 ngx.location.capture 发送一个子请求，此处因为是子请求，所有请求头继承自当前请求，还有如 ngx.ctx 和ngx.var 是否继承可以参考官方文档 http://wiki.nginx.org/HttpLuaModule#ngx.location.capture。另外还提供了 ngx.location.capture_multi用于并发发出多个请求，这样总的响应时间是最慢的一个，批量调用时有用。

example.conf 配置文件

Java 代码

```
location /lua_http_2 {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_http_2.lua;
}
```

访问如 http://192.168.1.2/lua_http_2 进行测试可以看到淘宝搜索界面。

我们通过 upstream+ngx.location.capture 方式虽然麻烦点，但是得到更好的性能和upstream 的连接池、负载均衡、故障转移、proxy cache 等特性。

不过因为继承在当前请求的请求头，所以可能会存在一些问题，比较常见的就是 gzip 压缩问题，ngx.location.capture 不会解压缩后端服务器的 GZIP 内容，解决办法可以参考https://github.com/openresty/lua-nginx-module/issues/12；因为我们大部分这种http 调用的都是内部服务，因此完全可以在 proxy location 中添加proxy_pass_request_headers off; 来不传递请求头。

6

JSON 库、编码转换、字符串处理

JSON 库

在进行数据传输时 JSON 格式目前应用广泛，因此从 Lua 对象与 JSON 字符串之间相互转换是一个非常常见的功能；目前 Lua 也有几个 JSON 库，本人用过 cjson、dkjson。其中 cjson的语法严格（比如 unicode \u0020\u7eaf ），要求符合规范否则会解析失败（如 \u002 ），而 dkjson 相对宽松，当然也可以通过修改 cjson 的源码来完成一些特殊要求。而在使用dkjson 时也没有遇到性能问题，目前使用的就是 dkjson。使用时要特别注意的是大部分 JSON库都仅支持 UTF-8 编码；因此如果你的字符编码是如 GBK 则需要先转换为 UTF-8 然后进行处理。

test_cjson.lua

Java 代码

```
local cjson = require("cjson")

--lua对象到字符串
local obj = {
    id = 1,
    name = "zhangsan",
    age = nil,
    is_male = false,
    hobby = {"film", "music", "read"}
}

local str = cjson.encode(obj)
ngx.say(str, "<br/>")

--字符串到lua对象
str = '{"hobby":["film","music","read"],"is_male":false,"name":"zhangsan","id":1,"age":null}'
local obj = cjson.decode(str)

ngx.say(obj.age, "<br/>")
ngx.say(obj.age == nil, "<br/>")
ngx.say(obj.age == cjson.null, "<br/>")
ngx.say(obj.hobby[1], "<br/>")


--循环引用
obj = {
  id = 1
```

```
}
obj.obj = obj
-- Cannot serialise, excessive nesting
--ngx.say(cjson.encode(obj), "<br/>")
local cjson_safe = require("cjson.safe")
--nil
ngx.say(cjson_safe.encode(obj), "<br/>")
```

null 将会转换为 cjson.null；循环引用会抛出异常 Cannot serialise, excessive nesting，默认解析嵌套深度是 1000，可以通过 cjson.encode_max_depth() 设置深度提高性能；使用 cjson.safe 不会抛出异常而是返回 nil。

example.conf 配置文件

Java 代码

```
location ~ /lua_cjson {
  default_type 'text/html';
  lua_code_cache on;
  content_by_lua_file /usr/example/lua/test_cjson.lua;
}
```

访问如 http://192.168.1.2/lua_cjson 将得到如下结果

Java 代码

```
{"hobby":["film","music","read"],"is_male":false,"name":"zhangsan","id":1}
null
false
true
film
nil
```

lua-cjson 文档 http://www.kyne.com.au/~mark/software/lua-cjson-manual.html。

接下来学习下 dkjson。

下载 dkjson 库

Java 代码

```
cd /usr/example/lualib/
wget http://dkolf.de/src/dkjson-lua.fsl/raw/dkjson.lua?name=16cbc26080996d9da827df42cb0844a25518eeb3 -O dkjson.
```

test_dkjson.lua

Java 代码

```
local dkjson = require("dkjson")

--lua对象到字符串
local obj = {
    id = 1,
    name = "zhangsan",
    age = nil,
    is_male = false,
    hobby = {"film", "music", "read"}
}

local str = dkjson.encode(obj, {indent = true})
ngx.say(str, "<br/>")

--字符串到lua对象
str = '{"hobby":["film","music","read"],"is_male":false,"name":"zhangsan","id":1,"age":null}'
local obj, pos, err = dkjson.decode(str, 1, nil)

ngx.say(obj.age, "<br/>")
ngx.say(obj.age == nil, "<br/>")
ngx.say(obj.hobby[1], "<br/>")

--循环引用
obj = {
  id = 1
}
obj.obj = obj
--reference cycle
--ngx.say(dkjson.encode(obj), "<br/>")
```

默认情况下解析的 json 的字符会有缩排和换行，使用 {indent = true} 配置将把所有内容放在一行。和 cjson 不同的是解析 json 字符串中的 null 时会得到 nil。

example.conf 配置文件

Java 代码

```
location ~ /lua_dkjson {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_dkjson.lua;
}
```

访问如 http://192.168.1.2/lua_dkjson 将得到如下结果

Java 代码

```
```` { "hobby":["film","music","read"], "is_male":false, "name":"zhangsan", "id":1 }
nil
true
film
```

```
dkjson 文档 [http://dkolf.de/src/dkjson-lua.fsl/home和http://dkolf.de/src/dkjson-lua.fsl/wiki?name=Documentation](http://dk

## 编码转换

我们在使用一些类库时会发现大部分库仅支持 UTF-8 编码，因此如果使用其他编码的话就需要进行编码转换的处理；而 Linux 上最常

安装 lua-iconv 可以通过如下两种方式：

ubuntu 下可以使用如下方式

Java **代码**
```

```
apt-get install luarocks
luarocks install lua-iconv
cp /usr/local/lib/lua/5.1/iconv.so /usr/example/lualib/
```

```
源码安装方式，需要有 gcc 环境

Java **代码**
```
```

wget https://github.com/do^Cloads/ittner/lua-iconv/lua-iconv-7.tar.gz
tar -xvf lua-iconv-7.tar.gz
cd lua-iconv-7
gcc -O2 -fPIC -I/usr/include/lua5.1 -c luaiconv.c -o luaiconv.o -I/usr/include
gcc -shared -o iconv.so -L/usr/local/lib luaiconv.o -L/usr/lib
cp iconv.so /usr/example/lualib/

test_iconv.lua

Java 代码

ngx.say("中文")

此时文件编码必须为 UTF-8，即 Lua 文件编码为什么里边的字符编码就是什么。

example.conf 配置文件

Java **代码**

location ~ /lua_iconv {
default_type 'text/html';
charset gbk;
lua_code_cache on;
content_by_lua_file /usr/example/lua/test_iconv.lua;
}

通过 charset 告诉浏览器我们的字符编码为 gbk。

访问 `http://192.168.1.2/lua_iconv` 会发现输出乱码；

此时需要我们将 test_iconv.lua 中的字符进行转码处理：

Java **代码**

local iconv = require("iconv")
local togbk = iconv.new("gbk", "utf-8")
local str, err = togbk:iconv("中文")
ngx.say(str)

通过转码我们得到最终输出的内容编码为 gbk， 使用方式 iconv.new (目标编码, 源编码)。

有如下可能出现的错误：

Java **代码**

nil
没有错误成功。
iconv.ERROR_NO_MEMORY
内存不足。
iconv.ERROR_INVALID
有非法字符。
iconv.ERROR_INCOMPLETE
有不完整字符。
iconv.ERROR_FINALIZED
使用已经销毁的转换器，比如垃圾回收了。
iconv.ERROR_UNKNOWN
未知错误

iconv 在转换时遇到非法字符或不能转换的字符就会失败，此时可以使用如下方式忽略转换失败的字符

Java **代码**

local togbk_ignore = iconv.new("GBK//IGNORE", "UTF-8")

另外在实际使用中进行 UTF-8 到 GBK 转换过程时，会发现有些字符在 GBK 编码表但是转换不了，此时可以使用更高的编码 GB180

更多介绍请参考 http://ittner.github.io/lua-iconv/。

位运算

Lua 5.3 之前是没有提供位运算支持的，需要使用第三方库，比如 LuaJIT 提供了 bit 库。

test_bit.lua

Java **代码**

local bit = require("bit")
ngx.say(bit.lshift(1, 2))

lshift 进行左移位运算，即得到4。

其他位操作 API 请参考 http://bitop.luajit.org/api.html。Lua 5.3 的位运算操作符 [http://cloudwu.

cache

ngx_lua 模块本身提供了全局共享内存 ngx.shared.DICT 可以实现全局共享，另外可以使用如 Redis 来实现缓存。另外还一个 lua-

创建缓存模块来实现只初始化一次：

Java **代码**

vim /usr/example/lualib/mycache.lua

Java **代码**

```
local lrucache = require("resty.lrucache")
--创建缓存实例，并指定最多缓存多少条目
local cache, err = lrucache.new(200)
if not cache then
ngx.log(ngx.ERR, "create cache error : ", err)
end

local function set(key, value, ttlInSeconds)
cache:set(key, value, ttlInSeconds)
end

local function get(key)
return cache:get(key)
end

local _M = {
set = set,
get = get
}

return _M
```

此处利用了模块的特性实现了每个 Worker 进行只初始化一次 cache 实例。

test_lrucache.lua

Java **代码**

```
local mycache = require("mycache")
local count = mycache.get("count") or 0
count = count + 1
mycache.set("count", count, 10 * 60 * 60) --10分钟
ngx.say(mycache.get("count"))
```

可以实现诸如访问量统计，但仅是每 Worker 进程的。

example.conf 配置文件

Java **代码**

```
location ~ /lua_lrucache {
default_type 'text/html';
lua_code_cache on;
content_by_lua_file /usr/example/lua/test_lrucache.lua;
}
```

访问如 `http://192.168.1.2/lua_lrucache` 测试。

更多介绍请参考 https://github.com/openresty/lua-resty-lrucache。

字符串处理

Lua 5.3 之前没有提供字符操作相关的函数，如字符串截取、替换等都是字节为单位操作；在实际使用时尤其包含中文的场景下显然不能

Lua UTF-8 库
https://github.com/starwing/luautf8

LuaRocks 安装

Java **代码**

```
#首先确保git安装了
apt-get install git
luarocks install utf8
cp /usr/local/lib/lua/5.1/utf8.so /usr/example/lualib/
```

源码安装

Java 代码

```
wget https://github.com/starwing/luautf8/archive/master.zip
unzip master.zip
cd luautf8-master/
gcc -O2 -fPIC -I/usr/include/lua5.1 -c utf8.c -o utf8.o -I/usr/include
gcc -shared -o utf8.so -L/usr/local/lib utf8.o -L/usr/lib
```

test_utf8.lua

Java 代码

```
local utf8 = require("utf8")
local str = "abc中文"
ngx.say("len : ", utf8.len(str), "
")
ngx.say("sub : ", utf8.sub(str, 1, 4))
```

文件编码必须为 UTF8，此处我们实现了最常用的字符串长度计算和字符串截取。

example.conf 配置文件

Java 代码

```
location ~ /lua_utf8 {
default_type 'text/html';
lua_code_cache on;
content_by_lua_file /usr/example/lua/test_utf8.lua;
}
```

访问如 `http://192.168.1.2/lua_utf8` 测试得到如下结果

```
len : 5
sub : abc中
```

字符串转换为 unicode 编码：

Java **代码**

```
local bit = require("bit")
local bit_band = bit.band
local bit_bor = bit.bor
local bit_lshift = bit.lshift
local string_format = string.format
local string_byte = string.byte
local table_concat = table.concat

local function utf8_to_unicode(str)
if not str or str == "" or str == ngx.null then
return nil
end
local res, seq, val = {}, 0, nil
for i = 1, #str do
local c = string_byte(str, i)
if seq == 0 then
if val then
res[#res + 1] = string_format("%04x", val)
end
```

```
      seq = c < 0x80 and 1 or c < 0xE0 and 2 or c < 0xF0 and 3 or
                    c < 0xF8 and 4 or --c < 0xFC and 5 or c < 0xFE and 6 or
                    0
       if seq == 0 then
          ngx.log(ngx.ERR, 'invalid UTF-8 character sequence' .. ",,," .. tostring(str))
          return str
       end

       val = bit_band(c, 2 ^ (8 - seq) - 1)
    else
       val = bit_bor(bit_lshift(val, 6), bit_band(c, 0x3F))
    end
    seq = seq - 1
end
if val then
    res[#res + 1] = string_format("%04x", val)
end
if #res == 0 then
```

```
    return str
end
return "\\u" .. table_concat(res, "\\u")
```

end

ngx.say("utf8 to unicode : ", utf8_to_unicode("abc中文"), "
")

如上方法将输出 utf8 to unicode : \u0061\u0062\u0063\u4e2d\u6587。

删除空格:

Java **代码**

```
local function ltrim(s)
if not s then
return s
end
local res = s
local tmp = string_find(res, '%S')
if not tmp then
res = ''
elseif tmp ~= 1 then
res = string_sub(res, tmp)
end
return res
end

local function rtrim(s)
if not s then
return s
end
local res = s
local tmp = string_find(res, '%S%s*$')
if not tmp then
res = ''
elseif tmp ~= #res then
```

```
res = string_sub(res, 1, tmp)
end

return res

end

local function trim(s)
if not s then
return s
end
local res1 = ltrim(s)
local res2 = rtrim(res1)
return res2
end

字符串分割：

Java **代码**

function split(szFullString, szSeparator)
local nFindStartIndex = 1
local nSplitIndex = 1
local nSplitArray = {}
while true do
local nFindLastIndex = string.find(szFullString, szSeparator, nFindStartIndex)
if not nFindLastIndex then
nSplitArray[nSplitIndex] = string.sub(szFullString, nFindStartIndex, string.len(szFullString))
break
end
nSplitArray[nSplitIndex] = string.sub(szFullString, nFindStartIndex, nFindLastIndex - 1)
nFindStartIndex = nFindLastIndex + string.len(szSeparator)
nSplitIndex = nSplitIndex + 1
end
return nSplitArray
end
```

如 split("a,b,c", ",") 将得到一个分割后的 table。

到此基本的字符串操作就完成了，其他 luautf8 模块的 API 和 LuaAPI 类似可以参考 http://cloudwu.github.io/lua53doc/manual.html#6.4 http://cloudwu.github.io/lua53doc/manual.html#6.5

另外对于 GBK 的操作，可以先转换为 UTF-8，最后再转换为 GBK 即可。

7

常用 Lua 开发库 3–模板渲染

动态 web 网页开发是 Web 开发中一个常见的场景，比如像京东商品详情页，其页面逻辑是非常复杂的，需要使用模板技术来实现。而 Lua 中也有许多模板引擎，如目前我在使用的 lua-resty-template，可以渲染很复杂的页面，借助 LuaJIT 其性能也是可以接受的。

如果学习过 JavaEE 中的 servlet 和 JSP 的话，应该知道 JSP 模板最终会被翻译成Servlet 来执行；而 lua-resty-template 模板引擎可以认为是 JSP，其最终会被翻译成 Lua 代码，然后通过 ngx.print 输出。

而 lua-resty-template 和大多数模板引擎是类似的，大体内容有：

- 模板位置：从哪里查找模板；
- 变量输出/转义：变量值输出；
- 代码片段：执行代码片段，完成如 if/else、for 等复杂逻辑，调用对象函数/方法；
- 注释：解释代码片段含义；
- include：包含另一个模板片段；
- 其他：lua-resty-template 还提供了不需要解析片段、简单布局、可复用的代码块、宏指令等支持。

首先需要下载 lua-resty-template

Java 代码

```
cd /usr/example/lualib/resty/
wget https://raw.githubusercontent.com/bungle/lua-resty-template/master/lib/resty/template.lua
mkdir /usr/example/lualib/resty/html
cd /usr/example/lualib/resty/html
wget https://raw.githubusercontent.com/bungle/lua-resty-template/master/lib/resty/template/html.lua
```

接下来就可以通过如下代码片段引用了

Java 代码

```
local template = require("resty.template")
```

模板位置

我们需要告诉 lua-resty-template 去哪儿加载我们的模块，此处可以通过 set 指令定义template_location、template_root 或者从 root 指令定义的位置加载。

如我们可以在 example.conf 配置文件的 server 部分定义

Java 代码

```
\#first match ngx location
set $template_location "/templates";
\#then match root read file
set $template_root "/usr/example/templates";
也可以通过在server部分定义root指令
```

Java 代码

```
root /usr/example/templates;
```

其顺序是

Java 代码

```
local function load_ngx(path)
    local file, location = path, ngx_var.template_location
    if file:sub(1)  == "/" then file = file:sub(2) end
    if location and location ~= "" then
        if location:sub(-1) == "/" then location = location:sub(1, -2) end
        local res = ngx_capture(location .. '/' .. file)
        if res.status == 200 then return res.body end
    end
    local root = ngx_var.template_root or ngx_var.document_root
    if root:sub(-1) == "/" then root = root:sub(1, -2) end
    return read_file(root .. "/" .. file) or path
end
```

1. 通过 ngx.location.capture从template_location 查找，如果找到（状态为为 200 ）则使用该内容作为模板；此种方式是一种动态获取模板方式；
2. 如果定义了 template_root，则从该位置通过读取文件的方式加载模板；
3. 如果没有定义 template_root，则默认从 root 指令定义的 document_root 处加载模板。

此处建议首先 template_root，如果实在有问题再使用 template_location，尽量不要通过 root 指令定义的 document_root 加载，因为其本身的含义不是给本模板引擎使用的。

接下来定义模板位置

Java 代码

```
mkdir /usr/example/templates
mkdir /usr/example/templates2
```

example.conf 配置 server 部分

Java 代码

```
\#first match ngx location
set $template_location "/templates";
\#then match root read file
set $template_root "/usr/example/templates";

location /templates {
    internal;
    alias /usr/example/templates2;
}
```

首先查找 /usr/example/template2，找不到会查找 /usr/example/templates。

然后创建两个模板文件

Java 代码

```
vim /usr/example/templates2/t1.html
```

内容为

Java 代码

```
template2
```

Java 代码

```
vim /usr/example/templates/t1.html
```

内容为

Java 代码

```
template1
```

test_temlate_1.lua

Java 代码

```
local template = require("resty.template")
template.render("t1.html")
```

example.conf 配置文件

Java 代码

```
location /lua_template_1 {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_template_1.lua;
}
```

访问如 `http://192.168.1.2/lua_template_1` 将看到 template2 输出。然后 rm/usr/example/templates2/t1.html，reload nginx 将看到 template1 输出。

接下来的测试我们会把模板文件都放到 /usr/example/templates 下。

API

使用模板引擎目的就是输出响应内容；主要用法两种：直接通过 ngx.print 输出或者得到模板渲染之后的内容按照想要的规则输出。

test_template_2.lua

Java 代码

```
local template = require("resty.template")
--是否缓存解析后的模板，默认true
template.caching(true)
--渲染模板需要的上下文(数据)
local context = {title = "title"}
--渲染模板
template.render("t1.html", context)


ngx.say("<br/>")
```

```
--编译得到一个lua函数
local func = template.compile("t1.html")
--执行函数，得到渲染之后的内容
local content = func(context)
--通过ngx API输出
ngx.say(content)
```

常见用法即如下两种方式：要么直接将模板内容直接作为响应输出，要么得到渲染后的内容然后按照想要的规则输出。

examle.conf 配置文件

Java **代码**

```
location /lua_template_2 {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_template_2.lua;
}
```

使用示例

test_template_3.lua

Java 代码

```
local template = require("resty.template")

local context = {
    title = "测试",
    name = "张三",
    description = "<script>alert(1);</script>",
    age = 20,
    hobby = {"电影", "音乐", "阅读"},
    score = {语文 = 90, 数学 = 80, 英语 = 70},
    score2 = {
        {name = "语文", score = 90},
        {name = "数学", score = 80},
        {name = "英语", score = 70},
    }
}

template.render("t3.html", context)
```

请确认文件编码为 UTF-8；context 即我们渲染模板使用的数据。

模板文件 /usr/example/templates/t3.html

Java 代码

```
{(header.html)}
  <body>
    {# 不转义变量输出 #}
    姓名：{* string.upper(name) *}<br/>
    {# 转义变量输出 #}
    简介：{{description}}<br/>
    {# 可以做一些运算 #}
    年龄: {* age + 1 *}<br/>
    {# 循环输出 #}
    爱好:
    {% for i, v in ipairs(hobby) do %}
```

```
      {% if i > 1 then %}，{% end %}
      {* v *}
    {% end %}<br/>

    成绩：
    {% local i = 1; %}
    {% for k, v in pairs(score) do %}
      {% if i > 1 then %}，{% end %}
      {* k *} = {* v *}
      {% i = i + 1 %}
    {% end %}<br/>
    成绩2：
    {% for i = 1, #score2 do local t = score2[i] %}
      {% if i > 1 then %}，{% end %}
       {* t.name *} = {* t.score *}
    {% end %}<br/>
    {# 中间内容不解析 #}
    {-raw-}{(file)}{-raw-}
{(footer.html)}
```

- {(include_file)}：包含另一个模板文件；
- {* var *}：变量输出；
- {{ var }}：变量转义输出；
- {% code %}：代码片段；
- {# comment #}：注释；
- {-raw-}：中间的内容不会解析，作为纯文本输出；

模板最终被转换为 Lua 代码进行执行，所以模板中可以执行任意 Lua 代码。

example.conf 配置文件

Java 代码

```
location /lua_template_3 {
    default_type 'text/html';
    lua_code_cache on;
    content_by_lua_file /usr/example/lua/test_template_3.lua;
}
```

访问如 `http://192.168.1.2/lua_template_3` 进行测试。

基本的模板引擎使用到此就介绍完了。

8

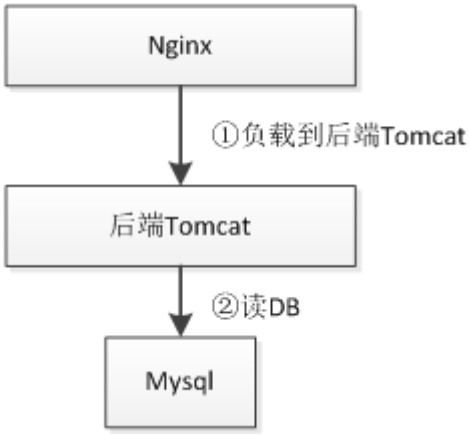
HTTP 服务

此处我说的 HTTP 服务主要指如访问京东网站时我们看到的热门搜索、用户登录、实时价格、实时库存、服务支持、广告语等这种非 Web 页面，而是在 Web 页面中异步加载的相关数据。这些服务有个特点即访问量巨大、逻辑比较单一；但是如实时库存逻辑其实是非常复杂的。在京东这些服务每天有几亿十几亿的访问量，比如实时库存服务曾经在没有任何 IP 限流、DDos 防御的情况被刷到600多万/分钟的访问量，而且能轻松应对。支撑如此大的访问量就需要考虑设计良好的架构，并很容易实现水平扩展。

架构

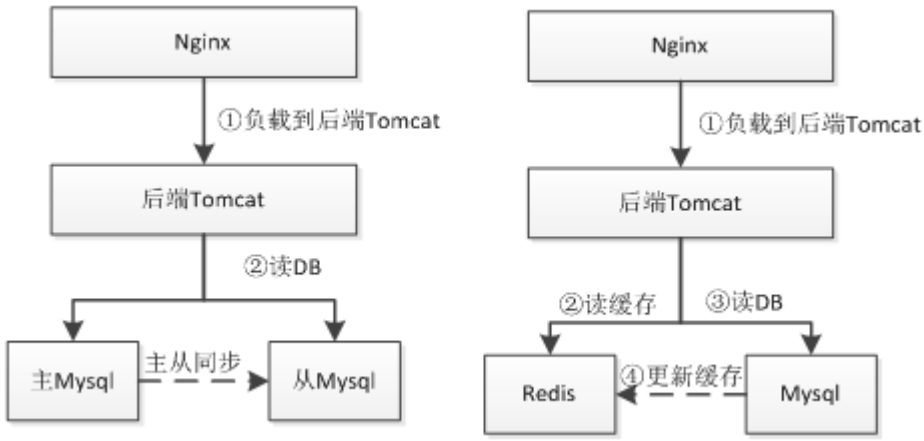
此处介绍下我曾使用过 Nginx+JavaEE 的架构。

单 DB 架构



早期架构可能就是 Nginx 直接 upstream 请求到后端 Tomcat，扩容时基本是增加新的 Tomcat 实例，然后通过 Nginx 负载均衡 upstream 过去。此时数据库还不是瓶颈。当访问量到一定级别，数据库的压力就上来了，此处单纯的靠单个数据库可能扛不住了，此时可以通过数据库的读写分离或加缓存来实现。

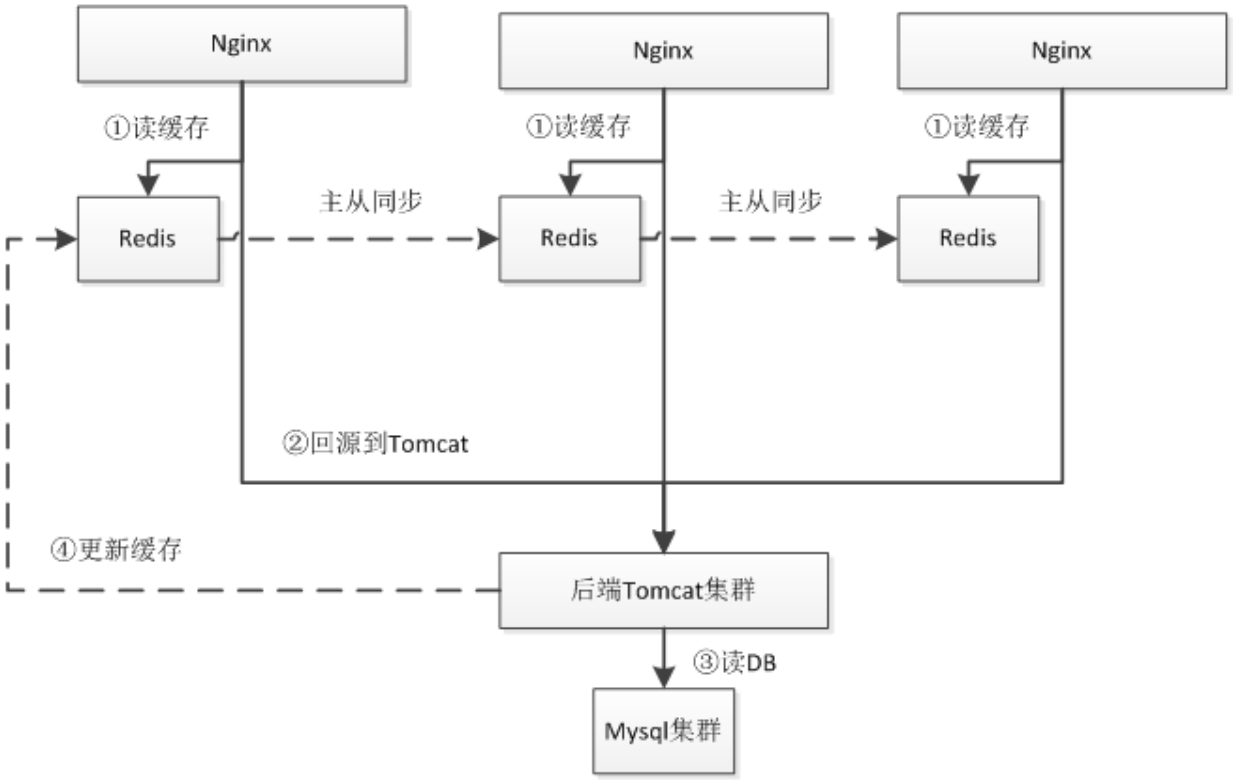
DB+Cache/ 数据库读写分离架构



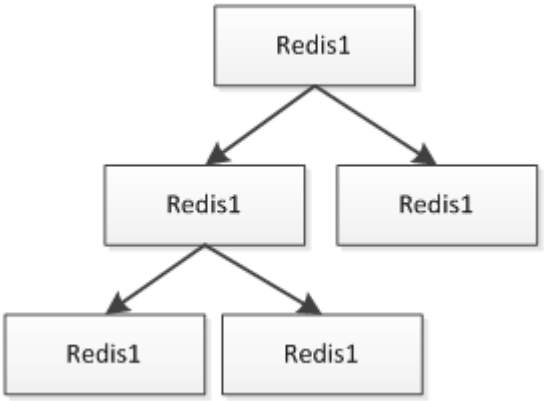
此时就通过使用如数据库读写分离或者 Redis 这种缓存来支撑更大的访问量。使用缓存这种架构会遇到的问题诸如缓存与数据库数据不同步造成数据不一致（一般设置过期时间），或者如 Redis 挂了，此时会直接命中数据库导致数据库压力过大；可以考虑 Redis 的主从或者一致性Hash 算法做分片的 Redis 集群；使用缓存这种架构要

求应用对数据的一致性要求不是很高；比如像下订单这种要落地的数据不适合用 Redis 存储，但是订单的读取可以使用缓存。

Nginx+Lua+Local Redis+Mysql 集群架构

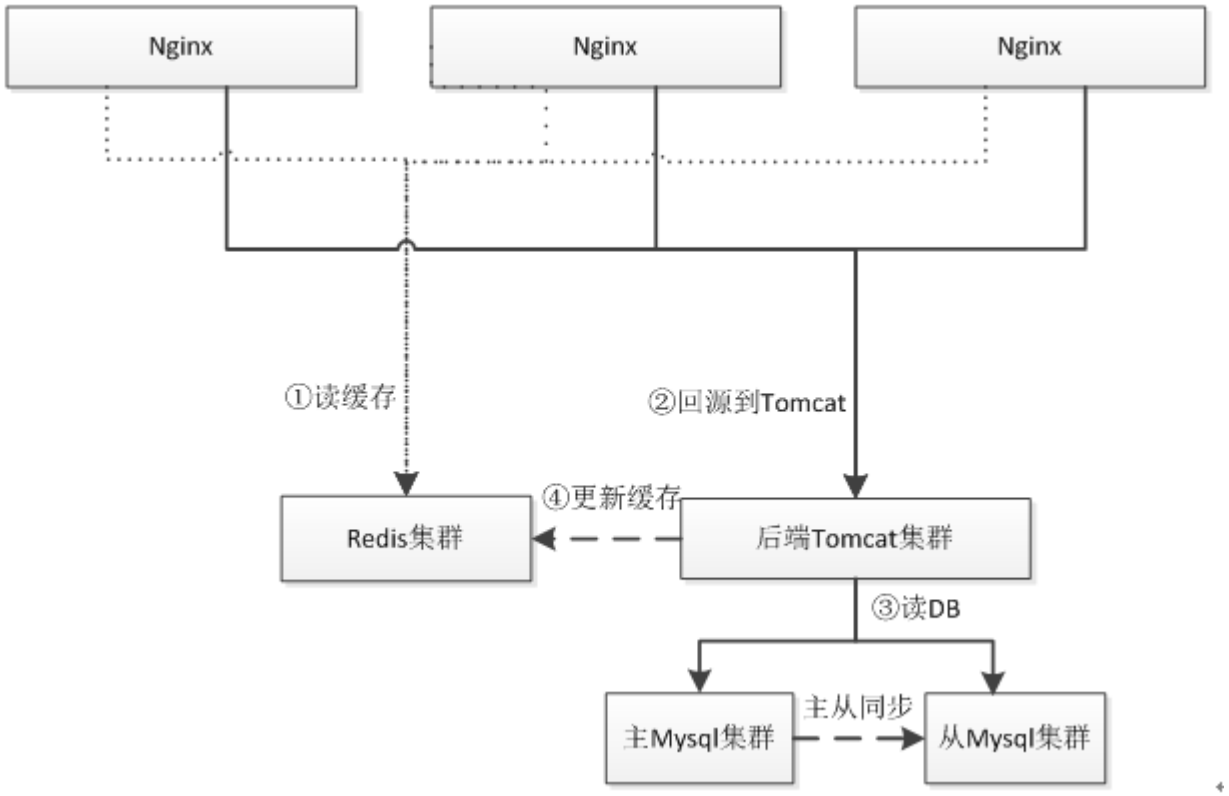


首先 Nginx 通过 Lua 读取本机 Redis 缓存，如果不命中才回源到后端 Tomcat 集群；后端Tomcat 集群再读取 Mysql 数据库。Redis 都是安装到和 Nginx 同一台服务器，Nginx 直接读本机可以减少网络延时。Redis 通过主从方式同步数据，Redis 主从一般采用树的方式实现：

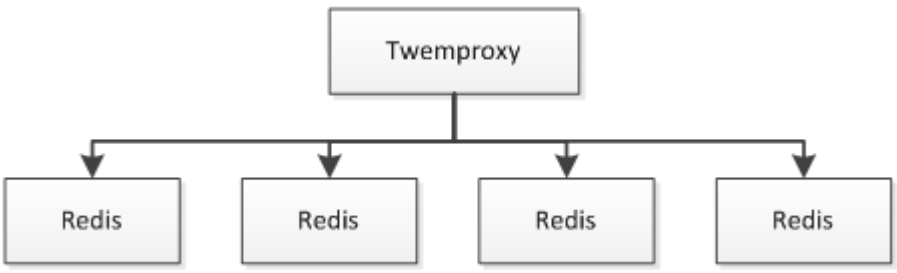


在叶子节点可以做 AOF 持久化，保证在主 Redis 挂时能进行恢复；此处假设对 Redis 很依赖的话，可以考虑多主 Redis 架构，而不是单主，来防止单主挂了时数据的不一致和击穿到后端Tomcat 集群。这种架构的缺点就是要求 Redis 实例数据量较小，如果单机内存不足以存储这么多数据，当然也可以通过如尾号为 1 的在 A 服务器，尾号为 2 的在 B 服务器这种方式实现；缺点也很明显，运维复杂、扩展性差。

Nginx+Lua+Redis 集群 +Mysql 集群架构

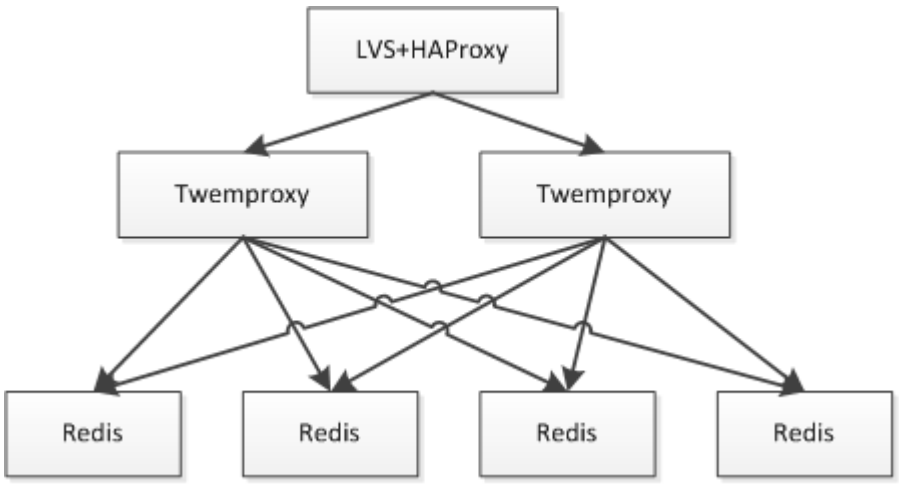


和之前架构不同的点是此时我们使用一致性 Hash 算法实现 Redis 集群而不是读本机 Redis，保证其中一台挂了，只有很少的数据会丢失，防止击穿到数据库。Redis 集群分片可以使用Twemproxy；如果 Tomcat 实例很多的话，此时就要考虑 Redis 和 Mysql 链接数问题，因为大部分 Redis/Mysql 客户端都是通过连接池实现，此时的链接数会成为瓶颈。一般方法是通过中间件来减少链接数。



Twemproxy 与 Redis 之间通过单链接交互，并 Twemproxy 实现分片逻辑；这样我们可以水平扩展更多的 Twemproxy 来增加链接数。

此时的问题就是 Twemproxy 实例众多，应用维护配置困难；此时就需要在之上做负载均衡，比如通过 LVS/HAProxy 实现 VIP（虚拟 IP ），可以做到切换对应用透明、故障自动转移；还可以通过实现内网 DNS 来做其负载均衡。



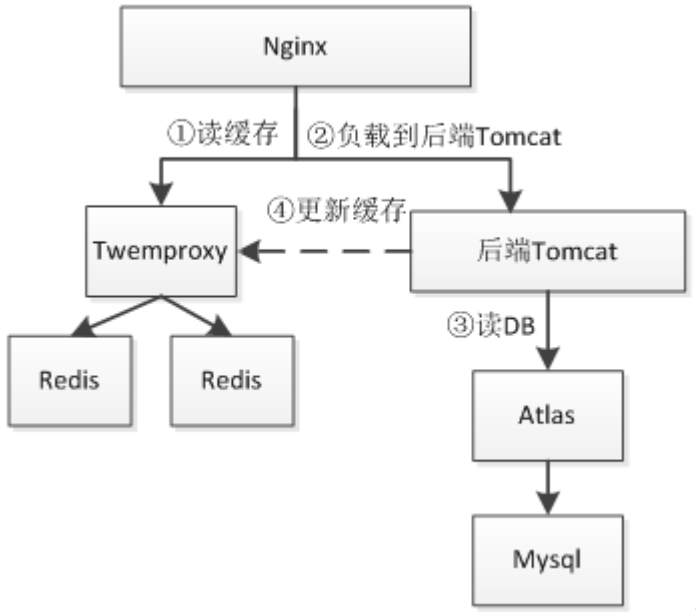
本文没有涉及 Nginx 之上是如何架构的，对于 Nginx、Redis、Mysql 等的负载均衡、资源的CDN 化不是本文关注的点，有兴趣可以参考

很早的 Taobao CDN 架构

Nginx/LVS/HAProxy 负载均衡软件的优缺点详解

实现

接下来我们来搭建一下第四种架构。



以获取如京东商品页广告词为例，如下图

假设京东有10亿商品，那么广告词极限情况是10亿；所以在设计时就要考虑：

1. 数据量，数据更新是否频繁且更新量是否很大；
2. 是K-V还是关系，是否需要批量获取，是否需要按照规则查询。

而对于本例，广告词更新量不会很大，每分钟可能在几万左右；而且是 K-V 的，其实适合使用关系存储；因为广告词是商家维护，因此后台查询需要知道这些商品是哪个商家的；而对于前台是不关心商家的，是 KV 存储，所以前台显示的可以放进如 Redis 中。 即存在两种设计：

1. 所有数据存储到 Mysql，然后热点数据加载到 Redis；
2. 关系存储到 Mysql，而数据存储到如SSDB这种持久化KV存储中。

基本数据结构：商品 ID、广告词、所属商家、开始时间、结束时间、是否有效。

后台逻辑

1. 商家登录后台；
2. 按照商家分页查询商家数据，此处要按照商品关键词或商品类目查询的话，需要走商品系统的搜索子系统，如通过 Solr或elasticsearch 实现搜索子系统；
3. 进行广告词的增删改查；
4. 增删改时可以直接更新 Redis 缓存或者只删除 Redis 缓存（第一次前台查询时写入缓存）；

前台逻辑

1. 首先 Nginx 通过 Lua 查询 Redis 缓存；
2. 查询不到的话回源到 Tomcat，Tomcat 读取数据库查询到数据，然后把最新的数据异步写入 Redis（一般设置过期时间，如5分钟）；此处设计时要考虑假设 Tomcat 读取 Mysql 的极限值是多少，然后设计降级开关，如假设每秒回源达到 100，则直接不查询 Mysql 而返回空的广告词来防止 Tomcat 应用雪崩。

为了简单，我们不进行后台的设计实现，只做前端的设计实现，此时数据结构我们简化为[商品ID、广告词]。另外有朋友可能看到了，可以直接把 Tomcat 部分干掉，通过 Lua 直接读取Mysql 进行回源实现。为了完整性此处我们还是做回源到 Tomcat 的设计，因为如果逻辑比较复杂的话或一些限制（比如使用 Java 特有协议的 RPC）还是通过 Java 去实现更方便一些。

项目搭建

项目部署目录结构。

Java 代码

```
/usr/chapter6
  redis_6660.conf
  redis_6661.conf
  nginx_chapter6.conf
  nutcracker.yml
  nutcracker.init
  webapp
WEB-INF
   lib
   classes
   web.xml
```

Redis+Twemproxy 配置

此处根据实际情况来决定 Redis 大小，此处我们已两个 Redis 实例（6660、6661），在Twemproxy 上通过一致性 Hash 做分片逻辑。

安装

之前已经介绍过 Redis 和 Twemproxy 的安装了。

**Redis配置redis_6660.conf和redis_6661.conf **

Java 代码

```
\#分别为6660 6661
port 6660
\#进程ID 分别改为redis_6660.pid redis_6661.pid
pidfile "/var/run/redis_6660.pid"
\#设置内存大小，根据实际情况设置，此处测试仅设置20mb
maxmemory 20mb
\#内存不足时，按照过期时间进行LRU删除
maxmemory-policy volatile-lru
\#Redis的过期算法不是精确的而是通过采样来算的，默认采样为3个，此处我们改成10
maxmemory-samples 10
\#不进行RDB持久化
save “”
\#不进行AOF持久化
appendonly no
```

将如上配置放到 redis_6660.conf 和 redis_6661.conf 配置文件最后即可，后边的配置会覆盖前边的。

Twemproxy配置nutcracker.yml

Java 代码

```
server1:
  listen: 127.0.0.1:1111
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  servers:
```

```
  - 127.0.0.1:6660:1 server1
  - 127.0.0.1:6661:1 server2
```

复制 nutcracker.init 到 /usr/chapter6 下，并修改配置文件为 /usr/chapter6/nutcracker.yml。

启动

Java 代码

```
nohup /usr/servers/redis-2.8.19/src/redis-server  /usr/chapter6/redis_6660.conf &
nohup /usr/servers/redis-2.8.19/src/redis-server  /usr/chapter6/redis_6661.conf &
/usr/chapter6/nutcracker.init start
ps -aux | grep -e redis  -e nutcracker
```

Mysql+Atlas 配置

Atlas 类似于 Twemproxy，是 Qihoo 360 基于 Mysql Proxy 开发的一个 Mysql 中间件，据称每天承载读写请求数达几十亿，可以实现分表、读写分离、数据库连接池等功能，缺点是没有实现跨库分表（分库）功能，需要在客户端使用分库逻辑。另一个选择是使用如阿里的 TDDL，它是在客户端完成之前说的功能。到底选择是在客户端还是在中间件根据实际情况选择。

此处我们不做 Mysql 的主从复制（读写分离），只做分库分表实现。

Mysql 初始化

为了测试我们此处分两个表。

Java 代码

```
CREATE DATABASE chapter6 DEFAULT CHARACTER SET utf8;
use chapter6;
CREATE TABLE  chapter6.ad_0(
    sku_id BIGINT,
    content VARCHAR(4000)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8;
CREATE TABLE  chapter6.ad_1
    sku_id BIGINT,
    content VARCHAR(4000)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8;
```

Atlas 安装

Java 代码

```
cd /usr/servers/
wget https://github.com/Qihoo360/Atlas/archive/2.2.1.tar.gz -O Atlas-2.2.1.tar.gz
tar -xvf Atlas-2.2.1.tar.gz
cd Atlas-2.2.1/
\#Atlas依赖mysql_config，如果没有可以通过如下方式安装
apt-get install libmysqlclient-dev
\#安装Lua依赖
wget http://www.lua.org/ftp/lua-5.1.5.tar.gz
tar -xvf lua-5.1.5.tar.gz
```

```
cd lua-5.1.5/
make linux && make install
\#安装glib依赖
apt-get install libglib2.0-dev
\#安装libevent依赖
apt-get install libevent
\#安装flex依赖
apt-get install flex
\#安装jemalloc依赖
apt-get install libjemalloc-dev
\#安装OpenSSL依赖
apt-get install openssl
apt-get install libssl-dev
 apt-get install libssl0.9.8

./configure --with-mysql=/usr/bin/mysql_config
./bootstrap.sh
make && make install
```

Atlas 配置

Java 代码

```
vim /usr/local/mysql-proxy/conf/chapter6.cnf
Java代码  收藏代码
[mysql-proxy]
\#Atlas代理的主库，多个之间逗号分隔
proxy-backend-addresses = 127.0.0.1:3306
\#Atlas代理的从库，多个之间逗号分隔，格式ip:port@weight，权重默认1
\#proxy-read-only-backend-addresses = 127.0.0.1:3306,127.0.0.1:3306
\#用户名/密码，密码使用/usr/servers/Atlas-2.2.1/script/encrypt 123456加密
pwds = root:/iZxz+0GRoA=
\#后端进程运行
daemon = true
\#开启monitor进程，当worker进程挂了自动重启
keepalive = true
\#工作线程数，对Atlas的性能有很大影响，可根据情况适当设置
event-threads = 64
\#日志级别
log-level = message
\#日志存放的路径
log-path = /usr/chapter6/
\#实例名称，用于同一台机器上多个Atlas实例间的区分
instance = test
```

```
\#监听的ip和port
proxy-address = 0.0.0.0:1112
\#监听的管理接口的ip和port
admin-address = 0.0.0.0:1113
\#管理接口的用户名
admin-username = admin
\#管理接口的密码
admin-password = 123456
\#分表逻辑
tables = chapter6.ad.sku_id.2
\#默认字符集
charset = utf8
```

因为本例没有做读写分离，所以读库 proxy-read-only-backend-addresses 没有配置。分表逻辑即：数据库名.表名.分表键.表的个数，分表的表名格式是 table_N，N 从 0 开始。

Atlas 启动/重启/停止

Java 代码

```
/usr/local/mysql-proxy/bin/mysql-proxyd chapter6 start
/usr/local/mysql-proxy/bin/mysql-proxyd chapter6 restart
/usr/local/mysql-proxy/bin/mysql-proxyd chapter6 stop
```

如上命令会自动到 /usr/local/mysql-proxy/conf 目录下查找 chapter6.cnf 配置文件。

Atlas 管理

通过如下命令进入管理接口

Java 代码

```
mysql -h127.0.0.1 -P1113  -uadmin -p123456
```

通过执行 SELECT * FROM help 查看帮助。还可以通过一些 SQL 进行服务器的动态添加/移除。

Atlas 客户端

通过如下命令进入客户端接口

Java 代码

```
mysql -h127.0.0.1 -P1112  -uroot -p123456
```

Java 代码

```
use chapter6;
insert into ad values(1 '测试1);
insert into ad values(2, '测试2');
insert into ad values(3 '测试3);
select * from ad where sku_id=1;
select * from ad where sku_id=2;
 #通过如下sql可以看到实际的分表结果

select * from ad_0;
select * from ad_1;
```

此时无法执行 select * from ad，需要使用如 “select * from ad where sku_id=1” 这种 SQL 进行查询；即需要带上 sku_id 且必须是相等比较；如果是范围或模糊是不可以的；如果想全部查询，只能挨着遍历所有表进行查询。即在客户端做查询-聚合。

此处实际的分表逻辑是按照商家进行分表，而不是按照商品编号，因为我们后台查询时是按照商家维度的，此处是为了测试才使用商品编号的。

到此基本的 Atlas 就介绍完了，更多内容请参考如下资料： Mysql 主从复制 http://369369.blog.51cto.com/319630/790921/ Mysql中间件介绍 http://www.guokr.com/blog/475765/ Atlas使用 http://www.0550go.com/database/mysql/mysql-atlas.html Atlas文档 [https://github.com/Qihoo360/Atlas/blob/master/README_ZH.mdhref="https://github.com/Qihoo360/Atlas/blob/master/README_ZH.md")

Java+Tomcat 安装

Java 安装

Java 代码

```
cd /usr/servers/
\#首先到如下网站下载JDK
\#http://www.oracle.com/technetwork/cn/java/javase/downloads/jdk7-downloads-1880260.html
\#本文下载的是 jdk-7u75-linux-x64.tar.gz。
tar -xvf jdk-7u75-linux-x64.tar.gz
vim ~/.bashrc
在文件最后添加如下环境变量
export JAVA_HOME=/usr/servers/jdk1.7.0_75/
export PATH=$JAVA_HOME/bin:$JAVA_HOME/jre/bin:$PATH
export CLASSPATH=$CLASSPATH:.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib

\#使环境变量生效
source ~/.bashrc
```

Tomcat 安装

Java 代码

```
cd /usr/servers/
wget http://ftp.cuhk.edu.hk/pub/packages/apache.org/tomcat/tomcat-7/v7.0.59/bin/apache-tomcat-7.0.59.tar.gz
tar -xvf apache-tomcat-7.0.59.tar.gz
cd apache-tomcat-7.0.59/
\#启动
/usr/servers/apache-tomcat-7.0.59/bin/startup.sh
\#停止
/usr/servers/apache-tomcat-7.0.59/bin/shutdown.sh
\#删除tomcat默认的webapp
rm -r apache-tomcat-7.0.59/webapps/*
\#通过Catalina目录发布web应用
cd apache-tomcat-7.0.59/conf/Catalina/localhost/
vim ROOT.xml
```

ROOT.xml

Java 代码

```
<!-- 访问路径是根，web应用所属目录为/usr/chapter6/webapp -->
<Context path="" docBase="/usr/chapter6/webapp"></Context>
```

Java **代码 **

```
\#创建一个静态文件随便添加点内容
vim /usr/chapter6/webapp/index.html
\#启动
/usr/servers/apache-tomcat-7.0.59/bin/startup.sh
```

访问如 `http://192.168.1.2:8080/index.html` 能处理内容说明配置成功。

Java **代码 **

```
\#变更目录结构
cd /usr/servers/
mv apache-tomcat-7.0.59 tomcat-server1
\#此处我们创建两个tomcat实例
cp  -r tomcat-server1 tomcat-server2
vim tomcat-server2/conf/server.xml
```

Java **代码 **

```
\#如下端口进行变更
8080--->8090
8005--->8006
```

启动两个 Tomcat

Java **代码 **

```
/usr/servers/tomcat-server1/bin/startup.sh
/usr/servers/tomcat-server2/bin/startup.sh
```

分别访问，如果能正常访问说明配置正常。

- `http://192.168.1.2:8080/index.html`
- `http://192.168.1.2:8090/index.html`

如上步骤使我们在一个服务器上能启动两个 tomcat 实例，这样的好处是我们可以做本机的 Tomcat 负载均衡，假设一个 tomcat 重启时另一个是可以工作的，从而不至于不给用户返回响应。

Java+Tomcat 逻辑开发

搭建项目

我们使用 Maven 搭建 Web 项目，Maven 知识请自行学习。

项目依赖

本文将最小化依赖，即仅依赖我们需要的 servlet、mysql、druid、jedis。

Java 代码

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.27</version>
  </dependency>
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.5</version>
  </dependency>
  <dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.5.2</version>
  </dependency>
</dependencies>
```

核心代码

`com.github.zhangkaitao.chapter6.servlet.AdServlet`

Java 代码

```
public class AdServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String idStr = req.getParameter("id");
        Long id = Long.valueOf(idStr);
        //1、读取Mysql获取数据
        String content = null;
        try {
            content = queryDB(id);
        } catch (Exception e) {
            e.printStackTrace();
            resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
            return;
        }
        if(content != null) {
            //2.1、如果获取到，异步写Redis
            asyncSetToRedis(idStr, content);
            //2.2、如果获取到，把响应内容返回
            resp.setCharacterEncoding("UTF-8");
            resp.getWriter().write(content);
        } else {
            //2.3、如果获取不到，返回404状态码
            resp.setStatus(HttpServletResponse.SC_NOT_FOUND);
        }
    }

    private DruidDataSource datasource = null;
    private JedisPool jedisPool = null;

    {
        datasource = new DruidDataSource();
        datasource.setUrl("jdbc:mysql://127.0.0.1:1112/chapter6?useUnicode=true&characterEncoding=utf-8&autoReconne
        datasource.setUsername("root");
        datasource.setPassword("123456");
        datasource.setMaxActive(100);

        GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();
        poolConfig.setMaxTotal(100);
        jedisPool = new JedisPool(poolConfig, "127.0.0.1", 1111);
    }

    private String queryDB(Long id) throws Exception {
        Connection conn = null;
```

```
        try {
            conn = datasource.getConnection();
            String sql = "select content from ad where sku_id = ?";
            PreparedStatement psst = conn.prepareStatement(sql);
            psst.setLong(1, id);
            ResultSet rs = psst.executeQuery();
            String content = null;
            if(rs.next()) {
                content = rs.getString("content");
            }
            rs.close();
            psst.close();
            return content;
        } catch (Exception e) {
            throw e;
        } finally {
            if(conn != null) {
                conn.close();
            }
        }
    }

    private ExecutorService executorService = Executors.newFixedThreadPool(10);
    private void asyncSetToRedis(final String id, final String content) {
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                Jedis jedis = null;
                try {
                    jedis = jedisPool.getResource();
                    jedis.setex(id, 5 * 60, content);//5分钟
                } catch (Exception e) {
                    e.printStackTrace();
                    jedisPool.returnBrokenResource(jedis);
                } finally {
                    jedisPool.returnResource(jedis);
                }

            }
        });
    }
}
```

整个逻辑比较简单，此处更新缓存一般使用异步方式去更新，这样不会阻塞主线程；另外此处可以考虑走 Servlet 异步化来提示吞吐量。

web.xml 配置

Java 代码

```
<servlet>
    <servlet-name>adServlet</servlet-name>
    <servlet-class>com.github.zhangkaitao.chapter6.servlet.AdServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>adServlet</servlet-name>
    <url-pattern>/ad</url-pattern>
</servlet-mapping>
```

打 WAR 包

Java 代码

```
cd D:\workspace\chapter6
mvn clean package
此处使用maven命令打包，比如本例将得到chapter6.war，然后将其上传到服务器的/usr/chapter6/webapp，然后通过unzip chapt
```

测试

启动 Tomcat 实例，分别访问如下地址将看到广告内容：

Java 代码

```
http://192.168.1.2:8080/ad?id=1
http://192.168.1.2:8090/ad?id=1
```

nginx 配置

vim /usr/chapter6/nginx_chapter6.conf

Java 代码

```
upstream backend {
    server 127.0.0.1:8080 max_fails=5 fail_timeout=10s weight=1 backup=false;
    server 127.0.0.1:8090 max_fails=5 fail_timeout=10s weight=1 backup=false;
    check interval=3000 rise=1 fall=2 timeout=5000 type=tcp default_down=false;
    keepalive 100;
}
server {
```

```
    listen       80;
    server_name  _;

    location ~ /backend/(.*) {
        keepalive_timeout   30s;
        keepalive_requests  100;

        rewrite /backend(/.*) $1 break;
        #之后该服务将只有内部使用，ngx.location.capture
        proxy_pass_request_headers off;
        #more_clear_input_headers Accept-Encoding;
        proxy_next_upstream error timeout;
        proxy_pass http://backend;
    }
}
```

upstream 配置：http://nginx.org/cn/docs/http/ngx_http_upstream_module.html。
server：指定上游到的服务器， weight：权重，权重可以认为负载均衡的比例； fail_timeout+max_fails：在指定时间内失败多少次认为服务器不可用，通过proxy_next_upstream来判断是否失败。
check：ngx_http_upstream_check_module模块，上游服务器的健康检查，interval：发送心跳包的时间间隔，rise：连续成功rise次数则认为服务器up，fall：连续失败fall次则认为服务器down，timeout：上游服务器请求超时时间，type：心跳检测类型（比如此处使用 tcp ）更多配置请参考 https://github.com/yaoweibin/ nginx_upstream_check_module 和 http://tengine.taobao.org/document_cn/http_upstream_check_cn.html。
keepalive：用来支持 upstream server http keepalive 特性(需要上游服务器支持，比如 tomcat)。默认的负载均衡算法是 round-robin，还可以根据 ip、url 等做 hash 来做负载均衡。更多资料请参考官方文档。

tomcat keepalive 配置： http://tomcat.apache.org/tomcat-7.0-doc/config/http.html。 maxKeepAliveRequests：默认100； keepAliveTimeout：默认等于 connectionTimeout，默认 60 秒；

location proxy 配置：http://nginx.org/cn/docs/http/ngx_http_proxy_module.html。 rewrite：将当前请求的 url 重写，如我们请求时是 /backend/ad，则重写后是 /ad。 proxy_pass：将整个请求转发到上游服务器。 proxy_next_upstream：什么情况认为当前 upstream server 失败，需要 next upstream，默认是连接失败/超时，负载均衡参数。 proxy_pass_request_headers：之前已经介绍过了，两个原因：1、假设上游服务器不需要请求头则没必要传输请求头；2、ngx.location.capture时防止 gzip 乱码（也可以使用more_clear_input_headers 配置）。 keepalive：keepalive_timeout：keepalive 超时设置，keepalive_requests：长连接数量。此处的 keepalive（别人访问该 location 时的长连接）和 upstream keepalive（nginx 与上游服务器的长连接）是不一样的；此处注意，如果您的服务是面向客户的，而且是单个动态内容就没必要使用长连接了。

```
vim /usr/servers/nginx/conf/nginx.conf
```

Java 代码

```
include /usr/chapter6/nginx_chapter6.conf;
\#为了方便测试，注释掉example.conf
\#include /usr/example/example.conf;
```

重启 nginx

/usr/servers/nginx/sbin/nginx -s reload

访问如 192.168.1.2/backend/ad?id=1 即看到结果。可以 kill 掉一个 tomcat，可以看到服务还是正常的。

vim /usr/chapter6/nginx_chapter6.conf

Java 代码

```
location ~ /backend/(.*) {
    internal;
    keepalive_timeout   30s;
    keepalive_requests  1000;
    #支持keep-alive
    proxy_http_version 1.1;
    proxy_set_header Connection "";

    rewrite /backend(/.*) $1 break;
    proxy_pass_request_headers off;
    #more_clear_input_headers Accept-Encoding;
    proxy_next_upstream error timeout;
    proxy_pass http://backend;
}
```

加上 internal，表示只有内部使用该服务。

Nginx+Lua 逻辑开发

核心代码

/usr/chapter6/ad.lua

Java 代码

```
local redis = require("resty.redis")
local cjson = require("cjson")
local cjson_encode = cjson.encode
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local ngx_exit = ngx.exit
local ngx_print = ngx.print
local ngx_re_match = ngx.re.match
local ngx_var = ngx.var

local function close_redis(red)
    if not red then
        return
    end
    --释放连接(连接池实现)
    local pool_max_idle_time = 10000 --毫秒
    local pool_size = 100 --连接池大小
    local ok, err = red:set_keepalive(pool_max_idle_time, pool_size)

    if not ok then
        ngx_log(ngx_ERR, "set redis keepalive error : ", err)
    end
end
local function read_redis(id)
    local red = redis:new()
    red:set_timeout(1000)
    local ip = "127.0.0.1"
    local port = 1111
    local ok, err = red:connect(ip, port)
    if not ok then
        ngx_log(ngx_ERR, "connect to redis error : ", err)
        return close_redis(red)
    end

    local resp, err = red:get(id)
```

```
    if not resp then
        ngx_log(ngx_ERR, "get redis content error : ", err)
        return close_redis(red)
    end
        --得到的数据为空处理
    if resp == ngx.null then
        resp = nil
    end
    close_redis(red)

    return resp
end

local function read_http(id)
    local resp = ngx.location.capture("/backend/ad", {
        method = ngx.HTTP_GET,
        args = {id = id}
    })

    if not resp then
        ngx_log(ngx_ERR, "request error :", err)
        return
    end

    if resp.status ~= 200 then
        ngx_log(ngx_ERR, "request error, status :", resp.status)
        return
    end

    return resp.body
end


--获取id
local id = ngx_var.id

--从redis获取
local content = read_redis(id)

--如果redis没有，回源到tomcat
if not content then
   ngx_log(ngx_ERR, "redis not found content, back to http, id : ", id)
    content = read_http(id)
end
```

```
--如果还没有返回404
if not content then
   ngx_log(ngx_ERR, "http not found content, id : ", id)
   return ngx_exit(404)
end

--输出内容
ngx.print("show_ad(")
ngx_print(cjson_encode({content = content}))
ngx.print(")")
```

将可能经常用的变量做成局部变量，如 local ngx_print = ngx.print；使用 jsonp 方式输出，此处我们可以将请求 url 限定为 /ad/id 方式，这样的好处是1、可以尽可能早的识别无效请求；2、可以走 nginx 缓存 /CDN 缓存，缓存的 key 就是 URL，而不带任何参数，防止那些通过加随机数穿透缓存；3、jsonp 使用固定的回调函数 show_ad()，或者限定几个固定的回调来减少缓存的版本。

vim /usr/chapter6/nginx_chapter6.conf

Java 代码

```
location ~ ^/ad/(\d+)$ {
    default_type 'text/html';
    charset utf-8;
    lua_code_cache on;
    set $id $1;
    content_by_lua_file /usr/chapter6/ad.lua;
}
```

重启 nginx

Java 代码

```
/usr/servers/nginx/sbin/nginx -s reload
```

访问如 `http://192.168.1.2/ad/1` 即可得到结果。而且注意观察日志，第一次访问时不命中Redis，回源到 Tomcat；第二次请求时就会命中 Redis 了。

第一次访问时将看到 /usr/servers/nginx/logs/error.log 输出类似如下的内容，而第二次请求相同的 url 不再有如下内容：

Java 代码

```
redis not found content, back to http, id : 2
```

到此整个架构就介绍完了，此处可以直接不使用 Tomcat，而是 Lua 直连 Mysql 做回源处理；另外本文只是介绍了大体架构，还有更多业务及运维上的细节需要在实际应用中根据自己的场景自己摸索。后续如使用 LVS/HA Proxy 做负载均衡、使用 CDN 等可以查找资料学习。

9

Web 开发实战2——商品详情页

本章以京东商品详情页为例，京东商品详情页虽然仅是单个页面，但是其数据聚合源是非常多的，除了一些实时性要求比较高的如价格、库存、服务支持等通过 AJAX 异步加载加载之外，其他的数据都是在后端做数据聚合然后拼装网页模板的。 http://item.jd.com/1217499.html

如图所示，商品页主要包括商品基本信息（基本信息、图片列表、颜色/尺码关系、扩展属性、规格参数、包装清单、售后保障等）、商品介绍、其他信息（分类、品牌、店铺【第三方卖家】、店内分类【第三方卖家】、同类相关品牌）。更多细节此处就不阐述了。

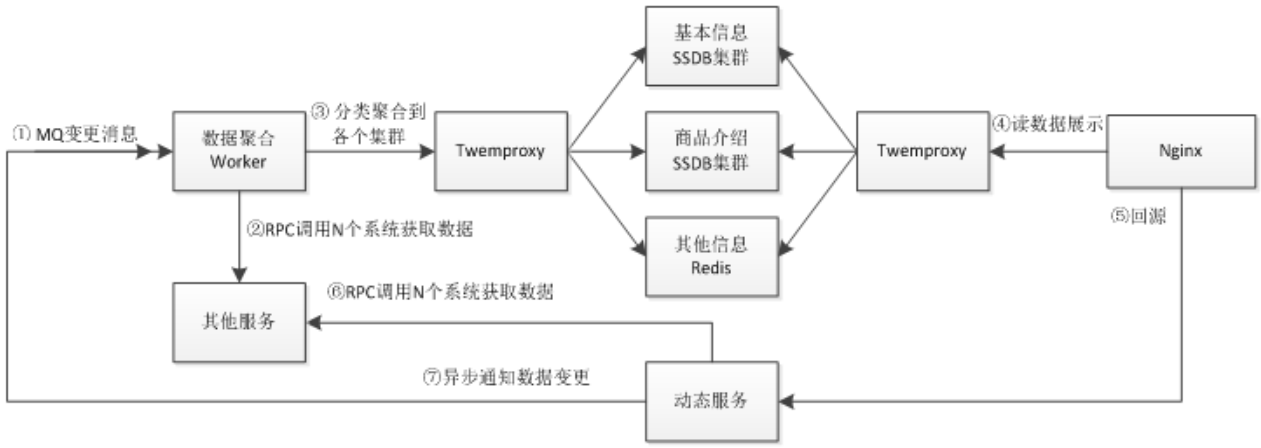
整个京东有数亿商品，如果每次动态获取如上内容进行模板拼装，数据来源之多足以造成性能无法满足要求；最初的解决方案是生成静态页，但是静态页的最大的问题：1、无法迅速响应页面需求变更；2、很难做多版本线上对比测试。如上两个因素足以制约商品页的多样化发展，因此静态化技术不是很好的方案。

通过分析，数据主要分为四种：商品页基本信息、商品介绍（异步加载）、其他信息（分类、品牌、店铺等）、其他需要实时展示的数据（价格、库存等）。而其他信息如分类、品牌、店铺是非常少的，完全可以放到一个占用内存很小的 Redis 中存储；而商品基本信息我们可以借鉴静态化技术将数据做聚合存储，这样的好处是

数据是原子的，而模板是随时可变的，吸收了静态页聚合的优点，弥补了静态页的多版本缺点；另外一个非常严重的问题就是严重依赖这些相关系统，如果它们挂了或响应慢则商品页就挂了或响应慢；商品介绍我们也通过 AJAX 技术惰性加载（因为是第二屏，只有当用户滚动鼠标到该屏时才显示）；而实时展示数据通过 AJAX 技术做异步加载；因此我们可以做如下设计：

1. 接收商品变更消息，做商品基本信息的聚合，即从多个数据源获取商品相关信息如图片列表、颜色尺码、规格参数、扩展属性等等，聚合为一个大的 JSON 数据做成数据闭环，以 key-value存储；因为是闭环，即使依赖的系统挂了我们商品页还是能继续服务的，对商品页不会造成任何影响；
2. 接收商品介绍变更消息，存储商品介绍信息；
3. 介绍其他信息变更消息，存储其他信息。

整个架构如下图所示：



技术选型

MQ 可以使用如 Apache ActiveMQ； Worker/ 动态服务可以通过如 Java 技术实现； RPC 可以选择如 alibaba Dubbo； KV 持久化存储可以选择 SSDB（如果使用 SSD 盘则可以选择 SSDB+RocksDB 引擎）或者 ARDB（ LMDB 引擎版）； 缓存使用 Redis； SSDB/Redis 分片使用如 Twemproxy，这样不管使用 Java 还是 Nginx+Lua，它们都不关心分片逻辑； 前端模板拼装使用 Nginx+Lua； 数据集群数据存储的机器可以采用 RAID 技术或者主从模式防止单点故障； 因为数据变更不频繁，可以考虑 SSD 替代机械硬盘。

核心流程

1. 首先我们监听商品数据变更消息；
2. 接收到消息后，数据聚合 Worker 通过 RPC 调用相关系统获取所有要展示的数据，此处获取数据的来源可能非常多而且响应速度完全受制于这些系统，可能耗时几百毫秒甚至上秒的时间；
3. 将数据聚合为 JSON 串存储到相关数据集群；
4. 前端 Nginx 通过 Lua 获取相关集群的数据进行展示；商品页需要获取基本信息+其他信息进行模板拼装，即拼装模板仅需要两次调用（另外因为其他信息数据量少且对一致性要求不高，因此我们完全可以缓存到 Nginx 本地全局内存，这样可以减少远程调用提高性能）；当页面滚动到商品介绍页面时异步调用商品介绍服务获取数据；
5. 如果从聚合的 SSDB 集群 /Redis 中获取不到相关数据；则回源到动态服务通过 RPC 调用相关系统获取所有要展示的数据返回（此处可以做限流处理，因为如果大量请求过来的话可能导致服务雪崩，需要采取保护措施），此处的逻辑和数据聚合 Worker 完全一样；然后发送 MQ 通知数据变更，这样下次访问时就可以从聚合的 SSDB 集群 /Redis 中获取数据了。

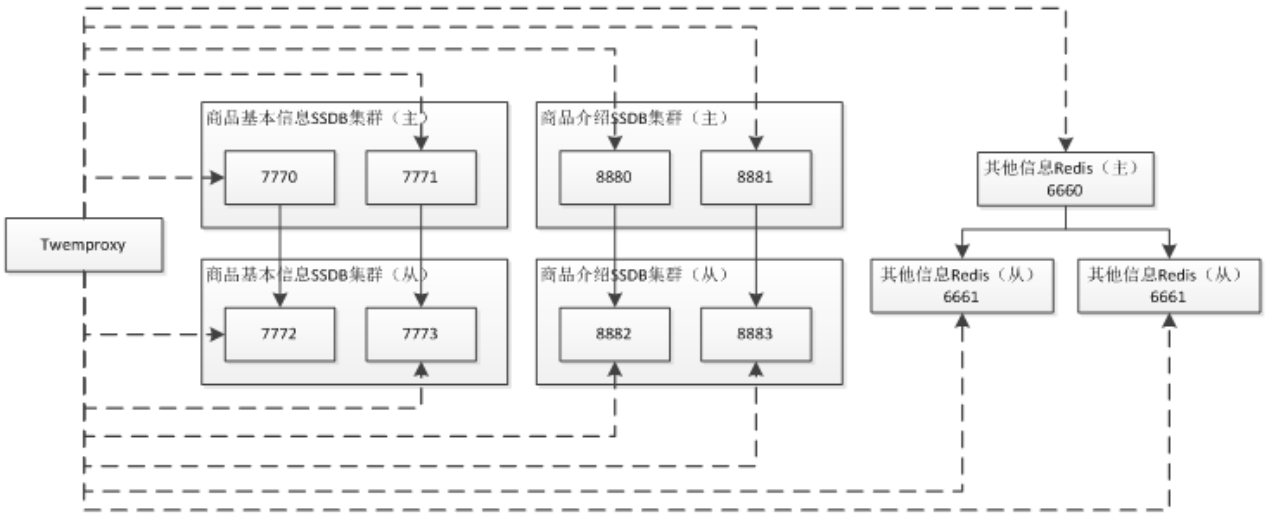
基本流程如上所述，主要分为 Worker、动态服务、数据存储和前端展示；因为系统非常复杂，只介绍动态服务和前端展示、数据存储架构；Worker 部分不做实现。

项目搭建

项目部署目录结构。

```
/usr/chapter7
  ssdb_basic_7770.conf
  ssdb_basic_7771.conf
  ssdb_basic_7772.conf
  ssdb_basic_7773.conf
  ssdb_desc_8880.conf
  ssdb_desc_8881.conf
  ssdb_desc_8882.conf
  ssdb_desc_8883.conf
  redis_other_6660.conf
  redis_other_6661.conf
  nginx_chapter7.conf
  nutcracker.yml
  nutcracker.init
  item.html
  header.html
  footer.html
  item.lua
  desc.lua
  lualib
    item.lua
    item
      common.lua
  webapp
WEB-INF
   lib
   classes
   web.xml
```

数据存储实现



整体架构为主从模式，写数据到主集群，读数据从从集群读取数据，这样当一个集群不足以支撑流量时可以使用更多的集群来支撑更多的访问量；集群分片使用 Twemproxy 实现。

商品基本信息 SSDB 集群配置

vim /usr/chapter7/ssdb_basic_7770.conf \

Java 代码

```
work_dir = /usr/data/ssdb_7770
pidfile = /usr/data/ssdb_7770.pid

server:
        ip: 0.0.0.0
        port: 7770
        allow: 127.0.0.1
        allow: 192.168

replication:
        binlog: yes
        sync_speed: -1
        slaveof:
logger:
        level: error
```

```
    output: /usr/data/ssdb_7770.log
    rotate:
        size: 1000000000

leveldb:
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

vim /usr/chapter7/ssdb_basic_7771.conf

Java 代码

```
work_dir = /usr/data/ssdb_7771
pidfile = /usr/data/ssdb_7771.pid

server:
    ip: 0.0.0.0
    port: 7771
    allow: 127.0.0.1
    allow: 192.168

replication:
    binlog: yes
    sync_speed: -1
    slaveof:
logger:
    level: error
    output: /usr/data/ssdb_7771.log
    rotate:
        size: 1000000000

leveldb:
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

vim /usr/chapter7/ssdb_basic_7772.conf

Java 代码

```
work_dir = /usr/data/ssdb_7772
pidfile = /usr/data/ssdb_7772.pid

server:
    ip: 0.0.0.0
    port: 7772
    allow: 127.0.0.1
    allow: 192.168

replication:
    binlog: yes
    sync_speed: -1
    slaveof:
        type: sync
        ip: 127.0.0.1
        port: 7770

logger:
    level: error
    output: /usr/data/ssdb_7772.log
    rotate:
        size: 1000000000

leveldb:
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

vim /usr/chapter7/ssdb_basic_7773.conf

Java 代码

```
work_dir = /usr/data/ssdb_7773
pidfile = /usr/data/ssdb_7773.pid

server:
    ip: 0.0.0.0
    port: 7773
    allow: 127.0.0.1
    allow: 192.168

replication:
    binlog: yes
    sync_speed: -1
```

```
    slaveof:
        type: sync
        ip: 127.0.0.1
        port: 7771

logger:
    level: error
    output: /usr/data/ssdb_7773.log
    rotate:
        size: 1000000000

leveldb:
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

配置文件使用 Tab 而不是空格做缩排，（复制到配置文件后请把空格替换为 Tab ）。主从关系：7770(主)-->7772(从)，7771(主)--->7773(从)；配置文件如何配置请参考 [https://github.com/ideawu/ssdb-docs/blob/master/src/zh_cn/config.mdhref="https://github.com/ideawu/ssdb-docs/blob/master/src/zh_cn/config.md")。

创建工作目录

Java 代码

```
mkdir -p /usr/data/ssdb_7770
mkdir -p /usr/data/ssdb_7771
mkdir -p /usr/data/ssdb_7772
mkdir -p /usr/data/ssdb_7773
```

启动

Java 代码

```
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_basic_7770.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_basic_7771.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_basic_7772.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_basic_7773.conf &
```

通过 ps -aux | grep ssdb 命令看是否启动了，tail -f nohup.out 查看错误信息。

商品介绍 SSDB 集群配置

vim /usr/chapter7/ssdb_desc_8880.conf

Java 代码

```
work_dir = /usr/data/ssdb_8880
pidfile = /usr/data/ssdb8880.pid

server:
        ip: 0.0.0.0
        port: 8880
        allow: 127.0.0.1
        allow: 192.168

replication:
        binlog: yes
        sync_speed: -1
        slaveof:
logger:
        level: error
        output: /usr/data/ssdb_8880.log
        rotate:
                size: 1000000000

leveldb:
        cache_size: 500
        block_size: 32
        write_buffer_size: 64
        compaction_speed: 1000
        compression: yes
```

vim /usr/chapter7/ssdb_desc_8881.conf

Java 代码

```
work_dir = /usr/data/ssdb_8881
pidfile = /usr/data/ssdb8881.pid

server:
        ip: 0.0.0.0
        port: 8881
        allow: 127.0.0.1
        allow: 192.168
```

```
logger:
    level: error
    output: /usr/data/ssdb_8881.log
    rotate:
        size: 1000000000

leveldb:
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

vim /usr/chapter7/ssdb_desc_8882.conf

Java 代码

```
work_dir = /usr/data/ssdb_8882
pidfile = /usr/data/ssdb_8882.pid

server:
    ip: 0.0.0.0
    port: 8882
    allow: 127.0.0.1
    allow: 192.168

replication:
    binlog: yes
    sync_speed: -1
    slaveof:
replication:
    binlog: yes
    sync_speed: -1
    slaveof:
        type: sync
        ip: 127.0.0.1
        port: 8880

logger:
    level: error
    output: /usr/data/ssdb_8882.log
    rotate:
        size: 1000000000

leveldb:
```

```
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

vim /usr/chapter7/ssdb_desc_8883.conf

Java 代码

```
work_dir = /usr/data/ssdb_8883
pidfile = /usr/data/ssdb_8883.pid

server:
    ip: 0.0.0.0
    port: 8883
    allow: 127.0.0.1
    allow: 192.168

replication:
    binlog: yes
    sync_speed: -1
    slaveof:
        type: sync
        ip: 127.0.0.1
        port: 8881

logger:
    level: error
    output: /usr/data/ssdb_8883.log
    rotate:
        size: 1000000000

leveldb:
    cache_size: 500
    block_size: 32
    write_buffer_size: 64
    compaction_speed: 1000
    compression: yes
```

配置文件使用 Tab 而不是空格做缩排（复制到配置文件后请把空格替换为 Tab ）。主从关系：7770(主)-->7772(从)，7771(主)--->7773(从)；配置文件如何配置请参考 [https://github.com/ideawu/ssdb-docs/blob/master/src/zh_cn/config.mdhref="https://github.com/ideawu/ssdb-docs/blob/master/src/zh_cn/config.md")。

创建工作目录

Java 代码

```
mkdir -p /usr/data/ssdb_888{0,1,2,3}
```

启动

Java 代码

```
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_desc_8880.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_desc_8881.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_desc_8882.conf &
nohup /usr/servers/ssdb-1.8.0/ssdb-server  /usr/chapter7/ssdb_desc_8883.conf &
```

通过 ps -aux | grep ssdb 命令看是否启动了，tail -f nohup.out 查看错误信息。

其他信息 Redis 配置

vim /usr/chapter7/redis_6660.conf

Java 代码

```
port 6660
pidfile "/var/run/redis_6660.pid"
\#设置内存大小，根据实际情况设置，此处测试仅设置20mb
maxmemory 20mb
\#内存不足时，所有KEY按照LRU算法删除
maxmemory-policy allkeys-lru
\#Redis的过期算法不是精确的而是通过采样来算的，默认采样为3个，此处我们改成10
maxmemory-samples 10
\#不进行RDB持久化
save “”
\#不进行AOF持久化
appendonly no
```

vim /usr/chapter7/redis_6661.conf

Java 代码

```
port 6661
pidfile "/var/run/redis_6661.pid"
\#设置内存大小，根据实际情况设置，此处测试仅设置20mb
maxmemory 20mb
\#内存不足时，所有KEY按照LRU算法进行删除
maxmemory-policy allkeys-lru
\#Redis的过期算法不是精确的而是通过采样来算的，默认采样为3个，此处我们改成10
maxmemory-samples 10
\#不进行RDB持久化
save “”
\#不进行AOF持久化
appendonly no
\#主从
slaveof 127.0.0.1 6660
```

vim /usr/chapter7/redis_6662.conf

Java 代码

```
port 6662
pidfile "/var/run/redis_6662.pid"
\#设置内存大小，根据实际情况设置，此处测试仅设置20mb
maxmemory 20mb
\#内存不足时，所有KEY按照LRU算法进行删除
maxmemory-policy allkeys-lru
\#Redis的过期算法不是精确的而是通过采样来算的，默认采样为3个，此处我们改成10
maxmemory-samples 10
\#不进行RDB持久化
save “”
\#不进行AOF持久化
appendonly no
\#主从
slaveof 127.0.0.1 6660
```

如上配置放到配置文件最末尾即可；此处内存不足时的驱逐算法为所有 KEY 按照 LRU 进行删除（实际是内存基本上不会遇到满的情况）；主从关系：6660(主)-->6661(从)和6660(主)-->6662(从)。

启动

Java 代码

```
nohup /usr/servers/redis-2.8.19/src/redis-server /usr/chapter7/redis_6660.conf &
nohup /usr/servers/redis-2.8.19/src/redis-server /usr/chapter7/redis_6661.conf &
nohup /usr/servers/redis-2.8.19/src/redis-server /usr/chapter7/redis_6662.conf &
```

通过 ps -aux | grep redis 命令看是否启动了，tail -f nohup.out 查看错误信息。

测试

测试时在主 SSDB/Redis 中写入数据，然后从从 SSDB/Redis 能读取到数据即表示配置主从成功。

测试商品基本信息 SSDB 集群

Java 代码

```
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli  -p 7770
127.0.0.1:7770> set i 1
OK
127.0.0.1:7770>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli  -p 7772
127.0.0.1:7772> get i
"1"
```

测试商品介绍 SSDB 集群

Java 代码

```
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli  -p 8880
127.0.0.1:8880> set i 1
OK
127.0.0.1:8880>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli  -p 8882
127.0.0.1:8882> get i
"1"
```

测试其他信息集群

Java 代码

```
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli  -p 6660
127.0.0.1:6660> set i 1
OK
127.0.0.1:6660> get i
"1"
127.0.0.1:6660>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli  -p 6661
127.0.0.1:6661> get i
"1"
```

Twemproxy 配置

vim /usr/chapter7/nutcracker.yml

Java 代码

```
basic_master:
  listen: 127.0.0.1:1111
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:7770:1 server1
   - 127.0.0.1:7771:1 server2

basic_slave:
  listen: 127.0.0.1:1112
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:7772:1 server1
   - 127.0.0.1:7773:1 server2


desc_master:
  listen: 127.0.0.1:1113
  hash: fnv1a_64
  distribution: ketama
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:8880:1 server1
   - 127.0.0.1:8881:1 server2

desc_slave:
  listen: 127.0.0.1:1114
  hash: fnv1a_64
```

```
  distribution: ketama
  redis: true
  timeout: 1000
  servers:
   - 127.0.0.1:8882:1 server1
   - 127.0.0.1:8883:1 server2

other_master:
  listen: 127.0.0.1:1115
  hash: fnv1a_64
  distribution: random
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:6660:1 server1


other_slave:
  listen: 127.0.0.1:1116
  hash: fnv1a_64
  distribution: random
  redis: true
  timeout: 1000
  hash_tag: "::"
  servers:
   - 127.0.0.1:6661:1 server1
   - 127.0.0.1:6662:1 server2
```

1. 因为我们使用了主从，所以需要给 server 起一个名字如 server1、server2；否则分片算法默认根据 ip:port:weight，这样就会主从数据的分片算法不一致；
2. 其他信息 Redis 因为每个 Redis 是对等的，因此分片算法可以使用 random；
3. 我们使用了 hash_tag，可以保证相同的 tag 在一个分片上（本例配置了但没有用到该特性）。

复制第六章的 nutcracker.init，帮把配置文件改为 usr/chapter7/nutcracker.yml。然后通过 /usr/chapter7/nutcracker.init start 启动 Twemproxy。

测试主从集群是否工作正常：

Java 代码

```
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli -p 1111
127.0.0.1:1111> set i 1
OK
```

```
127.0.0.1:1111>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli -p 1112
127.0.0.1:1112> get i
"1"
127.0.0.1:1112>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli -p 1113
127.0.0.1:1113> set i 1
OK
127.0.0.1:1113>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli -p 1114
127.0.0.1:1114> get i
"1"
127.0.0.1:1114>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli -p 1115
127.0.0.1:1115> set i 1
OK
127.0.0.1:1115>
root@kaitao:/usr/chapter7# /usr/servers/redis-2.8.19/src/redis-cli -p 1116
127.0.0.1:1116> get i
"1"
```

到此数据集群配置成功。

动态服务实现

因为真实数据是从多个子系统获取，很难模拟这么多子系统交互，所以此处我们使用假数据来进行实现。

项目搭建

我们使用 Maven 搭建 Web 项目，Maven 知识请自行学习。

项目依赖

本文将最小化依赖，即仅依赖我们需要的 servlet、jackson、guava、jedis。

Java 代码

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>17.0</version>
  </dependency>
  <dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.5.2</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.3.3</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.3.3</version>
```

```
 </dependency>
</dependencies>
```

guava 是类似于 apache commons 的一个基础类库，用于简化一些重复操作，可以参考http://ifeve.com/google-guava/。

核心代码

com.github.zhangkaitao.chapter7.servlet.ProductServiceServlet

Java 代码

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    String type = req.getParameter("type");
    String content = null;
    try {
        if("basic".equals(type)) {
            content = getBasicInfo(req.getParameter("skuId"));
        } else if("desc".equals(type)) {
            content = getDescInfo(req.getParameter("skuId"));
        } else if("other".equals(type)) {
            content = getOtherInfo(req.getParameter("ps3Id"), req.getParameter("brandId"));
        }
    } catch (Exception e) {
        e.printStackTrace();
        resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        return;
    }
    if(content != null) {
        resp.setCharacterEncoding("UTF-8");
        resp.getWriter().write(content);
    } else {
        resp.setStatus(HttpServletResponse.SC_NOT_FOUND);
    }
}
```

根据请求参数 type 来决定调用哪个服务获取数据。

基本信息服务

Java 代码

```
private String getBasicInfo(String skuId) throws Exception {
    Map<String, Object> map = new HashMap<String, Object>();
    //商品编号
    map.put("skuId", skuId);
    //名称
    map.put("name", "苹果（Apple）iPhone 6 (A1586) 16GB 金色 移动联通电信4G手机");
    //一级二级三级分类
    map.put("ps1Id", 9987);
    map.put("ps2Id", 653);
    map.put("ps3Id", 655);
    //品牌ID
    map.put("brandId", 14026);
    //图片列表
    map.put("imgs", getImgs(skuId));
    //上架时间
    map.put("date", "2014-10-09 22:29:09");
    //商品毛重
    map.put("weight", "400");
    //颜色尺码
    map.put("colorSize", getColorSize(skuId));
    //扩展属性
    map.put("expands", getExpands(skuId));
    //规格参数
    map.put("propCodes", getPropCodes(skuId));
    map.put("date", System.currentTimeMillis());
    String content = objectMapper.writeValueAsString(map);
    //实际应用应该是发送MQ
    asyncSetToRedis(basicInfoJedisPool, "p:" + skuId + ":", content);
    return objectMapper.writeValueAsString(map);
}

private List<String> getImgs(String skuId) {
    return Lists.newArrayList(
            "jfs/t277/193/1005339798/768456/29136988/542d0798N19d42ce3.jpg",
            "jfs/t352/148/1022071312/209475/53b8cd7f/542d079bN3ea45c98.jpg",
            "jfs/t274/315/1008507116/108039/f70cb380/542d0799Na03319e6.jpg",
            "jfs/t337/181/1064215916/27801/b5026705/542d079aNf184ce18.jpg"
    );
}

private List<Map<String, Object>> getColorSize(String skuId) {
    return Lists.newArrayList(
        makeColorSize(1217499, "金色", "公开版（16GB ROM）"),
        makeColorSize(1217500, "深空灰", "公开版（16GB ROM）"),
        makeColorSize(1217501, "银色", "公开版（16GB ROM）"),
```

```
        makeColorSize(1217508, "金色", "公开版（64GB ROM）"),
        makeColorSize(1217509, "深空灰", "公开版（64GB ROM）"),
        makeColorSize(1217509, "银色", "公开版（64GB ROM）"),
        makeColorSize(1217493, "金色", "移动4G版（16GB）"),
        makeColorSize(1217494, "深空灰", "移动4G版（16GB）"),
        makeColorSize(1217495, "银色", "移动4G版（16GB）"),
        makeColorSize(1217503, "金色", "移动4G版（64GB）"),
        makeColorSize(1217503, "金色", "移动4G版（64GB）"),
        makeColorSize(1217504, "深空灰", "移动4G版（64GB）"),
        makeColorSize(1217505, "银色", "移动4G版（64GB）")
    );
}
private Map<String, Object> makeColorSize(long skuId, String color, String size) {
    Map<String, Object> cs1 = Maps.newHashMap();
    cs1.put("SkuId", skuId);
    cs1.put("Color", color);
    cs1.put("Size", size);
    return cs1;
}

private List<List<?>> getExpands(String skuId) {
    return Lists.newArrayList(
            (List<?>)Lists.newArrayList("热点", Lists.newArrayList("超薄7mm以下", "支持NFC")),
            (List<?>)Lists.newArrayList("系统", "苹果（IOS）"),
            (List<?>)Lists.newArrayList("系统", "苹果（IOS）"),
            (List<?>)Lists.newArrayList("购买方式", "非合约机")
    );
}

private Map<String, List<List<String>>> getPropCodes(String skuId) {
    Map<String, List<List<String>>> map = Maps.newHashMap();
    map.put("主体", Lists.<List<String>>newArrayList(
            Lists.<String>newArrayList("品牌", "苹果（Apple）"),
            Lists.<String>newArrayList("型号", "iPhone 6 A1586"),
            Lists.<String>newArrayList("颜色", "金色"),
            Lists.<String>newArrayList("上市年份", "2014年")
    ));
    map.put("存储", Lists.<List<String>>newArrayList(
            Lists.<String>newArrayList("机身内存", "16GB ROM"),
            Lists.<String>newArrayList("储存卡类型", "不支持")
    ));
    map.put("显示", Lists.<List<String>>newArrayList(
            Lists.<String>newArrayList("屏幕尺寸", "4.7英寸"),
            Lists.<String>newArrayList("触摸屏", "Retina HD"),
            Lists.<String>newArrayList("分辨率", "1334 x 750")
```

```
    ));
    return map;
}
```

本例基本信息提供了如商品名称、图片列表、颜色尺码、扩展属性、规格参数等等数据；而为了简化逻辑大多数数据都是 List/Map 数据结构。

商品介绍服务

Java 代码

```
private String getDescInfo(String skuId) throws Exception {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("content", "<div><img data-lazyload='http://img30.360buyimg.com/jgsq-productsoa/jfs/t448/127/574781110/10
    map.put("date", System.currentTimeMillis());
    String content = objectMapper.writeValueAsString(map);
    //实际应用应该是发送MQ
    asyncSetToRedis(descInfoJedisPool, "d:" + skuId + ":", content);
    return objectMapper.writeValueAsString(map);
}
```

其他信息服务

Java 代码

```
private String getOtherInfo(String ps3Id, String brandId) throws Exception {
    Map<String, Object> map = new HashMap<String, Object>();
    //面包屑
    List<List<?>> breadcrumb = Lists.newArrayList();
    breadcrumb.add(Lists.newArrayList(9987, "手机"));
    breadcrumb.add(Lists.newArrayList(653, "手机通讯"));
    breadcrumb.add(Lists.newArrayList(655, "手机"));
    //品牌
    Map<String, Object> brand = Maps.newHashMap();
    brand.put("name", "苹果（Apple）");
    brand.put("logo", "BrandLogo/g14/M09/09/10/rBEhVlK6vdkIAAAAAAAFLXzp-IIAAHWawP_QjwAAAVF472.png");
    map.put("breadcrumb", breadcrumb);
    map.put("brand", brand);
    //实际应用应该是发送MQ
    asyncSetToRedis(otherInfoJedisPool, "s:" + ps3Id + ":", objectMapper.writeValueAsString(breadcrumb));
    asyncSetToRedis(otherInfoJedisPool, "b:" + brandId + ":", objectMapper.writeValueAsString(brand));
    return objectMapper.writeValueAsString(map);
}
```

本例中其他信息只使用了面包屑和品牌数据。

辅助工具

Java 代码

```
private ObjectMapper objectMapper = new ObjectMapper();
private JedisPool basicInfoJedisPool = createJedisPool("127.0.0.1", 1111);
private JedisPool descInfoJedisPool = createJedisPool("127.0.0.1", 1113);
private JedisPool otherInfoJedisPool = createJedisPool("127.0.0.1", 1115);

private JedisPool createJedisPool(String host, int port) {
    GenericObjectPoolConfig poolConfig = new GenericObjectPoolConfig();
    poolConfig.setMaxTotal(100);
    return new JedisPool(poolConfig, host, port);
}

private ExecutorService executorService = Executors.newFixedThreadPool(10);
private void asyncSetToRedis(final JedisPool jedisPool, final String key, final String content) {
    executorService.submit(new Runnable() {
        @Override
        public void run() {
            Jedis jedis = null;
            try {
                jedis = jedisPool.getResource();
                jedis.set(key, content);
            } catch (Exception e) {
                e.printStackTrace();
                jedisPool.returnBrokenResource(jedis);
            } finally {
                jedisPool.returnResource(jedis);
            }

        }
    });
}
```

本例使用 Jackson 进行 JSON 的序列化；Jedis 进行 Redis 的操作；使用线程池做异步更新（实际应用中可以使用 MQ 做实现）。

web.xml 配置

Java 代码

```
<servlet>
  <servlet-name>productServiceServlet</servlet-name>
  <servlet-class>com.github.zhangkaitao.chapter7.servlet.ProductServiceServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>productServiceServlet</servlet-name>
  <url-pattern>/info</url-pattern>
</servlet-mapping>
```

打 WAR 包

Java 代码

```
cd D:\workspace\chapter7
mvn clean package
```

此处使用 maven 命令打包，比如本例将得到 chapter7.war，然后将其上传到服务器的 /usr/chapter7/webapp，然后通过 unzip chapter6.war 解压。

配置 Tomcat

复制第六章使用的 tomcat 实例：

Java 代码

```
cd /usr/servers/
cp -r tomcat-server1 tomcat-chapter7/
vim /usr/servers/tomcat-chapter7/conf/Catalina/localhost/ROOT.xml
```

Java 代码

```
<!-- 访问路径是根，web应用所属目录为/usr/chapter7/webapp -->
<Context path="" docBase="/usr/chapter7/webapp"></Context>
```

指向第七章的 web 应用路径。

测试

启动 tomcat 实例。

Java 代码

```
/usr/servers/tomcat-chapter7/bin/startup.sh
```

访问如下 URL 进行测试。

Java 代码

```
http://192.168.1.2:8080/info?type=basic&skuId=1
http://192.168.1.2:8080/info?type=desc&skuId=1
http://192.168.1.2:8080/info?type=other&ps3Id=1&brandId=1
```

nginx 配置

vim /usr/chapter7/nginx_chapter7.conf

Java 代码

```
upstream backend {
    server 127.0.0.1:8080 max_fails=5 fail_timeout=10s weight=1;
    check interval=3000 rise=1 fall=2 timeout=5000 type=tcp default_down=false;
    keepalive 100;
}

server {
    listen       80;
    server_name  item2015.jd.com item.jd.com d.3.cn;

    location ~ /backend/(.*) {
        #internal;
        keepalive_timeout   30s;
        keepalive_requests  1000;
        #支持keep-alive
        proxy_http_version 1.1;
        proxy_set_header Connection "";

        rewrite /backend/(.*) $1 break;
        proxy_pass_request_headers off;
```

```
        #more_clear_input_headers Accept-Encoding;
        proxy_next_upstream error timeout;
        proxy_pass http://backend;
    }
}
```

此处 server_name 我们指定了 item.jd.com (商品详情页)和 d.3.cn (商品介绍)。其他配置可以参考第六章内容。另外实际生产环境要把 #internal 打开，表示只有本 nginx 能访问。

vim /usr/servers/nginx/conf/nginx.conf

Java 代码

```
include /usr/chapter7/nginx_chapter7.conf;
\#为了方便测试，注释掉example.conf
include /usr/chapter6/nginx_chapter6.conf;
```

Java 代码

```
\#lua模块路径，其中”;;”表示默认搜索路径，默认到/usr/servers/nginx下找
lua_package_path "/usr/chapter7/lualib/?.lua;;"; #lua 模块
lua_package_cpath "/usr/chapter7/lualib/?.so;;"; #c模块
lua模块从/usr/chapter7目录加载，因为我们要写自己的模块使用。
```

重启 nginx

/usr/servers/nginx/sbin/nginx -s reload

绑定 hosts

192.168.1.2 item.jd.com 192.168.1.2 item2015.jd.com 192.168.1.2 d.3.cn

访问如 http://item.jd.com/backend/info?type=basic&skuId=1 即看到结果。

前端展示实现

我们分为三部分实现：基础组件、商品介绍、前端展示部分。

基础组件

首先我们进行基础组件的实现，商品介绍和前端展示部分都需要读取 Redis 和 Http 服务，因此我们可以抽取公共部分出来复用。

vim /usr/chapter7/lualib/item/common.lua

Java 代码

```
local redis = require("resty.redis")
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local function close_redis(red)
    if not red then
        return
    end
    --释放连接(连接池实现)
    local pool_max_idle_time = 10000 --毫秒
    local pool_size = 100 --连接池大小
    local ok, err = red:set_keepalive(pool_max_idle_time, pool_size)

    if not ok then
        ngx_log(ngx_ERR, "set redis keepalive error : ", err)
    end
end

local function read_redis(ip, port, keys)
    local red = redis:new()
    red:set_timeout(1000)
    local ok, err = red:connect(ip, port)
    if not ok then
        ngx_log(ngx_ERR, "connect to redis error : ", err)
        return close_redis(red)
    end
    local resp = nil
    if #keys == 1 then
        resp, err = red:get(keys[1])
```

```
    else
        resp, err = red:mget(keys)
    end
    if not resp then
        ngx_log(ngx_ERR, "get redis content error : ", err)
        return close_redis(red)
    end

    --得到的数据为空处理
    if resp == ngx.null then
        resp = nil
    end
    close_redis(red)

    return resp
end

local function read_http(args)
    local resp = ngx.location.capture("/backend/info", {
        method = ngx.HTTP_GET,
        args = args
    })

    if not resp then
        ngx_log(ngx_ERR, "request error")
        return
    end
    if resp.status ~= 200 then
        ngx_log(ngx_ERR, "request error, status :", resp.status)
        return
    end
    return resp.body
end

local _M = {
    read_redis = read_redis,
    read_http = read_http
}
return _M
```

整个逻辑和第六章类似；只是 read_redis 根据参数 keys 个数支持 get 和 mget。比如read_redis(ip, port, {"key1"}) 则调用 get 而 read_redis(ip, port, {"key1", "key2"}) 则调用 mget。

商品介绍

核心代码

vim /usr/chapter7/desc.lua

Java 代码

```
local common = require("item.common")
local read_redis = common.read_redis
local read_http = common.read_http
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local ngx_exit = ngx.exit
local ngx_print = ngx.print
local ngx_re_match = ngx.re.match
local ngx_var = ngx.var

local descKey = "d:" .. skuId .. ":"
local descInfoStr = read_redis("127.0.0.1", 1114, {descKey})
if not descInfoStr then
  ngx_log(ngx_ERR, "redis not found desc info, back to http, skuId : ", skuId)
  descInfoStr = read_http({type="desc", skuId = skuId})
end
if not descInfoStr then
  ngx_log(ngx_ERR, "http not found basic info, skuId : ", skuId)
  return ngx_exit(404)
end
ngx_print("showdesc(")
ngx_print(descInfoStr)
ngx_print(")")
```

通过复用逻辑后整体代码简化了许多；此处读取商品介绍从集群；另外前端展示使用 JSONP 技术展示商品介绍。

nginx 配置

vim /usr/chapter7/nginx_chapter7.conf

Java 代码

```
location ~^/desc/(\d+)$ {
    if ($host != "d.3.cn") {
      return 403;
    }
    default_type application/x-javascript;
    charset utf-8;
    lua_code_cache on;
    set $skuId $1;
    content_by_lua_file /usr/chapter7/desc.lua;
}
```

因为 item.jd.com 和 d.3.cn 复用了同一个配置文件，此处需要限定只有 d.3.cn 域名能访问，防止恶意访问。

重启 nginx 后，访问如 http://d.3.cn/desc/1 即可得到 JSONP 结果。

前端展示

核心代码

vim /usr/chapter7/item.lua

Java 代码

```
local common = require("item.common")
local item = require("item")
local read_redis = common.read_redis
local read_http = common.read_http
local cjson = require("cjson")
local cjson_decode = cjson.decode
local ngx_log = ngx.log
local ngx_ERR = ngx.ERR
local ngx_exit = ngx.exit
local ngx_print = ngx.print
local ngx_var = ngx.var


local skuId = ngx_var.skuId

--获取基本信息
local basicInfoKey = "p:" .. skuId .. ":"
local basicInfoStr = read_redis("127.0.0.1", 1112, {basicInfoKey})
if not basicInfoStr then
  ngx_log(ngx_ERR, "redis not found basic info, back to http, skuId : ", skuId)
  basicInfoStr = read_http({type="basic", skuId = skuId})
end
if not basicInfoStr then
  ngx_log(ngx_ERR, "http not found basic info, skuId : ", skuId)
  return ngx_exit(404)
end

local basicInfo = cjson_decode(basicInfoStr)
local ps3Id = basicInfo["ps3Id"]
local brandId = basicInfo["brandId"]
--获取其他信息
local breadcrumbKey = "s:" .. ps3Id .. ":"
local brandKey = "b:" .. brandId ..":"
local otherInfo = read_redis("127.0.0.1", 1116, {breadcrumbKey, brandKey}) or {}
```

```
local breadcrumbStr = otherInfo[1]
local brandStr = otherInfo[2]
if breadcrumbStr then
  basicInfo["breadcrumb"] = cjson_decode(breadcrumbStr)
end
if brandStr then
  basicInfo["brand"] = cjson_decode(brandStr)
end
if not breadcrumbStr and not brandStr then
  ngx_log(ngx_ERR, "redis not found other info, back to http, skuId : ", brandId)
  local otherInfoStr = read_http({type="other", ps3Id = ps3Id, brandId = brandId})
  if not otherInfoStr then
     ngx_log(ngx_ERR, "http not found other info, skuId : ", skuId)
  else
    local otherInfo = cjson_decode(otherInfoStr)
    basicInfo["breadcrumb"] = otherInfo["breadcrumb"]
    basicInfo["brand"] = otherInfo["brand"]
  end
end

local name = basicInfo["name"]
--name to unicode
basicInfo["unicodeName"] = item.utf8_to_unicode(name)
--字符串截取，超长显示...
basicInfo["moreName"] = item.trunc(name, 10)
--初始化各分类的url
item.init_breadcrumb(basicInfo)
--初始化扩展属性
item.init_expand(basicInfo)
--初始化颜色尺码
item.init_color_size(basicInfo)
local template = require "resty.template"
template.caching(true)
template.render("item.html", basicInfo)
```

整个逻辑分为四部分：1、获取基本信息；2、根据基本信息中的关联关系获取其他信息；3、初始化/格式化数据；4、渲染模板。

初始化模块

vim /usr/chapter7/lualib/item.lua

Java 代码

```
local bit = require("bit")
local utf8 = require("utf8")
local cjson = require("cjson")
local cjson_encode = cjson.encode
local bit_band = bit.band
local bit_bor = bit.bor
local bit_lshift = bit.lshift
local string_format = string.format
local string_byte = string.byte
local table_concat = table.concat

--utf8转为unicode
local function utf8_to_unicode(str)
    if not str or str == "" or str == ngx.null then
        return nil
    end
    local res, seq, val = {}, 0, nil
    for i = 1, #str do
        local c = string_byte(str, i)
        if seq == 0 then
            if val then
                res[#res + 1] = string_format("%04x", val)
            end

            seq = c < 0x80 and 1 or c < 0xE0 and 2 or c < 0xF0 and 3 or
                        c < 0xF8 and 4 or --c < 0xFC and 5 or c < 0xFE and 6 or
                        0
            if seq == 0 then
                ngx.log(ngx.ERR, 'invalid UTF-8 character sequence' .. ",,," .. tostring(str))
                return str
            end

            val = bit_band(c, 2 ^ (8 - seq) - 1)
        else
            val = bit_bor(bit_lshift(val, 6), bit_band(c, 0x3F))
        end
        seq = seq - 1
    end
    if val then
        res[#res + 1] = string_format("%04x", val)
    end
    if #res == 0 then
        return str
    end
    return "\\u" .. table_concat(res, "\\u")
```

```
end

--utf8字符串截取
local function trunc(str, len)
  if not str then
    return nil
  end

  if utf8.len(str) > len then
    return utf8.sub(str, 1, len) .. "..."
  end
  return str
end

--初始化面包屑
local function init_breadcrumb(info)
   local breadcrumb = info["breadcrumb"]
   if not breadcrumb then
      return
   end

   local ps1Id = breadcrumb[1][1]
   local ps2Id = breadcrumb[2][1]
   local ps3Id = breadcrumb[3][1]

   --此处应该根据一级分类查找url
   local ps1Url = "http://shouji.jd.com/"
   local ps2Url = "http://channel.jd.com/shouji.html"
   local ps3Url = "http://list.jd.com/list.html?cat=" .. ps1Id .. "," .. ps2Id .. "," .. ps3Id

   breadcrumb[1][3] = ps1Url
   breadcrumb[2][3] = ps2Url
   breadcrumb[3][3] = ps3Url
end

--初始化扩展属性
local function init_expand(info)
  local expands = info["expands"]
  if not expands then
    return
  end
  for _, e in ipairs(expands) do
    if type(e[2]) == "table" then
      e[2] = table_concat(e[2], "，")
    end
```

```
  end
end

--初始化颜色尺码
local function init_color_size(info)
  local colorSize = info["colorSize"]

  --颜色尺码JSON串
  local colorSizeJson = cjson_encode(colorSize)
  --颜色列表（不重复）
  local colorList = {}
  --尺码列表（不重复）
  local sizeList = {}
  info["colorSizeJson"] = colorSizeJson
  info["colorList"] = colorList
  info["sizeList"] = sizeList

  local colorSet = {}
  local sizeSet = {}
  for _, cz in ipairs(colorSize) do
    local color = cz["Color"]
    local size = cz["Size"]
    if color and color ~= "" and not colorSet[color] then
      colorList[#colorList + 1] = {color = color, url = "http://item.jd.com/" ..cz["SkuId"] .. ".html"}
      colorSet[color] = true
    end
    if size and size ~= "" and not sizeSet[size] then
      sizeList[#sizeList + 1] = {size = size, url = "http://item.jd.com/" ..cz["SkuId"] .. ".html"}
      sizeSet[size] = ""
    end
  end
end

local _M = {
  utf8_to_unicode = utf8_to_unicode,
  trunc = trunc,
  init_breadcrumb = init_breadcrumb,
  init_expand = init_expand,
  init_color_size = init_color_size
}

return _M
```

比如utf8_to_unicode 代码之前已经见过了，其他的都是一些逻辑代码。

模板 html 片段

Java 代码

```
var pageConfig = {
    compatible: true,
    product: {
        skuid: {* skuId *},
        name: '{* unicodeName *}',
        skuidkey:'AFC266E971535B664FC926D34E91C879',
        href: 'http://item.jd.com/{* skuId *}.html',
        src: '{* imgs[1] *}',
        cat: [{* ps1Id *},{* ps2Id *},{* ps3Id *}],
        brand: {* brandId *},
        tips: false,
        pType: 1,
        venderId:0,
        shopId:'0',
        specialAttrs:["HYKHSP-0","isDistribution","isHaveYB","isSelfService-0","isWeChatStock-0","packType","IsNewGood
        videoPath:'',
        desc: 'http://d.3.cn/desc/{* skuId *}'
    }
};
var warestatus = 1;
{% if colorSizeJson then %} var ColorSize = {* colorSizeJson *};{% end %}
        {-raw-}
        try{(function(flag){ if(!flag){return;} if(window.location.hash == '#m'){var exp = new Date();exp.setTime(exp.getTime()
        {-raw-}
{* var *}输出变量，{% code %} 写代码片段，{-raw-} 不进行任何处理直接输出。
```

面包屑

Java 代码

```
<div class="breadcrumb">
    <strong><a href='{* breadcrumb[1][3] *}'>{* breadcrumb[1][2] *}</a></strong>
    <span>
         &gt; 
        <a href='{* breadcrumb[2][3] *}'>{* breadcrumb[2][2] *}</a>
         &gt; 
        <a href='{* breadcrumb[3][3] *}'>{* breadcrumb[3][2] *}</a>
         &gt; 
    </span>
    <span>
```

```
        {% if brand then %}
        <a href='http://www.jd.com/pinpai/{* ps3Id *}-{* brandId *}.html'>{* brand['name'] *}</a>
         &gt; 
       {% end %}
       <a href='http://item.jd.com/{* skuId *}.html'>{* moreName *}</a>
    </span>
</div>
```

图片列表

Java 代码

```
<div id="spec-n1" class="jqzoom" onclick="window.open('http://www.jd.com/bigimage.aspx?id={* skuId *}')" clstag="shang
    <img data-img="1" width="350" height="350" src="http://img14.360buyimg.com/n1/{* imgs[1] *}" alt="{* name *}"/>
</div>
<div id="spec-list" clstag="shangpin|keycount|product|spec-n5">
    <a href="javascript:;" class="spec-control" id="spec-forward"></a>
    <a href="javascript:;" class="spec-control" id="spec-backward"></a>
    <div class="spec-items">
        <ul class="lh">
            {% for _, img in ipairs(imgs) do %}
            <li><img class='img-hover' alt='{* name *}' src='http://img14.360buyimg.com/n5/{* img *}' data-url='{* img *}' data-i
            {% end %}
        </ul>
    </div>
</div>
```

颜色尺码选择

Java 代码

```
<div class="dt">选择颜色：</div>
    <div class="dd">
        {% for _, color in ipairs(colorList) do %}
            <div class="item"><b></b><a href="{* color['url'] *}" title="{* color['color'] *}"><i>{* color['color'] *}</i></a></div>
        {% end %}
    </div>
</div>
<div id="choose-version" class="li">
    <div class="dt">选择版本：</div>
    <div class="dd">
        {% for _, size in ipairs(sizeList) do %}
            <div class="item"><b></b><a href="{* size['url'] *}" title="{* size['size'] *}">{* size['size'] *}</a></div>
        {% end %}
    </div>
</div>
```

扩展属性

Java 代码

```
<ul id="parameter2" class="p-parameter-list">
    <li title='{* name *}'>商品名称：{* name *}</li>
    <li title='{* skuId *}'>商品编号：{* skuId *}</li>
    {% if brand then %}
    <li title='{* brand["name"] *}'>品牌：  <a href='http://www.jd.com/pinpai/{* ps3Id *}-{* brandId *}.html' target='_blank'>{* br
    {% end %}
    {% if date then %}
    <li title='{* date *}'>上架时间：{* date *}</li>
    {% end %}
    {% if weight then %}
    <li title='{* weight *}'>商品毛重：{* weight *}</li>
    {% end %}
    {% for _, e in pairs(expands) do %}
    <li title='{* e[2] *}'>{* e[1] *}：{* e[2] *}</li>
    {% end %}
</ul>
```

规格参数

Java 代码

```
<table cellpadding="0" cellspacing="1" width="100%" border="0" class="Ptable">
    {% for group, pc in pairs(propCodes) do  %}
    <tr><th class="tdTitle" colspan="2">{* group *}</th><tr>
    {% for _, v in pairs(pc) do %}
    <tr><td class="tdTitle">{* v[1] *}</td><td>{* v[2] *}</td></tr>
    {% end %}
    {% end %}
</table>
```

nginx 配置

vim /usr/chapter7/nginx_chapter7.conf

Java 代码

```
\#模板加载位置
set $template_root "/usr/chapter7";
```

```
location ~ ^/(\d+).html$ {
    if ($host !~ "^(item|item2015)\.jd\.com$") {
        return 403;
    }
    default_type 'text/html';
    charset utf-8;
    lua_code_cache on;
    set $skuId $1;
    content_by_lua_file /usr/chapter7/item.lua;
}
```

测试

重启 nginx，访问 http://item.jd.com/1217499.html 可得到响应内容，本例和京东的商品详情页的数据是有些出入的，输出的页面可能是缺少一些数据的。

优化

local cache

对于其他信息，对数据一致性要求不敏感，而且数据量很少，完全可以在本地缓存全量；而且可以设置如5-10分钟的过期时间是完全可以接受的；因此可以 lua_shared_dict 全局内存进行缓存。具体逻辑可以参考

Java 代码

```
local nginx_shared = ngx.shared
--item.jd.com配置的缓存
local local_cache = nginx_shared.item_local_cache
local function cache_get(key)
    if not local_cache then
        return nil
    end
    return local_cache:get(key)
end

local function cache_set(key, value)
    if not local_cache then
        return nil
    end
    return local_cache:set(key, value, 10 * 60) --10分钟
end
```

```
local function get(ip, port, keys)
    local tables = {}
    local fetchKeys = {}
    local resp = nil
    local status = STATUS_OK
    --如果tables是个map #tables拿不到长度
    local has_value = false
    --先读取本地缓存
    for i, key in ipairs(keys) do
        local value = cache_get(key)
        if value then
            if value == "" then
                value = nil
            end
            tables[key] = value
            has_value = true
        else
            fetchKeys[#fetchKeys + 1] = key
        end
    end

    --如果还有数据没获取 从redis获取
    if #fetchKeys > 0 then
        if #fetchKeys == 1 then
            status, resp = redis_get(ip, port, fetchKeys[1])
        else
            status, resp = redis_mget(ip, port, fetchKeys)
        end
        if status == STATUS_OK then
            for i = 1, #fetchKeys do
                local key = fetchKeys[i]
                local value = nil
                if #fetchKeys == 1 then
                  value = resp
                else
                  value = get_data(resp, i)
                end
                tables[key] = value
                 has_value = true
                 cache_set(key, value or "", ttl)
            end
        end
    end
    --如果从缓存查到 就认为ok
    if has_value and status == STATUS_NOT_FOUND then
```

```
        status = STATUS_OK
    end
    return status, tables
end
```

nginx proxy cache

为了防止恶意刷页面/热点页面访问频繁，我们可以使用 nginx proxy_cache 做页面缓存，当然更好的选择是使用 CDN 技术，如通过 Apache Traffic Server、Squid、Varnish。

nginx.conf 配置

Java 代码

```
proxy_buffering on;
proxy_buffer_size 8k;
proxy_buffers 256 8k;
proxy_busy_buffers_size 64k;
proxy_temp_file_write_size 64k;
proxy_temp_path /usr/servers/nginx/proxy_temp;
\#设置Web缓存区名称为cache_one，内存缓存空间大小为200MB，1分钟没有被访问的内容自动清除，硬盘缓存空间大小为30GB。
proxy_cache_path  /usr/servers/nginx/proxy_cache levels=1:2 keys_zone=cache_item:200m inactive=1m max_size=30g
```

增加 proxy_cache 的配置，可以通过挂载一块内存作为缓存的存储空间。更多配置规则请参考http://nginx.org/cn/docs/http/ngx_http_proxy_module.html。

nginx_chapter7.conf 配置

与 server 指令配置同级

Java 代码

```
\############ 测试时使用的动态请求
map $host $item_dynamic {
    default                 "0";
    item2015.jd.com         "1";
}
```

即如果域名为 item2015.jd.com则item_dynamic=1。

Java 代码

```
location ~ ^/(\d+).html$ {
    set $skuId $1;
    if ($host !~ "^(item|item2015)\.jd\.com$") {
      return 403;
    }

    expires 3m;
    proxy_cache cache_item;
    proxy_cache_key $uri;
    proxy_cache_bypass $item_dynamic;
    proxy_no_cache $item_dynamic;
    proxy_cache_valid 200 301 3m;
    proxy_cache_use_stale updating error timeout invalid_header http_500 http_502 http_503 http_504;
    proxy_pass_request_headers off;
    proxy_set_header Host $host;
    #支持keep-alive
    proxy_http_version 1.1;
    proxy_set_header Connection "";
    proxy_pass http://127.0.0.1/proxy/$skuId.html;
    add_header X-Cache '$upstream_cache_status';
}

location ~ ^/proxy/(\d+).html$ {
    allow 127.0.0.1;
    deny all;
    keepalive_timeout   30s;
    keepalive_requests  1000;
    default_type 'text/html';
    charset utf-8;
    lua_code_cache on;
    set $skuId $1;
    content_by_lua_file /usr/chapter7/item.lua;
}
```

- expires：设置响应缓存头信息，此处是3分钟；将会得到 Cache-Control:max-age=180 和类似 Expires:Sat, 28 Feb 2015 10:01:10 GMT 的响应头；
- proxy_cache：使用之前在 nginx.conf 中配置的 cache_item 缓存；
- proxy_cache_key：缓存 key 为 uri，不包括 host 和参数，这样不管用户怎么通过在 url 上加随机数都是走缓存的；
- proxy_cache_bypass：nginx 不从缓存取响应的条件，可以写多个；如果存在一个字符串条件且不是“0”，那么 nginx 就不会从缓存中取响应内容；此处如果我们使用的 host 为item2015.jd.com 时就不会从缓存取响应内容；

- proxy_no_cache：nginx 不将响应内容写入缓存的条件，可以写多个；如果存在一个字符串条件且不是“0”，那么 nginx 就不会从将响应内容写入缓存；此处如果我们使用的 host 为item2015.jd.com 时就不会将响应内容写入缓存；
- proxy_cache_valid：为不同的响应状态码设置不同的缓存时间，此处我们对 200、301 缓存3分钟；
- proxy_cache_use_stale：什么情况下使用不新鲜（过期）的缓存内容；配置和proxy_next_upstream 内容类似；此处配置了如果连接出错、超时、404、500 等都会使用不新鲜的缓存内容；此外我们配置了 updating 配置，通过配置它可以在 nginx 正在更新缓存（其中一个 Worker 进程）时（其他的 Worker 进程）使用不新鲜的缓存进行响应，这样可以减少回源的数量；
- proxy_pass_request_headers：我们不需要请求头，所以不传递； proxy_http_version 1.1 和 proxy_set_header Connection ""：支持keepalive； add_header X-Cache '$upstream_cache_status'：添加是否缓存命中的响应头；比如命中 HIT、不命中 MISS、不走缓存 BYPASS；比如命中会看到 X-Cache：HIT响应头；
- allow/deny：允许和拒绝访问的 ip 列表，此处我们只允许本机访问； keepalive_timeout 30s 和 keepalive_requests 1000：支持 keepalive；

nginx_chapter7.conf 清理缓存配置

Java 代码

```
location /purge {
    allow    127.0.0.1;
    allow    192.168.0.0/16;
    deny     all;
    proxy_cache_purge  cache_item $arg_url;
}
```

只允许内网访问。访问如 http://item.jd.com/purge?url=/11.html；如果看到Successful purge 说明缓存存在并清理了。

修改 item.lua 代码

Java 代码

```
--添加Last-Modified，用于响应304缓存
ngx.header["Last-Modified"] = ngx.http_time(ngx.now())

local template = require "resty.template"
template.caching(true)
```

```
template.render("item.html", basicInfo)
~
```

在渲染模板前设置 Last-Modified，用于判断内容是否变更的条件，默认 Nginx 通过等于去比较，也可以通过配置 if_modified_since 指令来支持小于等于比较；如果请求头发送的 If-Modified-Since 和 Last-Modified 匹配则返回 304 响应，即内容没有变更，使用本地缓存。此处可能看到了我们的 Last-Modified 是当前时间，不是商品信息变更的时间；商品信息变更时间由：商品信息变更时间、面包屑变更时间和品牌变更时间三者决定的，因此实际应用时应该取三者最大的；还一个问题就是模板内容可能变了，但是商品信息没有变，此时使用 Last-Modified 得到的内容可能是错误的，所以可以通过使用 ETag 技术来解决这个问题，ETag 可以认为是内容的一个摘要，内容变更后摘要就变了。

GZIP 压缩

修改 nginx.conf 配置文件

Java 代码

```
gzip on;
gzip_min_length  4k;
gzip_buffers     4 16k;
gzip_http_version 1.0;
gzip_proxied       any; #前端是squid的情况下要加此参数，否则squid上不缓存gzip文件
gzip_comp_level 2;
gzip_types       text/plain application/x-javascript text/css application/xml;
gzip_vary on;
```

此处我们指定至少 4k 时才压缩，如果数据太小压缩没有意义。

到此整个商品详情页逻辑就介绍完了，一些细节和运维内容需要在实际开发中实际处理，无法做到面面俱到。

10

流量复制 /AB 测试/协程

流量复制

在实际开发中经常涉及到项目的升级，而该升级不能简单的上线就完事了，需要验证该升级是否兼容老的上线，因此可能需要并行运行两个项目一段时间进行数据比对和校验，待没问题后再进行上线。这其实就需要进行流量复制，把流量复制到其他服务器上，一种方式是使用如 tcpcopy 引流；另外我们还可以使用 nginx 的 HttpLuaModule 模块中的 ngx.location.capture_multi 进行并发执行来模拟复制。

构造两个服务

Java 代码

```
location /test1 {
    keepalive_timeout 60s;
    keepalive_requests 1000;
    content_by_lua '
        ngx.print("test1 : ", ngx.req.get_uri_args()["a"])
        ngx.log(ngx.ERR, "request test1")
    ';
}
location /test2 {
    keepalive_timeout 60s;
    keepalive_requests 1000;
    content_by_lua '
        ngx.print("test2 : ", ngx.req.get_uri_args()["a"])
        ngx.log(ngx.ERR, "request test2")
    ';
}
```

通过 ngx.location.capture_multi 调用

Java 代码

```
location /test {
    lua_socket_connect_timeout 3s;
    lua_socket_send_timeout 3s;
    lua_socket_read_timeout 3s;
    lua_socket_pool_size 100;
    lua_socket_keepalive_timeout 60s;
    lua_socket_buffer_size 8k;

    content_by_lua '
        local res1, res2 = ngx.location.capture_multi{
```

```
            { "/test1", { args = ngx.req.get_uri_args() } },
            { "/test2", { args = ngx.req.get_uri_args()} },
        }
        if res1.status == ngx.HTTP_OK then
            ngx.print(res1.body)
        end
        if res2.status ~= ngx.HTTP_OK then
            --记录错误
        end
    ';
}
```

此处可以根据需求设置相应的超时时间和长连接连接池等；ngx.location.capture 底层通过cosocket 实现，而其支持 Lua 中的协程，通过它可以以同步的方式写非阻塞的代码实现。

此处要考虑记录失败的情况，对失败的数据进行重放还是放弃根据自己业务做处理。

AB 测试

AB 测试即多版本测试，有时候我们开发了新版本需要灰度测试，即让一部分人看到新版，一部分人看到老版，然后通过访问数据决定是否切换到新版。比如可以通过根据区域、用户等信息进行切版本。

比如京东商城有一个 cookie 叫做__jda，该 cookie 是在用户访问网站时种下的，因此我们可以拿到这个 cookie，根据这个 cookie 进行版本选择。

比如两次清空 cookie 访问发现第二个数字串是变化的，即我们可以根据第二个数字串进行判断。
__jda=122270672.1059377902.1425691107.1425691107.1425699059.1 __jda=122270672.556927616.1425699216.1425699216.1425699216.1。

判断规则可以比较多的选择，比如通过尾号；要切 30% 的流量到新版，可以通过选择尾号为 1，3,5 的切到新版，其余的还停留在老版。

使用 map 选择版本

Java 代码

```
map $cookie___jda $ab_key {
    default                         "0";
    ~^\d+\.\d+(?P<k>(1|3|5))\.      "1";
}
```

使用 map 映射规则，即如果是到新版则等于 "1"，到老版等于 “0”； 然后我们就可以通过 ngx.var.ab_key 获取到该数据。

Java 代码

```
location /abtest1 {
    if ($ab_key = "1") {
        echo_location /test1 ngx.var.args;
    }
    if ($ab_key = "0") {
        echo_location /test2 ngx.var.args;
    }
}
```

此处也可以使用 proxy_pass 到不同版本的服务器上

Java 代码

```
location /abtest2 {
    if ($ab_key = "1") {
        rewrite ^ /test1 break;
        proxy_pass http://backend1;
    }
    rewrite ^ /test2 break;
    proxy_pass http://backend2;
}
```

直接在 Lua 中使用 lua-resty-cookie 获取该 Cookie 进行解析

首先下载 lua-resty-cookie

Java 代码

```
cd /usr/example/lualib/resty/
wget https://raw.githubusercontent.com/cloudflare/lua-resty-cookie/master/lib/resty/cookie.lua
```

Java 代码

```
location /abtest3 {
    content_by_lua '

        local ck = require("resty.cookie")
        local cookie = ck:new()
        local ab_key = "0"
        local jda = cookie:get("__jda")
        if jda then
            local v = ngx.re.match(jda, [[^\d+\.\d+(1|3|5)\.]])
            if v then
                ab_key = "1"
            end
        end

        if ab_key == "1" then
            ngx.exec("/test1", ngx.var.args)
        else
            ngx.print(ngx.location.capture("/test2", {args = ngx.req.get_uri_args()}).body)
        end
    ';

}
```

首先使用 lua-resty-cookie 获取 cookie，然后使用 ngx.re.match 进行规则的匹配，最后使用 ngx.exec 或者 ngx.location.capture 进行处理。此处同时使用 ngx.exec 和ngx.location.capture 目的是为了演示，此外没有对 ngx.location.capture 进行异常处理。

协程

Lua 中没有线程和异步编程编程的概念，对于并发执行提供了协程的概念，个人认为协程是在A运行中发现自己忙则把 CPU 使用权让出来给B使用，最后 A 能从中断位置继续执行，本地还是单线程，CPU 独占的；因此如果写网络程序需要配合非阻塞 I/O 来实现。

ngx_lua 模块对协程做了封装，我们可以直接调用 ngx.thread API 使用，虽然称其为“轻量级线程”，但其本质还是 Lua 协程。该 API 必须配合该 ngx_lua 模块提供的非阻塞 I/O API 一起使用，比如我们之前使用的 ngx.location.capture_multi 和 lua-resty-redis、lua-resty-mysql 等基于 cosocket 实现的都是支持的。

通过 Lua 协程我们可以并发的调用多个接口，然后谁先执行成功谁先返回，类似于 BigPipe 模型。

依赖的 API

Java 代码

```
location /api1 {
    echo_sleep 3;
    echo api1 : $arg_a;
}
location /api2 {
    echo_sleep 3;
    echo api2 : $arg_a;
}
```

我们使用 echo_sleep 等待 3 秒。

串行实现

Java 代码

```
location /serial {
    content_by_lua '
        local t1 = ngx.now()
        local res1 = ngx.location.capture("/api1", {args = ngx.req.get_uri_args()})
        local res2 = ngx.location.capture("/api2", {args = ngx.req.get_uri_args()})
        local t2 = ngx.now()
        ngx.print(res1.body, "<br/>", res2.body, "<br/>", tostring(t2-t1))
```

```
    ';
}
```

即一个个的调用，总的执行时间在6秒以上，比如访问 `http://192.168.1.2/serial?a=22`

Java 代码

```
api1 : 22
api2 : 22
6.0040001869202
```

ngx.location.capture_multi 实现

Java 代码

```
location /concurrency1 {
    content_by_lua '
        local t1 = ngx.now()
        local res1,res2 = ngx.location.capture_multi({
            {"/api1", {args = ngx.req.get_uri_args()}},
            {"/api2", {args = ngx.req.get_uri_args()}}

        })
        local t2 = ngx.now()
        ngx.print(res1.body, "<br/>", res2.body, "<br/>", tostring(t2-t1))
    ';
}
```

直接使用 ngx.location.capture_multi 来实现，比如访问 `http://192.168.1.2/concurrency1?a=22`

Java 代码

```
api1 : 22
api2 : 22
3.0020000934601
```

协程 API 实现

Java 代码

```
location /concurrency2 {
    content_by_lua '
        local t1 = ngx.now()
```

```
        local function capture(uri, args)
          return ngx.location.capture(uri, args)
        end
        local thread1 = ngx.thread.spawn(capture, "/api1", {args = ngx.req.get_uri_args()})
        local thread2 = ngx.thread.spawn(capture, "/api2", {args = ngx.req.get_uri_args()})
        local ok1, res1 = ngx.thread.wait(thread1)
        local ok2, res2 = ngx.thread.wait(thread2)
        local t2 = ngx.now()
        ngx.print(res1.body, "<br/>", res2.body, "<br/>", tostring(t2-t1))
    ';
}
```

使用 ngx.thread.spawn 创建一个轻量级线程，然后使用 ngx.thread.wait 等待该线程的执行成功。比如访问 `http://192.168.1.2/concurrency2?a=22`

Java 代码

```
api1 : 22
api2 : 22
3.0030000209808
```

其有点类似于 Java 中的线程池执行模型，但不同于线程池，其每次只执行一个函数，遇到 IO 等待则让出 CPU 让下一个执行。我们可以通过下面的方式实现任意一个成功即返回，之前的是等待所有执行成功才返回。

Java 代码

```
local  ok, res = ngx.thread.wait(thread1, thread2)
```