跟我学Nginx+Lua开发

作者: jinnianshilongnian
http://jinnianshilongnian.iteye.com

跟我学Nginx+Lua开发

目 录

1. 跟我学Nginx+Lua开发

1.1 第一章 安装OpenResty(Nginx+Lua)开发环境

发表时间: 2015-02-17 关键字: nginx, lua, ngx_lua, openresty

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。

1、创建目录/usr/servers，以后我们把所有软件安装在此目录

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

2、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）

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

3、下载ngx_openresty-1.7.7.2.tar.gz并解压

```
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。

3、安装LuaJIT

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

4、下载ngx_cache_purge模块，该模块用于清理nginx缓存

```
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
```

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

```
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
```

6、安装ngx_openresty

```
cd /usr/servers/ngx_openresty-1.7.7.2
./configure --prefix=/usr/servers --with-http_realip_module  --with-pcre  --with-luajit --add-m
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

8、到/usr/servers目录下

```
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

7、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://wiki.nginx.org/HttpLuaModule。

1、编辑nginx.conf配置文件

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

2、在http部分添加如下配置

```
#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
```

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

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

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

```
include lua.conf;
```

5、测试是否正常

```
/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

1、在lua.conf中server部分添加如下配置

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

2、测试配置是否正确

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

3、重启nginx

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

4、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

5、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。

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

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

然后lua.conf修改为

```
    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。

6、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。

```
    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

7、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。

```
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)

```
#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

```
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。

附件下载:

- nginx.zip (118.5 KB)
- dl.iteye.com/topics/download/de2966ff-790b-3834-90b4-78a46974f56e

1.2 第二章 OpenResty(Nginx+Lua)开发入门

发表时间: 2015-02-22 关键字: nginx, lua, ngx_lua, openresty

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

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

Lua5.3 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

1、example.conf配置文件

```
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";
}
```

2、test_request.lua

```
--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

```
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

1.1、example.conf配置文件

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

1.2、test_response_1.lua

```
--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。

2.1、example.conf配置文件

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

2.2、test_response_2.lua

```
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

1、example.conf配置文件

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

2、test_other.lua

```
--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。

1、首先在nginx.conf的http部分分配内存大小

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

2、example.conf配置文件

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

3、 test_lua_shared_dict.lua

```
--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_by_lua_file	access tail	http,server,location,location if	请求访问阶段处理，用于访问控制
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。

1、nginx.conf配置文件中的http部分添加如下代码

```
    #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;
```

2、init.lua

```
--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)
```

3、test.lua

```
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")
```

4、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。

1、nginx.conf配置文件中的http部分添加如下代码

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

2、init_worker.lua

```
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；

1.1、example.conf配置文件

```
    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；

1.2、test_set_1.lua

```
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

2.1、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

```
############ 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

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

但是我们想把商品编号为为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

```
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：

```
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。

1.1、example.conf配置文件

```
    location /lua_rewrite_1 {
        default_type "text/html";
        rewrite_by_lua_file /usr/example/lua/test_rewrite_1.lua;
        echo "no rewrite";
    }
```

1.2、test_rewrite_1.lua

```
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。

2.1、example.conf配置文件

```
    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";
    }
```

2.2、test_rewrite_2.lua

```
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

3.1、example.conf配置文件

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

3.2、test_rewrite_3.lua

```
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

```
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;
```

1.1、example.conf配置文件

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

1.2、test_access.lua

```
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。

1.3 第三章 Redis/SSDB+Twemproxy安装与使用

发表时间: 2015-02-26 关键字: reids, ssdb, twemproxy

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

1、下载redis并安装

```
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。

2、后台启动Redis服务器

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

3、查看是否启动成功

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

4、进入客户端

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

5、执行如下命令

```
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

1、下载SSDB并安装

```
#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
```

2、后台启动SSDB服务器

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

3、查看是否启动成功

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

4、进入客户端

```
/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

5、执行如下命令

```
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/ssdb。

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

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

1、下载Twemproxy并安装

```
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。

2、配置

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

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

3、启动Twemproxy代理

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

-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。

4、查看是否启动成功

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

5、进入客户端

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

6、执行如下命令

```
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

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

Redis内存

```
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128GB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。

```
#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
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，而每次设置的数据都特别大而且速度特别快（大量设置html片段）那么就需要加大repl-backlog-size。

主从示例

```
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

```
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

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

进入从客户端，挂主

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

```
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

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

```
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

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

```
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

```
#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 “”
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。

```
#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
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)

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

设置新的maxmemory(20mb)

```
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

```
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

```
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

```
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

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

```
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

```
/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.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。

```
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。

```
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。

1.4 第五章 常用Lua开发库3-模板渲染

发表时间: 2015-03-01 关键字: nginx, lua, ngx_lua, openresty, lua-resty-template

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

```
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
```

接下来就可以通过如下代码片段引用了

```
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

```
#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

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

其顺序是

```
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

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

example.conf配置server部分

```
    #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

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

内容为

```
template2
```

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

内容为

```
template1
```

test_temlate_1.lua

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

example.conf配置文件

```
    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。

1、test_template_2.lua

```
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。

2、examle.conf配置文件

```
    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

1、test_template_3.lua

```
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。

2、模板文件/usr/example/templates/t3.html

```
{(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。

3、example.conf配置文件

```
    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。

1.5 第六章 Web开发实战1——HTTP服务

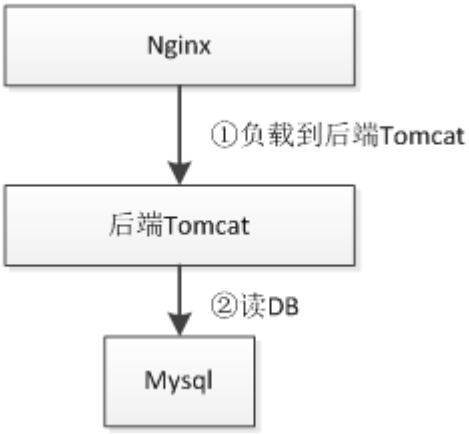
发表时间: 2015-03-02 关键字: nginx, lua, ngx_lua, openresty

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

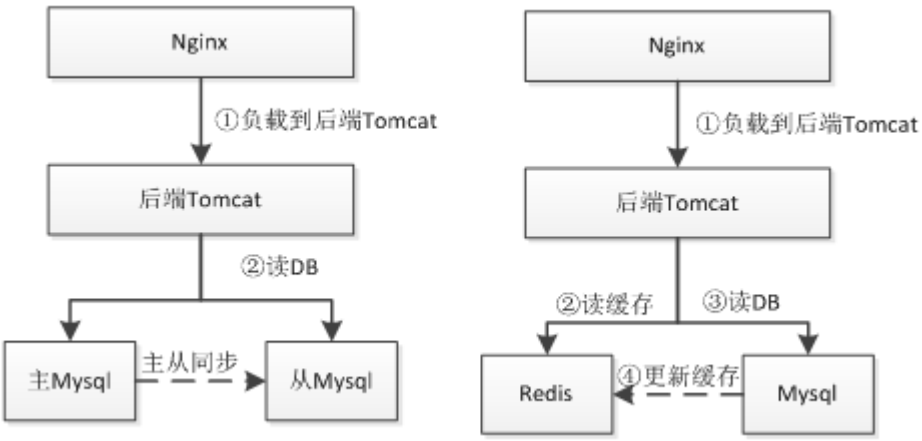
此处介绍下我曾使用过Nginx+JavaEE的架构。

1、单DB架构



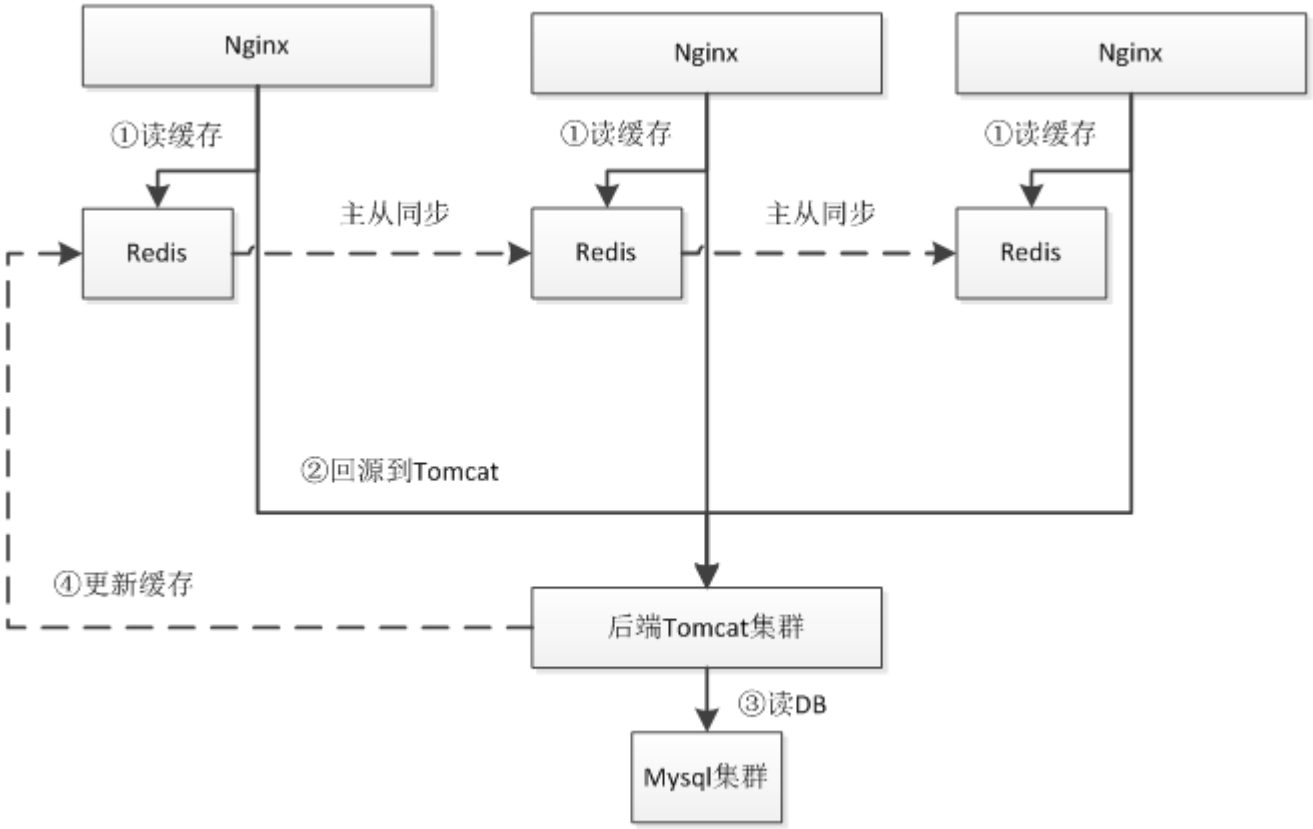
早期架构可能就是Nginx直接upstream请求到后端Tomcat，扩容时基本是增加新的Tomcat实例，然后通过Nginx负载均衡upstream过去。此时数据库还不是瓶颈。当访问量到一定级别，数据库的压力就上来了，此处单纯的靠单个数据库可能扛不住了，此时可以通过数据库的读写分离或加缓存来实现。

2、DB+Cache/数据库读写分离架构



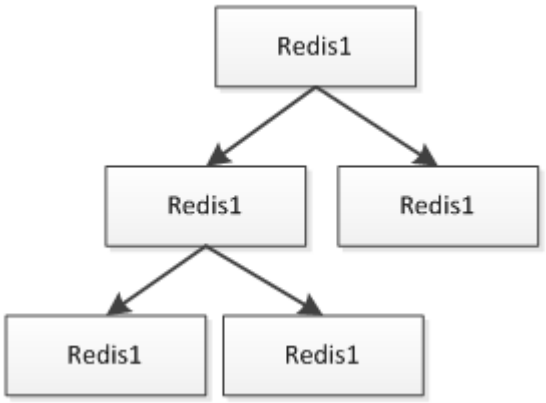
此时就通过使用如数据库读写分离或者Redis这种缓存来支撑更大的访问量。使用缓存这种架构会遇到的问题诸如缓存与数据库数据不同步造成数据不一致（一般设置过期时间），或者如Redis挂了，此时会直接命中数据库导致数据库压力过大；可以考虑Redis的主从或者一致性Hash 算法做分片的Redis集群；使用缓存这种架构要求应用对数据的一致性要求不是很高；比如像下订单这种要落地的数据不适合用Redis存储，但是订单的读取可以使用缓存。

3、Nginx+Lua+Local Redis+Mysql集群架构



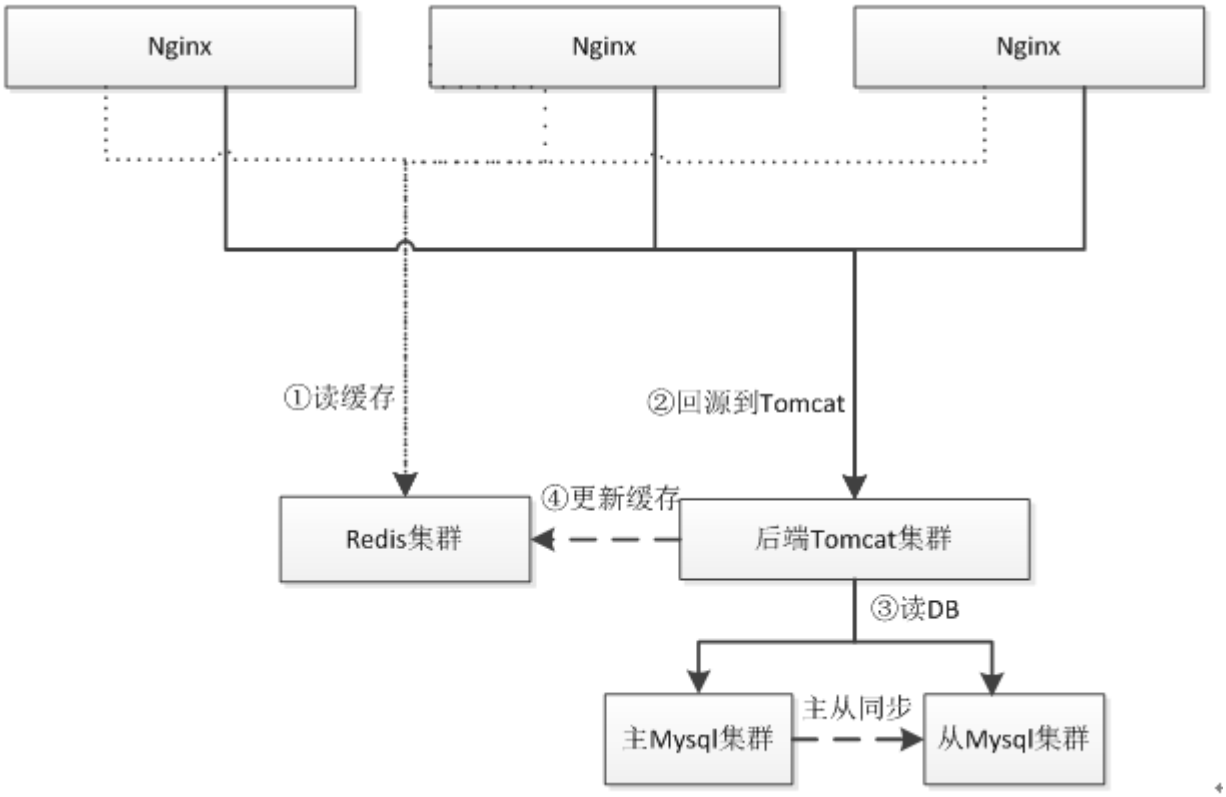
首先Nginx通过Lua读取本机Redis缓存，如果不命中才回源到后端Tomcat集群；后端Tomcat集群再读取Mysql数据库。Redis都是安装到和Nginx同一台服务器，Nginx直接读本机可以减少网络延时。Redis通过主从方式同步数据，

Redis主从一般采用树的方式实现：



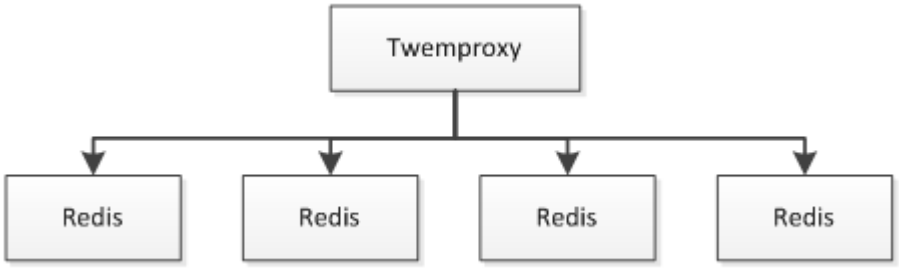
在叶子节点可以做AOF持久化，保证在主Redis挂时能进行恢复；此处假设对Redis很依赖的话，可以考虑多主Redis架构，而不是单主，来防止单主挂了时数据的不一致和击穿到后端Tomcat集群。这种架构的缺点就是要求Redis实例数据量较小，如果单机内存不足以存储这么多数据，当然也可以通过如尾号为1的在A服务器，尾号为2的在B服务器这种方式实现；缺点也很明显，运维复杂、扩展性差。

4、Nginx+Lua+ Redis集群+Mysql????



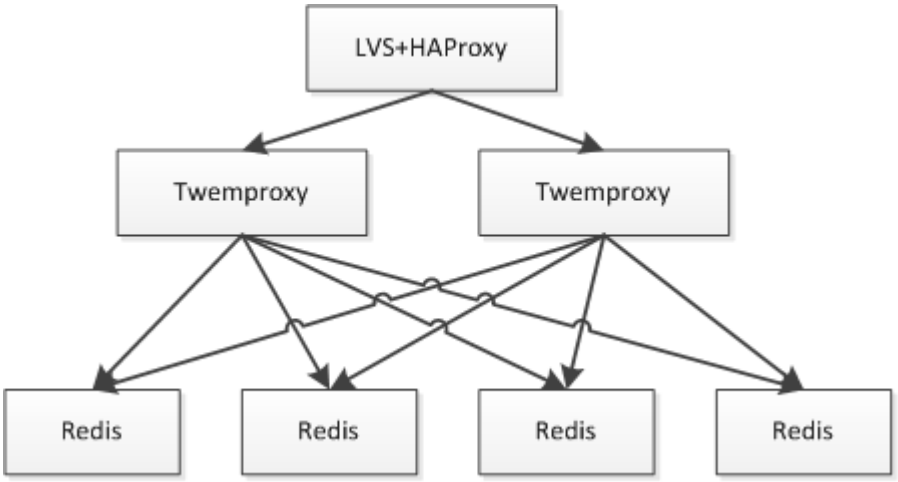
和之前架构不同的点是此时我们使用一致性Hash算法实现Redis集群而不是读本机Redis，保证其中一台挂了，只有很少的数据会丢失，防止击穿到数据库。Redis集群分片可以使用Twemproxy；如果 Tomcat实例很多的话，此时就要考虑Redis和Mysql链接数问题，因为大部分Redis/Mysql客户端都是通过连接池实现，此时的链接数会成为瓶颈。一般方

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。



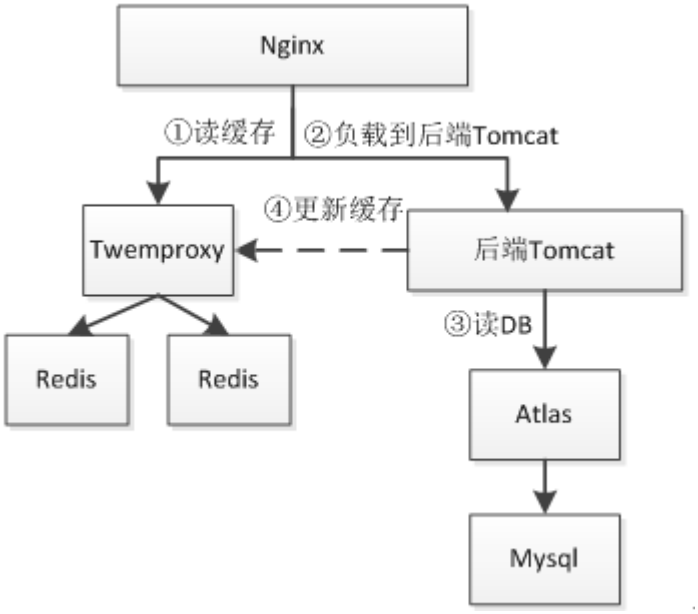
本文没有涉及Nginx之上是如何架构的，对于Nginx、Redis、Mysql等的负载均衡、资源的CDN化不是本文关注的点，有兴趣可以参考

很早的Taobao CDN架构

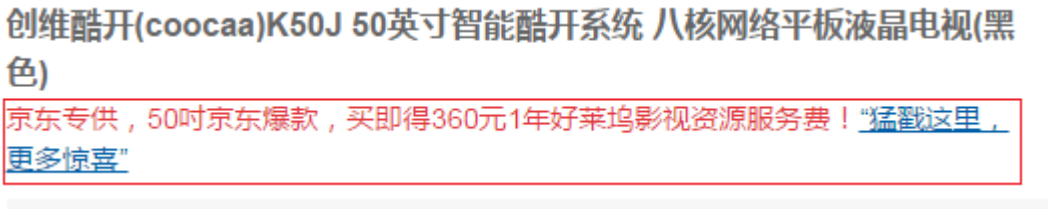
Nginx/LVS/HAProxy负载均衡软件的优缺点详解

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。

```
/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

```
#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

```
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

```
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、分库（sharding版本）、读写分离、数据库连接池等功能，缺点是没有实现跨库分表功能，需要在客户端使用分库逻辑，目前Atlas不活跃。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。

```
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

```
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

```
vim /usr/local/mysql-proxy/conf/chapter6.cnf
```

```
[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

```
/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

```
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

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

```
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.md

Java+Tomcat安装

Java安装

```
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

```
cd /usr/servers/
wget http://ftp.cuhk.edu.hk/pub/packages/apache.org/tomcat/tomcat-7/v7.0.59/bin/apache-tomcat-7
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

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

```
#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。

```
#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
```

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

启动两个Tomcat

```
/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。

```
<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

```
public class AdServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletExcept
        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&characterEncodi
        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

```
  <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

```
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 chapter6.war解压。

测试

启动Tomcat实例，分别访问如下地址将看到广告内容：

```
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

```
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

```
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

```
    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

```
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

```
    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

```
/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：

```
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/HAProxy做负载均衡、使用CDN等可以查找资料学习。

附件下载:

- chapter6.zip (2.8 MB)
- dl.iteye.com/topics/download/d2dddc19-5b40-3b33-88dc-0263d8b84bc4

1.6 第七章 Web开发实战2——商品详情页

发表时间: 2015-03-03 关键字: nginx, lua, ngx_lua, openresty

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

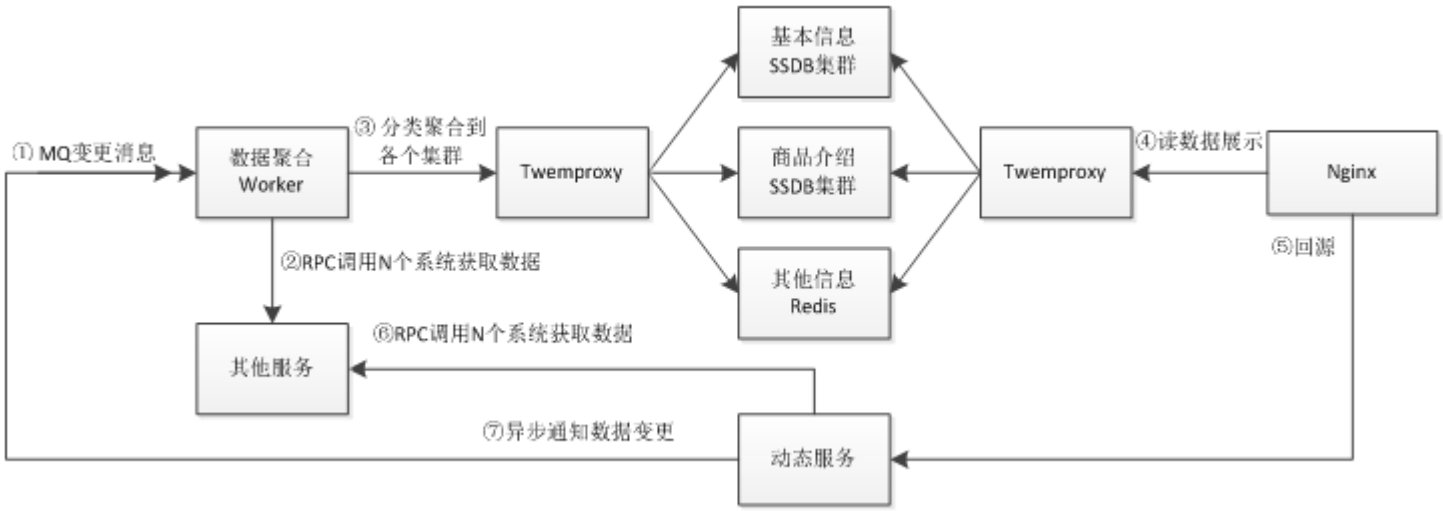
通过AJAX技术惰性加载（因为是第二屏，只有当用户滚动鼠标到该屏时才显示）；而实时展示数据通过AJAX技术做异步加载；因此我们可以做如下设计：

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

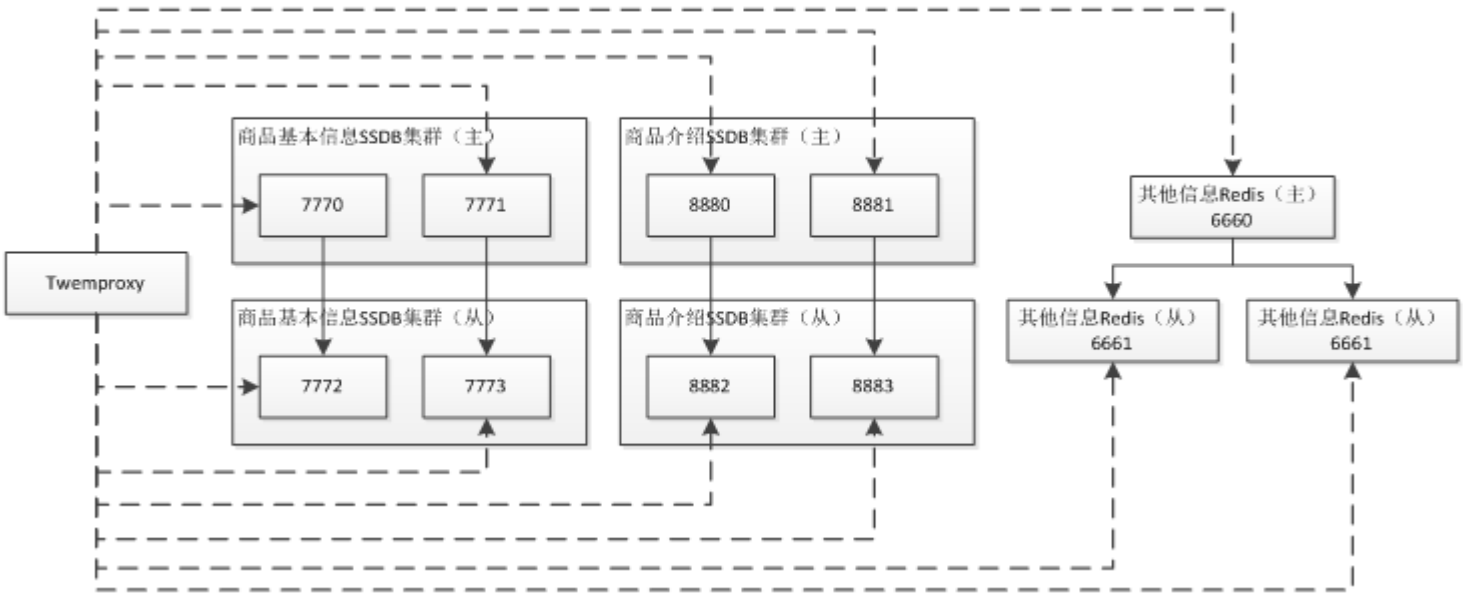
WEB-INF

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

```
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

```
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

```
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

```
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.md。

创建工作目录

```
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

```
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

```
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

```
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

```
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

```
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.md。

创建工作目录

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

启动

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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

```
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：

```
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。

```
<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

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletExcept
    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

```
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

```
    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
        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

```
    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-lIAAHWawP_QjwAAAVF4
        map.put("breadcrumb", breadcrumb);
        map.put("brand", brand);
        //实际应用应该是发送MQ
        asyncSetToRedis(otherInfoJedisPool, "s:" + ps3Id + ":", objectMapper.writeValueAsString
        asyncSetToRedis(otherInfoJedisPool, "b:" + brandId + ":", objectMapper.writeValueAsStri
        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

```
    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 cont
        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

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

打WAR包

```
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：

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

```
<!-- 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。

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

访问如下URL进行测试。

```
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

```
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

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

```
#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

```
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

```
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

```
    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

```
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

```
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"]
         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"] .. '
         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

```
        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","isWeCha
                 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 ex
                 {-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

```
<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'] *}</
         &gt; 
       {% end %}
       <a href='http://item.jd.com/{* skuId *}.html'>{* moreName *}</a>
    </span>
</div>
```

图片列表

```
<div id="spec-n1" class="jqzoom" onclick="window.open('http://www.jd.com/bigimage.aspx?id={* sk
    <img data-img="1" width="350" height="350" src="http://img14.360buyimg.com/n1/{* imgs[1] *}
</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/{* i
            {% end %}
        </ul>
    </div>
</div>
```

颜色尺码选择

```
<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'] *}">
        {% 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'] *}">{*
        {% end %}
    </div>
</div>
```

扩展属性

```
<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 *}-{* bra
    {% 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

```
<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

```
    #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

```
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。

1、nginx.conf配置

```
    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
    proxy_cache_path  /usr/servers/nginx/proxy_cache levels=1:2 keys_zone=cache_item:200m inact
```

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。

2、nginx_chapter7.conf配置

与server指令配置同级

```
############ 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。

```
    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
        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

```
    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。

3、修改item.lua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

```
    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。

1.7 第八章 流量复制/AB测试/协程

发表时间: 2015-03-07 关键字: nginx, lua, coroutine, ngx_lua, openresty

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

```
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

```
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。

1、使用map选择版本

```
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。

```
    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

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

2、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

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

```
    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。

1、依赖的API

```
    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。

2、串行实现

```
    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

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

3、ngx.location.capture_multi实现

```
    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

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

4、协程API实现

```
    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

```
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。

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

Lua协程参考资料

《Programming in Lua》

http://timyang.net/lua/lua-coroutine-vs-java-wait-notify/

https://github.com/andycai/luaprimer/blob/master/05.md

http://my.oschina.net/wangxuanyihaha/blog/186401

http://manual.luaer.cn/2.11.html