**畅购电商系统开发**

**第4天**

传智播客.黑马程序员.深圳

# 学习目标

* **Lua介绍**

Lua语法 输出、变量定义、数据类型、流程控制(if..)、循环操作、函数、表(数组)、模块

* **OpenResty介绍(理解配置)**

封装了Nginx，并且提供了Lua扩展，大大提升了Nginx对并发处理的能，10K-1000K

Lua->广告缓存操作

* **广告缓存载入与读取**
* **Nginx讲解**

限流操作:漏斗限流原理

1.控制速率

2.并发量控制

* **Canal讲解**

实现数据同步操作->MySQL

* **Canal实现首页缓存同步**

# 首页分析

首页门户系统需要展示各种各样的广告数据。如图，以jd为例：

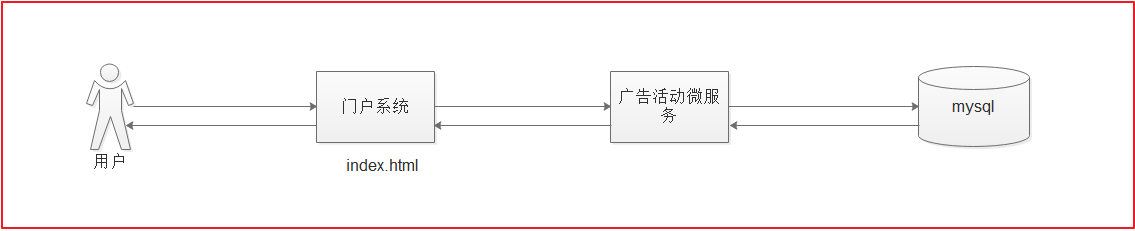


变更频率低的数据，如何提升访问速度？

1.数据做成静态页[商品详情页]

2.做缓存[Redis]

基本的思路如下：



如上图此种方式 简单，直接通过数据库查询数据展示给用户即可，但是通常情况下，首页（门户系统的流量一般非常的高）不适合直接通过mysql数据库直接访问的方式来获取展示。

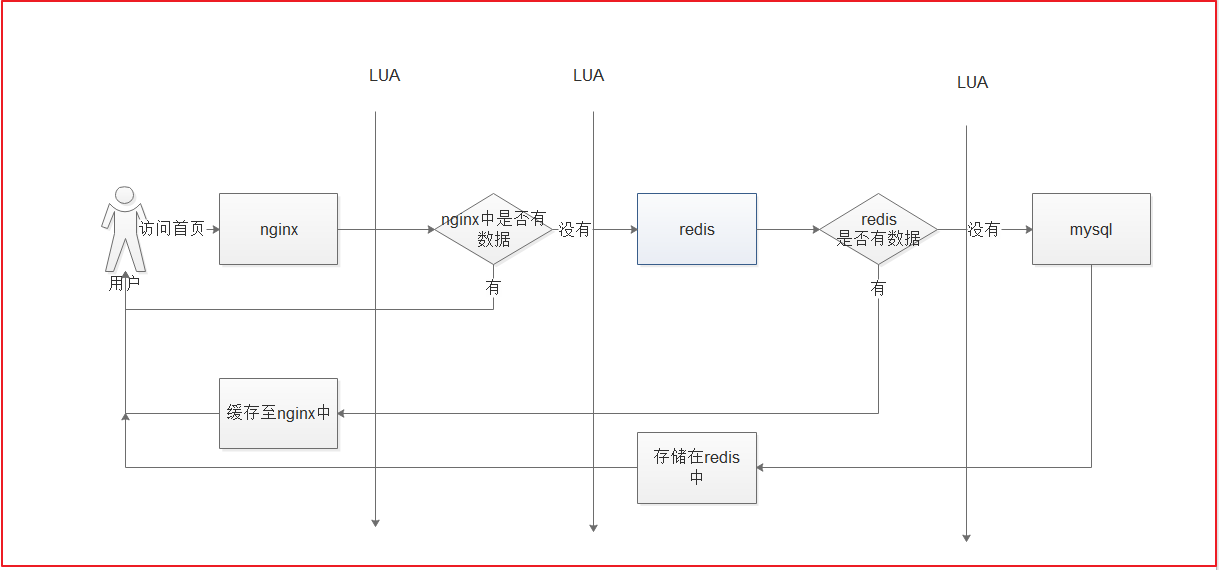
如下思路：

1.首先访问nginx ，我们可以采用缓存的方式，先从nginx本地缓存中获取，获取到直接响应

2.如果没有获取到，再次访问redis，我们可以从redis中获取数据，如果有 则返回，并缓存到nginx中

3.如果没有获取到,再次访问mysql,我们从mysql中获取数据，再将数据存储到redis中，返回。

而这里面，我们都可以使用LUA脚本嵌入到程序中执行这些查询相关的业务。



畅购的首页广告数据库为：changgou\_content，数据表分别为：tb\_content、tb\_content\_category

# Lua(了解)

## lua是什么

Lua [1] 是一个小巧的脚本语言。它是巴西里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro）里的一个由Roberto Ierusalimschy、Waldemar Celes 和 Luiz Henrique de Figueiredo三人所组成的研究小组于1993年开发的。 其设计目的是为了通过灵活嵌入应用程序中从而为应用程序提供灵活的扩展和定制功能。Lua由标准C编写而成，几乎在所有操作系统和平台上都可以编译，运行。Lua并没有提供强大的库，这是由它的定位决定的。所以Lua不适合作为开发独立应用程序的语言。Lua 有一个同时进行的JIT项目，提供在特定平台上的即时编译功能。

简单来说：

Lua 是一种轻量小巧的脚本语言，用标准C语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

## 特性

* 支持面向过程(procedure-oriented)编程和函数式编程(functional programming)；
* 自动内存管理；只提供了一种通用类型的表（table），用它可以实现数组，哈希表，集合，对象；
* 语言内置模式匹配；闭包(closure)；函数也可以看做一个值；提供多线程（协同进程，并非操作系统所支持的线程）支持；
* 通过闭包和table可以很方便地支持面向对象编程所需要的一些关键机制，比如数据抽象，虚函数，继承和重载等。

## 应用场景

* 游戏开发
* 独立应用脚本
* Web 应用脚本
* 扩展和数据库插件如：MySQL Proxy 和 MySQL WorkBench
* 安全系统，如入侵检测系统
* redis中嵌套调用实现类似事务的功能
* web容器中应用处理一些过滤 缓存等等的逻辑，例如nginx。

## lua的安装

本课程中已经在提供的虚拟机中装好了lua软件，下载目录为：/usr/loacl/server

有linux版本的安装也有mac版本的安装。我们采用linux版本的安装，首先我们准备一个linux虚拟机。

安装步骤,在linux系统中执行下面的命令。

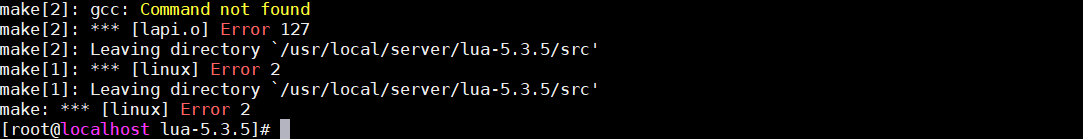
curl -R -O **http**://www.lua.org/**ftp**/lua-5.3.5.tar.gz

tar zxf lua-5.3.5.tar.gz

cd lua-5.3.5

make linux test

注意：此时安装，有可能会出现如下错误：



此时需要安装lua相关依赖库的支持，执行如下命令即可：

yum **install** libtermcap-devel ncurses-devel libevent-devel readline-devel

此时再执行lua测试看lua是否安装成功

[**root@localhost** ~]# lua

Lua 5.1.4 Copyright (**C**) 1994-2008 Lua.org, PUC-Rio

## 入门程序

创建hello.lua文件，内容为

编辑文件hello.lua

**vi** hello.**lua**

在文件中输入：

print("hello");

保存并退出。

执行命令

**lua** hello.**lua**

输出为：

**Hello**

效果如下：



## LUA的基本语法

lua有交互式编程和脚本式编程。

交互式编程就是直接输入语法，就能执行。

脚本式编程需要编写脚本，然后再执行命令 执行脚本才可以。

一般采用脚本式编程。（例如：编写一个hello.lua的文件，输入文件内容，并执行lua hell.lua即可）

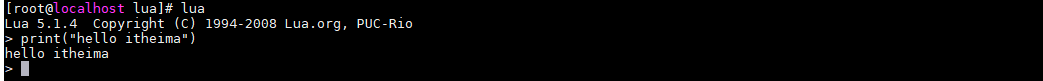
**(1)交互式编程**

Lua 提供了交互式编程模式。我们可以在命令行中输入程序并立即查看效果。

Lua 交互式编程模式可以通过命令 lua -i 或 lua 来启用：

**lua** -**i**

如下图：



**(2)脚本式编程**

我们可以将 Lua 程序代码保持到一个以 lua 结尾的文件，并执行，该模式称为脚本式编程，例如上面入门程序中将lua语法写到hello.lua文件中。

### 注释

**一行注释**：两个减号是单行注释:

--

**多行注释**：

--[[

多行注释

多行注释

--]]

### 定义变量

全局变量，默认的情况下，定义一个变量都是全局变量，

如果要用局部变量 需要声明为local.例如：

-- 全局变量赋值

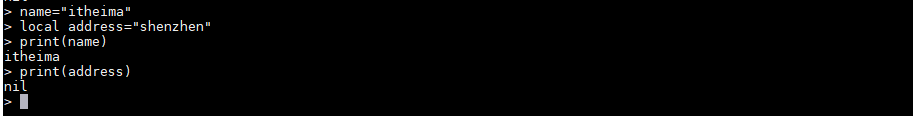
a=1

-- 局部变量赋值

**local** b=2

如果变量没有初始化：则 它的值为nil 这和java中的null不同。

如下图案例：



### Lua中的数据类型

Lua 是动态类型语言，变量不要类型定义,只需要为变量赋值。 值可以存储在变量中，作为参数传递或结果返回。

Lua 中有 8 个基本类型分别为：nil、boolean、number、string、userdata、function、thread 和 table。

| **数据类型** | **描述** |
| --- | --- |
| nil | 这个最简单，只有值nil属于该类，表示一个无效值（在条件表达式中相当于false）。 |
| boolean | 包含两个值：false和true。 |
| number | 表示双精度类型的实浮点数 |
| string | 字符串由一对双引号或单引号来表示 |
| function | 由 C 或 Lua 编写的函数 |
| userdata | 表示任意存储在变量中的C数据结构 |
| thread | 表示执行的独立线路，用于执行协同程序 |
| table | Lua 中的表（table）其实是一个“关联数组”（associative arrays），数组的索引可以是数字、字符串或表类型。在 Lua 里，table 的创建是通过“构造表达式”来完成，最简单构造表达式是{}，用来创建一个空表。 |

**实例：**

print(type("Hello world")) --> string

print(type(10.4\*3)) --> number

print(type(print)) --> function

print(type(type)) --> function

print(type(**true**)) --> boolean

print(type(**nil**)) --> nil

### 流程控制

**(1)if语句**

Lua **if 语句** 由一个布尔表达式作为条件判断，其后紧跟其他语句组成。

语法：

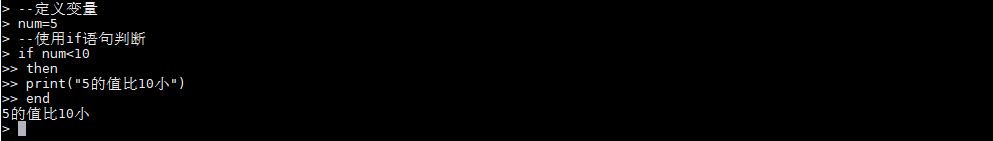
**if**(布尔表达式)

**then**

--[ 在布尔表达式为 true 时执行的语句 --]

**end**

实例：



**(2)if..else语句**

Lua if 语句可以与 else 语句搭配使用, 在 if 条件表达式为 false 时执行 else 语句代码块。

语法：

**if**(布尔表达式)

**then**

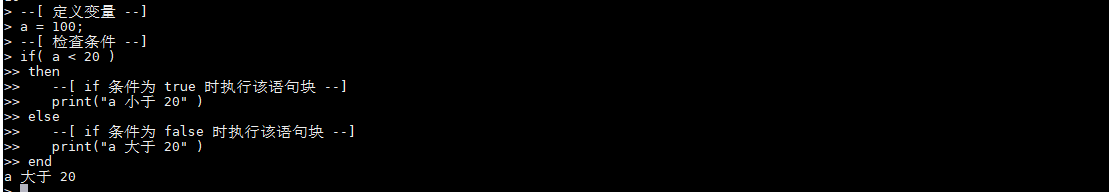
--[ 布尔表达式为 true 时执行该语句块 --]

**else**

--[ 布尔表达式为 false 时执行该语句块 --]

**end**

实例：



### 循环

**学员完成**

**(1)while循环[==满足条件就循环==]**

Lua 编程语言中 while 循环语句在判断条件为 true 时会重复执行循环体语句。  
语法：

**while**(condition)

do

statements

**end**

实例：

a=10

**while**( a < 20 )

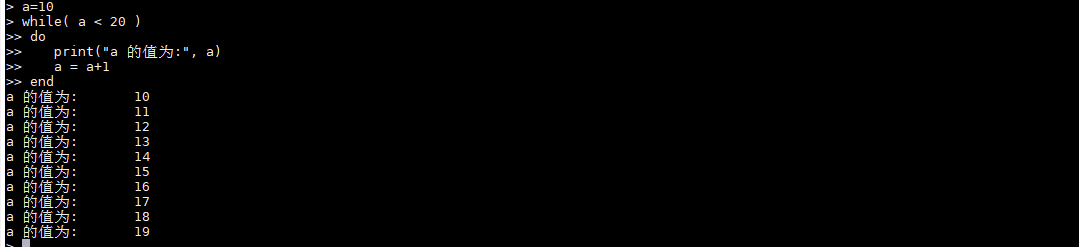
do

print("a 的值为:", a)

a = a+1

**end**

效果如下：



**(2)for循环**

Lua 编程语言中 for 循环语句可以重复执行指定语句，重复次数可在 for 语句中控制。

语法： 1->10 1:exp1 10:exp2 2:exp3:递增的数量

**for** var=exp1,exp2,exp3

do

<执行体>

**end**

var 从 exp1 变化到 exp2，每次变化以 exp3 为步长递增 var，并执行一次 **“执行体”**。exp3 是可选的，如果不指定，默认为1。

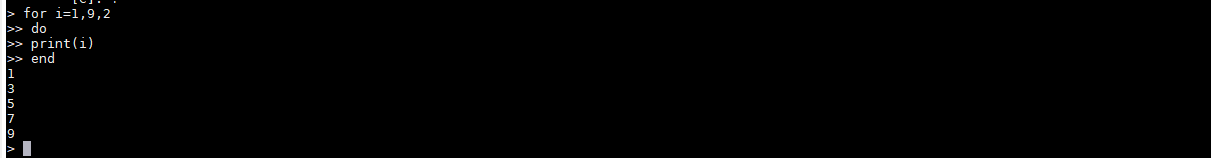
例子：for i=1,9,2 :i=1从1开始循环，9循环数据到9结束，2每次递增2

**for** i=1,9,2

**do**

**print**(i)

end



**(3)repeat…until语句[==满足条件结束==]**

Lua 编程语言中 repeat…until 循环语句不同于 for 和 while循环，for 和 while 循环的条件语句在当前循环执行开始时判断，而 repeat…until 循环的条件语句在当前循环结束后判断。

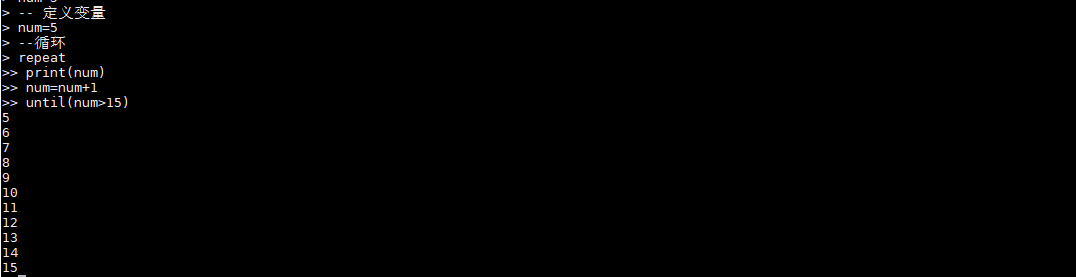
语法：

repeat

statements

**until**( condition )

案例：



### 函数

lua中也可以定义函数，类似于java中的方法。例如：

..表示字符拼接 ,表示隔开输出

--[[ 函数返回两个值的最大值 --]]

**function** **max**(num1, num2)

**if** (num1 > num2) **then**

result = num1;

**else**

result = num2;

**end**

**return** result;

**end**

-- 调用函数

print("两值比较最大值为 ",max(10,4))

print("两值比较最大值为 "..max(5,6))

执行之后的结果：

两值比较最大值为 10

两值比较最大值为 6

### 表

table 是 Lua 的一种数据结构用来帮助我们创建不同的数据类型，如：数组、字典等。

Lua也是通过table来解决模块（module）、包（package）和对象（Object）的。

案例：

-- 初始化表

mytable = {}

-- 指定值

mytable[1]= "Lua"

-- 移除引用

mytable = nil

### 模块

**(1)模块定义(定义模块,被调用)**

模块类似于一个封装库，从 Lua 5.1 开始，Lua 加入了标准的模块管理机制，可以把一些公用的代码放在一个文件里，以 API 接口的形式在其他地方调用，有利于代码的重用和降低代码耦合度。

创建一个文件叫module.lua，在module.lua中创建一个独立的模块，代码如下：

-- 文件名为 module.lua

-- 定义一个名为 module 的模块

module = {}

-- 定义一个常量

module.constant = "这是一个常量"

-- 定义一个函数

**function** **module.func1**()

print("这是一个公有函数")

**end**

**local** **function** **func2**()

print("这是一个私有函数！")

**end**

**function** **module.func3**()

func2()

**end**

**return** module

由上可知，模块的结构就是一个 table 的结构，因此可以像操作调用 table 里的元素那样来操作调用模块里的常量或函数。

上面的 func2 声明为程序块的局部变量，即表示一个私有函数，因此是不能从外部访问模块里的这个私有函数，必须通过模块里的公有函数来调用.

**(2)require 函数(引入其他,模块)**

require 用于 引入其他的模块，类似于java中的类要引用别的类的效果。

用法：

**require**("<模块名>")

require "<模块名>"

两种都可以。

我们可以将上面定义的module模块引入使用,创建一个test\_module.lua文件，代码如下：

-- test\_module.lua 文件

-- **module** 模块为上文提到到 **module**.**lua**

**require**("module")

print(**module**.**constant**)

**module**.**func3**()

# OpenResty介绍

OpenResty(又称：ngx\_openresty) 是一个基于 nginx的可伸缩的 Web 平台，由中国人章亦春发起，提供了很多高质量的第三方模块。

OpenResty是一个基于 Nginx 与 Lua 的高性能 Web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。

OpenResty通过汇聚各种设计精良的 Nginx 模块（主要由 OpenResty 团队自主开发），从而将 Nginx 有效地变成一个强大的通用 Web 应用平台。这样，Web 开发人员和系统工程师可以使用 Lua 脚本语言调动 Nginx 支持的各种 C 以及 Lua 模块，快速构造出足以胜任 10K 乃至 1000K 以上单机并发连接的高性能 Web 应用系统。

OpenResty的目标是让你的Web服务直接跑在 Nginx 服务内部，充分利用 Nginx 的非阻塞 I/O 模型，不仅仅对 HTTP 客户端请求,甚至于对远程后端诸如 MySQL、PostgreSQL、Memcached 以及 Redis 等都进行一致的高性能响应。

360，UPYUN，阿里云，新浪，腾讯网，去哪儿网，酷狗音乐等都是 OpenResty 的深度用户。

OpenResty 简单理解成 就相当于封装了nginx,并且集成了LUA脚本，开发人员只需要简单的其提供了模块就可以实现相关的逻辑，而不再像之前，还需要在nginx中自己编写lua的脚本，再进行调用了。

官网：<http://openresty.org/cn/>

## 安装openresty

linux安装openresty，本课程的虚拟机中已经装好了openresty

1.添加仓库执行命令

yum install yum-utils

yum-config-manager --add-repo https://openresty.org/package/centos/openresty.repo

2.执行安装

yum **install** openresty

3.安装成功后 会在默认的目录如下：

/usr/**local**/openresty

## 安装nginx

默认已经安装好了nginx,在目录：/usr/local/openresty/nginx 下。

修改/usr/local/openresty/nginx/conf/nginx.conf,将配置文件使用的根设置为root,目的就是将来要使用lua脚本的时候 ，直接可以加载在root下的lua脚本。

**cd** /usr/local/openresty/nginx/**conf**

**vi** nginx.**conf**

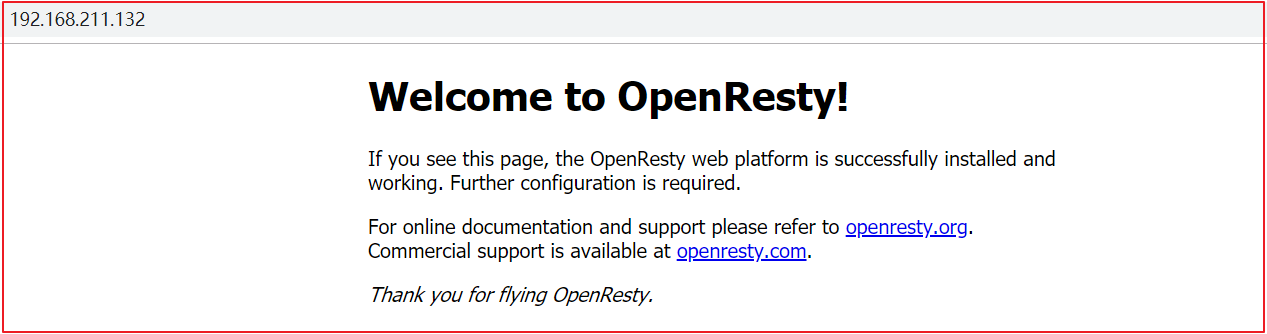
修改代码如下：



## 测试访问

重启下centos虚拟机，然后访问测试Nginx

访问地址：<http://192.168.211.132/>



# 广告缓存的载入与读取

## 需求分析

需要在页面上显示广告的信息。

## Lua+Nginx配置

### 查询数据放入redis中

实现思路：

a定义请求：用于查询数据库中的数据更新到redis中。

b.连接mysql ，按照广告分类ID读取广告列表，转换为json字符串。

c.连接redis，将广告列表json字符串存入redis 。

定义请求：

请求：

/update\_content

参数：

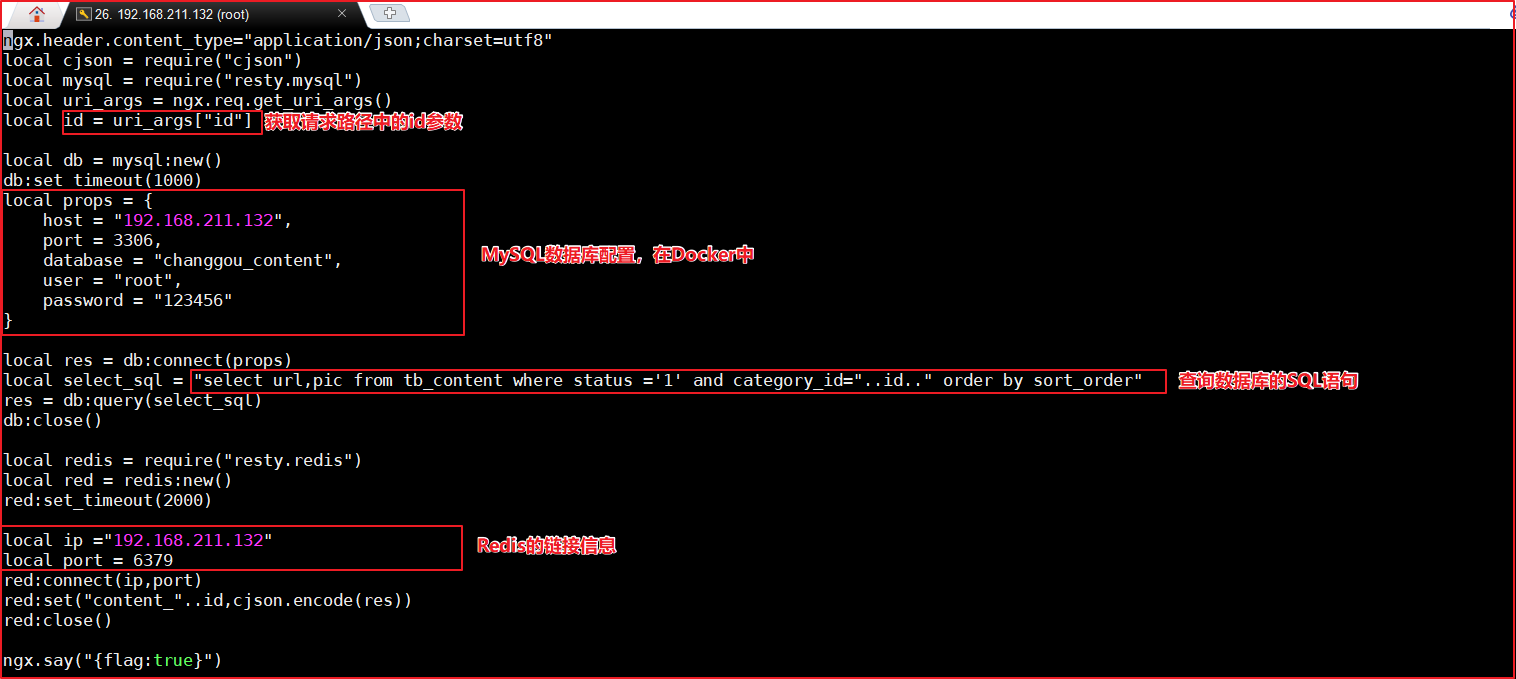
**id** --指定广告分类的**id**

返回值：

json

请求地址：<http://192.168.211.132/update_content?id=1>

创建/root/lua目录，在该目录下创建update\_content.lua： 目的就是连接mysql 查询数据 并存储到redis中。



上图代码如下：

ngx.header.content\_type="application/json;charset=utf8"

**local** cjson = require("cjson")

**local** mysql = require("resty.mysql")

**local** uri\_args = ngx.req.get\_uri\_args()

**local** id = uri\_args["id"]

**local** db = mysql:new()

db:set\_timeout(1000)

**local** props = {

host = "192.168.211.132",

port = 3306,

database = "changgou\_content",

user = "root",

password = "123456"

}

**local** res = db:connect(props)

**local** select\_sql = "select url,pic from tb\_content where status ='1' and category\_id="..id.." order by sort\_order"

res = db:query(select\_sql)

db:close()

**local** redis = require("resty.redis")

**local** red = redis:new()

red:set\_timeout(2000)

**local** ip ="192.168.211.132"

**local** port = 6379

red:connect(ip,port)

red:set("content\_"..id,cjson.encode(res))

red:close()

ngx.say("{flag:true}")

修改/usr/local/openresty/nginx/conf/nginx.conf文件： 添加头信息，和 location信息，代码如下：

**server** {

**listen** 80;

**server\_name** localhost;

**location** /update\_content {

#nginx的拦截，将/update\_content的请求拦截，交给update\_content.lua脚本执行

**content\_by\_lua\_file** /root/lua/update\_content.lua;

}

}

定义lua缓存命名空间，修改nginx.conf，添加如下代码即可：

代码如下：

lua\_shared\_dict dis\_cache 128m; 设置缓存大小

进入nignx的sbin目录下，执行：./nginx -s reload，重新加载nginx配置文件

请求 <http://192.168.211.132/update_content?id=1> 可以实现缓存的添加



### 从Redis缓存中获取广告数据

实现思路：

定义请求，用户根据广告分类的ID 获取广告的列表。通过lua脚本直接从redis中获取数据即可。

定义请求：<http://192.168.211.132/read_content?id=1>

请求:/read\_content

参数：id

返回值：json

在/root/lua目录下创建read\_content.lua

--设置响应头类型

ngx.header.content\_type="application/json;charset=utf8"

--获取请求中的参数ID

**local** uri\_args = ngx.req.get\_uri\_args();

**local** id = uri\_args["id"];

--引入redis库

**local** redis = require("resty.redis");

--创建redis对象

**local** red = redis:new()

--设置超时时间

red:set\_timeout(2000)

--连接

**local** ok, err = red:connect("192.168.211.132", 6379)

--获取key的值

**local** rescontent=red:get("content\_"..id)

--输出到返回响应中

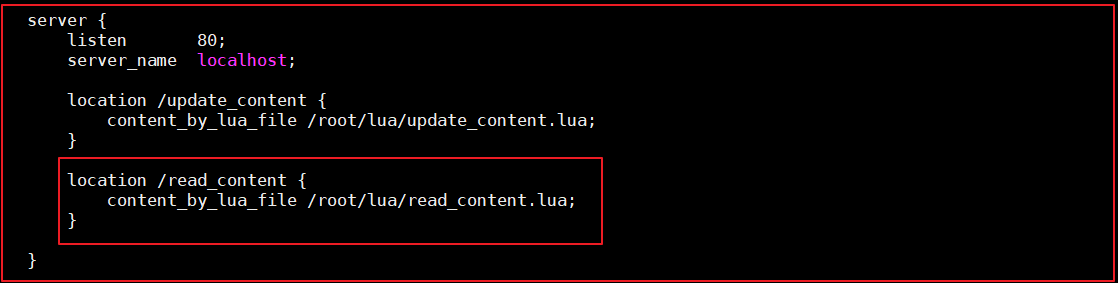
ngx.say(rescontent)

--关闭连接

red:close()

在/usr/local/openresty/nginx/conf/nginx.conf中配置如下：

如图：



代码：当遇到read\_content的请求时：就通过/root/lua 下的read\_content.lua lua脚本来处理这个请求

**location** /read\_content {

**content\_by\_lua\_file** /root/lua/read\_content.lua;

}

### 加入openresty本地缓存

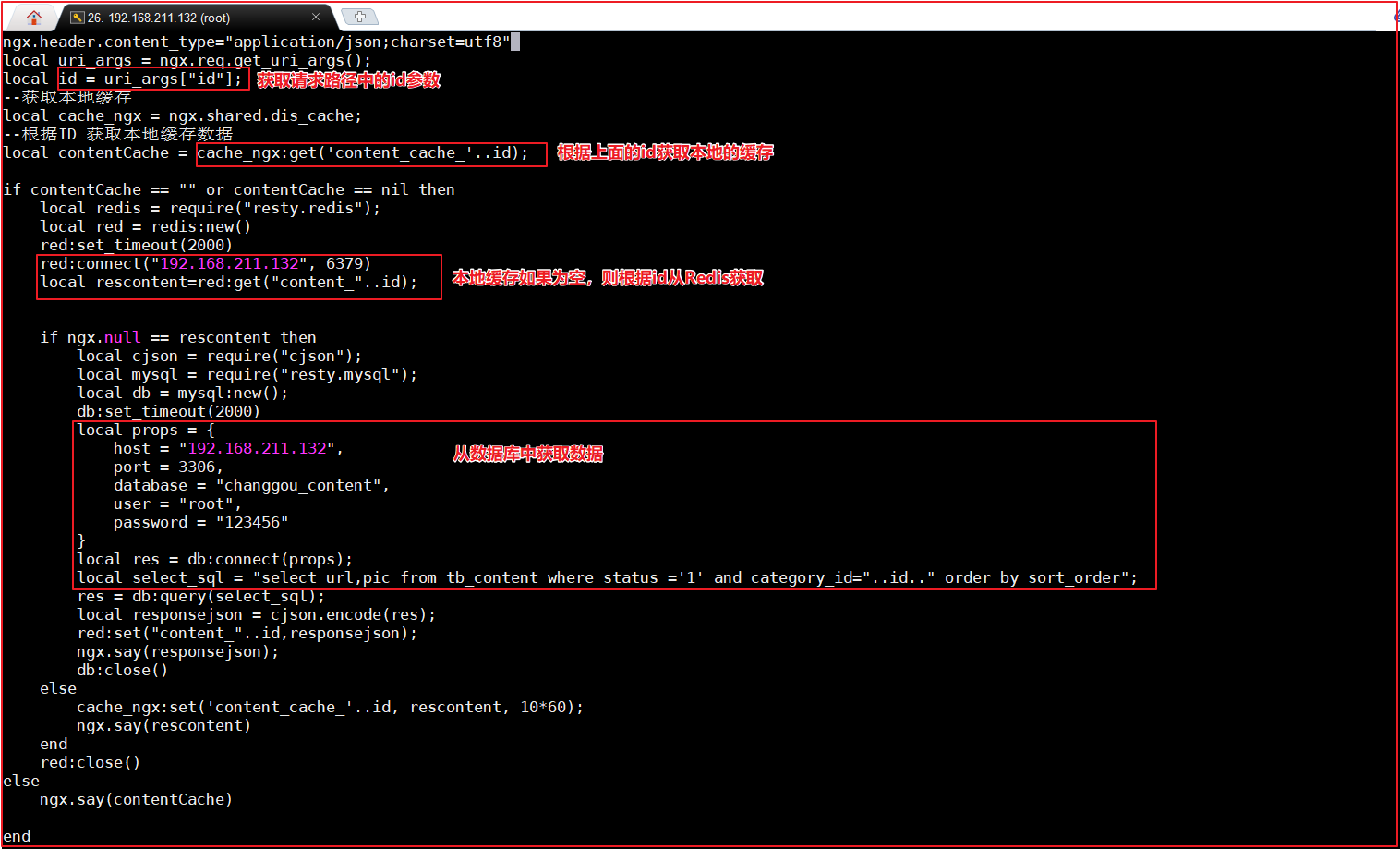
如上的方式没有问题，但是如果请求都到redis，redis压力也很大，所以我们一般采用多级缓存的方式来减少下游系统的服务压力。参考基本思路图的实现。

先查询openresty本地缓存 如果 没有

再查询redis中的数据，如果没有

再查询mysql中的数据，但凡有数据 则返回即可。

修改read\_content.lua文件，代码如下：



上图代码如下：这就是一整套多级缓存的逻辑

ngx.header.content\_type="application/json;charset=utf8"

**local** uri\_args = ngx.req.get\_uri\_args();

**local** id = uri\_args["id"];

--获取本地缓存

**local** cache\_ngx = ngx.shared.dis\_cache;

--根据ID 获取本地缓存数据

**local** contentCache = cache\_ngx:get('content\_cache\_'..id);

**if** contentCache == "" **or** contentCache == **nil** **then**

**local** redis = require("resty.redis");

**local** red = redis:new()

red:set\_timeout(2000)

red:connect("192.168.211.132", 6379)

**local** rescontent=red:get("content\_"..id);

**if** ngx.null == rescontent **then**

**local** cjson = require("cjson");

**local** mysql = require("resty.mysql");

**local** db = mysql:new();

db:set\_timeout(2000)

**local** props = {

host = "192.168.211.132",

port = 3306,

database = "changgou\_content",

user = "root",

password = "123456"

}

**local** res = db:connect(props);

**local** select\_sql = "select url,pic from tb\_content where status ='1' and category\_id="..id.." order by sort\_order";

res = db:query(select\_sql);

**local** responsejson = cjson.encode(res);

red:set("content\_"..id,responsejson); //redis保存数据

ngx.say(responsejson); // //nginx保存数据

db:close()

**else**

cache\_ngx:set('content\_cache\_'..id, rescontent, 10\*60);

ngx.say(rescontent) //nginx保存数据

**end**

red:close()

**else**

ngx.say(contentCache)

**end**

注意改完配置后，要让nginx重新reload一下才生效

1. 测试地址：<http://192.168.211.132/read_content?id=1>
2. 此时会获取分类ID=1的所有广告信息
3. 删除redis中的所有缓存
4. 再次访问<http://192.168.211.132/read_content?id=1>
5. 还是会获取分类ID=1的所有广告信息，因为有nginx缓存(有效时间10分钟)

# nginx限流

一般情况下，首页的并发量是比较大的，即使 有了多级缓存，当用户不停的刷新页面的时候，也是没有必要的，另外如果有恶意的请求 大量达到，也会对系统造成影响。

而限流就是保护措施之一。

## 生活中限流对比

* 水坝泄洪，通过闸口限制洪水流量（控制流量速度）。
* 办理银行业务：所有人先领号，各窗口叫号处理。每个窗口处理速度根据客户具体业务而定，所有人排队等待叫号即可。若快下班时，告知客户明日再来(拒绝流量)
* 火车站排队买票安检，通过排队 的方式依次放入。（缓存带处理任务）

## nginx的限流

nginx提供两种限流的方式：

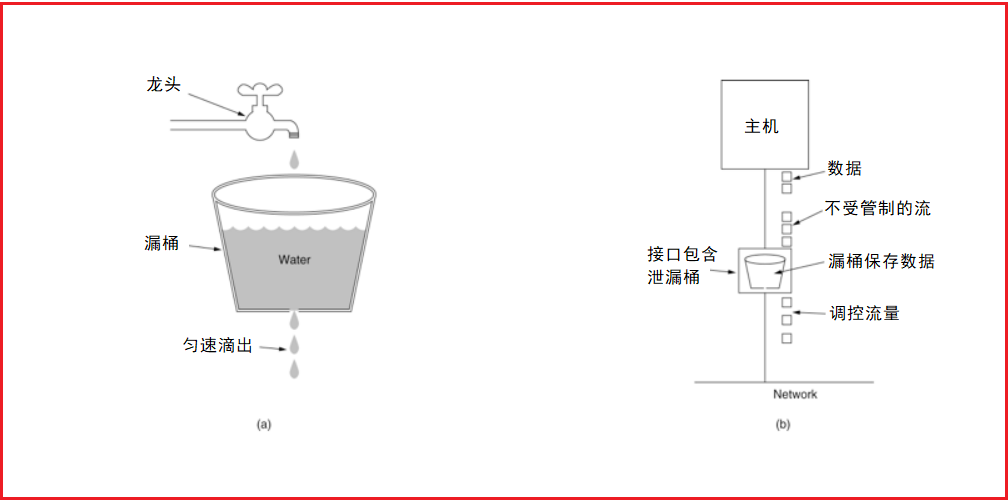
* 一是控制速率
* 二是控制并发连接数

### 控制速率

控制速率的方式之一就是采用漏桶算法。

#### 漏桶算法实现控制速率限流

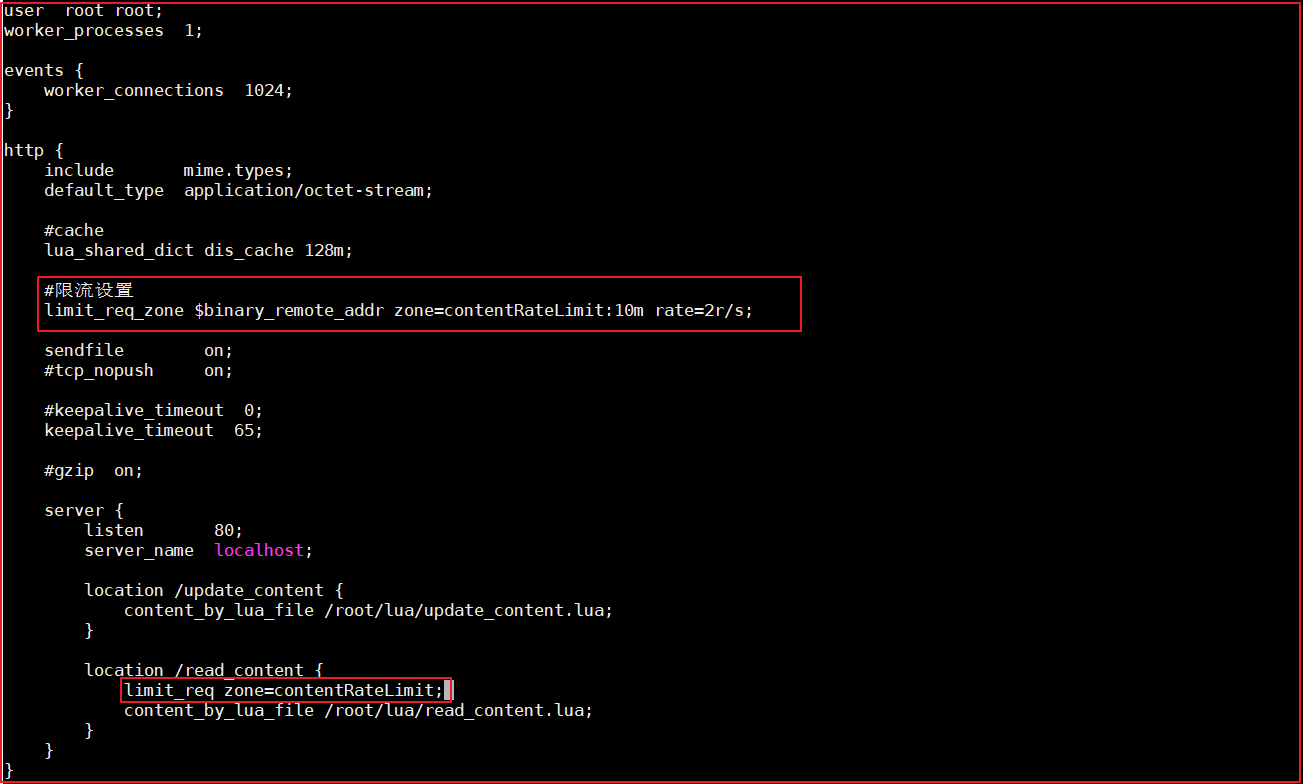
漏桶(Leaky Bucket)算法思路很简单,水(请求)先进入到漏桶里,漏桶以一定的速度出水(接口有响应速率),当水流入速度过大会直接溢出(访问频率超过接口响应速率),然后就拒绝请求,可以看出漏桶算法能强行限制数据的传输速率.示意图如下:



#### nginx的配置

nginx的限流

配置示意图如下：



修改/usr/local/openresty/nginx/conf/nginx.conf:

**user** root root;

**worker\_processes** 1;

**events** {

**worker\_connections** 1024;

}

**http** {

**include** mime.types;

**default\_type** application/octet-stream;

#cache

**lua\_shared\_dict** dis\_cache 128m;

#限流设置

**limit\_req\_zone** $binary\_remote\_addr zone=contentRateLimit:10m rate=2r/s;

**sendfile** on;

#tcp\_nopush on;

#keepalive\_timeout 0;

**keepalive\_timeout** 65;

#gzip on;

**server** {

**listen** 80;

**server\_name** localhost;

**location** /update\_content {

**content\_by\_lua\_file** /root/lua/update\_content.lua;

}

**location** /read\_content {

#使用限流配置

**limit\_req** zone=contentRateLimit;

**content\_by\_lua\_file** /root/lua/read\_content.lua;

}

}

}

配置说明：

* binary\_remote\_addr 是一种key，表示基于 remote\_addr(客户端IP) 来做限流，binary\_ 的目的是压缩内存占用量。
* zone：定义共享内存区来存储访问信息， contentRateLimit:10m 表示一个大小为10M，名字为contentRateLimit的内存区域。1M能存储16000 IP地址的访问信息，10M可以存储16W IP地址访问信息。
* rate 用于设置最大访问速率，rate=10r/s 表示每秒最多处理10个请求。Nginx 实际上以毫秒为粒度来跟踪请求信息，因此 10r/s 实际上是限制：每100毫秒处理一个请求。这意味着，自上一个请求处理完后，若后续100毫秒内又有请求到达，将拒绝处理该请求.我们这里设置成2 方便测试。

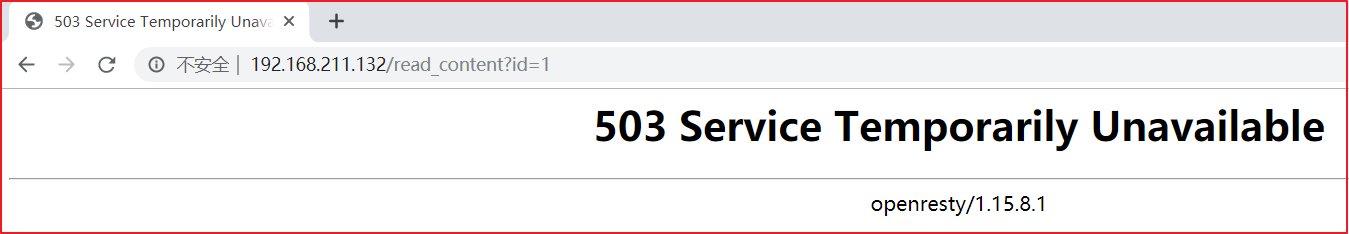
#### 测试

重新加载配置文件

cd /usr/local/openresty/nginx/sbin

./nginx -s reload

访问页面：<http://192.168.211.132/read_content?id=1>  ,连续刷新会直接报错。



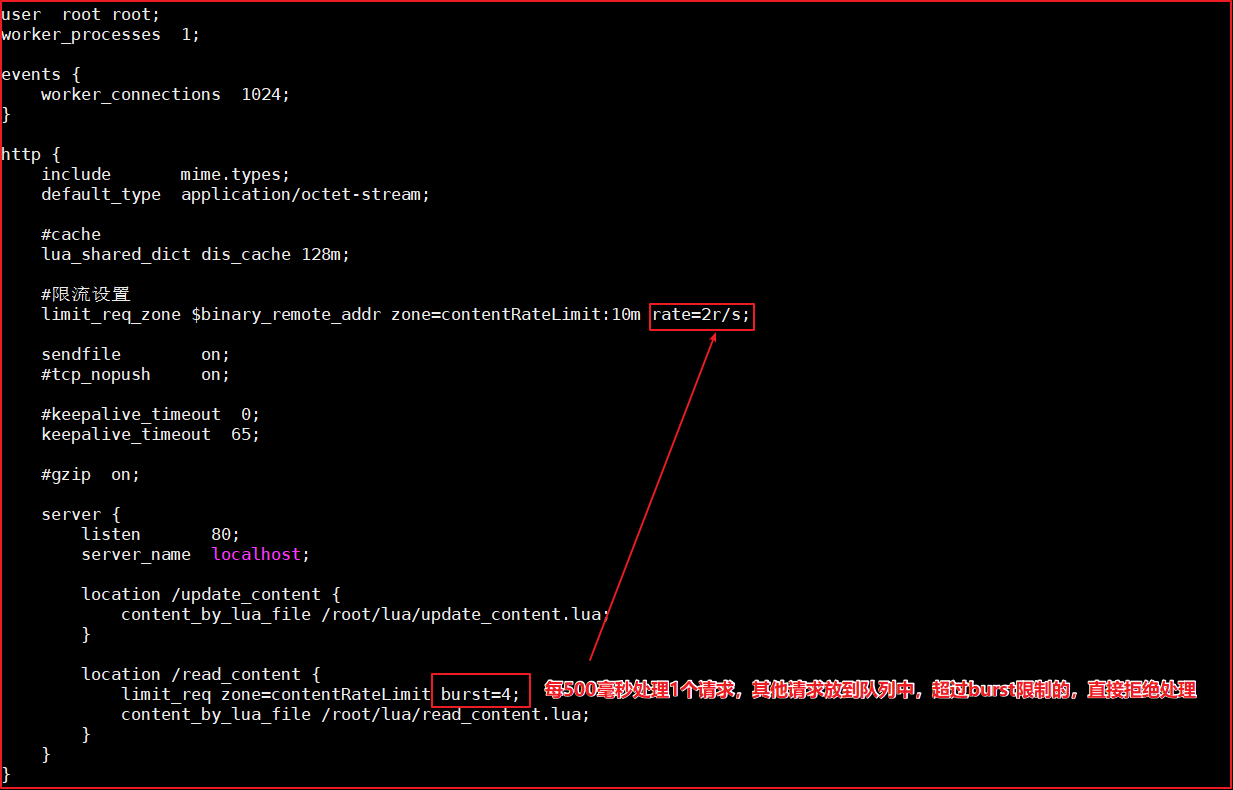
注意问题：

1. 漏桶漏出请求是匀速的。10r/s是怎样匀速的呢？每100ms漏出一个请求。在这样的配置下，桶是空的，所有不能实时漏出的请求，都会被拒绝掉。所以如果10次请求同时到达，那么只有一个请求能够得到执行，其它的，都会被拒绝。
2. Jmeter的Ramp-Up时间(秒)，表示多少秒完成线程数的所有请求，默认为0，表示同时并发！

#### 处理突发流量

上面例子限制 2r/s，如果有时正常流量突然增大，超出的请求将被拒绝，无法处理突发流量，可以结合 burst 参数使用来解决该问题。

例如，如下配置表示：



上图代码如下：

**server** {

**listen** 80;

**server\_name** localhost;

**location** /update\_content {

**content\_by\_lua\_file** /root/lua/update\_content.lua;

}

**location** /read\_content {

**limit\_req** zone=contentRateLimit burst=4;

**content\_by\_lua\_file** /root/lua/read\_content.lua;

}

}

burst 译为突发、爆发，表示在超过设定的处理速率后能额外处理的请求数,当 rate=10r/s 时，将1s拆成10份，即每100ms可处理1个请求。

此处，\*\*burst=4 \*\*，若同时有4个请求到达，Nginx 会处理第一个请求，剩余3个请求将放入队列，然后每隔500ms从队列中获取一个请求进行处理。若请求数大于4，将拒绝处理多余的请求，直接返回503.

不过，单独使用 burst 参数并不实用。假设 burst=50 ，rate依然为10r/s，排队中的50个请求虽然每100ms会处理一个，但第50个请求却需要等待 50 \* 100ms即 5s，这么长的处理时间自然难以接受。

因此，burst 往往结合 nodelay 一起使用。

例如：如下配置：

**server** {

**listen** 80;

**server\_name** localhost;

**location** /update\_content {

**content\_by\_lua\_file** /root/lua/update\_content.lua;

}

**location** /read\_content {

**limit\_req** zone=contentRateLimit burst=4 nodelay;

**content\_by\_lua\_file** /root/lua/read\_content.lua;

}

}

如上表示：

平均每秒允许不超过2个请求，突发不超过4个请求，并且处理突发4个请求的时候，没有延迟，等到完成之后，按照正常的速率处理。

如上两种配置结合就达到了速率稳定，但突然流量也能正常处理的效果。完整配置代码如下：

**user** root root;

**worker\_processes** 1;

**events** {

**worker\_connections** 1024;

}

**http** {

**include** mime.types;

**default\_type** application/octet-stream;

#cache

**lua\_shared\_dict** dis\_cache 128m;

#限流设置

**limit\_req\_zone** $binary\_remote\_addr zone=contentRateLimit:10m rate=2r/s;

**sendfile** on;

#tcp\_nopush on;

#keepalive\_timeout 0;

**keepalive\_timeout** 65;

#gzip on;

**server** {

**listen** 80;

**server\_name** localhost;

**location** /update\_content {

**content\_by\_lua\_file** /root/lua/update\_content.lua;

}

**location** /read\_content {

**limit\_req** zone=contentRateLimit burst=4 nodelay;

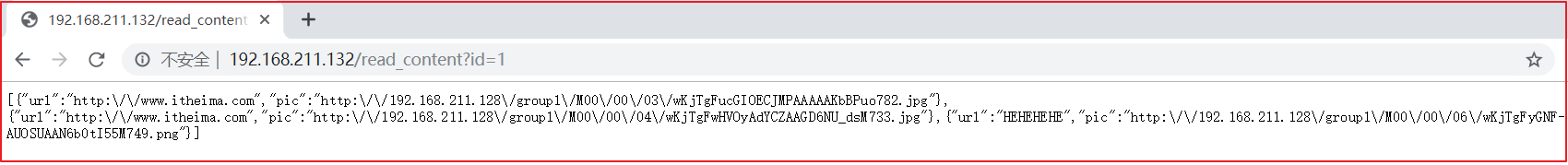
**content\_by\_lua\_file** /root/lua/read\_content.lua;

}

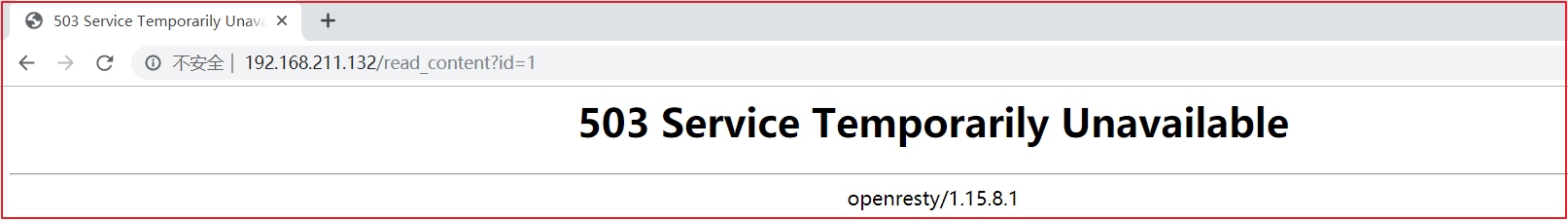
}

}

测试：如下图 在1秒钟之内可以刷新4次，正常处理。



但是超过之后，连续刷新5次，抛出异常。



### 控制并发量（连接数）

ngx\_http\_limit\_conn\_module 提供了限制连接数的能力。主要是利用limit\_conn\_zone和limit\_conn两个指令。

利用连接数限制 某一个用户的ip连接的数量来控制流量。

注意：并非所有连接都被计算在内 只有当服务器正在处理请求并且已经读取了整个请求头时，才会计算有效连接。此处忽略测试。

配置语法：

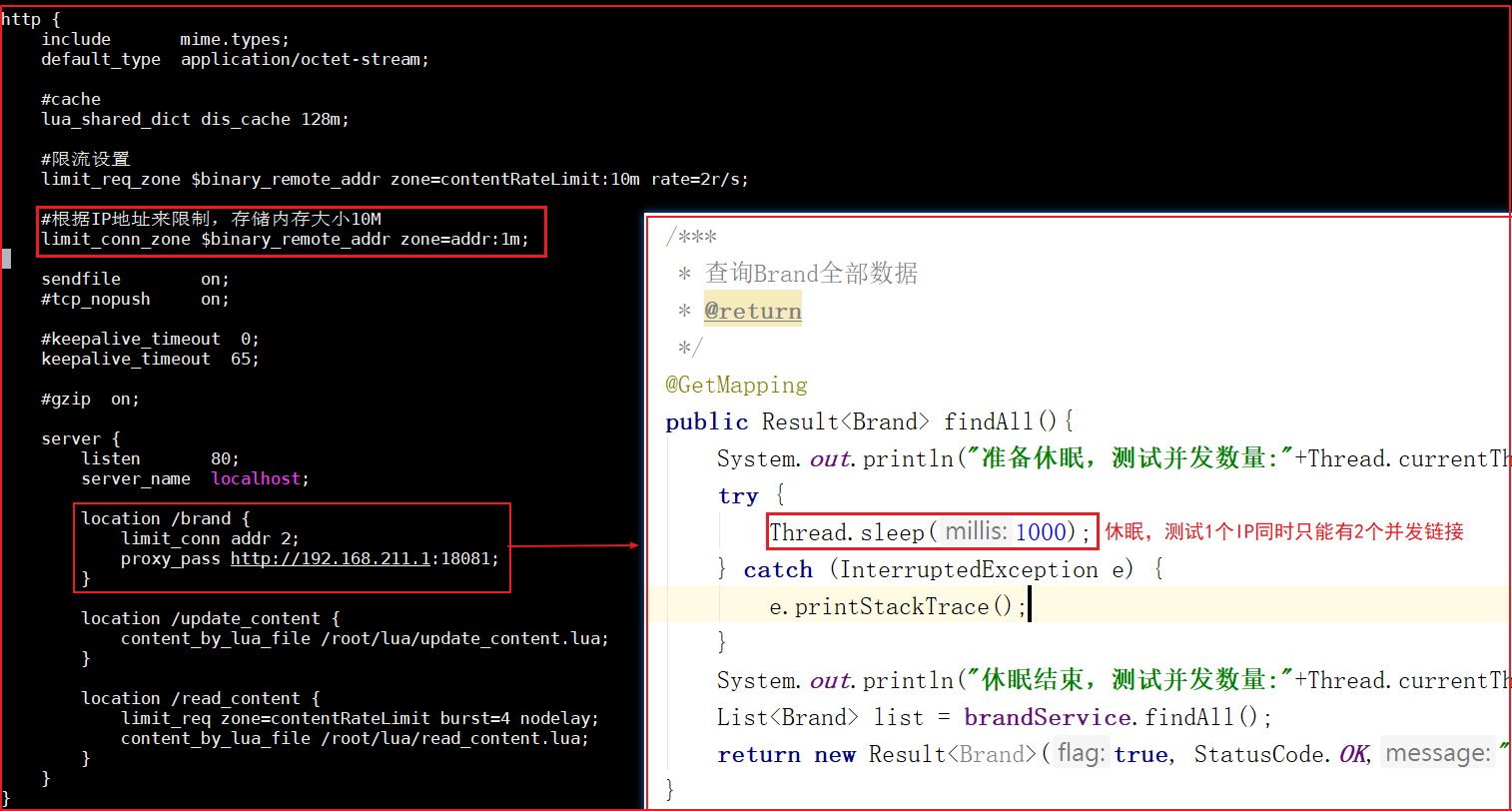
Syntax: limit\_conn zone number;

Default: —;

Context: http, server, location;

#### 配置限制ip固定连接数

如下，配置如下：



上图配置如下：

**http** {

**include** mime.types;

**default\_type** application/octet-stream;

#cache

**lua\_shared\_dict** dis\_cache 128m;

#限流设置

**limit\_req\_zone** $binary\_remote\_addr zone=contentRateLimit:10m rate=2r/s;

#根据IP地址来限制，存储内存大小10M

**limit\_conn\_zone** $binary\_remote\_addr zone=addr:1m;

**sendfile** on;

#tcp\_nopush on;

#keepalive\_timeout 0;

**keepalive\_timeout** 65;

#gzip on;

**server** {

**listen** 80;

**server\_name** localhost;

#所有以brand开始的请求，访问本地changgou-service-goods微服务

**location** /brand {

**limit\_conn** addr 2;

**proxy\_pass** http://192.168.211.1:18081;

}

**location** /update\_content {

**content\_by\_lua\_file** /root/lua/update\_content.lua;

}

**location** /read\_content {

**limit\_req** zone=contentRateLimit burst=4 nodelay;

**content\_by\_lua\_file** /root/lua/read\_content.lua;

}

}

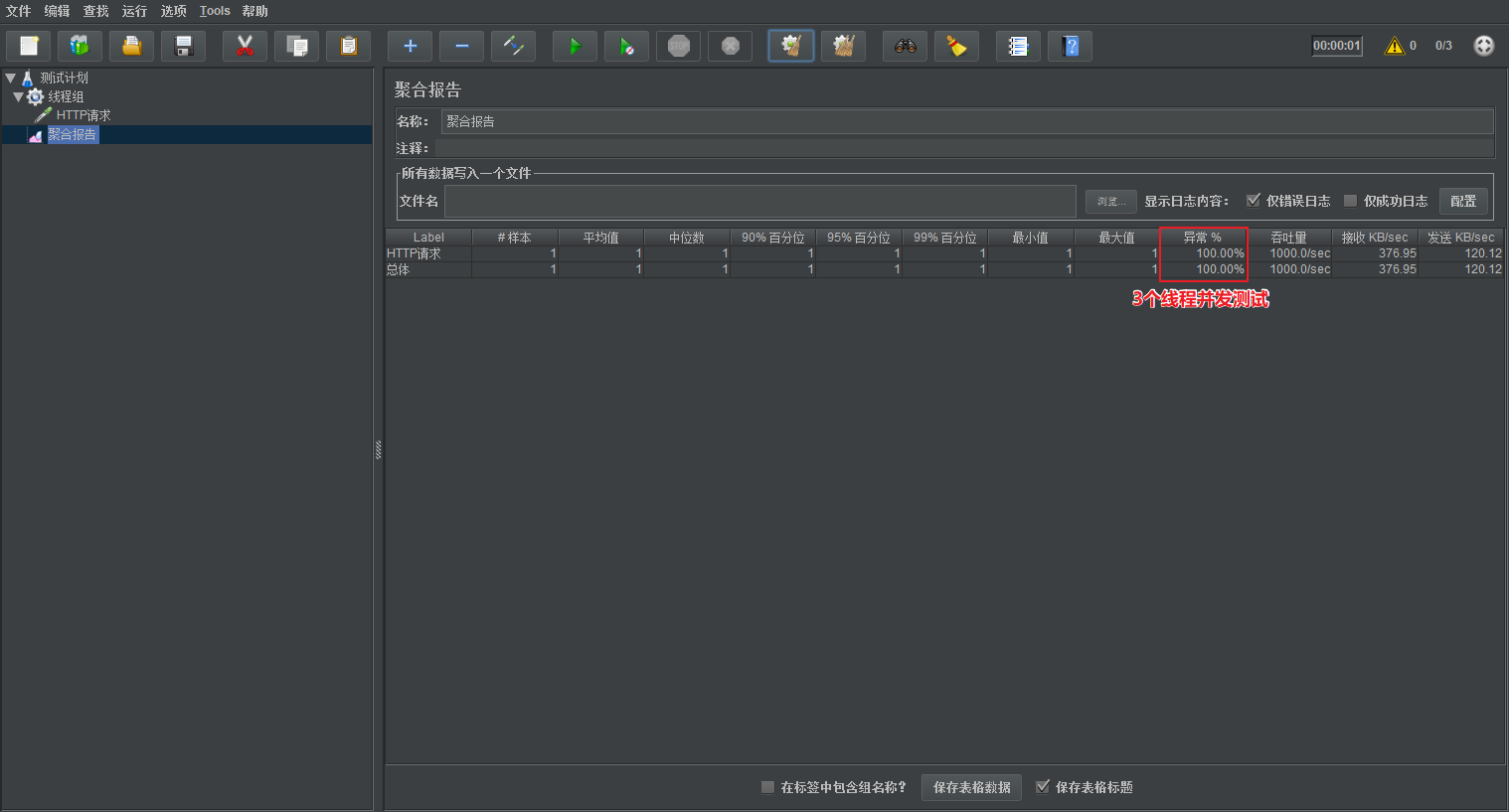
}

表示：

* limit\_conn\_zone $binary\_remote\_addr zone=**addr**:10m; 表示限制根据用户的IP地址来显示，设置存储地址为的内存大小10M
* limit\_conn **addr** 2; 表示 同一个地址只允许连接2次。

测试：

此时开3个线程，测试的时候会发生异常，开2个就不会有异常



测试工具可以使用Jmeter。

#### 配置限制单个ip连接数与服务总共连接数(了解)

限制每个客户端IP与服务器的连接数，同时限制与虚拟服务器的连接总数，如下配置：

**limit\_conn\_zone** $binary\_remote\_addr zone=perip:10m;

**limit\_conn\_zone** $server\_name zone=perserver:10m;

**server** {

**listen** 80;

**server\_name** localhost;

**charset** utf-8;

**location** / {

**limit\_conn** perip 10; #单个客户端ip与服务器的连接数．

**limit\_conn** perserver 100; #限制与服务器的总连接数

**root** html;

**index** index.html index.htm;

}

}

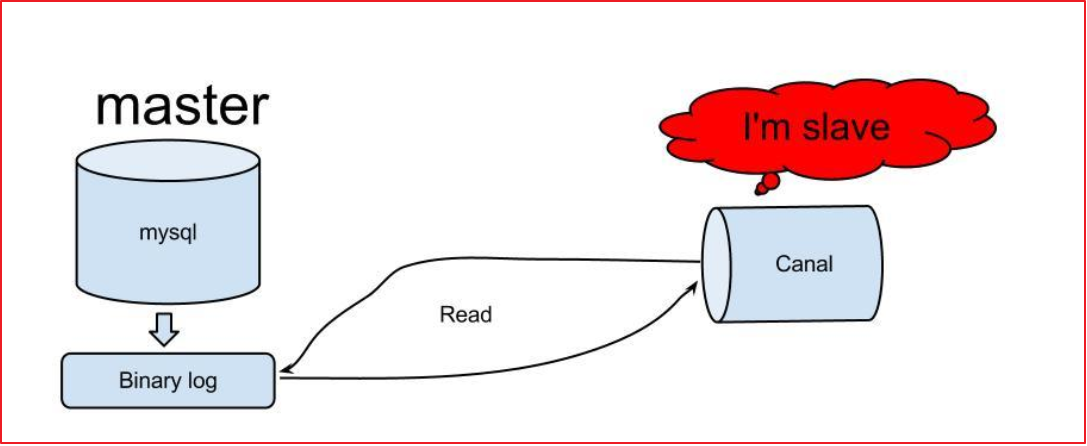
# canal同步广告

canal可以用来监控数据库数据的变化，从而获得新增数据，或者修改的数据。

canal是应阿里巴巴存在杭州和美国的双机房部署，存在跨机房同步的业务需求而提出的。

阿里系公司开始逐步的尝试基于数据库的日志解析，获取增量变更进行同步，由此衍生出了增量订阅&消费的业务。

## Canal工作原理



原理相对比较简单：

1. canal模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议
2. mysql master收到dump请求，开始推送binary log给slave(也就是canal)
3. canal解析binary log对象(原始为byte流)

canal需要使用到mysql，我们需要先安装mysql,给大家发的虚拟机中已经安装了mysql容器，但canal是基于mysql的主从模式实现的，所以必须先开启binlog.

## 开启binlog模式

linux上安装mysql容器，此处不在演示，提供的虚拟机已经配置好了。

**(1) 修改/mysqld.cnf，开启binlog模式。**

执行如下命令，编辑mysql配置文件



命令行如下：

docker exec -it mysql /bin/bash

**cd** /etc/mysql/mysql.**conf**.**d**

**vi** mysqld.**cnf**

修改mysqld.cnf配置文件，添加如下配置：



上图配置如下：

log-bin=/var/lib/mysql/mysql-bin

server-id=12345

**(2) 创建账号 用于测试使用,**

使用root账号创建用户并授予权限

**create** **user** canal@'%' **IDENTIFIED** **by** 'canal';

**GRANT** **SELECT**, REPLICATION **SLAVE**, REPLICATION CLIENT,SUPER **ON** \*.\* **TO** 'canal'@'%';

**FLUSH** **PRIVILEGES**;

**(3)重启mysql容器**

**docker** restart mysql

## canal容器安装

下载镜像：

docker pull docker.io/canal/canal-server

容器安装

docker run -p 11111:11111 --name canal -d docker.io/canal/canal-server

进入容器,修改核心配置canal.properties 和instance.properties，canal.properties 是canal自身的配置，instance.properties是需要同步数据的数据库连接配置。

执行代码如下:

docker exec -it canal /bin/bash

**cd** canal-server/**conf**/

**vi** canal.properties

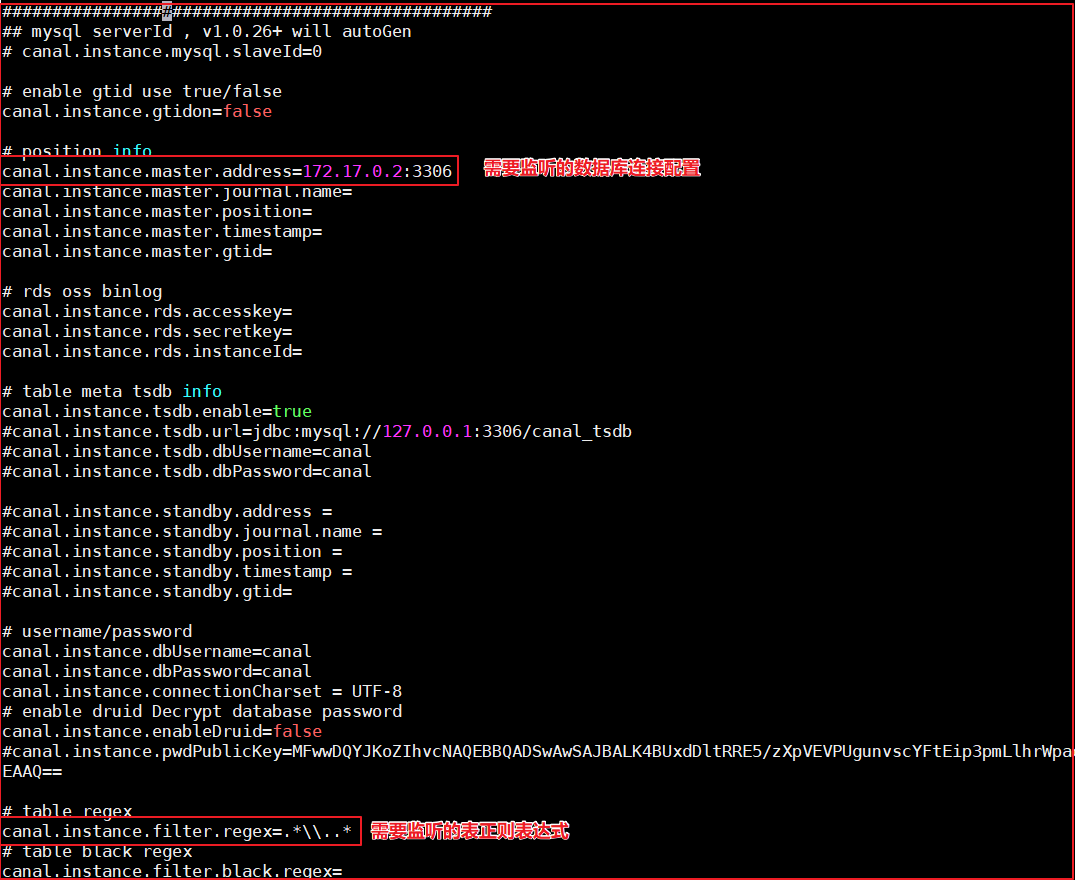
**cd** example/

**vi** instance.properties

修改canal.properties的id，不能和mysql的server-id重复，如下图：



修改instance.properties,配置数据库连接地址:



这里的canal.instance.filter.regex有多种配置，如下：

mysql 数据解析关注的表，Perl正则表达式.

多个正则之间以逗号(,)分隔，转义符需要双斜杠(\\)

常见例子：

1. 所有表：.\* or .\*\\..\*

2. canal schema下所有表： canal\\..\*

3. canal下的以canal打头的表：canal\\.canal.\*

4. canal schema下的一张表：canal.test1

5. 多个规则组合使用：canal\\..\*,mysql.test1,mysql.test2 (逗号分隔)

注意：此过滤条件只针对row模式的数据有效(ps. mixed/statement因为不解析sql，所以无法准确提取tableName进行过滤)

配置完成后，设置开机启动，并记得重启canal。

docker **update** --restart=always canal

docker restart canal

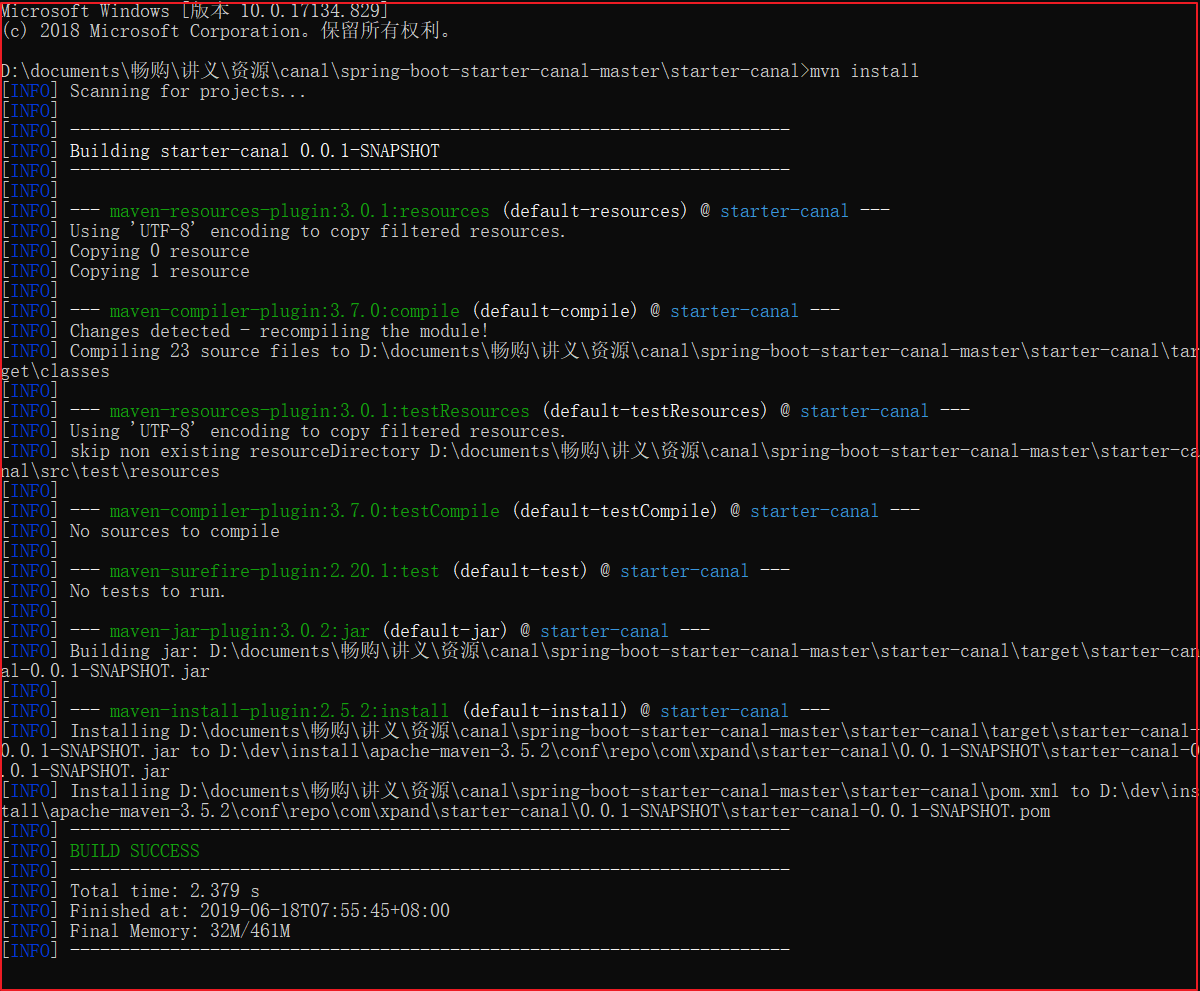
## canal微服务搭建

当用户执行 数据库的操作的时候，binlog 日志会被canal捕获到，并解析出数据。我们就可以将解析出来的数据进行同步到redis中即可。

思路：创建一个独立的程序，并监控canal服务器，获取binlog日志，解析数据，将数据更新到redis中。这样广告的数据就更新了。

### 安装辅助jar包

在canal\spring-boot-starter-canal-master中有一个工程starter-canal，它主要提供了SpringBoot环境下canal的支持，我们需要先安装该工程，在starter-canal目录下执行mvn install，如下图：



### canal微服务工程搭建

#### 加入依赖

在changgou\_service下创建changgou\_service\_canal工程，并引入相关配置。

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>changgou\_service</**artifactId**>  
 <**groupId**>com.changgou</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>changgou\_service\_canal</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter</**artifactId**>  
 </**dependency**>  
  
 *<!--canal依赖-->* <**dependency**>  
 <**groupId**>com.xpand</**groupId**>  
 <**artifactId**>starter-canal</**artifactId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### application.yml配置

**server**:  
 **port**: 18083  
**spring**:  
 **application**:  
 **name**: canal  
**eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka  
 **instance**:  
 **prefer-ip-address**: **true  
feign**:  
 **hystrix**:  
 **enabled**: **true***#hystrix 配置***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **timeout**:  
 *#如果enabled设置为false，则请求超时交给ribbon控制* **enabled**: true  
 **isolation**:  
 *# Hystrix隔离策略参数，2个模式：默认值 THREAD  
 # 1.THREAD 在固定大小线程池中，以单独线程执行，并发请求数受限于线程池大小。  
 # 2. SEMAPHORE 在调用线程中执行，通过信号量(限定值)来限制并发量* **strategy**: SEMAPHORE  
*#canal配置***canal**:  
 **client**:  
 **instances**:  
 **example**:  
 **host**: 192.168.211.132  
 **port**: 11111

#### 启动引导类创建

在com.changgou中创建启动类，代码如下

@SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**})  
@EnableEurekaClient  
@EnableCanalClient *//开启Canal客户端***public class** CanalApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(CanalApplication.**class**,args);  
 }  
}

#### 监听类创建

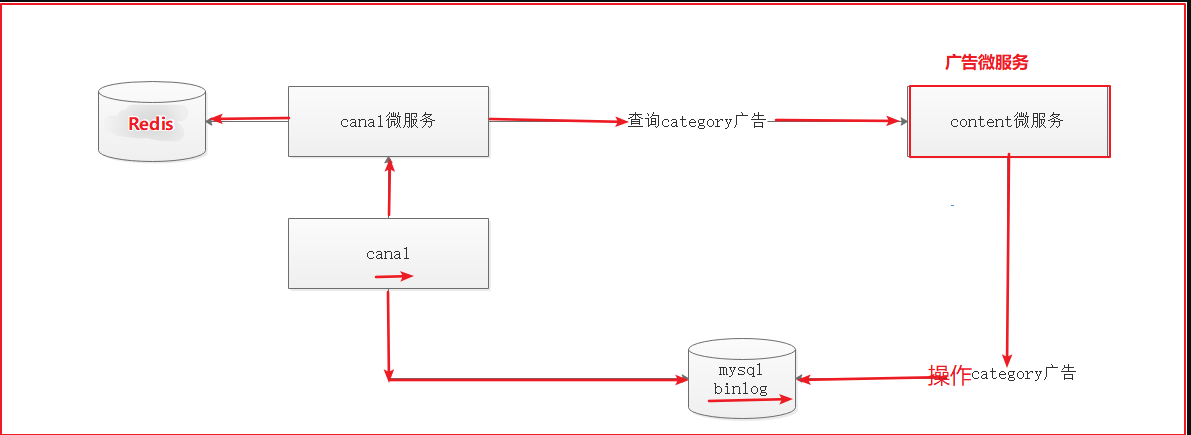
创建com.changgou.canal.listener.CanalDataEventListener

*/\*\*  
 \* 实现数据库变更监听  
 \** ***@description*** *com.changgou.canal.listener  
 \*/*@CanalEventListener  
**public class** CanalDataEventListener {  
 */\*\*  
 \* 新增监听  
 \** ***@InsertListenPoint：新增监听*** *\* CanalEntry.EventType:变更操作类型  
 \* CanalEntry.RowData：此次变更的数据  
 \*/* @InsertListenPoint  
 **public void** onEventInsert(CanalEntry.EventType eventType,CanalEntry.RowData rowData){  
 *//rowData.getBeforeColumnsList():数据变更前的内容  
 //rowData.getAfterColumnsList()：数据变更后的内容* System.***out***.println(**"---------新增监听--------"**);  
 **for** (CanalEntry.Column column : rowData.getAfterColumnsList()) {  
 System.***out***.println(column.getName() + **":"** + column.getValue());  
 }  
 }  
 */\*\*  
 \* 修改监听  
 \** ***@UpdateListenPoint：更新监听*** *\* CanalEntry.EventType:变更操作类型  
 \* CanalEntry.RowData：此次变更的数据  
 \*/* @UpdateListenPoint  
 **public void** onEventUpdate(CanalEntry.EventType eventType,CanalEntry.RowData rowData){  
 *//rowData.getBeforeColumnsList():数据变更前的内容  
 //rowData.getAfterColumnsList()：数据变更后的内容* System.***out***.println(**"---------更新监听--------"**);  
 **int** i = 0;  
 **for** (CanalEntry.Column column : rowData.getAfterColumnsList()) {  
 *//获取修改前数据* CanalEntry.Column beforeColumns = rowData.getBeforeColumns(i);  
 *//如果修改了字段* **if**(!beforeColumns.getValue().equals(column.getValue())) {  
 System.***out***.print(**"修改了字段:"** + column.getName() + **" "**);  
 System.***out***.println(beforeColumns.getValue() + **"-->"** + column.getValue());  
 }  
 i++;  
 }  
 }  
 */\*\*  
 \* 删除监听  
 \** ***@DeleteListenPoint：删除监听*** *\* CanalEntry.EventType:变更操作类型  
 \* CanalEntry.RowData：此次变更的数据  
 \*/* @DeleteListenPoint  
 **public void** onEventDelete(CanalEntry.EventType eventType,CanalEntry.RowData rowData){  
 *//rowData.getBeforeColumnsList():数据变更前的内容  
 //rowData.getAfterColumnsList()：数据变更后的内容* System.***out***.println(**"---------删除监听--------"**);  
 **for** (CanalEntry.Column column : rowData.getBeforeColumnsList()) {  
 System.***out***.println(column.getName() + **":"** + column.getValue());  
 }  
 }  
  
 */\*\*  
 \* 自定义监听  
 \** ***@ListenPoint：自定义监听*** *\* destination：必须使用canal.properties配置文件中canal.destinations属性的名字  
 \* schema：监听的数据库  
 \* table：监听的表  
 \* eventType：监听的操作类型  
 \*/* @ListenPoint(destination = **"example"**,schema = **"changgou\_content"**,table = **"tb\_content"**,eventType = CanalEntry.EventType.***DELETE***)  
 **public void** onEventCustomUpdate(CanalEntry.EventType eventType,CanalEntry.RowData rowData){  
 System.***out***.println(**"---------自定义监听--------"**);  
 **for** (CanalEntry.Column column : rowData.getBeforeColumnsList()) {  
 System.***out***.println(column.getName() + **":"** + column.getValue());  
 }  
 }  
}

#### 测试

启动canal微服务，然后修改任意数据库的表数据，查看canal微服务后台输出。

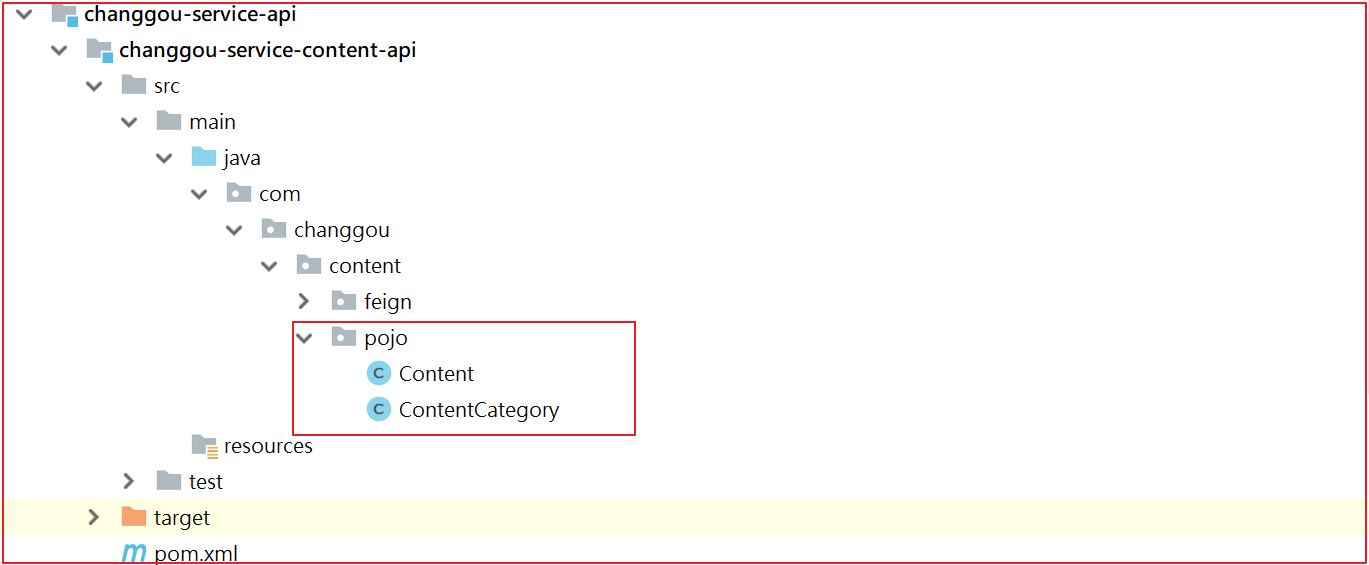
## 广告同步(作业)



如上图，每次执行广告操作的时候，会记录操作日志到，然后将操作日志发送给canal，canal将操作记录发送给canal微服务，canal微服务根据修改的分类ID调用content微服务查询分类对应的所有广告，canal微服务再将所有广告存入到Redis缓存。

### content微服务搭建

首先在changgou\_service\_api中创建**changgou\_service\_content\_api** ,将pojo拷贝到API工程中，如下图：



在changgou\_service中搭建**changgou\_service\_content**微服务，对应的dao、service、controller、pojo由代码生成器生成。

#### pom.xml配置

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**parent**>  
 <**artifactId**>changgou\_service</**artifactId**>  
 <**groupId**>com.changgou</**groupId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>changgou\_service\_content</**artifactId**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>com.changgou</**groupId**>  
 <**artifactId**>changgou\_service\_content\_api</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### application.yml配置

**server**:  
 **port**: 18084  
**spring**:  
 **application**:  
 **name**: content  
 **datasource**:  
 **driver-class-name**: com.mysql.jdbc.Driver  
 **url**: jdbc:mysql://192.168.211.132:3306/changgou\_content?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **username**: root  
 **password**: 123456  
**eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka  
 **instance**:  
 **prefer-ip-address**: **true  
feign**:  
 **hystrix**:  
 **enabled**: **true  
mybatis**:  
 **configuration**:  
 **map-underscore-to-camel-case**: **true** *#开启驼峰功能  
  
#hystrix 配置***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **timeout**:  
 *#如果enabled设置为false，则请求超时交给ribbon控制* **enabled**: true  
 **isolation**:  
 **strategy**: SEMAPHORE

#### 启动引导类创建

@SpringBootApplication  
@EnableEurekaClient  
@MapperScan(basePackages = {**"com.changgou.content.dao"**})  
**public class** ContentApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ContentApplication.**class**);  
 }  
}

### 广告查询

在content微服务中，添加根据分类查询广告，提供feign请求包装，供后续的canal微服务调用。

#### 业务层

业务层相关的代码可以不写，我们这次使用代码生成器的findList(Content content)方法。

#### 控制层

修改changgou-service-content的com.changgou.content.controller.ContentController,添加根据分类ID查询广告数据，代码如下：

*/\*\*\*  
 \* 根据categoryId查询广告集合  
 \*/*@GetMapping(value = **"/list/category/{id}"**)  
**public** Result<List<Content>> findByCategory(@PathVariable Long id){  
 *//根据分类ID查询广告集合* Content where = **new** Content();  
 where.setCategoryId(id);  
 List<Content> contents = **contentService**.findList(where);  
 **return new** Result<List<Content>>(**true**,StatusCode.***OK***,**"查询成功！"**,contents);  
}

#### feign配置

在changgou-service-content-api工程中添加feign，代码如下：

*/\*\*  
 \* 根据分类id查询所有广告列表  
 \** ***@param id*** *\** ***@return*** *\*/*@GetMapping(value = **"/list/category/{id}"**)  
**public** Result<List<Content>> findByCategory(@PathVariable Long id);

### 同步实现

#### 配置redis

changgou-cancal-service工程中修改application.yml配置文件，添加redis配置，如下代码：

**server**:  
 **port**: 18083  
**spring**:  
 **application**:  
 **name**: canal  
 **redis**:  
 *# 指向docker容器地址* **host**: 192.168.211.132  
 **port**: 6379

#省略其它配置….

#### 启动类中开启feign

修改CanalApplication，添加@EnableFeignClients注解，代码如下：

@SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**})  
@EnableEurekaClient  
@EnableCanalClient *//开启Canal客户端*@EnableFeignClients *//开启Feign客户端***public class** CanalApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(CanalApplication.**class**,args);  
 }  
}

#### 同步实现

修改监听类CanalDataEventListener，实现监听广告的增删改，并根据增删改的数据使用feign查询对应分类的所有广告，将广告存入到Redis中，代码如下：

*/\*\*  
 \* 实现数据库变更监听  
 \** ***@author*** *Steven  
 \** ***@description*** *com.changgou.canal.listener  
 \*/*@CanalEventListener  
**public class** CanalDataEventListener {  
  
 @Autowired  
 **private** ContentFeign **contentFeign**;  
 @Autowired  
 **private** StringRedisTemplate **stringRedisTemplate**;  
  
 */\*\*  
 \* 广告内容监听  
 \** ***@ListenPoint：自定义监听*** *\* destination：必须使用canal.properties配置文件中canal.destinations属性的名字  
 \* schema：监听的数据库  
 \* table：监听的表  
 \* eventType：监听的操作类型  
 \*/* @ListenPoint(destination = **"example"**,schema = **"changgou\_content"**,table = **"tb\_content"**,eventType =  
 {CanalEntry.EventType.***INSERT***,CanalEntry.EventType.***UPDATE***,CanalEntry.EventType.***DELETE***})  
 **public void** onEventContent(CanalEntry.EventType eventType,CanalEntry.RowData rowData){  
 System.***out***.println(**"---------广告监听--------"**);  
 *//先获取categoryId* String categoryId = **""**;  
 *//新增操作，取修改后的值* **if**(eventType == CanalEntry.EventType.***INSERT***){  
 *//tb\_content表中categoryId在第二列，所以下标为1* categoryId = rowData.getAfterColumns(1).getValue();  
 *//修改操作，取修改后的值* }**else if**(eventType == CanalEntry.EventType.***UPDATE***){  
 *//tb\_content表中categoryId在第二列，所以下标为1* categoryId = rowData.getAfterColumns(1).getValue();  
 *//如果修改过程中，把categoryId修改了，要把修改前的广告分类缓存也更新了* String categoryIdBefore = rowData.getBeforeColumns(1).getValue();  
 **if**(!categoryIdBefore.equals(categoryId)){  
 Result<List<Content>> result = **contentFeign**.findByCategory(**new** Long(categoryIdBefore));  
 **if**(result.getData() != **null**){  
 *//更新缓存* **stringRedisTemplate**.boundValueOps(**"content\_"** + categoryIdBefore).set(JSON.*toJSONString*(result.getData()));  
 }  
 }  
 }**else**{ *//删除操作，取修改前的值  
 //tb\_content表中categoryId在第二列，所以下标为1* categoryId = rowData.getBeforeColumns(1).getValue();  
 }  
 *//根据categoryId查询所有广告列表* Result<List<Content>> result = **contentFeign**.findByCategory(**new** Long(categoryId));  
 **if**(result.getData() != **null**){  
 *//更新缓存* **stringRedisTemplate**.boundValueOps(**"content\_"** + categoryId).set(JSON.*toJSONString*(result.getData()));  
 }  
 }  
 //省略其它代码……….  
 }

#### 测试

修改数据库数据，可以看到Redis中的缓存跟着一起变化。