# 《亿级流量系统多级缓存架构10》

40 59 60 61 65 68 70 73 75 78\_20210404\_031819

79 82 83 87 88 89 90 94 95 98

99 100 103 104 105 106【共26节】

# Lua +

## **课程主要内容**

l 多级缓存架构模型

l Redis整合Lua

l 利用Redis+Lua开发分布式锁

l Openresty 安装部署

l Nginx下lua脚本访问Nginx内核变量

l Nginx下利用Lua脚本访问Redis

Lua 是由巴西里约热内卢天主教大学（Pontifical Catholic University of Rio de Janeiro）里的一个研究小组于1993年开发的一种轻量、小巧的脚本语言，用标准 C 语言编写，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。

官网：<http://www.lua.org>/

Redis 在 2.6 版本中推出了脚本功能，允许开发者将 Lua 语言编写的脚本传到 Redis 中执行。使用 Lua 脚本的优点有如下几点:

l 减少网络开销：本来需要多次请求的操作，可以一次请求完成，从而节约网络开销；

l 原子操作：Redis 会将整个脚本作为一个整体执行，中间不会执行其它命令；

l 复用：客户端发送的脚本会存储在 Redis 中，从而实现脚本的复用。

## **Redis 与 Lua 整合**

### 测试lua执行

#### **在redis中执行简单脚本**

登录到客户端后执行

#### **hello world**

eval   "return 1+1"   0

#命令   脚本       参数个数

#### **参数**

EVAL "local msg='hello world' return msg..KEYS[1]" 1 AAA BBB

表是基于1的，也就是说索引以数值1开始。所以在表中的第一个元素就是mytable[1]，第二个就是mytable[2]等等。 表中不能有nil值。如果一个操作表中有[1, nil, 3, 4]，那么结果将会是[1]——表将会在第一个nil截断。

### 独立脚本

#### **获取key的value**

local key=KEYS[1]

​

local list=redis.call("get",key);

​

return list;

#### **读取redis集合中的数据**

local key=KEYS[1]

​

local list=redis.call("lrange",key,0,-1);

​

return list;

#### **统计点击次数**

local msg='count:'

local count = redis.call("get","count")

if not count then

       redis.call("set","count",1)

end

​

redis.call("incr","count")

​

return msg..count+1

​

#### **执行lua脚本**

##### **本地执行**

redis-cli --eval test.lua aaa,bbb

​

##### **远程执行**

redis-cli -h 192.168.2.161 -a密码 --eval /usr/local/luascript/test.lua name age , xiao6

### Lua 与 Redis 交互

#### **Lua 脚本获取 EVAL & EVALSHA 命令的参数**

通过 Lua 脚本的全局变量 KEYS 和 ARGV，能够访问 EVAL 和 EVALSHA 命令的 key [key ...] 参数和 arg [arg ...] 参数。

作为 Lua Table，能够将 KEYS 和 ARGV 作为一维数组使用，其下标从 1 开始。

#### **Lua 脚本内部执行 Redis 命令**

Lua 脚本内部允许通过内置函数执行 Redis 命令：

redis.call()

redis.pcall()

两者非常相似，区别在于：

若 Redis 命令执行错误，redis.call() 将错误抛出（即 EVAL & EVALSHA 执行出错）；

redis.pcall() 将错误内容返回。

local msg='count:' local count = redis.call("get","count") if not count then redis.call("set","count",1) end redis.call("incr","count") return msg..count+1

### redis WATCH/MULTI/EXEC 与Lua

redis 原生支持 监听、事务、批处理，那么还需要lua吗？

* 两者不存在竞争关系，而是增强关系，lua可以完成redis自身没有的功能
* 在lua中可以使用上一步的结果，也就是可以开发**后面操作依赖前面操作的执行结果的应用**，MULT中的命令都是独立操作
* redis可以编写模块增强功能，但是c语言写模块，太难了，lua简单的多
* 计算向移动数据
* 原子操作

lua脚本尽量短小并且尽量保证同一事物写在一段脚本内，因为redis是单线程的，过长的执行会造成阻塞，影响服务器性能。

### Redis Lua 脚本管理

1.script load 此命令用于将Lua脚本加载到Redis内存中

2.script exists scripts exists sha1 [sha1 …] 此命令用于判断sha1是否已经加载到Redis内存中

3.script flush 此命令用于清除Redis内存已经加载的所有Lua脚本,在执行script flush后,sha1不复存在

4.script kill 此命令用于杀掉正在执行的Lua脚本

### 死锁

下面代码会进入死循环，导致redis无法接受其他命令。

eval "while true do end" 0

127.0.0.1:6379> keys \*

(error) BUSY Redis is busy running a script. You can only call SCRIPT KILL or SHUTDOWN NOSAVE.

但是可以接受 SCRIPT KILL or SHUTDOWN NOSAVE. 两个命令

SHUTDOWN NOSAVE 不会进行持久化的操作

SCRIPT KILL 可以杀死正在执行的进程

### 生产环境下部署

#### **加载到redis**

redis-cli script load "$(cat test.lua)"

得到sha1值

执行

redis-cli evalsha "7a2054836e94e19da22c13f160bd987fbc9ef146" 0

## **Openresty Nginx + Lua**

Nginx是一个主进程配合多个工作进程的工作模式，每个进程由单个线程来处理多个连接。

在生产环境中，我们往往会把cpu内核直接绑定到工作进程上，从而提升性能。

### 安装

#### **预编译安装**

以CentOS举例 其他系统参照：<http://openresty.org/cn/linux-packages.html>

你可以在你的 CentOS 系统中添加 openresty 仓库，这样就可以便于未来安装或更新我们的软件包（通过 yum update 命令）。运行下面的命令就可以添加我们的仓库：

 yum install yum-utils

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

然后就可以像下面这样安装软件包，比如 openresty：

 yum install openresty

如果你想安装命令行工具 resty，那么可以像下面这样安装 openresty-resty 包：

 sudo yum install openresty-resty

#### **源码编译安装**

#### **下载**

<http://openresty.org/cn/download.html>

./configure

然后在进入 openresty-VERSION/目录, 然后输入以下命令配置:

./configure

默认, --prefix=/usr/local/openresty 程序会被安装到/usr/local/openresty目录。

依赖 gcc openssl-devel pcre-devel zlib-devel

安装：yum install gcc openssl-devel pcre-devel zlib-devel postgresql-devel

您可以指定各种选项，比如

./configure --prefix=/opt/openresty \

​

          --with-luajit \

​

          --without-http\_redis2\_module \

​

          --with-http\_iconv\_module \

​

          --with-http\_postgres\_module

试着使用 ./configure --help 查看更多的选项。

make && make install

#### **服务命令**

##### **启动**

Service openresty start

##### **停止**

Service openresty stop

##### **检查配置文件是否正确**

Nginx -t

重新加载配置文件

Service openresty reload

##### **查看已安装模块和版本号**

Nginx -V

### 测试lua脚本

在Nginx.conf 中写入

  location /lua {

​

       default\_type text/html;

       content\_by\_lua '

          ngx.say("<p>Hello, World!</p>")

        ';

    }

### lua-nginx-module

#### **创建配置文件lua.conf**

  server {

       listen       80;

       server\_name localhost;

​

  location /lua {

​

       default\_type text/html;

​

       content\_by\_lua\_file conf/lua/hello.lua;

​

        }

}

#### **在Nginx.conf下引入lua配置**

include lua.conf;

#### **创建外部lua脚本**

conf/lua/hello.lua

内容：

ngx.say("<p>Hello, World!</p>")

#### **获取Nginx uri中的单一变量**

    location /nginx\_var {

​

         default\_type text/html;

​

        content\_by\_lua\_block {

​

            ngx.say(ngx.var.arg\_a)

​

        }

    }

#### **获取Nginx uri中的所有变量**

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

#### **获取Nginx请求头信息**

local headers = ngx.req.get\_headers()

​

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

#### **获取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

#### **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/>")

#### **body内容体**

# 《亿级流量系统多级缓存架构2》

# Lua +

## **课程主要内容**

### Nginx缓存

#### **Nginx全局共享内存缓存**

lua\_shared\_dict shared\_data 1m;

​------

local shared\_data = ngx.shared.shared\_data

local i = shared\_data:get("i")

if not i then

​

   i = 1

​

   shared\_data:set("i", i)

​

   ngx.say("lazy set i ", i, "<br/>")

end

i = shared\_data:incr("i", 1)

ngx.say("i=", i, "<br/>")

#### **lua-resty-lrucache**

Lua 实现的一个简单的 LRU 缓存，适合在 Lua 空间里直接缓存较为复杂的 Lua 数据结构：

它相比 ngx\_lua 共享内存字典可以省去较昂贵的序列化操作，相比 memcached 这样的外部服务又能省去较昂贵的 socket 操作

lrucache 有两种实现

resty.lrucache

适合用来缓存命中率高或读操作远远大于写操作的缓存业务

resty.lrucache.pureffi

适合用来缓存命中率低或需要对key进行频繁增、删操作的缓存业务

local lrucache = require "resty.lrucache"

​

local c, err = lrucache.new(200)

c:set("dog", 32)

   c:set("cat", 56)

   ngx.say("dog: ", c:get("dog"))

   ngx.say("cat: ", c:get("cat"))

​

   c:set("dog", { age = 10 }, 0.1)  -- expire in 0.1 sec

   c:delete("dog")

[https://github.com/openresty/lua-resty-lrucache#name](https://github.com/openresty/lua-resty-lrucache" \l "name)

##### **实例**

local mycache = require("mycache")

local count = mycache.get("count") or 0

count = count + 1

mycache.set("count", count, 10 \* 60 \* 60) --10分钟

ngx.say(mycache.get("count"))

​

**mycache.lua**

local lrucache = require("resty.lrucache")

--创建缓存实例，并指定最多缓存多少条目

local cache, err = lrucache.new(200)

if not cache then

  ngx.log(ngx.ERR, "create cache error : ", err)

end

local function set(key, value, ttlInSeconds)

   cache:set(key, value, ttlInSeconds)

end

local function get(key)

   return cache:get(key)

end

local \_M = {

 set = set,

 get = get

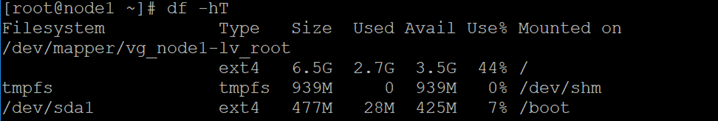
}

return \_M

##### **tmpfs**

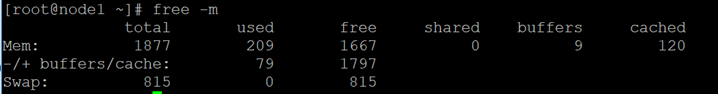
在Linux系统内存中的虚拟磁盘映射，可以理解为使用物理内存当做磁盘，利用这种文件系统，可以有效提高在高并发场景下的磁盘读写，但是重启后数据会丢失。

###### **查看tmpfs命令：**



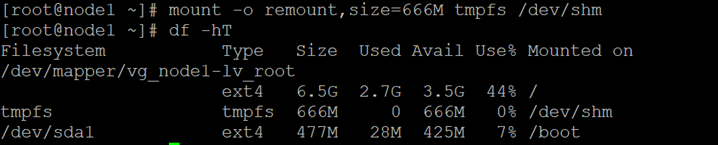
###### **系统默认开启，大小约为物理内存一半**

###### **查看物理内存利用情况**



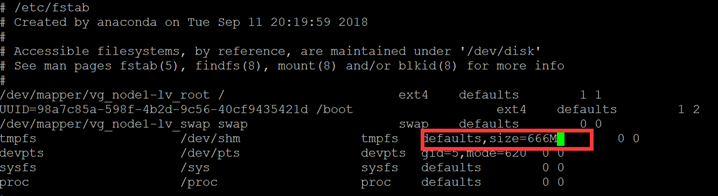
tmpfs没有占用内存空间，只有在写入数据的时候才会占用实际的物理内存

###### **调整大小**



临时修改方式如下，立即生效但重启后会恢复

###### **永久修改 /etc/fstab文件**



#### **http\_proxy 本地磁盘缓存**

proxy\_cache\_path /path/to/cache levels=1:2 keys\_zone=my\_cache:10m max\_size=10g inactive=60m use\_temp\_path=off;

​

server {

​

     set $upstream http://ip:port

​

          location / {

​

                   proxy\_cache my\_cache;

​

                   proxy\_pass $upstream;

            }

​

}

/path/to/cache #本地路径，用来设置Nginx缓存资源的存放地址

levels #默认所有缓存文件都放在同一个/path/to/cache下，但是会影响缓存的性能，因此通常会在/path/to/cache下面建立子目录用来分别存放不同的文件。假设levels=1:2，Nginx为将要缓存的资源生成的key为f4cd0fbc769e94925ec5540b6a4136d0，那么key的最后一位0，以及倒数第2-3位6d作为两级的子目录，也就是该资源最终会被缓存到/path/to/cache/0/6d目录中

key\_zone #在共享内存中设置一块存储区域来存放缓存的key和metadata（类似使用次数），这样nginx可以快速判断一个request是否命中或者未命中缓存，1m可以存储8000个key，10m可以存储80000个key

max\_size #最大cache空间，如果不指定，会使用掉所有disk space，当达到配额后，会删除最少使用的cache文件

inactive #未被访问文件在缓存中保留时间，本配置中如果60分钟未被访问则不论状态是否为expired，缓存控制程序会删掉文件。inactive默认是10分钟。需要注意的是，inactive和expired配置项的含义是不同的，expired只是缓存过期，但不会被删除，inactive是删除指定时间内未被访问的缓存文件

use\_temp\_path #如果为off，则nginx会将缓存文件直接写入指定的cache文件中，而不是使用temp\_path存储，official建议为off，避免文件在不同文件系统中不必要的拷贝

proxy\_cache #启用proxy cache，并指定key\_zone。另外，如果proxy\_cache off表示关闭掉缓存。

### lua-resty-redis访问redis

#### **常用方法**

local res, err = red:get("key")

​

local res, err = red:lrange("nokey", 0, 1)

​

ngx.say("res:",cjson.encode(res))

#### **创建连接**

red, err = redis:new()

​

ok, err = red:connect(host, port, options\_table?)

#### **timeout**

red:set\_timeout(time)

#### **keepalive**

red:set\_keepalive(max\_idle\_timeout, pool\_size)

#### **close**

ok, err = red:close()

null

nil

error

exception

crash

#### **pipeline**

red:init\_pipeline()

​

results, err = red:commit\_pipeline()

#### **认证**

   local res, err = red:auth("foobared")

​

   if not res then

​

      ngx.say("failed to authenticate: ", err)

​

       return

end

#### **redis-cluster支持**

<https://github.com/steve0511/resty-redis-cluster>

### redis2-nginx-module

redis2-nginx-module是一个支持 Redis 2.0 协议的 Nginx upstream 模块，它可以让 Nginx 以非阻塞方式直接防问远方的 Redis 服务，同时支持 TCP 协议和 Unix Domain Socket 模式，并且可以启用强大的 Redis 连接池功能。

<https://github.com/openresty/redis2-nginx-module>

#### **test**

location = /foo {

​

default\_type text/html;

​

    redis2\_query auth 123123;

​

    set $value 'first';

​

    redis2\_query set one $value;

​

    redis2\_pass 192.168.199.161:6379;

​

}

#### **get**

location = /get {

​

default\_type text/html;

​

    redis2\_pass 192.168.199.161:6379;

​

    redis2\_query auth 123123;

​

    set\_unescape\_uri $key $arg\_key;  # this requires ngx\_set\_misc

​

    redis2\_query get $key;

​

}

#### **set**

# GET /set?key=one&val=first%20value

​

location = /set {

​

default\_type text/html;

​

redis2\_pass 192.168.199.161:6379;

​

redis2\_query auth 123123;

    set\_unescape\_uri $key $arg\_key;  # this requires ngx\_set\_misc

​

    set\_unescape\_uri $val $arg\_val;  # this requires ngx\_set\_misc

​

    redis2\_query set $key $val;

​

}

#### **pipeline**

    set $value 'first';

​

    redis2\_query set one $value;

​

    redis2\_query get one;

    redis2\_query set one two;

​

    redis2\_query get one;

​

redis2\_query del key1;

#### **list**

   redis2\_query lpush key1 C;

​

   redis2\_query lpush key1 B;

​

   redis2\_query lpush key1 A;

​

redis2\_query lrange key1 0 -1;

#### **集群**

upstream redis\_cluster {

​

    server 192.168.199.161:6379;

​

    server 192.168.199.161:6379;

​

}

​

location = /redis {

​

default\_type text/html;

​

        redis2\_next\_upstream error timeout invalid\_response;

​

        redis2\_query get foo;

​

        redis2\_pass redis\_cluster;

  }

## **URL一致性哈希负载均衡**

有针对性的对url进行一致性hash 定向负载到后端Nginx

提高Nginx缓存系统命中率

#### **nginx url\_hash**

Nginx第三方模块，在转发请求时如果后端服务器宕机，会导致503错误

#### **lua-resty-http**

### GitHub主页

<https://github.com/ledgetech/lua-resty-http>

### 安装组件

wget <https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http_headers.lua>

wget <https://raw.githubusercontent.com/pintsized/lua-resty-http/master/lib/resty/http.lua>

local http = require("resty.http")

​

local httpc = http.new()

local resp, err = httpc:request\_uri("http://www.sogou.com", {

   method = "GET",

   path = "/sogou?query=resty.http",

   headers = {

      ["User-Agent"] = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.111 Safari/537.36"

  }

})

if not resp then

   ngx.say("request error :", err)

   return

end

ngx.status = resp.status

for k, v in pairs(resp.headers) do

   if k ~= "Transfer-Encoding" and k ~= "Connection" then

       ngx.header[k] = v

   end

end

ngx.say(resp.body)

httpc:close()

​

http模块加入

resolver 8.8.8.8;

### lua-resty-http实现一致性hash负载均衡

local http = require("resty.http")

local httpc = http.new()

​

local hosts = {"192.168.150.111","192.168.150.112"}

​

local item\_id= ngx.var.arg\_id

​

local id\_hash = ngx.crc32\_long(item\_id)

local index = (id\_hash % 2) +1

local resp, err = httpc:request\_uri("http://"..hosts[index], {

   method = "GET",

   path = "/sogou?query=resty.http",

   headers = {

      ["User-Agent"] = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.111 Safari/537.36"

  }

})

if not resp then

   ngx.say("request error :", err)

   return

end

ngx.say(resp.body)

httpc:close()

## **模板实时渲染 lua-resty-template**

<https://github.com/bungle/lua-resty-template>

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。

lua-resty-template大体内容有：

l 模板位置：从哪里查找模板；

l 变量输出/转义：变量值输出；

l 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；

l 注释：解释代码片段含义；

l include：包含另一个模板片段；

l 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。

基础语法

l {(include\_file)}：包含另一个模板文件；

l {\* var \*}：变量输出；

l {{ var }}：变量转义输出；

l {% code %}：代码片段；

l {# comment #}：注释；

l {-raw-}：中间的内容不会解析，作为纯文本输出；

### lua代码热加载

在http模块中加入

lua\_code\_cache off;

reload后Nginx会提示影响性能，记得在生产环境中关掉。

### 1569585068623

### 测试

### 一、初始化

-- Using template.new

local template = require "resty.template"

local view = template.new "view.html"

view.message = "Hello, World!"

view:render()

​

-- Using template.render

-- template.render("view.html", { message = "Hel11lo, Worl1d!" })

### 二、执行函数，得到渲染之后的内容

local func = template.compile("view.html")

​

local content = func(context)

​

ngx.say("xx:",content)

### resty.template.html

local template = require("resty.template")

local html = require "resty.template.html"

​

template.render([[

<ul>

{% for \_, person in ipairs(context) do %}

   {\*html.li(person.name)\*} --

{% end %}

</ul>

<table>

{% for \_, person in ipairs(context) do %}

   <tr data-sort="{{(person.name or ""):lower()}}">

       {\*html.td{ id = person.id }(person.name)\*}

   </tr>

{% end %}

</table>]], {

  { id = 1, name = "Emma"},

  { id = 2, name = "James" },

  { id = 3, name = "Nicholas" },

  { id = 4 }

})

### 模板内容

<!DOCTYPE html>

<html>

<body>

 <h1>{{message}}</h1>

</body>

</html>

### 多值传入

template.caching(false)

local template = require("resty.template")

local context = {

   name = "lucy",

   age = 50,

}

template.render("view.html", context)

### 模板内容

<!DOCTYPE html>

<html>

<body>

<h1>name:{{name}}</h1>

<h1>age:{{age}}</h1>

</body>

</html>

### 模板管理与缓存

模板缓存：默认开启，开发环境可以手动关闭

template.caching(true)

模板文件需要业务系统更新与维护，当模板文件更新后，可以通过模板版本号或消息通知Openresty清空缓存重载模板到内存中

template.cache = {}

### 完整页面

local template = require("resty.template")

template.caching(false)

local context = {

   title = "测试",

   name = "lucy",

   description = "<script>alert(1);</script>",

   age = 40,

   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("view.html", context)

### 模板

{(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}}

          简介：{\* description \*}<br/>

    {# 可以做一些运算 #}

    年龄: {\* age + 10 \*}<br/>

    {# 循环输出 #}

    爱好：

    {% for i, v in ipairs(hobby) do %}

        {% if v == '电影' then %} - xxoo

            {%else%} - {\* v \*}

{% end %}

    {% 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)}

​

### layout 布局统一风格

使用模板内容嵌套可以实现全站风格同一布局

#### **lua**

local template = require "resty.template"

一、

local layout = template.new "layout.html"

layout.title = "Testing lua-resty-template"

layout.view = template.compile "view.html" { message = "Hello, World!" }

layout:render()

二、

template.render("layout.html", {

title = "Testing lua-resty-template",

msg = "type=2",

view = template.compile "view.html" { message = "Hello, World!" }

})

三、

此方式重名变量值会被覆盖

local view = template.new("view.html", "layout.html")

view.title = "Testing lua-resty-template"

view.msg = "type=3"

view.message = "Hello, World!"

view:render()

四、

可以区分一下

local layout = template.new "layout.html"

layout.title = "Testing lua-resty-template"

layout.msg = "type=4"

local view = template.new("view.html", layout)

view.message = "Hello, World!"

view:render()

#### **layout.html**

<!DOCTYPE html>

<html>

<head>

​ <title>{{title}}</title>

</head>

<h1>layout</h1>

<body>

​ {\*view\*}

</body>

</html>

#### **view.html·**

msg:{{message}}

#### **多级嵌套**

lua

local view = template.new("view.html", "layout.html")

view.title = "Testing lua-resty-template"

view.message = "Hello, World!"

view:render()

view.html

{% layout="section.html" %}

<h1>msg:{{message}}</h1>section.html

<divid="section">

​ {view} - sss

</div>

layout.html

<!DOCTYPE html>

<html>

<head>

​ <title>{{title}}</title>

</head>

<h1>layout {{msg}}</h1><body>

​ {view}

</body>

</html>

## **IDE Lua 脚本调试**

### EmmyLua插件

<https://github.com/EmmyLua/IntelliJ-EmmyLua>

<https://emmylua.github.io/zh_CN>/

### LDT 基于eclipse

<https://www.eclipse.org/ldt>/

## **限流**

### 限流算法

#### **漏桶算法**​

#### **令牌桶算法**

#### **计数器**

常见的连接池、线程池等简单粗暴的按照数量限流的方式

### Tomcat限流

server.xml配置文件中的Connector节点

<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443"

maxConnections="800" acceptCount="500" maxThreads="400" />

l maxThreads：tomcat能并发处理的最大线程数

l acceptCount：当tomcat起动的线程数达到最大时，接受排队的请求个数，默认值为100

l maxConnections：瞬时最大连接数，超出会排队等待

## **Lua 开源项目**

### WAF

<https://github.com/unixhot/waf>

<https://github.com/loveshell/ngx_lua_waf>

l 防止 SQL 注入，本地包含，部分溢出，fuzzing 测试，XSS/SSRF 等 Web 攻击

l 防止 Apache Bench 之类压力测试工具的攻击

l 屏蔽常见的扫描黑客工具，扫描器

l 屏蔽图片附件类目录执行权限、防止 webshell 上传

l 支持 IP 白名单和黑名单功能，直接将黑名单的 IP 访问拒绝

l 支持 URL 白名单，将不需要过滤的 URL 进行定义

l 支持 User-Agent 的过滤、支持 CC 攻击防护、限制单个 URL 指定时间的访问次数

l 支持支持 Cookie 过滤，URL 与 URL 参数过滤

l 支持日志记录，将所有拒绝的操作，记录到日志中去

### Kong 基于Openresty的流量网关

<https://konghq.com>/

<https://github.com/kong/kong>

Kong 基于 OpenResty，是一个云原生、快速、可扩展、分布式的微服务抽象层（Microservice Abstraction Layer），也叫 API 网关（API Gateway），在 Service Mesh 里也叫 API 中间件（API Middleware）。

Kong 开源于 2015 年，核心价值在于高性能和扩展性。从全球 5000 强的组织统计数据来看，Kong 是现在依然在维护的，在生产环境使用最广泛的 API 网关。

Kong 宣称自己是世界上最流行的开源微服务 API 网关（The World’s Most Popular Open Source Microservice API Gateway）。

核心优势：

l 可扩展：可以方便的通过添加节点水平扩展，这意味着可以在很低的延迟下支持很大的系统负载。

l 模块化：可以通过添加新的插件来扩展 Kong 的能力，这些插件可以通过 RESTful Admin API 来安装和配置。

l 在任何基础架构上运行：Kong 可以在任何地方都能运行，比如在云或混合环境中部署 Kong，单个或全球的数据中心。

### ABTestingGateway

<https://github.com/CNSRE/ABTestingGateway>

ABTestingGateway 是一个可以动态设置分流策略的网关，关注与灰度发布相关领域，基于 Nginx 和 ngx-lua 开发，使用 Redis 作为分流策略数据库，可以实现动态调度功能。

ABTestingGateway 是新浪微博内部的动态路由系统 dygateway 的一部分，目前已经开源。在以往的基于 Nginx 实现的灰度系统中，分流逻辑往往通过 rewrite 阶段的 if 和 rewrite 指令等实现，优点是性能较高，缺点是功能受限、容易出错，以及转发规则固定，只能静态分流。ABTestingGateway 则采用 ngx-lua，通过启用 lua-shared-dict 和 lua-resty-lock 作为系统缓存和缓存锁，系统获得了较为接近原生 Nginx 转发的性能。

l 支持多种分流方式，目前包括 iprange、uidrange、uid 尾数和指定uid分流

l 支持多级分流，动态设置分流策略，即时生效，无需重启

l 可扩展性，提供了开发框架，开发者可以灵活添加新的分流方式，实现二次开发

l 高性能，压测数据接近原生 Nginx 转发

l 灰度系统配置写在 Nginx 配置文件中，方便管理员配置

l 适用于多种场景：灰度发布、AB 测试和负载均衡等

# 《亿级流量系统多级缓存架构3》

## **服务并发化**

## **其他缓存**

### 客户端

#### **浏览器缓存**

首先，限定在get请求

由于浏览器缓存涉及到的应用比较多，所以针对不同的应用和版本效果也会有差异

浏览器：chrome

服务器：nginx

在系统中不常改变的资源上可以使用浏览器缓存，

其中在frame下

Cache-Control: max-age=2592000

ETag: "5d8c4a06-a0fc"

Expires: Sat, 26 Oct 2019 15:16:24 GMT

##### ****ETag**:**

http1.1支持

在HTTP协议中If-Modified-Since和If-None-Match分别对应Last-Modified和ETag

Entity Tag 的缩写，中文译过来就是实体标签的意思.

HTTP中并没有指定如何生成ETag，哈希是比较理想的选择。

在计算Etag的时候，会产生CPU的耗费，所以也可以用时间戳，但这样直接使用Last-Modified即可。

ETag 用来校验用户请求的资源是否有变化，作用和lastmodified很像，区别是lastmodified精确到秒，ETag可以用hash算法来生成更精确的比对内容。

当用户首次请求资源的时候返回给用户数据和200状态码并生成ETag，再次请求的时候服务器比对ETag，没有发生变化的话返回304

**java实现**

package com.mashibing.httpcache.controller;

​

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.Locale;

​

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

​

@RestController

@RequestMapping("/cache")

public class CacheController {

@RequestMapping("/")

public ResponseEntity<String> last(@RequestHeader(value="IF-Modified-Since",required = false) Date ifModifiedSince) {

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss 'GMT'", Locale.US);

long now = System.currentTimeMillis() / 1000 \*1000;

HttpHeaders headers = new HttpHeaders();

String body = "<a href =''>hi点我</a>";

String ETag = getMd5(body);

headers.add("Date", simpleDateFormat.format(new Date(now)));

headers.add("ETag", ETag);

return new ResponseEntity<>(body,headers,HttpStatus.OK);

}

/\*\*

\* 字符串转md5

\* @param msg

\* @return

\*/

private String getMd5(String msg) {

MessageDigest md5 = null;

try {

md5 = MessageDigest.getInstance("MD5");

} catch (NoSuchAlgorithmException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

md5.update(msg.getBytes());

byte[] digest = md5.digest();

StringBuffer buf = null;

buf = new StringBuffer(digest.length \* 2);

//遍历

for (int i = 0; i < digest.length; i++) {

if (((int) digest[i] & 0xff) < 0x10) { //(int) b[i] & 0xff 转换成无符号整型

buf.append("0");

}

//Long.toHexString( 无符号长整数的十六进制字符串表示

buf.append(Long.toHexString((int) digest[i] & 0xff));

}

return buf.toString();

}

​

}

##### ****Cache-Control**、 Last-Modified 、Expires**

Last-Modified : 表示文档最后修改时间，浏览器在访问重复资源的时候会发送IF-Modified-Since 携带此时间去服务器验证，如果时间匹配则返回304，浏览器加载本地资源

Expires： 文档过期时间，在浏览器内可以通过这个时间来判断是否发送请求

Cache-Control ：http1.1的规范，使用max-age表示文件可以在浏览器中缓存的时间以秒为单位

**Cache-Control**缓存头,分为响应头和请求头

| **标记** | **类型** | **功能** |
| --- | --- | --- |
| public | 响应头 | 响应的数据可以被缓存，客户端和代理层都可以缓存 |
| private | 响应头 | 可私有缓存，客户端可以缓存，代理层不能缓存（CDN，proxy\_pass） |
| no-cache | 请求头 | 可以使用本地缓存，但是必须发送请求到服务器回源验证 |
| no-store | 请求和响应 | 应禁用缓存 |
| max-age | 请求和响应 | 文件可以在浏览器中缓存的时间以秒为单位 |
| s-maxage | 请求和响应 | 用户代理层缓存，CDN下发，当客户端数据过期时会重新校验 |
| max-stale | 请求和响应 | 缓存最大使用时间，如果缓存过期，但还在这个时间范围内则可以使用缓存数据 |
| min-fresh | 请求和响应 | 缓存最小使用时间， |
| must-revalidate | 请求和响应 | 当缓存过期后，必须回源重新请求资源。比no-cache更严格。因为HTTP 规范是允许客户端在某些特殊情况下直接使用过期缓存的，比如校验请求发送失败的时候。那么带有must-revalidate的缓存必须校验，其他条件全部失效。 |
| proxy-revalidate | 请求和响应 | 和must-revalidate类似，只对CDN这种代理服务器有效，客户端遇到此头，需要回源验证 |
| stale-while-revalidate | 响应 | 表示在指定时间内可以先使用本地缓存，后台进行异步校验 |
| stale-if-error | 响应 | 在指定时间内，重新验证时返回状态码为5XX的时候，可以用本地缓存 |
| only-if-cached | 响应 | 那么只使用缓存内容，如果没有缓存 则504 getway timeout |

在浏览器和服务器端验证文件是否过期的时候，浏览器在二次请求的时候会携带IF-Modified-Since属性

**Cache-Control和ETag的区别**Cache-Control直接是通过不请求来实现，而ETag是会发请求的，只不过服务器根据请求的东西的内容有无变化来判断是否返回请求的资源

java实现**Cache-Control** **Last-Modified:**

**Controller**

package com.mashibing.httpcache.controller;

​

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.Locale;

import java.util.Map;

​

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

​

@RestController

@RequestMapping("/cache")

public class CacheController {

private MyFile file = MyFile.getInstance();

@RequestMapping("/")

public ResponseEntity<String> last(@RequestHeader(value="IF-Modified-Since",required = false) Date ifModifiedSince) {

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss 'GMT'", Locale.US);

long now = System.currentTimeMillis() / 1000 \*1000;

// 系统当前时间

System.out.println(now);

// 缓存时间

long maxAge = 20;

HttpHeaders headers = new HttpHeaders();

if (null != ifModifiedSince && ifModifiedSince.getTime() == file.getLastModified() ) {

System.out.println(304);

}

headers.add("Date", simpleDateFormat.format(new Date(now)));

headers.add("Expires", simpleDateFormat.format(new Date(now + maxAge \* 1000)));

headers.add("Cache-Control", "max-age="+maxAge);

headers.add("Last-Modified", simpleDateFormat.format(new Date(file.getLastModified())));

String body = "<a href =''>hi点我</a>";

return new ResponseEntity<>(body,headers,HttpStatus.OK);

}

​

}

**Bean**

package com.mashibing.httpcache.controller;

​

public class MyFile {

​

private String name;

private long lastModified;

private long expireTime;

private static MyFile file ;

private MyFile() {

​

this.name = "file...";

this.lastModified = System.currentTimeMillis() /1000 \* 1000;

this.expireTime = 10 \*1000;

}

public long getExpireTime() {

return expireTime;

}

public void setExpireTime(long expireTime) {

this.expireTime = expireTime;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public long getLastModified() {

return lastModified;

}

public void setLastModified(long lastModified) {

this.lastModified = lastModified;

}

​

public static MyFile getInstance() {

// TODO Auto-generated method stub

if(file==null) {

file = new MyFile();

}

return file;

}

}​

##### **强制刷新**

在强制刷新的时候浏览器就不在发送IF-Modified-Since了，而会带上

##### **from disk cache & from memory cache**

可以验证请求是否使用了浏览器缓存和是否发送请求给服务器端。

当点击链接、引入外部资源和浏览器的前进后退的时候。

**from memory cache**

字面理解是从内存中，其实也是字面的含义，这个资源是直接从内存中拿到的，**不会请求服务器**一般已经加载过该资源且缓存在了内存当中，当关闭该页面时，此资源就被内存释放掉了，再次重新打开相同页面时不会出现from memory cache的情况

**from disk cache**

是从磁盘当中取出的，也是在已经在之前的某个时间加载过该资源，**不会请求服务器**但是此资源不会随着该页面的关闭而释放掉，因为是存在硬盘当中的，下次打开仍会from disk cache

不做深入研究

js脚本，css，图片，音视频，字体

##### **Age**

是CDN添加的属性表示在CDN中缓存了多少秒

##### ****via****

用来标识CDN缓存经历了哪些服务器，缓存是否命中，使用的协议

#### **浏览器缓存原则**

**首页**可以看做是框架 应该禁用缓存，以保证加载的资源都是最新的

还有一些场景下我们希望禁用浏览器缓存。比如轮训api上报数据数据

浏览器缓存很难彻底禁用，大家的做法是加版本号，随机数等方法。

只缓存200响应头的数据，像3XX这类跳转的页面不需要缓存。

对于js，css这类可以缓存很久的数据，可以通过加版本号的方式更新内容

不需要强一致性的数据，可以缓存几秒

异步加载的接口数据，可以使用ETag来校验。

在服务器添加Server头，有利于排查错误

#### **应用缓存**

分为手机APP和Client以及是否遵循http协议

在没有联网的状态下可以展示数据

流量消耗过多

漂亮的加载过程

提前下发 避免秒杀时同时下发数据造成流量短时间暴增

兜底数据 在服务器崩溃和网络不可用的时候展示

临时缓存 退出即清理

固定缓存 展示框架这种，可能很长时间不会更新，可用随客户端下发

父子连接 页面跳转时有一部分内容不需要重新加载，可用从父菜单带过来

预加载 某些逻辑可用判定用户接下来的操作，那么可用异步加载那些资源

异步加载 先展示框架，然后异步加载内容，避免主线程阻塞

# 亿级流量系统多级缓存架构4

## **日志**

## **异常控制/错误编码**

细分异常，多下功夫，功夫不是白费的，有可能一天、一个月、一年一个错误编码也没抛出来，但是一旦出了问题能够急速定位，根本不用看代码。

我们永远不能保证系统没有bug，bug可以藏的 很深埋的很久，但我们不怕，因为我们的伏兵也一 直在，你一跳我们立马抓，毫不犹豫。

## **异步**

异步本身不是什么高深的技术，关键是哪些 业务可以走异步，这更体现架构师的业务理解能力 和综合能力

## **6个9**

99.9999%

可用性、数据一致性

## **ACID特性**

以传统数据库为代表，依然坚守：原子性、一致性、隔离性、持久性

## **CAP原则**

EricBrewer

<https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem>/

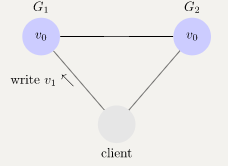
对于开发或设计分布式系统的架构师工程师来说，CAP是必须要掌握的理论。

CAP定理又被成为布鲁尔定理，是加州大学计算机科学家埃里克·布鲁尔提出来的猜想，后来被证明成为分布式计算领域公认的定理

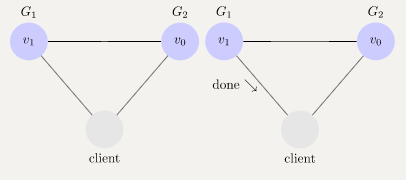
CAP定义，在高并发的场景下要做取舍，在大型集群中分区容错很难保证，一旦要确保容错性，那么就会损失数据一致性和高可用特性。

#### ****Consistency**** -> 一致性

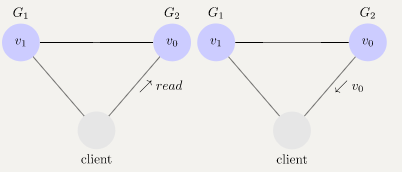
G1，G2为数据节点，同时存储了键值对 key=v:value=0



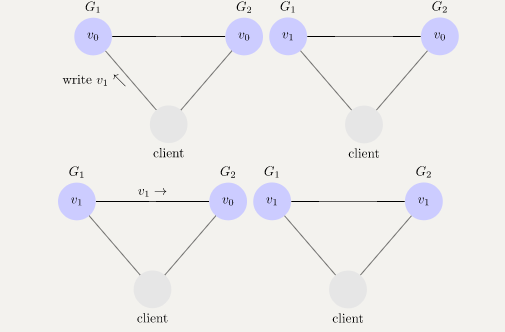
⭕️ 上图：向G1 写入数据

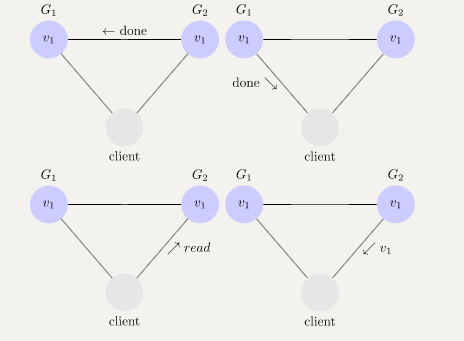


⭕️ 上图：数据写入完成，G1->v:1，写入完成后，在向G1读取数据的时候就会得到v:1，此时是一致性



⭕️ 上图：那么此时如果向G2发起读请求的话，因为数据没有同步，就会得到v:0,此时数据不一致





上图：如果流程变成这样的

写入G1

G1向G2同步数据

等待同步完成

通知写完成

读取数据

似乎得到了一致性

一致性是指分布式系统中，数据在多节点存在副本，那么数据如果**一直不修改**，在读的时候是不存在问题的，你访问哪个节点的数据都一样可一旦要是发生了**修改**，那么数据同步无法在修改的**瞬间**广播到所有副本节点那么在读的时候就可能发生**数据脏读**

脑裂

数据库

item\_id = 100,谁在改，时间

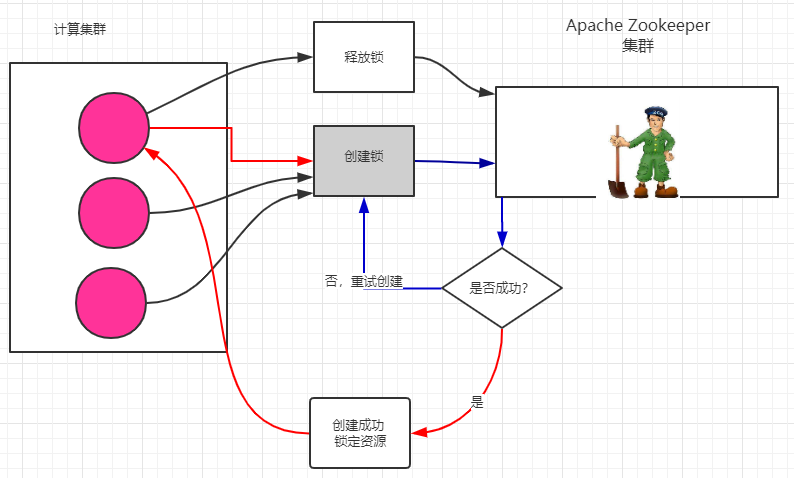
单点/AP

Redis 高性能

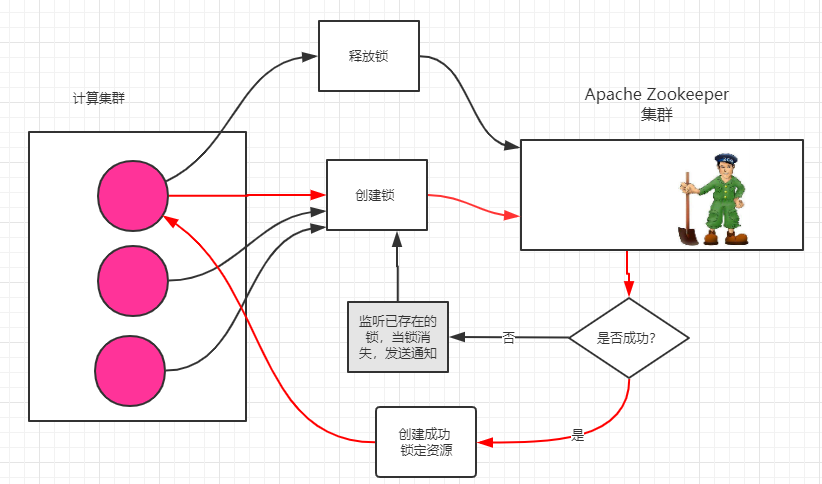
A/P

#### **Zookeeper**

饥渴式

****

懒汉式

****

**数据篡改**

拜占庭将军问题

分布式系统 一致性问题

* 强一致性
* 弱一致性
* 最终一致性

CAP中的一致性

**Availability** -> 可用性

指的是服务是否可用，范围涵盖终端客户访问我们的系统或者是集群内部相互通讯交换数据

也就是说在Client向Server发起请求时，服务器返回了正确的响应，称之为可用，反之为不可用

**这里有一个问题，如果发送请求在300年后给我返回数据了，那么算不算可用？**

所以要提出访问延迟的概念，在某个时间范围内响应才算可用。

**1s法则**

1S法则”是面向Web侧，H5链路上加载性能 和体验方向上的一个指标，具体指：

* “强网” (4G/WIFI)下，1秒完全完成页面加载，包括首屏 资源，可看亦可用;
* 3G下1秒完成首包的返回 ;
* 2G下1秒完成建连。

**高可用架构在实施的时候可以分两个方向**

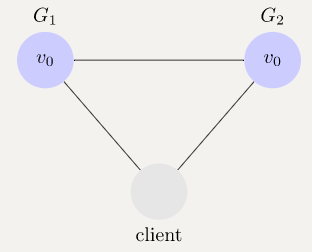
#### **客户端容错**

例如：游戏服务器

#### **服务器端容错**

例如：Nginx负载均衡

#### Partition tolerance -> 分区容错性



啥叫**分区容错**？

发生在分布式系统**内部**互访通信，是指分布式网络中部分**网络**不可用，但系统依然正常对外提供**服务**。

上图中 G1,G2是两台服务器，G1 向 G2 发送一条消息，G2 可能无法收到

比如：北京的订单系统，访问上海的库存系统

分区容错性是指分区具有容错性，我们可以尽可能的提高容错性，但是无法避免，

如果发生失败，就要在A和C之间做出选择。要么停止系统进行错误恢复，要么继续服务但是降低一致性，所以我们说只能保证AP或CP。

在互相隔离的空间中，提供数据服务的系统。CAP抽象：不同空间的数据，在同一时间，状态一致。

C：代表状态一致A：代表同一时间P：代表不同空间

**CP 不同空间中的数据，如果要求他们所有状态一致，则必然不在同一时间。**

**AP 不同空间中，如果要求同一时间都可以从任意的空间拿到数据，则必然数据的状态不一致。**

**CA 不同空间的数据，如果要求任意时间都可以从任意空间拿到状态一致的数据，则空间数必然为1.**

**要强一致性的地方**:

* 唯一ID生成, 这种性能很差. 并发不高.
* 对一致性要求比较高的系统，例如银行转账

要高可用: 最终一致, 即有可能读到脏数据. 但是一段时间之后总是能够读到新数据.

### Zookeeper和Eureka

**zookeepr**

保证CP，即任何时刻对zookeeper的访问请求能得到一致性的数据结果，同时系统对网络分割具备容错性，但是它不能保证每次服务的可用性。从实际情况来分析，在使用zookeeper获取服务列表时，如果zk正在选举或者zk集群中半数以上的机器不可用，那么将无法获取数据。所以说，zk不能保证服务可用性。

**eureka**

保证AP，eureka在设计时优先保证可用性，每一个节点都是平等的，一部分节点挂掉不会影响到正常节点的工作，不会出现类似zk的选举leader的过程，客户端发现向某个节点注册或连接失败，会自动切换到其他的节点，只要有一台eureka存在，就可以保证整个服务处在可用状态，只不过有可能这个服务上的信息并不是最新的信息。

## **BASE 理论**

eBay的架构师Dan Pritchett**源于对大规模分布式系统的实践总结**，在ACM上发表文章提出BASE理论，BASE理论是对CAP理论的延伸，核心思想是即使无法做到强一致性（StrongConsistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。

ACM国际大学生程序设计竞赛（英文全称：ACM International Collegiate Programming Contest（简称ACM-ICPC或ICPC））

### Basically Available

**基本可用**

在分布式系统出现故障的时候，允许损失部分可用性,支持分区失败，即保证核心可用。

##### **Soft State**

**软状态**

接受一段时间的状态不同步，及中间状态，而改中间状态不影响系统整体可用性。这里的中间状态就是CAP理论中的数据不一致性。

##### **Eventually Consistent**

**最终一致性**

上面说软状态，然后不可能一直是软状态，必须有个时间期限。在期限过后系统能够保证在没有其他新的更新操作的情况下，数据最终一定能够达到一致的状态，因此所有客户端对系统的数据访问最终都能够获取到最新的值。

强一致性

# 亿级流量系统多级缓存架构5

**资源隔离服务降级**

什么叫资源隔离？

服务隔离是指在一个大型系统中，可以把原连接在一起的组件，模块，服务，资源拆分开。那么在系统出现故障的时候，可以隔离故障，阻止传播，不会出现滚雪球和雪崩的效应。隔离的方式主要有。

* 线程隔离
* 进程隔离
* 集群隔离
* 机房隔离
* 读写隔离
* 动静隔离
* 爬虫隔离
* 等....

**线程隔离**

主要是在多线程环境下，对线程池进行治理，把核心业务和非核心业务分割开。但是在多线程池下，不同线程池中的线程

**在使用Netty的时候**

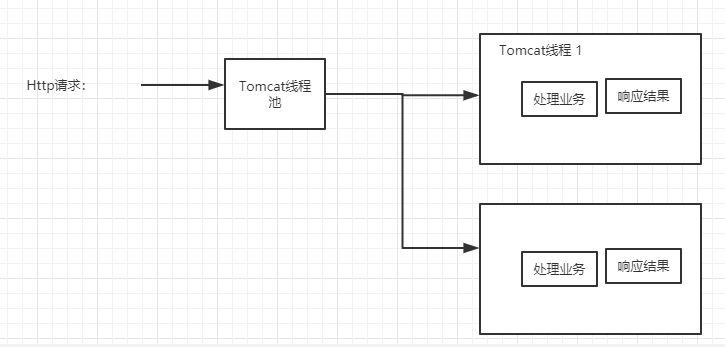
netty本身是负载网络io的框架，想要做到网络服务和业务隔离， 首先它不应该和项目中的业务逻辑在同一线程池中

如果同一线程池中

**Tomcat**中的资源隔离

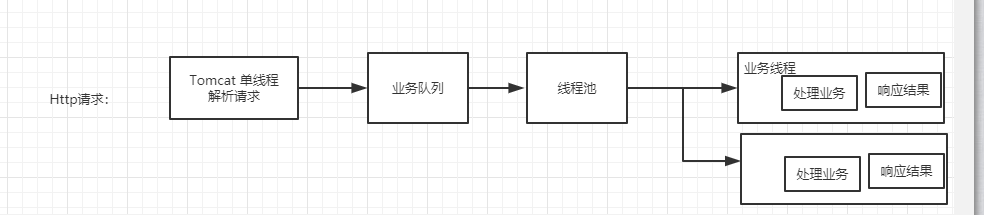
**servlet3请求隔离**

servlet 2.0



需要Tomcat7以上版本

Tomcat把连接介入和业务处理拆分成两个线程池来处理，即

****

**Connector** 介入连接，配置socket处理方式

可选

* Http11Protocol Http11NioProtocol Http11Nio2Protocol
* Http11AprProtocol

Connector\*\*

在异介入连接，可以采用BI/O，nio，ajp,apr.

性能提现的是qps，即介入连接数

**Executor** **Servlet**

可以使用独立的线程池来维护servlet的创建。

**那么**

connector能介入的请求肯定比业务复杂的servlet处理的个数要多，在中间，Tomcat还加入了队列，来等待servlet线程池空闲。

这两步是Tomcat内核完成的，在一阶段无法区分具体业务或资源，所以只能在连接介入，servlet初始化完成后我们根据自己的业务线去划分独立的连接池。

那么在独立的业务或资源中如果出现崩溃，不会影响其他的业务线程，从而达到资源隔离和服务降级的效果。

在使用了servlet3之后，系统线程隔离变得更灵活了。可以划分核心业务队列和非核心业务队列，

**但是**

资源一旦出现问题，虽然是隔离状态，想要让资源重新可用，很难做到不重启jvm。

线程池内部线程如果出现OOM、FullGC、cpu耗尽等问题也是无法控制的

**结论**

**线程隔离，只能保证在分配线程这个资源上进行隔离，并不能保证整体稳定性**

#### ****进程隔离****

在项目最初期，一般都是allinone的技术架构，然后做负载均衡，session共享。

通过线程隔离无法完全避免雪崩。

java cpu、内存这些资源可以通过不同的虚拟机进程来做隔离。

集群式

分布式

#### **集群隔离**

如果系统中某个业务模块包含像

* 抢购、秒杀
* 存储I/O密集度高
* 网络I/o高
* 计算I/O高

这类需求的时候，很容易在并发量高的时候因为这种功能把整个模块占有的资源全部耗尽，导致响应编码甚至节点不可用。

解决方案：

* 独立拆分模块
* 微服务化

可以使用hystrix在微服务中隔离分布式服务故障。他可以通过线程和信号量进行隔离。

#### **机房隔离异地多活**

解决数据容量大，计算，i/o（网络）密集度高的问题。把服务建立整体副本（计算服务、数据存储），在多机房内做异地多活或冷备份、是微服务数据异构的放大版。当在机房层面出现问题的时候，可以通过智能dns、httpdns、负载均衡等技术快速切换

数据（读写）分离：

通过主从模式，将mysql、redis等数据存储服务集群化，读写分离，那么在写入数据不可用的时候，也可以通过**重试机制**临时通过其他节点读取到数据。

多节点在做子网划分的时候，除了异地多活，还可以做数据中心，所有数据在本地机房crud 异步同步到数据中心，数据中心再去分发数据给其他机房

那么数据临时在本地机房不可用的时候，就可以尝试连接异地机房或数据中心。

#### **动静隔离**

把静态资源从原有服务中隔离出来，参考cdn技术

#### **爬虫隔离**

目前我们开发的都是API接口，并且多数都是开放的API接口。也就是说，只要有人拿到这个接口，任何人都可以通过这个API接口获取数据，那么像网络爬虫的，请求速度又快，获取的数据又多，不用多久，爬虫方完全可以用我们API的接口来开发一个同样的网站，这样的话，后果就有点严重了，所以我们需要限流，限制访问的频率

开放平台的API接口调用需要限制其频率，以节约服务器资源和避免恶意的频繁调用

在大型互联网项目中，对于web服务和网络爬虫的访问流量能达到5:1，甚至更高，有的系统有时候就会因为爬虫流量过高而导致资源耗尽，服务不可用。

**限流维度**

* 登录/会话限制
* 下载限流
* 访问频率
* ip限制，黑白名单

想要分辨出来一个访问是不是爬虫，可以简单的使用nginx来分析ua处理

nginx不仅可以处理ua来分离流量，还可以通过更强大的openresty来完成更复杂的逻辑，实现一个流量网关，软防火墙。

#### **资源隔离**

#### 磁盘

数据库

# 亿级流量系统多级缓存架构6

## **服务限流**

### 什么叫限流？

Ab测试

yum install httpd-tools

即限制流量进入

**缓存**，是用来增加系统吞吐量，提升访问速度提供高并发。

**降级**，是在系统某些服务组件不可用的时候、流量暴增、资源耗尽等情况下，暂时屏蔽掉出问题的服务，继续提供降级服务，给用户尽可能的友好提示，返回兜底数据，不会影响整体业务流程，待问题解决再重新上线服务

**限流**，是指在使用缓存和降级无效的场景。比如当达到阈值后限制接口调用频率，访问次数，库存个数等，在出现服务不可用之前，提前把服务降级。只服务好一部分用户。

在我们使用微信、支付宝、短信等等这些api的时候，每个接口都会有调用上的限流。

限流是对某一时间窗口内的请求数进行限制，保持系统的可用性、稳定性和安全性，防止因流量暴增而导致的系统运行缓慢或宕机，

app 统计

友盟 cnzz .....

### 限流算法

#### **计数器算法**

简单粗暴

比如线程池大小，数据库连接池大小、nginx连接数等都属于计数器算法。

全局或某段时间范围达到阈值则限流。

#### **漏桶算法**

削峰

缓冲

消费速度固定 因为计算性能固定

保证桶不能忙

#### **令牌桶算法**

平滑的流入速率限制，消费/秒。

可以用于对外服务接口，内部集群调用

**区别**

令牌桶是按照固定速率从桶里拿令牌消费，如果令牌为0，则拒绝新请求

漏桶是按照固定速率流出请求，流入速率不控制，当桶内请求达到阈值，新请求则被拒绝。

令牌桶支持每次拿多个令牌，平均流入速率，并支持突发流入，还可以支持缓慢提升流入速度

#### **并发限流**

设置系统阈值总的qps个数

Tomcat中配置的

* acceptCount 响应连接数
* maxConnections 瞬时最大连接数
* maxThreads 最大线程数

#### **接口限流**

##### **接口总数**

可以使用atomic类或者semaphore进行限流

这种方式简单粗暴。没有平滑处理。使用限制某个接口的总并发数，或限制某账号服务调用总次数。

比如某些开放平台限制试用账号。

if (atomic.incrementAndGet() > 100){

   // 拒绝

}finally{

atomic.decrementAndGet();

​

}

##### **接口时间窗口**

此时可以使用Guava Cache，类似于一个ConcurrentMap,但并不完全一样。

最基础的不同是ConcurrentMap保存所有的元素知道它们被明确删除,Guava Cache可以配置自动过期

//计数器

counter;

// 限制数量

limit;

// 限制单位 1000=秒

unit;

// 获得当前时间

current = system.currentTimeMillis() / unit

//判断时间窗内是否限制访问

​

if (counter.get(current).incrementAndGet() > limit){

   // 拒绝

}

### 使用guava实现

**引入包**

<!-- https://mvnrepository.com/artifact/com.google.guava/guava -->

<dependency>

   <groupId>com.google.guava</groupId>

   <artifactId>guava</artifactId>

   <version>28.1-jre</version>

</dependency>

**核心代码**

LoadingCache<Long, AtomicLong> counter = CacheBuilder.newBuilder().

expireAfterWrite(2, TimeUnit.SECONDS)

.build(new CacheLoader<Long, AtomicLong>() {

​

@Override

public AtomicLong load(Long secend) throws Exception {

// TODO Auto-generated method stub

return new AtomicLong(0);

}

});

counter.get(1l).incrementAndGet();

#### **令牌桶实现**

**稳定模式(SmoothBursty:令牌生成速度恒定)**

public static void main(String[] args) {

// RateLimiter.create(2)每秒产生的令牌数

RateLimiter limiter = RateLimiter.create(2);

       // limiter.acquire() 阻塞的方式获取令牌

System.out.println(limiter.acquire());;

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

}

`RateLimiter.create(2) 容量和突发量，令牌桶算法允许将一段时间内没有消费的令牌暂存到令牌桶中，用来突发消费。

**渐进模式(SmoothWarmingUp:令牌生成速度缓慢提升直到维持在一个稳定值)**

// 平滑限流，从冷启动速率（满的）到平均消费速率的时间间隔

RateLimiter limiter = RateLimiter.create(2,1000l,TimeUnit.MILLISECONDS);

System.out.println(limiter.acquire());;

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

System.out.println(limiter.acquire());;

**超时**

boolean tryAcquire = limiter.tryAcquire(Duration.ofMillis(11));

在timeout时间内是否能够获得令牌，异步执行

### 分布式系统限流

#### **Nginx + Lua**

可以使用resty.lock保持原子特性，请求之间不会产生锁的重入

<https://github.com/openresty/lua-resty-lock>

使用lua\_shared\_dict存储数据

local locks = require "resty.lock"

​

local function acquire()

   local lock =locks:new("locks")

   local elapsed, err =lock:lock("limit\_key") --互斥锁 保证原子特性

   local limit\_counter =ngx.shared.limit\_counter --计数器

​

   local key = "ip:" ..os.time()

   local limit = 5 --限流大小

   local current =limit\_counter:get(key)

​

   if current ~= nil and current + 1> limit then --如果超出限流大小

      lock:unlock()

      return 0

   end

   if current == nil then

      limit\_counter:set(key, 1, 1) --第一次需要设置过期时间，设置key的值为1，

过期时间为1秒

   else

       limit\_counter:incr(key, 1) --第二次开始加1即可

   end

   lock:unlock()

   return 1

end

ngx.print(acquire())

**nginx配置**

# 扩容

扩容的原因

单体应用 -> 平均

## **硬件与云服务扩容**

### 整机硬件

整机：IBM、浪潮、DELL、HP等

### cpu

Intel、Amd 单路，双路，四路

### 网卡

千兆 -> 万兆

### 内存

ECC校验

### 磁盘

SCSI HDD（机械）、HHD（混合）、SATA SSD、PCI-e SSD、 MVMe SSD

## **水平扩容**

复制

统一入口 负载均衡

## **系统拆分**

随着业务的增长，系统变得越来越庞大

根据系统功能拆分成独立而又互通的项目

比如交易系统、财务系统、生产流程系统、物流系统、网站系统等等

拆分完成会发生以下问题

系统间共享数据

中央数据中心，数据库集群

独立资源，消息中间件异步消费数据

接口调用

rpc rmi dubbo

http restful

持久化数据雪崩

数据库分库分表

资源隔离

对访问量大的

静态资源分布在各个系统中

防刷

cdn技术统一分发管理

高并发问题

缓存

数据闭环

数据一致性问题

分布式锁

### 多库join

#### **增加冗余字段设计**

#### **向多库发n次查询**

#### **复制表到本地数据库**

#### **使用搜素引擎**

#### **连接表**

以MySQL为例，链接表与常见的 MyISAM, InnoDB 等等，都是一种表的结构类型（称之为 存储引擎）。使用 show engines; 命令即可看到数据库所支持的存储引擎，默认 FEDERATED 引擎 是关闭的，我们需要去mysql配置文件开启它。

开启的方法很简单，在windows下只需要在mysql的配置文件 my.ini 最末尾加上一句 federated ，然后重启mysql即可。重启后输入 show engines;命令即可看到 FEDERATED 的 Support 变为 YES，表示federated引擎已经开启。

**链接的表，那么会有以下几个限制：**

本地的表结构必须与远程的完全一样

远程数据库目前仅限MySQL

不支持事务

不支持表结构修改

**一个例子**

       CREATE TABLE TABLE2 (

      id     INT(20) NOT NULL AUTO\_INCREMENT,

      name   VARCHAR(32) NOT NULL DEFAULT '',

      other  INT(20) NOT NULL DEFAULT '0',

      PRIMARY KEY (id),

      INDEX name (name),

      INDEX other\_key (other)

      )

      ENGINE=FEDERATED

      DEFAULT CHARSET=latin1

      CONNECTION='mysql://192.168.1.2:3307/DB2/TABLE2';

## **分布式ID**

并发场景下的id生成策略

### 分布式ID需求

全局唯一性：不能出现重复的ID号，既然是唯一标识，这是最基本的要求。

递增：保证下一个ID一定大于上一个ID，例如事务版本号、IM增量消息、排序等特殊需求。

信息安全：如果ID是连续的，恶意用户的扒取工作就非常容易做了，直接按照顺序下载指定URL即可；如果是订单号就更危险了，竞对可以直接知道我们一天的单量。所以在一些应用场景下，会需要ID无规则、不规则。

极高的可用性

高并发

单机id 自增和随机哪个效率高？

### 分布式ID方案

#### **全局唯一数据库**

集群中唯一的一台数据库服务器

**特点**

实现简单

性能一般，难扩展

维护

整形id自增，分页，排序等需求很方便

由于每个数据库产品对id生成策略实现不同，数据库中的数据迁移跨数据库平台需要处理

单点故障问题

#### **数据库集群生成方案**

避免重复，设置成起始数字不一样，步长一样

#### **UUID**

可以用数据库生成，也可以用代码生成

简单

不可读

变种，替换或增加字段用来存储生成时间

生成性能好，没有网络开销

全球唯一，数据合并一般不会出问题

mac地址，原子钟

保证服务器时间不回退

无法排序，没有递增关系

字符串存储空间大，查询效率较低

数据传输量大

#### **zookeeper**

自身高可用，原子性

有序，使用临时序号节点

使用节点的数据版本号

高并发下性能不好

#### **MongoDB**

使用ObjectId，原生分布式数据库

#### **Redis**

单线程的redis可以提供原子特性及高性能

实现分布式id可以用Redis的原子操作 INCR和INCRBY来实现起始数字不一样，步长一样

高性能

数字id，排序，分页方便

局部有序，全局无法保证有序

#### **snowflake**

snowflake是Twitter开源的分布式ID生成算法，结果是一个long型的ID

这种方案大致来说是一种以划分命名空间（UUID也算，由于比较常见，所以单独分析）来生成ID的一种算法，这种方案把64-bit分别划分成多段，分开来标示机器、时间等，比如在snowflake中的64-bit分别表示如下图

性能好

单机上是递增的，强依赖机器时钟，分布式系统中无法保证每台机器时钟都一样，所以不是全局递增

机器上时钟回拨

## **分布式事务**

## **弹性扩容**

### 虚拟机

第一，虚拟机弹性能力较弱。使用虚拟机部署业务，在弹性扩容时，需要经过申请虚拟机、创建和部署虚拟机、配置业务环境、启动业务实例这几个步骤。前面的几个步骤属于私有云平台，后面的步骤属于业务工程师。一次扩容需要多部门配合完成，扩容时间以小时计，过程难以实现自动化。如果可以实现自动化“一键快速扩容”，将极大地提高业务弹性效率，释放更多的人力，同时也消除了人工操作导致事故的隐患。

第二，IT成本高。由于虚拟机弹性能力较弱，业务部门为了应对流量高峰和突发流量，普遍采用预留大量机器和服务实例的做法。即先部署好大量的虚拟机或物理机，按照业务高峰时所需资源做预留，一般是非高峰时段资源需求的两倍。资源预留的办法带来非常高的IT成本，在非高峰时段，这些机器资源处于空闲状态，也是巨大的浪费。

# 亿级流量系统多级缓存架构8-服务降级

在高并发场景下，当系统中的一些功能组件出现异常，无法继续提供服务器的时候，为了保证整体系统可用性，可以牺牲一部分功能依旧提供**有损服务**

## **SLA**

服务等级定义 SLA（Service Level Agreement）是判定压测是否异常的重要依据。压测过程中，通过监控核心服务状态的 SLA 指标数据，您可以更直观地了解压测业务的状态。

SLA则是服务商与您达成的正常运行时间保证。

详见

<https://help.aliyun.com/document_detail/111729.html?spm=a2c4g.11186623.3.3.19693e74AnPY9O>

SLA 分为网络服务/云服务提供商的在线保证率和项目自身故障率

## **6个9**

2个9 = （1-99%）X24 X 365 = 87.6 小时 = 3.65天

3个9 = （1-99.9%）X24 X 365 = 8.76 小时

4个9 = （1-99.99%）X24 X 365 = 0.876 小时 = 52.56分钟

5个9 = （1-99.999%）X24 X 365 = 0.0876 小时 = 5.256分钟

6个9 = （1-99.9999%）X24 X 365 = 0.00876 小时 = 0.5256分钟 = 31秒

**AP**

12306 强一致性

## **降级处理**

### 兜底数据

默认值: 设置安全的默认值，不会引起数据问题，比如库存为0

静态值:请求的页面或api无法返回数据，提供一套静态数据展示，比如加载失败提示重试，或默认菜单

缓存: 缓存无法更新便使用旧的缓存

## **限流降级**

当流量洪峰到达的时候，对于丢弃的用户可以提供友好的提示

比如提示用户当前繁忙稍后重试等

### 超时降级

对调用的数据设置超时时间，当调用失败，对服务降级

### 重试/自动处理

客户端高可用：提供多个可调用的服务地址

微服务重试：dubbo重试机制

API调用重试：当达到重试次数后，增加访问标记，服务降级，异步探测服务是否恢复。

WEB端：在服务不可用时，web端增加重试按钮或自动重试可以提供更友好的体验。

自动重试需设置重试次数和数据幂等处理

### 降级开关

在服务调用方设置一个flag，标记服务是否可用

存储在第三方，配置文件、数据库、redis、zookeeper中。

### 数据组装降级

多库/多维度组装JSON/XML时，如果有一些不重要的数据无法获取或数据出错，可以忽略继续。

### 爬虫和机器人

分析机器人行为：短时间连续操作，agent，行为轨迹、拖拽（模拟登陆/秒杀/灌水）

爬虫：引到到静态页或缓存页

### 读降级

在一个请求内，多级缓存架构下，后端缓存或db不可用，可以使用前端缓存或兜底数据

### 写降级

在cap原理和BASE理论中写操作存在于**数据一致性**这个环节，

降级的目的是为了提供高可用性，在多数的互联网架构中，可用性是大于数据一致性的。

所以丧失写入数据同步，通过上面的理论，我们也能勉强接受数据最终一致性。

高并发场景下，写入操作无法及时到达或抗压，可以异步消费数据/cache更新/log等方式

### 前端降级

当系统出现问题的时候，尽量让请求隔离在离用户最近的位置，避免无效链路访问

在后端服务部分或完全不可用的时候，可以使用本地缓存或兜底数据

在一些特殊场景下，对数据一致性要求不高的时候，比如秒杀、抽奖等可以做假数据

#### **js降级**

在js中埋降级开关，在访问不到达，系统阈值的时候可以避免发送请求

#### **接入层**

可以在接入层，在用户请求还没到达服务的时候，通过、Nginx + Lua、Haproxy + lua

过滤无效请求和服务降级

#### **应用层降级**

参考常老师项目中的springcloud

### 片段降级

页面中异步加载的数据，有一部分无法得到

js、css无法加载 -> 兜底 css

### 静态化处理

在活动前可以通过静态化技术，把原来需要调用多个动态接口的页面整体缓存下来保存成一个静态文件

当动态服务无法支撑的时候，降级为静态化服务

静态文件内尽量不去异步加载数据

静态文件包含可变内容，可以使用模板技术定时生成更新

### 提前预埋

APP 浏览器 js/css缓存大促前

# 亿级流量系统多级缓存架构9-分布式事务

## **为什么分布式事务？**

### 传统事务的基本特性（ACID）

**mysql**中，我们可以使用begin开始事务，rollback回滚事务，commit提交事务

redolog 记录变更、undolog回滚

Spring中，使用@Transaction标记事务

#### **原子性（Atomic）**

要么全部成功，要么全部失败，没有中间状态

#### **一致性（Consistency）**

指的是在执行事务前后，事务外访问数据的时候，数据是一致的，要么看到的是成功的，要么看到的是失败的结果，不会多任务查询到的数据不一样。

#### **隔离性（Isolation）**

一个事务在未完成时，另一个事务不会影响到他

#### **持久性（Durability）**

会有持久化效果，改变是永久的

随着系统越来越庞大，我们为了提高可用性、维护性、吞吐量等等技术指标，可以采用SOA来改善原有架构，业务计算的问题解决后，数据库便成了整个系统中的瓶颈

### 分库分表

数据库集群/分布式存储方案在当前最主流的便是采用分库分表方案做多机存储和负载

分库分表能减少单表

在多机存储的环境下，传统的事务机制便无法正常运行了。

#### **表垂直拆分**

把表中的列拆分成多个部分，就是把一张很多字段的表，拆分成多个表

减少查询时候的网络i/o

提高查询效率

#### **表水平拆分**

按行拆分，一张表里的行数越多，查询效率越低

水平拆分指定就是把原来的一张表中存放的数据按照固定行数拆分成多个表来存储

有效的提高查询效率

分表后依然可以使用本地事务，但是单机负载依然是瓶颈。

#### **分库**

分库指的是

同样的表结构

不同的数据库中

每个节点中保存副本数据或分布式存储

可以把事务路由到同一库中，则可以保证事务特性，尤其是强一致性

当数据分散到数据集群中做跨库查询的时候，无法保证强一致性

#### **微服务下的多库存储**

#### **回想CAP定理和BASE理论**

base通过允许损失部分可用性来提高数据最终一致性

## **海量数据/高并发系统分布式事务解决方案**

### 强一致性

强一致性会带来系统大量的损耗，存在单点故障问题，在提交事务阶段会产生阻塞，直到结束才会释放资源。高并发场景下表现并不好。

#### **二阶段提交协议**

二阶段提交协议可以保证数据强一致性

二阶段

准备

提交

角色:

协调者

负责收集准备信息（预提交）

准备完成，发起正式提交

参与者

数据库

二阶段提交把事务分为两个阶段，即准备阶段和提交阶段，准备阶段负责收集每个参与者提交数据的预提交信息，其实是收集每个数据节点是否能够成功执行命令，

如果其中某些节点无法执行，那么会反馈给协调者失败信息，然后协调者会发送回滚信息到每个数据节点以保证一致性

如果全部返回准备好的消息，那么协调者会正式向数据节点提交数据变更。

##### **问题**

**同步执行**

每一次收集准备信息都需要等所有数据节点返回信息，造成同步等待。

**单点故障**

需要考虑数据节点和协调者的单点故障问题

**执行提交**

数据节点提交变更，无法保证是否真正提交了

二阶段提交存在很多潜在和小概率问题，那么在数据一致性要求高的场景下会造成数据不一致的现象发生，一旦发生数据不一致的情况后，后续人工干预的操作会浪费大量的时间，对银行、余额这类业务，还有可能造成资产流失。

#### **三阶段提交协议**

三阶段提交是对二阶段提交的改良版本，增加了预备阶段，解决了同步阻塞问题。

#### **TCC模式**

**TCC**

Try

尝试

完成所有业务检查

隔离预留资源

Confirm

真正的执行业务

使用try阶段预留的资源

幂等

Cancel

混滚

主业务

发起事务

事务管理器

执行tcc操作

创建事务ID来记录整个事务链路

实现嵌套事务业务逻辑

执行流程

第一阶段

主业务调用所有子业务的try操作

事务管理器记录操作日志

第二阶段

所有try操作成功时，事务管理器执行confirm操作

有失败时，执行回滚cancel操作

t -> cc **关系绑定**

T是由主业务发起的，CC操作和T操作不在一台机器上

通过配置文件

通过Spring注解

try阶段进行了锁定资源操作，锁定不成功就认为是不可进行的，锁定成功就最终一定会成功。

##### **框架**

**tcc-transaction**

<https://github.com/changmingxie/tcc-transaction>

**Hmily**

<https://dromara.org/website/zh-cn>/

**ByteTCC**

<https://github.com/liuyangming/ByteTCC>

**AOP**

TCC 通过AOP，面向切面编程来对confirm/cancel操作透明化

**幂等**

数据库中的唯一索引

分布式锁

状态机

#### **补偿模式**

重试

固定次数，固定时间，消息队列，定时任务

下一次访问修复

定时校对/核对 quartz、xxl-job、elastic-job

事务 一定是由多个操作组成的吗？

# 亿级流量系统多级缓存架构 -分布式事务 2

## **刚性事物和柔性事物**

刚性事务：遵循ACID原则，强一致性。

柔性事务：遵循BASE理论，最终一致性

## **分布式事务解决方案**

### XA两阶段提交方案

数据库实现xa协议，保证事务，mysql oracle等。

#### **一阶段提交**

由单数据源数据库实现，以mysql为例，

记录redolog

数据库通过redolog完成事务

单节点提交简单直观，没有和其他节点的通讯，性能好

#### **二阶段提交**

主要由商业数据库实现，但并不常用，强一致性，性能低

增加中间角色，事务管理器，来收集各数据库事务状态

阻塞

事务管理器 是由数据库提供的中间件

### 接口查询

### TCC

在互联网项目，电商系统，app领域占据主力地位

接口需要自己实现或使用第三方开源框架

二阶段提交改进

对代码有侵入

Try

confirm

cancel

有侵入性、需要做幂等操作

### 商业产品

**GTS**

全局事务服务（Global Transaction Service ，简称GTS）用于实现分布式环境下特别是微服务架构下的高性能事务一致性。可以与RDS、MySQL、PostgreSQL、DRDS等数据源，Spring Cloud、Dubbo、EDAS及其他RPC框架，MQ消息队列等中间件产品配合使用，轻松实现分布式数据库事务、多库事务、消息事务、服务链路级事务及各种组合。

阿里云服务中间件

<https://www.aliyun.com/aliware/txc>

<https://yq.aliyun.com/articles/334238>

作者

<https://weibo.com/jiangyu666>

### LCN

<http://www.txlcn.org/zh-cn>/

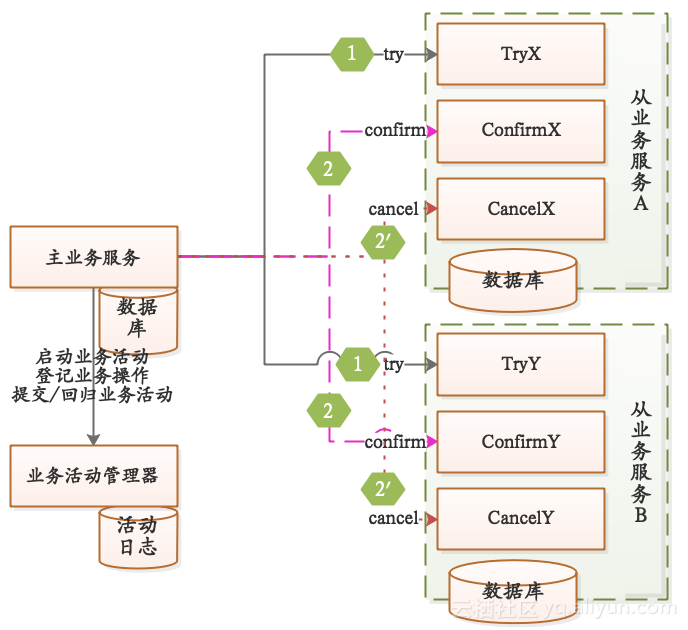
**Lock**、**confirm**、**notify**

兼容dubbo、springcloud框架，支持RPC框架拓展，支持各种ORM框架、NoSQL、负载均衡、事务补偿

#### **原理**

<https://txlcn.org/zh-cn/docs/principle/control.html>

### tcc-transaction



### TXC

无侵入式

# 亿级流量系统多级缓存架构10 阿里云实战

## **域名购买解析**

## **域名的价值**

域名的价值就像商标一样，体现在你注册一个域名之后，别人就不可以再注册这个域名了，域名的稀缺性就产生了。域名可以买卖，一个域名我注册了，有人也想要的话，我可以选择卖给他。域名没有标准的价格，一般域名越短越好记越有意义，价格越高，但这也不是特别绝对的。

十几年前，互联网还不是很发达，能提供域名注册的商家不多，每年的费用也很高。当然，现在是便宜多了， .com一年大概六七十。

*域名可以买卖，当然可以用来赚钱。域名赚钱也不难，买一些域名，有的域名别人没有注册过是可以抢注的。最值钱的是 .com的域名。当我要卖这个域名的时候，假如一年60块钱注册的，注册了5年，那成本就是300，卖的话肯定不能低于300块钱。有闲钱的话可以去投资一些，但现在投资域名也不太赚钱了。因为好的域名大家都知道，想从别人那买它不会便宜，想抢注刚释放的域名很难，因为很多专业的机构也在抢。*

## **域名解析原理**

#### **1、域名是什么**

域名由两部分组成 ：名（name）和域（domain）。比如 [lanrenzaixian.com](http://lanrenzaixian.com) 。lanrenzaixain 是名，.com是域。

和windows server 域管理器很像，公司里电脑太多的话，作为一个网管想要管理几千台电脑，要把电脑划分到不同的区域里，domain就是一个一个的区域，相当于一个分组。名称就是自己电脑的主机名。电脑想加入一个区域里，要有加入这个域的权限，就要给域的管理商付费。

每个区域里的名称是唯一的并且是独立的。不同的区域里可以有相同的名称出现。lanrenzaixian 在 .com里出现了，就不能再有和它同名的了，但是在别的域里可以有，比如 lanrenzaixain .net。

当我们注册完一个域名时，相当于我们在这个域里注册了一个名称。我就可以通过名称加域的地址定位到我的主机上。在我的域名的上一级还可以再加一个名称，比如www.lanrenzaixian .com 这就是我的子域名。这个子域名就是二级域名，二级域名上还可以加一级，就是三级域名，每一级域名都有管理它上一级域名的权限。

#### **2、域名访问网站**

域名的主要功能就是访问网站。

把域名变成访问网站的入口，或者传递数据到另一台主机，要走tcp/ip协议，这意味着我们只能通过ip地址去访问另一台服务器。现在用的是ipv4地址，如 60.66.58.21。**在公网上每台主机的ip都是唯一的。**用域名解析到一台主机，当我们访问一个域名时可以自动变成一个ip，这时就可以访问这台主机了。

#### **3、域名是如何变成ip的**

**在互联网上访问一台主机必须要通过IP地址，只有域名是不行的，还需要一个把域名转换成ip地址的过程。**

找到域名对应的ip，一部分的工作是在客户端完成的，访问网页时我们的客户端通常是浏览器。浏览器接收到你访问某个网址的请求后：

1、浏览器先在本机的hosts文件中找到对应的映射关系，主机到ip地址的映射关系。

hosts文件的主要作用是定义IP地址和主机名的映射关系。当用户在浏览器中输入一个网址时，系统会首先自动从hosts文件中寻找对应的IP地址，一旦找到，浏览器会立即打开对应网页，如果没有找到，则浏览器会将网址提交DNS服务器进行IP地址解析。

2、如果hosts文件中没有，在本机的dns缓存中查找。

找这个域名对应的ip地址。可以在命令行中用ping命令，如ping qq.com ，返回对应的ip。

3、dns缓存中没有，在tcp/ip协议里设置的dns服务器查找。



这个IPv4的dns服务器是可以自己配置的，或者通过路由器的dhcp功能下发的。

192.168.31.1是内网ip，很明显指向的是我的路由器。路由器中也有dns缓存。

4、如果还是没找到这个域名对应的ip。路由里面一定会配外网dns服务器。

外网dns服务器：当我们域名配好ip后，会提交到一台dns服务器上，保存域名对ip地址的解析。这台dns服务器就有了你的域名和ip的映射关系，并且会广播到外网dns服务器上。

谷歌的外网dns服务器：8.8.8.8。在外网服务器找到后，会返回给我们的主机。

5、如果外网dns服务器没有。

接收请求后，这台服务器会通过域名找到，保存了我的域名解析的dns服务器，因为外网dns服务器并发量很高，它会把请求做**批处理**。每批处理一次，就把能拿到的数据做本地缓存。当再访问这台服务器是就有了这个域名对应的ip了。

流程：hosts文件 --> 本机dns缓存 --> 路由器dns缓存 --> 外网dns服务器缓存 --> 外网dns服务器去找保存了那个域名解析的dns服务器。

## **域名管理**

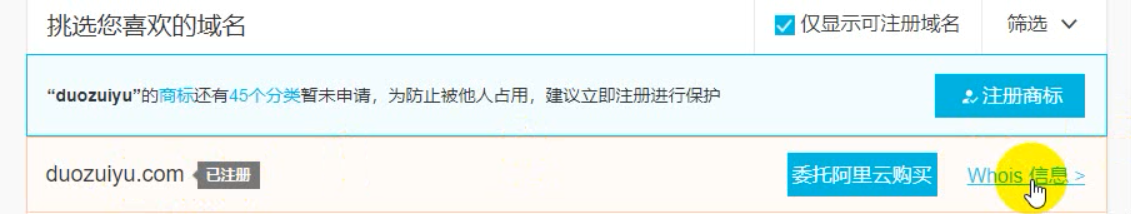


在域名的后台管理中心，我们可以：

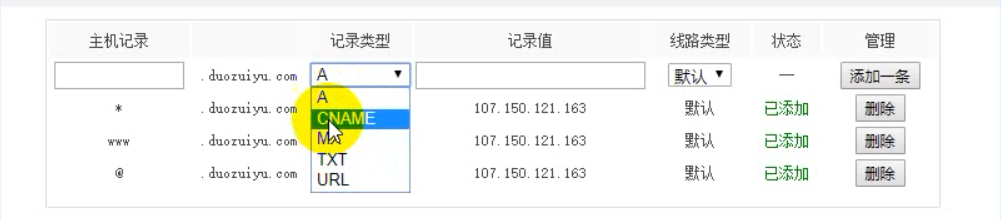
1、修改DNS服务器，每个平台都会提供几个可用的dns服务器，但不一定必须用这个，也可以自己搭一个dns服务器。一个域名也可以有多个dns服务器。

2、查看域名状态。域名的状态表示你域名现在是什么情况。active ：激活状态、delete ：被删除状态。状态有很多种，可以百度查一查。

3、域名联系人信息，信息是注册域名的时候填写的。这些信息是可以公开也可以不公开的。通过Whois能看到域名注册人的信息。



域名解析。域名解析的类型有A、CNAME等

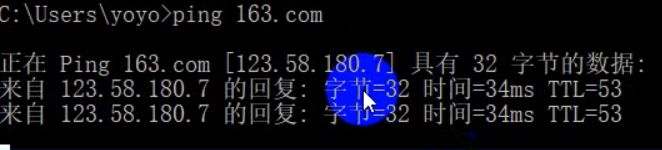


A记录：把某个域名直接指向一个ip地址，比如把qq .com 指向59.37.96.63。

CNAME记录：对应另一个已经解析好的域名，假如qq .com 是已经解析好的域名，CNAME是把qq.lanrenzaixain .com 指向 qq .com。

4、线路类型：线路类型是dns服务器给我们提供的一个功能。在很久以前，中国的网络有两大家：电信、连通。很多游戏服务器都分电信服务器和网通服务器。如果你家网是电信的，你玩网通服务器就会卡，网通玩电信也会卡。所以你最好是选择对应的服务器玩，这样电信的用户玩电信，网通的用户玩网通，就不会延迟太高。但是一个网站的用户有电信的也有网通的，那网站应该用电信还是网通呢？用电信的网通用户慢，用网通的电信访问的就慢。这种问题有几个解决方案：

在一台主机上接入两条网络，一个电信一个联通。对应的外网ip就有两个，通过两个网络都能接到这个主机里。但是电脑不会直接访问ip，要通过域名转成ip。这时当我们访问一个网址后，会通过一个dns服务器，dns服务器会拿到你的ip地址，分析你是电信还是联通。是电信的就返回给你电信的ip，联通就返回联通的ip。所以每个人得到的ip可能会不一样，你可以命令行ping 163.com，我得到的ip是 123.58.180.7，你的不一定哦。dns服务器不仅可以分析你是电信还是联通，还能分析你是那个区域：北京，山东、哈尔滨等。根据dns服务器的逻辑判断，哪条线路对你最优，就给你哪个ip。



BGP技术。由网络提供商整合多线路。这个线路既不是电信也不是联通。是网络提供商同时租了这两个网络，用BGP技术整合出一条通用的线路。当服务器接入到这一条线路后，用户在访问网络时，就会在网络内部自动适应。

## **云服务器部署**

## **https部署**

## **CDN全网加速**

# 亿级流量系统多级缓存架构11 流量网关

## **流量网关和业务网关**

### 流量网关

全局性流控

日志统计

防止 SQL 注入

防止 Web 攻击

屏蔽工具扫描

黑白名单控制

### ****业务网关****

请求接入：作为所有 API 接口服务请求的接入点，管理所有的接入请求；

业务聚合：作为所有后端业务服务的聚合点，所有的业务服务都可以在这里被调用；

中介策略：实现安全、验证、路由、过滤、流控，缓存等策略，进行一些必要的中介处理；

统一管理：提供配置管理工具，对所有 API 服务的调用生命周期和相应的中介策略进行统一管理。

## **开源网关**

目前常见的开源网关大致上按照语言分类有如下几类：

Nginx+lua：Open Resty、Kong、Orange、Abtesting gateway 等

Java：Zuul/Zuul2、Spring Cloud Gateway、Kaazing KWG、gravitee、Dromara soul 等

Go：Janus、fagongzi、Grpc-gateway

Dotnet：Ocelot

NodeJS：Express Gateway、Micro Gateway

按照使用数量、成熟度等来划分，主流的有 4 个：

OpenResty

Kong

Zuul/Zuul2

Spring Cloud Gateway

### ABTesingGateway

项目地址：

<https://github.com/CNSRE/ABTestingGateway>

### ****Zuul/Zuul2****

<https://github.com/Netflix/zuul>

Zuul 1.x

Zuul 2.x

### Spring Cloud Gateway

<https://github.com/spring-cloud/spring-cloud-gateway>/

## ****OpenResty/Kong/Zuul2/SpringCloudGateway 重要特性对比****

| **网关** | **限流** | **鉴权** | **监控** | **易用性** | **可维护性** | **成熟度** |
| --- | --- | --- | --- | --- | --- | --- |
| Spring Cloud Gateway | 可以通过IP，用户，集群限流，提供了相应的接口进行扩展 | 普通鉴权、auth2.0 | Gateway Metrics Filter | 简单易用 | spring系列可扩展强，易配置 可维护性好 | spring社区成熟，但gateway资源较少 |
| Zuul2 | 可以通过配置文件配置集群限流和单服务器限流亦可通过filter实现限流扩展 | filter中实现 | filter中实现 | 参考资料较少 | 可维护性较差 | 开源不久，资料少 |
| OpenResty | 需要lua开发 | 需要lua开发 | 需要开发 | 简单易用，但是需要进行的lua开发很多 | 可维护性较差，将来需要维护大量lua脚本 | 很成熟资料很多 |
| Kong | 根据秒，分，时，天，月，年，根据用户进行限流。可在原码的基础上进行开发 | 普通鉴权，Key Auth鉴权，HMAC，auth2.0 | 可上报datadog，记录请求数量，请求数据量，应答数据量，接收于发送的时间间隔，状态码数量，kong内运行时间 | 简单易用，api转发通过管理员接口配置，开发需要lua脚本 | "可维护性较差，将来需要维护大量lua库 | 相对成熟，用户问题汇总，社区，插件开源 |

## **Kong**

<https://konghq.com>/

<https://github.com/kong/kong>

**主要有三个组件**

Kong Server ：基于nginx的服务器，用来接收API请求。

Apache Cassandra/PostgreSQL ：用来存储操作数据。

Kong dashboard：官方推荐UI管理工具，当然，也可以使用 restfull 方式 管理admin api。

Kong采用插件机制进行功能定制，插件集（可以是0或N个）在API请求响应循环的生命周期中被执行。插件使用Lua编写，目前已有几个基础功能：HTTP基本认证、密钥认证、CORS（Cross-Origin Resource Sharing，跨域资源共享）、TCP、UDP、文件日志、API请求限流、请求转发以及Nginx监控。

### 安装

安装环境：centOS6.5

#获取下载的rpm

yum install https://download.postgresql.org/pub/repos/yum/9.5/redhat/rhel-7-x86\_64/pgdg-centos95-9.5-3.noarch.rpm

#执行安装

yum install postgresql95-server

初始化数据库

service postgresql-9.5 initdb

启动服务

service postgresql-9.5 start

### 远程连接

修改

vi /var/lib/pgsql/9.5/data/pg\_hba.conf

# TYPE DATABASE       USER           ADDRESS                 METHOD

​

# "local" is for Unix domain socket connections only

local   all             all                                     peer

​

​

# IPv4 local connections:

host   all             all             127.0.0.1/32           trust

# IPv6 local connections:

host   all             all             ::1/128                 ident

host   all             all             0.0.0.0/0               trust

# Allow replication connections from localhost, by a user with the

# replication privilege.

#local   replication     postgres                               peer

#host   replication     postgres       127.0.0.1/32           ident

#host   replication     postgres       ::1/128                 ident

​

vi /var/lib/pgsql/9.5/data/postgresql.conf

listen\_addresses = '\*'         # what IP address(es) to listen on;

                                      # comma-separated list of addresses;

                                      # defaults to 'localhost'; use '\*' for all

                                      # (change requires restart)

port = 5432                             # (change requires restart)

### ****创建数据库****

su - postgres//进入数据库

-bash-4.2$ psql

CREATE USER kong WITH PASSWORD 'kong';

create database kong owner kong;

grant all on database kong to kong;

### 1.3下载地址

https://kong.bintray.com/kong-community-edition-rpm/centos/6/

### 安装

rpm -ivh kong-community-edition-0.13.1.el6.noarch.rpm

### 拷贝配置文件

/etc/kong/kong.conf.default

/etc/kong/kong.conf

### 初始化数据库

1.4 kong migrations bootstrap

1.3 kong migrations up

### 启动服务

kong start

### 安装dashboard

<https://github.com/PGBI/kong-dashboard>

最新的Kong是0.14.x，而Kong-Dashboard暂时支持最高版本0.13.x

安装npm

curl --silent --location https://rpm.nodesource.com/setup\_7.x | bash -

sudo yum install -y nodejs

安装dashboard

npm install -g kong-dashboard

启动

kong-dashboard start --kong-url http://localhost:8001

RPM常用命令

**1、安装**

rpm -i 需要安装的包文件名

举例如下：

rpm -i example.rpm 安装 example.rpm 包；

rpm -iv example.rpm 安装 example.rpm 包并在安装过程中显示正在安装的文件信息；

rpm -ivh example.rpm 安装 example.rpm 包并在安装过程中显示正在安装的文件信息及安装进度；

**2、查看安装完成的软件**

[root@jacky zookeeper]# rpm -qa | grep jdkjava-1.6.0-openjdk-1.6.0.0-1.66.1.13.0.el6.i686java-1.7.0-openjdk-1.7.0.45-2.4.3.3.el6.i686

**3、卸载软件**

rpm -e --nodeps 要卸载的软件包

root@jacky zookeeper]# rpm -e --nodeps java-1.6.0-openjdk-1.6.0.0-1.66.1.13.0.el6.i686

# 亿级流量系统多级缓存架构12 电商详情页开发实战

1.如果去面试，公司的实力和自己的能力都不强，让你做架构师，带团队你会去做吗？理由是啥？

2.描述一下互联网创业公司的产品从无到有，应该从哪里入手，包含哪些岗位分工？

## **静态化文件生成方案**

### 管理后台

发布新商品/添加商品流程

事务中包含哪些操作

写db，强一致性

写缓存

生成文件

生成文件

文件结构化数据，分片，商品模板

直接生成

异步生成

更新商品

## **缓存**

## **先写DB还是先写缓存？**

### Redis 大Key/value问题

#### **什么是 大key/value？**

字符串：大于10k

hash、list、set、zset等 < 5000个

#### **问题**

读取数据的时候网络阻塞，单线程处理网络请求，频繁出现等待，线程hang住，甚至超时。

LRU算法问题，当新增value过大时，导致占用空间过大，后续新增的value会把之前的顶掉，缓存命中率降低。

del大value的时候，或缓存过期后调起的的del操作，会造成线程阻塞

#### **解决方案**

压缩数据

数据分组，客户端聚合

* 手动拆分成几个k/v存储，mget取值，客户端手动合并，部分更新
* hset,hmset更新，存储hash结构在redis中，hgetall取出，key节省空间

## **静态数据补偿**

# 亿级流量系统多级缓存架构12 电商详情页开发实战

## **rsync**

<https://www.samba.org/ftp/rsync/rsync.html>

remote synchronize是一个远程数据同步工具，可通过 LAN/WAN 快速同步多台主机之间的文件。也可以使用 rsync 同步本地硬盘中的不同目录。rsync 是用于替代 rcp 的一个工具，rsync 使用所谓的 rsync算法 进行数据同步，这种算法只传送两个文件的不同部分，而不是每次都整份传送，因此速度相当快。

rsync 基于inotify 开发

## **安装**

两端安装

yum install -y rsync

### 服务端配置

#### **密码文件**

/etc/rsync.password

hello:123

修改权限

chmod 600 /etc/rsync.password

/etc/rsyncd.conf

#/etc/rsyncd.conf

uid = root

gid = root

use chroot = no

max connections = 200

hosts allow = \*

timeout = 600

log file = /var/log/rsyncd.log

pid file = /var/run/rsyncd.pid

lock file = /var/run/rsync.lock

ignore errors

read only = false

list = false

host deny = 0.0.0.0/32

auth users = hello

secrets file = /etc/rsync.password

transfer logging = yes

#Module definitions

​

[www]

path = /usr/local/openresty/nginx/html/

#### **监控目录**

mkdir -p /data/www

#### **启动服务**

rsync --daemon

### 同步数据

rsync -zvaP --password-file=/etc/rsync.password hello@192.168.150.113::www /usr/local/openresty/nginx/html/

### 实时同步

rsync + inotify-tools 或 rsync+sersync

### 客户端配置

创建密码文件

vi /etc/rsync.password

123

## **Nginx SSI**

### nginx开启配置

ssi on;

ssi\_silent\_errors on;

ssi\_types text/html;

### 文件引入

<!--# include file="file.html" -->

## **TMPFS**

mount -t tmpfs -o size=500m tmpfs /data/tmp/

## **模板生成**

### Jfinal 介绍

国产开源框架

<https://www.jfinal.com>/

### HelloWorld

**POM坐标**

<dependency>

   <groupId>com.jfinal</groupId>

   <artifactId>jfinal-undertow</artifactId>

   <version>1.9</version>

</dependency>

<dependency>

   <groupId>com.jfinal</groupId>

   <artifactId>jfinal</artifactId>

   <version>4.7</version>

</dependency>

**APP.java**

package com.mashibing.jfinal.demo;

​

import com.jfinal.server.undertow.UndertowServer;

​

public class APP {

​

public static void main(String[] args) {

UndertowServer.start(MyConfig.class,80,true);

}

}

helloController.java

package com.mashibing.jfinal.demo;

​

import com.jfinal.core.Controller;

​

public class HelloController extends Controller {

​

public void index() {

renderText("hi222213!");

redirect("http://mashibing.com");

}

}

Config

package com.mashibing.jfinal.demo;

​

import com.jfinal.config.Constants;

import com.jfinal.config.Handlers;

import com.jfinal.config.Interceptors;

import com.jfinal.config.JFinalConfig;

import com.jfinal.config.Plugins;

import com.jfinal.config.Routes;

import com.jfinal.template.Engine;

​

public class MyConfig extends JFinalConfig {

​

@Override

public void configConstant(Constants me) {

// TODO Auto-generated method stub

}

​

@Override

public void configRoute(Routes me) {

​

me.add("/hello",HelloController.class);

}

​

@Override

public void configEngine(Engine me) {

// TODO Auto-generated method stub

}

​

@Override

public void configPlugin(Plugins me) {

// TODO Auto-generated method stub

}

​

@Override

public void configInterceptor(Interceptors me) {

// TODO Auto-generated method stub

}

​

@Override

public void configHandler(Handlers me) {

// TODO Auto-generated method stub

}

​

}

### Jfinal Enjoy

#### **maven坐标**

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.9.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.mashibing.jfinal02</groupId>

<artifactId>jfinal02</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>jfinal02</name>

<description>jfinal02</description>

​

<properties>

<java.version>1.8</java.version>

</properties>

​

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

​

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

 <groupId>com.jfinal</groupId>

 <artifactId>enjoy</artifactId>

 <version>4.7</version>

</dependency>

</dependencies>

​

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

​

</project>

#### **模板**

1 #(name) ,age = #(age)

#### **入口**

package com.mashibing.jfinal02;

​

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

​

@SpringBootApplication

public class Jfinal02Application {

​

public static void main(String[] args) {

SpringApplication.run(Jfinal02Application.class, args);

MyApp myApp = new MyApp();

myApp.hi();

}

}

#### **主程序**

package com.mashibing.jfinal02;

​

import java.util.HashMap;

​

import com.jfinal.kit.Kv;

import com.jfinal.template.Engine;

import com.jfinal.template.Template;

​

public class MyApp {

​

public void hi() {

Engine engine = Engine.use();

engine.setDevMode(true);

engine.setToClassPathSourceFactory();

Kv kv = Kv.by("name", "lucy");

kv.put("age", 18);

Template template = engine.getTemplate("index.tpl");

String str = template.renderToString(kv);

template.render(kv, "index.html");

System.out.println(str);

System.out.println("!!!");

}

}

### FreeMarker

javacc

#### **maven坐标**

<?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 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.2.1.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.mashibing.jfinal02</groupId>

<artifactId>freemk</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>freemk</name>

<description>jfinal02</description>

​

<properties>

<java.version>1.8</java.version>

</properties>

​

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-freemarker</artifactId>

</dependency>

​

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.junit.vintage</groupId>

<artifactId>junit-vintage-engine</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

​

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

​

</project>

​

#### **模板**

<h1>Girls</h1>

name = ${girl.name},

<br>

price = ${girl.age}

#### **实体类**

package com.mashibing.freemk;

​

public class Girl {

​

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

#### **Controller**

@Autowired

private Configuration conf;

@RequestMapping("aa")

public String aa() throws TemplateNotFoundException, MalformedTemplateNameException, ParseException, IOException, TemplateException {

int id = 101;

String filePath = "c:/dev";

File htmlFile = new File(filePath + "/" + id + ".html");

if(!htmlFile.exists()) {

//如果文件不存在

Template template = conf.getTemplate("tp.ftl");

htmlFile.createNewFile();

//写入数据

BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(htmlFile), "utf-8"));

HashMap<Object, Object> map = new HashMap<>();

Girl girl = new Girl();

girl.setName("lucy");

girl.setAge(18);

map.put("girl", girl);

template.process(map, writer);

}

return "aa";

}

# 亿级流量系统多级缓存架构14 静态文件生成方案

外部路径加载 模板

/usr/local/openresty/nginx/html

java -jar Arica-0.0.1-SNAPSHOT.jar --nginx.html.root=/usr/loca

l/openresty/nginx/html --spring.datasource.url=jdbc:mysql://192.168.110.140:3306/ssm?characterEncoding=utf8&useSSL=true&serverTimezone=UTC&allowMultiQueries=true

# 亿级流量系统多级缓存架构14 电商详情页开发实战

# Jrebel 最新版热部署问题

JRebel启动报Couldn't write to C:\Users

一般是由Windows登录用户名是中文引起的

### 解决方案1

重新创建一个纯英文不带空格的用户

### 解决方案2

麻烦..

更改当前用户主目录

C:\Users下直接修改用户主目录会提示失败

#### **第一步**

开启管理员账户

#### **第二步**

注销当前用户

使用administrator登录系统

#### **第三步**

修改文件夹名

把c:/users下的中文主目录改成纯英文，不带空格的那种

#### **第四步**

修改注册表

win + r 之后输入 **regedit**

找到 HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Profilelist

双击修改成纯英文

#### **第五步**

注销登录，使用原来的用户名重新登录系统

#### **搞定**

修改主目录后，可能会有权限问题

在主目录上，右击属性->安全->选择everyone

权限全选

## **开发实战**

### mybatis返回主键

<insert id="insert" parameterType="com.mashibing.Arica.entity.Item" useGeneratedKeys="true" keyProperty="id" keyColumn="id">

  insert into item (id, title, content,

    last\_generate)

  values (#{id,jdbcType=INTEGER}, #{title,jdbcType=VARCHAR}, #{content,jdbcType=VARCHAR},

    #{lastGenerate,jdbcType=TIMESTAMP})

</insert>

# 亿级流量系统多级缓存架构15 静态文件生成方案

## **模板文件外部路径 √**

## **项目部署 √**

## **多Nginx同步数据 √**

### 镜像服务器启动服务

在213上 启动rsync服务

检查密码文件

hello:123

rsync --daemon

### 在数据源创建密码文件

vi /etc/rsync.password

chmod 600 /etc/rsync.password

123

sersync 的方式监控目录中变化的数据，通过rsync客户端推送给rsync daemon

### 数据源的服务配置

yum install rsync -y

vi /etc/rsync.password

`123`

chmod 600 /etc/rsync.password

​

下载sersync

​

wget https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/sersync/sersync2.5.4\_64bit\_binary\_stable\_final.tar.gz

​

tar xf sersync2.5.4\_64bit\_binary\_stable\_final.tar.gz

​

cd GNU-Linux-x86/

​

mkdir -p /opt/sersync/{logs,conf,bin}

cp sersync2 /opt/sersync/bin

cp confxml.xml /opt/sersync/conf/

cd /opt/sersync/conf/

vi confxml.xml

../bin/sersync2 -d -r -o ./confxml.xml

​

cd /data

touch {1,2,3,4,5,6}

## **批量生成文件 √**

## **首页生成 √**

## **健康检查 √**

## **静态文件缓存架构**

## **搜索**

## **分页**

# 亿级流量系统多级缓存架构16 静态文件生成方案

## **发布Item并生成，事务单机 √**

## **并发编辑 锁，提示 只读 √**

## **文件检查**

## **分层 后台 生成的文件**

## **Nginx 合并输出 SSI**

## **动静性能对比 压力测试**

Tomcat + 动态输出 ~= 900

Tomcat + 静态html ~= 2000

## **静态化方案对比**

## **静态化内容**

不包含有状态信息

浏览者因素

时间因素

地域因素

异步加载状态信息

## **Html失效**

主动失效

被动失效

# 亿级流量系统多级缓存架构17 静态文件生成方案

## **本套架构方案适用场景**

适用于常见的互联网项目，应对高并发、高吞吐量的应用场景

电商详情页

CMS、News类

广告展示系统

多用户高并发的评论系统、bbs系统

优点：

* 系统结构简单
* 架构复杂度低，易实现
* 整体吞吐量没有瓶颈
* 系统负载易扩展
* 低延迟

缺点

* 数据量大的时候单机容量会成为瓶颈
* 缓存文件批量失效造成高I/O
* 负载策略单一，容易出现热点
* 数据一致性低

## **文件分层**

## **SSI**

## **搜索/分页 静态化**

## **FastDfs & CDN**

# 亿级流量系统多级缓存架构18 静态文件生成方案

## **使用Thymeleaf**

## **lua-resty-mysql**

<https://github.com/openresty/lua-resty-mysql>

# 亿级流量系统多级缓存架构19 静态文件生成方案

引入

local template = require "resty.template"

配置模板默认路径

  server {

set $template\_root /usr/local/openresty/nginx/html/templates;

关闭缓存

template.caching(false)

lua\_code\_cache off;

## **测试**

lua

local template = require "resty.template"

​

local view = template.new "view.html"

view.message = "Hello, World!"

view:render()

​

view.html

<!DOCTYPE html>

<html>

<body>

<h1>{{message}}</h1>

</body>

</html>

## **引入头尾文件**

view.html

{(header.html)}

<h1>{{message}}</h1>

{(footer.html)}

header.html

<!DOCTYPE html>

<html>

<head>

<title>{{title}}</title>

</head>

<body>

footer.html

<h1>copyright！</h1>

</body>

</html>

lua

local template = require "resty.template"

local view = template.new "view.html"

view.message = "Hello, World!"

view.title = "xxx"

view:render()

## **其他渲染方式**

local template = require "resty.template"

local context = {title = "title"}

template.render("view.html", context)

local func = template.compile("view.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)

## **lsp**

    location ~ \.lsp$ {

      default\_type text/html;

      content\_by\_lua '

local template = require "resty.template"

template.render(ngx.var.uri)

​';

  }