

第2章 亿级流量红包雨缓存实战

目标1: 掌握清理Nginx缓存

目标2: 掌握Lua流程控制语法

目标3: 能实现Nginx+Lua多级缓存控制

目标4: 掌握红包雨的缓存架构设计

目标5: 掌握队列溢出控制设计方案

目标6: 掌握队列削峰填谷的设计思想

目标7: Nginx限流

1 Nginx缓存清理

很多时候我们如果不想等待缓存的过期, 想要主动清除缓存, 可以采用第三方的缓存清除模块清除缓存 `nginx ngx_cache_purge`。安装nginx的时候, 需要添加 `purge` 模块, `purge` 模块我们已经下载了, 在 `/usr/local/server` 目录下, 添加该模块 `--add-module=/usr/local/server/ngx_cache_purge-2.3/`, 这一个步骤我们在安装 `OpenResty` 的时候已经实现了。

安装好了后, 我们配置一个清理缓存的地址:

```
#清理缓存
location ~ /purge(/.*) {
    #清理缓存
    proxy_cache_purge openresty_cache $host$1$is_args$args;
}
```

这里 `$1` 代表的是 `(/.*)` 的数据。

每次请求 `$host$key` 就可以删除指定缓存, 我们可以先查看缓存文件的可以:

```
[root@localhost 69]# cat 4c759a88208fe148da0155637f64a69b
3_yyyyyyyy) _^2)
KEY: 192.168.211.141/user/wangwu
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Date: Wed, 08 Jul 2020 08:34:49 GMT
Connection: close

{"username":"wangwu","sex":"男","name":"王五","level":1,"age":29}[root@localhost 69]# XshellXshell
-bash: XshellXshell: command not found
[root@localhost 69]#
```

此时访问 `<http://192.168.211.141/purge/user/wangwu>`



此时再查看缓存文件，已经删除了。

参数说明：

`$args`

参数: `$args`

解释: HTTP请求中的完整参数。

访问: `http://test.itheima.com/192.168.1.200?a=10`

返回: `"a=10"`

`$uri`

参数: `$uri`

解释: 表示当前请求的URI,不带任何参数

访问: `http://test.itheima.com/user/wangwu?a=10`

返回: `"/user/wangwu"`

`$host`

参数: `$host`

解释: 表示客户端请求头部中的Host字段。如果Host字段不存在,则以实际处理的server name名称代替。如果Host字段中带有端口,如IP:PORT,那么\$host会去掉端口

访问: `http://test.itheima.com/192.168.1.200?a=10`

返回: `"test.itheima.com"`

`$is_args`

参数: `$is_args`

解释: 表示请求中的URL是否带参数,如果带参数,\$is_args值为"?"。如果不带参数,则是空字符串

访问: `http://test.wanglei.com/192.168.1.200?a=10`

返回: `"?"`

访问: `http://test.wanglei.com/192.168.1.200`

返回: `""`

2 Lua高性能脚本语言

2.1 Lua介绍

Lua是一门以其性能著称的脚本语言,被广泛应用在很多方面。Lua一般用于嵌入式应用,现在越来越多应用于游戏当中,魔兽世界,愤怒的小鸟都有用到。

优势:

- 1、Lua极易嵌入到其他程序，可当做一种配置语言。
- 2、提升应用性能，比如：游戏脚本，nginx，wireshark的脚本
- 3、兼容性强，可以直接使用 C 代码写的函数。比如corona 移动应用开发平台，跟PhonePap类似，不过使用Lua做开发语言，应用可以build到ios，Android，kindle fire，nook平台；服务器端应该也是Lua。
- 4、在给软件提供嵌入式脚本编程能力上，Lua是绝佳选择。

Lua脚本的作用：嵌入到应用程序中，给应用程序提供扩展功能。

2.2 Lua安装

依赖安装：`yum install libtermcap-devel ncurses-devel libevent-devel readline-devel pcre-devel gcc openssl openssl-devel`

下载安装包：`curl -R -O http://www.lua.org/ftp/lua-5.3.5.tar.gz`

解压安装：`tar xzf lua-5.3.5.tar.gz`

安装：

```
进入解压目录
cd lua-5.3.5

执行安装
make linux test
```

此时原来系统自带的lua其实是5.1.4，我们需要替换原来系统自带的lua：

```
rm -rf /usr/bin/lua
ln -s /usr/local/server/lua-5.3.5/src/lua /usr/bin/lua
```

查看版本

```
[root@localhost 69]# lua -v
Lua 5.3.5 Copyright (C) 1994-2018 Lua.org, PUC-Rio
[root@localhost 69]#
```

2.3 Lua常用知识

2.3.1 Lua案例

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：

```
[root@localhost lua]# vi hello.lua
[root@localhost lua]# lua hello.lua
hello
[root@localhost lua]#
```

lua有交互式编程(在控制台编写代码)和脚本式编程(在文件中写代码叫脚本编程)。

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：

```
[root@localhost lua]# lua
Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio
> print("hello itheima")
hello itheima
>
```

(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。

2.3.2 Lua语法

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：

```
> name="itheima"  
> local address="shenzhen"  
> print(name)  
itheima  
> print(address)  
nil  
>
```

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。

lua红有个type函数，能出对象的类型。

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
```

2.3.3 流程控制

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：

```
> --定义变量
> num=5
> --使用if语句判断
> if num<10
> then
> print("5的值比10小")
> end
5的值比10小
```

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:

```
> --[ 定义变量 --]
> a = 100;
> --[ 检查条件 --]
> if( a < 20 )
> then
>     --[ if 条件为 true 时执行该语句块 --]
>     print("a 小于 20" )
> else
>     --[ if 条件为 false 时执行该语句块 --]
>     print("a 大于 20" )
> end
a 大于 20
```

循环[练习]

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:

```

> a=10
> while( a < 20 )
> do
>>   print("a 的值为:", a)
>>   a = a+1
>> end
a 的值为:      10
a 的值为:      11
a 的值为:      12
a 的值为:      13
a 的值为:      14
a 的值为:      15
a 的值为:      16
a 的值为:      17
a 的值为:      18
a 的值为:      19

```

for循环[练习]

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
do
    print(i)
end

```

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
1
3
5
7
9

```

repeat...until语句[满足条件结束][练习]

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：

```

-- 定义变量
num=5
-- 循环
repeat
  print(num)
  num=num+1
until(num>15)
5
6
7
8
9
10
11
12
13
14
15

```

函数

lua中也可以定义函数，类似于java中的方法。例如：

```

--[[ 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

```

...表示拼接

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[0] = "Lua"

-- 移除引用
mytable = nil
```

模块

(1) 模块定义

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

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)

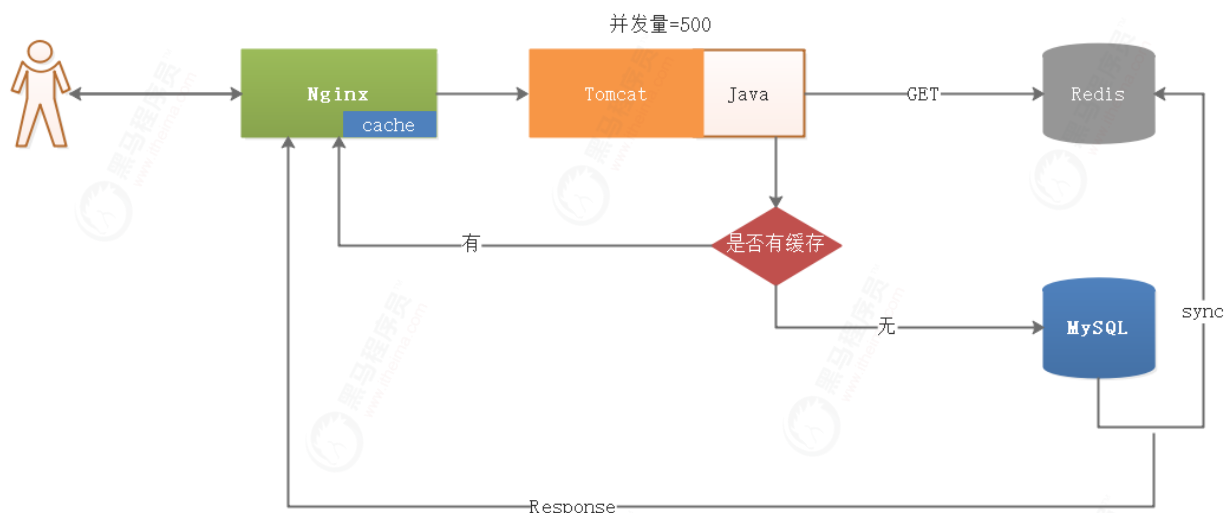
module.func3()
```

2 多级缓存架构

任何项目中我们都有一些频繁的查询,而那些频繁的查询数据基本都是相同的,比如项目中查看用户个人信息,购物狂欢查询活动信息,这些功能点使用频率非常高,我们一般需要对它们做特殊处理。

我们以狂欢活动信息为例,当双十一的时候,很多人会去查看今天优惠活动有哪些,对这些活动信息我们可以采用多级缓存架构来抗住高并发。

2.1 多级缓存架构对比



java版多级缓存架构

基于Java版的缓存架构实现流程如下：

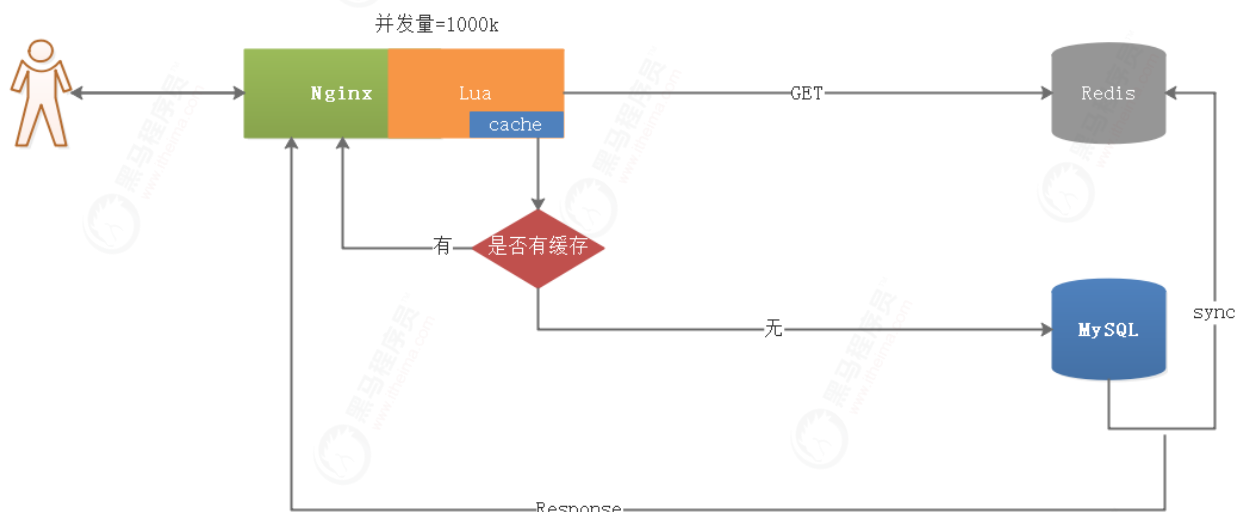
- 1、用户请求经过Nginx
- 2、Nginx检查是否有缓存，如果Nginx有缓存，直接响应用户数据
- 3、Nginx如果没有缓存，则将请求路由给后端Java服务
- 4、Java服务查询Redis缓存，如果有数据，则将数据直接响应给Nginx，并将数据存入缓存，Nginx将数据响应给用户
- 5、如果Redis没有缓存，则使用Java程序查询MySQL，并将数据存入到Redis，再将数据存入到Nginx中

优点：

- 1、采用了Nginx缓存，减少了数据加载的路径，从而提升站点数据加载效率
- 2、多级缓存有效防止了缓存击穿、缓存穿透问题

缺点：

Tomcat并发量偏低，导致缓存同步并发量失衡，缓存首次加载效率偏低，Tomcat 大规模集群占用资源高



Lua版多级缓存架构

优点：

- 1、采用了Nginx缓存，减少了数据加载的路径，从而提升站点数据加载效率
- 2、多级缓存有效防止了缓存击穿、缓存穿透问题
- 3、使用了Nginx+Lua集成，无论是哪次缓存加载，效率都高
- 4、Nginx并发量高，Nginx+Lua集成，大幅提升了并发能力

2.2 Nginx+Lua多级缓存实战

我们以双十一活动信息加载为例，通过多级缓存架构来加载双十一活动信息，双十一活动信息表结构如下：

```
CREATE TABLE `activitt_info` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(100) DEFAULT NULL COMMENT '活动名称',
  `desc` varchar(3000) DEFAULT NULL COMMENT '活动介绍',
  `starttime` datetime DEFAULT NULL,
  `endtime` datetime DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

2.2.1 链接MySQL封装

创建一个lua脚本 `mysql.lua`，用于提供根据SQL语句执行查询操作，代码如下：

```
--引入依赖库mysql
local mysql = require "resty.mysql"

--配置数据库链接信息
local props = {
  host = "192.168.211.141",
  port = 3306,
  database = "redpackage",
  user = "root",
  password = "123456"
}

--创建一个对象
local mysqlobj = {}

--根据SQL语句查询
function mysqlobj.query(sql)
  --创建链接、设置超时时间、编码格式
  local db = mysql:new()
  db:set_timeout(10000)
  db:connect(props)
  db:query("SET NAMES utf8")

  --执行SQL语句
  local result = db:query(sql)

  --关闭链接
  db:close()

  --返回结果集
  return result
end

return mysqlobj
```

2.2.2 链接Redis集群封装

我们需要安装 `lua-resty-redis-cluster` ,用该依赖库实现Redis集群的操作, 这里提供github地址, 大家下载后按操作配置即可, 下载地址: <https://github.com/cuiweixie/lua-resty-redis-cluster> , 下载该文件配置后即可实现Redis集群操作。

接下来我们实现Redis集群操作, 创建一个脚本 `redis.lua` , 脚本中实现Redis中数据的查询和添加操作, 代码如下:

```
--redis连接配置
local config = {
    name = "test",
    serv_list = {
        {ip="192.168.211.141", port = 7001},
        {ip="192.168.211.141", port = 7002},
        {ip="192.168.211.141", port = 7003},
        {ip="192.168.211.141", port = 7004},
        {ip="192.168.211.141", port = 7005},
        {ip="192.168.211.141", port = 7006},
    },
    idle_timeout    = 1000,
    pool_size      = 10000,
}

--引入redis集群配置
local redis_cluster = require "resty.rediscluster"

--定义一个对象
local lredis = {}

--根据key查询
function lredis.get(key)
    --创建链接
    local red = redis_cluster:new(config)
    red:init_pipeline()

    --根据key获取数据
    red:get(key)
    local rresult = red:commit_pipeline()

    --关闭链接
    red:close()

    return rresult
end

--添加带过期的数据
function lredis.set(key,value)
    --创建链接
    local red = redis_cluster:new(config)
    red:init_pipeline()

    --添加key, 同时设置过期时间
```

```
    red:set(key,value)
    local rresult = red:commit_pipeline()
end

return lredis
```

2.2.3 多级缓存操作

配置创建Nginx缓存共享空间

```
lua_shared_dict act_cache 128m;
```

创建多级缓存操作脚本 `activity.lua`，代码如下：

```
--多级缓存流程操作
--1) Lua脚本查询Nginx缓存
--2) Nginx如果没有缓存
--2.1) Lua脚本查询Redis
--2.1.1) Redis如果有数据，则将数据存入到Nginx缓存，并响应用户
--2.1.2) Redis没有数据，Lua脚本查询MySQL
--      MySQL有数据，则将数据存入到Redis、Nginx缓存[需要额外定义]，响应用户
--3) Nginx如果有缓存，则直接将缓存响应给用户

--响应数据为JSON类型
ngx.header.content_type="application/json;charset=utf8"
--引入依赖库
--cjson: 对象转JSON或者JSON转对象
local cjson = require("cjson")
local mysql = require("mysql")
local lredis = require("redis")

--获取请求参数ID   http://192.168.211.141/act?id=1
local id = ngx.req.get_uri_args()["id"];

--加载本地缓存
local cache ngx = ngx.shared.act_cache;

--组装本地缓存的key,并获取nginx本地缓存
local ngx_key = 'ngx_act_cache_'..id
local actCache = cache ngx:get(ngx_key)

--如果nginx中没有缓存，则查询Redis集群缓存
if actCache == "" or actCache == nil then
    --从Redis集群中加载数据
    local redis_key = 'redis_act_'..id
    local result = lredis.get(redis_key)

    --Redis中数据为空，查询数据库
```

```

if result[1]==nil or result[1]==ngx.null then
    --组装SQL语句
    local sql = "select * from activity_info where id ="..id
    --执行查询
    result = mysql.query(sql)
    --数据不为空，则添加到Redis中
    if result[1]==nil or result[1]==ngx.null then
        ngx.say("no data")
    else
        --数据添加到Nginx缓存和Redis缓存
        lrredis.set(redis_key,cjson.encode(result))
        cache_ngx:set(ngx_key, cjson.encode(result), 2*60);
        ngx.say(cjson.encode(result))
    end
else
    --将数据添加到Nginx缓存中
    cache_ngx:set(ngx_key, result, 2*60);
    --直接输出
    ngx.say(result)
end
else
    --输出缓存数据
    ngx.say(actCache)
end
end

```

Nginx配置：

```

#活动查询
location /act {
    content_by_lua_file /usr/local/openresty/nginx/lua/activity.lua;
}

```

3 红包雨案例剖析

每次在双十一或者618的时候，各大电商平台为了吸引用户，都会给与用户很大的优惠，比如送优惠券、抢红包等，抢优惠券或者红包的用户群体非常大，这时候会给服务器带来的并发也是非常大，解决这快问题，架构是关键。

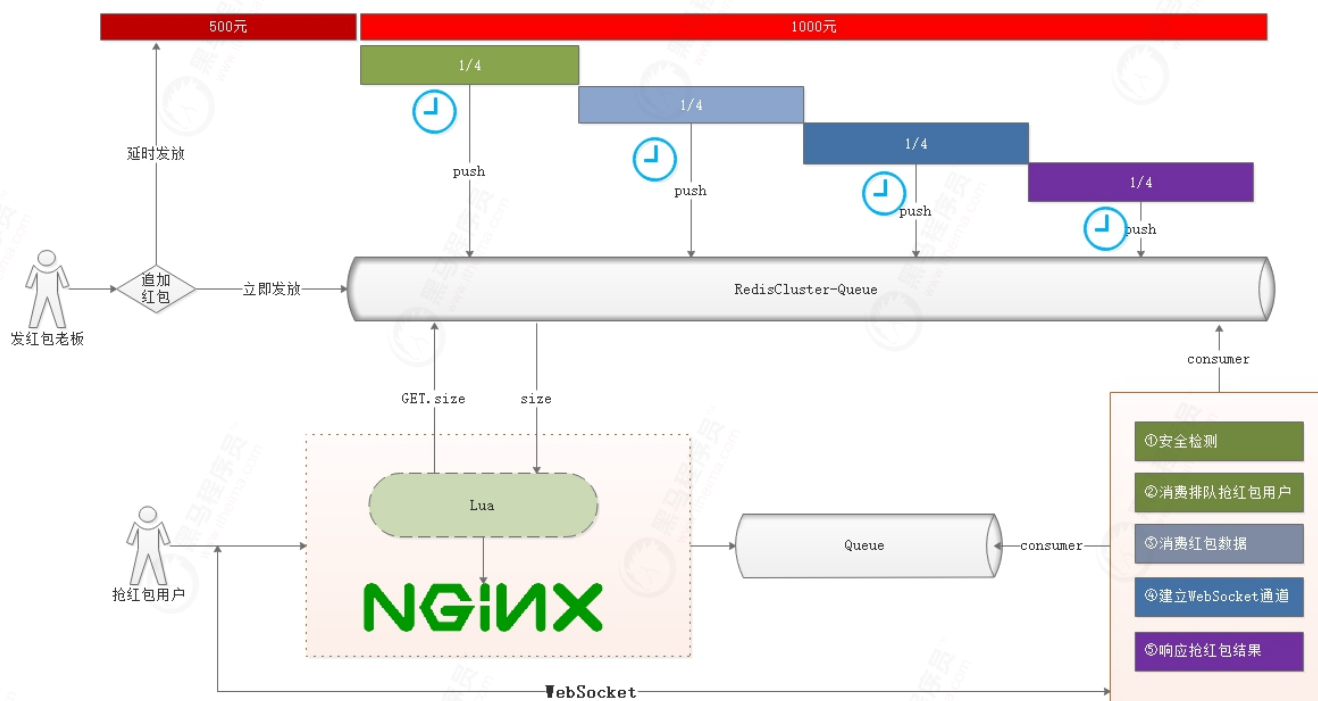
3.1 抢红包案例分析



双十一抢红包或者过年微信、QQ红包雨活动的并发量非常庞大，红包雨存在的规则也是非常多的，我们先来分析一下红包雨的特点。

- 1、并发量大
- 2、红包并非一下全部发完，按时间段发放
- 3、每个用户抢红包存在概率问题和上限问题
- 4、用户抢到的红包总金额不能超过发放的总金额
- 5、中途可以追加发放的红包金额，追击的金额可以立即给用户抢，也可以延时让用户抢

3.2 红包雨多级缓存架构设计



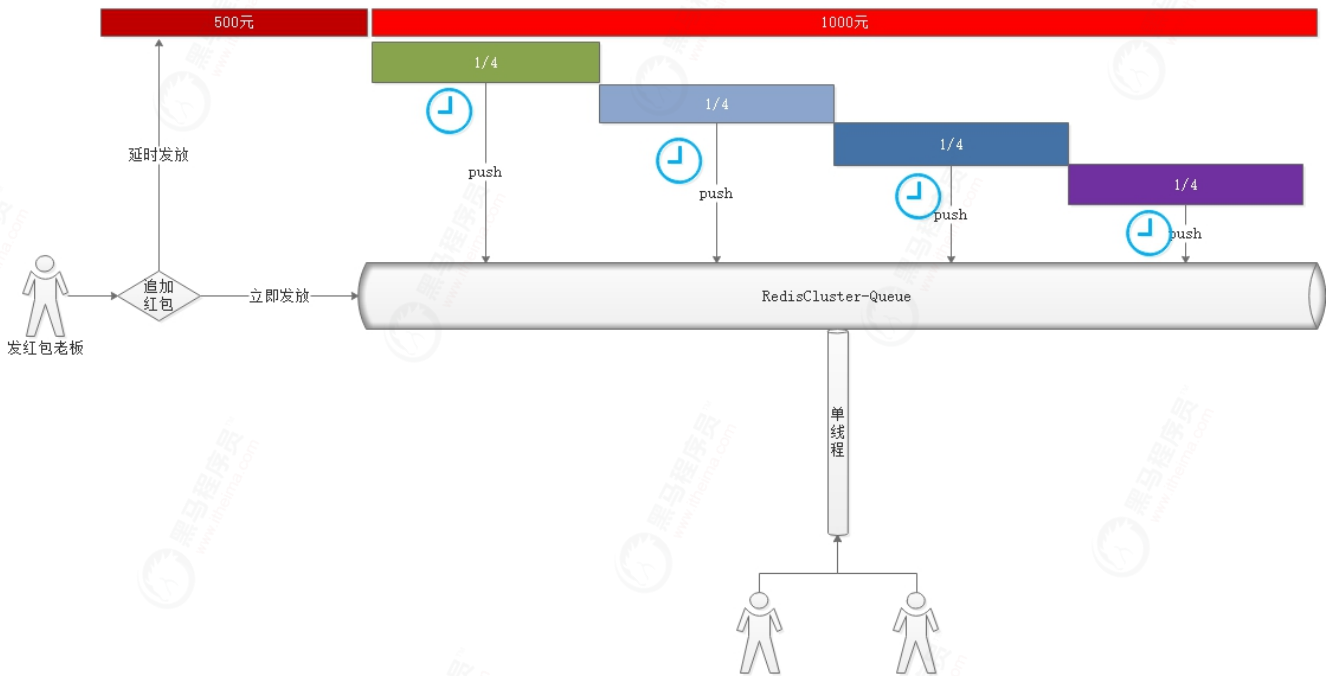
上面我们已经分析过红包雨的特点，要想实现一套高效的红包雨系统，缓存架构是关键。我们根据红包雨的特点设计了如上图所示的红包雨缓存架构体系。

- 1、红包雨分批次导入到Redis缓存而不要每次操作数据库
- 2、很多用户抢红包的时候，为了避免1个红包被多人抢到，我们要采用Redis的队列存储红包
- 3、追加红包的时候，可以追加延时发放红包，也可以直接追加立即发放红包
- 4、用户抢购红包的时候，会先经过Nginx，通过Lua脚本查看缓存中是否存在红包，如果不存在红包，则直接终止抢红包
- 5、如果还存在红包，为了避免后台同时处理很多请求，这里采用队列术缓存用户请求，后端通过消费队列执行抢红包

4 缓存队列-并发溢出高效控制设计

并发量非常大的系统，例如秒杀、抢红包、抢票等操作，都是存在溢出现象，比如秒杀超卖、抢红包超额、一票多单等溢出现象，如果采用数据库锁来控制溢出问题，效率非常低，在高并发场景下，很有可能直接导致数据库崩溃，因此针对高并发场景下数据溢出解决方案我们可以采用Redis缓存提升效率。

4.1 设计分析

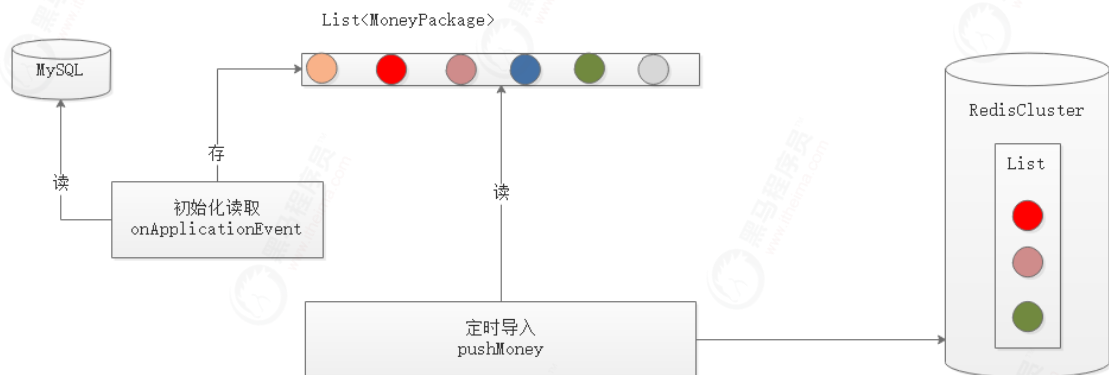


用户抢红包的时候，我们会分批次发放红包，每次发放红包会先根据发放红包的金额和红包的个数把每个红包计算好，然后存入到Redis队列中，存入到Redis队列后，用户每次抢红包都直接从Redis队列中获取一个红包即可，由于Redis是单线程，在这里可以有效的避免多个人同时抢到了一个红包，类似一种超卖现象。

表结构设计：

```
CREATE TABLE `money_package` (
  `id` int(11) NOT NULL COMMENT '主键ID',
  `money` int(11) NOT NULL COMMENT '红包总金额',
  `count` int(11) NOT NULL COMMENT '红包数量',
  `sort` int(2) DEFAULT NULL COMMENT '发红包顺序',
  `type` int(11) DEFAULT '1' COMMENT '红包发放类型 1: 延时发放 2: 立即发放',
  `hasload` int(1) DEFAULT '1' COMMENT '加载位置 1: 未加载 2: 加载到程序 3: 加载到Redis缓存',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

4.2 红包定时导入缓存队列



初始化读取

创建容器监听类 `com.itheima.quartz.MoneyPushTask`，让该类实现接口 `ApplicationListener`，当容器初始化完成后，会调用 `onApplicationEvent` 方法，我们可以在该方法中实现初始化读取数据，将数据填充到变量中。

`MoneyPushTask` 代码如下：

```
@Component
public class MoneyPushTask implements ApplicationListener<ContextRefreshedEvent>{

    //批次下发的金额和红包个数
    private List<MoneyPackage> moneyPackages;

    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private MoneyPackageService moneyPackageService;

    /**
     * 初始化加载
     * @param event
     */
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        //清空历史红包记录
        redisTemplate.delete(Constant.MONEY_QUEUE_NAME);
        //加载所有延时发放的红包金额
        moneyPackages = moneyPackageService.loadDelayData();
    }
}
```

定时加载

定时加载我们需要开启 `Quartz`，在启动类上添加 `@EnableScheduling` 注解，在 `MoneyPushTask` 中添加 `pushMoney` 方法，代码如下：

```
/**
 * 金额推送到Redis缓存队列中
 */
@Scheduled(cron = "0/30 * * * * ?")
public void pushMoney(){
    if(moneyPackages!=null && moneyPackages.size()>0){
        //从集合中移除第1个
        MoneyPackage moneyPackage = moneyPackages.remove(0);
        if(moneyPackage!=null){
            //变更加载状态
            int issuccess = moneyPackageService.modifyHasload(moneyPackage.getId(),3,2);
            if(issuccess<=0){
                return;
            }
        }
    }
}
```

```

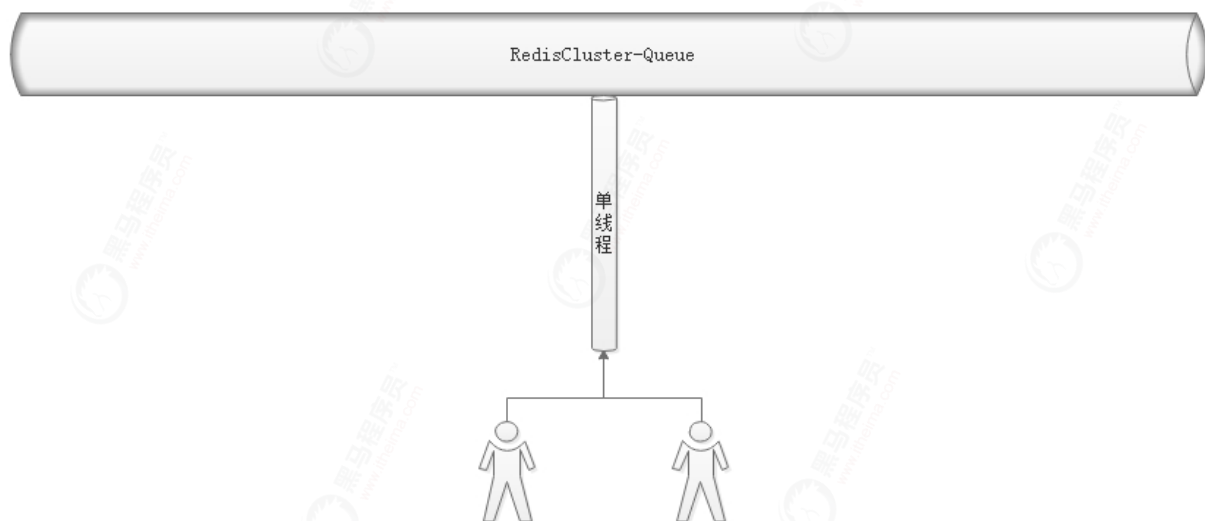
    }

    //红包个数
    Integer count = moneyPackage.getCount();
    //红包总金额
    Integer total = moneyPackage.getMoney();
    //执行金额计算
    List<Double> moneys = ReducePriceUtils.splitReducePrice(total, count);
    Double[] moneyArray = new Double[moneys.size()];
    moneys.toArray(moneyArray);

    //金额存入到Redis缓存队列中
    redisTemplate.boundListOps(Constant.MONEY_QUEUE_NAME).leftPushAll(moneyArray);
}
}
}

```

抢红包缓存队列溢出控制



上面已经实现将红包存入到Redis缓存队列中，用户每次抢红包的时候，只需要从Redis缓存队列中获取即可。

`com.itheima.controller.RedPacketController` 代码：

```

@RestController
@RequestMapping(value = "/red/packet")
public class RedPacketController {

    @Autowired
    private RedPacketService redPacketService;

    /**
     * 抢红包
     */
    @GetMapping(value = "/grab")
    public String grab(){
        return "抢红包金额: "+redPacketService.grab();
    }
}

```

```
}  
}
```

com.itheima/service/RedPacketService.java 代码:

```
public interface RedPacketService {  
    //抢红包  
    Double grab();  
}
```

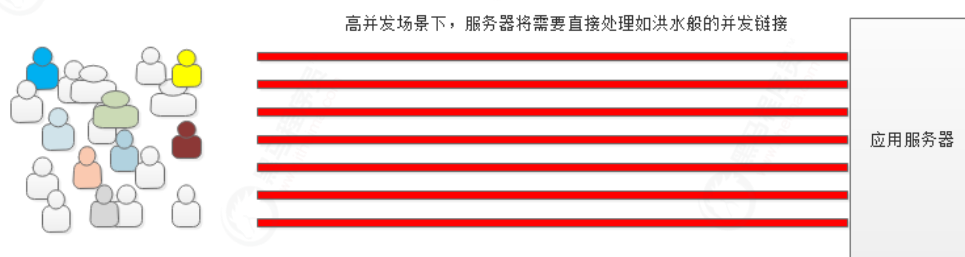
com.itheima.service.impl.RedPacketServiceImpl 代码:

```
@Service  
public class RedPacketServiceImpl implements RedPacketService {  
  
    @Autowired  
    private SessionThreadLocal sessionThreadLocal;  
  
    @Autowired  
    private RedisTemplate redisTemplate;  
  
    /**  
     *  
     * @return  
     */  
    @Override  
    public Double grab() {  
        //获取用户登录信息  
        String username = sessionThreadLocal.get().getUsername();  
  
        //抢红包  
        Object money = redisTemplate.boundListOps(Constant.MONEY_QUEUE_NAME).rightPop();  
        if(StringUtils.isEmpty(money)){  
            return 0.0;  
        }  
        return Double.valueOf(money.toString());  
    }  
}
```

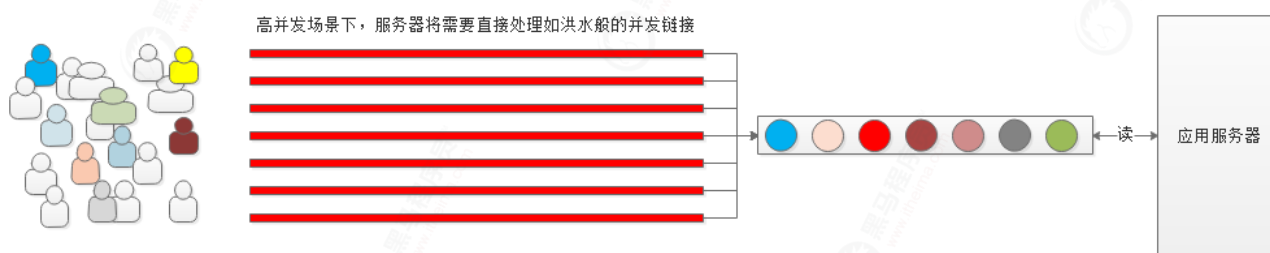
5 队列术限流

5.1 高并发场景分析

用户抢红包的高并发场景下，如果让后端服务器直接处理所有抢红包操作，服务器很有可能会崩溃，就像高铁站如果很多人蜂拥挤进站而不排队，很有可能导致整个高铁站秩序混乱，高铁站服务崩溃，程序也是如此。



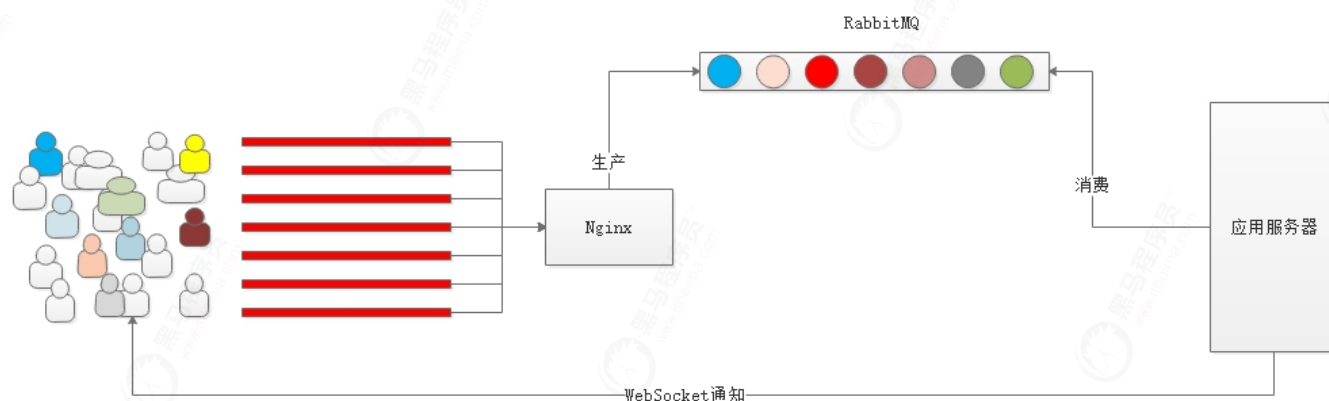
普通模式



队列术

解决大量并发用户蜂拥而上的方法可以采用队列术将用户的请求用队列缓存起来，后端服务从队列缓存中有序消费，可以防止后端服务同时面临处理大量请求。缓存用户请求可以用RabbitMQ、Kafka、RocketMQ、ActiveMQ，我们这里采用RabbitMQ即可。

5.2 队列削峰实战



用户抢红包的时候，我们用Lua脚本实现将用户抢红包的信息以生产者角色将消息发给RabbitMQ，后端应用服务以消费者身份从RabbitMQ获取消息并抢红包，再将抢红包信息以WebSocket方式通知给用户。

如果想使用Lua识别用户令牌，我们需要引入lua-resty-jwt模块，是用于ngx_lua和LuaJIT的Lua实现库，在该模块能实现Jwt令牌生成、Jwt令牌校验，依赖库的地址：<https://github.com/SkyLothar/lua-resty-jwt>

lua-resty-jwt安装

在资料\lua中已经下载好了该依赖库lua-resty-jwt-master.zip,我们将该库文件上传到服务器上,并解压,当然,我们也可以使用opm直接安装lua-resty-jwt,配置lua-resty-jwt之前,我们需要先安装resty和opm。

安装仓库管理工具包:

```
yum install yum-utils
```

添加仓库地址:

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

安装resty:

```
yum install openresty-resty
```

安装opm:

```
yum install openresty-opm
```

安装jwt组件:

```
opm get SkyLothar/lua-resty-jwt
```

此时lua-resty-jwt安装好了,可以直接使用了。

令牌校验库token.lua

```
--依赖jwt库
local jwt = require("resty.jwt")

-- 定义一个名为 jwttoken 的模块
jwttoken = {}

--密钥
--令牌校验auth_header->Bearer DSFLSFJLDLKSDJLJF
function jwttoken.check(auth_header,secret)
    --定义响应数据
    local response = {}

    --如果请求头中没有令牌,则直接返回401
    if auth_header == nil then
        response["code"]=401
        response["message"]="没有找到令牌数据"
        return response
    end

    --查找令牌中的Bearer前缀字符,并进行截取
    local _, _, token = string.find(auth_header, "Bearer%s+(.)")
```

```

--如果没有Bearer, 则表示令牌无效
if token == nil then
    response["code"]=401
    response["message"]="令牌格式不正确"
    return response
end

--校验令牌
local jwt_obj = jwt:verify(secret, token)

--如果校验结果中的verified==false, 则表示令牌无效
if jwt_obj.verified == false then
    response["code"]=401
    response["message"]="令牌无效"
    return response
end

--全部校验完成后, 说明令牌有效, 返回令牌数据
response["code"]=200
response["message"]="令牌校验通过"
response["body"]=jwt_obj
return response
end

return jwttoken

```

抢红包队列脚本:mq.lua,参考地址 <<https://github.com/wingify/lua-resty-rabbitmqstomp>>

```

--输出json数据
ngx.header.content_type="application/json;charset=utf-8"

--依赖库
local cJSON = require "cjson"
local rabbitmq = require "resty.rabbitmqstomp"
local jwttoken = require "token"

--账号密码
local opts = { username = "guest",
                password = "guest",
                vhost = "/" }

--链接RabbitMQ
local mq, err = rabbitmq:new(opts)
mq:set_timeout(10000)
local ok, err = mq:connect("192.168.211.141",61613)

--秘钥
local secret="5pi16a005YaN57605Lmf5q+U5LiN5LiK5bCP6ZuF55qE56yR"

--获取请求头中的令牌数据

```

```

local auth_header = ngx.var.http_Authorization

--调用令牌校验
local result = jwttoken.check(auth_header,secret)

if result.code==200 then
    --要发送的消息
    local msg = {}
    msg["user"]=result.body.payload.username

    --消息封装
    local headers = {}
    headers["destination"] = "/queue/red.queue"
    headers["receipt"] = "msg#1"
    headers["app-id"] = "luaresty"
    headers["persistent"] = "true"
    headers["content-type"] = "text/plain"
    headers["content_encoding"] = "UTF-8"

    --响应消息
    local resp = {}

    --消息发送
    local content = cJSON.encode(msg)
    local ok, err = mq:send(cJSON.encode(msg), headers)

    if not ok then
        resp["code"]=500
        resp["message"]="手太慢了"
        resp["body"]=err
        ngx.say(cJSON.encode(resp))
        return
    end

    --成功
    resp["code"]=200
    resp["message"]="排队抢红包! "
    resp["body"]=err
    ngx.say(cJSON.encode(resp))
    return
else
    ngx.say(cJSON.encode(result))
end

```

Java监听

```

@Component
public class MoneyListener {

    @Autowired

```

```

private RedPacketService redPacketService;

@Autowired
private RabbitTemplate rabbitTemplate;

@Autowired
private MoneyLogService moneyLogService;

/**
 * 监听队列money
 */
@RabbitListener(queues = "red.queue")
public void moneyList(String message) throws IOException {
    try {
        System.out.println(message);
        Map<String, Object> map = JSON.parseObject(message, Map.class);

        //抢红包
        Double money = redPacketService.grab(map.get("user").toString());

        if(money>0){
            //抢红包记录
            moneyLogService.add(new MoneyLog(money, map.get("user").toString()));
        }

        //封装抢红包信息
        Map<String, Object> resultMap = new HashMap<String, Object>();
        resultMap.put("user", map.get("user").toString());
        resultMap.put("money", money);
        resultMap.put("code", money>0? 200 :204);

        //发送MQ消息
        rabbitTemplate.convertAndSend("", "redmessage.queue", JSON.toJSONString(resultMap));
        System.out.println("消息发送成功! ");
    } catch (Exception e) {
    }
}
}

```

5.3 抢红包测试

打开资料中的 `html/websocket.html`，执行抢红包测试，效果如下：

抢红包

服务端响应数据如下:

```
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 861.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 923.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1156.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1383.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 652.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 268.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 965.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 932.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 451.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 36.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 5.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 1.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 40.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 15.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 251.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 302.0
用户: lisi, 抢红包信息: 恭喜你, 成功抢到红包! 红包金额是: 214.0
```

每次抢红包的金额不一样

6 Nginx限流(作业)

一般情况下, 双十一很多查询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。

6.1 限流理解

限流和生活中很多现实场景类似, 如下场景:

- 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)

而程序限流, 主要是控制应对高并发和恶意操作的一种技术手段。

6.2 Nginx限流实战

nginx提供两种限流的方式:

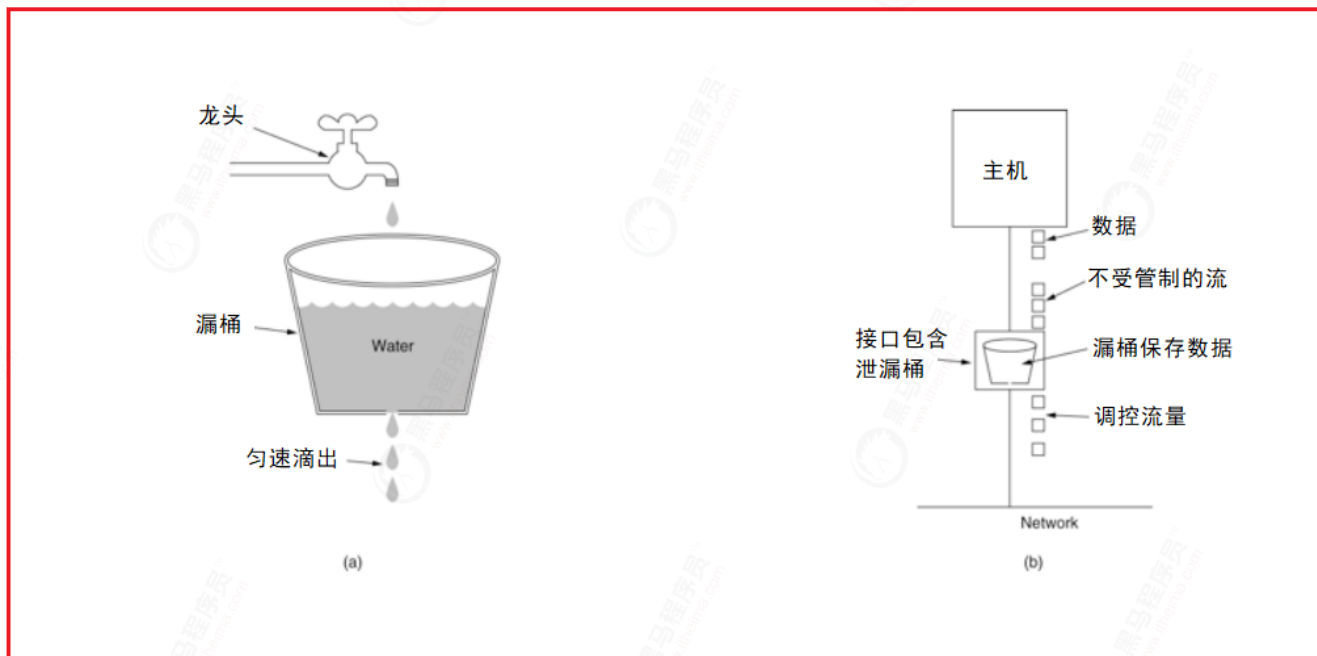
- 一是控制速率
- 二是控制并发连接数

6.2.1 速率限流

控制速率的方式之一就是采用漏桶算法。

(1)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:



(2)nginx的配置

配置示如下:

创建限流缓存空间:

```
#限流设置
limit_req_zone $binary_remote_addr zone=contentRateLimit:10m rate=2r/s;
```

配置限流:

```
#使用限流配置
limit_req zone=contentRateLimit;
```

参数说明:

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。

如果恶意一直刷新会出现如下现象:

192.168.211.141/user/wangwu
503 Service Temporarily Unavailable
openresty/1.11.2.5

(3)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:

```
limit_req zone=contentRateLimit burst=4;
```

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 $50 * 100\text{ms}$ 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:

```
limit_req zone=contentRateLimit burst=4 nodelay;
```

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。

完整配置如下:

```
#限流设置
limit_req_zone $binary_remote_addr zone=contentRateLimit:10m rate=2r/s;

server {
    listen      80;
    server_name localhost;

    #跨域配置
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Methods 'GET,POST';
    add_header Access-Control-Allow-Headers 'DNT,X-Mx-ReqToken,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Authorization';

    #静态页访问
    location ~ /\.html {
        root /usr/local/server/html;
        #配置有效期
        expires 10h;
    }

    #双十一活动查询
    location ~ /act {
        content_by_lua_file /usr/local/openresty/nginx/lua/activity.lua;
    }

    #抢红包
    location ~ /money {
        content_by_lua_file /usr/local/openresty/nginx/lua/mq.lua;
    }

    #限流配置-速率限流
    location ~ /user {
        #使用指定的缓存空间来实现限流
        limit_req zone=contentRateLimit burst=4 nodelay;
        proxy_pass http://myip:18082;
    }
}
```

6.2.2 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;
```

(1)配置限制固定连接数

如下，配置如下：

配置限流缓存空间：

```
#根据IP地址来限制，存储内存大小10M
limit_conn_zone $binary_remote_addr zone=addr:1m;
```

location配置：

```
limit_conn addr 2;
```

参数说明：

`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。

(2)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。

限流缓存空间配置:

```
#IP限流
limit_conn_zone $binary_remote_addr zone=perip:10m;
#根据server的名字限流
limit_conn_zone $server_name zone=perserver:10m;
```

location配置

```
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
```

每个IP限流 3个

总量5个