多级缓存

亿级流量的缓存方案

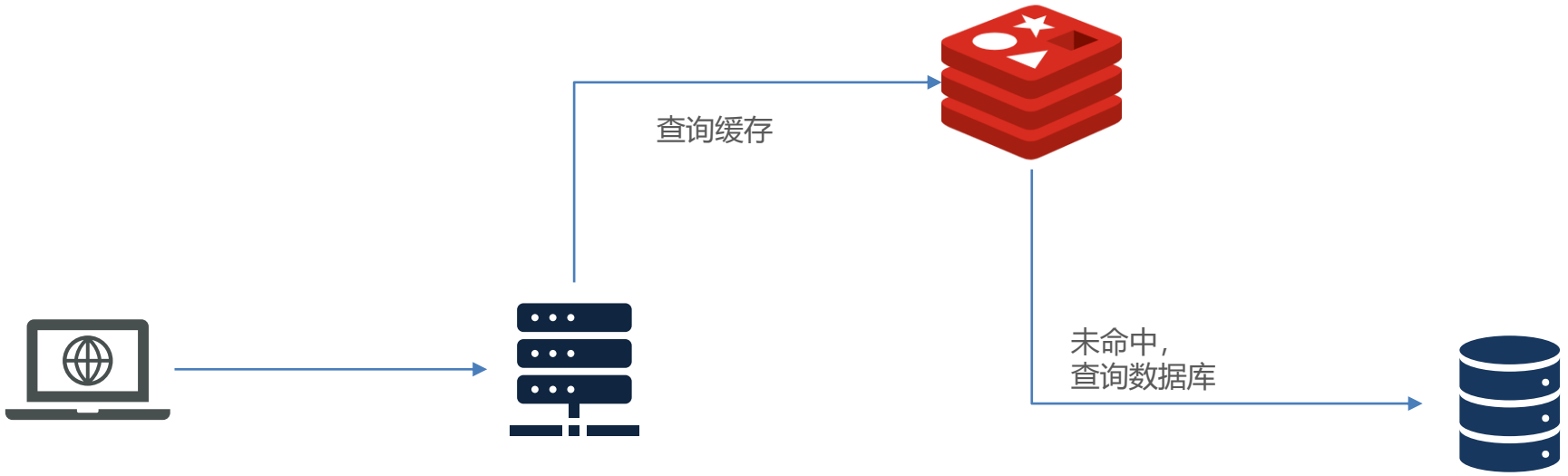
黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

传统缓存的问题

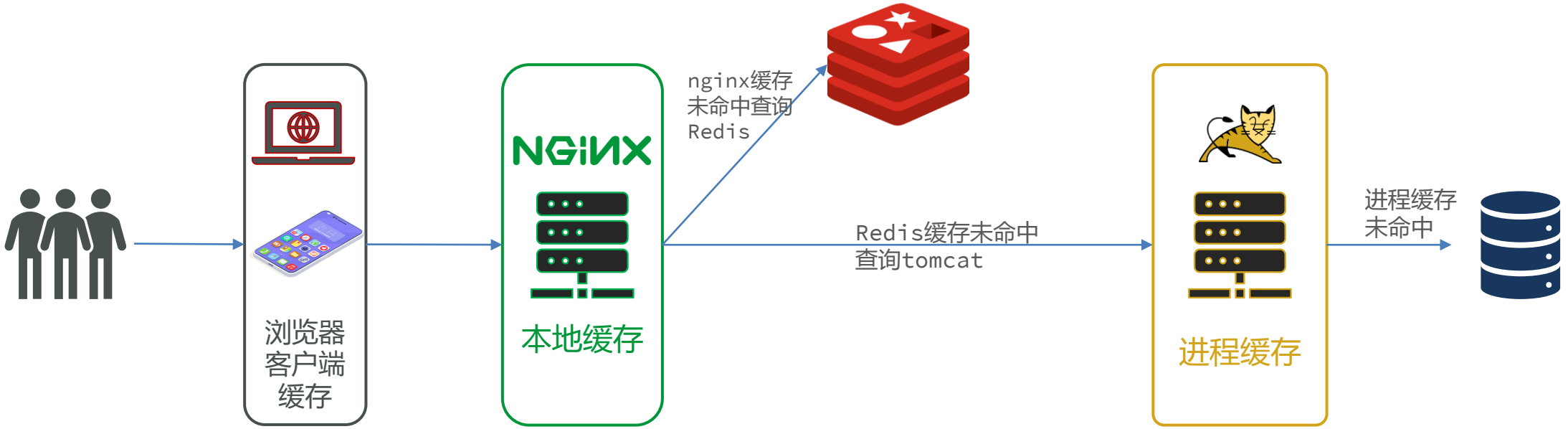
传统的缓存策略一般是请求到达Tomcat后，先查询Redis，如果未命中则查询数据库，存在下面的问题：

- 请求要经过Tomcat处理，Tomcat的性能成为整个系统的瓶颈
- Redis缓存失效时，会对数据库产生冲击



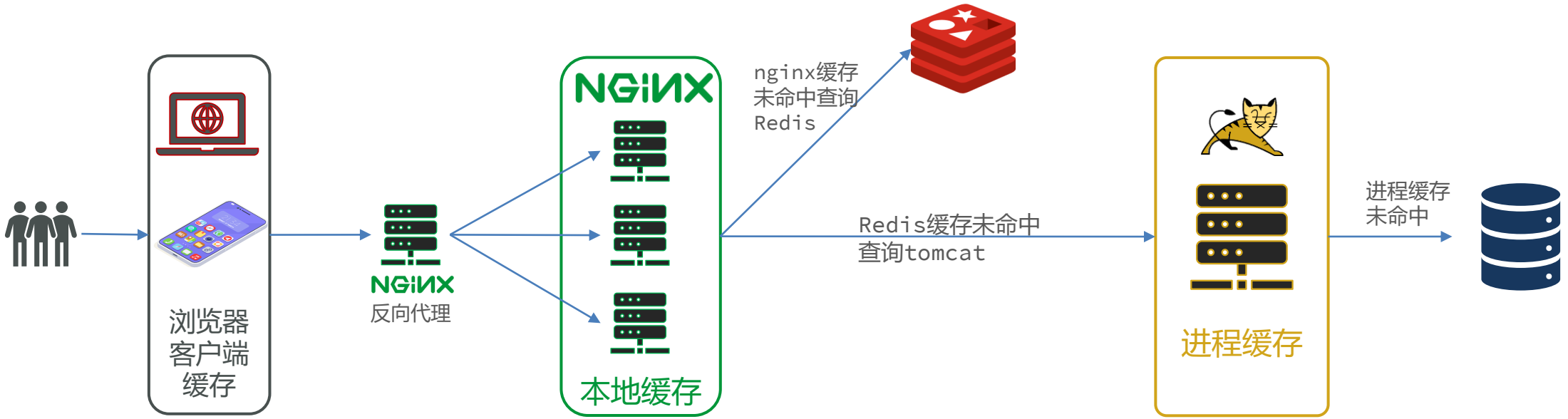
多级缓存方案

多级缓存就是充分利用请求处理的每个环节，分别添加缓存，减轻Tomcat压力，提升服务性能：



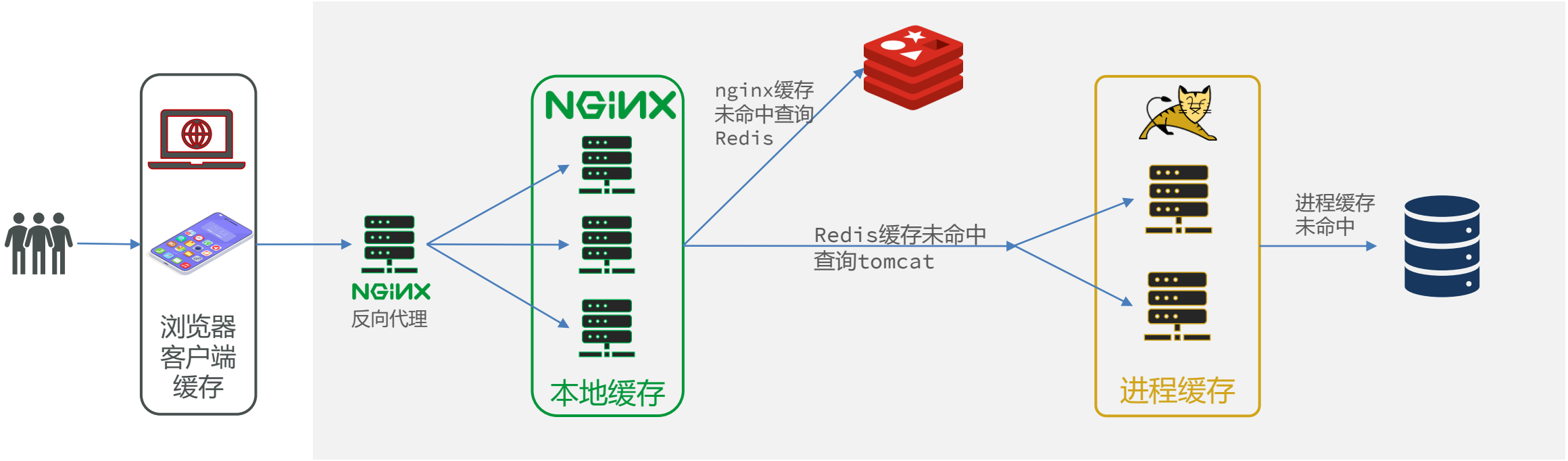
多级缓存方案

用作缓存的Nginx是业务Nginx，需要部署为集群，再有专门的Nginx用来做反向代理：



多级缓存方案

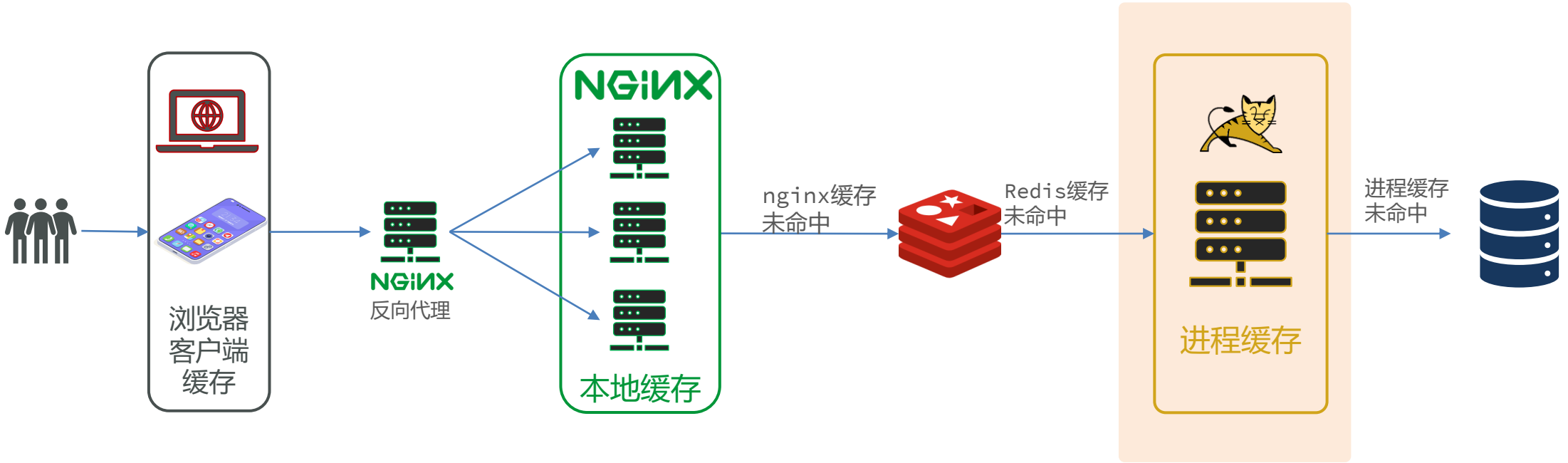
用作缓存的Nginx是业务Nginx，需要部署为集群，再有专门的Nginx用来做反向代理：



1 JVM进程缓存　2 Lua语法入门　3 实现多级缓存　4 缓存同步策略

目录

Contents


浏览器
客户端
缓存
NGINX
反向代理
NGINX
本地缓存
nginx缓存
未命中
Redis缓存
未命中
进程缓存
进程缓存
未命中

JVM进程缓存

- 导入商品案例
- 初识Caffeine
- 实现进程缓存

目录

Contents

导入商品管理案例

参考课前资料提供的文档来导入案例：

案例导入说明.md

目录

Contents

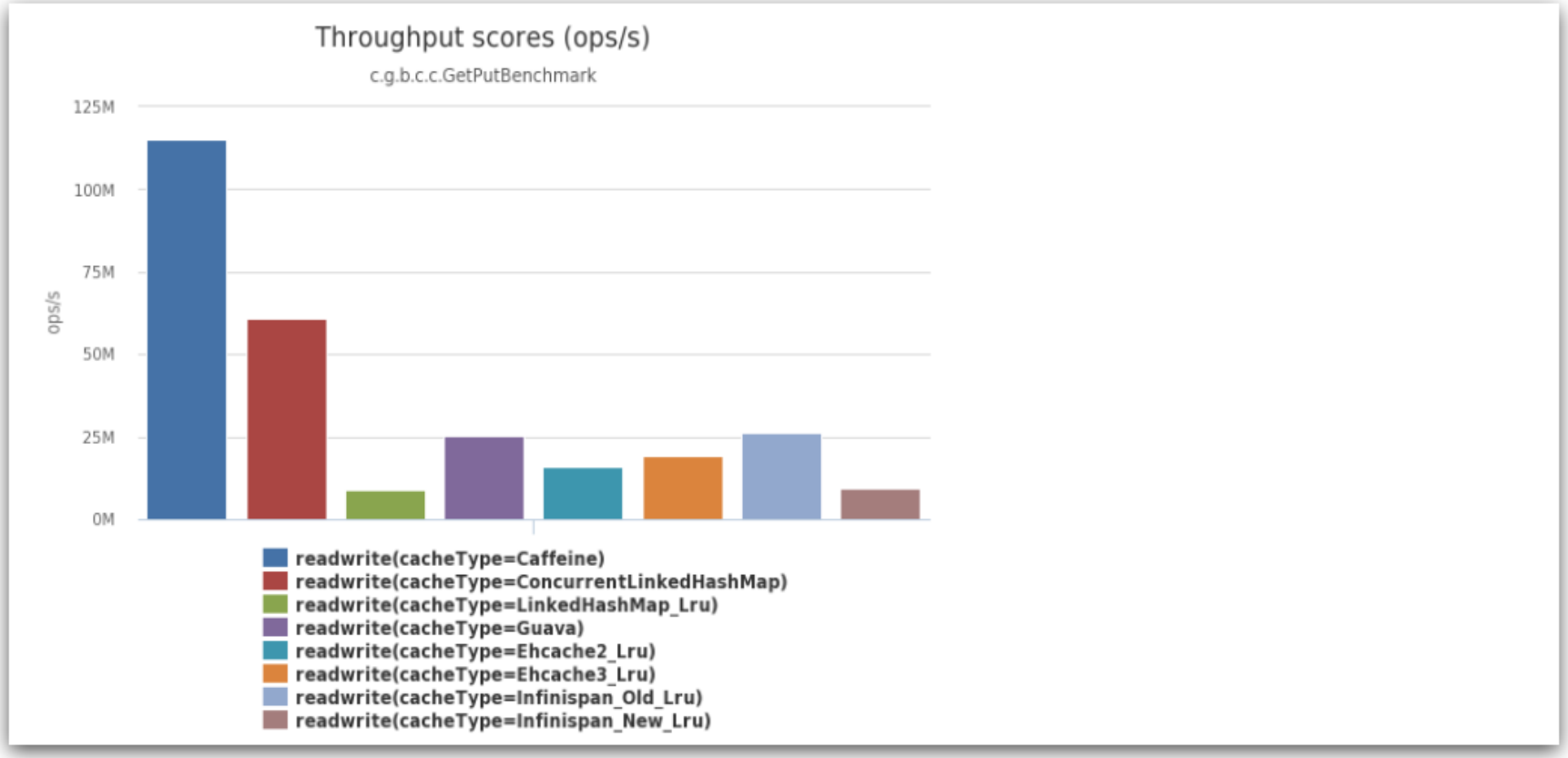
本地进程缓存

缓存在日常开发中启动至关重要的作用，由于是存储在内存中，数据的读取速度是非常快的，能大量减少对数据库的访问，减少数据库的压力。我们把缓存分为两类：

- 分布式缓存，例如Redis：
  - 优点：存储容量更大、可靠性更好、可以在集群间共享
  - 缺点：访问缓存有网络开销
  - 场景：缓存数据量较大、可靠性要求较高、需要在集群间共享
- 进程本地缓存，例如HashMap、GuavaCache：
  - 优点：读取本地内存，没有网络开销，速度更快
  - 缺点：存储容量有限、可靠性较低、无法共享
  - 场景：性能要求较高，缓存数据量较小

本地进程缓存

Caffeine是一个基于Java8开发的，提供了近乎最佳命中率的高性能的本地缓存库。目前Spring内部的缓存使用的就是Caffeine。GitHub地址：https://github.com/ben-manes/caffeine



Caffeine示例

可以通过item-service项目中的单元测试来学习Caffeine的使用：

```
@Test
void testBasicOps() {
    // 创建缓存对象
    Cache<String, String> cache = Caffeine.newBuilder().build();

    // 存数据
    cache.put("gf", "迪丽热巴");

    // 取数据，不存在则返回null
    String gf = cache.getIfPresent("gf");
    System.out.println("gf = " + gf);

    // 取数据，包含两个参数：
    // 参数一：缓存的key
    // 参数二：Lambda表达式，表达式参数就是缓存的key，方法体是查询数据库的逻辑
    // 优先根据key查询JVM缓存，如果未命中，则执行参数二的Lambda表达式
    String defaultGF = cache.get("defaultGF", key -> {
        // 这里可以去数据库根据 key查询value
        return "柳岩";
    });
    System.out.println("defaultGF = " + defaultGF);
}
```

Caffeine示例

Caffeine提供了三种缓存驱逐策略：

- **基于容量**：设置缓存的数量上限

```
// 创建缓存对象
Cache<String, String> cache = Caffeine.newBuilder()
        .maximumSize(1) // 设置缓存大小上限为 1
        .build();
```

- **基于时间**：设置缓存的有效时间

```
// 创建缓存对象
Cache<String, String> cache = Caffeine.newBuilder()
        .expireAfterWrite(Duration.ofSeconds(10)) // 设置缓存有效期为 10 秒，从最后一次写入开始计时
        .build();
```

- **基于引用**：设置缓存为软引用或弱引用，利用GC来回收缓存数据。性能较差，不建议使用。

在默认情况下，当一个缓存元素过期的时候，Caffeine不会自动立即将其清理和驱逐。而是在一次读或写操作后，或者在空闲时间完成对失效数据的驱逐。

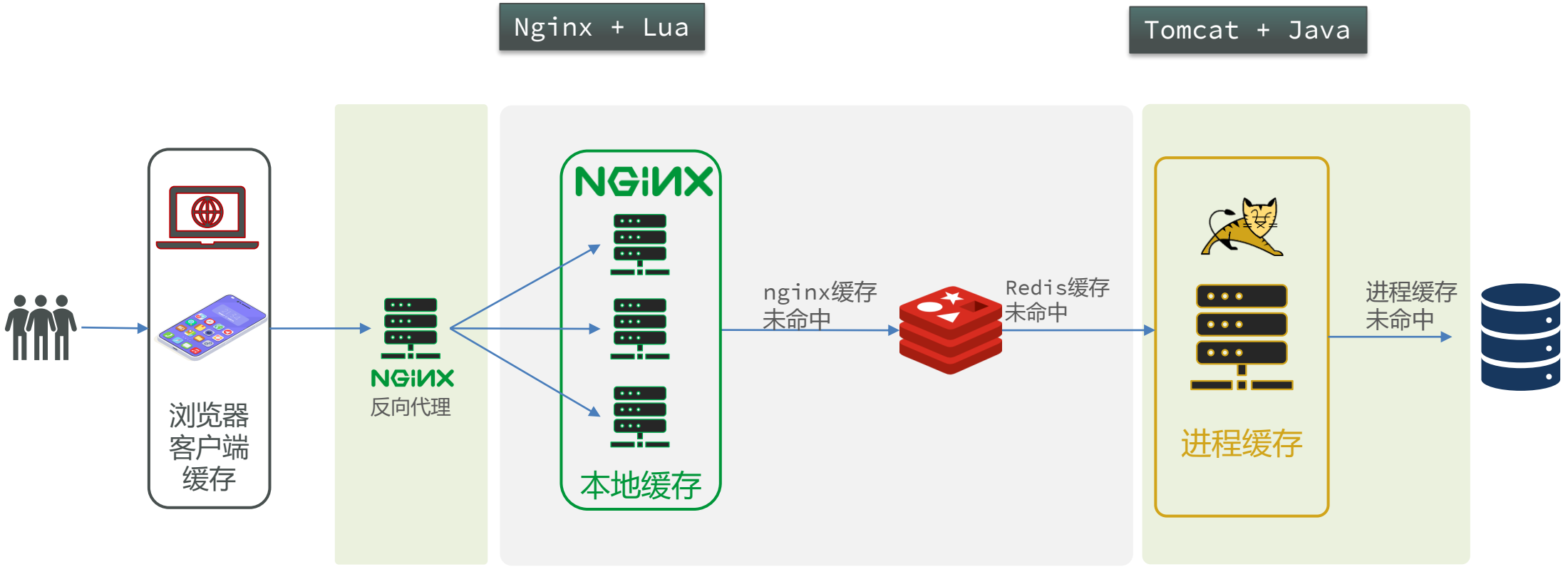
目录

Contents

案例 实现商品的查询的本地进程缓存

利用Caffeine实现下列需求：

- 给根据id查询商品的业务添加缓存，缓存未命中时查询数据库
- 给根据id查询商品库存的业务添加缓存，缓存未命中时查询数据库
- 缓存初始大小为100
- 缓存上限为10000


Nginx + Lua
Tomcat + Java
浏览器
客户端
缓存
NGINX
反向代理
NGINX
本地缓存
nginx缓存
未命中
Redis缓存
未命中
进程缓存
进程缓存
未命中

Lua语法入门

- 初识Lua
- 变量和循环
- 条件控制、函数

目录

Contents

初识Lua

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

HelloWorld

1. 在Linux虚拟机的任意目录下，新建一个hello.lua文件

```
[root@node1 ~]# touch hello.lua
[root@node1 ~]#
```

2. 添加下面的内容

```
print("Hello World!")
```

3. 运行

```
[root@node1 ~]# lua hello.lua
hello world
```

目录

Contents

数据类型

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
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。

可以利用type函数测试给定变量或者值的类型：

```
> print(type("Hello world"))
string
> print(type(10.4*3))
number
```

变量

Lua声明变量的时候，并不需要指定数据类型：

```lua
-- 声明字符串
local str = 'hello'
-- 字符串拼接可以使用 ..
local str2 = 'hello' .. 'world'
-- 声明数字
local num = 21
-- 声明布尔类型
local flag = true
-- 声明数组 key为索引的 table
local arr = {'java', 'python', 'lua'}
-- 声明table，类似java的map
local map =  {name='Jack', age=21}
```

访问table：

```lua
-- 访问数组，lua数组的角标从1开始
print(arr[1])
-- 访问table
print(map['name'])
print(map.name)
```

循环

数组、table都可以利用for循环来遍历：

- 遍历数组：

```
-- 声明数组 key为索引的 table
local arr = {'java', 'python', 'lua'}
-- 遍历数组
for index,value in ipairs(arr) do
    print(index, value)
end
```

- 遍历table：

```
-- 声明map，也就是table
local map = {name='Jack', age=21}
-- 遍历table
for key,value in pairs(map) do
   print(key, value)
end
```

目录

Contents

函数

定义函数的语法：

```
function 函数名( argument1, argument2..., argumentn)
    -- 函数体
    return 返回值
end
```

例如，定义一个函数，用来打印数组：

```
function printArr(arr)
    for index, value in ipairs(arr) do
        print(value)
    end
end
```

条件控制

类似Java的条件控制，例如if、else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
```

与java不同，布尔表达式中的逻辑运算是基于英文单词：

操作符	描述	实例
and	逻辑与操作符。 若 A 为 false，则返回 A，否则返回 B。	(A and B) 为 false。
or	逻辑或操作符。 若 A 为 true，则返回 A，否则返回 B。	(A or B) 为 true。
not	逻辑非操作符。与逻辑运算结果相反，如果条件为 true，逻辑非为 false。	not(A and B) 为 true。

案例 自定义函数，打印table

需求：自定义一个函数，可以打印table，当参数为nil时，打印错误信息

多级缓存

- 安装OpenResty
- OpenResty快速入门
- 请求参数处理
- 查询Tomcat
- Redis缓存预热
- 查询Redis缓存
- Nginx本地缓存

目录

Contents

初识OpenResty

OpenResty® 是一个基于 Nginx的高性能 Web 平台，用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。具备下列特点：

- 具备Nginx的完整功能
- 基于Lua语言进行扩展，集成了大量精良的 Lua 库、第三方模块
- 允许使用Lua自定义业务逻辑、自定义库

官方网站： https://openresty.org/cn/

初识OpenResty

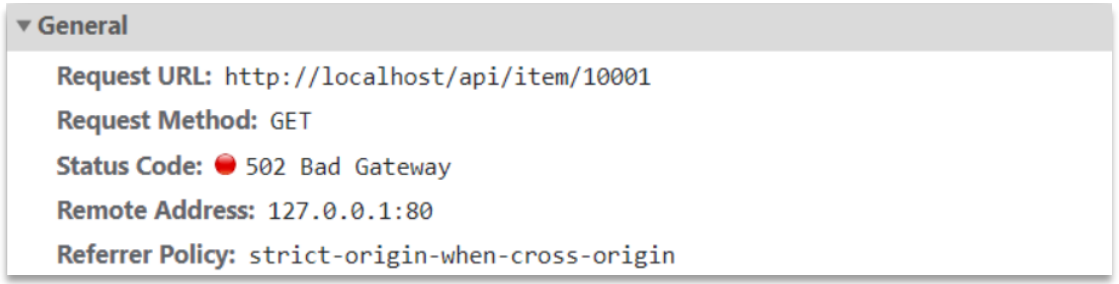
安装OpenResty可以参考课前资料：

目录

Contents

案例 OpenResty快速入门，实现商品详情页数据查询

商品详情页面目前展示的是假数据，在浏览器的控制台可以看到查询商品信息的请求：



而这个请求最终被反向代理到虚拟机的OpenResty集群：

需求：在OpenResty中接收这个请求，并返回一段商品的假数据。

步骤

步骤一：修改nginx.conf文件

1. 在nginx.conf的http下面，添加对OpenResty的Lua模块的加载：

```
# 加载lua 模块
lua_package_path "/usr/local/openresty/lualib/?.lua;;";
# 加载c模块
lua_package_cpath "/usr/local/openresty/lualib/?.so;;";
```

2. 在nginx.conf的server下面，添加对/api/item这个路径的监听：

```
location /api/item {
    # 响应类型，这里返回json
    default_type application/json;
    # 响应数据由 lua/item.lua这个文件来决定
    content_by_lua_file lua/item.lua;
}
```

步骤

步骤二：编写item.lua文件

1. 在nginx目录创建文件夹：lua

```
[root@node1 nginx]# pwd
/usr/local/openresty/nginx
[root@node1 nginx]# mkdir lua
```

2. 在lua文件夹下，新建文件：item.lua

```
[root@node1 nginx]# pwd
/usr/local/openresty/nginx
[root@node1 nginx]# touch lua/item.lua
```

3. 内容如下：

```
-- 返回假数据，这里的ngx.say()函数，就是写数据到Response中
ngx.say('{"id":10001,"name":"SALSA AIR}')
```

4. 重新加载配置

```
nginx -s reload
```

目录

Contents

OpenResty获取请求参数

OpenResty提供了各种API用来获取不同类型的请求参数：

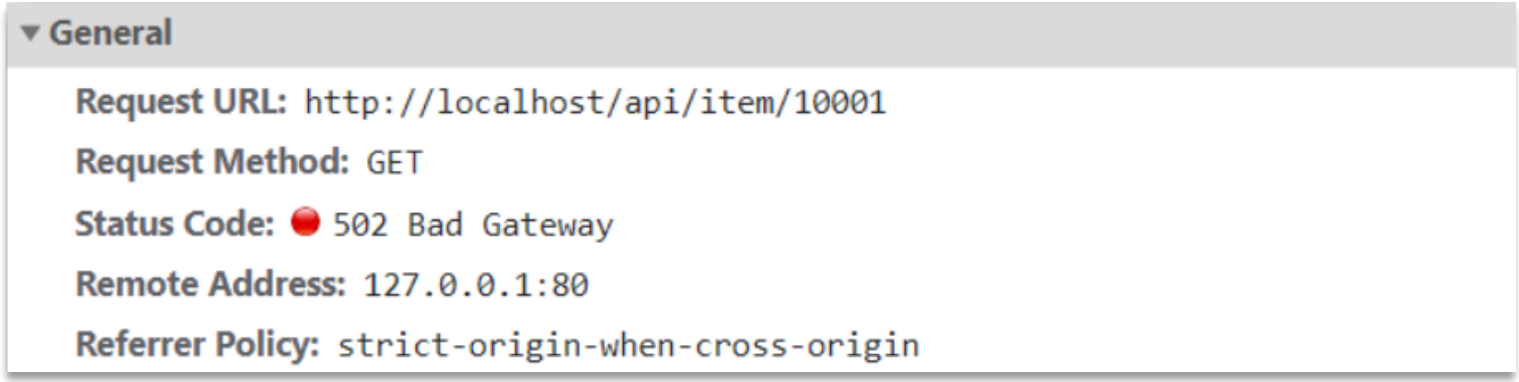
参数格式	参数示例	参数解析代码示例

路径占位符	/item/1001	# 1.正则表达式匹配：
		location ~ /item/(\d+) {
		  content_by_lua_file lua/item.lua;
		}
		
		-- 2. 匹配到的参数会存入ngx.var数组中，
		-- 可以用角标获取
		local id = ngx.var[1]

请求头	id: 1001	-- 获取请求头，返回值是table类型 local headers = ngx.req.get_headers()
Get请求参数	?id=1001	-- 获取GET请求参数，返回值是table类型 local getParams = ngx.req.get_uri_args()
Post表单参数	id=1001	-- 读取请求体 ngx.req.read_body() -- 获取POST表单参数，返回值是table类型 local postParams = ngx.req.get_post_args()
JSON参数	{"id": 1001}	-- 读取请求体 ngx.req.read_body() -- 获取body中的json参数，返回值是string类型 local jsonBody = ngx.req.get_body_data()

案例 获取请求路径中的商品id信息，拼接到json结果中返回

在查询商品信息的请求中，通过路径占位符的方式，传递了商品id到后台：

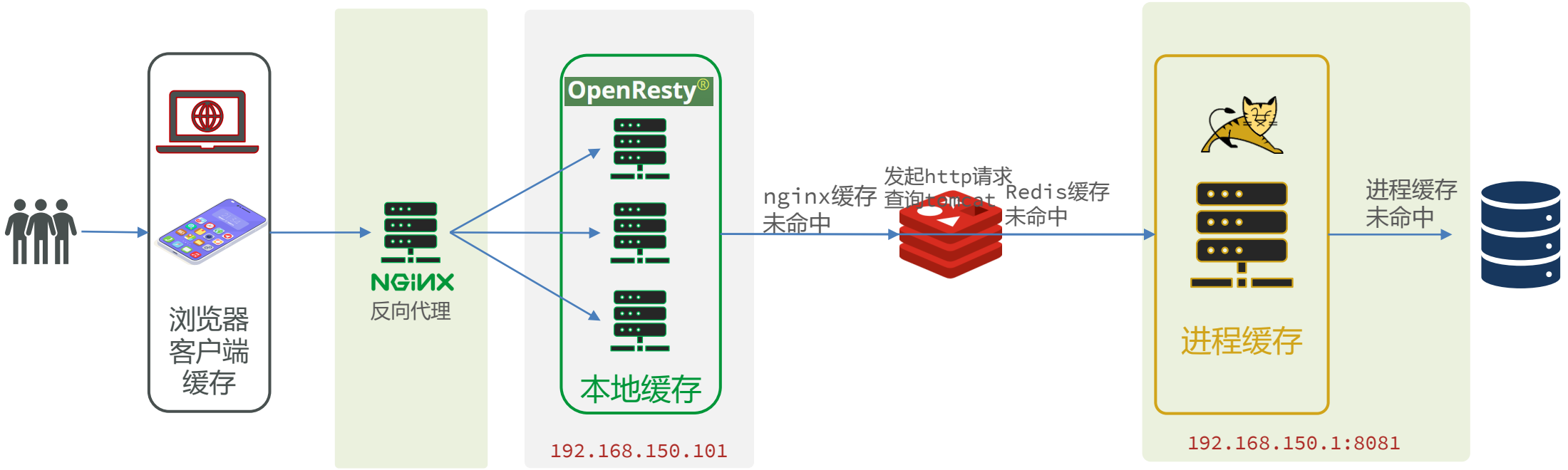


需求：在OpenResty中接收这个请求，并获取路径中的id信息，拼接到结果的json字符串中返回

目录

Contents

多级缓存需求


浏览器
客户端
缓存
NGINX
反向代理
OpenResty®
本地缓存
192.168.150.101
nginx缓存
未命中
发起http请求
查询tomcat
Redis缓存
未命中
进程缓存
进程缓存
192.168.150.1:8081

案例 获取请求路径中的商品id信息，根据id向Tomcat查询商品信息

这里要修改item.lua，满足下面的需求：

1. 获取请求参数中的id
2. 根据id向Tomcat服务发送请求，查询商品信息
3. 根据id向Tomcat服务发送请求，查询库存信息
4. 组装商品信息、库存信息，序列化为JSON格式并返回

nginx内部发送Http请求

nginx提供了内部API用以发送http请求：

```
local resp = ngx.location.capture("/path",{
    method = ngx.HTTP_GET,   -- 请求方式
    args = {a=1,b=2},  -- get方式传参数
    body = "c=3&d=4" -- post方式传参数
})
```

返回的响应内容包括：

- resp.status：响应状态码
- resp.header：响应头，是一个table
- resp.body：响应体，就是响应数据

注意：这里的path是路径，并不包含IP和端口。这个请求会被nginx内部的server监听并处理。

但是我们希望这个请求发送到Tomcat服务器，所以还需要编写一个server来对这个路径做反向代理：

```
location /path {
    # 这里是windows电脑的ip和Java服务端口，需要确保windows防火墙处于关闭状态
    proxy_pass http://192.168.150.1:8081;
}
```

封装http查询的函数

我们可以把http查询的请求封装为一个函数，放到OpenResty函数库中，方便后期使用。

1. 在/usr/local/openresty/lualib目录下创建common.lua文件：

```
vi /usr/local/openresty/lualib/common.lua
```

2. 在common.lua中封装http查询的函数

```
-- 封装函数，发送http请求，并解析响应
local function read_http(path, params)
    local resp = ngx.location.capture(path,{
        method = ngx.HTTP_GET,
        args = params,
    })
    if not resp then
        -- 记录错误信息，返回404
        ngx.log(ngx.ERR, "http not found, path: ", path , ", args: ", args)
        ngx.exit(404)
    end
    return resp.body
end
-- 将方法导出
local _M = {
    read_http = read_http
}
return _M
```

使用Http函数查询数据

我们刚才已经把http查询的请求封装为一个函数，放到OpenResty函数库中，接下来就可以使用这个库了。

- 修改item.lua文件

```
-- 引入自定义工具模块
local common = require("common")
local read_http = common.read_http

-- 获取路径参数
local id = ngx.var[1]

-- 根据id查询商品
local itemJSON = read_http("/item/".. id, nil)
-- 根据id查询商品库存
local itemStockJSON = read_http("/item/stock/".. id, nil)
```

查询到的是商品、库存的json格式数据，我们需要将两部分数据组装，需要用到JSON处理函数库。

JSON结果处理

OpenResty提供了一个cjson的模块用来处理JSON的序列化和反序列化。

官方地址： https://github.com/openresty/lua-cjson/

- 引入cjson模块：

```
local cjson = require "cjson"
```

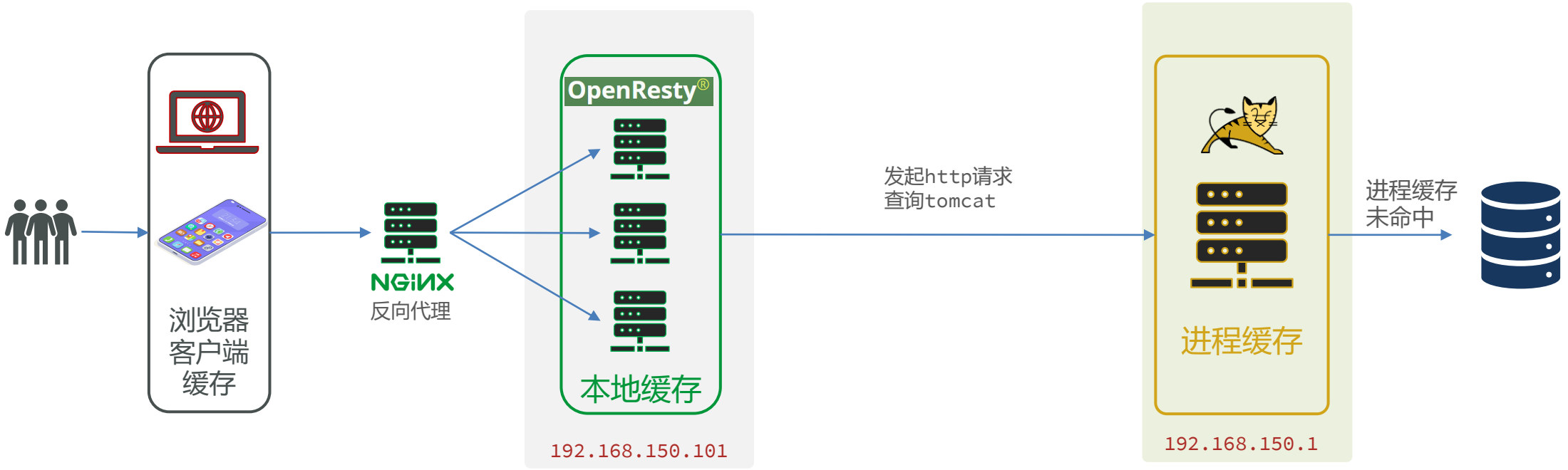
- 序列化：

```
local obj = {
    name = 'jack',
    age = 21
}
local json = cjson.encode(obj)
```

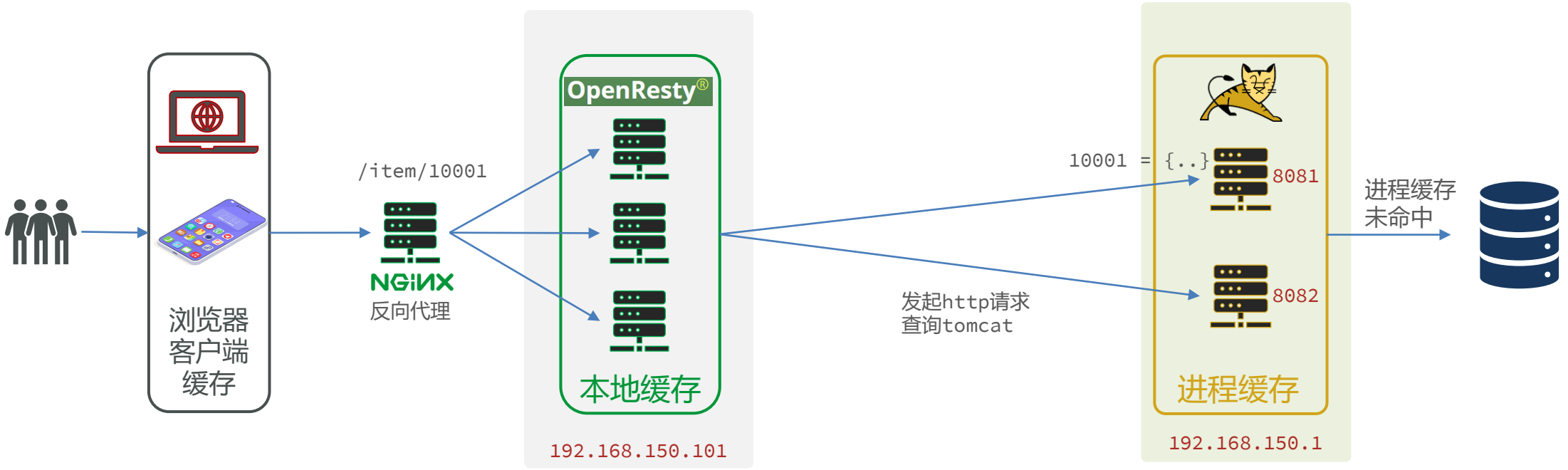
- 反序列化：

```
local json = '{"name": "jack", "age": 21}'
-- 反序列化
local obj = cjson.decode(json);
print(obj.name)
```

Tomcat集群的负载均衡



Tomcat集群的负载均衡



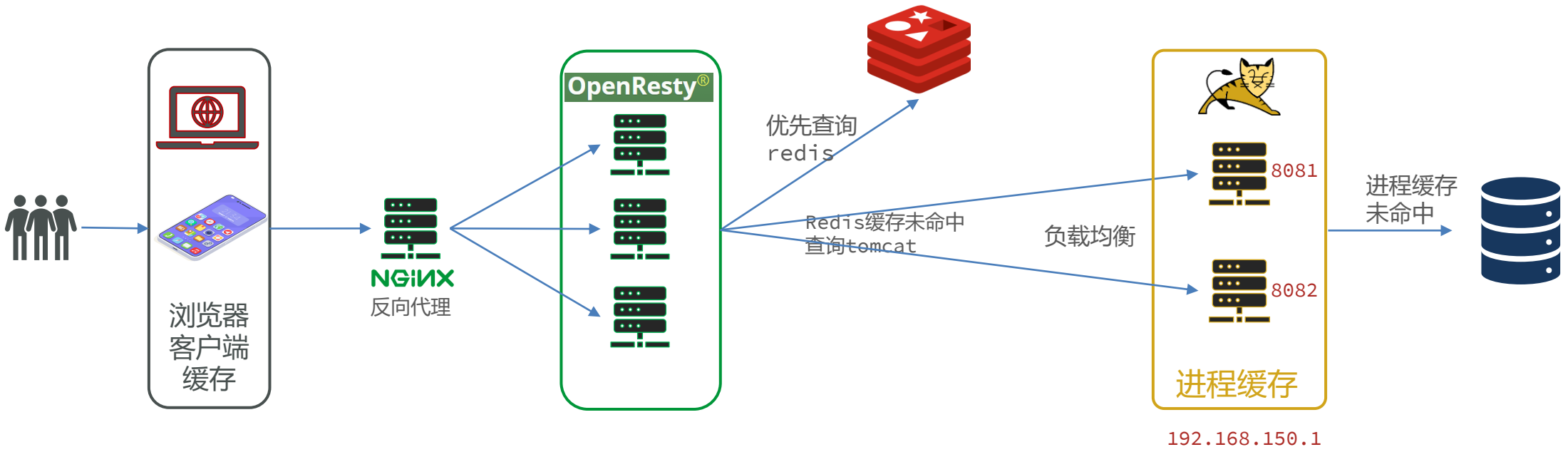

```
# 反向代理配置，将/item路径的请求代理到tomcat集群
location /item {
    proxy_pass  http://tomcat-cluster;
}
```

```
# tomcat集群配置
upstream tomcat-cluster{
    hash $request_uri;
    server 192.168.150.1:8081;
    server 192.168.150.1:8082;
}
```

目录

Contents

添加redis缓存的需求



冷启动与缓存预热

冷启动：服务刚刚启动时，Redis中并没有缓存，如果所有商品数据都在第一次查询时添加缓存，可能会给数据库带来较大压力。

缓存预热：在实际开发中，我们可以利用大数据统计用户访问的热点数据，在项目启动时将这些热点数据提前查询并保存到Redis中。

我们数据量较少，可以在启动时将所有数据都放入缓存中。

缓存预热

1. 利用Docker安装Redis

```
docker run --name redis -p 6379:6379 -d redis redis-server --appendonly yes
```

2. 在item-service服务中引入Redis依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

3. 配置Redis地址

```
spring:
  redis:
    host: 192.168.150.101
```

4. 编写初始化类

```
@Component
public class RedisHandler implements InitializingBean {
    @Autowired
    private StringRedisTemplate redisTemplate;
    @Override
    public void afterPropertiesSet() throws Exception { // 初始化缓存 ... }
}
```

目录

Contents

OpenResty的Redis模块

OpenResty提供了操作Redis的模块，我们只要引入该模块就能直接使用：

- 引入Redis模块，并初始化Redis对象

```
-- 引入redis模块
local redis = require("resty.redis")
-- 初始化Redis对象
local red = redis:new()
-- 设置Redis超时时间
red:set_timeouts(1000, 1000, 1000)
```

- 封装函数，用来释放Redis连接，其实是放入连接池

```
-- 关闭redis连接的工具方法，其实是放入连接池
local function close_redis(red)
    local pool_max_idle_time = 10000 -- 连接的空闲时间，单位是毫秒
    local pool_size = 100 --连接池大小
    local ok, err = red:set_keepalive(pool_max_idle_time, pool_size)
    if not ok then
        ngx.log(ngx.ERR, "放入Redis连接池失败: ", err)
    end
end
```

OpenResty的Redis模块

OpenResty提供了操作Redis的模块，我们只要引入该模块就能直接使用：

- 封装函数，从Redis读数据并返回

```
-- 查询redis的方法 ip和port是redis地址，key是查询的key
local function read_redis(ip, port, key)
    -- 获取一个连接
    local ok, err = red:connect(ip, port)
    if not ok then
        ngx.log(ngx.ERR, "连接redis失败 : ", err)
        return nil
    end
    -- 查询redis
    local resp, err = red:get(key)
    -- 查询失败处理
    if not resp then
        ngx.log(ngx.ERR, "查询Redis失败: ", err, ", key = " , key)
    end
    --得到的数据为空处理
    if resp == ngx.null then
        resp = nil
        ngx.log(ngx.ERR, "查询Redis数据为空, key = ", key)
    end
    close_redis(red)
    return resp
end
```

案例 查询商品时，优先Redis缓存查询

需求：

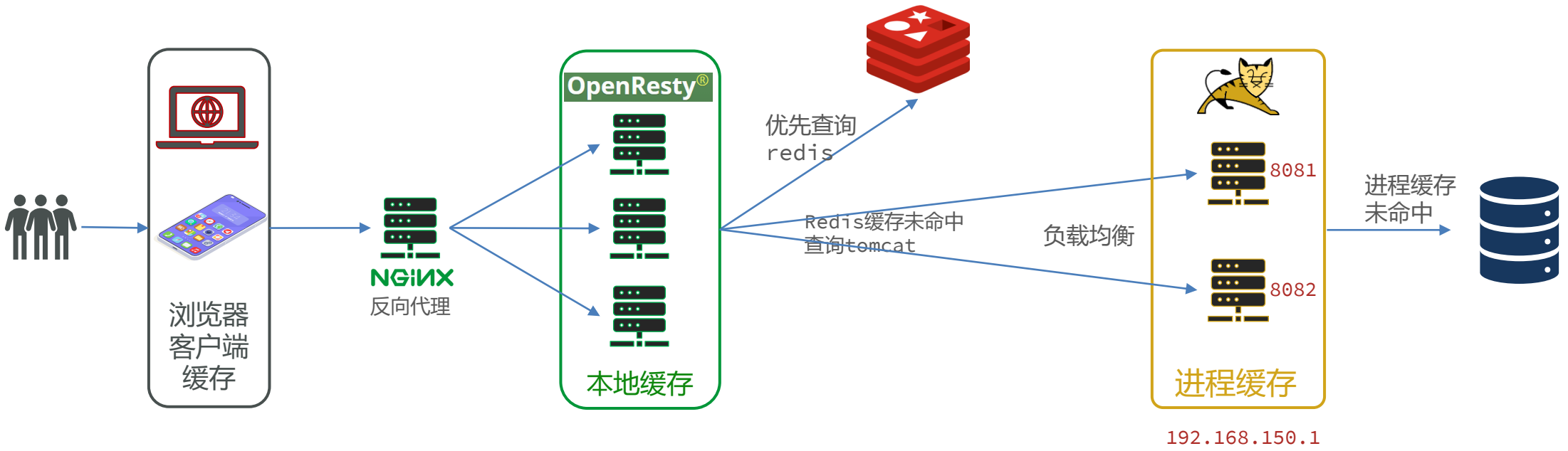
- 修改item.lua，封装一个函数read_data，实现先查询Redis，如果未命中，再查询tomcat
- 修改item.lua，查询商品和库存时都调用read_data这个函数

```
-- 封装函数，先查询redis，再查询http
local function read_data(key, path, params)
    -- 查询redis
    local resp = read_redis("127.0.0.1", 6379, key)
    -- 判断redis是否命中
    if not resp then
        -- Redis查询失败，查询http
        resp = read_http(path, params)
    end
    return resp
end
```

目录

Contents

Nginx本地缓存需求



nginx本地缓存

OpenResty为Nginx提供了**shard dict**的功能，可以在nginx的多个worker之间共享数据，实现缓存功能。

- 开启共享字典，在nginx.conf的http下添加配置：

```
# 共享字典，也就是本地缓存，名称叫做：item_cache，大小150m
lua_shared_dict item_cache 150m;
```

- 操作共享字典：

```
-- 获取本地缓存对象
local item_cache = ngx.shared.item_cache
-- 存储，指定key、value、过期时间，单位s，默认为0代表永不过期
item_cache:set('key', 'value', 1000)
-- 读取
local val = item_cache:get('key')
```

案例 在查询商品时，优先查询OpenResty的本地缓存

需求：

- 修改item.lua中的read_data函数，优先查询本地缓存，未命中时再查询Redis、Tomcat
- 查询Redis或Tomcat成功后，将数据写入本地缓存，并设置有效期
- 商品基本信息，有效期30分钟
- 库存信息，有效期1分钟

nginx本地缓存

修改后的查询逻辑：

```
-- 封装函数，先查询本地缓存，再查询redis，再查询http
local function read_data(key, expire, path, params)
    -- 读取本地缓存
    local val = item_cache:get(key)
    if not val then
        -- 缓存未命中，记录日志
        ngx.log(ngx.ERR, "本地缓存查询失败，key: ", key , ", 尝试redis查询")
        -- 查询redis
        val = read_redis("127.0.0.1", 6379, key)
        -- 判断redis是否命中
        if not val then
            ngx.log(ngx.ERR, "Redis缓存查询失败，key: ", key , ", 尝试http查询")
            -- Redis查询失败，查询http
            val = read_http(path, params)
        end
    end
    -- 写入本地缓存
    item_cache:set(key, val, expire)
    return val
end
-- 根据id查询商品
local itemJSON = read_data('item:id:' .. id, 1800, "/item/".. id, nil)
-- 根据id查询商品库存
local itemStockJSON = read_data('item:stock:id:' .. id, 60, "/item/stock/".. id, nil)
```

缓存同步

- 数据同步策略
- 安装Canal
- 监听Canal

目录

Contents

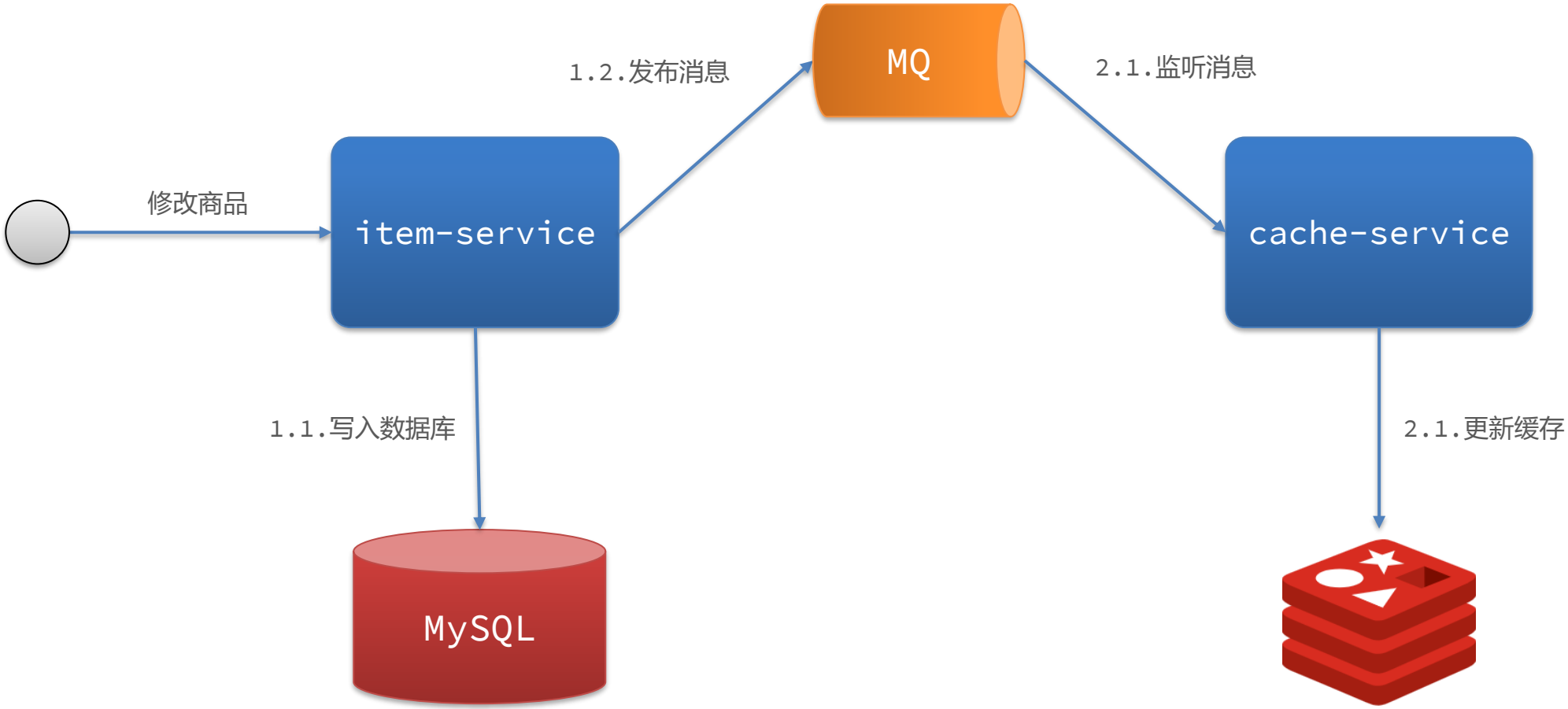
缓存同步策略

缓存数据同步的常见方式有三种：

- **设置有效期**：给缓存设置有效期，到期后自动删除。再次查询时更新
  - 优势：简单、方便
  - 缺点：时效性差，缓存过期之前可能不一致
  - 场景：更新频率较低，时效性要求低的业务
- **同步双写**：在修改数据库的同时，直接修改缓存
  - 优势：时效性强，缓存与数据库强一致
  - 缺点：有代码侵入，耦合度高；
  - 场景：对一致性、时效性要求较高的缓存数据
- **异步通知**：修改数据库时发送事件通知，相关服务监听到通知后修改缓存数据
  - 优势：低耦合，可以同时通知多个缓存服务
  - 缺点：时效性一般，可能存在中间不一致状态
  - 场景：时效性要求一般，有多个服务需要同步

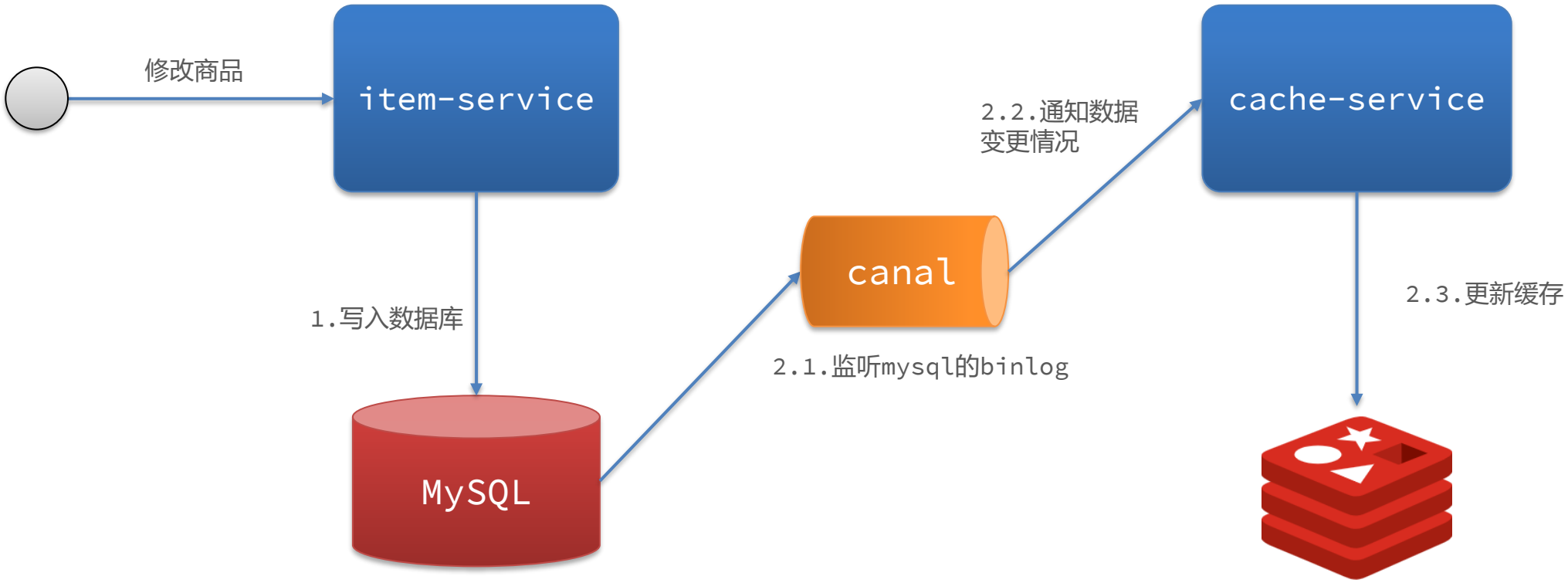
缓存同步策略

基于MQ的异步通知：



缓存同步策略

基于Canal的异步通知：



目录

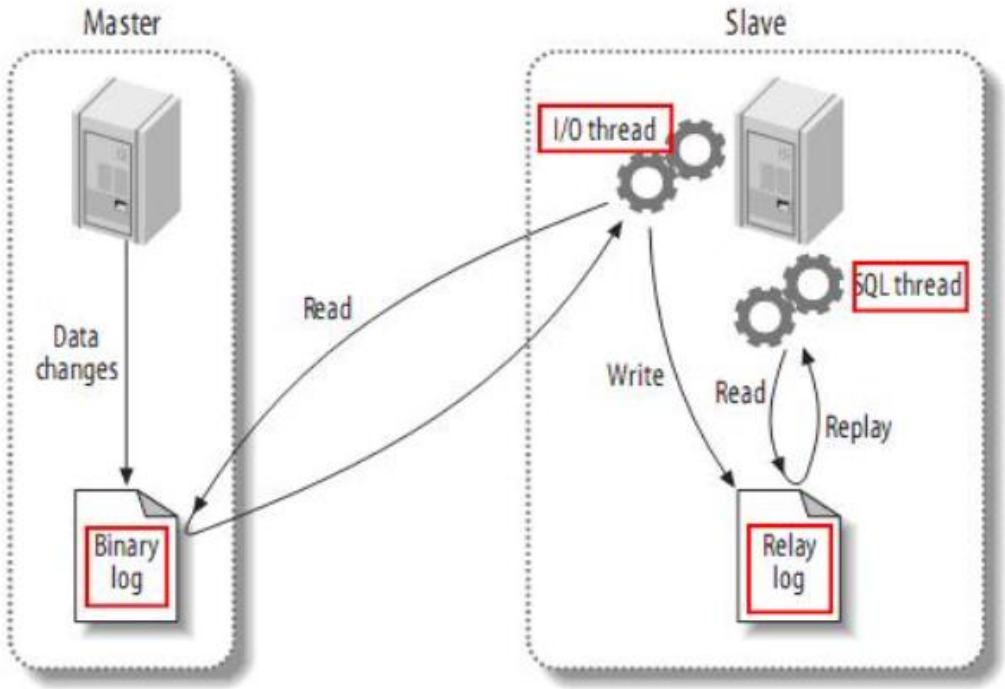
Contents

初识Canal

Canal [kə'næl]，译意为水道/管道/沟渠，canal是阿里巴巴旗下的一款开源项目，基于Java开发。基于数据库增量日志解析，提供增量数据订阅&消费。GitHub的地址：https://github.com/alibaba/canal

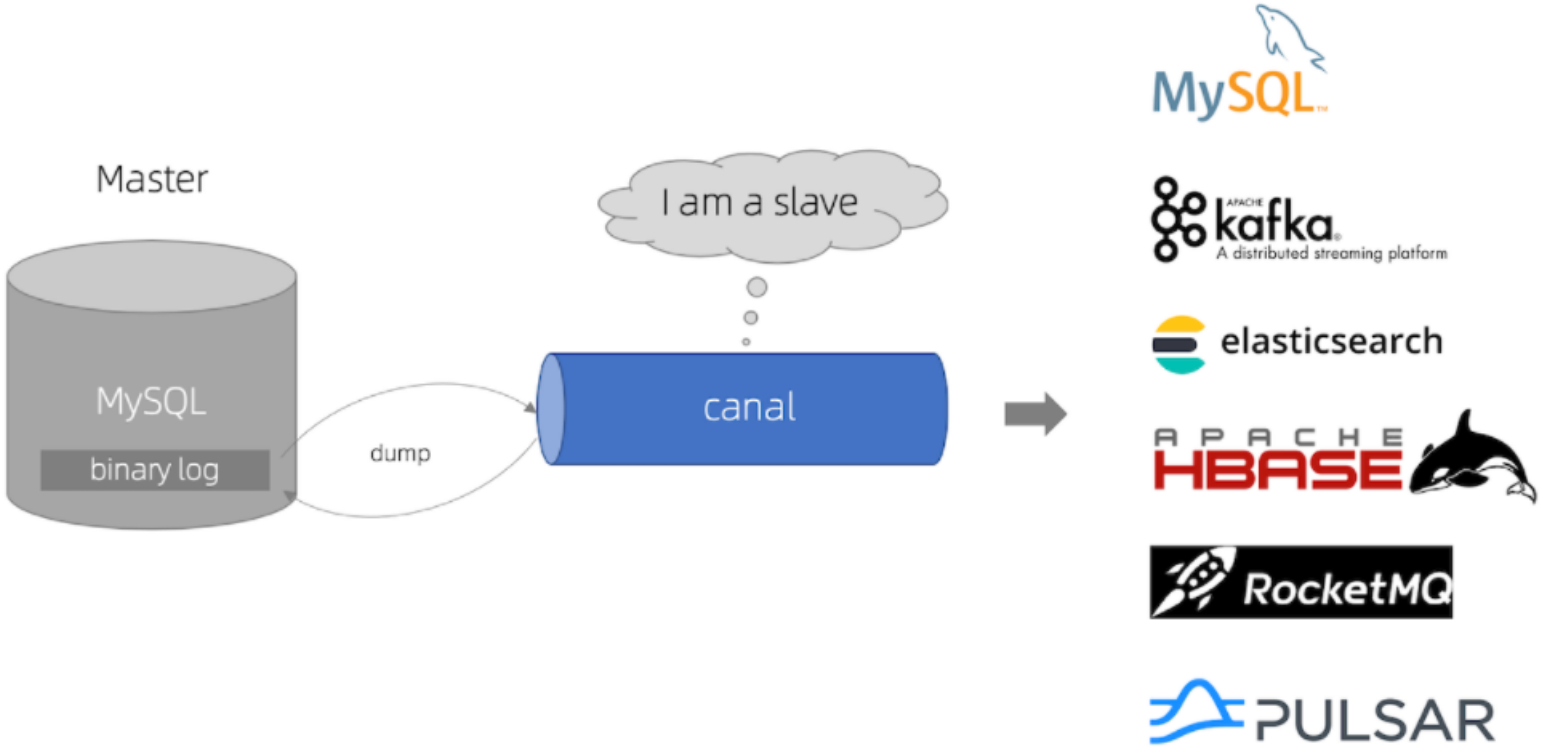
Canal是基于mysql的主从同步来实现的，MySQL主从同步的原理如下：

- MySQL master 将数据变更写入二进制日志(binary log)，其中记录的数据叫做binary log events
- MySQL slave 将 master 的 binary log events拷贝到它的中继日志(relay log)
- MySQL slave 重放 relay log 中事件，将数据变更反映它自己的数据



初识Canal

Canal就是把自己伪装成MySQL的一个slave节点，从而监听master的binary log变化。再把得到的变化信息通知给Canal的客户端，进而完成对其它数据库的同步。



安装和配置Canal

安装和配置Canal参考课前资料文档：

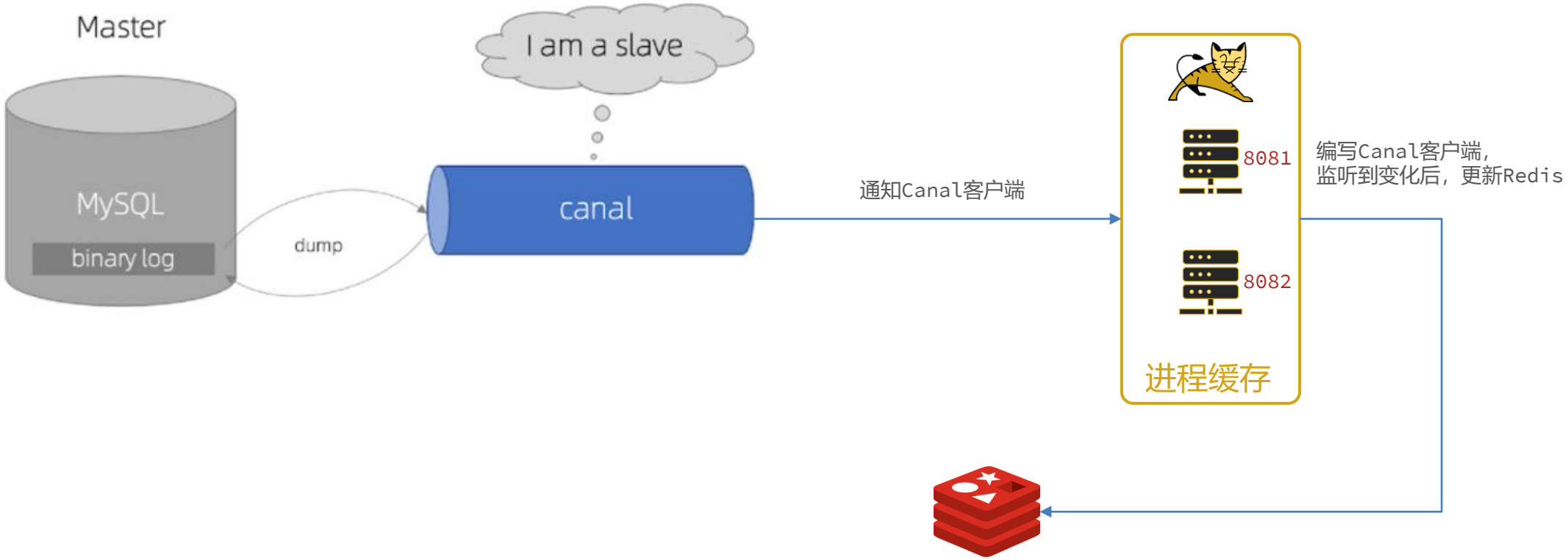
安装Canal.md

目录

Contents

Canal客户端

Canal提供了各种语言的客户端，当Canal监听到binlog变化时，会通知Canal的客户端。



Canal客户端

Canal提供了各种语言的客户端，当Canal监听到binlog变化时，会通知Canal的客户端。不过这里我们会使用GitHub上的第三方开源的canal-starter。地址：https://github.com/NormanGyllenhaal/canal-client

引入依赖：

```
<!--canal-->
<dependency>
    <groupId>top.javatool</groupId>
    <artifactId>canal-spring-boot-starter</artifactId>
    <version>1.2.1-RELEASE</version>
</dependency>
```

编写配置：

```
canal:
  destination: heima # canal实例名称，要跟canal-server运行时设置的destination一致
  server: 192.168.150.101:11111 # canal地址
```

Canal客户端

编写监听器，监听Canal消息：

```
package com.heima.item.canal;

@CanalTable("tb_item")
@Component
public class ItemHandler implements EntryHandler<Item> {

    @Override
    public void insert(Item item) {
        // 新增数据到redis
    }
    @Override
    public void update(Item before, Item after) {
        // 更新redis数据
        // 更新本地缓存
    }
    @Override
    public void delete(Item item) {
        // 删除redis数据
        // 清理本地缓存
    }
}
```

指定要监听的表

指定表关联的实体类

监听到数据库的增、改、删 的消息

Canal客户端

Canal推送给canal-client的是被修改的这一行数据（row），而我们引入的canal-client则会帮我们把行数据封装到Item实体类中。这个过程中需要知道数据库与实体的映射关系，要用到JPA的几个注解：

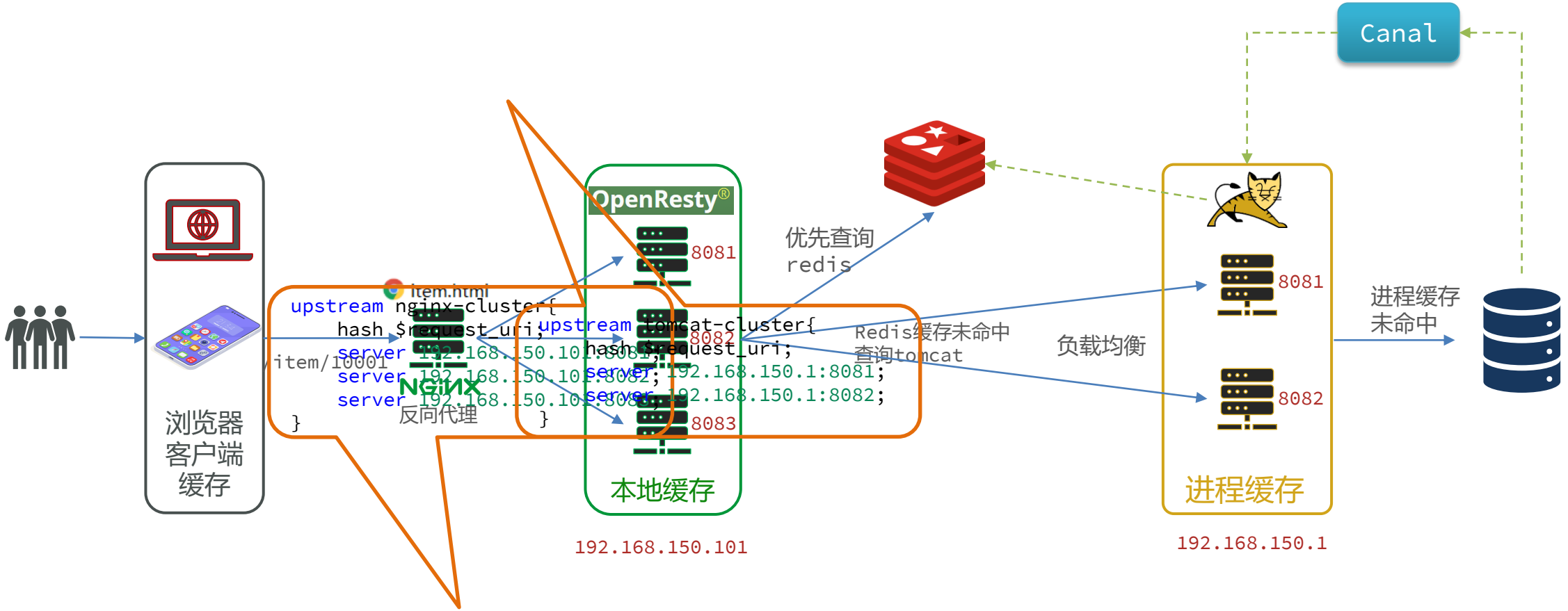
```
@Data
@TableName("tb_item")
public class Item {
    @TableId(type = IdType.AUTO)
    @Id
    private Long id;
    @Column(name = "name")
    private String name;
    // ... 其它字段略
    private Date updateTime;
    @TableField(exist = false)
    @Transient
    private Integer stock;
    @TableField(exist = false)
    @Transient
    private Integer sold;
}
```

标记表中的id字段

标记表中与属性名不一致的字段

标记不属于表中的字段

多级缓存总结


Canal
OpenResty®
8081
8083
优先查询
redis
item.html
/item/10001
upstream nginx-cluster{
hash $request_uri;
server 192.168.150.101:8081;
server 192.168.150.101:8082;
server 192.168.150.101:8083;
}
NGINX
反向代理
upstream tomcat-cluster{
hash $request_uri;
server 192.168.150.1:8081;
server 192.168.150.1:8082;
}
Redis缓存未命中
查询tomcat
负载均衡
8081
8082
进程缓存
未命中
浏览器
客户端
缓存
本地缓存
192.168.150.101
进程缓存
192.168.150.1

传智教育旗下高端IT教育品牌