

第3章 缓存灾难问题解决

- 目标1：缓存穿透解决方案
- 目标2：理解布隆过滤器原理
- 目标3：缓存击穿解决方案
- 目标4：缓存雪崩解决方案
- 目标5：缓存一致性解决方案
- 目标6：RESTful规范安全控制方案

1 缓存穿透实战

有些数据查询频率很高的时候，我们会将数据存入到缓存，用户每次查询直接查询缓存即可，从而提高用户访问数据的效率。

比如获取用户为lisi的抢红包记录，此时如果每次查询数据库效率都很低，我们可以第1次从数据库查询lisi最近的前10条抢红包记录，然后将记录存入到Redis缓存，下次直接查询Redis缓存即可。

每次用户抢红包，谁抢到了红包，我们会将抢到红包的用户信息按照抢红包的金额大小的前100名用户信息公示出去，这里也可以采用这种方式来做。

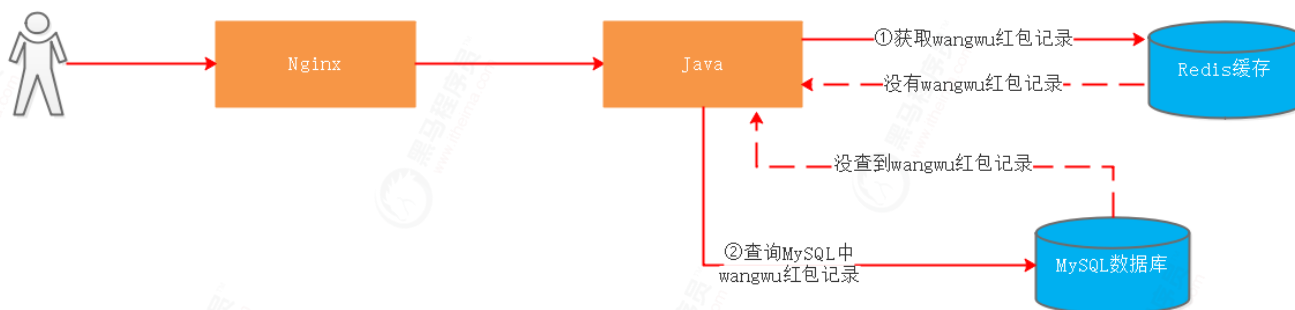
当然，也不是所有数据都适合做缓存，需要根据数据特点来决定，如下图：

数据访问频率	访问频率高	适合缓存	效果好
	访问频率低	不建议缓存	效果不佳
数据读写比例	读多写少	适合缓存	效果好
	读少写多	不建议缓存	效果不佳
数据一致性	一致性要求低	适合缓存	效果好
	一致性要求高	不建议缓存	效果不佳

不是所有数据都适合做缓存
缓存数据需要根据数据操作特点来决定

1.1 缓存穿透介绍

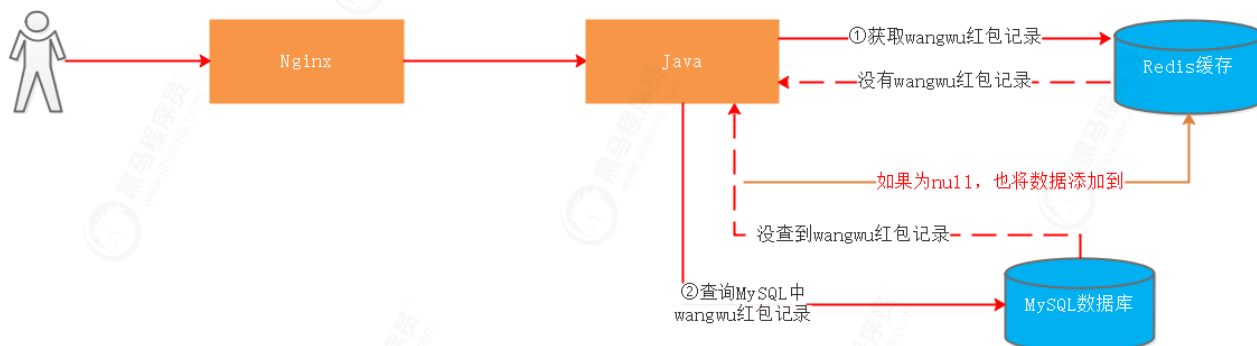
查询wangwu的抢红包记录，如果没有抢红包记录，每次查询都将直接走数据库，这种现象就叫缓存穿透



上面查询最近红包记录提升用户访问效率，这种操作是一种正常操作，但也存在一些非正常操作，比如 wangwu 没有抢到红包，但用户恶意频繁去查询抢红包记录，此时Redis缓存中将一直没有数据，每次都会查询数据库，这种现象叫缓存穿透。

缓存穿透该如何解决呢？我们提供这一种思路，如下图所示：

查询wangwu的抢红包记录，第一次查询如果为null，则往redis缓存中添加一个空对象，下次查询直接返回空，从而避免穿透发生



1.2 穿透问题解决

```

/**
 * 查询用户的抢红包记录
 * @return
 */
@Override
public List<MoneyLog> list() {
    String key = sessionThreadLocal.get().getUsername();
    Object moneyLogs = redisTemplate.boundHashOps(key: "UserMoneyLog").get(key);
    if(moneyLogs!=null){
        return (List<MoneyLog>) moneyLogs;
    }

    //查询数据
    moneyLogs = moneyLogDao.list(sessionThreadLocal.get().getUsername());
    //没有数据，也存空信息到Redis缓存
    redisTemplate.boundHashOps(key: "UserMoneyLog").put(key, moneyLogs==null? "" : moneyLogs);
    return (List<MoneyLog>) moneyLogs;
}

```

①先查Redis缓存

②缓存没有数据，再查询数据库

③有没有数据都将数据存入到Redis缓存

1.3 布隆过滤器

防止缓存穿透，有多种方案，上面所实现的是一种最简单的方案，也有其他方案，比如布隆过滤器也是缓存穿透方案之一。布隆过滤器主要是解决大规模数据下不需要精确过滤的业务场景，如检查垃圾邮件地址，爬虫URL地址去重，解决缓存穿透问题等。

布隆过滤器：在一个存在一定数量的集合中过滤一个对应的数据，判断该数据是否在该集合中。

1.3.1 原理

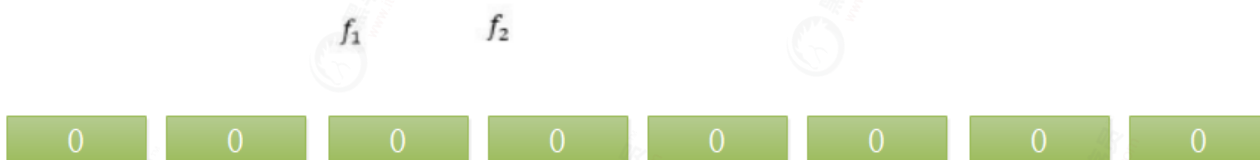
布隆过滤器的巨大用处就是，能够迅速判断一个元素是否在一个集合中。因此他有如下三个使用场景：

1. 网页爬虫对URL的去重，避免爬取相同的URL地址
2. 反垃圾邮件，从数十亿个垃圾邮件列表中判断某邮箱是否垃圾邮箱（同理，垃圾短信）
3. 缓存穿透，将所有可能存在的数据缓存放到布隆过滤器中，当黑客访问不存在的缓存时迅速返回避免缓存及DB挂掉。

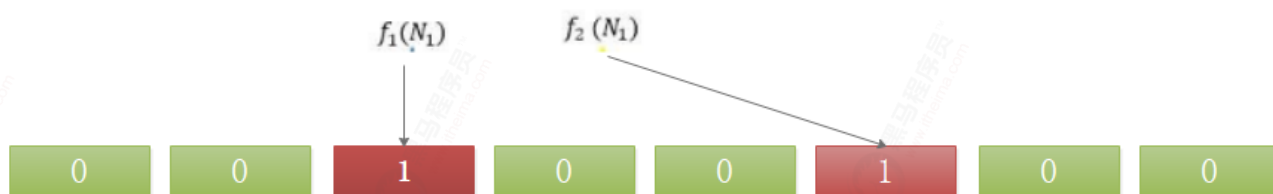
我们来谈谈布隆过滤器的原理

其内部维护一个全为0的bit数组，需要说明的是，布隆过滤器有一个误判率的概念，误判率越低，则数组越长，所占空间越大。误判率越高则数组越小，所占的空间越小。

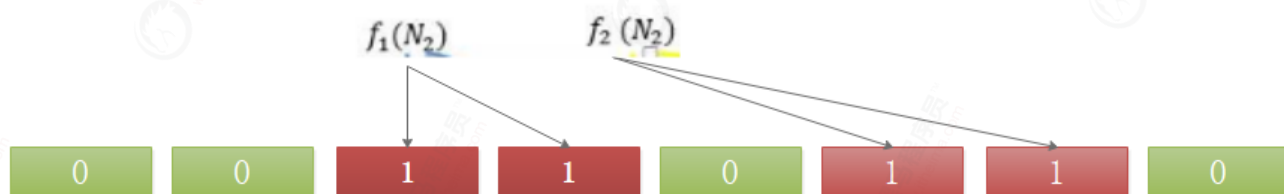
假设，根据误判率，我们生成一个10位的bit数组，以及2个hash函数 (f_1, f_2)，如下图所示(生成的数组的位数和hash函数的数量，我们不用去关心是如何生成的，有数学论文进行过专业的证明)。



假设输入集合为 $((N_1, N_2))$, 经过计算 $(f_1(N_1))$ 得到的数值为2, $(f_2(N_1))$ 得到的数值为5, 则将数组下标为2和下标为5的位置置为1, 如下图所示



同理, 经过计算 $(f_1(N_2))$ 得到的数值为3, $(f_2(N_2))$ 得到的数值为6, 则将数组下标为3和下标为6的位置置为1, 如下图所示



这个时候, 我们有第三个数 (N_3) , 我们判断 (N_3) 在不在集合 $((N_1, N_2))$ 中, 就进行 $(f_1(N_3), f_2(N_3))$ 的计算

1. 若值恰巧都位于上图的红色位置中, 我们则认为, (N_3) 在集合 $((N_1, N_2))$ 中
2. 若值有一个不位于上图的红色位置中, 我们则认为, (N_3) 不在集合 $((N_1, N_2))$ 中

以上就是布隆过滤器的计算原理。

1.3.2 布隆过滤器案例

引入依赖包

```
<!--google的布隆过滤器-->
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>22.0</version>
</dependency>
```

编写测试类

```
public class BloomFilterTest {

    private static int size = 1000000;
    //Google的布隆过滤器
    private static BloomFilter<Integer> bloomFilter
    =BloomFilter.create(Funnels.integerFunnel(), size,0.0001);

    public static void main(String[] args) {
```

```
//放一百万个key到布隆过滤器中 1000000
for (int i = 0; i < size; i++) {
    bloomFilter.put(i);
}

List<Integer> list = new ArrayList<Integer>(1000);
//取10000个不在过滤器里的值, 看看有多少个会被认为在过滤器里
for (int i = size + 10000; i < size + 20000; i++) {
    if (bloomFilter.mightContain(i)) {
        list.add(i);
    }
}
System.out.println("误判的数量: " + list.size());
}
```

我们可以发现有330个被误判,误判的概率为0.03, 源码中也有说明。

```
public static <T> BloomFilter<T> create(Funnel<? super T> funnel, long expectedInsertions) {
    return create(funnel, expectedInsertions, fpp:0.03); // FYI, for 3%, we always get 5 hash functions
}
```

这里的误判概率是可以调整的, 每次创建 BloomFilter 的时候, 指定误判概率值即可, 这个值必须大于0。

```
//Google的布隆过滤器
private static BloomFilter<Integer> bloomFilter =BloomFilter.create(Funnels.integerFunnel(), size, fpp:0.01);
```

优点

1. 思路简单
2. 保证一致性
3. 性能强

缺点

1. 代码复杂度增大
2. 需要另外维护一个集合来存放缓存的Key
3. 布隆过滤器不支持删值操作

2 缓存击穿实战

上面案例我们实现了某个用户抢红包的信息查询, 接下来我们实现公示抢到红包并且按照红包金额大小排序查询出前100名用户信息, 这块数据并发量将更大, 我们需要做缓存处理。

2.1 抢红包排行查询

Controller

```
@RestController
@RequestMapping(value = "/mlog")
public class MoneyLogController {

    @Autowired
    private MoneyLogService moneyLogService;

    /**
     * 抢红包列表
     */
    @GetMapping(value = "/top")
    public List<MoneyLog> top() {
        return moneyLogService.top();
    }
}
```

/mlog/top

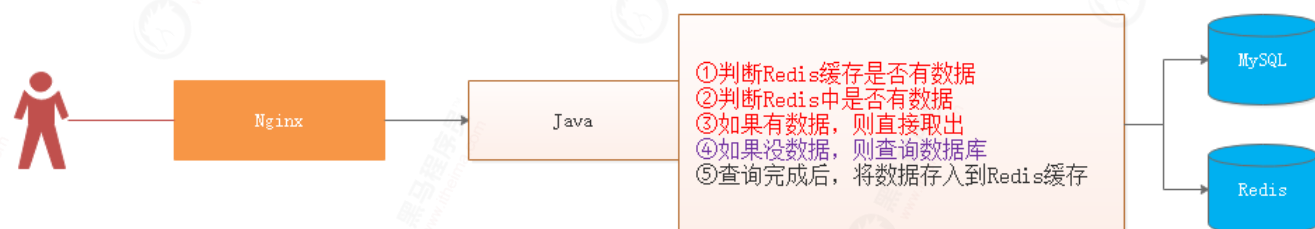
Service 这里做了缓存操作，缓存1分钟，1分钟过后，会再次查询数据库

```
/**
 * 抢红包额度排名
 * @return
 */
@Override
public List<MoneyLog> top() {
    //检查缓存中是否有数据，如果有，直接返回
    String key = "MoneyLogTop100";
    List<MoneyLog> mlogs = (List<MoneyLog>) redisTemplate.boundValueOps(key).get();
    if(mlogs==null){
        //如果没有数据，查询缓存，再将数据存入到Redis，并缓存1分钟
        mlogs = moneyLogDao.top();
        if(mlogs!=null && mlogs.size()>0){
            redisTemplate.boundValueOps(key).set(mlogs);
            redisTemplate.boundValueOps(key).expire(1L, TimeUnit.MINUTES);
        }
    }
    return mlogs;
}
```

测试结果如下：

```
JSON :
[
  {
    "id": "27a66c8c-9358-4b04-8eb0-823cdab82e17",
    "money": 1352,
    "createtime": "2020-07-31T02:06:10.000+0000",
    "username": "lisi"
  },
  {
    "id": "29c40348-554b-4e20-acf9-76d376235107",
    "money": 1269,
    "createtime": "2020-07-31T02:06:11.000+0000",
    "username": "lisi"
  },
  {
    "id": "2a8d3f4c-f4b8-4382-8acd-1a4e1c6b57ff",
    "money": 511,
    "createtime": "2020-07-31T02:06:11.000+0000",
    "username": "lisi"
  },
  {
    "id": "470afae7-95af-4c89-9cf0-ece7cca8be02",
    "money": 133,
    "createtime": "2020-07-31T02:06:11.000+0000",
    "username": "lisi"
  }
]
```

2.2 击穿现象分析



我们先来了解下缓存击穿，缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

上面查询红包排名就存在击穿现象，比如10万用户请求，此时缓存刚好过期，10万用户同时到达了第②个步骤，而且此时Redis中都判断没有数据，那么此时就都查询数据库，给数据库带来巨大的压力，甚至是宕机。

2.3 击穿解决方案

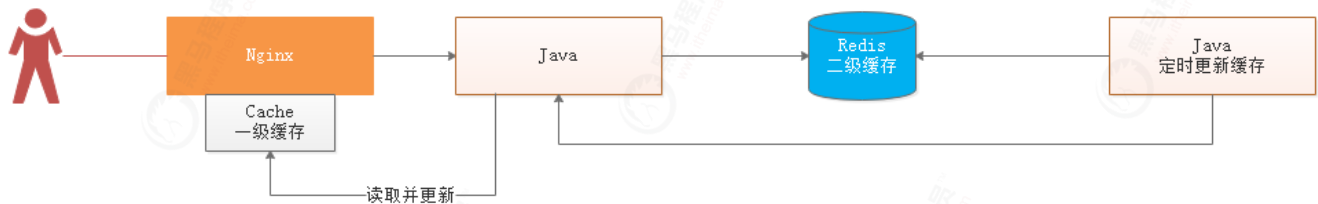
针对缓存击穿现象，可以有多重解决方案。我们这里给大家讲解一下实用的5种方案。

2.3.1 定时器

后台定义一个job(定时任务)专门主动更新缓存数据.比如,一个缓存中的数据过期时间是1分钟,那么job每隔25秒刷新数据(将从数据库中查到的数据更新到缓存中),或者缓存不过期,直接写定时任务定时更新即可。定时器需要择优选择,比如可以用eIastic-job,xxl-job。

这种方案比较容易理解,但会增加系统复杂度。比较适合那些key相对固定,cache粒度较大的业务,key比较分散的则不太适合,实现起来也比较复杂。

2.3.2 多级缓存



采用多级缓存也可以有效防止击穿现象,首先通过程序将缓存存入到Redis缓存,且永不过期,用户查询的时候,先查询Nginx缓存,如果Nginx缓存没有,则查询Redis缓存,并将Redis缓存存入到Nginx一级缓存中,并设置更新时间。这种方案不仅可以提升查询速度,同时又能防止击穿问题,并且提升了程序的抗压能力。

2.3.3 分布式锁

解决上面超卖问题,我们可以采用分布式锁来控制,分布式锁的原理很简单。

分布式锁主要是实现在分布式场景下保证数据的最终一致性。在单进程的系统中,存在多个线程可以同时改变某个变量(可变共享变量)时,就需要对变量或代码块做同步(lock—synchronized),使其在修改这种变量时能够线性执行消除并发修改变量。但分布式系统是多部署、多进程的,开发语言提供的并发处理API在此场景下就无能为力了。

目前市面上分布式锁常见的实现方式有三种:

1. 基于数据库实现分布式锁; (性能效率比较低, 不推荐)
2. 基于缓存 (Redis等) 实现分布式锁; (推荐)
3. 基于Zookeeper实现分布式锁; (推荐)

大部分网站使用的分布式锁是基于缓存的,有更好的性能,而缓存一般是以集群方式部署,保证了高可用性。而Redis分布式锁官方推荐使用redisson。

Redisson分布式锁说明:

- 1、redisson获取锁释放锁的使用和JDK里面的lock很相似,底层的实现采用了类似lock的处理方式
- 2、redisson 依赖redis, 因此使用redisson 锁需要服务端安装redis, 而且redisson 支持单机和集群两种模式下的锁的实现
- 3、redisson 在多线程或者说是分布式环境下实现机制, 其实是通过设置key的方式进行实现, 也就是说多个线程为了抢占同一个锁, 其实就是争抢设置key。

基于Redisson分布式锁实现

步骤:

- 1、引入依赖包
- 2、定义锁的操作方法
 - a. 获取锁的方法
 - b. 解锁
- 3、创建链接Redis集群的配置文件，在里面做相关配置
- 4、RedissonClient，实现锁的操作->获取锁和解锁
- 5、创建工厂对象Redisson的工厂

1)引入依赖

```
<dependency>
  <groupId>org.redisson</groupId>
  <artifactId>redisson-spring-boot-starter</artifactId>
  <version>3.11.0</version>
</dependency>
```

2)锁操作方法实现

要想用到分布式锁，我们就必须要实现获取锁和释放锁，获取锁和释放锁可以编写一个 DistributedLocker 接口，代码如下：

```
public interface DistributedLocker {
    /**
     * lock(), 拿不到lock就不罢休，不然线程就一直block
     * @param lockKey
     * @return
     */
    RLock lock(String lockKey);
    /**
     * timeout为加锁时间，单位为秒
     * @param lockKey
     * @param timeout
     * @return
     */
    RLock lock(String lockKey, long timeout);
    /**
     * timeout为加锁时间，时间单位由unit确定
     * @param lockKey
     * @param unit
     * @param timeout
     * @return
     */
    RLock lock(String lockKey, TimeUnit unit, long timeout);
    /**
     * tryLock(), 马上返回，拿到lock就返回true，不然返回false。
     * 带时间限制的tryLock(), 拿不到lock，就等一段时间，超时返回false。
     */
}
```

```

    * @param lockKey
    * @param unit
    * @param waitTime
    * @param leaseTime 租赁时间, 如果超过指定时间还没解锁, 就强制解锁
    * @return
    */
    boolean tryLock(String lockKey, TimeUnit unit, long waitTime, long leaseTime);
    /**
     * 解锁
     * @param lockKey
     */
    void unlock(String lockKey);
    /**
     * 解锁
     * @param lock
     */
    void unlock(RLock lock);
}

```

实现上面接口中对应的锁管理方法,编写一个锁管理类 `RedissonDistributedLocker`, 代码如下:

```

@Component
public class RedissonDistributedLocker implements DistributedLocker {

    @Autowired
    private RedissonClient redissonClient;

    /**
     * lock(), 拿不到lock就不罢休, 不然线程就一直block
     * @param lockKey
     * @return
     */
    @Override
    public RLock lock(String lockKey) {
        RLock lock = redissonClient.getLock(lockKey);
        lock.lock();
        return lock;
    }

    /**
     * timeout为加锁时间, 单位为秒
     * @param lockKey
     * @param timeout
     * @return
     */
    @Override
    public RLock lock(String lockKey, long timeout) {
        RLock lock = redissonClient.getLock(lockKey);
        lock.lock(timeout, TimeUnit.SECONDS);
        return lock;
    }
}

```

```

/**
 * timeout为加锁时间，时间单位由unit确定
 * @param lockKey
 * @param unit
 * @param timeout
 * @return
 */
@Override
public RLock lock(String lockKey, TimeUnit unit, long timeout) {
    RLock lock = redissonClient.getLock(lockKey);
    lock.lock(timeout, unit);
    return lock;
}

/**
 * tryLock(), 马上返回，拿到lock就返回true，不然返回false。
 * 带时间限制的tryLock(), 拿不到lock，就等一段时间，超时返回false。
 * @param lockKey
 * @param unit
 * @param waitTime
 * @param leaseTime
 * @return
 */
@Override
public boolean tryLock(String lockKey, TimeUnit unit, long waitTime, long leaseTime) {
    RLock lock = redissonClient.getLock(lockKey);
    try {
        return lock.tryLock(waitTime, leaseTime, unit);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return false;
}

/**
 * 解锁
 * @param lockKey
 */
@Override
public void unlock(String lockKey) {
    RLock lock = redissonClient.getLock(lockKey);
    lock.unlock();
}

/**
 * 解锁
 * @param lock
 */
@Override
public void unlock(RLock lock) {
    lock.unlock();
}

```

```
}
```

3)配置Redis链接

在resources下新建文件 `redisson.yml`，主要用于配置redis集群节点链接配置，代码如下：

```
clusterServersConfig:
  # 连接空闲超时，单位：毫秒 默认10000
  idleConnectionTimeout: 10000
  pingTimeout: 1000
  # 同任何节点建立连接时的等待超时。时间单位是毫秒 默认10000
  connectTimeout: 10000
  # 等待节点回复命令的时间。该时间从命令发送成功时开始计时。默认3000
  timeout: 3000
  # 命令失败重试次数
  retryAttempts: 3
  # 命令重试发送时间间隔，单位：毫秒
  retryInterval: 1500
  # 重新连接时间间隔，单位：毫秒
  reconnectionTimeout: 3000
  # 执行失败最大次数
  failedAttempts: 3
  # 密码
  #password: test1234
  # 单个连接最大订阅数量
  subscriptionsPerConnection: 5
  clientName: null
  # loadBalancer 负载均衡算法类的选择
  loadBalancer: !<org.redisson.connection.balancer.RoundRobinLoadBalancer> {}
  # 从节点发布和订阅连接的最小空闲连接数
  slaveSubscriptionConnectionMinimumIdleSize: 1
  # 从节点发布和订阅连接池大小 默认值50
  slaveSubscriptionConnectionPoolSize: 50
  # 从节点最小空闲连接数 默认值32
  slaveConnectionMinimumIdleSize: 32
  # 从节点连接池大小 默认64
  slaveConnectionPoolSize: 64
  # 主节点最小空闲连接数 默认32
  masterConnectionMinimumIdleSize: 32
  # 主节点连接池大小 默认64
  masterConnectionPoolSize: 64
  # 订阅操作的负载均衡模式
  subscriptionMode: SLAVE
  # 只在从服务器读取
  readMode: SLAVE
  # 集群地址
  nodeAddresses:
    - "redis://192.168.211.141:7001"
    - "redis://192.168.211.141:7002"
    - "redis://192.168.211.141:7003"
    - "redis://192.168.211.141:7004"
```

```

- "redis://192.168.211.141:7005"
- "redis://192.168.211.141:7006"
# 对Redis集群节点状态扫描的时间间隔。单位是毫秒。默认1000
scanInterval: 1000
#这个线程池数量被所有RTopic对象监听器，RRemoteService调用者和RExecutorService任务共同共享。默认2
threads: 0
#这个线程池数量是在一个Redisson实例内，被其创建的所有分布式数据类型和服务，以及底层客户端所一同共享的线程池里
保存的线程数量。默认2
nettyThreads: 0
# 编码方式 默认org.redisson.codec.JsonJacksonCodec
codec: !<org.redisson.codec.JsonJacksonCodec> {}
#传输模式
transportMode: NIO
# 分布式锁自动过期时间，防止死锁，默认30000
lockWatchdogTimeout: 30000
# 通过该参数来修改是否按订阅发布消息的接收顺序出来消息，如果选否将对消息实行并行处理，该参数只适用于订阅发布消
息的情况，默认true
keepPubSubOrder: true
# 用来指定高性能引擎的行为。由于该变量值的选用与使用场景息息相关（NORMAL除外）我们建议对每个参数值都进行尝试。
#
#该参数仅限于Redisson PRO版本。
#performanceMode: HIGHER_THROUGHPUT

```

4)创建Redisson管理对象

Redisson管理对象有2个，分别为 `RedissonClient` 和 `RedissonConnectionFactory`，我们只用在项目的 `RedisConfig` 中配置一下这2个对象即可，在 `RedisConfig` 中添加的代码如下：

```

/****
 * Redisson客户端
 * @return
 * @throws IOException
 */
@Bean
public RedissonClient redisson() throws IOException {
    ClassPathResource resource = new ClassPathResource("redisson.yml");
    Config config = Config.fromYAML(resource.getInputStream());
    RedissonClient redisson = Redisson.create(config);
    return redisson;
}

/****
 * Redisson工厂对象
 * @param redisson
 * @return
 */
@Bean
public RedissonConnectionFactory redissonConnectionFactory(RedissonClient redisson) {
    return new RedissonConnectionFactory(redisson);
}

```

5)分布式锁实现

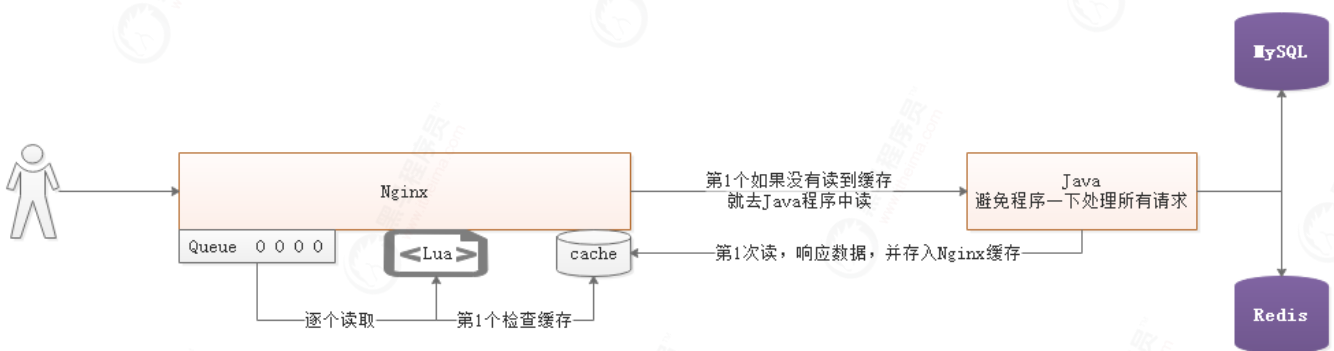
```
@Override
public List<MoneyLog> top() {
    //检查缓存中是否有数据，如果有，直接返回
    String key = "MoneyLogTop100";
    List<MoneyLog> mlogs = (List<MoneyLog>) redisTemplate.boundValueOps(key).get();
    if(mlogs==null){
        //分布式锁
        boolean bo = distributedLocker.tryLock(lockKey: "LOCK_ITHEIMA_LIST", TimeUnit.SECONDS, waitTime: 10L, leaseTime: 10L);
        //获取到了锁
        if(bo){
            //如果没有数据，查询缓存，再将数据存入到Redis，并缓存1分钟
            mlogs = moneyLogDao.top();
            if(mlogs!=null && mlogs.size()>0){
                redisTemplate.boundValueOps(key).set(mlogs);
                redisTemplate.boundValueOps(key).expire(1L, TimeUnit.MINUTES);
            }
            //解锁
            distributedLocker.unlock(lockKey: "LOCK_ITHEIMA_LIST");
        }
    }
    return mlogs;
}
```

2.3.4 队列术



上面我们已经使用过队列术了，队列术在面对零点洪峰流量时，是相当有效，可以直接将所有流量存入到队列中，让后台不用同时处理很多请求，而是从队列中逐个消费逐个处理，上图是实现流程，由于前面抢单已经实现过该流程，所以这里不再重复讲解。

基于Nginx缓存队列术



针对一些特定操作，如果请求并发量极高，我们可以采用Nginx自身的队列术，在上一章我们已经学过了Nginx的代理缓存，其中有一个属性叫 `proxy_cache_lock`，该属性的意思是：当多个客户端请求一个缓存中不存在的文件（或称之为一个MISS），只有这些请求中的第一个被允许发送至服务器。其他请求在第一个请求得到满意结果之后在缓存中得到文件。如果不启用 `proxy_cache_lock`，则所有在缓存中找不到文件的请求都会直接与服务器通信。

`proxy_cache_lock` 的作用其实和队列术及其类似，只不过发生的地方以及处理的语言不同而已。

我们正好可以使用代理缓存来处理一些查询量大的相同数据，例如查询抢红包Top100就可以用Nginx的代理缓存中 `proxy_cache_lock` 来实现。

```
location / {
    #启用缓存openresty_cache
    proxy_cache openresty_cache;
    #针对指定请求缓存
    #proxy_cache_methods GET;
    #设置指定请求会缓存
    proxy_cache_valid 200 304 10s;
    #最少请求1次才会缓存
    proxy_cache_min_uses 3;
    #如果并发请求，只有第1个请求会去服务器获取数据
    proxy_cache_lock on;
    #唯一的key
    proxy_cache_key $host$uri$is_args$args;
    proxy_pass http://myip:18082;
}
```

后台代码移除分布式锁：

```
@Override
public List<MoneyLog> top() {
    System.out.println("Java后台查找数据！");
    //检查缓存中是否有数据，如果有，直接返回
    String key = "MoneyLogTop100";
    List<MoneyLog> mlogs = moneyLogDao.top();
    return mlogs;
}
```

此时只有第1次从后台获取数据，当然这里会请求3次后才会从缓存拿数据，因为有个属性 `proxy_cache_min_uses 3` 属性。

3 缓存雪崩解决方案

缓存雪崩介绍

缓存雪崩是指，由于缓存层承载着大量请求，有效的保护了存储层，但是如果缓存层由于某些原因整体不能提供服务，于是所有的请求都会达到存储层，存储层的调用量会暴增，造成存储层也会挂掉的情况。

解决方案

1)缓存高可用

即使个别节点、个别机器、甚至是机房宕掉，依然可以提供服务，比如 Redis Sentinel 和 Redis Cluster 都实现了高可用。

2)限流

微服务网关或者Nginx做好限流操作，防止大量请求直接进入后端，使后端载荷过重最后宕机。

3)数据预热

预先去更新缓存，再即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀，不要同时失效。

4)队列术限流

使用Nginx队列或者MQ队列，缓存用户的请求，让所有相同操作只有1次查询数据库，并将查询的数据加入到缓存中，下次查询直接从缓存中获取数据。

5)加锁

数据操作，如果是带有缓存查询的，均使用分布式锁，防止大量请求直接操作数据库。

6)多级缓存（推荐）

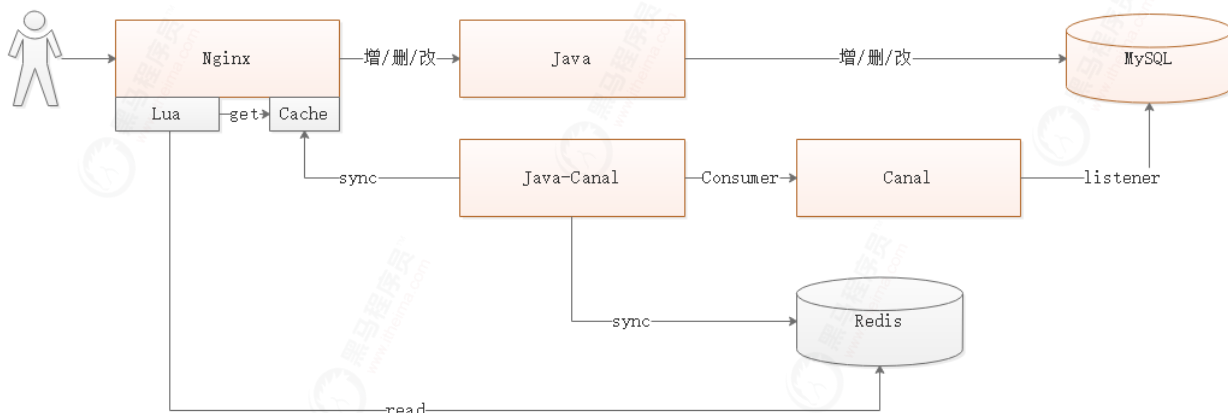
采用多级缓存，Nginx+Redis+MyBatis二级缓存，当Nginx缓存失效时，查找Redis缓存，Redis缓存失效查找MyBatis二级缓存。

4 缓存一致性

用户每次抢完红包，要查看自己抢红包记录，此时需要查询数据库表 money_log ,如果每次都查询 money_log 就会占用大量数据库资源。此时我们应该将数据存储到缓存中，每次查询直接从缓存获取即可。

但现在面临的问题是如果用户抢到了不同的红包，缓存没法及时更新，因此我们需要实现抢红包数据库数据和Redis缓存中的数据同步。

4.1 缓存一致性解决方案



用户每次操作数据库的时候，使用Canal监听数据库指定表的增量变化，在Java程序中消费Canal监听到的增量变化，并在Java程序中实现对Redis缓存或者Nginx缓存的更新。

用户查询的时候，先通过Lua查询Nginx的缓存，如果Nginx缓存没有数据，则查询Redis缓存，Redis缓存如果也没有数据，可以去数据库查询。

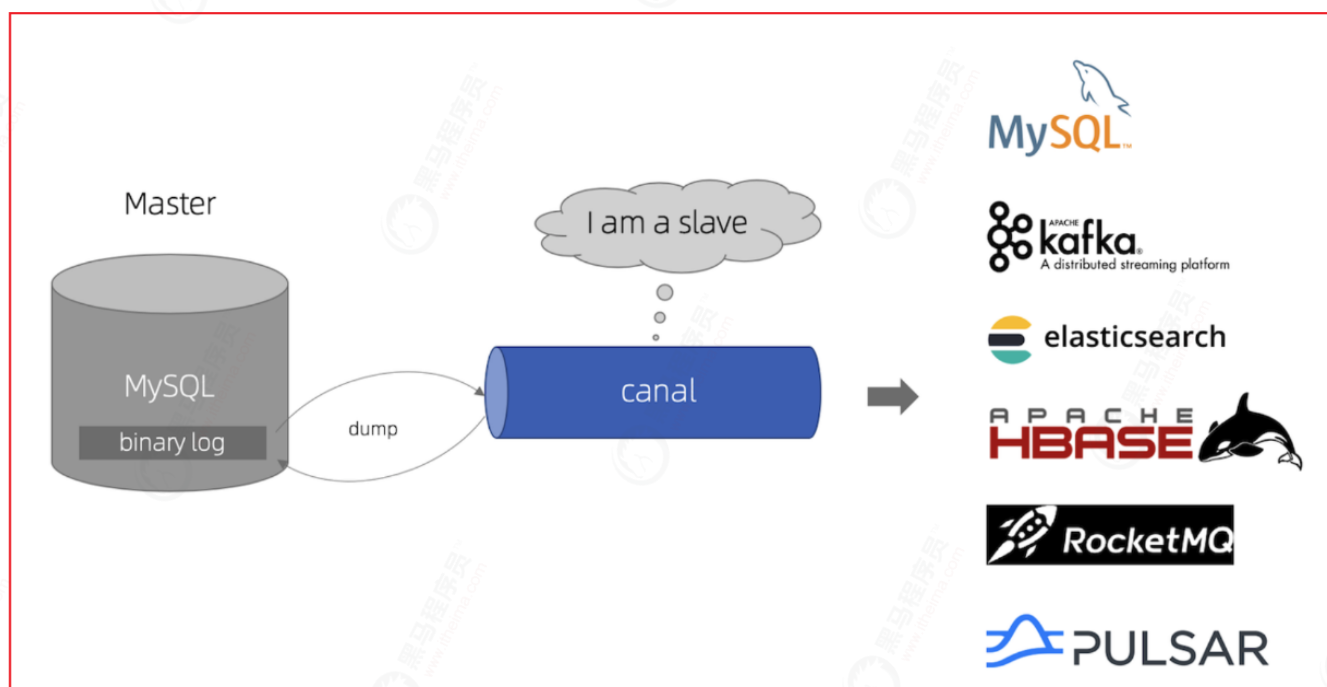
4.2 Canal介绍

Canal主要用途是基于 MySQL 数据库增量日志解析，并能提供增量数据订阅和消费，应用场景十分丰富。

github地址: <https://github.com/alibaba/canal>

版本下载地址: <https://github.com/alibaba/canal/releases>

文档地址: <https://github.com/alibaba/canal/wiki/Docker-QuickStart>



Canal应用场景

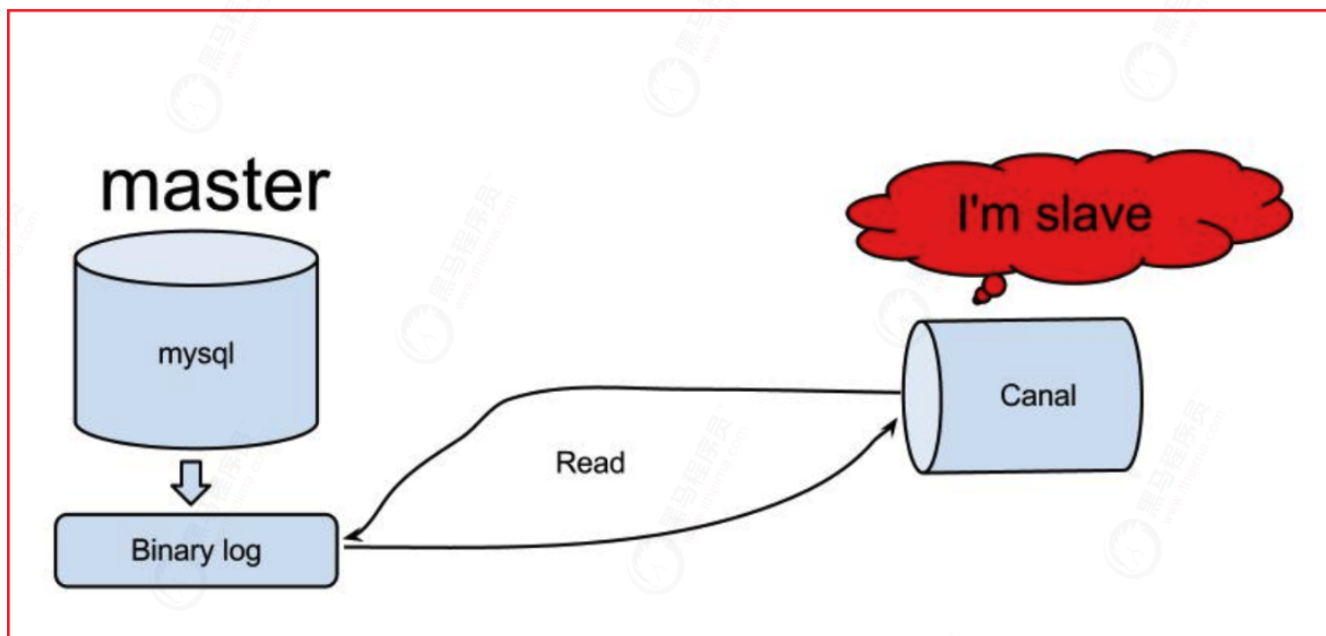
1.电商场景下商品实时更新同步到至Elasticsearch、solr等搜索引擎； 2.价格、库存发生变更实时同步到redis； 3.数据库异地备份、数据同步； 4.代替使用轮询数据库方式来监控数据库变更，有效改善轮询耗费数据库资源。

MySQL主从复制原理

1. MySQL master 将数据变更写入二进制日志(binary log , 其中记录叫做二进制日志事件 binary log events , 可以通过 show binlog events 进行查看) 2. MySQL slave 将 master 的 binary log events 拷贝到它的中继日志(relay log) 3. MySQL slave 重放 relay log 中事件，将数据变更反映它自己的数据

Canal工作原理

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



4.3 Canal配置

1)开启MySQL的bin-log

开启MySQL的binlog日志功能:

```
cd /etc/mysql/mysql.conf.d
```

在mysqld.cnf最下面添加如下配置

```
# 开启 binlog
```

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

```
# 选择 ROW 模式
```

```
binlog-format=ROW
```

```
# 配置 MySQL replaction 需要定义, 不要和 canal 的 slaveId 重复
```

```
server-id=12345
```

2)Canal安装

这里采用容器安装

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

配置CanalServer

修改 /home/admin/canal-server/conf/canal.properties ,将它的id属性修改成和mysql数据库中server-id不同的值, 如下图:

```
#####
#####      common argument      #####
#####
#canal.manager.jdbc.url=jdbc:mysql://127.0.0.1:3306/canal_manager?useUnicode=true&characterEncoding=UTF-8
#canal.manager.jdbc.username=root
#canal.manager.jdbc.password=121212
canal.id = 1
# tcp bind ip
canal.ip =
# register ip to zookeeper
canal.register.ip =
canal.port = 11111
canal.metrics.pull.port = 11112
canal.admin.jmx.port = 11113
```

修改 /home/admin/canal-server/conf/example/instance.properties ,配置要监听的数据库服务地址和监听数据变化的数据库以及表, 修改如下:

```
# position info
canal.instance.master.address=172.17.0.4:3306 监听的数据库
canal.instance.master.journal.name=
canal.instance.master.position=
canal.instance.master.timestamp=
canal.instance.master.gtid=
```

```
# table regex
#canal.instance.filter.regex=.*\\.*
canal.instance.filter.regex=redpackage\\.*
# table black regex
canal.instance.filter.black.regex=
```

指定监听数据库表的配置如下 canal.instance.filter.regex :

mysql 数据解析关注的表, Perl正则表达式.

多个正则之间以逗号(,)分隔, 转义符需要双斜杠(\\)

常见例子:

1. 所有表: .* or .*\\.*
2. canal schema下所有表: canal\\.*
3. canal下的以canal打头的表: canal\\.canal.*
4. canal schema下的一张表: canal.test1
5. 多个规则组合使用: canal\\.*,mysql.test1,mysql.test2 (逗号分隔)

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

重启canal:

```
docker restart canal
```

不要忘了MySQL创建账号并授权:

```
create user canal@%' IDENTIFIED by 'canal';
GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT,SUPER ON *.* TO 'canal'@'%';
FLUSH PRIVILEGES;
```

4.4 同步更新Redis缓存

修改 com.itheima.service.impl.MoneyLogServiceImpl , 添加方法 list(String username) , 代码如下:

```

/**
 * 查询用户的抢红包记录
 * @param username
 * @return
 */
@Override
public List<MoneyLog> list(String username) {
    //查询数据
    return moneyLogDao.list(username);
}

```

直接查询数据

创建类 `com.itheima.canal.MoneyLogSync`，实现消费Canal中监听到的增量数据，代码如下：

```

@CanalTable(value = "money_log")
@Component
public class MoneyLogSync implements EntryHandler<MoneyLog> {

    @Autowired
    private MoneyLogService moneyLogService;

    @Autowired
    private RedisTemplate redisTemplate;

    /**
     * 数据增加变更
     * @param moneyLog
     */
    @Override
    public void insert(MoneyLog moneyLog) {
        //查询用户的抢红包列表
        List<MoneyLog> moneyLogs = moneyLogService.list(moneyLog.getUsername());
        //将数据存入到Redis
        redisTemplate.boundHashOps("UserMoneyLog").put(moneyLog.getUsername(), moneyLogs);
    }
}

```

application.yml中配置Canal的地址：

```

#Canal配置
canal:
  server: 192.168.211.141:11111
  destination: example

```

关于微服务中如何消费Canal监听到的数据，参考参考地址：<https://github.com/NormanGyllenhaal/canal-client>

我们可以测试实现抢单，抢单后，数据会自动同步到Redis缓存中。

Key: Size: 4 bytes
lisi
Value: Size: 724 bytes
[{ "@type": "com.itheima.domain.MoneyLog", "createtime": 1596205402000, "id": "418db9cf-fef2-458f-ac12-c399fe606773", "money": 812.00, "username": "lisi" }, { "@type": "com.itheima.domain.MoneyLog", "createtime": 1596205401000, "id": "68775145-2c98-4de3-91f9-a4ca02a923ba", "money": 1303.00, "username": "lisi" }]

4.5 清理Nginx缓存

清理Nginx缓存，可以利用 `purge` 来清理,请求地址: `<http://192.168.211.141/purge/mlog/top>`。

Successful purge
Key : 192.168.211.141/mlog/top Path: /usr/local/openresty/nginx/cache/5/d3/851262c2086a722424c6a05105de5d35
openresty/1.11.2.5

我们现在要用程序访问，这里需要做相关操作，在java代码中访问上面地址即可,在这里大家可以把前面所学的基于Jwt令牌身份安全校验弄进去，实现安全清理缓存，也可以基于Nginx安全配置实现缓存清理操作。

```
@Override
public void insert(MoneyLog moneyLog) {
    //查询用户的抢红包列表
    List<MoneyLog> moneyLogs = moneyLogService.list(moneyLog.getUsername());
    //将数据存入到Redis
    redisTemplate.boundHashOps(key: "UserMoneyLog").put(moneyLog.getUsername(), moneyLogs);

    //清理Nginx缓存
    try {
        restTemplate.getForObject(url: "http://192.168.211.141/purge/mlog/top", String.class);
    } catch (RestClientException e) {
        System.out.println("暂无缓存");
    }
}
```

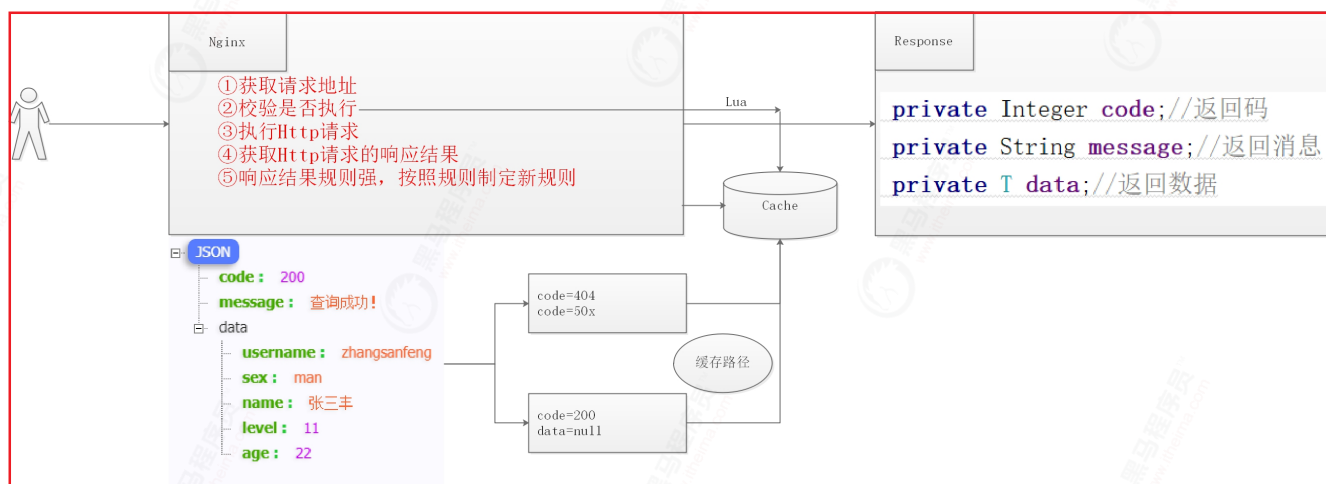
5 RESTful站点安全终极解决方案

当前主流架构都属于微服务架构，主流的开发模式是前后端分离，前后端分离有较强的数据规则，前面针对缓存击穿和穿透解决方案适用于普通项目，不约束项目数据规则，但针对规则如此强的前后端分离开发模式和RESTful的微服务架构，应该是有更优秀的解决方案，不仅限于缓存穿透、缓存击穿问题解决。

解决基于RESTful开发的站点的安全风控解决方案。【缓存穿透】、【缓存击穿】、【缓存雪崩】、【黑白名单】、【定向日志收集】、【防止攻击】、【限流】、【熔断】

5.1 RESTful特性分析

刚才我们做了一个分析，所有前后端分离项目数据规则极强，我们设计了一套如下安全控制解决方案。



我们如果能在Nginx端使用Lua脚本取代Nginx自发执行操作，可以实现很多高级功能。按照上面架构图规则可以过滤很多不安全因素，实现一些高级功能，例如：限流、故障自动切换、缓存击穿、缓存穿透、用户行为收集等，所有能做的事几乎都能做，我们这里拿恶意请求为例，可以制作出如下高质量的功能。

- 1、无效路径收集
- 2、缓存击穿
- 3、队列限流
- 4、无效请求拦截
- 5、黑白名单过滤
- 6、异常熔断降级
- 7、智能切换
-

5.2 功能实现分析



1)无效路径收集

如上图，如果我们能在Nginx执行Http请求，并获取Http请求后的结果，如果请求后的结果是404，证明地址无效，我们可以把该请求地址存储到缓存中，标记为无效标识，下次有相同地址请求，直接跳转到404界面，而不需要请求后端微服务。

2)缓存击穿

定期检测缓存过期时间，一旦要过期，立即将请求做队列限流控制。

3)黑白名单过滤

导入黑白名单IP，每次获取用户IP，检测用户IP是否为黑名单IP，一旦为黑名单IP，则直接拒绝访问。

4)异常熔断降级

收集指定路径返回数据，返回code=500的链接指定时间内频率超过N，则将该路径链接存入缓存，一段时间内禁止用户访问，防止大量错误连接导致服务器宕机。

5.3 Lua执行Http请求

5.3.1 响应接口设计

1)规范接口设计

响应数据必须是规范的数据，我们可以为后端设计一个响应数据封装对象，代码如下：

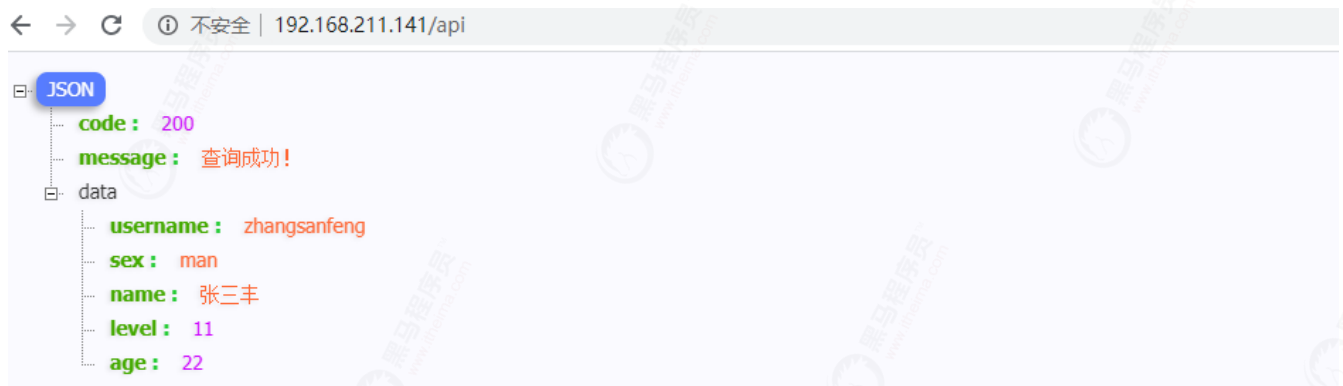
```
public class Result<T> {  
  
    private Integer code;//返回码  
    private String message;//返回消息  
    private T data;//返回数据  
  
    //..get  set  
  
}
```

2)请求测试

创建测试 `RestUserController` , 代码如下:

```
@RestController  
@RequestMapping(value = "/api")  
public class RestUserController {  
  
    @GetMapping(value = "/userinfo/one")  
    public Result<UserInfo> one(){  
        UserInfo userInfo = new UserInfo();  
        userInfo.setAge(22);  
        userInfo.setLevel(11);  
        userInfo.setName("张三丰");  
        userInfo.setSex("man");  
        userInfo.setUsername("zhangsanfeng");  
        return new Result<UserInfo>(200,"查询成功! ", userInfo);  
    }  
}
```

测试: `<http://192.168.211.141/api>`



5.3.2 Lua执行Http请求实现

Lua执行Http请求，需要依赖http模块，Http模块下载地址 <https://github.com/ledgetech/lua-resty-http>，下载该模块后，解压到 `/usr/local/openresty/lua-resty-http-master` 下，并且在nginx的nginx.conf中指定依赖库地址 `lua_package_path "/usr/local/openresty/nginx/lua/?.lua;/usr/local/openresty/lua-resty-http-master/lib/?.lua;/usr/local/openresty/lua-resty-jwt-master/lib/resty?.lua;;"`;

编写lua脚本用于处理用户请求，`resthttp.lua` 脚本如下：

```
--输出json类型数据
ngx.header.content_type="application/json;charset=utf8"

local http = require "resty.http"

local httpc = http.new()

--执行请求
local res, err = httpc:request_uri("http://192.168.0.105:18082/api/userinfo/one", {
    method = "GET",
    body = "a=1&b=2",
    headers = {
        ["Content-Type"] = "application/x-www-form-urlencoded",
    },
    --当前连接保存时间
    keepalive_timeout = 60000,
    --连接池数量
    keepalive_pool = 10
})

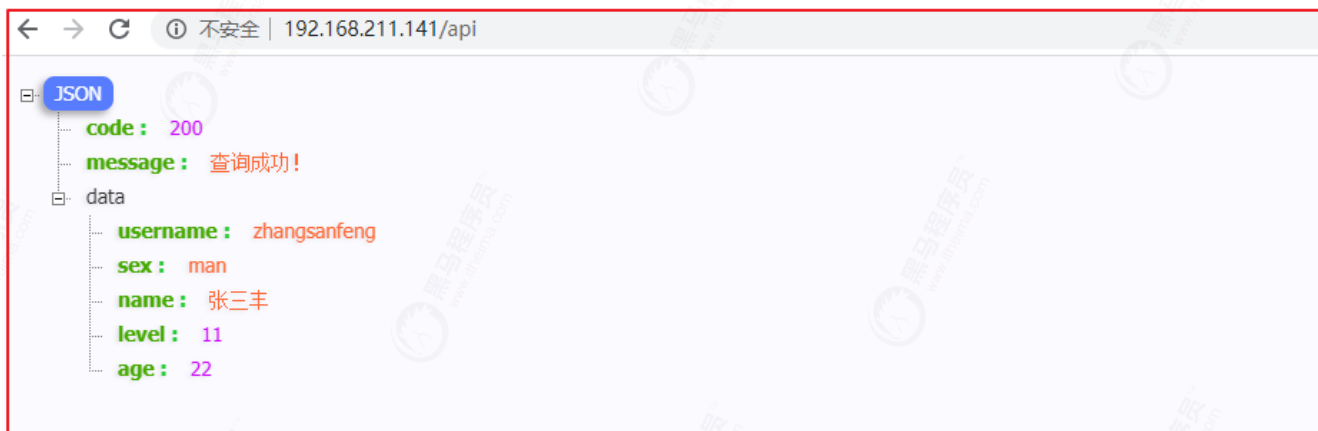
if not res then
    ngx.say("failed to request: ", err)
    return
end

ngx.say(res.body)
```

修改nginx.conf，添加如下配置：

```
#api
location ~ /api {
    content_by_lua_file /usr/local/openresty/nginx/lua/resthttp.lua;
}
```

访问 `<http://192.168.211.141/api>` 效果如下：



此时如果我们在 `resthttp.lua` 织入缓存脚本，可以实现各种条件判断，前面已经实现过相关操作，这里我们就不做演示了。

如果想把地址换成动态的，可以获取用户访问地址，这里提供了一些关于Http参数处理的操作，大家可以参考实现想要的功能。

```
-- 这个变量等于包含一些客户端请求参数的原始URI，它无法修改，请查看$uri更改或重写URI。
local request_uri = ngx.var.request_uri

-- HTTP方法（如http, https）。按需使用，例：
local scheme = ngx.var.scheme
-- 服务器地址，在完成一次系统调用后可以确定这个值，如果要绕开系统调用，则必须在listen中指定地址并且使用bind参数。
local server_addr = ngx.var.server_addr

-- 服务器名称
local server_name = ngx.var.server_name

-- 请求到达服务器的端口号。
local server_port = ngx.var.server_port
local resp = {}
resp["request_uri"]=request_uri
resp["scheme"]=scheme
resp["server_addr"]=server_addr
resp["server_name"]=server_name
resp["server_port"]=server_port
resp["url"]=scheme.."://"..server_addr.." "..server_port..request_uri

--获取完整请求路径
local url= scheme.."://"..server_addr.." "..server_port..request_uri
```