# Redis高级

## AOP缓存实现

### AOP缓存分析

1. 自定义注解Cache\_Find

Key: 1.自动生成key 2.使用用户key

枚举类型: 让用户选择key类型 1.自动 2.自己key

超时时间定义: 单位秒

1. AOP设定

1.环绕通知 控制目标方法执行

2.切入点 拦截注解

### 定义枚举类型

//定义key的类型

**public** **enum** KEY\_ENUM {

***AUTO***,

//使用用户自己的key

//第一个参数 ITEM\_CAT\_0

***EMPTY***; //使用用户自己的key

}

### 自定义注解

//定义一个查询的注解

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***METHOD***) //注解的作用范围

**public** **@interface** Cache\_Find {

String key() **default** "";//接收用户key值

KEY\_ENUM keyType() **default** KEY\_ENUM.***AUTO***;

//定义key类型

**int** secondes() **default** 0; //永不失效

}

### 编辑AOP切面

@Component //将对象交给spring容器管理

@Aspect //标识切面

**public** **class** RedisAspect {

//容器初始化时不需要实例化该对象,

//只有用户使用时才初始化.

//一般工具类中添加该注解

@Autowired(required = **false**)

**private** Jedis jedis;

//使用该方法可以直接获取注解的对象

/\*\*

\* key:value

\* **@param** cache\_find

\* **@return**

\*/

@SuppressWarnings("unchecked")

@Around("@annotation(cache\_find)")

**public** Object around

(ProceedingJoinPoint joinPoint,Cache\_Find cache\_find) {

//1.获取key的值

String key = getKey(joinPoint,cache\_find);

//2.根据key查询缓存

String result = jedis.get(key);

Object data = **null**;

**try** {

**if**(StringUtils.*isEmpty*(result)) {

//如果结果为null,表示缓存中没有数据

//查询数据库

data = joinPoint.proceed(); //表示业务方法执行.

//将数据转化为JSON串

String json =

ObjectMapperUtil.*toJSON*(data);

//判断用户是否设定超时时间

**if**(cache\_find.secondes() == 0)

//表示不要超时

jedis.set(key, json);

**else**

jedis.setex(key, cache\_find.secondes(), json);

System.***out***.println("第一次查询数据库!!!!");

}**else** {

Class targetClass = getClass(joinPoint);

//如果缓存中有数据,则将json串转化为对象返回

data = ObjectMapperUtil.*toObject*(result, targetClass);

System.***out***.println("AOP查询缓存!!!!!!");

}

} **catch** (Throwable e) {

e.printStackTrace();

**throw** **new** RuntimeException(e);

}

**return** data;

}

//获取返回值类型

**private** Class getClass(ProceedingJoinPoint joinPoint) {

MethodSignature signature =

(MethodSignature) joinPoint.getSignature();

**return** signature.getReturnType();

}

/\*\*

\* 1.判断用户key类型 auto empty

\* **@param** joinPoint

\* **@param** cache\_find

\* **@return**

\*/

**private** String getKey

(ProceedingJoinPoint joinPoint, Cache\_Find cache\_find) {

//1.获取key类型

KEY\_ENUM key\_ENUM = cache\_find.keyType();

//2.判断key类型

**if**(key\_ENUM.equals(KEY\_ENUM.***EMPTY***)) {

//表示使用用户自己的key

**return** cache\_find.key();

}

//表示用户的key需要拼接 key+"\_"+第一个参数

String strArgs =

String.*valueOf*(joinPoint.getArgs()[0]);

String key = cache\_find.key()+"\_"+strArgs;

**return** key;

}

}

### 作业自定义缓存更新操作

名称:Cache\_update

业务:当标识该注解,表示进行缓存更新操作.

一般做缓存更新时一般都是删除缓存.

避免不必要的内存开销.

难点: 动态获取key即可.

## Spring整合redis分片

### 编辑properties文件

redis.nodes=192.168.175.129:6379,192.168.175.129:6380,192.168.175.129:6381

### 编辑配置类

//表示redis配置类

@Configuration

@PropertySource("classpath:/properties/redis.properties")

**public** **class** RedisConfig {

@Value("${redis.nodes}")

**private** String redisNodes; //node1,node2,node3

@Bean

**public** ShardedJedis shardedJedis() {

List<JedisShardInfo> shards = **new** ArrayList<>();

//ip:端口,ip:端口

String[] nodes = redisNodes.split(",");

**for** (String node : nodes) {

String host = node.split(":")[0];

**int** port = Integer.*parseInt*(node.split(":")[1]);

JedisShardInfo info =

**new** JedisShardInfo(host, port);

shards.add(info);

}

**return** **new** ShardedJedis(shards);

}

}

## Redis持久化策略

### 说明

Redis中的数据都在内存中,如果断电宕机则内存数据丢失.其中数据应该持久化保存.不允许丢失.

持久化策略:

1. RDB模式
2. AOF模式

### Redis持久化工作原理

说明:按照配置的时间,定期将内存数据保存到redis中的持久化文件中.

当redis服务器宕机之后重启时,首先读取指定的持久化文件,恢复内存数据,方便用户使用.

### RDB模式

概念:

RDB模式是Redis中默认的持久化策略.保存的是redis的内存快照.

占用的资源少.持久化效率最高的.

RDB特点:

1.RDB模式能够定期持久化,但是有丢失数据的风险.

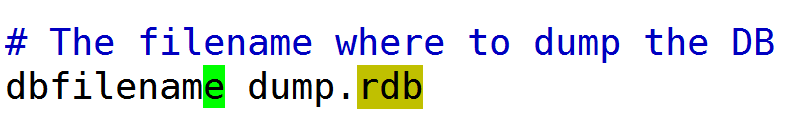
2.Redis中默认的持久化策略

3.RDB模式做内存的快照. 效率高

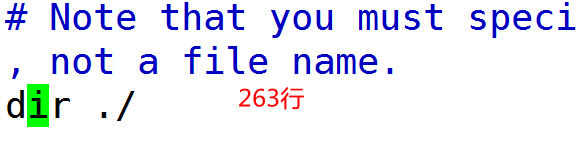
4.占用磁盘空间较小.

### RDB模式的配置

1. 修改持久化文件名称



2.持久化文件目录设定



3.持久化策略

save 900 1 用户在900秒内做1次set操作时持久化1次

save 300 10 用户在300秒内做10次set草时持久化1次

save 60 10000 60秒内做10000次set 持久化一次

使用save持久化化时,如果执行持久化操作,则程序陷入阻塞.用户不能执行set.

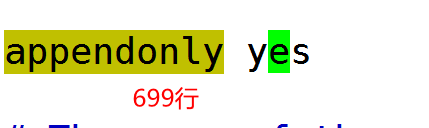
### AOF模式

特点:

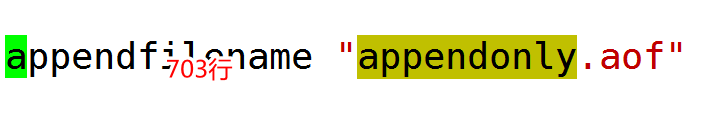
1. AOF模式可以实现数据的实时持久化.
2. 记录的是用户的操作过程.
3. 持久化文件会比较大.
4. 持久化效率低.
5. AOF模式默认是关闭的
6. AOF模式持久化是异步的.

### AOF模式持久化策略

1.开启AOF模式



1. 修改持久化文件名称



1. 持久化策略:

# appendfsync always 只要用户执行”更新”命令则持久化一次

appendfsync everysec 每秒持久化一次.

# appendfsync no 由默认的操作系统决定什么时候持久化.

### Redis缓存策略说明

1. 如果有并发查询时.如果缓存服务器宕机/缓存失效.则查询数据库.可能导致数据库宕机. **俗称缓存雪崩**.
2. 如果用户,**高并发**查询一个**不存在的数据**时.后台数据库有宕机的风险.

**俗称:缓存穿透**

限流 直至封杀IP地址. IP模拟器

1. 如果**高并发条件下**.当某一个热点的key,超时或者失效时.数据库有宕机的风险.

**俗称:缓存击穿**

## Redis内存机制

### 业务需求

Redis中的数据都保存内存中.内存中的数据如果一味的新增,不删除则内存数据很快存满.导致新的数据保存错误.

需求:用户每次都能存储数据,但是内存大小是可控的.要求动态维护内存大小.

### Redis中内存策略

#### LRU算法:

[内存管理](https://baike.baidu.com/item/%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86/5633616)的一种页面置换算法，对于**在内存中但又不用的**[**数据块**](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%9D%97/107672)（内存块）叫做LRU，操作系统会根据哪些数据属于LRU而将其移出内存而腾出空间来加载另外的数据。

#### LFU算法

LFU（least frequently used (LFU) page-replacement algorithm）。即最不经常使用页置换算法，要求在页置换时置换引用计数最小的页，因为经常使用的页应该有一个较大的引用次数。但是有些页在开始时使用次数很多，但以后就不再使用，这类页将会长时间留在内存中，因此可以将引用计数寄存器定时右移一位，形成指数衰减的平均使用次数。

LFU:根据数据使用的次数多少删除数据.

#### 内存优化策略

1. volatile-lru

设定超时时间的数据采用LRU算法删除数据.

1. allkeys-lru

所有的数据采用LRU算法删除数据

1. volatile-lfu

设定了超时时间的数据采用LFU删除数据

1. allkeys-lfu

所有的数据采用LFU算法删除数据

1. volatile-random

设定了超时时间的随机删除

1. allkeys-random

所有key随机删除

1. volatile-ttl

设定了超时时间的数据排序.将马上要超时的数据提前删除.

1. Noeviction

不删除数据.如果内存数据存满则报错返回. 该策略是默认策略

