# 整合实现步骤

# 自动配置的逻辑

### 导入依赖

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

### 提供自动配置属性

根据redis的结构提供属性

单节点redis

spring.redis.host=ip地址

spring.redis.port=端口

哨兵集群

spring.redis.sentinel.master=主从代号

spring.redis.sentinel.nodes=哨兵节点信息 ip1:port1,ip2,port2

集群结构

spring.redis.cluster.nodes=若干个集群节点信息 ip1:port1,ip2:port2

不同结构都共享连接池属性配置

spring.redis.pool.maxActive=最大连接

spring.redis.pool.maxIdle=最大空闲

spring.redis.pool.minIdle=最小空闲

#### 自动配置的逻辑

springboot编写了一个配置类RedisAutoConfiguration.

使用注解

* @Configuration:配置类的标识
* @Conditional衍生注解
* @Bean生成方法返回值对象容器管理StringRedisTemplate

### 客户端对象注入

可以在系统代码中注入springboot帮你自动创建的一个对象StringRedisTemplate.

|  |
| --- |
| public String setOrGet(String key){  //StringRedisTemplate的api方法  //分为基础命令exists expire del等等  //判断key是否在cluster集群存在  Boolean exists = template.hasKey(key);  System.out.println("key在集群是"+(exists?"存在的":"不存在的"));  template.expire("name",500, TimeUnit.SECONDS);  template.getExpire("name",TimeUnit.SECONDS);  //5种类型String hash list set zset  //String  ValueOperations<String, String> StringOps = template.opsForValue();  StringOps.set(key, UUID.randomUUID().toString());  //hash  HashOperations<String, Object, Object> hashOps = template.opsForHash();  //list  ListOperations<String, String> listOps = template.opsForList();  //set opsForSet  //zset opsForZset  return StringOps.get(key);  } |

* 二次封装的意义

直接使用jedis的底层客户端代码实现redis的程序应用,但是对于开发者来讲,接触到越底层的api,学习成本就越高.

# redis应用

### redis解决分布式系统Session共享问题

#### 功能的问题和扩展

登录逻辑是有业务问题的?

* 没有实现登录顶替的功能.

一个用户可以在多个客户端同时登录使用.视频,等网站的vip权限,就要想办法限制同一个用户的使用登录次数.

* 超时问题

登录时间2小时,有时候用户使用系统超过2小时.造成登录状态突然掉线.简单的处理逻辑延长登录时间.时间延长的太长,大量用户不使用系统时,用户状态依然存储.延长时间短,不够用.伸缩性的访问时长.

可以通过续租来解决.

#### 登录顶替问题

业务逻辑：如何实现后面的用户登录同一个账户是,前面的用户登录状态消失?

基本思路：后面登录的用户,登陆过程,只要能把前面登录的用户生成的ticket删除就可以实现顶替了.

基本思路中解决的问题:后面登录的用户怎么能知道前面登录的用户的ticket值是谁?可以通过一个用户名生成key保存系统中唯一一个有效的ticket（实际应用中用token，用户名相同无法解决登陆顶替问题）

|  |
| --- |
| //存在user 先解决登录顶替的问题  //生成一个和userName有关的key  String userLoginKey="user\_login\_"+user.getUserName();  //判断userLoginKey是否存在,存在则说明有相同用户曾经登录过  if(template.hasKey(userLoginKey)){  //说明有人登陆过,相同用户,把上次ticket获取删除  //获取value值就是上次的ticket  String oldTicket=template.opsForValue().get(userLoginKey);  template.delete(oldTicket); |
| //为后续登录顶替我做准备  template.opsForValue().set(userLoginKey,ticket,2,TimeUnit.HOURS); |

#### 超时续租

问题:程序中是突然超时.

解决思路:给程序用户登录状态获取,添加一个续租的逻辑.

业务逻辑:

每次访问用户登录状态,都判断登录剩余时间,一旦剩余时间小于某个固定值 1小时,30分钟,就给重新定义超时时长2小时

|  |
| --- |
| //添加续租逻辑  public String queryTicket(String ticket) {  //查看剩余时间  Long leftTime = template.getExpire(ticket, TimeUnit.SECONDS);  //正整数,-2超时,对一个超时数据设置重新超时2小时不影响删除的情况  if(leftTime>0&&leftTime<60\*60){  //正在使用的状态,已经使用超过1小时  template.expire(ticket,60\*60\*2,TimeUnit.SECONDS);  //userLoginKey记录唯一有效ticket也要重新设置  //EM\_TICKETeeee1590996942130  String userName=ticket.substring(9,(ticket.length()-13));  String userLoginKey="user\_login\_"+userName;  template.expire(userLoginKey,60\*60\*2,TimeUnit.SECONDS);  }  return template.opsForValue().get(ticket);  } |

#### zuul网关的敏感头

通过域名location测试用户登录生成cookie值

[www.easymall.com域名测试,没有看到cookie](http://www.easymall.com域名测试,没有看到cookie) nginx 或者zuul有关.zuul默认对访问时添加的各种头信息cookie就是其中一个表示敏感,过滤掉了.

zuul配置 忽略敏感头

#忽略敏感头

zuul.sensitive-headers=

### 数据缓存

redis缓存功能引入,减少了数据库的访问压力,将大量请求并发负载承受在缓存层.为什么缓存能解决故障高可用的问题才是好缓存---解决缓存雪崩

#### 缓存雪崩

redis或者其他缓存一旦引入,承受大量的并发请求,只要缓存崩溃,会导致大量请求涌入数据库,崩溃,系统崩溃.

如果缓存能够迅速恢复,并且数据也能保证可靠性.雪崩就解决了.

#### 被动缓存

使用数据时才加入到缓存.

用户查询数据库,先判断redis中是否有数据,有就直接使用,没有缓存数据为后续访问提供缓存,先从数据查出来,放到缓存一份.

#### 主动缓存

新增数据时,添加到缓存中

新增商品数据时,不管查询用不用,都会像缓存添加一份,一旦有查询查到这个新数据不需要走被动逻辑.

#### 缓存和数据库一致性问题

**解决思路**：

先主动删除缓存（被动缓存重新读取）

更新数据库

最后被动缓存会因为有人查询商品，然后从数据库读取更新后到内容存放到缓存

理想执行顺序：

1、删除缓存

2、更新数据库

3、判断缓存没有数据

4、读取数据库

5、新增缓存

**问题**：

高并发下,执行顺序未必按照最合理的执行

有可能是：

1 删除缓存 在2还么有来的急执行

3 判断缓存为空

4 读取数据库旧数据

5 存放缓存一份旧数据

2 更新数据库

**解决办法**: 更新时添加缓存锁

**代码实现 ：修改商品上锁 保证别人不会同时修改，同时查询时也会判断**

|  |
| --- |
| @Override **public void** editProduct**(**Product product**) {** //生成一个key 表示锁  String lock=**"product\_"**+product.getProductId**()**+**".lock"**;  //将 key value保存到redis中  //setIfAbsent 表示redis命令中的 set key value NX  //含义是：该key value不存在则存储，否则存储失败  //这样别人在编辑商品时会设置key value ，就会设置失败，所以也不能编辑商品 防止多人同时编辑商品  Boolean aBoolean = **stringRedisTemplate**.opsForValue**()**.setIfAbsent**(**lock, **"")**;  **if (**aBoolean**){**//设置成功 可编辑商品  //删除之前 保存到redis中的旧的商品缓存  **stringRedisTemplate**.delete**("product\_"**+product.getProductId**())**;  //执行修改数据库  **productMapper**.updateProductById**(**product**)**;  *//释放锁* **stringRedisTemplate**.delete***(***lock***)***;  **}else{  throw new** RuntimeException**("其他人正在编辑，请稍后再试")**;  **} }** |

**代码实现 ：**修改被动缓存

原有的逻辑：判断缓存是否有数据,有则直接使用,没有则从数据查询并且添加到缓存一份

现有逻辑：判断有没有该商品的锁

有:说明有人在更新,不操作缓存,直接从数据库获取数据

没有:没有更新,正常执行原有缓存逻辑

|  |
| --- |
| **public** Product queryOneProduct**(**String productId**) {** //#######解决缓存一致性问题  //判断锁是否存在  String lock=**"product\_"**+productId+**".lock"**;  **if (stringRedisTemplate**.hasKey**(**lock**)) {**//有人在编辑数据  //锁存在 所以缓存不一致 直接返回数据库数据  **return productMapper**.selectProductById**(**productId**)**;    **}** String productKey=**"product\_"**+productId;  //读取redis缓存  **try {  if(stringRedisTemplate**.hasKey**(**productKey**)){  return objectMapper**.readValue**(stringRedisTemplate**.opsForValue**()**.get**(**productKey**)**,Product.**class)**;  **}  } catch (**Exception e**) {** e.printStackTrace**()**;  **}** Product product = **productMapper**.selectProductById**(**productId**)**;  **try {** //保存到缓存S  String objectJson = **objectMapper**.writeValueAsString**(**product**)**;  **stringRedisTemplate**.opsForValue**()**.set**(**productKey,objectJson**)**;  **} catch (**JsonProcessingException e**) {** e.printStackTrace**()**;  **}  return** product; **}** |