

初识redis

NoSQL

	SQL	NoSQL
数据结构	结构化(Structured)	非结构化
数据关联	关联的(Relational)	无关联的
查询方式	SQL查询	非SQL
事务特性	ACID	BASE
存储方式	磁盘	内存
扩展性	垂直	水平
使用场景	1) 数据结构固定 2) 相关业务对数据安全性、一致性要求较高	1) 数据结构不固定 2) 对一致性、安全性要求不高 3) 对性能要求

- #1 键值类型 (Redis)
- #2 文档类型 (MongoDB)
- #3 列类型 (HBase)
- #4 Graph类型 (Neo4j)

- 数据结构：SQL可以使用字段、约束保持结构化；NoSQL根据不同类型的数据库有不同的结构
- 数据关联：SQL表之间能够以外键互相关联；NoSQL本身不支持相互关联，需要编写逻辑进行关联。
- 事务特性：

BASE理论是针对大规模分布式系统的三个主要属性的缩写，它代表了基本可用（Basically Available）、软状态（Soft state）、最终一致性（Eventual consistency）。BASE理论与ACID（原子性、一致性、隔离性、持久性）原则相对，更强调系统在分区容错性下的可用性和数据的灵活性。以下是BASE理论的三个组成部分的详细介绍：

1. 基本可用 (Basically Available) :

- 基本可用是指分布式系统在出现故障时，仍能保证提供部分功能服务，而不是完全不可用。这意味着系统可以损失一些非关键的功能或性能来确保核心服务的可用性。

2. 软状态 (Soft state) :

- 软状态允许系统中的数据存在中间状态，这种状态不是必须是严格一致的，并且这个状态可能会持续一段时间而无需立即达到一致性。这与硬状态相反，硬状态要求系统数据必须保持一致性。

3. 最终一致性 (Eventual consistency) :

- 最终一致性表明系统中的所有节点经过一定时间后，能够达到一个一致的状态。这意味着系统不要求所有时刻的数据都是一致的，但随着时间的推移和没有新的更新操作发生，所有的副本最终会同步并达成一致。

BASE理论特别适用于设计需要高可用性的大规模分布式系统，例如云服务、大数据应用等场景，在这些场景中，系统可能无法同时满足传统数据库的ACID特性，尤其是在面临网络分区的情况下。通过采用BASE原则，这些系统可以在一定程度上牺牲强一致性来换取更好的可用性和分区容忍度。这样做的目的是为了在面对故障时，仍然能够为用户提供服务，尽管在某些情况下这些服务可能暂时不完全一致。

- 存储方式：Nosql在内存中，读写性能快。

Redis

Remote Dictionary Server，远程词典服务，是一个基于内存的键值型NoSQL数据库

特征：

- 键值（key-value）型，value支持多种不同数据结构，功能丰富
- 单线程，每个命令具备原子性
- 低延迟，速度快（基于内存、IO多路复用、良好的编码）。
- 支持数据持久化
- 支持主从集群、分片集群
- 支持多语言客户端

启动redis

默认安装路径在/usr/local/bin目录下

该目录以及默认配置到环境变量，因此可以在任意目录下运行这些命令。其中：

- redis-cli：是redis提供的命令行客户端
- redis-server：是redis的服务端启动脚本
- redis-sentinel：是redis的哨兵启动脚本

指定配置启动：

如果要让Redis以后台方式启动，则必须修改Redis配置文件，就在我们之前解压的redis安装包下（/usr/local/src/redis-7.0.15），名字叫redis.conf

我们先将这个配置文件备份一份：`cp redis.conf redis.conf.bck`

然后修改redis.conf文件中的一些配置：

```
# 允许访问的地址，默认是127.0.0.1，会导致只能在本地访问。修改为0.0.0.0则可以在任意IP访问，生产环境不要设置为0.0.0.0
bind 0.0.0.0
# 守护进程，修改为yes后即可后台运行
daemonize yes
# 密码，设置后访问Redis必须输入密码
requirepass 123321
```

Redis的其它常见配置：

```
# 监听的端口
port 6379
# 工作目录, 默认是当前目录, 也就是运行redis-server时的命令, 日志、持久化等文件会保存在这个目录
dir .
# 数据库数量, 设置为1, 代表只使用1个库, 默认有16个库, 编号0~15
databases 1
# 设置redis能够使用的最大内存
maxmemory 512mb
# 日志文件, 默认为空, 不记录日志, 可以指定日志文件名
logfile "redis.log"
```

启动Redis:

```
# 进入redis安装目录
cd /usr/local/src/redis-6.2.6
# 启动
redis-server redis.conf
```

停止服务:

```
# 利用redis-cli来执行 shutdown 命令, 即可停止 Redis 服务,
# 因为之前配置了密码, 因此需要通过 -u 来指定密码
redis-cli -u 123321 shutdown
```

开机自启

我们也可以通过配置来实现开机自启。

首先, 新建一个系统服务文件:

```
vi /etc/systemd/system/redis.service
```

内容如下:

```
[Unit]
Description=redis-server
After=network.target

[Service]
Type=forking
ExecStart=/usr/local/bin/redis-server /usr/local/src/redis-7.0.15/redis.conf
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

然后重载系统服务:

```
systemctl daemon-reload
```

现在，我们可以用下面这组命令来操作redis了：

```
# 启动  
systemctl start redis  
# 停止  
systemctl stop redis  
# 重启  
systemctl restart redis  
# 查看状态  
systemctl status redis
```

执行下面的命令，可以让redis开机自启：

```
systemctl enable redis
```

客户端

命令行客户端

Redis安装完成后就自带了命令行客户端：redis-cli，使用方式如下：

```
redis-cli [options] [commands]
```

其中常见的options有：

- `-h 127.0.0.1`：指定要连接的redis节点的IP地址，默认是127.0.0.1
- `-p 6379`：指定要连接的redis节点的端口，默认是6379
- `-a 123321`：指定redis的访问密码

其中的commands就是Redis的操作命令，例如：

- `ping`：与redis服务端做心跳测试，服务端正常会返回 `pong`

不指定command时，会进入 `redis-cli` 的交互控制台：

```
[root@heima redis-6.2.6]# redis-cli -a 123321      通过-a参数指定密码并连接  
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.  
127.0.0.1:6379>  
[root@heima redis-6.2.6]# redis-cli  
127.0.0.1:6379> auth 123321          进入控制台后，通过auth命令来指定密码  
OK  
127.0.0.1:6379> ping  
PONG  
127.0.0.1:6379> 
```

可以使用 `redis-cli -h 192.168.14.128 -p 6379` 进入，然后用 `AUTH` 密码 登录

Redis命令

Redis是一个key-value的数据库，key一般是String类型，不过value的类型多种多样：

String	hello world
Hash	{name: "Jack", age: 21}
List	[A -> B -> C -> C]
Set	{A, B, C}
SortedSet	{A: 1, B: 2, C: 3}
GEO	{A: (120.3, 30.5)}
BitMap	0110110101110101011
HyperLog	0110110101110101011

基本类型 特殊类型

官方文档: [Commands | Docs](#)

命令行查看: `help @generic`

通用命令

`help [command]` 查看命令的具体用法

`KEYS pattern`：查看符合模板的所有key，支持模糊查询（但是数据量大时效率很低下，又因为redis是单线程的，会造成redis阻塞）。此命令不建议在生产环境设备或主节点上使用

`DEL key`：删除指定key

`EXIST key`：判断key是否存在

`EXPIRE key seconds`：给key设置一个有效期，有效期到期会自动删除key

`TTL key`：查看key的有效期，-1代表永久；-2代表没有这个key

String类型

String可以分为三类：

- string: 普通字符串
- int: 整数类型，可以自增自减
- float: 浮点类型，可以自增自减

不管哪种格式，底层都是字节数组形式存储，只不过编码方式不同（数字可以转成二进制进行存储，普通字符串只能以字节码存储占用内存更多）。字符串类型的最大空间不能超过512M。

`set k v`：添加或修改

`get key`：根据key获取String类型的value

`mset k v k v...`：批量添加或修改多个

`mget k k...`：根据多个key获取多个String类型的value

`incr k`：让整型的value自增1

`incrby k 数字`：整型v自增指定步长，可以为负

`increbyfloat k 数值`：让浮点类型的v自增指定步长，可以为负

`setnx k v`：添加（不能修改），也可写作 `set k v nx`

`setex k 有效期 v`：添加键值对，并指定有效期，也可写作 `set k v ex 有效期`

Key的层级格式

Redis的key允许有多个单词形成层级结构，多个单词之间用'：'隔开，格式如下：

项目名:业务名:类型:id

这个格式并非固定，也可以根据自己的需求来删除或添加词条。

例如我们的项目名称叫 heima，有user和product两种不同类型的数据，我们可以这样定义key：

- ◆ user相关的key: `heima:user:1`
- ◆ product相关的key: `heima:product:1`

如果Value是一个Java对象，例如一个User对象，则可以将对象序列化为JSON字符串后存储：

KEY	VALUE
<code>heima:user:1</code>	<code>{"id":1, "name": "Jack", "age": 21}</code>
<code>heima:product:1</code>	<code>{"id":1, "name": "小米11", "price": 4999}</code>

采用这种命名方式，在图形化客户端中会自动生成包结构（命令行中不会）

Hash类型

Hash类型，也叫散列，其value是一个无序字典，类似于Java中的HashMap结构。

String结构是将对象序列化为JSON字符串后存储，当需要修改对象某个字段时很不方便：

KEY	VALUE
heima:user:1	{name:"Jack", age:21}
heima:user:2	{name:"Rose", age:18}

Hash结构可以将对象中的每个字段独立存储，可以针对单个字段做CRUD：

KEY	VALUE	
	field	value
heima:user:1	name	Jack
	age	21
heima:user:2	name	Rose
	age	18

`hset key field value`：添加或修改value

`hget key field`

`hmset k field value field value...`：设置一个key中的多个value

`hmget k field field...`：获取一个key的多个value

`hgetall k`：获取一个key的所有field和value，key和value交替打印。

`hkeys k`：获取所有field

`hvals k`：获取所有value

`hincrby k f 数值`：value值增长

`hsetnx k field value`：添加一条f-v。前提是field不存在

List类型

Redis中的List类型与Java中的LinkedList类似，可以看做是一个双向链表结构。既可以支持正向检索和也可以支持反向检索。

特征也与LinkedList类似：

- 有序
- 元素可以重复
- 插入和删除快
- 查询速度一般

常用来存储一个有序数据，例如：朋友圈点赞列表，评论列表等。

`lpush k element...`：向列表左侧插入一个或多个元素

`lpop k`：弹出并返回列表左侧第一个元素

`rpush k element...`: 向列表右侧插入一个或多个元素

`rpop key`: 弹出并返回列表右侧第一个元素

`lrange key start end`: 返回范围内所有元素 (下标0开始, 不会弹出)

`blpop` 和 `brpop`: 与`lpop`和`rpop`类似, 只是在没有元素时进行等待, 而不是直接返回nil

Set类型

- 无序
- 元素不可重复
- 查找快
- 支持交集、并集、差集等功能

`sadd k member...`: 向set中添加一个或多个元素

`srem k member...`: 移除

`scard k`: 返回此key中元素的个数

`sismember k member`: 判断一个元素是否存在在set中

`smembers k`: 获取此key的set中所有元素

`sinter k1 k2...` 求交集

`sdiff k1 k2...` 差集

`sunion k1 k2...` 并集

SortedSet类型

Redis的SortedSet是一个可排序的set集合, 与Java中的TreeSet有些类似, 但底层数据结构却差别很大。SortedSet中的每一个元素都带有一个score属性, 可以基于score属性对元素排序, 底层的实现是一个跳表 (SkipList) 加 hash表。

SortedSet具备下列特性:

- 可排序
- 元素不重复
- 查询速度快

因为SortedSet的可排序特性, 经常被用来实现排行榜这样的功能。

`zadd key score member`: 添加一个或多个元素到sortedset, 如果已经存在, 覆盖其score值。添加数据会自动排序

`zrem key member`: 删除sorted set中一个指定元素

`zscore k m`: 获取set中指定元素的score

`zrank k m`: 获取set中指定元素的排名

`zcard k`: 获取元素个数

`zcount k min max`: 统计给定范围内元素个数 (min、max指的是score)

`zincrby k 数字 member`: set中指定元素自增指定步长

`zrange k min max`: 按照score排序后, 获取指定排名范围的元素 (这里的min、max指的是排名, 从0开始)

`zrangebyscore k min max`: 按照score排序后, 获取指定score范围内的元素 (min、max指的是score)

`zdiff、zinter、zunion`: 差交并

以上所有排名都默认是升序

如果要降序, 在 z 后加上 `rev`

Redis的java客户端

Jedis快速入门

首先引入依赖:

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>5.2.0</version>
</dependency>
```

jedis的方法名就跟命令名称相同。

对String类型进行基本操作:

```
public class JedisTest {
    private static Jedis jedis;
    @BeforeEach
    void setUp() {
        //1. 建立连接
        jedis=new Jedis("192.168.14.128",6379);
        //2. 设置密码
        jedis.auth("abc123");
        //3. 选择库
        jedis.select(0);
    }
    @Test
```

```

void testString(){
    //存
    String result = jedis.set("name", "jack");
    System.out.println(result);
    //取
    String name = jedis.get("name");
    System.out.println(name);
}

@AfterEach
void tearDown(){
    if(jedis!=null){
        jedis.close();
    }
}
}

```

hash类型（同样要获取jedis、关闭jedis）：

```

@Test
void testHash(){
    jedis.hset("user:1", "name", "jeak");
    jedis.hset("user:1", "age", "21");
    Map<String, String> map = jedis.hgetAll("user:1");
    System.out.println(map);
}

```

Jedis的连接池

jedis本身是线程不安全的，并且频繁的创建和销毁连接会有性能损耗，因此推荐使用jedis连接池代替jedis的直连方式。

```

public class JedisConnectionFactory {

    private static final JedisPool jedisPool;

    static {
        JedisPoolConfig config = new JedisPoolConfig();
        //最大连接数（空闲+正在使用）
        config.setMaxTotal(8);
        //最大空闲连接
        config.setMaxIdle(8);
        //最小空闲连接
        config.setMinIdle(0);
        //池没有可用连接时，获取连接的线程最大等待时间
        config.setMaxWaitMillis(1000);
        jedisPool = new JedisPool(config, "192.168.14.128", 6379,
                1000, "abc123");
    }

    public static Jedis getJedis() {
        return jedisPool.getResource();
    }
}

```

```
}
```

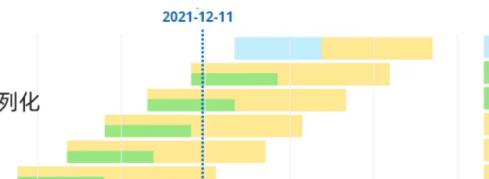
这样，在获取jedis的时候，直接调用 `jedis = jedisConnectionFactory.getJedis();`；使用close方法也是归还jedis，而不是销毁。

SpringDataRedis

SpringDataRedis

SpringData是Spring中数据操作的模块，包含对各种数据库的集成，其中对Redis的集成模块就叫做SpringDataRedis，官网地址：<https://spring.io/projects/spring-data-redis>

- 提供了对不同Redis客户端的整合（Lettuce和Jedis）
- 提供了RedisTemplate统一API来操作Redis
- 支持Redis的发布订阅模型
- 支持Redis哨兵和Redis集群
- 支持基于Lettuce的响应式编程
- 支持基于JDK、JSON、字符串、Spring对象的数据序列化及反序列化
- 支持基于Redis的JDKCollection实现



SpringDataRedis中提供了RedisTemplate工具类，其中封装了各种对Redis的操作。并且将不同数据类型的操作API封装到了不同的类型中：

API	返回值类型	说明
<code>redisTemplate.opsForValue()</code>	<code>ValueOperations</code>	操作String类型数据
<code>redisTemplate.opsForHash()</code>	<code>HashOperations</code>	操作Hash类型数据
<code>redisTemplate.opsForList()</code>	<code>ListOperations</code>	操作List类型数据
<code>redisTemplate.opsForSet()</code>	<code>SetOperations</code>	操作Set类型数据
<code>redisTemplate.opsForZSet()</code>	<code>ZSetOperations</code>	操作SortedSet类型数据
<code>redisTemplate</code>		通用的命令

快速入门

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

```
spring:
  data:
    redis:
      host: 192.168.14.128
      port: 6379
      password: abc123
      #spring默认lettuce, 在这里就是用这个
      lettuce:
        pool:
          max-active: 8
          max-idle: 8
          min-idle: 0
          max-wait: 1000ms
```

```
@Autowired
RedisTemplate redisTemplate;
@Test
void contextLoads() {
    redisTemplate.opsForValue().set("name", "world");
    System.out.println(redisTemplate.opsForValue().get("name"));
}
```

set还有第三第四个参数，分别为有效期及有效期的单位。

-RedisSerializer

Spring Data Redis 的 `RedisTemplate` 默认使用 `JdkSerializationRedisSerializer` 来序列化键和值。使用的是 `ObjectOutputStream`。

这种序列化方式会将字符串或其他对象转换为二进制格式（即字节数组），以便支持复杂的 Java 对象存储。

例如，字符串 `"name"` 被序列化后变成了类似 `"\xac\xed\x00\x05t\x00\x04name"` 的形式。

可以使用的序列化器在 `RedisSerializer` 接口下。也可以通过实现此接口自定义序列化器。

key一般使用`StringRedisSerializer`, value一般用`GenericJackson2JsonRedisSerializer`

设置序列化器：

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
</dependency>
```

如果使用了springmvc就不需要引入了

```
@Configuration
public class RedisConfig {
```

```

@Bean
public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {
    //创建RedisTemplate对象
    RedisTemplate<String, Object> template = new RedisTemplate<>();
    //设置工厂
    template.setConnectionFactory(factory);
    //创建JSON序列化工具并设置
    GenericJackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
    GenericJackson2JsonRedisSerializer();
    //key
    template.setKeySerializer(RedisSerializer.string());
    template.setHashKeySerializer(RedisSerializer.string());
    //value
    template.setValueSerializer(jackson2JsonRedisSerializer);
    template.setHashValueSerializer(jackson2JsonRedisSerializer);
    return template;
}
}

```

再将自动注入的redisTemplate泛型加上 (与上面对应)

```

@.Autowired
RedisTemplate<String, Object> redisTemplate;

```

*StringRedisTemplate

尽管JSON的序列化方式可以满足我们的需求，但依然存在一些问题，如图：



为了在反序列化时知道对象的类型，JSON序列化器会将类的class类型写入json结果中，存入Redis，会带来额外的内存开销。

为了节省内存空间，我们并不会使用JSON序列化器来处理value，而是统一使用String序列化器，要求只能存储String类型的key和value。而当需要存储java对象时，手动完成对象的序列化和反序列化。

Spring提供了一个StringRedisTemplate类，它的key和value序列化方式默认就是String方式。省去了我们自定义RedisTemplate的过程：

```

private static final ObjectMapper mapper=new ObjectMapper();

@Test
void testSaveUser() throws JsonProcessingException {
    User user = new User("world",18);
    String json=mapper.writeValueAsString(user);
    stringRedisTemplate.opsForValue().set("user:200", json);
    String jsonUser=stringRedisTemplate.opsForValue().get("user:200");
    User user1=mapper.readValue(jsonUser,User.class);
    System.out.println(user1);
}

```

也可以把手动序列化反序列化过程封装成工具类。

redisTemplate操作hash类型

```

@Test
void testHash(){
    stringRedisTemplate.opsForHash().put("user:2","name","zhangsan");
    stringRedisTemplate.opsForHash().put("user:2","age","12");
    Map<Object, Object>
entries=stringRedisTemplate.opsForHash().entries("user:2");
    System.out.println("entries="+entries);
}

```

hash的方法名跟命令不同。

将对象采用hash的方式存入redis:

```

Map<String, String> userMap=BeanUtil.beanToMap(userDTO,new HashMap<>(),
CopyOptions.create()
.setIgnoreNullValue(true)
//将对象每个属性值都转为string类型才能用
stringRedisTemplate存储

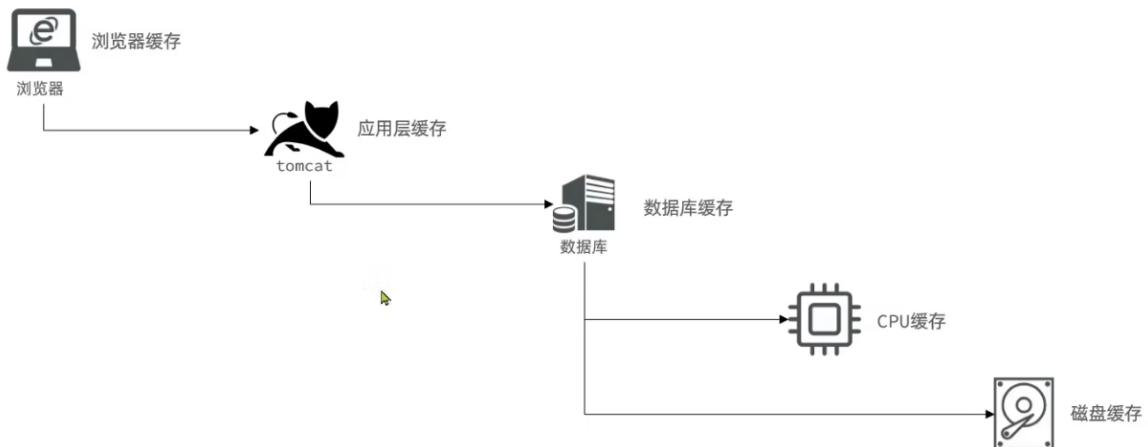
.setFieldValueEditor((fieldName, fieldValue)->fieldValue.toString());
stringRedisTemplate.opsForHash().putAll("login:token:"+token,userMap);

```

实战篇

缓存

缓存就是数据交换的缓冲区（cache），是贮存数据的临时地方，一般读写性能较高。



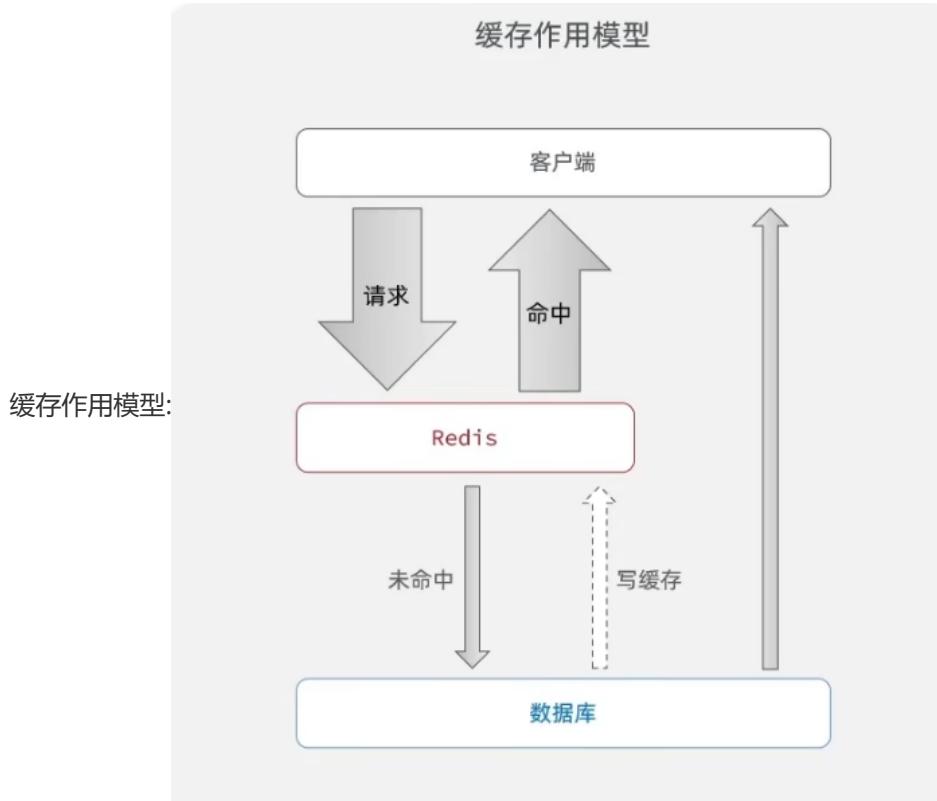
(还不止上图这么多)

缓存的作用：

- 降低后端负载（比如数据库）
- 提高读写效率，降低响应时间

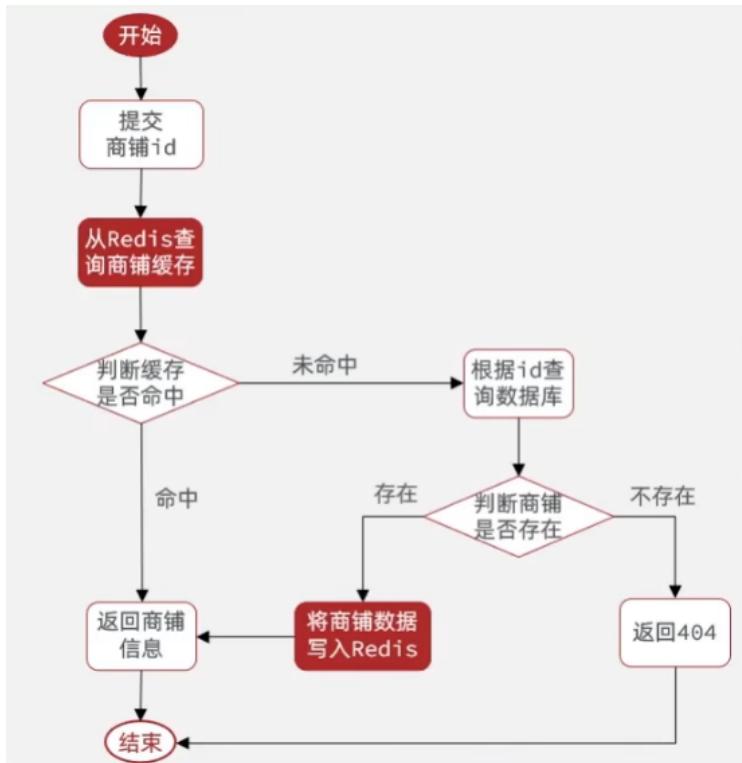
缓存的成本（不足）

- 数据一致性成本
- 代码维护成本
- 运维成本



代码演示

以商户数据查询为例：



一个基本实现：

```
@Service
public class ShopServiceImpl extends ServiceImpl<ShopMapper, Shop> implements IShopService {
    @Autowired
    private StringRedisTemplate stringRedisTemplate;
    @Override
    public Result queryById(Long id) {
        //1.从redis查询商铺缓存
        String shopJson=stringRedisTemplate.opsForValue().get("cache:shop:"+id);
        //2.判断是否存在
        if(StrUtil.isNotBlank(shopJson)){
            //3.存在，直接返回
            Shop shop= JSONUtil.toBean(shopJson, Shop.class);
            return Result.ok(shop);
        }
        //4.不存在，根据id查询数据库
        Shop shop=this.getById(id);
        //5.不存在，返回错误
        if(shop!=null){
            return Result.fail("店铺不存在！");
        }
        //6.存在，写入redis

        stringRedisTemplate.opsForValue().set("cache:shop:"+id,JSONUtil.toJsonStr(shop));
        return Result.ok(shop);
    }
}
```

缓存更新策略（双写一致性）

缓存带来诸多好处的同时，也会带来一些问题，首先就是一致性问题：

在数据库数据更新时，缓存不会及时更新，导致查询到的数据与数据库中的数据不一致。

于是引入缓存更新策略。

	内存淘汰	超时剔除	主动更新
说明	不用自己维护，利用Redis的内存淘汰机制，当内存不足时自动淘汰部分数据。下次查询时更新缓存。	给缓存数据添加TTL时间，到期后自动删除缓存。下次查询时更新缓存。	编写业务逻辑，在修改数据库的同时，更新缓存。 
一致性	差	一般	好
维护成本	无	低	高

业务场景：

- 低一致性需求：使用内存淘汰机制。例如店铺类型的查询缓存
- 高一致性需求：主动更新，并以超时剔除作为兜底方案。例如店铺详情查询的缓存

主动更新策略（问题待解决）

主要的三种策略：



第三种方法效率高（因为每隔一段时间检查缓存有没有变化，所以假如在一段时间内缓存变化了n次，但是只要向数据库更新最后一次就行了）。但是一致性和可靠性较差（如果宕机，内存中的更新数据会丢失）

一般采用第一种方式较多。

采用第一种方式有三个问题需要考虑：

1. 删除缓存还是更新缓存?

- ◆ 更新缓存: 每次更新数据库都更新缓存, 无效写操作较多 X
- ◆ 删除缓存: 更新数据库时让缓存失效, 查询时再更新缓存 ✓

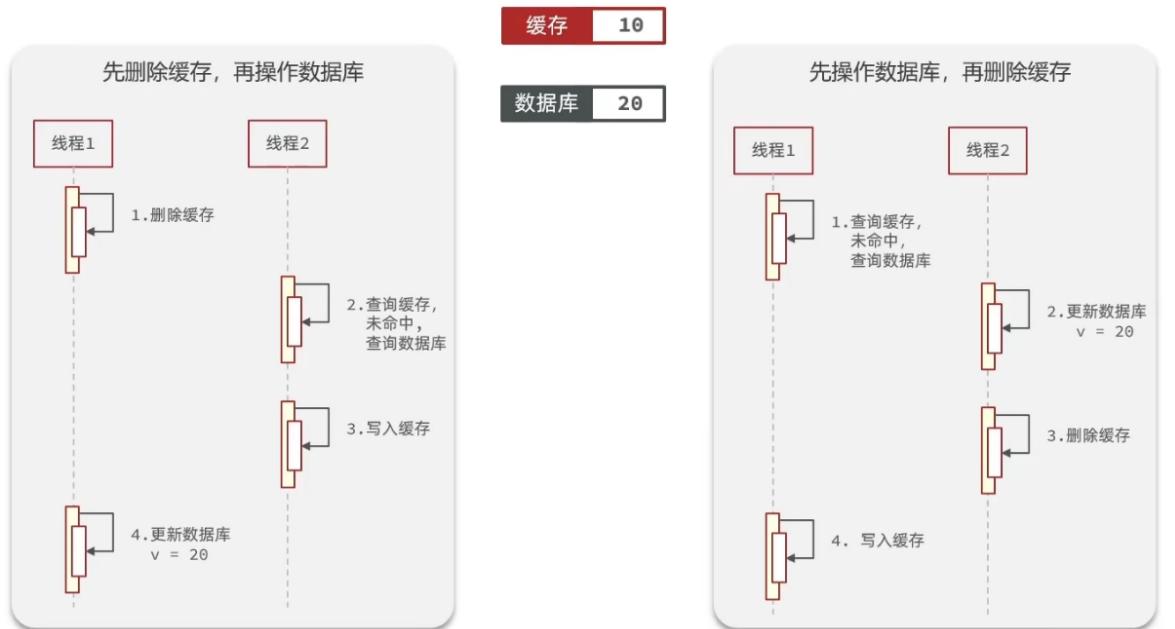
2. 如何保证缓存与数据库的操作的同时成功或失败?

- ◆ 单体系统, 将缓存与数据库操作放在一个事务
- ◆ 分布式系统, 利用TCC等分布式事务方案

3. 先操作缓存还是先操作数据库?

- ◆ 先删除缓存, 再操作数据库
- ◆ 先操作数据库, 再删除缓存

其中, 在第三个问题中, 两个方案在多线程下都有可能出现问题:



又因为缓存操作比数据库操作快得多, 所以方案二出错概率小。

所以采用方案二, 但在方案二出现问题的时候怎么办呢? -用超时剔除作为兜底方案 (根据需要也可以再加上延迟双删)。 (存疑: 即使加了超时剔除, 在出现多线程问题且未剔除的时候, 也还是会出问题啊。)

总结:

缓存更新策略的最佳实践方案：

1. 低一致性需求：使用Redis自带的内存淘汰机制
2. 高一致性需求：主动更新，并以超时剔除作为兜底方案

◆ 读操作：

- 缓存命中则直接返回
- 缓存未命中则查询数据库，并写入缓存，设定超时时间

◆ 写操作：

- 先写数据库，然后再删除缓存
- 要确保数据库与缓存操作的原子性

所以，在【代码演示】中，可以给第六步加上TTL，并在service层的update方法中先操作数据库，再删缓存。

缓存穿透

缓存穿透是指客户端请求的数据在缓存中和数据库中都不存在，这样永远不会有数据写入缓存，如果这个请求发送了很多，每个都达到了数据库，可能会造成问题。

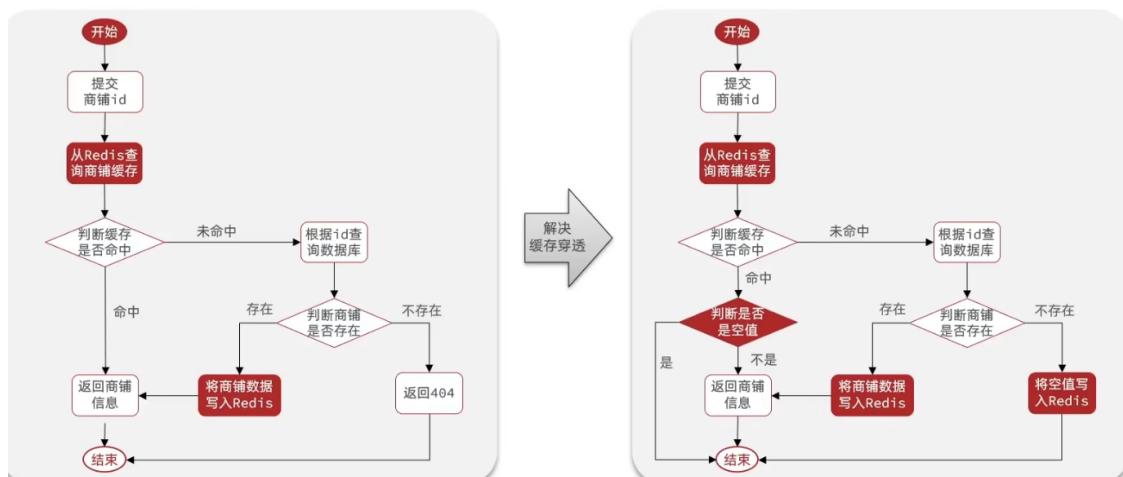
解决方案：

- 缓存空对象：查询不到，缓存null，并加上ttl

优点：实现简单，维护方便

缺点：

- 额外内存消耗
- 可能造成短期数据不一致（即使设置了ttl，数据库在这段时间内更新了，但内存仍是null）



- 布隆过滤器：

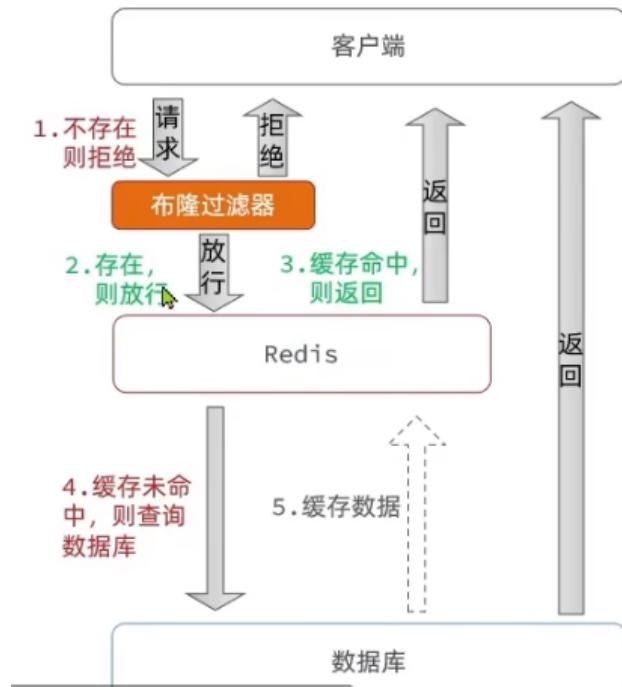
布隆过滤器（Bloom Filter）是一种空间效率非常高的概率型数据结构，主要用于判断一个元素是否在一个集合中。它能够快速地告诉你某个元素“可能在集合中”或“绝对不在集合中”。布隆过滤器通过允许一定程度的误报（即报告某个元素可能存在但实际上不存在的情况），以换取极低的存储需求和高效的查询速度。

布隆过滤器的工作原理

1. **初始化**: 创建一个长度为 mm 的位数组，并将其所有位初始化为 0。
2. **哈希函数**: 选择 kk 个独立的哈希函数，这些哈希函数将输入元素映射到位数组的不同位置上。
3. **插入元素**
 - 当要向布隆过滤器中添加一个元素时，使用这 kk 个哈希函数对该元素进行哈希计算，得到 kk 个不同的哈希值。
 - 将这些哈希值对应到位数组的位置上的位设置为 1。
4. **查询元素**
 - 要检查一个元素是否存在于布隆过滤器中，同样使用这 kk 个哈希函数对该元素进行哈希计算。
 - 如果所有对应的位都是 1，则该元素可能存在于集合中；如果有任何一个位是 0，则该元素绝对不在集合中。

主要特点

- **空间高效**: 相比直接存储元素，布隆过滤器只需要少量的位来表示大量元素的存在情况。
- **时间高效**: 无论是插入还是查询操作的时间复杂度均为 $O(k)$ ，其中 kk 是哈希函数的数量，通常是一个小常数。
- **存在误报率**: 布隆过滤器可能会错误地认为一个不在集合中的元素存在于集合中，但不会漏报（即将存在的元素报告为不存在）。
- **无法删除元素**: 由于多个元素可能共享同一个位，因此不能简单地通过将位重置为 0 来实现删除操作。



布隆过滤器优点：内存占用少，没有多余key

缺点：

- 实现复杂
- 存在误判可能

- 增强id复杂度，避免被猜测id规律，并在接收id时做好校验
- 加强权限验证

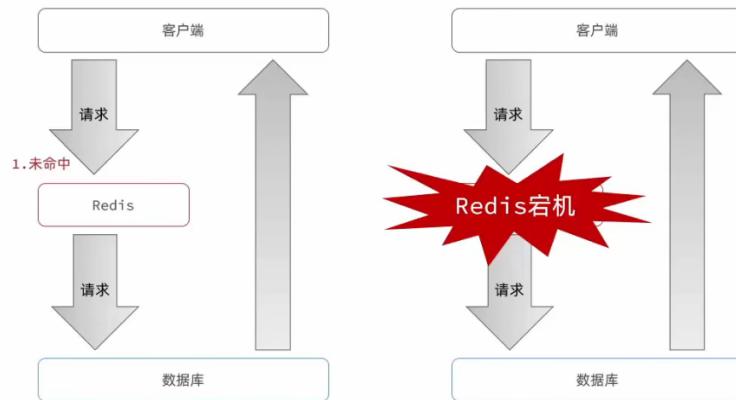
- 热点参数限流

缓存雪崩

缓存雪崩是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。

解决方案：

- ◆ 给不同的Key的TTL添加随机值
- ◆ 利用Redis集群提高服务的可用性
- ◆ 给缓存业务添加降级限流策略
- ◆ 给业务添加多级缓存

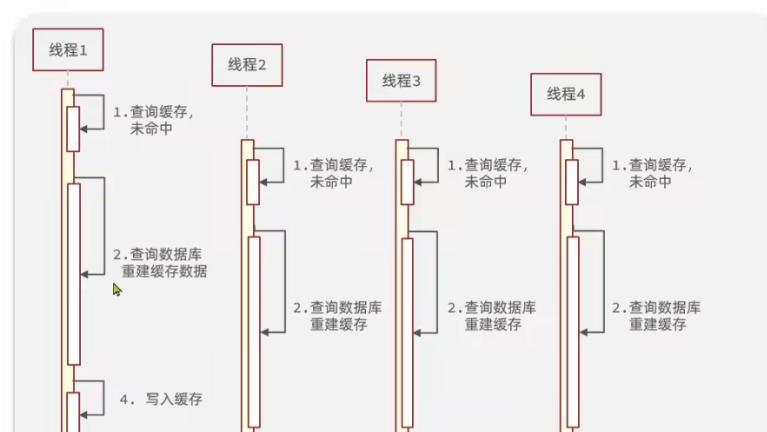


缓存击穿 (热点key问题)

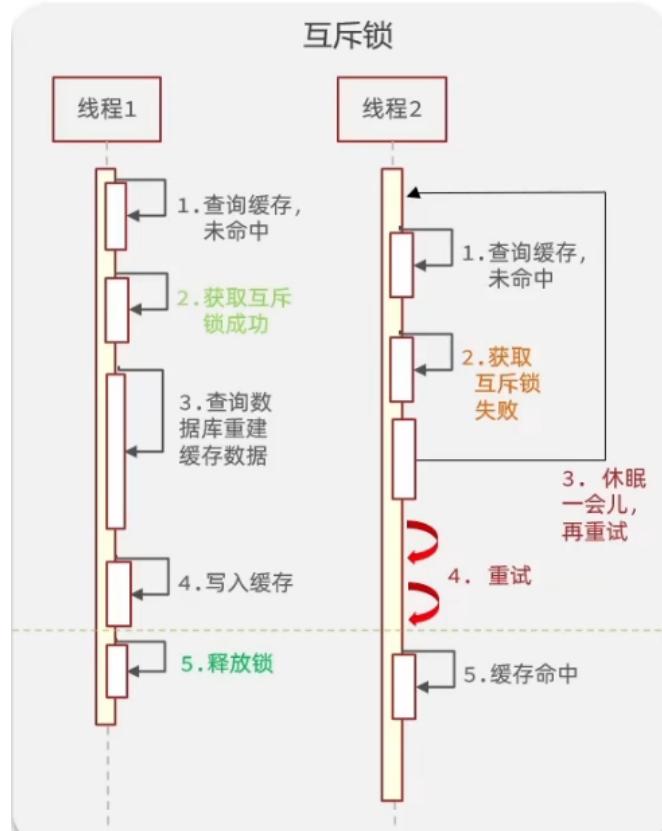
缓存击穿问题也叫热点Key问题，就是一个被高并发访问并且缓存重建业务较复杂的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击。

常见的解决方案有两种：

- ◆ 互斥锁
- ◆ 逻辑过期



互斥锁：



分布式锁:

```

public boolean tryLock(String key) {
    Boolean flag=stringRedisTemplate.opsForValue().setIfAbsent(key, "locked",
10, TimeUnit.SECONDS);
    return BooleanUtil.isTrue(flag);
}

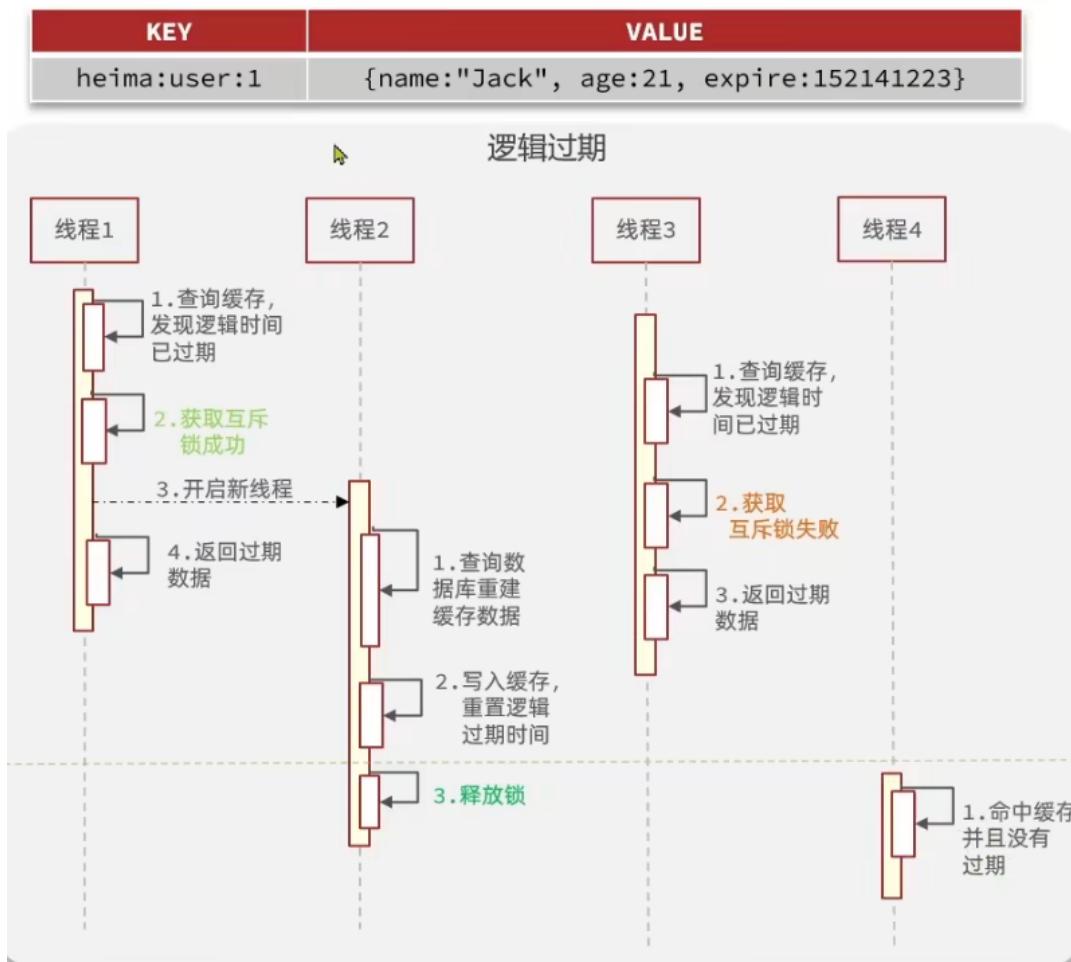
public void unlock(String key) {
    stringRedisTemplate.delete(key);
}

```

但是会造成一个在操作数据库，剩下大量线程在等待

逻辑过期:

不设置显式的ttl了，而是将ttl作为键值对存储，查询时再进行判断是否过期。



解决方案	优点	缺点
互斥锁	<ul style="list-style-type: none">没有额外的内存消耗保证一致性实现简单	<ul style="list-style-type: none">线程需要等待，性能受影响可能有死锁风险
逻辑过期	<ul style="list-style-type: none">线程无需等待，性能较好	<ul style="list-style-type: none">不保证一致性有额外内存消耗实现复杂

需要根据需求（一致性还是可用性）进行选择。 (**CAP定理**)

缓存工具封装

需求：

基于StringRedisTemplate封装一个缓存工具类，满足下列需求：

- ✓ 方法1：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置TTL过期时间
- ✓ 方法2：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置逻辑过期时间，用于处理缓存击穿问题
- ✓ 方法3：根据指定的key查询缓存，并反序列化为指定类型，利用缓存空值的方式解决缓存穿透问题
- ✓ 方法4：根据指定的key查询缓存，并反序列化为指定类型，需要利用逻辑过期解决缓存击穿问题

带有逻辑过期参数的类：

```
package com.hmdp.utils;
import lombok.Data;
import java.time.LocalDateTime;
@Data
public class RedisData{
    private LocalDateTime expireTime;
    private Object data;
}
```

工具类：

```
@Slf4j
@Component
public class CacheClient {
    private final StringRedisTemplate stringRedisTemplate;

    public CacheClient(StringRedisTemplate stringRedisTemplate) {
        this.stringRedisTemplate = stringRedisTemplate;
    }

    //方法一
    public void set(String key, Object value, Long time, TimeUnit timeUnit){
        stringRedisTemplate.opsForValue().set(key,
JSONUtil.toJsonStr(value),time,timeUnit);
    }

    //方法二
    public void setWithLogicalExpire(String key, Object value, Long time,
TimeUnit timeUnit){
        //设置逻辑过期
        RedisData redisData = new RedisData();
        redisData.setData(value);

        redisData.setExpireTime(LocalDateTime.now().plusSeconds(timeUnit.toSeconds(time)));
        //写入redis
        stringRedisTemplate.opsForValue().set(key,
JSONUtil.toJsonStr(redisData));
    }

    //方法三
    public <R, ID> R queryWithPenetration(String keyPrefix, ID id, Class<R>
clazz,
                                            Function<ID, R> dbFallback,
                                            Long time, TimeUnit timeUnit){
        String key = keyPrefix+id;
        //1. 查询缓存
        String json=stringRedisTemplate.opsForValue().get(key);
```

```

//2.判断是否存在
if(strUtil.isNotBlank(json)){
    //3.存在，直接返回
    return JSONUtil.toBean(json, clazz);
}
//判断命中的是否是空值
if(json!=null){
    //返回一个错误信息
    return null;
}
//4.缓存不存在，根据id查询数据库
R r=dbFallback.apply(id);
//5.数据库不存在，返回错误
if(r==null){
    //将空值写入redis
    stringRedisTemplate.opsForValue().set(key, "", 2L, TimeUnit.MINUTES);
    //返回错误信息
    return null;
}
//6.数据库存在，写入redis
this.set(key,r,time,timeUnit);
return r;
}

//方法四
//线程池
private static final ExecutorService CACHE_REBUILD_EXECUTOR =
Executors.newFixedThreadPool(10);
//分布式锁
public boolean tryLock(String key) {
    Boolean flag=stringRedisTemplate.opsForValue().setIfAbsent(key,
"locked", 10, TimeUnit.SECONDS);
    return BooleanUtil.isTrue(flag);
}
public void unlock(String key) {
    stringRedisTemplate.delete(key);
}
public <R,ID> R queryWithCacheBreakdown(String keyPrefix, ID id, Class<R>
clazz,
                                            Function<ID,R> dbFallback,
                                            Long time, TimeUnit timeUnit){
    String key=keyPrefix+id;
    //1.从redis查询缓存
    String json=stringRedisTemplate.opsForValue().get(key);
    //2.判断缓存是否存在
    if(strUtil.isBlank(json)){
        //3.因为是热点key，不存在，判定为数据库也没有，直接返回
        return null;
    }
    //4.命中，先把json反序列化成对象
    RedisData redisData=JSONUtil.toBean(json, RedisData.class);
    R r=JSONUtil.toBean(JSONObject redisData.getData(), clazz);
    LocalDateTime expireTime=redisData.getExpireTime();
    //5.判断是否过期
    if(expireTime.isAfter(LocalDateTime.now())){
        //5.1未过期，直接返回
        return r;
    }
}

```

```

//5.2已过期需要重建缓存
//6.缓存重建
//6.1获取互斥锁
String localKey="lock:shop:"+id;
boolean isLock=tryLock(localKey);
//6.2判断是否成功获取锁
if(isLock){
    //6.3成功，开启独立线程，实现缓存重建
    CACHE_REBUILD_EXECUTOR.submit(() ->{
        try {
            //查询数据库
            R r1=dbFallback.apply(id);
            //写入redis
            this.setWithLogicalExpire(key,r1,time,timeUnit);
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            unlock(localKey);
        }
    });
}
//返回过期信息
return r;
}
}

```

Feed流

关注推送也叫做Feed流。即用户可以通过无限下拉获取新的信息。

Feed流产品有两种常见模式：

- **Timeline**: 不做内容筛选，简单的按照内容发布时间排序，常用于好友或关注。例如朋友圈
 - 优点：信息全面，不会有缺失。并且实现也相对简单
 - 缺点：信息噪音较多，用户不一定感兴趣，内容获取效率低
- **智能排序**: 利用智能算法屏蔽掉违规的、用户不感兴趣的内容。推送用户感兴趣信息来吸引用户
 - 优点：投喂用户感兴趣信息，用户粘度很高，容易沉迷
 - 缺点：如果算法不精准，可能起到反作用

timeline的模式有三种：

以推送关注的人的消息为例

- 拉模式（读扩散）

每个人将信息发送到发件箱，而粉丝在读的时候，才会将消息拉取到自己的收件箱并进行排序。

优点：节省内存空间（收件箱里的消息用完就删，用到再读）

缺点：延迟高

- 推模式（写扩散）

每个人发信息会直接将消息推送到每个粉丝的收件箱并排序。

优点：延迟低

缺点：内存占用高

- 推拉模式（读写混合）

粉丝量较少的用户，直接采用推模式。

粉丝量多的用户，再把粉丝进行分类。活跃粉丝采用推模式，普通粉丝采用拉模式。

	拉模式	推模式	推拉结合
写比例	低	高	中
读比例	高	低	中
用户读取延迟	高	低	低
实现难度	复杂	简单	很复杂
使用场景	很少使用	用户量少、没有大V	过千万的用户量，有大V

滚动分页查询

以滚动分页查询收件箱为例，以时间戳为score

```
zdevrangebyscore key max min withscores offset count:
```

max: 上一次查询的最小时时间戳

min: 0

offset: 与上一次查询最小时时间戳一致的所有元素的个数

count: 每页几个

本条语句的意思是，数据按时间戳从大到小排，在max到min的范围内，从小于等于max的第offset个元素开始，查询count条。并带上score。

GEO数据结构

GEO:Geolocation，地理坐标。

redis允许存储地理坐标信息，方便根据经纬度来检索数据。

```
help @geo 查询相关命令
```

`geoadd key 经度 纬度 member ...`: 添加地理空间信息，包含经度、纬度、值 (member) 。可以添加多个。

`geodist key member1 member2 单位`: 计算指定两点之间的距离并返回。默认单位为米

`geohash`: 将指定member的坐标转为hash字符串形式返回

`geopos key member`: 返回指定member坐标

`geosearch`: 在指定范围内搜索member，并按照与指定点之间的距离排序后返回。范围可以是圆形或矩形。

`geosearchstore`: 与geosearch功能一致，不过可以把结果存储。

java中调用时调用 `stirngRedisTemplate.opsForGeo()`. 具体方法

BitMap

redis中用string类型数据结构实现BitMap，因此最大上限是512M，转换为bit则是 2^{32} 个bit位。

`setbit key offset value`: 向指定位置 (offset) 存入一个0或1。默认为0

`getbit key offset`: 获取指定位置的bit值

`bitcount key`: 统计值为1的bit位的数量

`bitfield`: 操作 (查询、修改、自增) bit数组的指定位置的值。用help @bitfield查询。

由于 `getbit` 只能读取一位，可以使用 `bitfield` 查询多位。`bitfield key get u2 0` 表示从第0位开始读2位，其中u表示无符号。

`bitfield_ro`: 获取bit数组，并以十进制形式返回

`bitop`: 将多个bitmap的结果做位运算 (与、或、异或)

`bitpos key 0/1`: 查找bit数组中指定范围内第一个0或1出现的位置

因为bitmap底层是基于string，所以其操作被封装在字符串相关操作中了。

`stirngRedisTemplate.opsForValue()`.方法

HyperLogLog(HLL)

- **UV**: 全称Unique Visitor，也叫独立访客量，是指通过互联网访问、浏览这个网页的自然人。1天内同一个用户多次访问该网站，只记录1次。
- **PV**: 全称Page View，也叫页面访问量或点击量，用户每访问网站的一个页面，记录1次PV，用户多次打开页面，则记录多次PV。往往用来衡量网站的流量。

UV统计在服务端做会比较麻烦，因为要判断该用户是否已经统计过了，需要将统计过的用户信息保存。但是如果每个访问的用户都保存到Redis中，数据量会非常恐怖。

这时就需要用到HLL

Hyperloglog(HLL)是从Loglog算法派生的概率算法，用于确定非常大的集合的基数，而不需要存储其所有值。相关算法原理大家可以参考：<https://juejin.cn/post/6844903785744056333#heading-0>

Redis中的HLL是基于string结构实现的，单个HLL的内存永远小于16kb，内存占用低的令人发指！作为代价，其测量结果是概率性的，有小于0.81%的误差。不过对于UV统计来说，这完全可以忽略。

`pfadd key element...`: 添加一或多个元素

`pfcount key...`: 获取数量，如果传入多个key，先合并再输出

`pfmerge destkey sourcekey...`: 合并

悲观锁和乐观锁

悲观锁：

认为线程安全问题一定会发生，因此在操作数据之前先获取锁，确保线程串行执行。例如Lock、Synchronized。

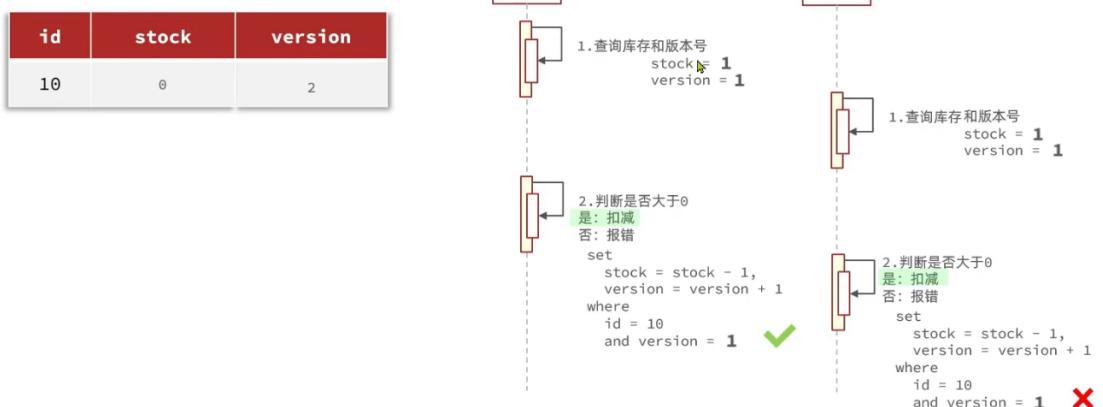
乐观锁：

认为线程安全问题不一定会发生，因此不加锁，只是在更新数据时去判断有没有其他线程对数据做了修改。

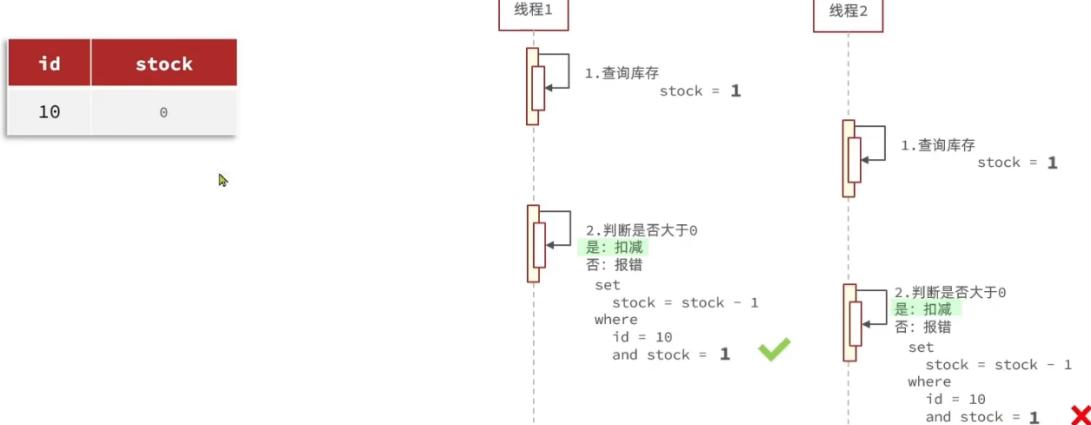
- 如果没有修改则认为是安全的，自己才更新数据
- 如果已经被其他线程修改说明发生了线程安全问题，此时可以重试或异常。

乐观锁的关键是判断之前查询得到的数据是否有被修改过，常见的方式有两种：

◆ 版本号法



◆ CAS法



但是在实际情况中，可能根据业务需求进行修改。比如CAS法中即使stock被修改了，但是只要大于0，那么其他线程还是可以对其进行操作的。

1. 悲观锁：添加同步锁，让线程串行执行

- 优点：简单粗暴
- 缺点：性能一般

2. 乐观锁：不加锁，在更新时判断是否有其它线程在修改

- 优点：性能好
- 缺点：存在成功率低的问题

分布式锁

由于synchronized和lock只能在单个jvm内部使用，在集群或分布式条件下，每个jvm内部都有一个线程能拿到锁，这样就会造成线程安全问题。于是引入分布式锁。

在jvm内部是通过锁监视器实现锁的，多个jvm的锁监视器不一样，就会造成问题。因此，我们要让多个jvm共享同一个锁监视器。

所以，**分布式锁**：

满足分布式系统或集群模式下多进程可见并且互斥的锁。

分布式锁的核心是实现多进程之间互斥，而满足这一点的方式有很多，常见的有三种：

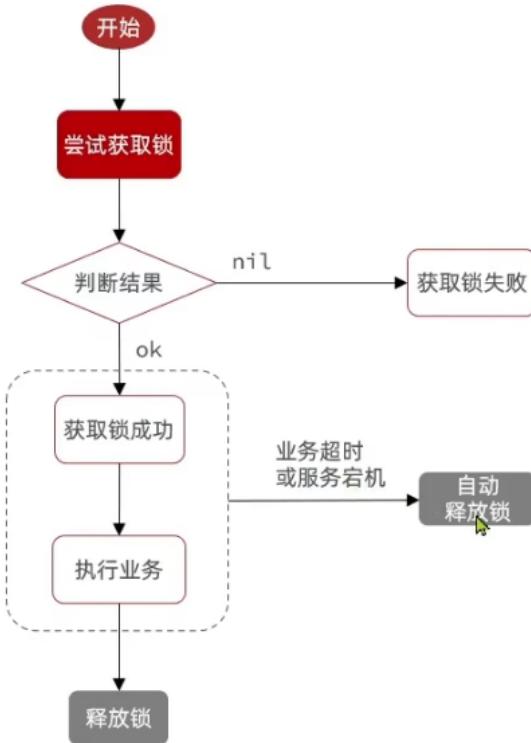
	MySQL	Redis	Zookeeper
互斥	利用mysql本身的互斥锁机制	利用setnx这样的互斥命令	利用节点的唯一性和有序性实现互斥
高可用	好	好	好
高性能	一般	好	一般
安全性	断开连接，自动释放锁	利用锁超时时间，到期释放	临时节点，断开连接自动释放

基于redis的分布式锁

通过setnx的特性，可以实现分布式锁。

但是如果业务超时或服务宕机，锁就得不到释放，所以在设置的时候应该加上超时时间，且应该与setnx在一条语句中。且value最好带上线程标识。

并且获取失败会有两种设置：阻塞式和非阻塞式。阻塞式会一直等到锁被释放，非阻塞式会立即结束并返回一个结果。在这里采用非阻塞式。



先来简单实现一下：

```

public interface ILock {
    /**
     * 尝试获取锁
     * @param timeoutSec 锁持有的超时时间，过期后自动释放
     * @return true代表获取成功，反之失败
     */
    boolean tryLock(long timeoutSec);
    //释放锁
    void unlock();
}

public class SimpleRedisLock implements ILock {
    //业务名称，也是锁的名称
    private String name;
    private StringRedisTemplate stringRedisTemplate;

    public SimpleRedisLock(String name, StringRedisTemplate stringRedisTemplate) {
        this.name = name;
        this.stringRedisTemplate = stringRedisTemplate;
    }
    @Override
    public boolean tryLock(long timeoutSec) {
        //获取线程标识
        Long threadId=Thread.currentThread().getId();
        //获取锁
        Boolean success=stringRedisTemplate.opsForValue().setIfAbsent("lock:"+name+threadId+"", timeoutSec, TimeUnit.SECONDS);
        return Boolean.TRUE.equals(success); //避免自动拆箱空指针风险
    }
}

```

```
@Override  
public void unlock() {  
    stringRedisTemplate.delete("lock:"+name);  
}  
}
```

使用：

```
//创建锁对象  
SimpleRedisLock lock=new SimpleRedisLock("order:"+userId,stringRedisTemplate);  
//获取锁  
boolean isLock= lock.tryLock(5);  
//判断是否获取成功  
if(!isLock){  
    //获取锁失败操作  
    return ...;  
}  
try{  
    //获取锁成功操作  
} finally{  
    lock.unlock();  
}
```

分布式锁误删问题

假设线程一获取锁成功，但是发生阻塞，一直到锁的ttl结束仍没有恢复，于是锁被释放。

线程二在此时可以拿到锁并执行自己的业务，但此时线程一又恢复了并执行完业务开始释放锁，释放了线程二的锁，这是其他进程又能拿到锁并与线程二并行执行。这就会造成线程安全问题。

所以，在上面的代码中，释放锁时需要判断是不是自己的锁。

上面添加的线程标识就起作用了，但集群或分布式环境还要考虑线程id是否相同。于是我们干脆使用UUID+线程id的方式作为value。同一服务器使用的UUID相同。

```
private static final String ID_PREFIX= UUID.randomUUID().toString(true)+"-";
```

```
//获取线程标识  
String threadId=ID_PREFIX+Thread.currentThread().getId();
```

改进的unLock：

```
@Override  
public void unlock() {  
    //获取线程标识  
    String threadId=ID_PREFIX+Thread.currentThread().getId();  
    String id = stringRedisTemplate.opsForValue().get("lock:" + name);  
    //判断标志是否一致  
    if(threadId.equals(id)){  
        stringRedisTemplate.delete("lock:"+name);  
    }  
}
```

分布式锁的原子性问题 (lua)

假设线程一拿到锁，执行业务完毕并经过判断确认redis中存的锁是自己的，但此时发生阻塞。锁被超时释放。

此时线程二拿到了锁，正在执行业务，线程一又恢复了，释放了线程二的锁，会造成线程安全问题。

问题产生的原因是判断锁和释放锁并非原子性操作。

可以使用lua脚本：

Redis提供了Lua脚本功能，在一个Lua脚本中编写多条redis命令，确保多条命令执行时的原子性。(Lua是一种编程语言。)

这里重点介绍Redis提供的调用函数，语法如下：

```
# 执行redis命令  
redis.call('命令名称', 'key', '其它参数', ...)
```

例如，我们要执行set name jack，则脚本是这样：

```
# 执行 set name jack  
redis.call('set', 'name', 'jack')
```

例如，我们要先执行set name Rose，再执行get name，则脚本如下：

```
# 先执行 set name jack  
redis.call('set', 'name', 'jack')  
# 再执行 get name  
local name = redis.call('get', 'name')  
# 返回  
return name
```

写好脚本以后，需要用Redis命令来调用脚本，调用脚本的常见命令如下：

```
127.0.0.1:6379> help @scripting
EVAL script numkeys key [key ...] arg [arg ...]
summary: Execute a Lua script server side
since: 2.6.0
```

例如，我们要执行 `redis.call('set', 'name', 'jack')` 这个脚本，语法如下：

```
# 调用脚本
EVAL "return redis.call('set', 'name', 'jack')"
0
```

脚本内容 脚本需要的key类型的参数个数

如果脚本中的key、value不想写死，可以作为参数传递。key类型参数会放入KEYS数组，其它参数会放入ARGV数组，在脚本中可以从KEYS和ARGV数组获取这些参数：

```
# 调用脚本
EVAL "return redis.call('set', KEYS[1], ARGV[1])"
1 name Rose
```

脚本内容 脚本需要的key类型的参数个数

注意：lua中的下标从1开始

改进：

在resources目录下创建unlock.lua

```
-- 获取锁中的线程标识 get key
local id=redis.call("get", KEYS[1])
-- 比较线程标识与锁中的标识是否一致
if(id==ARGV[1]) then
    -- 释放锁
    return redis.call("del", KEYS[1])
end
return 0
```

提前加载脚本防止多余IO流

```
private static final DefaultRedisScript<Long> UNLOCK_SCRIPT;
static {
    UNLOCK_SCRIPT=new DefaultRedisScript<>();
    UNLOCK_SCRIPT.setLocation(new ClassPathResource("unlock.lua"));
    UNLOCK_SCRIPT.setResultType(Long.class);
}
```

unLock方法：

```
@Override
public void unlock() {
    //调用lua脚本
    stringRedisTemplate.execute(
        UNLOCK_SCRIPT,
        Collections.singletonList("lock:"+name),
        ID_PREFIX+Thread.currentThread().getId());
}
```

分布式锁的优化（源码）

```

public class SimpleRedisLock implements ILock {
    //业务名称，也是锁的名称
    private String name;
    private StringRedisTemplate stringRedisTemplate;
    private static final String ID_PREFIX= UUID.randomUUID().toString(true)+"-";
    private static final DefaultRedisScript<Long> UNLOCK_SCRIPT;
    static {
        UNLOCK_SCRIPT=new DefaultRedisScript<>();
        UNLOCK_SCRIPT.setLocation(new ClassPathResource("unlock.lua"));
        UNLOCK_SCRIPT.setResultType(Long.class);
    }

    public SimpleRedisLock(String name, StringRedisTemplate stringRedisTemplate) {
        this.name = name;
        this.stringRedisTemplate = stringRedisTemplate;
    }

    @Override
    public boolean tryLock(Long timeoutSec) {
        //获取线程标识
        String threadId=ID_PREFIX+Thread.currentThread().getId();
        //获取锁
        Boolean success=stringRedisTemplate.opsForValue().setIfAbsent("lock:"+name,threadId,time
        outSec, TimeUnit.SECONDS);
        return Boolean.TRUE.equals(success); //避免自动拆箱空指针风险
    }

    @Override
    public void unlock() {
        //调用lua脚本
        stringRedisTemplate.execute(
            UNLOCK_SCRIPT,
            Collections.singletonList("lock:"+name),
            ID_PREFIX+Thread.currentThread().getId());
    }
}

```

经过改进，分布式锁变成如上这样。但仍有地方可以改进。

- 不可重入：即一个线程无法多次获取同一把锁。
(可重入锁 (Reentrant Lock)，在Java中指的是一个线程可以多次获取同一个锁而不会导致死锁的机制。这意味着，如果一个线程已经持有了某个锁，并试图再次获取这个锁，它将能够成功获取而不被阻塞。这种锁被称为“可重入”的，因为它允许同一个线程重新进入 (re-enter) 一个它已经持有的锁。)
- 不可重试：获取锁只尝试一次就返回false，没有重试机制
- 超时释放：锁超时释放虽然可以避免死锁问题，但如果业务执行时间较长，锁释放了但业务未完毕，可能造成安全隐患。
- 主从一致性：如果Redis提供了主从集群，主从同步存在延迟，一个线程拿到锁了，刚好主节点宕机，如果从节点还没有同步主节点中的锁数据，另一个线程也能拿到锁。

如果自己去实现上述功能实在麻烦，于是引入Redisson

Redisson 是一个用于 Redis 的 Java 客户端，它使得在 Java 应用中使用 Redis 变得更加简单和便捷。Redisson 提供了许多高级功能，如分布式对象、分布式集合、分布式锁、信号量等，帮助开发者构建高可用的分布式系统。Redisson 支持多种部署模式，包括单节点模式、主从模式、哨兵模式和集群模式。

Redisson快速入门

导入依赖：

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.26.1</version>
</dependency>
```

创建bean：

```
@Configuration
public class RedissonConfig {
    @Bean
    public RedissonClient redissonClient() {
        //配置
        Config config = new Config();
        config.useSingleServer()
            .setAddress("redis://192.168.14.128:6379")
            .setPassword("abc123");
        //创建RedissonClient对象
        return Redisson.create(config);
    }
}
```

使用：

```
@Autowired
private RedissonClient redissonclient;

RLock lock=redissonclient.getLock("lockName");
//尝试获取锁，参数分别是：获取锁的最大等待时间（期间会重试），锁自动释放时间，时间单位
boolean isLock=lock.tryLock();
if(!isLock){
    //失败机制
    return ...;
}
try{
    //业务
} finally{
    lock.unlock();
}
```

Redisson可重入锁原理

锁存储结构：

KEY	VALUE	
	field	value
lock	thread1	0

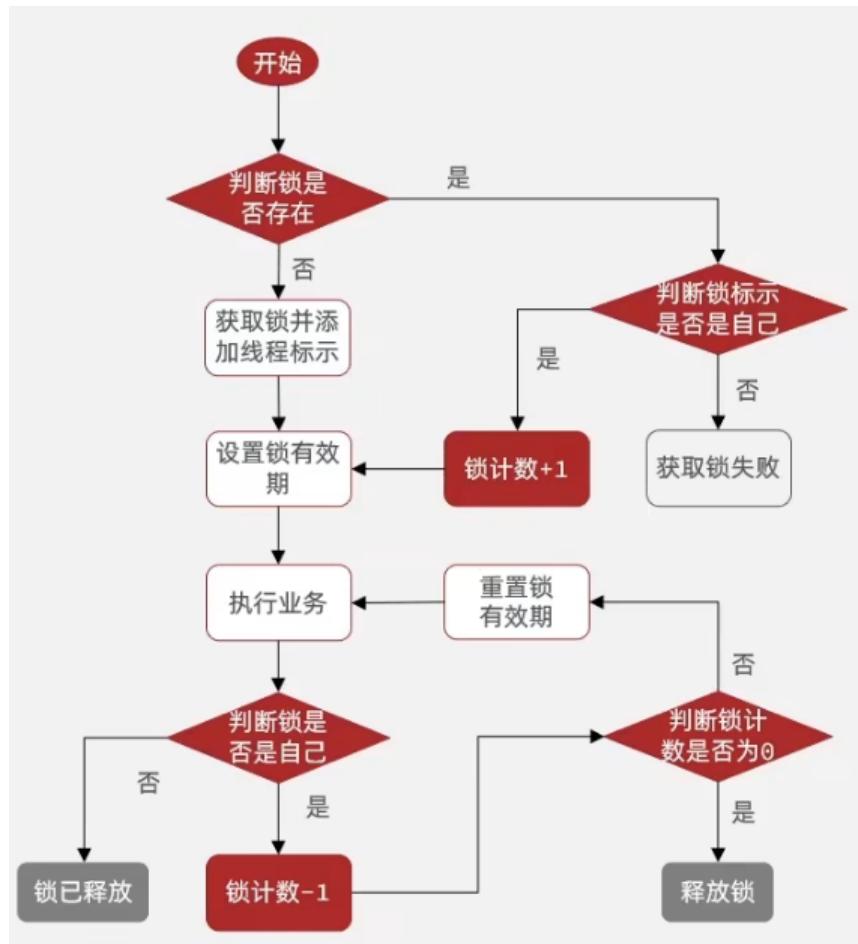
key是我们为锁指定的名字， field标记了线程， value： 使用计数器原理记录进入了几层锁。

举例：

```
// 创建锁对象
RLock lock = redissonClient.getLock("lock");

@Test
void method1() {
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 1");
        return;
    }
    try {
        log.info("获取锁成功, 1");
        method2();
    } finally {
        log.info("释放锁, 1");
        lock.unlock();
    }
}
void method2(){
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 2");
        return;
    }
    try {
        log.info("获取锁成功, 2");
    } finally {
        log.info("释放锁, 2");
        lock.unlock();
    }
}
```

在以上的例子中，使用RLock是可以重入锁的，怎么实现的？



举例：

获取锁的Lua脚本：

```

local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间
-- 判断是否存在
if(redis.call('exists', key) == 0) then
    -- 不存在, 获取锁
    redis.call('hset', key, threadId, '1');
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
-- 锁已经存在, 判断threadId是否是自己
if(redis.call('hexists', key, threadId) == 1) then
    -- 不存在, 获取锁, 重入次数+1
    redis.call('hincrby', key, threadId, '1');
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
return 0; -- 代码走到这里, 说明获取锁的不是自己, 获取锁失败

```

释放锁的Lua脚本：

lock:order threa

```
local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间
-- 判断当前锁是否还是被自己持有
if (redis.call('HEXISTS', key, threadId) == 0) then
    return nil; -- 如果已经不是自己，则直接返回
end;
-- 是自己的锁，则重入次数-1
local count = redis.call('HINCRBY', key, threadId, -1);
-- 判断是否重入次数是否已经为0
if (count > 0) then
    -- 大于0说明不能释放锁，重置有效期然后返回
    redis.call('EXPIRE', key, releaseTime);
    return nil;
else -- 等于0说明可以释放锁，直接删除
    redis.call('DEL', key);
    return nil;
end;
```

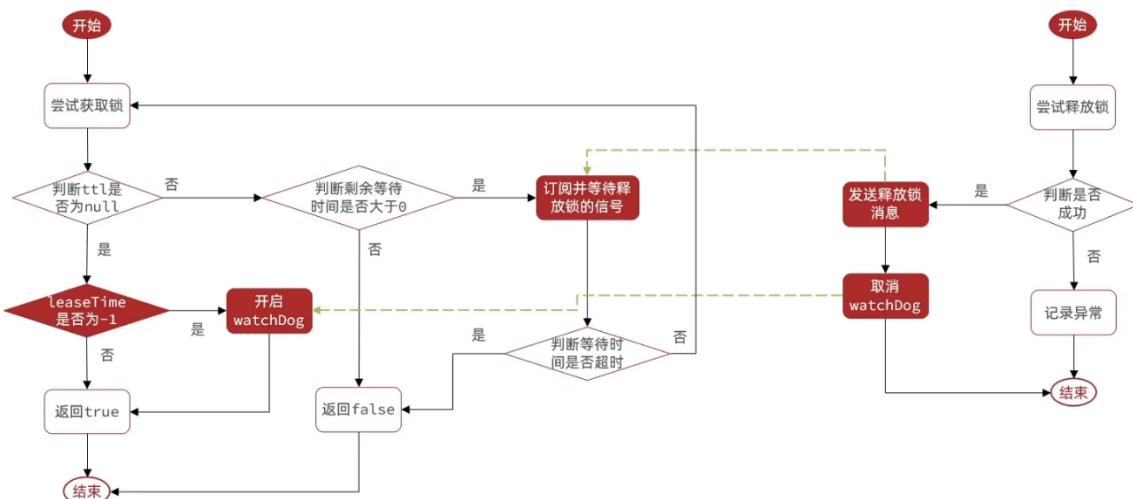
如果在第二层锁释放时发现所已经不是自己的了，那么第一层锁接下来的代码还是会执行，会造成线程安全问题。这里要用到续约机制：

为了避免锁因过期而被其他线程抢占，Redisson 提供了自动续期机制（Watchdog）。只要线程仍然持有锁，Redisson 就会定期延长锁的有效期。

- 默认情况下，Redisson 会每 10 秒检查一次锁的有效期，并将其续期为 30 秒。
- 如果业务逻辑执行时间超过锁的初始过期时间，续期机制可以确保锁不会被意外释放。

Redisson的锁重试和watchdog机制（续约）

https://www.bilibili.com/video/BV1cr4y1671t?p=67&vd_source=c054be8430afebb3d00e5f2d0b77f9fc



左边是获取锁，右边是释放锁。其中watchdog用来续约。

Redisson分布式锁原理：

- **可重入**: 利用hash结构记录线程id和重入次数
- **可重试**: 利用信号量和PubSub功能实现等待、唤醒，获取锁失败的重试机制
- **超时续约**: 利用watchDog，每隔一段时间（releaseTime / 3），重置超时时间

multiLock原理(主从一致性)

主节点用于增删改，从节点只用于一些读操作。因而需要主从同步，而主从同步需要一定的时间，而此时主节点宕机了，导致在主节点获取的锁还没有同步到从节点，这就导致了主从一致性问题。

于是，我们可以先不用主从，建立多个节点，在获取锁的时候，只有向所有节点获取锁都成功，才算锁获取成功。（这些节点之间不分主从，但是可以和另外的节点构成主从），这样就解决了主从一致性问题。

```
RLock lock1=redissonClient.getLock("order");
RLock lock2=redissonClient2.getLock("order");
RLock lock3=redissonClient3.getLock("order");
lock=redissonClient.getMultiLock(lock1,lock2,lock3);
```

源码：`RedissonMultiLock.class` 中的tryLock方法

总结：

1) 不可重入Redis分布式锁:

- ◆ 原理：利用setnx的互斥性；利用ex避免死锁；释放锁时判断线程标示
- ◆ 缺陷：不可重入、无法重试、锁超时失效

2) 可重入的Redis分布式锁:

- ◆ 原理：利用hash结构，记录线程标示和重入次数；利用watchDog延续锁时间；利用信号量控制锁重试等待
- ◆ 缺陷：redis宕机引起锁失效问题

3) Redisson的multiLock:

- ◆ 原理：多个独立的Redis节点，必须在所有节点都获取重入锁，才算获取锁成功
- ◆ 缺陷：运维成本高、实现复杂

高级篇

以下是高级篇

分布式缓存

单点redis的问题即解决方案概述

- redis是内存存储，服务重启可能会丢失数据
--redis数据持久化
- redis虽然并发能力较强，但仍无法满足一些场景
--搭建主从集群，实现读写分离
- 如果redis宕机，则服务不可用，需要一种自动的故障恢复手段。
--利用redis哨兵，实现健康检测和自动恢复。
- 基于内存，单节点能存储的数据量难以满足海量数据的存储需求。
--参考es，搭建分片集群，利用插槽机制实现动态扩容。

redis持久化

RDB

redis database backup file (redis数据备份文件)，也叫redis数据快照。简单来说就是把内存中所有的数据记录到磁盘中。当redis实例故障重启后，从磁盘读取快照文件，恢复数据。
快照文件称为RDB文件，默认保存在当前运行目录。

执行：

进入redis-cli

`save`: 由redis主进程执行rdb, 因为redis是单线程的, 所以会阻塞所有命令。不推荐

`bgsave`: fork () 一个子进程执行RDB, 避免主进程受到影响。

redis非故障关机前执行一次RDB

Redis内部也有触发RDB的机制, 可以在redis.conf文件中找到, 格式如下:

```
save 900 1 #每900秒内至少有一次写操作, 则出发bgsave
save 300 10
save 60 10000
```

Redis 支持在运行时动态修改配置, 可以通过 `CONFIG SET` 命令来更新 `save` 规则。

```
CONFIG SET save "900 1 300 10 60 10000"
```

可以使用以下命令查看当前的 `save` 规则:

```
CONFIG GET save
```

(改为`save ""`, 表示禁用此配置)

```
# 是否压缩 ,建议不开启, 压缩也会消耗cpu, 磁盘的话不值钱
rdbcompression yes
```

其他配置:
RDB文件名称
`dbfilename dump.rdb`

文件保存的路径目录
`dir ./`

RDB的fork原理(异步持久化)

`bgsave`开始时会fork主进程得到子进程, 子进程共享主进程的内存数据。完成fork后读取内存数据并写入RDB文件。fork的过程是阻塞的, 因此必须要快。

linux系统中, 进程无法操作实际物理内存, 而是操作虚拟内存。用页表存储了从虚拟页到物理帧的映射关系, 从而完成对物理内存数据的读写。

在fork时, 子进程只拷贝页表, 也就是虚拟内存和物理内存的映射关系。于是子进程在操作自己的虚拟内存, 就可以操作映射的那块相同的物理内存。在读取完毕后, 就可以存入磁盘 (用新的rdb文件替换旧的)

这就使fork加快, 防止阻塞过长时间。

fork采用COW(copy-on-write)技术:

为什么需要 Copy-On-Write?

传统的 `fork` 实现会直接复制父进程的所有内存页到子进程中，这会导致以下问题：

1. **高内存消耗**：如果父进程占用了大量内存，`fork` 时需要分配与父进程相同大小的内存空间。
2. **低效率**：即使子进程可能只使用少量内存（例如仅执行 `exec` 加载新程序），也会浪费大量时间和资源来复制整个地址空间。

通过 Copy-On-Write，操作系统可以：

- 初始时不复制任何内存页，而是让父子进程共享内存。
- 仅在需要修改时才复制相关页面，从而节省内存和 CPU 开销。

Copy-On-Write 的工作原理

共享内存页

- 当 `fork` 被调用时，操作系统并不会立即复制父进程的内存页。
- 父进程和子进程的页表都指向相同的物理内存页，并且这些页被标记为 **只读**。

检测写操作

- 如果父进程或子进程尝试对某个共享页进行写操作，会触发 **页错误 (Page Fault)**。
- 操作系统捕获这个页错误，并检查该页是否被标记为共享。

复制页面

- 操作系统为触发写操作的进程创建一个新的物理内存页，并将原始页的内容复制到新页中。
- 修改该进程的页表，使其指向新的物理页，原始页的引用计数减1，而另一个进程仍然指向原始页。
- 新页被标记为可写，允许进程完成写操作。

继续执行

- 写操作完成后，进程恢复正常执行，后续对该页的访问将使用新复制的页。

优点

1. **节省内存**：父子进程共享内存页，直到需要修改时才复制，减少了不必要的内存分配。
2. **提高性能**：避免了在 `fork` 时复制大量内存页的操作，特别是对于大内存进程。
3. **支持快速进程创建**：`fork` 的开销主要在于页表的复制，而不是实际内存数据的复制。

缺点

1. **页错误开销**：首次写操作会触发页错误，导致一定的性能开销（需要分配新页并复制数据）。
2. **复杂性增加**：操作系统需要管理共享页的状态，并处理页错误，增加了实现的复杂性。

RDB的缺点：

- 执行间隔长，两次RDB之间写入的数据由丢失风险
- `fork`子进程，写出、压缩RDB文件都比较耗时

AOF

Append Only File (追加文件)。redis处理的每一个写命令（还有选择库）都会记录在AOF文件，可以看作是命今日志文件。

RDB容易数据丢失，AOF可以大大提高数据安全性，弥补这一缺陷。

AOF默认是关闭的，需要修改redis.conf配置文件来开启：

```
appendonly yes
```

```
appendfilename "xxx.aof"
```

AOF的记录命令频率也可以通过配置文件配置：

```
# 表示每执行一次写命令，立即记录到AOF文件
appendfsync always
# 写命令执行完先放入AOF缓冲区，然后表示每隔1秒将缓冲区数据写到AOF文件，是默认方案
appendfsync everysec
# 写命令执行完先放入AOF缓冲区，由操作系统决定何时将缓冲区内容写回磁盘
appendfsync no
```

配置项	刷盘时机	优点	缺点
Always	同步刷盘	可靠性高，几乎不丢数据	性能影响大
everysec	每秒刷盘	性能适中	最多丢失1秒数据
no	操作系统控制	性能最好	可靠性较差，可能丢失大量数据

因为是记录命令，AOF文件会比RDB文件大得多。而且AOF会记录对同一个key的多次写操作，但其实最后一次修改才有意义。

通过执行bgrewriteaof命令，可以让AOF文件执行重写功能，用最少的命令达到同样的效果。

这条命令的自动执行条件：

```
# AOF文件比上次文件 增长超过多少百分比则触发重写
auto-aof-rewrite-percentage 100
# AOF文件体积最小多大以上才触发重写
auto-aof-rewrite-min-size 64mb
```

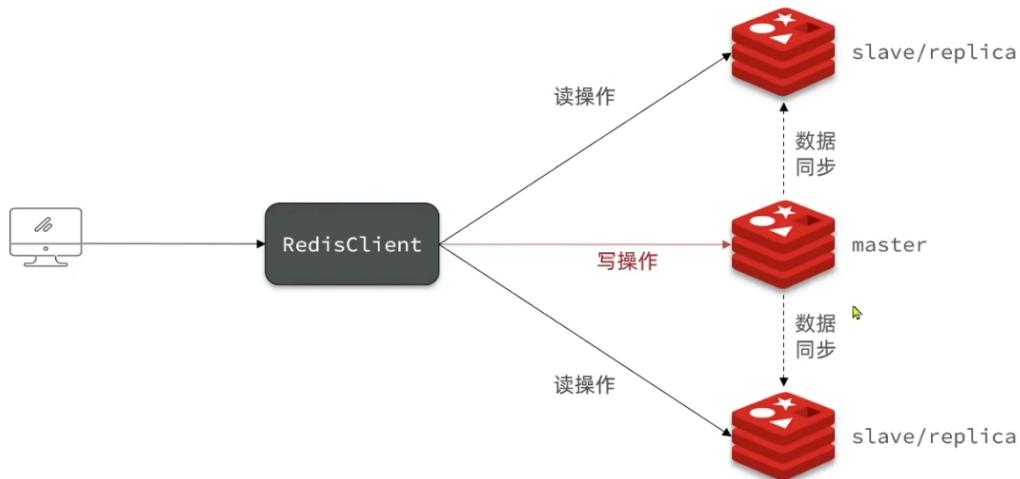
RDB和AOF对比

RDB和AOF各有自己的优缺点，如果对数据安全性要求较高，在实际开发中往往结合两者来使用。

	RDB	AOF
持久化方式	定时对整个内存做快照	记录每一次执行的命令
数据完整性	不完整，两次备份之间会丢失	相对完整，取决于刷盘策略
文件大小	会有压缩，文件体积小	记录命令，文件体积很大
宕机恢复速度	很快	慢
数据恢复优先级	低，因为数据完整性不如AOF	高，因为数据完整性更高
系统资源占用	高，大量CPU和内存消耗	低，主要是磁盘IO资源 但AOF重写时会占用大量CPU和内存资源
使用场景	可以容忍数分钟的数据丢失，追求更快的启动速度	对数据安全性要求较高常见

主从

单节点Redis的并发能力是有上限的，要进一步提高Redis的并发能力，就需要搭建主从集群，实现读写分离。



搭建主从集群

仅作参考。

在/tmp目录下创建7001,7002,7003文件夹。

将/usr/local/src/redis-7.0.15下的redis.conf复制到每一个文件夹下。更改端口号和 数据保存的目录
(默认当前目录，我们希望写在各自的文件夹中)

```
sed -i -e 's/6379/7001/g' -e 's/dir .\/dir \tmp\7001\g' 7001/redis.conf
sed -i -e 's/6379/7002/g' -e 's/dir .\/dir \tmp\7002\g' 7002/redis.conf
sed -i -e 's/6379/7003/g' -e 's/dir .\/dir \tmp\7003\g' 7003/redis.conf
```

接下来，可以启动redis。

输入 redis-server 7001/redis.conf 启动。

但是他们三者之间目前没有主从关系。要配置主从可以使用replicaof命令。

有临时和永久两种模式：

- 修改配置文件 (永久生效)
 - 在redis.conf中添加一行配置： slaveof <masterip> <masterport>
- 使用redis-cli客户端连接到redis服务，执行slaveof命令（重启后失效）：

```
slaveof <masterip> <masterport>
```

如果主节点启用了密码认证（通过 requirepass 配置），而从节点未提供正确的密码，则主节点会拒绝从节点的连接。

- 在从节点的配置文件中或运行时设置主节点的密码：

```
masterauth <主节点密码>
```

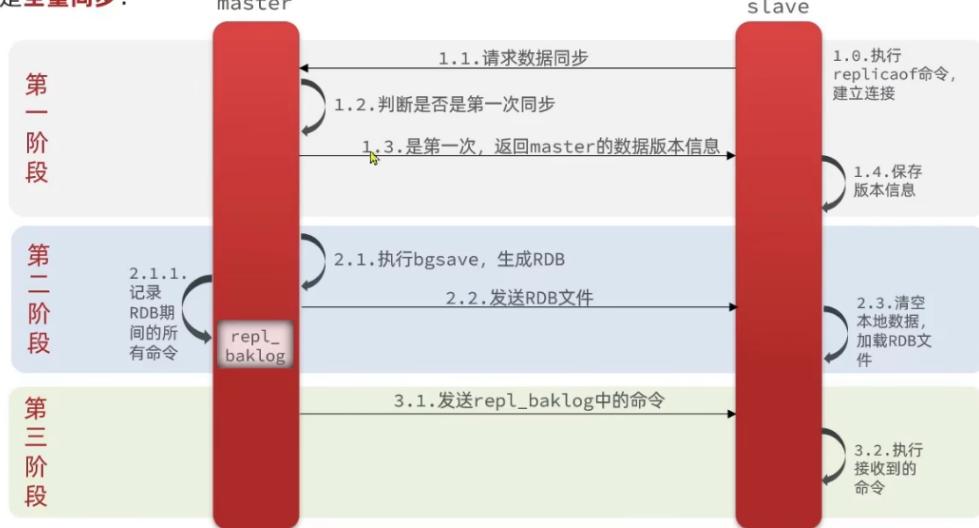
- 或者在运行时动态设置：

```
CONFIG SET masterauth <主节点密码>
```

数据同步原理

全量同步(sync)

主从第一次同步是**全量同步**：



这个过程是比较慢的。

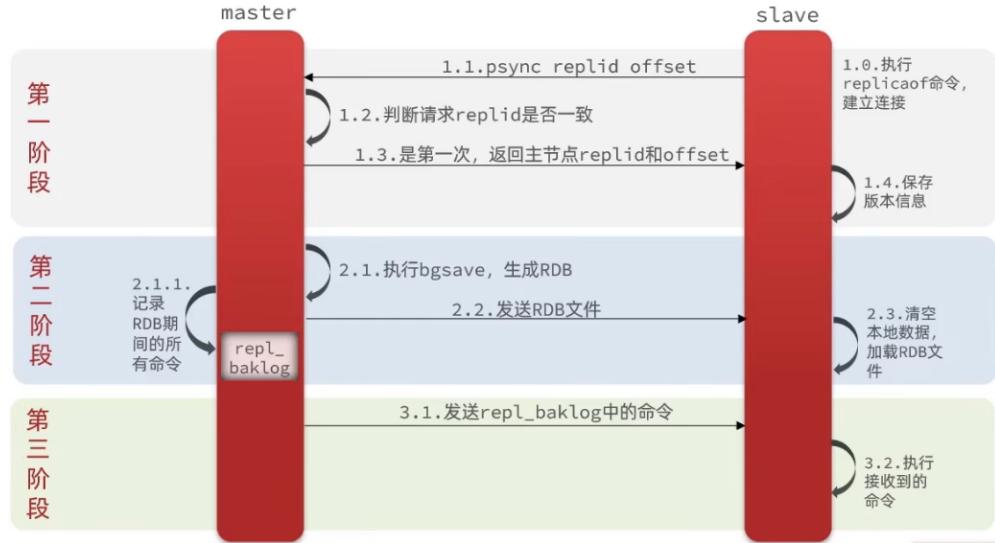
master如何判断slave是不是第一次来同步数据？两个重要概念：

- Replication Id: 简称replid，是数据集的标记，id一致则说明是同一数据集。每一个master都有唯一的replid，slave会继承master节点的replid。
- offset: 偏移量，随着记录在repl_baklog中的数据增多而越来越大。slave完成同步时也会记录当前同步的eoffset。如果slave的offset小于master的offset，说明数据落后，需要更新。

因此slave做数据同步，必须向master声明自己的replication id和offset，master才可以判断到底需要同步哪些数据。

所以，如果从节点replid和主节点不一致，说明是第一次来，全量同步。

主从第一次同步是全量同步



流程总结：

简述全量同步的流程？

- slave节点请求增量同步
- master节点判断replid，发现不一致，拒绝增量同步
- master将完整内存数据生成RDB，发送RDB到slave
- slave清空本地数据，加载master的RDB
- master将RDB期间的命令记录在repl_baklog，并持续将log中的命令发送给slave
- slave执行接收到的命令，保持与master之间的同步

增量同步(psync)

主从第一次同步是全量同步，但如果slave重启后同步，则执行增量同步



注意

repl_baklog大小有上限，写满后会覆盖最早的数据。如果slave断开时间过久，导致尚未备份的数据被覆盖，则无法基于log做增量同步，只能再次全量同步。

又因为全量同步是比较慢的，所以上情况怎么减少/缓解呢？

可以采用以下几个方面来优化Redis主从集群：

- master中配置 `repl-diskless-sync yes` 启用无磁盘复制，避免全量同步时的磁盘IO（在写RDB是不写到磁盘中了，直接写到网络中发给slave）。（适用于网络快、磁盘慢的情况）
- Redis单节点上的内存占用不要太大，减少RDB导致的过多磁盘IO
- 适当提高`repl_baklog`的大小，并且在slave宕机时尽快恢复，尽可能避免全量同步
- 为了避免master主从复制压力过大，限制一个master上的slave数量，如果是在太多slave，则可以采用主-从-从的链式结构。

总结

简述全量同步和增量同步区别？

- 全量同步：master将完整内存数据生成RDB，发送RDB到slave。后续命令则记录在`repl_baklog`，逐个发送给slave。
- 增量同步：slave提交自己的offset到master，master获取`repl_baklog`中从offset之后的命令给slave

什么时候执行全量同步？

- slave节点第一次连接master节点时
- slave节点断开时间太久，`repl_baklog`中的offset已经被覆盖时

什么时候执行增量同步？

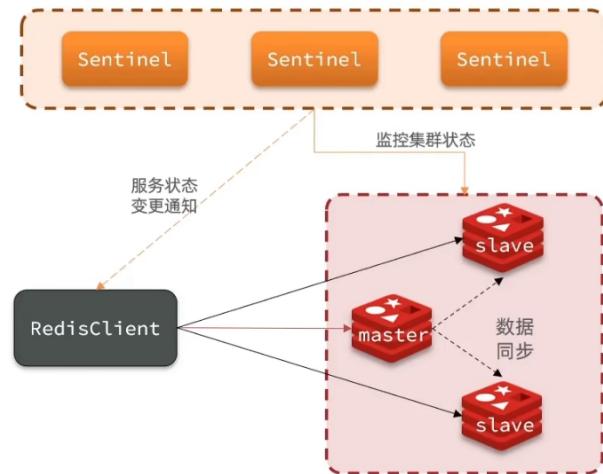
- slave节点断开又恢复，并且在`repl_baklog`中能找到offset时

哨兵

哨兵作用和工作原理

Redis提供了哨兵（Sentinel）机制来实现主从集群的自动故障恢复。哨兵的结构和作用如下：

- **监控**: Sentinel 会不断检查您的master和slave 是否按预期工作
- **自动故障恢复**: 如果master故障, Sentinel会将一个slave提升为master。当故障实例恢复后也可以新的master为主
- **通知**: Sentinel充当Redis客户端的服务发现来源, 当集群发生故障转移时, 会将最新信息推送给Redis的客户端



Sentinel基于心跳机制监测服务状态，每隔一秒向集群的每个实例发送ping命令：

- 主观下线: 如果Sentinel节点发现某实例未在规定时间内响应，则认为该实例**主观下线**
- 客观下线: 若超过指定数量 (quorum) 的sentinel都认为该实例主观下线，则该实例客观下线。quorum值最好超过Sentinel实例数量的一半。

一旦发现master故障, sentinel需要在slave中选择一个作为新的master (sentinel也会选取一个领导者执行此操作), 依据:

- 首先会判断slave节点与master节点断开时间长短, 如果超过指定值 (down-after-milliseconds * 10) 则会排除该 slave节点
- 然后判断slave节点的slave-priority值, 越小优先级越高, 如果是0则永不参与选举
- 如果slave-priority一样, 则判断slave节点的offset值, 越大说明数据越新, 优先级越高
- 最后是判断slave节点的运行id大小, 越小优先级越高。

选举slave成为master的步骤如下:

- sentinel给备选的slave1节点发送slaveof no one命令, 让该节点成为master
- sentinel给所有其它slave发送slaveof 192.168.150.101 7002 命令, 让这些slave成为新master的从节点, 开始从新的master上同步数据。
- 最后, sentinel将故障节点标记为slave, 当故障节点恢复后会自动成为新的master的slave节点



搭建哨兵集群

/tmp目录下创建三个文件夹s1、s2、s3。

```
mkdir s1 s2 s3
```

在s1目录创建一个sentinel.conf, 添加下面内容:

```
port 27001
sentinel announce-ip 192.168.14.128
sentinel monitor mymaster 192.168.14.128 7001 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 60000
dir "/tmp/s1"
sentinel auth-pass mymaster <密码>
```

解读：

- `port 27001`：是当前sentinel实例的端口
- `sentinel monitor mymaster 192.168.14.128 7001 2`：指定主节点信息
 - `mymaster`：主节点名称，自定义，任意写
 - `192.168.14.128 7001`：主节点的ip和端口
 - `2`：选举master时的quorum值
- `sentinel auth-pass mymaster <密码>`：redis的密码。如果redis主从节点密码不一致，需要分别配置此项。

注意redis从节点配置文件中也要配置主节点的密码。

然后将s1/sentinel.conf文件拷贝到s2、s3两个目录中（在/tmp目录执行下列命令）：

```
# 方式一：逐个拷贝
cp s1/sentinel.conf s2
cp s1/sentinel.conf s3
# 方式二：管道组合命令，一键拷贝
echo s2 s3 | xargs -t -n 1 cp s1/sentinel.conf
```

修改s2、s3两个文件夹内的配置文件，将端口分别修改为27002、27003：

```
sed -i -e 's/27001/27002/g' -e 's/s1/s2/g' s2/sentinel.conf
sed -i -e 's/27001/27003/g' -e 's/s1/s3/g' s3/sentinel.conf
```

启动：

```
redis-sentinel s1/sentinel.conf
```

RedisTemplate的哨兵模式

在Sentinel集群监管下的Redis主从集群，其节点会因为自动故障转移而发生变化。Redis的客户端必须感知这种变化，及时更新连接信息。Spring的RedisTemplate底层利用lettuce实现了节点的感知和自动切换。

1.引入依赖。这里面会基于lettuce完成各种自动功能。

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

2.在配置文件application.yml中指定sentinel相关信息：

```
spring:
  redis:
    password: abc123 #给redis配置的密码
    sentinel:
      master: mymaster #在sentinel配置文件中指定的master名称
      nodes:           # 指定redis=sentinel集群信息
        - 192.168.14.128:27001
        - 192.168.14.128:27002
        - 192.168.14.128:27003
```

因为redis主从的地址可能会发生变更，所以redis客户端不需要知道redis集群的具体地址，只要知道sentinel的地址就行了。

3.配置主从读写分离

```
@Bean
public LettuceClientConfigurationBuilderCustomizer
clientConfigurationBuilderCustomizer(){
    return clientConfigurationBuilder ->
    clientConfigurationBuilder.readFrom(ReadFrom.REPLICA_PREFERRED);
}
```

这里的ReadFrom是配置Redis的读取策略，是一个枚举，包括下面的选择：

- MASTER：从主节点读取
- MASTER_PREFERRED：优先从master节点读取，master不可用才读取replica
- REPLICA：从slave节点读取
- REPLICA_PREFERRED：优先从slave读取，所有的slave都不可用才读取master。

接着就可以实现业务逻辑了。

分片集群

为了提高主从同步时的性能，单节点的redis设置不能太高，否则内存占用过多，在RDB持久化或全量同步时会产生大量io，导致性能下降。

但是如果单节点redis的内存上线降低了，就满足不了大量数据的存储；

且主从集群只能应对高并发读的问题，如果出现高并发写又该怎么办呢？

-----引入分片集群

分片集群结构（如何解决问题）

分片集群特征：

- 集群中有多个master，每个master保存不同数据
- 每个master都可以有多个slave节点
- master之间通过ping监测彼此健康状态
- 客户端请求可以访问集群任意节点，最终都会被转发到正确节点（节点之间会做一种自动的路由，把请求路由到正确的节点上去访问），故不再需要哨兵机制。

搭建分片集群

准备实例和配置

删除之前的7001、7002、7003这几个目录，重新创建出7001、7002、7003、8001、8002、8003目录：

```
# 进入/tmp目录
cd /tmp
# 删除旧的，避免配置干扰
rm -rf 7001 7002 7003
# 创建目录
mkdir 7001 7002 7003 8001 8002 8003
```

在/tmp下准备一个新的redis.conf文件，内容如下：

```
port 6379
# 开启集群功能
cluster-enabled yes
# 集群的配置文件名称，不需要我们创建，由redis自己维护
cluster-config-file /tmp/6379/nodes.conf
# 节点心跳失败的超时时间
cluster-node-timeout 5000
# 持久化文件存放目录
dir /tmp/6379
# 绑定地址
bind 0.0.0.0
# 让redis后台运行
daemonize yes
# 注册的实例ip
replica-announce-ip 192.168.14.128
# 保护模式
protected-mode no
# 数据库数量
databases 1
# 日志
logfile /tmp/6379/run.log
```

将这个文件拷贝到每个目录下：

```
# 进入/tmp目录
cd /tmp
# 执行拷贝
echo 7001 7002 7003 8001 8002 8003 | xargs -t -n 1 cp redis.conf
```

修改每个目录下的redis.conf，将其中的6379修改为与所在目录一致：

```
# 进入/tmp目录
cd /tmp
# 修改配置文件
printf '%s\n' 7001 7002 7003 8001 8002 8003 | xargs -I{} -t sed -i 's/6379/{}/g'
{}/redis.conf
```

启动

因为已经配置了后台启动模式，所以可以直接启动服务：

```
# 进入/tmp目录
cd /tmp
# 一键启动所有服务
printf '%s\n' 7001 7002 7003 8001 8002 8003 | xargs -I{} -t redis-server
{}/redis.conf
```

通过ps查看状态：

```
ps -ef | grep redis
```

如果要关闭所有进程，可以执行命令：

```
ps -ef | grep redis | awk '{print $2}' | xargs kill
```

或者（推荐这种方式）：

```
printf '%s\n' 7001 7002 7003 8001 8002 8003 | xargs -I{} -t redis-cli -p {} shutdown
```

创建集群

虽然服务启动了，但是目前每个服务之间都是独立的，没有任何关联。我们需要执行命令来创建集群。

```
redis-cli --cluster create --cluster-replicas 1 192.168.14.128:7001
192.168.14.128:7002 192.168.14.128:7003 192.168.14.128:8001 192.168.14.128:8002
192.168.14.128:8003
```

命令说明：

- redis-cli --cluster 或者 ./redis-trib.rb：代表集群操作命令
- create：代表是创建集群
- --replicas 1 或者 --cluster-replicas 1：指定集群中每个master的副本个数为1，此时节点总数 ÷ (replicas + 1) 得到的就是master的数量。因此节点列表中的前n个就是master，其它节点都是slave节点，随机分配到不同master

通过命令可以查看集群状态：

```
redis-cli -p 7001 cluster nodes
```

测试

尝试连接7001节点，存储一个数据：

```
# 连接  
redis-cli -p 7001  
# 存储数据  
set num 123  
# 读取数据  
get num  
# 再次存储  
set a 1
```

结果悲剧了。

集群操作时，需要给 redis-cli 加上 -c 参数才可以：

```
redis-cli -c -p 7001
```

这次可以了。

散列插槽

Redis会把每一个master节点映射到0~16383共16384个插槽（hash slot）上，查看集群信息时能看到映射情况。

数据key不是与节点绑定，而是与插槽绑定。**这是因为**节点可能删除或宕机，数据就丢失了，如果跟插槽绑定，在节点删除或丢失时，可以将插槽交予其他节点。

redis会根据key的有效部分计算插槽值，分两种情况：

- key中包含“{}”，且“{}”中至少包含1个字符，“{}”中的部分是有效部分
- key中不包含“{}”，整个key都是有效部分

例如：key是num，那么就根据num计算，如果是{itcast}num，则根据itcast计算。计算方式是利用CRC16算法得到一个hash值，然后对16384取余，得到的结果就是slot值。

当访问不在本节点的插槽存储的key时，会进行路由，转到正确的节点上查询数据。

思考：如何将同一类数据固定保存在某个redis实例？

答：这一类数据使用相同的有效部分，例如key都已{typeid}作为前缀。

集群伸缩

redis-cli --cluster提供了很多操作集群的命令，可以通过下面的方式查看：

```
redis-cli --cluster help
```

比如，添加节点的命令：

```
redis-cli --cluster add-node new_host:new_port existing_host:existing_port [--cluster-slave --cluster-master-id <arg>]
```

，其中中括号内的内容不写，则添加为master节点，写了则添加为指定节点的slave。

向集群中添加一个新的master节点，并向其中存储 num = 10

需求：

操作举例：

- 启动一个新的redis实例，端口为7004
- 添加7004到之前的集群，并作为一个master节点
- 给7004节点分配插槽，使得num这个key可以存储到7004实例

添加过程很简单，但是添加过后7004中是没有插槽的。那么num原本是属于7001的插槽的，如何将插槽分配给7004？

从7001移出插槽：

```
redis-cli --cluster reshard 192.168.14.128:7001
```

在删除节点的时候，如果是主节点，需要先将插槽转移给其他主节点，才能够进行删除。否则可能会导致数据丢失或不可用。

故障转移

当集群中一个master节点发生宕机：

- 首先是该实例与其它实例失去连接
- 然后是疑似宕机：

```
1fa6d68d590827c24c237b1c490b78e5c7fe2ca9 192.168.150.101:8003@18003 slave f5fc58defbebb957e47fb0d8327a09dc4f1678f5 0 1625207711535 8 connected  
f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001@17001 myself,master - 0 1625207710000 8 connected 0-5460  
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master,fail2 - 1625207705198 1625207703000 10 disconnected 5461-10922  
6ec60fb5af950a465f05c8024bf8f75d89b014 192.168.150.101:8002@18002 slave 1c00e5f9e158b169f199f15884ab43bc433bla06 0 1625207710000 3 connected  
1c00e5f9e158b169f199f15884ab43bc433bla06 192.168.150.101:7003@17003 master - 0 1625207711000 3 connected 10923-16383  
7b6d5ffc9a985d614dc5aeb2ee3abac1adfd3e22 192.168.150.101:8001@18001 slave afaaa70d6528fc72490e0f3f7b32731a12c12bb8 0 1625207709420 10 connected
```

- 最后是确定下线，自动提升一个slave为新的master：

```
1fa6d68d590827c24c237b1c490b78e5c7fe2ca9 192.168.150.101:8003@18003 slave f5fc58defbebb957e47fb0d8327a09dc4f1678f5 0 1625208023157 8  
f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001@17001 myself,master - 0 1625208022000 8 connected 0-5460  
afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002@17002 master,fail - 1625207705198 1625207703000 10 disconnected  
6ec60fb5af950a465f05c8024bf8f75d89b014 192.168.150.101:8002@18002 slave 1c00e5f9e158b169f199f15884ab43bc433bla06 0 1625208021035 3  
1c00e5f9e158b169f199f15884ab43bc433bla06 192.168.150.101:7003@17003 master - 0 1625208022084 3 connected 10923-16383  
7b6d5ffc9a985d614dc5aeb2ee3abac1adfd3e22 192.168.150.101:8001@18001 master - 0 1625208023000 11 connected 5461-10922
```

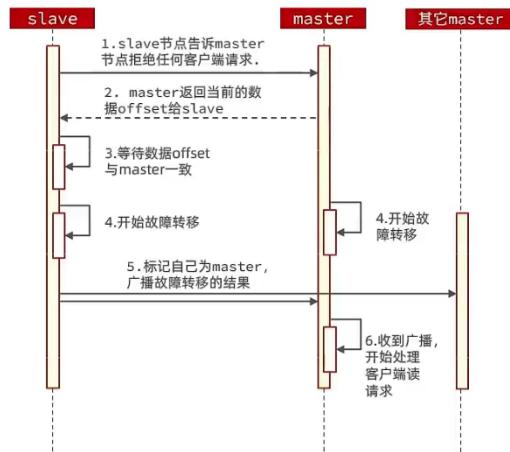
以上是自动故障转移，那么要如何进行手动故障转移呢？

为什么要进行手动故障转移：比如机器7001需要升级，就需要开启一个节点成为7001的slave，然后进行故障转移，而7001会变成该节点的slave，可以让7001下线去进行升级。

利用cluster failover命令可以手动让集群中的某个master宕机，切换到执行cluster failover命令的这个slave节点，实现无感知的数据迁移。其流程如下：

手动的Failover支持三种不同模式：

- 缺省：默认的流程，如图1~6步
- force：省略了对offset的一致性校验
- takeover：直接执行第5步，忽略数据一致性、忽略master状态和其它master的意见



RedisTemplate访问分片集群

RedisTemplate底层同样基于lettuce实现了分片集群的支持，而使用的步骤与哨兵模式基本一致：

1. 引入redis的starter依赖
2. 配置分片集群地址
3. 配置读写分离

与哨兵模式相比，其中只有分片集群的配置方式略有差异，如下：

```
spring:  
  redis:  
    cluster:  
      nodes: # 指定分片集群的每一个节点信息  
        - 192.168.150.101:7001  
        - 192.168.150.101:7002  
        - 192.168.150.101:7003  
        - 192.168.150.101:8001  
        - 192.168.150.101:8002  
        - 192.168.150.101:8003
```

多级缓存

传统缓存的问题

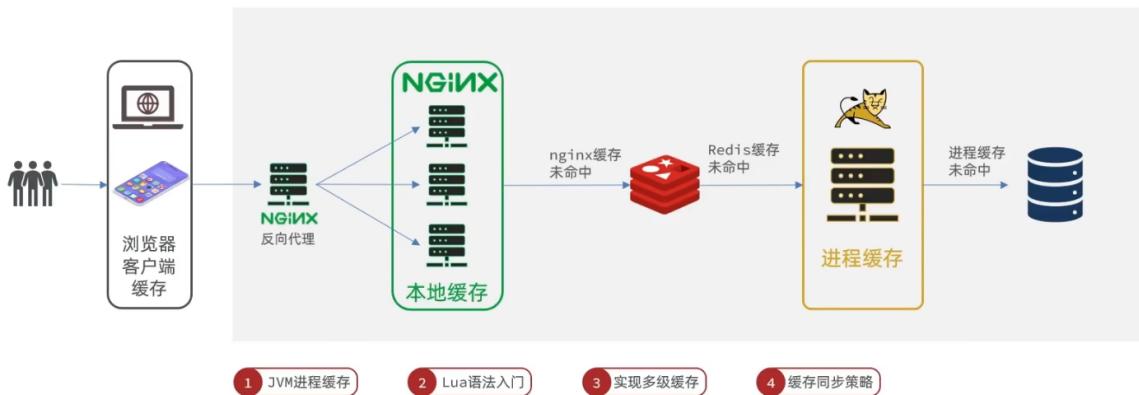
传统的缓存策略一般是请求到达tomcat后，先查询redis，如果未命中则查询数据库，存在下面问题：

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

如何解决：

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。

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：



其中redis、tomcat、mysql也都可以部署成集群。

下面使用商品一个商品的业务实现简单的多级缓存。

JVM进程缓存

导入商品案例

详见 "C:\users\14693\Desktop\note\note-md\redis\案例导入说明.md"

关于nginx部分的相关解释

在nginx目录下运行nginx服务。

运行命令：

```
start nginx.exe
```

然后访问 <http://localhost/item.html?id=10001>。

现在，页面是假数据展示的。我们需要向服务器发送ajax请求，查询商品数据。

打开控制台，可以看到页面有发起ajax查询数据。

```
▼ General
Request URL: http://localhost/api/item/10001
Request Method: GET
Status Code: 502 Bad Gateway
Remote Address: 127.0.0.1:80
Referrer Policy: strict-origin-when-cross-origin
```

这个请求地址是80端口，所以被当前的nginx反向代理了。

查看nginx的conf目录下的nginx.conf文件：

```

upstream nginx-cluster{
    server 192.168.150.101:8081;
}

server {
    listen      80;
    server_name localhost;

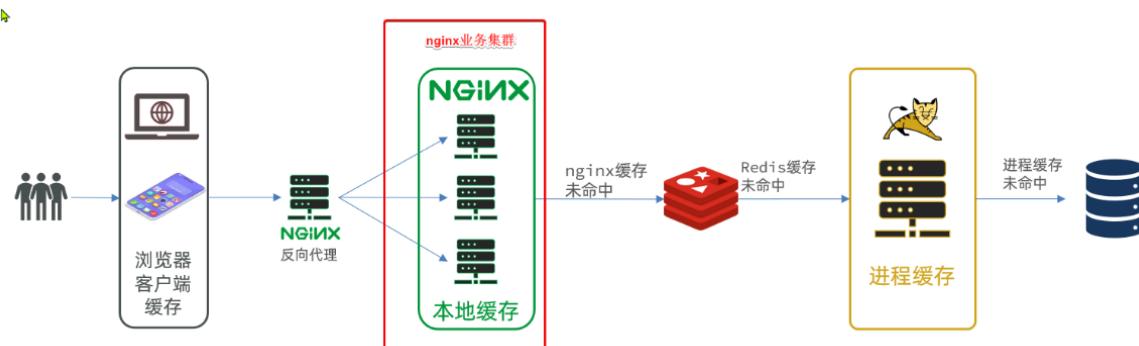
    location /api {
        proxy_pass http://nginx-cluster;
    }
}

```

nginx-cluster集群，就是将来的nginx业务集群

监听/api路径，反向代理到nginx-cluster集群

其中的192.168.150.101是我的虚拟机IP，也就是我的Nginx业务集群要部署的地方：



导入完成后，上图的客户端、nginx反向代理服务器、tomcat服务器、mysql就有了
还差中间的三个缓存。

初识Caffeine

缓存在日常开发中起到至关重要的作用，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、可靠性好、可以被共享
 - 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

Caffeine是一个基于java8开发的，提供了极高命中率的高性能的本地缓存库。目前Spring内部的缓存就是使用的Caffeine。Github地址：<https://github.com/ben-manes/caffeine>

Caffeine存取数据：

```

@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，不存在则去数据库查询
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);
}

```

缓存驱逐策略

- 基于容量：设置缓存的数量上限

```

/*
 基于大小设置驱逐策略:
 */
@Test
void testEvictByNum() throws InterruptedException {
    // 创建缓存对象
    Cache<String, String> cache = Caffeine.newBuilder()
        // 设置缓存大小上限为 1
        .maximumSize(1)
        .build();
    // 存数据
    cache.put("gf1", "柳岩");
    cache.put("gf2", "范冰冰");
    cache.put("gf3", "迪丽热巴");
    // 延迟10ms，给清理线程一点时间
    Thread.sleep(10L);
    // 获取数据
    System.out.println("gf1: " + cache.getIfPresent("gf1"));
    System.out.println("gf2: " + cache.getIfPresent("gf2"));
    System.out.println("gf3: " + cache.getIfPresent("gf3"));
}

```

- 基于时间：设置缓存的有效时间

```

/*
 基于时间设置驱逐策略:
 */
@Test
void testEvictByTime() throws InterruptedException {
    // 创建缓存对象
    Cache<String, String> cache = Caffeine.newBuilder()
        .expireAfterWrite(Duration.ofSeconds(1)) // 设置缓存有效期为
10 秒
}

```

```
        .build();
    // 存数据
    cache.put("gf", "柳岩");
    // 获取数据
    System.out.println("gf: " + cache.getIfPresent("gf"));
    // 休眠一会儿
    Thread.sleep(1200L);
    System.out.println("gf: " + cache.getIfPresent("gf"));
}
```

- 基于引用：java有内存回收机制，如果一个对象没人应用，就会回收。而Caffeine底层可以基于软引用或弱引用来配置key和value，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。

实现进程缓存

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



提前将bean定义好：

```
@Configuration
public class CaffeineConfig {
    @Bean
    public Cache<Long, Item> itemCache() {
        return Caffeine.newBuilder()
            .initialCapacity(100)
            .maximumSize(10000)
            .build();
    }
    @Bean
    public Cache<Long, ItemStock> stockCache() {
        return Caffeine.newBuilder()
            .initialCapacity(100)
            .maximumSize(10000)
            .build();
    }
}
```

自动注入：

```
@Autowired  
private Cache<Long,Item> itemCache;  
@Autowired  
private Cache<Long,ItemStock> itemStockCache;
```

使用：

```
@GetMapping("/{id}")  
public Item findById(@PathVariable("id") Long id){  
    return itemCache.get(id, key->itemService.query()  
        .ne("status", 3).eq("id", id)  
        .one());  
}  
  
@GetMapping("/stock/{id}")  
public ItemStock findStockById(@PathVariable("id") Long id){  
    return stockCache.get(id, key->stockService.getById(id));  
}
```

Nginx本地缓存(1)

Lua

接下来要做nginx的业务集群，在其中实现本地缓存。

所以要用nginx+lua开发

初始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/>

linux运行lua：

```
lua xxx.lua
```

变量与循环

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：

```
-- 声明字符串
local str = 'hello'
-- 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：

```
-- 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)
```

数组其实也是table的一种，key为数字，数组下标从1开始。

String用 .. 拼接。

local声明的是局部变量，不加local则为全局变量。

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
```

其中for后面跟的是变量名，前者为key，后者为value。

ipairs表示解析数组，pairs表示解析table。

条件控制、函数

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。

举例

```
print("hello world!")

local arr={"java","lua","hello"}
local map={name="jack",age=21}

local function printArr(arr)
    if (not arr) then
        print("数组不能为空!")
        return nil
    end
    for i,val in pairs(arr) do
        print(val)
    end
end

printArr(arr)
printArr(nil)

for key,value in pairs(map) do
    print(key,value)
end
```

OpenResty

安装

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

安装及配置：

```
"C:\Users\14693\Desktop\note\note-md\redis\安装OpenResty.md"
```

这里开放的是8081端口。

快速入门

在【导入商品案例】笔记中，我们已经知道，访问<http://localhost/item.html?id=10001>其实是访问<http://localhost/api/item/10001>，而在nginx的配置文件中配置了访问[http://localhost/api/...](http://localhost/api/)的端口，就是访问 192.168.14.128:8081，而这个8081已经在上一篇笔记中被我们配置为linux上的openresty。因此我们要在openresty中接收请求。

我们这里仅以返回假数据为例。

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;
}
```

```
#user    nobody;
worker_processes  1;
error_log  logs/error.log;

events {
    worker_connections  1024;
}

http {
    include      mime.types;
    default_type application/octet-stream;
    sendfile     on;
    keepalive_timeout  65;
    #lua 模块
    lua_package_path "/usr/local/openresty/lualib/?.lua;;";
    #c模块
    lua_package_cpath "/usr/local/openresty/lualib/?.so;;";

    server {
        listen      8081;
        server_name localhost;
        location /api/item {
            default_type application/json;
        }
    }
}
```

```
        content_by_lua_file lua/item.lua;
    }

    location / {
        root    html;
        index  index.html index.htm;
    }
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
        root    html;
    }
}

}
```

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
```

```
ngx.say('{"id":10001,"name":"SALSA AIR","title":"RIMOWA 26寸托运箱拉杆箱 SALSA AIR
系列果绿色
820.70.36.4","price":19900,"image":"https://m.360buyimg.com/mobilecms/s720x720_j
fs/t6934/364/1195375010/84676/e9f2c55f/597ece38N0ddcbc77.jpg!q70.jpg.webp","cate
gory":"拉杆箱","brand":"RIMOWA","spec":"","status":1,"createTime":"2019-04-
30T16:00:00.000+00:00","updateTime":"2019-04-
30T16:00:00.000+00:00","stock":2999,"sold":31290}')
```

请求参数处理

上面示例是返回的一个假数据（写死的数据）。但在实际开发中不可能这么写，一定会根据请求查询不同的商品。而第一个问题就是：如何获取不同请求参数。

OpenResty提供了各种API用来获取不同类型的需求参数：

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 queryParams = 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()

所以，修改nginx.conf：

```
location ~ /api/item/(\d+) {  
    default_type application/json;  
    content_by_lua_file lua/item.lua;  
}
```

修改item.lua：

```
-- 获取路径参数  
local id=ngx.var[1]  
-- 返回结果  
ngx.say('{"id": ' .. id ..',"name":"SALSA AIR","title":"RIMOWA 26寸托运箱拉杆箱  
SALSA AIR系列果绿色  
820.70.36.4","price":19900,"image":"https://m.360buyimg.com/mobilecms/s720x720_j  
fs/t6934/364/1195375010/84676/e9f2c55f/597ece38N0ddcbc77.jpg!q70.jpg.webp","cate  
gory":"拉杆箱","brand":"RIMOWA","spec":"","status":1,"createTime":"2019-04-  
30T16:00:00.000+00:00","updateTime":"2019-04-  
30T16:00:00.000+00:00","stock":2999,"sold":31290}')
```

查询tomcat完成页面渲染

下面根据id信息向tomcat查询商品信息

这里要修改item.lua，满足下列需求：

- 根据id向tomcat服务器发送请求，查询商品信息
- 根据id向tomcat发送请求，查询库存信息
- 组装商品信息、库存信息，序列化为JSON格式并返回

修改nginx.conf：

将请求反向代理到windows的tomcat，而windows的tomcat可能有商品缓存，没有则到linux的数据库去查。

```
location /item {
    proxy_pass http://192.168.14.1:8081;
}
```

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

```
-- 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
```

修改item.lua：

```
--导入common函数库
local common=require('common')
local read_http=common.read_http
local cjson=require('cjson')
--获取路径参数
local id=ngx.var[1]

--查询商品信息
local itemJSON=read_http("/item/" .. id,nil)

--查询库存信息
local stockJSON=read_http("/item/stock/" .. id,nil)

--json转化成lua对象
local item=cjson.decode(itemJSON)
local stock=cjson.decode(stockJSON)

--组合数据
item.stock=stock.stock
item.sold=stock.sold
--把item序列化为json返回结果
ngx.say(cjson.encode(item))
```

根据id对tomcat集群负载均衡

tomcat如果按集群部署，那么linux上nginx的配置就要修改：

```
# 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{
    server 192.168.150.1:8081;
    server 192.168.150.1:8082;
}
```

这样nginx会自动对tomcat进行轮询的负载均衡。

但是由于是本地缓存，每个tomcat无法共享缓存，如果进行轮询，在每次访问同一页面，可能不是同一个tomcat，这样就会产生冗余的缓存。

所以我们要将同一个请求指向同一个tomcat服务器（在这里就是查询同一个id的商品时应该查询同一个tomcat服务器）。即修改nginx的负载均衡算法。

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

Redis缓存

Redis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。

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 ... }
```

初始化类：

```
@Component
public class RedisHandler implements InitializingBean {
    @Autowired
    private StringRedisTemplate redisTemplate;
    @Autowired
    private IItemStockService stockService;
    @Autowired
    private IItemService itemService;

    private static final ObjectMapper MAPPER = new ObjectMapper();

    @Override
    public void afterPropertiesSet() throws Exception {
        //初始化缓存
        //1.查询商品信息
        List<Item> itemList=itemService.list();
        //2.放入缓存
        for (Item item : itemList) {
            //2.1把item序列化为json
            String json = MAPPER.writeValueAsString(item);
            //2.2存入redis
            redisTemplate.opsForValue().set("item:id"+item.getId(),json);
        }
        //3.查询商品库存信息
        List<ItemStock> stockList=stockService.list();
        //4.放入缓存
        for (ItemStock stock : stockList) {
            //2.1把stock序列化为json
            String json = MAPPER.writeValueAsString(stock);
            //2.2存入redis
            redisTemplate.opsForValue().set("item:stock:id"+stock.getId(),json);
        }
    }
}
```

先查询redis，未命中再查询tomcat

OpenResty提供了操作Redis的模块，我们只要引入改模块就能直接使用：（都写在common.lua中）

- 引入Redis模块，并初始化Redis对象

```
-- 导入redis
local redis=require('resty.redis')
-- 初始化redis
local red=redis:new()
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
```

- 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
```

- 将方法导出方便别处引用：

```
-- 将方法导出
local _M = {
    read_http = read_http,
    read_redis=read_redis
}
```

修改item.lua:

- 导入函数

```
local read_redis=common.read_redis
```

- 封装查询函数并修改查询信息的语句

```
function read_data(key,path,params)
    -- 查询redis
    local resp=read_redis("192.168.14.128",6379,key)
    -- 判断查询结果
    if not resp then
        ngx.log(ngx.ERR,"redis查询失败, 尝试插叙http, key:",key)
        -- redis查询失败, 去查询http
        resp=read_http(path,params)
    end
    return resp
end

-- 获取路径参数
local id=ngx.var[1]

-- 查询商品信息
local itemJSON=read_data("item:id:" .. id,"/item/" .. id,nil)

-- 查询库存信息
local stockJSON=read_data("item:stock:id:" .. id,"/item/stock/" .. id,nil)
```

然后, 即使停掉tomcat, 也可以获取数据 (从redis)

Nginx本地缓存(2)

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')
```

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

- 开启共享词典，在nginx.conf的http下添加配置：

```
lua_shared_dict item_cache 50m;
```

- 在item.lua中写入：

```
local item_cache=ngx.shared.item_cache
```

下面，修改逻辑：

- 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

注意：数据特别敏感（变化快且重要）不要存进nginx本地缓存，因为nginx缓存策略是到期更新，如果在到期之前发生变化，那么在查询时就会发生不一致

实现：

```
function read_data(key, expire, path, params)
    -- 查询nginx本地缓存
    local val=item_cache:get(key)
    if not val then
        ngx.log(ngx.ERR, "本地缓存查询失败，尝试查询redis, key:", key)
        -- 查询redis
        val=read_redis("192.168.14.128", 6379, key)
        -- 判断查询结果
        if not val then
            ngx.log(ngx.ERR, "redis查询失败，尝试插叙http, key:", key)
            -- redis查询失败，去查询http
            val=read_http(path,params)
        end
    end
end
```

```
-- 查询成功，把数据写入本地缓存
item_cache:set(key, val, expire)
-- 返回数据
return val
end

--获取路径参数
local id=ngx.var[1]

--查询商品信息
local itemJSON=read_data("item:id:" .. id, 1800, "/item/" .. id, nil)

--查询库存信息
local stockJSON=read_data("item:stock:id:" .. id, 60, "/item/stock/" .. id, nil)
```

缓存同步

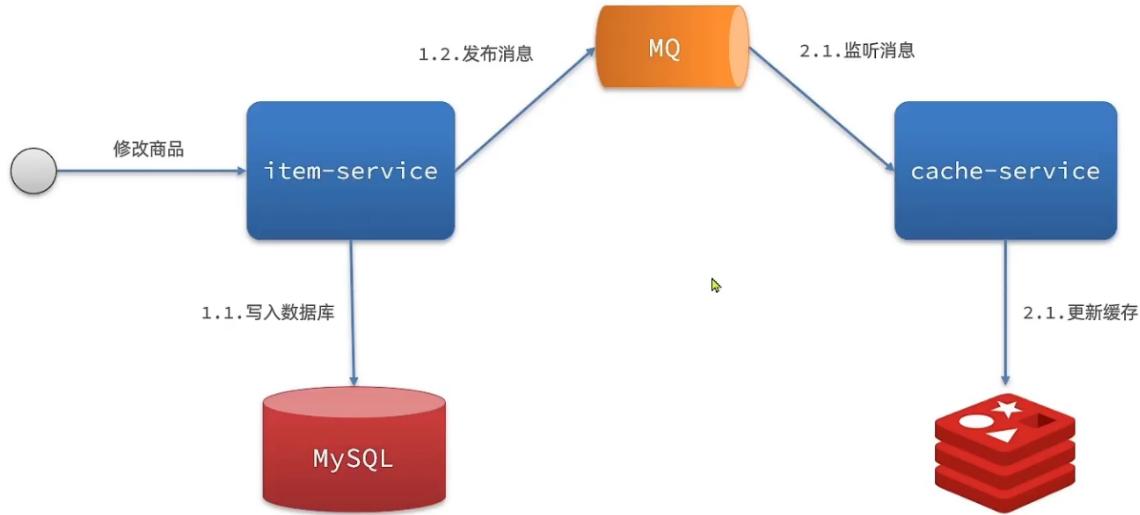
数据同步策略

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

基于mq的异步通知：

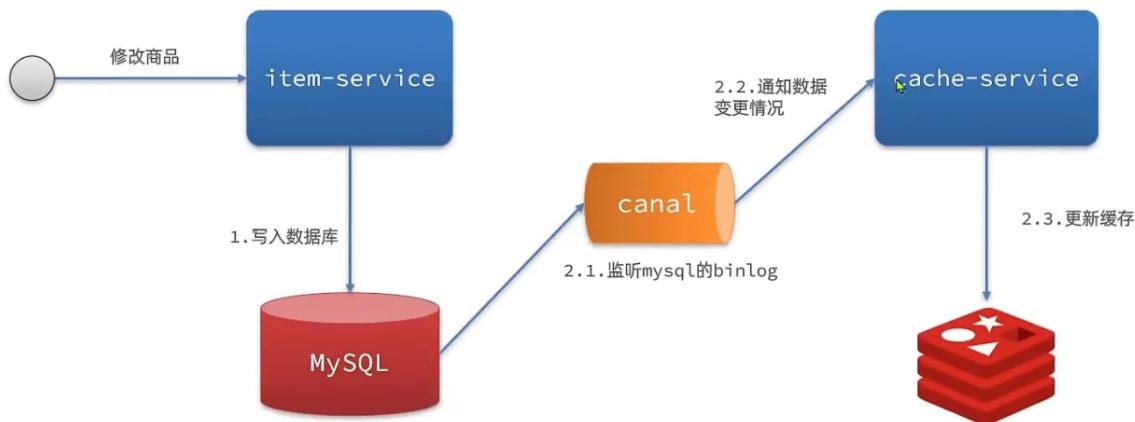
基于MQ的异步通知：



我们这里采用另一种进行举例：

基于Canal的异步通知：

基于Canal的异步通知：



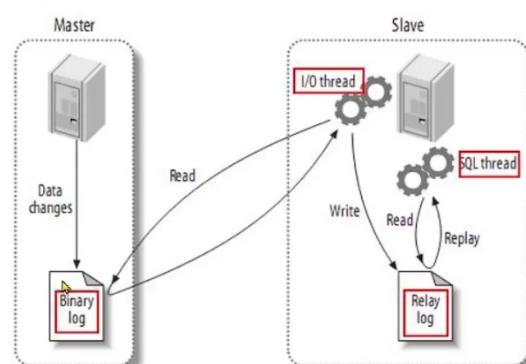
openresty的缓存时设置有效期进行更新的，我们接下来只对redis、JVM缓存、mysql作缓存一致性。

Canal

Canal基于数据库增量解析，提供增量数据订阅&消费。

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



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，进而完成对其他地方数据的同步。

安装参考 "c:\Users\14693\Desktop\note\note-md\redis\安装Canal.md"

监听Canal实现缓存同步

参考https://www.bilibili.com/video/BV1cr4y1671t?p=132&vd_source=c054be8430afebb3d00e5f2d0b77f9fc

多级缓存总结

多级缓存总结

