美团一面：项目中使用过Redis吗？我说用Redis做缓存。他对我哦了一声
==========

引言

Redis，作为一种开源的、基于内存且支持持久化的键值存储系统，以其卓越的性能、丰富灵活的数据结构和高度可扩展性在全球范围内广受欢迎。Redis不仅提供了一种简单直观的方式来存储和检索数据，更因其支持数据结构如字符串、哈希、列表、集合、有序集合等多种类型，使得其在众多场景下表现出强大的适用性和灵活性。

Redis的核心特点包括：

1. **高性能**：基于内存操作，读写速度极快，特别适用于对性能要求高的实时应用。

> 关于Redis高性能的原因，请参考：[京东二面：Redis为什么快？我说Redis是纯内存操作的，然后他对我笑了笑。。。。。。](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkyNzYzMTY0MA==&mid=2247484198&idx=1&sn=32b0992f1b7d9862e2c983f9cb2e633f&chksm=c2245351f553da472f19d57c7d7dfa56c58e8ea4a9e814fc2d4a8fadf72afeadae4b0b0bec35#rd”)

2. **数据持久化**：支持RDB和AOF两种持久化方式，确保即使在服务器重启后也能恢复数据。
3. **分布式的特性**：通过主从复制、哨兵模式或集群模式，Redis可以轻松地构建高可用和可扩展的服务。
4. **丰富的数据结构**：提供了多种数据结构支持，便于开发人员根据实际需求进行数据建模和处理。

Redis的广泛应用跨越了多个行业和技术领域，诸如网站加速、缓存服务、会话管理、实时统计、排行榜、消息队列、分布式锁、社交网络功能、限流控制等。本文将深入探讨Redis在这些场景下的具体应用方法及其背后的工作原理，旨在帮助开发者更好地理解和掌握Redis，以应对各种复杂的业务需求，并充分发挥其潜能。同时，我们也将如何在实践中平衡Redis的性能、安全性、一致性等方面的挑战，为实际项目带来更高的价值。

数据缓存

在高并发访问的场景下，数据库经常成为系统的瓶颈。Redis因其内存存储、读取速度快的特点，常被用作数据库查询结果的缓存层，有效降低数据库负载，提高整体系统的响应速度。这也是我们使用场景频率最高的一个。

通常我们选择使用String类型来存储数据库查询结果，如单个实体对象的JSON序列化形式。

```
@Service
public class ProductService {

    @Autowired
    private RedisTemplate<String, Product> redisTemplate;

    // 使用@Cacheable注解进行缓存
    @Cacheable(value = "productCache", key = "#id")
    public Product getProductById(String id) {
        // 此处是从数据库或其他数据源获取商品的方法
        // 在实际场景中，如果缓存命中，则不会执行下面的数据库查询逻辑
        return getProductFromDatabase(id);
    }
}
```

而使用Redis作为缓存使用时，有一些特别需要注意的事项：

1. **缓存穿透**：当查询的数据在数据库和缓存中均不存在时，可能会导致大量的无效请求直接打到数据库。可通过布隆过滤器预防缓存穿透。
2. **缓存雪崩**：若大量缓存在同一时刻失效，所有请求都会涌向数据库，造成瞬时压力过大。可通过设置合理的过期时间分散、预加载或采用Redis集群等方式避免。
3. **缓存一致性**：当数据库数据发生变化时，需要及时更新缓存，避免数据不一致。可以采用主动更新策略（如监听数据库binlog）或被动更新策略（如在读取时判断数据新鲜度）。

而对于数据缓存，我们常使用的业务场景如热点数据存储、全页缓存等。

会话管理

在说会话管理之前，我们来简单介绍一下`Spring Session`。

Spring Session 是 Spring Framework 的一个项目，旨在简化分布式应用程序中的会话管理。在传统的基于 Servlet 的应用程序中，会话管理是通过 HttpSession 接口实现的，但在分布式环境中，每个节点上的 HttpSession 不能简单地共享，因此需要一种机制来管理会话并确保会话在集群中的一致性。

Spring Session 提供了一种简单的方法来解决这个问题，它将会话数据从容器（如 Tomcat 或 Jetty）中分离出来，并存储在外部数据存储（如 Redis、MongoDB、JDBC 等）中。这样，不同节点上的应用程序实例可以共享相同的会话数据，实现分布式环境下的会话管理。

所以在Web应用中，Redis用于会话管理时，可以取代传统基于服务器内存或Cookie的会话存储方案。通过将会话数据序列化后存储为Redis中的键值对，实现跨多个服务器实例的会话共享。

```
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
    <version>3.2.0</version>
</dependency>
```

然后我们在启动类中，使用`@EnableRedisHttpSession`启用Redis作为会话存储。

```
@Configuration
@EnableRedisHttpSession
public class RedisSessionConfig {

    @Bean
    public RedisConnectionFactory connectionFactory() {
        // 这里假设你已经在application.properties或application.yml中配置了Redis的信息
        // 根据实际情况填写Redis服务器地址、端口等信息
        return new LettuceConnectionFactory();
```

```
    }

}
```

以上是一个简单的`Spring Session`使用`Redis`进行会话管理的示例代码。通过这种方式，我们可以轻松地在分布式环境中管理会话，并确保会话数据的一致性和可靠性。如果需要了解一些具体的用法，请自行参考`Spring Session`。

排行榜与计分板

有序集合（Sorted Sets）是Redis的一种强大数据结构，可以用来实现动态排行榜，每个成员都有一个分数，按分数排序。有序集合中的每一个成员都有一个分数（score），成员依据其分数进行排序，且成员本身是唯一的。

当需要给某个用户增加积分或改变其排名时，可以使用`ZADD`命令向有序集合中添加或更新成员及其分数。例如，`ZADD leaderboard score member`，这里的`ranking`是有序集合的名称，`score`是用户的积分值，`member`是用户ID。

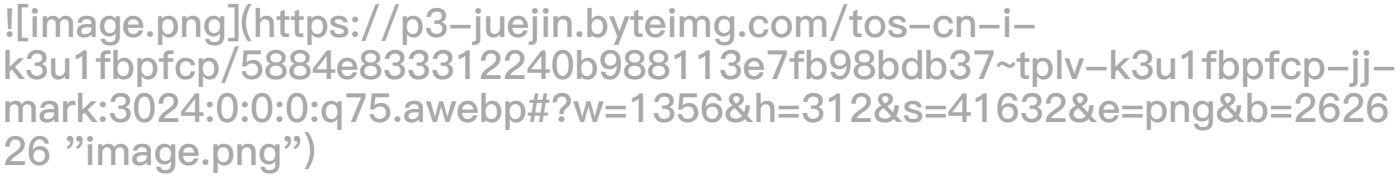
查询排行榜时，可以使用`ZRANGE`命令获取指定范围内的成员及其分数，例如，`ZRANGE ranking 0 -1 WITHSCORES`，这条命令会返回集合中所有的成员及其对应的分数，按照分数从低到高排序。

若要按照分数从高到低显示排行榜，使用`ZREVRANGE`命令，如`ZREVRANGE ranking 0 -1 WITHSCORES`。

```
@Service
public class RankingService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    public void addToRanking(String playerName, int score) {
        redisTemplate.opsForZSet().add(”ranking”, playerName, score);
    }

    public List<RankingInfo> getRanking() {
        List<RankingInfo> rankingInfos = new ArrayList<>();
```

```
        Set<ZSetOperations.TypedTuple<String>> rankingSet = redisTempl
ate.opsForZSet().rangeWithScores("ranking", 0, -1);
        for (ZSetOperations.TypedTuple<String> tuple : rankingSet) {
            RankingInfo rankingInfo = new RankingInfo();
            rankingInfo.setPlayerName(tuple.getValue());
            rankingInfo.setScore(tuple.getScore().intValue());
            rankingInfos.add(rankingInfo);
            System.out.println("playerName: " + tuple.getValue() + ", score:
" + tuple.getScore().intValue());
        }
        return rankingInfos;
    }
}
```

我们模拟请求，往redis中填入一些数据，在获取排行榜：



在实际场景中，有序集合非常适合处理实时动态变化的排行榜数据，比如京东的月度销量榜单、商品按时间的上新排行榜等，因为它的更新和查询操作都是原子性的，并且能高效地支持按分数排序的操作。

计数器与统计

Redis的原子性操作如`INCR`和`DECR`可以用于计数，确保在高并发环境下的计数准确性。比如在流量统计、电商网站商品的浏览量、视频网站视频的播放数赞等场景的应用。

```
@Service
public class CounterService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;
```

```
    public void incrementLikeCount(String postId) {
        redisTemplate.opsForValue().increment(postId + ":likes");
    }

    public void decrementLikeCount(String postId) {
        redisTemplate.opsForValue().decrement(postId + ":likes");
    }

    public long getLikeCount(String postId) {
        String value = redisTemplate.opsForValue().get(postId + ":likes");
        return StringUtils.isBlank(value) ? 0 : Long.parseLong(value);
    }
}
```

在使用Redis实现点赞，统计等功能时一定要考虑设置计数值的最大值或最小值限制，以及过期策略。

分布式锁

分布式锁

Redis的`SETNX`（设置并检查是否存在）和`EXPIRE`命令组合可以实现分布式锁，因其操作时原子性的，所以可以确保在分布式环境下同一资源只能被一个客户端修改。

使用 Redis 实现分布式锁通常会使用 Redis 的 SETNX 命令。这个命令用于设置一个键的值，如果这个键不存在的话，它会设置成功并返回 1，如果这个键已经存在，则设置失败并返回 0。结合 Redis 的 EXPIRE 命令，可以为这个键设置一个过期时间，确保即使获取锁的客户端异常退出，锁也会在一段时间后自动释放。

```
@Component
public class DistributedLock {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    public boolean acquireLock(String lockKey, String requestId, long expi
```

```
reTime) {
        Boolean result = redisTemplate.opsForValue().setIfAbsent(lockKey,
requestId, expireTime);
        return result != null && result;
    }

    public void releaseLock(String lockKey, String requestId) {
        String value = redisTemplate.opsForValue().get(lockKey);
        if (value != null && value.equals(requestId)) {
            redisTemplate.delete(lockKey);
        }
    }
}
```

使用分布式锁时，务必确保在加锁和解锁操作之间处理完临界区代码，否则可能出现死锁。并且要注意锁定超时时间应当合理设置，以避免锁定资源长时间无法释放。

> 关于分布式锁，推荐使用一些第三方的分布式锁框架，例如Redisson

全局ID

在全局ID生成的场景中，我们可以使用 Redis 的原子递增操作来实现。通过对 Redis 中的一个特定的 key 进行原子递增操作，可以确保生成的ID是唯一的。

```
@Component
public class UniqueIdGenerator {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    public long generateUniqueId(String key) {
        return redisTemplate.opsForValue().increment(key, 1);
    }
}
```

库存扣减

在扣减库存的场景中，我们可以使用 Redis 的原子递减操作来实现。将库存数量存储在 Redis 的一个特定key中（例如仓库编码:SKU），然后通过递减操作来实现库存的扣减。这样可以保证在高并发情况下，库存扣减的原子性。

```
@Component
public class StockService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**商品库存的key*/
    private static final String STOCK_PREFIX = ”stock:%s:%s”;

    /**
     * 扣减库存
     * @param warehouseCode
     * @param productId
     * @param quantity
     * @return
     */
    public boolean decreaseStock(String warehouseCode, String productI
d, long quantity) {
        String key = String.format(STOCK_PREFIX, warehouseCode, produc
tId);
        Long stock = redisTemplate.opsForValue().decrement(key, quantity)
;
        return stock >= 0;
    }
}
```

秒杀

在秒杀场景中，使用Lua脚本。Lua 脚本可以在 Redis 服务器端原子性地执行多个命令，这样可以避免在多个命令之间出现竞态条件。

我们使用Lua脚本来检查库存是否足够并进行扣减操作。如果库存足够，则减少库存并返回 true；如果库存不足，则直接返回 false。通过 Lua 脚本的原子性执行，可以确保在高并发情况下，库存扣减操作的正确性和一致性。

我们先定义一个扣减库存的lua脚本，使用Lua脚本一次性执行获取库存、判断库存是否充足以及扣减库存这三个操作，确保了操作的原子性

```
-- 获取Lua脚本参数：商品ID和要购买的数量
local productId = KEYS[1]
local amount = tonumber(ARGV[1])

-- 获取当前库存
local currentStock = tonumber(redis.call('GET', 'seckill:product:'..productId))

-- 判断库存是否充足
if currentStock <= 0 or currentStock < amount then
    return 0
end

-- 扣减库存
redis.call('DECRBY', 'seckill:product:'..productId, amount)

-- 返回成功标志
return 1
```

然后在秒杀服务中使用Redis的`DefaultRedisScript`执行lua脚本，完成秒杀

```
@Component
public class SeckillService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**
     * 初始化RedisScript对象
     */
    private final DefaultRedisScript<Long> seckillScript = new DefaultRedisScript<>();
    {
        seckillScript.setLocation(new ClassPathResource("rate_limiter.lua"));
        seckillScript.setResultType(Long.class);
```

```
    }

    public boolean seckillyLua(String productId, int amount){
        // 设置Lua脚本参数
        List<String> keys = Collections.singletonList(productId);
        List<String> args = Collections.singletonList(Integer.toString(amou
nt));

        // 执行Lua脚本
        Long result = redisTemplate.execute(seckillScript, keys, args);

        // 如果执行结果为1，表示秒杀成功
        return Objects.equals(result, 1L);
    }
}
```

关于秒杀场景，我们也可以使用`WATCH`命令监视库存键，然后尝试获取并扣减库存。如果在`WATCH`之后、`EXEC`之前库存发生了变化，`exec`方法会返回`null`，此时我们取消`WATCH`并重新尝试整个流程，直到成功扣减库存为止。这样就实现了基于Redis乐观锁的秒杀场景，有效防止了超卖现象。

```
/**
     * 秒杀方法
     * @param productId 商品ID
     * @param amount 要购买的数量
     * @return 秒杀成功与否
     */
    @Transactional(rollbackFor = Exception.class)
    public boolean seckilByWatch(String productId, int amount) {
        // 乐观锁事务操作
        while (true) {
            // WATCH指令监控库存键
            redisTemplate.watch("stock:" + productId);

            // 获取当前库存
            String currentStockStr = redisTemplate.opsForValue().get("stock
:" + productId);
            if (currentStockStr == null) {
                // 库存不存在，可能是商品已售罄或异常情况
                return false;
            }
            int currentStock = Integer.parseInt(currentStockStr);
```

```
            // 判断库存是否充足
            if (currentStock < amount) {
                // 库存不足，取消WATCH并退出循环
                redisTemplate.unwatch();
                return false;
            }

            // 开启Redis事务
            redisTemplate.multi();

            // 执行扣减库存操作
            redisTemplate.opsForValue().decrement("stock:" + productId, a
mount);

            // 执行其他与秒杀相关的操作，如增加订单、更新用户余额等...

            // 提交事务，如果在此期间库存被其他客户端修改，则exec返回null
            List<Object> results = redisTemplate.exec();

            // 如果事务执行成功，跳出循环
            if (!results.isEmpty()) {
                return true;
            }
        }
    }
```

消息队列与发布/订阅

Redis的发布/订阅（Pub/Sub）模式，可以实现一个简单的消息队列。发布/订阅模式允许消息的发布者（发布消息）和订阅者（接收消息）之间解耦，消息的发布者不需要知道消息的接收者是谁，从而实现了一对多的消息传递。

首先我们需要定义一个消息监听器，我们可以实现这个借口并实现其中的方法来处理接收到的消息。这样可以根据具体的业务需求来定义消息的处理逻辑。

```
public interface MessageListener {
    void onMessage(String channel, String message);
}
```

然后我们就可以定义消息的生产者以及消费者。`publish` 方法用于向指定频道发布消息，我们使用 RedisTemplate 的 `convertAndSend` 方法来发送消息到指定的频道。

而`subscribe`方法用于订阅指定的频道，并设置消息监听器。当有消息发布到指定的频道时，消息监听器会收到消息并进行处理。

```
@Component
public class MessageQueue {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    public void publish(String channel, String message) {
        redisTemplate.convertAndSend(channel, message);
    }

    public void subscribe(String channel, MessageListener listener) {
        redisTemplate.getConnectionFactory().getConnection().subscribe((
message, pattern) -> {
            listener.onMessage(channel, message);
        }, channel.getBytes());
    }
}
```

使用Redis的发布订阅模式实现一个轻量级的队列时要注意：Pub/Sub是非持久化的，一旦消息发布，没有订阅者接收的话，消息就会丢失。还有就是Pub/Sub不适合大规模的消息堆积场景，因为它不保证消息顺序和重复消费，更适合实时广播型消息推送。

社交网络

在社交网络中，Redis可以利用集合(Set)、哈希(Hash)和有序集合(Sorted Set)等数据结构构建用户关系图谱。

使用哈希（Hash）数据结构存储用户的个人资料信息，每个用户对应一个哈希表，其中包含用户的各种属性，比如用户名、年龄、性别等。

```
@Component
public class RelationshipGraphService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**用户资料*/
    private static final String USER_PROFILE_PREFIX = "user_profile:";

    /**
     * 存储用户个人资料
     * @param userId
     * @param profile
     */
    public void setUserProfile(String userId, Map<String, String> profile) {
        String key = USER_PROFILE_PREFIX + userId;
        redisTemplate.opsForHash().putAll(key, profile);
    }

    /**
     * 获取用户个人资料
     * @param userId
     * @return
     */
    public Map<Object, Object> getUserProfile(String userId) {
        String key = USER_PROFILE_PREFIX + userId;
        return redisTemplate.opsForHash().entries(key);
    }
}
```

使用集合（Set）数据结构来存储用户的好友关系。每个用户都有一个集合，其中包含了他的所有好友的用户ID。

```
@Component
public class RelationshipGraphService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

     /**用户好友*/
    private static final String FRIENDS_PREFIX = "friends:";
```

```
    /**
     * 添加好友关系
     * @param userId
     * @param friendId
     */
    public void addFriend(String userId, String friendId) {
        String key = FRIENDS_PREFIX + userId;
        redisTemplate.opsForSet().add(key, friendId);
    }

    /**
     * 获取用户的所有好友
     * @param userId
     * @return
     */
    public Set<String> getFriends(String userId) {
        String key = FRIENDS_PREFIX + userId;
        return redisTemplate.opsForSet().members(key);
    }
}
```

同理，我们还可以实现点赞的业务场景

```
@Service
public class LikeService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**
     * 点赞
     * @param objectId
     * @param userId
     */
    public void like(String objectId, String userId) {
        // 将点赞人放入zset中
        redisTemplate.opsForSet().add(getLikeKey(objectId), userId);
    }

    /**
     * 取消点赞
     * @param objectId
     * @param userId
```

```
     */
    public void unlike(String objectId, String userId) {
        // 减少点赞人数
        redisTemplate.opsForSet().remove(getLikeKey(objectId), userId);
    }

    /**
     * 是否点赞
     * @param objectId
     * @param userId
     * @return
     */
    public Boolean isLiked(String objectId, String userId) {
        return redisTemplate.opsForSet().isMember(getLikeKey(objectId), u
serId);
    }

    /**
     * 获取点赞数
     * @param objectId
     * @return
     */
    public Long getLikeCount(String objectId) {
       return redisTemplate.opsForSet().size(getLikeKey(objectId));
    }

    /**
     * 获取所有点赞的用户
     * @param objectId
     * @return
     */
    public Set<String> getLikedUsers(String objectId) {
        return redisTemplate.opsForSet().members(getLikeKey(objectId));
    }

    private String getLikeKey(String objectId) {
        return ”likes:” + objectId;
    }

}
```

使用有序集合（Sorted Set）数据结构来存储用户的者列表。有序集合中的成员是者的用户ID，而分数可以是时间或者其他指标，比如活跃度。

```
@Component
public class RelationshipGraphService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**用户者*/
    private static final String FOLLOWERS_PREFIX = "followers:";

    /**
     * 添加者
     * @param userId
     * @param followerId
     * @param score
     */
    public void addFollower(String userId, String followerId, double score)
{
        String key = FOLLOWERS_PREFIX + userId;
        redisTemplate.opsForZSet().add(key, followerId, score);
    }

    /**
     * 获取用户的者列表（按照时间排序）
     * @param userId
     * @return
     */
    public Set<String> getFollowers(String userId) {
        String key = FOLLOWERS_PREFIX + userId;
        return redisTemplate.opsForZSet().range(key, 0, -1);
    }

}
```

除此之外，我们还可以实现可能认识的人，共同好友等业务场景。

限流与速率控制

Redis可以精确地实施限流策略，如使用INCR命令结合Lua脚本实现滑动窗口限流。

创建一个Lua脚本，该脚本负责检查在一定时间段内请求次数是否超过限制。

```
-- rate_limiter.lua
local key = KEYS[1]
local limit = tonumber(ARGV[1])
local timeWindow = tonumber(ARGV[2]) -- 时间窗口，例如单位为秒

-- 获取当前时间戳
local currentTime = redis.call('TIME')[1]

-- 获取最近timeWindow秒内的请求次数
local count = redis.call('ZCOUNT', key .. ':requests', currentTime -
 timeWindow, currentTime)

-- 如果未超过限制，则累加请求次数，并返回true
if count < limit then
  redis.call('ZADD', key .. ':requests', currentTime, currentTime)
  return 1
else
  return 0
end
```

限流服务中Redis使用`DefaultRedisScript`执行Lua脚本

```
@Component
public class RateLimiter {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**限流Key*/
    private static final String TATE_LIMITER_KEY = "rate-limit:%s";

    /**规定的时间窗口内允许的最大请求数量*/
    private static final Integer LIMIT = 100;

    /**限流策略的时间窗口长度，单位是秒*/
    private static final Integer TIME_WINDOW = 60;

    /**
     * 初始化RedisScript对象
     */
```

```
    private final DefaultRedisScript<Long> rateLimiterScript = new Defaul
tRedisScript<>();
    {
        rateLimiterScript.setLocation(new ClassPathResource("rate_limiter.
lua"));
        rateLimiterScript.setResultType(Long.class);
    }


    /**
     * 限流方法 1分钟内最多100次请求
     * @param userId
     * @return
     */
    public boolean allowRequest(String userId) {
        String key = String.format(TATE_LIMITER_KEY, userId);
        List<String> keys = Collections.singletonList(key);
        List<String> args = Arrays.asList(String.valueOf(LIMIT), String.value
Of(TIME_WINDOW));

        // 执行Lua脚本
        Long result = redisTemplate.execute(rateLimiterScript, keys, args);

        // 结果为1表示允许请求，0表示请求被限流
        return Objects.equals(result, 1L);
    }
}

```

位运算与位图应用

Redis的位图（BitMap）是一种特殊的数据结构，它允许我们在单一的字符串键（String Key）中存储一系列二进制位（bits），每个位对应一个布尔值（0或1），并通过偏移量（offset）来定位和操作这些位。位图极大地节省了存储空间，尤其适合于大规模数据的标记、统计和筛选场景。

在位图中，每一位相当于一个标识符，例如可以用来表示用户是否在线、商品是否有库存、用户是否已读邮件等。相对于传统的键值对存储。位图可以非常快速地统计满足特定条件的元素个数，如统计在线用户数、激活用户数等。


```
@Service
public class UserOnlineStatusService {
```

```
    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    private static final String ONLINE_STATUS_KEY = "online_status";
    private static final String RETENTION_RATE_KEY_PREFIX = "retention
_rate:";
    private static final String DAILY_ACTIVITY_KEY_PREFIX = "daily_activi
ty:";

    /**
     * 设置用户在线状态为在线
     * @param userId
     */
    public void setUserOnline(long userId) {
        redisTemplate.opsForValue().setBit(ONLINE_STATUS_KEY, userId, t
rue);
    }

    /**
     * 设置用户在线状态为离线
     * @param userId
     */
    public void setUserOffline(long userId) {
        redisTemplate.opsForValue().setBit(ONLINE_STATUS_KEY, userId, f
alse);
    }

    /**
     * 获取用户在线状态
     * @param userId
     * @return
     */
    public boolean isUserOnline(long userId) {
        return redisTemplate.opsForValue().getBit(ONLINE_STATUS_KEY, u
serId);
    }

    /**
     * 统计在线用户数量
     * @return
     */
    public long countOnlineUsers() {
        return  getCount(ONLINE_STATUS_KEY);
    }

    /**
     * 记录用户的留存情况
```

```
     * @param userId
     * @param daysAgo
     */
    public void recordUserRetention(long userId, int daysAgo) {
        String key = RETENTION_RATE_KEY_PREFIX + LocalDate.now().mi
nusDays(daysAgo).toString();
        redisTemplate.opsForValue().setBit(key, userId, true);
    }

    /**
     * 获取指定日期的留存率
     * @param daysAgo
     * @return
     */
    public double getRetentionRate(int daysAgo) {
        String key = RETENTION_RATE_KEY_PREFIX + LocalDate.now().mi
nusDays(daysAgo).toString();
        long totalUsers = countOnlineUsers();
        long retainedUsers = getCount(key);
        return (double) retainedUsers / totalUsers * 100;
    }

    /**
     * 记录用户的每日活跃情况
     * @param userId
     */
    public void recordUserDailyActivity(long userId) {
        String key = DAILY_ACTIVITY_KEY_PREFIX + LocalDate.now().toStr
ing();
        redisTemplate.opsForValue().setBit(key, userId, true);
    }

    /**
     * 获取指定日期的活跃用户数量
     * @param date
     * @return
     */
    public long countDailyActiveUsers(LocalDate date) {
        String key = DAILY_ACTIVITY_KEY_PREFIX + date.toString();
        return getCount(key);
    }

    /**
     * 获取最近几天每天的活跃用户数量列表
     * @param days
     * @return
     */
    public List<Long> getDailyActiveUsers(int days) {
```

```
        LocalDate currentDate = LocalDate.now();
        List<Long> results = Lists.newArrayList();
        for (int i = 0; i < days; i++) {
            LocalDate date = currentDate.minusDays(i);
            String key = DAILY_ACTIVITY_KEY_PREFIX + date.toString();
            results.add(getCount(key));
        }
        return results;
    }

    /**
     * 获取key下的数量
     * @param key
     * @return
     */
    private long getCount(String key) {
        return (long) redisTemplate.execute((RedisCallback<Long>) connec
tion -> connection.bitCount(key.getBytes()));
    }
}
```

最新列表

Redis的List（列表）是一个基于双向链表实现的数据结构，允许我们在列表头部（左端）和尾部（右端）进行高效的插入和删除操作。

`LPUSH`命令：全称是`LIST PUSH LEFT`，用于将一个或多个值插入到列表的最左边（头部），在这里用于将最新生成的内容ID推送到列表顶部，保证列表中始终是最新的内容排在前面。

`LTRIM`命令用于修剪列表，保留指定范围内的元素，从而限制列表的长度。在这个场景中，每次添加新ID后都会执行`LTRIM`操作，只保留最近的N个ID，确保列表始终保持固定长度，即只包含最新的内容ID。

```
@Service
public class LatestListService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    private static final String LATEST_LIST_KEY = ”latest_list”;
```

```
    /**
     * 添加最新内容ID到列表头部
     * @param contentId 内容ID
     */
    public void addLatestContent(String contentId) {
        ListOperations<String, String> listOps = redisTemplate.opsForList()
;
        listOps.leftPush(LATEST_LIST_KEY, contentId);
        // 限制列表最多存储N个ID，假设N为100
        listOps.trim(LATEST_LIST_KEY, 0, 99);
    }

    /**
     * 获取最新的N个内容ID
     * @param count 要获取的数量，默认为10
     * @return 最新的内容ID列表
     */
    public List<String> getLatestContentIds(int count) {
        ListOperations<String, String> listOps = redisTemplate.opsForList()
;
        return listOps.range(LATEST_LIST_KEY, 0, count - 1);
    }
}

```

抽奖

借助Redis的Set数据结构以及其内置的`Spop`命令，我们能够高效且随机地选定抽奖获胜者。Set作为一种不允许包含重复成员的数据集合，其特性天然适用于防止抽奖过程中出现重复参与的情况，确保每位参与者仅拥有一个有效的抽奖资格。

由于Set内部元素的排列不具备确定性，这意味着在对集合执行随机获取操作时，每一次选取都将独立且不可预测，这与抽奖活动中所要求的随机公平原则高度契合。

Redis的`Spop`命令允许我们在单个原子操作下，不仅随机选取，还会从Set中移除指定数量（默认为1）的元素。这一原子操作机制尤为关键，在高并发环境下，即便有多个请求同时进行抽奖，`Spop`也能够确保同一时刻只有一个请求能成功获取并移除一个元素，有效避免了重复选择同一位参与者作为获奖者的可能性。

```
@Service
public class LotteryService {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    private static final String PARTICIPANTS_SET_KEY = "lottery:participa
nts";

    /**
     * 添加参与者到抽奖名单
     * @param participant 参与者ID
     */
    public void joinLottery(String participant) {
        redisTemplate.opsForSet().add(PARTICIPANTS_SET_KEY, participa
nt);
    }

    /**
     * 抽取一名幸运儿
     * @return 幸运儿ID
     */
    public String drawWinner() {
        // 使用Spop命令随机抽取一个参与者
        return redisTemplate.opsForSet().pop(PARTICIPANTS_SET_KEY);
    }

    /**
     * 抽取N个幸运儿
     * @param count 抽取数量
     * @return 幸运儿ID列表
     */
    public List<String> drawWinners(int count) {
        return redisTemplate.opsForSet().pop(PARTICIPANTS_SET_KEY, co
unt);
    }
}
```

Stream类型

Redis Stream作为一种自Redis 5.0起引入的高级数据结构，专为存储和处理有序且持久的消息流而设计。可视作一个分布式的、具备持久特性的消息队列，通过唯一的键名来标识每个Stream，其中容纳了多个携带时间戳和唯一标识

符的消息实体。

每条存储于Stream中的消息都具有全球唯一的message ID，该ID内嵌时间戳和序列编号，旨在确保即使在复杂的集群部署中仍能保持消息的严格时序性。这些消息内容会持久存储在Redis中，确保即使服务器重启也能安全恢复。

生产者利用`XADD`指令将新消息添加到Stream中，而消费者则通过`XREAD`或针对多消费者组场景优化的`XREADGROUP`命令来读取并处理消息。`XREADGROUP`尤其擅长处理多消费者组间的公平分配和持久订阅，确保消息的公正、有序送达各个消费者。

Stream核心特性之一是支持消费者组机制，消费者组内的不同消费者可独立地消费消息，并通过`XACK`命令确认已消费的消息，从而实现了消息的持久化消费和至少一次（at-least-once）交付保证。当消息量超出消费者处理能力时，未处理的消息可在Stream中积压，直到达到预设的最大容量限制。此外，还能设定消息的有效期（TTL），逾期未被消费的消息将自动剔除。即使在网络传输过程中消息遭受损失，亦可通过message ID保障消息的幂等性重新投递。尽管网络条件可能导致消息到达消费者的时间顺序与生产者发出的顺序有所偏差，但Stream机制确保了每个消息在其内在的时间上下文中依然保持着严格的顺序关系。

Redis Stream作为一个集消息持久化、多消费者公平竞争、消息追溯和排序等功能于一体的强大消息队列工具，已在日志采集、实时数据分析、活动追踪等诸多领域展现出卓越的适用性和价值。

```
@Component
public class LogCollector {

    private static final String LOGS_STREAM_KEY = "logs";
    private static final String GROUP_NAME = "log_consumers";
    private static final String CONSUMER_NAME = "log_consumer";

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    // 发送日志事件至 Redis Stream
    public void sendLogEvent(String message, Map<String, String> attributes) {
        StreamOperations<String, Object, Object> streamOperations = redisTemplate.opsForStream();
        RecordId messageId = streamOperations.add(StreamRecords.newR
```

```
ecord()
                .ofStrings(attributes)
                .withStreamKey(LOGS_STREAM_KEY));
    }

    // 实时消费日志事件
    public StreamRecords<String, String> consumeLogs(int batchSize) {
        Consumer consumer = Consumer.from(CONSUMER_NAME, GROUP
_NAME);
        StreamOffset<String> offset = StreamOffset.create(LOGS_STREAM
_KEY, ReadOffset.lastConsumed());
        StreamReadOptions<String, String> readOptions = StreamReadOpti
ons.empty().count(batchSize);
        return redisTemplate.opsForStream().read(readOptions, StreamOffs
et.create(LOGS_STREAM_KEY, ReadOffset.lastConsumed()), consumer);
    }
}

```

GEO类型

Redis的GEO数据类型自3.2版本起引入，专为存储和高效操作含有经纬度坐标的地理位置信息而设计。开发人员利用这一类型可以轻松管理地理位置数据，同时兼顾内存效率和响应速度。

利用`GEOADD`命令，可以将带有精确经纬度坐标的数据点归档至指定键名下的集合中。

可借助`GEOPOS`命令获取某一成员的具体经纬度坐标。

通过`GEODIST`命令，可以准确计算任意两个地理位置成员之间的地球表面距离，支持多种计量单位，包括米、千米、英里和英尺。

使用`GEORADIUS`命令，系统可以根据指定的经纬度中心点及半径范围检索出处于该区域内的所有成员地理位置。

`GEORADIUSBYMEMBER`命令也用于范围查询，但其查询依据是选定成员自身的位置，以此为圆心划定搜索范围。

GEO类型在许多场景下都非常有用，例如移动应用中的附近好友查找、商店位

置搜索、物流配送中的最近司机调度等。

```
@Service
public class FriendService {

    private static final String FRIEND_LOCATIONS_KEY = ”friend_location
s”;

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    @Autowired
    private GeoOperations<String, FriendLocation> geoOperations; // 自
动装配GeoOperations

    public void saveFriendLocation(FriendLocation location) {
        geoOperations.add(FRIEND_LOCATIONS_KEY, location.getLongitud
e(), location.getLatitude(), location);
    }

    public List<FriendLocation> findFriendsNearby(double myLongitude, d
ouble myLatitude, Distance radius) {
        Circle circle = new Circle(new Point(myLongitude, myLatitude), radi
us);
        return geoOperations.radius(FRIEND_LOCATIONS_KEY, circle, Metr
ic.KILOMETERS).getContent();
    }
}
```

总结

Redis作为一款高性能、内存型的NoSQL数据库，凭借其丰富的数据结构、极高的读写速度以及灵活的数据持久化策略，在现代分布式系统中扮演着至关重要的角色。它的关键价值体现在以下几个方面：

1. 缓存优化：Redis将频繁访问的数据存储在内存中，显著减少了数据库的读取压力，提升了系统的整体性能和响应速度。
2. 分布式支持：通过主从复制、哨兵和集群模式，Redis实现了高度可扩展性和高可用性，满足大规模分布式系统的需求。
3. 数据结构多样性：Redis支持字符串、哈希、列表、集合、有序集合、Bitmaps、HyperLogLog、Geo等多样化的数据结构，为多种应用场景提供了

便利，如排行榜、社交关系、消息队列、计数器、限速器等。
4. 实时处理与分析：随着Redis 5.0引入Stream数据结构，使得Redis在日志收集、实时分析、物联网数据流处理等方面有了更多的可能性。
5. 地理位置服务：GEO类型提供了便捷的空间索引和距离计算功能，使得Redis能够在电商、出行、社交等领域提供附近地点搜索、路线规划等服务。

本文已收录于我的个人博客：[码农Academy的博客，专注分享Java技术干货，包括Java基础、Spring Boot、Spring Cloud、Mysql、Redis、Elasticsearch、中间件、架构设计、面试题、程序员攻略等](http://cxyroad.com/ ”https://www.coderacademy.online/”)

原文链接: https://juejin.cn/post/7351321303247257637