基于Redission布隆过滤器-优化版

概述：

原因：

juejin.cn/post/735501… （基于Redission布隆过滤器原理,优缺点及工具类和使用示例）文中简单介绍Redission布隆过滤器使用，这边基于此上再次给出优化版本，防止元素存在误差的场景：

优化：

优化理论：

为了进一步优化 `RedissonBloomFilterUtil`，我们可以考虑以下几点来增强其功能和鲁棒性：

1. 配置管理：允许动态配置布隆过滤器的参数，如预期插入数量和误报率。
2. 异常处理：增加异常处理机制来确保系统稳定运行，即使底层服务（缓存或数据库）出现问题。
3. 日志记录：记录关键操作和异常，以便于问题追踪和性能监控。
4. 重建和同步布隆过滤器：提供机制定期或根据需要重建布隆过滤器，以减少误报率。
5. 分布式锁：在更新缓存和布隆过滤器时使用分布式锁，以防止并发更新导致的数据不一致。
6. 回退策略：在缓存和数据库服务不可用时提供回退策略，以保证服务的持续可用性。

优化流程：

使用布隆过滤器的基本流程如下：

1. 当请求到来时，首先查询布隆过滤器。

2. 如果布隆过滤器认为数据不存在，则直接返回，不继续查询数据库。
3. 如果布隆过滤器认为数据可能存在，继续查询缓存。
4. 如果缓存中有数据，返回缓存数据。
5. 如果缓存中没有数据，继续查询数据库。
6. 如果数据库中有数据，将其放入缓存并返回。
7. 如果数据库中也没有数据，可以选择更新布隆过滤器（如果布隆过滤器支持删除操作），或者记录这个不存在的查询，以便在布隆过滤器重建时排除这些误报。

优化实现：

以下是考虑了上述优化点的 `RedissonBloomFilterUtil` 类的代码实现：

```
import org.redisson.api.RBloomFilter;
import org.redisson.api.RedissonClient;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class RedissonBloomFilterUtil {

    private static final Logger logger =
LoggerFactory.getLogger(RedissonBloomFilterUtil.class);

    private RBloomFilter<String> bloomFilter;
    private RedissonClient redissonClient;
    private CacheService cacheService; // 假设这是你的缓存服务
    private DatabaseService databaseService; // 假设这是你的数据库服务

    public RedissonBloomFilterUtil(RedissonClient redissonClient,
CacheService cacheService, DatabaseService databaseService, long
expectedInsertions, double falseProbability) {
        this.redissonClient = redissonClient;
        this.cacheService = cacheService;
        this.databaseService = databaseService;
        this.bloomFilter = redissonClient.getBloomFilter("bloomFilter");
        this.bloomFilter.tryInit(expectedInsertions, falseProbability);
    }

    public Object getData(String key) {
        try {
            if (!bloomFilter.contains(key)) {
                logger.debug("Bloom filter does not contain key: {}", key);
                return null;
```

```
            }

            Object cachedData = cacheService.getFromCache(key);
            if (cachedData != null) {
                logger.debug("Returning data from cache for key: {}", key);
                return cachedData;
            }

            Object databaseData = databaseService.getFromDatabase(key);
            if (databaseData != null) {
                cacheService.addToCache(key, databaseData);
                logger.debug("Data added to cache for key: {}", key);
                return databaseData;
            }

            logger.debug("Data not found in database for key: {}", key);
            recordFalsePositive(key);
        } catch (Exception e) {
            logger.error("Error retrieving data for key: {}", key, e);
            // Implement fallback strategy (e.g., retry logic, circuit breaker)
        }
        return null;
    }

    private void recordFalsePositive(String key) {
        // Implement logic to record false positives
        // This could involve logging or updating the bloom filter
        logger.info("Recorded false positive for key: {}", key);
    }

    public void rebuildBloomFilter() {
        // Implement logic to rebuild the bloom filter
        // This might involve clearing the bloom filter and re-adding keys
from a reliable source
        logger.info("Rebuilding bloom filter");
    }

    // ... (shutdown method and other methods)
}

```

优化总结：

在这个实现中，我们添加了日志记录来帮助跟踪操作和捕获异常。我们还为 `getData` 方法添加了异常处理逻辑，以便在发生错误时记录错误并执行回退策略（例如重试逻辑或断路器模式）。

`recordFalsePositive` 方法现在记录了误报的信息，这有助于监控布隆过滤器的性能，并在必要时进行调整。

`rebuildBloomFilter` 方法是一个存根，你可以在这里实现布隆过滤器的重建逻辑。这可能包括从数据库或其他可靠源重新填充布隆过滤器，以确保其准确性。

扩展：

请注意，为了实现分布式锁和回退策略，可能需要更复杂的逻辑和额外的依赖。
可以使用 Redisson 提供的分布式锁功能，在更新缓存和布隆过滤器时保持数据一致性。
对于回退策略，你可能需要使用像 Hystrix 这样的库来实现断路器模式，或者自己实现重试逻辑。