16.精确掌控并发：漏桶算法在分布式环境下并发流量控制的设计与实现_V20240115

本篇重点讲清楚【漏桶】原理，在支付系统的应用场景，以及使用reids实现的核心代码。

1. 前言

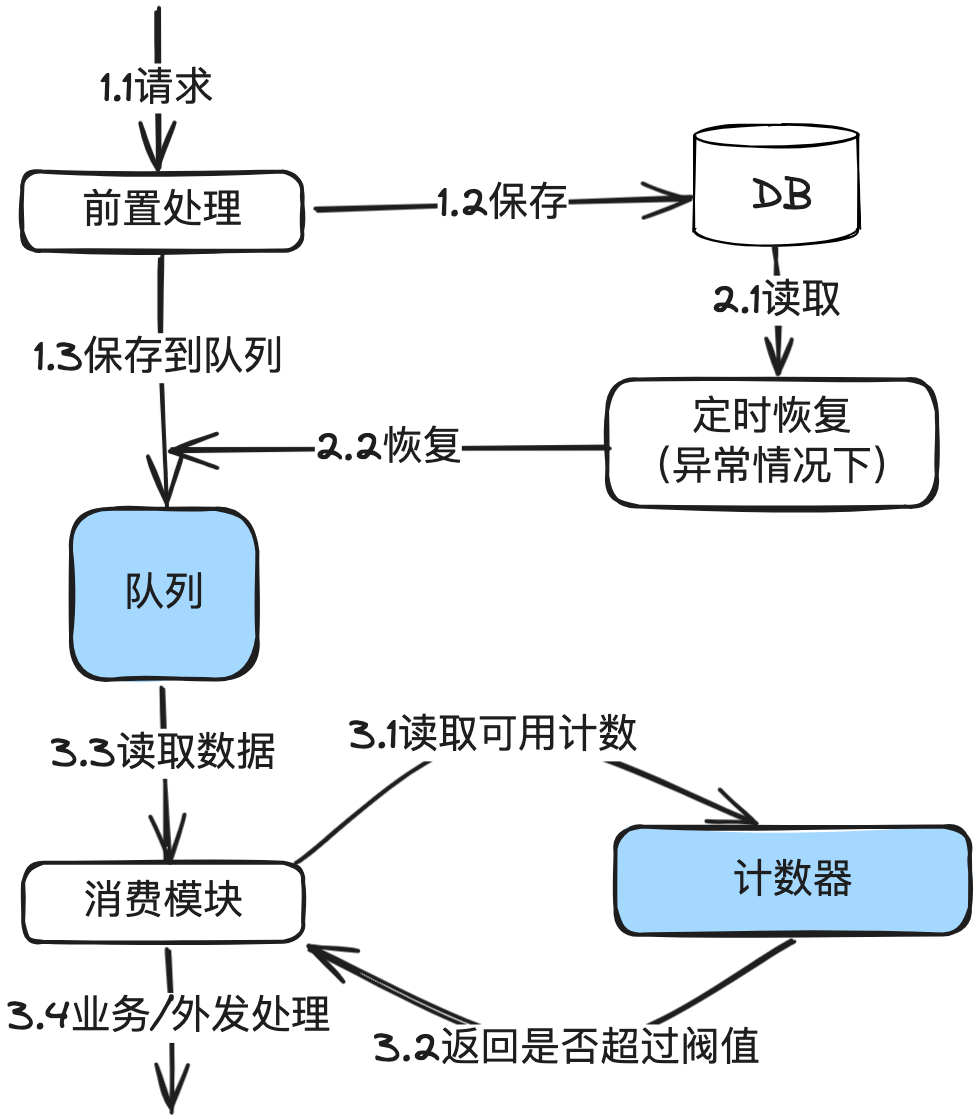
在流量控制系列文章中的前两篇，分别介绍了固定时间窗口算法和滑动时间窗口算法在支付渠道限流的应用以及使用redis实现的核心代码。

这两个算法有一个共同的问题：那就是超过阀值的数据会直接拒绝掉。如果超过阀值也不想拒绝请求，后面仍然发出去，怎么办？这就是本篇要说的漏桶及下篇要讲的令牌桶解决的问题。

2. 漏桶原理

漏桶算法通过模拟水桶漏水的过程来控制数据的传输速率。它允许短时间的突发数据流，随后以恒定的速率排空积聚的数据。这种机制特别适合于需要平滑处理瞬时高流量冲击，但后端需要恒定速率处理的场景。比如批量接收上游商户的退款，然后根据渠道的要求以极低的TPS慢慢退出去到渠道。

最简单的理解，漏桶 = 队列 + 固定窗口算法。其中队列用于先保存数据。固定窗口算法用于获取可用计数，获取到就从队列获取一个请求进行业务处理。



工作原理：

1. 桶容量：漏桶有一个固定的容量，代表在任何时刻系统能够容纳的最大请求量。比如上面图中的队列。
2. 数据流入：数据来了后就保存到桶（队列）中，如果桶已满，则溢出的数据会被丢弃。
3. 恒定速率流出：数据以固定的速率从桶中“漏出”，即被处理。这个速率是预先设定的，与请求量无关。
4. 计数器最简单的做法，就是把固定时间窗口的代码用起来。
5. 保存到数据库，是为了持久化，以及队列出现问题时，可以重新恢复。

3. 在支付系统下的应用场景

中国的IT基础设施领先于全球各个国家，各大银行和第三方钱包也被各电商双十一等大促场景狂虐之后进化到支持极高的TPS，但是在跨境场景下，比如东南亚或南美的国家，他们的银行IT基础设施差，系统老旧，无法支持高并发流量。甚至碰到过一些银行要求退款只能有1TPS。

在分布式场景下，要做到**1TPS**的高精度限流，只能依赖漏桶来做。

4. Redis实现漏桶的核心代码

漏桶算法通常通过队列 + 固定时间窗口计数法来实现。队列存储待处理的请求，而一个线程以固定速率从队列中取出并处理这些请求。

为什么又是Redis？因为前面已经实现过Redis版本的固定时间窗口算法，再加一个队列就可以搞定。当然大家也可以选择其它的方案实现，这只是一个抛砖引玉。

下面是单机版本的伪代码：

```java
public class LeakyBucket {
    private final int capacity;
    private final long leakIntervalInMillis;
    private final LinkedBlockingQueue<Data> bucket;

    public LeakyBucket(int capacity, long leakRateInMillis) {
        this.capacity = capacity;
        this.leakIntervalInMillis = leakRateInMillis;
        this.bucket = new LinkedBlockingQueue<>(capacity);
    }

    // 尝试添加数据到桶中
    public boolean addToBucket(Data data) {
        return bucket.offer(data);
    }

    // 启动桶的漏水过程
    public void startLeaking() {
        new Thread(() -> {
            while (true) {
                try {
                    Data data = bucket.poll(leakIntervalInMillis, TimeUnit.MILLISECONDS);
                    if (data != null) {
                        process(data);
                    }
                } catch (InterruptedException e) {
                    log.debug("Leaking process interrupted");
                    continue;
                }
            }
        }).start();
    }

    // 处理桶中的数据
    private void process(Data data) {
        // 业务处理
        ... ...
    }
}
```

上面单机的代码实用性不高，因为在分布式环境下，并发请求量是根据部署机器累计起来的，1台机器限流1TPS，20台机器就到了20TPS。

优化为分布式：

```java
class LeakyBucketHolding {
    private final LinkedBlockingQueue<Data> bucket;
    private int limit;
    private String bizType;

    public LeakyBucketHolding(String bizType, int capacity, int limit) {
        this.bizType = bizType;
        this.bucket = new LinkedBlockingQueue<>(capacity);
        this.limit = limit;
    }

    // 其它代码略
}


class LeakyBucket {

    @Autowired
    private RedisLimitUtil redisLimitUtil;

    private Map<String, LeakyBucketHolding> leakyBucketHoldingMap = new HashMap();


    // 添加数据到桶中
    public boolean addData(Data data) {
        String key = buildKey(data);
        LeakyBucketHolding holding = leakyBucketHoldingMap.get(key);
        if (null == holding) {
            holding = buildHolding(data);
            leakyBucketHoldingMap.put(key, holding);
        }
        return holding.getLinkedBlockingQueue().offer(data);
    }

    public Data getData() {
        for(LeakyBucketHolding holding : leakyBucketHoldingMap.values()) {
            if(holding.getBucket().size() == 0) {
                return null;
            }

            /* RedisLimitUtil的实现参考
             * "精确掌控并发：固定时间窗口算法在分布式环境下并发流量控制的设计与实现"中的示例代码
             */
```

```
            boolean limited = RedisLimitUtil.isLimited(holding.getBizType(
), holding.getLimit());
            if (limited) {
                  return null;
            }

            try {
                return holding.getBucket().poll(10, TimeUnit.MILLISECONDS)
;
            } catch (InterruptedException e) {
                log.log("Leaking process interrupted");
            }
            return null;
        }
    }
}
```

上面的代码只是写一个示例，也没有做方法的抽取，真实的代码会比这个写得更优雅一点，大家将就看一下，理解思路就行。

代码使用的是内存列队，也就是请求过来后，先保存到DB，然后发到内存队列。在重启服务器时，内存列队的数据会丢失，这种情况下，依赖定时任务从DB中恢复任务到内存列队。

还有一种做法，就不使用内存队列，而是使用redis来实现队列。代码如下：

```java
public class LeakyBucket {
    // 其它代码略
    ... ...

    // 添加数据到队列中
    public void addData(Data data) {
        return redisTemplate.rpush(data.getBizType(), data);
    }


    // 添加数据到队列中
    public Data getData(String bizType) {
        return redisTemplate.lpop(bizType);
    }

    // 其它代码略
    ... ...
}

```

退款流量控制实例：RefundServiceImpl

```java
/**
 * 支付服务示例
 */
public class RefundServiceImpl implements RefudnService {
    @Autowiread
    private LeakyBucket leakyBucket;

    @Override
    public RefundOrder refund(RefundRequest request) {
        // 前置业务处理
        ... ...

        Data data = buildData(request);
        leakyBucket.addData(data);

        // 其它业务处理
        ... ...
    }

    @PostConstruct
    public void init() {
        new Thread(() -> {
            while (true) {
                Data data = leakyBucket.getData();
                if (null != data) {
                    process(data);
                } else {
                    sleep(10);
                }
            }
        }).start();
    }
}
```

在代码中可以看到，退款请求来后，只需要往桶里扔就完事。然后等另外的线程按固定速度发出去。

代码中还存在的问题：

1. 上述代码只是示例，真实的代码还有很多异常处理，比如队列数据丢失，需要重新处理。
2. 暂时只能用于退款，因为退款的时效要求不高。另外，单机只需要开一个线程就行，因为服务器是分布式部署，多个服务器合并起来仍然是多个线程在并发处理。对退款是足够的。

5. 为什么不使用消息中间件来做队列

为什么不直接使用RabbitMQ或Kafaka等消息中间件来做队列？主要是因为有些公司使用自码的消息中间件，可能只有推模型而没有拉的模式。

如果只有推的模式，就会出现推下来后发现限流，又抛回来，来回做无用功。

如果消息中间件有拉的模式，同时配合redis的固定窗口实现，也是完全没有问题的。

6. 为什么不直接使用消息中间件来做流控

消息中间件是另外的选型方案，会在后面的文章中介绍。

7. 结束语

今天主要介绍了漏桶原理、在支付系统中的使用场景，以及基于redis实现的核心代码。

下一篇将介绍令牌桶在分布式场景下流量控制的应用和核心代码实现。

这是《百图解码支付系统设计与实现》专栏系列文章中的第（16）篇。和墨哥（隐墨星辰）一起深入解码支付系统的方方面面。

欢迎转载。

Github（PDF文档全集，不定时更新）：https://github.com/yinmo-sc/Decoding-Payment-System-Book

公众号：隐墨星辰。

有个小群不定时解答一些问题或知识点，有兴趣的同学可先加微信（yinmo_sc）后进入，添加微信请备注：加支付系统设计与实现讨论群。