使用 Redis 实现限流——滑动窗口算法

用 Go 语言实现滑动窗口限流算法，并利用 Redis 作为存储后端，可以按照以下步骤进行设计和编码。滑动窗口限流的核心思想是**维护一个固定时间窗口**，并在窗口内记录请求次数，当窗口滑动时，旧的请求计数被移除，新的请求计数被添加。这里以 Redis 的有序集合（Sorted Set，简称 ZSet）作为数据结构，因为它可以方便地实现时间排序和计数功能。

步骤一：定义滑动窗口参数

确定滑动窗口的几个关键参数：

* 时间窗口宽度（如：1秒、5分钟等）
* 允许的最大请求数量（如：每秒100次、每分钟1000次等）

步骤二：选择 Redis 操作

使用 Redis 的有序集合（ZSet），其成员为请求发生的时间戳（Unix 时间戳），分值为请求的计数值（通常初始为1）。ZSet 可以自动按分值排序，便于我们管理滑动窗口内的请求。

步骤三：编写 Go 代码实现限流逻辑

以下是使用 Go 语言和 Redis 实现滑动窗口限流的基本流程：

1. **初始化 Redis 客户端**：
使用 `github.com/go-redis/redis/v8` 库或其他熟悉的 Redis 客户端库创建一个 Redis 客户端实例。

```
import (
    "github.com/go-redis/redis/v8"
)
```

```
var rdb *redis.Client

func init() {
    rdb = redis.NewClient(&redis.Options{
        Addr:     "localhost:6379",
        Password: "", // no password set
        DB:       0,  // use default DB
    })
}

```

2. **定义限流方法**:
创建一个名为 `limitRequest` 的函数，接收请求的标识符（如 API 路径或用户 ID）和当前时间作为参数。在函数内部执行以下操作：

a. **计算窗口边界**:
根据当前时间计算出滑动窗口的起始和结束时间戳。

b. **清理过期请求**:
使用 ZSet 的范围删除（ZREMRANGEBYSCORE）命令移除窗口起始时间之前的元素，确保窗口内仅包含有效请求。

c. **累加新请求**:
将当前请求的时间戳作为成员加入 ZSet，如果已有相同时间戳的成员，则使用 ZINCRBY 命令递增其分值（表示多次请求在同一时刻）。

d. **检查请求是否超限**:
使用 ZCARD 命令获取窗口内请求总数，与允许的最大请求数比较。若超出限制，则返回限流结果；否则，允许此次请求并返回成功。

```
func limitRequest(identifier string, now int64, maxRequestsInWindow int) (bool, error) {
    key := fmt.Sprintf("rate_limit:%s", identifier)

    // 计算窗口边界
    windowStart := now - timeWindowWidth
    windowEnd := now

    // 清理过期请求
    if err := rdb.ZRemRangeByScore(ctx, key, "-inf", windowStart).Err();
```

```
err != nil {
        return false, err
    }

    // 累加新请求
    pipe := rdb.Pipeline()
    exists, _ := pipe.ZScore(ctx, key, now).Result()
    if exists == nil {
        pipe.ZAdd(ctx, key, &redis.Z{Score: float64(now), Member: now})
    } else {
        pipe.ZIncrBy(ctx, key, 1, now)
    }
    _, err := pipe.Exec(ctx)
    if err != nil {
        return false, err
    }

    // 检查请求是否超限
    count, err := rdb.ZCard(ctx, key).Result()
    if err != nil {
        return false, err
    }
    if count > int64(maxRequestsInWindow) {
        return false, nil // 超限，拒绝请求
    }

    return true, nil // 未超限，允许请求
}

```

3. **调用限流方法**:
在实际处理请求的逻辑中，先调用 `limitRequest` 方法判断是否允许此次请求，根据返回值决定是否继续处理请求或返回限流响应。

请注意，以上代码示例假设您已经设置了 Redis 连接信息和相关常量（如 `timeWindowWidth`）。实际应用中，请根据您的具体需求调整代码，例如添加错误处理、优化性能（如使用 pipeline 或 Lua 脚本）、考虑分布式环境下的锁机制（如 RedLock）等。

此外，滑动窗口限流算法在高并发场景下可能会对 Redis 造成一定压力，尤其是窗口较短且请求量大的情况下。为了提高性能，可以考虑使用布隆过滤器提前过滤掉部分请求，或者结合客户端缓存减少对 Redis 的直接访问。

Redis 发生故障时

使用 Redis 作为限流工具时，确实需要考虑其服务的稳定性。尽管 Redis 是一个高可用、高性能的键值数据库，但在实际生产环境中，任何服务都可能因为各种原因（如硬件故障、网络问题、软件错误等）出现暂时不可用的情况。针对 Redis 崩溃或不可用的情况，可以采取以下几种策略来应对：

1. **冗余与高可用部署**：

* **主从复制**：配置 Redis 主从架构，确保数据在多个节点间同步。当主节点崩溃时，可以通过自动或手动切换到已同步数据的从节点继续提供服务。
* **哨兵模式**（Sentinel）：使用 Redis Sentinel 提供自动故障检测和主节点切换功能，进一步提升系统的自我恢复能力。
* **集群模式**：部署 Redis 集群，将数据和负载分散在多个节点上，即使部分节点不可用，整个集群仍能继续提供服务。
2. **客户端容错与重试**：

* **连接池管理**：在客户端实现连接池管理，当连接失败时能够自动重新建立连接或从池中获取其他可用连接。
* **重试策略**：对于因 Redis 临时不可用导致的失败操作，实施合理的重试策略。比如，短暂延迟后重试（指数退避或固定间隔重试），避免短时间内频繁重试加重 Redis 服务器负担。
* **降级策略**：在 Redis 不可用时，客户端可以暂时执行降级逻辑，如放宽限流条件、允许一定比例的请求通过（牺牲一部分限流效果），或者暂时禁用限流功能，确保服务的基本可用性。
3. **本地缓存与兜底逻辑**：

* **本地计数**：在客户端（如应用程序服务器）维持一个本地计数器，用于在短时间内（如几秒钟）进行限流。这样，在 Redis 短暂不可用期间，可以依赖本地计数器进行限流，待 Redis 恢复后，再将本地计数同步回 Redis。
* **熔断与降级**：在客户端或服务治理框架中设置熔断机制，当连续检测到 Redis 服务不可用时，触发熔断状态，直接拒绝部分非关键请求或返回默认值，防止请求堆积导致系统雪崩。
4. **监控与报警**：

* **实时监控**：对 Redis 服务的运行状态、性能指标、故障事件进行实时监控，及时发现异常情况。
* **报警通知**：设置警报阈值和通知机制，一旦 Redis 出现故障或性能下降，立即通知运维人员进行干预。

通过上述措施，可以在 Redis 发生故障时降低对限流功能的影响，保障系统的

整体稳定性和可用性。同时，应定期对 Redis 集群进行健康检查、性能调优和数据备份，预防潜在问题，提升系统的健壮性。

欢迎公-众-号【TaonyDaily】、留言、评论，一起学习。

> Don’t reinvent the wheel, library code is there to help.

文章来源：[刘俊涛的博客](http://cxyroad.com/ ”https://www.cnblogs.com/lovebing”)

若有帮助到您，欢迎点赞、转发、支持，您的支持是对我坚持最好的肯定(*^_^*)

原文链接: https://juejin.cn/post/7361555165185130534