Redis 热 key 解决方案

讲师：tojson

1、热 key 背景介绍

✧ 什么是 redis 热 key

在 Redis 中，热 key 是指那些在极短的时间内访问频次非常高的 key。

✧ 热 key 出现的场景

- ✓ 节假日或双十一活动某商品促销降价成为爆款
- ✓ 新浪微博或头条某某明星结婚了，大量的用户点击浏览这条新闻
- ✓ 秒杀活动，瞬间大量爬虫用户或机器人涌入系统

✧ 热 key 产生的影响

- ✓ 对应节点的网卡带宽被打满，出现丢包重传，请求波动耗时大幅上升
- ✓ 请求过多，热点 key 引起 redis 节点数据倾斜，缓存服务被打垮
- ✓ 大量的请求穿透到 DB，DB 扛不住宕机

热点 key 的出现可能会对系统的稳定性和可用性造成巨大的影响，在日常的工作中，我们需要尽可能避免这种情况的出现，比如在设计和编码阶段避免引入全局性热 key，或者在设计时考虑热 key 出现时的应对方案。

2、如何发现热 key

平常我们在设计和开发需求的时候，已经尽可能预判一些热点的场景，但是在真实的环境中还是会有不可预料的场景出现，比如十倍或百倍的突发流量流入。

既然不可能完全避免，我们就需要有一种方法能够在出问题的时候快速定位有没有热key以及是哪些热 key，通过这个帮助业务快速定位问题的根源。那究竟如何发现热 key 呢？

我们可以从 redis 请求入手来探测热 key，具体如下所示：

✧ 开发独立的热 key 检测系统（比较推荐）

提供单独的热 key 检测的接入 sdk，应用系统引入该 sdk 后，热 key 检测系统自动计算是否热 key 并推送相关结果给应用系统，应用系统根据业务实际情况进行相应处理。

✧ 改写 redis 客户端收集上报数据

改写 Redis SDK，记录每个请求，定时把收集到的数据上报，然后由一个统一的服务进行聚合计算。方案直观简单，但没法适应多语言架构，一方面多语言 SDK 对齐是个问题，另外一方面后期 SDK 的维护升级会面临比较大的困难，成本很高。

✧ 改写 redis 代理层收集上报数据

如果所有的 redis 请求都经过代理的话，可以考虑改动 proxy 代码进行收集，思路与客户端基本类似。该方案对使用方完全透明，能够解决客户端SDK的语言异构和版本升级问题，不过开发成本会比客户端高些。

✧ 利用 redis 新特性定时扫描上报数据

redis 在 4.0 版本之后添加了 hotkeys 查找特性[1]，可以直接利用 redis-cli --hotkeys 获取当前 keyspace 的热点 key。该方案无需二次开发，能够直接利用现成的工具，但由于需要扫描整个 keyspace，实时性上比较差，另外扫描耗时与 key 的数量正相关，如果 key 的数量比较多，耗时可能会非常长。

✧ redis 节点抓包解析上报数据

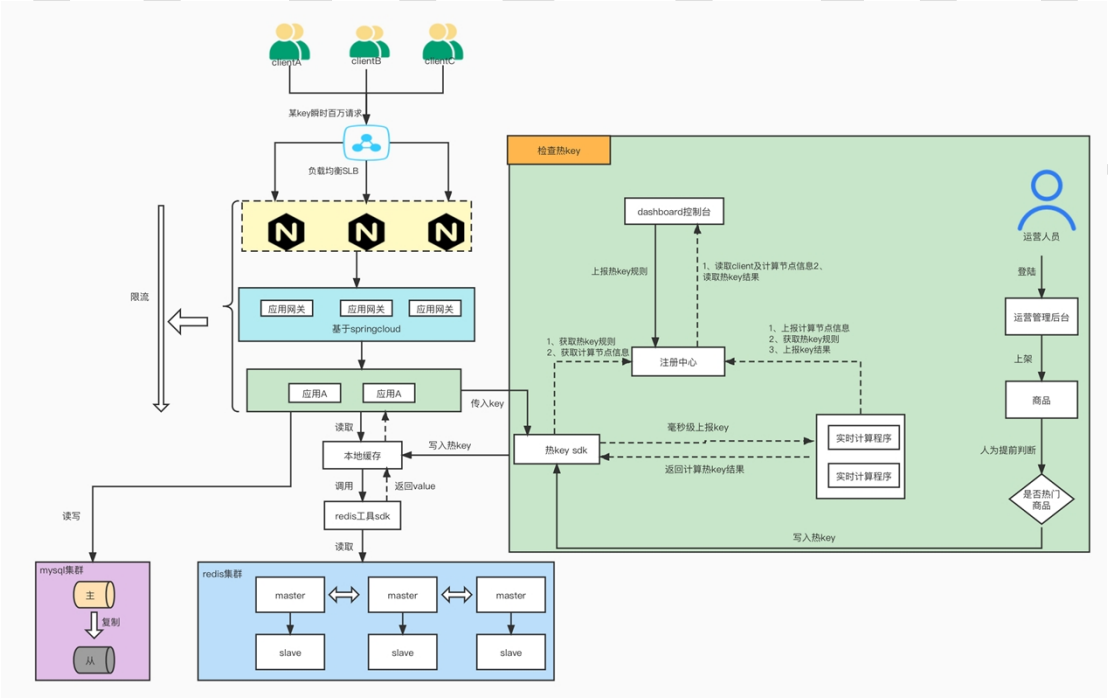
在可能存在热 key 的节点上(流量倾斜判断)，通过 tcpdump 抓取一段时间内的流量并上报，然后由一个外部的程序进行解析、聚合和计算。该方案无需侵入现有的 SDK 或者 Proxy 中间件，开发维护成本可控，但也存在缺点的，具体是热 key 节点的网络流量和系统负载已经比较高了，抓包可能会情况进一步恶化。

✧ 凭业务经验提前预估热 key

3、解决方案

✧ 利用本地缓存（guava cache 或 caffeine）

在你发现热 key 以后，把热 key 加载到系统的 JVM 中。针对这种热 key 请求，会直接从 jvm 中取，而不会走到 redis 层。常见的本地缓存可以利用 guava cache 或者 caffeine 实现。现在假设，你的应用有 100 个节点，OK，你也有 jvm 缓存了。这 10 万个请求平均分散开来，每个节点有 1000 个请求，会从 JVM 中取到值然后返回数据。避免了 10 万个请求打到同一台 redis 上的情形。而如果该 key 是在本地内存中，读取一个内存中的值，每秒多少个万次都是很正常的，不存在任何数据层的瓶颈。当然，如果通过增加 redis 集群规模的形式，也能提升数据的访问上限，但问题是事先不知道热 key 在哪里，而全量增加 redis 的规模，性价比很低，毕竟热 key 不是经常会有。下面我们来看看整体架构流程图，如下所示：



优点：

内存访问和 redis 访问的速度不在一个量级，基于本地缓存，接口性能非常好， 可以大大增加单实例的 QPS。

缺点：

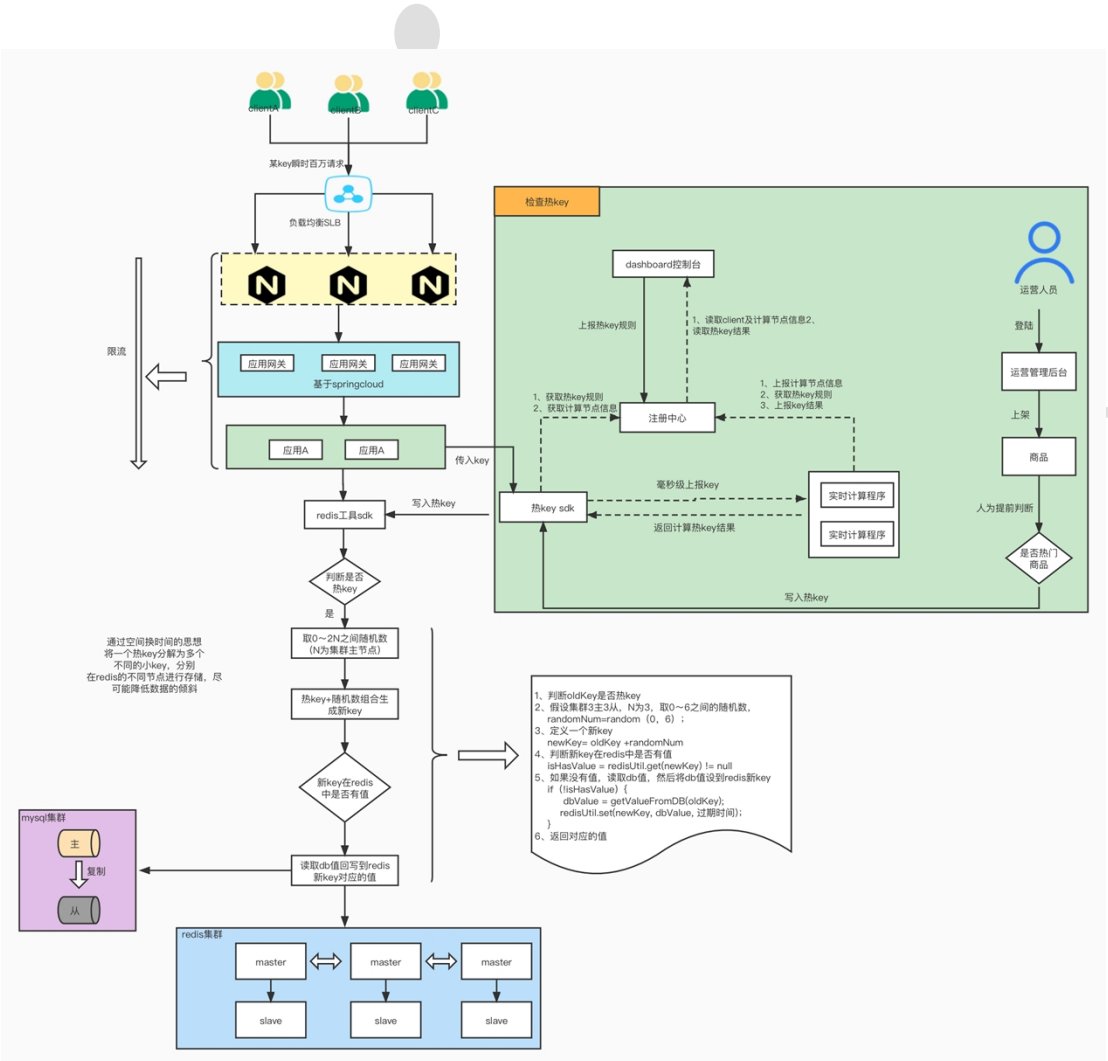
受应用内存限制，容量有限，数据量非常大的时候，占用太多内存，不太适合。

部分热点数据，需要提前预知。

热点数据自动检测有一定的延迟，系统短时间内承受的风险比较大。

✧ 冗余存储备份 key

通过空间换时间的思想，将一个热 key 分解为多个不同的小 key，分别在 redis 集群的不同节点进行存储，尽可能降低数据的倾斜，当瞬间有几十万的请求去访问 redis 上某个 key，通过 key 分流，请求会分散打到不同的节点获取数据，这样就没有所谓的热点 key 问题。下面我们来看看整体架构流程图，如下所示：



优点：

不受应用内存限制，请求量比较大的时候可以轻松水平扩容。

缺点：

同一份数据，冗余多份存储，浪费了 redis 部分内存。

✧ 限流熔断（兜底方案，保护系统高可用）

1）基于 nginx 层限流（集成 lua 限流模块）

2）基于应用网关限流

3）基于微服务限流

常用的限流框架：hystrix 或阿里的 sentinel

常用的限流算法：漏桶、令牌桶、计数器统计、滑动窗口