

Redis 知识点总结 1

digest-time: 2021-10-02

什么是 Redis?

- C 语言写成, 高性能 key-value 非关系缓存数据库。
- 支持存储的 value 类型更多, 包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hash (哈希类型)。

Redis 有哪些优缺点?

- 优点
 - 读写性能优异, Redis 能读的速度是 110000 次/s, 写的速度是 81000 次
 - 支持数据持久化, 支持 AOF 和 RDB 两种持久化方式。
 - 支持事务, Redis 的所有操作都是原子性的, 同时 Redis 还支持对几个操作合并后的原子性执行。
 - 数据结构丰富, 除了支持 string 类型的 value 外还支持 hash、set、zset、list 等数据结构。
 - 支持主从复制, 主机自动将数据同步到从机, 可以进行读写分离。
- 缺点
 - 数据库容量受到物理内存的限制, 主要局限在较小数据量的高性能操作和运算上。
 - 不具备自动容错和恢复功能, 主机从机的宕机都会导致前端部分读写请求失败, 需要等待机器重启或者手动切换前端的 IP 才能恢复。
 - 主机宕机, 宕机前有部分数据未能及时同步到从机, 切换 IP 后还会引入数据不一致的问题, 降低了系统的可用性。
 - 较难支持在线扩容, 在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题, 运维人员在系统上线时必须确保有足够的空间, 这对资源造成了很大的浪费。

为什么要用 Redis 而不用 map/guava 做缓存?

- 缓存分为本地缓存和分布式缓存。以 Java 为例, 使用自带的 map 或者 guava 实现的是**本地缓存**, 最主要的特点是轻量以及快速, 生命周期随着 jvm 的销毁而结束, 并且在多实例的情况下, 每个实例都需要各自保存一份缓存, **缓存不具有一致性**。
- 使用 redis 或 memcached 之类的称为分布式缓存, 在多实例的情况下, 各实例共用一份缓存数据, 缓存具有一致性。缺点是需保持 redis 或 memcached 服务的高可用, 整个程序架构上较为复杂。

Redis 为什么这么快

- 1、**完全基于内存**, 绝大部分请求是纯粹的内存操作, 非常快速。数据存在内存中, 类似于 HashMap, HashMap 的优势就是查找和操作的时间复杂度都是 $O(1)$;

- 2、数据结构经过专门设计，如 String；
- 3、采用单线程，避免了不必要的上下文切换和竞争条件，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- 4、使用多路 I/O 复用模型，非阻塞 IO；
- 5、Redis 构建了通信机制和协议（VM 机制），因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

Redis 有哪些数据类型

- Redis 主要有 5 种数据类型，包括 String，List，Set，Zset，Hash，满足大部分的使用要求

数据类型	可以存储的值	操作	应用场景
String	字符串、整数或者浮点数	对整个字符串或者字符串的其中一部分执行操作 对整数和浮点数执行自增或者自减操作	做简单的键值对缓存
List	列表	从两端压入或者弹出元素 对单个或者多个元素进行修剪， 只保留一个范围内的元素	存储一些列表型的数据结构，类似 粉丝列表、文章的评论列表 之类的数据
Set	无序集合	添加、获取、移除单个元素 检查一个元素是否存在于集合中 计算交集、并集、差集 从集合里面随机获取元素	交集、并集、差集的操作 ，比如交集，可以把两个人的粉丝列表整一个交集
Hash	包含键值对的无序散列表	添加、获取、移除单个键值对 获取所有键值对 检查某个键是否存在	结构化的数据，比如一个 对象
ZSet	有序集合	添加、获取、删除元素 根据分值范围或者成员来 获取元素 计算一个键的排名	去重但可以排序，如 获取排名前几名的用户

Redis 的应用场景

- 计数器

可以对 String 进行自增自减运算，从而实现计数器功能。Redis 这种内存型数据库的读写性能非常高，很适合存储频繁读写的计数量。

- 缓存

将**热点数据**放到内存中，设置内存的最大使用量以及淘汰策略来保证缓存的命中率。

- 会话缓存

可以使用 Redis 来统一存储多台应用服务器的会话信息。当应用服务器不再存储用户的会话信息，也就不再具有状态，一个用户可以请求任意一个应用服务器，从而更容易实现高可用性以及可伸缩性。

- 消息队列(发布/订阅功能)

List 是一个双向链表，可以通过 `lpush` 和 `rpop` 写入和读取消息。不过最好使用 Kafka、RabbitMQ 等消息中间件。

- 分布式锁实现

在分布式场景下，无法使用单机环境下的锁来对多个节点上的进程进行同步。可以使用 Redis 自带的 `SETNX` 命令实现分布式锁，除此之外，还可以使用官方提供的 **RedLock** 分布式锁实现。

- 其它

Set 可以实现**交集**、**并集**等操作，从而实现**共同好友**等功能。ZSet 可以实现有序性操作，从而实现**排行榜**等功能。

RDB: 是 Redis DataBase 缩写（快照）

- RDB 是 Redis 默认的持久化方式。按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为 `dump.rdb`。通过配置文件中的 `save` 参数来定义快照的周期。

- 优点:

- 1、只有一个文件 `dump.rdb`，方便持久化。
- 2、容灾性好，一个文件可以保存到安全的磁盘。
- 3、性能最大化，`fork` 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能
4. 相对于数据集大时，比 AOF 的启动效率更高。

- 缺点：

1、**数据安全性低，会发生数据丢失**。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)

AOF：持久化：

- AOF 持久化(即 Append Only File 持久化)，则是将 Redis 执行的**每次写命令记录**到单独的日志文件中，当重启 Redis 会重新将持久化的日志中文件恢复数据。

- 当两种方式同时开启时，**数据恢复 Redis 会优先选择 AOF 恢复**

- 优点：

1、**数据安全**，aof 持久化可以配置 `appendfsync` 属性，有 `always`，每进行一次 命令操作就记录到 aof 文件中一次。

2、通过 `append` 模式写文件，即使中途服务器宕机，可以通过 `redis-check-aof` 工具解决数据一致性问题。

3、AOF 机制的 **rewrite** 模式。AOF 文件没被 `rewrite` 之前（**文件过大时会对命令 进行合并重写**），可以删除其中的某些命令（比如误操作的 `flushall`））

- 缺点：

1、AOF 文件比 RDB 文件大，且**恢复速度慢**。

2、数据集大的时候，比 `rd` 启动效率低。

- 两种持久化的优缺点是什么？

- AOF 文件比 RDB 更新频率高，优先使用 AOF 还原数据。
- AOF 比 RDB 更安全也更大
- RDB 性能比 AOF 好
- 如果两个都配了优先加载 AOF

如何选择合适的持久化方式

- 一般来说， 如果想达到足以媲美 PostgreSQL 的数据安全性，你应该同时使用两种持久化功能。在这种情况下，当 Redis **重启的时候会优先载入 AOF 文件来恢复原始的数据**，因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集**要完整**。
- 如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失，那么你可以只使用 RDB 持久化。

- 有很多用户都只使用 AOF 持久化，但并不推荐这种方式，因为**定时生成 RDB 快照（snapshot）**非常便于进行数据库备份，并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快，除此之外，使用 RDB 还可以避免 AOF 程序的 bug。
- 如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。

Redis 持久化数据和缓存怎么做扩容？

- 如果 Redis 被当做缓存使用，使用**一致性哈希实现动态扩容缩容**。
- 如果 Redis 被当做一个持久化存储使用，必须使用固定的 **keys-to-nodes 映射关系**，节点的数量一旦确定不能变化。否则的话（即 Redis 节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有 Redis 集群可以做到这样。

Redis 的过期键的删除策略

可以设置 Redis 中缓存的 key 的过期时间。

- **定时过期**：每个设置过期时间的 key 都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的 CPU 资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。
- **惰性过期**：只有当访问一个 key 时，才会判断该 key 是否已过期，过期则清除。该策略可以最大化地节省 CPU 资源，却对内存非常不友好。极端情况可能出现大量的过期 key 没有再次被访问，从而不会被清除，占用大量内存。
- **定期过期**：每隔一定的时间，会扫描一定数量的数据库的 **expires 字典** 中一定数量的 key，并清除其中已过期的 key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得 CPU 和内存资源达到最优的平衡效果。（expires 字典会保存所有设置了过期时间的 key 的过期时间数据，其中，key 是指向键空间中的某个键的指针，value 是该键的毫秒精度的 UNIX 时间戳表示的过期时间。键空间是指该 Redis 集群中保存的所有键。）

Redis 中同时使用了惰性过期和定期过期两种过期策略。

Redis key 的过期时间和永久有效分别怎么设置？

- **expire** 和 **persist** 命令。

我们知道通过 **expire** 来设置 key 的过期时间，那么对过期的数据怎么处理呢？

- 除了缓存服务器自带的缓存失效策略之外（Redis 默认的有 6 中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

- 1、定时去清理过期的缓存;
- 2、当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种缺点是维护大量缓存的 key 是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据

- redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

Redis 的内存淘汰策略有哪些（区分过期策略）

Redis 的内存淘汰策略是指在 Redis 的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

- 全局的键空间选择性移除
 - **noeviction**: 当内存不足以容纳新写入数据时，新写入操作会报错。
 - **allkeys-lru**: 当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 key。（这个是最常用的）
 - **allkeys-random**: 当内存不足以容纳新写入数据时，在键空间中，随机移除某个 key。
- 设置过期时间的键空间选择性移除
 - **volatile-lru**: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 key。
 - **volatile-random**: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个 key。
 - **volatile-ttl**: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 key 优先移除。
- 总结

Redis 的内存淘汰策略的选取并不会影响过期的 key 的处理。内存淘汰策略用于处理内存不足时的需要申请额外空间的数据；过期策略用于处理过期的缓存数据。

Redis 如何做内存优化？

- 可以好好利用 Hash,list,sorted set,set 等集合类型数据，因为通常情况下很多小的 Key-Value 可以用更紧凑的方式存放到一起。尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key，而是应该把这个用户的所有信息存储到一张散列表里面

线程模型

Redis 线程模型

Redis 基于 **Reactor 模式** 开发了网络事件处理器，这个处理器被称为文件事件处理器（**file event handler**）。它的组成结构为 4 部分：多个套接字、IO 多路复用程序、文件事件分派器、事件处理器。因为**文件事件分派器队列的消费是单线程的，所以 Redis 才叫单线程模型**。

- 文件事件处理器使用 I/O 多路复用（multiplexing）程序来同时监听多个套接字，并根据套接字目前执行的任务来为套接字关联不同的事件处理器。
- 当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时，与操作相对应的文件事件就会产生，这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

虽然文件事件处理器以单线程方式运行，但通过使用 I/O 多路复用程序来监听多个套接字，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与 redis 服务器中其他同样以单线程方式运行的模块进行对接，这保持了 Redis 内部单线程设计的简单性。

事务

什么是事务？

- 事务是一个单独的隔离操作：事务中的**所有命令都会序列化、按顺序地执行**。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

Redis 事务的概念（一次性、顺序性、排他性）

- Redis 事务的本质是**通过 MULTI、EXEC、WATCH 等一组命令的集合**。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。
- 总结说：redis 事务就是一**次性、顺序性、排他性**的执行一个队列中的一系列命令。

Redis 事务相关命令

Redis 事务功能是通过 **MULTI**、**EXEC**、**DISCARD** 和 **WATCH** 四个原语实现的

Redis 会将一个事务中的所有命令序列化，然后按顺序执行。

1. **redis 不支持回滚**，“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”，所以 Redis 的内部可以保持简单且快速。
 2. 如果在一个事务中的命令出现错误，那么所有的命令都不会执行；
 3. 如果在一个事务中出现运行错误，那么正确的命令会被执行。
- **WATCH** 命令是一个乐观锁，监控一个或多个键。可以为 Redis 事务提供 check-and-set (CAS) 行为。一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到 EXEC 命令。
 - **MULTI**：用于开启一个事务，返回 OK。MULTI 执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当 EXEC 命令被调用时，所有队列中的命令才会被执行。
 - **EXEC**：顺序执行事务块内的命令。
 - **DISCARD**：清空事务队列，并放弃执行事务。
 - **UNWATCH**：取消 watch 对所有 key 的监控。

事务管理（ACID）概述

Redis 的事务总是具有 ACID 中的一致性和隔离性（一致性？？感觉有问题），其他特性是不支持的。当服务器运行在 `_AOF_` 持久化模式下，并且 `appendfsync` 选项的值为 `always` 时，事务也具有耐久性。

Redis 事务支持隔离性吗（单线程自带隔离性）

- Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis 的事务是总是带有隔离性的。

Redis 事务保证原子性吗，支持回滚吗

- Redis 中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

Redis 事务其他实现（Lua 脚本）

- 基于 **Lua 脚本**，Redis 可以保证脚本内的命令一次性、按顺序地执行，其同时也不提供事务运行错误的回滚，执行过程中如果部分命令运行错误，剩下的命令还是会继续运行完
- 基于中间标记变量，通过另外的标记变量来标识事务是否执行完成，读取数据时先读取该标记变量判断是否事务执行完成。但这样会需要额外写代码实现，比较繁琐

集群方案

1、哨兵模式

哨兵的介绍

sentinel，中文名是哨兵。哨兵是 redis 集群机构中非常重要的一个组件，主要有以下功能：

- **集群监控**：负责监控 redis master 和 slave 进程是否正常工作。
- **消息通知**：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
- **故障转移**：如果 master node 挂掉了，会自动转移到 slave node 上。
- **配置中心**：如果故障转移发生了，通知 client 客户端新的 master 地址。

哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

- 故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了**分布式选举**的问题。
- 即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了。

哨兵的核心知识

- 哨兵至少需要 **3 个实例**，来保证自己的健壮性。
- 哨兵 + redis 主从的部署架构，是**不保证数据零丢失**的，只能保证 redis 集群的高可用性。
- 对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

2、官方 Redis Cluster 方案(服务端路由查询)

简介

- Redis Cluster 是一种**服务端 Sharding 技术**，3.0 版本开始正式提供。Redis Cluster 并没有使用一致性 hash，而是采用 slot(槽)的概念，一共分成 **16384** 个槽。将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行

方案说明

- 通过哈希的方式，将数据分片，每个节点均分存储一定哈希槽(哈希值)区间的数据，默认分配了 16384 个槽位
 - 每份数据分片会存储在多个互为主从的多节点上
 - 数据写入先写主节点，再同步到从节点(支持配置为阻塞同步)
 - 同一分片多个节点间的数据不保持一致性
 - 读取数据时，当客户端操作的 key 没有分配在该节点上时，redis 会返回转向指令，指向正确的节点
 - 扩容时时需要把旧节点的数据迁移一部分到新节点
- 在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 **6379**，另外一个就是 加 1w 的端口号，比如 **16379**。
 - 16379 端口号是用来进行节点间通信的**，也就是 cluster bus 的东西，cluster bus 的通信，用来进行故障检测、配置更新、故障转移授权。

cluster bus 用了另外一种二进制的协议，**gossip** 协议，用于节点间进行**高效的数据交换，占用更少的网络带宽和处理时间。**

节点间的内部通信机制

- 基本通信原理
- 集群元数据的维护有两种方式：集中式、Gossip 协议。redis cluster 节点间采用 gossip 协议进行通信。

分布式寻址算法

- hash 算法（大量缓存重建）
- 一致性 hash 算法（自动缓存迁移）+ 虚拟节点（自动负载均衡）
- redis cluster 的 hash slot 算法

优点

- 无中心架构**，支持动态扩容，对业务透明
- 具备 Sentinel 的监控和自动 Failover(故障转移)能力
- 客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可
- 高性能，客户端直连 redis 服务，免去了 proxy 代理的损耗

缺点

- 运维也很复杂，数据迁移需要人工干预
- 只能使用 0 号数据库
- 不支持批量操作(pipeline 管道操作)
- 分布式逻辑和存储模块耦合等

3、基于客户端分配

简介

- Redis Sharding 是 Redis Cluster 出来之前，业界普遍使用的多 Redis 实例集群方法。其主要思想是采用哈希算法将 Redis 数据的 key 进行散列，通过 hash 函数，特定的 key 会映射到特定的 Redis 节点上。Java redis 客户端驱动 jedis，支持 Redis Sharding 功能，即 ShardedJedis 以及结合缓存池的 ShardedJedisPool

优点

- 优势在于非常简单，服务端的 Redis 实例彼此独立，相互无关联，每个 Redis 实例像单服务器一样运行，非常容易线性扩展，系统的灵活性很强

缺点

- 由于 sharding 处理放到客户端，规模进一步扩大时给运维带来挑战。
- 客户端 sharding 不支持动态增删节点。服务端 Redis 实例群拓扑结构有变化时，每个客户端都需要更新调整。连接不能共享，当应用规模增大时，资源浪费制约优化

4、基于代理服务器分片

简介

- 客户端发送请求到一个代理组件，代理解析客户端的数据，并将请求转发至正确的节点，最后将结果回复给客户端

特征

- 透明接入，业务程序不用关心后端 Redis 实例，切换成本低
- Proxy 的逻辑和存储的逻辑是隔离的
- 代理层多了一次转发，性能有所损耗

业界开源方案

- Twitter 开源的 Twemproxy
- 豌豆荚开源的 Codis

5、Redis 主从架构

- 单机的 redis，能够承载的 QPS 大概就在上万到几万不等。对于缓存来说，一般都是用来支撑**读高并发**的。因此架构做成主从(master-slave)架构，一主多从，主负责写，并且将数据复制到其它的 slave 节点，从节点负责读。所有的**读请求全部走从节点**。这样也可以很轻松实现水平扩容，**支撑读高并发**。

redis replication -> 主从架构 -> 读写分离 -> 水平扩容支撑读高并发

redis replication 的核心机制

- redis 采用**异步方式**复制数据到 slave 节点，不过 redis2.8 开始，slave node 会**周期性地确认自己每次复制的数据量**；
- 一个 master node 是可以配置多个 slave node 的；
- slave node 也可以连接其他的 slave node；
- slave node 做复制的时候，不会 block master node 的正常工作；
- slave node 在做复制的时候，也不会 block 对自己的查询操作，它会用旧的数据集来提供服务；但是**复制完成的时候，需要删除旧数据集**，加载新数据集，这个时候就会暂停对外服务了；（类似 mvcc）
- slave node 主要用来进行横向扩容，做读写分离，扩容的 slave node 可以提高读的吞吐量。

注意：

- 如果采用了主从架构，那么建议必须**开启 master node 的持久化**，不建议用 slave node 作为 master node 的数据热备，因为那样的话，如果你关掉 master 的持久化，可能在 master 宕机重启的时候数据是空的，然后可能一经过复制，slave node 的数据也丢了。
- 另外，master 的各种备份方案，也需要做。万一本地的所有文件丢失了，从备份中挑选一份 rdb 去恢复 master，这样才能**确保启动的时候，是有数据的**，即使采用了后续讲解的高可用机制，slave node 可以自动接管 master node，但也可能 sentinel 还没检测到 master failure，master node 就自动重启了，还是可能导致上面所有的 slave node 数据被清空。

redis 主从复制的核心原理

- 当启动一个 slave node 的时候，它会发送一个 **PSYNC** 命令给 master node。
- 如果这是 slave node 初次连接到 master node，那么会触发一次 **full resynchronization** 全量复制。此时 master 会启动一个后台线程，开始生成一份 RDB 快照文件，

- 同时还会将从客户端 client 新收到的所有写命令缓存在内存中。RDB 文件生成完毕后，master 会将这个 RDB 发送给 slave，slave 会先写入本地磁盘，然后再从本地磁盘加载到内存中，
- 接着 master 会将内存中缓存的写命令发送到 slave，slave 也会同步这些数据。
- slave node 如果跟 master node 有网络故障，断开了连接，会自动重连，连接之后 master node 仅会复制给 slave 部分缺少的数据。

缺点

- 所有的 slave 节点数据的复制和同步都由 master 节点来处理，会造成 master 节点压力太大，使用主从结构来解决

生产环境中的 redis 是怎么部署的？

- redis cluster, 10 台机器，5 台机器部署了 redis 主实例，另外 5 台机器部署了 redis 的从实例，每个主实例挂了一个从实例，5 个节点对外提供读写服务，每个节点的读写高峰 qps 可能可以达到每秒 5 万，5 台机器最多是 25 万读写请求/s。
- 机器是什么配置？32G 内存+ 8 核 CPU + 1T 磁盘，但是分配给 redis 进程的是 10g 内存，一般线上生产环境，redis 的内存尽量不要超过 10g，超过 10g 可能会有问题。
- 5 台机器对外提供读写，一共有 50g 内存。
- 因为每个主实例都挂了一个从实例，所以是高可用的，任何一个主实例宕机，都会自动故障迁移，redis 从实例会自动变成主实例继续提供读写服务。
- 你往内存里写的是什么数据？每条数据的大小是多少？商品数据，每条数据是 10kb。100 条数据是 1mb，10 万条数据是 1g。常驻内存的是 200 万条商品数据，占用内存是 20g，仅仅不到总内存的 50%。目前高峰期每秒就是 3500 左右的请求量。

其实大型的公司，会有基础架构的 team 负责缓存集群的运维。

说说 Redis 哈希槽的概念？

- Redis 集群没有使用一致性 hash，而是引入了哈希槽的概念，Redis 集群有 16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。

Redis 集群会有写操作丢失吗？为什么？

- Redis 并不能保证数据的强一致性，这意味这在实际中集群在特定的条件下可能会丢失写操作。

Redis 集群之间是如何复制的？

- 异步复制

Redis 集群最大节点个数是多少？

- 16384 个

Redis 集群如何选择数据库？

- Redis 集群目前无法做数据库选择，默认在 **0 数据库**。

分区

Redis 是单线程的，如何提高多核 CPU 的利用率？

- 可以在**同一个服务器部署多个 Redis 的实例**，并把他们当作不同的服务器来使用，在某些时候，无论如何一个服务器是不够的，所以，如果你想使用多个 CPU，你可以考虑一下分片（shard）。

为什么要做 Redis 分区？

- 分区可以让 Redis 管理更大的内存，Redis 将可以使用所有机器的内存。如果没有分区，你最多只能使用一台机器的内存。**分区使 Redis 的计算能力通过简单地增加计算机得到成倍提升，Redis 的网络带宽也会随着计算机和网卡的增加而成倍增长。**

Redis 分区有什么缺点？

- 涉及多个 key 的操作通常不会被支持。例如**你不能对两个集合求交集**，因为他们可能被存储到不同的 Redis 实例（实际上这种情况也有办法，但是不能直接使用交集指令）。
- 同时操作多个 key,则**不能使用 Redis 事务**。

- 分区使用的粒度是 key，不能使用一个非常长的排序 key 存储一个数据集（The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set）
- 当使用分区的时候，数据处理会非常复杂，例如为了备份你必须从不同的 Redis 实例和主机同时收集 RDB / AOF 文件。
- 分区时动态扩容或缩容可能非常复杂。Redis 集群在运行时增加或者删除 Redis 节点，能做到最大程度对用户透明地数据再平衡，但其他一些客户端分区或者代理分区方法则不支持这种特性。然而，有一种**预分片**的技术也可以较好的解决这个问题。

分布式问题

Redis 实现分布式锁

- Redis 为单进程单线程模式，**采用队列模式将并发访问变成串行访问**，且多客户端对 Redis 的连接并不存在竞争关系 Redis 中可以使用 **setNx** 命令实现分布式锁。
- 当且仅当 key 不存在，将 key 的值设为 value。若给定的 key 已经存在，则 setNx 不做任何动作
- SETNX 是『SET if Not eXists』（如果不存在，则 SET）的简写。
- 返回值：设置成功，返回 1 。设置失败，返回 0 。
- 使用 setNx 完成同步锁的流程及事项如下：
- 使用 SETNX 命令获取锁，若返回 0（key 已存在，锁已存在）则获取失败，反之获取成功
- 为了防止获取锁后程序出现异常，导致其他线程/进程调用 setNx 命令总是返回 0 而进入死锁状态，**需要为该 key 设置一个“合理”的过期时间释放锁，使用 DEL 命令将锁数据删除**

如何解决 Redis 的并发竞争 Key 问题

- 所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！
- 推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）
- 基于 zookeeper **临时有序节点**可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在 zookeeper 上的与该方法对应的指定节点的目录下，生成一个唯一的临时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中**序号最小的一个**。当释放锁的时候，只需将

这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推 Zookeeper。

分布式 Redis 是前期做还是后期规模上来了再做好？为什么？

- 既然 Redis 是如此的轻量（单实例只使用 1M 内存），为防止以后的扩容，最好的办法就是一开始就启动较多实例。即便你只有一台服务器，你也可以一开始就让 Redis 以分布式的方式运行，使用分区，在同一台服务器上启动多个实例。
- 一开始就多设置几个 Redis 实例，例如 32 或者 64 个实例，对大多数用户来说这操作起来可能比较麻烦，但是从长久来看做这点牺牲是值得的。
- 这样的话，当你的数据不断增长，需要更多的 Redis 服务器时，你需要做的就是仅仅将 Redis 实例从一台服务迁移到另外一台服务器而已（而不用考虑重新分区的问题）。一旦你添加了另一台服务器，你需要将你一半的 Redis 实例从第一台机器迁移到第二台机器。

什么是 RedLock

- Redis 官方站提出了一种权威的基于 Redis 实现分布式锁的方式名叫 Redlock，这种方式比原先的单节点的方法更安全。它可以保证以下特性：
 1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
 2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区
 3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

缓存异常

什么是 redis 穿透？

- 就是用户请求透过 redis 去请求 mysql 服务器，导致 mysql 压力过载。但一个 web 服务里，极容易出现瓶颈的就是 mysql，所以才让 redis 去分担 mysql 的压力，所以这种问题是万万要避免的
- 解决方法：
 1. 从缓存取不到的数据，在数据库中也未取到，这时也可以将 key-value 对写为 key-null，缓存有效时间可以设置短点，如 30 秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个 id 暴力攻击
 2. 接口层增加校验，如用户鉴权校验，id 做基础校验，id<=0 的直接拦截；

3. **采用布隆过滤器**，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力

什么是 redis 雪崩？

- 就是 redis 服务由于**负载过大**而宕机，导致 mysql 的负载过大也宕机，最终整个系统瘫痪
- 解决方法：
 1. **redis 集群**，将原来一个人干的工作，分发给多个人干
 2. **缓存预热**（关闭外网访问，先开启 mysql，通过预热脚本将热点数据写入缓存中，启动缓存。开启外网服务）
 3. **数据不要设置相同的生存时间**，不然过期时，redis 压力会大

什么是 redis 穿透？

- 高并发下，由于一个 key 失效，而导致多个线程去 mysql 查同一业务数据并存到 redis（并发下，存了多份数据），而一段时间后，多份数据同时失效。导致压力骤增
- 解决方法：
 1. **分级缓存**（缓存两份数据，第二份数据生存时间长一点作为备份，第一份数据用于被请求命中，如果第二份数据被命中说明第一份数据已经过期，要去 mysql 请求数据重新缓存两份数据）
 2. **计划任务**（假如数据生存时间为 30 分钟，计划任务就 20 分钟执行一次更新缓存数据）

缓存预热

- **缓存预热**就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户**直接查询事先被预热的缓存数据**！
- **解决方案**
 1. 直接写个缓存刷新页面，上线时手工操作一下；
 2. 数据量不大，可以在项目启动的时候自动进行加载；
 3. 定时刷新缓存；

缓存降级

- 当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

- 缓存降级的最终目的是**保证核心服务可用，即使是有损的**。而且有些服务是无法降级的（如加入购物车、结算）。
- 服务降级的目的，是为了防止 Redis 服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis 出现问题，不去数据库查询，而是**直接返回默认值给用户**。

热点数据和冷数据

- 热点数据，缓存才有价值
- 对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。**频繁修改的数据，看情况考虑使用缓存**
- 对于热点数据，比如我们的某 IM 产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次。再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次。
- 数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。
- 那存不存在，修改频率很高，但是又不得不考虑缓存的场景呢？有！比如，这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，**点赞数，收藏数，分享数**等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到 Redis 缓存，减少数据库压力。

缓存热点 key

- 缓存中的一个 Key(比如一个促销商品)，在某个时间点过期的时候，恰好在这个时间点对这个 Key 有大量的并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端 DB 压垮。

解决方案

- **对缓存查询加锁，如果 KEY 不存在，就加锁，然后查 DB 入缓存，然后解锁**；其他进程如果发现有锁就等待，然后等解锁后返回数据或者进入 DB 查询

常用工具

Redis 支持的 Java 客户端都有哪些？官方推荐用哪个？

- Redisson、Jedis、lettuce 等等，官方推荐使用 **Redisson**。

Redis 和 Redisson 有什么关系？

- **Redisson 是一个高级的分布式协调 Redis 客户端，能帮助用户在分布式环境中轻松实现一些 Java 的对象** (Bloom filter, BitSet, Set, SetMultimap, SortedSortedSet, SortedSet, Map, ConcurrentMap, List, ListMultimap, Queue, BlockingQueue, Deque, BlockingDeque, Semaphore, Lock, ReadWriteLock, AtomicLong, CountDownLatch, Publish / Subscribe, HyperLogLog)。

Jedis 与 Redisson 对比有什么优缺点？

- Jedis 是 Redis 的 Java 实现的客户端，其 API 提供了比较全面的 Redis 命令的支持；Redisson 实现了 **分布式和可扩展的 Java 数据结构**，和 Jedis 相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等 Redis 特性。Redisson 的宗旨是促进使用者对 Redis 的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

其他问题

Redis 与 Memcached 的区别

- 两者都是非关系型内存键值数据库，现在公司一般都是用 Redis 来实现缓存，而且 Redis 自身也越来越强大了！Redis 与 Memcached 主要有以下不同：

对比参数	Redis	Memcached
类型	1. 支持内存 2. 非关系型数据库	1. 支持内存 2. 键值对形式 3. 缓存形式
数据存储类型	1. String 2. List 3. Set 4. Hash 5. Sort Set 【俗称 ZSet】	1. 文本型 2. 二进制类型
查询【操作】类型	1. 批量操作 2. 事务支持 3. 每个类型不同的 CRUD	1. 常用的 CRUD 2. 少量的其他命令
附加功能	1. 发布/订阅模式 2. 主从分区 3. 序列化支持 4. 脚本支持【Lua 脚本】	1. 多线程服务支持
网络 IO 模型	1. 单线程的多路 IO 复用模型	1. 多线程，非阻塞 IO 模式
事件库	自封转简易事件库 AeEvent	贵族血统的 LibEvent 事件库

对比参数	Redis	Memcached
持久化支持	1. RDB 2. AOF	不支持
集群模式	原生支持 cluster 模式, 可以实现主从复制, 读写分离	没有原生的集群模式, 需要依靠客户端来实现往集群中分片写入数据
内存管理机制	在 Redis 中, 并不是所有数据都一直存储在内存中, 可以将一些很久没用的 value 交换到磁盘	Memcached 的数据则会一直在内存中, Memcached 将内存分割成特定长度的块来存储数据, 以完全解决内存碎片的问题。但是这种方式会使得内存的利用率不高, 例如块的大小为 128 bytes, 只存储 100 bytes 的数据, 那么剩下的 28 bytes 就浪费掉了。
适用场景	复杂数据结构, 有持久化, 高可用需求, value 存储内容较大	纯 key-value, 数据量非常大, 并发量非常大的业务

1. memcached 所有的值均是简单的字符串, redis 作为其替代者, 支持更为**丰富的数据类型**
2. redis 的**速度比 memcached 快很多**
3. redis 可以持久化其数据

如何保证缓存与数据库双写时的数据一致性?

- 你只要用缓存, 就可能会涉及到缓存与数据库双存储双写, 你只要是双写, 就一定会有数据一致性的问题, 那么你如何解决一致性问题?
- 一般来说, 就是如果你的系统不是严格要求缓存+数据库必须一致性的话, 缓存可以稍微的跟数据库偶尔有不一致的情况, 最好不要做这个方案, **读请求和写请求串行化, 串到一个内存队列里去**, 这样就可以保证一定不会出现不一致的情况
- **串行化**之后, 就会导致系统的吞吐量会大幅度的降低, 用比正常情况下多几倍的机器去支撑线上的一个请求。
- 还有一种方式就是可能会暂时产生不一致的情况, 但是发生的几率特别小, 就是**先更新数据库, 然后再删除缓存**。

问题场景	描述	解决
先写缓存, 再写数据库, 缓存写成功, 数据库写失败	缓存写成功, 但写数据库失败或者响应延迟, 则下次读取 (并发读) 缓存时, 就出现脏读	这个写缓存的方式, 本身就是错误的, 需要改为先写数据库, 把旧缓存置为失效; 读取数据的时候, 如果缓存不存在, 则读取数据库再写缓存
先写数据库, 再写缓存, 数据库写成功, 缓存写失败	写数据库成功, 但写缓存失败, 则下次读取 (并发读) 缓存时, 则读不到数据	缓存使用时, 假如读缓存失败, 先读数据库, 再回写缓存的方式实现

问题场景	描述	解决
需要缓存异步刷新	指数据库操作和写缓存不在一个操作步骤中，比如在分布式场景下，无法做到同时写缓存或需要异步刷新（补救措施）时候	确定哪些数据适合此类场景，根据经验值确定合理的数据不一致时间，用户数据刷新的时间间隔

Redis 常见性能问题和解决方案？

1. Master 最好不要做任何持久化工作，包括内存快照和 AOF 日志文件，特别是不要启用内存快照做持久化。
2. 如果数据比较关键，某个 Slave 开启 AOF 备份数据，策略为每秒同步一次。
3. 为了主从复制的速度和连接的稳定性，Slave 和 Master 最好在同一个局域网内。
4. 尽量避免在压力较大的主库上增加从库
5. Master 调用 BGREWRITEAOF 重写 AOF 文件，**AOF 在重写的时候会占大量的 CPU 和内存资源，导致服务 load 过高，出现短暂服务暂停现象。**
6. 为了 Master 的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master←Slave1←Slave2←Slave3…，这样的结构也方便解决单点故障问题，实现 Slave 对 Master 的替换，也即，如果 Master 挂了，可以立马启用 Slave1 做 Master，其他不变。

一个字符串类型的值能存储最大容量是多少？

- 512M

Redis 如何做大量数据插入？

- Redis2.6 开始 redis-cli 支持一种新的被称之为 **pipe mode** 的新模式用于执行大量数据插入工作。

假如 Redis 里面有 1 亿个 key，其中有 10w 个 key 是以某个固定的已知的前缀开头的，如果将它们全部找出来？

- 使用 **keys 指令** 可以扫出指定模式的 key 列表。
- 对方接着追问：如果这个 redis 正在给线上的业务提供服务，那使用 keys 指令会有什么问题？
这个时候你要回答 redis 关键的一个特性：redis 的单线程的。**keys 指令会导致线程阻塞一段时间，线上服务会停顿**，直到指令执行完毕，服务才能恢复。这个时候可以使用 **scan 指令**，scan 指令可以无阻塞的提取出指定模式的 key 列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用 keys 指令长。

使用 Redis 做过异步队列吗，是如何实现的

- 使用 **list** 类型保存数据信息，**rpush** 生产消息，**lpop** 消费消息，当 **lpop** 没有消息时，可以 **sleep** 一段时间，然后再检查有没有信息，如果不想 **sleep** 的话，可以使用 **blpop**，在没有信息的时候，会一直阻塞，直到信息的到来。redis 可以通过 **pub/sub** 主题订阅模式实现一个生产者，多个消费者，当然也存在一定的缺点，当**消费者下线时，生产的消息会丢失**。

Redis 如何实现**延时队列**

- 使用 **sortedset**，使用时间戳做 **score**，消息内容作为 **key**，调用 **zadd** 来生产消息，消费者使用 **zrangbyscore** 获取 **n** 秒之前的数据做轮询处理。

Redis 回收使用的是**什么算法**？

- LRU 算法