Redis 基础应用

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 1579 字 大约 5 分钟

问题清单

Questions

1. 为什么用 Redis 作为缓存?
2. Redis 除了缓存之外的应用场景?
3. Redis 实现消息队列有哪些方式?
4. Redis 与 Memcached 对比?
5. 本地缓存、分布式缓存、多级缓存如何实现?
6. 为什么 Redis 采用跳表而不采用二叉树或平衡树?
7. Redis 为什么不能做持久化数据库?

问题回答

1. 为什么用 Redis 作为缓存?

Answer

Redis 通常可以用来当作缓存中间件

- 基于内存的数据库，读写性能高
- `Redis` 底层设计了多种数据结构，被上层应用，提升了性能
- `Redis` 是单线程模型，避免了多线程的上下文切换、锁竞争、并发问题，对开发与调试友好，可维护性高
  - `Redis Server` 是多线程的，但是其请求处理流程为单线程，所以一般认为 `Redis` 为单线程模型
- `Redis` 采用基于多路复用的 `I/O模型`，为 `select/epoll` 机制，一旦监听到 `FD` 上有请求到达，则会触发事件并放入消息队列中，等待处理

  `Redis` 一直在对事件队列进行处理，谁有数据就处理谁，实现了高性能

2. Redis 除了缓存之外的应用场景?

Answer

- 实现分布式锁
- 实现消息队列
- 实现限流器
- 实现幂等性接口
- 实现单点 `Token` 登陆（有状态）分布式 `Session`
- 实现分布式 ID
- 通过消息订阅机制实现聊天
- 基于 `Redis` 提供的数据结构，实现排行榜、点赞关注列表、社交 Feed 流、签到统计、网站UV统计、地理坐标等

3. Redis 实现消息队列有哪些方式?

Answer

1. 采用 `Redis` 的双向阻塞 `List`
   - 提供了 `LPUSH` 与 `RPOP` 以及阻塞式的 `BRPOP` 命令，满足消息有序且阻塞式获取消息的需求
   - 通过为每条消息设置全局唯一 ID 来确保可以处理重复消息
   - 消息可靠性：通过 `BRPOPLPUSH` 命令实现备份 `List`
   - 缺点：不支持多个消费者消费同一条消息
2. 采用 `Redis` 发布订阅模式 `PUB/SUB`
   - 只要发布者发布消息，所有订阅者都能收到消息，订阅者都是平等的。
   - 缺点：不支持持久化
3. 采用 `Redis` 的 `Stream`
   - 支持消息的持久化、支持自动生成全局唯一 ID、支持 ack 确认消息的模式、支持消费组模式
     - `Redis` 本身数据结构自带持久化功能
     - `XADD` 命令在插入数据时，自动生成全局唯一ID
     - `XREAD` 命令读取消息队列时，支持设置 `BLOCK` 实现阻塞读
     - 消费组读取消息：同一消费组的消费者不能读取同一条消息，不同消费组的消费者可以读取同一条消息
     - 消息可靠性：`XPENDING` 命令查看已读取但未确认的消息，`XACK` 命令确认消息
   - 缺点：
     - `Redis` 本身虽然做到了持久化机制，但仍然存在持久化失败的情况
     - 主从切换时造成数据丢失
     - `Stream` 存在最大长度限制，消息积压概率大

如果需要满足高可靠性，采用专业的消息队列 `RocketMQ`、`RabbitMQ`、`Kafka` 等。如：下单之后的一系列操作

否则，可以使用 `Stream`。如：实时日志处理、实时数据采集、消息通知

4. Redis 与Memcached 对比?

Answer

- `Redis` 支持多种类型的数据结构，而 `Memcached` 只支持 `key-value` 字符串形式
- `Redis` 支持持久化，而 `Memcached` 不支持
- `Redis` 为单线程模型，而 `Memcached` 为多线程模型，采用多线程处理请求，充分利用多核，提升了 `IO` 效率

  数据量较大时，性能高于 `Redis`；数据量较小时，差别不大
- `Redis` 的内存淘汰策略较多，并且不需要设置内存上限；而 `Memcached` 只有 `LRU` 内存淘汰策略，且必须设置内存上限
- `Redis` 可以设置主从与哨兵来保证高可用性，而 `Memcached` 没有此功能
- `Redis` 采用固定哈希槽位实现集群化，而 `Memcached` 采用一致性哈希算法实现集群化

5. 本地缓存、分布式缓存、多级缓存如何实现?

Answer

- 本地缓存：存储在本地应用进程当中，随着应用进程的消亡而消失
  - 优点：读取速度快，没有网络请求发送与传输的成本
  - 缺点：不支持持久化，不能够存储大量缓存数据，只存在于单个进程中，不能被多个进程共享
  - 实现：可以采用 `Java` 自身的 `HashMap`，也可以使用 `Guava`、`Hutool` 等本地缓存工具类库
- 分布式缓存：部署于独立的缓存数据库中间件上，可以被多个分布式微服务应用共享
  - 优点：支持大量数据存储，部分中间件支持持久化，基于内存存储，速度较快
  - 缺点：性能低于本地缓存；在分布式场景下，部署和运维成本高（需要搭建集群保证高可用）
  - 实现：`Memcached`、`Redis`
- 多级缓存：本地缓存作为一级缓存，分布式缓存作为二级缓存，最后一级为数据库

6. 为什么 Redis 采用跳表而不采用二叉树或平衡树?

Answer

对比 `MySQL` 的索引数据结构来看：

- 第一，B+ 树相对于跳表的优势就在于层数少，磁盘 `IO` 次数少

  而 `Redis` 是基于内存的数据库，因此不存在磁盘 `IO`，当然也不需要考虑此问题。`MySQL` 是基于磁盘存储，所以需要考虑磁盘 `IO`
- 第二，B+ 树在进行插入时，需要进行合并与拆分节点以保持树的平衡，而跳表插入则不需要考虑这一点

  因此跳表写性能优于 B+ 树

7. Redis 为什么不能做持久化数据库?

Answer

- 内存宝贵
- 不支持事务回滚
- 断电故障问题
- 数据量过大时，会出现内存溢出
- 数据导出与数据分析，`MySQL` 支持将表结构与数据导出为 `SQL` 文件，而 `Redis` 不支持
- `Redis6.0` 之前无法做到细粒度的权限控制

上次编辑于: 2023/6/23 15:31:06
贡献者: ltyzzz

Redis 线程模型

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 1246 字 大约 4 分钟

问题清单

Questions

1. 如何理解 Redis 单线程模型?
2. Redis 6.0 之前为什么不引入多线程? 6.0 之后为什么引入多线程?
3. 对比一下各类网络 IO 模型?
4. 实现IO多路复用有哪几种方式? 对比一下

问题回答

1. 如何理解 Redis 单线程模型?

Answer

- `Redis Server` 为多线程模型，有一些耗时的操作比如 `AOF` 刷盘操作等需要后台开启一个线程去执行
- 但 `Redis` 处理客户端命令请求为单线程，其采用了 `IO多路复用` 模型实现了高性能

具体 `IO多路复用` 模型是为了解决阻塞与非阻塞模型的效率低问题

阻塞与非阻塞模型中，用户应用均需要以阻塞或自旋的方式读取网卡数据以及等待数据从内核态拷贝到用户态缓冲区，无法有效利用CPU

`IO多路复用` 则是依赖于 `FD` 文件描述符，优先处理已经就绪的 `IO事件`

- 利用一个线程监听多个 `FD`
- 当某个 `FD` 可读、可写时得到通知，用户进程找到就绪的 `Socket`，依次调用 `recvfrom`
- 此时内核拷贝数据到用户空间，用户进程处理数据

2. Redis 6.0 之前为什么不引入多线程？6.0 之后为什么引入多线程？

Answer

- Redis的瓶颈并不在于性能，而是在于内存和网络延迟。
- 引入多线程会提升复杂度，需要面临多线程下上下文切换成本、并发安全问题，开发与调试难度增大
- Redis6.0之后引入多线程专门用来处理网络I/O，以此充分利用CPU资源

3. 对比一下各类网络 IO 模型？

Answer

- 阻塞 IO

  1. 用户进程调用 `recvfrom` 尝试读取数据，但此时数据未到达，进入阻塞等待状态
  2. 数据到达并拷贝到内核缓冲区，此时数据准备就绪
  3. 切换到内核态，将内核数据拷贝到用户缓冲区
  4. 用户进程解除阻塞，开始处理数据

  阻塞 IO 模式下，用户会从发起 `recvfrom` 指令开始，一直阻塞到数据拷贝到用户缓冲区

- 非阻塞 IO

  1. 用户进程调用 `recvfrom` 尝试读取数据，但此时数据未到达，返回异常并循环读取
  2. 数据到达并拷贝到内核缓冲区，此时数据准备就绪，与此同时，用户进程进入阻塞态
  3. 切换到内核态，将内核数据拷贝到用户缓冲区
  4. 用户进程解除阻塞，开始处理数据

  `非阻塞IO` 模式下，用户会从发起 `recvform` 指令开始，循环读取数据。直到数据抵达内核缓冲区，用户进程进入阻塞状态。直到数据拷贝到用户缓冲区，用户进程解除阻塞，开始读取数据

- IO 多路复用

  利用单个线程来同时监听多个 `FD`，并在某个 `FD` 可读可写时得到通知，避免无效的等待，充分利用 `CPU` 资源

  1. 用户进程调用 `select` 函数，指定监听的 `FD` 集合，之后用户进程阻塞
  2. 任意一个或多个 `socket` 数据准备就绪则返回 `readable`，用户进程解除阻塞
  3. 用户进程找到就绪的 `socket`，依次调用 `recvfrom` 读取数据，内核拷贝数据到用户缓冲区，用户进程开始处理数据

4. 实现IO多路复用有哪几种方式？对比一下

Answer

有三种方式：select、poll、epoll

- select / poll 与 epoll 对比：

  `select` 和 `poll` 都是当被监听的数据准备好之后，内核会将 `FD` 整个数据返回给用户态，然后用户态在遍历一遍 `FD` 集合，用户态获取数据的时间复杂度为 `O(n)`

  `epoll` 相当于内核数据准备好之后，会将准备好的这部分数据写入用户态缓冲区，用户态获取数据的时间复杂度为 `O(1)`，省去了遍历操作

- `select` 与 `poll` 对比

  `select` 的 `fd` 通过 `long` 数组进行存储，共 1024 个 `bit` 位，一个 `long` 位 32 位 4 字节，因此数组长度为 32。每个 `bit` 位代表一个 `fd`，0 为未就绪，1 为已就绪。

  `poll` 的 `fd` 通过链表进行存储，理论无上限，但是每次返回给用户态的数据仍然是整个 `fd` 数据。这意味着，监听的 `fd` 越多，遍历消耗时间越久，性能会下降

- `epoll` 通过红黑树保存需要监听的 `fd`，为每个 `fd` 设置一个监听函数，当 `fd` 准备就绪时触发该监听函数，并将准备就绪的 `fd` 添加到链表中。用户进程通过 `epoll_wait` 函数等待数据准备完毕

上次编辑于: 2023/6/23 15:31:06
贡献者: ltyzzz

Redis 内存管理

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 955 字 大约 3 分钟

问题清单

Questions

1. Redis 的内存淘汰机制是什么?
2. LRU 与 LFU 是如何实现的?
3. Redis 的过期删除策略是什么?
4. 定期删除的具体实现流程是什么?

问题回答

1. Redis 的内存淘汰机制是什么?

Answer

- 不进行数据淘汰：`noeviction`

  当运行内存超过最大限度时，会抛出 `OOM` 错误

- 进行数据淘汰
  - 在设置过期时间的 `key` 中进行淘汰：
    - `volatile-random`：随机淘汰
    - `volatile-ttl`：优先淘汰更早过期的key
    - `volatile-lru`：优先淘汰最久未使用的key
    - `volatile-lfu`：优先淘汰使用频率最低的key
  - 在所有 `key` 中进行淘汰：`volatile-random` / `volatile-lru` / `volatile-lfu`

2. LRU 与 LFU 是如何实现的?

Answer

传统 LRU 实现：

- `LRU` 算法原理为通过链表从前向后记录最近一次使用过的 `key`。每次使用 `key` 时，都会将该 `key` 移动到链表的头部。
- 需要进行内存淘汰时，删除链表尾部的 `key`
- 问题为：需要用链表管理全部缓存数据，增加内存开销；每次使用 `key` 时，需要移动链表数据，性能低

Redis LRU 实现：

- `Redis` 每一个 `key` 都有一个字段用于记录最后访问的时间。每次进行内存淘汰时，随机选取 5 个key，淘汰最久没有使用的 `key`。

LRU 实质问题为：

- 存在 `key` 污染，当一个大 `key` 只被读取一次，但是会长时间停留在内存中

LFU实现：

- LFU算法原理是根据数据访问频率来淘汰数据，核心思想为："若数据过去被访问过多次，那么将来也可能访问多次"
- 相比于 `LRU`，`LFU` 除了记录上次访问的时间，还记录了数据访问频率
  - 高 16bit 数据用于记录上次访问的时间，低 8bit 数据用于记录数据访问频率 `logc`
  - 访问时的衰减操作：数据访问频率 `logc` 是会随着时间而衰减的，衰减的值与前后访问的时间差有关
    - 时间差越大，衰减值越大
  - 访问时的自增操作：对于 `logc` 值越大的 `key`，其 `logc` 值就越难增加

3. Redis 的过期删除策略是什么?

Answer

Redis 过期删除策略为：定时删除、定期删除、惰性删除

- 定时删除：每个 `key` 绑定一个定时器，当超过过期时间时，将该 `key` 删除
  - 优点：可以保证过期 `key` 一旦过期，立马被删除，对于内存很友好
  - 缺点：对 `CPU` 不友好，每个 `key` 均需要对应一个定时器，不利于性能
- 惰性删除
  - 优点：`CPU` 友好，简单实现
  - 缺点：内存不友好
- 定期删除：每隔一段时间随机从 `Redis` 中选取一定数量的 `key` 进行判断。原理是 `serverCron` 定时任务
  - 优点：综合考虑了 `CPU` 与内存
  - 缺点：难以控制定期删除的时间间隔和每次删除花费的时间

4. 定期删除的具体实现流程是什么?

Answer

- 遍历数据库，从过期字典获取当前数据库的带有过期时间 `key` 的数量
  - 若该数量为 0，则直接跳过这个数据库，接着遍历
  - 否则，挑选限定数量的 `key`，检查是否过期，若过期则直接删除
  - 如果本轮检查的已过期 `key` 数量超过了 25%，则继续检查；否则检查下一个数据库
- 还需要判断当前处理时间是否到达时间上限，若到达则停止处理
- 通过全局变量 `current_db` 记录检查进度，便于之后接着检查
- 当数据库遍历一遍后，会重置 `current_db` 为 0

上次编辑于: 2023/6/23 15:31:06
贡献者: ltyzzz

Redis 持久化

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 1018 字 大约 3 分钟

问题清单

Questions

1. Redis 如何实现持久化?
2. RDB 持久化的实现原理是什么?
3. AOF 持久化的实现原理是什么?
4. AOF 重写原理是什么?
5. AOF 重写过程中，主进程什么时候会阻塞?

问题回答

1. Redis 如何实现持久化?

Answer

通过 `RDB` 持久化与 `AOF` 持久化实现， `RDB` 持久化为默认持久化方式，当开启 `AOF` 持久化后，采用 `AOF` 持久化。

`Redis` 还提供了混合持久化方式，结合了 `RDB` 与 `AOF`，即 `AOF` 文件中第一部分是 `RDB` 的全量数据，第二部分是追加的写命令增量数据。

2. RDB 持久化的实现原理是什么?

Answer

`Redis` 通过自动保存机制，当满足一定条件，会将 `Redis` 内存上的数据持久化到 `RDB` 文件中。

`RDB` 持久化的原理为利用 `serverCron` 定时任务，每隔一段时间自动执行一次，其中一项工作则是检查 `RDB` 持久化条件是否满足。

默认的持久化条件为：服务器在 900s / 300s / 60s 内对数据库至少进行 1 / 10 / 10000 次操作，则开启持久化

- 通过 `dirty` 和 `lastsave` 属性判断

持久化的具体实现命令为：`save` / `bgsave`

- `save` 命令会阻塞 `redis` 服务器进程，而 `bgsave` 会创建一个子进程，后台进行持久化文件

`RDB` 缺点在于其可能会造成数据丢失，在两次持久化的间隔时间内，`redis` 宕机会导致数据丢失

3. AOF 持久化的实现原理是什么?

Answer

`Redis` 在执行一个写命令后，会将正在执行的写命令追加到服务器 `aof_buff` 缓冲区末尾并写入 `AOF` 文件，但是真正的刷盘操作需要根据策略确定

- Always：每次执行写命令后，均会将 `page_cache` 中的写命令写入到硬盘
- Everysec：每隔一秒将 `page_cache` 中的写命令写入到硬盘，该操作由专门的线程进行负责
- No：不负责刷盘，何时刷盘由操作系统自行决定

4. AOF 重写原理是什么?

Answer

由于 `AOF` 原理是追加写命令，这会导致 `AOF` 文件过大，当文件大小超过一定阈值后，就会执行 `AOF` 重写机制

- 创建新的 `aof` 文件，遍历全部数据库以及数据库中的全部 `key`，忽略已过期的 `key`，根据 `key` 类型对 `key` 进行重写，最后覆盖旧的 `aof` 文件
- 创建新的 `aof` 文件目的在于防止 `aof` 重写失败，污染原有的 `aof` 文件

`AOF` 可以通过 `bgrewriteof` 命令实现后台重写

- 新创建一个后台子进程，进行重写操作，不影响主进程处理客户端请求
- 后台重写的原理是利用了 **写时复制技术**
  - 当进行后台重写时，主进程会将页表复制一份给子进程，但是物理内存并没有进行复制，这时实现了共享数据，但是数据是只读的。
  - 当主进程收到客户端请求，去修改数据时，会修改物理内存数据，此时 `CPU` 会触发写保护中断，对被修改数据执行物理内存复制，不影响其他共享数据，并重新设置映射关系
  - 发生写时复制时，会造成主进程与子进程数据不一致，因此 `Redis` 设置了 `AOF` 重写缓冲区，主进程会将写命令追加到 `AOF` 缓冲区与重写缓冲区。

    当子进程结束重写后，会发送信号给父进程。父进程收到信号后，会调用信号处理函数，将AOF重写缓冲区中的所有内容追加到新的 `AOF` 文件中，并替换旧的 `AOF` 文件

5. AOF 重写过程中，主进程什么时候会阻塞?

Answer

1. fork子进程
2. 发生写时复制
3. 主进程调用信号处理函数

上次编辑于: 2023/6/23 15:31:06

贡献者: ltyzzz

Redis 缓存常见问题

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 1381 字 大约 5 分钟

问题清单

Questions

1. 常见的缓存问题有哪些?
2. 缓存与数据库的一致性问题?
3. 如何解决第二步失败的问题?
4. 主从复制场景下带来了什么一致性问题?
5. Redis 并发竞争 Key 问题如何解决?

问答回答

1. 常见的缓存问题有哪些?

Answer

- 缓存穿透：客户端数据在缓存和数据库中都不存在，最终请求会到达数据库
  - 缓存空对象：当请求到达数据库，当数据库不存在时，则缓存一个空对象。之后再次请求时，则返回空对象
  - 布隆过滤器
  - 非法请求限制：用户明知道数据不存在，但是仍然在恶意查询，因此需要在 `API` 入口处判断请求参数是否合理
- 缓存雪崩：同一时间段内有大量的key同时失效，导致大量请求进入数据库
  - 给不同的 `key` 设置随机的 `TTL` 值
  - 给缓存业务添加限流、降级、熔断等策略
  - 设置多级缓存
  - 构建高可用的 `Redis` 集群
- 缓存击穿：热点 `key` 问题，被高并发访问的 `key` 失效，导致大量请求进入数据库
  - 互斥锁实现：并发线程中其中一个线程获取互斥锁并重建缓存，其余线程阻塞等待，重建之后其他线程再去查询
  - 逻辑过期：并发线程中其中一个线程发现逻辑时间过期，去获取互斥锁并重建缓存，其余线程直接返回过期数据，重建之后其他线程查询的为最新数据

2. 缓存与数据库的一致性问题?

Answer

引入缓存后，选择更新缓存与数据库策略时，会引起不同程度的一致性问题。

1. 先更新缓存后更新数据库 or 先更新数据库后更新缓存

> 当二者之一宕机后，都会造成整体操作失败，从而造成一致性问题

2. 并发操作共享资源会带来一致性问题，使得数据库与缓存的数据不一致

并发操作需要加分布式锁去解决，而且从缓存利用率来看，及时地更新缓存并不可取，因为缓存数据并不会被马上被读取。

因此需要采用新的策略：删除缓存

1. 先删除缓存，后更新数据库

> 并发问题：当 A 先删除了缓存，此时 B 读缓存发现不存在，然后从数据库中查询读到旧值，此时A更新了数据库，最后 B 将旧值写入了缓存
>
> - 此时缓存为旧值，数据库为新值，二者不一致

2. 先更新数据库，后删除缓存

> 并发问题：A 读缓存发现缓存不存在，去读取数据库得到旧值，B 更新数据库，然后删除缓存，A 将旧值写入缓存
>
> - 此时缓存为旧值，数据库为新值，二者不一致
> - 发生此问题的条件
>   - 缓存刚好失效
>   - 更新数据库 + 删除缓存时间 < 读数据库 + 写入缓存时间
>   - 读写请求并发
> - 发生此问题的概率很小，因为写数据库需要加锁，耗时长

3. **如何解决第二步失败的问题?**

   需要确保两步均成功执行，如果第二步失败，则需要去重试执行第二步，保证最终一致性

   - 同步重试会导致一直占据 CPU 线程资源，并且如果当前系统宕机，则重试请求会丢失，将永远无法实现一致。

   因此需要依靠消息队列，采用异步重试方式实现一致性

   - 消息队列保证了可靠性，写到队列的消息，成功消费前不会丢失，可以持久化
   - 消息队列保证了消息成功投递，下游成功获取到消息并消费，消息才会被删除，否则将重试投递

   也可以依靠 canal 订阅 MySQL 更新日志，然后自动将日志投递到下游的消息队列，消息队列再投递消息去删除缓存

4. **主从复制场景下带来了什么一致性问题?**

   1. Redis 主库与从库之间的一致性问题

      - 主库有从库没有 - 复制延迟
      - 主库没有从库有 - 主库某 key 设置了过期时间，过期后短时间内能从从库读取到
      - 主库新增了数据，但还未来得及同步到从库中，主库宕机，从库晋升为新的主机时，丢失了部分数据

   2. MySQL 主从复制延迟，造成数据库与缓存之间的一致性问题

      - A 更新 MySQL 主库，并删除了缓存，B 查询缓存未命中，查询 MySQL 从库得到了旧值。此时 MySQL 主从同步完成，B 将旧值回种缓存

        此时 MySQL 存储的为新值，Redis 存储的为旧值

      - 需要采用延迟双删策略，且延迟时间需要大于 MySQL 主从同步时间

5. **Redis 并发竞争 Key 问题如何解决?**

   1. 采用分布式锁
   2. 对 `key` 添加时间戳，即乐观锁思想。若获取的时间戳小于当前 `key` 的时间戳，则取消执行或继续重试

3. 采用消息队列，将 `set` 操作串行化

上次编辑于: 2023/6/23 15:31:06
贡献者: ltyzzz

Redis 性能评估

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 977 字 大约 3 分钟

问题清单

Questions

1. 如何评估 Redis 的性能?
2. Redis 的慢查询功能是什么？如何使用它定位 Redis 性能瓶颈?
3. Redis 内存碎片是什么？产生原因是什么?
4. 什么是 Big Key?
5. Big Key的危害?
6. 如何避免或解决 Big Key 问题?

问题回答

1. 如何评估 Redis 的性能?

Answer

可以从 5 个方面评估：性能 `Performance`、内存 `Memory`、基本活动 `Basic Activity`、持久性 `Persistence`、错误 `Error`

对于性能：请求响应时间、每秒处理的请求数量、缓存命中率

对于内存：内存占用率、内存碎片率、内存淘汰 `key` 的数量

对于基本活动：客户端连接数、`key` 数量、`slave` 数量、最近一次主从交互时间

对于持久性：最后一次持久化时间、最后一次持久化后数据库的更改次数

对于错误：`key` 查找失败次数

2. Redis 的慢查询功能是什么？如何使用它定位 Redis 性能瓶颈?

> **Answer**
>
> 慢查询日志功能用于记录执行时间超过给定时长的命令请求， 用户可以通过这个功能产生的日志来监视和优化查询速度。
>
> `Redis` 将所有的慢查询日志保存在 `slowlog` 链表中，每个链表节点为一个 `slowlogEntry` 结构，代表一条慢查询日志
>
> 可以通过 `config set` 命令配置命令执行时间阈值与慢查询日志的最大长度
>
> 超出最大长度，则将链表末尾日志移除，新日志位于链表头部

3. Redis内存碎片是什么？产生原因是什么?

> **Answer**
>
> Redis分配内存空间大于实际的占用空间，多余的部分即是内存碎片
>
> 内存碎片形成原因分为内部原因与外部原因:
>
> - 底层内存分配器的分配策略无法真正做到按需分配，其按照一系列固定大小划分内存空间。当程序申请的内存接近某个固定值时，会分配相应的大小
> - `key-value` 会被频繁的修改，造成空间的扩容或收缩

4. 什么是Big Key?

> **Answer**
>
> 大 `key` 并不是指 `key` 的值很大，而是 `key` 对应的 `value` 很大。
>
> 一般而言，下面这两种情况被称为大 `key`：
>
> - `String` 类型的值大于 `10` KB;
> - `Hash`、`List`、`Set`、`ZSet` 类型的元素的个数超过 5000 个;

5. Big Key 的危害?

Answer

1. 减慢 `Redis` 持久化速度，加长服务端进程的阻塞时间

   - `AOF` 后台重写时 `fork` 子进程，如果数据过大，会造成页表过大，加长 `fork` 时间，进而长时间阻塞主进程

   - `AOF` 持久化追加命令到 `AOF` 文件中时，最终需要将其落入磁盘中，如果数据过大，会加长这一过程，进而长时间阻塞主进程

   - `AOF` 和 `RDB` 后台执行时，有可能发生写时复制，如果数据过大，则会造成物理内存复制时间过长，进而长时间阻塞主进程

2. 客户端请求时间超长甚至超时，响应速度慢
3. 对 `Big Key` 的读取、更新等操作会阻塞服务端进程
4. 引起流量高峰，可能使 `Redis` 服务器崩溃
5. `Big Key` 过大或过多时，会造成 `OOM` 错误，也可能导致重要的 `key` 由于内存淘汰策略被删除
6. 影响主从同步

6. 如何避免或解决 Big Key 问题?

Answer

1. 设计阶段将可能存在的 `Big Key` 进行拆分，通过 `hash` 数据结构进行存储
2. 通过 `redis-cli --bigkeys` 命令找到 `Big Key`，并通过 `unlink` 命令将其异步删除
3. 实时对 `Redis` 进行性能指标监控
4. 定期清理失效的 `key`，防止内存溢出

上次编辑于: 2023/6/23 15:31:06
贡献者: ltyzzz

Redis 集群

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 2530 字 大约 8 分钟

问题清单

Questions

1. Redis 主从复制如何实现?
2. 从服务器宕机或出现网络故障，如何保证主从一致性?
3. 主从模式下，如何判断某个节点是否正常工作?
4. 主从复制中，出现的 replication buffer 和 repl_backlog_buffer 有什么区别?
5. 主从切换如何减少数据丢失?
6. Sentinel 哨兵节点作用是什么?
7. Sentinel 如何判断主节点是否宕机?
8. 由哪个 Sentinel 节点完成最终的故障转移?
9. Leader Sentinel 如何选出新的主节点?
10. Sentinel 集群如何组成?
11. Redis 如何保证高可用?

问题回答

1. Redis 主从复制如何实现?

Answer

主从通过 `replicaof` 实现主从关系

第一次同步共有建立连接、同步数据、发送写命令三个阶段

- 建立连接

  从机会向主机发送 `psync` 命令，表示需要进行数据同步

  `psync` 命令包含两个参数：主服务器 `runID` 与复制进度 `offset`

  - `runID`：每个 `redis` 启动时都会随机生成一个 `ID`。由于第一次主从复制时，从机并不知道主机的 `runID`，所以为 `?`
  - `offset`：表示复制的进度。第一次复制时，该值为-1

  主机收到命令后，由于是第一次主从复制，会执行完整同步操作，会返回 `FULLERSYNC` 并携带 `runID` 与 `offset`。

  从服务器收到后，会记录这两个值。

- 同步数据

  主服务器执行 `bgsave` 命令生成 `RDB` 文件，并发送给从服务器

  从服务器收到 `RDB` 文件后，先清空当前数据库，然后载入 `RDB` 文件

  为了保证复制期间主从一致性，主服务器会将复制期间执行的写命令，写入到 `replication buffer` 缓冲区中

- 主服务器发送写命令到从服务器

  从服务器完成 `RDB` 载入后，会发送一个确认消息给主服务器。

  主服务器收到后，便将 `replication buffer` 中的写命令发送给从服务器，从服务器执行，此时达成数据一致

第一次主从复制之后，双方之间会维护一个 `TCP` 连接，主服务器可以将写命令传播给从服务器，此过程为命令传播

2. 从服务器宕机或出现网络故障，如何保证主从一致性?

Answer

Redis 2.8 之前，仍然采用全量复制保证主从一致性，即网络恢复或从服务器恢复后，再次与主从服务器执行全量同步。

Redis 2.8 之后，采用新的增量复制方式

- 从服务器恢复后，发送psync命令给主服务器，并携带之前存储的参数 `runID` 与 `offset`
- 主服务器收到命令后，回复CONTINUE响应告诉从服务器接下来采用增量复制来恢复数据
- 主服务器发送这段时间内的写命令给从服务器

通过一个环形缓冲区与各自的同步偏移量，来确定发送哪些写命令给从服务器

- repl_backlog_buffer：复制积压环形缓冲区
- replication offset：主服务器标记写进度，从服务器标记读进度

主服务器在进行命令传播时，不止会将写命令发送给从服务器，还会存储写命令到 `repl_backlog_buffer` 中

当网络恢复后，从服务器会将自己的复制偏移量 `offset` 发送给主服务器，主服务器根据两个偏移量，确定同步方式

- 若从服务器需要读取的数据还在 `repl_backlog_buffer` 中，采取增量复制
- 否则，采取全量复制

repl_backlog_buffer默认大小为 1MB，当缓冲区写满后，如果继续写入，将会覆盖之前的数据（固定长度的 `FIFO` 队列）

为了网络恢复后，主服务器采取全量复制，需要调整 `repl_backlog_buffer` 的大小

```text
second * write_size_per_second
```

3. 主从模式下，如何判断某个节点是否正常工作?

Answer

- 主节点默认 10s 去 ping 一次从节点，判断节点是否存活
- 从节点默认 1s 发送一次 `replconf ack{offset}` 命令，给主节点上报自身当前的复制偏移量
  - 实时监测主节点网络状态
  - 上报自身复制偏移量，防止出现命令丢失情况

4. 主从复制中，出现的 replication buffer 和 repl_backlog_buffer 有什么区别?

Answer

- replication buffer：主节点会为每一个从节点分配，因为不同从节点连接并开始进行主从复制的时间不同，所以后续发送的写命令也不同，需要单独管理
- repl_backlog_buffer：只用于增量复制阶段，只存在于主节点中

并且当 `replication_buffer` 满了之后，会导致连接断开，并重新连接，重新进行主从复制

5. 主从切换如何减少数据丢失?

Answer

- 异步复制同步丢失

  主节点还没来得及将数据同步给从节点时，主节点宕机，造成数据丢失

  - 通过配置 `min-slaves-max-lag`，一旦所有从节点数据复制和同步延迟超过了该值，主节点会拒绝任何请求。

    即使主节点宕机，损失的数据这只是该值时间内的数据。

  - 客户端发现主节点拒绝请求后，可以采取降级措施，将数据存储在本地缓存或数据库中，等恢复后再重新写入。也可以写入消息队列

- 集群脑裂现象

  当主节点与从节点之间出现网络故障时，无法实现主从命令传播，而 `Sentinel` 节点会判断主节点宕机，重新选举出新的主节点。

  但此时客户端与主节点之间通信正常，还会继续发送写请求。而主节点被降级为从节点后，会清空自身数据，导致客户端写入的数据全部丢失。

  当主节点发现「从节点下线的数量太多」，或者「网络延迟太大」的时候，那么主节点会禁止写操作，直接把错误返回给客户端。

  在 Redis 的配置文件中有两个参数我们可以设置：

  - min-slaves-to-write x，主节点必须要有**至少 x 个从节点连接**，如果小于这个数，主节点会禁止写数据。
  - min-slaves-max-lag x，主从数据复制和同步的延迟**不能超过 x 秒**，如果主从同步的延迟超过 x 秒，主节点会禁止写数据。

6. Sentinel 哨兵节点作用是什么?

Answer

主从节点故障转移。

它会监测主节点是否存活，如果发现主节点挂了，它就会选举一个从节点切换为主节点，并且把新主节点的相关信息通知给从节点和客户端。

7. Sentinel 如何判断主节点是否宕机?

Answer

哨兵会每隔 1 秒给所有主从节点发送 PING 命令，当主从节点收到 PING 命令后，会发送一个响应命令给哨兵，这样就可以判断它们是否在正常运行。

如果主节点或者从节点没有在规定的时间内响应哨兵的 PING 命令，哨兵就会将它们标记为「**主观下线**」。

- 这个「规定的时间」是配置项 `down-after-milliseconds` 参数设定的，单位是毫秒。

为了防止出现误判，会用多个节点组成哨兵集群

当主观下线赞同票数达到哨兵配置文件中的 quorum 配置项设定的值后，这时主节点就会被该哨兵标记为「客观下线」。

- `quorum` 一般配置为 `哨兵个数 / 2 + 1`

8. 由哪个 Sentinel 节点完成最终的故障转移?

Answer

由哨兵集群中的 `leader` 来进行故障转移

`leader` 的选举过程为：

1. 当主观下线赞成票数达到了当前哨兵的 `quorum` 配置值时，当前哨兵就会成为 `leader` 候选者
2. 候选者会向其他哨兵发送命令，表明希望成为 `Leader` 来执行主从切换，并让所有其他哨兵对它进行投票。
    - 每个哨兵只有一次投票机会，如果用完后就不能参与投票了，可以投给自己或投给别人，但是只有候选者才能把票投给自己。
3. 成为 `leader` 的条件为：

```
1.  自身为 `leader` 候选者

2.  拿到半数以上的赞成票

3.  赞成票数大于 `quorum`
```

9. Leader Sentinel 如何选出新的主节点?

Answer

对所有从服务器进行筛选

- 过滤掉所有处于下线状态的从节点
- 过滤掉所有与主服务器连接断开超过 `down-after-milliseconds * 10` 毫秒的从服务器
- 根据优先级、复制进度、ID 号依次进行排序：优先级高、复制偏移量大、ID 号小的节点

之后将选出的节点升级为主节点，让其他从节点更换复制目标，并将旧的主节点变为从节点

0. Sentinel 集群如何组成?

Answer

哨兵节点之间是通过 Redis 的发布者/订阅者机制来相互发现的。

在主从集群中，主节点上有一个名为 `__sentinel__:hello` 的频道，不同哨兵就是通过它来相互发现，实现互相通信的。

哨兵 A 把自己的 IP 地址和端口的信息发布到 `__sentinel__:hello` 频道上，哨兵 B 和 C 订阅了该频道。那么此时，哨兵 B 和 C 就可以从这个频道直接获取哨兵 A 的 IP 地址和端口号。然后，哨兵 B、C 可以和哨兵 A 建立网络连接。

Sentinel默认会以10s一次的频率，向主服务器发送INFO命令，获取其信息，其中包含有主服务器下的所有从服务器信息。

根据此信息，Sentinel与其他从服务器建立命令连接与订阅连接

1. **Redis 如何保证高可用?**

Answer

主从复制、Sentinel集群、Cluster集群

上次编辑于: 2023/6/23 15:31:06
贡献者: ltyzzz

Redis 应用

ltyzzz (https://ltyzzz.com/) 2023年6月23日 约 766 字 大约 3 分钟

问题清单

Questions

1. **Redis 如何实现延时队列?**
2. **Redis 限流器如何实现? 有哪些常见的限流算法?**
3. **如何用 Redis 实现动态 feed 流?**
4. **布隆过滤器原理是什么?**

问题回答

1. Redis 如何实现延时队列?

Answer

采用 `String` 与 `SortedSet` 数据结构实现

对于 `Push` 操作

- 采用 `String`：生成随机 `UUID` 作为 `key`，具体数据作为 `value`
- 采用 `SortedSet`：将 `key` 添加到 `SortedSet` 中，`score` 为当前时间戳

对于 `Pop` 操作

- 计算 `previous` 秒之前的时间戳，使用 `SortedSet` 的 `zrangebyscore` 方法获取这个时间之前的所有 `key` 值
- 利用 `Redis` 删除的原子性，删掉 `SortedSet` 中的 `key`
- 通过这些 `key`，定位到 `String` 中的 `value` 值，得到真正的数据

2. Redis 限流器如何实现？有哪些常见的限流算法？

Answer

Redis + lua 脚本实现令牌桶限流器

1. 用户发起请求，经过网关或拦截器或AOP前置方法
2. Redis 生成令牌：`(当前时间 - 最后一次生成令牌的时间) / 1000 * 每秒生成令牌数量`
3. 放入桶中：若生成令牌数大于 0，则更新最后一次生成令牌时间；生成令牌数加剩余令牌数大于桶容量，取桶容量
4. 取出令牌：桶中剩余令牌小于所取令牌数，则全部取出；否则只取固定数量
5. 获取到令牌则放行请求

漏桶算法、固定窗口算法、计数器算法

3. 如何用 Redis 实现动态 feed 流？

Answer

1. 当用户 `A` 发布视频时，写入 `Redis` 该用户对应的 `sendbox` 中，将该条消息封装，发送给消息队列
2. 消息队列根据 `A` 用户的粉丝列表，并判断当前粉丝是否为活跃用户

    - 若是，则将此消息添加到粉丝的 `inbox` 中
    - 若不是，则不做处理

3. 用户登陆时，首先从 `inbox` 中获取动态，若判断用户不是活跃用户，则遍历其关注列表的 `sendbox`，添加到 `inbox`
4. `feed` 数据清理：定时任务清理大于 300 条以上的最早的数据

4. 布隆过滤器原理是什么？

Answer

布隆过滤器主要是由位数组和 `hash` 函数构成的

- 位数组初始状态都为 0
- 哈希函数用于对 `key` 进行计算，并得到该 `key` 对应于位数组的位置
- 判断元素是否存在
  1. 先通过多个哈希函数得到当前元素的多个哈希值，并进行取余，得到多个位数组的位置
  2. 判断多个位置上是否都为 1
     - 若其中有一个为 0，则说明元素不存在
     - 若都为 1，则说明元素有可能存在

应用场景：

- 解决缓存穿透问题：先查看布隆过滤器，再查看 `Redis`，最后查看 `MySQL`
- 判断用户是否看过某帖子、某视频、某网站
- 爬虫系统对已爬取的页面去重

上次编辑于: 2023/6/23 15:31:06
贡献者: ltyzzz