Redis

> 为什么要用NoSQL?

用户个人信息，社交网络，地理位置，用户产生的数据，用户日志爆发式增长。

> NoSQL是什么?

非关系型数据库。复杂数据类型（个人信息，社交网络，地理位置）的存储不需要固定的格式，不需要多余操作就可以横向扩展。

1. 方便扩展（数据之间没有关系，很好扩展）
2. 大数据量高性能（Redis一秒写8万次，读取11万，NoSQL的缓存记录级，是一种细粒度的缓存，性能较高）
3. 数据类型多样（不需要实现设计数据库，随取随用）
4. 传统RDBMS和NoSQL：

   传统的RDBMS：

   - 结构化组织
   - SQL
   - 数据和关系都存在单独的表中
   - 操作语言，数据定义语言
   - 严格的一致性
   - 基础事务操作

   NoSQL：

   - 不仅仅是数据
   - 没有固定的查询语言
   - 键值对存储，列存储，文档存储，图形数据库
   - 最终一致性
   - CAP定理和BASE（异地多活）
   - 高性能，高可用，高可扩展

NoSQL四大分类

KV键值对

Redis

文档型数据库（bson格式和json格式一样）

MangoDB（基于分布式文件存储的数据库，主要用来处理大量的文档）

MangoDB是一个介于关系型和非关系型数据库中中间的产品。MangoDB是非关系型数据库中功能最丰富，最像关系型数据库的。

列存储数据库

HBase

分布式文件系统

图关系数据库

放的是关系（朋友圈社交网络，广告推荐）

Neo4j

Redis是单线程

Redis是基于内存操作，CPU不是Redis性能瓶颈。Redis瓶颈是机器的内存和网络带宽，既然可以使用单线程实现，便使用单线程。

为什么单线程这么快

- Redis将所有数据全部放在内存中，使用单线程取操作效率就是最高的，多线程（CPU上下文切换，会耗时）。对于内存系统来说，如果没有上下文切换效率就是最高的。

实现

字符串

字符串底层为SDS，主要是为了解决C中char*的缺点：

- 追加操作和长度计算操作
- 二进制安全

类内有三个成员变量：

1. int len：buf已占用长度
2. int free：buf剩余可用长度
3. char buf[]：实际保存字符串数据的地方

当大小小于 `1MB` 的字符串执行追加操作时， `sdsMakeRoomFor` 就为它们分配多于所需大小一倍的空间； 当字符串的大小大于 `1MB` ， 那么 `sdsMakeRoomFor` 就为它们额外多分配 `1MB` 的空间。**会浪费内存吗?** 执行过APPEND命令的字符串会带有额外的预分配空间，这些预分配空间不会被释放， 除非该字符串所对应的键被删除， 或者等到关闭 Redis 之后， 再次启动时重新载入的字符串对象将不会有预分配空间。通常这种字符串数量不多，占用体积不大。真的太大的话可能就需要修改 Redis 服务器， 让它定时释放一些字符串键的预分配空间， 从而更有效地使用内存。

Lists

Lists类型是使用双向链表实现的，在两端插入或删除操作较快，读取中间节点数据慢

Zset

ziplist：满足以下两个条件的时候

- 元素数量少于128的时候
- 每个元素的长度小于64字节

skiplist：不满足上述两个条件就会使用跳表，具体来说是组合了map和skiplist

- map用来存储member到score的映射，这样就可以在O(1)时间内找到member对应的分数
- skiplist按从小到大的顺序存储分数
- skiplist每个元素的值都是[score,value]对

zset使用散列表和跳跃表（skip list）实现，即使读取位于中间的数据速度也很快（ O(logN) ）

事务

错误处理

1. 语法错误

   命令不存在或命令参数个数不对。只要一个命令有语法错误，执行EXEC命令后Redis就会直接返回错误，连语法正确的命令也不会执行。

2. 运行错误

   指在命令执行时出现的错误，比如使用散列类型的命令操作集合类型的键，这种错误在实际执行之前Redis是无法发现的。如果事务里的一条命令出现了运行错误，事务里其他的命令依然会继续执行（包括出错命令之后的命令）

redis的事务没有提供rollback功能，开发者必须在事务执行出错后将数据库傅园慧事务执行前的状态。

WATCH

WATCH命令可以监控一个或多个键，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。

由于WATCH命令的作用只是当被监控的键值被修改后阻止之后一个事务的执行，不能保证其他客户端不修改这一键值。

使用UNWATCH取消，避免影响下一个事务的执行。

过期时间

使用EXPIRE命令设置一个键的过期时间，到时间后Redis会自动删除它。

使用TTL查看过期时间

使用PERSIST，SET或GETSET清除键的过期时间

淘汰键规则

表 4-1　Redis 支持的淘汰键的规则

规　则	说　明
volatile-lru	使用 LRU 算法删除一个键（只对设置了过期时间的键）
allkeys-lru	使用 LRU 算法删除一个键
volatile-random	随机删除一个键（只对设置了过期时间的键）
allkeys-random	随机删除一个键
volatile-ttl	删除过期时间最近的一个键
noeviction	不删除键，只返回错误

SORT命令

- BY参数，GET参数，STORE参数
- SORT tag:ruby:posts BY post:->*time DESC GET post:*->*title GET post:*->time GET # STORE sort.result
- 性能优化：

  时间复杂度O(n+mlog(m))，n表示要排序的列表中的元素个数，m表示要返回的元素个数。Redis在排序前会建立一个长度为n的容器来存储排序的元素。

发布/订阅模式

发布者发布消息的命令是PUBLISH，用法是PUBLISH channel message，发出去的消息不会被持久化。

订阅频道的命令是SUBSCRIBE，用法是SUBSCRIBE channel [channel ...]。执行SUBSCRIBE命令后客户端会进入订阅状态，此状态下客户端不能使用除SUBSCRIBE UNSUBSRIBE PSUBSCRIBE PUNSUBSCRIBE之外的命令。

进入订阅状态后客户端可能收到三种类型的回复。每种类型的回复都包含三个值，第一个值是消息的类型，根据消息类型的不同，第二、三个值的含义也不同。

1. subscribe 第二个值是订阅成功的频道名称，第三个值是当前客户端订阅的频道数量
2. message 最关心的类型。第二个值表示产生消息的频道名称，第三个值是消息的内容。
3. unsubscribe 第二个值是对应的频道名称，第三个值是当前客户端订阅的频道数量，为0时退出订阅状态，之后可以执行其他非“发布/订阅”模式的命令

管道

客户端和Redis使用TCP协议连接。Redis的底层通信协议对管道（pipelining）提供了支持。通过管道可以一次性发送多条命令并在执行完后一次性将结果返回，当一组命令中每条命令都不依赖于之前命令的执行结果时就可以将这组命令一起通过管道发出。

持久化

RDB

持久化通过快照完成

- 根据配置规则进行自动快照
- 用户执行SAVE或BGSAVE命令
- 执行FLUSHALL命令
- 执行复制（replication）

1. **根据配置规则进行自动快照**

   时间窗口M内更改的键个数大于N时，符合自动快照条件

   save 900 1 在900秒内有一个或以上的键被更改进行快照

   save 300 10 在300秒内有至少10个上的键被更改进行快照

   save 60 10000

2. **用户执行SAVE或GBSAVE命令**

   执行SAVE命令时，Redis同步进行快照操作，执行过程中会堵塞所有来自客户端的请求。尽量避免在生产环境使用这一命令。

   BGSAVE可以在后台异步进行快照操作，同时服务器还能继续响应来自客户端的请求。

3. **执行FLUSHALL命令**

   执行FLUSHALL命令时，Redis会清除数据库中所有数据。无论清空数据库的操作是否触发了自动快照条件，只要自动快照条件不为空，Redis就会执行一次快照操作。如果没有定义自动快照条件时，执行FLUSHALL不会进行快照

4. **执行复制时**

   当设置了主从模式时，Redis会在复制初始化时自动进行快照。即使没有定义自动快照条件，而且没有手动执行过快照操作，也会生成RDB文件

快照原理

1. Redis使用fork函数复制一份当前进程（父进程）的副本（子进程）
2. 父进程继续接收并处理客户端发来的命令，子进程开始将内存中的数据写入硬盘中的临时文件
3. 当子进程写入完所有数据后会用该临时文件替换旧的RDB文件，至此一次快照操作完成

AOF

使用Redis存储非临时数据时，一般需要打开AOF持久化来降低进程中止导致的数据丢失。AOF可以将Redis执行的每一条命令追加到硬盘文件中。默认不开启。

实现

AOF以纯文本的方式记录了Redis执行的写命令。AOF文件的内容正是Redis客户端向Redis发送的原始通信协议的内容。

每当达到一定条件Redis就会自动重写AOF文件：

- auto-aof-rewrite-percentage 当目前的AOF文件大小超过上一次重写时的AOF文件大小的百分之多少时会再次重写
- auto-aof-rewrite-min-size 限制了允许重写的最小的AOF文件大小

同步硬盘数据

虽然每次执行更改数据库内容的操作时，AOF都会将命令记录在AOF文件中，但是事实上由于操作系统的缓存机制，数据并没有真正写入硬盘，而是进入了系统的硬盘缓存，默认情况下系统每30秒执行一次同步操作将内容真正写入硬盘。在这30秒钟内如果系统异常退出会导致硬盘缓存中数据库丢失。在Redis中可以设置appendfsync参数设置同步时机：

- 默认everysec，每秒执行一次同步操作
- always 每次执行写入都会执行同步，最安全也最慢
- no 不需要主动进行同步，交由系统来做（30秒一次），最快但最不安全

集群

复制

一个数据库启动后，会向主数据库发送SYNC命令。主数据库接到SYNC命令后会开始在后台保存快照（即RDB持久化过程），并将保存快照期间收到的命令缓存起来。快照完成后，会将快照文件和所有缓存的命令发送给从数据库。从数据库收到后，载入快照文件并执行收到的缓存的命令。以上过程称为复制初始化。复制初始化结束后，主数据库每收到写命令时就会将命令同步给从数据库。

主从数据库连接断开重连后，Redis2.6及以前需要重新进行复制初始化。2.8开始断线重连能够支持有条件的增量数据传输。主数据库只需要将断线期间执行的命令传送给从数据库。

同步时从数据库不会堵塞，可以继续处理请求。默认情况下从数据库会用同步前的数据对命令进行响应。

乐观复制

Redis采用了乐观复制的复制策略，容忍一定时间内主从数据库内容是不同的，但两者的数据会最终同步。具体来说，Redis在主从数据库之间复制数据的过程本身是异步的。主数据库不会等待从数据库接受该命令后再返回给客户端。

不过Redis提供了两个配置选项来限制只有当数据至少同步给指定数量的从数据库时，主数据才是可写的：

min-slaves-to-write 3

min-slaves-max-lag 10

第一行表示只有当3或3个以上从数据库连接到主数据库时，主数据库才是可写的。第二行表示从数据库最长失去连接的时间。如果从数据库最后与主数据库联系的时间小于这个值，则认为从数据库还在保持与主数据库的连接

从数据库持久化

为了提高性能，可以通过复制功能建立一个（或若干个）从数据库，并在从数据库中启用持久化，同时在主数据库中禁用持久化

- 从数据库崩溃重启后，主数据库会自动将数据同步过来
- 若主数据库崩溃：
    1. 在从数据库中使用SLAVEOF NO ONE命令将数据库提升成主数据库继续服务
    2. 启动之前崩溃的主数据库，使用SLAVEOF命令将其设置成新的从数据库，即可将数据同步回来

增量复制

1. 从数据库会存储主数据库的运行ID，每个Redis实例均会拥有一个唯一的运行ID
2. 复制同步阶段，主数据库每将一个命令传送给从数据库时，都会把该命令存放到一个积压队列（backlog）中并记录下当前积压队列中存放命令的偏移量范围
3. 从数据库接收到主数据库传来的命令时，会记录下该命令的偏移量。

2.8版后，从数据库连接时不再发送SYNC，而是PSYNC，格式为 “PSYNC 主数据库运行ID 断开前最新的命令偏移量”

主数据库操作：

1. 判断从数据库传送过来的运行ID是否和自己的运行ID相同，确保从数据库之前确实是和自己同步的
2. 判断从数据库最后同步成功的命令偏移量是否在积压队列中。若存在则可以进行增量复制，并将积压队列中相应的命令发送给从数据库。若此次重连不满足增量复制的条件，主数据库会进行一次全部同步

哨兵

1. 监控主数据库和从数据库是否正常运行
2. 主数据库出现故障时自动将从数据库转换为主数据库

哨兵是一个独立的进程

原理

哨兵进程启动时会读取配置文件的内容，找到需要监控的主数据库，同时发现其从数据库。烧饼启动后，会与要监控的主数据库建立两条连接，一条用来订阅该主数据库的_sentinel_:hello频道以获取其他同样监控该数据库的哨兵节点的信息。另外哨兵也需要定期向主数据库发送INFO等命令来获取主数据库本身的信息。

和主数据库连接建立后：

1. 每10秒哨兵会向主数据库和从数据库发送INFO命令
2. 每2秒哨兵会向主数据库和从数据库的_sentinel_:hello频道发送自己的信息
3. 每1秒哨兵会向主数据库、从数据库和其他哨兵节点发送PING命令

- INFO命令使得哨兵可以获得当前数据库的相关信息，从而实现新节点的自动发现。
- 哨兵向主从数据库的_sentinel_:hello频道发送信息来与同样监控该数据库的哨兵分享自己的信息，可以看到哨兵的基本信息以及其监控的主数据库的信息。其他哨兵会判断是不是新发现的哨兵，如果是则将其加入已发现的哨兵列表并创建一个到齐的连接。同时哨兵如果发现信息中的主数据库版本比自己的高，会更新主数据库的数据。
- 每隔一段时间发送PING命令来定时监控数据库和节点有没有停止服务。超过指定时间仍未回复，则哨兵认为其**主观下线**，开始询问其他哨兵节点以了解他们是否也认为该主数据库主观下线，达到指定数量时，哨兵认为其**客观下线**，并选举领头的哨兵节点对主从系统发起故障恢复

选举领头哨兵的过程使用了Raft算法。

1. 发现主数据库客观下线的哨兵节点（A）向每个哨兵节点发送命令，要求对方选自己称为领头哨兵。
2. 如果目标哨兵节点没有选过其他人，则会同意A称为领头哨兵
3. 如果A发现超过半数且超过qurorum参数值的哨兵节点同意，则A成功成为领头哨兵
4. 当有多个哨兵节点同时参选领头哨兵，会出现没有任何节点当选的可能。此时每个参选节点将等待一个随机时间重新发起参选请求进行下一轮选举，直到选举成功。

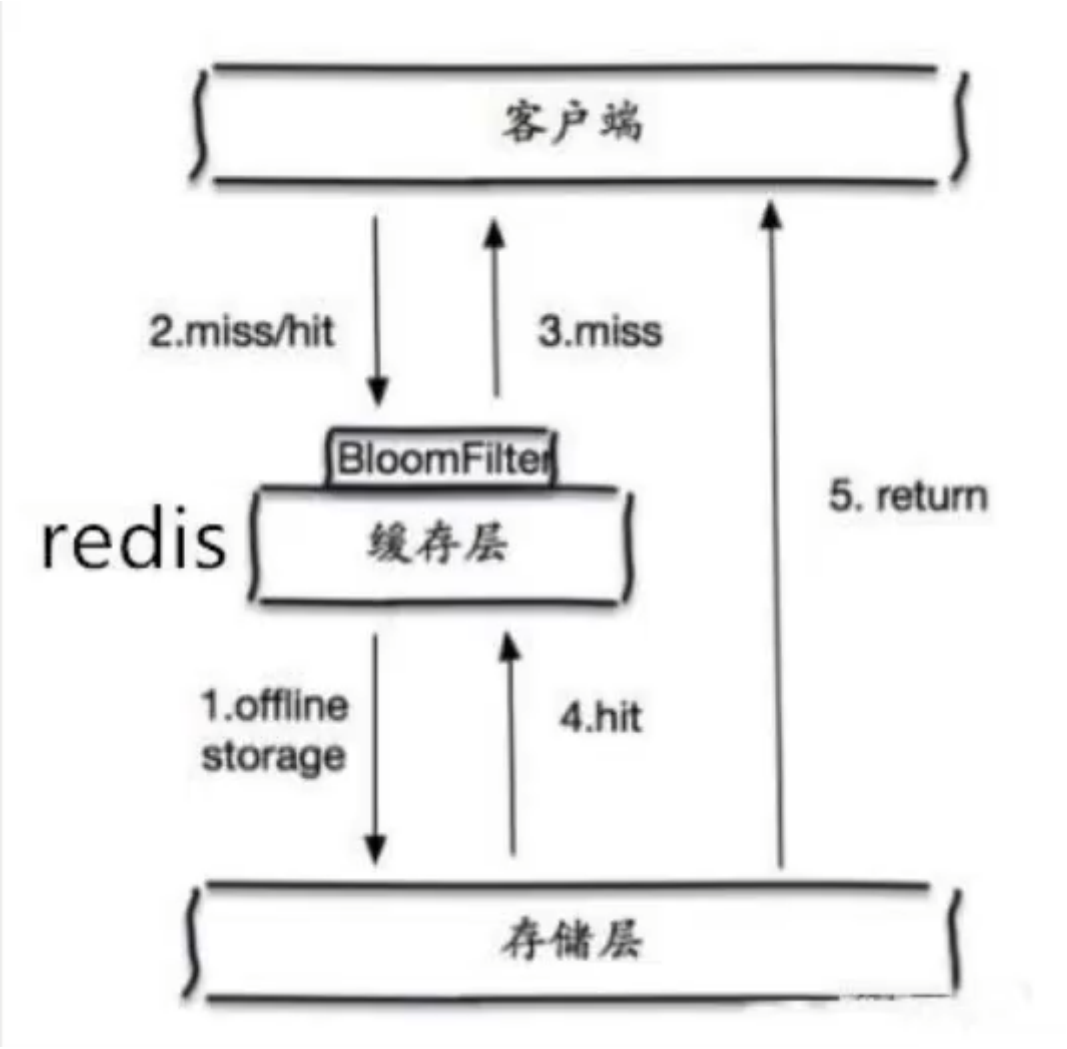
缓存穿透和雪崩

缓存穿透（查不到）

用户查询一个数据，缓存没有命中，向持久层数据库查询也没有，于是本次查询失败。用户很多的时候，会给持久层造成很大的压力。

解决：布隆过滤器

一种数据结构，对所有可能查询的参数以hash形式存储，在控制层进行校验，不符合则丢弃，从而避免了对底层系统的查询压力。



空值缓存

有两个问题：

1. 需要更多空间存储更多的键，因为当中可能有很多的空值的键
2. 即使对空值设置了过期时间，还是会存在缓存层和储存层数据会有一段时间窗口的不一致，这对于需要保持一致性的业务会有影响

缓存击穿（量太大，缓存过期）

需要注意和缓存穿透的区别：缓存击穿是指一个key非常热点，在不停地扛着大并发，大并发集中在这一个点进行访问，当这个key失效的瞬间，持续的大并发就穿破缓存，直接请求数据库。

当某个key在过期的瞬间，有大量的请求并发访问，这类数据一般是热点数据，由于缓存过期，会同时访问数据库来查询最新数据，并且回写缓存，会导致数据库瞬间压力过大。

解决：

- 设置热点数据永不过期
- 加互斥锁

  分布式锁：保证每个key同时只有一个线程去查询后端服务，其他线程只需要等待。这种方式将高并发压力转移到了分布式锁。

缓存雪崩

在某一个时间段，缓存集中过期失效（或Redis宕机）

比如双十二抢购前，商品时间比较集中地放入了缓存，过了一段时间这批商品缓存都过期了，对这批商品地访问查询都落到了数据库上，对于数据库而言，会产生周期性的压力波峰。所有的请求都会到达存储层，存储层调用量暴增，造成存储层也会挂掉。比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。

解决：

- Redis高可用：多增设几台Redis，一台挂掉后其他的还可以继续工作。
- 限流降级：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量，比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。
- 数据预热：在正式部署之前，先把可能的数据预先访问一边。这样部分可能大量访问的数据就会加载到缓存中，在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

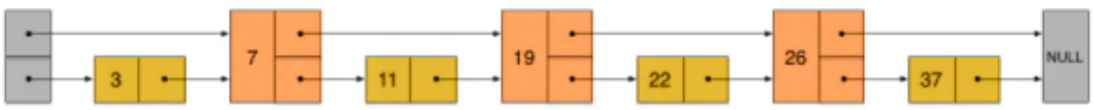
常见问题

> zset是怎么实现的?

- ziplist：满足以下两个条件的时候
- - 元素数量少于128的时候
  - 每个元素的长度小于64字节
- skiplist：不满足上述两个条件就会使用跳表，具体来说是组合了map和skiplist
- - map用来存储member到score的映射，这样就可以在O(1)时间内找到member对应的分数
  - skiplist按从小到大的顺序存储分数
  - skiplist每个元素的值都是[score,value]对
- **为什么不直接用跳跃表**
- 假如我们单独使用 字典，虽然能以 O(1) 的时间复杂度查找成员的分值，但是因为字典是以无序的方式来保存集合元素，所以每次进行范围操作的时候都要进行排序；假如我们单独使用跳跃表来实现，虽然能执行范围操作，但是查找操作有 O(1)的复杂度变为了O(logN)。因此Redis使用了两种数据结构来共同实现有序集合。

1. 压缩列表（ziplist）是Redis为了节省内存而开发的，是由一系列特殊编码的**连续内存块组成的顺序型数据结构**，一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。
2. 因为有了skiplist，才1能在O(logN)的时间内插入一个元素，并且实现快速的按分数范围查找元素

跳表：



总结一下跳表原理：

- 每个跳表都必须设定一个最大的连接层数MaxLevel
- 第一层连接会连接到表中的每个元素

- 插入一个元素会随机生成一个连接层数值[1, MaxLevel]之间，根据这个值跳表会给这元素建立N个连接
- 插入某个元素的时候先从最高层开始，当跳到比目标值大的元素后，回退到上一个元素，用该元素的下一层连接进行遍历，周而复始直到第一层连接，最终在第一层连接中找到合适的位置

> redis为什么这么快

- redis是纯内存操作：数据存放在内存中，内存的响应时间大约是100纳秒，这是Redis每秒万亿级别访问的重要基础。
- 非阻塞I/O：Redis采用epoll作为I/O多路复用技术的实现，再加上Redis自身的事件处理模型将epoll中的连接，读写，关闭都转换为了时间，不在I/O上浪费过多的时间。
- 单线程避免了线程切换和竞态产生的消耗。