Redis

- 消息模型
  - 1、点对点
    - 消息生产者向消息队列中发送了一个消息之后，只能被一个消费者消费一次。
  - 2、发布/订阅
    - 消息生产者向频道发送一个消息之后，多个消费者可以从该频道订阅到这条消息并消费。
  - 发布与订阅模式和观察者模式有以下不同：
    - 观察者模式中，观察者和主题都知道对方的存在；而在发布与订阅模式中，生产者与消费者不知道对方的存在，它们之间通过频道进行通信。
    - 观察者模式是同步的，当事件触发时，主题会调用观察者的方法，然后等待方法返回；而发布与订阅模式是异步的，生产者向频道发送一个消息之后，就不需要关心消费者何时去订阅这个消息，可以立即返回。

观察者模式定义了对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知，并自动更新。观察者模式属于行为型模式，行为型模式关注的是对象之间的通讯，观察者模式就是观察者和被观察者之间的通讯

- 项目中缓存是如何使用的？为什么要用缓存？
  - 因为传统的关系型数据库如Mysql已经不能适用所有的场景了，比如秒杀的库存扣减，APP首页的访问流量高峰等等，都很容易把数据库打崩，所以引入了缓存中间件，目前市面上比较常用的缓存中间件有Redis和Memcached不过中和考虑了他们的优缺点，最后选择了Redis。
  - 常见问题
    - 缓存和数据库双写不一致
    - 缓存雪崩、缓存穿透
    - 缓存并发竞争

多个用户同时更新一个key

- redis和memcached有什么区别
  - redis支持复杂的数据结构
  - redis原生支持集群模式（redis3.x开始支持cluster模式），memcached不支持
  - redis只能使用单线程，memcached多线程，100k以上的数据中m的性能要比redis高
  - Memcached
    - MC 处理请求时使用多线程异步 IO 的方式，可以合理利用 CPU 多核的优势，性能非常优秀；
    - MC 功能简单，使用内存存储数据；
    - MC 对缓存的数据可以设置失效期，过期后的数据会被清除；
    - 失效的策略采用延迟失效，就是当再次使用数据时检查是否失效；
    - 当容量存满时，会对缓存中的数据进行剔除，剔除时除了会对过期 key 进行清理，还会按 LRU 策略对数据进行剔除。

- Mencached的问题
  - key 不能超过 250 个字节；
  - value 不能超过 1M 字节；
  - key 的最大失效时间是 30 天；
  - 只支持 K-V 结构，不提供持久化和主从同步功能
- redis线程模型
  - Redis基于Reactor模式开发了自己的网络事件处理器： 这个处理器被称为**文件事件处理器**（file event handler）
  - 组成结构为4部分
    - 多个套接字
    - IO多路复用程序
    - 事件分派器
    - 事件处理器（命令请求处理器、命令回复处理器、连接应答处理器）
    - 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。
  - 消息处理流程
    - 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
    - 尽管多个文件事件可能会并发地出现，但I/O多路复用程序总是会将所有产生事件的套接字都推到一个队列里面，然后通过这个队列，以有序（sequentially）、同步（synchronously）、每次一个套接字的方式向文件事件分派器传送套接字
    - 当上一个套接字产生的事件被处理完毕之后（该套接字为事件所关联的事件处理器执行完毕）， I/O多路复用程序才会继续向文件事件分派器传送下一个套接字。
  - 文件事件的类型
    - AE_READABLE事件
    - AE_WRITABLE事件
  - 文件事件处理器
    - Redis为文件事件编写了多个处理器，这些事件处理器分别用于实现不同的网络通讯需求，常用的处理器如下：
    - 为了对连接服务器的各个客户端进行应答， 服务器要为监听套接字关联**连接应答处理器**。
    - 为了接收客户端传来的命令请求， 服务器要为客户端套接字关联**命令请求处理器**。
    - 为了向客户端返回命令的执行结果， 服务器要为客户端套接字关联**命令回复处理器**。
- 为啥 redis 单线程模型也能效率这么高

- 纯内存操作
- 非阻塞同步多路复用 epoll
- 单线程避免了上下文切换的性能损耗

- Redis 数据类型和使用场景
  - // dbsize在计算键的总数时直接获取，O(1)；keys命令会遍历所有键，所以它的时间复杂度是O(n)，一般不使用
  - // setnx可以作为分布式锁的一种方法
  - 内部编码
    - string: int（8个字节的长整型）、embstr（小于等于44个字节的字符串）、raw（大于44个字节的字符串）
      - 如果字符串对象保存的是一个字符串值， 并且这个字符串值的长度大于 39 字节， 那么字符串对象将使用一个简单动态字符串（SDS）来保存这个字符串值，并将对象的编码设置为 raw 。
      - 如果字符串对象保存的是一个字符串值， 并且这个字符串值的长度小于等于 39 字节， 那么字符串对象将使用 embstr 编码的方式来保存这个字符串值。
      - embstr 编码是专门用于保存短字符串的一种优化编码方式， 这种编码和 raw 编码一样， 都使用 redisObject 结构和 sdshdr 结构来表示字符串对象， 但 raw 编码会调用两次内存分配函数来分别创建 redisObject 结构和 sdshdr 结构， 而 embstr 编码则通过调用一次内存分配函数来分配一块连续的空间， 空间中依次包含 redisObject 和 sdshdr 两个结构， 如图 8-3 所示。

        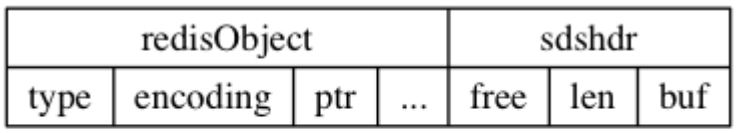

        图 8-3　embstr 编码创建的内存块结构

      - embstr 编码的字符串对象在执行命令时， 产生的效果和 raw 编码的字符串对象执行命令时产生的效果是相同的， 但使用 embstr 编码的字符串对象来保存短字符串值有以下好处：
      - embstr 编码将创建字符串对象所需的内存分配次数从 raw 编码的两次降低为一次。
      - 释放 embstr 编码的字符串对象只需要调用一次内存释放函数， 而释放 raw 编码的字符串对象需要调用两次内存释放函数。
      - 因为 embstr 编码的字符串对象的所有数据都保存在一块连续的内存里面， 所以这种编码的字符串对象比起 raw 编码的字符串对象能够更好地利用缓存带来的优势。
    - hash: hashtable、ziplist（小于512元素时，单个数据小于64字节）
    - list: linkedlist、ziplist（小于512个元素时）
    - set: hashtable、intset（小于512个元素，数据都是整数）

- zset: skiplist、ziplist（小于128个时，数据的大小都要小于64字节）
- Redis这样设计有两个好处：第一，可以改进内部编码，而对外的数据结构和命令没有影响；第二，多种内部编码实现可以在不同场景下发挥各自的优势，Redis会根据配置选项转换不同的类型
- ziplist
  - 压缩列表。它并不是基础数据结构，而是Redis自己设计的一种数据存储结构。它有点儿类似数组，通过**一片连续的内存空间，来存储数据**。不过，它跟数组不同的一点是，它**允许存储的数据大小不同。**
  - 之所以说这种存储结构节省内存，是相较于数组的存储思路而言的。数组要求每个元素的大小相同，要存储不同长度的字符串，就需要用最大长度的字符串大小作为元素的大小。当我们存储小于2最长字节长度的字符串的时候，便会浪费部分存储空间。
  - 压缩列表这种存储结构，一**方面比较节省内存，另一方面可以支持不同类型数据的存储**。而且，因为数据存储在一片连续的内存空间，通过键来获取值为列表类型的数据，读取的效率也非常高。

- string
  - 字符串类型是Redis最基础的数据结构。首先键都是字符串类型，而且其他几种数据结构都是在字符串类型基础上构建的
  - 使用场景
  - 1.缓存功能2.计数器 3.分布式锁
- hash
  - 使用场景：缓存对象信息、用户信息
- list
  - 列表是一种比较灵活的数据结构，它可以充当栈和队列的角色，在实际开发上有很多应用场景。
  - 使用场景：
  - 1.阻塞消息队列（lpush+brpop）
  - 2.文章列表（列表不但是有序的，同时支持按照索引范围获取元素，基于lrange实现分页查询）
- set
  - 集合中不允许有重复元素，并且集合中的元素是无序的，不能通过索引下标获取元素。一个集合最多可以存储232 -1个元素。
  - Redis除了支持集合内的增删改查，同时还支持多个集合取交集、并集、差集。集合间的运算在元素较多的情况下会比较耗时，所以Redis提供了上面三个命令（原命令+store）将集合间交集sinter、并集sunion、差集sdiff的结果保存在destination key中
  - 使用场景：集合类型比较典型的使用场景是标签（tag）通过集合间运算
- sorted set
  - 它给每个元素设置一个分数（score）作为排序的依据。

    - 使用场景：排行榜功能
- Redis 的过期策略都有哪些？内存淘汰策略有哪些？
    - redis过期策略：**定性删除+惰性删除**
    - 定期删除，指的是 redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 key，检查其是否过期，如果过期就删除。
    - 惰性删除，是指在获取某个 key 的时候，redis 会检查一下，这个 key 如果设置了过期时间那么是否过期了？如果过期了此时就会删除，不会给你返回任何东西。
    - 大量过期 key 堆积在内存里，导致 redis 内存块耗尽了：走**内存淘汰策略**
    - **内存淘汰策略**
    - noeviction: 当内存不足以容纳新写入数据时，新写入操作会报错，这个一般没人用吧，实在是太恶心了。
    - **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，这个一般没人用吧，为啥要随机，肯定是把最近最少使用的 key 给干掉啊。
    - 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（这个一般不太合适）。
    - 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。
    - **手写一个lru代码**
- list做异步队列
    - 一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。
    - 如果不用sleep，list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。
    - 如果想要生产一次消费多次，可以使用pub/sub主题订阅者模式，可以实现1:N的消息队列，但在消费者下线后，生产的消息会丢失，想要持久化的话，需要使用消息队列如rabbitmq等。
- 如何保证redis高并发（主从架构）
    - redis高并发的瓶颈：单机（qps上万到几万不等）
    - 水平扩容支撑读高并发
    - **读写分离/主从（master-slave）**：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。
    - 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；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。

- 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。
- **过期KEY处理**
- slave 不会过期 key，只会等待 master 过期 key。如果 master 过期了一个 key，或者通过 LRU 淘汰了一个 key，那么会模拟一条 del 命令发送给 slave。

- 主从复制的步骤
  - **复制过程**
    - slave node 启动时，会在自己本地保存 master node 的信息，包括 master node 的host 和ip，但是复制流程没开始
    - slave node 内部有个定时任务，每秒检查是否有新的 master node 要连接和复制，如果发现，就跟 master node 建立 socket 网络连接，
    - 然后 slave node 发送 ping 命令给 master node/会发送一个 PSYNC 命令给 master node。如果 master 设置了 requirepass，那么 slave node 必须发送 masterauth 的口令过去进行认证。
    - master node 执行**全量复制/增量复制**，将数据发给 slave node。
    - 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。
    - 在后续，master node 持续将写命令，异步复制给 slave node。
  - **全量复制**
    - master 执行 bgsave ，在本地生成一份 rdb 快照文件。
    - master node 将 rdb 快照文件发送给 slave node，如果 rdb 复制时间超过 60秒（repl-timeout），那么 slave node 就会认为复制失败，可以适当调大这个参数(对于千兆网卡的机器，一般每秒传输 100MB，6G 文件，很可能超过 60s)
    - master node 在生成 rdb 时，会将所有新的写命令缓存在内存中，在 slave node 保存了 rdb 之后，再将新的写命令复制给 slave node。
    - 如果在复制期间，内存缓冲区持续消耗超过 64MB，或者一次性超过 256MB，那么停止复制，复制失败。
    - slave node 接收到 rdb 之后，清空自己的旧数据，然后重新加载 rdb 到自己的内存中，同时基于旧的数据版本对外提供服务。
  - **增量复制**/断点续传
    - 如果全量复制过程中，master-slave 网络连接断掉，那么 slave 重新连接 master 时，会触发增量复制。
    - master 直接从自己的 backlog 中获取部分丢失的数据，发送给 slave node，默认 backlog 就是 1MB。

    - master 就是根据 slave 发送的 psync 中的 offset 来从 backlog 中获取数据的。
    - 从 redis2.8 开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始复制一份。master node 会在内存中维护一个 **backlog（默认1MB）**，master 和 slave 都会保存一个 **replica offset** 还有一个 master run id，offset 就是保存在 backlog 中的。如果 master 和 slave 网络连接断掉了，slave 会让 master 从上次 replica offset 开始继续复制，如果没有找到对应的 offset，那么就会执行一次 **resynchronization全量复制**。
  - **异步复制**
    - master 每次接收到写命令之后，先在内部写入数据，然后异步发送给 slave node。
  - **heartbeat**
    - 主从节点互相都会发送 heartbeat 信息。
    - master 默认每隔 10秒 发送一次 heartbeat，slave node 每隔 1秒 发送一个 heartbeat。
- redis高可用（哨兵/redis cluster切换）
  - **哨兵**
    - 哨兵至少需要3个实例来保证自己的健壮性，哨兵本身也是分布式；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。
    - quorum为判断master宕机的数量，majority为选举master故障转移的数量（即超过一半），sdown 是主观宕机，就一个哨兵如果自己觉得一个 master 宕机了，那么就是主观宕机；odown 是客观宕机，如果 quorum 数量的哨兵都觉得一个 master 宕机了，那么就是客观宕机
    - 哨兵会负责自动纠正 slave 的一些配置，比如 slave 如果要成为潜在的 master 候选人，哨兵会确保 slave 复制现有 master 的数据；如果 slave 连接到了一个错误的 master 上，比如故障转移之后，那么哨兵会确保它们连接到正确的 master 上。
    - 功能
      - **故障转移/主备切换**：如果 master node 挂掉了，会自动转移到 slave node 上。
      - 集群监控：负责监控 redis master 和 slave 进程是否正常工作。
      - 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。
      - 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。
  - **哨兵主备切换的数据丢失问题**
    - 异步复制导致的数据丢失：因为 master->slave 的复制是异步的，所以可能有部分数据还没复制到 slave，master 就宕机了，此时这部分数据就丢失了。
    - 脑裂导致的数据丢失：master短暂脱离了正常的网络，重新选举了新的master，而此时client还在往旧的master发数据，旧的master恢复后成为了slave，原有数据丢失。
    - 解决：min-slaves-to-write 1min-slaves-max-lag 10
    - 表示，要求至少有 1 个 slave，数据复制和同步的延迟不能超过 10 秒。
  - **主从架构slave->master选举算法**

- 如果一个 master 被认为 odown 了，而且 majority 数量的哨兵都允许主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个 slave 来，会考虑 slave 的一些信息：
- 跟 master 断开连接的时长、slave 优先级、复制 offset、run id
- 接下来会对 slave 进行排序：
- 1.按照 slave 优先级进行排序，slave priority 越低，优先级就越高。
- 2.如果 slave priority 相同，那么看 replica offset，哪个 slave 复制了越多的数据，offset 越靠后，优先级就越高。
- 3.如果上面两个条件都相同，那么选择一个 run id 比较小的那个 slave。

- **redis cluster的高可用与主备切换原理**
  - redis cluster 的高可用的原理，几乎跟哨兵是类似的。
  - 如果一个节点认为另外一个节点宕机，那么就是 pfail，主观宕机。那么会在 gossip ping 消息中，ping 给其他节点，如果超过半数的节点都认为 pfail 了，那么就会变成 fail，客观宕机。跟哨兵的原理几乎一样，sdown，odown。如果一个节点认为某个节点 pfail 了
  - **从节点过滤:**对宕机的 master node，从其所有的 slave node 中，选择一个切换成 master node。检查每个 slave node 与 master node 断开连接的时间，如果超过了 cluster-node-timeout * cluster-slave-validity-factor，那么就没有资格切换成 master。
  - **从节点选举:**每个从节点，都根据自己对 master 复制数据的 offset，来设置一个选举时间，offset 越大（复制数据越多）的从节点，选举时间越靠前，优先进行选举。所有的 master node 开始 slave 选举投票，给要进行选举的 slave 进行投票，如果大部分 master node（N/2 + 1）都投票给了某个从节点，那么选举通过，那个从节点可以切换成 master。

- Redis持久化的两种方式和比较
  - 持久化主要是做**灾难恢复、数据恢复**，也可以归类到高可用的一个环节中去
  - **RDB:** 持久化机制，是对 redis 中的数据执行周期性的持久化，RDB 会生成多个数据文件，每个数据文件都代表了某一个时刻中 redis 的数据
  - **AOF:**AOF 机制对每条写入命令作为日志，以 append-only 的模式写入一个日志文件中，在 redis 重启的时候，可以通过回放 AOF 日志中的写入指令来重新构建整个数据集。
  - RDB和AOF的优点：
    - RDB相对于 AOF 持久化机制来说，直接基于 RDB 数据文件来重启和恢复 redis 进程，更加快速。
    - AOF 可以更好的保护数据不丢失，一般 AOF 会每隔 1 秒，通过一个后台线程执行一次 fsync操作，最多丢失 1 秒钟的数据。
    - AOF 日志文件即使过大的时候，出现后台重写操作，也不会影响客户端的读写。因为在 rewrite log 的时候，会对其中的指令进行压缩，创建出一份需要恢复数据的最小日志出来。在创建新日志文件的时候，老的日志文件还是照常写入。当新的 merge 后的日志文件 ready 的时候，再交换新老日志文件即可。

- RDB和AOF的缺点：
  - 如果想要在 redis 故障时，尽可能少的丢失数据，那么 RDB 没有 AOF 好。一般来说，RDB 数据快照文件，都是每隔 5 分钟，或者更长时间生成一次，这个时候就得接受一旦 redis 进程宕机，那么会丢失最近 5 分钟的数据。
  - 对于同一份数据来说，AOF 日志文件通常比 RDB 数据快照文件更大。
- RBD和AOF的选择
  - 不要仅仅使用 RDB，因为那样会导致你丢失很多数据；
  - 也不要仅仅使用 AOF，因为那样有两个问题：第一，你通过 AOF 做冷备，没有 RDB 做冷备来的恢复速度更快；第二，RDB 每次简单粗暴生成数据快照，更加健壮，可以避免 AOF 这种复杂的备份和恢复机制的 bug；
  - redis 支持同时开启开启两种持久化方式，我们可以综合使用 AOF 和 RDB 两种持久化机制，用 AOF 来保证数据不丢失，作为数据恢复的第一选择; 用 RDB 来做不同程度的冷备，在 AOF 文件都丢失或损坏不可用的时候，还可以使用 RDB 来进行快速的数据恢复。

- redis集群模式的工作原理/节点间通信/选举主备切换
  - redis cluster介绍
    - 相当于多master+读写分离+高可用
    - 自动将数据进行分片，每个 master 上放一部分数据；提供内置的高可用支持，部分 master 不可用时，还是可以继续工作
    - redis 集群模式，可以做到在多台机器上，部署多个 redis 实例，每个实例存储一部分的数据，同时每个 redis 主实例可以挂 redis 从实例，自动确保说，如果 redis 主实例挂了，会自动切换到 redis 从实例上来。
    - 如果你数据量很少，主要是承载高并发高性能的场景，比如你的缓存一般就几个 G，单机就足够了，可以使用 replication，一个 master 多个 slaves，要几个 slave 跟你要求的读吞吐量有关，然后自己搭建一个 **sentinel 集群去保证 redis 主从架构的高可用性。**
    - redis cluster，主要是针对**海量数据+高并发+高可用**的场景。redis cluster 支撑 **N 个 redis master node**，每个 master node 都可以挂载多个 slave node。这样整个 redis 就可以横向扩容了。如果你要支撑更大数据量的缓存，那就横向扩容更多的 master 节点，每个 master 节点就能存放更多的数据了。
  - 节点间内部通信
    - 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
    - 集群元数据的维护有两种方式：**集中式、Gossip 协议**。redis cluster 节点间采用二进制协议- gossip 协议进行通信。
    - 集中式是将集群元数据（节点信息、故障等等）几种存储在某个节点上。集中式元数据集中存储的一个典型代表，就是大数据领域的 storm。它是分布式的大数据实时计算引擎，是集中式的元数据存储的结构，底层基于 zookeeper（分布式协调的中间件）对所有元数据进行存储维护。

    - gossip 协议，所有节点都持有一份元数据，不同的节点如果出现了元数据的变更，就不断将元数据发送给其它的节点，让其它节点也进行元数据的变更。
    - 集中式的好处在于，**元数据的读取和更新，时效性非常好**，一旦元数据出现了变更，就立即更新到集中式的存储中，其它节点读取的时候就可以感知到；不好在于，所有的元数据的**更新压力**全部集中在一个地方，可能会导致元数据的存储有压力。
    - gossip 好处在于，元数据的更新比较分散，不是集中在一个地方，更新请求会陆陆续续打到所有节点上去更新，降低了压力；不好在于，元数据的更新有延时，可能导致集群中的一些操作会有一些滞后。
  - gossip协议
    - gossip 协议包含多种消息，包含 ping,pong,meet,fail 等等
    - meet：某个节点发送 meet 给新加入的节点，让新节点加入集群中，然后新节点就会开始与其它节点进行通信。
    - ping：每个节点都会频繁给其它节点发送 ping，其中包含自己的状态还有自己维护的集群元数据，互相通过 ping 交换元数据。
    - pong：返回 ping 和 meeet，包含自己的状态和其它信息，也用于信息广播和更新。
    - fail：某个节点判断另一个节点 fail 之后，就发送 fail 给其它节点，通知其它节点说，某个节点宕机啦。
- 分布式寻址算法/一致性hash
  - hash算法、一致性hash算法、hash slot算法（redis）
  - hash算法
    - 对key进行hash取值之后，对master的数量取模
    - 问题：某一个 master 节点宕机，所有请求过来，都会基于最新的剩余 master 节点数去取模，尝试去取数据。这会导致大部分的请求无法拿到有效的缓存（取模得到的节点错了），导致大量的流量涌入数据库。（需要重新取模）
  - 一致性hash算法+虚拟节点（负载均衡）
    - 一致性 hash 算法将整个 hash 值空间组织成一个虚拟的圆环，整个空间按顺时针方向组织，下一步将各个 master 节点（使用服务器的 ip 或主机名）进行 hash。这样就能确定每个节点在其哈希环上的位置。来了一个 key，首先计算 hash 值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，遇到的第一个 master 节点就是 key 所在位置。
    - 如果一个节点挂了，受影响的数据仅仅是此节点到环空间前一个节点（沿着逆时针方向行走遇到的第一个节点）之间的数据，其它不受影响。增加一个节点也同理。
    - 一致性哈希算法在**节点太少**时，容易因为节点分布不均匀而造成**缓存热点**的问题。为了解决这种热点问题，一致性 hash 算法引入了虚拟节点机制，即对**每一个节点计算多个 hash，每个计算结果位置都放置一个虚拟节点**。这样就实现了数据的均匀分布，负载均衡。
  - **redis cluster 的 hash slot 算法**

    - redis cluster 有固定的 16384 个 hash slot，对每个 key 计算 CRC16 值，然后对 16384 取模，可以获取 key 对应的 hash slot。redis cluster 中每个 master 都会持有部分 slot
    - hash slot 让 node 的增加和移除很简单，增加一个 master，就将其他 master 的 hash slot 移动部分过去，减少一个 master，就将它的 hash slot 移动到其他 master 上去。移动 hash slot 的成本是非常低的。客户端的 api，可以对指定的数据，让他们走同一个 hash slot，通过 hash tag 来实现。任何一台机器宕机，另外两个节点，不影响的。因为 key 找的是 hash slot，不是机器。
- Redis雪崩、穿透、击穿
    - 缓存雪崩
        - 缓存机器意外发生了全盘宕机。缓存挂了,此时请求全部落数据库，数据库必然扛不住，然后宕机。哪怕恢复之后也会马上宕机
        - redis中的key在同一个时间段同时过期，直接落到数据库中
    - 缓存雪崩解决
        - 事前：redis 高可用，主从+哨兵/redis cluster，避免全盘崩溃。
        - 给过期的key设置一个随机时间，避免不要同时过期
        - 事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 被打死。
        - 事后：redis 持久化，一旦重启，自动从磁盘上加载数据，快速恢复缓存数据。
    - 缓存穿透
        - 黑客发出的恶意攻击，缓存中查不到，每次你去数据库里查，也查不到。缓存中不会有，请求每次都“视缓存于无物”，直接查询数据库。这种恶意攻击场景的缓存穿透就会直接把数据库给打死。
    - 缓存穿透解决
        - 1、在接口层增加校验，不合法的参数就直接返回
        - 2、每次系统 A 从数据库中只要没查到，就写一个空值到缓存里去，比如 set -999 UNKNOWN。然后设置一个过期时间，这样的话，下次有相同的 key 来访问的时候，在缓存失效之前，都可以直接从缓存中取数据。
        - 3、使用redis的布隆过滤器
    - 缓存击穿
        - 缓存击穿，就是说某个 key 非常热点，访问非常频繁，处于集中式高并发访问的情况，当这个 key 在失效的瞬间，大量的请求就击穿了缓存，直接请求数据库，就像是在一道屏障上凿开了一个洞。
    - 缓存击穿解决
        - 若缓存的数据是基本不会发生更新的，则可尝试将该热点数据设置为永不过期。
        - 若缓存的数据更新不频繁，且缓存刷新的整个流程耗时较少的情况下，则可以采用基于 redis、zookeeper 等分布式中间件的分布式互斥锁，或者本地互斥锁以保证仅少量的请求能请求数据库并重新构建缓存，其余线程则在锁释放后能访问到新缓存。

    - 若缓存的数据更新频繁或者缓存刷新的流程耗时较长的情况下，可以利用定时线程在缓存过期前主动的重新构建缓存或者延后缓存的过期时间，以保证所有的请求能一直访问到对应的缓存。

- 缓存和数据库双写一致性
    - 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
    - **读请求和写请求串行化：**串行化可以保证一定不会出现不一致的情况，但是它也会导致系统的吞吐量大幅度降低，用比正常情况下多几倍的机器去支撑线上的一个请求。
    - **Cache Aside Pattern**
        - 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。
        - 更新的时候，先更新数据库，然后再删除缓存（为什么删除而不是更新？缓存的值可能是由数据库计算出来的值、这个值可能是频繁修改而不是频繁访问的）
    - 缓存不一致问题1：
        - 先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。
        - 解决：先删除缓存，再更新数据库。
    - 缓存不一致问题2：（复杂、高并发出现）
        - 数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，**查到了修改前的旧数据**，放到了缓存中。随后数据变更的程序完成了数据库的修改。

- Redis并发问题/Redis事务的CAS
    - 多客户端同时并发写一个 key，可能本来应该先到的数据后到了，导致数据版本错了；或者是多客户端同时获取一个 key，修改值之后再写回去，只要顺序错了，数据就错了。
    - 可以基于 zookeeper 实现分布式锁。每个系统通过 zookeeper 获取分布式锁，确保同一时间，只能有一个系统实例在操作某个 key，别人都不允许读和写。

- Redis在生成环境中是怎样部署的
    - 看看你了解不了解你们公司的 redis 生产集群的部署架构，如果你不了解，那么确实你就很失职了，你的 redis 是主从架构？集群架构？用了哪种集群方案？有没有做高可用保证？有没有开启持久化机制确保可以进行数据恢复？线上 redis 给几个 G 的内存？设置了哪些参数？压测后你们 redis 集群承载多少 QPS？
    - 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。