Redis是什么？

Redis（`Remote Dictionary Server`）是一个使用 C 语言编写的，高性能非关系型的键值对数据库。与传统数据库不同的是，Redis 的数据是存在内存中的，所以读写速度非常快，被广泛应用于缓存方向。Redis可以将数据写入磁盘中，保证了数据的安全不丢失，而且Redis的操作是原子性的。

Redis优缺点?

优点：

1. **基于内存操作**，内存读写速度快。
2. **支持多种数据类型**，包括String、Hash、List、Set、ZSet等。
3. **支持持久化。**Redis支持RDB和AOF两种持久化机制，持久化功能可以有效地避免数据丢失问题。
4. **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。
5. **支持主从复制。**主节点会自动将数据同步到从节点，可以进行读写分离。
6. Redis命令的处理是**单线程**的。Redis6.0引入了多线程，需要注意的是，**多线程用于处理网络数据的读写和协议解析**，Redis命令执行还是单线程的。

缺点：

1. 对结构化查询的支持比较差。
2. 数据库容量受到物理内存的限制，不适合用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的操作。
3. Redis 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。

Redis为什么这么快?

- **基于内存**：Redis是使用内存存储，没有磁盘IO上的开销。数据存在内存中，读写速度快。
- **IO多路复用模型**：Redis 采用 IO 多路复用技术。Redis 使用单线程来轮询描述符，将数据库的操作都转换成了事件，不在网络I/O上浪费过多的时间。
- **高效的数据结构**：Redis 每种数据类型底层都做了优化，目的就是为了追求更快的速度。

讲讲Redis的线程模型?

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。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accept、read、write、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。

Redis应用场景有哪些?

1. **缓存热点数据**，缓解数据库的压力。
2. 利用 Redis 原子性的自增操作，可以实现**计数器**的功能，比如统计用户点赞数、用户访问数等。
3. **简单的消息队列**，可以使用Redis自身的发布/订阅模式或者List来实现简单的消息队列，实现异步操作。

4. **限速器**，可用于限制某个用户访问某个接口的频率，比如秒杀场景用于防止用户快速点击带来不必要的压力。
5. **好友关系**，利用集合的一些命令，比如交集、并集、差集等，实现共同好友、共同爱好之类的功能。

Memcached和Redis的区别?

1. MemCached 数据结构单一，仅用来缓存数据，而 **Redis 支持多种数据类型。**
2. MemCached 不支持数据持久化，重启后数据会消失。**Redis 支持数据持久化。**
3. **Redis 提供主从同步机制和 cluster 集群部署能力**，能够提供高可用服务。Memcached 没有提供原生的集群模式，需要依靠客户端实现往集群中分片写入数据。
4. Redis 的速度比 Memcached 快很多。
5. Redis 使用**单线程的多路 IO 复用模型**，Memcached使用多线程的非阻塞 IO 模型。（Redis6.0引入了多线程IO，**用来处理网络数据的读写和协议解析**，但是命令的执行仍然是单线程）
6. value 值大小不同：Redis 最大可以达到 512M；memcache 只有 1mb。

为什么要用 Redis 而不用 map/guava 做缓存?

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。

Redis 数据类型有哪些?

基本数据类型:

1、**String**：最常用的一种数据类型，String类型的值可以是字符串、数字或者二进制，但值最大不能超过512MB。

2、**Hash**：Hash 是一个键值对集合。

3、**Set**：无序去重的集合。Set 提供了交集、并集等方法，对于实现共同好友、共同关注等功能特别方便。

4、**List**：有序可重复的集合，底层是依赖双向链表实现的。

5、**SortedSet**：有序Set。内部维护了一个`score`的参数来实现。适用于排行榜和带权重的消息队列等场景。

特殊的数据类型:

1、**Bitmap**：位图，可以认为是一个以位为单位数组，数组中的每个单元只能存0或者1，数组的下标在 Bitmap 中叫做偏移量。Bitmap的长度与集合中元素个数无关，而是与基数的上限有关。

2、**Hyperloglog。**HyperLogLog 是用来做基数统计的算法，其优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。典型的使用场景是统计独立访客。

3、**Geospatial** ：主要用于存储地理位置信息，并对存储的信息进行操作，适用场景如定位、附近的人等。

SortedSet和List异同点?

相同点:

1. 都是有序的；
2. 都可以获得某个范围内的元素。

不同点：

1. 列表基于链表实现，获取两端元素速度快，访问中间元素速度慢；
2. 有序集合基于散列表和跳跃表实现，访问中间元素时间复杂度是OlogN；
3. 列表不能简单的调整某个元素的位置，有序列表可以（更改元素的分数）；
4. 有序集合更耗内存。

Redis的内存用完了会怎样?

如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回）。

也可以配置内存淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

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。

keys命令存在的问题?

redis的单线程的。keys指令会导致线程阻塞一段时间，直到执行完毕，服务才能恢复。scan采用渐进式遍历的方式来解决keys命令可能带来的阻塞问题，每次scan命令的时间复杂度是`O(1)`，但是要真正实现keys的功能，需要执行多次scan。

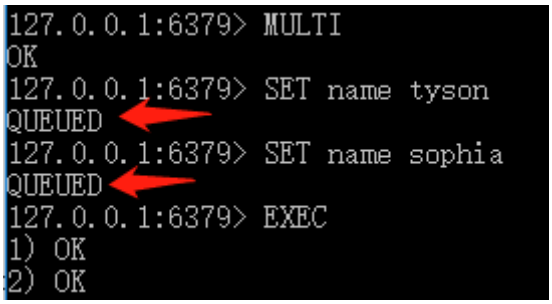
scan的缺点：在scan的过程中如果有键的变化（增加、删除、修改），遍历过程可能会有以下问题：新增的键可能没有遍历到，遍历出了重复的键等情况，也就是说scan并不能保证完整的遍历出来所有的键。

Redis事务

事务的原理是将一个事务范围内的若干命令发送给Redis，然后再让Redis依次执行这些命令。

事务的生命周期：

1. 使用MULTI开启一个事务
2. 在开启事务的时候，每次操作的命令将会被插入到一个队列中，同时这个命令并不会被真的执行
3. EXEC命令进行提交事务



一个事务范围内某个命令出错不会影响其他命令的执行，不保证原子性：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set a 1
QUEUED
127.0.0.1:6379> set b 1 2
QUEUED
127.0.0.1:6379> set c 3
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) ERR syntax error
3) OK
```

WATCH命令

`WATCH`命令可以监控一个或多个键，一旦其中有一个键被修改，之后的事务就不会执行（类似于乐观锁）。执行`EXEC`命令之后，就会自动取消监控。

```
127.0.0.1:6379> watch name
OK
127.0.0.1:6379> set name 1
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set name 2
QUEUED
127.0.0.1:6379> set gender 1
QUEUED
127.0.0.1:6379> exec
(nil)
127.0.0.1:6379> get gender
(nil)
```

比如上面的代码中：

1. `watch name`开启了对`name`这个`key`的监控
2. 修改`name`的值
3. 开启事务a
4. 在事务a中设置了`name`和`gender`的值
5. 使用`EXEC`命令进提交事务
6. 使用命令`get gender`发现不存在，即事务a没有执行

使用`UNWATCH`可以取消`WATCH`命令对`key`的监控，所有监控锁将会被取消。

Redis事务支持隔离性吗?

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。

持久化机制

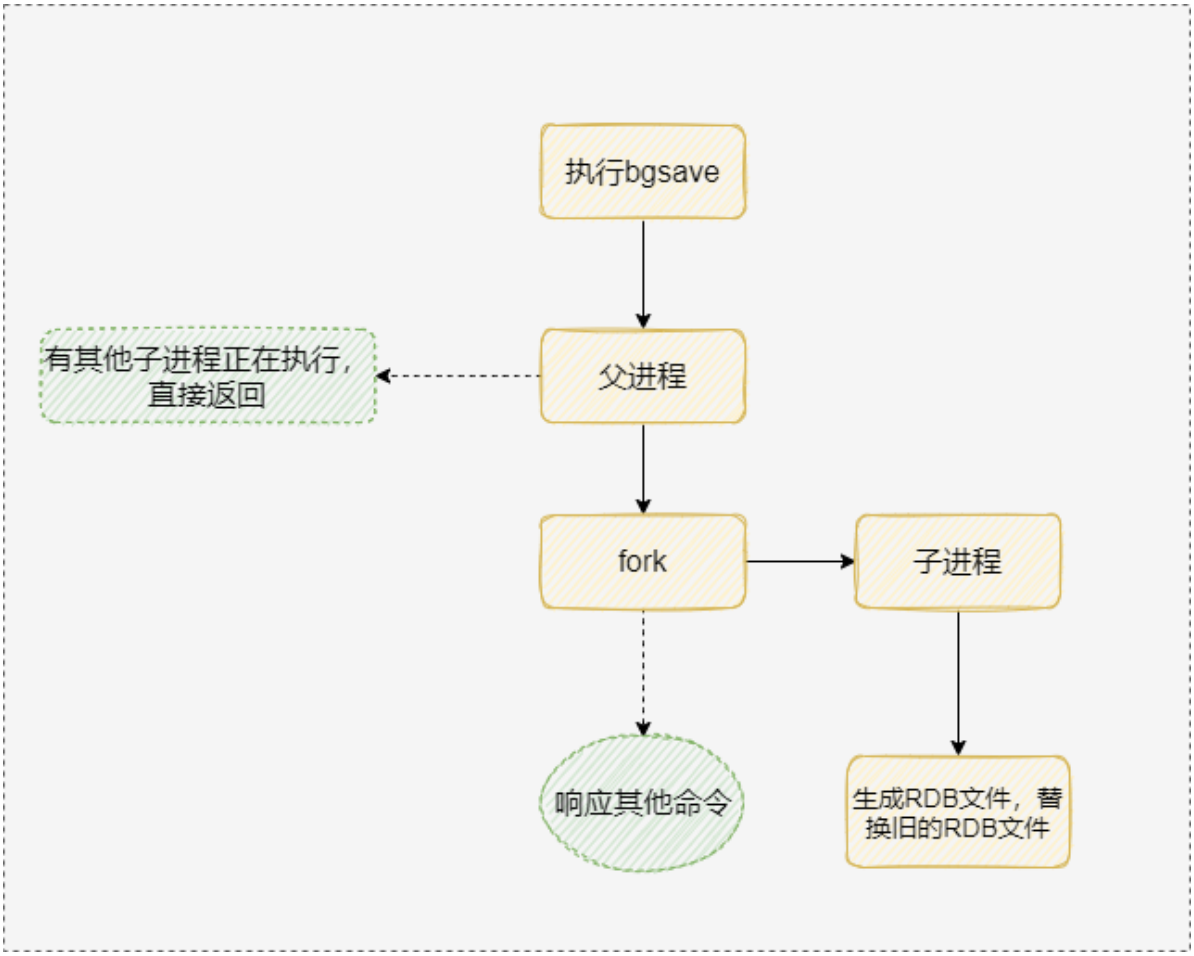
持久化就是把**内存的数据写到磁盘中**，防止服务宕机导致内存数据丢失。

Redis支持两种方式的持久化，一种是`RDB`的方式，一种是`AOF`的方式。**前者会根据指定的规则定时将内存中的数据存储在硬盘上**，而**后者在每次执行完命令后将命令记录下来**。一般将两者结合使用。

RDB方式

`RDB`是 Redis 默认的持久化方案。RDB持久化时会将内存中的数据写入到磁盘中，在指定目录下生成一个`dump.rdb`文件。Redis 重启会加载`dump.rdb`文件恢复数据。

`bgsave`是主流的触发 RDB 持久化的方式，执行过程如下：



- 执行`BGSAVE`命令
- Redis 父进程判断当前**是否存在正在执行的子进程**，如果存在，`BGSAVE`命令直接返回。
- 父进程执行`fork`操作**创建子进程**，fork操作过程中父进程会阻塞。
- 父进程`fork`完成后，**父进程继续接收并处理客户端的请求**，而**子进程开始将内存中的数据写进硬盘的临时文件**；
- 当子进程写完所有数据后会**用该临时文件替换旧的 RDB 文件。**

Redis启动时会读取RDB快照文件，将数据从硬盘载入内存。通过 RDB 方式的持久化，一旦Redis异常退出，就会丢失最近一次持久化以后更改的数据。

触发 RDB 持久化的方式：

1. **手动触发**：用户执行`SAVE`或`BGSAVE`命令。`SAVE`命令执行快照的过程会阻塞所有客户端的请求，应避免在生产环境使用此命令。`BGSAVE`命令可以在后台异步进行快照操作，快照的同时服务器还可以继续响应客户端的请求，因此需要手动执行快照时推荐使用`BGSAVE`命令。
2. **被动触发**：
    - 根据配置规则进行自动快照，如`SAVE 100 10`，100秒内至少有10个键被修改则进行快照。
    - 如果从节点执行全量复制操作，主节点会自动执行`BGSAVE`生成 RDB 文件并发送给从节点。

- 默认情况下执行`shutdown`命令时，如果没有开启 AOF 持久化功能则自动执行·BGSAVE·。

优点：

1. **Redis 加载 RDB 恢复数据远远快于 AOF 的方式。**
2. 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。**

缺点：

1. **RDB方式数据无法做到实时持久化。**因为`BGSAVE`每次运行都要执行`fork`操作创建子进程，属于重量级操作，频繁执行成本比较高。
2. RDB 文件使用特定二进制格式保存，Redis 版本升级过程中有多个格式的 RDB 版本，**存在老版本 Redis 无法兼容新版 RDB 格式的问题。**

AOF方式

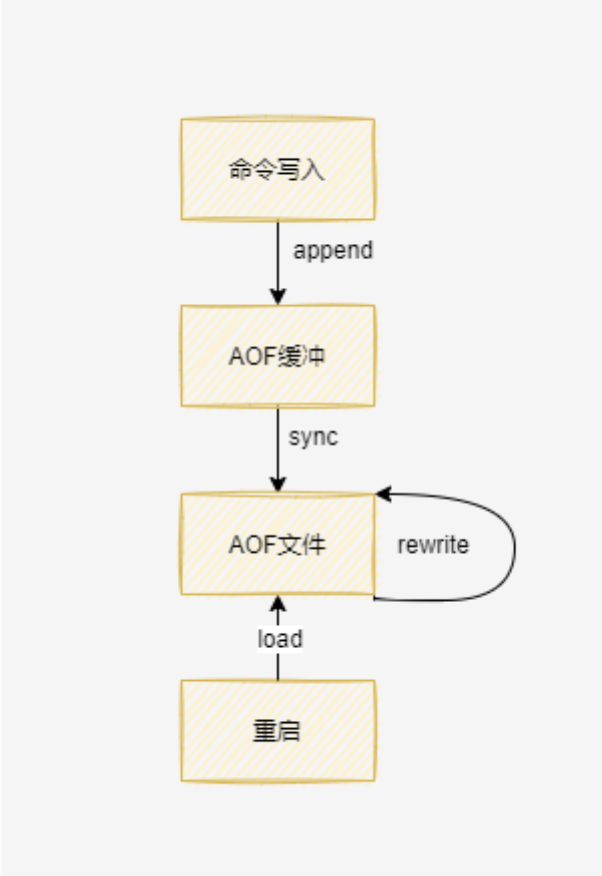
AOF（append only file）持久化：以独立日志的方式记录每次写命令，Redis重启时会重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是**解决了数据持久化的实时性**，AOF 是Redis持久化的主流方式。

默认情况下Redis没有开启AOF方式的持久化，可以通过`appendonly`参数启用：`appendonly yes`。开启AOF方式持久化后每执行一条写命令，Redis就会将该命令写进`aof_buf`缓冲区，AOF缓冲区根据对应的策略向硬盘做同步操作。

默认情况下系统**每30秒**会执行一次同步操作。为了防止缓冲区数据丢失，可以在Redis写入AOF文件后主动要求系统将缓冲区数据同步到硬盘上。可以通过`appendfsync`参数设置同步的时机。

```
appendfsync always //每次写入aof文件都会执行同步，最安全最慢，不建议配置
appendfsync everysec  //既保证性能也保证安全，建议配置
appendfsync no //由操作系统决定何时进行同步操作
```

接下来看一下 AOF 持久化执行流程：



1. 所有的写入命令会追加到 AOP 缓冲区中。
2. AOF 缓冲区根据对应的策略向硬盘同步。
3. 随着 AOF 文件越来越大，需要定期对 AOF 文件进行重写，达到压缩文件体积的目的。AOF文件重写是把Redis进程内的数据转化为写命令同步到新AOF文件的过程。
4. 当 Redis 服务器重启时，可以加载 AOF 文件进行数据恢复。

优点：

1. AOF可以更好的保护数据不丢失，可以配置 AOF 每秒执行一次`fsync`操作，如果Redis进程挂掉，最多丢失1秒的数据。
2. AOF以`append-only`的模式写入，所以没有磁盘寻址的开销，写入性能非常高。

缺点：

1. 对于同一份文件AOF文件比RDB数据快照要大。
2. 数据恢复比较慢。

RDB和AOF如何选择?

通常来说，应该同时使用两种持久化方案，以保证数据安全。

- 如果数据不敏感，且可以从其他地方重新生成，可以关闭持久化。
- 如果数据比较重要，且能够承受几分钟的数据丢失，比如缓存等，只需要使用RDB即可。
- 如果是用做内存数据，要使用Redis的持久化，建议是RDB和AOF都开启。
- 如果只用AOF，优先使用everysec的配置选择，因为它在可靠性和性能之间取了一个平衡。

当RDB与AOF两种方式都开启时，Redis会优先使用AOF恢复数据，因为AOF保存的文件比RDB文件更完整。

Redis有哪些部署方案?

单机版：单机部署，单机redis能够承载的 QPS 大概就在上万到几万不等。这种部署方式很少使用。存在的问题：1、内存容量有限 2、处理能力有限 3、无法高可用。

主从模式：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。master 节点挂掉后，需要手动指定新的 master，可用性不高，基本不用。

哨兵模式：主从复制存在不能自动故障转移、达不到高可用的问题。哨兵模式解决了这些问题。通过哨兵机制可以自动切换主从节点。master 节点挂掉后，哨兵进程会主动选举新的 master，可用性高，但是每个节点存储的数据是一样的，浪费内存空间。数据量不是很多，集群规模不是很大，需要自动容错容灾的时候使用。

Redis cluster：服务端分片技术，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。主要是针对海量数据+高并发+高可用的场景，如果是海量数据，如果你的数据量很大，那么建议就用Redis cluster，所有主节点的容量总和就是Redis cluster可缓存的数据容量。

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的复制功能是支持多个数据库之间的数据同步。主数据库可以进行读写操作，当主数据库的数据发生变化时会自动将数据同步到从数据库。从数据库一般是只读的，它会接收主数据库同步过来的数据。一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

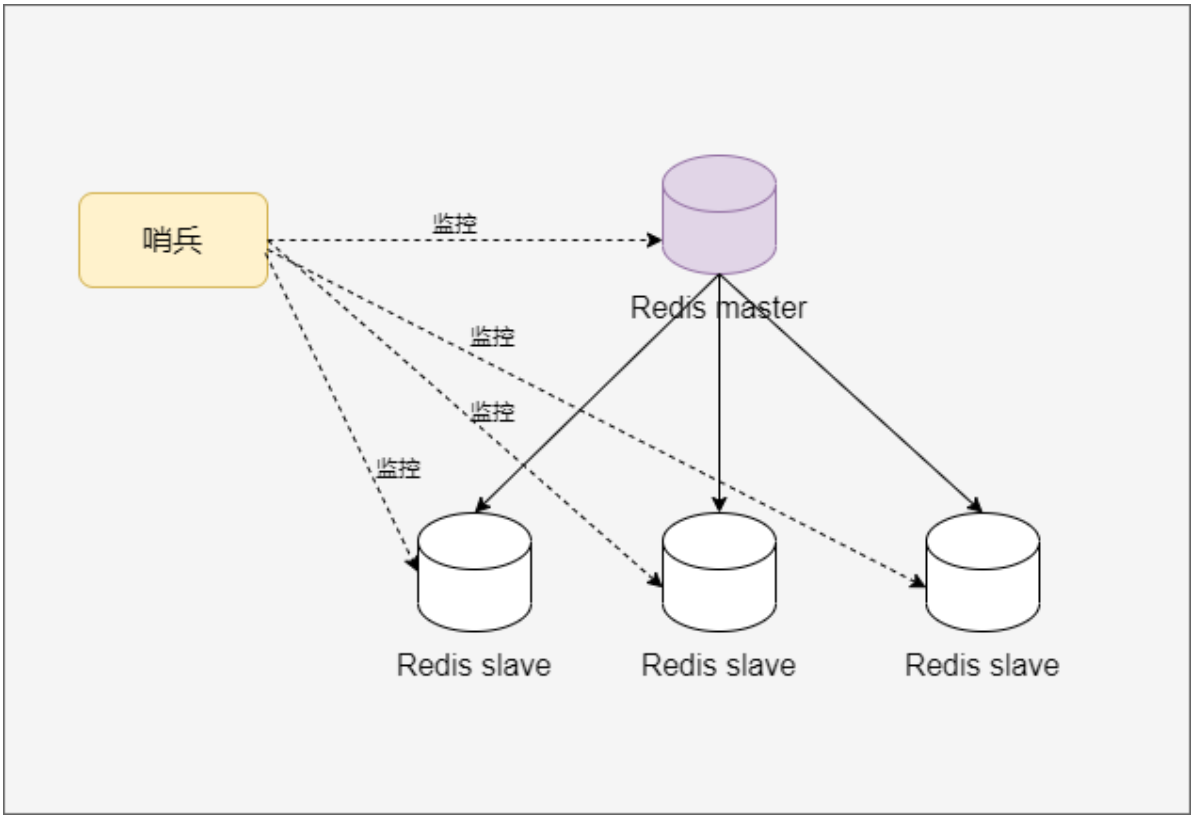
主从复制的原理?

1. 当启动一个从节点时，它会发送一个 `PSYNC` 命令给主节点；
2. 如果是从节点初次连接到主节点，那么会触发一次全量复制。此时主节点会启动一个后台线程，开始生成一份 `RDB` 快照文件；
3. 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， 主节点会将 `RDB` 文件发送给从节点，从节点会先将 `RDB` 文件**写入本地磁盘，然后再从本地磁盘加载到内存中**；
4. 接着主节点会将内存中缓存的写命令发送到从节点，从节点同步这些数据；
5. 如果从节点跟主节点之间网络出现故障，连接断开了，会自动重连，连接之后主节点仅会将部分缺失的数据同步给从节点。

哨兵Sentinel

主从复制存在不能自动故障转移、达不到高可用的问题。哨兵模式解决了这些问题。通过哨兵机制可以自动切换主从节点。

客户端连接Redis的时候，先连接哨兵，哨兵会告诉客户端Redis主节点的地址，然后客户端连接上Redis并进行后续的操作。当主节点宕机的时候，哨兵监测到主节点宕机，会重新推选出某个表现良好的从节点成为新的主节点，然后通过发布订阅模式通知其他的从服务器，让它们切换主机。



工作原理

- 每个 `Sentinel` 以每秒钟一次的频率向它所知道的 `Master`， `Slave` 以及其他 `Sentinel` 实例发送一个 `PING` 命令。
- 如果一个实例距离最后一次有效回复 `PING` 命令的时间超过指定值， 则这个实例会被 `Sentine` 标记为主观下线。
- 如果一个 `Master` 被标记为主观下线，则正在监视这个 `Master` 的所有 `Sentinel` 要以每秒一次的频率确认 `Master` 是否真正进入主观下线状态。
- 当有足够数量的 `Sentinel` （大于等于配置文件指定值）在指定的时间范围内确认 `Master` 的确进入了主观下线状态， 则 `Master` 会被标记为客观下线 。若没有足够数量的 `Sentinel` 同意 `Master`

已经下线，`Master` 的客观下线状态就会被解除。 若 `Master` 重新向 `Sentinel` 的 `PING` 命令返回有效回复，`Master` 的主观下线状态就会被移除。

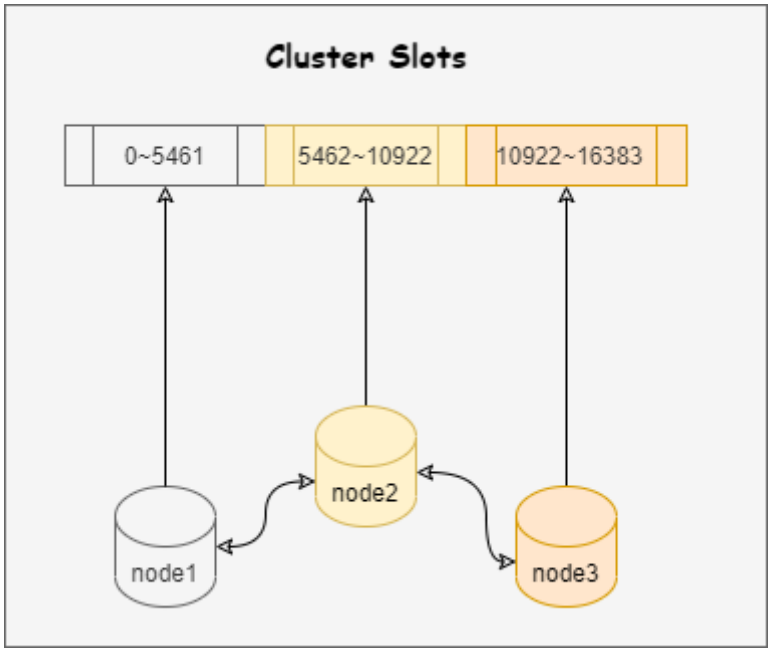
- 哨兵节点会选举出哨兵 leader，负责故障转移的工作。
- 哨兵 leader 会推选出某个表现良好的从节点成为新的主节点，然后通知其他从节点更新主节点信息。

Redis cluster

哨兵模式解决了主从复制不能自动故障转移、达不到高可用的问题，但还是存在主节点的写能力、容量受限于单机配置的问题。而cluster模式实现了Redis的分布式存储，每个节点存储不同的内容，解决主节点的写能力、容量受限于单机配置的问题。

Redis cluster集群节点最小配置6个节点以上（3主3从），其中主节点提供读写操作，从节点作为备用节点，不提供请求，只作为故障转移使用。

Redis cluster采用**虚拟槽分区**，所有的键根据哈希函数映射到0 ~ 16383个整数槽内，每个节点负责维护一部分槽以及槽所映射的键值数据。



工作原理：

1. 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
2. 每份数据分片会存储在多个互为主从的多节点上
3. 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)
4. 同一分片多个节点间的数据不保持一致性
5. 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
6. 扩容时时需要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。

优点：

- 无中心架构，**支持动态扩**容；
- 数据按照`slot`存储分布在多个节点，节点间数据共享，**可动态调整数据分布**；

- **高可用性。**部分节点不可用时，集群仍可用。集群模式能够实现自动故障转移（failover），节点之间通过`gossip`协议交换状态信息，用投票机制完成`Slave`到`Master`的角色转换。

缺点：

- **不支持批量操作**（pipeline）。
- 数据通过异步复制，**不保证数据的强一致性。**
- **事务操作支持有限**，只支持多`key`在同一节点上的事务操作，当多个`key`分布于不同的节点上时无法使用事务功能。
- `key`作为数据分区的最小粒度，不能将一个很大的键值对象如`hash`、`list`等映射到不同的节点。
- **不支持多数据库空间**，单机下的Redis可以支持到16个数据库，集群模式下只能使用1个数据库空间。
- 只能使用0号数据库。

哈希分区算法有哪些?

节点取余分区。使用特定的数据，如Redis的键或用户ID，对节点数量N取余：hash（key）%N计算出哈希值，用来决定数据映射到哪一个节点上。
优点是简单性。扩容时通常采用翻倍扩容，避免数据映射全部被打乱导致全量迁移的情况。

一致性哈希分区。为系统中每个节点分配一个token，范围一般在0~232，这些token构成一个哈希环。数据读写执行节点查找操作时，先根据key计算hash值，然后顺时针找到第一个大于等于该哈希值的token节点。
这种方式相比节点取余最大的好处在于加入和删除节点只影响哈希环中相邻的节点，对其他节点无影响。

虚拟槽分区，所有的键根据哈希函数映射到0~16383整数槽内，计算公式：slot=CRC16（key）&16383。每一个节点负责维护一部分槽以及槽所映射的键值数据。**Redis Cluser采用虚拟槽分区算法。**

过期键的删除策略?

1、**被动删除。**在访问key时，如果发现key已经过期，那么会将key删除。

2、**主动删除。**定时清理key，每次清理会依次遍历所有DB，从db随机取出20个key，如果过期就删除，如果其中有5个key过期，那么就继续对这个db进行清理，否则开始清理下一个db。

3、**内存不够时清理。**Redis有最大内存的限制，通过maxmemory参数可以设置最大内存，当使用的内存超过了设置的最大内存，就要进行内存释放， 在进行内存释放的时候，会按照配置的淘汰策略清理内存。

内存淘汰策略有哪些?

当Redis的内存超过最大允许的内存之后，Redis 会触发内存淘汰策略，删除一些不常用的数据，以保证Redis服务器正常运行。

Redisv4.0前提供 6 种数据淘汰策略：

- **volatile-lru**：LRU（`Least Recently Used`），最近使用。利用LRU算法移除设置了过期时间的key
- **allkeys-lru**：当内存不足以容纳新写入数据时，从数据集中移除最近最少使用的key
- **volatile-ttl**：从已设置过期时间的数据集中挑选将要过期的数据淘汰
- **volatile-random**：从已设置过期时间的数据集中任意选择数据淘汰
- **allkeys-random**：从数据集中任意选择数据淘汰
- **no-eviction**：禁止删除数据，当内存不足以容纳新写入数据时，新写入操作会报错

Redisv4.0后增加以下两种：

- **volatile-lfu**：LFU，Least Frequently Used，最少使用，从已设置过期时间的数据集中挑选最不经常使用的数据淘汰。
- **allkeys-lfu**：当内存不足以容纳新写入数据时，从数据集中移除最不经常使用的key。

内存淘汰策略可以通过配置文件来修改，相应的配置项是`maxmemory-policy`，默认配置是`noeviction`。

如何保证缓存与数据库双写时的数据一致性?

1、先删除缓存再更新数据库

进行更新操作时，先删除缓存，然后更新数据库，后续的请求再次读取时，会从数据库读取后再将新数据更新到缓存。

存在的问题：删除缓存数据之后，更新数据库完成之前，这个时间段内如果有新的读请求过来，就会从数据库读取旧数据重新写到缓存中，再次造成不一致，并且后续读的都是旧数据。

2、先更新数据库再删除缓存

进行更新操作时，先更新MySQL，成功之后，删除缓存，后续读取请求时再将新数据回写缓存。

存在的问题：更新MySQL和删除缓存这段时间内，请求读取的还是缓存的旧数据，不过等数据库更新完成，就会恢复一致，影响相对比较小。

3、异步更新缓存

数据库的更新操作完成后不直接操作缓存，而是把这个操作命令封装成消息扔到消息队列中，然后由Redis自己去消费更新数据，消息队列可以保证数据操作顺序一致性，确保缓存系统的数据正常。

以上几个方案都不完美，需要根据业务需求，评估哪种方案影响较小，然后选择相应的方案。

缓存常见问题

缓存穿透

缓存穿透是指查询一个**不存在的数据**，由于缓存是不命中时被动写的，如果从DB查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到DB去查询，失去了缓存的意义。在流量大时，可能DB就挂掉了。

怎么解决?

1. **缓存空值**，不会查数据库。
2. 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，查询不存在的数据会被这个`bitmap`拦截掉，从而避免了对`DB`的查询压力。

布隆过滤器的原理：当一个元素被加入集合时，通过K个哈希函数将这个元素映射成一个位数组中的K个点，把它们置为1。查询时，将元素通过哈希函数映射之后会得到k个点，如果这些点有任何一个0，则被检元素一定不在，直接返回；如果都是1，则查询元素很可能存在，就会去查询Redis和数据库。

布隆过滤器一般用于在大数据量的集合中判定某元素是否存在。

缓存雪崩

缓存雪崩是指在我们设置缓存时采用了相同的过期时间，**导致缓存在某一时刻同时失效**，请求全部转发到DB，DB瞬时压力过重挂掉。

解决方法：

1. 在原有的失效时间基础上**增加一个随机值**，使得过期时间分散一些。这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。
2. **加锁排队可以起到缓冲的作用**，防止大量的请求同时操作数据库，但它的缺点是**增加了系统的响应时间**，**降低了系统的吞吐量**，牺牲了一部分用户体验。当缓存未查询到时，对要请求的 key 进行加锁，只允许一个线程去数据库中查，其他线程等候排队。
3. 设置二级缓存。二级缓存指的是除了 Redis 本身的缓存，**再设置一层缓存**，当 Redis 失效之后，先去查询二级缓存。例如可以设置一个本地缓存，在 Redis 缓存失效的时候先去查询本地缓存而非查询数据库。

缓存击穿

缓存击穿：大量的请求同时查询一个 key 时，此时这个 key 正好失效了，就会导致大量的请求都落到数据库。**缓存击穿是查询缓存中失效的 key，而缓存穿透是查询不存在的 key。**

解决方法：

1、**加互斥锁。**在并发的多个请求中，只有第一个请求线程能拿到锁并执行数据库查询操作，其他的线程拿不到锁就阻塞等着，等到第一个线程将数据写入缓存后，直接走缓存。可以使用Redis分布式锁实现，代码如下：

```
public String get(String key) {
    String value = redis.get(key);
    if (value == null) { //缓存值过期
        String unique_key = systemId + ":" + key;
        //设置30s的超时
        if (redis.set(unique_key, 1, 'NX', 'PX', 30000) == 1) {  //设置成功
            value = db.get(key);
            redis.set(key, value, expire_secs);
            redis.del(unique_key);
        } else {  //其他线程已经到数据库取值并回写到缓存了，可以重试获取缓存值
            sleep(50);
            get(key);  //重试
        }
    } else {
        return value;
    }
}
```

2、**热点数据不过期。**直接将缓存设置为不过期，然后由定时任务去异步加载数据，更新缓存。这种方式适用于比较极端的场景，例如流量特别特别大的场景，使用时需要考虑业务能接受数据不一致的时间，还有就是异常情况的处理，保证缓存可以定时刷新。

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）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

1. 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
2. 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；
3. 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；
4. 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

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。

Redis 怎么实现消息队列?

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。

```
BLPOP queue 0  //0表示不限制等待时间
```

> BLPOP和LPOP命令相似，唯一的区别就是当列表没有元素时BLPOP命令会一直阻塞连接，直到有新元素加入。

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。

```
PUBLISH channel1 hi
SUBSCRIBE channel1
UNSUBSCRIBE channel1 //退订通过SUBSCRIBE命令订阅的频道。
```

> `PSUBSCRIBE channel?*` 按照规则订阅。
> `PUNSUBSCRIBE channel?*` 退订通过PSUBSCRIBE命令按照某种规则订阅的频道。其中订阅规则要进行严格的字符串匹配，`PUNSUBSCRIBE *`无法退订`channel?*`规则。

Redis 怎么实现延时队列

使用sortedset，拿时间戳作为score，消息内容作为key，调用zadd来生产消息，消费者用`zrangebyscore`指令获取N秒之前的数据轮询进行处理。

pipeline的作用?

redis客户端执行一条命令分4个过程： 发送命令、命令排队、命令执行、返回结果。使用`pipeline`可以批量请求，批量返回结果，执行速度比逐条执行要快。

使用`pipeline`组装的命令个数不能太多，不然数据量过大，增加客户端的等待时间，还可能造成网络阻塞，可以将大量命令的拆分多个小的`pipeline`命令完成。

原生批命令（mset和mget）与`pipeline`对比：

1. 原生批命令是原子性，`pipeline`是**非原子性**。pipeline命令中途异常退出，之前执行成功的命令**不会回滚**。
2. 原生批命令只有一个命令，但`pipeline`**支持多命令**。

LUA脚本

Redis 通过 LUA 脚本创建具有原子性的命令： 当lua脚本命令正在运行的时候，不会有其他脚本或 Redis 命令被执行，实现组合命令的原子操作。

在Redis中执行Lua脚本有两种方法：`eval`和`evalsha`。`eval`命令使用内置的 Lua 解释器，对 Lua 脚本进行求值。

```
//第一个参数是lua脚本，第二个参数是键名参数个数，剩下的是键名参数和附加参数
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

lua脚本作用

1、Lua脚本在Redis中是原子执行的，执行过程中间不会插入其他命令。

2、Lua脚本可以将多条命令一次性打包，有效地减少网络开销。

应用场景

举例：限制接口访问频率。

在Redis维护一个接口访问次数的键值对，`key`是接口名称，`value`是访问次数。每次访问接口时，会执行以下操作：

- 通过`aop`拦截接口的请求，对接口请求进行计数，每次进来一个请求，相应的接口访问次数`count`加1，存入redis。
- 如果是第一次请求，则会设置`count=1`，并设置过期时间。因为这里`set()`和`expire()`组合操作不是原子操作，所以引入`lua`脚本，实现原子操作，避免并发访问问题。
- 如果给定时间范围内超过最大访问次数，则会抛出异常。

```
private String buildLuaScript() {
    return "local c" +
        "\nc = redis.call('get',KEYS[1])" +
        "\nif c and tonumber(c) > tonumber(ARGV[1]) then" +
        "\nreturn c;" +
        "\nend" +
        "\nc = redis.call('incr',KEYS[1])" +
        "\nif tonumber(c) == 1 then" +
        "\nredis.call('expire',KEYS[1],ARGV[2])" +
        "\nend" +
        "\nreturn c;";
}

String luaScript = buildLuaScript();
```

```
RedisScript<Number> redisScript = new DefaultRedisScript<>(luaScript,
Number.class);
Number count = redisTemplate.execute(redisScript, keys, limit.count(),
limit.period());
```

PS：这种接口限流的实现方式比较简单，问题也比较多，一般不会使用，接口限流用的比较多的是令牌桶算法和漏桶算法。

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*，此种方式比原先的单节点的方法更安全。

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 挂掉了
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

Redis大key怎么处理?

> BigKey的危害:
>
> - 网络阻塞：对BigKey执行读请求时，少陵的QPS就可能导致带宽被占满，导致redis实例乃至所在物理机变慢
> - 数据倾斜：BigKey所在的Redis实例内存使用率远超其他实例，无法使数据分片的内存资源达到均衡
> - Redis阻塞：对元素较多的hash、list、zset等做运算会耗时较久，使主线程阻塞
> - CPU压力：对BigKey的数据序列化和反序列化会导致CPU的使用率飙升，影响Redis实例和本机其他应用
>
> 如何发现BigKey :
>
> - redis-cli --bigkeys
>
> 利用redis-cli提供的--bigkeys参数，可以遍历分析所有key，并返回Key的整体统计信息与每个数据类型的Top1的BigKey
>
> - scan扫描
>
> 自己编程，利用scan扫描Redis中的所有key，利用strlen、hlen等命令判断Key的长度（此处不建议使用MEMORY USAGE）
>
> - 第三方工具
>
> 利用第三方工具，如Redis-Rdb-Tols分析RDB快照文件，全面分析内存使用情况
>
> - 网络加快
>
> 自定义工具，监控进出Redis的网络数据，超出预警值时主动告警

通常我们会将含有较大数据或含有大量成员、列表数的Key称之为大Key。

以下 是对各个数据类型大key的描述：

- value是STRING类型，它的值超过5MB
- value是ZSET、Hash、List、Set等集合类型时，它的成员数量超过1w个

上述的定义并不绝对，主要是根据value的成员数量和大小来确定，根据业务场景确定标准。

怎么处理：

1. 当vaule是string时，可以使用序列化、压缩算法将key的大小控制在合理范围内，但是序列化和反序列化都会带来更多时间上的消耗。或者将key进行拆分，一个大key分为不同的部分，记录每个部分的key，使用multiget等操作实现事务读取。
2. 当value是list/set等集合类型时，根据预估的数据规模来进行分片，不同的元素计算后分到不同的片。

> 怎么删除BigKey?
>
> - Redis3.0以前
>
> 如果是集合类型，则遍历BigKey的元素，先逐个删除子元素，最后删除BigKey
> - Redis4.0以后
>
> Redis在4.0后提供了异步删除的命令:unlink

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。