第三版：Redis 60 道

Redis 相比 Memcached 有哪些优势：

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

Redis 是单线程

Redis 利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销

Reids 常用 5 种数据类型

string，list，set，sorted set，hash

Reids6 种淘汰策略：

noeviction：不删除策略，达到最大内存限制时，如果需要更多内存，直接返回错误信息。大多数写命令都会导致占用更多的内存(有极少数会例外。

allkeys-lru:所有 key 通用；优先删除最近最少使用(less recently used ,LRU) 的 key。

volatile-lru:只限于设置了 expire 的部分；优先删除最近最少使用(less recently used ,LRU) 的 key。

allkeys-random:所有 key 通用；随机删除一部分 key。

volatile-random: 只限于设置了 **expire** 的部分；随机删除一部分 key。

volatile-ttl: 只限于设置了 **expire** 的部分；优先删除剩余时间(time to live,TTL) 短的 key。

Redis 的并发竞争问题如何解决?

单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis 本身没有锁的概念，Redis 对于多个客户端连接并不存在竞争，利用 setnx 实现锁。

Redis 前端启动命令

./Redis-server

Reids 支持的语言:

java、C、C#、C++、php、Node.js、Go 等。

Redis 持久化方案:

Rdb 和 Aof

Redis 的主从复制

持久化保证了即使 Redis 服务重启也不会丢失数据，因为 Redis 服务重启后会将硬盘上持久化的数据恢复到内存中，但是当 Redis 服务器的硬盘损坏了可能会导致数据丢失，如果通过 Redis 的主从复制机制就可以避免这种单点故障，

Redis 是单线程的，但 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、 数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；

3、 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，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；**这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程**

5、 使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 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；

为什么 Redis 是单线程的?

Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且 CPU 不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

Redis 内存模型

used_memory：Redis 分配器分配的内存总量（单位是字节），包括使用的虚拟内存（即 swap）；Redis 分配器后面会介绍。used_memory_human 只是显示更友好。

used_memory_rss：Redis 进程占据操作系统的内存（单位是字节），与 top 及 ps 命令看到的值是一致的；除了分配器分配的内存之外，used_memory_rss 还包括进程运行本身需要的内存、内存碎片等，但是不包括虚拟内存。

mem_fragmentation_ratio：内存碎片比率，该值是 used_memory_rss / used_memory 的比值。

mem_allocator：Redis 使用的内存分配器，在编译时指定；可以是 libc 、jemalloc 或者 tcmalloc，默认是 jemalloc；截图中使用的便是默认的 jemalloc。

进程本身运行需要的内存

Redis 主进程本身运行肯定需要占用内存，如代码、常量池等等；这部分内存大约几兆，在大多数生产环境中与 Redis 数据占用的内存相比可以忽略。这部分内存不是由 jemalloc 分配，因此不会统计在 used_memory 中。

缓冲内存

缓冲内存包括客户端缓冲区、复制积压缓冲区、AOF 缓冲区等；其中，客户端缓冲存储客户端连接的输入输出缓冲；复制积压缓冲用于部分复制功能；AOF 缓冲区用于在进行 AOF 重写时，保存最近的写入命令。在了解相应功能之前，不需要知道这些缓冲的细节;这部分内存由 jemalloc 分配,因此会统计在 used_memory 中。

内存碎片

内存碎片是 Redis 在分配、回收物理内存过程中产生的。例如，如果对数据的更改频繁，而且数据之间的大小相差很大，可能导致 Redis 释放的空间在物理内存中并没有释放，但 Redis 又无法有效利用，这就形成了内存碎片。内存碎片不会统计在 used_memory 中。

Redis 对象有 5 种类型

无论是哪种类型，Redis 都不会直接存储，而是通过 RedisObject 对象进行存储。

Redis 没有直接使用 C 字符串

(即以空字符’\0’结尾的字符数组)作为默认的字符串表示，而是使用了 SDS。SDS 是简单动态字符串(Simple Dynamic String)的缩写。

Reids 主从复制

复制是高可用 Redis 的基础，哨兵和集群都是在复制基础上实现高可用的。复制主要实现了数据的多机备份，以及对于读操作的负载均衡和简单的故障恢复。缺陷：故障恢复无法自动化；写操作无法负载均衡；存储能力受到单机的限制。

Redis 哨兵

在复制的基础上，哨兵实现了自动化的故障恢复。缺陷：写操作无法负载均衡；存储能力受到单机的限制。

Reids 持久化触发条件

RDB 持久化的触发分为手动触发和自动触发两种。

Redis 开启 AOF

Redis 服务器默认开启 RDB，关闭 AOF；要开启 AOF，需要在配置文件中配置：

appendonly yes

AOF 常用配置总结

下面是 AOF 常用的配置项，以及默认值；前面介绍过的这里不再详细介绍。

1、 appendonly no：是否开启 AOF

2、 appendfilename "appendonly.aof"：AOF 文件名

3、 dir ./：RDB 文件和 AOF 文件所在目录

4、 appendfsync everysec：fsync 持久化策略

5、 no-appendfsync-on-rewrite no：AOF 重写期间是否禁止 fsync；如果开启该选项，可以减轻文件重写时 CPU 和硬盘的负载（尤其是硬盘），但是可能会丢失 AOF 重写期间的数据；需要在负载和安全性之间进行平衡

6、 auto-aof-rewrite-percentage 100：文件重写触发条件之一

7、 auto-aof-rewrite-min-size 64mb：文件重写触发提交之一

8、 aof-load-truncated yes：如果 AOF 文件结尾损坏，Redis 启动时是否仍载入 AOF 文件

RDB 和 AOF 的优缺点

RDB 持久化

优点：RDB 文件紧凑，体积小，网络传输快，适合全量复制；恢复速度比 AOF 快很多。当然，与 AOF 相比，RDB 最重要的优点之一是对性能的影响相对较小。

缺点：RDB 文件的致命缺点在于其数据快照的持久化方式决定了必然做不到实时持久化，而在数据越来越重要的今天，数据的大量丢失很多时候是无法接受的，因此 AOF 持久化成为主流。此外，RDB 文件需要满足特定格式，兼容性差（如老版本的 Redis 不兼容新版本的 RDB 文件）。

AOF 持久化

与 RDB 持久化相对应，AOF 的优点在于支持秒级持久化、兼容性好，缺点是文件大、恢复速度慢、对性能影响大。

持久化策略选择

1、 如果 Redis 中的数据完全丢弃也没有关系（如 Redis 完全用作 DB 层数据的 cache），那么无论是单机，还是主从架构，都可以不进行任何持久化。

2、 在单机环境下（对于个人开发者，这种情况可能比较常见），如果可以接受十几分钟或更多的数据丢失，选择 RDB 对 Redis 的性能更加有利；如果只能接受秒级别的数据丢失，应该选择 AOF。

3、 但在多数情况下，我们都会配置主从环境，slave 的存在既可以实现数据的热备，也可以进行读写分离分担 Redis 读请求，以及在 master 宕掉后继续提供服务。

Redis 缓存被击穿处理机制

使用 mutex。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去 load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如 Redis 的 SETNX 或者 Memcache 的 ADD）去 set 一个 mutex key，当操作返回成功时，再进行 load db 的操作并回设缓存；否则，就重试整个 get 缓存的方法

Redis 还提供的高级工具

像慢查询分析、性能测试、Pipeline、事务、Lua 自定义命令、Bitmaps、HyperLogLog、/订阅、Geo 等个性化功能。

Redis 常用管理命令

```
# dbsize 返回当前数据库 key 的数量。
```

info 返回当前 Redis 服务器状态和一些统计信息。
monitor 实时监听并返回 Redis 服务器接收到的所有请求信息。
shutdown 把数据同步保存到磁盘上，并关闭 Redis 服务。
config get parameter 获取一个 Redis 配置参数信息。（个别参数可能无法获取）
config set parameter value 设置一个 Redis 配置参数信息。（个别参数可能无法获取）
config resetstat 重置 info 命令的统计信息。（重置包括：keyspace 命中数、
keyspace 错误数、 处理命令数，接收连接数、过期 key 数）
debug object key 获取一个 key 的调试信息。
debug segfault 制造一次服务器当机。
flushdb 删除当前数据库中所有 key,此方法不会失败。小心慎用
flushall 删除全部数据库中所有 key，此方法不会失败。小心慎用

Reids 工具命令

#Redis-server：Redis 服务器的 daemon 启动程序
#Redis-cli：Redis 命令行操作工具。当然，你也可以用 telnet 根据其纯文本协议来操作
#Redis-benchmark：Redis 性能测试工具，测试 Redis 在你的系统及你的配置下的读写性能
$Redis-benchmark -n 100000 –c 50
#模拟同时由 50 个客户端发送 100000 个 SETs/GETs 查询
#Redis-check-aof：更新日志检查
#Redis-check-dump：本地数据库检查

为什么需要持久化?

由于 Redis 是一种内存型数据库，即服务器在运行时，系统为其分配了一部分内存存储数据，一旦服务器挂了，或者突然宕机了，那么数据库里面的数据将会丢失，为了使服务器即使突然关机也能保存数据，必须通过持久化的方式将数据从内存保存到磁盘中。

判断 key 是否存在

exists key +key 名字

删除 key

```
del key1 key2 ...
```

缓存和数据库间数据一致性问题

分布式环境下（单机就不用说了）非常容易出现缓存和数据库间的数据一致性问题，针对这一点的话，只能说，如果你的项目对缓存的要求是强一致性的，那么请不要使用缓存。我们只能采取合适的策略来降低缓存和数据库间数据不一致的概率，而无法保证两者间的强一致性。合适的策略包括 合适的缓存更新策略，更新数据库后要及时更新缓存、缓存失败时增加重试机制，例如 MQ 模式的消息队列。

布隆过滤器

bloomfilter 就类似于一个 hash set，用于快速判某个元素是否存在于集合中，其典型的应用场景就是快速判断一个 key 是否存在于某容器，不存在就直接返回。布隆过滤器的关键就在于 hash 算法和容器大小

缓存雪崩问题

存在同一时间内大量键过期（失效），接着来的一大波请求瞬间都落在了数据库中导致连接异常。

解决方案：

1、 也是像解决缓存穿透一样加锁排队。

2、 建立备份缓存，缓存 A 和缓存 B，A 设置超时时间，B 不设值超时时间，先从 A 读缓存，A 没有读 B，并且更新 A 缓存和 B 缓存;

缓存并发问题

这里的并发指的是多个 Redis 的 client 同时 set key 引起的并发问题。比较有效的解决方案就是把 Redis.set 操作放在队列中使其串行化，必须的一个一个执行，具体的代码就不上了，当然加锁也是可以的，至于为什么不用 Redis 中的事务，留给各位看官自己思考探究。

Redis 分布式

Redis 支持主从的模式。原则：Master 会将数据同步到 slave，而 slave 不会将数据同步到 master。Slave 启动时会连接 master 来同步数据。

这是一个典型的分布式读写分离模型。我们可以利用 master 来插入数据，slave 提供检索服务。这样可以有效减少单个机器的并发访问数量

读写分离模型

通过增加 Slave DB 的数量，读的性能可以线性增长。为了避免 Master DB 的单点故障，集群一般都会采用两台 Master DB 做双机热备，所以整个集群的读和写的可用性都非常高。读写分离架构的缺陷在于，不管是 Master 还是 Slave，每个节点都必须保存完整的数据，如果在数据量很大的情况下，集群的扩展能力还是受限于单个节点的存储能力，而且对于 Write-intensive 类型的应用，读写分离架构并不适合。

数据分片模型

为了解决读写分离模型的缺陷，可以将数据分片模型应用进来。

可以将每个节点看成都是独立的 master，然后通过业务实现数据分片。

结合上面两种模型，可以将每个 master 设计成由一个 master 和多个 slave 组成的模型。

Redis 常见性能问题和解决方案：

Master 最好不要做任何持久化工作，如 RDB 内存快照和 AOF 日志文件

如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一次

为了主从复制的速度和连接的稳定性，Master 和 Slave 最好在同一个局域网内

尽量避免在压力很大的主库上增加从库

Redis 通讯协议

RESP 是 Redis 客户端和服务端之前使用的一种通讯协议；RESP 的特点：实现简单、快速解析、可读性好

Redis 分布式锁实现

先拿 setnx 来争抢锁，抢到之后，再用 expire 给锁加一个过期时间防止锁忘记了释放。如果在 setnx 之后执行 expire 之前进程意外 crash 或者要重启维护了，那会怎么样？set 指令有非常复杂的参数，这个应该是可以同时把 setnx 和 expire 合成一条指令来用的！

Redis 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。缺点：在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如 rabbitmq 等。能不能生产一次消费多次呢？使用 pub/sub 主题订阅者模式，可以实现 1:N 的消息队列。

Redis 中海量数据的正确操作方式

利用 SCAN 系列命令（SCAN、SSCAN、HSCAN、ZSCAN）完成数据迭代。

SCAN 系列命令注意事项

1、 SCAN 的参数没有 key，因为其迭代对象是 DB 内数据；

2、 返回值都是数组，第一个值都是下一次迭代游标；

3、 时间复杂度：每次请求都是 O(1)，完成所有迭代需要 O(N)，N 是元素数量；

4、 可用版本：version >= 2.8.0；

Redis 管道 Pipeline

在某些场景下我们在**一次操作中可能需要执行多个命令**，而如果我们只是一个命令一个命令去执行则会浪费很多网络消耗时间，如果将命令一次性传输到 Redis 中去再执行，则会减少很多开销时间。但是需要注意的是 pipeline 中的命令并不是原子性执行的，也就是说管道中的命令到达 Redis 服务器的时候可能会被其他的命令穿插

手写一个 LRU 算法

```
class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int CACHE_SIZE;

    /**
     * 传递进来最多能缓存多少数据
     *
     * @param cacheSize 缓存大小
     */
    public LRUCache(int cacheSize) {
        // true 表示让 linkedHashMap 按照访问顺序来进行排序，最近访问的放在头部，最老访问的放在尾部。
        super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);
```

```
        CACHE_SIZE = cacheSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        // 当 map 中的数据量大于指定的缓存个数的时候，就自动删除最老的数据。
        return size() > CACHE_SIZE;
    }
}
```

多节点 Redis 分布式锁：Redlock 算法

获取当前时间（start）。

依次向 N 个 Redis 节点请求锁。请求锁的方式与从单节点 Redis 获取锁的方式一致。为了保证在某个 Redis 节点不可用时该算法能够继续运行，获取锁的操作都需要设置超时时间，需要保证该超时时间远小于锁的有效时间。这样才能保证客户端在向某个 Redis 节点获取锁失败之后，可以立刻尝试下一个节点。

计算获取锁的过程总共消耗多长时间（consumeTime = end - start）。如果客户端从大多数 Redis 节点（>= N/2 + 1）成功获取锁，并且获取锁总时长没有超过锁的有效时间，这种情况下，客户端会认为获取锁成功，否则，获取锁失败。

如果最终获取锁成功，锁的有效时间应该重新设置为锁最初的有效时间减去 consumeTime。

如果最终获取锁失败，客户端应该立刻向所有 Redis 节点发起释放锁的请求。

Redis 中设置过期时间主要通过以下四种方式

1、 expire key seconds：设置 key 在 n 秒后过期；

2、 pexpire key milliseconds：设置 key 在 n 毫秒后过期；

3、 expireat key timestamp：设置 key 在某个时间戳（精确到秒）之后过期；

4、 pexpireat key millisecondsTimestamp：设置 key 在某个时间戳（精确到毫秒）之后过期；

Reids 三种不同删除策略

定时删除：在设置键的过期时间的同时，创建一个定时任务，当键达到过期时间时，立即执行对键的删除操作

惰性删除：放任键过期不管，但在每次从键空间获取键时，都检查取得的键是否过期，如果过期的话，就删除该键，如果没有过期，就返回该键

定期删除：每隔一点时间，程序就对数据库进行一次检查，删除里面的过期键，至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

定时删除

优点：对内存友好，定时删除策略可以保证过期键会尽可能快地被删除，并释放国期间所占用的内存

缺点：对 cpu 时间不友好，在过期键比较多时，删除任务会占用很大一部分 cpu 时间，在内存不紧张但 cpu 时间紧张的情况下，将 cpu 时间用在删除和当前任务无关的过期键上，影响服务器的响应时间和吞吐量

定期删除

由于定时删除会占用太多 cpu 时间，影响服务器的响应时间和吞吐量以及惰性删除浪费太多内存，有内存泄露的危险，所以出现一种整合和折中这两种策略的定期删除策略。

1、 定期删除策略每隔一段时间执行一次删除过期键操作，并通过限制删除操作执行的时长和频率来减少删除操作对 CPU 时间的影响。

2、 定时删除策略有效地减少了因为过期键带来的内存浪费。

惰性删除

优点：对 cpu 时间友好，在每次从键空间获取键时进行过期键检查并是否删除，删除目标也仅限当前处理的键，这个策略不会在其他无关的删除任务上花费任何 cpu 时间。

缺点：对内存不友好，过期键过期也可能不会被删除，导致所占的内存也不会释放。甚至可能会出现内存泄露的现象，当存在很多过期键，而这些过期键又没有被访问到，这会可能导致它们会一直保存在内存中，造成内存泄露。

Redis 常见的几种缓存策略

1、 Cache-Aside

2、 Read-Through

3、 Write-Through

4、 Write-Behind

Redis Module 实现布隆过滤器

Redis module 是 Redis 4.0 以后支持的新的特性，这里很多国外牛逼的大学和机构提供了很多牛逼的 Module 只要编译引入到 Redis 中就能轻松的实现我们某些需求的功能。在 Redis 官方 Module 中有一些我们常见的一些模块，我们在这里就做一个简单的使用。

1、 neural-Redis 主要是神经网络的机器学，集成到 Redis 可以做一些机器训练感兴趣的可以尝试

2、 RedisSearch 主要支持一些富文本的的搜索

3、 RedisBloom 支持分布式环境下的 Bloom 过滤器

Redis 到底是怎么实现“附近的人”

```
GEOADD key longitude latitude member [longitude latitude member ...]
```

将给定的位置对象（纬度、经度、名字）添加到指定的 key。其中，key 为集合名称，member 为该经纬度所对应的对象。在实际运用中，当所需存储的对象数量过多时，可通过设置多 key(如一个省一个 key)的方式对对象集合变相做 sharding，避免单集合数量过多。

成功插入后的返回值：

```
(integer) N
```

其中 N 为成功插入的个数。