序言

Redis是业界普遍应用的缓存组件，研究一个组件框架，最直观的办法就是从应用方的角度出发，将每个步骤的考虑一番，从这些步骤入手去研究往往能够最快的体会到一个组件框架的设计哲学。以Redis为例，每当发起一条请求时，redis是如何管理管理网络请求，收到请求后又是通过什么样的数据结构进行组织并操作内存，这些数据又是如何dump到磁盘实现持久化，再到多机环境下如何同步和保证一致性。

一丶网络模型

Redis是典型的基于Reactor的事件驱动模型，单进程单线程，高效的框架总是类似的。网络模型与spp的异步模型几乎一致。

Redis流程上整体分为接受请求处理器、响应处理器和应答处理器三个同步模块，每一个请求都是要经历这三个部分。

Redis集成了libevent/epoll/kqueue/select等多种事件管理机制，可以根据操作系统版本自由选择合适的管理机制，其中libevent是最优选择的机制。

Redis的网络模型有着所有事件驱动模型的优点，高效低耗。但是面对耗时较长的操作的时候，同样无法处理请求，只能等到事件处理完毕才能响应，之前在业务中也遇到过这样的场景，删除redis中全量的key-value，整个操作时间较长，操作期间所有的请求都无法响应。所以了解清楚网络模型有助于在业务中扬长避短，减少长耗时的请求，尽可能多一些简单的短耗时请求发挥异步模型的最大的威力，事实上在Redis的设计中也多次体现这一点。

二丶数据结构和内存管理

1.字符串

1.1 结构

Redis的字符串是对C语言原始字符串的二次封装，结构如下：

structsdshdr {longlen;longfree;charbuf[];};

可以看出，每当定义一个字符串时，除了保存字符的空间，Redis还分配了额外的空间用于管理属性字段。

1.2 内存管理方式

动态内存管理方式，动态方式最大的好处就是能够较为充分的利用内存空间，减少内存碎片化，与此同时带来的劣势就是容易引起频繁的内存抖动，通常采用“空间预分配”和“惰性空间释放”两种优化策略来减少内存抖动，redis也不例外。

每次修改字符串内容时，首先检查内存空间是否符合要求，否则就扩大2倍或者按M增长；减少字符串内容时，内存并不会立刻回收，而是按需回收。

关于内存管理的优化，最基本的出发点就是浪费一点空间还是牺牲一些时间的权衡，像STL、tcmalloc、protobuf3的arena机制等采用的核心思路都是“预分配迟回收”，Redis也是一样的。

1.3 二进制安全

判断字符串结束与否的标识是len字段，而不是C语言的''，因此是二进制安全的。

放心的将pb序列化后的二进制字符串存入redis。

简而言之，通过redis的简单封装，redis的字符串的操作更加方便，性能更友好，并且屏蔽了C语言字符串的一些需要用户关心的问题。

2.字典（哈希）

字典的底层一定是hash，涉及到hash一定会涉及到hash算法、冲突的解决方法和hash表扩容和缩容。

2.1 hash算法

Redis使用的就是常用的Murmurhash2，Murmurhash算法能够给出在任意输入序列下的散列分布性，并且计算速度很快。之前做共享内存的Local-Cache的需求时也正是利用了Murmurhash的优势，解决了原有结构的hash函数散列分布性差的问题。

2.2 hash冲突解决方法

链地址法解决hash冲突，通用解决方案没什么特殊的。多说一句，如果选用链地址解决冲突，那么势必要有一个散列性非常好的hash函数，否则hash的性能将会大大折扣。Redis选用了Murmurhash，所以可以放心大胆的采用链地址方案。

2.3 hash扩容和缩容

维持hash表在一个合理的负载范围之内，简称为rehash过程。

rehash的过程也是一个权衡的过程，在做评估之前首先明确一点，不管中间采用什么样的rehash策略，rehash在宏观上看一定是：分配一个新的内存块，老数据搬到新的内存块上，释放旧内存块。

老数据何时搬？怎么搬？就变成了一个需要权衡的问题。

第一部分的网络模型上明确的指出Redis的事件驱动模型特点，不适合玩长耗时操作。如果一个hashtable非常大，需要进行扩容就一次性把老数据copy过去，那就会非常耗时，违背事件驱动的特点。所以Redis依旧采用了一种惰性的方案：

新空间分配完毕后，启动rehashidx标识符表明rehash过程的开始；之后所有增删改查涉及的操作时都会将数据迁移到新空间，直到老空间数据大小为0表明数据已经全部在新空间，将rehashidx禁用，表明rehash结束。

将一次性的集中问题分而治之，在Redis的设计哲学中体现的淋漓尽致，主要是为了避免大耗时操作，影响Redis响应客户请求。

3.整数集合

变长整数存储，整数分为16/32/64三个变长尺度，根据存入的数据所属的类型，进行规划。

每次插入新元素都有可能导致尺度升级（例如由16位涨到32位），因此插入整数的时间复杂度为O（n）。这里也是一个权衡，内存空间和时间的一个折中，尽可能节省内存。

4.跳跃表

Redis的skilplist和普通的skiplist没什么不同，都是冗余数据实现的从粗到细的多层次链表，Redis中应用跳表的地方不多，常见的就是有序集合。

Redis的跳表和普通skiplist没有什么特殊之处。

5.链表

Redis的链表是双向非循环链表，拥有表头和表尾指针，对于首尾的操作时间复杂度是O(1)，查找时间复杂度O(n)，插入时间复杂度O(1)。

Redis的链表和普通链表没有什么特殊之处。

三丶AOF和RDB持久化

AOF持久化日志，RDB持久化实体数据，AOF优先级大于RDB。

1.AOF持久化

机制：通过定时事件将aof缓冲区内的数据定时写到磁盘上。

2.AOF重写

为了减少AOF大小，Redis提供了AOF重写功能，这个重写功能做的工作就是创建一个新AOF文件代替老的AOF，并且这个新的AOF文件没有一条冗余指令。（例如对list先插入A/B/C，后删除B/C，再插入D共6条指令，最终状态为A/D，只需1条指令就可以）

实现原理就是读现有数据库的状态，根据状态反推指令，跟之前的AOF无关。同样，为了避免长时间耗时，重写工作放在子进程进行。

3.RDB持久化

SAVE和BGSAVE两个命令都是用于生成RDB文件，区别在于BGSAVE会fork出一个子进程单独进行，不影响Redis处理正常请求。

定时和定次数后进行持久化操作。

简而言之，RDB的过程其实是比较简单的，满足条件后直接去写RDB文件就结束了。

四丶多机和集群

1.主从服务器

避免单点是所有服务的通用问题，Redis也不例外。解决单点就要有备机，有备机就要解决固有的数据同步问题。

1.1 sync——原始版主从同步

Redis最初的同步做法是sync指令，通过sync每次都会全量数据，显然每次都全量复制的设计比较消耗资源。改进思路也是常规逻辑，第一次全量，剩下的增量，这就是现在的psync指令的活。

1.2 psync

部分重同步实现的技术手段是“偏移序号+积压缓冲区”，具体做法如下：

（1）主从分别维护一个seq，主每次完成一个请求便seq+1，从每同步完后更新自己seq；

（2）从每次打算同步时都是携带着自己的seq到主，主将自身的seq与从做差结果与积压缓冲区大小比较，如果小于积压缓冲区大小，直接从积压缓冲区取相应的操作进行部分重同步；

（3）否则说明积压缓冲区不能够cover掉主从不一致的数据，进行全量同步。

本质做法用空间换时间，显然在这里牺牲部分空间换回高效的部分重同步，收益比很大。

2.Sentinel

本质：多主从服务器的Redis系统，多台主从上加了管理监控，以保证系统高可用性。

3.集群

Redis的官方版集群尚未在工业界普及起来，下面主要介绍一下集群的管理体系和运转体系。

2.1 slot-集群单位

集群的数据区由slot组成，每个节点负责的slot是在集群启动时分配的。

2.2 客户请求

客户请求时如果相应数据hash后不属于请求节点所管理的slots，会给客户返回MOVED错误，并给出正确的slots。

从这个层面看，redis的集群还不够友好，集群内部的状态必须由客户感知。

2.3 容灾

主从服务器，从用于备份主，一旦主故障，从代替主。