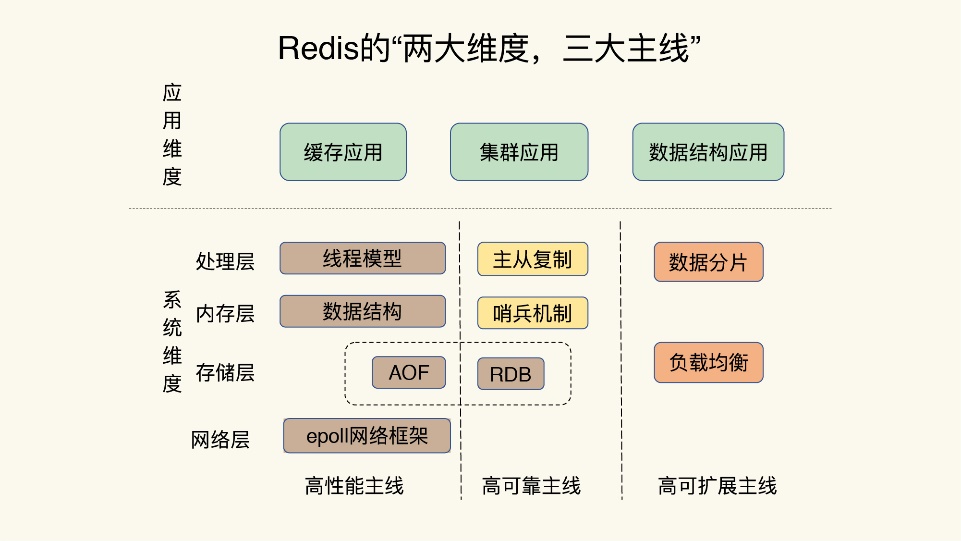
## 开篇部分

* 1. Redis知识全景图： 两大维度，三大主线



两大维度： 系统维度 和 应用维度

系统维度上 连接 Redis关键技术设计原理

应用维度：

三大主线： 高性能，高可靠，高可扩展

高性能：

线程模型，数据结构，持久化，网络框架

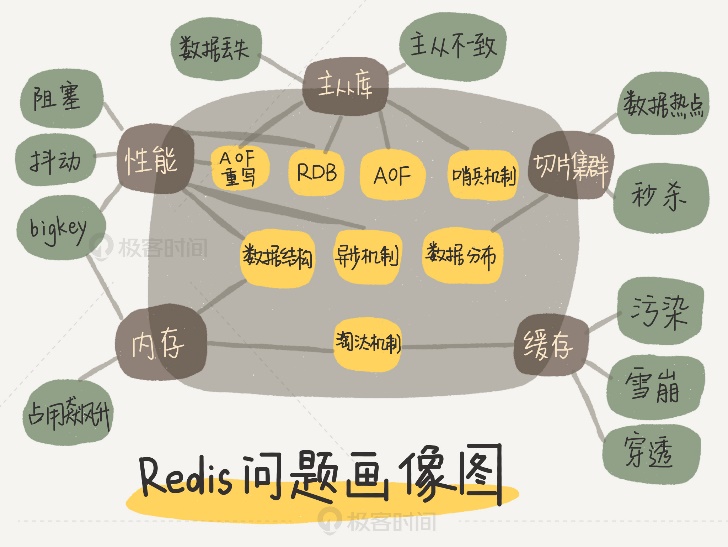
高性能：

主从复制， 哨兵机制

高可扩展：

数据分片，负载均衡

* 1. Redis 的问题画像图：典型问题与技术点



## 基础部分

Redis是典型的键值数据库，非关系性数据库

#### 2.1 基本框架：简易版本的KV数据库SimpeKV设计

##### 2.1.1 数据模型：存什么样的数据

对键值对数据库而言，基本数据类型是 key-value模型， 不同的键值对数据库所支持的key值差异不大，而value支持的类型差别较大， 比如Memcached支持的value的类型只有string， Redis支持的类型包括string，哈希，列表，集合等。在选择键值对数据库时，value支持的类型也成为一个重要的选择。

##### 2.1.2操作接口：对数据可做什么样的操作

SimpleKV是一个简单的kv键值对数据库，基本操作为增删查改，支持三种基本操作： PUT， GET， DELETE。

PUT ：新写入或更新一个 key-value 对

GET ： 根据一个key读取对应的value值

DELETE：根据key值删除整个key-value对

SCAN: 根据一段 key 的范围返回相应的value值

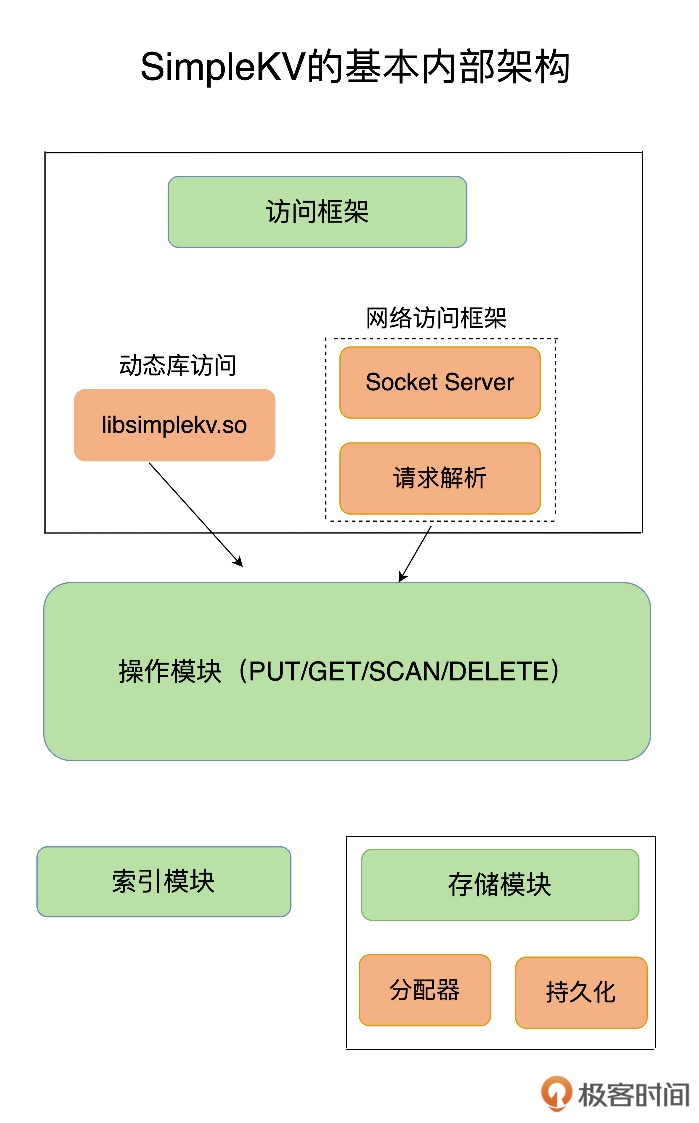
##### 2.1.3 键值对保存位置：内存 or 外存

保存在内存： 读写块，速度约在 百ns级别 但是存在潜在风险，一旦断电，所有数据将丢失

保存在外存： 数据可以避免丢失，但是磁盘读写速度慢(通常几ms级别)，会拉低kv数据库的整体性能。

因此，如何设计选择，通过根据kv数据库的主要应用场景。 Memcached和redis都是属于内存键值数据库。

##### 2.1.4 键值数据库主要部分： 访问框架 、 索引模块、 操作模块 、 存储模块



###### 采用什么访问模式

* + - * 1. 通过函数库调用的方式供外部应用使用（本地端）
        2. 通过网络框架以 socket 通信的形式对外提供键值对访问（Memcached和Redis）

###### 如何定位键值对的位置

SimpleKV依赖于 索引模块 来定位。

索引的作用在于让键值kv数据库根据 key 找到相应的 value 的存储位置，进而执行操作。

索引的类型：哈希表 ， B+树， 字典树， 跳表，不同的索引结构在性能，空间消耗，并发控制具有不同的特征。

哈希表： Memcached和Redis采用 key-value索引，采用哈希表作为索引的原因主要是键值数据保存在内存中，而内存的高性能可以和哈希表的O(1)操作复杂度想匹配。

RoscksDB：采用调表作为内存中的key-value索

###### 不同操作的具体逻辑

GET/SCAN: 根据 value的存储位置返回value的值即可。

PUT 会涉及到内存空间分配

DELETE 操作，SimpleKV 需要删除键值对，需要释放相应的空间，由 分配器 完成。

以上操作都涉及到 SimpleKV的存储模块

###### 存储模块

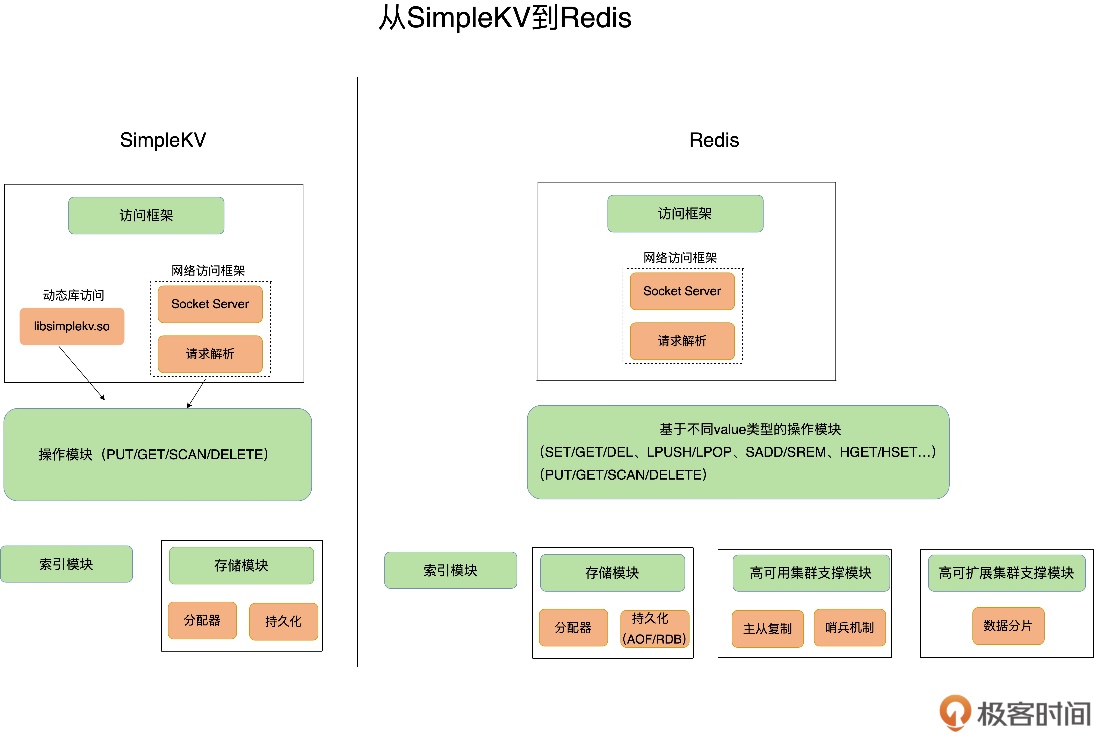
SimpleKV采用内存分配器glibc的malloc和free， 因此，SimpleKV不需要考虑内存管理问题，但是键值数据库的键值对大小通常不一样，而glibc的分配器在处理随机大小的内存块分配时，表现并不好，一旦数据过大，很容易出现严重的内存碎片问题。因此，分配器 是KV数据库的一个关键因数。 Redis 提供了多种内存分配器的选择。

一般而言，都需要重启后快速提供服务 涉及到 持久化

SimpleKV的存储模块增加了持久化功能，这里采用 文件形式，将键值数据通过调用本地文件系统接口保存到磁盘。

SimpleKV两种 持久化 方式：

1. SimpleKV直接对键值对落盘保存(每次都保存)，它的可靠性强，但是每次都要写盘，性能会略低。
2. SimpleKV只是周期性的把内存中的键值保存到文件中，避免频繁写盘操作，但SimpleKV数据有丢失的可能。



SimpleKV 与 Redis对比图

#### 2.2 数据结构： Redis的数据结构

##### 2.2.1 Redis速度快的原因

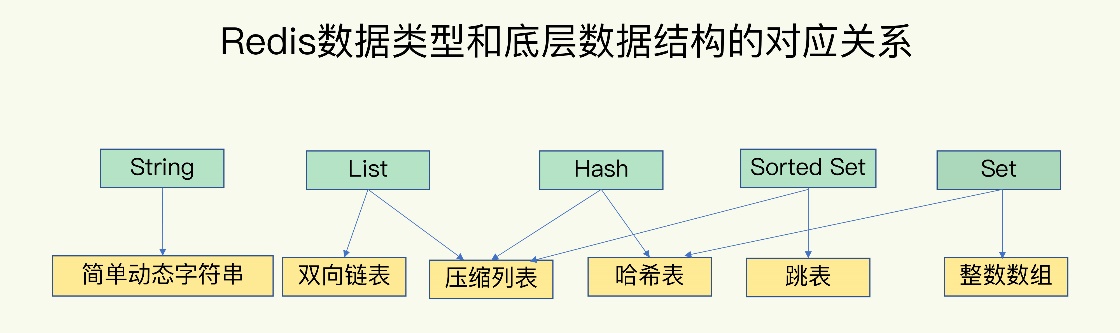
1. Redis是内存数据库，所有操作都是在内存上完成的，而内存访问速度本来就快。
2. Redis自身的数据结构，高效的数据结构是Redis快速处理数据的基础。

##### 2.2.2 Redis数据结构 和 底层数据实现

###### 2.2.2.1 Redis数据类型(5种)

1. String(字符串) 底层是 sds
2. List(列表) 底层是 ziplist 和 doublelist
3. Hash(哈希) 底层是 ziplist 和 hashtable
4. Set(集合) 底层是 ziplist 和 intset
5. Sorted Set(有序集合) 底层是 ziplist 和 skiplist

其中 list，hash，Set 和 SortSet统称为 集合类型，特点是一个键对应一个集合的数据

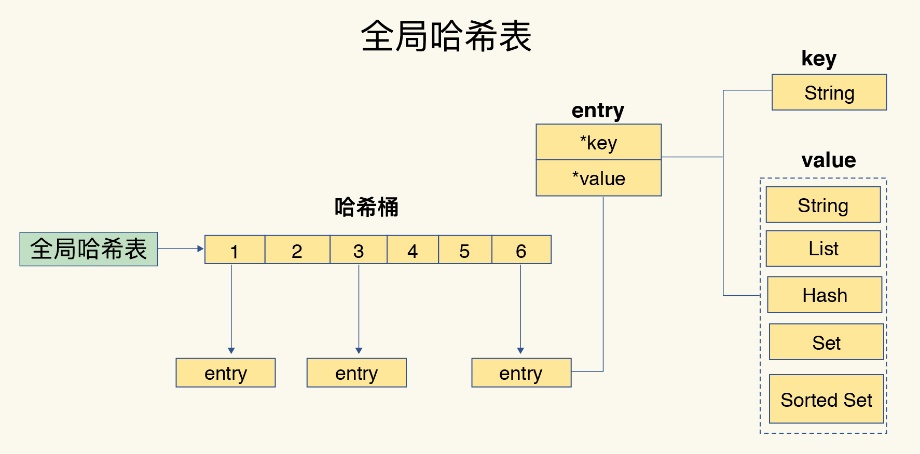


###### 2.2.2.2 Redis底层数据结构(6种)

1. 简单动态字符串 simple dynamic string
2. 双向链表 doublelist O(N)
3. 压缩列表 ziplist O(N)
4. 哈希表 hashtable O(1)
5. 跳表 sliplist O(logN)
6. 整数数组 inset O(N)

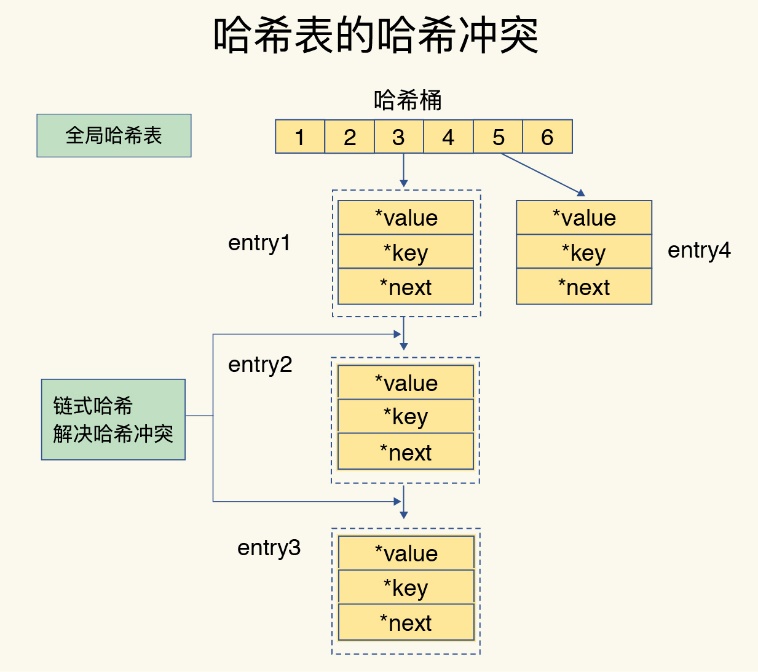
###### 2.2.2.3 键和值组织结构：

Redis使用了一个 哈希表 来保存所有的键值对： 一个哈希表，实际上就是一个数组，数组的每个元素称位一个哈希桶，也即是哈希表由多个哈希桶组成，每个哈希桶保存了键值对数据，哈希桶中的元素保存的并不是值本身，而是指向值的指针。即不管是string还是集合，哈希桶中的元素都是指向他们的指针。



##### 2.2.3 Redis中哈希冲突和rehash

Redis中，采用 链地址法(链式哈希) 来处理哈希冲突：同一个哈希桶中的多个元素用一个链表来保存，它们之间依次用指针连接



若哈希填充过多，就可能导致某些哈希冲突链过长，进而导致链表上元素查找过长，效率降低，从而影响效率。因此，Redis会对哈希表做rehash 操作，rehash也即是增加现有哈希桶的数量，逐个让增多的entry元素能够在更多的桶之间分散保存。

Redis默认使用了两个全局哈希表：哈希表1 和 哈希表2 ；一开始，当插入数据时，默认使用哈希表1，此时哈希表2没有被分配空间，随着数据逐步增多，Redis开始执行rehash。主要分为三步：

1. 给哈希表2分配更大的空间， 比如时哈希表1大小的两倍
2. 把哈希表1的数据重新映射并拷贝到哈希表2中（涉及到大量数据的拷贝，若一次性拷贝则可能导致Redis线程拥塞，为避免此问题，Redis采用了渐进式rehash）
3. 释放哈希表1的空间。

##### 2.2.4 集合数据操作效率

###### 2.2.4.1 Redis集合数据

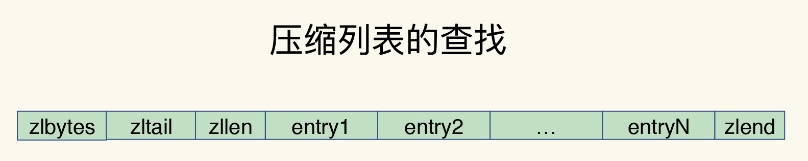
* + - * 1. 对于集合类型的value，第一步时通过全局哈希表找到对应的哈希桶位置，第二步时在集合中再增删改查。
        2. 集合操作的效率与集合首先是底层数据结构相关，比如使用哈希表实现的集合比用链表实现的集合的访问效率要高，其次是这些操作效率与操作本身有关，比如读写一个元素比读写所有元素的操作效率要高。

###### 2.2.4.2 Redis集合底层数据结构

1. 主要有 5 种， 整数数组(intset), 双向链表(doublelist), 哈希表(hashtable), 压缩列表(ziplist), 跳表(skiplist)

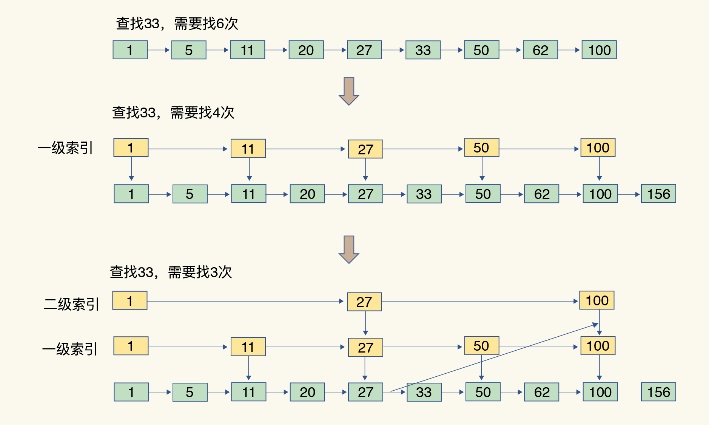


1. 压缩列表(ziplist)类似一个数组，数组中每个元素对应保存一个数据，每个数据有三个字段 zlbytes， zltail， zllen，分别表示列表长度，列表尾，列表中entry个数；压缩列表在表尾还有一个zlend，表示列表结束。



也即是 表尾和表头的查找为O(1)，其余元素O(N)

1. 跳表(ziplist)是在链表基础上，增加了多级索引，通过索引位置的跳转，实现数据的快速定位，跳表的查找复杂度是O(lonN)



跳表的查找过程

每日一问:整数数组(intset)和压缩列表(ziplist)在查找时间复杂度方面并没有太大优势，那为什么Redis还会把它们作为底层数据结构呢？

两个原因：

1. 内存利用率，数组和压缩列表都是非常紧凑的数据结构，它们占用内存比链表更少。Redis是内存数据库，大量的数据存储在内存中，因此需要尽可能的优化，提高内存利用率。
2. 数组对CPU高速缓存支持更加友好，所以Redis在设计时，集合数据较少情况下，默认采用内存紧凑排列的方式进行存储，同时利用CPU高速缓存不会降低访问速度。当元素超过设定的阈值后，避免查询时间复杂度过高，转为哈希和跳表数据结构存储，保证查询效率。

#### 2.3 高性能IO模型 ： 为什么单线程Redis能够那么快？

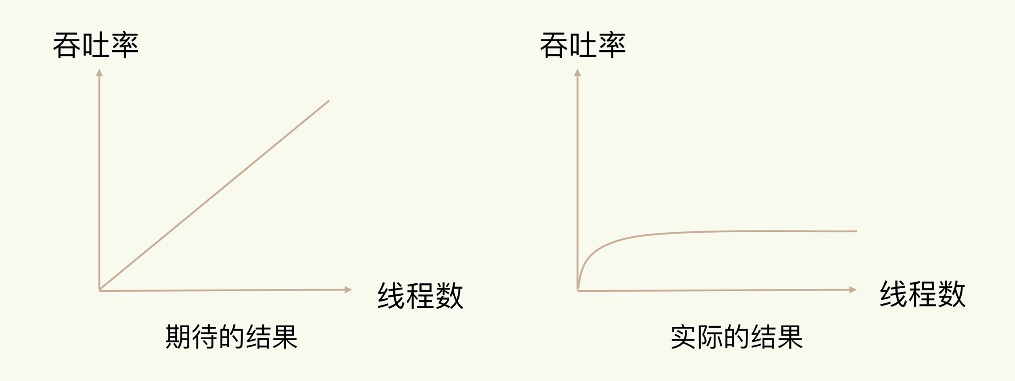
##### 2.3.1 Redis真的是单线程吗？

通常所说的 Redis单线程 主要是指Redis的网IO和键值对读写是由一个线程来完成的，这也是Redis对外提供键值存储服务的主要线程，其他功能，如持久化，异步删除，集群数据同步等，其实是由额外的线程执行的。

##### 2.3.2 Redis 为什么用单线程

###### 2.3.2.1 多线程的开销

1. 多线程可以增加系统吞吐率或增加系统可扩展性，在合理的资源分配情况下，可以增加系统中处理请求操作的的资源实体，进而提升系统能够同时处理的请求数，即吞吐率。



线程数与系统吞吐率

1. 原因在于：系统中通常存在被多线程访问的共享资源，比如临界资源。当多个线程需要修改访问临界资源时，为了保证共享资源的正确性，就需要额外的机制进行保证(比如锁)，而这个额外的机制，会带来额外的开销(加解锁的开销)。
2. 多线程编程模式面临的共享资源的并发访问控制问题。
3. 并发访问控制是多线程编程开发的难点，如果没有精细的设计，比如只是简单采用粗粒度的互斥锁，往往会导致不理想的结果，即使增加了线程，大部分线程也在等待获取访问共享资源的互斥锁，导致并行变串行，使得系统吞吐率没有随着线程的增加而增加。
4. 多线程开发中一般都会引入 同步原语 来保护共享资源的并发访问， 但这会降低易调试性和可维护性。

**为避免以上这些问题，Redis直接采用了单线程模式。**

##### 2.3.3 单线程Redis为什么那么快(每秒十万级别的处理能力)

* + 1. Redis的大部分操作直接在内存中完成，而内存的速度本来就快。
    2. Redis的高效数据结构，Redis采用全局哈希表，通过key值可以很快获取value的位置。比如哈希表和跳表。
    3. Redis采用了**多路复用机制**，使其在网络IO操作中能够并发处理大量的客户请求，实现高吞吐率。

##### 2.3.4基于多路复用的高性能 I/O 模型

###### 2.3.4.1 基本IO模型与阻塞点

## 实践部分

## 加餐部分