Redis(Remote Dictionary Server)，远程字典服务，每秒10万次读写操作。

Redis是一个key-value database，是nosql的数据库。与其类似的还有HBase，MonoDB等。

关系型数据库如mysql通常用于处理复杂的数据，效率不够高。nosql的数据库就应运而生，用来存取一些简单的数据，非常适合web服务。

Redis可以将数据持久化到硬盘上，不过很多时候还是把Redis当成cache来用。

同样的机器上，Memcached可以达到百万qps，Redis可以达到几十万qps，mysql只能达到1kqps。

Redis的DB是用0,1,2等表示，不像mysql需要给数据库取一个名字，例如本项目就是twitter。

Redis中，如果key不存在，执行lpush或rpush，它就会针对key创建了一个空列表，并且把value添加到列表中，如果执行set，就会存储一个普通的字符串。

Redis可以通过配置设置其最大占用内存大小。

Redis热点缓存导致缓存穿透。

Redis的五种基本数据结构：

1. 字符串：可用于存储图片，视频，还可以用作计算器，也可以用作分布式锁。

默认最大存储值为512M。set key value, get key等操作，redis使用sds(simple dynamic string，简单动态字符串)来实现字符串，它采用预分配冗余空间的方式来减少内存频繁分配。

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr5 {

unsigned char flags; /\* 3 lsb of type, and 5 msb of string length \*/

char buf[];

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr8 {

uint8\_t len; /\* used \*/

uint8\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr16 {

uint16\_t len; /\* used \*/

uint16\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr32 {

uint32\_t len; /\* used \*/

uint32\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr64 {

uint64\_t len; /\* used \*/

uint64\_t alloc; /\* excluding the header and null terminator \*/

unsigned char flags; /\* 3 lsb of type, 5 unused bits \*/

char buf[];

};

其中，\_\_attribute\_\_ ((\_\_packed\_\_))是告诉编译器取消字节对齐，则结构体的大小就是结构体成员

结构体中的成员：

len：目前字符串的长度，不包含终止字符\0

alloc：分配的总长度，不包含sds头部(就是len，alloc本身，flags的组合)和尾部的\0，这里的\0是尾部隐含的\0，在buf数组中可以包含多个\0，sds以len来判断是否到达字符串尾部了，所以sds是二进制安全的。C语言中的char\*不是二进制安全的，它碰到\0会终止。

flags：标志位，表示sds头类型，有sdshdr5,sdshdr8,sdshdr16,sdshdr32,sdshdr64共五种类型，其中sdshdr5不用。

buf：存储字符串的字符数组

使用sds，定义如下：

sds s

sds是一个char\*，指向char[] buf。

由于sds取消了字节对齐，所以s[-1]其实指向的就是flags

C/C++其实支持负数下标，不过除非像sds这样肯定之前的地址存在有效的值，否则不要这样用。

空间的预分配：当sds的len增加addlen(如果创建一个新的sds，sdsnewlen()方法，不会进行空间预分配)时，如果sds空间足够，则什么都不干；否则就会触发空间预分配。新的长度newlen=len+addlen，如果newlen小于1M，则分配2\*newlen的大小，如果大于1M，则分配newlen+1M的大小

惰性空间释放：

void sdsclear(sds s) {

sdssetlen(s, 0);

s[0] = '\0';

}

从上述代码可以看出，清空sds时，不会真正释放其内存，而是将len设置为0，并且将buf数组第一个字符设置为\0就行了。这样再次使用该sds时，就不用重新分配内存空间了。

sds相比C语言中的char\*的优势：

1. O(1)的时间复杂度获取字符串长度，通过len来获取
2. 杜绝缓冲区溢出。char\*包括char[]不记录自身的长度以及空闲的空间，在进行拼接操作时，如果不计算好长度，容易溢出；而sds可以通过len和alloc进行判断空间是否足够，不够就进行预分配。
3. 减少字符串修改时带来的内存重新分配次数

char\*字符串修改时，无论增长还是缩短，都会重新分配内存，但sds不需要，它采用预分配机制和惰性空间释放(redis中也有真正释放内存空间的操作)

1. 二进制安全，可以存储任何二进制数据(包括图片，视频等)

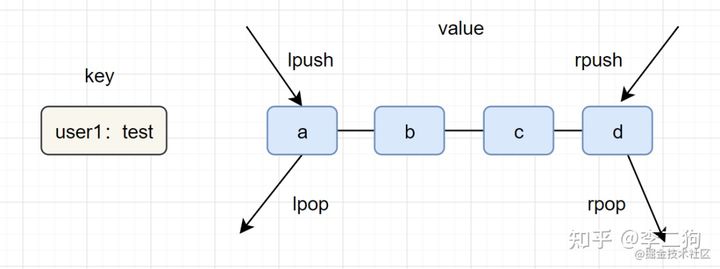
sds的缺点可能是操作比char\*要复杂。

1. 哈希表：键值对，类似python中的字典，可以用来存放用户的信息等。

操作有hset key field value, hget key field，有点类似一个key-key-value。最大长度为2^32这个量级。

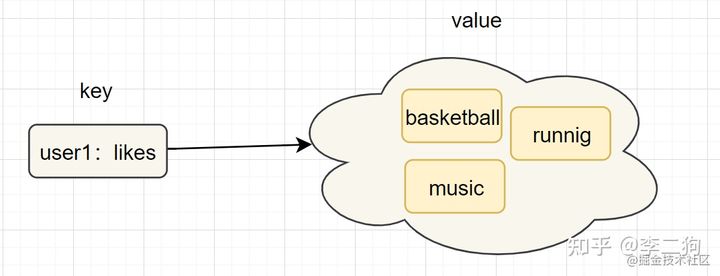
1. List：可用于redis的分页功能

有点类似python中的list，是一个双向的链表。最大长度为2^32-1



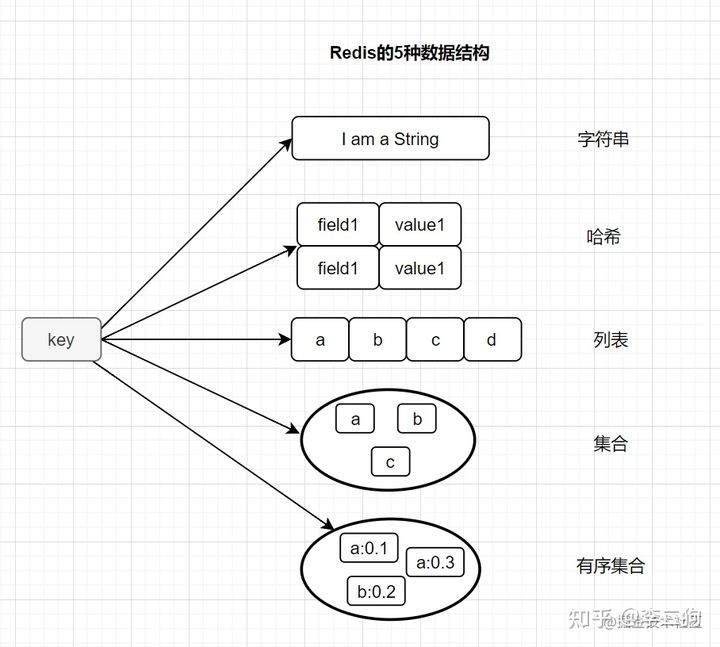
1. Set(集合)：底层是hashtable，查找速度快，可用户全局去重，判断用户是否注册。可以利用Set的交集，并集，差集去判断共同的喜好，全部的喜好，独有的喜好等。

Set是无序的字符串集合，不允许重复元素。sadd myset 1 2 3，在集合myset中插入3个元素。smembers myset可以查看集合元素。



1. Zset(有序集合)：可用于范围查找，排行榜应用，取TOP N等。

已排序的字符串结合，元素也不能重复。



三种特殊额数据结构：

1. Bitmap：

它不是实际的数据类型，用bit来映射某个元素的状态。底层是基于String类型实现的，字符串的最大值为512M，总共有2^32个bits。

1. HyperLogLog：

是一种用来统计不同事物的概率统计数据结构。

1. Geo：地理位置定位，用于存储地理位置信息，并对存储的信息进行操作

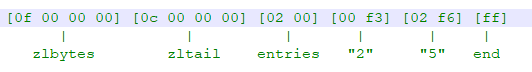
Redis的超时时间可以设置为秒或毫秒，它的精度是1毫秒。

Redis的key最大长度为512M，但官方文档上推荐别太长，1024个字节就是个坏主意。

Redis中的内部数据结构：

1. sds，简单动态字符串，String和Bitmap都是基于sds实现的。
2. ziplist，压缩链表，redis3.2之前是list和hash的底层实现之一，redis3.2之后是zset的底层实现之一，之所以是之一，是因为当数据量较大或元素较长时，ziplist会被转换为其它数据结构。

ziplist像一个内存连续的数组，但每个元素的长度不同。相比较而言，C++中的list内存肯定是不连续的。python中的list存储的是对象的引用(C语言中的指针)，如果list存储的元素是相同的，则元素的地址看起来是连续的，如果不同，则不是连续的；它的元素对应的对象的地址则是不连续。



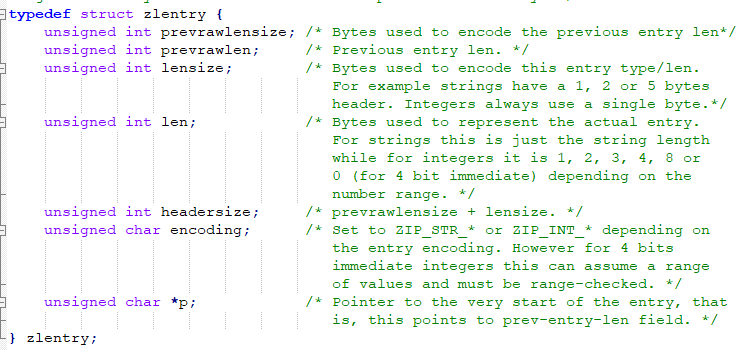
zlbytes：存储整个ziplist占用的内存字节数，上图中总共有15个字节，0f表示一个字节，十六进制显示，对应的是15。

zltail：entries中尾结点的偏移量，上图中对应的是结点”5”，它前面有12个字节，所以记录的偏移量为12.通过这个变量，可以在不遍历整个ziplist的情况下，弹出尾结点。

zlen：结点的数量。

zlend：固定为ff，一个字节，表示ziplist的末端。

ziplist中的结点entry



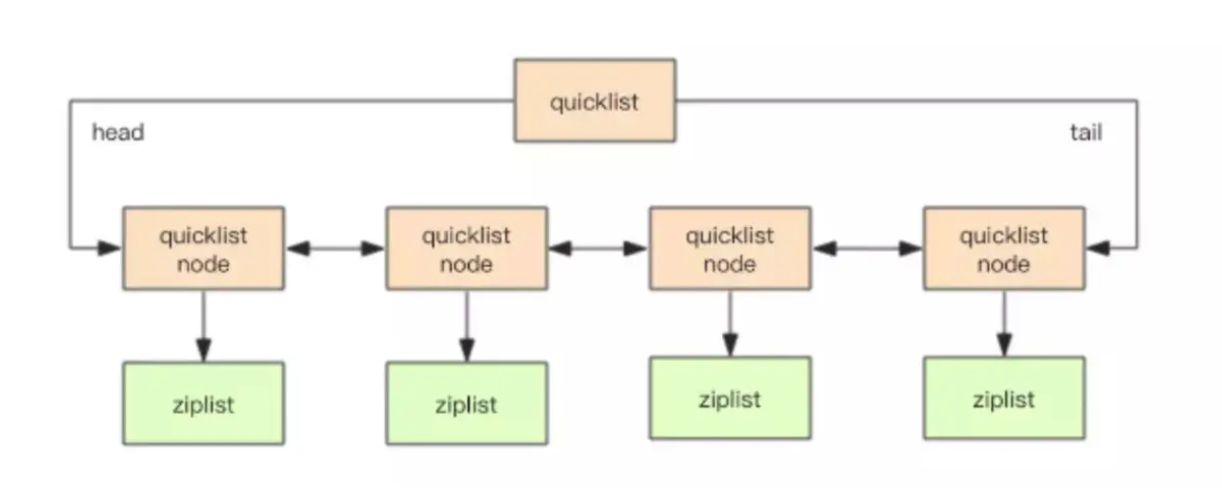
ziplist的缺点：

每次插入或删除一个元素时，都需要进行频繁的调用realloc()函数进行内存的扩展或减小，然后进行数据”搬移”，甚至可能引发连锁更新，造成严重效率的损失。

连锁更新：当插入新结点时，需要更新新结点的后继结点中保存前驱结点(也就是新结点)的长度的值prerawlen，如果新结点的长度小于254字节，则prerawlen的字节长度为1，否则它的字节长度为5，假如后继结点之前的prerawlen的字节长度为1，而插入新结点后，需要扩展为4(还有一种缩小的情况，但redis不处理，以防止频繁扩展或缩小产生的抖动)，如果后继结点以及其之后的结点的长度都是250-253之间，那么当后继结点扩容后，其之后的结点也会跟着扩容，毕竟后继结点的长度肯定会大于254了，从而导致连锁反应。

1. quicklist：redis3.2之后，列表的底层实现

quicklist是由ziplist组成的双端链表，每个结点都包含一个ziplist。



quicklist具有双向链表的优点，两段插入删除的时间复杂度为O(1)，中间插入删除结点的时间复杂度为O(n)，但不需要内存的复制。

1. dict，也就是hashtable，哈希表的底层实现就是dict，哈希冲突采用链表法。
2. skiplist，跳跃表，是实现有序集合zset的底层数据机构之一(元素数量较多或元素中字符串较长时使用skiplist，否则使用ziplist)。redis独有的数据结构，可以实现平均时间复杂度O(logn)，最坏时间复杂度O(n)的结点查找。

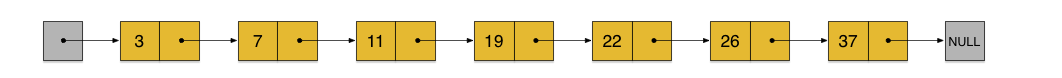
skiplist是概率型数据结构，是用来替代平衡查找树的，例如红黑树，B+树等。

平衡查找树的增删查操作都是O(logn)的，但它的缺点是：自平衡过程比较复杂，实现起来麻烦，加锁的开销会比较大。

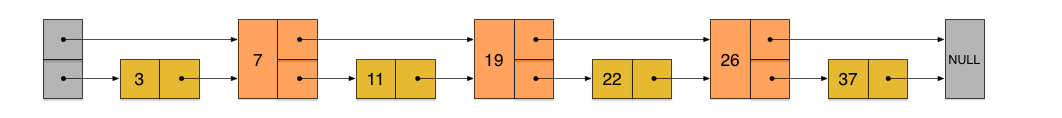
skiplist是一种基于有序链表的扩展。普通的有序链表在查询时不支持二分查找，原因是无法像数组那样很快获得middle元素，也没法快速获取链表长度。链表如果从头结点访问某个中间结点，需要一步一步挪过去，这样时间复杂度就是O(n)，就不需要二分了，直接逐个比较。

skiplist(跳跃表)的基本原理：

有序的链表：查找，插入，删除的时间复杂度都是O(n)。



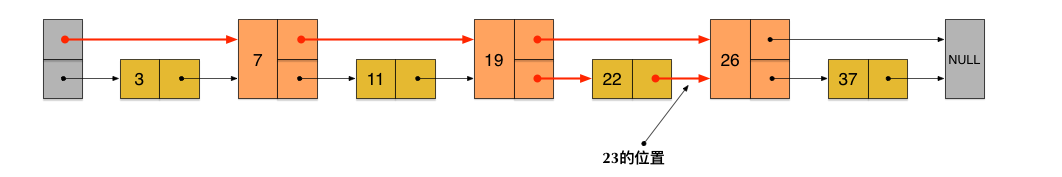
假如我们每相邻两个结点增加一个指针，指向其下下个结点：



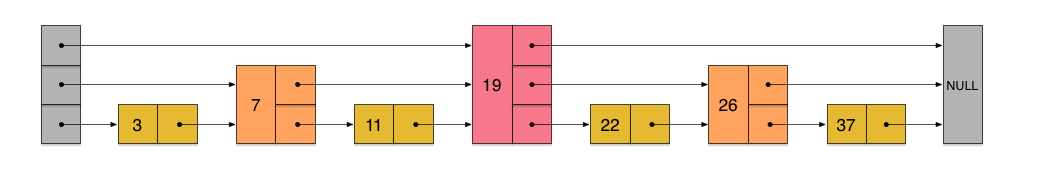
新增加的指针形成了一个新的链表，包含的结点数只有原来的一半(上图中的7,19,26)。查找或插入数据时，可以先在这个新的链表上查找，碰到比待查找数据大的情况，再回到原来的链表中查找，例：插入23，红色的箭头路径。

23先和7比较，大于7，再和19比较，之后再和26比较，23比26小，所以要回到原链表中，返回时从结点19回到原链表，然后和22比较，23比22大，继续向后比较，碰到了26,比它小，所以23要插入到22和26之间。

在比较过程中，由于有新增的链表，所以没有和原链表的结点进行逐个比较，大约只比较了一半。



利用同样的方式，可以再产生一层新的链表：

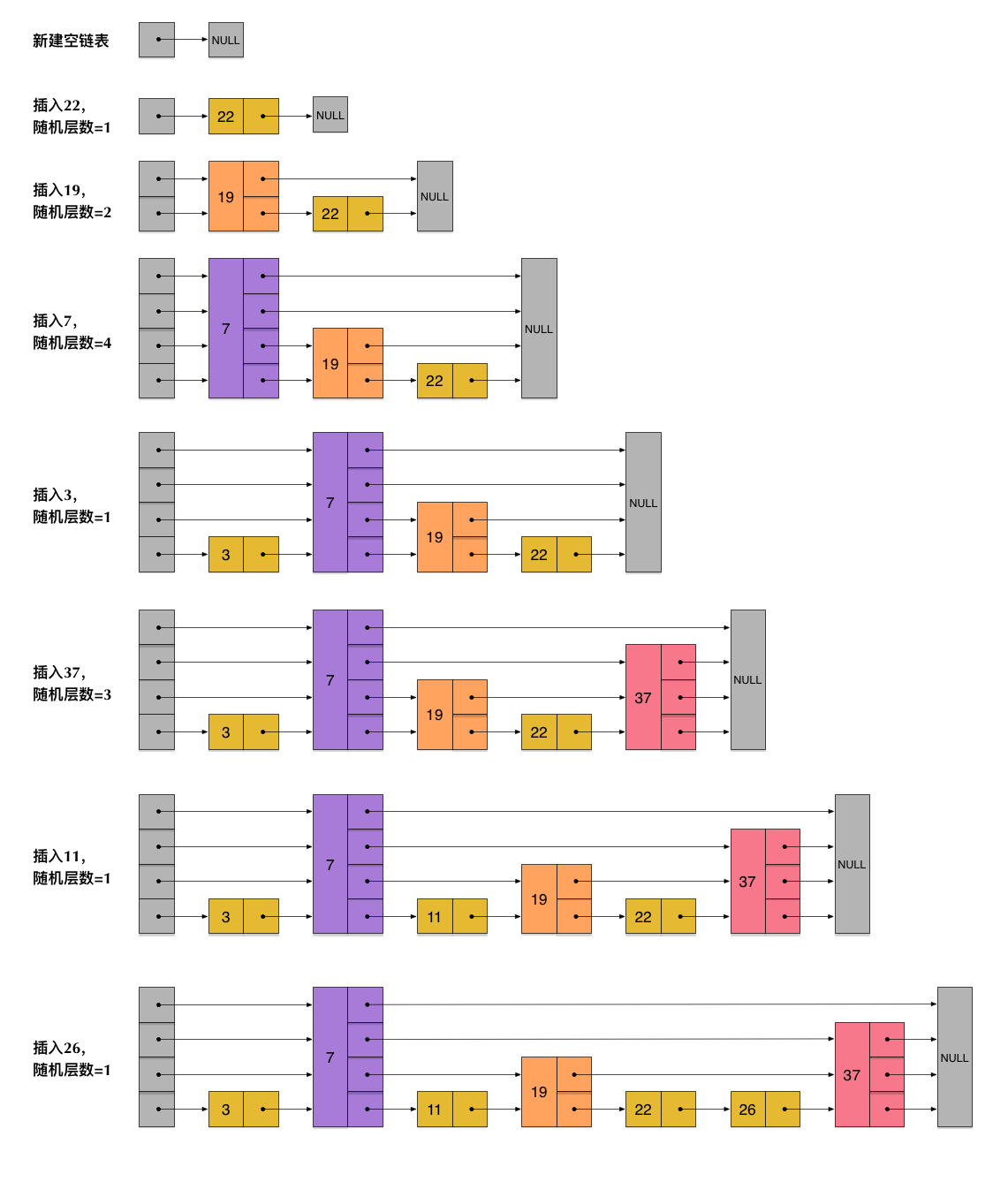


如果还是插入23，会发现比较的次数会进一步减少。可以想象，当链表足够长的时候，这种多层链表的查找方式能让我们跳过很多下层节点，大大加快查找的速度。

redis中的skiplist就是受到上述思想的启发产生的，但上述的方法会有一个比较大的问题：

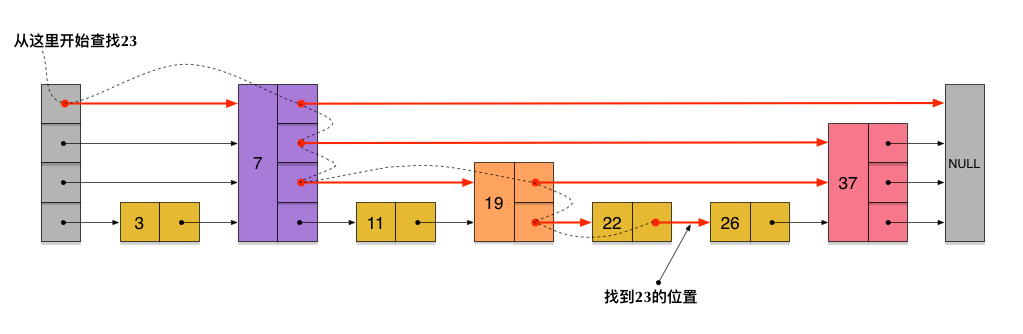
新插入一个节点之后，就会打乱上下相邻两层链表上节点个数严格的2:1的对应关系。如果要维持这种对应关系，就必须把新插入的节点后面的所有节点（也包括新插入的节点）重新进行调整，这会让时间复杂度重新蜕化成O(n)。删除数据也有同样的问题。

skiplist为了避免这种问题，不要求相邻两层链表之间有严格的对应关系。而是为每个结点随机出一个层数level，比如一个结点随机出的层数是3，那么就把它链入到第1层到第3层这3层链表中。



从上图skiplist的创建和插入来看，每一个节点的层数（level）是随机出来的，而且新插入一个节点不会影响其它节点的层数。因此，插入操作只需要修改插入节点前后的指针，而不需要对很多节点都进行调整。这就降低了插入操作的复杂度。

从上图创建的链表中，依然插入23：



从第4层头结点开始，先和7比较，23大于7，之后第4层没有结点了，跳回第3层和结点7之后的37比较，23比37小，跳回第2层的结点7，和7之后的19比较，23大于19，和19之后的37比较，23小于37，跳回第1层的结点19，和19之后的22比较，23大于22，和之后的26比较，23小于26，找到了插入的位置。

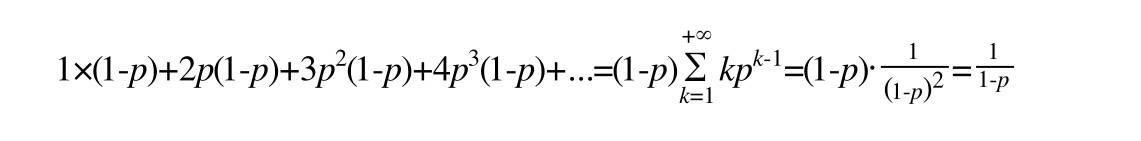
Redis中计算随机数是一个重要的操作。它的要求如下：

1. 每个结点都有第1层指针，即每个结点都在第1层链表中
2. 结点层数最大不允许超过32
3. 如果一个结点有第i层(i>=1)指针（即结点已经在第1层到第i层链表中），那么它有第(i+1)层指针的概率为p，redis中p为1/4.

根据上述的要求，可以得出下面结论：

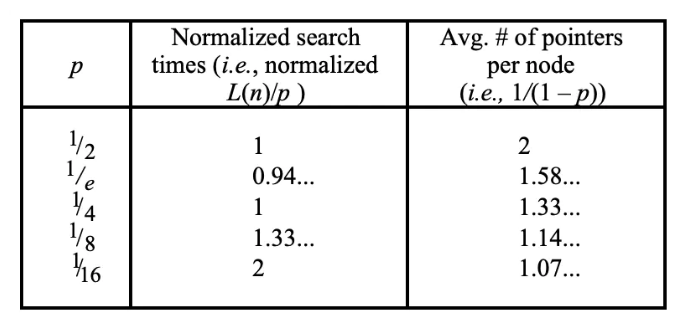
1. 结点的层数恰好为1的概率为1-p，也就是没有i+1层指针的概率。
2. 结点层数恰好为2的概率为p(1-p)，其中p代表有第2层的概率，1-p表示没有第i+1层指针的概率
3. 依次类推，结点层数恰好为3的概率为。p^2表示既有第2层又有第层的概率，1-表示没有第i+1层指针的概率。

因此，一个结点的平均层数为：

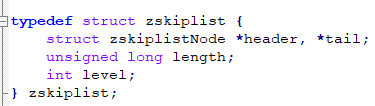


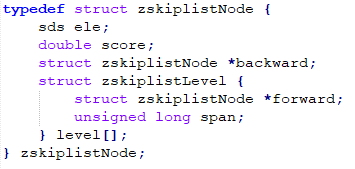
p=1/2时，结点平均层数(也就是每个结点平均包含的指针数)为2，当p=1/4时，结点平均层数为1.33，相比1/2，redis的内存开销要小不少。

下表是p的取值，以及对应的归一化搜索次数和每个结点的平均包含的指针数，可以看出综合来看，p=1/4时的，两个指标都比较合理。



Redis中的zset使用的skiplist：

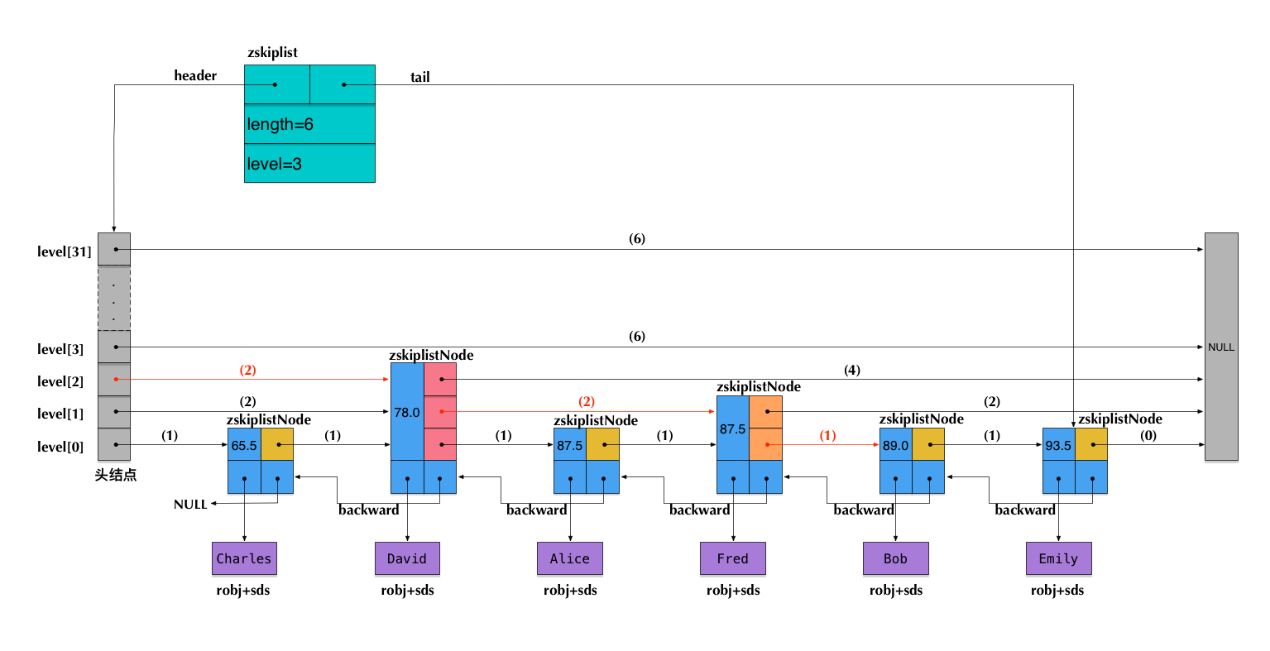




Redis中zskiplistNode存储的元素变成了sds。

Redis中的skiplist最大为32层，可以满足2^64个元素。

一个实际的skiplist：



skiplist与红黑树相比：

缺点：

1. 占用的内存空间更大，每个结点的大小取决于该结点的层数
2. 速度应该比红黑树慢，最坏情况下就是O(n)，而红黑树都是O(logn)

优点：

1. 实现比红黑树简单
2. 红黑树插入删除时为了平衡高度需要旋转附近节点，高并发时需要锁。而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速，不需要考虑加锁的情况。
3. 比红黑树更容易扩展，redis作者自己说的。

Redis速度快的原因：

1. 内存中操作
2. 设计巧妙的数据结构

Redis中主要的事件：文件事件和时间事件

文件事件：主要是网络I/O的读写，请求(客户端发的命令)的接收和回复。

时间事件：单次/多次执行的定时器，如主从复制、定时删除过期数据、字典rehash等

Redis中的单线程： 6.0版本之前

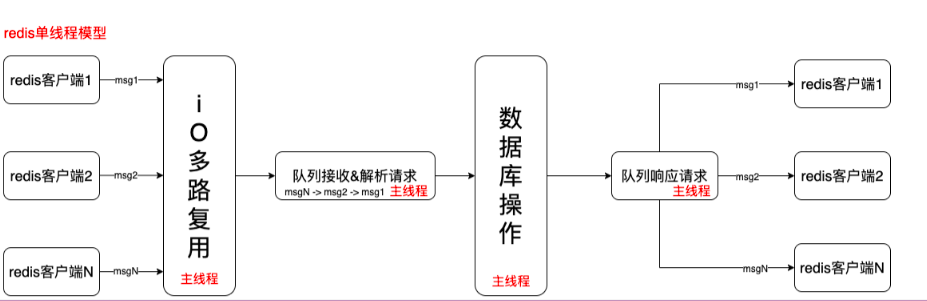
主要是指 Redis 的网络IO和命令执行是由一个线程来完成的，这也是 Redis 对外提供键值存储服务的主要流程。但 Redis 的其他功能，比如持久化、异步删除、集群数据同步等，其实是由额外的线程执行的

在Redis的单线程下，由于这两种事件在同一个线程中执行，就会出现互相影响的问题，如时间事件到了还在等待/执行文件事件，或者文件事件已经就绪却在执行时间事件，这就是单线程的缺点，所以在实现上要将这些影响降到最低。redis其实要保证每个事件任务的处理时间不能太长，否则就会相互影响。

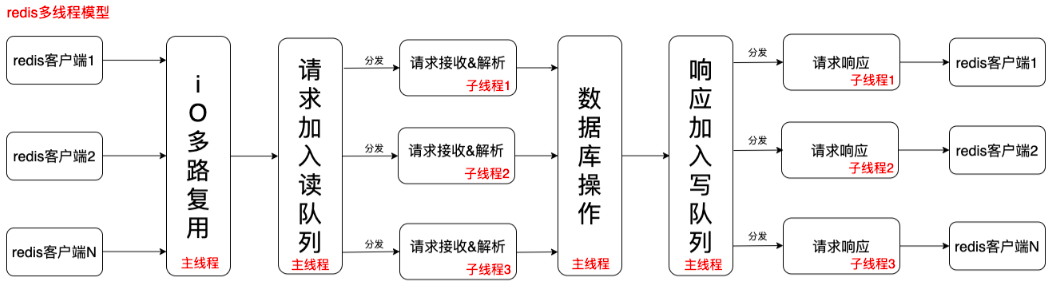
Redis请求的执行流程如下：

1. 服务启动，开始网络端口监听，等待客户端请求
2. 客户端向服务端发起连接请求，创建客户端连接对象，完成连接
3. 将socket信息注册到epoll，设置超时时间为时间事件的周期时长(这个时间事件就是最近需要执行的，是从一个链表遍历取到的)，等待客户端发起请求
4. 客户端发起操作数据库的请求(如GET)
5. epoll收到客户端的请求，可能多个，按顺序处理请求
6. 接收请求参数，解析请求协议得到请求命令
7. 执行请求命令，操作redis数据库
8. 将结果返回给客户端

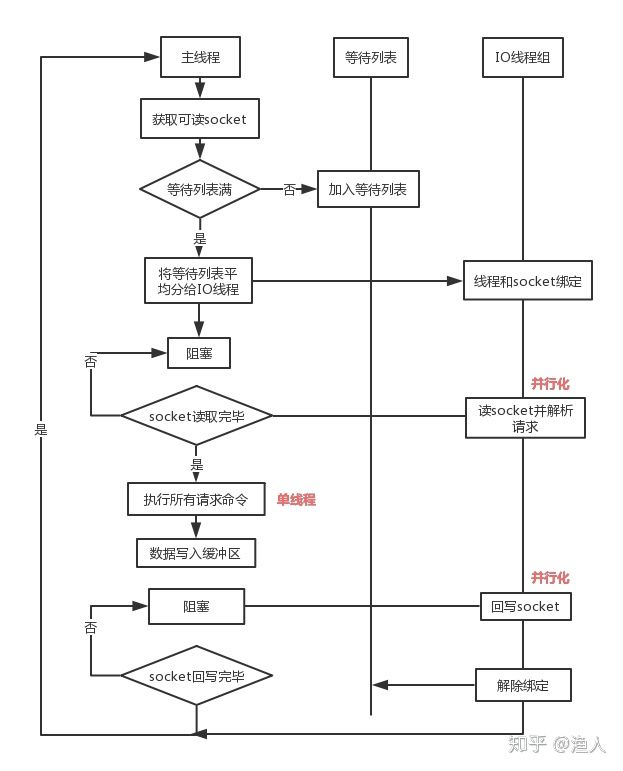
Redis的单线程模型：redis 6.0之前的版本



Redis多线程模型：



多线程相比于单线程，核心思路是：将主线程的IO读写任务拆分出来给一组独立的线程执行，使得多个socket的读写可以并行化，而命令的操作仍然是单线程的，都是在主线程中执行。



核心流程如下：

1. 主线程获取socket放入等待列表
2. 将socket分配给各个IO线程
3. 主线程阻塞等待IO线程读取socket完毕
4. 主线程执行命令-单线程
5. 主线程阻塞等待IO线程将数据回写socket完毕
6. 解除绑定，清空等待队列

I/O多线程的特点：

1. IO线程要么同时在读socket，要么同时在写，不会同时读或写
2. IO线程只负责读写socket解析命令，不负责命令处理
3. IO线程数可自行配置（目前代码限制上限为512，默认为1(关闭此功能)）

多线程貌似没有用到锁，主线程执行请求入队列 -> I/O线程并行进行网络读 -> 主线程串行执行Redis命令 -> I/O线程并行进行网络写 -> 主线程清空队列，并接收下一批请求。这样以来，主线程和I/O线程不会出现同时访问请求队列的清空，就不需要锁了。

单线程的优点：

避免了上下文切换，也不用去考虑各种锁的问题，不存在加锁释放锁操作，没有死锁问题导致的性能消耗。

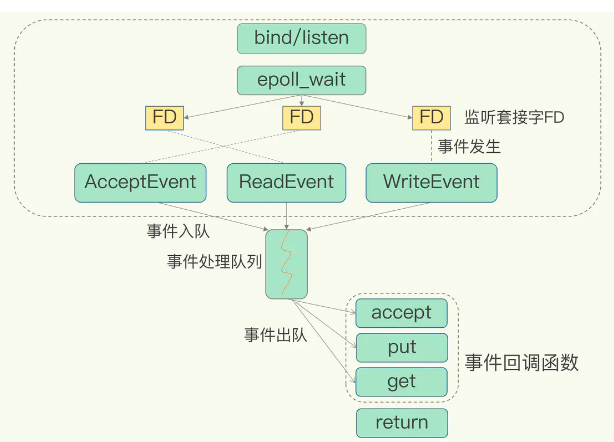
缺点：无法利用多CPU的性能

多线程的优点：

提高了响应速度，充分利用了多cpu的特点

缺点：代码复杂性提高。

I/O多路复用：非阻塞的I/O



redis的压力测试：redis-benchmark

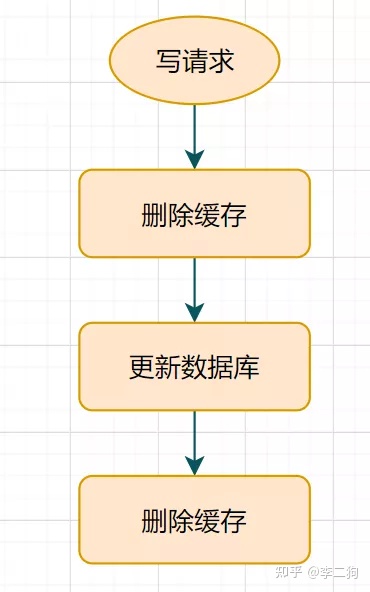
Redis的常用场景：

1. 缓存，应该是最常用的
2. 计数器，redis提供了相应的+1和-1的原子操作
3. 分布式锁
4. 排行榜，利用zset
5. 消息队列
6. 社交网络，其实也是缓存的功能。

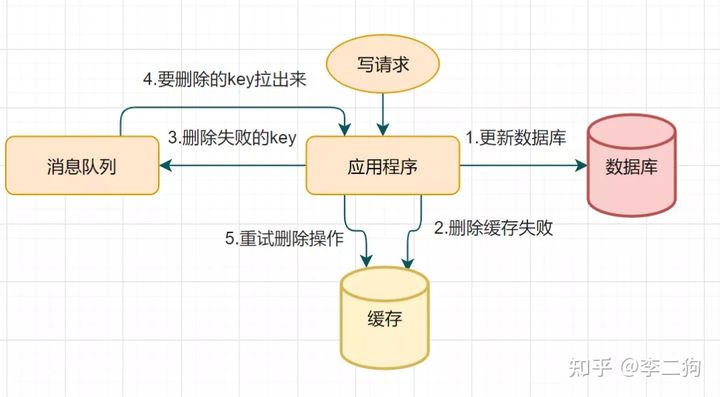
MySQL与Redis的双写一致性：

1. 缓存延时删除：
2. 写请求来临时，先删除缓存，在更新数据库(不要先更新数据库，再删除缓存，这样经过实践出现不一致的概率更大)
3. 休眠一定的时间，再次删除缓存

原因：request A是写请求，删除缓存后，更新数据库，如果此时有request B读缓存，会读不到，然后就去数据库读取，并回填缓存，这样之后的request又会读的是旧数据(直到超时，如果设置了超时时间的话)，这样就需要更新数据库后，再删除一次，这样之后的request就会从数据库读取并回填缓存。流程示意图如下：



删除缓存重试机制：如果删除缓存失败，可以使用消息队列进行重试。



1. 监听binlog：通过监听binlog的消息队列来进行缓存删除。binlog是mysql记录数据更新或潜在更新的日志，可以通过监听它来删除缓存。

Redis事务：

MULTI ：开启事务，redis会将后续的命令逐个放入队列中，然后使用EXEC命令来原子化执行这个命令系列。

EXEC：执行事务中的所有操作命令。

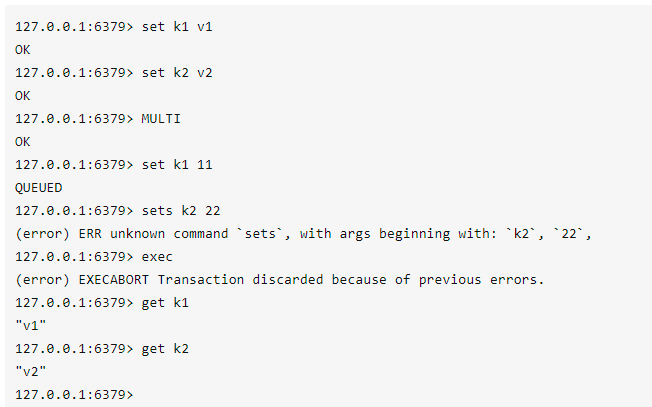
DISCARD：取消事务，放弃执行事务块中的所有命令。

WATCH：监视一个或多个key,如果事务在执行前，这个key(或多个key)被其他命令修改，则事务被中断，不会执行事务中的任何命令。

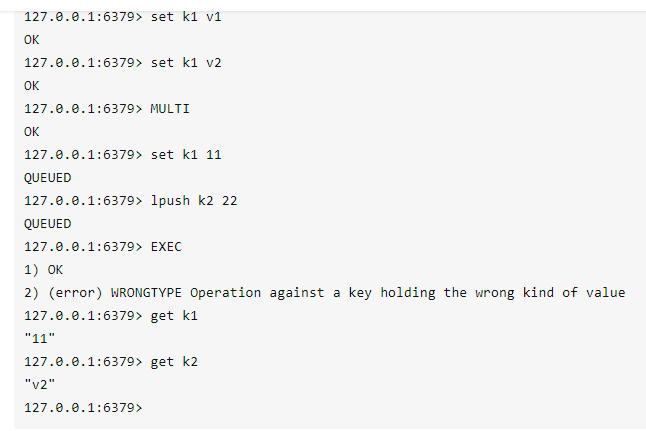
UNWATCH：取消WATCH对所有key的监视。

Redis事务就是顺序性、一次性、排他性的执行一个队列中的一系列命令，但Redis事务不支持回滚操作，如果队列中某条命令执行识别，则可能会产生两种情况：

1. 语法错误：类似C++中的编译错误，这种情况下，所有命令都不会执行成功，k1，k2都保留原值



1. Redis类型错误：类似C++的运行时错误，这种情况下，会有部分命令执行成功。



Redis中的哈希冲突：

采用链表法和Rehash来解决哈希冲突。

布隆过滤器：

假设我们有个集合A，A中有n个元素。利用k个哈希散列函数，将A中的每个元素映射到一个长度为a位的二进制数组B中的不同位置上，这些位置上的二进制数均设置为1。如果待检查的元素，经过这k个哈希散列函数的映射后，发现其k个位置上的二进制数全部为1，这个元素很可能属于集合A，反之，一定不属于集合A

布隆过滤器可能会存在误判，提高的方法：

1. 多用几个哈希函数映射，降低哈希碰撞的概率
2. 同时增加二进制B数组的长度，可以增大hash函数生成的数据的范围，也可以降低哈希碰撞的概率