课前准备

- 准备redis安装包

课堂主题

Redis底层数据结构、IO多路复用、缓存淘汰策略、Redis事务、Redis持久化

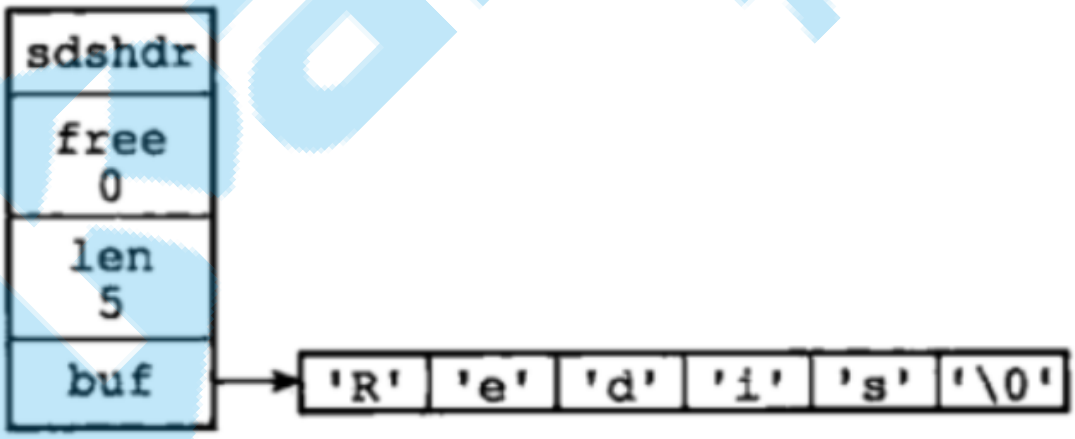
课堂目标

- 掌握Redis数据类型的底层数据结构
- Redis的主要应用有哪些?
- Redis单机安装要掌握?
- Redis数据类型有哪些?
- Redis的数量类型各自的使用场景及注意事项是什么?
- Redis的消息模式是如何实现的?

知识要点

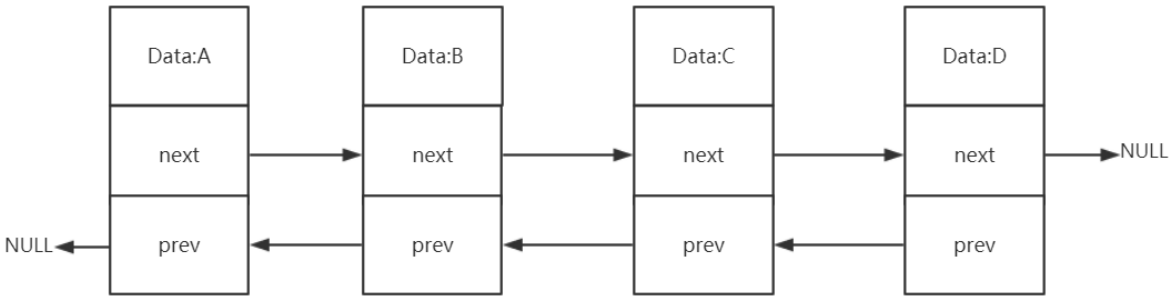
Redis数据结构

简单动态字符串（SDS）



链表

双向链表:可以从两个方向遍历



字典

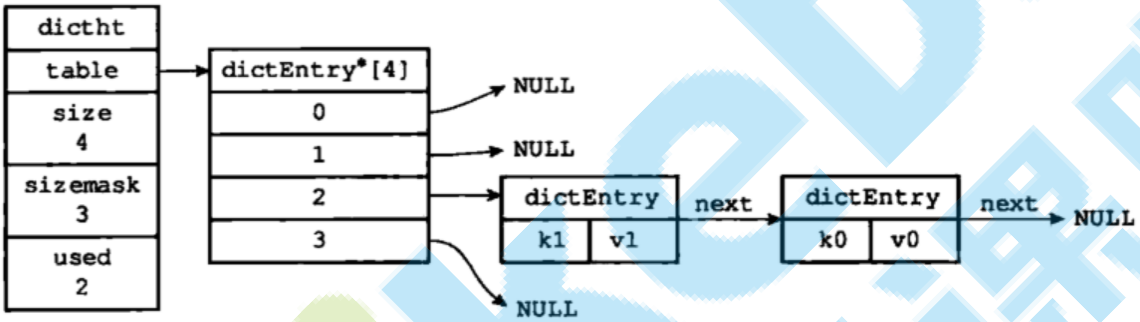
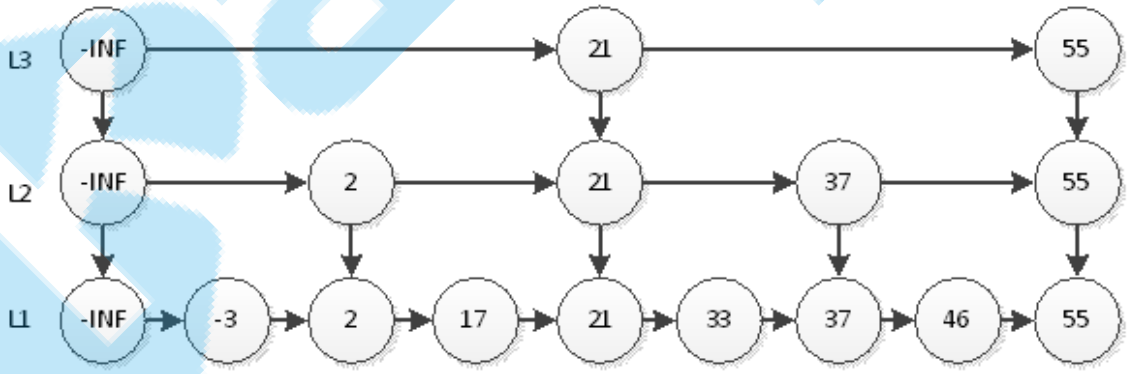


图 4-2　连接在一起的键 K1 和键 K0

跳跃表

查询

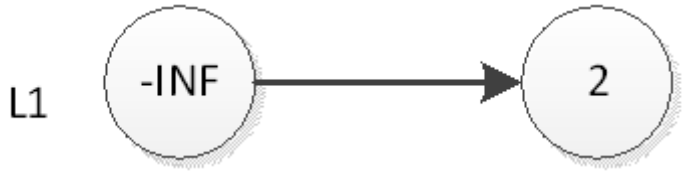


插入

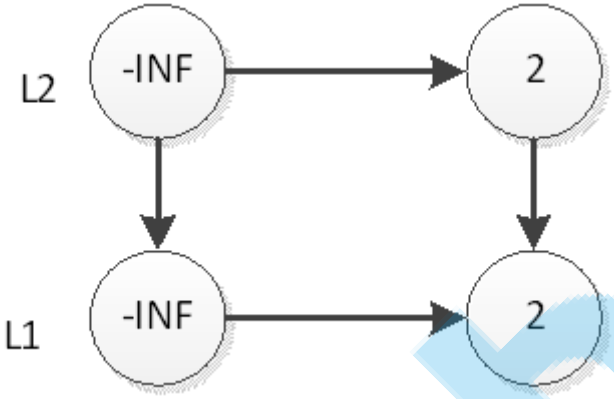
概率算法

在此还是以上图为例：跳跃表的初试状态如下图，表中没有一个元素：

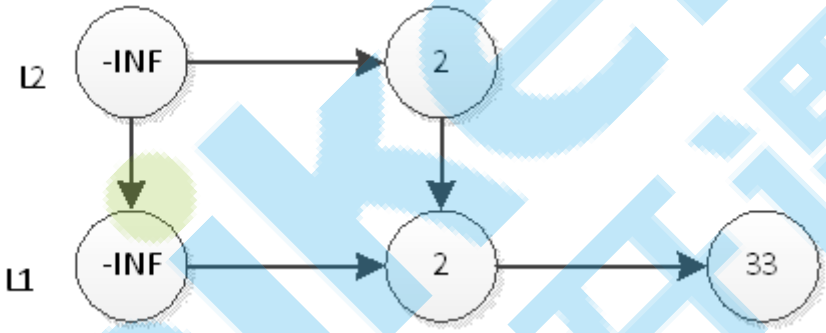
如果我们要插入元素2，首先是在底部插入元素2，如下图：



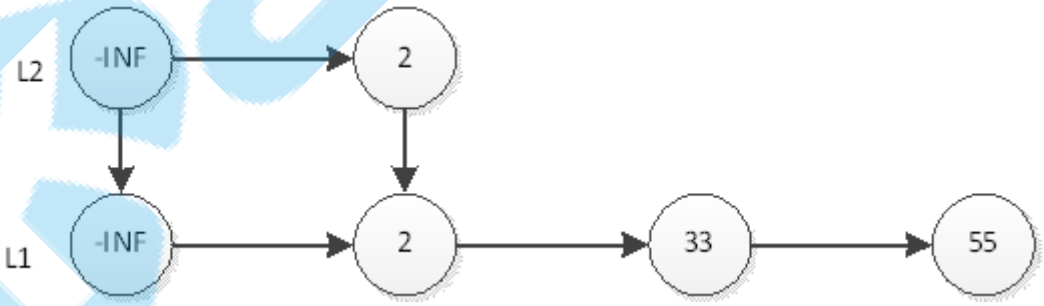
然后我们抛硬币，结果是正面，那么我们要将2插入到L2层，如下图



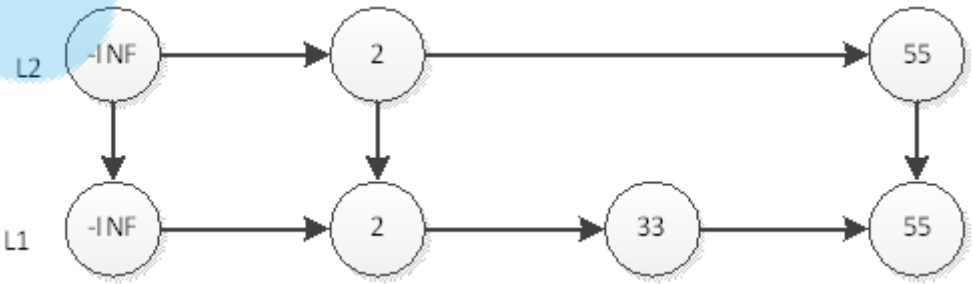
继续抛硬币，结果是反面，那么元素2的插入操作就停止了，插入后的表结构就是上图所示。接下来，我们插入元素33，跟元素2的插入一样，现在L1层插入33，如下图：



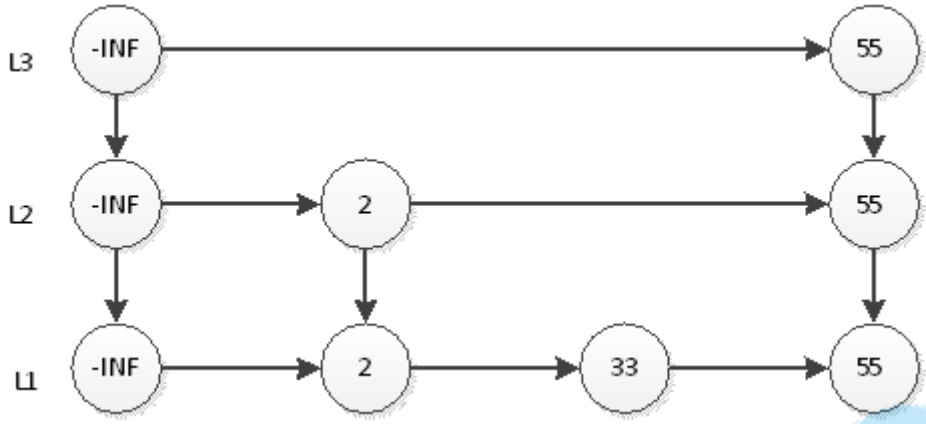
然后抛硬币，结果是反面，那么元素33的插入操作就结束了，插入后的表结构就是上图所示。接下来，我们插入元素55，首先在L1插入55，插入后如下图：



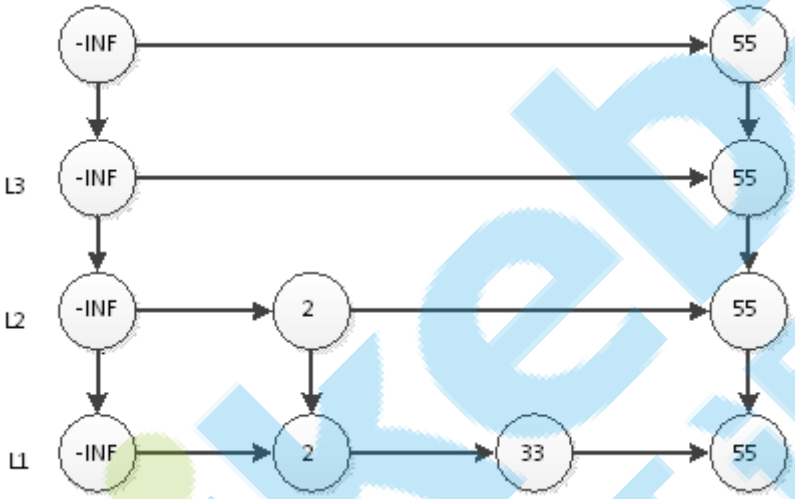
然后抛硬币，结果是正面，那么L2层需要插入55，如下图：



继续抛硬币，结果又是正面，那么L3层需要插入55，如下图：



继续抛硬币，结果又是正面，那么要在L4插入55，结果如下图：



继续抛硬币，结果是反面，那么55的插入结束，表结构就如上图所示。

以此类推，我们插入剩余的元素。当然因为规模小，结果很可能不是一个理想的跳跃表。但是如果元素个数n的规模很大，学过概率论的同学都知道，最终的表结构肯定非常接近于理想跳跃表。

删除

直接删除元素，然后调整一下删除元素后的指针即可。跟普通的链表删除操作完全一样。

整数集合

整数集合（intset）是集合（set）的底层实现之一，当一个集合（set）只包含整数值元素，并且这个集合的元素不多时，Redis就会使用整数集合(intset)作为该集合的底层实现。整数集合（intset）是Redis用于保存整数值的集合抽象数据类型，它可以保存类型为int16_t、int32_t 或者int64_t 的整数值，并且保证集合中不会出现重复元素。

压缩列表

压缩列表（ziplist）是列表键和哈希键的底层实现之一。当一个列表只包含少量列表项时，并且每个列表项时小整数值或短字符串，那么Redis会使用压缩列表来做该列表的底层实现。

压缩列表（ziplist）是Redis为了节省内存而开发的，是由一系列特殊编码的连续内存块组成的顺序型数据结构，一个压缩列表可以包含任意多个节点（entry），每个节点可以保存一个字节数组或者一个整数值。

对象

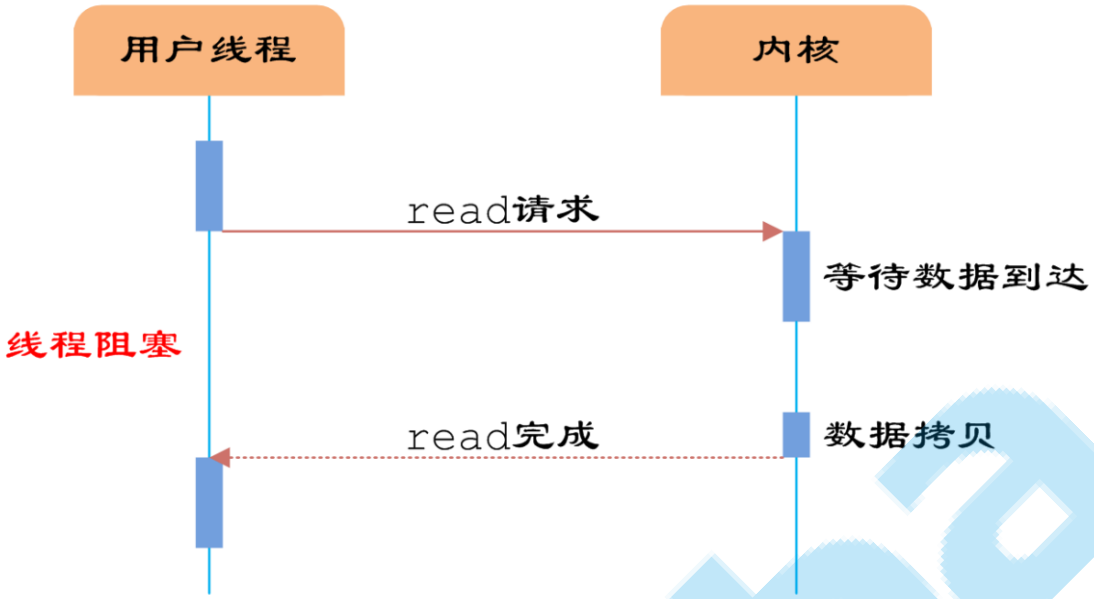
前面我们讲了Redis的数据结构，Redis不是用这些数据结构直接实现Redis的键值对数据库，而是基于这些数据结构创建了一个对象系统。包含字符串对象，列表对象，哈希对象，集合对象和有序集合对象。根据对象的类型可以判断一个对象是否可以执行给定的命令，也可针对不同的使用场景，对象设置有多种不同的数据结构实现，从而优化对象在不同场景下的使用效率。

类型	编码	OBJECT ENCODING命令输出	对象
REDIS_STRING	REDIS_ENCODING_INT	“int”	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	“embstr”	使用embstr编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	“raw”	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	“linkedlist”	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	“hashtable”	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	“intset”	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	“hashtable”	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	“ziplist”	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	“skiplist”	使用跳跃表和字典实现的有序集合对象

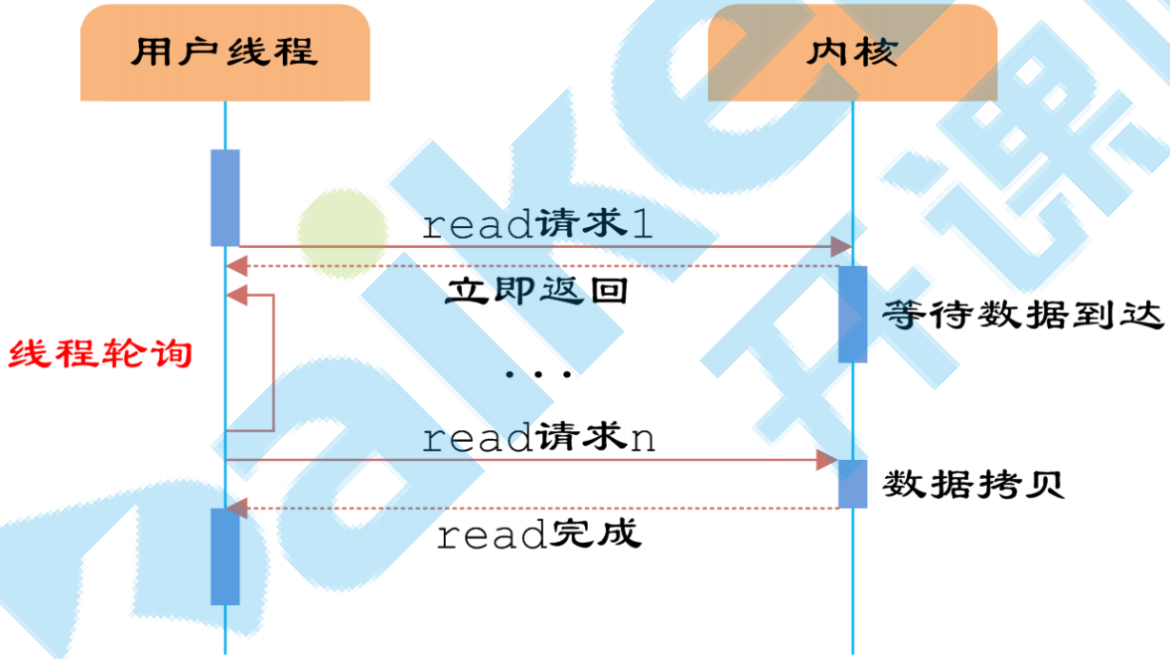
IO多路复用

IO模型

同步阻塞IO（Blocking IO）

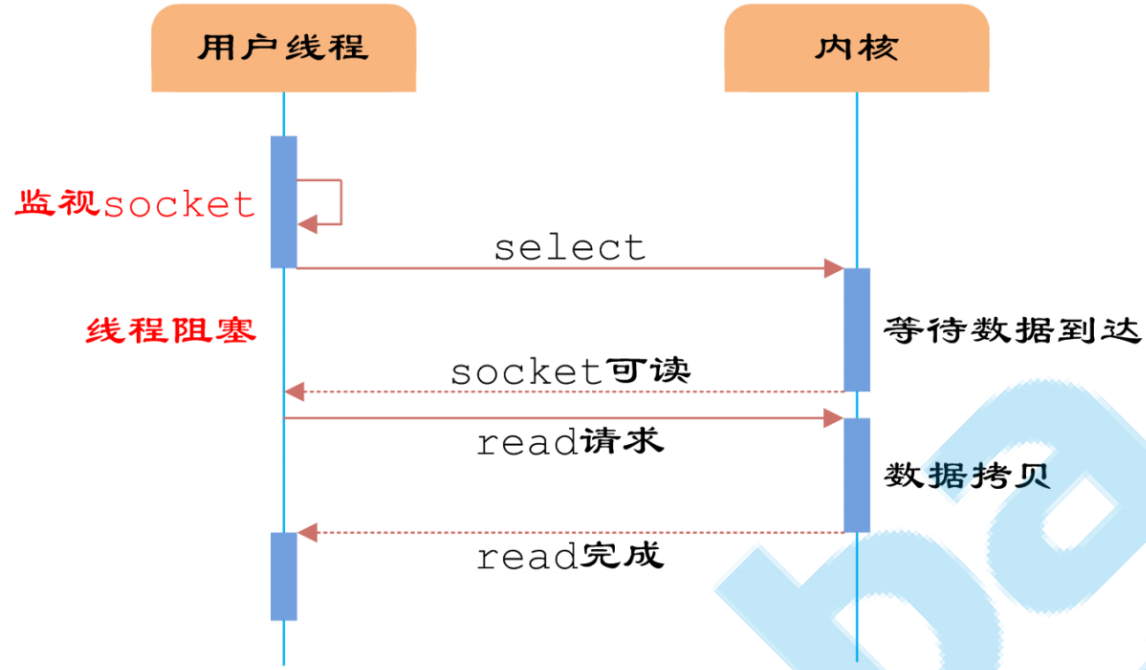


同步非阻塞IO（Non-blocking IO）

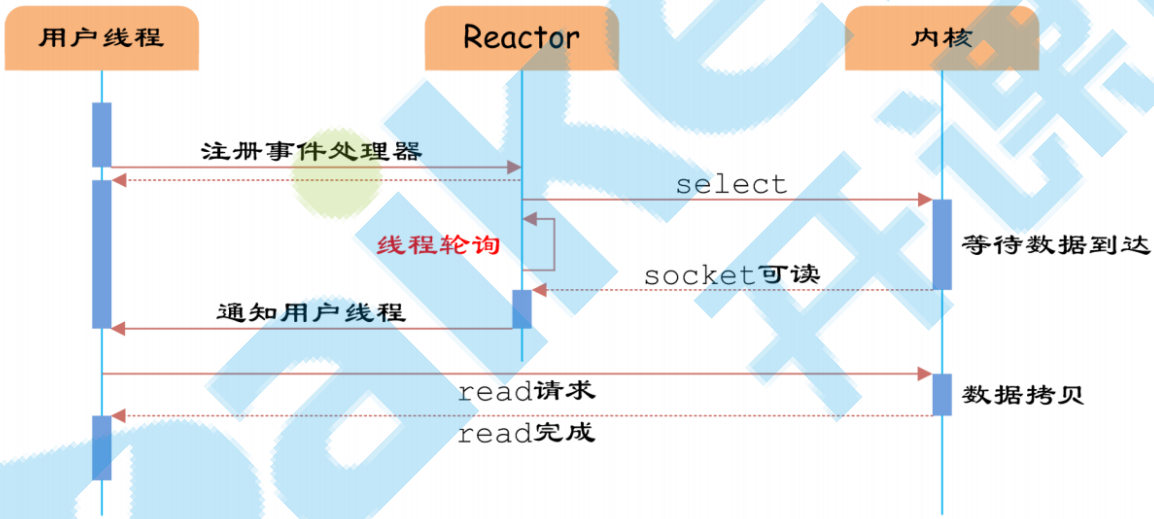


IO多路复用（IO Multiplexing）

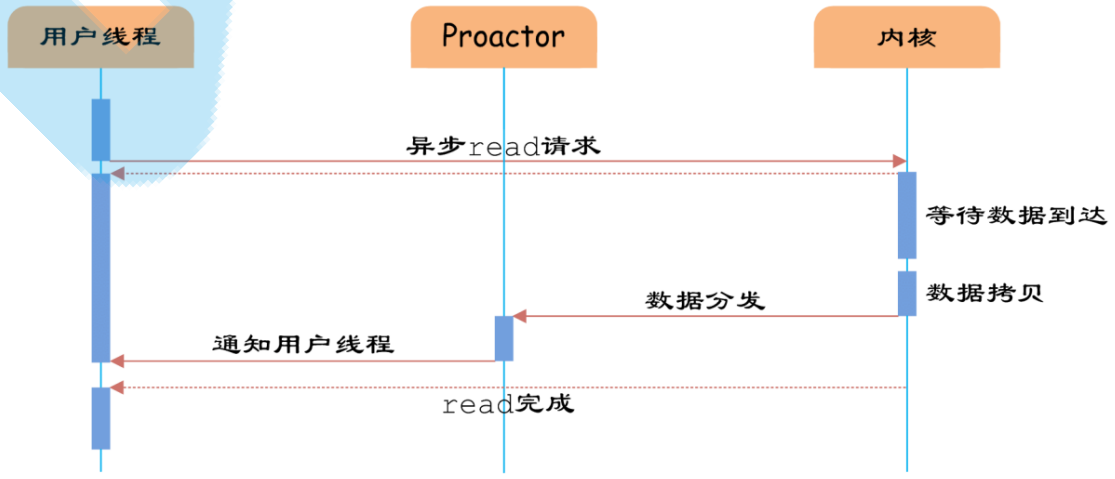
多路分离函数 select



Reactor模式



异步IO（Asynchronous IO）



Redis IO多路复用技术以及epoll实现原理

为什么Redis中要使用I/O多路复用呢?

Redis 是跑在单线程中的，所有的操作都是按照顺序线性执行的，但是由于读写操作等待用户输入或输出都

是阻塞的，所以 I/O 操作在一般情况下往往不能直接返回，这会导致某一文件的 I/O 阻塞导致整个进程无法对其

它客户提供服务，而 I/O 多路复用就是为了解决这个问题而出现的。

IO多路复用实现机制

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个

描述符就绪，能够通知程序进行相应的操作。

redis epoll底层实现

当某一进程调用epoll_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll

的使用方式密切相关。

```
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root  rbr;
    /*双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};
```

每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll_ctl方法向epoll对象中添加进来的事

件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间

效率是lgn，其中n为树的高度)。

而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个

回调方法。这个回调方法在内核中叫ep_poll_callback,它会将发生的事件添加到rdlist双链表中。

缓存淘汰策略

最大缓存

- 在 redis 中，允许用户设置最大使用内存大小**maxmemory**，默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。
- redis 内存数据集大小上升到一定大小的时候，就会实行**数据淘汰策略。**

淘汰策略

redis淘汰策略配置：**maxmemory-policy** voltile-lru，支持热配置

redis 提供 6种数据淘汰策略：

1. voltile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
6. no-enviction（驱逐）：禁止驱逐数据

LRU原理

LRU（**Least recently used**，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

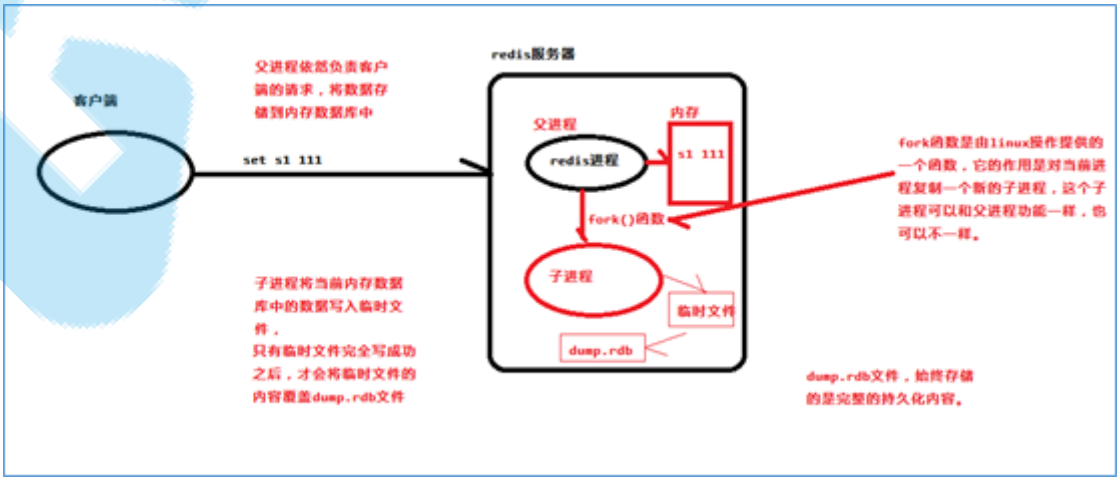
Redis事务

- `Redis` 的事务是通过 `MULTI` 、 `EXEC` 、 `DISCARD` 和 `WATCH` 这四个命令来完成的。
- `Redis` 的单个命令都是原子性的，所以这里需要确保事务性的对象是**命令集合。**
- `Redis` 将命令集合序列化并确保处于同一事务的**命令集合连续且不被打断**的执行
- `Redis` 不支持回滚操作。

Redis持久化

`Redis` 是一个**内存**数据库，为了保证数据的持久性，它提供了两种持久化方案：

- `RDB` 方式（默认）



- `AOF` 方式

  默认情况下 `Redis` 没有开启 `AOF` （`append only file`）方式的持久化。

开启`AOF`持久化后，每执行一条会**更改`Redis`中的数据的命令**，`Redis`就会将该命令写入硬盘中的`AOF`文件，这一过程显然**会降低`Redis`的性能**，但大部分情况下这个影响是能够接受的，另外使**用较快的硬盘可以提高`AOF`的性能。**