# Redis笔记

1. redis是可以在qt下调试，需要删除qmake步骤替换为make步骤，然后attach to runing program.
2. 通过阅读注释版本的redis速度很快。

项目地址：

<https://github.com/huangz1990/redis-3.0-annotated>

redis设计与实现

<http://redisbook.com/>

# 一．事件

Main函数中：aeMain(server.el);开始事件循环

|  |
| --- |
| /\*  \* 事件处理器的主循环  \*/  void aeMain(aeEventLoop \*eventLoop) {  eventLoop->stop = 0;  while (!eventLoop->stop) {  // 如果有需要在事件处理前执行的函数，那么运行它  if (eventLoop->beforesleep != NULL)  eventLoop->beforesleep(eventLoop);  // 开始处理事件  aeProcessEvents(eventLoop, AE\_ALL\_EVENTS);  }  } |

EventLoop.fired中 包含了所有被触发的事件。

方法：numevents = aeApiPoll(eventLoop, tvp); 从操作系统中获取事件,因为ｉｏ复用的策略的多样性所以需要抽出一个函数aeApiPoll

|  |
| --- |
| for (j = 0; j < numevents; j++) {  // 从已就绪数组中获取事件  aeFileEvent \*fe = &eventLoop->events[eventLoop->fired[j].fd];  int mask = eventLoop->fired[j].mask;  int fd = eventLoop->fired[j].fd;  int rfired = 0;  // 读事件  if (fe->mask & mask & AE\_READABLE) {  // rfired 确保读/写事件只能执行其中一个  rfired = 1;  fe->rfileProc(eventLoop,fd,fe->clientData,mask);  }  // 写事件  if (fe->mask & mask & AE\_WRITABLE) {  if (!rfired || fe->wfileProc != fe->rfileProc)  fe->wfileProc(eventLoop,fd,fe->clientData,mask);  }  processed++;  } |

根据读事件还是写事件处理。

aeFileEvent 中定义了 读写事件的处理。这个结构体直接定义了事件处理函数和数据，唯一需要确定的就是事件处理时机．

|  |
| --- |
| /\* File event structure  \*  \* 文件事件结构  \*/  typedef struct aeFileEvent {  // 监听事件类型掩码，  // 值可以是 AE\_READABLE 或 AE\_WRITABLE ，  // 或者 AE\_READABLE | AE\_WRITABLE  int mask; /\* one of AE\_(READABLE|WRITABLE) \*/  // 读事件处理器  aeFileProc \*rfileProc;  // 写事件处理器  aeFileProc \*wfileProc;  // 多路复用库的私有数据  void \*clientData;  } aeFileEvent; |

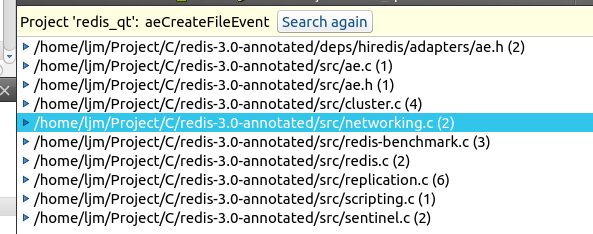
## 1.1绑定函数 aeCreateFileEvent,所有的函数都通过这个函数绑定

|  |
| --- |
| /\*  \* 根据 mask 参数的值，监听 fd 文件的状态，  \* 当 fd 可用时，执行 proc 函数，直接将clientdata赋值给eventloop[fd].clientData  \* 也就是所有必要条件都集中在eventLoop中  \*/  int aeCreateFileEvent(aeEventLoop \*eventLoop, int fd, int mask,  aeFileProc \*proc, void \*clientData)  {  核心函数，绑定事件，还是ｉｏ复用策略  aeApiAddEvent()  } |

Mask取值 复合值

|  |
| --- |
| /\*  \* 文件事件状态  \*/  // 未设置  #define AE\_NONE 0  // 可读  #define AE\_READABLE 1  // 可写  #define AE\_WRITABLE 2 |

这个函数应该要绑定socket的链接,和其他方法,调用如下



网络的绑定如下:

createClient的时候绑定事件处理函数到 **readQueryFromClient**

**当一个客户端请求链接到来时候创建客户端对象,**

|  |
| --- |
| /\*  \* 创建一个新客户端  \*/  redisClient \*createClient(int fd) {  // 分配空间  redisClient \*c = zmalloc(sizeof(redisClient));  /\* passing -1 as fd it is possible to create a non connected client.  \* This is useful since all the Redis commands needs to be executed  \* in the context of a client. When commands are executed in other  \* contexts (for instance a Lua script) we need a non connected client. \*/  // 当 fd 不为 -1 时，创建带网络连接的客户端  // 如果 fd 为 -1 ，那么创建无网络连接的伪客户端  // 因为 Redis 的命令必须在客户端的上下文中使用，所以在执行 Lua 环境中的命令时  // 需要用到这种伪终端  if (fd != -1) {  // 非阻塞  anetNonBlock(NULL,fd);  // 禁用 Nagle 算法  anetEnableTcpNoDelay(NULL,fd);  // 设置 keep alive  if (server.tcpkeepalive)  anetKeepAlive(NULL,fd,server.tcpkeepalive);  // 绑定读事件到事件 loop （开始接收命令请求）  if (aeCreateFileEvent(server.el,fd,AE\_READABLE,  readQueryFromClient, c) == AE\_ERR)  {  close(fd);  zfree(c);  return NULL;  }  }  // 初始化各个属性  // 默认数据库  selectDb(c,0);  // 套接字  c->fd = fd;  // 名字  c->name = NULL;  // 回复缓冲区的偏移量  c->bufpos = 0;  // 查询缓冲区  c->querybuf = sdsempty();  // 查询缓冲区峰值  c->querybuf\_peak = 0;  // 命令请求的类型  c->reqtype = 0;  // 命令参数数量  c->argc = 0;  // 命令参数  c->argv = NULL;  // 当前执行的命令和最近一次执行的命令  c->cmd = c->lastcmd = NULL;  // 查询缓冲区中未读入的命令内容数量  c->multibulklen = 0;  // 读入的参数的长度  c->bulklen = -1;  // 已发送字节数  c->sentlen = 0;  // 状态 FLAG  c->flags = 0;  // 创建时间和最后一次互动时间  c->ctime = c->lastinteraction = server.unixtime;  // 认证状态  c->authenticated = 0;  // 复制状态  c->replstate = REDIS\_REPL\_NONE;  // 复制偏移量  c->reploff = 0;  // 通过 ACK 命令接收到的偏移量  c->repl\_ack\_off = 0;  // 通过 AKC 命令接收到偏移量的时间  c->repl\_ack\_time = 0;  // 客户端为从服务器时使用，记录了从服务器所使用的端口号  c->slave\_listening\_port = 0;  // 回复链表  c->reply = listCreate();  // 回复链表的字节量  c->reply\_bytes = 0;  // 回复缓冲区大小达到软限制的时间  c->obuf\_soft\_limit\_reached\_time = 0;  // 回复链表的释放和复制函数  listSetFreeMethod(c->reply,decrRefCountVoid);  listSetDupMethod(c->reply,dupClientReplyValue);  // 阻塞类型  c->btype = REDIS\_BLOCKED\_NONE;  // 阻塞超时  c->bpop.timeout = 0;  // 造成客户端阻塞的列表键  c->bpop.keys = dictCreate(&setDictType,NULL);  // 在解除阻塞时将元素推入到 target 指定的键中  // BRPOPLPUSH 命令时使用  c->bpop.target = NULL;  c->bpop.numreplicas = 0;  c->bpop.reploffset = 0;  c->woff = 0;  // 进行事务时监视的键  c->watched\_keys = listCreate();  // 订阅的频道和模式  c->pubsub\_channels = dictCreate(&setDictType,NULL);  c->pubsub\_patterns = listCreate();  c->peerid = NULL;  listSetFreeMethod(c->pubsub\_patterns,decrRefCountVoid);  listSetMatchMethod(c->pubsub\_patterns,listMatchObjects);  // 如果不是伪客户端，那么添加到服务器的客户端链表中  if (fd != -1) listAddNodeTail(server.clients,c);  // 初始化客户端的事务状态  initClientMultiState(c);  // 返回客户端  return c;  } |

准备回复客户端 绑定函数到: sendReplyToClient, 也就是client内存在回复数据的时候触发,一个客户端可写意味着有数据需要被回复.

|  |
| --- |
| /\* This function is called every time we are going to transmit new data  \* to the client. The behavior is the following:  \*  \* 这个函数在每次向客户端发送数据时都会被调用。函数的行为如下：  \*  \* If the client should receive new data (normal clients will) the function  \* returns REDIS\_OK, and make sure to install the write handler in our event  \* loop so that when the socket is writable new data gets written.  \*  \* 当客户端可以接收新数据时（通常情况下都是这样），函数返回 REDIS\_OK ，  \* 并将写处理器（write handler）安装到事件循环中，  \* 这样当套接字可写时，新数据就会被写入。  \*  \* If the client should not receive new data, because it is a fake client,  \* a master, a slave not yet online, or because the setup of the write handler  \* failed, the function returns REDIS\_ERR.  \*  \* 对于那些不应该接收新数据的客户端，  \* 比如伪客户端、 master 以及 未 ONLINE 的 slave ，  \* 或者写处理器安装失败时，  \* 函数返回 REDIS\_ERR 。  \*  \* Typically gets called every time a reply is built, before adding more  \* data to the clients output buffers. If the function returns REDIS\_ERR no  \* data should be appended to the output buffers.  \*  \* 通常在每个回复被创建时调用，如果函数返回 REDIS\_ERR ，  \* 那么没有数据会被追加到输出缓冲区。  \*/  int prepareClientToWrite(redisClient \*c) {  // LUA 脚本环境所使用的伪客户端总是可写的  if (c->flags & REDIS\_LUA\_CLIENT) return REDIS\_OK;    // 客户端是主服务器并且不接受查询，  // 那么它是不可写的，出错  if ((c->flags & REDIS\_MASTER) &&  !(c->flags & REDIS\_MASTER\_FORCE\_REPLY)) return REDIS\_ERR;  // 无连接的伪客户端总是不可写的  if (c->fd <= 0) return REDIS\_ERR; /\* Fake client \*/  // 一般情况，为客户端套接字安装写处理器到事件循环  if (c->bufpos == 0 && listLength(c->reply) == 0 &&  (c->replstate == REDIS\_REPL\_NONE ||  c->replstate == REDIS\_REPL\_ONLINE) &&  aeCreateFileEvent(server.el, c->fd, AE\_WRITABLE,  sendReplyToClient, c) == AE\_ERR) return REDIS\_ERR;  return REDIS\_OK;  } |

在redis.c中的initserver()中,绑定了服务端口的的可读事件(只要有客户端链接或者请求就会触发)

|  |
| --- |
| /\* Create an event handler for accepting new connections in TCP and Unix  \* domain sockets. \*/  // 为 TCP 连接关联连接应答（accept）处理器  // 用于接受并应答客户端的 connect() 调用  for (j = 0; j < server.ipfd\_count; j++) {  if (aeCreateFileEvent(server.el, server.ipfd[j], AE\_READABLE,  acceptTcpHandler,NULL) == AE\_ERR)  {  redisPanic(  "Unrecoverable error creating server.ipfd file event.");  }  } |

然后建立:创建一个 TCP 连接处理器

创建一个createClient 处理tcp请求, 里面会绑定事件.

这里会对每一个ip地址绑定链接回调函数.

## 1.2 输入命令 set mykey myvalue

指令协议内容:

|  |
| --- |
| "\*3**\r\n**$3**\r\n**SET**\r\n**$5**\r\n**mykey**\r\n**$7**\r\n**myvalue**\r\n**" |

\*号是指令的开始符号,

3表示三个参数,参数个数不能超过1024\*1024

|  |
| --- |
| **\*3**  **$3**  **SET**  **$5**  **mykey**  **$7**  **myvalue**  \*<参数数量> CR LF  $<参数 1 的字节数量> CR LF  <参数 1 的数据> CR LF  ...  $<参数 N 的字节数量> CR LF  <参数 N 的数据> CR LF |

详细参见:

<http://redis.readthedocs.org/en/latest/topic/protocol.html>

<http://redisdoc.com/>

指令会进入方法:

readQueryFromClient,

processInputBuffer

processMultibulkBuffer

在次函数中设置client的属性值

// 设置参数数量

c->multibulklen = ll;

c->argv = zmalloc(sizeof(robj\*)\*c->multibulklen);

最终解析出argv,argc,

## 1.3 执行指令:

执行在:int processCommand(redisClient \*c)

首先查找指令:

/\*

\* 根据给定命令名字（SDS），查找命令

\*/

struct redisCommand \*lookupCommand(sds name) {

return dictFetchValue(server.commands, name);

}

指令key的hash算法是:

|  |
| --- |
| /\* And a case insensitive hash function (based on djb hash) \*/  unsigned int dictGenCaseHashFunction(const unsigned char \*buf, int len) {  unsigned int hash = (unsigned int)dict\_hash\_function\_seed;  while (len--)  hash = ((hash << 5) + hash) + (tolower(\*buf++)); /\* hash \* 33 + c \*/  return hash;  } |

调用方法

|  |
| --- |
| unsigned int dictSdsHash(const void \*key) {  return dictGenHashFunction((unsigned char\*)key, sdslen((char\*)key));  } |

如果一切条件检查正常执行命令:

call(c,REDIS\_CALL\_FULL);

当然所有的指令都会通过redisCommandTable映射到相应的函数中去

set指令:

void setCommand(redisClient \*c)

调用setGenericCommand

执行了多种类似的set指令.

# 二.数据结构

## 2.1 字典

Redis 的字典使用哈希表作为底层实现， 一个哈希表里面可以有多个哈希表节点， 而每个哈希表节点就保存了字典中的一个键值对。

|  |
| --- |
| void testDict()  {  sds s=sdsnew("dict1");  dict\* d=dictCreate(&optionSetDictType,s);//包括hash方法,复制方法等等必要函数  sds key=sdsnew("mykey");  sds v=sdsnew("vv5566");  if(dictAdd(d,key,v)==DICT\_OK){  dictEntry\* dictentry=dictFind(d,key);  printf("action ok v=%s",dictentry->v.val);  }  else  showerror("add value to dict error");  } |

补充迭代器,rehash使用.