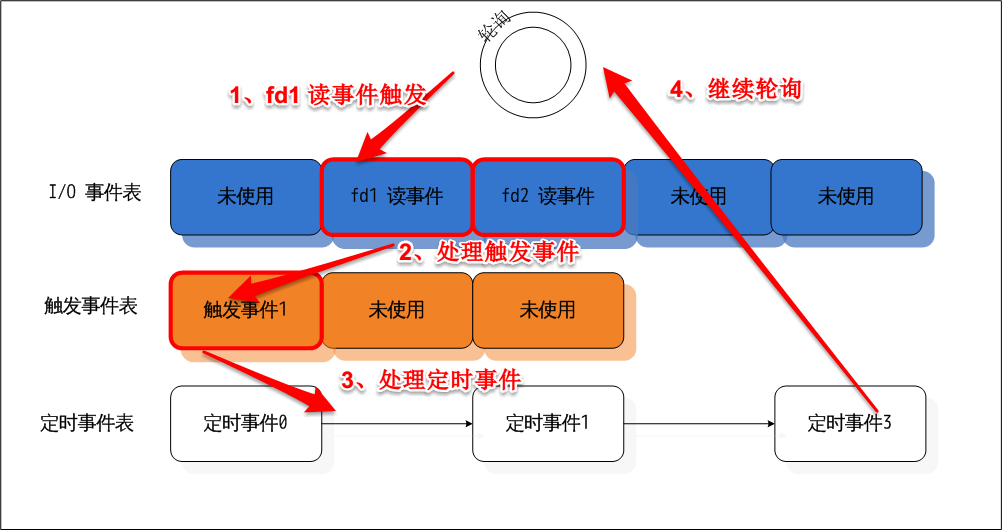
**概述**

**redis 内部有一个小型的事件驱动**，它和 libevent 网络库的事件驱动一样，都是**依托 I/O 多路复用技术支撑起来的。**

利用 I/O 多路复用技术，**监听感兴趣的文件 I/O 事件**，例如读事件，写事件等，同时也要维护一个以文件描述符为主键，数据为某个预设函数的事件表，这里其实就是一个数组或者链表 。**当事件触发时，比如某个文件描述符可读，系统会返回文件描述符值，用这个值在事件表中找到相应的数据项(Handler)，从而实现回调。**同样的，定时事件也是可以实现的，因为系统提供的 I/O 多路复用技术中的函数允许我们设定时间值。

[](http://daoluan.net/blog/wp-content/uploads/2014/03/redis_event_summary.png)

上面一段话比较综合，可能需要一些 linux 系统编程和网络编程的基础，但你会看到多数事件驱动程序都是这么实现的（？）。

**redis 事件驱动数据结构**

**redis 事件驱动内部有四个主要的数据结构，分别是：事件循环结构体，文件事件结构体，时间事件结构体和触发事件结构体。**

|  |
| --- |
| // 文件事件结构体  /\* File event structure \*/  typedef struct aeFileEvent {      int mask; /\* one of AE\_(READABLE|WRITABLE) \*/        // 回调函数指针      aeFileProc \*rfileProc;      aeFileProc \*wfileProc;        // clientData 参数一般是指向 redisClient 的指针      void \*clientData;  } aeFileEvent;    // 时间事件结构体  /\* Time event structure \*/  typedef struct aeTimeEvent {      long long id; /\* time event identifier. \*/      long when\_sec; /\* seconds \*/      long when\_ms; /\* milliseconds \*/        // 定时回调函数指针      aeTimeProc \*timeProc;        // 定时事件清理函数，当删除定时事件的时候会被调用      aeEventFinalizerProc \*finalizerProc;        // clientData 参数一般是指向 redisClient 的指针      void \*clientData;        // 定时事件表采用链表来维护      struct aeTimeEvent \*next;  } aeTimeEvent;    // 触发事件  /\* A fired event \*/  typedef struct aeFiredEvent {      int fd;      int mask;  } aeFiredEvent;    // 事件循环结构体  /\* State of an event based program \*/  typedef struct aeEventLoop {      int maxfd;   /\* highest file descriptor currently registered \*/      int setsize; /\* max number of file descriptors tracked \*/        // 记录最大的定时事件 id + 1      long long timeEventNextId;        // 用于系统时间的矫正      time\_t lastTime;     /\* Used to detect system clock skew \*/        // I/O 事件表      aeFileEvent \*events; /\* Registered events \*/        // 被触发的事件      aeFiredEvent \*fired; /\* Fired events \*/        // 定时事件表      aeTimeEvent \*timeEventHead;        // 事件循环结束标识      int stop;        // 对于不同的 I/O 多路复用技术，有不同的数据，详见各自实现      void \*apidata; /\* This is used for polling API specific data \*/        // 新的循环前需要执行的操作      aeBeforeSleepProc \*beforesleep;  } aeEventLoop; |

上面的数据结构能给我们很好的提示：**事件循环结构体维护 I/O 事件表，定时事件表和触发事件表。**

**事件循环中心**

redis 的主函数中调用 initServer() 函数从而初始化事件循环中心（EventLoop），它的主要工作是在 aeCreateEventLoop() 中完成的。

|  |
| --- |
| aeEventLoop \*aeCreateEventLoop(int setsize) {      aeEventLoop \*eventLoop;      int i;        // 分配空间      if ((eventLoop = zmalloc(sizeof(\*eventLoop))) == NULL) goto err;        // 分配文件事件结构体空间      eventLoop->events = zmalloc(sizeof(aeFileEvent)\*setsize);        // 分配已触发事件结构体空间      eventLoop->fired = zmalloc(sizeof(aeFiredEvent)\*setsize);      if (eventLoop->events == NULL || eventLoop->fired == NULL) goto err;        eventLoop->setsize = setsize;      eventLoop->lastTime = time(NULL);        // 时间事件链表头      eventLoop->timeEventHead = NULL;        // 后续提到      eventLoop->timeEventNextId = 0;      eventLoop->stop = 0;      eventLoop->maxfd = -1;        // 进入事件循环前需要执行的操作，此项会在 redis main() 函数中设置      eventLoop->beforesleep = NULL;        // 在这里，aeApiCreate() 函数对于每个 IO 多路复用模型的实现都有不同，具体参见源代码，因为每种 IO 多路复用模型的初始化都不同      if (aeApiCreate(eventLoop) == -1) goto err;        /\* Events with mask == AE\_NONE are not set. So let's initialize the       \* vector with it. \*/      // 初始化事件类型掩码为无事件状态      for (i = 0; i < setsize; i++)          eventLoop->events[i].mask = AE\_NONE;      return eventLoop;    err:      if (eventLoop) {          zfree(eventLoop->events);          zfree(eventLoop->fired);          zfree(eventLoop);      }      return NULL;  } |

有上面初始化工作只是完成了一个空空的事件中心而已。要想驱动事件循环，还需要下面的工作。

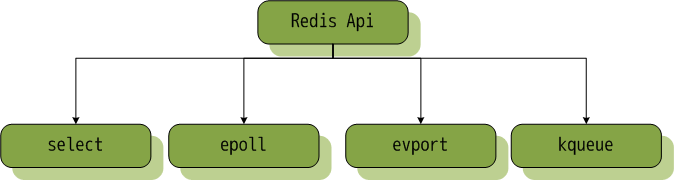
**事件注册详解**

**文件 I/O 事件注册主要操作在 aeCreateFileEvent() 中完成**。aeCreateFileEvent() 会根据文件描述符的数值大小在**事件循环结构体的 I/O 事件表中取一个数据空间，利用系统提供的 I/O 多路复用技术监听感兴趣的 I/O 事件，并设置回调函数**。

[](http://daoluan.net/blog/wp-content/uploads/2014/03/io_event_table.png)

|  |
| --- |
| int aeCreateFileEvent(aeEventLoop \*eventLoop, int fd, int mask,          aeFileProc \*proc, void \*clientData)  {      if (fd >= eventLoop->setsize) {          errno = ERANGE;          return AE\_ERR;      }      // 在 I/O 事件表中选择一个空间      aeFileEvent \*fe = &eventLoop->events[fd];        // aeApiAddEvent() 只在此函数中调用，对于不同 IO 多路复用实现，会有所不同      if (aeApiAddEvent(eventLoop, fd, mask) == -1)          return AE\_ERR;        fe->mask |= mask;        // 设置回调函数      if (mask & AE\_READABLE) fe->rfileProc = proc;      if (mask & AE\_WRITABLE) fe->wfileProc = proc;      fe->clientData = clientData;      if (fd > eventLoop->maxfd)          eventLoop->maxfd = fd;      return AE\_OK;  } |

对于不同版本的 I/O 多路复用，比如 epoll，select，kqueue 等，redis 有各自的版本，但接口统一，譬如 aeApiAddEvent()。

[](http://daoluan.net/blog/wp-content/uploads/2014/03/redis_event_api.png)

之于定时事件，在事件循环结构体中用链表来维护。定时事件操作在 aeCreateTimeEvent() 中完成：分配定时事件结构体，设置触发时间和回调函数，插入到定时事件表中。

[](http://daoluan.net/blog/wp-content/uploads/2014/03/time_event_table.png)

|  |
| --- |
| long long aeCreateTimeEvent(aeEventLoop \*eventLoop, long long milliseconds,          aeTimeProc \*proc, void \*clientData,          aeEventFinalizerProc \*finalizerProc)  {  /\*    自增      timeEventNextId 会在处理执行定时事件时会用到，用于防止出现死循环。      如果超过了最大 id，则跳过这个定时事件，为的是避免死循环，即：      如果事件一执行的时候注册了事件二，事件一执行完毕后事件二得到执行，紧接着如果事件一有得到执行就会成为循环，因此维护了 timeEventNextId 。\*/      long long id = eventLoop->timeEventNextId++;      aeTimeEvent \*te;        // 分配空间      te = zmalloc(sizeof(\*te));      if (te == NULL) return AE\_ERR;        // 填充时间事件结构体      te->id = id;        // 计算超时时间      aeAddMillisecondsToNow(milliseconds,&te->when\_sec,&te->when\_ms);        // proc == serverCorn      te->timeProc = proc;      te->finalizerProc = finalizerProc;      te->clientData = clientData;        // 头插法      te->next = eventLoop->timeEventHead;      eventLoop->timeEventHead = te;      return id;  } |

**准备监听工作**

initServer() 中调用了 aeCreateEventLoop() 完成了事件中心的初始化，initServer() 还做了监听的准备。

|  |
| --- |
| /\* Open the TCP listening socket for the user commands. \*/  // listenToPort() 中有调用 listen()  if (server.port != 0 &&      listenToPort(server.port,server.ipfd,&server.ipfd\_count) == REDIS\_ERR)      exit(1);    // UNIX 域套接字  /\* Open the listening Unix domain socket. \*/  if (server.unixsocket != NULL) {      unlink(server.unixsocket); /\* don't care if this fails \*/      server.sofd = anetUnixServer(server.neterr,server.unixsocket,server.unixsocketperm);      if (server.sofd == ANET\_ERR) {          redisLog(REDIS\_WARNING, "Opening socket: %s", server.neterr);          exit(1);      }  } |

从上面可以看出，redis 提供了 TCP 和 UNIX 域套接字两种工作方式。以 TCP 工作方式为例，listenPort() 创建绑定了套接字并启动了监听。

**为监听套接字注册事件**

在进入事件循环前还需要做一些准备工作。紧接着，initServer() 为所有的监听套接字注册了读事件，响应函数为 acceptTcpHandler() 或者 acceptUnixHandler()。

|  |
| --- |
| // 创建接收 TCP 或者 UNIX 域套接字的事件处理  // TCP  /\* Create an event handler for accepting new connections in TCP and Unix   \* domain sockets. \*/  for (j = 0; j < server.ipfd\_count; j++) {        // acceptTcpHandler() tcp 连接接受处理函数      if (aeCreateFileEvent(server.el, server.ipfd[j], AE\_READABLE,          acceptTcpHandler,NULL) == AE\_ERR)          {              redisPanic(                  "Unrecoverable error creating server.ipfd file event.");          }  }    // UNIX 域套接字  if (server.sofd > 0 && aeCreateFileEvent(server.el,server.sofd,AE\_READABLE,      acceptUnixHandler,NULL) == AE\_ERR) redisPanic("Unrecoverable error creating server.sofd file event."); |

来看看acceptTcpHandler() 做了什么：

|  |
| --- |
| // 用于 TCP 接收请求的处理函数  void acceptTcpHandler(aeEventLoop \*el, int fd, void \*privdata, int mask) {      int cport, cfd;      char cip[REDIS\_IP\_STR\_LEN];      REDIS\_NOTUSED(el);      REDIS\_NOTUSED(mask);      REDIS\_NOTUSED(privdata);        // 接收客户端请求      cfd = anetTcpAccept(server.neterr, fd, cip, sizeof(cip), &cport);        // 出错      if (cfd == AE\_ERR) {          redisLog(REDIS\_WARNING,"Accepting client connection: %s", server.neterr);          return;      }        // 记录      redisLog(REDIS\_VERBOSE,"Accepted %s:%d", cip, cport);        // 真正有意思的地方      acceptCommonHandler(cfd,0);  } |

接收套接字与客户端建立连接后，调用 acceptCommonHandler()。acceptCommonHandler() 主要工作就是：

1. 建立并保存服务端与客户端的连接信息，这些信息保存在一个 struct redisClient 结构体中；
2. 为与客户端连接的套接字注册读事件，相应的回调函数为 readQueryFromClient()，readQueryFromClient() **作用是从套接字读取数据，执行相应操作并回复客户端。**

**redis 事件循环**

以上做好了准备工作，可以进入事件循环。跳出 initServer() 回到 main() 中，main() 会调用 aeMain()。进入事件循环发生在 aeProcessEvents() 中：

1. **根据定时事件表计算需要等待的最短时间；**
2. **调用 redis api aeApiPoll() 进入监听轮询，如果没有事件发生就会进入睡眠状态，其实就是 I/O 多路复用 select() epoll() 等的调用；**
3. **有事件发生会被唤醒，处理已触发的 I/O 事件和定时事件。**

|  |
| --- |
| void aeMain(aeEventLoop \*eventLoop) {      eventLoop->stop = 0;      while (!eventLoop->stop) {            // 进入事件循环可能会进入睡眠状态。在睡眠之前，执行预设置的函数 aeSetBeforeSleepProc()。          if (eventLoop->beforesleep != NULL)              eventLoop->beforesleep(eventLoop);            // AE\_ALL\_EVENTS 表示处理所有的事件          aeProcessEvents(eventLoop, AE\_ALL\_EVENTS);      }  }    // 先处理定时事件，然后处理套接字事件  int aeProcessEvents(aeEventLoop \*eventLoop, int flags)  {      int processed = 0, numevents;        /\* Nothing to do? return ASAP \*/      if (!(flags & AE\_TIME\_EVENTS) && !(flags & AE\_FILE\_EVENTS)) return 0;        /\* Note that we want call select() even if there are no       \* file events to process as long as we want to process time       \* events, in order to sleep until the next time event is ready       \* to fire. \*/      if (eventLoop->maxfd != -1 ||          ((flags & AE\_TIME\_EVENTS) && !(flags & AE\_DONT\_WAIT))) {            int j;          aeTimeEvent \*shortest = NULL;          // tvp 会在 IO 多路复用的函数调用中用到，表示超时时间          struct timeval tv, \*tvp;            // 得到最短将来会发生的定时事件          if (flags & AE\_TIME\_EVENTS && !(flags & AE\_DONT\_WAIT))              shortest = aeSearchNearestTimer(eventLoop);            // 计算睡眠的最短时间          if (shortest) { // 存在定时事件              long now\_sec, now\_ms;                /\* Calculate the time missing for the nearest               \* timer to fire. \*/              // 得到当前时间              aeGetTime(&now\_sec, &now\_ms);              tvp = &tv;              tvp->tv\_sec = shortest->when\_sec - now\_sec;              if (shortest->when\_ms < now\_ms) { // 需要借位                  // 减法中的借位，毫秒向秒借位                  tvp->tv\_usec = ((shortest->when\_ms+1000) - now\_ms)\*1000;                  tvp->tv\_sec --;              } else { // 不需要借位，直接减                  tvp->tv\_usec = (shortest->when\_ms - now\_ms)\*1000;              }                // 当前系统时间已经超过定时事件设定的时间              if (tvp->tv\_sec < 0) tvp->tv\_sec = 0;              if (tvp->tv\_usec < 0) tvp->tv\_usec = 0;          } else {              /\* If we have to check for events but need to return               \* ASAP because of AE\_DONT\_WAIT we need to set the timeout               \* to zero \*/              // 如果没有定时事件，见机行事              if (flags & AE\_DONT\_WAIT) {                  tv.tv\_sec = tv.tv\_usec = 0;                  tvp = &tv;              } else {                  /\* Otherwise we can block \*/                  tvp = NULL; /\* wait forever \*/              }          }            // 调用 IO 多路复用函数阻塞监听          numevents = aeApiPoll(eventLoop, tvp);            // 处理已经触发的事件          for (j = 0; j < numevents; j++) {              // 找到 I/O 事件表中存储的数据              aeFileEvent \*fe = &eventLoop->events[eventLoop->fired[j].fd];              int mask = eventLoop->fired[j].mask;              int fd = eventLoop->fired[j].fd;              int rfired = 0;             /\* note the fe->mask & mask & ... code: maybe an already processed               \* event removed an element that fired and we still didn't               \* processed, so we check if the event is still valid. \*/              // 读事件              if (fe->mask & mask & AE\_READABLE) {                  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++;          }      }        // 处理定时事件      /\* Check time events \*/      if (flags & AE\_TIME\_EVENTS)          processed += processTimeEvents(eventLoop);        return processed; /\* return the number of processed file/time events \*/  } |

**事件触发**

这里以 select 版本的 redis api 实现作为讲解，aeApiPoll() 调用了 select() 进入了监听轮询。aeApiPoll() 的 tvp 参数是最小等待时间，它会被预先计算出来，它主要完成：

1. 拷贝读写的 fdset。select() 的调用会破坏传入的 fdset，实际上有两份 fdset，一份作为备份，另一份用作调用。每次调用 select() 之前都从备份中直接拷贝一份；
2. 调用 select()；
3. 被唤醒后，检查 fdset 中的每一个文件描述符，并将可读或者可写的描述符记录到触发表当中。

接下来的操作便是执行相应的回调函数，代码在上一段中已经贴出：先处理 I/O 事件，再处理定时事件。

|  |
| --- |
| static int aeApiPoll(aeEventLoop \*eventLoop, struct timeval \*tvp) {      aeApiState \*state = eventLoop->apidata;      int retval, j, numevents = 0;        /\*      真有意思，在 aeApiState 结构中：      typedef struct aeApiState {          fd\_set rfds, wfds;          fd\_set \_rfds, \_wfds;      } aeApiState;      在调用 select() 的时候传入的是 \_rfds 和 \_wfds，所有监听的数据在 rfds 和 wfds 中。      在下次需要调用 selec() 的时候，会将 rfds 和 wfds 中的数据拷贝进 \_rfds 和 \_wfds 中。\*/      memcpy(&state->\_rfds,&state->rfds,sizeof(fd\_set));      memcpy(&state->\_wfds,&state->wfds,sizeof(fd\_set));        retval = select(eventLoop->maxfd+1,                  &state->\_rfds,&state->\_wfds,NULL,tvp);      if (retval > 0) {          // 轮询          for (j = 0; j <= eventLoop->maxfd; j++) {              int mask = 0;              aeFileEvent \*fe = &eventLoop->events[j];                if (fe->mask == AE\_NONE) continue;              if (fe->mask & AE\_READABLE && FD\_ISSET(j,&state->\_rfds))                  mask |= AE\_READABLE;              if (fe->mask & AE\_WRITABLE && FD\_ISSET(j,&state->\_wfds))                  mask |= AE\_WRITABLE;                // 添加到触发事件表中              eventLoop->fired[numevents].fd = j;              eventLoop->fired[numevents].mask = mask;              numevents++;          }      }      return numevents;  } |

**总结**

redis 的事件驱动总结如下：

1. **初始化事件循环结构体**
2. **注册监听套接字（socket）的读事件**
3. **注册定时事件**
4. **进入事件循环**
5. **如果监听套接字（socket）变为可读，会接收客户端请求，并为对应的套接字（socket）注册读事件**
6. **如果与客户端连接的套接字变为可读，执行相应的操作**

