链滴

【Redis 源码】Redis 事件监听

作者：zeekling

原文链接：https://ld246.com/article/1609681363067

来源网站：链滴

简介

Redis服务器是典型的事件驱动程序，而事件又分为文件事件（socket的可读可写事件）与时间事件定时任务）两大类。无论是文件事件还是时间事件都封装在结构体aeEventLoop中：

```
typedef struct aeEventLoop {
    int maxfd;   /* highest file descriptor currently registered */
    int setsize; /* max number of file descriptors tracked */
    long long timeEventNextId;
    time_t lastTime;     /* Used to detect system clock skew */
    aeFileEvent *events; /* Registered events */
    aeFiredEvent *fired; /* Fired events */
    aeTimeEvent *timeEventHead;
    int stop;
    void *apidata; /* This is used for polling API specific data */
    aeBeforeSleepProc *beforesleep;
    aeBeforeSleepProc *aftersleep;
    int flags;
} aeEventLoop;
```

- stop标识事件循环是否结束；
- events为文件事件数组，存储已经注册的文件事件；
- fired存储被触发的文件事件；Redis有多个定时任务，因此理论上应该有多个时间事件，多个时间件形成链表，
- timeEventHead即为时间事件链表头节点；Redis服务器需要阻塞等待文件事件的发生，进程阻塞前会调用beforesleep函数，进程因为某种原因被唤醒之后会调用aftersleep函数。Redis底层可以使4种I/O多路复用模型（kqueue、epoll等），
- apidata是对这4种模型的进一步封装。

事件驱动程序通常存在while/for循环，循环等待事件发生并处理，Redis也不例外，其事件循环如下：

```
void aeMain(aeEventLoop *eventLoop) {
    eventLoop->stop = 0;
    while (!eventLoop->stop) {
        aeProcessEvents(eventLoop, AE_ALL_EVENTS|
                                   AE_CALL_BEFORE_SLEEP|
                                   AE_CALL_AFTER_SLEEP);
    }
}
```

函数aeProcessEvents为事件处理主函数，其第2个参数是一个标志位，AE_ALL_EVENTS表示函数需处理文件事件与时间事件，AE_CALL_AFTER_SLEEP表示阻塞等待文件事件之后需要执行aftersleep数。

文件监听事件

Redis客户端通过TCP socket与服务端交互，文件事件指的就是socket的可读可写事件。socket读写作有阻塞与非阻塞之分。采用阻塞模式时，一个进程只能处理一条网络连接的读写事件，为了同时处多条网络连接，通常会采用多线程或者多进程，效率低下；非阻塞模式下，可以使用目前比较成熟的IO多路复用模型，如select/epoll/kqueue等，视不同操作系统而定。

根据操作系统选择I/O复用模型，选择顺序为epoll、kqueue、select。

```
#ifdef HAVE_EVPORT
#include "ae_evport.c"
#else
    #ifdef HAVE_EPOLL
    #include "ae_epoll.c"
    #else
        #ifdef HAVE_KQUEUE
        #include "ae_kqueue.c"
        #else
        #include "ae_select.c"
        #endif
    #endif
#endif
```

以epoll为例，aeApiCreate函数是对epoll_create的封装；aeApiAddEvent函数用于添加事件，是对poll_ctl的封装；aeApiDelEvent函数用于删除事件，是对epoll_ctl的封装；aeApiPoll是对epoll_wai的封装。

```
static int aeApiCreate(aeEventLoop *eventLoop);
static int aeApiAddEvent(aeEventLoop *eventLoop, int fd, int mask);
static void aeApiDelEvent(aeEventLoop *eventLoop, int fd, int delmask)
static int aeApiPoll(aeEventLoop *eventLoop, struct timeval *tvp);
```

4个函数的输入参数含义如下。

- eventLoop：事件循环，与文件事件相关的最主要字段有3个，
- apidata指向I/O多路复用模型对象，注意4种I/O多路复用模型对象的类型不同，因此此字段是void类型；
- events存储需要监控的事件数组，以socket文件描述符作为数组索引存取元素；fired存储已触发的件数组。

时间监听事件

前面介绍了Redis文件事件，已经知道事件循环执行函数aeProcessEvents的主要逻辑：①查找最早发生的时间事件，计算超时时间；②阻塞等待文件事件的产生；③处理文件事件；④处理时间事件。间事件的执行函数为processTimeEvents。

Redis服务器内部有很多定时任务需要执行，比如定时清除超时客户端连接，定时删除过期键等，定任务被封装为时间事件aeTimeEvent对象，多个时间事件形成链表，存储在aeEventLoop结构体的ti eEventHead字段，它指向链表首节点。时间事件aeTimeEvent定义如下：

```
typedef struct aeTimeEvent {
    long long id; /* time event identifier. */
    long when_sec; /* seconds */
    long when_ms; /* milliseconds */
    aeTimeProc *timeProc;
    aeEventFinalizerProc *finalizerProc;
    void *clientData;
    struct aeTimeEvent *prev;
    struct aeTimeEvent *next;
    int refcount; /* refcount to prevent timer events from being
```

```
         * freed in recursive time event calls. */
} aeTimeEvent;
```

各字段含义如下。

- id：时间事件唯一ID，通过字段eventLoop->timeEventNextId实现；
- when_sec与when_ms：时间事件触发的秒数与毫秒数；
- timeProc：函数指针，指向时间事件处理函数；
- finalizerProc：函数指针，删除时间事件节点之前会调用此函数；
- clientData：指向对应的客户端对象；
- prev：指向上一个时间事件节点。
- next：指向下一个时间事件节点。

时间事件执行函数processTimeEvents的处理逻辑比较简单，只是遍历时间事件链表，判断当前时间件是否已经到期，如果到期则执行时间事件处理函数timeProc：

```
static int processTimeEvents(aeEventLoop *eventLoop) {
    int processed = 0;
    aeTimeEvent *te;
    long long maxId;
    time_t now = time(NULL);

    /* 校对系统时间 */
    if (now < eventLoop->lastTime) {
        te = eventLoop->timeEventHead;
        while(te) {
            te->when_sec = 0;
            te = te->next;
        }
    }
    eventLoop->lastTime = now;

    te = eventLoop->timeEventHead;
    maxId = eventLoop->timeEventNextId-1;
    while(te) {
        long now_sec, now_ms;
        long long id;

        /* 删除已经指定要删除的事件 */
        if (te->id == AE_DELETED_EVENT_ID) {
            aeTimeEvent *next = te->next;
            if (te->refcount) {
                te = next;
                continue;
            }
            if (te->prev)
                te->prev->next = te->next;
            else
                eventLoop->timeEventHead = te->next;
            if (te->next)
                te->next->prev = te->prev;
```

```
            if (te->finalizerProc)
                te->finalizerProc(eventLoop, te->clientData);
            zfree(te);
            te = next;
            continue;
        }

        /* 确保只执行Redis自己创建的时间事件 */
        if (te->id > maxId) {
            te = te->next;
            continue;
        }
        aeGetTime(&now_sec, &now_ms);
        if (now_sec > te->when_sec ||
            (now_sec == te->when_sec && now_ms >= te->when_ms))
        {
            int retval;

            id = te->id;
            te->refcount++;
            // 处理时间事件
            retval = te->timeProc(eventLoop, id, te->clientData);
            te->refcount--;
            processed++;
            // 重新设置时间事件到期时间
            if (retval != AE_NOMORE) {
                aeAddMillisecondsToNow(retval,&te->when_sec,&te->when_ms);
            } else {
                te->id = AE_DELETED_EVENT_ID;
            }
        }
        te = te->next;
    }
    return processed;
}
```

注意时间事件处理函数timeProc返回值retval，其表示此时间事件下次应该被触发的时间，单位为毫，且是一个相对时间，即从当前时间算起，retval毫秒后此时间事件会被触发。

Redis创建时间事件节点的函数为aeCreateTimeEvent，内部实现非常简单，只是创建时间事件并添到时间事件链表。aeCreateTimeEvent函数定义如下：

```
long long aeCreateTimeEvent(aeEventLoop *eventLoop, long long milliseconds,
        aeTimeProc *proc, void *clientData,
        aeEventFinalizerProc *finalizerProc);
```

各字段含义如下。

- eventLoop：输入参数指向事件循环结构体；
- milliseconds：表示此时间事件触发时间，单位毫秒，注意这是一个相对时间，即从当前时间算起milliseconds毫秒后此时间事件会被触发；
- proc：指向时间事件的处理函数；
- clientData：指向对应的结构体对象；

● finalizerProc：同样是函数指针，删除时间事件节点之前会调用此函数。

```
int serverCron(struct aeEventLoop *eventLoop, long long id, void *clientData) {
    run_with_period(100) {
      //100毫秒周期执行
    }
    run_with_period(5000) {
     //5000毫秒周期执行
    }

    /* Show information about connected clients */
    if (!server.sentinel_mode) {
        run_with_period(5000) {
            serverLog(LL_DEBUG,
                "%lu clients connected (%lu replicas), %zu bytes in use",
                listLength(server.clients)-listLength(server.slaves),
                listLength(server.slaves),
                zmalloc_used_memory());
        }
    }

    /* 清除无用客户端连接 */
    clientsCron();

    /* 执行数据库定时任务 */
    databasesCron();

    /* 检查是否手动执行了AOF写入命令*/
    if (!hasActiveChildProcess() &&
        server.aof_rewrite_scheduled)
    {
        rewriteAppendOnlyFileBackground();
    }

    /* 检查AOF写入是否完成等 */
    if (hasActiveChildProcess() || ldbPendingChildren())
    {
        checkChildrenDone();
    } else {

    }

    /* AOF定时写入磁盘 */
    if (server.aof_flush_postponed_start) flushAppendOnlyFile(0);

    /* 定时检查AOF写入结果*/
    run_with_period(1000) {
        if (server.aof_last_write_status == C_ERR)
            flushAppendOnlyFile(0);
    }

    /* 清理无用的客户端连接 */
    clientsArePaused(); /* Don't check return value, just use the side effect.*/
```

```
    /* cluster 定时检查 */
    run_with_period(1000) replicationCron();

    /* 运行cluster定时任务 */
    run_with_period(100) {
        if (server.cluster_enabled) clusterCron();
    }

    /* 哨兵模式定时任务执行 */
    if (server.sentinel_mode) sentinelTimer();

    /* Cleanup expired MIGRATE cached sockets. */
    run_with_period(1000) {
        migrateCloseTimedoutSockets();
    }

    /* 停止无用的I/O线程 */
    stopThreadedIOIfNeeded();

    /* 定时检查slot信息 */
    if (server.tracking_clients) trackingLimitUsedSlots();

    /*定时模块定时任务执行 */
    RedisModuleCronLoopV1 ei = {REDISMODULE_CRON_LOOP_VERSION,server.hz};
    moduleFireServerEvent(REDISMODULE_EVENT_CRON_LOOP,
                          0,
                          &ei);

    server.cronloops++;
    return 1000/server.hz;
}
```