

## 13-Redis6.0多IO线程的效率提高了吗？

你好，我是蒋德钧。

通过上节课的学习，我们知道Redis server启动后的进程会以单线程的方式，执行客户端请求解析和处理工作。但是，Redis server也会通过bioInit函数启动三个后台线程，来处理后台任务。也就是说，Redis不再让主线程执行一些耗时操作，比如同步写、删除等，而是交给后台线程异步完成，从而避免了对主线程的阻塞。

实际上，在2020年5月推出的Redis 6.0版本中，Redis在执行模型中还进一步使用了多线程来处理IO任务，这样设计的目的，就是为了充分利用当前服务器的多核特性，使用多核运行多线程，让多线程帮助加速数据读取、命令解析以及数据写回的速度，提升Redis整体性能。

**那么，这些多线程具体是在什么时候启动，又是通过什么方式来处理IO请求的呢？**

今天这节课，我就来给你介绍下Redis 6.0实现的多IO线程机制。通过这部分内容的学习，你可以充分了解到Redis 6.0是如何通过多线程来提升IO请求处理效率的。这样你也就可以结合实际业务来评估，自己是否需要使用Redis 6.0了。

好，接下来，我们先来看下多IO线程的初始化。注意，因为我们之前课程中阅读的是Redis 5.0.8版本的代码，所以在开始学习今天的课程之前，你还需要下载[Redis 6.0.15](#)的源码，以便能查看到和多IO线程机制相关的代码。

### 多IO线程的初始化

我在上一讲给你介绍过，Redis 5.0中的三个后台线程，是server在初始化过程的最后，调用InitServerLast函数，而InitServerLast函数再进一步调用bioInit函数来完成的。如果我们在Redis 6.0中查看InitServerLast函数，会发现和Redis 5.0相比，该函数在调完bioInit函数后，又调用了**initThreadedIO函数**。而initThreadedIO函数正是用来初始化多IO线程的，这部分的代码调用如下所示：

```
void InitServerLast() {
    bioInit();
    initThreadedIO(); //调用initThreadedIO函数初始化IO线程
    set_jemalloc_bg_thread(server.jemalloc_bg_thread);
    server.initial_memory_usage = zmalloc_used_memory();
}
```

所以下面，我们就来看下initThreadedIO函数的主要执行流程，这个函数是在[networking.c](#)文件中实现的。

**首先，initThreadedIO函数会设置IO线程的激活标志。**这个激活标志保存在redisServer结构体类型的全局变量server当中，对应redisServer结构体的成员变量io\_threads\_active。initThreadedIO函数会把io\_threads\_active初始化为0，表示IO线程还没有被激活。这部分代码如下所示：

```
void initThreadedIO(void) {
    server.io_threads_active = 0;
    ...
}
```

```
}
```

这里，你要注意一下，刚才提到的**全局变量server**是Redis server运行时，用来保存各种全局信息的结构体变量。我在[第8讲](#)给你介绍Redis server初始化过程的时候，提到过Redis server的各种参数初始化配置，都是保存在这个全局变量server中的。所以，当你在阅读Redis源码时，如果在某个函数中看到变量server，要知道其实就是这个全局变量。

**紧接着，initThreadedIO函数会对设置的IO线程数量进行判断。**这个数量就是保存在全局变量server的成员变量io\_threads\_num中的。那么在这里，IO线程的数量判断会有三种结果。

第一种，如果IO线程数量为1，就表示只有1个主IO线程，initThreadedIO函数就直接返回了。此时，Redis server的IO线程和Redis 6.0之前的版本是相同的。

```
if (server.io_threads_num == 1) return;
```

第二种，如果IO线程数量大于宏定义IO\_THREADS\_MAX\_NUM（默认值为128），那么initThreadedIO函数会报错，并退出整个程序。

```
if (server.io_threads_num > IO_THREADS_MAX_NUM) {  
    ... //报错日志记录  
    exit(1); //退出程序  
}
```

第三种，如果IO线程数量大于1，并且小于宏定义IO\_THREADS\_MAX\_NUM，那么，initThreadedIO函数会执行一个循环流程，该流程的循环次数就是设置的IO线程数量。

如此一来，在该循环流程中，initThreadedIO函数就会给以下四个数组进行初始化操作。

- **io\_threads\_list数组**：保存了每个IO线程要处理的客户端，将数组每个元素初始化为一个List类型的列表；
- **io\_threads\_pending数组**：保存等待每个IO线程处理的客户端个数；
- **io\_threads\_mutex数组**：保存线程互斥锁；
- **io\_threads数组**：保存每个IO线程的描述符。

这四个数组的定义都在networking.c文件中，如下所示：

```
pthread_t io_threads[IO_THREADS_MAX_NUM]; //记录线程描述符的数组  
pthread_mutex_t io_threads_mutex[IO_THREADS_MAX_NUM]; //记录线程互斥锁的数组  
_Atomic unsigned long io_threads_pending[IO_THREADS_MAX_NUM]; //记录线程待处理的客户端个数  
list *io_threads_list[IO_THREADS_MAX_NUM]; //记录线程对应处理的客户端
```

然后，在对这些数组进行初始化的同时，initThreadedIO函数还会根据IO线程数量，**调用pthread\_create函数创建相应数量的线程**。我在上节课给你介绍过，pthread\_create函数的参数包括创建线程要运行的函数和函数参数（\*tidp、\*attr、\*start\_routine、\*arg）。

所以，对于initThreadedIO函数来说，它创建的线程要运行的函数是**IOThreadMain**，参数是当前创建线程的编号。不过要注意的是，这个编号是从1开始的，编号为0的线程其实是运行Redis server主流程的主IO线程。

以下代码就展示了initThreadedIO函数对数组的初始化，以及创建IO线程的过程，你可以看下。

```
for (int i = 0; i < server.io_threads_num; i++) {

    io_threads_list[i] = listCreate();
    if (i == 0) continue; //编号为0的线程是主IO线程

    pthread_t tid;
    pthread_mutex_init(&io_threads_mutex[i], NULL); //初始化io_threads_mutex数组
    io_threads_pending[i] = 0; //初始化io_threads_pending数组
    pthread_mutex_lock(&io_threads_mutex[i]);
    //调用pthread_create函数创建IO线程，线程运行函数为IOThreadMain
    if (pthread_create(&tid, NULL, IOThreadMain, (void*)(long)i) != 0) {
        ... //出错处理
    }
    io_threads[i] = tid; //初始化io_threads数组，设置值为线程标识
}
```

好了，现在我们再来看下，刚才介绍的IO线程启动后要运行的函数IOThreadMain。了解这个函数，可以帮助我们掌握IO线程实际做的工作。

## IO线程的运行函数IOThreadMain

IOThreadMain函数也是在networking.c文件中定义的，它的主要执行逻辑是一个**while(1)循环**。在这个循环中，IOThreadMain函数会把io\_threads\_list数组中，每个IO线程对应的列表读取出来。

就像我在前面给你介绍的一样，io\_threads\_list数组中会针对每个IO线程，使用一个列表记录该线程要处理的客户端。所以，IOThreadMain函数就会从每个IO线程对应的列表中，进一步取出要处理的客户端，然后判断线程要执行的操作标记。这个操作标记是用变量io\_threads\_op表示的，它有两种取值。

- **io\_threads\_op的值为宏定义IO\_THREADS\_OP\_WRITE**：这表明该IO线程要做的是写操作，线程会调用writeToClient函数将数据写回客户端。
- **io\_threads\_op的值为宏定义IO\_THREADS\_OP\_READ**：这表明该IO线程要做的是读操作，线程会调用readQueryFromClient函数从客户端读取数据。

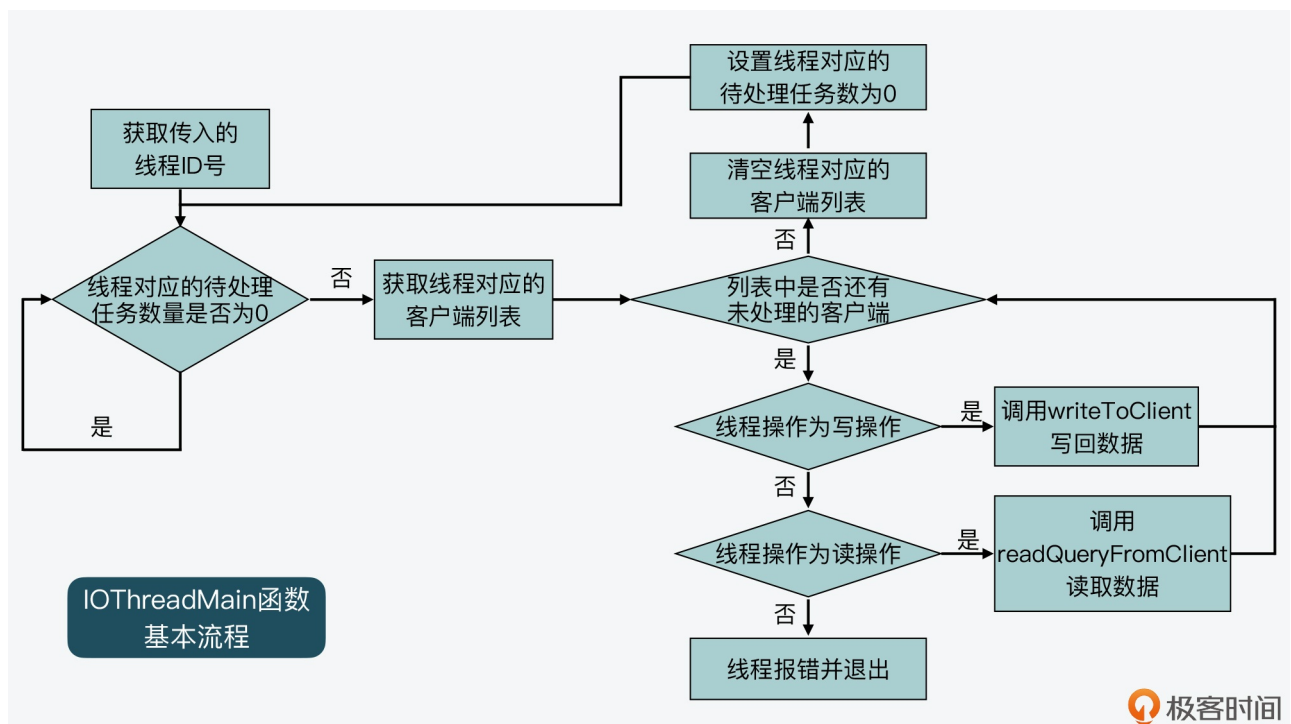
这部分的代码逻辑你可以看看下面的代码。

```

void *IOThreadMain(void *myid) {
...
while(1) {
    listIter li;
    listNode *ln;
    //获取IO线程要处理的客户端列表
    listRewind(io_threads_list[id],&li);
    while((ln = listNext(&li))) {
        client *c = listNodeValue(ln); //从客户端列表中获取一个客户端
        if (io_threads_op == IO_THREADS_OP_WRITE) {
            writeToClient(c,0); //如果线程操作是写操作，则调用writeToClient将数据写回客户端
        } else if (io_threads_op == IO_THREADS_OP_READ) {
            readQueryFromClient(c->conn); //如果线程操作是读操作，则调用readQueryFromClient从客户端读取数据
        } else {
            serverPanic("io_threads_op value is unknown");
        }
    }
    listEmpty(io_threads_list[id]); //处理完所有客户端后，清空该线程的客户端列表
    io_threads_pending[id] = 0; //将该线程的待处理任务数量设置为0
}
}
}

```

我也画了下面这张图，展示了IOThreadMain函数的基本流程，你可以看下。



好了，到这里你应该就了解了，每一个IO线程运行时，都会不断检查是否有等待它处理的客户端。如果有，就根据操作类型，从客户端读取数据或是将数据写回客户端。你可以看到，这些操作都是Redis要和客户端完成的IO操作，所以，这也是为什么我们把这些线程称为IO线程的原因。

那么，你看到这里，可能也会产生一些疑问，**IO线程要处理的客户端是如何添加到io\_threads\_list数组中的呢？**

这就要说到Redis server对应的全局变量server了。server变量中有两个List类型的成员变量：

clients\_pending\_write和clients\_pending\_read，它们分别记录了待写回数据的客户端和待读取数据的客户端，如下所示：

```
struct redisServer {  
    ...  
    list *clients_pending_write; //待写回数据的客户端  
    list *clients_pending_read; //待读取数据的客户端  
    ...  
}
```

你要知道，Redis server在接收到客户端请求和给客户端返回数据的过程中，会根据一定条件，推迟客户端的读写操作，并分别把待读写的客户端保存到这两个列表中。然后，Redis server在每次进入事件循环前，会再把列表中的客户端添加到io\_threads\_list数组中，交给IO线程进行处理。

所以接下来，我们就先来看下，Redis是如何推迟客户端的读写操作，并把这些客户端添加到clients\_pending\_write和clients\_pending\_read这两个列表中的。

## 如何推迟客户端读操作？

Redis server在和一个客户端建立连接后，就会开始监听这个客户端上的可读事件，而处理可读事件的回调函数是**readQueryFromClient**。我在[第11讲](#)中给你介绍了这个过程，你可以再去回顾下。

那么这里，我们再来看下Redis 6.0版本中的readQueryFromClient函数。这个函数一开始会先从传入参数conn中获取客户端c，紧接着就调用postponeClientRead函数，来判断是否推迟从客户端读取数据。这部分的执行逻辑如下所示：

```
void readQueryFromClient(connection *conn) {  
    client *c = connGetPrivateData(conn); //从连接数据结构中获取客户  
    ...  
    if (postponeClientRead(c)) return; //判断是否推迟从客户端读取数据  
    ...  
}
```

现在，我们就来看下**postponeClientRead函数**的执行逻辑。这个函数会根据四个条件判断能否推迟从客户端读取数据。

### 条件一：全局变量server的io\_threads\_active值为1

这表示多IO线程已经激活。我刚才说过，这个变量值在initThreadedIO函数中是会被初始化为0的，也就是说，多IO线程初始化后，默认还没有激活（我一会儿还会给你介绍这个变量值何时被设置为1）。

### 条件二：全局变量server的io\_threads\_do\_read值为1

这表示多IO线程可以用于处理延后执行的客户端读操作。这个变量值是在Redis配置文件redis.conf中，通过配置项io-threads-do-reads设置的，默认值为no，也就是说，多IO线程机制默认并不会用于客户端读操

作。所以，如果你想用多IO线程处理客户端读操作，就需要把io-threads-do-reads配置项设为yes。

### 条件三：ProcessingEventsWhileBlocked变量值为0

这表示processEventsWhileBlocked函数没有在执行。ProcessingEventsWhileBlocked是一个全局变量，它会在processEventsWhileBlocked函数执行时被设置为1，在processEventsWhileBlocked函数执行完成时被设置为0。

而processEventsWhileBlocked函数是在[networking.c](#)文件中实现的。当Redis在读取RDB文件或是AOF文件时，这个函数会被调用，用来处理事件驱动框架捕获到的事件。这样就避免了因读取RDB或AOF文件造成Redis阻塞，而无法及时处理事件的情况。所以，当processEventsWhileBlocked函数执行处理客户端可读事件时，这些客户端读操作是会被推迟执行的。

### 条件四：客户端现有标识不能有CLIENT\_MASTER、CLIENT\_SLAVE和CLIENT\_PENDING\_READ

其中，CLIENT\_MASTER和CLIENT\_SLAVE标识分别表示客户端是用于主从复制的客户端，也就是说，这些客户端不会推迟读操作。CLIENT\_PENDING\_READ本身就表示一个客户端已经被设置为推迟读操作了，所以，对于已带有CLIENT\_PENDING\_READ标识的客户端，postponeClientRead函数就不会再推迟它的读操作了。

总之，只有前面这四个条件都满足了，postponeClientRead函数才会推迟当前客户端的读操作。具体来说，postponeClientRead函数会给该客户端设置CLIENT\_PENDING\_READ标识，并调用listAddNodeHead函数，把这个客户端添加到全局变量server的clients\_pending\_read列表中。

我把postponeClientRead函数的代码放在这里，你可以看下。

```
int postponeClientRead(client *c) {
    //判断IO线程是否激活，
    if (server.io_threads_active && server.io_threads_do_reads &&
        !ProcessingEventsWhileBlocked &&
        !(c->flags & (CLIENT_MASTER|CLIENT_SLAVE|CLIENT_PENDING_READ)))
    {
        c->flags |= CLIENT_PENDING_READ; //给客户端的flag添加CLIENT_PENDING_READ标记，表示推迟该客户端的读操作
        listAddNodeHead(server.clients_pending_read,c); //将客户端添加到clients_pending_read列表中
        return 1;
    } else {
        return 0;
    }
}
```

好，现在你已经知道，Redis是在客户端读事件回调函数readQueryFromClient中，通过调用postponeClientRead函数来判断和推迟客户端读操作。下面，我再带你来看下Redis是如何推迟客户端写操作的。

### 如何推迟客户端写操作？

Redis在执行了客户端命令，要给客户端返回结果时，会调用**addReply函数**将待返回结果写入客户端输出缓冲区。



而在addReply函数的一开始，该函数会调用**prepareClientToWrite函数**，来判断是否推迟执行客户端写操作。下面代码展示了addReply函数对prepareClientToWrite函数的调用，你可以看下。

```
void addReply(client *c, robj *obj) {
    if (prepareClientToWrite(c) != C_OK) return;
    ...
}
```

所以这里，我们继续来看下prepareClientToWrite函数。这个函数会根据客户端设置的标识进行一系列的判断。其中，该函数会调用**clientHasPendingReplies函数**，判断当前客户端是否还有留存在输出缓冲区中的数据等待写回。

如果没有的话，那么，prepareClientToWrite就会调用**clientInstallWriteHandler函数**，再进一步判断能否推迟该客户端写操作。下面的代码展示了这一调用过程，你可以看下。

```
int prepareClientToWrite(client *c) {
    ...
    //如果当前客户端没有待写回数据，调用clientInstallWriteHandler函数
    if (!clientHasPendingReplies(c)) clientInstallWriteHandler(c);
    return C_OK;
}
```

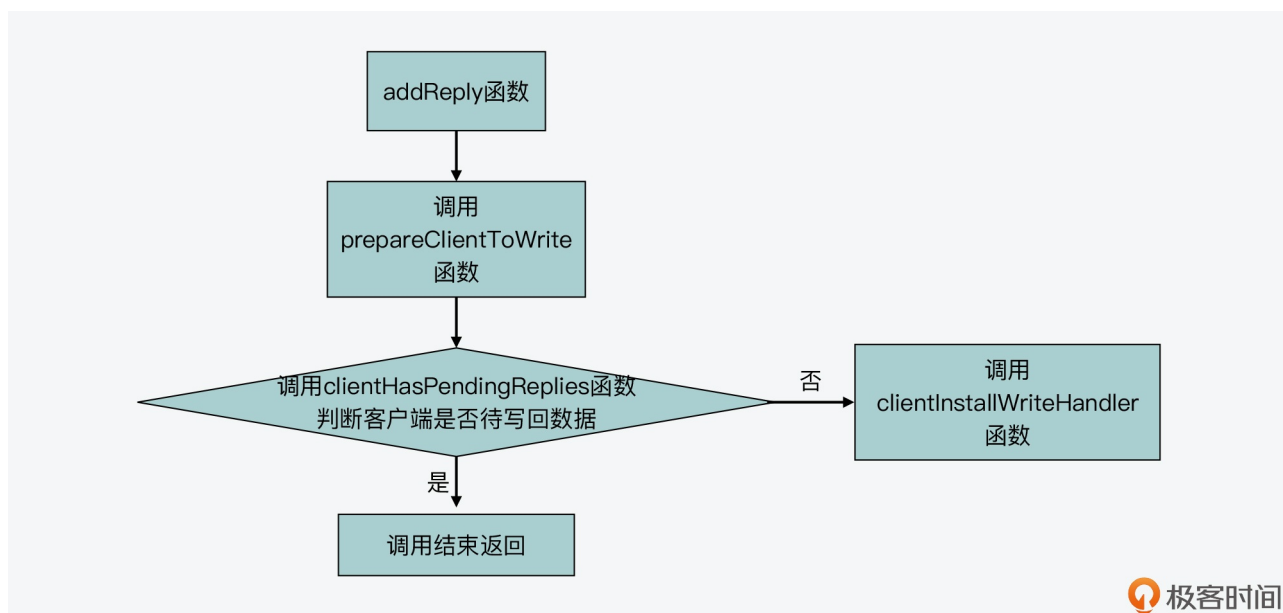
那么这样一来，我们其实就知道了，能否推迟客户端写操作，最终是由clientInstallWriteHandler函数来决定的，这个函数会判断两个条件。

- **条件一**：客户端没有设置过CLIENT\_PENDING\_WRITE标识，即没有被推迟过执行写操作。
- **条件二**：客户端所在实例没有进行主从复制，或者客户端所在实例是主从复制中的从节点，但全量复制的RDB文件已经传输完成，客户端可以接收请求。

一旦这两个条件都满足了，clientInstallWriteHandler函数就会把客户端标识设置为CLIENT\_PENDING\_WRITE，表示推迟该客户端的写操作。同时，clientInstallWriteHandler函数会把这个客户端添加到全局变量server的待写回客户端列表中，也就是clients\_pending\_write列表中。

```
void clientInstallWriteHandler(client *c) {
    //如果客户端没有设置过CLIENT_PENDING_WRITE标识，并且客户端没有在进行主从复制，或者客户端是主从复制中的从节点，已经能接收
    if (!(c->flags & CLIENT_PENDING_WRITE) &&
        (c->replstate == REPL_STATE_NONE ||
         (c->replstate == SLAVE_STATE_ONLINE && !c->repl_put_online_on_ack)))
    {
        //将客户端的标识设置为待写回，即CLIENT_PENDING_WRITE
        c->flags |= CLIENT_PENDING_WRITE;
        listAddNodeHead(server.clients_pending_write,c); //将可获得加入clients_pending_write列表
    }
}
```

为了便于你更好地理解，我画了一张图，展示了Redis推迟客户端写操作的函数调用关系，你可以再回顾下。



不过，当Redis使用clients\_pending\_read和clients\_pending\_write两个列表，保存了推迟执行的客户端后，**这些客户端又是如何分配给多IO线程执行的呢？**这就和下面两个函数相关了。

- handleClientsWithPendingReadsUsingThreads函数：该函数主要负责将clients\_pending\_read列表中的客户端分配给IO线程进行处理。
- handleClientsWithPendingWritesUsingThreads函数：该函数主要负责将clients\_pending\_write列表中的客户端分配给IO线程进行处理。

所以接下来，我们就来看下这两个函数的具体操作。

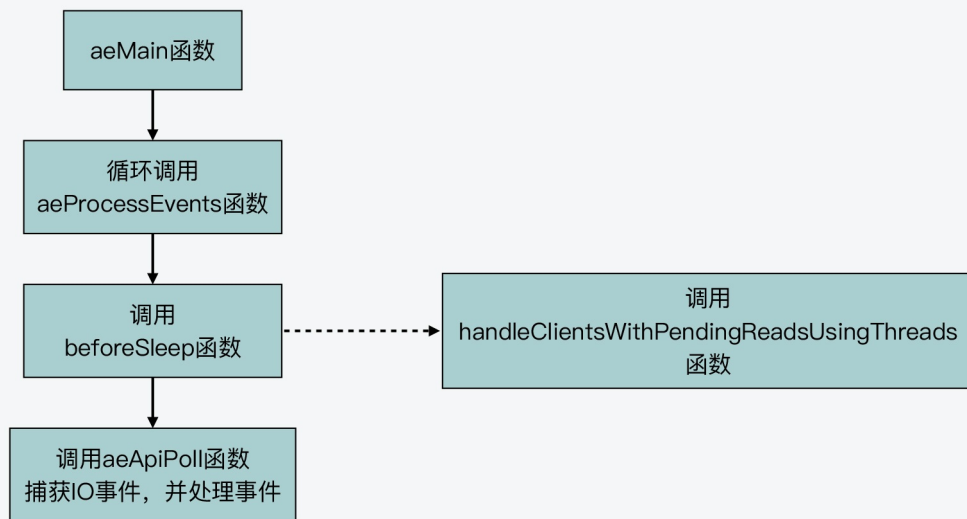
## 如何把待读客户端分配给IO线程执行？

首先，我们来了解**handleClientsWithPendingReadsUsingThreads函数**。这个函数是在beforeSleep函数中调用的。

在Redis 6.0版本的代码中，事件驱动框架同样是调用aeMain函数来执行事件循环流程，该循环流程会调用aeProcessEvents函数处理各种事件。而在aeProcessEvents函数实际调用aeApiPoll函数捕获IO事件之前，beforeSleep函数会被调用。

这个过程如下图所示，你可以看下。





handleClientsWithPendingReadsUsingThreads函数的主要执行逻辑可以分成四步。

**第一步**，该函数会先根据全局变量server的io\_threads\_active成员变量，判定IO线程是否激活，并且根据server的io\_threads\_do\_reads成员变量，判定用户是否设置了Redis可以用IO线程处理待读客户端。只有在IO线程激活，并且IO线程可以用于处理待读客户端时，handleClientsWithPendingReadsUsingThreads函数才会继续执行，否则该函数就直接结束返回了。这一步的判断逻辑如以下代码所示：

```
if (!server.io_threads_active || !server.io_threads_do_reads)
    return 0;
```

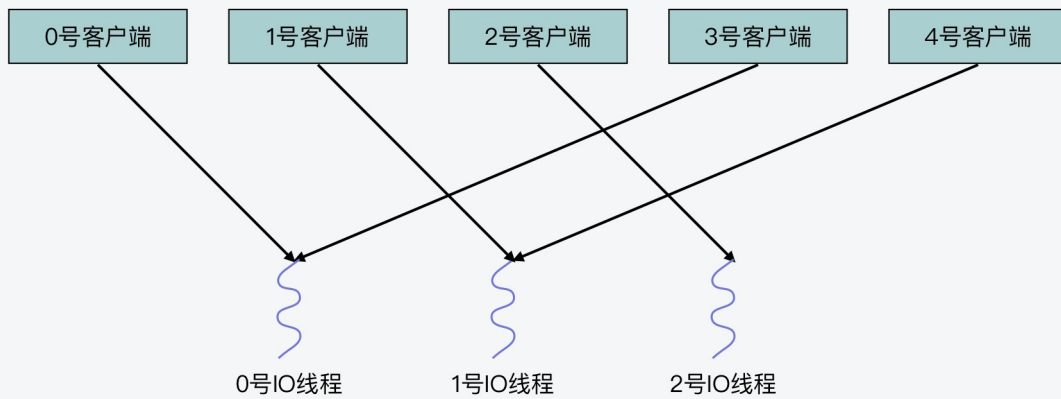
**第二步**，handleClientsWithPendingReadsUsingThreads函数会获取clients\_pending\_read列表的长度，这代表了要处理的待读客户端个数。然后，该函数会从clients\_pending\_read列表中逐一取出待处理的客户端，并用客户端在列表中的序号，对IO线程数量进行取模运算。

这样一来，我们就可以根据取模得到的余数，把该客户端分配给对应的IO线程进行处理。紧接着，handleClientsWithPendingReadsUsingThreads函数会**调用listAddNodeTail函数，把分配好的客户端添加到io\_threads\_list列表的相应元素中**。我刚才给你介绍过，io\_threads\_list数组的每个元素是一个列表，对应保存了每个IO线程要处理的客户端。

为了便于你理解，我来给你举个例子。

假设IO线程数量设置为3，clients\_pending\_read列表中一共有5个待读客户端，它们在列表中的序号分别是0，1，2，3和4。在这一步中，0号到4号客户端对线程数量3取模的结果分别是0，1，2，0，1，这也对应了即将处理这些客户端的IO线程编号。这也就是说，0号客户端由0号线程处理，1号客户端有1号线程处理，以此类推。你可以看到，这个分配方式其实就是把待处理客户端，以**轮询方式**逐一分配给各个IO线程。

我画了下面这张图，展示了这个分配结果，你可以再看下。



以下代码展示的就是以轮询方式将客户端分配给IO线程的执行逻辑：

```
int processed = listLength(server.clients_pending_read);
listRewind(server.clients_pending_read,&li);
int item_id = 0;
while((ln = listNext(&li))) {
    client *c = listNodeValue(ln);
    int target_id = item_id % server.io_threads_num;
    listAddNodeTail(io_threads_list[target_id],c);
    item_id++;
}
```

这样，当handleClientsWithPendingReadsUsingThreads函数完成客户端的IO线程分配之后，它会将IO线程的操作标识设置为**读操作**，也就是IO\_THREADS\_OP\_READ。然后，它会遍历io\_threads\_list数组中的每个元素列表长度，等待每个线程处理的客户端数量，赋值给io\_threads\_pending数组。这一过程如下所示：

```
io_threads_op = IO_THREADS_OP_READ;
for (int j = 1; j < server.io_threads_num; j++) {
    int count = listLength(io_threads_list[j]);
    io_threads_pending[j] = count;
}
```

**第三步**，handleClientsWithPendingReadsUsingThreads函数会将io\_threads\_list数组0号列表（也就是io\_threads\_list[0]元素）中的待读客户端逐一取出来，并调用readQueryFromClient函数进行处理。

其实，handleClientsWithPendingReadsUsingThreads函数本身就是由IO主线程执行的，而io\_threads\_list数组对应的0号线程正是IO主线程，所以，这里就是让主IO线程来处理它的待读客户端。

```
listRewind(io_threads_list[0],&li); //获取0号列表中的所有客户端
while((ln = listNext(&li))) {
    client *c = listNodeValue(ln);
    readQueryFromClient(c->conn);
}
```

```
}  
listEmpty(io_threads_list[0]); //处理完后，清空0号列表
```

紧接着，handleClientsWithPendingReadsUsingThreads函数会执行一个while(1)循环，等待所有IO线程完成待读客户端的处理，如下所示：

```
while(1) {  
    unsigned long pending = 0;  
    for (int j = 1; j < server.io_threads_num; j++)  
        pending += io_threads_pending[j];  
    if (pending == 0) break;  
}
```

**第四步**，handleClientsWithPendingReadsUsingThreads函数会再次遍历一遍clients\_pending\_read列表，依次取出其中的客户端。紧接着，它会判断客户端的标识中是否有CLIENT\_PENDING\_COMMAND。如果有CLIENT\_PENDING\_COMMAND标识，表明该客户端中的命令已经被某一个IO线程解析过，已经可以被执行了。

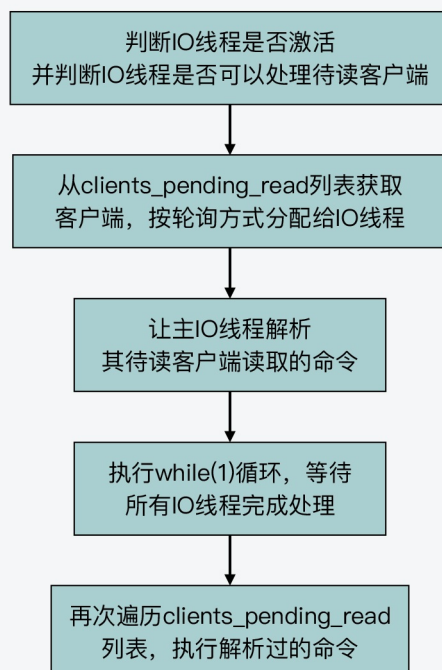
此时，handleClientsWithPendingReadsUsingThreads函数会调用processCommandAndResetClient函数执行命令。最后，它会直接调用processInputBuffer函数解析客户端中所有命令并执行。

这部分的代码逻辑如下所示，你可以看下。

```
while(listLength(server.clients_pending_read)) {  
    ln = listFirst(server.clients_pending_read);  
    client *c = listNodeValue(ln);  
    ...  
    //如果命令已经解析过，则执行该命令  
    if (c->flags & CLIENT_PENDING_COMMAND) {  
        c->flags &= ~CLIENT_PENDING_COMMAND;  
        if (processCommandAndResetClient(c) == C_ERR) {  
            continue;  
        }  
    }  
    //解析并执行所有命令  
    processInputBuffer(c);  
}
```

好了，到这里，你就了解了clients\_pending\_read列表中的待读客户端，是如何经过以上四个步骤来分配给IO线程进行处理的。下图展示了这个主要过程，你可以再回顾下。

### handleClientsWithPendingReadsUsingThreads函数基本流程



那么，接下来，我们再来看下待写客户端的分配和处理。

## 如何把待写客户端分配给IO线程执行？

和待读客户端的分配处理类似，待写客户端分配处理是由

**handleClientsWithPendingWritesUsingThreads函数**来完成的。该函数也是在beforeSleep函数中被调用的。

handleClientsWithPendingWritesUsingThreads函数的主要流程同样也可以分成4步，其中，第2、3和4步的执行逻辑，和handleClientsWithPendingReadsUsingThreads函数类似。

简单来说，在第2步，handleClientsWithPendingWritesUsingThreads函数会把待写客户端，按照**轮询方式**分配给IO线程，添加到io\_threads\_list数组各元素中。

然后，在第3步，handleClientsWithPendingWritesUsingThreads函数会让主IO线程处理其待写客户端，并执行while(1)循环等待所有IO线程完成处理。

在第4步，handleClientsWithPendingWritesUsingThreads函数会再次检查clients\_pending\_write列表中，是否还有待写的客户端。如果有的话，并且这些客户端还有留存在缓冲区中的数据，那么，handleClientsWithPendingWritesUsingThreads函数就会调用connSetWriteHandler函数注册可写事件，而这个可写事件对应的回调函数是**sendReplyToClient函数**。

等到事件循环流程再次执行时，刚才handleClientsWithPendingWritesUsingThreads函数注册的可写事件就会被处理，紧接着sendReplyToClient函数会执行，它会直接调用writeToClient函数，把客户端缓冲区中的数据写回。

这里，**你需要注意的是**，connSetWriteHandler函数最终会映射为connSocketSetWriteHandler函数，而connSocketSetWriteHandler函数是在[connection.c](#)文件中实现的。connSocketSetWriteHandler函数会调用aeCreateFileEvent函数创建AE\_WRITABLE事件，这就是刚才介绍的可写事件的注册（关于aeCreateFileEvent函数的使用，你也可以再回顾下第11讲）。

不过，和handleClientsWithPendingReadsUsingThreads函数不同的是在第1步，handleClientsWithPendingWritesUsingThreads函数，**会判断IO线程数量是否为1，或者待写客户端数量是否小于IO线程数量的2倍。**

如果这两个条件中有一个条件成立，那么handleClientsWithPendingWritesUsingThreads函数就不会用多线程来处理客户端了，而是会调用handleClientsWithPendingWrites函数由主IO线程直接处理待写客户端。这样做的目的，主要是为了在待写客户端数量不多时，避免采用多线程，从而**节省CPU开销**。

这一步的条件判断逻辑如下所示。其中，stopThreadedIOIfNeeded函数主要是用来判断待写客户端数量，是否不足为IO线程数量的2倍。

```
if (server.io_threads_num == 1 || stopThreadedIOIfNeeded()) {  
    return handleClientsWithPendingWrites();  
}
```

另外，handleClientsWithPendingWritesUsingThreads函数在第1步中，还会**判断IO线程是否已激活**。如果没有激活，它就会调用startThreadedIO函数，把全局变量server的io\_threads\_active成员变量值设置为1，表示IO线程已激活。这步判断操作如下所示：

```
if (!server.io_threads_active) startThreadedIO();
```

总之你要知道的就是，Redis是通过handleClientsWithPendingWritesUsingThreads函数，把待写客户端按轮询方式分配给各个IO线程，并由它们来负责写回数据的。

## 小结

今天这节课，我给你介绍了Redis 6.0中新设计实现的多IO线程机制。这个机制的设计主要是为了使用多个IO线程，来并发处理客户端读取数据、解析命令和写回数据。使用了多线程后，Redis就可以充分利用服务器的多核特性，从而**提高IO效率**。

总结来说，Redis 6.0先是在初始化过程中，根据用户设置的IO线程数量，创建对应数量的IO线程。

当Redis server初始化完成后正常运行时，它会在readQueryFromClient函数中通过调用postponeClientRead函数来决定是否推迟客户端读操作。同时，Redis server会在addReply函数中通过调用prepareClientToWrite函数，来决定是否推迟客户端写操作。而待读写的客户端会被分别加入到clients\_pending\_read和clients\_pending\_write两个列表中。

这样，每当Redis server要进入事件循环流程前，都会在beforeSleep函数中分别调用handleClientsWithPendingReadsUsingThreads函数和handleClientsWithPendingWritesUsingThreads函数，将待读写客户端**以轮询方式分配给IO线程**，加入到IO线程的待处理客户端列表io\_threads\_list中。

而IO线程一旦运行后，本身会一直检测io\_threads\_list中的客户端，如果有待读写客户端，IO线程就会调用

readQueryFromClient或writeToClient函数来进行处理。

最后，我也想再提醒你一下，**多IO线程本身并不会执行命令**，它们只是利用多核并行地读取数据和解析命令，或是将server数据写回（下节课我还会结合分布式锁的原子性保证，来给你介绍这一部分的源码实现。）。所以，**Redis执行命令的线程还是主IO线程**。这一点对于你理解多IO线程机制很重要，可以避免你误解Redis有多线程同时执行命令。

这样一来，我们原来针对Redis 单个主IO线程做的优化仍然有效，比如避免bigkey、避免阻塞操作等。

## 每课一问

Redis 多IO线程机制使用startThreadedIO函数和stopThreadedIO函数，来设置IO线程激活标识io\_threads\_active为1和为0。此处，这两个函数还会对线程互斥锁数组进行解锁和加锁操作，如下所示。你知道为什么这两个函数要执行解锁和加锁操作么？

```
void startThreadedIO(void) {
    ...
    for (int j = 1; j < server.io_threads_num; j++)
        pthread_mutex_unlock(&io_threads_mutex[j]); //给互斥锁数组中每个线程对应的互斥锁做解锁操作
    server.io_threads_active = 1;
}

void stopThreadedIO(void) {
    ...
    for (int j = 1; j < server.io_threads_num; j++)
        pthread_mutex_lock(&io_threads_mutex[j]); //给互斥锁数组中每个线程对应的互斥锁做加锁操作
    server.io_threads_active = 0;
}
```

欢迎在留言区分享你的答案和思考过程，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

## 精选留言：

- Kaito 2021-08-24 18:05:54
  - 1、Redis 6.0 之前，处理客户端请求是单线程，这种模型的缺点是，只能用到「单核」CPU。如果并发量很高，那么在读写客户端数据时，容易引发性能瓶颈，所以 Redis 6.0 引入了多 IO 线程解决这个问题
  - 2、配置文件开启 io-threads N 后，Redis Server 启动时，会启动 N - 1 个 IO 线程（主线程也算一个 IO 线程），这些 IO 线程执行的逻辑是 networking.c 的 IOThreadMain 函数。但默认只开启多线程「写」client socket，如果要开启多线程「读」，还需配置 io-threads-do-reads = yes
  - 3、Redis 在读取客户端请求时，判断如果开启了 IO 多线程，则把这个 client 放到 clients\_pending\_read 链表中（postponeClientRead 函数），之后主线程在处理每次事件循环之前，把链表数据轮询放到 IO 线程的链表（io\_threads\_list）中
  - 4、同样地，在写回响应时，是把 client 放到 clients\_pending\_write 中（prepareClientToWrite 函数），执行事件循环之前把数据轮询放到 IO 线程的链表（io\_threads\_list）中
  - 5、主线程把 client 分发到 IO 线程时，自己也会读写客户端 socket（主线程也要分担一部分读写操作），之后「等待」所有 IO 线程完成读写，再由主线程「串行」执行后续逻辑

6、每个 IO 线程，不停地从 io\_threads\_list 链表中取出 client，并根据指定类型读、写 client socket

7、IO 线程在处理读、写 client 时有些许差异，如果 write\_client\_peding < io\_threads \* 2，则直接由「主线程」负责写，不再交给 IO 线程处理，从而节省 CPU 消耗

8、Redis 官方建议，服务器最少 4 核 CPU 才建议开启 IO 多线程，4 核 CPU 建议开 2-3 个 IO 线程，8 核 CPU 开 6 个 IO 线程，超过 8 个线程性能提升不大

9、Redis 官方表示，开启多 IO 线程后，性能可提升 1 倍。当然，如果 Redis 性能足够用，没必要开 IO 线程

课后题：为什么 startThreadedIO / stopThreadedIO 要执行加解锁？

既然涉及到加锁操作，必然是为了「互斥」从而控制某些逻辑。可以在代码中检索这个锁变量，看存在哪些逻辑对 io\_threads\_mutex 操作了加解锁。

跟踪代码可以看到，在 networking.c 的 IOThreadMain 函数，也对这个变量进行了加解锁操作，那就说明 startThreadedIO / stopThreadedIO 函数，可以控制 IOThreadMain 里逻辑的执行，IOThreadMain 代码如下。

```
void *IOThreadMain(void *myid) {
...
while(1) {
...
/* Give the main thread a chance to stop this thread. */
if (io_threads_pending[id] == 0) {
pthread_mutex_lock(&io_threads_mutex[id]);
pthread_mutex_unlock(&io_threads_mutex[id]);
continue;
}
// 读写 client socket
// ...
}
```

这个函数正是 IO 多线程的主逻辑。

从注释可以看到，这是为了给主线程停止 IO 线程的机会。也就是说，这里的目的是为了让主线程可以控制 IO 线程的开启 / 暂停。

因为每次 IO 线程在执行时必须先拿到锁，才能执行后面的逻辑，如果主线程执行了 stopThreadedIO，就会先拿到锁，那么 IOThreadMain 函数在执行时就会因为拿不到锁阻塞「等待」，这就达到了 stop IO 线程的目的。

同样地，调用 startThreadedIO 函数后，会释放锁，IO 线程就可以拿到锁，继续「恢复」执行。

[4赞]

- 可怜大灰狼 2021-08-24 12:22:36  
networking.c中IOThreadMain方法有如下一小段代码：



```
/* Give the main thread a chance to stop this thread. */  
if (getIOPendingCount(id) == 0) {  
    pthread_mutex_lock(&io_threads_mutex[id]);  
    pthread_mutex_unlock(&io_threads_mutex[id]);  
    continue;  
}
```

就像代码里说的，给主线程暂停子线程的机会。

如果主线程没有在startThreadedIO做unlock和在stopThreadedIO做lock，主线程也无法暂停和开始子线程，进而会导致cpu资源浪费。 [1赞]

- Milittle 2021-08-24 01:26:49

课后题我的猜测：就是对于一个线程完整的释放和触发，启动线程，将线程的mutex释放，意味着你在这个线程中，去访问一些共享资源，那么你可以使用这个mutex。关闭线程，将线程的mutex获取，让线程中其他获取mutex的能力失效。一点猜测，不知道对不对。