Rabbitmq-c实现中的心酸故事

## 背景

【一切为了让程序更稳定】

测试程序：

|  |
| --- |
| main()  {  ret = amqp\_connection\_create();  ret = amqp\_channel\_open();  ret = amqp\_exchange\_declare();  ret = amqp\_queue\_declare();  ret = amqp\_queue\_bind();  while (1) {  ret = amqp\_basic\_publish();  }  } |

## 为什么放弃uv\_try\_write

【一切源于内部逻辑BUG】

现象：rabbitmq客户端性能测试时，大概率出现core dump的情况。

分析过程：

* 堆栈信息

|  |
| --- |
| Thread 2 (Thread 0x7ffff796d700 (LWP 121670)):  #0 uv\_\_write (stream=0x606998) at src/unix/stream.c:874  #1 0x00007ffff7bd27a6 in uv\_\_stream\_io (loop=<value optimized out>, w=0x606a20, events=<value optimized out>) at src/unix/stream.c:1332  #2 0x00007ffff7bc8dcb in uv\_\_run\_pending (loop=0x606010, mode=UV\_RUN\_DEFAULT) at src/unix/core.c:772  #3 uv\_run (loop=0x606010, mode=UV\_RUN\_DEFAULT) at src/unix/core.c:354  #4 0x00007ffff7de82a4 in on\_run (arg=0x606010)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_connection.c:175  #5 0x0000003853807aa1 in start\_thread () from /lib64/libpthread.so.0  #6 0x00000038534e8aad in clone () from /lib64/libc.so.6  Thread 1 (Thread 0x7ffff7990700 (LWP 121648)):  #0 0x000000385380e7cd in write () from /lib64/libpthread.so.0  #1 0x00007ffff7bd12d0 in uv\_\_write (stream=0x606998) at src/unix/stream.c:843  #2 0x00007ffff7bd17e7 in uv\_write2 (req=0x7fffffffd0b0, stream=0x606998, bufs=0x7fffffffd1d0, nbufs=<value optimized out>,  send\_handle=<value optimized out>, cb=<value optimized out>) at src/unix/stream.c:1472  #3 0x00007ffff7bd191b in uv\_try\_write (stream=0x606998, bufs=0x7fffffffd1d0, nbufs=1) at src/unix/stream.c:1522  #4 0x00007ffff7df690d in conn\_write (c=0x606400, data=0x6094a0, len=30)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_io.c:21  #5 0x00007ffff7dee8de in amqp\_publish\_frame (channel=0x606450, id=3932200, decoded=0x7fffffffd400, properties=0x6098b0, body=...,  encoded=0x7fffffffd3f0)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_frame.c:1590  #6 0x00007ffff7de7bfe in amqp\_basic\_publish (channel=0x606450, exchange=0x403e52 "HelloWorld", routing\_key=0x403f11 "A.B", mandatory=0,  immediate=0, properties=0x6098b0, body=...)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_channel.c:863  #7 0x00000000004013c3 in main (argc=5, argv=0x7fffffffd5e8) at main.c:112  (gdb) p buf  $1 = (uv\_buf\_t \*) 0x404  (gdb) f 1  #1 0x00007ffff7bd27a6 in uv\_\_stream\_io (loop=<value optimized out>, w=0x606a20, events=<value optimized out>) at src/unix/stream.c:1332  1332 uv\_\_write(stream);  (gdb) f 0  #0 uv\_\_write (stream=0x606998) at src/unix/stream.c:874  874 size\_t len = buf->len;  (gdb) set print pretty  (gdb) p \*req  $3 = {  data = 0x7fffffffd100,  type = UV\_WRITE,  active\_queue = {0x606030, 0x606030},  reserved = {0xb, 0xce000000ffffd400, 0x7fffffffd160, 0x7ffff7dee485},  cb = 0x7ffff7bd10e0 <uv\_try\_write\_cb>,  send\_handle = 0x0,  handle = 0x606998,  queue = {0x606a58, 0x606a58},  write\_index = 0,  bufs = 0x7fffffffd130,  nbufs = 1,  error = 0,  bufsml = {{  base = 0x6094a0 "\001",  len = 30  }, {  base = 0x1200001000 <Address 0x1200001000 out of bounds>,  len = 884763262976  }, {  base = 0x400 <Address 0x400 out of bounds>,  len = 140737488345371  }, {  base = 0x7fffffffd1f0 "\240\323\377\377\377\177",  len = 140737351968699  }}  }  (gdb) p req->bufsml  $4 = {{  base = 0x6094a0 "\001",  len = 30  }, {  base = 0x1200001000 <Address 0x1200001000 out of bounds>,  len = 884763262976  }, {  base = 0x400 <Address 0x400 out of bounds>,  len = 140737488345371  }, {  base = 0x7fffffffd1f0 "\240\323\377\377\377\177",  len = 140737351968699  }}  (gdb) p req->bufs  $5 = (uv\_buf\_t \*) 0x7fffffffd130  (gdb) p req->bufs[req->write\_index]  $6 = {  base = 0x6094a0 "\001",  len = 30  }  (gdb) p &(req->bufs[req->write\_index])  $7 = (uv\_buf\_t \*) 0x7fffffffd130  (gdb) p /x sizeof(uv\_buf\_t)  $11 = 0x10 |

* 问题代码

|  |
| --- |
| 相关函数  uv\_try\_write()  {  uv\_write;  …………  /\* Unqueue request, regardless of immediateness \*/  QUEUE\_REMOVE(&req.queue);  uv\_\_req\_unregister(stream->loop, &req);  if (req.bufs != req.bufsml)  uv\_\_free(req.bufs);  req.bufs = NULL;  }  uv\_write2()  {  …………  /\* If the queue was empty when this function began, we should attempt to  \* do the write immediately. Otherwise start the write\_watcher and wait  \* for the fd to become writable.  \*/  if (stream->connect\_req) {  /\* Still connecting, do nothing. \*/  }  else if (empty\_queue) {  uv\_\_write(stream);  }  else {  /\*  \* blocking streams should never have anything in the queue.  \* if this assert fires then somehow the blocking stream isn't being  \* sufficiently flushed in uv\_\_write.  \*/  assert(!(stream->flags & UV\_STREAM\_BLOCKING));  uv\_\_io\_start(stream->loop, &stream->io\_watcher, POLLOUT);  uv\_\_stream\_osx\_interrupt\_select(stream);  }  }  uv\_\_write()  {  …………  uv\_buf\_t\* buf = &(req->bufs[req->write\_index]);  size\_t len = buf->len;  } |
| 调用关系  IO复用线程：uv\_\_run\_pending🡪uv\_\_stream\_io🡪uv\_\_write  用户线程：uv\_try\_write🡪uv\_write2🡪uv\_\_write |

原因：

1. uv\_try\_write在发送消息时，如果碰到socket缓冲区满的情况，在调用write操作发送数据时，并不能完整将数据发出，继而在下次调用uv\_try\_write的时候会调用uv\_\_io\_start，触发写事件；
2. IO复用线程会触发调用uv\_\_stream\_io流程；
3. 在IO复用线程调用到uv\_buf\_t\* buf = &(req->bufs[req->write\_index]);操作之前，用户线程调用QUEUE\_REMOVE(&req.queue);将req资源进行删除；
4. 此时，IO复用线程在调用调用uv\_buf\_t\* buf = &(req->bufs[req->write\_index]);时，致使buf地址非法，从而造成了core dump情况。

上述分析，最重要一点原因是在uv\_try\_write流程中，使用栈变量req，在处理req资源的逻辑中并没有考虑周详，从而造成了频繁core dump的情况。

下一步方案：由于uv\_try\_write内部BUG，决定放弃uv\_try\_write，使用uv\_write方式进行数据发送。

## 为什么在不开启heartbeat时，无法产生write\_done操作

【一切源于蛋疼的内部机制】

现象：在定位分析完uv\_try\_write的问题后，使用uv\_write代替uv\_try\_write实现数据发送，在调试中仍然出现core dump的情况。测试程序中，在创建AMQP连接时，启动heartbeat，为了排除模块引起的问题，尝试关闭心跳来定位分析。在不开启heartbeat后，出现无法产生write\_done写回调的操作。

分析过程：

* 堆栈信息

|  |
| --- |
| (gdb) thread apply all bt  Thread 2 (Thread 0x7ffff796b700 (LWP 7967)):  #0 0x00000038534e5309 in syscall () from /lib64/libc.so.6  #1 0x00007ffff7bd628a in uv\_\_epoll\_wait (epfd=<value optimized out>, events=<value optimized out>, nevents=<value optimized out>,  timeout=<value optimized out>) at src/unix/linux-syscalls.c:321  #2 0x00007ffff7bd495b in uv\_\_io\_poll (loop=0x606010, timeout=-1) at src/unix/linux-core.c:290  #3 0x00007ffff7bc6e6c in uv\_run (loop=0x606010, mode=UV\_RUN\_DEFAULT) at src/unix/core.c:362  #4 0x00007ffff7de75a3 in on\_run (arg=0x606010)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_connection.c:180  #5 0x0000003853807aa1 in start\_thread () from /lib64/libpthread.so.0  #6 0x00000038534e8aad in clone () from /lib64/libc.so.6  Thread 1 (Thread 0x7ffff798e700 (LWP 7964)):  #0 0x0000003853484444 in \_\_memset\_sse2 () from /lib64/libc.so.6  #1 0x00007ffff7dee09f in amqp\_publish\_frame (channel=0x606450, id=3932200, decoded=0x7fffffffd400, properties=0x609930, body=...,  encoded=0x7fffffffd3f0)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_frame.c:1702  #2 0x00007ffff7de6d97 in amqp\_basic\_publish (channel=0x606450, exchange=0x403e62 "HelloWorld", routing\_key=0x403f21 "A.B", mandatory=0,  immediate=0, properties=0x609930, body=...)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_channel.c:987  #3 0x00000000004013d6 in main (argc=5, argv=0x7fffffffd5f8) at main.c:114  通过调试，可以看到w->events = 1，即epoll\_wait在等待读事件 |

* 问题代码

|  |
| --- |
| int uv\_\_next\_timeout(const uv\_loop\_t\* loop) {  const struct heap\_node\* heap\_node;  const uv\_timer\_t\* handle;  uint64\_t diff;  heap\_node = heap\_min((const struct heap\*) &loop->timer\_heap);  if (heap\_node == NULL)  return -1; /\* block indefinitely \*/  handle = container\_of(heap\_node, uv\_timer\_t, heap\_node);  if (handle->timeout <= loop->time)  return 0;  diff = handle->timeout - loop->time;  if (diff > INT\_MAX)  diff = INT\_MAX;  return diff;  }  int uv\_backend\_timeout(const uv\_loop\_t\* loop) {  if (loop->stop\_flag != 0)  return 0;  if (!uv\_\_has\_active\_handles(loop) && !uv\_\_has\_active\_reqs(loop))  return 0;  if (!QUEUE\_EMPTY(&loop->idle\_handles))  return 0;  if (!QUEUE\_EMPTY(&loop->pending\_queue))  return 0;  if (loop->closing\_handles)  return 0;  return uv\_\_next\_timeout(loop);  }  int uv\_run(uv\_loop\_t\* loop, uv\_run\_mode mode) {  …………  timeout = 0;  if ((mode == UV\_RUN\_ONCE && !ran\_pending) || mode == UV\_RUN\_DEFAULT)  timeout = uv\_backend\_timeout(loop);  uv\_\_io\_poll(loop, timeout);  }  事件加入时机  void uv\_\_io\_start(uv\_loop\_t\* loop, uv\_\_io\_t\* w, unsigned int events) {  …………  w->pevents |= events;  maybe\_resize(loop, w->fd + 1);  if (QUEUE\_EMPTY(&w->watcher\_queue))  QUEUE\_INSERT\_TAIL(&loop->watcher\_queue, &w->watcher\_queue);  if (loop->watchers[w->fd] == NULL) {  loop->watchers[w->fd] = w;  loop->nfds++;  }  }  static void uv\_\_stream\_io(uv\_loop\_t\* loop, uv\_\_io\_t\* w, unsigned int events) {  …………  /\* Write queue drained. \*/  if (QUEUE\_EMPTY(&stream->write\_queue))  uv\_\_drain(stream);  }  static void uv\_\_drain(uv\_stream\_t\* stream) {  uv\_shutdown\_t\* req;  int err;  assert(QUEUE\_EMPTY(&stream->write\_queue));  uv\_\_io\_stop(stream->loop, &stream->io\_watcher, POLLOUT);  uv\_\_stream\_osx\_interrupt\_select(stream);  ……  }  void uv\_\_io\_stop(uv\_loop\_t\* loop, uv\_\_io\_t\* w, unsigned int events) {  …………  w->pevents &= ~events;  if (w->pevents == 0) {  QUEUE\_REMOVE(&w->watcher\_queue);  QUEUE\_INIT(&w->watcher\_queue);  if (loop->watchers[w->fd] != NULL) {  assert(loop->watchers[w->fd] == w);  assert(loop->nfds > 0);  loop->watchers[w->fd] = NULL;  loop->nfds--;  w->events = 0;  }  }  else if (QUEUE\_EMPTY(&w->watcher\_queue))  QUEUE\_INSERT\_TAIL(&loop->watcher\_queue, &w->watcher\_queue);  } |
| IO复用线程函数执行顺序：uv\_run🡪 uv\_backend\_timeout🡪 uv\_\_next\_timeout🡪uv\_io\_poll  IO复用线程在发送完数据后，会判断是否队列为空uv\_\_stream\_io🡪uv\_\_drain🡪uv\_\_io\_stop |

原因：

1. IO复用线程在发送完数据后，会判断当前write\_queue是否为空，如果为空，则执行w->pevents &= ~events;将写事件类型移除，此时最多只会存在其他类型事件（w->events = 1，即epoll\_wait在等待读事件）；
2. IO复用线程在下次程序就会处于uv\_\_epoll\_wait，等待读事件上来，同时传入到uv\_\_io\_poll中的timeout值为-1（heap\_node == NULL）；
3. 此后，在调用uv\_write时，由于watcher\_queue非空，导致写事件并不会加入；
4. 最终导致uv\_\_epoll\_wait在没有timeout的情况下，等待读事件；如果没有任何数据接收，用户数据也不会真正发送出去。

下一步方案：由于libuv内部限制，需要配合POLLIN事件（heartbeat），防止epoll\_wait无限等待。

## 在开启heartbeat后，write\_done会存在core dump的情况

【一切源于蛋疼的内部机制】

现象：在定位分析完epoll\_wait的问题后，测试程序中加入心跳机制，在调试中仍然出现core dump的情况。

分析过程：

Rabbitmq-c的心跳机制在实现过程中采用了uv\_timer，在uv\_backend\_timeout调用过程中，返回的timeout为0，防止epoll\_wait处于忙等状态。

|  |
| --- |
| int uv\_\_next\_timeout(const uv\_loop\_t\* loop) {  …………  diff = handle->timeout - loop->time;  if (diff > INT\_MAX)  diff = INT\_MAX;  return diff;  } |

* 堆栈信息

|  |
| --- |
| (gdb) bt  #0 0x00000038534325e5 in raise () from /lib64/libc.so.6  #1 0x0000003853433dc5 in abort () from /lib64/libc.so.6  #2 0x00000038534704f7 in \_\_libc\_message () from /lib64/libc.so.6  #3 0x0000003853475f3e in malloc\_printerr () from /lib64/libc.so.6  #4 0x00007f42795ea380 in write\_done (req=0x7f42740008c0, status=0)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_io.c:20  #5 0x00007f42793c2f61 in uv\_\_write\_callbacks (stream=0x1191a18) at src/unix/stream.c:961  #6 0x00007f42793c482e in uv\_\_stream\_io (loop=<value optimized out>, w=0x1191aa0, events=<value optimized out>) at src/unix/stream.c:1333  #7 0x00007f42793bae1b in uv\_\_run\_pending (loop=0x1191010, mode=UV\_RUN\_DEFAULT) at src/unix/core.c:770  #8 uv\_run (loop=0x1191010, mode=UV\_RUN\_DEFAULT) at src/unix/core.c:354  #9 0x00007f42795db4f1 in on\_run (arg=0x1191010)  at /home04/28565/VCS/VideoCloudClusterBase/Branches/P\_2016.11.14\_VideoCloudClusterBase\_3.0.X/OpenSrc/rabbitmq-c/Src/amqp\_connection.c:175  #10 0x0000003853807aa1 in start\_thread () from /lib64/libpthread.so.0  #11 0x00000038534e8aad in clone () from /lib64/libc.so.6  (gdb) p \*req  $2 = {  data = 0x0,  type = UV\_WRITE,  active\_queue = {0x1191030, 0x1191030},  reserved = {0x0, 0x0, 0x0, 0x0},  cb = 0x7f42795ea314 <write\_done>,  send\_handle = 0x0,  handle = 0x1191a18,  queue = {0x1191ae8, 0x1191ae8},  write\_index = 1,  bufs = 0x0,  nbufs = 1,  error = 0,  bufsml = {{  base = 0x7f427915bc80 "",  len = 8  }, {  base = 0x0,  len = 0  }, {  base = 0x0,  len = 0  }, {  base = 0x0,  len = 0  }}  } |

* 问题代码

|  |
| --- |
| static void uv\_\_write(uv\_stream\_t\* stream) {  …………  if ((size\_t)n < len) {  buf->base += n;  buf->len -= n;  stream->write\_queue\_size -= n;  n = 0;  /\* There is more to write. \*/  if (stream->flags & UV\_STREAM\_BLOCKING) {  /\*  \* If we're blocking then we should not be enabling the write  \* watcher - instead we need to try again.  \*/  goto start;  } else {  /\* Break loop and ensure the watcher is pending. \*/  break;  }  } else {  /\* Finished writing the buf at index req->write\_index. \*/  req->write\_index++;  assert((size\_t)n >= len);  n -= len;  assert(stream->write\_queue\_size >= len);  stream->write\_queue\_size -= len;  if (req->write\_index == req->nbufs) {  /\* Then we're done! \*/  assert(n == 0);  uv\_\_write\_req\_finish(req);  /\* TODO: start trying to write the next request. \*/  return;  }  …………  }  int uv\_write2(uv\_write\_t\* req,  uv\_stream\_t\* stream,  const uv\_buf\_t bufs[],  unsigned int nbufs,  uv\_stream\_t\* send\_handle,  uv\_write\_cb cb) {  …………  req->bufs = req->bufsml;  if (nbufs > ARRAY\_SIZE(req->bufsml))  req->bufs = uv\_\_malloc(nbufs \* sizeof(bufs[0]));  if (req->bufs == NULL)  return -ENOMEM;  memcpy(req->bufs, bufs, nbufs \* sizeof(bufs[0]));  req->nbufs = nbufs;  …………  }  void write\_done(uv\_write\_t\* req, int status)  {  if (0 > status) {  AMQP\_LOG\_ERROR("write done failed, status = %d", status);  }  if (NULL != req) {  if (NULL != req->bufsml[0].base) {  free(req->bufsml[0].base);  req->bufsml[0].base = NULL;  req->bufsml[0].len = 0;  }  free(req);  req = NULL;  }  }  void conn\_write(amqp\_connection\_t \* c, const void \* data, unsigned int len)  {  int ret = 0;  uv\_buf\_t buf;  uv\_write\_t\* req = (uv\_write\_t\*)malloc(sizeof(uv\_write\_t));  buf.base = (char\*)data;  buf.len = len;  ret = uv\_write(req, &c->handle.stream, &buf, 1, write\_done);  if (0 != ret) {  AMQP\_LOG\_ERROR("uv write failed, ret = %d", ret);  }  } |

原因：

1. uv\_write在发送消息时，如果碰到socket缓冲区满的情况，内部在调用uv\_\_write操作发送数据时，并不能完整将数据发出，从而走进了代码if ((size\_t)n < len) {逻辑中，将base值进行偏移；
2. 在IO复用线程在epoll\_wait返回后，处理write\_queue后，等这块数据剩余部分发送完时，继而会调用callback，即write\_done函数，这里在处理资源释放的逻辑中，只是判断if (NULL != req->bufsml[0].base) {，此时，base地址并非是当时malloc时候的地址，导致在free过程中产生core dump的问题。

案例DEMO：

|  |
| --- |
| int main(int argc, char\* argv[])  {  char \*p = (char\*)malloc(100);  free(p+30);  return 0;  }  堆栈信息：  Program terminated with signal 6, Aborted.  #0 0x00000038534325e5 in raise () from /lib64/libc.so.6  Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.192.el6.x86\_64 libgcc-4.4.7-17.el6.x86\_64  (gdb) bt  #0 0x00000038534325e5 in raise () from /lib64/libc.so.6  #1 0x0000003853433dc5 in abort () from /lib64/libc.so.6  #2 0x00000038534704f7 in \_\_libc\_message () from /lib64/libc.so.6  #3 0x0000003853475f3e in malloc\_printerr () from /lib64/libc.so.6  #4 0x0000003853478d8d in \_int\_free () from /lib64/libc.so.6  #5 0x0000000000400531 in main (argc=1, argv=0x7ffe6429b318) at test.c:8 |

下一步方案：由于libuv内部蛋疼的实现逻辑，需要对非一次性发送出去的数据，做特殊处理，防止释放非法指针。

方案：

* 修改libuv内部实现，记录原始data的地址
* 实现req与data之前的映射关系

|  |  |  |
| --- | --- | --- |
|  | 修改libuv内部实现，记录原始data的地址 | 实现req与data之前的映射关系 |
| 优点 | 天然的映射关系 | 用户层面可理解之间关系 |
| 缺点 | 与libuv依赖增强，并且对于libuv需要深入的理解 | 1. 效率低，每次done之后都有查询操作 2. 实现麻烦 |

## 代码演进过程

过程中需要了解的技术点：

1. gdb调试
2. free非堆变量或者非malloc出来的指针
3. 善用valgrind等工具

不合理的实现：

1. 在任何conn\_write调用后，都调用receive\_new\_frame，触发读事件；
2. 实现初期并未考虑好内存管理，存在泄漏的情况

|  |
| --- |
| valgrind --tool=memcheck --leak-check=full ./publish 172.25.7.176 5672 root dahuacloud |

## 下一步

1. 深入研究libuv
2. 通过社区对libuv的运用，找到相对合理使用的方式
3. 研究类libuv技术，如libevent
4. 对开源的思考