# 警惕 jdk8 UDP 和 Thread.interrupt 的 Bug

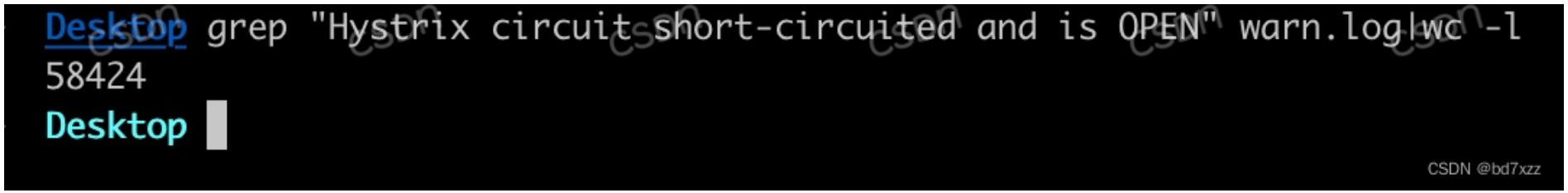
bd7xzz 已于 2023-02-11 20:32:12 修改



## 背景

线上业务在热点流量大的情况下（业务采用 Java 编程语言实现），单机偶发出现 Hystrix 熔断，接口无法提供服务。如下图所示：


```
Hystrix circuit short-circuited and is OPEN 代表Hystrix在一个窗口时间内失败了N次，发生短路。短路时间默认为 5 秒，5秒之内拒绝所有的请求，之后开始运行。
```




统计了一下 1 个小时内居然有 5w 多条。
最初怀疑依赖下游的某个 rpc 接口出现问题，但从监控来看，异常时间点多个 rpc 接口异常率都有所上升。所以考虑是自身单机存在什么问题，导致调用依赖的 rpc 接口失败，产生了大量的熔断。

这里还有很重要的一点：**在单机出现问题第一时间，摘掉了流量，过半小时后恢复流量，依旧熔断无法恢复**

- 若摘掉流量后，静置一段时间放开流量，服务恢复，证明可能是由于流量过载导致的问题，如：JVM 内存不够，gc 回收不过来，频繁 gc。这样的话流量掐掉后，gc 一段时间后内存够用，服务恢复正常。
- 若摘掉流量后，静置一段时间放开流量，服务依旧熔断，在单机出现问题的场景下，很可能是 JVM 内部的问题（包括应用自身问题或者 JVM 的一些问题，如：死锁）

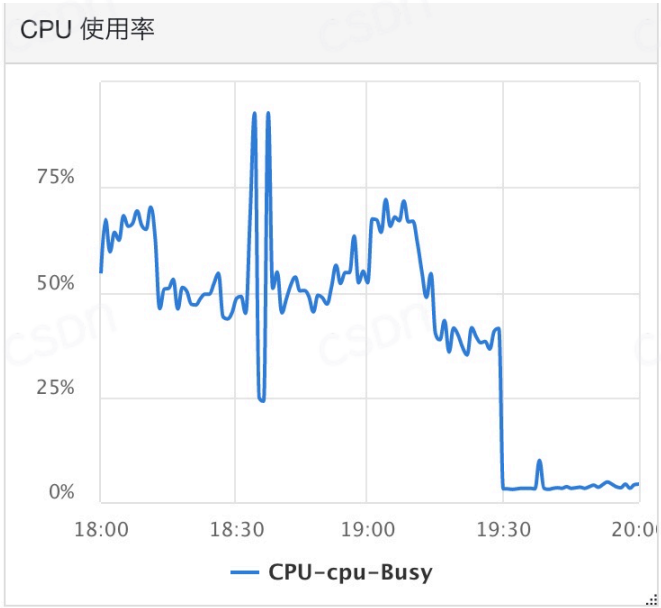
**排查**

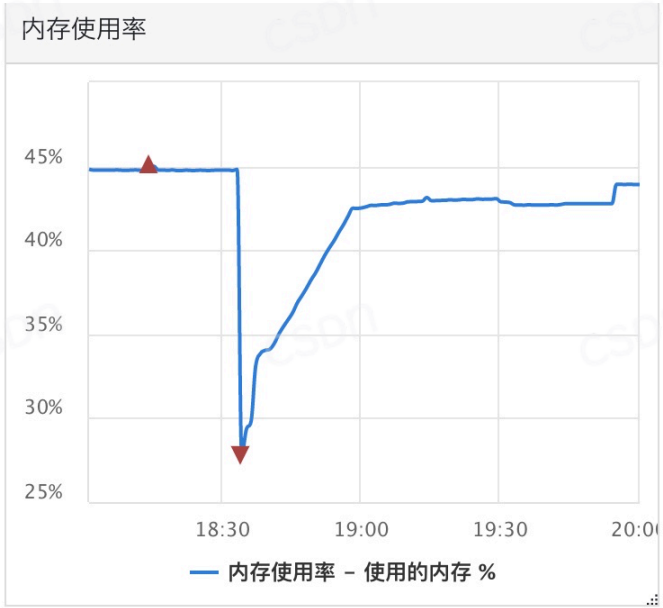
通常排查这种自身问题可从几个方面入手：

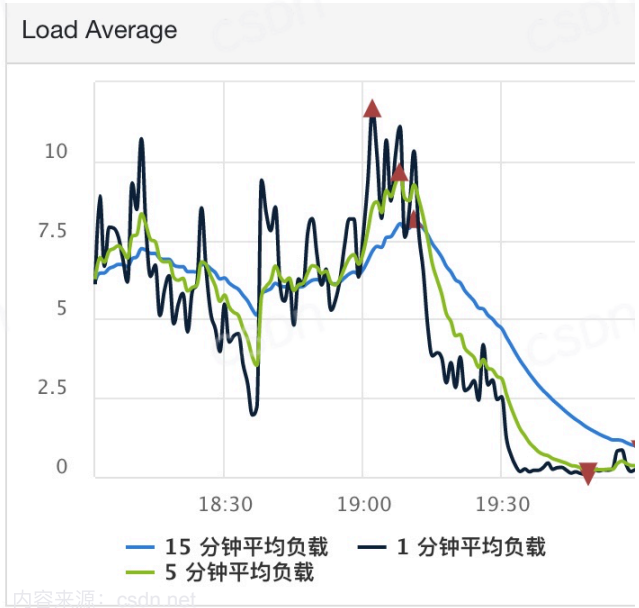
1. CPU load、使用率、内存使用率是否过高
2. 网卡出入速率是否有抖动，包括 TCP 重传
3. 磁盘 IOPS 是否过高，导致一些 sys call 阻塞
4. JVM 是否出现 stw 动作，包括不限于：gc、JIT 等
5. 是否有 oom、stack overflow、死锁等现象
6. 代码出现 bug
7. 虚拟化对应用进行 stw 动作

根据这些逐一分析：

1. 监控 CPU 和内存如下图所示：CPU 有明显的抖动，内存使用率正常（18:30 断崖式下滑是对服务进行了上线，可忽略不管），注意问题发生在 19:00 以后，可以看到 CPU 在出问题时间段内有明显的波动。

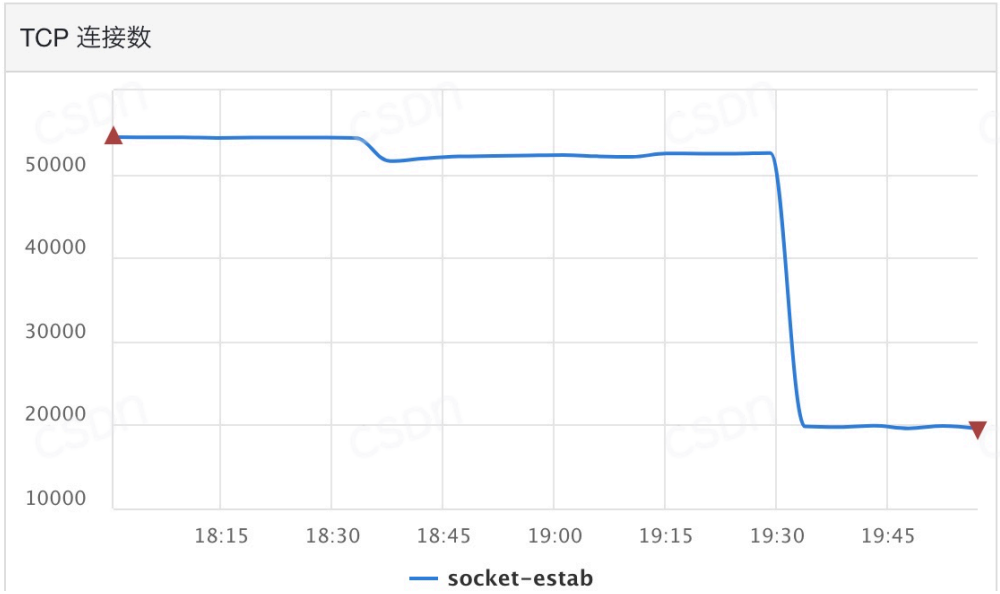


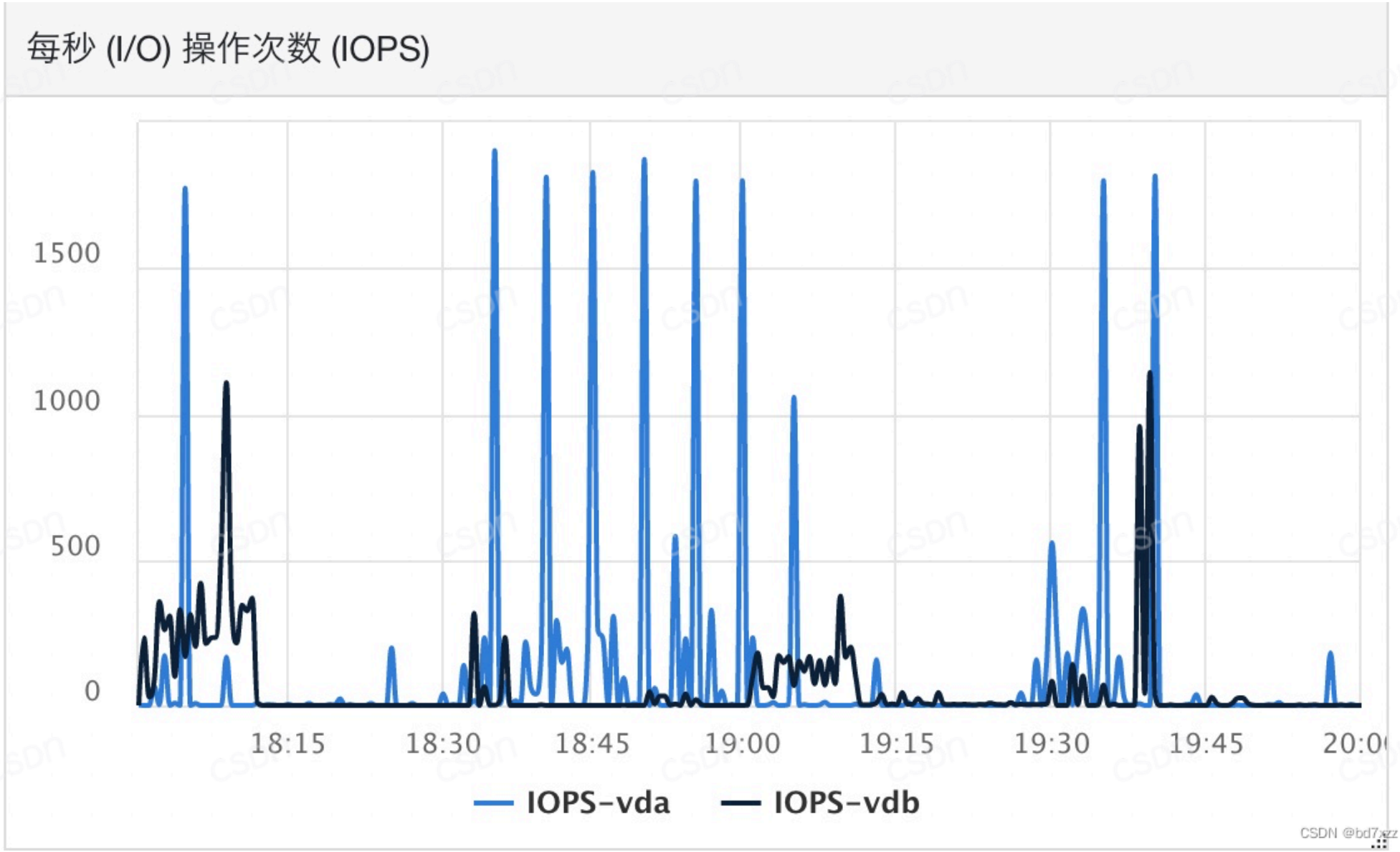



2. 查看监控网卡如下图所示：在 19:00 之后出问题的时间段网卡出入速率略有波动，可忽略。





3. 查看磁盘 IOS 如下图所示：磁盘 IOPS 虽然比较高，但 19:00 前后都比较平均。



4. 查看 gclog 如下图所示：虽然 young gc 比较频繁，但回收的比例还是比较大，耗时也不高。没有明显的其他原因导致 stw 占用时间长的迹象。

5. 打印线程栈如下图所示：jstack 打出的线程栈就很明显了，最下面直接显示了死锁！

```
Found one Java-level deadlock:
=============================
"pool-486-thread-132":
  waiting to lock monitor 0x00007fafc2a1a018 (object 0x00000006b77f2098, a java.lang.Object),
  which is held by "hystrix-[redacted]"
"hystrix[redacted]
  waiting to lock monitor 0x00007fafc22e3578 (object 0x00000006b77f2128, a java.lang.Object),
  which is held by "pool-486-thread-132"

Java stack information for the threads listed above:
===================================================
"pool-486-thread-132":
        at java.lang.Thread.blockedOn(Thread.java:239)
        - waiting to lock <0x00000006b77f2098> (a java.lang.Object)
        at java.lang.System$2.blockedOn(System.java:1252)
        at java.nio.channels.spi.AbstractInterruptibleChannel.blockedOn(AbstractInterruptibleChannel.java:211)
        at java.nio.channels.spi.AbstractInterruptibleChannel.end(AbstractInterruptibleChannel.java:198)
        at sun.nio.ch.DatagramChannelImpl.receive(DatagramChannelImpl.java:389)
        - locked <0x00000006b77f20f8> (a java.lang.Object)
        at sun.nio.ch.DatagramChannelImpl.connect(DatagramChannelImpl.java:767)
        - locked <0x00000006b77f20e0> (a java.lang.Object)
        - locked <0x00000006b77f2128> (a java.lang.Object)
        - locked <0x00000006b77f2110> (a java.lang.Object)
        - locked <0x00000006b77f20f8> (a java.lang.Object)
        at org.xbill.DNS.UDPClient.connect(UDPClient.java:107)
        at org.xbill.DNS.UDPClient.sendrecv(UDPClient.java:150)
        at org.xbill.DNS.SimpleResolver.send(SimpleResolver.java:257)
        at org.xbill.DNS.ExtendedResolver$Resolution.start(ExtendedResolver.java:95)
        at org.xbill.DNS.ExtendedResolver.send(ExtendedResolver.java:358)
        at org.xbill.DNS.Lookup.lookup(Lookup.java:477)
        at org.xbill.DNS.Lookup.resolve(Lookup.java:529)
        at org.xbill.DNS.Lookup.run(Lookup.java:543)
        at org.xbill.DNS.spi.DNSJavaNameService.lookupAllHostAddr(DNSJavaNameService.java:143)
        at org.xbill.DNS.spi.DNSJavaNameService.invoke(DNSJavaNameService.java:97)
        at com.sun.proxy.$Proxy2.lookupAllHostAddr(Unknown Source)
        at java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1324)
        at java.net.InetAddress.getAllByName0(InetAddress.java:1277)
        at java.net.InetAddress.getAllByName(InetAddress.java:1193)
        at java.net.InetAddress.getAllByName(InetAddress.java:1127)
```

6. review 了最近上线的代码，改动很小没有问题。

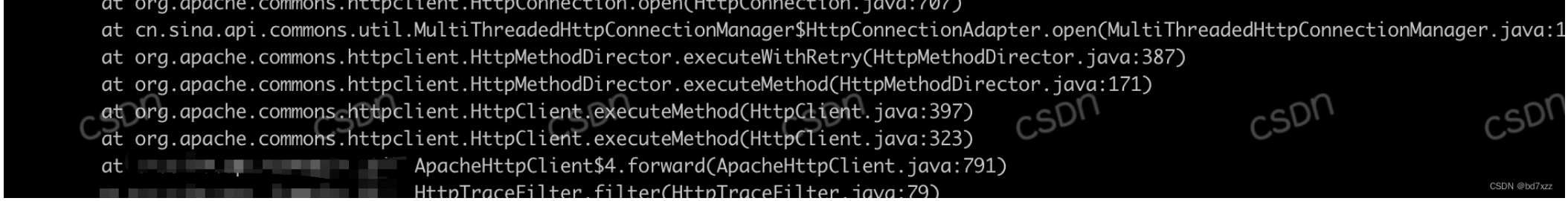
7. 由于打印线程栈出现死锁，问题很明显，不用排查虚拟化了。

问题很明显了，JVM 出现了死锁，导致 Hystirx 熔断。
从死锁的日志来看，似乎跟 DNS 相关。难道 Java 的 DNS 有啥 Bug?

**分析**

```
"pool-486-thread-132":
  waiting to lock monitor 0x00007fafc2a1a018 (object 0x00000006b77f2098, a java.lang.Object),
  which is held by "hystrix-[redacted]
"hystrix-[redacted]":
  waiting to lock monitor 0x00007fafc22e3578 (object 0x00000006b77f2128, a java.lang.Object),
  which is held by "pool-486-thread-132"

Java stack information for the threads listed above:
===================================================
"pool-486-thread-132":
        at java.lang.Thread.blockedOn(Thread.java:239)
        - waiting to lock <0x00000006b77f2098> (a java.lang.Object)
        at java.lang.System$2.blockedOn(System.java:1252)
        at java.nio.channels.spi.AbstractInterruptibleChannel.blockedOn(AbstractInterruptibleChannel.java:211)
        at java.nio.channels.spi.AbstractInterruptibleChannel.end(AbstractInterruptibleChannel.java:198)
        at sun.nio.ch.DatagramChannelImpl.receive(DatagramChannelImpl.java:389)
        - locked <0x00000006b77f20f8> (a java.lang.Object)
        at sun.nio.ch.DatagramChannelImpl.connect(DatagramChannelImpl.java:767)
        - locked <0x00000006b77f20e0> (a java.lang.Object)
        - locked <0x00000006b77f2128> (a java.lang.Object)
        - locked <0x00000006b77f2110> (a java.lang.Object)
        - locked <0x00000006b77f20f8> (a java.lang.Object)
        at org.xbill.DNS.UDPClient.connect(UDPClient.java:107)
        at org.xbill.DNS.UDPClient.sendrecv(UDPClient.java:150)
        at org.xbill.DNS.SimpleResolver.send(SimpleResolver.java:257)
        at org.xbill.DNS.ExtendedResolver$Resolution.start(ExtendedResolver.java:95)
        at org.xbill.DNS.ExtendedResolver.send(ExtendedResolver.java:358)
        at org.xbill.DNS.Lookup.lookup(Lookup.java:477)
        at org.xbill.DNS.Lookup.resolve(Lookup.java:529)
        at org.xbill.DNS.Lookup.run(Lookup.java:543)
        at org.xbill.DNS.spi.DNSJavaNameService.lookupAllHostAddr(DNSJavaNameService.java:143)
        at org.xbill.DNS.spi.DNSJavaNameService.invoke(DNSJavaNameService.java:97)
        at com.sun.proxy.$Proxy2.lookupAllHostAddr(Unknown Source)
        at java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1324)
        at java.net.InetAddress.getAllByName0(InetAddress.java:1277)
        at java.net.InetAddress.getAllByName(InetAddress.java:1193)
        at java.net.InetAddress.getAllByName(InetAddress.java:1127)
        at java.net.InetAddress.getByName(InetAddress.java:1077)
        at org.apache.commons.httpclient.protocol.ReflectionSocketFactory.createSocket(ReflectionSocketFactory.java:124)
        at org.apache.commons.httpclient.protocol.DefaultProtocolSocketFactory.createSocket(DefaultProtocolSocketFactory.java:125)
```




在死锁线程栈中很明显地标记出了 `pool-486-thread-132` 在等 `0x00000006b77f2098` 这把锁，
并且这把锁被 `hstirx-***` 线程持有着，同时 `hstirx-***` 在等待 `0x00000006b77f2128` 这把锁，并且这把锁被 `pool-486-thread-132` 线程持有着。


这俩个线程栈中同时出现了 DNS 和 DatagramChannel 相关的类，证明死锁很大一部分原因是由 DNS 在发 UDP 包的情况下产生了死锁。我们知道死锁是俩个线程对相同临界区资源竞争时，加锁不当产生的。复习下死锁的四个必要条件：

1. 互斥条件：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程所占用。
2. 请求和保持条件：当进程因请求资源而阻塞时，对已获得的资源保持不放。

3. 不剥夺条件：进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。

4. 环路等待条件：在发生死锁时，必然存在一个进程–资源的环形链。

那我们使用的 DNS 为啥会出现这样的问题？难道是 JDK 或者是 dnsjava（我们使用了 dnsjava）的 bug？
排查 bug 的最好方法就是阅读源码，我们可以通过死锁的线程栈自底向上逐层跟踪代码。

先看 `histirx-***` 的死锁线程栈：

```
"hystrix-[redacted]":
        at sun.nio.ch.DatagramChannelImpl.implCloseSelectableChannel(DatagramChannelImpl.java:1017)
        - waiting to lock <0x00000006b77f2128> (a java.lang.Object)
        at java.nio.channels.spi.AbstractSelectableChannel.implCloseChannel(AbstractSelectableChannel.java:234)
        at java.nio.channels.spi.AbstractInterruptibleChannel$1.interrupt(AbstractInterruptibleChannel.java:165)
        - locked <0x00000006b77f20b0> (a java.lang.Object)
        at java.lang.Thread.interrupt(Thread.java:922)
        - locked <0x00000006b77f2098> (a java.lang.Object)
        at java.util.concurrent.FutureTask.cancel(FutureTask.java:174)
        at java.util.concurrent.AbstractExecutorService.invokeAll(AbstractExecutorService.java:303)
        at brave.internal.WrappingExecutorService.invokeAll(WrappingExecutorService.java:38)
        at [redacted].BatchExecutor.execute(BatchExecutor.java:219)
        at [redacted]va:252)
        at [redacted]java:376)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

1. 从业务调用点开始，这里我们的业务向 BatchExecutor（公司内部封装的线程池操作）提交了线程

```
try {
    List<Future<L>> futures = executorService.invokeAll(taskList, timeout, timeUnit);
    futures.forEach(future -> {
```

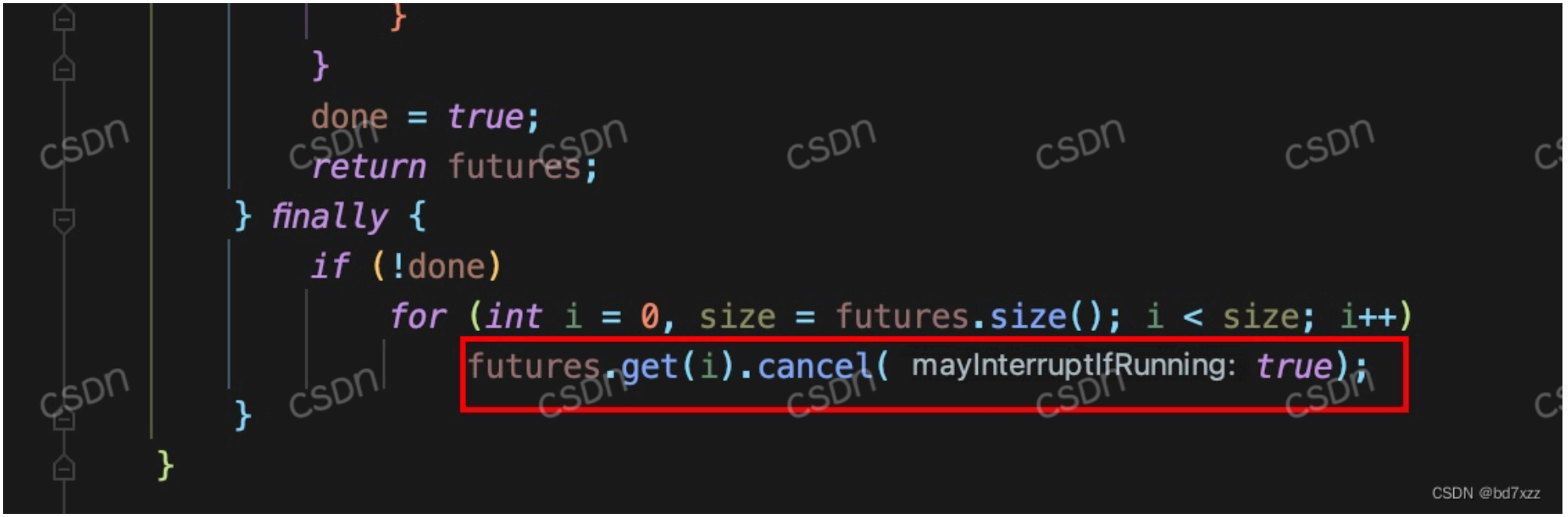
2. 本质上调用 AbstractExecutorService.invokeAll 方法，该方法在线程池中的线程执行结束时，finally 会触发 FutureTask.cancel 方法

```java
public <T> List<Future<T>> invokeAll( @NotNull Collection<? extends Callable<T>> tasks,
                                      long timeout,  @NotNull TimeUnit unit)
    throws InterruptedException {
    if (tasks == null)
        throw new NullPointerException();
    long nanos = unit.toNanos(timeout);
    ArrayList<Future<T>> futures = new ArrayList<~>(tasks.size());
    boolean done = false;
    try {
        for (Callable<T> t : tasks)
            futures.add(newTaskFor(t));

        final long deadline = System.nanoTime() + nanos;
        final int size = futures.size();

        // Interleave time checks and calls to execute in case
        // executor doesn't have any/much parallelism.
        for (int i = 0; i < size; i++) {
            execute((Runnable)futures.get(i));
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L)
                return futures;
        }

        for (int i = 0; i < size; i++) {
            Future<T> f = futures.get(i);
            if (!f.isDone()) {
                if (nanos <= 0L)
```

3. 由于在调用 cancel 方法对 mayInterruptIfRunning 实参传递的是 true，所以会执行 Thread.interrupt 方法

```
public boolean cancel(boolean mayInterruptIfRunning) {
    if (!(state == NEW &&
          UNSAFE.compareAndSwapInt( o: this, stateOffset, NEW,
              mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
        return false;
    try {    // in case call to interrupt throws exception
        if (mayInterruptIfRunning) {
            try {
                Thread t = runner;
                if (t != null)
                    t.interrupt();
            } finally { // final state
                UNSAFE.putOrderedInt( o: this, stateOffset, INTERRUPTED);
            }
        }
    } finally {
        finishCompletion();
    }
    return true;
}
```

4. Thread.interrupt 会对 blockerlock 加锁，根据线程栈，我们看到 intterupt 后会执行 DatagramChannelImpl.implCloseSelectableChannel 方法关闭 UDP 链接。

5. 而 implCloseSelectableChannel 方法要拿到 stateLock 的锁，根据线程栈，此时出现锁等待。

```
2 usages   1 override
protected void implCloseSelectableChannel() throws IOException {
    synchronized(this.stateLock) {
        if (this.state != 2) {
            nd.preClose(this.fd);
        }

        ResourceManager.afterUdpClose();
        if (this.registry != null) {
            this.registry.invalidateAll();
        }

        long var2;
        if ((var2 = this.readerThread) != 0L) {
            NativeThread.signal(var2);
        }

        if ((var2 = this.writerThread) != 0L) {
            NativeThread.signal(var2);
        }

        if (!this.isRegistered()) {
            this.kill();
        }
```

再看 `pool-486-thread-132` 的死锁线程栈：

1. 这个线程栈比较长，直接从 `at org.xbill.DNS.UDPClient.connect(UDPClient.java:107)` 开始看，可以从包名看出，这是做 dnsjava 向 DNS server 发起 UDP 请求。

```
Java stack information for the threads listed above:
===================================================
"pool-486-thread-132":
        at java.lang.Thread.blockedOn(Thread.java:239)
        - waiting to lock <0x00000006b77f2098> (a java.lang.Object)
        at java.lang.System$2.blockedOn(System.java:1252)
        at java.nio.channels.spi.AbstractInterruptibleChannel.blockedOn(AbstractInterruptibleChannel.java:211)
        at java.nio.channels.spi.AbstractInterruptibleChannel.end(AbstractInterruptibleChannel.java:198)
        at sun.nio.ch.DatagramChannelImpl.receive(DatagramChannelImpl.java:389)
        - locked <0x00000006b77f20f8> (a java.lang.Object)
        at sun.nio.ch.DatagramChannelImpl.connect(DatagramChannelImpl.java:767)
        - locked <0x00000006b77f20e0> (a java.lang.Object)
        - locked <0x00000006b77f2128> (a java.lang.Object)
        - locked <0x00000006b77f2110> (a java.lang.Object)
        - locked <0x00000006b77f20f8> (a java.lang.Object)
        at org.xbill.DNS.UDPClient.connect(UDPClient.java:107)
        at org.xbill.DNS.UDPClient.sendrecv(UDPClient.java:150)
        at org.xbill.DNS.SimpleResolver.send(SimpleResolver.java:257)
        at org.xbill.DNS.ExtendedResolver$Resolution.start(ExtendedResolver.java:95)
        at org.xbill.DNS.ExtendedResolver.send(ExtendedResolver.java:358)
        at org.xbill.DNS.Lookup.lookup(Lookup.java:477)
        at org.xbill.DNS.Lookup.resolve(Lookup.java:529)
        at org.xbill.DNS.Lookup.run(Lookup.java:543)
        at org.xbill.DNS.spi.DNSJavaNameService.lookupAllHostAddr(DNSJavaNameService.java:143)
        at org.xbill.DNS.spi.DNSJavaNameService.invoke(DNSJavaNameService.java:97)
        at com.sun.proxy.$Proxy2.lookupAllHostAddr(Unknown Source)
        at java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1324)
        at java.net.InetAddress.getAllByName0(InetAddress.java:1277)
        at java.net.InetAddress.getAllByName(InetAddress.java:1193)
        at java.net.InetAddress.getAllByName(InetAddress.java:1127)
        at java.net.InetAddress.getByName(InetAddress.java:1077)
        at org.apache.commons.httpclient.protocol.ReflectionSocketFactory.createSocket(ReflectionSocketFactory.java:124)
        at org.apache.commons.httpclient.protocol.DefaultProtocolSocketFactory.createSocket(DefaultProtocolSocketFactory.java:125)
        at org.apache.commons.httpclient.HttpConnection.open(HttpConnection.java:707)
        at cn.sina.api.commons.util.MultiThreadedHttpConnectionManager$HttpConnectionAdapter.open(MultiThreadedHttpConnectionManager.java:1391)
        at org.apache.commons.httpclient.HttpMethodDirector.executeWithRetry(HttpMethodDirector.java:387)
        at org.apache.commons.httpclient.HttpMethodDirector.executeMethod(HttpMethodDirector.java:171)
        at org.apache.commons.httpclient.HttpClient.executeMethod(HttpClient.java:397)
        at org.apache.commons.httpclient.HttpClient.executeMethod(HttpClient.java:323)
        at cn.sina.api.commons.util.ApacheHttpClient$4.forward(ApacheHttpClient.java:791)
        at cn.sina.api.commons.util.HttpTraceFilter.filter(HttpTraceFilter.java:79)
        at cn.sina.api.commons.util.ApacheHttpClient$LinkedFilterChain.forward(ApacheHttpClient.java:172)
        at cn.sina.api.commons.util.HttpTraceFilter.filter(HttpTraceFilter.java:79)
        at cn.sina.api.commons.util.ApacheHttpClient$LinkedFilterChain.forward(ApacheHttpClient.java:172)
```

```
at cn.sina.api.commons.util.ApacheHttpClient.doExecuteMethod(ApacheHttpClient.java:1281)
at cn.sina.api.commons.util.ApacheHttpClient.executeMethod(ApacheHttpClient.java:1215)
at cn.sina.api.commons.util.ApacheHttpClient$HttpClientRequestBuilder.executeGetResponseBody(ApacheHttpClient.java:1692)
at cn.sina.api.commons.util.ApacheHttpClient$HttpClientRequestBuilder.execute(ApacheHttpClient.java:1562)
```



2. connect 方法通过 DatagramChannel.connect 开启 UDP 链接。

```
.02  void
.03  connect(SocketAddress addr) throws IOException {
.04      if (!bound)
.05          bind(null);
.06      DatagramChannel channel = (DatagramChannel) key.channel();
.07      channel.connect(addr);
.08  }
.09
```



3. DatagramChannel.connect 中会锁住 readLock、writeLock、stateLock，即线程栈中的：

- readLock `locked <0x00000006b77f20f8> (a java.lang.Object)`
- writeLock `locked <0x00000006b77f2110> (a java.lang.Object)`
- stateLock `locked <0x00000006b77f2128> (a java.lang.Object)`

并锁住 bloackingLock 即线程栈中的 `locked <0x00000006b77f20e0> (a java.lang.Object)`
然后分配 Buffer 并执行 receive 方法。

```java
public DatagramChannel connect(SocketAddress var1) throws IOException {
    boolean var2 = false;
    synchronized(this.readLock) {
        synchronized(this.writeLock) {
            synchronized(this.stateLock) {
                this.ensureOpenAndUnconnected();
                InetSocketAddress var6 = Net.checkAddress(var1);
                SecurityManager var7 = System.getSecurityManager();
                if (var7 != null) {
                    var7.checkConnect(var6.getAddress().getHostAddress(), var6.getPort());
                }

                int var8 = Net.connect(this.family, this.fd, var6.getAddress(), var6.getPort());
                if (var8 <= 0) {
                    throw new Error();
                }

                this.state = 1;
                this.remoteAddress = var6;
                this.sender = var6;
                this.cachedSenderInetAddress = var6.getAddress();
                this.cachedSenderPort = var6.getPort();
                this.localAddress = Net.localAddress(this.fd);
                boolean var9 = false;
                synchronized(this.blockingLock()) {
                    try {
                        var9 = this.isBlocking();
                        ByteBuffer var11 = ByteBuffer.allocate( capacity: 1);
                        if (var9) {
                            this.configureBlocking(false);
                        }

                        do {
```

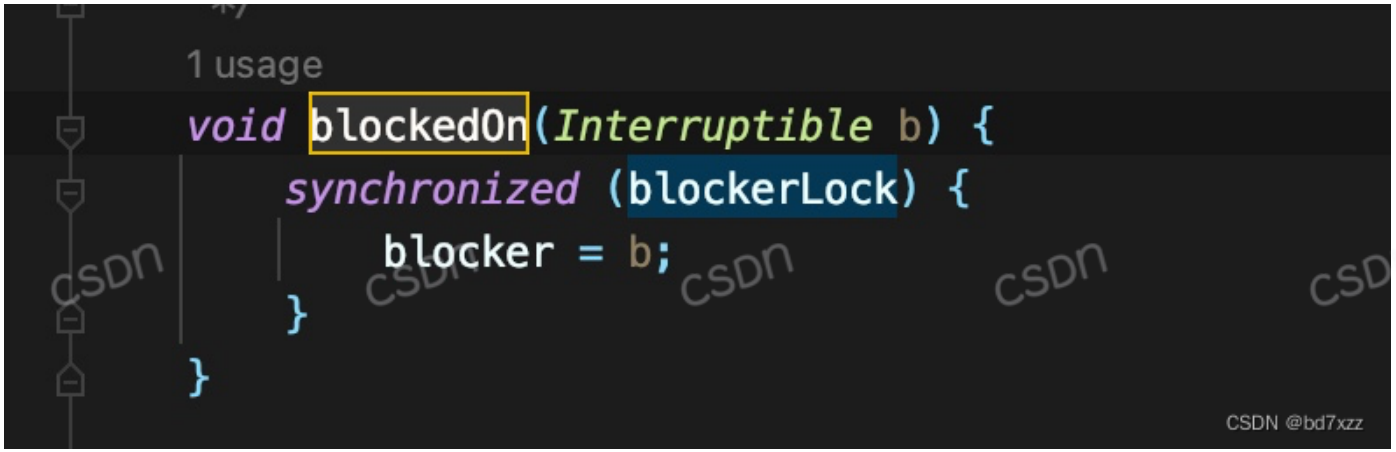
```
                var11.clear();
            } while(this.receive(var11) != null);
        } finally {
            if (var9) {
                this.configureBlocking(true);
```

CSDN @bd7xzz

4. 在 receive 方法并根据线程栈逐层往下，recevie 方法 finally 最后会执行 end 方法，end 方法调用到最后实际上调用了 Thread.blockedOn，bockedOn 需要拿到 blockerLock 锁。此时这个锁 `<0x00000006b77f2098>` 正在被 `histirx-***` 线程持有着。

```
    at java.lang.Thread.blockedOn(Thread.java:239)
    - waiting to lock <0x00000006b77f2098> (a java.lang.Object)
    at java.lang.System$2.blockedOn(System.java:1252)
    at java.nio.channels.spi.AbstractInterruptibleChannel.blockedOn(AbstractInterruptibleChannel.java:211)
    at java.nio.channels.spi.AbstractInterruptibleChannel.end(AbstractInterruptibleChannel.java:198)
    at sun.nio.ch.DatagramChannelImpl.receive(DatagramChannelImpl.java:389)
    - locked <0x00000006b77f20f8> (a java.lang.Object)
    at sun.nio.ch.DatagramChannelImpl.connect(DatagramChannelImpl.java:767)
```

CSDN @bd7xzz


这个链路看起来，似乎是由一个线程建立 UDP 并发包，一个线程对该线程进行 interrupt，就会产生死锁。
这一点，也有人在 openjdk 的 bug 列表中提出， JDK-8228765

Description

A DESCRIPTION OF THE PROBLEM :
If you interrupt a thread that is opening a DatagramChannel, it sometimes deadlocks.

STEPS TO FOLLOW TO REPRODUCE THE PROBLEM :
On one thread, open a DatagramChannel, send a packet, and close the channel.
On a second thread, interrupt the first thread.



那么线上为啥会出现 hystrix 线程对另外一个 http 请求正在 DNS 解析线程进行 interrupt 的动作？这里简单说明下：我们的业务中会通过 BatchExecutor 提交一个异步的业务逻辑，并且超时时间为 300ms。若 300ms 以后未正常返回结果，自然会执行 finally。

```
} finally {
    if (!done)
        for (int i = 0, size = futures.size(); i < size; i++)
            futures.get(i).cancel( mayInterruptIfRunning: true);
}
```



由于在异步业务逻辑比较复杂，请求多个 rpc 或其他资源，若一个请求超过 300ms，其他业务逻辑也会被超时终止，即被 interrupt，当有 DNS 时，便会触发这个 bug。业务流量大的情况下触发概率比较大。

**复现**

经过上面的分析并参考 JDK-8228765，我们可以构造 2 个线程，一个发 UDP 包，一个对其进行 interrupt 操作，参考如下代码：

```
1 import java.io.IOException;
2 import java.net.InetSocketAddress;
3 import java.net.StandardSocketOptions;
4 import java.nio.channels.DatagramChannel;
5 import java.util.concurrent.ThreadLocalRandom;
```

```
6
7  public class UDPReceiver {
8
9      public static void main(String[] args) throws IOException, InterruptedException {
10         DatagramChannel receiver = DatagramChannel.open();
11         receiver.setOption(StandardSocketOptions.SO_REUSEADDR, true);
12         receiver.bind(new InetSocketAddress("localhost",8000));//监听UDP 8000端口
13
14         UDPSender sender = new UDPSender();
15         new Thread(sender).start(); //启动一个线程开始发UDP包
16         while (true) {
17             System.out.println("before interrupt");
18             sender.interruptThis(); //这里会触发死锁
19             System.out.println("after interrupt");
20             Thread.sleep(ThreadLocalRandom.current().nextLong(0,100));
21         }
22     }
23 }
24
```

```
1  import java.net.InetSocketAddress;
2  import java.net.StandardSocketOptions;
3  import java.nio.ByteBuffer;
4  import java.nio.channels.DatagramChannel;
5  import java.util.concurrent.ThreadLocalRandom;
6
7  public class UDPSender implements Runnable {
8      private static final InetSocketAddress target = new InetSocketAddress("localhost", 8000);
9      private Thread thisThread;
10
11
12     public void interruptThis() {
13         if (null != thisThread) {
14             thisThread.interrupt(); //引发死锁的源头2
15
```

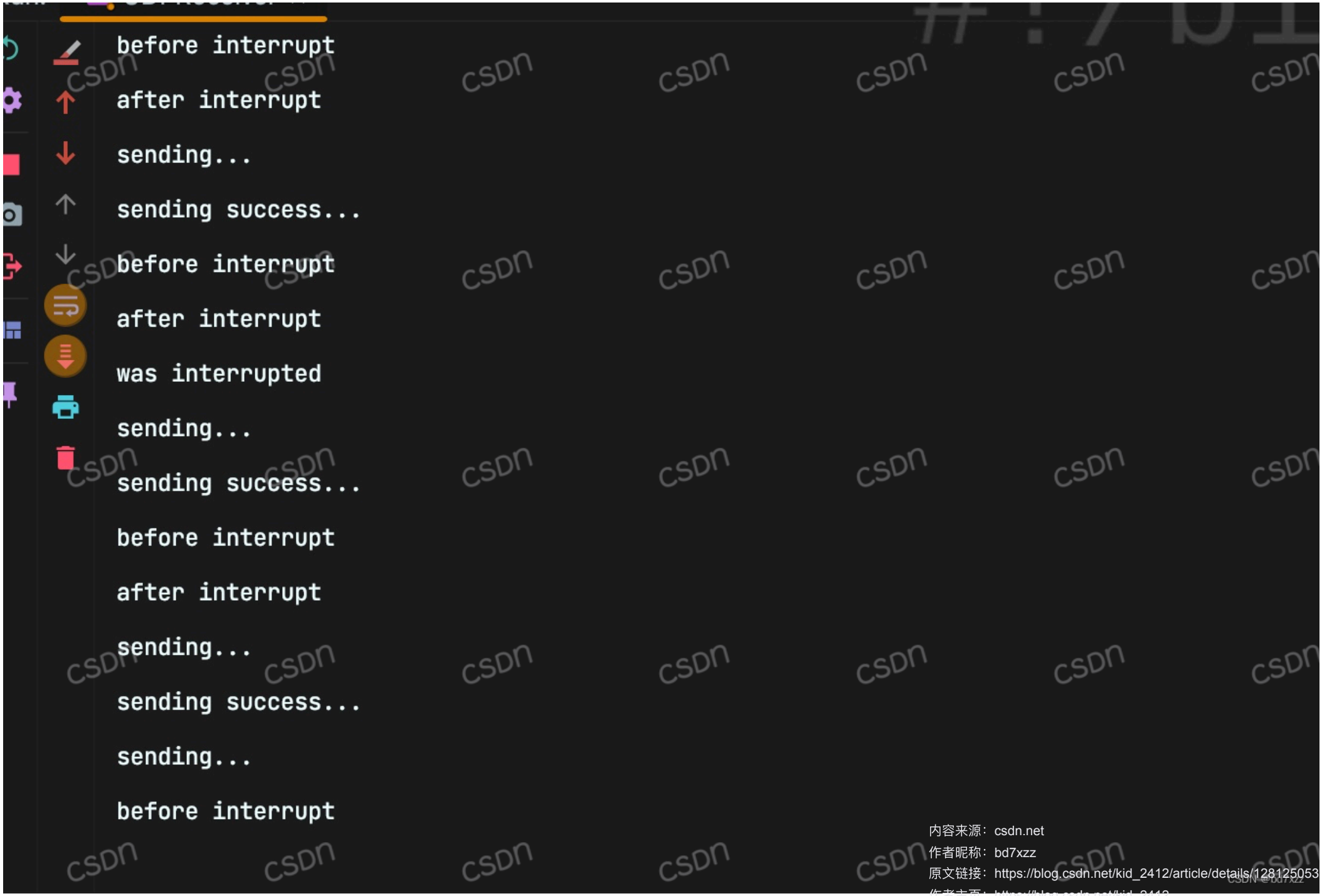
```
        }
    }

    @Override
    public void run() {
        thisThread = Thread.currentThread(); //保存一下sender的当前线程
        while (true) {
            try {
                DatagramChannel sender = DatagramChannel.open();
                sender.setOption(StandardSocketOptions.SO_REUSEADDR, true);
                sender.connect(target); //连接本地8000端口发包
                System.out.println("sending...");
                sender.send(ByteBuffer.allocate(100).putInt(1), target);//引发死锁的源头1
                System.out.println("sending success...");
                Thread.sleep(ThreadLocalRandom.current().nextLong(0, 100));
            } catch (Exception e) {

            }

        }
    }
}
```

如下图所示，在跑几轮后 before interrupt 的时候出现死锁。



```
jstack -l <pid>
```

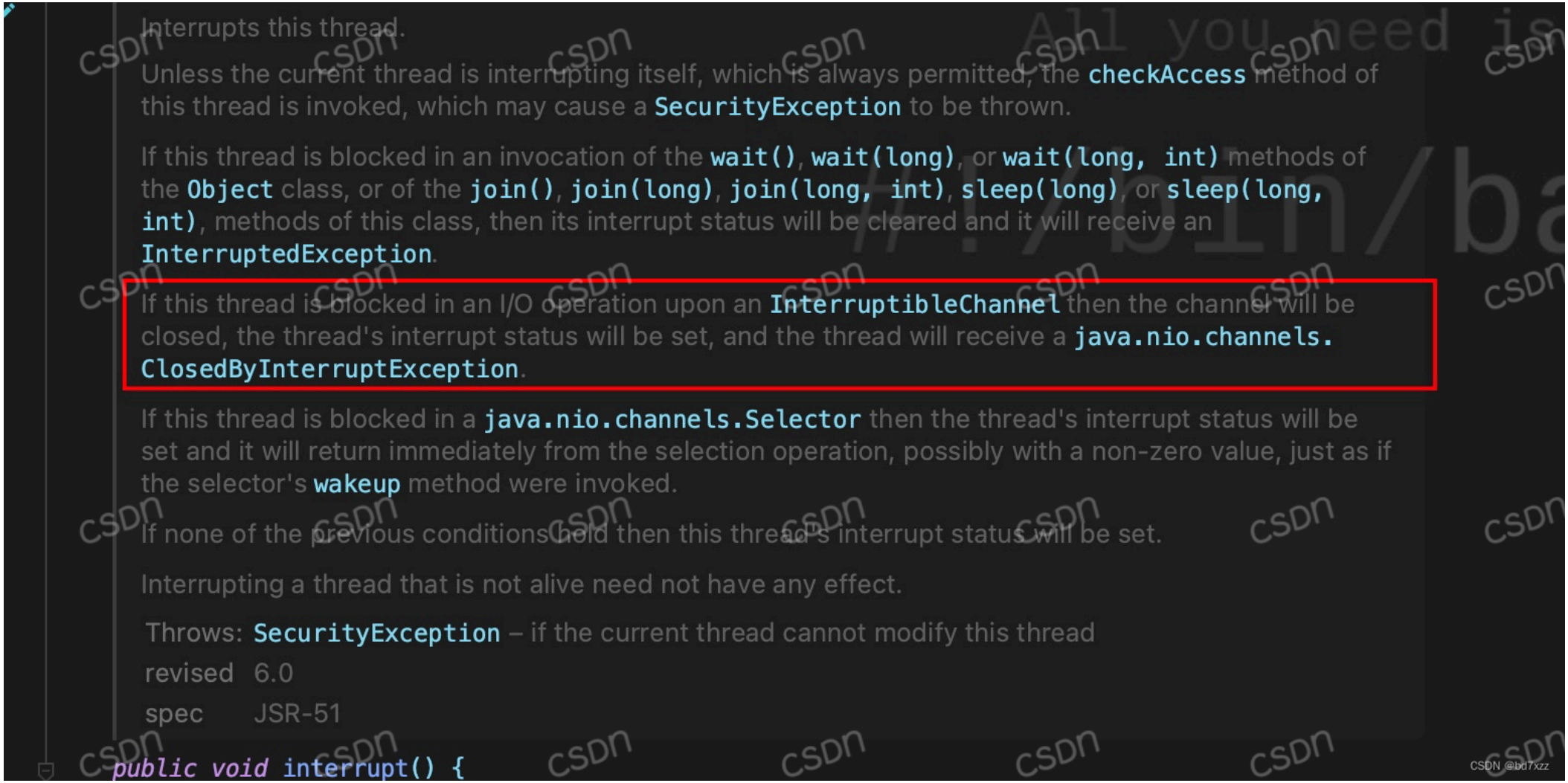
打印线程栈，最后可看到死锁信息。

```
Found one Java-level deadlock:
=============================
"Thread-0":
  waiting to lock monitor 0x00007f8a31826148 (object 0x000000076b0f9668, a java.lang.Object),
  which is held by "main"
"main":
  waiting to lock monitor 0x00007f8a31824d58 (object 0x000000076b1e2be8, a java.lang.Object),
  which is held by "Thread-0"

Java stack information for the threads listed above:
===================================================
"Thread-0":
        at java.lang.Thread.blockedOn(Thread.java:239)
        - waiting to lock <0x000000076b0f9668> (a java.lang.Object)
        at java.lang.System$2.blockedOn(System.java:1252)
        at java.nio.channels.spi.AbstractInterruptibleChannel.blockedOn(AbstractInterruptibleChannel.java:211)
        at java.nio.channels.spi.AbstractInterruptibleChannel.end(AbstractInterruptibleChannel.java:198)
        at sun.nio.ch.DatagramChannelImpl.write(DatagramChannelImpl.java:610)
        - locked <0x000000076b1e2bd8> (a java.lang.Object)
        at sun.nio.ch.DatagramChannelImpl.send(DatagramChannelImpl.java:451)
        - locked <0x000000076b1e2be8> (a java.lang.Object)
        - locked <0x000000076b1e2bd8> (a java.lang.Object)
        at UDPSender.run(UDPSender.java:27)
        at java.lang.Thread.run(Thread.java:748)
"main":
        at sun.nio.ch.DatagramChannelImpl.implCloseSelectableChannel(DatagramChannelImpl.java:1007)
        - waiting to lock <0x000000076b1e2be8> (a java.lang.Object)
        at java.nio.channels.spi.AbstractSelectableChannel.implCloseChannel(AbstractSelectableChannel.java:234)
        at java.nio.channels.spi.AbstractInterruptibleChannel$1.interrupt(AbstractInterruptibleChannel.java:165)
        - locked <0x000000076b1e2b98> (a java.lang.Object)
        at java.lang.Thread.interrupt(Thread.java:922)
        - locked <0x000000076b0f9668> (a java.lang.Object)
        at UDPSender.interruptThis(UDPSender.java:14)
        at UDPReceiver.main(UDPReceiver.java:17)

Found 1 deadlock.
```

线程 Thread-0 等待的锁对象 `0x000000076b0f9668` 由主线程在执行 UDPSender.interruptThis 时锁住，即对 Thread-0 线程进行 interrupt。根据 Thread.interrupt 方法的描述，若当前线程处于 IO 方法阻塞时，通道会被关闭，会抛出 ClosedByInterruptException 异常并设置中断状态为 true。




死锁的整个过程如下：

先看 new Thread (sender).start () 即 Thread-0 的执行。

可以看死锁的线程栈调用链路执行情况，参考代码推导：

```
==========================================================
"Thread-0":
        at java.lang.Thread.blockedOn(Thread.java:239)
        - waiting to lock <0x000000076b0f9668> (a java.lang.Object)
        at java.lang.System$2.blockedOn(System.java:1252)
        at java.nio.channels.spi.AbstractInterruptibleChannel.blockedOn(AbstractInterruptibleChannel.java:211)
        at java.nio.channels.spi.AbstractInterruptibleChannel.end(AbstractInterruptibleChannel.java:198)
        at sun.nio.ch.DatagramChannelImpl.write(DatagramChannelImpl.java:610)
        - locked <0x000000076b1e2bd8> (a java.lang.Object)
        at sun.nio.ch.DatagramChannelImpl.send(DatagramChannelImpl.java:451)
        - locked <0x000000076b1e2be8> (a java.lang.Object)
        - locked <0x000000076b1e2bd8> (a java.lang.Object)
        at UDPSender.run(UDPSender.java:27)
        at java.lang.Thread.run(Thread.java:748)
```



1. DatagramChannelImpl 实现了 AbstractInterruptibleChannel。在 DatagramChannelImpl.send 方法中会拿到两把锁，注意其中一把为 stateLock，即线程中的 `- locked <0x000000076b1e2be8> (a java.lang.Object)`

```java
public int send(ByteBuffer var1, SocketAddress var2) throws IOException {
    if (var1 == null) {
        throw new NullPointerException();
    } else {
        synchronized(this.writeLock) {
            this.ensureOpen();
            InetSocketAddress var4 = Net.checkAddress(var2);
            InetAddress var5 = var4.getAddress();
            if (var5 == null) {
                throw new IOException("Target address not resolved");
            } else {
                synchronized(this.stateLock) {
                    if (this.isConnected()) {
                        if (!var2.equals(this.remoteAddress)) {
                            throw new IllegalArgumentException("Connected address not equal to target address");
                        }

                        int var10000 = this.write(var1);
                        return var10000;
                    }

                    if (var2 == null) {
                        throw new NullPointerException();
                    }

                    SecurityManager var7 = System.getSecurityManager();
                    if (var7 != null) {
                        if (var5.isMulticastAddress()) {
                            var7.checkMulticast(var5);
                        } else {
                            var7.checkConnect(var5.getHostAddress(), var4.getPort());
                        }
```

2. 在 write 方法中，可以看到调用 this.begin 方法，即 AbstracInterruptibleChannel.begin。在 IOUtil.write 真正发包完成后，会执行 finally 中的 this.end 方法。

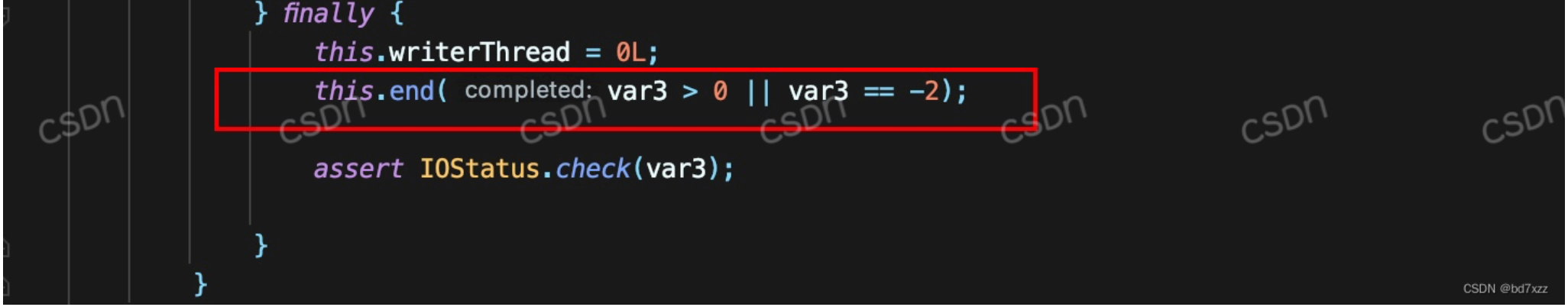
```java
public int write(ByteBuffer var1) throws IOException {
    if (var1 == null) {
        throw new NullPointerException();
    } else {
        synchronized(this.writeLock) {
            synchronized(this.stateLock) {
                this.ensureOpen();
                if (!this.isConnected()) {
                    throw new NotYetConnectedException();
                }
            }

            int var3 = 0;

            try {
                this.begin();
                if (!this.isOpen()) {
                    byte var13 = 0;
                    return var13;
                } else {
                    this.writerThread = NativeThread.current();

                    do {
                        var3 = IOUtil.write(this.fd, var1,  l: -1L, nd);
                    } while(var3 == -3 && this.isOpen());

                    int var4 = IOStatus.normalize(var3);
                    return var4;
                }
```



AbstracInterruptibleChannel.begin 注册了收到 interrupt 后要执行的逻辑，注意这里，下面会用。

```
protected final void begin() {
    if (interruptor == null) {
        interruptor = new Interruptible() {
                public void interrupt(Thread target) {
                    synchronized (closeLock) {
                        if (!open)
                            return;
                        open = false;
                        interrupted = target;
                        try {
                            AbstractInterruptibleChannel.this.implCloseChannel();
                        } catch (IOException x) { }
                    }
                }};
    }
    blockedOn(interruptor);
    Thread me = Thread.currentThread();
    if (me.isInterrupted())
        interruptor.interrupt(me);
}
```

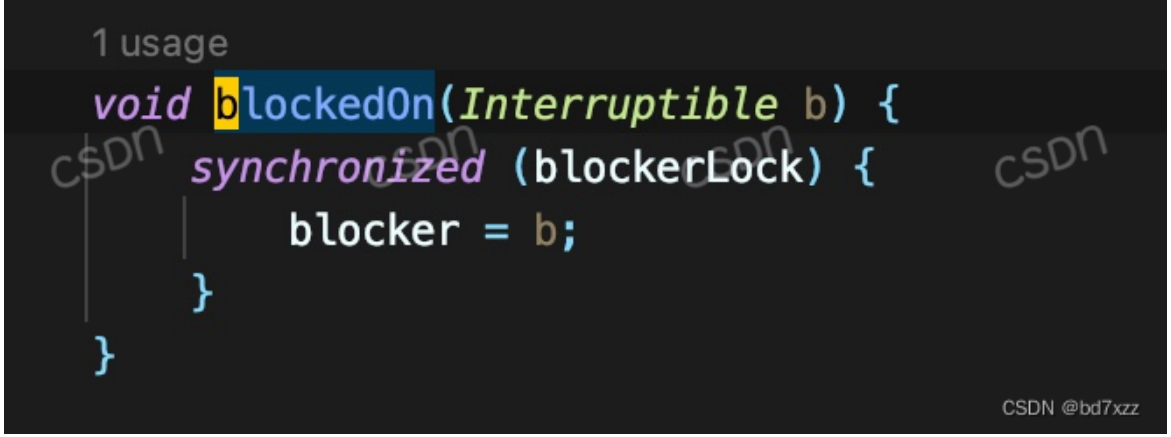
CSDN @bd7xzz

3. 在 AbstractInterruptibleChannel.end 方法中通过 blockedOn 方法标记 block 状态

```
protected final void end(boolean completed)
    throws AsynchronousCloseException
{
    blockedOn( intr: null);
    Thread interrupted = this.interrupted;
    if (interrupted != null && interrupted == Thread.currentThread()) {
        interrupted = null;
        throw new ClosedByInterruptException();
    }
    if (!completed && !open)
        throw new AsynchronousCloseException();
}
```

CSDN @bd7xzz

4. blockedOn 会最终调到 Thread.blockedOn 方法，且实参为当前 Thread-0 线程。注意这里，要拿到 blockerlock，即线程栈中的 `- waiting to lock <0x000000076b0f9668> (a java.lang.Object)` 。




再看 sender.interruptThis () 的执行：

可以看死锁的线程栈调用链路执行情况，参考代码推导：

```
"main":
        at sun.nio.ch.DatagramChannelImpl.implCloseSelectableChannel(DatagramChannelImpl.java:1007)
        - waiting to lock <0x000000076b1e2be8> (a java.lang.Object)
        at java.nio.channels.spi.AbstractSelectableChannel.implCloseChannel(AbstractSelectableChannel.java:234)
        at java.nio.channels.spi.AbstractInterruptibleChannel$1.interrupt(AbstractInterruptibleChannel.java:165)
        - locked <0x000000076b1e2b98> (a java.lang.Object)
        at java.lang.Thread.interrupt(Thread.java:922)
        - locked <0x000000076b0f9668> (a java.lang.Object)
        at UDPSender.interruptThis(UDPSender.java:14)
        at UDPReceiver.main(UDPReceiver.java:17)

Found 1 deadlock.
```

CSDN @bd7xzz

```
public static void main(String[] args) throws IOException, InterruptedException {
    DatagramChannel receiver = DatagramChannel.open();
    receiver.setOption(StandardSocketOptions.SO_REUSEADDR, value: true);
    receiver.bind(new InetSocketAddress( hostname: "localhost", port: 8000));
    UDPSender sender = new UDPSender();
    new Thread(sender).start();
    while (true) {
        System.out.println("before interrupt");
        sender.interruptThis();
        System.out.println("after interrupt");
        Thread.sleep(ThreadLocalRandom.current().nextLong( origin: 0, bound: 100));
    }
}
```

CSDN @bd7xzz

1. 在 Thread.interrupt 方法中会调用 b.interrupt，注意这里会拿到 blokcerLock，即线程栈中的 `- locked <0x000000076b0f9668> (a java.lang.Object)`

2. b.interrupt 会触发 AbstracInterruptibleChannel 在 begin 方法中实现的 interrupt 方法

```
protected final void begin() {
    if (this.interruptor == null) {
        this.interruptor = new Interruptible() {
            public void interrupt(Thread var1) {
                synchronized(AbstractInterruptibleChannel.this.closeLock) {
                    if (AbstractInterruptibleChannel.this.open) {
                        AbstractInterruptibleChannel.this.open = false;
                        AbstractInterruptibleChannel.this.interrupted = var1;

                        try {
                            AbstractInterruptibleChannel.this.implCloseChannel();
                        } catch (IOException var5) {
                        }

                    }
                }
            }
        };
    }

    blockedOn(this.interruptor);
    Thread var1 = Thread.currentThread();
    if (var1.isInterrupted()) {
        this.interruptor.interrupt(var1);
    }
}
```

3. 通过 AbstracInterruptibleChannel.implCloseChannel 关闭通道。

```
5 usages
protected final void implCloseChannel() throws IOException {
    implCloseSelectableChannel();
    synchronized (keyLock) {
        int count = (keys == null) ? 0 : keys.length;
        for (int i = 0; i < count; i++) {
            SelectionKey k = keys[i];
            if (k != null)
                k.cancel();
        }
    }
}
```

CSDN @bd7xzz

4. 在 DatagramChannelImpl.implCloseChannel 的实现中，要拿到 stateLock 这把锁。注意这里就是线程栈中的 `- waiting to lock <0x000000076b1e2be8> (a java.lang.Object)`

```
2 usages   1 override
protected void implCloseSelectableChannel() throws IOException {
    synchronized(this.stateLock) {
        if (this.state != 2) {
            nd.preClose(this.fd);
        }

        ResourceManager.afterUdpClose();
        if (this.registry != null) {
            this.registry.invalidateAll();
        }

        long var2;
        if ((var2 = this.readerThread) != 0L) {
            NativeThread.signal(var2);
        }

        if ((var2 = this.writerThread) != 0L) {
            NativeThread.signal(var2);
        }

        if (!this.isRegistered()) {
            this.kill();
        }
```

从上面的分析可以很明显的看出，主线程触发 Thread-0 的 interrupt 动作时，在执行 interupt 方法持有了 blockerLock 锁，关闭通道要拿 stateLock 锁，而 Thread-0 在发包时先拿到了 stateLock 锁，并且设置 block 状态时要拿 blockerLock 锁，此时出现死锁！

### 解决方法

1. 根据 OpenJDK 的 bug 列表中的说明 JDK-8228765 ，这个 bug 从 JDK-8039509 就开始有了，一直到 jdk13 才修复。所以，要么升级 jdk 到 13，要么死！我选择了死。
2. 我们项目中使用了 dnsjava，在 dnsjava 的 github 上 issues#69 里也是有报这个 bug 的。另外在 logstash 中也有类似的问题 issues#117。
3. 当然如果选择了死，那就是遇到了重启服务器了。

### 总结

1. JVM 的 bug 也很多，同时也很诡异，需要仔细观察运行时状态 + 阅读源码才能分析出原因，如果你不想分析原因，可以用 google 搜搜看，很可能已经有了答案。
2. 排查问题思路很重要，由于业务环境通常比较复杂，在大流量、大数据量的情况下会有各种千奇百怪的问题，需要有足够的耐心和较广的知识面。