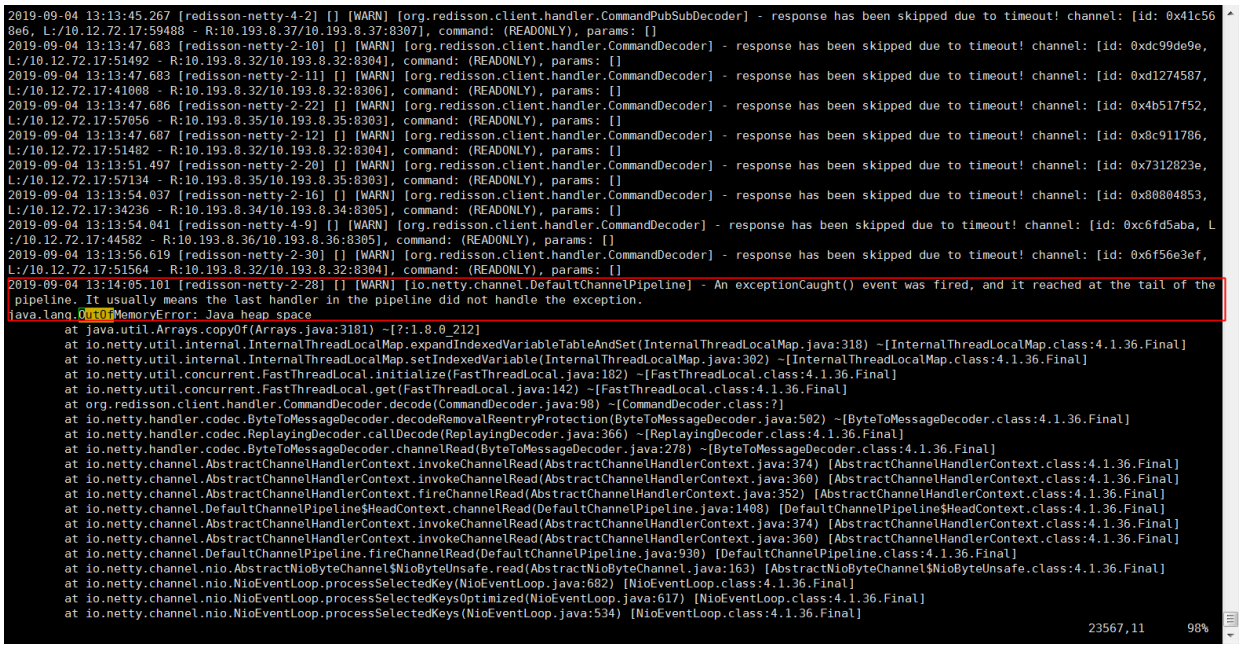
redisson客户端oom bug

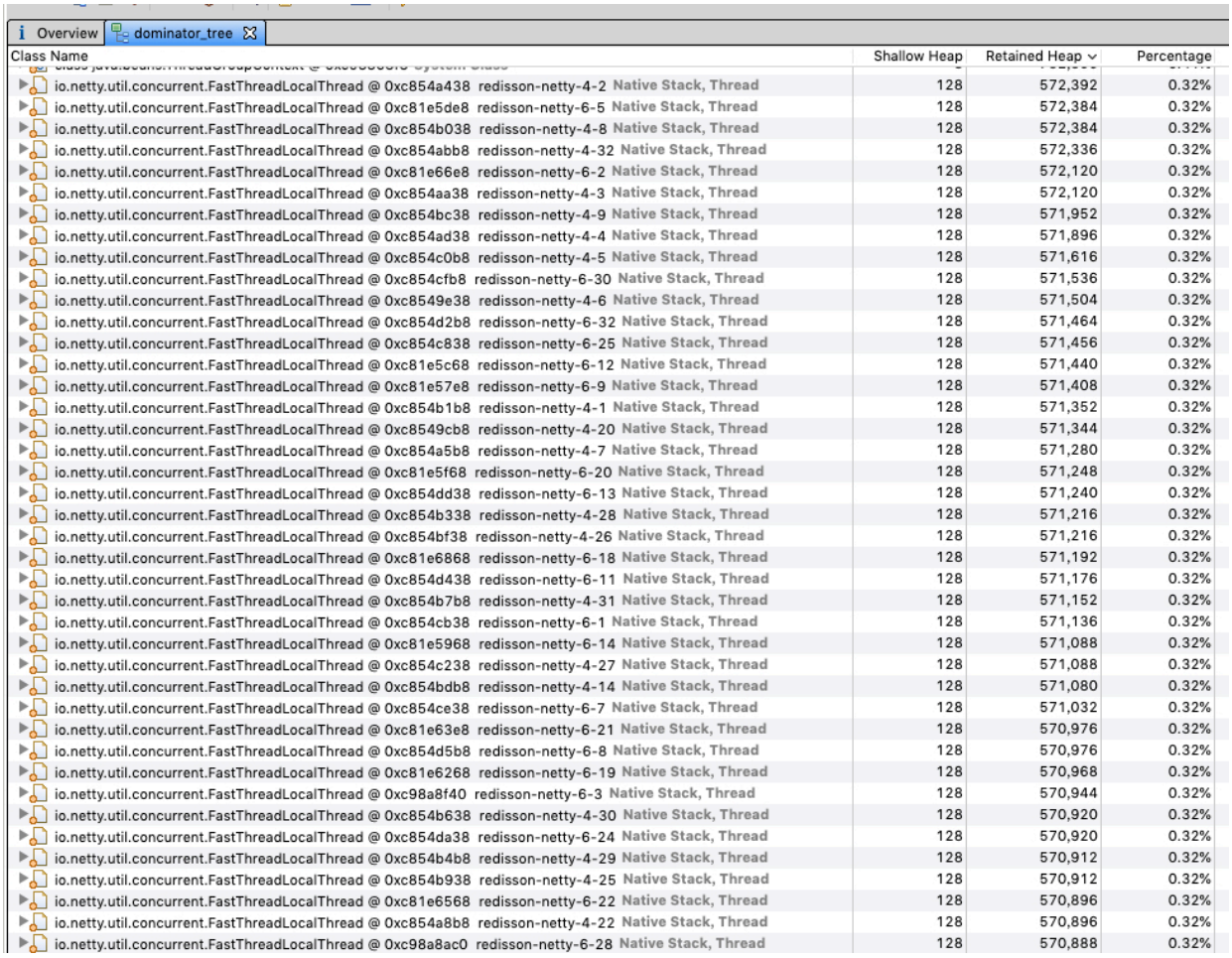
1、发现问题

在使用redisson作为redis的客户端，上线一段时间之后，发现时不时有app因为OutOfMemory（java.lang.OutOfMemoryError: Java heap space）退出进程。

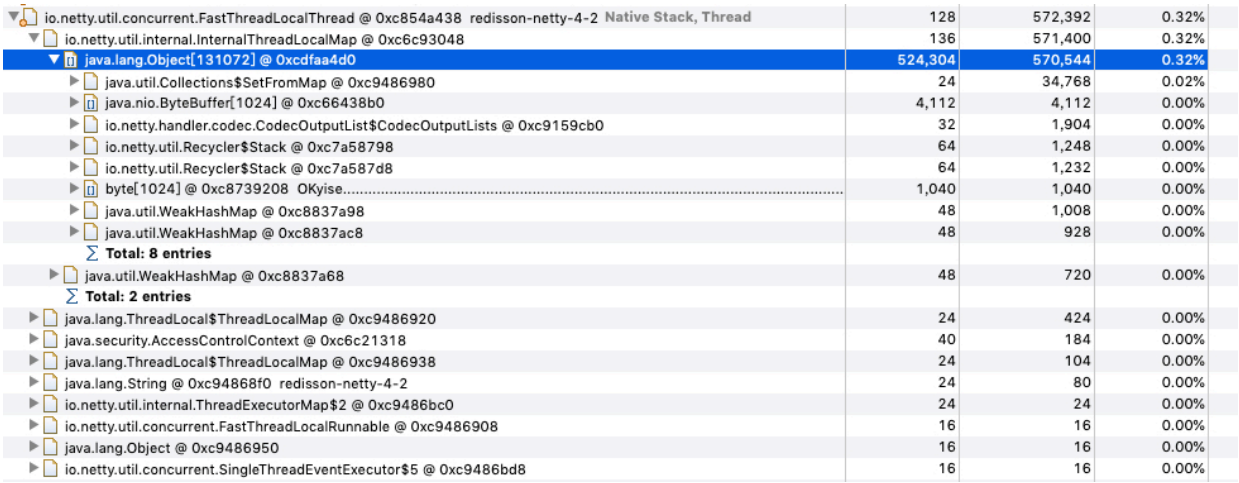


2、heap dump

对运行一段时间的应用进行heap dump，查看其内存占用情况。



Class Name	Shallow Heap	Retained Heap	Percentage
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854a438 redisson-netty-4-2 Native Stack, Thread	128	572,392	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e5de8 redisson-netty-6-5 Native Stack, Thread	128	572,384	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854b038 redisson-netty-4-8 Native Stack, Thread	128	572,384	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854abb8 redisson-netty-4-32 Native Stack, Thread	128	572,336	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e66e8 redisson-netty-6-2 Native Stack, Thread	128	572,120	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854aa38 redisson-netty-4-3 Native Stack, Thread	128	572,120	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854bc38 redisson-netty-4-9 Native Stack, Thread	128	571,952	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854ad38 redisson-netty-4-4 Native Stack, Thread	128	571,896	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854c0b8 redisson-netty-4-5 Native Stack, Thread	128	571,616	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854cfb8 redisson-netty-6-30 Native Stack, Thread	128	571,536	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc8549e38 redisson-netty-4-6 Native Stack, Thread	128	571,504	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854d2b8 redisson-netty-6-32 Native Stack, Thread	128	571,464	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854c838 redisson-netty-6-25 Native Stack, Thread	128	571,456	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e5c68 redisson-netty-6-12 Native Stack, Thread	128	571,440	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e57e8 redisson-netty-6-9 Native Stack, Thread	128	571,408	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854b1b8 redisson-netty-4-1 Native Stack, Thread	128	571,352	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc8549cb8 redisson-netty-4-20 Native Stack, Thread	128	571,344	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854a5b8 redisson-netty-4-7 Native Stack, Thread	128	571,280	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e5f68 redisson-netty-6-20 Native Stack, Thread	128	571,248	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854dd38 redisson-netty-6-13 Native Stack, Thread	128	571,240	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854b338 redisson-netty-4-28 Native Stack, Thread	128	571,216	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854bf38 redisson-netty-4-26 Native Stack, Thread	128	571,216	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e6868 redisson-netty-6-18 Native Stack, Thread	128	571,192	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854d438 redisson-netty-6-11 Native Stack, Thread	128	571,176	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854b7b8 redisson-netty-4-31 Native Stack, Thread	128	571,152	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854cb38 redisson-netty-6-1 Native Stack, Thread	128	571,136	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e5968 redisson-netty-6-14 Native Stack, Thread	128	571,088	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854c238 redisson-netty-4-27 Native Stack, Thread	128	571,088	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854bdb8 redisson-netty-4-14 Native Stack, Thread	128	571,080	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854ce38 redisson-netty-6-7 Native Stack, Thread	128	571,032	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e63e8 redisson-netty-6-21 Native Stack, Thread	128	570,976	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854d5b8 redisson-netty-6-8 Native Stack, Thread	128	570,976	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e6268 redisson-netty-6-19 Native Stack, Thread	128	570,968	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc98a8f40 redisson-netty-6-3 Native Stack, Thread	128	570,944	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854b638 redisson-netty-4-30 Native Stack, Thread	128	570,920	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854da38 redisson-netty-6-24 Native Stack, Thread	128	570,920	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854b4b8 redisson-netty-4-29 Native Stack, Thread	128	570,912	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854b938 redisson-netty-4-25 Native Stack, Thread	128	570,912	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc81e6568 redisson-netty-6-22 Native Stack, Thread	128	570,896	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc854a8b8 redisson-netty-4-22 Native Stack, Thread	128	570,896	0.32%
io.netty.util.concurrent.FastThreadLocalThread @ 0xc98a8ac0 redisson-netty-6-28 Native Stack, Thread	128	570,888	0.32%



io.netty.util.concurrent.FastThreadLocalThread @ 0xc854a438 redisson-netty-4-2 Native Stack, Thread	128	572,392	0.32%
io.netty.util.internal.InternalThreadLocalMap @ 0xc6c93048	136	571,400	0.32%
java.lang.Object[131072] @ 0xcdfaa4d0	524,304	570,544	0.32%
java.util.Collections$SetFromMap @ 0xc9486980	24	34,768	0.02%
java.nio.ByteBuffer[1024] @ 0xc66438b0	4,112	4,112	0.00%
io.netty.handler.codec.CodecOutputList$CodecOutputLists @ 0xc9159cb0	32	1,904	0.00%
io.netty.util.Recycler$Stack @ 0xc7a58798	64	1,248	0.00%
io.netty.util.Recycler$Stack @ 0xc7a587d8	64	1,232	0.00%
byte[1024] @ 0xc8739208 OKyise.....	1,040	1,040	0.00%
java.util.WeakHashMap @ 0xc8837a98	48	1,008	0.00%
java.util.WeakHashMap @ 0xc8837ac8	48	928	0.00%
Total: 8 entries			
java.util.WeakHashMap @ 0xc8837a68	48	720	0.00%
Total: 2 entries			
java.lang.ThreadLocal$ThreadLocalMap @ 0xc9486920	24	424	0.00%
java.security.AccessControlContext @ 0xc6c21318	40	184	0.00%
java.lang.ThreadLocal$ThreadLocalMap @ 0xc9486938	24	104	0.00%
java.lang.String @ 0xc94868f0 redisson-netty-4-2	24	80	0.00%
io.netty.util.internal.ThreadExecutorMap$2 @ 0xc9486bc0	24	24	0.00%
io.netty.util.concurrent.FastThreadLocalRunnable @ 0xc9486908	16	16	0.00%
java.lang.Object @ 0xc9486950	16	16	0.00%
io.netty.util.concurrent.SingleThreadEventExecutor$5 @ 0xc9486bd8	16	16	0.00%

可以看到FastThreadLocalThread对象持有一个大的Object数组。

3、分析代码，定位问题

为了定位问题，我们使用了最简单的spring boot+redisson官方提供的示例代码，发现能够完全复现线上问题。

1）结合第一步发生oom时候的业务日志，能够定位到是`CommandDecoder`的`decode`方法被调用的时候导致oom的发生。

2）`CommandDecoder`实例化的时候，会实例化一个成员变量`FastThreadLocal`。

3）`FastThreadLocal`实例化的时候，会初始化其成员变量`index`。源代码如下：

```
public FastThreadLocal() {
    index = InternalThreadLocalMap.nextVariableIndex();
}
```

也就是说，是因为`CommandDecoder`的频繁初始化导致的oom。那现在问题就集中在`CommandDecoder`为什么会频繁初始化。

4）从源代码搜索`CommandDecoder`初始化的代码，发现是`RedisChannelInitializer`的`initChannel`方法被调用的时候会`new CommandDecoder`。`initChannel`被调用，说明TCP链路注册成功。那现在问题集中在TCP链路为何会不断注册?

5）对netty有了基本的了解之后，应该能够想到redisson作为netty客户端，应该会有连接监控等日志。而且netty中实现这些功能一般也是通过在pipeline里增加`ChannelHandler`来实现的。继续观察`initChannel`的实现逻辑，发现其增加了一个名为`ConnectionWatchdog`的`ChannelHandler`。源代码如下：

```
    connectionWatchdog = new ConnectionWatchdog(bootstrap, channels, config.getTimer());
}

@Override
protected void initChannel(Channel ch) throws Exception {
    initSsl(config, ch);

    if (type == Type.PLAIN) {
        ch.pipeline().addLast(new RedisConnectionHandler(redisClient));
    } else {
        ch.pipeline().addLast(new RedisPubSubConnectionHandler(redisClient));
    }

    ch.pipeline().addLast(
        connectionWatchdog,
        CommandEncoder.INSTANCE,
        CommandBatchEncoder.INSTANCE,
        new CommandsQueue());

    if (pingConnectionHandler != null) {
        ch.pipeline().addLast(pingConnectionHandler);
    }

    if (type == Type.PLAIN) {
        ch.pipeline().addLast(new CommandDecoder(config.getExecutor(), config.isDecodeInExecutor()));
    } else {
        ch.pipeline().addLast(new CommandPubSubDecoder(config.getExecutor(), config.isKeepPubSubOrder(), config.isDecodeInExecutor()));
    }
}
```

6）`ConnectionWatchdog`实现了`channelInactive`方法，其实现了重连的逻辑。

```java
@Override
public void channelInactive(ChannelHandlerContext ctx) throws Exception {
    RedisConnection connection = RedisConnection.getFrom(ctx.channel());
    if (connection != null) {
        connection.fireDisconnected();
        if (!connection.isClosed()) {
            if (connection.isFastReconnect()) {
                tryReconnect(connection, nextAttempt: 1);
            } else {
                reconnect(connection, attempts: 1);
            }
        }
    }
    ctx.fireChannelInactive();
}
```

7）将app的日志级别改为debug级别。发现每隔5分钟就会打印重连日志：

```
2019-09-29 18:09:24.285 DEBUG 65271 --- [pool-1-thread-1]
o.r.client.handler.ConnectionWatchdog    : reconnecting
RedisConnection@1984655680 [redisClient=[addr=redis://10.193.8.37:8304],
channel=[id: 0x35f3baff, L:/10.2.121.35:53398 !
R:10.193.8.37/10.193.8.37:8304], currentCommand=null] to
10.193.8.37/10.193.8.37:8304
2019-09-29 18:09:24.286 DEBUG 65271 --- [pool-1-thread-1]
o.r.client.handler.ConnectionWatchdog    : reconnecting
RedisPubSubConnection@1938066514 [redisClient=
[addr=redis://10.193.8.37:8304], channel=[id: 0x84548cac,
L:/10.2.121.35:53411 ! R:10.193.8.37/10.193.8.37:8304],
currentCommand=null] to 10.193.8.37/10.193.8.37:8304
2019-09-29 18:09:24.291 DEBUG 65271 --- [pool-1-thread-1]
o.r.client.handler.ConnectionWatchdog    : reconnecting
RedisConnection@166080294 [redisClient=[addr=redis://10.193.8.37:8304],
channel=[id: 0x092bbd1e, L:/10.2.121.35:53405 !
R:10.193.8.37/10.193.8.37:8304], currentCommand=null] to
10.193.8.37/10.193.8.37:8304
```

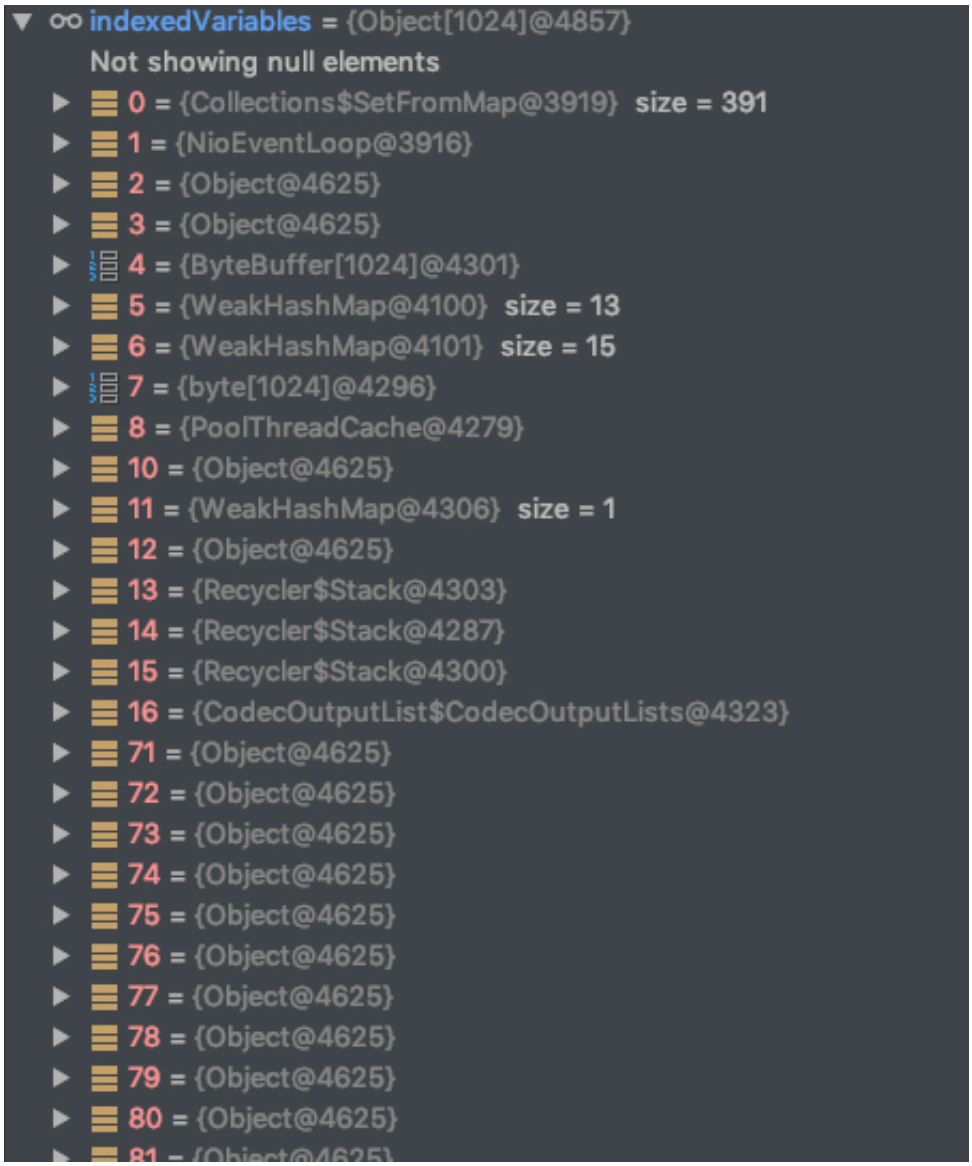
根据日志，定位到代码，发现`tryReconnect`方法会调用bootstrap的connect方法，当TCP链路注册成功之后，就会调用`RedisChannelInitializer`的`initChannel`方法，这就回到了第4）步的分析。

```java
private void tryReconnect(final RedisConnection connection, final int nextAttempt) {
    if (connection.isClosed() || bootstrap.config().group().isShuttingDown()) {
        return;
    }

    log.debug("reconnecting {} to {} ", connection, connection.getRedisClient().getAddr(), connection);

    try {
        bootstrap.connect(connection.getRedisClient().getAddr()).addListener(new ChannelFutureListener() {
```

8）另外，对app进行调试，观察`FastThreadLocalThread`所持有的Object数组的值，除了前几位有意义之外，其他均是无意义的空对象。



至此，我们基本可以确定redisson客户端存在bug，而且该bug就是对netty所提供的`FastThreadLocal`的不正确使用所导致。

9）在github上提issue向作者反应该问题，但是作者最开始并没有意识到redisson客户端存在的bug。

https://github.com/redisson/redisson/issues/2309

10）在同步和redisson沟通该问题的过程中，我们自己修改了redisson的源代码。其实代码改动很少，就是将其对`FastTheadLocal`的使用改为JDK自带的`ThreadLocal`，并且在decode完毕之后，调用`TheadLocal`的`remove`方法。通过在sylvanas预发环境和holmes的两台生产环境观察（3-4天的时间），并对这些app进行heap dump分析，发现oom的问题已经得到解决。

11）在我们准备大规模对生产环境的redisson版本进行升级的时候，发现redisson的作者意识到这是一个bug，并进行了修复。其修复的方式是：不用`FastThreadLocal`去存储变量，而是改为成员变量的方式。

> 其实这个方法在我们自己解决redisson的这个bug的时候也考虑过，但是没有十分的把握在这里直接将线程私有的变量修改为成员变量是否会有问题，所以最终我们选择了比较稳妥的用JDK自带的`ThreadLocal`替换`FastThreadLocal`。

4、总结

1）尽量隔离出一个简单独立的环境来复现问题。在复杂的业务系统中定位问题会受到业务系统中其他框架等的干扰，不利于定位问题。

2）要敢于阅读、能够阅读项目中所用开源框架的源代码。

3）要掌握JVM问题排查的基本方法和工具。

4）要保持良好的心态。不要慌，不要急。坚定问题一定能够得到解决，特别是能够稳定复现的问题一定能够解决。

延伸1：netty中FastThreadLocal的实现

https://www.jianshu.com/p/92c3832c0a8b