

常见的JVM内存溢出异常

1.大对象内存溢出

Java堆用于存储对象实例，只要不断地创建对象，当对象数量到达最大堆的容量限制后就会产生内存溢出异常。最常见的内存溢出就是存在大的容器，而没法回收，比如：Map，List等。

- 内存溢出：内存空间不足导致，新对象无法分配到足够的内存；
- 内存泄漏：应该释放的对象没有被释放，多见于自己使用容器保存元素的情况下。

出现下面信息就可以断定出现了堆内存溢出。

```
java.lang.OutOfMemoryError: Java heap space
```

保证GC Roots到对象之间有可达路径来避免垃圾回收机制清除这些对象。

实例:

```
-verbose:gc -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError -  
XX:HeapDumpPath=D:\dump
```

```
/**  
 * java 堆内存溢出  
 * <p>  
 * VM Args: -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError -  
XX:HeapDumpPath=D:\dump  
 *  
 * @author yuhao.wang3  
 */  
public class HeapOutOfMemoryErrorTest {  
    public static void main(String[] args) throws InterruptedException {  
        // 模拟大容器  
        List<Object> list = Lists.newArrayList();  
        for (long i = 1; i > 0; i++) {  
            list.add(new Object());  
            if (i % 100_000 == 0) {  
                System.out.println(Thread.currentThread().getName() + "::" + i);  
            }  
        }  
    }  
}
```

运行结果:

```
[GC (Allocation Failure) 5596K->1589K(19968K), 0.0422027 secs]  
main::100000  
main::200000  
[GC (Allocation Failure) 7221K->5476K(19968K), 0.0144103 secs]  
main::300000  
[GC (Allocation Failure) 9190K->9195K(19968K), 0.0098252 secs]
```

```

main::400000
main::500000
[Full GC (Ergonomics) 17992K->13471K(19968K), 0.3431052 secs]
main::600000
main::700000
main::800000
[Full GC (Ergonomics) 17127K->16788K(19968K), 0.1581969 secs]
[Full GC (Allocation Failure) 16788K->16758K(19968K), 0.1994445 secs]
java.lang.OutOfMemoryError: Java heap space
Dumping heap to D:\dump\java_pid7432.hprof ...
Heap dump file created [28774262 bytes in 0.221 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3210)
    at java.util.Arrays.copyOf(Arrays.java:3181)
    at java.util.ArrayList.grow(ArrayList.java:261)
    at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:235)
    at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:227)
    at java.util.ArrayList.add(ArrayList.java:458)
    at com.xiaolyuh.HeapOutOfMemoryErrorTest.main(HeapOutOfMemoryErrorTest.java:23)
Disconnected from the target VM, address: '127.0.0.1:61622', transport: 'socket'

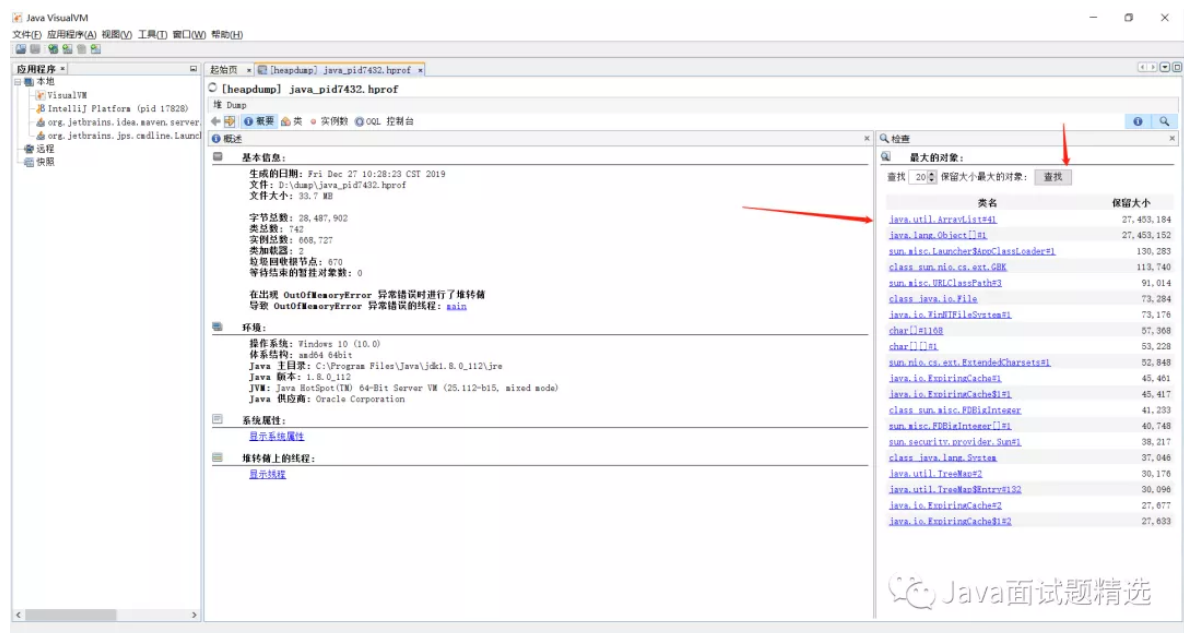
```

分析工具:

JDK自带的jvisualvm.exe工具可以分析.hprof和.dump文件。

首先需要找出最大的对象，判断最大对象的存在是否合理，如何合理就需要调整VM内存大小。如果不合理，那么这个对象的存在，就是最有可能是引起内存溢出的根源。通过GC Roots的引用链信息，就可以比较准确地定位出泄露代码的位置。

1.查询最大对象



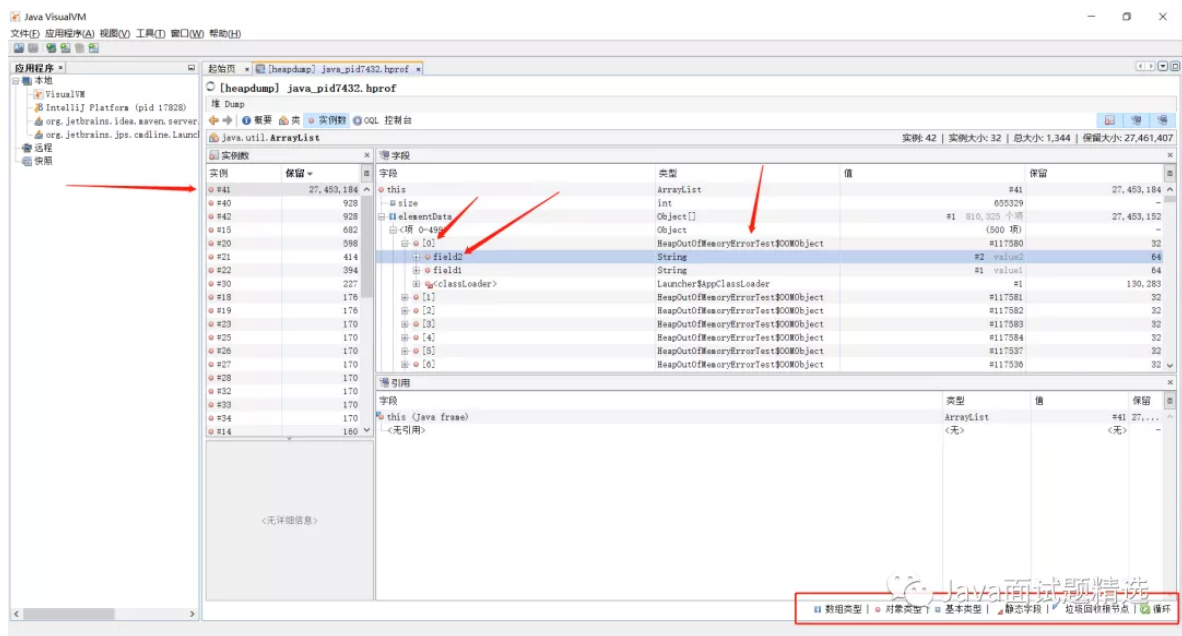
The screenshot shows the Java VisualVM interface. The 'Basic Information' tab is active, displaying the following details:

- 生成的日期:** Fri Dec 27 10:28:23 CST 2019
- 文件:** D:\dump\java_pid7432.hprof
- 文件大小:** 33.7 MB
- 字节总数:** 28,487,902
- 类总数:** 742
- 变量总数:** 608,727
- 类加载器:** 2
- 垃圾回收节点:** 870
- 等待结束的静态对象数:** 0

The 'Largest Objects' tab is also visible, showing a list of objects with their memory sizes. The objects are sorted by size, and the 'java.util.ArrayList' entry is highlighted, indicating it is the largest object.

类名	保留大小
java.util.ArrayList@1	27,453,184
java.lang.Object@1	27,453,152
sun.misc.Launcher\$AppClassLoader@1	130,283
class_sun.misc.cs.ext.GBK	113,740
sun.misc.URLClassPath@3	91,014
class_java.io.File	73,284
java.io.File@1	73,176
char[]@108	57,368
char[]@1	53,228
sun.misc.cs.ext.ExtendedCharsets@1	52,848
java.io.ExpiringCache@1	45,461
java.io.ExpiringCache@1	45,417
class_sun.misc.FBIInteger@1	41,233
sun.misc.FBIInteger@1	40,748
sun.security.provider.Sun@1	38,217
class_java.lang.String	37,048
java.util.TreeMap@2	30,176
java.util.TreeMap@1	30,096
java.io.ExpiringCache@2	27,677
java.io.ExpiringCache@2	27,633

2.找出具体的对象



3.解决方案

- 优化代码，去除大对象；
- 调整JVM内存大小（-Xmx与-Xms）；

2.超出GC开销限制

当出现 `java.lang.OutOfMemoryError: GC overhead limit exceeded` 异常信息时，表示超出了GC开销限制。当超过98%的时间用来做GC，但是却回收了不到2%的堆内存时会抛出此异常。

异常栈

```
[Full GC (Ergonomics) 19225K->19225K(19968K), 0.1044070 secs]
[Full GC (Ergonomics) 19227K->19227K(19968K), 0.0684710 secs]
java.lang.OutOfMemoryError: GC overhead limit exceeded
Dumping heap to D:\dump\java_pid17556.hprof ...
Heap dump file created [34925385 bytes in 0.132 secs]
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
[Full GC (Ergonomics) 19257K->933K(19968K), 0.0403569 secs]
at com.xiaolyuh.HeapOutOfMemoryErrorTest.main(HeapOutOfMemoryErrorTest.java:25)
ERROR: JDWP Unable to get JNI 1.2 environment, jvm->GetEnv() return code = -2
JDWP exit error AGENT_ERROR_NO_JNI_ENV(183): [util.c:840]
```

解决方案

- 通过 `-XX:-UseGCOverheadLimit` 参数来禁用这个检查，但是并不能从根本上来解决内存溢出的问题，最后还是会报出 `java.lang.OutOfMemoryError: Java heap space` 异常；
- 调整JVM内存大小（-Xmx与-Xms）；

虚拟机栈和本地方法栈溢出

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 `StackOverflowError` 异常。
- 如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 `OutOfMemoryError` 异常。

这里把异常分成两种情况，看似更加严谨，但却存在着一些互相重叠的地方：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质上只是对同一件事情的两种描述而已。

StackOverflowError

出现StackOverflowError异常的主要原因有两点：

- 单个线程请求的栈深度大于虚拟机所允许的最大深度
- 创建的线程过多

3.单个线程请求的栈深度过大

单个线程请求的栈深度大于虚拟机所允许的最大深度，主要表现有以下几点：

- 存在递归调用
- 存在循环依赖调用
- 方法调用链路很深，比如使用装饰器模式的时候，对已经装饰后的对象再进行装饰

异常信息 `java.lang.StackOverflowError`。

装饰器示例：

```
Collections.unmodifiableList(  
    Collections.unmodifiableList(  
        Collections.unmodifiableList(  
            Collections.unmodifiableList(  
                Collections.unmodifiableList(  
                    ...))))))));
```

递归示例：

```
/**  
 * java 虚拟机栈和本地方法栈内存溢出测试  
 * <p>  
 * VM Args: -Xss128k  
 *  
 * @author yuhao.wang3  
 */  
public class StackOverflowErrorErrorTest {  
    private int stackLength = 0;  
  
    public void stackLeak() {  
        stackLength++;  
        stackLeak();  
    }  
  
    public static void main(String[] args) {  
        StackOverflowErrorErrorTest sof = new StackOverflowErrorErrorTest();  
        try {  
            sof.stackLeak();  
        } catch (Exception e) {  
            System.out.println(sof.stackLength);  
            e.printStackTrace();  
        }  
    }  
}
```

```
}  
}
```

运行结果:

```
stackLength::1372  
java.lang.StackOverflowError  
    at  
com.xiaolyuh.StackOverflowErrorErrorTest.stackLeak(StackOverflowErrorErrorTest.j  
ava:16)  
    at  
com.xiaolyuh.StackOverflowErrorErrorTest.stackLeak(StackOverflowErrorErrorTest.j  
ava:16)  
    at  
com.xiaolyuh.StackOverflowErrorErrorTest.stackLeak(StackOverflowErrorErrorTest.j  
ava:16)  
    ...
```

当增大栈空间的时候我们会发现，递归深度会增加，修改栈空间-Xss1m，然后运行程序，运行结果如下:

```
stackLength::20641  
java.lang.StackOverflowError  
    at  
com.xiaolyuh.StackOverflowErrorErrorTest.stackLeak(StackOverflowErrorErrorTest.j  
ava:16)  
    at  
com.xiaolyuh.StackOverflowErrorErrorTest.stackLeak(StackOverflowErrorErrorTest.j  
ava:16)  
    ...
```

修改递归方法的参数列表后递归深度急剧减少:

```
public void stackLeak(String ags1, String ags2, String ags3) {  
    stackLength++;  
    stackLeak(ags1, ags2, ags3);  
}
```

运行结果如下:

```
stackLength::13154  
java.lang.StackOverflowError  
    at  
com.xiaolyuh.StackOverflowErrorErrorTest.stackLeak(StackOverflowErrorErrorTest.j  
ava:16)  
    at  
com.xiaolyuh.StackOverflowErrorErrorTest.stackLeak(StackOverflowErrorErrorTest.j  
ava:16)  
    ...
```

由此可见影响递归的深度因素有:

- 单个线程的栈空间大小 (-Xss)
- 局部变量表的大小

单个线程请求的栈深度超过内存限制导致的栈内存溢出，一般是由于非正确的编码导致的。从上面的示例我们可以看出，当栈空间在-Xss128k的时候，调用层级都在1000以上，一般情况下方法的调用是达不到这个深度的。如果方法调用的深度确实有这么大，那么我们可以通过-Xss配置来增大栈空间大小。

4. 创建的线程过多

不断地建立线程也可能导致栈内存溢出，因为我们机器的总内存是有限制的，所以虚拟机栈和本地方法栈对应的内存也是有最大限制的。如果单个线程的栈空间越大，那么整个应用允许创建的线程数就越少。异常信息 `java.lang.OutOfMemoryError: unable to create new native thread`。

虚拟机栈和本地方法栈内存 \approx 操作系统内存限制 - 最大堆容量(Xmx) - 最大方法区容量(MaxPermSize)

过多创建线程示例：

```
/**
 * java 虚拟机栈和本地方法栈内存溢出测试
 * <p>
 * 创建线程过多导致内存溢出异常
 * <p>
 * VM Args: -verbose:gc -Xss20M -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=D:\dump
 *
 * @author yuhao.wang3
 * @since 2019/11/30 17:09
 */
public class StackOutOfMemoryErrorTest {
    private static int threadCount;

    public static void main(String[] args) throws Throwable {
        try {
            while (true) {
                threadCount++;
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        try {
                            Thread.sleep(1000 * 60 * 10);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
            }
        } catch (Throwable e) {
            e.printStackTrace();
            throw e;
        } finally {
            System.out.println("threadCount=" + threadCount);
        }
    }
}
```

Java的线程是映射到操作系统的内核线程上，因此上述代码执行时有较大的风险，可能会导致操作系统假死。

运行结果：

```
java.lang.OutOfMemoryError: unable to create new native thread
  at java.lang.Thread.start0(Native Method)
  at java.lang.Thread.start(Thread.java:717)
  at StackOutOfMemoryErrorTest.main(StackOutOfMemoryErrorTest.java:17)
threadCount=4131
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
native thread
  at java.lang.Thread.start0(Native Method)
  at java.lang.Thread.start(Thread.java:717)
  at StackOutOfMemoryErrorTest.main(StackOutOfMemoryErrorTest.java:17)
```

需要重新上述异常，最好是在32位机器上，因为我在64位机器没有重现。

在有限的内存空间里面，当我们需要创建更多的线程的时候，我们可以减少单个线程的栈空间大小。

5. 元数据区域的内存溢出

元数据区域或方法区是用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。我们可以通过在运行时产生大量的类去填满方法区，直到溢出，如：代理的使用(CGLib)、大量JSP或动态产生JSP文件的应用（JSP第一次运行时需要编译为Java类）、基于OSGi的应用（即使是同一个类文件，被不同的加载器加载也会视为不同的类）等。

```
/**
 * java 元数据区域/方法区的内存溢出
 * <p>
 * VM Args JDK 1.6: set JAVA_OPTS=-verbose:gc -XX:PermSize=10m -
XX:MaxPermSize=10m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:\dump
 * <p>
 * VM Args JDK 1.8: set JAVA_OPTS=-verbose:gc -Xmx20m -XX:MetaspaceSize=5m -
XX:MaxMetaspaceSize=5m -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:\dump
 *
 * @author yuhao.wang3
 */
public class MethodAreaOutOfMemoryErrorTest {

    static class MethodAreaOOM {

    }

    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(MethodAreaOOM.class);
            enhancer.setCallback(new MethodInterceptor() {
                @Override
                public Object intercept(Object obj, Method method, Object[]
params, MethodProxy proxy) throws Throwable {
                    return proxy.invokeSuper(obj, params);
                }
            });
            enhancer.create();
        }
    }
}
```

运行结果：

```

[GC (Last ditch collection) 1283K->1283K(16384K), 0.0002585 secs]
[Full GC (Last ditch collection) 1283K->1226K(19968K), 0.0075856 secs]
java.lang.OutOfMemoryError: Metaspace
Dumping heap to D:\dump\java_pid18364.hprof ...
Heap dump file created [2479477 bytes in 0.015 secs]
[GC (Metadata GC Threshold) 1450K->1354K(19968K), 0.0003906 secs]
[Full GC (Metadata GC Threshold) 1354K->976K(19968K), 0.0073752 secs]
[GC (Last ditch collection) 976K->976K(19968K), 0.0002921 secs]
[Full GC (Last ditch collection) 976K->973K(19968K), 0.0045243 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:368)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:362)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:361)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at
    org.springframework.cglib.core.internal.LoadingCache.createEntry(LoadingCache.java:52)
    at
    org.springframework.cglib.core.internal.LoadingCache.get(LoadingCache.java:34)
    at
    org.springframework.cglib.core.AbstractClassGenerator$ClassLoaderData.get(AbstractClassGenerator.java:116)
    at
    org.springframework.cglib.core.AbstractClassGenerator.create(AbstractClassGenerator.java:291)
    at
    org.springframework.cglib.core.KeyFactory$Generator.create(KeyFactory.java:221)
    at org.springframework.cglib.core.KeyFactory.create(KeyFactory.java:174)
    at org.springframework.cglib.core.KeyFactory.create(KeyFactory.java:153)
    at org.springframework.cglib.proxy.Enhancer.<clinit>(Enhancer.java:73)
    at
    com.xiaolyuh.MethodAreaOutOfMemoryErrorTest.main(MethodAreaOutOfMemoryErrorTest.java:26)

```

6. 运行时常量池的内存溢出

String.intern()是一个Native方法，它的作用是：如果字符串常量池中已经包含一个等于此String对象的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。

在JDK 1.6的时候，运行时常量池是在方法区中，所以直接限制了方法区中大小就可以模拟出运行池常量池的内存溢出。

```

/**
 * java 方法区和运行时常量池溢出
 * <p>
 * VM Args JDK 1.6: set JAVA_OPTS=-verbose:gc -XX:PermSize10 -XX:MaxPermSize10m
 * -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:\dump
 *

```



```

* @author yuhao.wang3
*/
public class RuntimeConstantOutOfMemoryErrorTest {

    public static void main(String[] args) {
        // 使用List保存着常量池的引用，避免Full GC 回收常量池行为
        List<String> list = new ArrayList<>();
        for (int i = 0; ; i++) {
            list.add(String.valueOf(i).intern());
        }
    }
}

```

运行结果:

```

Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
    at java.lang.String.intern(Native Method)
    at
    RuntimeConstantOutOfMemoryErrorTest.main(RuntimeConstantOutOfMemoryErrorTest.java:18)

```

7.直接内存溢出

DirectMemory容量可通过 `-XX:MaxDirectMemorySize` 指定，如果不指定，则默认与Java堆最大值（`-Xmx`指定）一样。

```

/**
 * java 直接内存溢出
 * <p>
 * VM Args JDK 1.6: set JAVA_OPTS=-verbose:gc -Xms20m -
XX:MaxDirectMemorySize=10m -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=D:\dump
 *
 * @author yuhao.wang3
 */
public class DirectMemoryOutOfMemoryErrorTest {

    public static void main(String[] args) throws IllegalAccessException {
        int _1M = 1024 * 1024;
        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe) unsafeField.get(null);
        while (true) {
            unsafe.allocateMemory(_1M);
        }
    }
}

```

运行结果:

```

Exception in thread "main" java.lang.OutOfMemoryError
    at sun.misc.Unsafe.allocateMemory(Native Method)
    at
    com.xiaolyuh.DirectMemoryOutOfMemoryErrorTest.main(DirectMemoryOutOfMemoryErrorTest.java:23)

```

由DirectMemory导致的内存溢出，一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果读者发现OOM之后Dump文件很小，而程序中又直接或间接使用了NIO，那就可以考虑检查一下是不是这方面的原因。

解决方案

通过 `-XX:MaxDirectMemorySize` 指定直接内存大小。