图文 076、动手实验：自己模拟出JVM Metaspace内存溢出的场景体验一下！

详情 评论

动手实验：

自己模拟出JVM Metaspace内存溢出的场景体验一下！

狸猫技术窝专栏上新，基于**真实订单系统**的消息中间件（mq）实战，重磅推荐：

未来3个月，我的好朋友原子弹大侠将带你一起，全程实战，360度死磕MQ

（点击下方蓝字进行试听）

从 0 开始带你成为消息中间件实战高手

重要说明：

如何提问：每篇文章都有评论区，大家可以尽情在评论区留言提问，我都会逐一答疑

（**ps**：评论区还精选了一些小伙伴对**专栏每日思考题的作答**，有的答案真的非常好！大家可以通过看别人的思路，启发一下自己，从而加深理解）

如何加群：购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**。

（群里有不少**一二线互联网大厂的助教**，大家可以一起讨论交流各种技术）

具体**加群方式**请参见文末。

（**注**：以前通过其他专栏加过群的同学就不要重复加了）

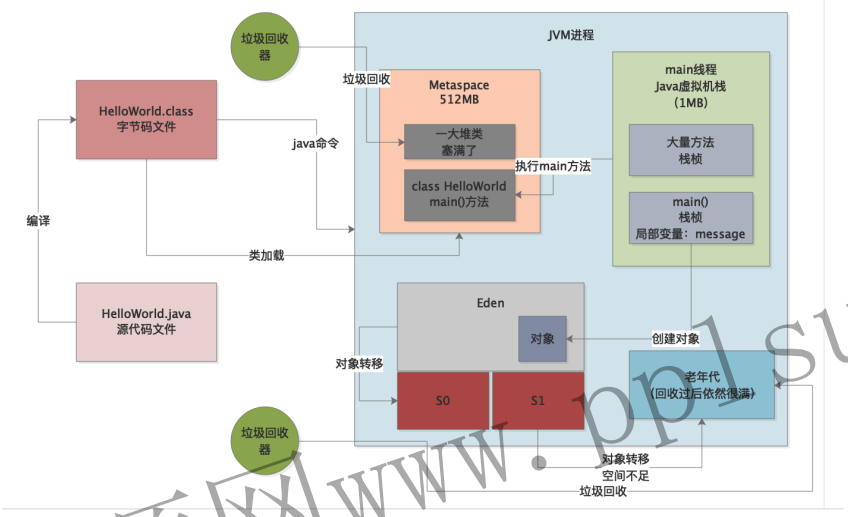
1、前文回顾

上一周我们已经把内存溢出问题产生的根本原理都给大家做了详细的说明，这一周我们就进入实际的动手实验环境，带着大家来动手体验一下Metaspace、栈内存、堆内存分别都是怎么溢出的，溢出的时候会出现什么样的错误。

然后我们本周稍晚还会给出两个真实的生产案例来告诉大家平时在工作的时候，什么样的场景下可能会导致这三个区域的内存溢出。

2、Metaspace内存溢出原理回顾

首先，我们来回顾一下之前一直用的一张图片，大家看下图：



在这张图片里，我们通过之前的讲解，可以清晰的看到JVM整体的运行原理，包括类加载，线程执行方法，虚拟机栈，堆内存创建对象，GC以及对象转移老年代，等等。而且在这个图里，我们也清晰标志出来了哪些环节可能会发生内存溢出。

因此在这里我们来回顾一下，Metaspace区域发生内存溢出的一个场景，说白了就是如果我们在程序里不停的动态生成类，就会导致不停的加载类到Metaspace区域里去，而且这些动态生成的类必须还得是不能被回收掉的。

接着一旦Metaspace区域满了，就会触发Full GC连带着回收Metaspace中的类，但是此时大量的类是不能被回收的。

因此即使触发过Full GC过后，Metaspace区域几乎还是不能放下任何一个类，此时必然会触发Metaspace区域的内存溢出，导致JVM也是崩溃掉，无法继续运行了。

3、到底什么是动态生成类?

可能有的人不太理解什么叫做动态生成类，其实很简单，我们平时正常情况下，类都是通过自己的双手一行一行代码写出来的，而且都是写的“.java”后缀的源代码文件，大家想想是不是这样?

所以很多人可能想当然的以为在JVM中的类都是我们双手写出来的，其实并不是这样子。

大家回忆一下，平时我们自己写出来的类大致长什么样子？是不是一般都包含一些静态变量、实例变量、静态方法、实例方法，里面还有一大堆的业务逻辑？大致其实类就是这么个东西。

所以既然你双手都能写出来这种普普通通的类，那么当然是有办法可以借助一些方法在系统运行的时候，通过程序动态的生成出更多的类了，这是没有问题的。

所以一旦我们程序中拼命的生成大量的类，而且这些类还不能被回收，那么必然会最终导致Metaspace区域被占满，进而导致Metaspace内存溢出了。

接着我们就来实际看看代码层面上，动态生成类到底是怎么做的吧!

4、一段CGLIB动态生成类的代码示例

相信大家平时无论用Eclipse还是用IntelliJ IDEA，应该都大部分人都是基于Maven来进行项目构建的，当然现在也有一些公司在用Gradle进行项目构建。

我们这里就以Maven来举例好了，如果要用CGLIB来动态生成一些类，那么必须在你项目的pom.xml中引入以下的一些依赖。

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>3.3.0</version>
</dependency>
```

接着我们就可以使用CGLIB来动态生成类了，大家看下面的代码：

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class Demo1 {

  public static void main(String[] args) {
    while(true) {
      Enhancer enhancer = new Enhancer();
      enhancer.setSuperclass(Car.class);
      enhancer.setUseCache(false);
      enhancer.setCallback(new MethodInterceptor() {
        public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
          if(method.getName().equals("run")) {
            System.out.println("启动汽车之间，先进行自动的安全检查.....");
            return methodProxy.invokeSuper(o, objects);
          } else {
            return methodProxy.invokeSuper(o, objects);
          }
        }
      });

      Car car = (Car) enhancer.create();
      car.run();
    }
  }

  static class Car {

    public void run() {
      System.out.println("汽车启动，开始行使.....");
    }

  }

}
```

上面那段代码稍微有点复杂是不是？

没关系，给大家解释一下，这个所谓的动态生成类大概是个什么意思。

首先我们可以看到我们在这里定义了一个类，代表了一个汽车，他有一个run()方法，执行的时候就会启动汽车，开始让汽车行驶，大家看下面的这个代码片段：

```
  static class Car {

    public void run() {
      System.out.println("汽车启动，开始行使.....");
    }

  }
```

相信关于小汽车的这个类的定义，大家都是没有问题的。那么我们接着来看下面的代码片段，我们通过CGLIB的Enhancer类生成了一个Car类的子类

注意，从这里开始，就是开始动态生成类了，大家要仔细看，看下面的代码片段：

```
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(Car.class);
enhancer.setUseCache(false);
```

你权且当做Enhancer是用来生成类的一个API吧，看到片段里我们给Enhancer设置了一个SuperClass没有？这里的意思就是说Enhancer生成的类是Car类的子类，Car类是生成类的父类。至于那个UseCache是什么意思，就先别管了。

既然Enhancer动态生成的类是Car的子类，那么是不是Car有的方法子类都有？所以子类是不是也有Car的run()方法?

答案是肯定的，但我们现在想要在调用子类的run()方法的时候做点手脚，如下面代码片段：

```
        enhancer.setCallback(new MethodInterceptor() {
          public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
            if(method.getName().equals("run")) {
              System.out.println("启动汽车之间，先进行自动的安全检查......");
              return methodProxy.invokeSuper(o, objects);
            } else {
              return methodProxy.invokeSuper(o, objects);
            }
          }
        });
```

这个片段的意思是：如果你调用子类对象的run()方法，会先被这里的MethodInterceptor拦截一下，拦截之后，各位看里面的代码，是不是判断了一下，如果你调用的Method是run方法，那么就先对汽车做一下安全检查。

安全检查做完之后，再通过“methodProxy.invokeSuper(o, objects);”调用父类Car的run()方法，去启动汽车，这行代码就会执行到Car类的run()方法里去了。

到此为止，我们就已经通过CGLIB的Enhancer生成了一个Car类的子类了，而且定义好了对这个子类调用继承自父类的run()方法的时候，先干点别的，再调用父类的run()方法。

这么一搞，是不是跟下面这种在IDE里手写一个Car的子类是类似的?

看看下面的手写版本的代码：

```
static class SafeCar extends Car {

    @Override
    public void run() {
        System.out.println("汽车启动，开始行使......");
        super.run();
    }

}
```

看看上面那个SafeCar作为Car的子类，是不是干了一样的事?

但是这个类需要你用双手提前写出来代码，而CGLIB Enhancer那种模式可以在系统运行期间动态的创建一个Car的子类出来，实现一样的效果

看到这里，各位应该理解这个动态创建类了!

5、限制Metaspace大小看看内存溢出效果

接着我们可以设置一下这个程序的JVM参数，限制他的Metaspace区域比较小一点，如下所示，我们把这个程序的JVM中的Metaspace区域设置为仅仅10m：

-XX:MetaspaceSize=10m -XX:MaxMetaspaceSize=10m

接着我们可以在上述代码中做点手脚，大家看到上面的代码是有一个while循环的，所以他会不停的创建Car类的子类

我们在里面可以加入一个计数器，就是看看当前创建了多少个Car的子类了，如下所示：

```
long counter = 0;

while(true) {
    System.out.println("目前创建了" + (++counter) + "个Car类的子类了");
}
```

在while循环外面加一个计数器，然后打印出当前创建了多少个类了。

接着大家用上述JVM参数来运行这个程序即可，可以看到如下所示的打印输出：

目前创建了263个Car类的子类了

Exception in thread "main" java.lang.IllegalStateException: Unable to load cache item
at net.sf.cglib.core.internal.LoadingCache.createEntry(LoadingCache.java:79)
at net.sf.cglib.core.internal.LoadingCache.get(LoadingCache.java:34)
at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData.get(AbstractClassGenerator.java:119)
at net.sf.cglib.core.AbstractClassGenerator.create(AbstractClassGenerator.java:294)
at net.sf.cglib.reflect.FastClass$Generator.create(FastClass.java:65)
at net.sf.cglib.proxy.MethodProxy.helper(MethodProxy.java:121)
at net.sf.cglib.proxy.MethodProxy.init(MethodProxy.java:75)
at net.sf.cglib.proxy.MethodProxy.invokeSuper(MethodProxy.java:226)
at com.limao.demo.jvm.Demo1$1.intercept(Demo1.java:22)
at com.limao.demo.jvm.Demo1$Car$$EnhancerByCGLIB$$7e5aa3a5_264.run(<generated>)
at com.limao.demo.jvm.Demo1.main(Demo1.java:30)
Caused by: java.lang.OutOfMemoryError: Metaspace
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Class.java:348)
at net.sf.cglib.core.ReflectUtils.defineClass(ReflectUtils.java:467)
at net.sf.cglib.core.AbstractClassGenerator.generate(AbstractClassGenerator.java:339)
at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassGenerator.java:96)
at net.sf.cglib.core.AbstractClassGenerator$ClassLoaderData$3.apply(AbstractClassGenerator.java:94)
at net.sf.cglib.core.internal.LoadingCache$2.call(LoadingCache.java:54)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at net.sf.cglib.core.internal.LoadingCache.createEntry(LoadingCache.java:61)
... 10 more

大家注意一下上述异常日志的两个地方，一个是在创建了263个类之后，10M的Metaspace区域就被耗尽了，接着就会看到异常中有如下的一个：

Caused by: java.lang.OutOfMemoryError: Metaspace。

这个OutOfMemoryError就是经典的内存溢出的问题，而且他明确告诉你，是Metaspace这块区域内存溢出了。

而且大家可以看到，一旦内存溢出，本来在运行的JVM进程直接会崩溃掉，你的程序会退出，这就是真实的内存溢出的日志。

6、本文总结

本文带着大家用一段CGLIB动态生成类的代码演示了一下Metaspace区域内存溢出的场景，相信大家自己动手做一下实验，就可以切实感受到这个内存溢出的效果了。

End
