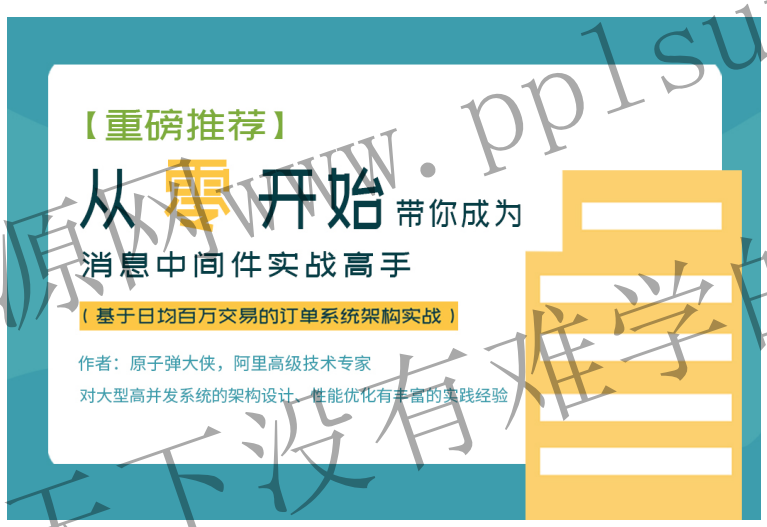
图文 080、案例实战：两个新手工程师误写代码是如何导致OOM的?

详情 评论

案例实战：
两个新手工程师误写代码是如何导致OOM的?

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

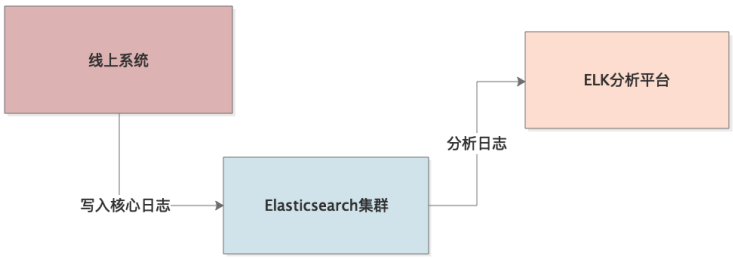
上周五的文章给大家介绍了一个系统设计上的缺陷导致的极端场景下的OOM，让大家从较为真实的系统案例中感受一下OOM到底是如何发生的。

今天的文章就给大家说一说之前遇到过的团队里两个新手工程师误写代码导致的OOM的问题。

2、第一个案例：一时迷糊写出了一个无限循环调用

第一个案例是当时团队里招聘的一个实习生同学，写出了一个代码上的bug导致线上系统出现栈内存溢出的场景。

当时有一个非常重要的系统，我们设计了一个链路监控机制，也就是会在一个比较核心的链路节点，写一些重要的日志到Elasticsearch集群里去，事后会基于ELK进行核心链路日志的一些分析，如下图所示。



同时我们对这个机制做了规定，如果在某个节点写日志时发生了某些异常，此时也必须将这个链路节点的异常写入ES集群里去，因为我们在分析的时候，需要知道系统运行到这里有一个异常。

因此当时那个实习生同学写出来的伪代码大致如下：

```
try {
    // 一大堆的业务逻辑
    log();
} catch(Exception e) {
    log();
}
public void log() {
    try {
        // 将日志写入ES集群
    } catch(Exception e) {
        log();
    }
}
```

不知道大家看了上面的代码是作何感想？当时这个同学居然在log()方法中一旦ES集群出现故障的时候再次调用了自己，继续尝试将日志写入ES集群。

因此在线上系统中，有一次ES集群短暂故障了一会儿，结果直接就导致log()方法中写ES集群每次都是失败的，都会抛异常。

而一旦抛异常进入了catch语句中，就会再次重新回过头来调用log()方法。

然后log()方法再次写ES集群发现不行，继续抛异常进入catch中，再次循环调用自己。

线上系统本来在ES集群故障的时候不该有什么问题的，因为核心业务逻辑都是可以运行的，最多不过就是无法把核心日志写入ES集群罢了。

但是因为这个bug，导致在ES故障时，所有系统全部在写日志的时候，陷入了一个无限循环调用log()方法的困境中。

之前给大家演示过，一旦无限循环调用方法自己，一定会在一定时间导致线程的栈内存溢出的，此时直接会导致JVM进程的崩溃

系统居然因为这么一个小问题崩溃了！这就是一次非常真实的线上案例。

后来针对此类问题，我们都是通过严格的持续集成+严格的Code Review标准来避免的

每个人每天都会写一点代码，这个代码必须是配套单元测试可以运行的，然后全部提交到持续集成服务器上去，代码集成到整体代码中，自动运行全部单元测试+集成测试。

在单元测试+集成测试中，我们都是要求针对一些try catch中可能走到catch的分支写一些测试的，因此一旦有这类代码，每天只要工程师提交到持续集成系统上去，立马就会自动运行测试触发这个问题，就可以立刻解决了。

而且每天这一点代码也必须交给指定的其他同事进行Code Review，别人会仔细看你今天写的每一行代码，做一个审查，一旦发现问题也会打回去重新修改代码。从此之后，这种低端的问题再也没有发生过。

3、第二个案例：没有缓存的动态代理

第二个案例同样是之前的一个新手工程师写的，这个并不是实习生，是一个校招生同学，在团队里工作了1年左右的时间。

但是确实因为经验不足，有一次在实现一块代码机制的时候，也是犯了一个很大的错误。

简单来说，当时这个同学想要实现一个动态代理机制，也就是说在系统运行的时候，针对已有的某个类，生成一个动态代理类，也就是动态生成类，然后对那个类的一些方法调用做一些额外的处理。

当时这个是一个我们自己研发的分布式事务框架，对于这个框架是有这位同学参与在里面的，因此在框架中有时候要对一些已经有的类实现动态代理，去实现分布式事务一些的复杂底层机制。

当时大概的一个伪代码其实跟之前给大家的代码是类似的：

```
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();
}
```

不知道大家发现类似这种代码里的一个问题没有？比如你用CGLIB的Enhancer针对某个类动态生成了一个子类，这个子类你完全可以缓存起来，下次直接用这个已经生成好的子类来创建对象就可以了

类似下面这样：

```
private volatile Enhancer enhancer = null;

public void doSomething() {
if(enhancer == null) {
this.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();
}
```

其实这个类只要生成一次就可以了，下次来直接用这个动态生成的类创建一个对象就可以了。

但是当时那个工程师没有缓存这个动态生成的类，就是每次调用方法都生成一个类，这就闯祸了。

有一次线上系统负载很高的时候，因为这个框架直接导致瞬间创建了一大堆的类，塞满了Metaspace区域无法回收，进而导致Metaspace区域直接内存溢出，系统也崩溃了，这也是一个很大的问题。

后来对于这类问题，是严格要求每次上线必须走严格的自动化压力测试，通过高并发压力下系统是否正常运行支撑24小时，来判断是否可以上线。

这样类似于这类代码在上线之前就会被压力测试露出马脚，因为压力一大，瞬间会引发这个问题。

4、本文总结

这周的文章，我们带着大家感受了一下各种内存溢出发生的场景，同时给出了几个真实的线上生产案例是如何导致各个内存区域溢出的

相信大家对内存溢出这个问题，有了一个更加深刻的理解。

接下来我们会带着大家一起来学习如何对线上的OOM进行监控，同时在OOM时如何让JVM自动保留现场，同时结合几个案例和工具学习，发生OOM之后如何快速排查和定位到底代码哪里出现了OOM，以及如何进行解决。

最后做一个说明，有同学反馈专栏的目录和最初的大纲写的略有出入。这是因为整个专栏的写作是一个很长的过程，在这过程中我会根据大家每天的提问反馈，对大家的掌握专栏的情况进行评估，然后在此基础上进行微调，以便于大家更好的掌握。专栏干货只会多，不会少，这个请大家放心。

End
