想要完成JVM的性能调优，需要具备以下几个方面。

知识，工具，数据，经验

# 系统卡顿

【内容】绩效考核系统，针对每个员工进行考核，生成一个全部考核点的考核结果，形成Excel文档，供用户下载。文档中包括用户提交的考核信息以及分析信息。Excel文档由用户请求的时候生成，下载，并保存到内存服务器一份。

【环境】

* 2个Intel Xeon E5 4620 V4 CPU
* 64G 内存
* CentOS
* Tomcat 7 (单服务应用)
* JDK7

【问题】

经常有用户反映长时间的出现卡顿的信息

【处理思路】

* 优化SQL（只能针对某个功能）
* 监控CPU使用率
* 监控内存
  + Full GC 的时间长（fullGC的时候整个JVM是阻塞的，会造成系统卡顿）分析可能的原因存在很大的对象（保存在老年代中），当老年代使用率高的时候回触发Full GC，由于堆内存较大Full GC的时间也会变成。

【解决方案】

部署多个web容器，每个web容器指定堆内存为4G。（注意点：不要给应用设置太大的堆内存，会影响gc回收时间）

# 内容溢出 OOM

### Java 堆内存溢出

在 Java 堆中只要不断的创建对象，并且 GC-Roots 到对象之间存在引用链，这样 JVM 就不会回收对象。

只要将-Xms(最小堆),-Xmx(最大堆) 设置为一样禁止自动扩展堆内存。

当使用一个 while(true) 循环来不断创建对象就会发生 OutOfMemory，还可以使用 -XX:+HeapDumpOutofMemoryErorr 当发生 OOM 时会自动 dump 堆栈到文件中。

伪代码:

public static void main(String[] args) {

List<String> list = new ArrayList<>(10) ;

while (true){

list.add("1") ;

}

}

当出现 OOM 时可以通过工具来分析 GC-Roots 引用链 ，查看对象和 GC-Roots 是如何进行关联的，是否存在对象的生命周期过长，或者是这些对象确实改存在的，那就要考虑将堆内存调大了。

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.crossoverjie.oom.HeapOOM.main(HeapOOM.java:18)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:147)

Process finished with exit code 1

java.lang.OutOfMemoryError: Java heap space表示堆内存溢出。

更多内存溢出相关实战请看这里：强如 Disruptor 也发生内存溢出？

### MetaSpace (元数据) 内存溢出

JDK8 中将永久代移除，使用 MetaSpace 来保存类加载之后的类信息，字符串常量池也被移动到 Java 堆。

PermSize 和 MaxPermSize 已经不能使用了，在 JDK8 中配置这两个参数将会发出警告。

JDK 8 中将类信息移到到了本地堆内存(Native Heap)中，将原有的永久代移动到了本地堆中成为 MetaSpace ,如果不指定该区域的大小，JVM 将会动态的调整。

可以使用 -XX:MaxMetaspaceSize=10M 来限制最大元数据。这样当不停的创建类时将会占满该区域并出现 OOM。

public static void main(String[] args) {

while (true){

Enhancer enhancer = new Enhancer() ;

enhancer.setSuperclass(HeapOOM.class);

enhancer.setUseCache(false) ;

enhancer.setCallback(new MethodInterceptor() {

@Override

public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

return methodProxy.invoke(o,objects) ;

}

});

enhancer.create() ;

}

}

使用 cglib 不停的创建新类，最终会抛出:

Caused by: java.lang.reflect.InvocationTargetException

at sun.reflect.GeneratedMethodAccessor1.invoke(Unknown Source)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at net.sf.cglib.core.ReflectUtils.defineClass(ReflectUtils.java:459)

at net.sf.cglib.core.AbstractClassGenerator.generate(AbstractClassGenerator.java:336)

... 11 more

Caused by: java.lang.OutOfMemoryError: Metaspace

at java.lang.ClassLoader.defineClass1(Native Method)

at java.lang.ClassLoader.defineClass(ClassLoader.java:763)

... 16 more

注意：这里的 OOM 伴随的是 java.lang.OutOfMemoryError: Metaspace 也就是元数据溢出。

# 线程池队列达到阈值触发报警

【背景】上午刚到公司，准备开始一天的摸鱼之旅时突然收到了一封监控中心的邮件。

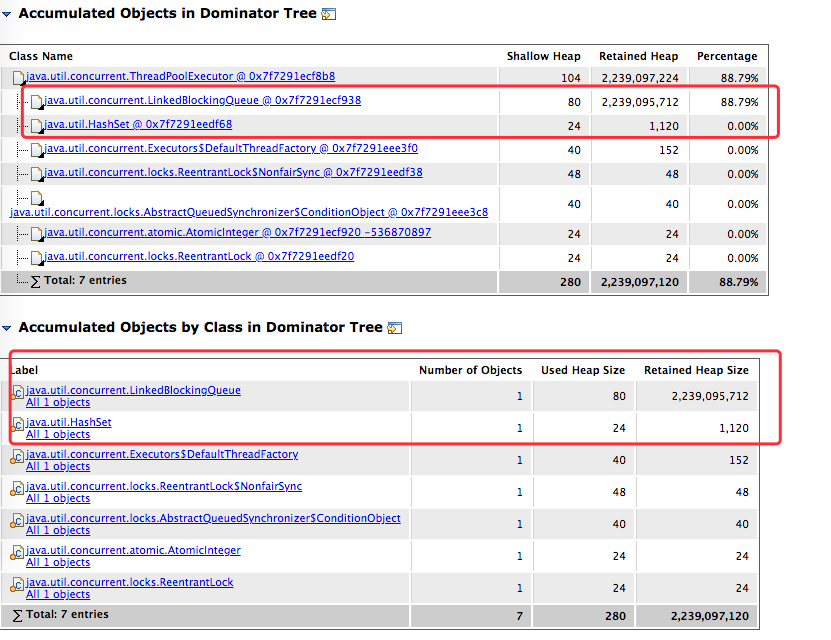
心中暗道不好，因为监控系统从来不会告诉我应用完美无 bug，其实系统挺猥琐。

打开邮件一看，果然告知我有一个应用的线程池队列达到阈值触发了报警。

由于这个应用出问题非常影响用户体验；

于是立马让运维保留现场 dump 线程和内存同时重启应用，还好重启之后恢复正常。于是开始着手排查问题。

【内存分析】先利用 MAT 分析了内存，的到了如下报告。



其中有两个比较大的对象，一个就是之前线程池存放任务的 LinkedBlockingQueue，还有一个则是 HashSet。

当然其中队列占用了大量的内存，所以优先查看，HashSet 一会儿再看。

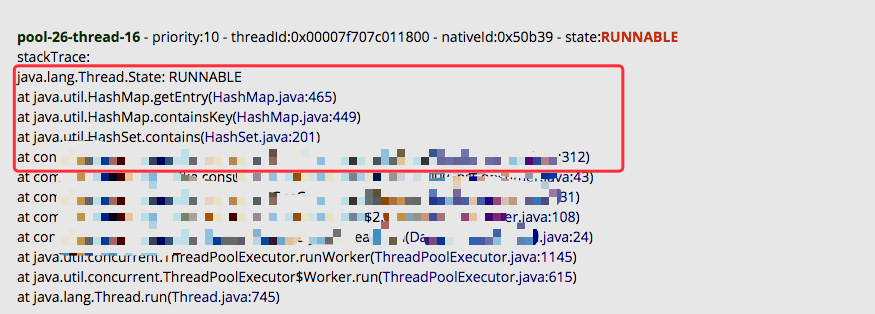
由于队列的大小给的够大，所以结合目前的情况来看应当是线程池里的任务处理较慢，导致队列的任务越堆越多，至少这是目前可以得出的结论。

【线程分析】http://fastthread.io/index.jsp

再来看看线程的分析，这里利用 fastthread.io 这个网站进行线程分析。

因为从表现来看线程池里的任务迟迟没有执行完毕，所以主要看看它们在干嘛。

正好他们都处于 RUNNABLE 状态，同时堆栈如下：



发现正好就是在处理上文提到的 HashSet，看这个堆栈是在查询 key 是否存在。通过查看 312 行的业务代码确实也是如此。这里的线程名字也是个坑，让我找了好久。

【定位】

分析了内存和线程的堆栈之后其实已经大概猜出一些问题了。

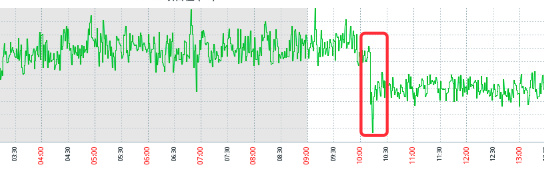
这里其实有一个前提忘记讲到：

这个告警是凌晨三点发出的邮件，但并没有电话提醒之类的，所以大家都不知道。

到了早上上班时才发现并立即 dump 了上面的证据。

所有有一个很重要的事实：这几个业务线程在查询 HashSet 的时候运行了 6 7 个小时都没有返回。

通过之前的监控曲线图也可以看出：



操作系统在之前一直处于高负载中，直到我们早上看到报警重启之后才降低。

同时发现这个应用生产上运行的是 JDK1.7 ，所以我初步认为应该是在查询 key 的时候进入了 HashMap 的环形链表导致 CPU 高负载同时也进入了死循环。

为了验证这个问题再次 review 了代码。

整理之后的伪代码如下：

//线程池

private ExecutorService executor;

private Set<String> set = new hashSet();

private void execute(){

while(true){

//从 MQ 中获取数据

String key = subMQ();

executor.excute(new Worker(key)) ;

}

}

public class Worker extends Thread{

private String key ;

public Worker(String key){

this.key = key;

}

@Override

private void run(){

if(!set.contains(key)){

//数据库查询

if(queryDB(key)){

set.add(key);

return;

}

}

//达到某种条件时清空 set

if(flag){

set = null ;

}

}

}

大致的流程如下：

* 程序从 MQ 中获取数据。
* 将数据丢到业务线程池中。
* 判断数据是否已经写入了 Set。
* 没有则查询数据库。
* 之后写入到 Set 中。

这里有一个很明显的问题，那就是作为共享资源的 Set 并没有做任何的同步处理。

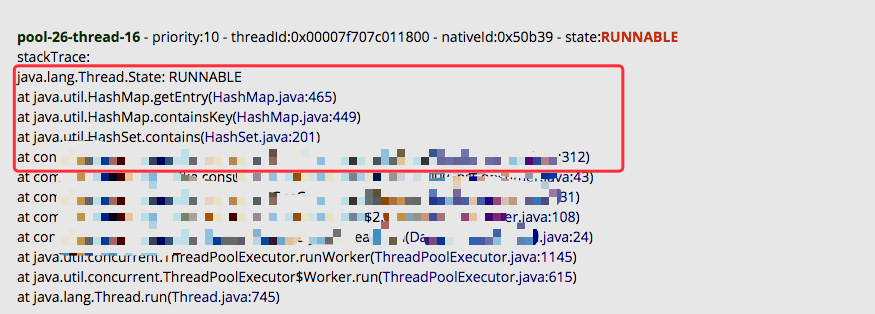
这里会有多个线程并发的操作，由于 HashSet 其实本质上就是 HashMap，所以它肯定是线程不安全的，所以会出现两个问题：

* Set 中的数据在并发写入时被覆盖导致数据不准确。
* 会在扩容的时候形成环形链表。

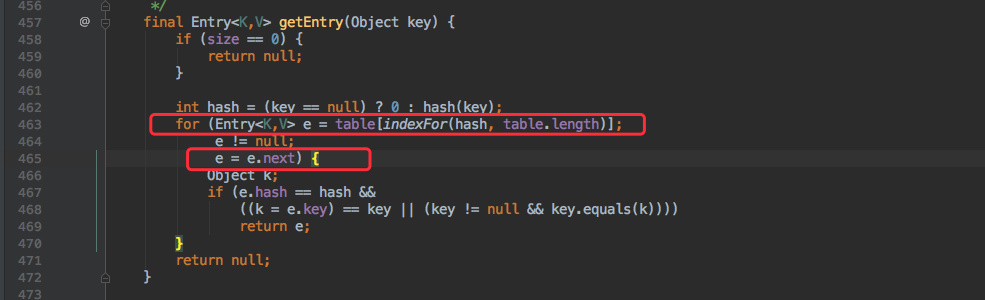
第一个问题相对于第二个还能接受。

通过上文的内存分析我们已经知道这个 set 中的数据已经不少了。同时由于初始化时并没有指定大小，仅仅只是默认值，所以在大量的并发写入时候会导致频繁的扩容，而在 1.7 的条件下又可能会形成环形链表。

不巧的是代码中也有查询操作（contains()）,观察上文的堆栈情况：



发现是运行在 HashMap 的 465 行，来看看 1.7 中那里具体在做什么：



已经很明显了。这里在遍历链表，同时由于形成了环形链表导致这个 e.next 永远不为空，所以这个循环也不会退出了。

到这里其实已经找到问题了，但还有一个疑问是为什么线程池里的任务队列会越堆越多。我第一直觉是任务执行太慢导致的。

仔细查看了代码发现只有一个地方可能会慢：也就是有一个数据库的查询。

把这个 SQL 拿到生产环境执行发现确实不快，查看索引发现都有命中。

但我一看表中的数据发现已经快有 7000W 的数据了。同时经过运维得知 MySQL 那台服务器的 IO 压力也比较大。

所以这个原因也比较明显了：

由于每消费一条数据都要去查询一次数据库，MySQL 本身压力就比较大，加上数据量也很高所以导致这个 IO 响应较慢，导致整个任务处理的就比较慢了。

但还有一个原因也不能忽视；由于所有的业务线程在某个时间点都进入了死循环，根本没有执行完任务的机会，而后面的数据还在源源不断的进入，所以这个队列只会越堆越多！

这其实是一个老应用了，可能会有人问为什么之前没出现问题。

这是因为之前数据量都比较少，即使是并发写入也没有出现并发扩容形成环形链表的情况。这段时间业务量的暴增正好把这个隐藏的雷给揪出来了。所以还是得信墨菲他老人家的话。

【总结】

至此整个排查结束，而我们后续的调整措施大概如下：

HashSet 不是线程安全的，换为 ConcurrentHashMap同时把 value 写死一样可以达到 set 的效果。

根据我们后面的监控，初始化 ConcurrentHashMap 的大小尽量大一些，避免频繁的扩容。

MySQL 中很多数据都已经不用了，进行冷热处理。尽量降低单表数据量。同时后期考虑分表。

查数据那里调整为查缓存，提高查询效率。

线程池的名称一定得取的有意义，不然是自己给自己增加难度。

根据监控将线程池的队列大小调整为一个具体值，并且要有拒绝策略。

升级到 JDK1.8。

再一个是报警邮件酌情考虑为电话通知😂。

HashMap 的死循环问题在网上层出不穷，没想到还真被我遇到了。现在要满足这个条件还是挺少见的，比如 1.8 以下的 JDK 这一条可能大多数人就碰不到，正好又证实了一次墨菲定律。