

## 20 | Java & C ++：代码级监控及常用计数器解析（下）

2020-02-05 高楼

性能测试实战30讲

[进入课程 >](#)



讲述：高楼

时长 21:26 大小 19.63M



在上一篇文章中，我们描述了在 Java 开发语言中如何抓取方法的执行时间，其中描述的操作也是我们在分析时经常使用的。

今天我们将接着描述如下几点内容：

1. Java 语言中如何查找有问题的内存对象。
2. 简单介绍一下在 C/C++ 语言中如何查找方法执行时间和对象的内存消耗。



之所以要描述 C/C++ 语言的相关内容，就是为了告诉你，几乎在任何一语言中都有相应的工具，都有办法捕获到相应的内容。

下面我们来看看如何抓取 Java 应用中对象占用多大内存，以及如何分辨占用是合理的和不合理的。

## Java 类应用查找对象内存消耗

对 Java 的内存分析通常都落在对 JVM 的使用上（不要认为我这句话说得片面），再具体一点，说的就是内存泄露和内存溢出。由于现在对象都是可变长的，内存溢出就不常见了；而由于底层框架的慢慢成熟，内存泄露现在也不常见了。

有人说了，那你还啰嗦个什么劲呢？别捉急呀，不常见不等于没有。只是说它不再是 No.1 级的问题，但是排在 No.2 级还是没问题的。

如果你的应用有了问题，看到了像这样的图：



这是我在一个项目中遇到的问题，图片不够清晰，我们只要关注黄线的趋势就好。

之所以把它拿出来事，是因为这个问题太极端了。上图是近 20 天的 JVM 使用率，从曲线的趋势上就可以看出来，它存在明显的内存泄露，但是又泄露得非常非常慢。这个系统要求 24x365 运行。

做过运维的人会知道，如此长时间的运行，运维时间长了之后，只会对这样的系统做常规的健康检查，因为前期天天关注它，又不出问题，眼睛都看瞎了，也不值得，于是后期就放松了警惕，慢慢懈怠。

而这个系统在生产上出现事故是在正常运行快到一年的时候，这个系统的业务量不大，十几个 TPS 的业务量级。这是一个外贸的系统，业务量虽然不大，但每个业务涉及的金额很大。其实出故障时间倒也不长，才几个小时，但是也干掉了几个总监级职位及相关人员。

如何对内存进行分析，仍然是性能测试分析的从业人员应该知道的知识点。

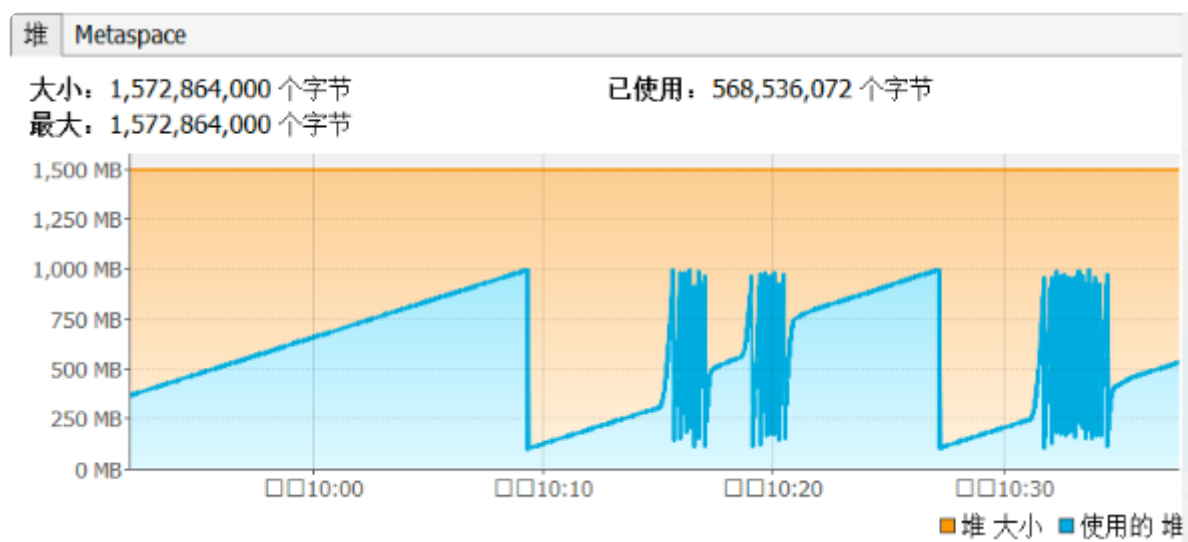
我们从技术的角度来说一下内存问题的排查思路。

这下我换个实例程序。我们照样用 jvisualvm，记住哦，这时候 Arthas 之类的工具就没得玩了，因为 Arthas 只会操作栈，有很多在 Java 方面做性能分析的工具都是只分析栈的。在 Java 中动态操作对象，其实资源消耗非常高。打个比方，你可以想像一下，在一个课间休息的校园，像寻找一个特定的孩子有多难。

其实操作一个对象还有迹可循，但是内存中那么多对象，要想全都控制，那几乎是不理智的。所以，我们首先要看内存整体的健康状态。

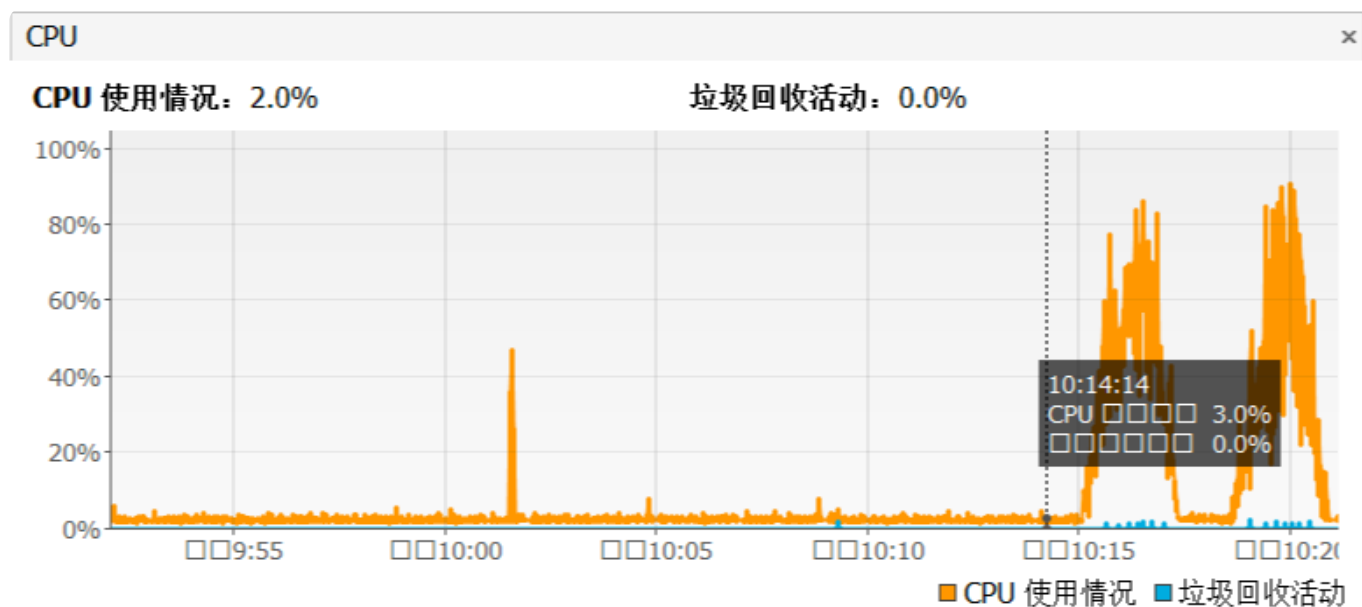
## 内存趋势判断

### 场景一：典型的正常内存的场景



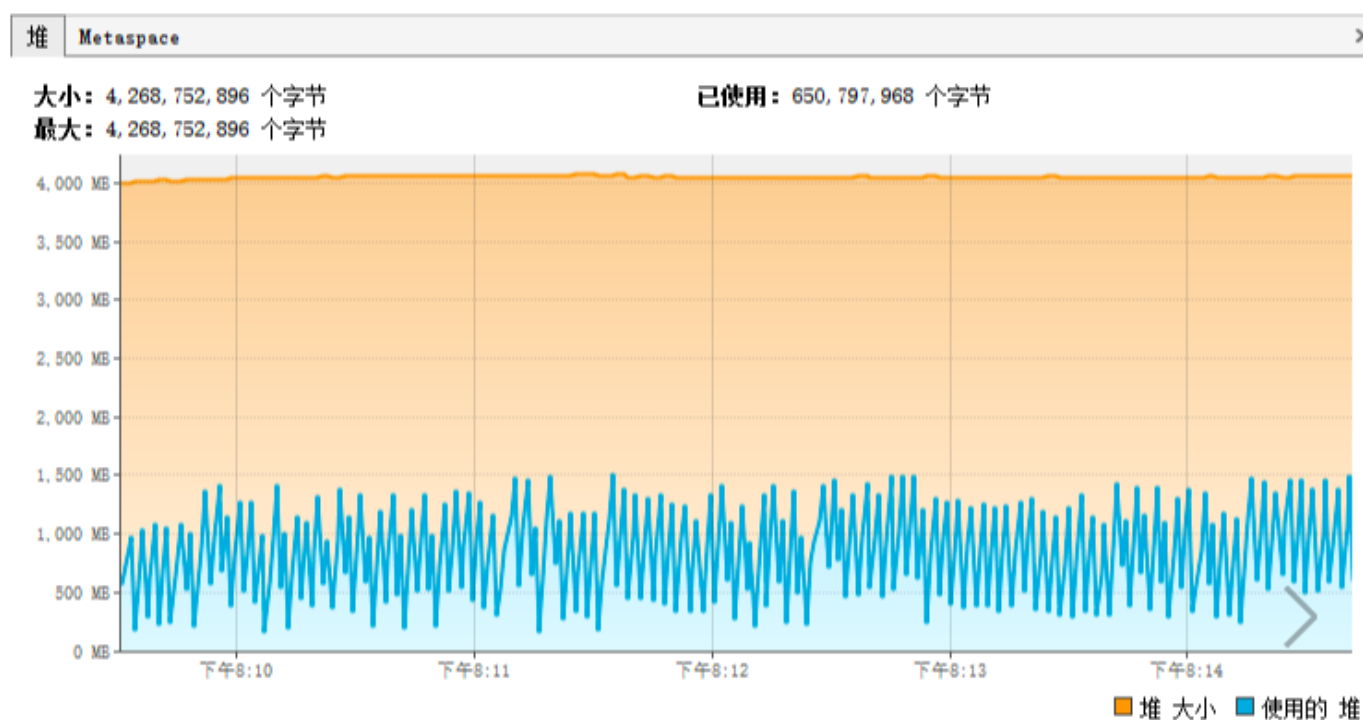
看了这个图后，要有如下几个反应：

1. 内存使用很正常，回收健康。
2. 内存从目前的压力级别上来看，够用，无需再增加。
3. 无内存泄露的情况，因为回收之后基本回到了同一水位上。
4. 基本也能看得出来 GC 够快。为什么说基本呢？因为最好还是看一下这张图。



从这张图可以看到，当应用在压力场景之后，GC 并没有消耗过多的 CPU。

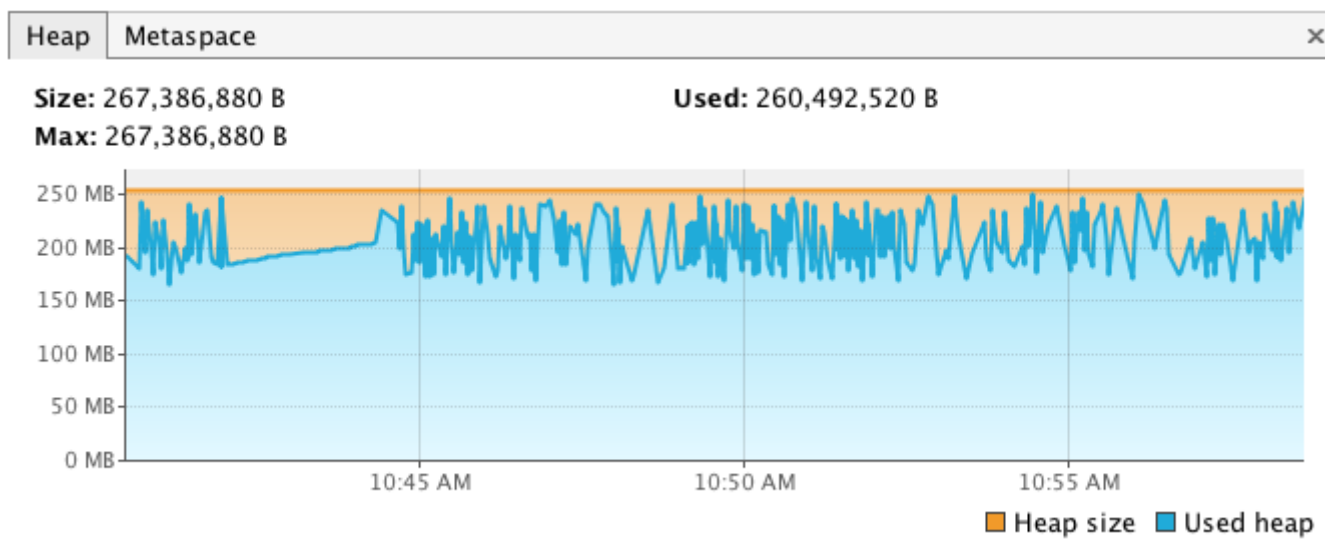
## 场景二：典型的内存分配过多的场景



从这张图我们可以看出来：

1. 内存使用很正常，回收健康。
2. 从目前的压力级别上来看，内存不仅够用，而且过多。
3. 无内存泄露的情况。

### 场景三：典型的内存不够用的场景



从这张图我们可以看出来：

1. 内存使用很正常，回收健康。
2. 从目前的压力级别上来看，**内存不够用，需再增加。**
3. CPU 可看可不看，因为现在看似乎没多大意义，先加了内存再说。
4. 无内存泄露的情况，因为回收之后基本回到了同一水位上。

### 场景四：典型的内存泄露到爆的场景

为了显示我能力的多样性，我换个工具的监控结果。



0.00	0.00	84.92	56.95	94.68	1569	12.541	70	178.193	190.734
00.00	0.00	0.00	93.66	94.68	1573	12.939	71	178.193	191.132
99.98	0.00	0.00	93.99	95.10	1575	13.624	72	180.447	194.071
0.00	0.00	100.00	100.00	95.34	1575	13.624	73	183.737	197.361
0.00	0.00	100.00	100.00	95.38	1575	13.624	74	186.081	199.705
0.00	0.00	100.00	100.00	95.80	1575	13.624	75	188.589	202.213
0.00	0.00	100.00	100.00	96.60	1575	13.624	76	191.062	204.686
0.00	0.00	100.00	100.00	96.99	1575	13.624	77	194.086	207.710
0.00	0.00	100.00	100.00	97.03	1575	13.624	78	196.668	210.292
0.00	0.00	100.00	100.00	97.84	1575	13.624	80	202.044	215.668
0.00	0.00	100.00	100.00	98.33	1575	13.624	81	204.932	218.557
0.00	0.00	100.00	100.00	98.37	1575	13.624	82	207.193	220.817
0.00	0.00	100.00	100.00	98.41	1575	13.624	83	209.685	223.309
0.00	0.00	100.00	100.00	98.77	1575	13.624	84	212.111	225.736
0.00	0.00	100.00	100.00	98.77	1575	13.624	85	214.691	228.315
0.00	0.00	100.00	100.00	99.35	1575	13.624	87	219.648	233.272
0.00	0.00	100.00	100.00	99.44	1575	13.624	88	222.091	235.715
0.00	0.00	100.00	100.00	99.35	1575	13.624	89	224.433	238.057
0.00	0.00	99.93	100.00	99.49	1575	13.624	91	229.049	242.673
0.00	0.00	100.00	100.00	99.53	1575	13.624	92	231.562	245.186
0.00	0.00	100.00	100.00	99.58	1575	13.624	93	234.172	247.796
0.00	0.00	100.00	100.00	99.62	1575	13.624	94	236.937	250.562
0.00	0.00	100.00	100.00	99.67	1575	13.624	95	239.747	253.371
0.00	0.00	100.00	100.00	99.67	1575	13.624	96	242.619	256.244
0.00	0.00	100.00	100.00	99.71	1575	13.624	97	245.798	259.422
0.00	0.00	100.00	100.00	99.71	1575	13.624	98	248.363	261.987
0.00	0.00	100.00	100.00	99.71	1575	13.624	99	250.964	264.588
0.00	0.00	100.00	100.00	99.76	1575	13.624	100	253.483	267.107
0.00	0.00	100.00	100.00	99.76	1575	13.624	102	258.912	272.536
0.00	0.00	100.00	100.00	99.76	1575	13.624	103	261.528	275.152
0.00	0.00	100.00	100.00	99.76	1575	13.624	104	264.060	277.684
0.00	0.00	100.00	100.00	99.76	1575	13.624	105	266.665	280.289
0.00	0.00	100.00	100.00	99.76	1575	13.624	106	269.093	282.718
0.00	0.00	100.00	100.00	99.80	1575	13.624	108	274.027	287.652
0.00	0.00	100.00	100.00	99.80	1575	13.624	109	276.561	290.185
0.00	0.00	100.00	100.00	99.80	1575	13.624	110	279.182	292.806
0.00	0.00	100.00	100.00	99.80	1575	13.624	111	281.555	295.180
0.00	0.00	100.00	100.00	99.80	1575	13.624	112	283.933	297.557
0.00	0.00	100.00	100.00	99.80	1575	13.624	113	286.633	300.257
0.00	0.00	100.00	100.00	99.80	1575	13.624	115	291.874	305.498
0.00	0.00	100.00	100.00	99.80	1575	13.624	116	294.165	307.789
0.00	0.00	100.00	100.00	99.80	1575	13.624	117	296.617	310.241

看到上面这张图，你可能觉得人生面对着挑战：“啥玩意？”

实际上，这张图说明以下四点：

1. 年轻代（第三列）、年老代（第四列）全满了，持久代在不断增加，并且也没有释放过。
2. 两个保留区（第一列、第二列）都是空的。
3. Yonug GC（第六列）已经不做了。
4. Full GC（第八列）一直都在尝试做回收的动作，但是一直也没成功，因为年轻代、年老代都没回收下来，持久代也在不停涨。

如果出现了 1 和 2 的话，不用看什么具体对象内存的消耗，只要像网上那些只玩 JVM 参数的人一样，调调参数就行了。

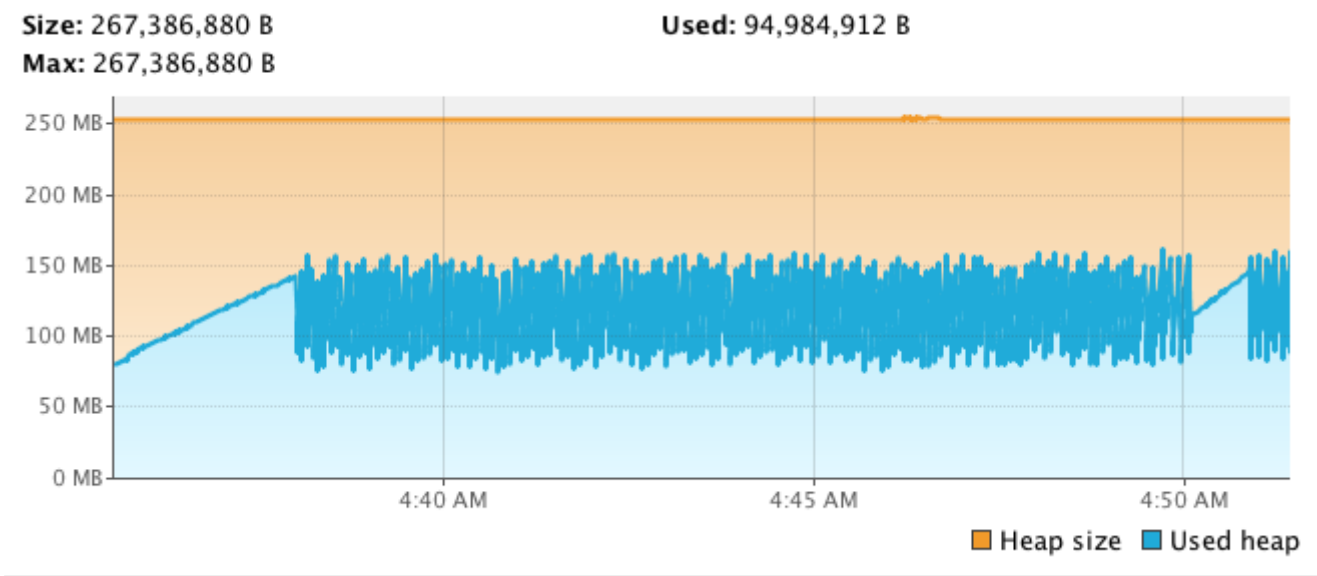
但是如果出现 3 和 4，对于 3 还要再判断一下，之前的内存是不是设置得太小了？如果是，就调大，看能不能到场景一的状态。如果不是，那就得像场景四一样，查一下内存到底

消耗在哪个对象上了。

## 查找增加的内存

### 逻辑一

下面我们来说说如何判断性能测试过程中内存的变化。



我们在内存中经常看到的对象是这样的。

Sampler Settings

Sample: CPU Memory Stop

Status: memory sampling in progress

Heap histogram Per thread allocations

Heap Deltas Snapshot Perform GC Heap Dump

Classes: 7,747    Instances: 1,893,850    Bytes: 102,423,112

Class Name	Bytes [%]	Bytes	Instances
char[]	29.4%	30,189,992	188,023 (9.9%)
java.lang.Object[]	7.6%	7,784,672	232,957 (12.3%)
byte[]	6.3%	6,478,136	28,347 (1.4%)
java.lang.reflect.Method	4.4%	4,598,704	52,258 (2.7%)
int[]	4.2%	4,336,104	19,249 (1.0%)
java.lang.String	3.8%	3,931,872	163,828 (8.6%)
java.util.HashMap	3.6%	3,744,000	78,000 (4.1%)
java.util.TreeMap\$Entry	3.0%	3,160,240	79,006 (4.1%)
java.io.ObjectStreamClass\$WeakClassKey	2.6%	2,667,360	83,355 (4.4%)
java.util.concurrent.ConcurrentHashMap\$Node	2.5%	2,620,896	81,903 (4.3%)
java.lang.Class	1.9%	2,034,536	18,486 (0.9%)
java.util.LinkedHashMap\$Entry	1.7%	1,753,240	43,831 (2.3%)
java.util.HashMap\$Node[]	1.5%	1,570,208	20,521 (1.0%)
org.springframework.util.ConcurrentReferenceHashMap\$SoftEntryReference	1.5%	1,543,200	32,150 (1.6%)
java.util.ArrayList	1.0%	1,117,200	46,550 (2.4%)
java.util.LinkedHashMap	1.0%	1,099,784	19,639 (1.0%)
java.util.concurrent.ConcurrentHashMap\$Node[]	1.0%	1,074,448	633 (0.0%)
java.lang.Class[]	1.0%	1,056,592	47,347 (2.5%)
java.lang.reflect.Field	1.0%	1,026,504	14,257 (0.7%)

Class Name Filter (Contains)

如果你用 jmap 的话，会看到如下信息。

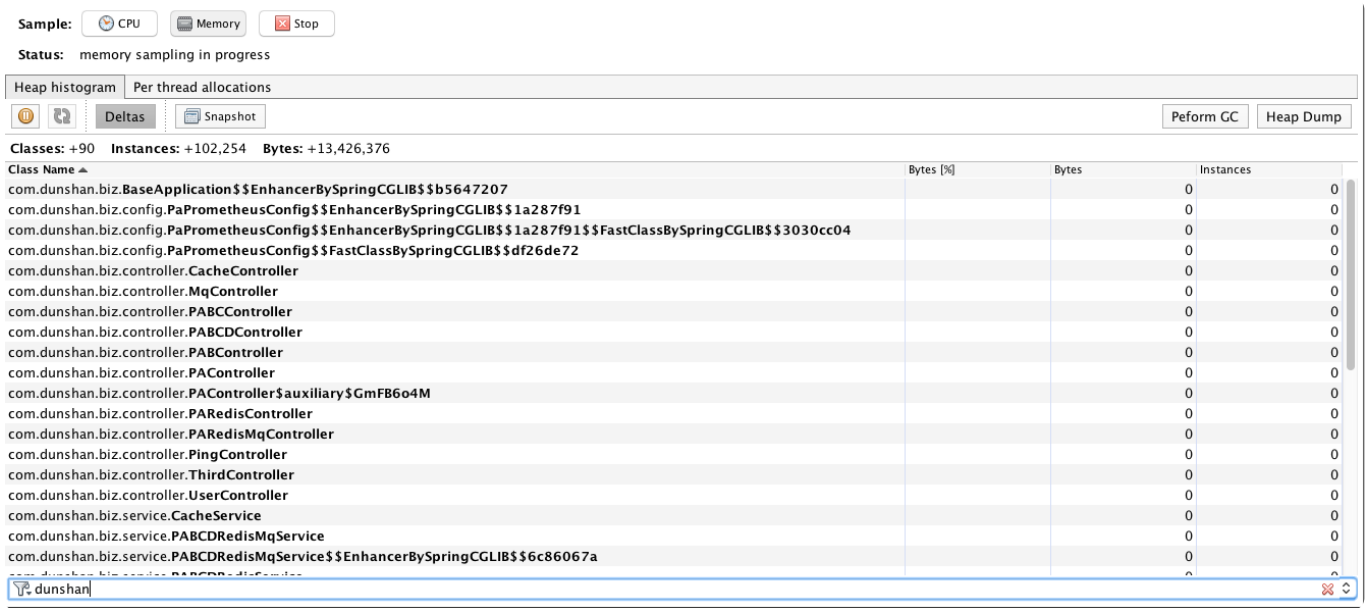
```
[root@7dgroup1 target]# jmap -histo 17953
```

num	#instances	#bytes	class name
1:	181607	30877824	[C
2:	206897	6980504	[Ljava.lang.Object;
3:	25840	5964368	[B
4:	18126	5383168	[I
5:	51991	4575208	java.lang.reflect.Method
6:	160536	3852864	java.lang.String
7:	77107	3701136	java.util.HashMap
8:	70051	2802040	java.util.TreeMap\$Entry
9:	81806	2617792	java.util.concurrent.ConcurrentHashMap\$Node
10:	72692	2326144	java.io.ObjectStreamClass\$WeakClassKey
11:	18493	2035264	java.lang.Class
12:	43875	1755000	java.util.LinkedHashMap\$Entry
13:	32150	1543200	org.springframework.util.ConcurrentReferenceHashMap\$SoftEntryReference
14:	20087	1472256	[Ljava.util.HashMap\$Node;
15:	19648	1100288	java.util.LinkedHashMap
16:	45192	1084608	java.util.ArrayList
17:	608	1072384	[Ljava.util.concurrent.ConcurrentHashMap\$Node;
18:	46745	1041248	[Ljava.lang.Class;
19:	14060	1012320	java.lang.reflect.Field
20:	25417	813344	java.util.HashMap\$Node
21:	9255	740400	org.apache.skywalking.apm.dependencies.net.bytebuddy.pool.TypePool\$DefaultLazyTypeDescription\$MethodToken
22:	42068	673088	java.lang.Object
23:	25526	612624	java.lang.Long
24:	17692	566144	java.util.TreeMap\$KeyIterator
25:	15648	479552	[Ljava.lang.String;
26:	18367	440808	org.springframework.util.ConcurrentReferenceHashMap\$Entry
27:	5495	439600	java.lang.reflect.Constructor
28:	3568	428160	org.springframework.boot.loader.jar.JarEntry
29:	8484	407232	java.util.TreeMap
30:	3142	326768	java.io.ObjectStreamClass
31:	5639	323152	[Ljava.lang.reflect.Method;
32:	6253	300144	org.springframework.core.ResolvableType
33:	2937	281952	java.lang.management.ThreadInfo
34:	3878	279216	org.springframework.core.annotation.AnnotationAttributes
35:	11143	267432	org.apache.skywalking.apm.dependencies.net.bytebuddy.pool.TypePool\$DefaultLazyTypeDescription\$MethodToken\$ParameterToken
36:	1826	265816	[Lorg.springframework.util.ConcurrentReferenceHashMap\$Reference;
37:	6250	250000	java.lang.ref.SoftReference
38:	9774	234576	java.io.SerialCallbackContext
39:	4064	227584	java.lang.Class\$ReflectionData

你可能会问，这么多的内容，我到底要看什么呢？这也是性能测试人员经常遇到的问题，明明数据都在眼前，就是不知道从哪下嘴。

我建议你不要看这些底层的对象类型，因为实在是有点多哇。在这里我们最好是看自己代码调用的对象的内存占用大小增量。

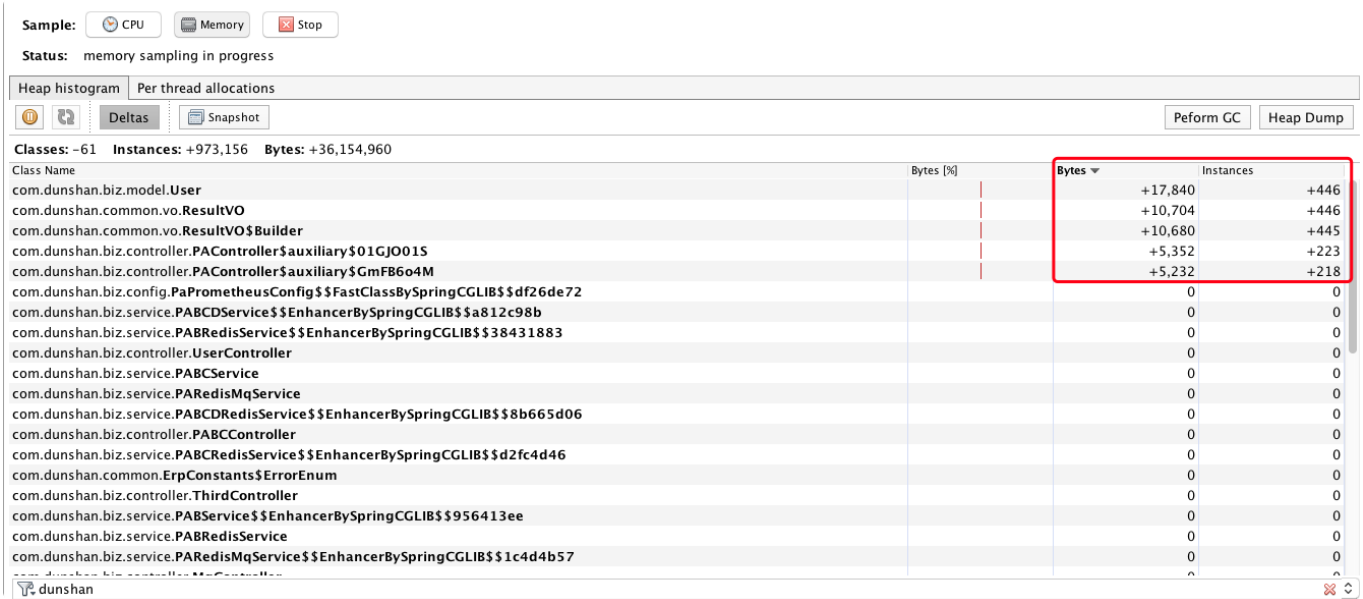
1. 先过滤下我们自己的包。
2. 点击一下 Deltas，就能看到下面的截图。





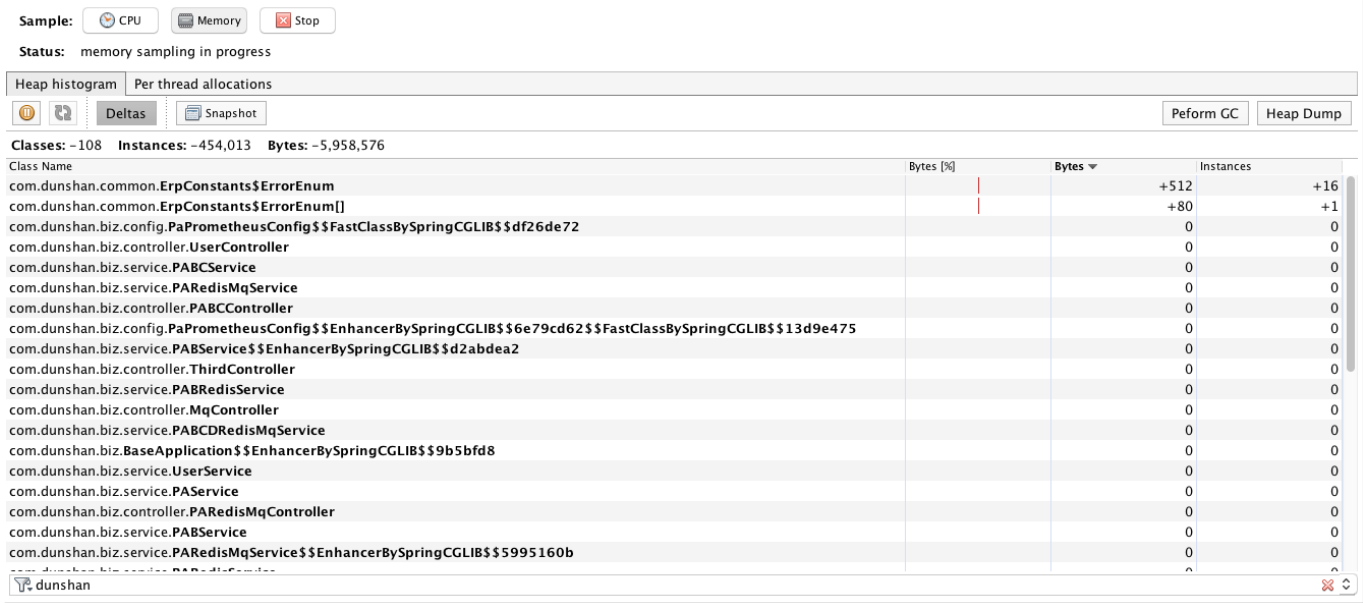
在刚开始点击 Deltas 之后，会看到全是零的对象。

下面我们来做下压力，观察一下。



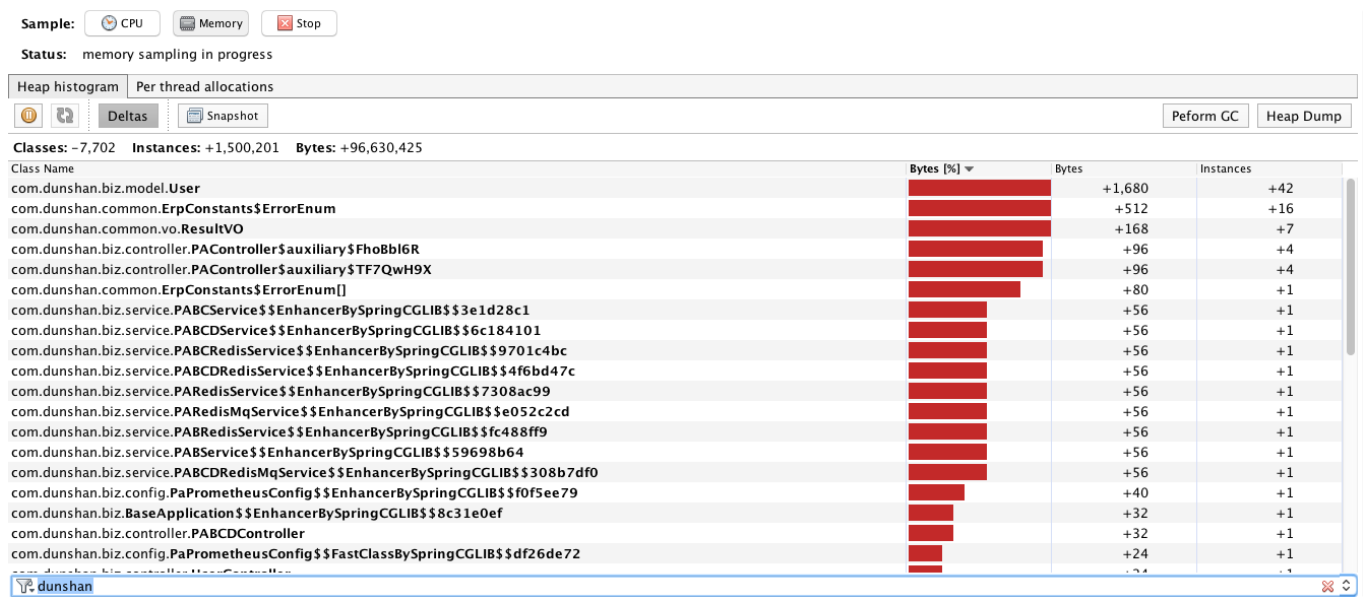
你看现在对象的实体都在往上增加对吧？但是当压力停止之后，该回收的都回收了，而有些必须长久使用的对象，在架构设计上也应该清晰地判断增量，不然就有可能导致内存不够。出现这种情况一般是架构师的失职。像这类东西应该写到公司的代码规范里。

当内存正常回收之后，再观察 Deltas，应该会看到大部分对象都回收了的状态。如下所示：



临时的对象也都清理了。这就是正常的结果。

如果停止压力之后，又做了正常的 FullGC 回收了之后，还是像下面这样。

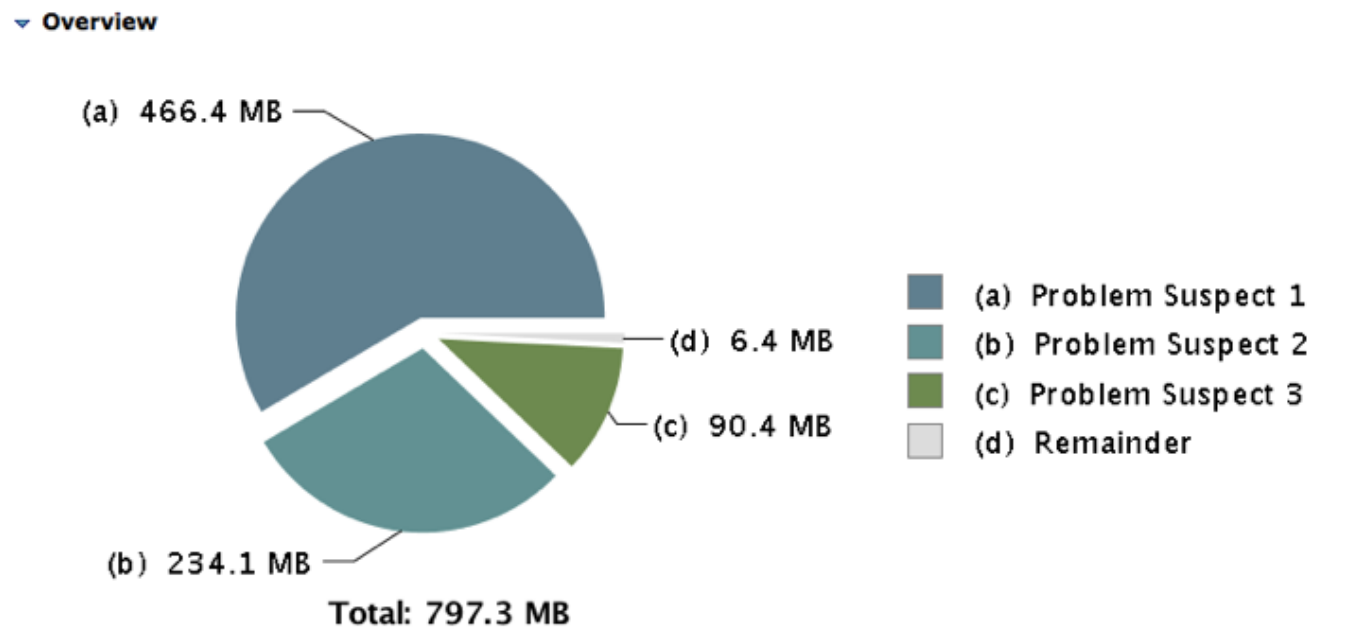


那就显然有问题了。回收不了的对象就是典型的内存泄露了。

















逻辑二

我们看下面这个图。这是 jmap 做出来的 heapdump，然后用 MAT 打开的。

1. 第一个可疑的内存泄露点占了 466.4MB 的内存。



2. 找到内存消耗点的多的内容。如下所示。

Class Name	Shallow Heap	Retained Heap	Percentage
 org.quartz.simpl.SimpleThreadPool\$WorkerThread @ 0xcfc223638 TaskFrameWork_Worker-1	128	489,046,544	58.49%
 java.util.ArrayList @ 0xdb3880c0	24	489,008,720	58.49%
 java.lang.Object[198578] @ 0xea0db6e8	794,328	489,008,696	58.49%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe4031498	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe983b028	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xd5f3d2a8	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe0b50540	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xd8a46cb0	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe61a0900	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe4358c08	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe3a011d0	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xdce2f378	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xd0cb8328	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe8399fc8	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe4d0daa0	64	2,720	0.00%
 com. . . . .crm.res.phone.bo.BOResPhoneNumUsedBean @ 0xe7bb6040	64	2,720	0.00%

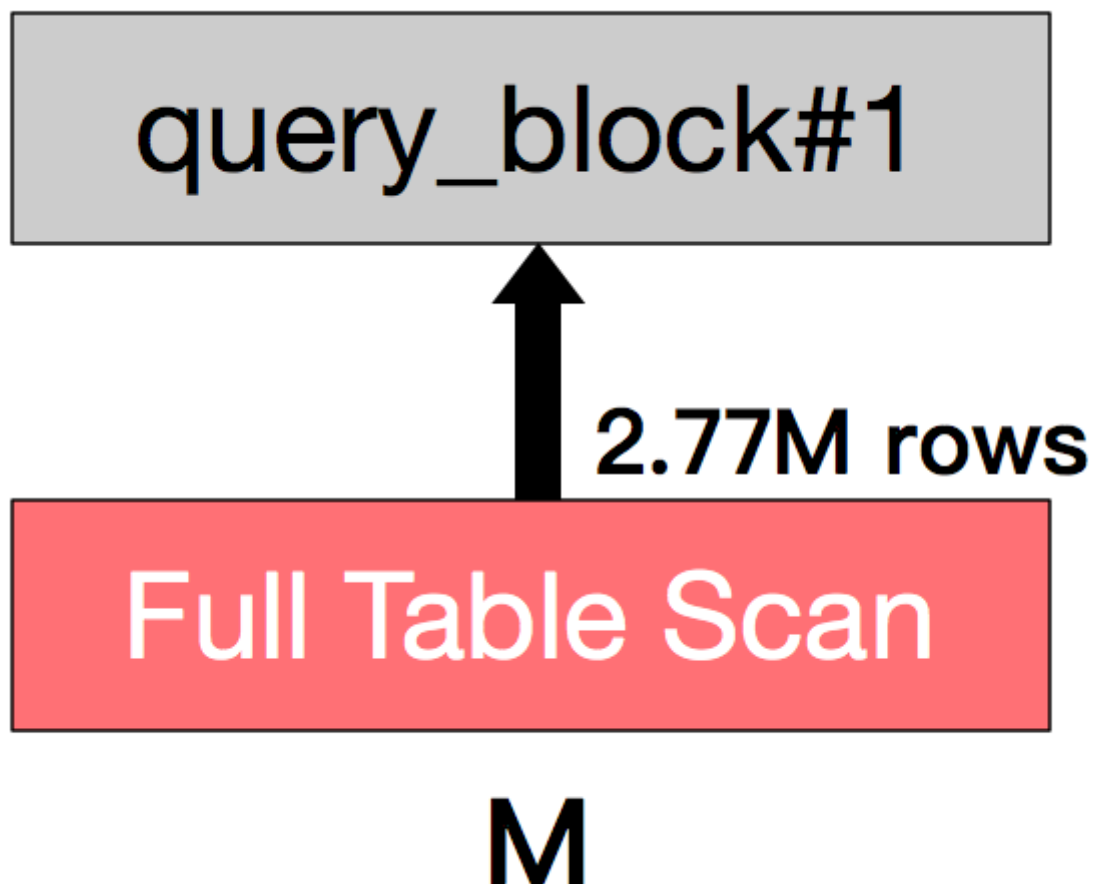
这是一个实体 bean。每个倒是不大，但是架不住有 79 万个。

### 3. 看它对应的栈。

```
TaskFrameWork_Worker-1
  at java.lang.StringCoding$StringDecoder.decode([BII)[C (StringCoding.java:133)
  at java.lang.StringCoding.decode(Ljava/lang/String;[BII)[C (StringCoding.java:173)
  at java.lang.String.<init>([BII)Ljava/lang/String; V (String.java:443)
  at com.mysql.jdbc.ResultSetRow.getString(Ljava/lang/String;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/String; (ResultSetRow.java:1006)
  at com.mysql.jdbc.ResultSetImpl.getString(Ljava/lang/String;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/String; (ResultSetImpl.java:1006)
  at com.mysql.jdbc.ResultSetImpl.getNativeConvertToString(ILcom/mysql/jdbc/Field;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/String; (ResultSetImpl.java:3725)
  at com.mysql.jdbc.ResultSetImpl.getNativeString(I)Ljava/lang/String; (ResultSetImpl.java:4569)
  at com.mysql.jdbc.ResultSetImpl.getStringInternal(IZ)Ljava/lang/String; (ResultSetImpl.java:5708)
  at com.mysql.jdbc.ResultSetImpl.getString(I)Ljava/lang/String; (ResultSetImpl.java:5509)
  at com.mysql.jdbc.ResultSetImpl.getObject(I)Ljava/lang/Object; (ResultSetImpl.java:4894)
  at com.mysql.jdbc.ResultSetImpl.getObject(Ljava/lang/String;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/Object; (ResultSetImpl.java:5012)
  at org.apache.commons.dbcp.DelegatingResultSet.getObject(Ljava/lang/String;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/Object; (DelegatingResultSet.java:335)
  at org.apache.commons.dbcp.DelegatingResultSet.getObject(Ljava/lang/String;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/Object; (DelegatingResultSet.java:335)
  at com. . . . .appframe2.complex.datasource.LogicResultSet.getObject(Ljava/lang/String;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/Object; (LogicResultSet.java:641)
  at com. . . . .appframe2.bo.DataStoreImpl.fillData(Ljava/sql/ResultSet;Lcom. . . . .appframe2/common/Object;Lcom. . . . .appframe2/common/DataStoreImpl;[BII)Ljava/lang/Object; (DataStoreImpl.java:1006)
  at com. . . . .appframe2.bo.DataStoreImpl.createDataContainerFromResultSet(Ljava/lang/Class;Lcom. . . . .appframe2/common/Object;Lcom. . . . .appframe2/common/DataStoreImpl;[BII)Ljava/lang/Object; (DataStoreImpl.java:1006)
  at com. . . . .crm.res.phone.bo.BOResPhoneNumUsedEngine.getBeans([Ljava/lang/String;Lcom/mysql/jdbc/ConnectionImpl;[BII)Ljava/lang/Object; (BOResPhoneNumUsedEngine.java:1006)
  at com. . . . .crm.res.phone.dao.impl.ResPhoneNumUsedDAOImpl.query(Lcom. . . . .crm.res.phone/ivalues/IBOResPhoneNumUsedValue;I)Ljava/lang/Object; (ResPhoneNumUsedDAOImpl.java:1006)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Ljava/lang/reflect/Method;Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (NativeMethodAccessorImpl.java:1006)
  at sun.reflect.NativeMethodAccessorImpl.invoke(Ljava/lang/reflect/Method;Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (NativeMethodAccessorImpl.java:1006)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(Ljava/lang/reflect/Method;Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (DelegatingMethodAccessorImpl.java:1006)
  at java.lang.reflect.Method.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (Method.java:597)
  at com. . . . .appframe2.complex.service.proxy.ProxyInvocationHandler.invoke(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (ProxyInvocationHandler.java:1006)
  at com.sun.proxy.$Proxy73.query(Lcom. . . . .crm.res.phone/ivalues/IBOResPhoneNumUsedValue;I)Lcom. . . . .crm.res.phone/ivalues/IBOResPhoneNumUsedValue; (Proxy73.java:1006)
  at com. . . . .crm.res.phone.service.impl.ResPhoneNumUsedSVImpl.query(Lcom. . . . .crm.res.phone/ivalues/IBOResPhoneNumUsedValue;I)Lcom. . . . .crm.res.phone/ivalues/IBOResPhoneNumUsedValue; (ResPhoneNumUsedSVImpl.java:1006)
  at com. . . . .crm.res.phone.service.impl.ResPhoneNumUsedSVImpl.query(Lcom. . . . .crm.res.phone/ivalues/IBOResPhoneNumUsedValue;I)Lcom. . . . .crm.res.phone/ivalues/IBOResPhoneNumUsedValue; (ResPhoneNumUsedSVImpl.java:1006)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Ljava/lang/reflect/Method;Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (NativeMethodAccessorImpl.java:1006)
  at sun.reflect.NativeMethodAccessorImpl.invoke(Ljava/lang/reflect/Method;Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (NativeMethodAccessorImpl.java:1006)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(Ljava/lang/reflect/Method;Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object; (DelegatingMethodAccessorImpl.java:1006)
```

就是一个数据库操作。

### 4. 取出 SQL，查看执行计划如下。



这是曲线的 SQL 查询数据过多，导致内存不够用。这个不叫泄露，这是溢出。因为要是一个人查询，就可能没事嘛，但是多个人一起查了，才会出问题。从业务的代码实现的角度上说，这绝对是个有问题的设计逻辑。如果真是必须全表扫描的，你得规定这个功能怎么用呀。如果不用全表扫描，干嘛不做过滤呢？

其实在 Java 中查找内存消耗的手段还有很多。你喜欢怎么玩就怎么玩，只要找得到就好。我只是给两种我觉得常用又易用的方式。

## C/C++ 类应用查找方法执行时间

对 C/C++ 的应用来说，我们可以用 google-perftools 查找方法执行时间。当然，在这之时，你需要先安装配置好 google-perftools 和 libunwind。


google-perftools 是针对 C/C++ 程序的性能分析工具。使用它，可以对 CPU 时间片、内存等系统资源的分配和使用进行分析。

使用步骤如下：

1. 编译目标程序，加入对 google-perftools 库的依赖。
2. 运行目标程序，在代码中加入启动 / 终止剖析的开关。
3. 将生成的结果通过剖析工具生成相应的调用图。

你可以在代码中加入固定输出剖析数据的开关，当运行到某段代码时就会执行。当然你也可以在代码中只加入接收信号的功能，然后在运行的过程中，通过 kill 命令给正在运行的程序发送指令，从而控制开关。

我来举个例子。如果我们有一个函数 f，我想知道它的执行效率。硬编码的方式就是在调用这个函数的前后加上剖析开关。

 复制代码

```
1 ProfilerStart("test.prof");//开启性能分析
2 f();
3 ProfilerStop();//停止性能分析
4
```

在程序编译之后，会在同目录生成一个叫 a.out 的可执行文件。

```
[root@7dgroup Sample6]# ll
总用量 52
-rwxr-xr-x 1 root root 8696 5月 29 17:09 a.out
-rwxr-xr-x 1 root root 9584 5月 21 2017 test
-rwxr-xr-x 1 root root 10389 5月 29 15:41 test6
-rw-r--r-- 1 root root 324 5月 29 16:45 test6_2.c
-rw-r--r-- 1 root root 922 5月 29 15:41 test6.c
-rw-r--r-- 1 root root 101 5月 21 2017 test.c
-rw-r--r-- 1 root root 216 11月 27 2016 test_heap_checker.cpp
```

执行这个文件，就会生成 test.prof 文件。



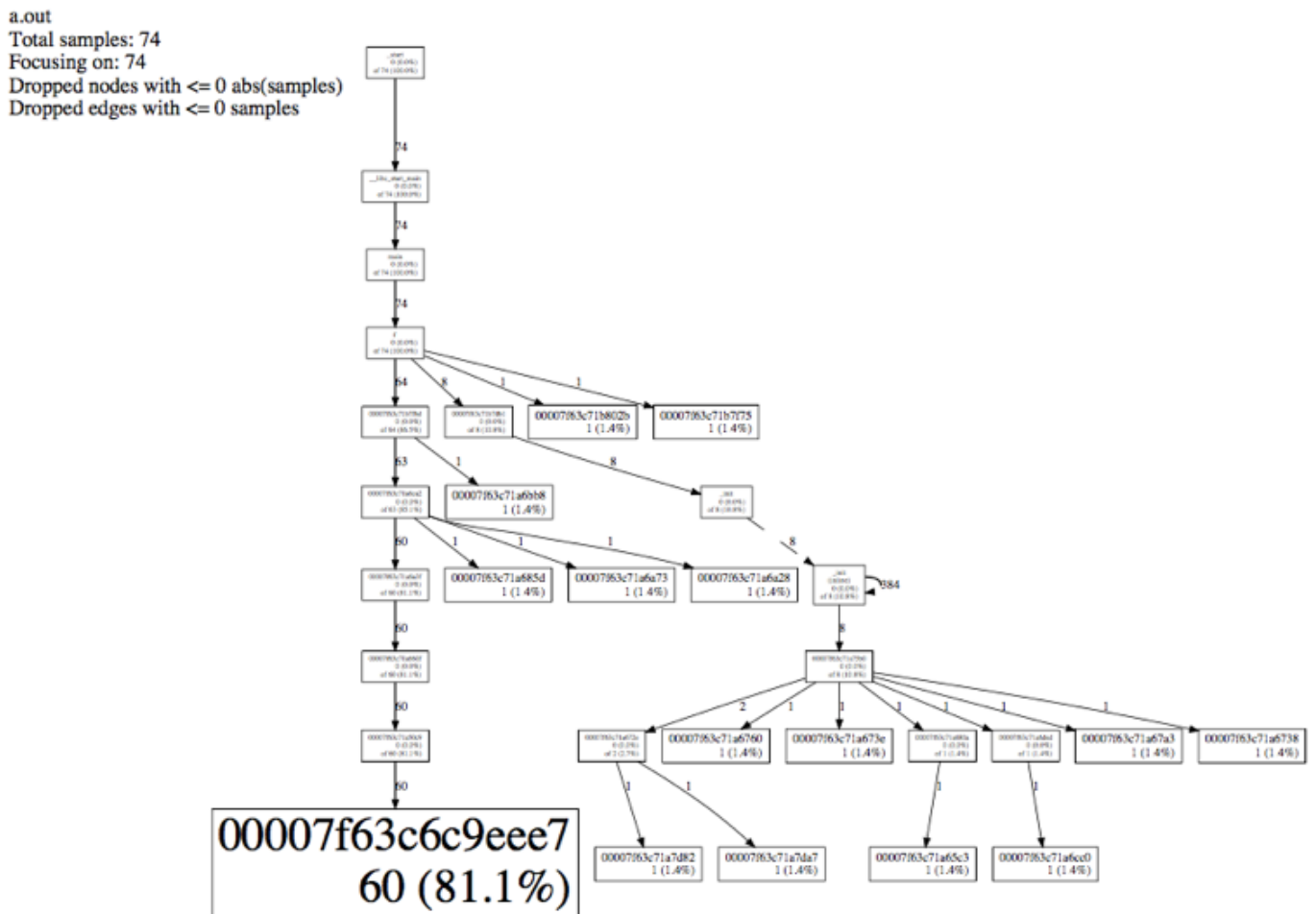
```
[root@7dgroup Sample6]# ./a.out
PROFILE: interrupts/evictions/bytes = 74/0/1320
[root@7dgroup Sample6]# ll
总用量 60
-rwxr-xr-x 1 root root 8696 5月 29 17:09 a.out
-rwxr-xr-x 1 root root 9584 5月 21 2017 test
-rwxr-xr-x 1 root root 10389 5月 29 15:41 test6
-rw-r--r-- 1 root root 324 5月 29 16:45 test6_2.c
-rw-r--r-- 1 root root 922 5月 29 15:41 test6.c
-rw-r--r-- 1 root root 101 5月 21 2017 test.c
-rw-r--r-- 1 root root 216 11月 27 2016 test_heap_checker.cpp
-rw-r--r-- 1 root root 5406 5月 29 17:11 test.prof
[root@7dgroup Sample6]#
```

然后执行命令：

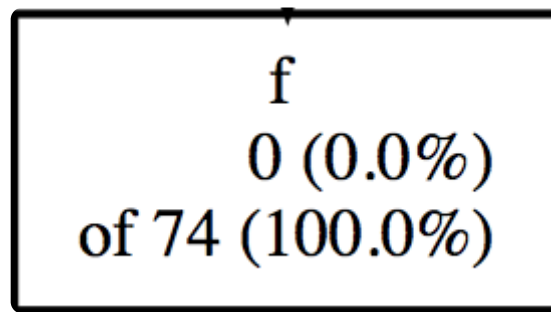
 复制代码

```
1 pprof --pdf a.out test.prof >test.pdf
```

打开这个 PDF 就可以看到如下图:



你看到上面有很多只有地址而没有函数名的调用吗？那是没有符号表。这里我们不分析那些不是我们自己的函数，我们只看自己的函数 f。



看这一段，它有三行。

第一行：函数名；

第二行：不包含内部函数调用的样本数 (百分比)；

第三行：of 包含内部函数调用的样本数 (百分比)。

是不是和 Java 中 self time/total time 有异曲同工之妙？它也可以实现从 CPU 使用率高到具体函数的定位。

你也许会说，这个有点复杂，还要在代码里加这么多，编译还要加上动态库啥的。当然了，你还可以用 perf 工具来跟踪 CPU clock，在代码编译时加上调试参数，就可以直接用 perf top -g 看调用过程由每个函数所消耗的 CPU 时钟。你还可以用 systemtap 来自己写代码进行动态跟踪。

## C/C++ 类应用查找对象内存消耗

其实 googler perftools 也可以分析内存，但是我觉得它没有 Valgrind 好使。所以在这一部分，我用 valgrind 来告诉你如何查找到 C/C++ 的内存消耗。

valgrind 能实现这些功能：

工具	功能
Memcheck	可以检测程序中出现的内存问题。比如说，未初始化变量、内存溢出、内存覆盖等等。它对malloc()/free()/new/delete的调用都会被捕获。
Callgrind	它可以收集运行时的数据，建立调用关系图，并且可以计算调用的成本。
Cachegrind	Cache分析器。
Helgrind	检查多线程程序中出现的竞争问题。
Massif	堆栈分析器，可以分析程序在运行时使用了多少内存。
Extension	可用此功能自扩展调试工具。

这里举一个内存泄露的小例子。这是一段再无聊不过的代码：

 复制代码

```

1  #include <stdlib.h>
2
3
4  void f(void)
5  {
6      int* x = malloc(10 * sizeof(int));
7      x[10] = 0;          // problem 1: heap block overrun
8  }                      // problem 2: memory leak -- x not freed
9
10
11 int main(void)
12 {
13     f();
14     return 0;

```

我们不断分配，而不释放。

编译运行之后，我们可以看到如下结果。

 复制代码

```

1  [root@7dgroup Sample10]# gcc -Wall -o test5 test5.c
2  [root@7dgroup Sample10]# valgrind --tool=memcheck --leak-check=full ./test5
3  ==318== Memcheck, a memory error detector
4  ==318== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
5  ==318== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info

```

```
6 ==318== Command: ./test5
7 ==318==
8 ==318== Invalid write of size 4
9 ==318==    at 0x40054E: f (in /root/GDB/Sample10/test5)
10 ==318==    by 0x40055E: main (in /root/GDB/Sample10/test5)
11 ==318== Address 0x51f7068 is 0 bytes after a block of size 40 alloc'd
12 ==318==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd
13 ==318==    by 0x400541: f (in /root/GDB/Sample10/test5)
14 ==318==    by 0x40055E: main (in /root/GDB/Sample10/test5)
15 ==318==
16 ==318==
17 ==318== HEAP SUMMARY:
18 ==318==    in use at exit: 40 bytes in 1 blocks
19 ==318== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
20 ==318==
21 ==318== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
22 ==318==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd
23 ==318==    by 0x400541: f (in /root/GDB/Sample10/test5)
24 ==318==    by 0x40055E: main (in /root/GDB/Sample10/test5)
25 ==318==
26 ==318== LEAK SUMMARY:
27 ==318==    definitely lost: 40 bytes in 1 blocks
28 ==318==    indirectly lost: 0 bytes in 0 blocks
29 ==318==    possibly lost: 0 bytes in 0 blocks
30 ==318==    still reachable: 0 bytes in 0 blocks
31 ==318==    suppressed: 0 bytes in 0 blocks
32 ==318==
33 ==318== For counts of detected and suppressed errors, rerun with: -v
34 ==318== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 1 from 1)
35 [root@7dgroup Sample10]#
```

主要看一下这行。

```
1 ==318==    total heap usage: 1 allocs, 0 frees, 40 bytes allocated
2
```

 复制代码

这里分配了 40 个字节的一块内存，但是 0 释放，所以就泄露了。

请你注意，在调试自己的程序时，要像 Java 一样，分析内存的泄露，在压力前和压力后做内存的比对。在压力中则不用做。

## 总结

不管是什么语言的应用，在性能分析的过程中，都是分析两个方法。

1. 执行速度够不够快。只有够快才能满足更高的 TPS。
2. 执行过程中内存用得不多。内存用得少，才可以同时支持更多的请求。

我觉得对性能测试过程中的分析来说，这两点足够你解决代码上的问题了。有人说，为什么不说 I/O 的事情呢。其实 I/O 仍然是读写量的多少，也会反应用内存中。至于磁盘本身性能跟不上，那是另一个话题。

## 思考题

最后给你留两个思考题吧。对代码的性能分析过程中，主要是哪两点呢？针对代码分析的这两点，有什么样的分析链路？

欢迎你在评论区写下自己的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

点击查看 

## 打卡学习，成为真正的性能测试高手



PC端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。



## 精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。