
# JVM问题定位典型案例分析

你假笨@PerfMa

GTLC
GLOBAL
TECH LEADERSHIP
CONFERENCE

全球技术领导力峰会

# 通往**年薪百万**的CTO的路上，
# 如何打造自己的技术**领导**力？

扫描二维码了解详情

# 李嘉鹏

## 江湖称号

你假笨

寒泉子

## 职业经历

2010.07~2017.08
【阿里巴巴】
蚂蚁中间件
阿里JVM

2017.09~至今
【PerfMa】
CEO

## 微信公众号

你假笨：
主要围绕JVM及性能优化展开

PerfMa

01

## 类加载
## 死锁

一把你看不见的锁

02

## FinalReference
## 堆积

FinalReference的回收会跨GC

03

## 堆外内存
## 泄漏

不受Xmx控制的内存

04

## YGC
## 不断拉长

类加载器其实也会影响YGC

为JVM参数而生

## XXFox

免费的JVM参数分析交流平台;
即使你不了解JVM参数，但我们也可以
让你的系统运行在合理的JVM参数上面
http://xxfox.perfma.com

# 小福利

## JCafeBabe

免费的JVM问答社区
http://www.jcafebabe.com

01

# 类加载死锁

```
→ grep "forName0" acjstack.log -B1
        at sun.nio.cs.ext.SJIS_0213.<clinit>(SJIS_0213.java:78)
        at java.lang.Class.forName0(Native Method)
--
    java.lang.Thread.State: BLOCKED (on object monitor)
        at java.lang.Class.forName0(Native Method)
--
    java.lang.Thread.State: BLOCKED (on object monitor)
        at java.lang.Class.forName0(Native Method)
--
    java.lang.Thread.State: BLOCKED (on object monitor)
        at java.lang.Class.forName0(Native Method)
--
    java.lang.Thread.State: BLOCKED (on object monitor)
        at java.lang.Class.forName0(Native Method)
--
        at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
        at java.lang.Class.forName0(Native Method)
--
    java.lang.Thread.State: BLOCKED (on object monitor)
        at java.lang.Class.forName0(Native Method)
--
        at sun.nio.cs.ext.SJIS_0213.<clinit>(SJIS_0213.java:78)
        at java.lang.Class.forName0(Native Method)
```

```
/** Called after security check for system loader access checks have been made. */
private static native Class<?> forName0(String name, boolean initialize,
                                        ClassLoader loader,
                                        Class<?> caller)
    throws ClassNotFoundException;
```

## 问题描述

- 线程Dump没有检测到死锁
- 有不少的线程Block在Class.forName0这个方法上
- 然而Class.forName0并没有加锁

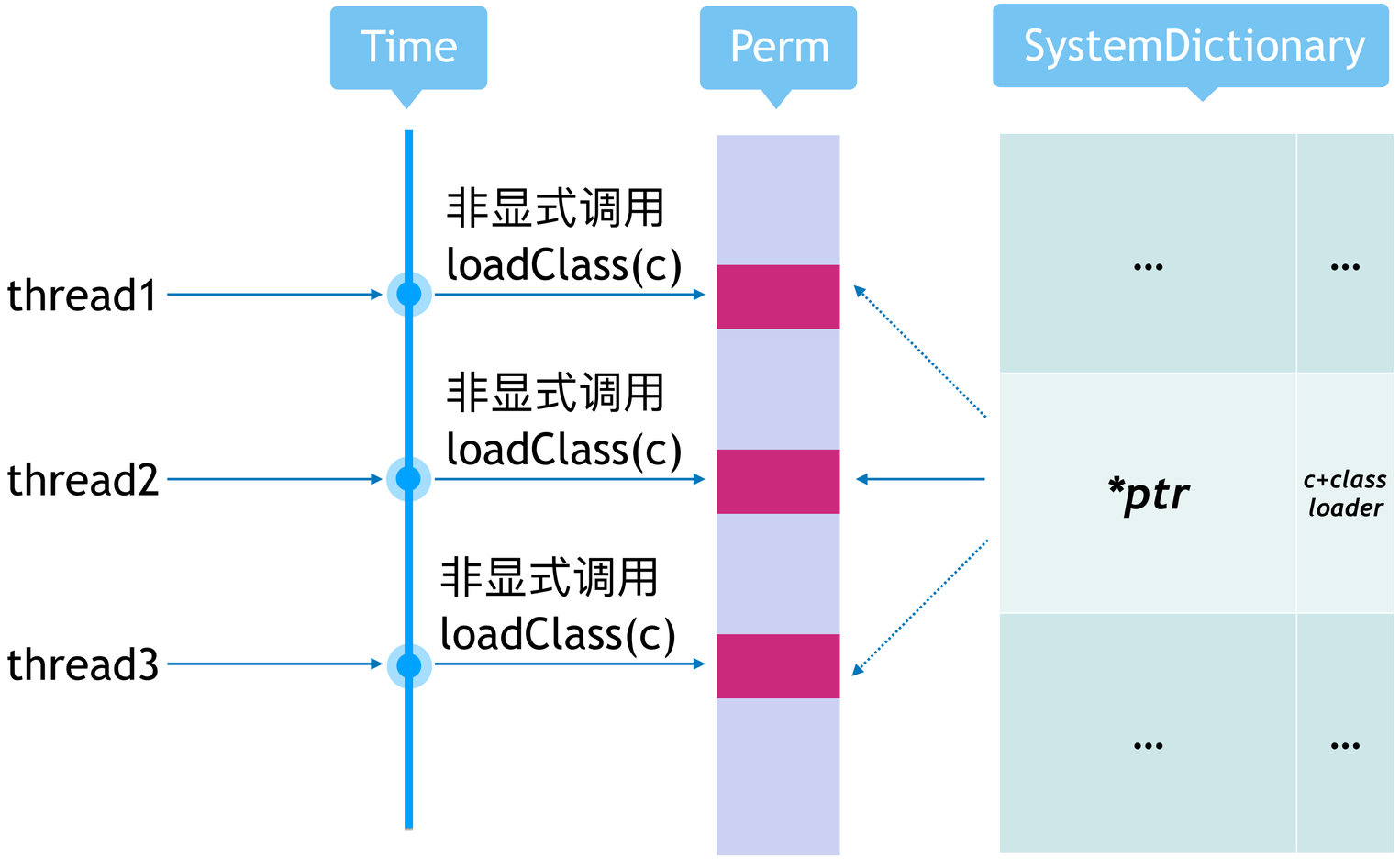
```
$ jstack -m 19417
Attaching to process ID 19417, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.102-b52
Deadlock Detection:

No deadlocks found.

----------------- 19419 -----------------
0x000000327880b43c      __pthread_cond_wait + 0xcc
0x00007fb694d2d92c      _ZN13ObjectMonitor4waitElbP6Thread + 0x96c
0x00007fb694b47e0f      JVM_MonitorWait + 0x19f
0x00007fb67e4c1b86      * java.lang.Object.wait(long) bci:0 (Interpreted frame)
```

```
1  Thread 799 (Thread 0x7faeabdfb700 (LWP 115355)):
2  #0  0x000000357480b43c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib64/libpthread.so.0
3  #1  0x00007faf3f33349b in os::PlatformEvent::park() () from /opt/install/jdk/jre/lib/amd64/server/libjvm.so
4  #2  0x00007faf3f325edc in ObjectMonitor::EnterI(Thread*) () from /opt/install/jdk/jre/lib/amd64/server/libjvm.so
5  #3  0x00007faf3f326f16 in ObjectMonitor::enter(Thread*) () from /opt/install/jdk/jre/lib/amd64/server/libjvm.so
6  #4  0x00007faf3f432bc7 in SystemDictionary::resolve_instance_class_or_null(Symbol*, Handle, Handle, Thread*) () from
   /opt/install/jdk/jre/lib/amd64/server/libjvm.so
7  #5  0x00007faf3f4336e2 in SystemDictionary::resolve_or_null(Symbol*, Handle, Handle, Thread*) () from
8  /opt/install/jdk/jre/lib/amd64/server/libjvm.so
9  #6  0x00007faf3f434d43 in SystemDictionary::resolve_or_fail(Symbol*, Handle, Handle, bool, Thread*) () from
   /opt/install/jdk/jre/lib/amd64/server/libjvm.so
10 #7  0x00007faf3f1a66df in find_class_from_class_loader(JNIEnv_*, Symbol*, unsigned char, Handle, Handle, unsigned char, Thread*) () from
   /opt/install/jdk/jre/lib/amd64/server/libjvm.so
11 #8  0x00007faf3f1b051e in JVM_FindClassFromCaller () from /opt/install/jdk/jre/lib/amd64/server/libjvm.so
12 #9  0x00007faf3e6ca740 in Java_java_lang_Class_forName0 () from /opt/install/jdk/jre/lib/amd64/libjava.so
13 #10 0x00007faf35012d98 in ?? ()
14 #11 0x00007faeabdf5c00 in ?? ()
15 #12 0x00007faf3500f3eb in ?? ()
16 #13 0x00007faf35005310 in ?? ()
17 #14 0x0000000b00000000 in ?? ()
18 #15 0x00007faeabdf5aa0 in ?? ()
19 #16 0x0000000000000000 in ?? ()
```

## 类加载有锁?

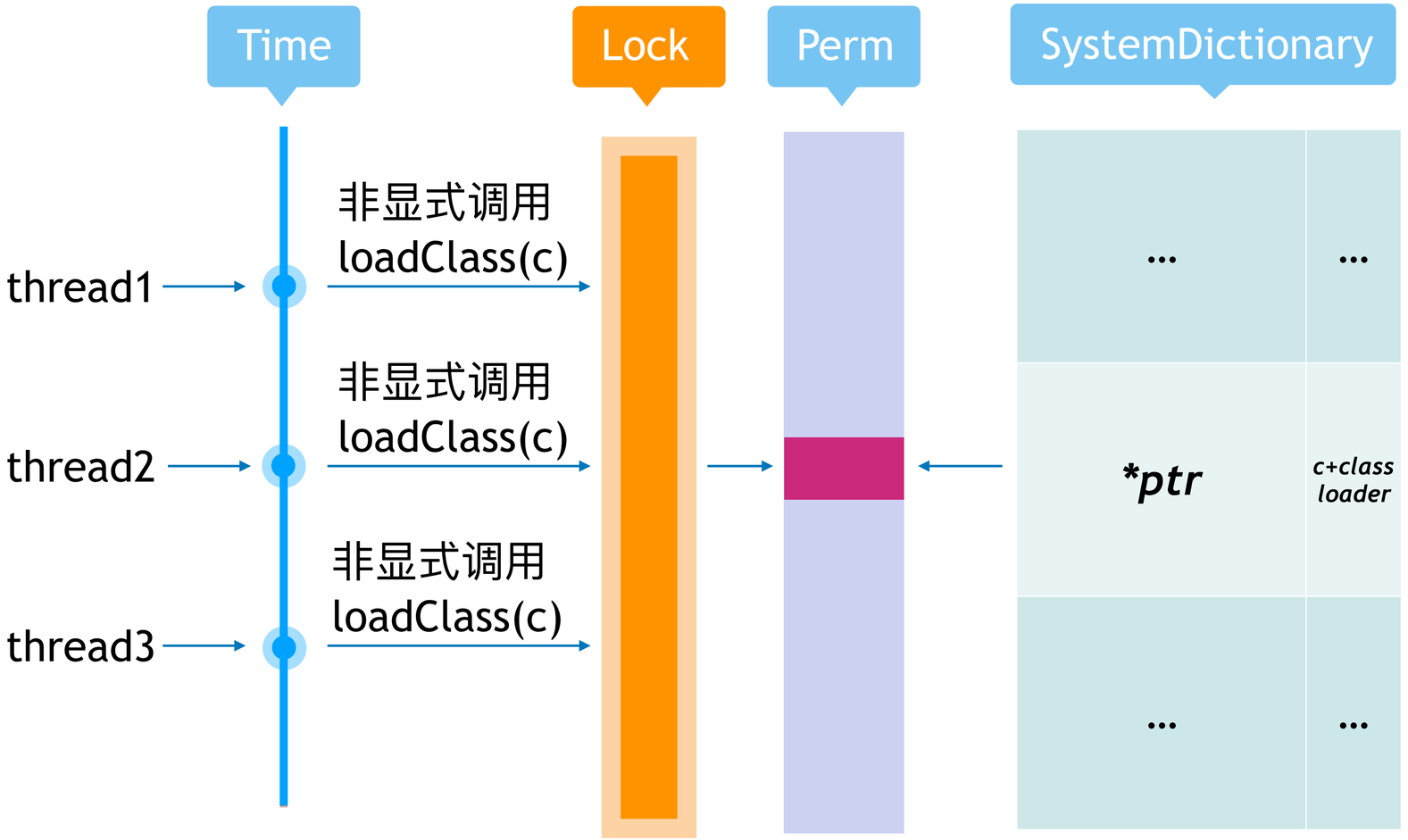
- Java级别的死锁会在jstack <pid>的输出最后看到
- 执行jstack -m <pid>

# 不带锁的场景




类加载过程为
何需要加锁

# 带锁的场景




类加载过程为
何需要加锁

```java
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
```

JDK

JVM

```
bool DoObjectLock = true;
if (is_parallelCapable(class_loader)) {
  DoObjectLock = false;
}

unsigned int p_hash = placeholders()->compute_hash(name, class_loader);
int p_index = placeholders()->hash_to_index(p_hash);

// Class is not in SystemDictionary so we have to do loading.
// Make sure we are synchronized on the class loader before we proceed
Handle lockObject = compute_loader_lock_object(class_loader, THREAD);
check_loader_lock_contention(lockObject, THREAD);
ObjectLocker ol(lockObject, THREAD, DoObjectLock);
```

## JDK7并行类加载

## 问题重现

如果类加载复写loadClass方法，

并在加载过程中加其他锁?

02

# FinalReference堆积

# FinalReference堆积

```
$ jmap -histo 23216

 num     #instances         #bytes  class name
----------------------------------------------
   1:      15063168      602526720  java.lang.ref.Finalizer
   2:      15063162      241010592  FinalTest
   3:            96        8953440  [I
   4:          1718         302400  [C
   5:           466          53376  java.lang.Class
   6:          1707          40968  java.lang.String
   7:           791          31640  java.util.TreeMap$Entry
   8:           505          28328  [Ljava.lang.Object;
   9:             8          24984  [B
  10:            17           6664  java.lang.Thread
  11:           193           6640  [Ljava.lang.String;
  12:            75           5400  java.lang.reflect.Field
  13:           256           4096  java.lang.Integer
  14:            91           3640  java.lang.ref.SoftReference
  15:           113           3616  java.util.Hashtable$Entry
  16:            11           2224  [S
```

## 问题描述

- jmap -histo可知java.lang.ref.Finalizer对象排在第一

# FinalReference堆积

```
class FinalReference<T> extends Reference<T> {
```

---

```
final class Finalizer extends FinalReference {
private Finalizer(Object finalizee) {
    super(finalizee, queue);
    add();
}

/* Invoked by VM */
static void register(Object finalizee) {
    new Finalizer(finalizee);
}
```

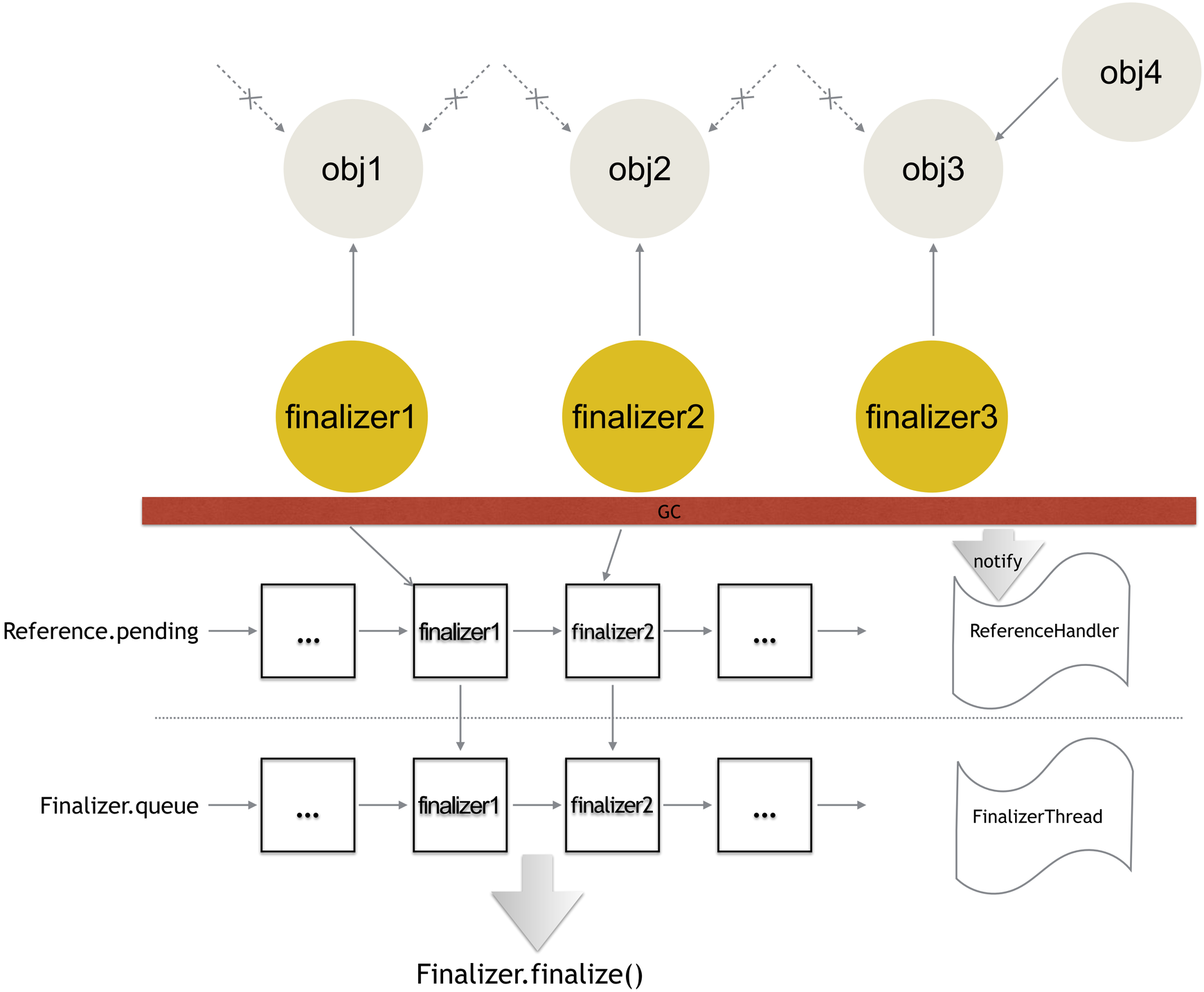
---

```
protected void finalize() throws IOException {
    close();
}
```

## FinalReference的存在形式

- FinalReference外面无法扩展
- Finalizer是FinalReference的子类
- 类加载的时候根据类里是否含有非空的void finalize()方法来决定是否其对象将来被一个FinalReference对象引用

# FinalReference堆积




## finalize方法何时被调用

- GC发生并找出Finalizer对象
- 判断Finalizer对象引用的对象是否没有别的引用了
- 将引用已死对象的Finalizer对象丢到一个Finalizer的ReferenceQueue里
- 在GC结束之后，FinalizerThread线程被唤醒并从ReferenceQueue里取出来间接调用finalize方法

# FinalReference堆积

```
private void runFinalizer(JavaLangAccess jla) {
    synchronized (this) {
        if (hasBeenFinalized()) return;
        remove();
    }
    try {
        Object finalizee = this.get();
        if (finalizee != null && !(finalizee instanceof java.lang.Enum)) {
            jla.invokeFinalize(finalizee);

            /* Clear stack slot containing this variable, to decrease
               the chances of false retention with a conservative GC */
            finalizee = null;
        }
    } catch (Throwable x) { }
    super.clear();
}
```

```
public void clear() {
    this.referent = null;
}
```

## FinalReference对象及引用的对象何时被回收

- 执行完finalize方法后会剥离Finalizer对象和被引用对象的关系
- 执行完了finalize方法的Finalizer对象及被引用的对象会在下个GC周期里被回收
- 如果finalize方法因为队列过长，不得不等待之前的对象执行完才能执行，因此可能存在跨多个GC周期

03

# 堆外内存泄露

-XX:+DisableExplicitGC

```
java.lang.OutOfMemoryError: Direct buffer memory
        at java.nio.Bits.reserveMemory(Bits.java:658)
        at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123)
        at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:311)
```

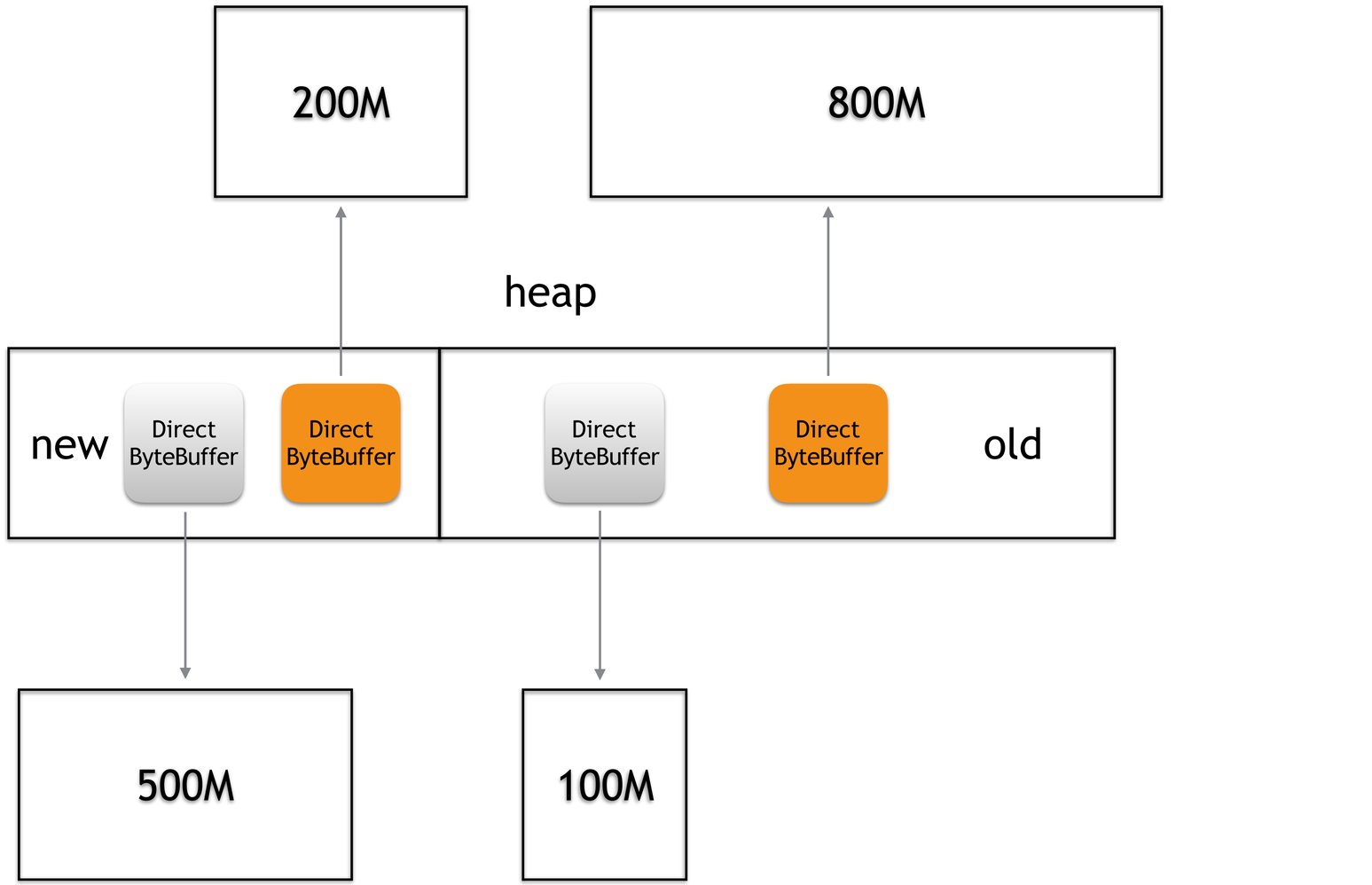
```
➜ jstat -gcutil 99691
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00   0.00   6.00   0.00  17.22  19.75      0    0.000     0    0.000    0.000
```

```
[Full GC (System.gc()) [CMS: 0K->287K(707840K), 0.0159560 secs] 10905K->287K(1014528K), [Metaspace: 2596K->2596K(1056768K)], 0.0165024 secs]
[Full GC (System.gc()) [CMS: 287K->285K(707840K), 0.0133971 secs] 293K->285K(1014528K), [Metaspace: 2597K->2597K(1056768K)], 0.0135743 secs]
[Full GC (System.gc()) [CMS: 285K->285K(707840K), 0.0028655 secs] 5745K->285K(1014528K), [Metaspace: 2597K->2597K(1056768K)], 0.0030372 secs]
[Full GC (System.gc()) [CMS: 285K->285K(707840K), 0.0043807 secs] 291K->285K(1014528K), [Metaspace: 2597K->2597K(1056768K)], 0.0045552 secs]
[Full GC (System.gc()) [CMS: 285K->285K(707840K), 0.0041310 secs] 5744K->285K(1014528K), [Metaspace: 2603K->2603K(1056768K)], 0.0043136 secs]
```

-XX:-DisableExplicitGC

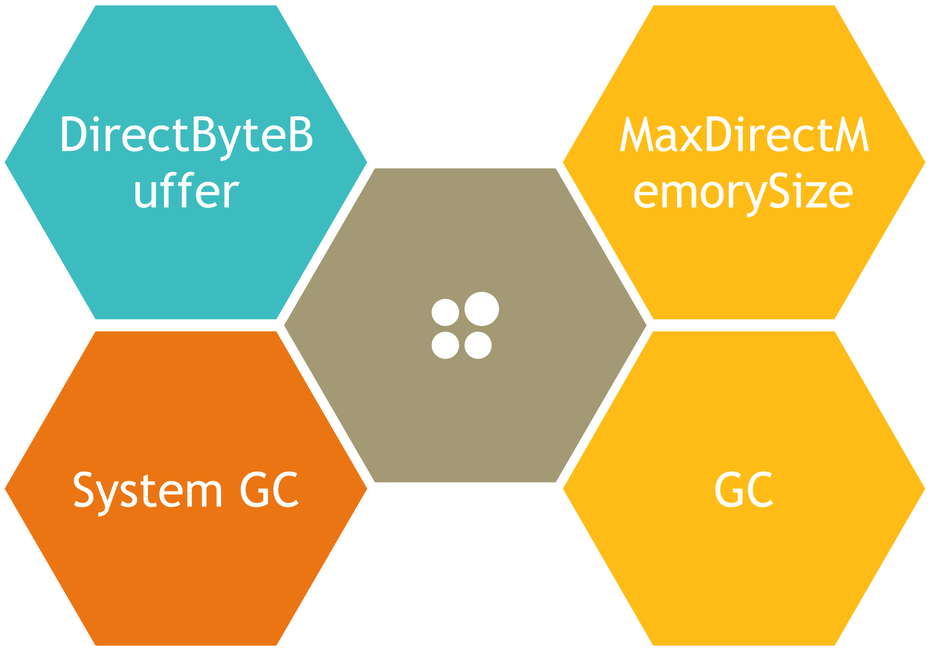
## 问题描述

- heap使用率很低，但是出现了OOM或者FullGC


200M
800M
heap
new
Direct ByteBuffer
Direct ByteBuffer
Direct ByteBuffer
Direct ByteBuffer
old
500M
100M
Dead
Live

## 堆外内存

## 堆外内存关键点

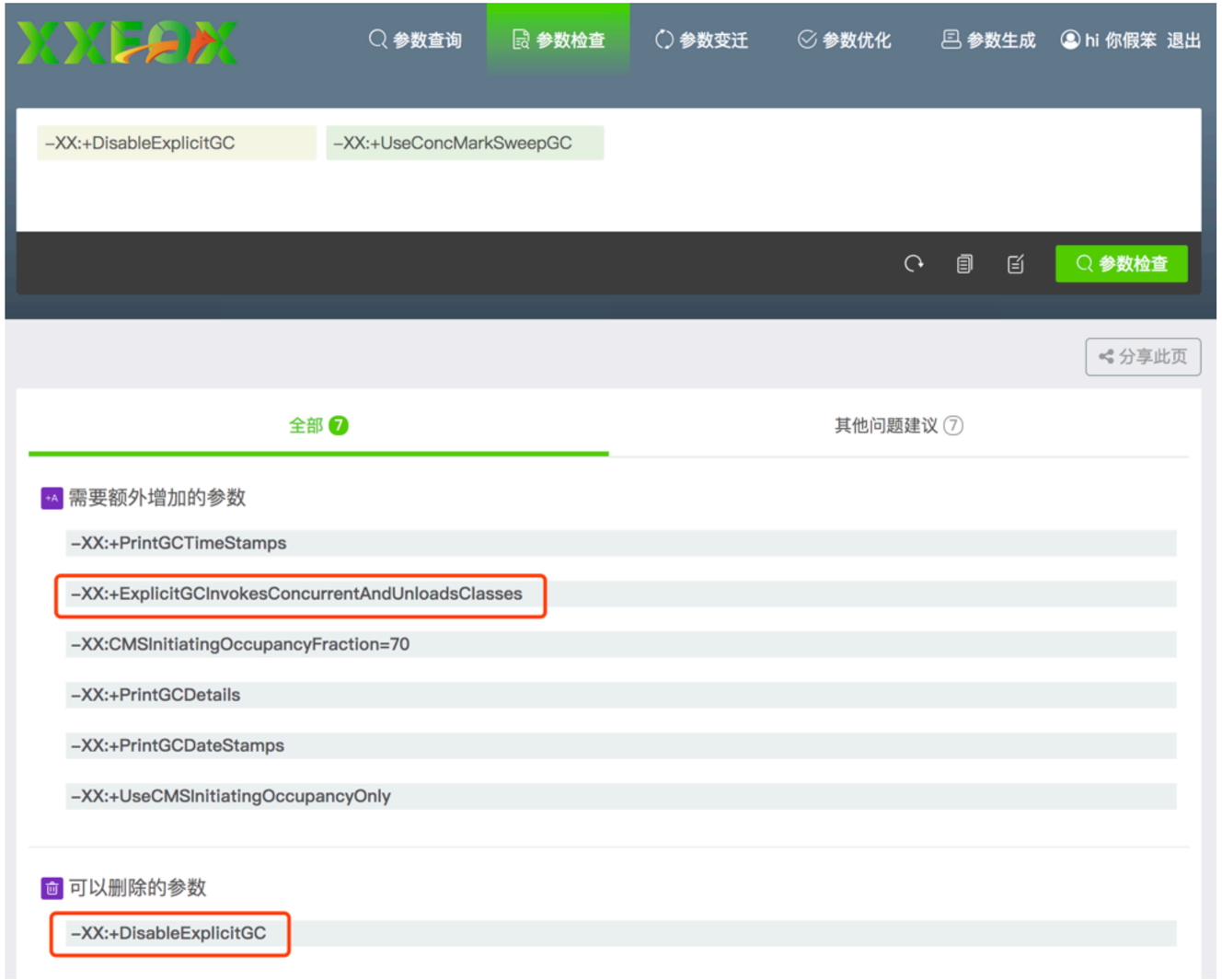



```
Total: 1670.0 MB
  1616.3  96.8%  96.8%   1616.3  96.8% zcalloc
    40.3   2.4%  99.2%     40.3   2.4% os::malloc
     9.4   0.6%  99.8%      9.4   0.6% init
     1.6   0.1%  99.9%      1.7   0.1% readCEN
     1.3   0.1%  99.9%      1.3   0.1% ObjectSynchronizer::omAlloc
     0.5   0.0% 100.0%   1591.0  95.3% Java_java_util_zip_Deflater_init
     0.1   0.0% 100.0%      0.1   0.0% _dl_allocate_tls
     0.1   0.0% 100.0%      0.2   0.0% addMetaName
     0.1   0.0% 100.0%      0.2   0.0% allocZip
     0.1   0.0% 100.0%      0.1   0.0% instanceKlass::add_dependent_nmethod
     0.1   0.0% 100.0%      0.1   0.0% newEntry
     0.0   0.0% 100.0%      0.0   0.0% strdup
     0.0   0.0% 100.0%     25.8   1.5% Java_java_util_zip_Inflater_init
```

## 如何定位

- 如果是Java滥用所致，可以使用btrace跟踪DirectByteBuffer的构造函数来定位
- 非Java层面的问题，可以使用google perftools来分析具体哪里分配，比如压缩解压缩

# 堆外内存泄漏


XXFOX
参数查询
参数检查
参数变迁
参数优化
参数生成
hi 你假笨 退出
-XX:+DisableExplicitGC
-XX:+UseConcMarkSweepGC
参数检查
分享此页
全部 7
其他问题建议 7
需要额外增加的参数
-XX:+PrintGCTimeStamps
-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses
-XX:CMSInitiatingOccupancyFraction=70
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+UseCMSInitiatingOccupancyOnly
可以删除的参数
-XX:+DisableExplicitGC

04

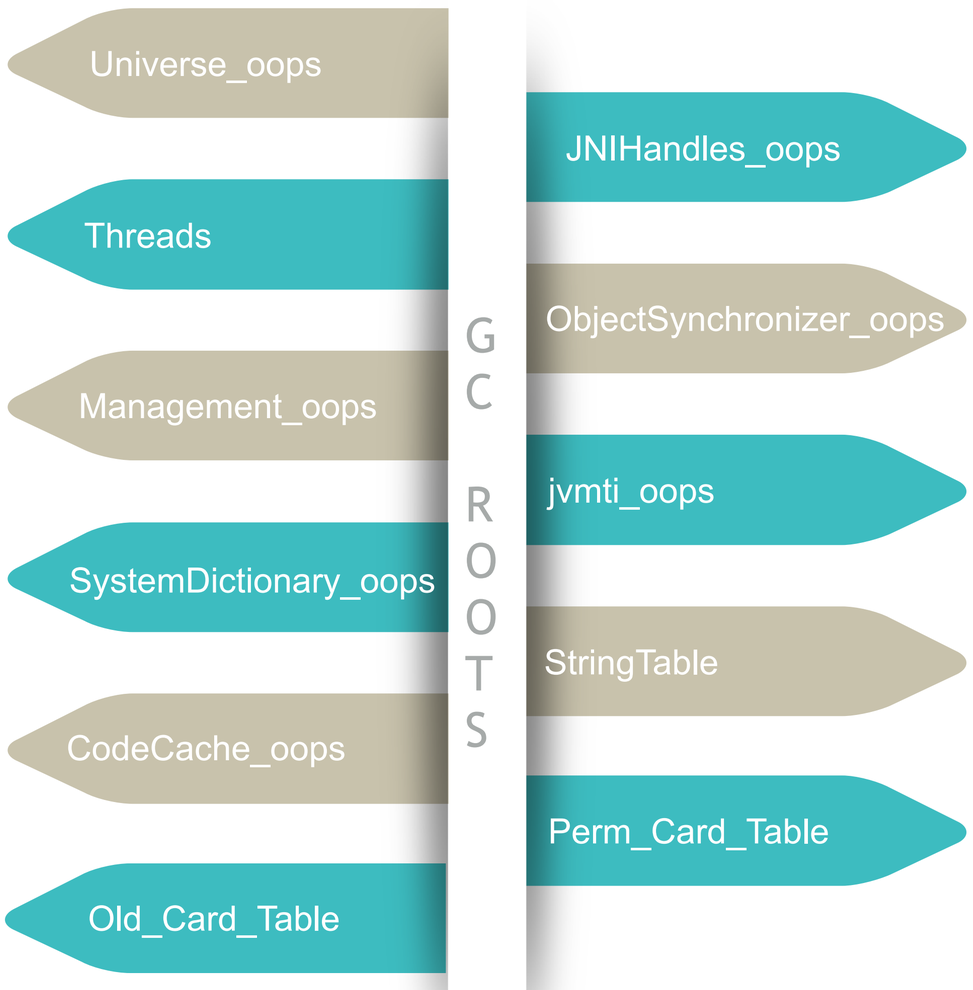
# YGC拉长

```java
import java.util.UUID;

/**
 * -XX:+UseConcMarkSweepGC -XX:+PrintGCDetails -Xmx2G -Xms2G -Xmn100M
 * @author nijiaben
 *
 */
public class StringTest {
    public static void main(String args[]){
        while(true){
            test();
        }
    }

    public static void test(){
        UUID.randomUUID().toString().intern();
    }
}
```

```
[GC (Allocation Failure) [ParNew: 81920K->9402K(92160K), 0.0189677 secs] 81920K->9402K(2086912K), 0.0190023 secs] [Times: user=0.06 sys=0.01, real=0
01 secs]
[GC (Allocation Failure) [ParNew: 91322K->10240K(92160K), 0.0688914 secs] 91322K->20848K(2086912K), 0.0689275 secs] [Times: user=0.18 sys=0.03,
real=0.07 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0353598 secs] 102768K->32190K(2086912K), 0.0353943 secs] [Times: user=0.12 sys=0.01,
real=0.03 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0366743 secs] 114110K->43658K(2086912K), 0.0367410 secs] [Times: user=0.13 sys=0.01,
real=0.04 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0416275 secs] 125578K->55126K(2086912K), 0.0416580 secs] [Times: user=0.14 sys=0.01,
real=0.04 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0461826 secs] 137046K->66595K(2086912K), 0.0462111 secs] [Times: user=0.16 sys=0.00,
real=0.05 secs]
[GC (Allocation Failure) [ParNew: 92160K->10238K(92160K), 0.0456932 secs] 148515K->78063K(2086912K), 0.0457261 secs] [Times: user=0.16 sys=0.01,
real=0.05 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0493550 secs] 159983K->89532K(2086912K), 0.0493805 secs] [Times: user=0.18 sys=0.01,
real=0.05 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0517893 secs] 171452K->101000K(2086912K), 0.0518136 secs] [Times: user=0.19 sys=0.01,
real=0.05 secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0644424 secs] 182920K->113308K(2086912K), 0.0644716 secs] [Times: user=0.21 sys=0.01,
real=0.06 secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.0617553 secs] 195228K->124785K(2086912K), 0.0617885 secs] [Times: user=0.22 sys=0.00,
real=0.06 secs]
[GC (Allocation Failure) [ParNew: 92159K->10239K(92160K), 0.0628551 secs] 206705K->136276K(2086912K), 0.0628835 secs] [Times: user=0.22 sys=0.01,
real=0.06 secs]
[GC (Allocation Failure) [ParNew: 92159K->10239K(92160K), 0.0638557 secs] 218196K->147746K(2086912K), 0.0638812 secs] [Times: user=0.23 sys=0.00,
real=0.06 secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.0794642 secs] 229666K->159217K(2086912K), 0.0795007 secs] [Times: user=0.24 sys=0.01,
real=0.08 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0794338 secs] 241137K->170752K(2086912K), 0.0794627 secs] [Times: user=0.25 sys=0.01,
real=0.08 secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0754222 secs] 252672K->182197K(2086912K), 0.0754474 secs] [Times: user=0.26 sys=0.01,
real=0.07 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0812658 secs] 264117K->193672K(2086912K), 0.0812939 secs] [Times: user=0.27 sys=0.01,
real=0.08 secs]
[GC (Allocation Failure) [ParNew: 92160K->10238K(92160K), 0.0812642 secs] 275592K->205142K(2086912K), 0.0812915 secs] [Times: user=0.28 sys=0.01,
real=0.08 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0833250 secs] 287062K->216611K(2086912K), 0.0833516 secs] [Times: user=0.30 sys=0.01,
real=0.08 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0876986 secs] 298531K->228080K(2086912K), 0.0877266 secs] [Times: user=0.30 sys=0.01,
real=0.09 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.0901577 secs] 310000K->239548K(2086912K), 0.0901828 secs] [Times: user=0.31 sys=0.00,
real=0.09 secs]
```

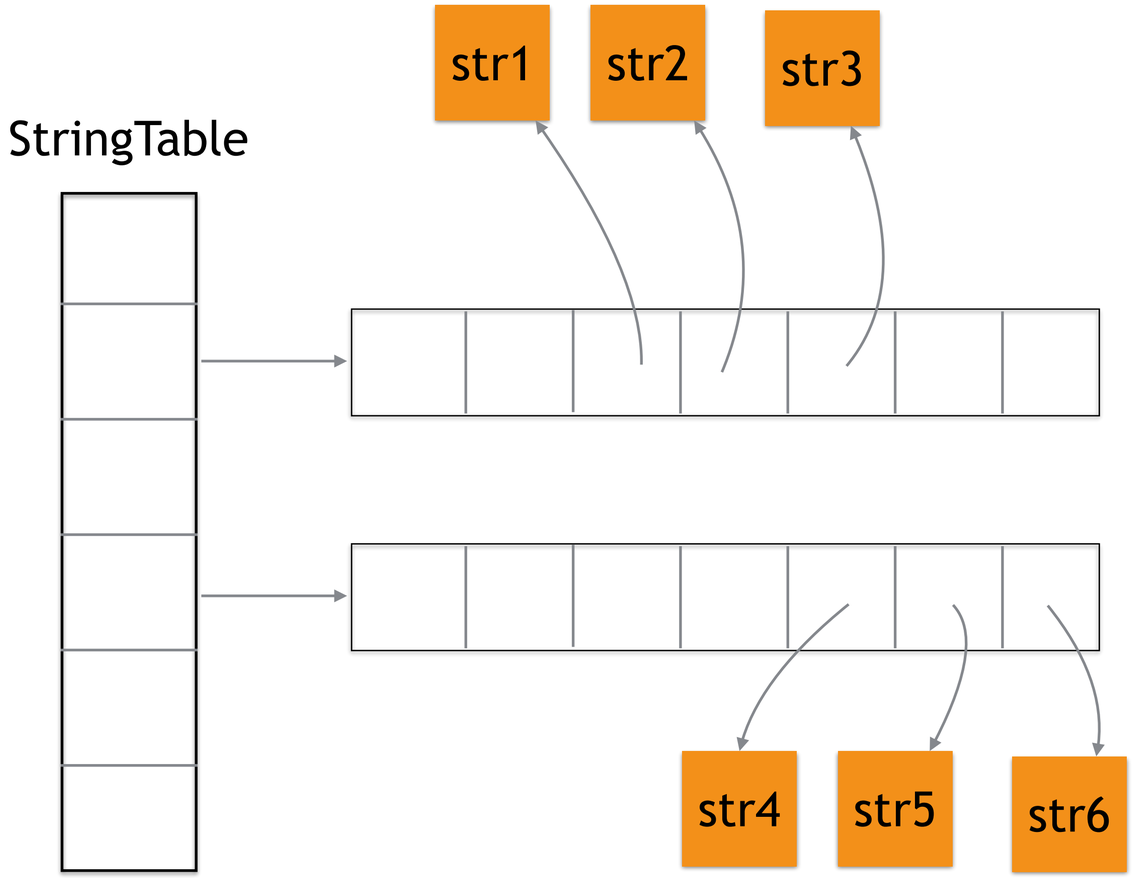
## 问题描述

- 案例的GC策略是CMS GC
- 发现ParNew GC的时间不断变长




# 影响YGC的主要因素

- YGC主要分三步骤，mark & copy & sweep
- mark和copy是同时进行的
- 被mark的对象的多少直接影响了耗时的多少




# StringTable & SystemDictionary

- StringTable主要记录了经过了String.intern后的String对象指针
- SystemDictionary主要是记录了加载的类
- 两者都是hashTable结构，因此存在Hash碰撞的风险

```
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.2069926 secs] 741843K->670951K(2086912K), 0.2070170 secs] [Times: user=0.73 sys=0.01,
real=0.21 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.2046205 secs] 752871K->681978K(2086912K), 0.2046473 secs] [Times: user=0.74 sys=0.01,
real=0.20 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.2067506 secs] 763898K->693006K(2086912K), 0.2067755 secs] [Times: user=0.75 sys=0.02,
real=0.21 secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.2096044 secs] 774926K->704033K(2086912K), 0.2096326 secs] [Times: user=0.76 sys=0.01,
real=0.21 secs]
[GC (Allocation Failure) [ParNew: 92160K->10239K(92160K), 0.2144534 secs] 785953K->715061K(2086912K), 0.2144798 secs] [Times: user=0.77 sys=0.01,
real=0.21 secs]
[GC (Allocation Failure) [ParNew: 92159K->10239K(92160K), 0.2161926 secs] 796981K->726088K(2086912K), 0.2162203 secs] [Times: user=0.79 sys=0.01,
real=0.21 secs]
[GC (Allocation Failure) [ParNew: 92159K->10239K(92160K), 0.2184479 secs] 808008K->737116K(2086912K), 0.2184988 secs] [Times: user=0.79 sys=0.01,
real=0.22 secs]
[GC (Allocation Failure) [ParNew: 92159K->10238K(92160K), 0.2535450 secs] 819036K->748143K(2086912K), 0.2535690 secs] [Times: user=0.84 sys=0.02,
real=0.26 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.2596731 secs] 830063K->759171K(2086912K), 0.2597021 secs] [Times: user=0.81 sys=0.01,
real=0.26 secs]
[GC (Allocation Failure) [ParNew: 92158K->10238K(92160K), 0.5572108 secs] 841091K->770198K(2086912K), 0.5572562 secs] [Times: user=0.93 sys=0.03,
real=0.55 secs]
[Full GC (Heap Inspection Initiated GC) [CMS: 759959K->408K(1994752K), 1.6282869 secs] 807883K->408K(2086912K), [Metaspace: 3001K->3001K(1056768K)],
1.6286912 secs] [Times: user=1.15 sys=0.41, real=1.63 secs]
[GC (Allocation Failure) [ParNew: 81920K->10238K(92160K), 0.0301978 secs] 82328K->12206K(2086912K), 0.0302262 secs] [Times: user=0.08 sys=0.00,
real=0.03 secs]
[GC (Allocation Failure) [ParNew: 92158K->10240K(92160K), 0.0503538 secs] 94126K->23392K(2086912K), 0.0503880 secs] [Times: user=0.13 sys=0.01,
real=0.05 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0356140 secs] 105312K->34248K(2086912K), 0.0356389 secs] [Times: user=0.13 sys=0.01,
real=0.04 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0422429 secs] 116168K->45307K(2086912K), 0.0422751 secs] [Times: user=0.14 sys=0.00,
real=0.04 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0473471 secs] 127227K->56495K(2086912K), 0.0473806 secs] [Times: user=0.15 sys=0.00,
real=0.05 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0580714 secs] 138415K->67379K(2086912K), 0.0581121 secs] [Times: user=0.17 sys=0.01,
real=0.06 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0462102 secs] 149299K->78407K(2086912K), 0.0462358 secs] [Times: user=0.17 sys=0.00,
real=0.05 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0624969 secs] 160327K->89649K(2086912K), 0.0625241 secs] [Times: user=0.19 sys=0.00,
real=0.06 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0650299 secs] 171569K->100490K(2086912K), 0.0650632 secs] [Times: user=0.20 sys=0.01,
real=0.07 secs]
[GC (Allocation Failure) [ParNew: 92160K->10240K(92160K), 0.0794302 secs] 182410K->111779K(2086912K), 0.0794629 secs] [Times: user=0.21 sys=0.00,
real=0.08 secs]
```

# 如何验证

- 类卸载或者StringTable清理在CMS GC或者Full GC的时候才会清理
- 通过jmap -histo:live <pid>触发一次Full GC
- 如果发现接下来的YGC的时间变短了，那说明和这两块的可能性比较大

PerfMa

感　谢　收　看

主办方 Geekbang. 极客邦科技 InfoQ

# GMTC 2018

## 全球大前端技术大会

大前端的下一站

<<扫码了解更多详情>>

极客邦科技
企业培训与咨询

Geekbang.

扫码关注

获取更多培训信息