JVM 调优经验分享：让你的 Java 应用性能更上一层楼

/ 如何用一行代码修复一个排查了一周的问题

主讲人：朱孟柱

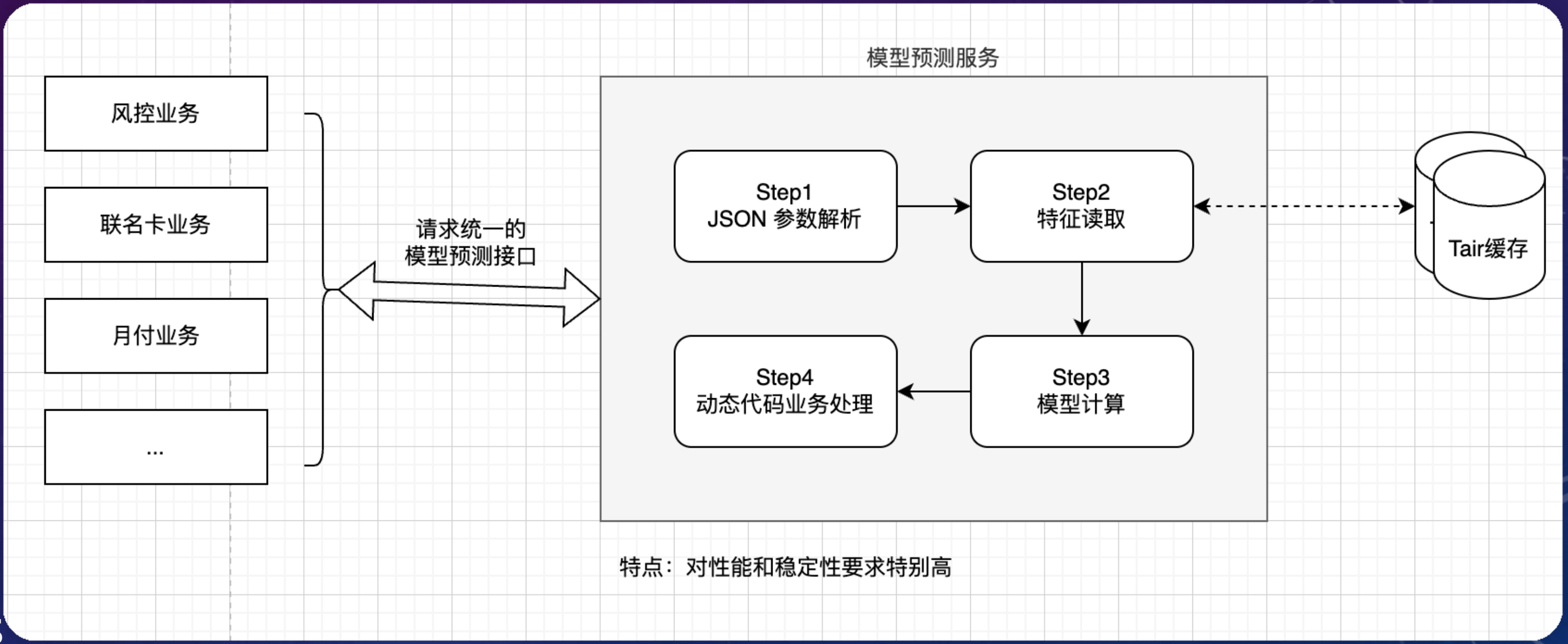
目标

通过这个案例，了解：

- 排查性能问题的一些基本思路；
- G1GC 的基本原理；
- Eclipse Memory Analyzer 的基本使用方法；
- JVM 内存 dump 方法及分析思路；

问题：服务请求成功率从 99.99% 下降到 99%

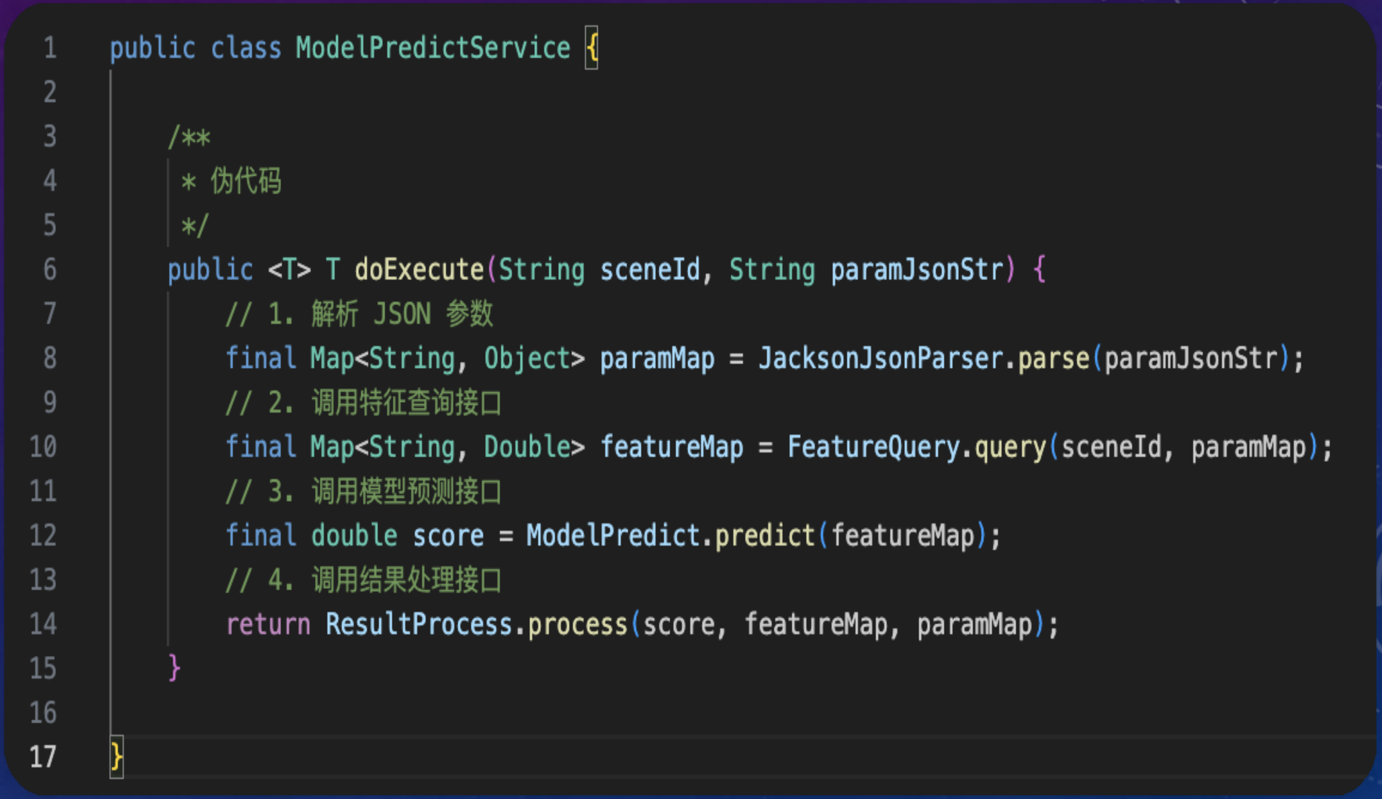
- 模型预测服务
- 语言：Java8，G1GC
- 机器：4C8G * 40
- QPS：峰值 1W
- SLA：99.99%
- 性能：TP9999 <= 50ms



特点：对性能和稳定性要求特别高

问题：服务请求成功率从 99.99% 下降到 99%

- 模型预测服务
- 语言：Java8，G1GC
- 机器：4C8G * 40
- QPS：峰值 1W
- SLA：99.99%
- 性能：TP9999 <= 50ms



```java
public class ModelPredictService {

    /**
     * 伪代码
     */
    public <T> T doExecute(String sceneId, String paramJsonStr) {
        // 1. 解析 JSON 参数
        final Map<String, Object> paramMap = JacksonJsonParser.parse(paramJsonStr);
        // 2. 调用特征查询接口
        final Map<String, Double> featureMap = FeatureQuery.query(sceneId, paramMap);
        // 3. 调用模型预测接口
        final double score = ModelPredict.predict(featureMap);
        // 4. 调用结果处理接口
        return ResultProcess.process(score, featureMap, paramMap);
    }

}
```

问题：服务请求成功率从 99.99% 下降到 99%

首轮排查：

- 最近没有代码变更
- 最近没有重启服务
- 最近没有修改机器配置
- 排查了服务器网络状况，正常
- 排查了 Tair 缓存服务状况，正常
- 排查了服务器和 Tair 之间的网络拓扑，正常

问题：服务请求成功率从 99.99% 下降到 99%

问题特征:

- 服务本身 TP 升高，TP9999 大于 50ms（客户端请求成功率下降的直接原因）
- 请求缓存 TP 指标升高
- YGC 次数变多
- YGC 耗时上升，从几毫秒慢慢上升到最高接近 200ms，直到 FullGC
- 每次 FullGC 性能以上指标变好，然后慢慢变差（每天凌晨 3 点定时主动 FullGC）

开始怀疑是 YGC 出现问题。

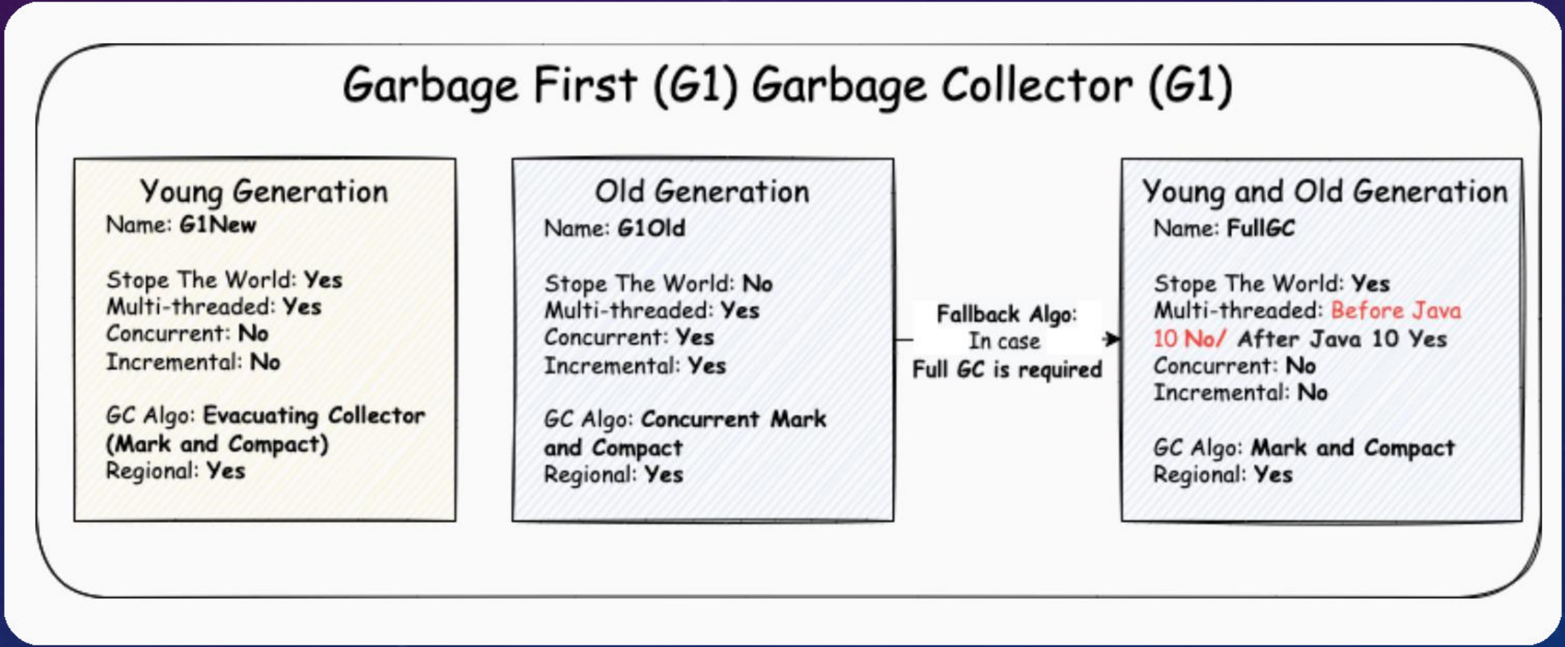
问题：服务请求成功率从 99.99% 下降到 99%

定位到时 YGC 问题后，首先，做止损操作：

- 对机器进行了扩容
- 在午高峰和晚高峰前，各增加一次主动 GC
- 在午高峰和晚高峰期间，紧盯告警，对性能异常的机器进行流量禁用以及手动 FullGC

排查：回顾 G1GC 基本原理

- 分代



Garbage First (G1) Garbage Collector (G1)

Young Generation
Name: **G1New**

Stope The World: **Yes**
Multi-threaded: **Yes**
Concurrent: **No**
Incremental: **No**

GC Algo: **Evacuating Collector (Mark and Compact)**
Regional: **Yes**

Old Generation
Name: **G1Old**

Stope The World: **No**
Multi-threaded: **Yes**
Concurrent: **Yes**
Incremental: **Yes**

GC Algo: **Concurrent Mark and Compact**
Regional: **Yes**

Fallback Algo:
In case
Full GC is required

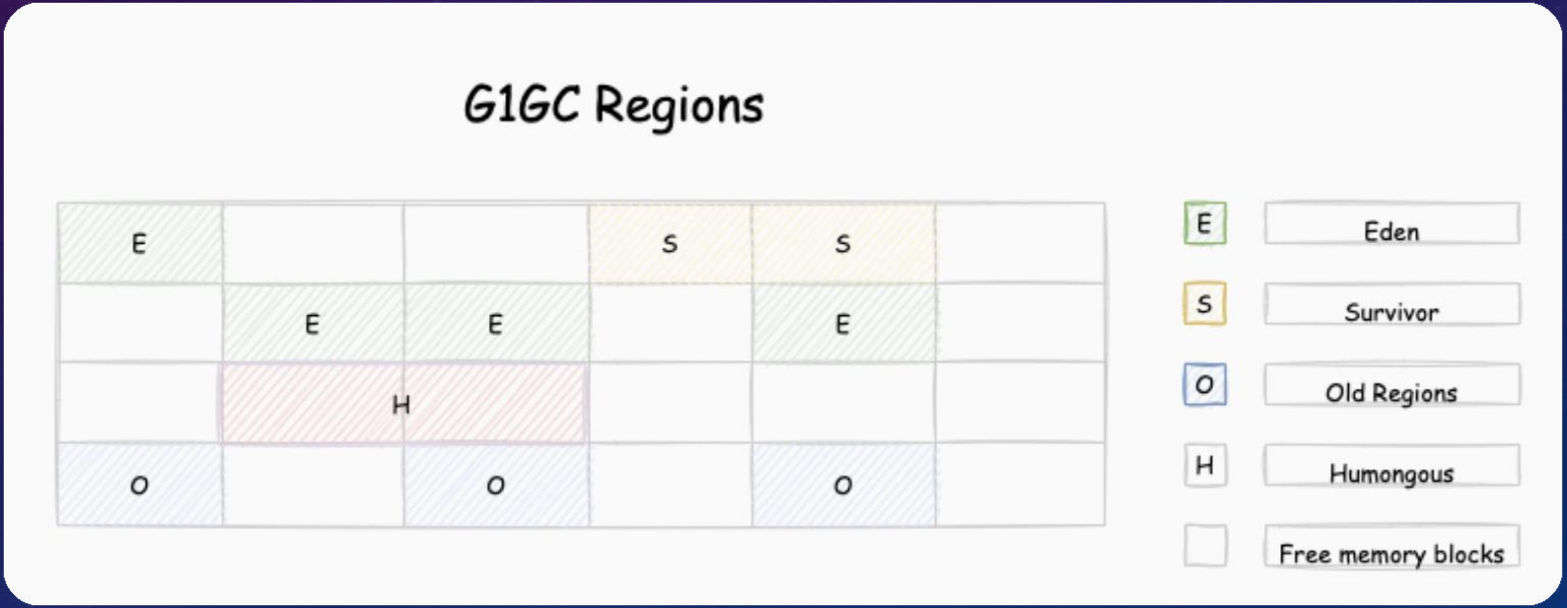
Young and Old Generation
Name: **FullGC**

Stope The World: **Yes**
Multi-threaded: Before Java 10 No/ **After Java 10 Yes**
Concurrent: **No**
Incremental: **No**

GC Algo: **Mark and Compact**
Regional: **Yes**

排查：回顾 G1GC 基本原理

- 分代
- 分区



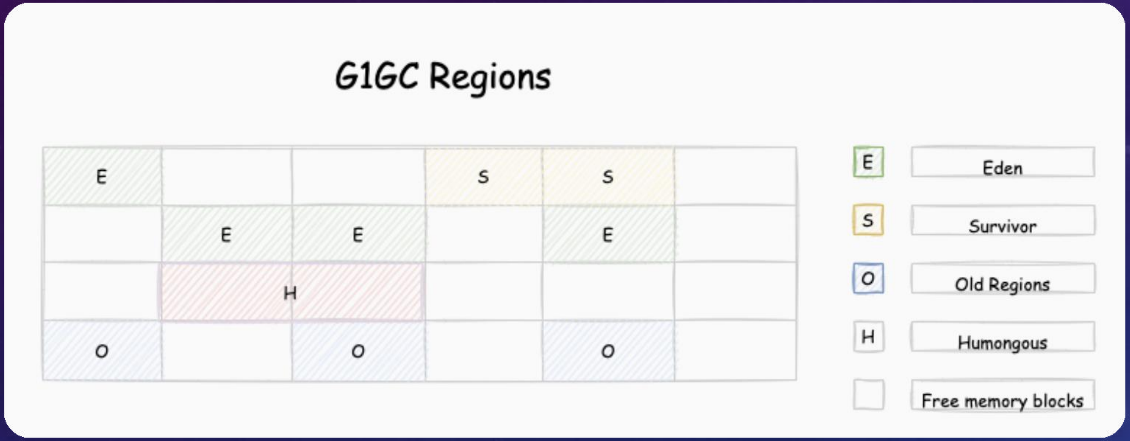
G1GC Regions

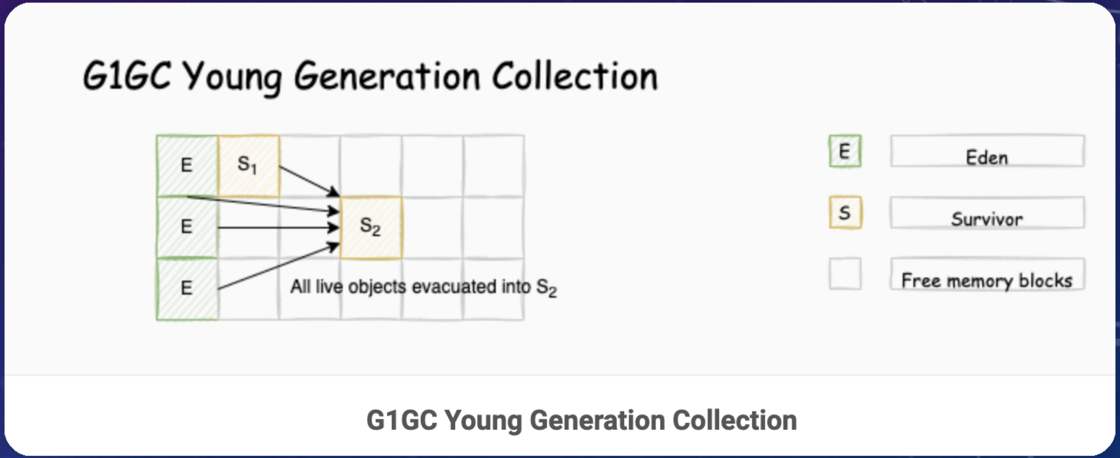
E			S	S	
	E	E		E	
	H				
O		O		O	

E	Eden
S	Survivor
O	Old Regions
H	Humongous
	Free memory blocks

排查：回顾 G1GC 基本原理

- 分代
- 分区

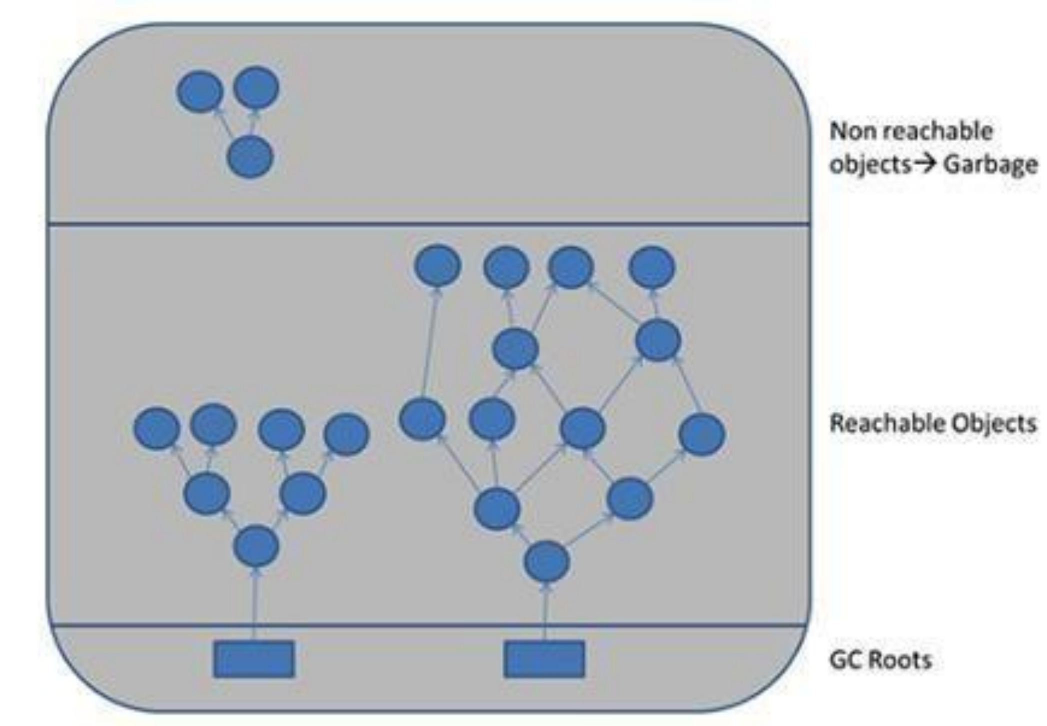



G1GC Young Generation Collection
E
E
E
S1
S2
All live objects evacuated into S2
E
Eden
S
Survivor
Free memory blocks

G1GC Young Generation Collection

排查：回顾 G1GC 基本原理

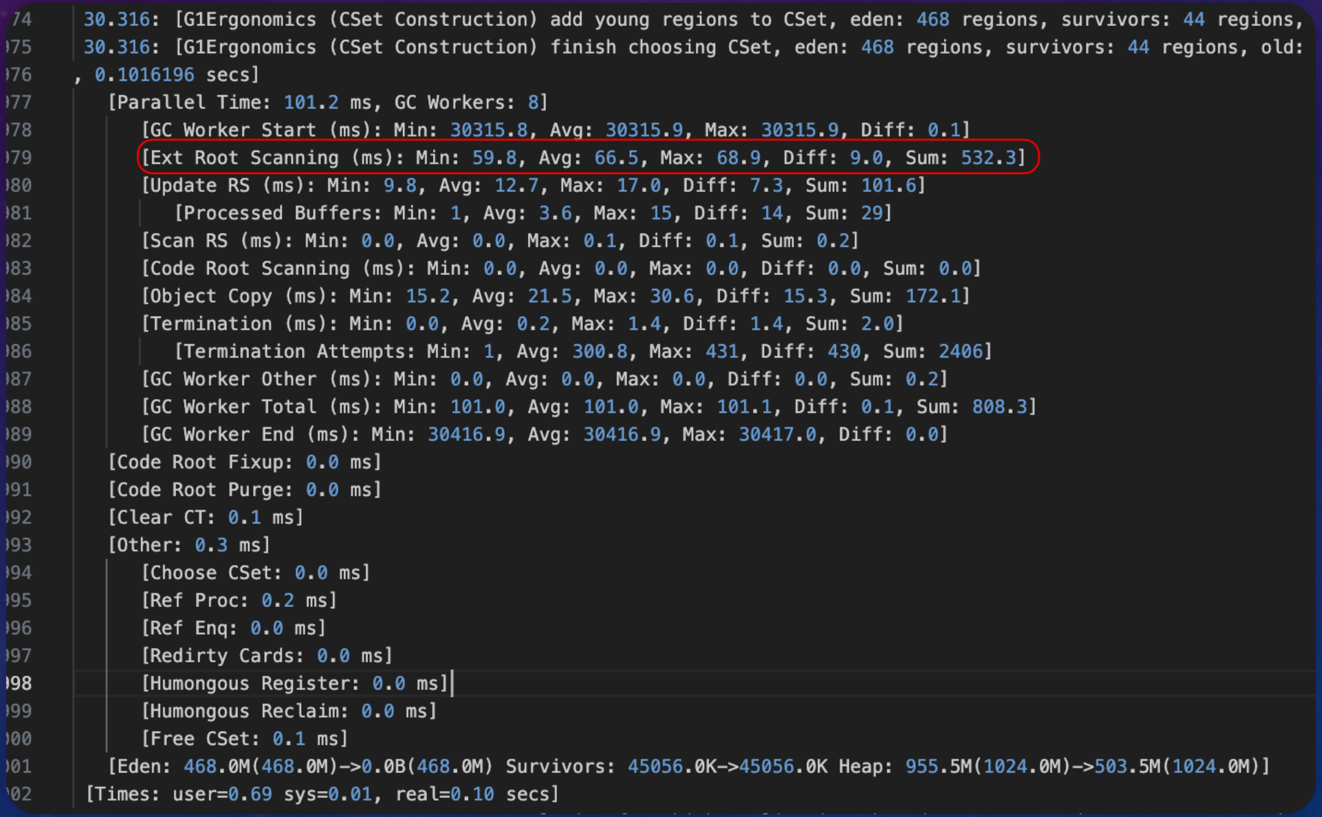
- 分区
- 分代
- 基于 GC Roots 可达性分析



排查：分析 GC 日志

查看 GC 日志:

- Root Scanning 耗时特别长(正常一般不超过 5ms，现在高达 66ms)



```
30.316: [G1Ergonomics (CSet Construction) add young regions to CSet, eden: 468 regions, survivors: 44 regions,
30.316: [G1Ergonomics (CSet Construction) finish choosing CSet, eden: 468 regions, survivors: 44 regions, old:
, 0.1016196 secs]
   [Parallel Time: 101.2 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 30315.8, Avg: 30315.9, Max: 30315.9, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 59.8, Avg: 66.5, Max: 68.9, Diff: 9.0, Sum: 532.3]
      [Update RS (ms): Min: 9.8, Avg: 12.7, Max: 17.0, Diff: 7.3, Sum: 101.6]
         [Processed Buffers: Min: 1, Avg: 3.6, Max: 15, Diff: 14, Sum: 29]
      [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Object Copy (ms): Min: 15.2, Avg: 21.5, Max: 30.6, Diff: 15.3, Sum: 172.1]
      [Termination (ms): Min: 0.0, Avg: 0.2, Max: 1.4, Diff: 1.4, Sum: 2.0]
         [Termination Attempts: Min: 1, Avg: 300.8, Max: 431, Diff: 430, Sum: 2406]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2]
      [GC Worker Total (ms): Min: 101.0, Avg: 101.0, Max: 101.1, Diff: 0.1, Sum: 808.3]
      [GC Worker End (ms): Min: 30416.9, Avg: 30416.9, Max: 30417.0, Diff: 0.0]
   [Code Root Fixup: 0.0 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.1 ms]
   [Other: 0.3 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.2 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.0 ms]
      [Humongous Register: 0.0 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 0.1 ms]
   [Eden: 468.0M(468.0M)->0.0B(468.0M) Survivors: 45056.0K->45056.0K Heap: 955.5M(1024.0M)->503.5M(1024.0M)]
 [Times: user=0.69 sys=0.01, real=0.10 secs]
```

排查：分析 GC 日志

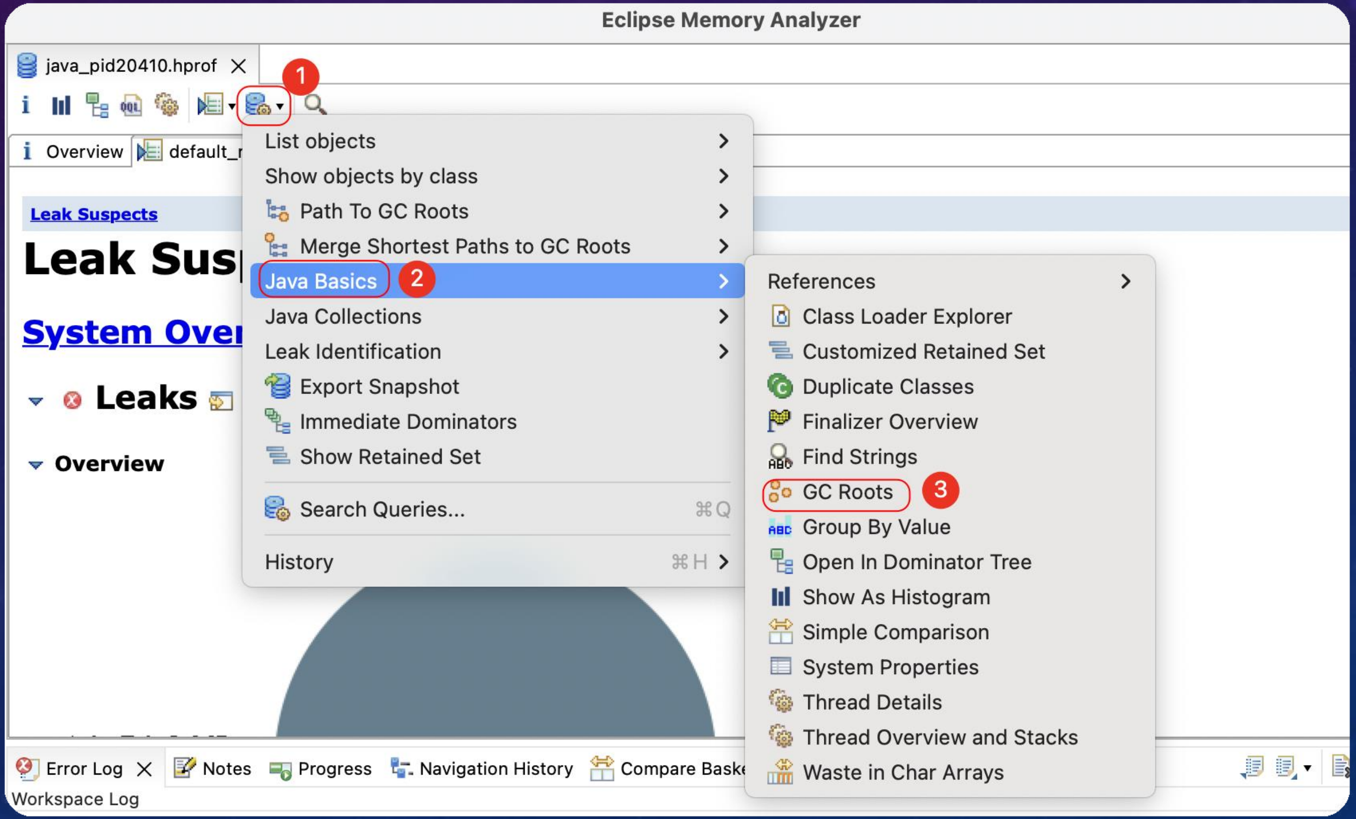
什么是 GC Roots? 网上会告诉你:

- JNI global reference
- JNI local reference
- Local variable on stack
- Monitor
- System class
- Thread
- **Others**

Others 包含哪些取决于 JVM 实现，约等于没说。

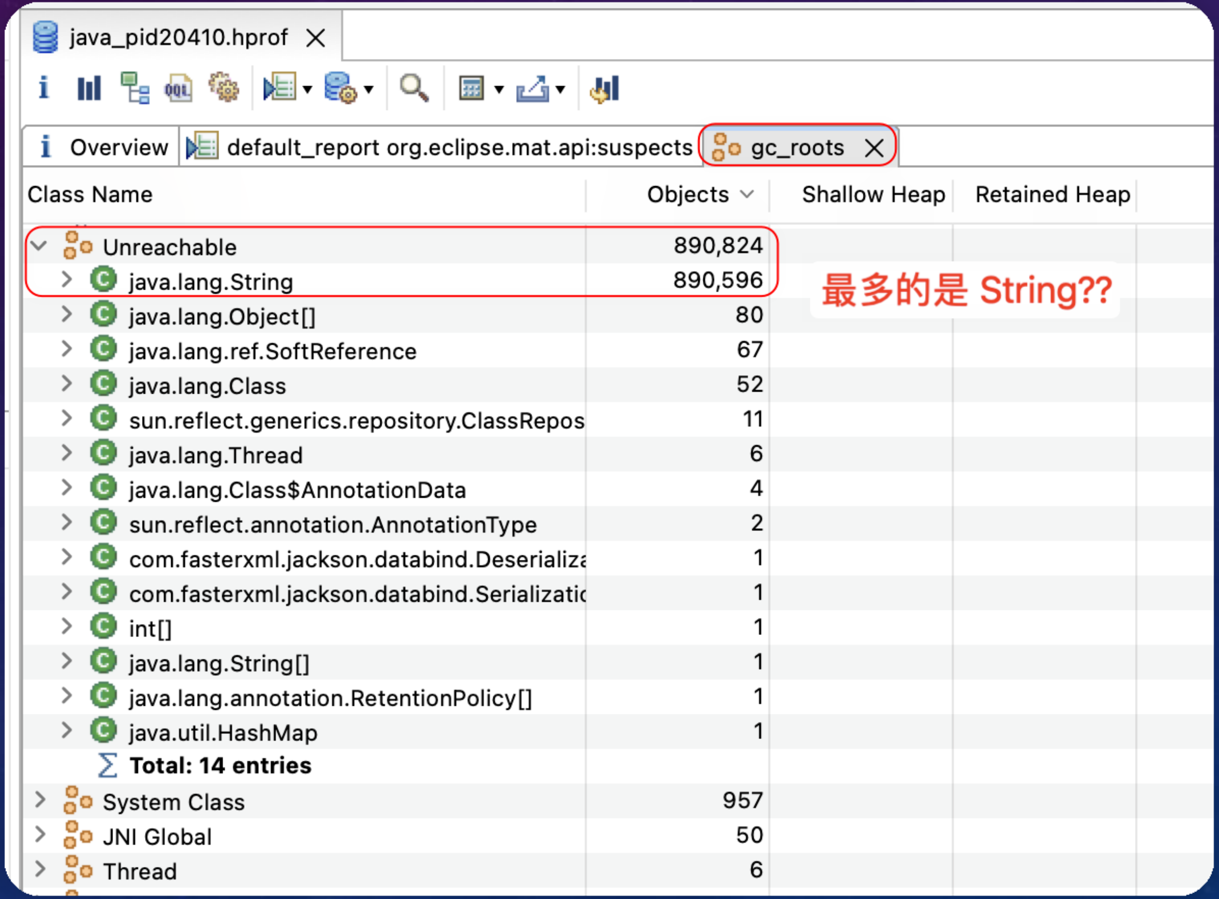
排查：堆内存分析

- Dump 问题现场内存
  - jmap -dump:format=b,file=dump.hprof <pid>
  - 这里一定不能不能加 live 参数，否则会先进行 FullGC，现场就丢失了
- 使用 Eclipse Memory Analyzer 分析



排查：堆内存分析

- GC roots 最多的竟然是 String?



java_pid20410.hprof

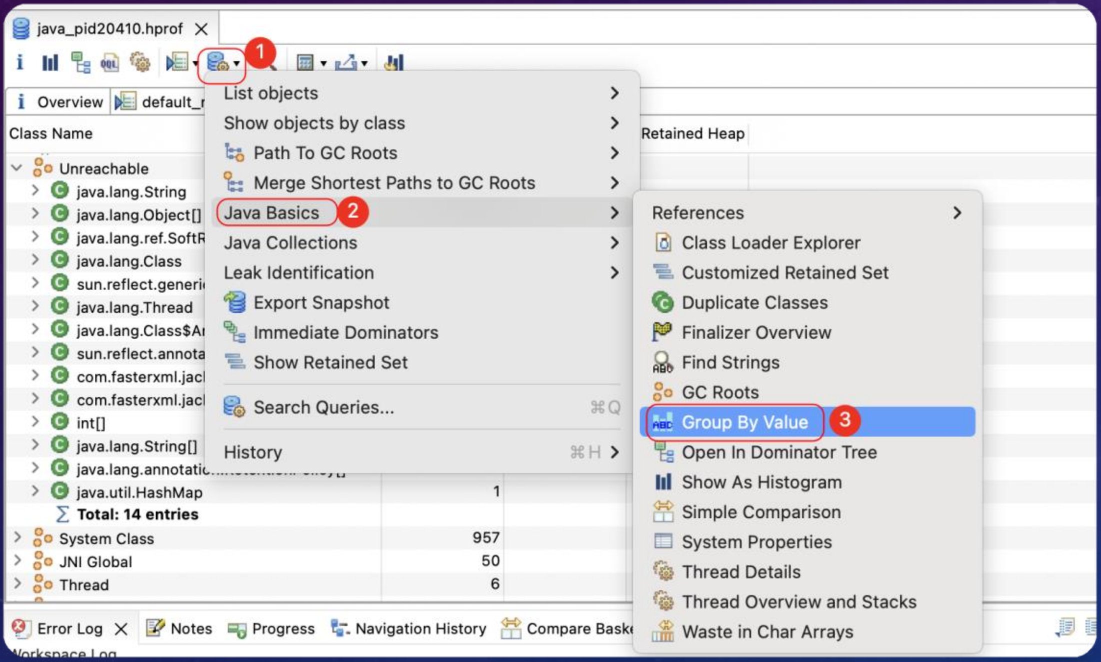
Overview | default_report org.eclipse.mat.api:suspects | gc_roots

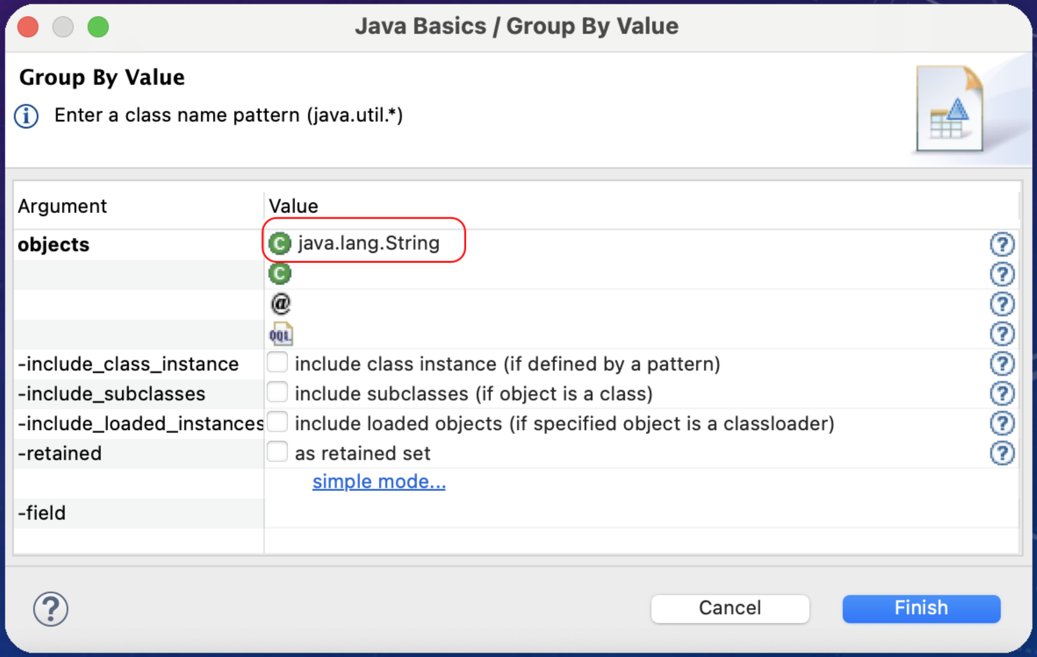
Class Name	Objects	Shallow Heap	Retained Heap
Unreachable	890,824		
java.lang.String	890,596		
java.lang.Object[]	80		
java.lang.ref.SoftReference	67		
java.lang.Class	52		
sun.reflect.generics.repository.ClassRepos	11		
java.lang.Thread	6		
java.lang.Class$AnnotationData	4		
sun.reflect.annotation.AnnotationType	2		
com.fasterxml.jackson.databind.Deserializa	1		
com.fasterxml.jackson.databind.Serializatic	1		
int[]	1		
java.lang.String[]	1		
java.lang.annotation.RetentionPolicy[]	1		
java.util.HashMap	1		
Total: 14 entries			
System Class	957		
JNI Global	50		
Thread	6		

最多的是 String??

排查：堆内存分析

- 笨办法，使用 MAT 的 Group By Value 功能，看看到底是哪些 String：





Java Basics / Group By Value

Group By Value

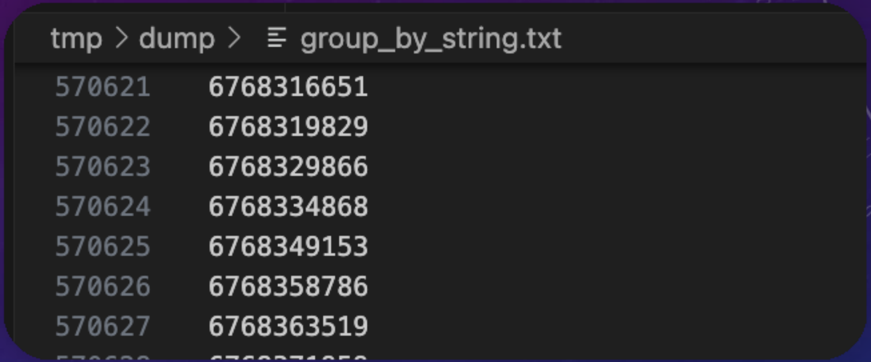
Enter a class name pattern (java.util.*)

Argument	Value
objects	java.lang.String
	@
	OQL
-include_class_instance	□ include class instance (if defined by a pattern)
-include_subclasses	□ include subclasses (if object is a class)
-include_loaded_instances	□ include loaded objects (if specified object is a classloader)
-retained	□ as retained set
	simple mode...
-field	

Cancel　Finish

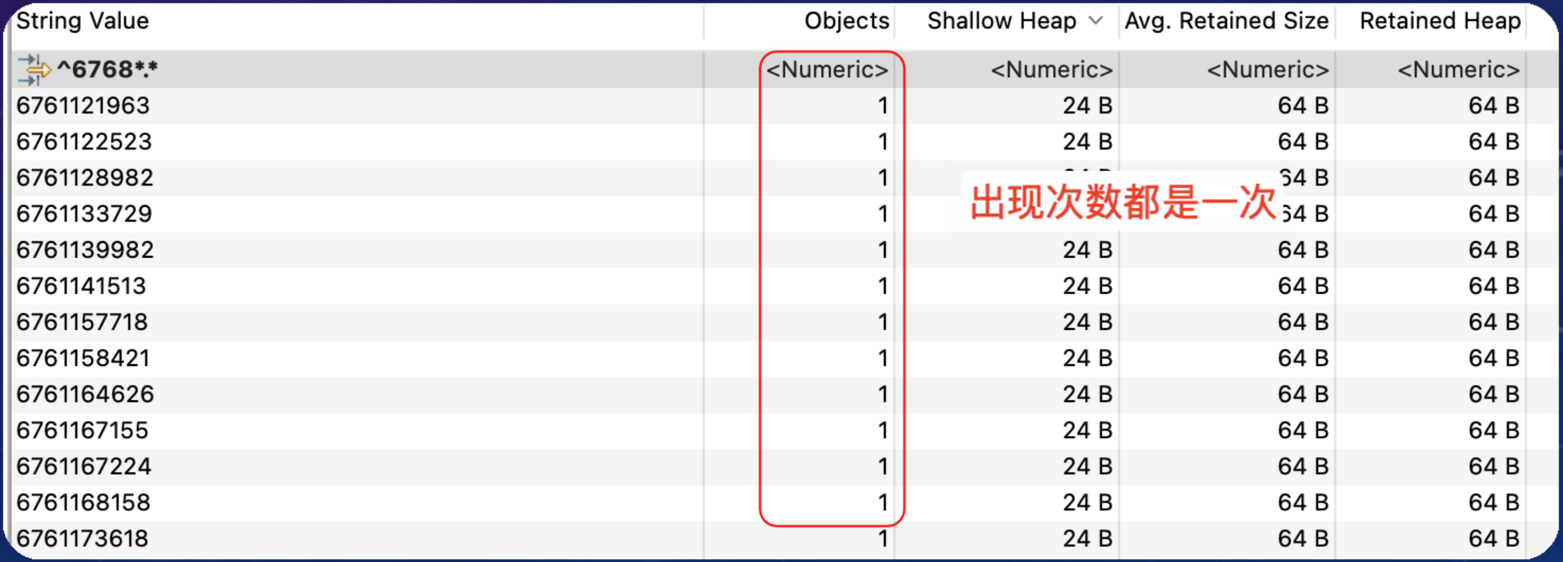
排查：堆内存分析

- 发现了上百万个纯数字组成的字符串
  - 每个字符串只出现一次
  - 看起来很像是 userId
- 这些字符串哪里来的呢?
- 为什么字符串会是 GC Roots?



tmp > dump > group_by_string.txt

```
6768316651
6768319829
6768329866
6768334868
6768349153
6768358786
6768363519
```

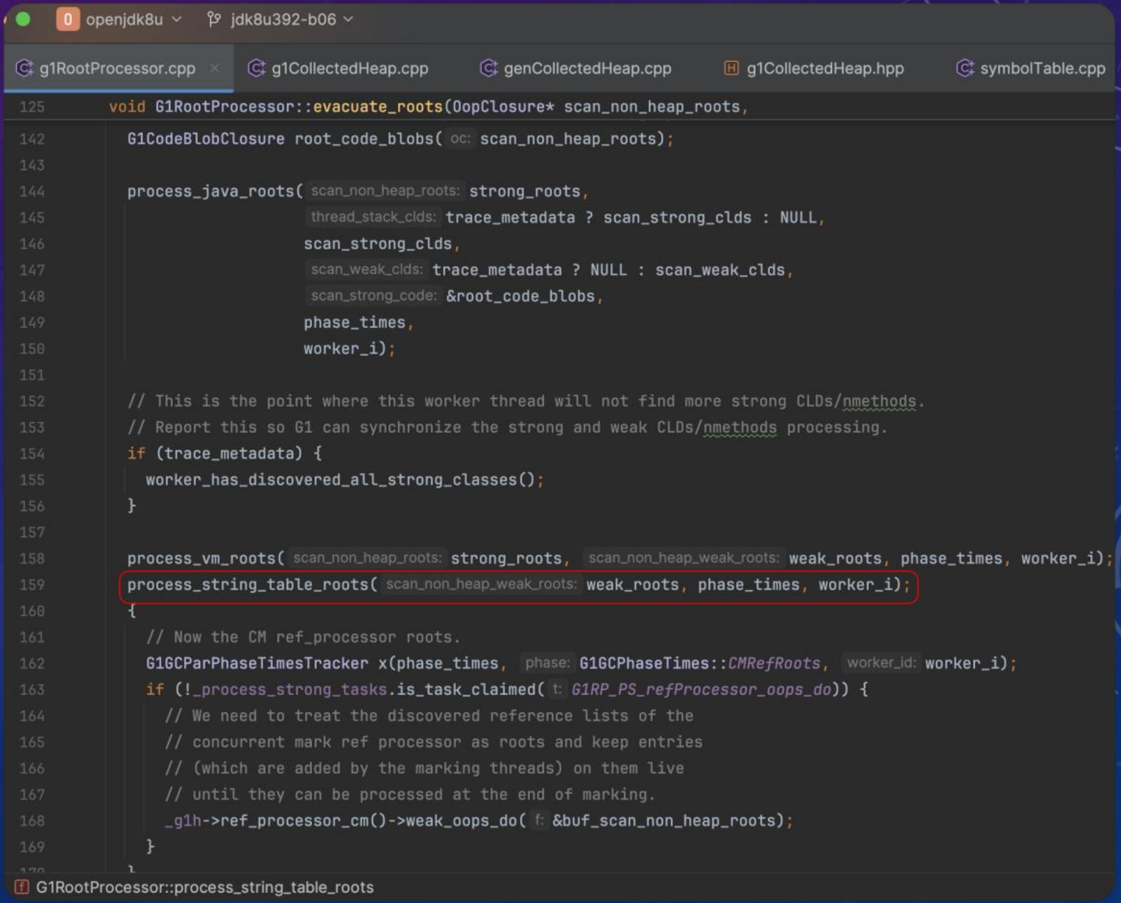


String Value	Objects	Shallow Heap	Avg. Retained Size	Retained Heap
^6768*.*	<Numeric>	<Numeric>	<Numeric>	<Numeric>
6761121963	1	24 B	64 B	64 B
6761122523	1	24 B	64 B	64 B
6761128982	1	24 B	64 B	64 B
6761133729	1	[illegible]	64 B	64 B
6761139982	1	24 B	64 B	64 B
6761141513	1	24 B	64 B	64 B
6761157718	1	24 B	64 B	64 B
6761158421	1	24 B	64 B	64 B
6761164626	1	24 B	64 B	64 B
6761167155	1	24 B	64 B	64 B
6761167224	1	24 B	64 B	64 B
6761168158	1	24 B	64 B	64 B
6761173618	1	24 B	64 B	64 B

出现次数都是一次

排查： 分析 JDK 源码

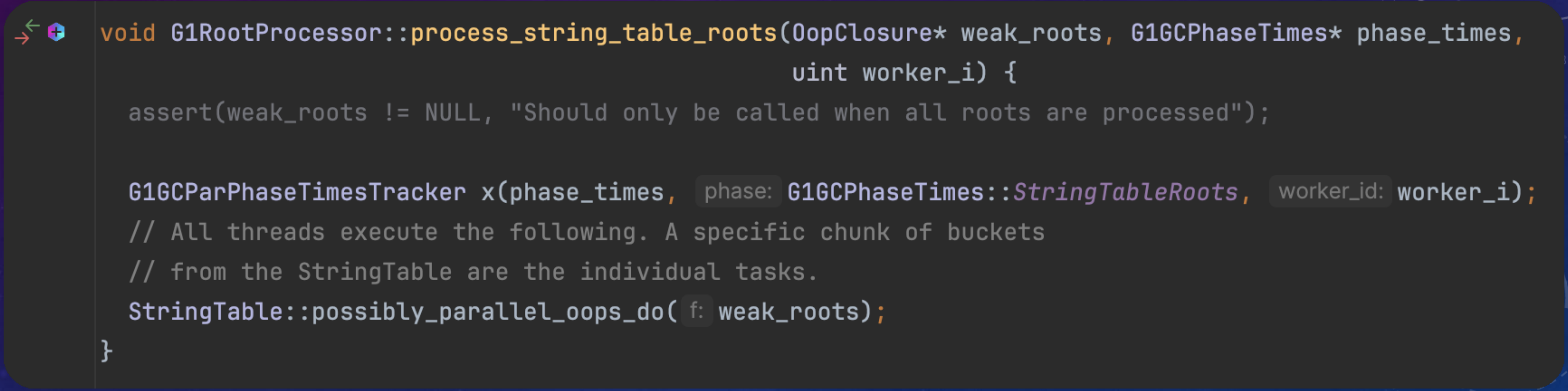
- G1RootProcessor::evacuate_roots
  - process_java_roots
  - process_vm_roots
  - process_string_table_roots
  - weak_oops_do
  - ……
- **YGC 时，会扫描 StringTable（为什么？）**



```
void G1RootProcessor::evacuate_roots(OopClosure* scan_non_heap_roots,
  G1CodeBlobClosure root_code_blobs( oc: scan_non_heap_roots);

  process_java_roots( scan_non_heap_roots: strong_roots,
                     thread_stack_clds: trace_metadata ? scan_strong_clds : NULL,
                     scan_strong_clds,
                     scan_weak_clds: trace_metadata ? NULL : scan_weak_clds,
                     scan_strong_code: &root_code_blobs,
                     phase_times,
                     worker_i);

  // This is the point where this worker thread will not find more strong CLDs/nmethods.
  // Report this so G1 can synchronize the strong and weak CLDs/nmethods processing.
  if (trace_metadata) {
    worker_has_discovered_all_strong_classes();
  }

  process_vm_roots( scan_non_heap_roots: strong_roots,  scan_non_heap_weak_roots: weak_roots, phase_times, worker_i);
  process_string_table_roots( scan_non_heap_weak_roots: weak_roots, phase_times, worker_i);
  {
    // Now the CM ref_processor roots.
    G1GCParPhaseTimesTracker x(phase_times,  phase: G1GCPhaseTimes::CMRefRoots,  worker_id: worker_i);
    if (!_process_strong_tasks.is_task_claimed( t: G1RP_PS_refProcessor_oops_do)) {
      // We need to treat the discovered reference lists of the
      // concurrent mark ref processor as roots and keep entries
      // (which are added by the marking threads) on them live
      // until they can be processed at the end of marking.
      _g1h->ref_processor_cm()->weak_oops_do( f: &buf_scan_non_heap_roots);
    }
  }
```

排查： 分析 JDK 源码

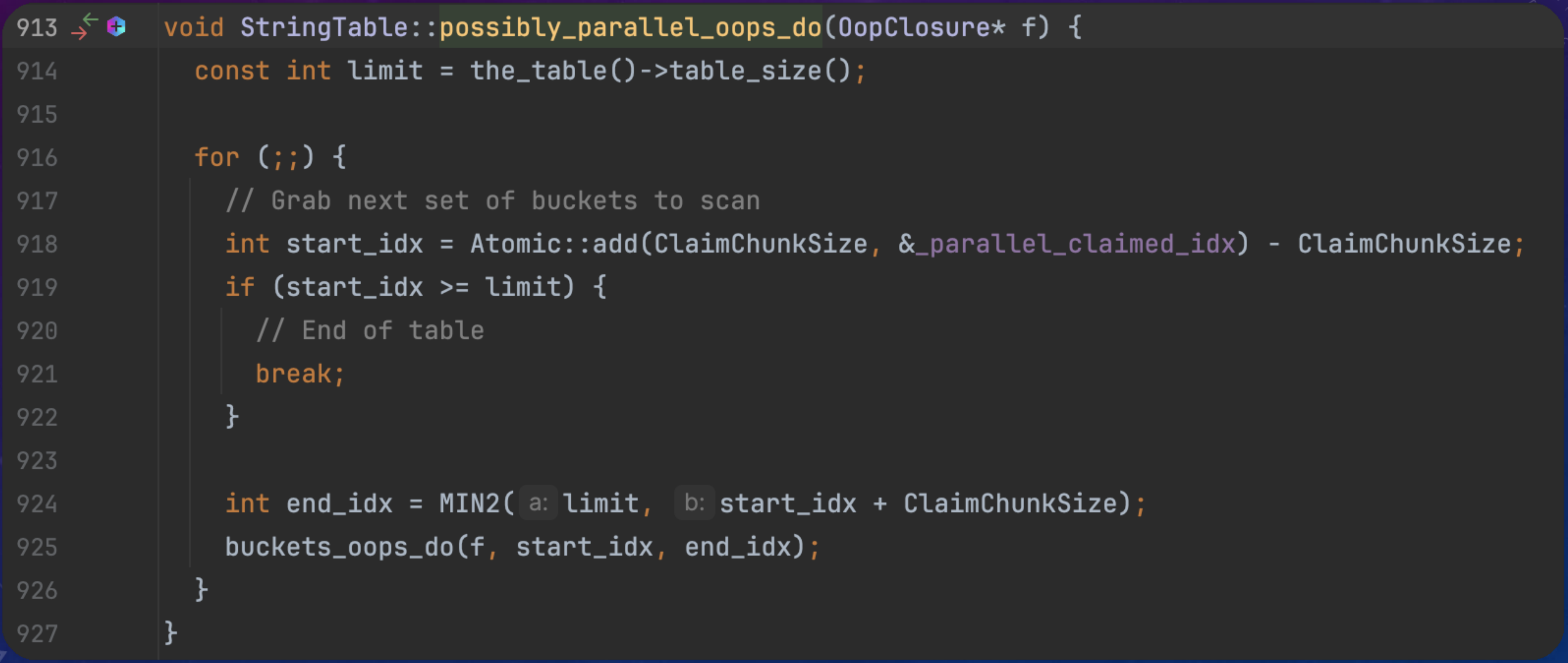
- G1RootProcessor ::process_string_table_roots



```
void G1RootProcessor::process_string_table_roots(OopClosure* weak_roots, G1GCPhaseTimes* phase_times,
                                                 uint worker_i) {
  assert(weak_roots != NULL, "Should only be called when all roots are processed");

  G1GCParPhaseTimesTracker x(phase_times, phase: G1GCPhaseTimes::StringTableRoots, worker_id: worker_i);
  // All threads execute the following. A specific chunk of buckets
  // from the StringTable are the individual tasks.
  StringTable::possibly_parallel_oops_do( f: weak_roots);
}
```

排查： 分析 JDK 源码

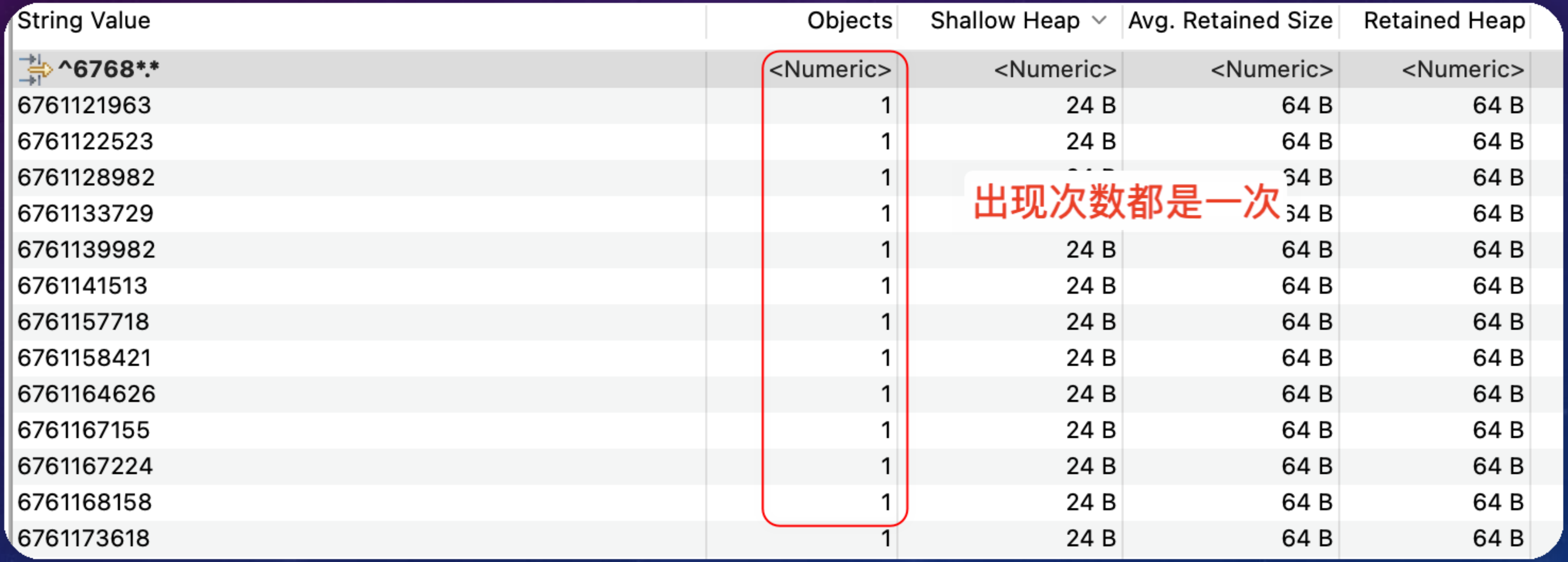
- StringTable::possibly_parallel_oops_do



```
void StringTable::possibly_parallel_oops_do(OopClosure* f) {
  const int limit = the_table()->table_size();

  for (;;) {
    // Grab next set of buckets to scan
    int start_idx = Atomic::add(ClaimChunkSize, &_parallel_claimed_idx) - ClaimChunkSize;
    if (start_idx >= limit) {
      // End of table
      break;
    }

    int end_idx = MIN2( a: limit,  b: start_idx + ClaimChunkSize);
    buckets_oops_do(f, start_idx, end_idx);
  }
}
```

罪魁祸首 ：？

疑问：这些字符串是怎么进入 StringTable 的?



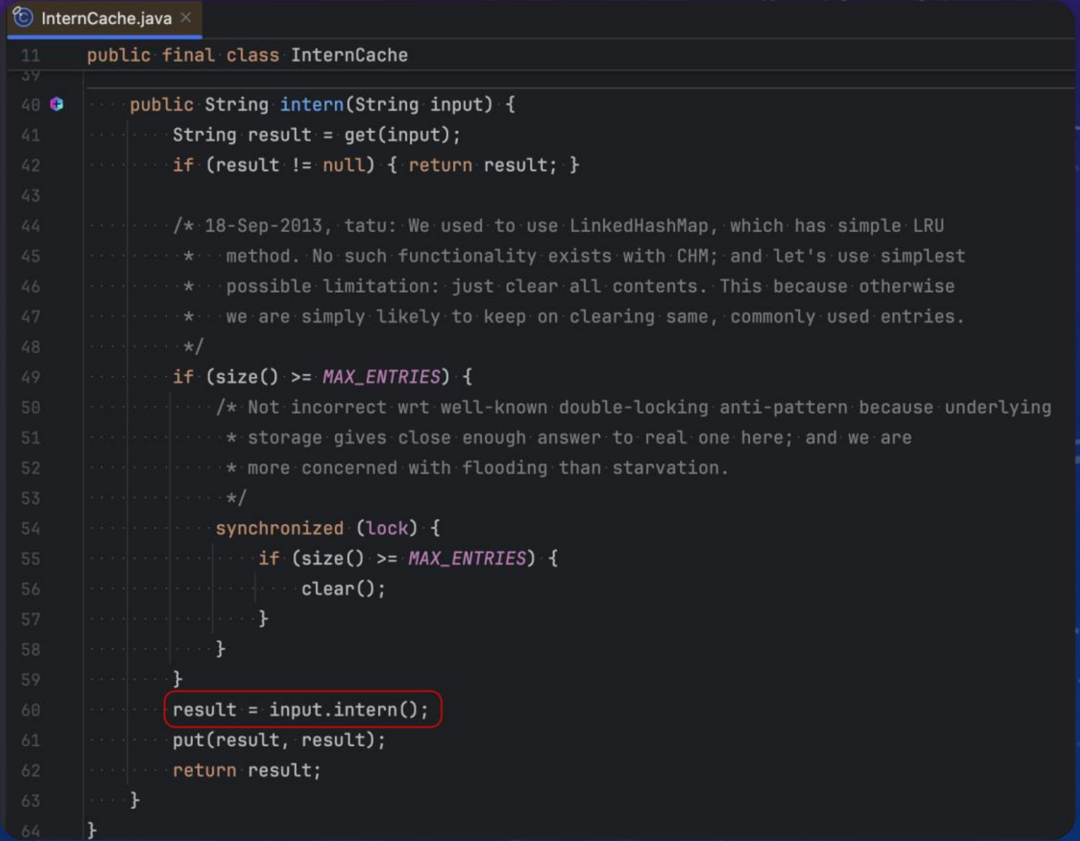
String Value	Objects	Shallow Heap	Avg. Retained Size	Retained Heap
^6768*.*	<Numeric>	<Numeric>	<Numeric>	<Numeric>
6761121963	1	24 B	64 B	64 B
6761122523	1	24 B	64 B	64 B
6761128982	1	24 B	64 B	64 B
6761133729	1		64 B	64 B
6761139982	1	24 B	64 B	64 B
6761141513	1	24 B	64 B	64 B
6761157718	1	24 B	64 B	64 B
6761158421	1	24 B	64 B	64 B
6761164626	1	24 B	64 B	64 B
6761167155	1	24 B	64 B	64 B
6761167224	1	24 B	64 B	64 B
6761168158	1	24 B	64 B	64 B
6761173618	1	24 B	64 B	64 B

出现次数都是一次

罪魁祸首 ： Jackson

- 我们使用 Jackson 进行反序列处理
- Jackson 会对 JSON 的属性名进行 intern 处理
- 早在 2016 年，就有人提了相关 issue：#332

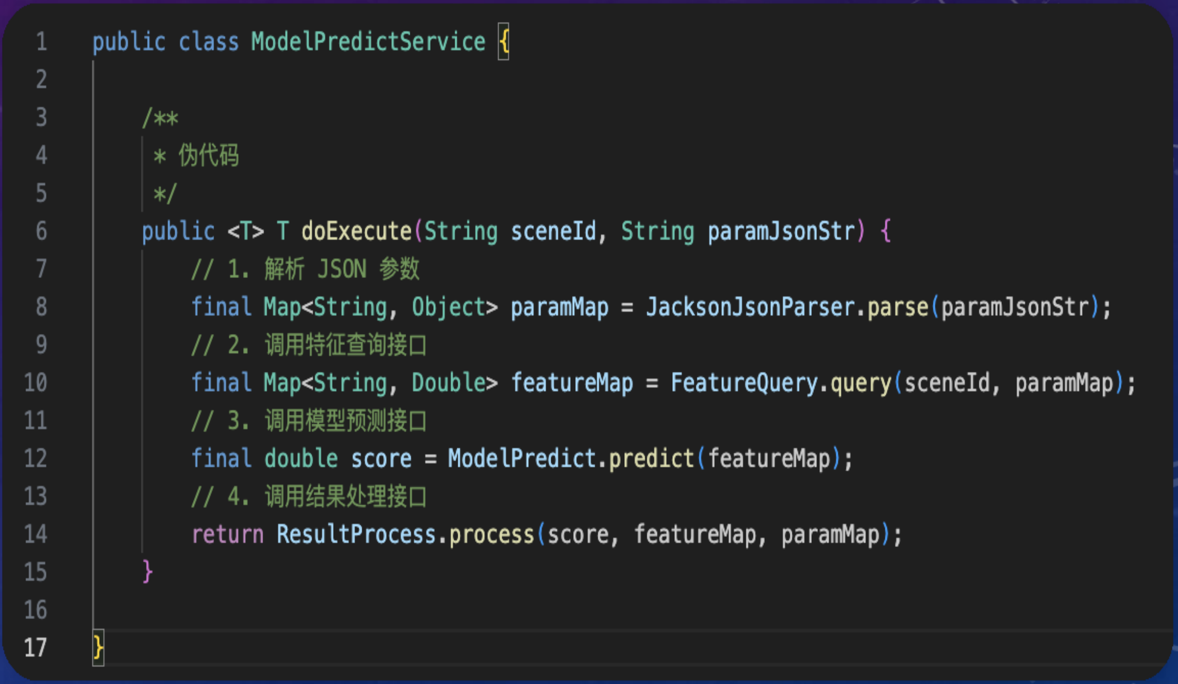
```
// 用户请求参数
{
    "6768349153": {
        "orderId": "xx",
        "age": 30,
        "orderTime": 1721195690135
    }
}
```



```java
public final class InternCache

    public String intern(String input) {
        String result = get(input);
        if (result != null) { return result; }

        /* 18-Sep-2013, tatu: We used to use LinkedHashMap, which has simple LRU
         *   method. No such functionality exists with CHM; and let's use simplest
         *   possible limitation: just clear all contents. This because otherwise
         *   we are simply likely to keep on clearing same, commonly used entries.
         */
        if (size() >= MAX_ENTRIES) {
            /* Not incorrect wrt well-known double-locking anti-pattern because underlying
             * storage gives close enough answer to real one here; and we are
             * more concerned with flooding than starvation.
             */
            synchronized (lock) {
                if (size() >= MAX_ENTRIES) {
                    clear();
                }
            }
        }
        result = input.intern();
        put(result, result);
        return result;
    }
}
```

罪魁祸首 ：Jackson

```
// 用户请求参数
{
    "6768349153": {
        "orderId": "xx",
        "age": 30,
        "orderTime": 1721195690135
    }
}
```



```java
public class ModelPredictService {

    /**
     * 伪代码
     */
    public <T> T doExecute(String sceneId, String paramJsonStr) {
        // 1. 解析 JSON 参数
        final Map<String, Object> paramMap = JacksonJsonParser.parse(paramJsonStr);
        // 2. 调用特征查询接口
        final Map<String, Double> featureMap = FeatureQuery.query(sceneId, paramMap);
        // 3. 调用模型预测接口
        final double score = ModelPredict.predict(featureMap);
        // 4. 调用结果处理接口
        return ResultProcess.process(score, featureMap, paramMap);
    }

}
```

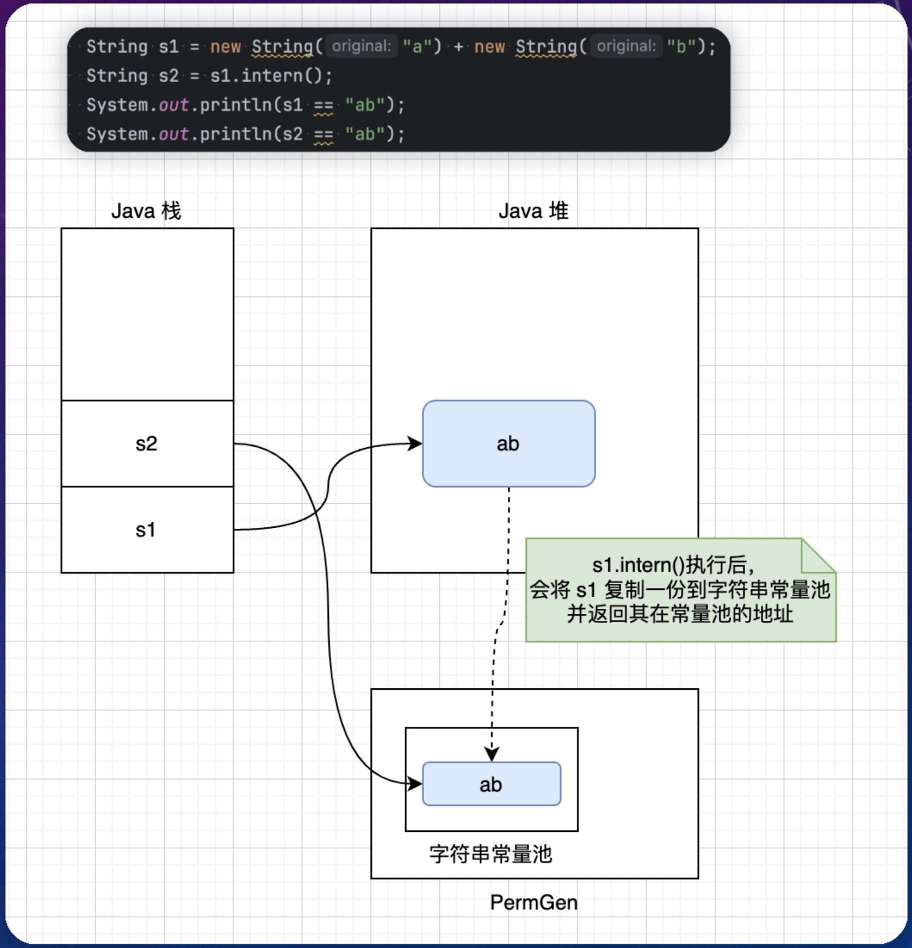
拓展知识：String.intern

- 不『经典』的『面试题』

```
public class StringIntern {

    public static void main(String[] args) {
        // Java 6 和 Java 7/8 运行结果分别是什么
   String s1 = new String("a") + new String("b");
        String s2 = s1.intern();
        System.out.println(s1 == "ab");
        System.out.println(s2 == "ab");
    }
}
```

拓展知识：String.intern

- Java 6
  - 字符串常量池在 PermGen
- 答案
  - s1==“ab” false
  - s2==“ab” true

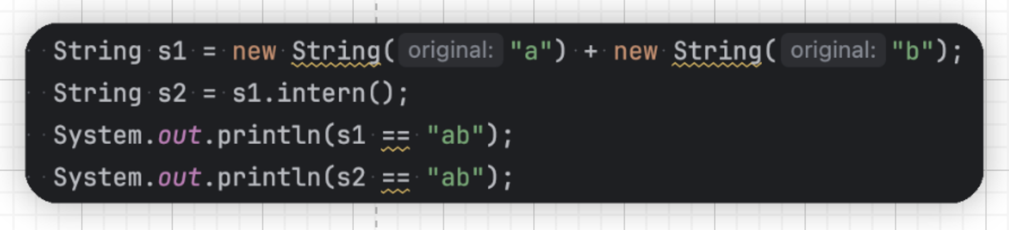


```
String s1 = new String( original: "a") + new String( original: "b");
String s2 = s1.intern();
System.out.println(s1 == "ab");
System.out.println(s2 == "ab");
```

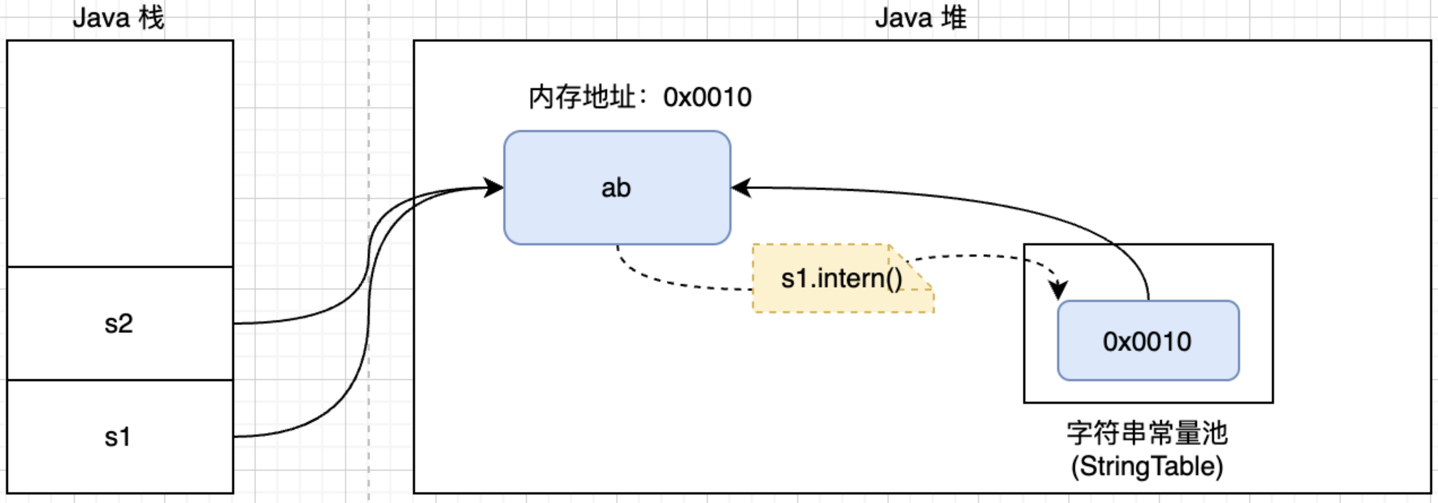
s1.intern()执行后，
会将 s1 复制一份到字符串常量池
并返回其在常量池的地址

拓展知识：String.intern

- Java 7/8
  - 字符串常量池在 Java 堆
- 答案
  - s1==“ab” true
  - s2==“ab” true



```
String s1 = new String( original: "a") + new String( original: "b");
String s2 = s1.intern();
System.out.println(s1 == "ab");
System.out.println(s2 == "ab");
```

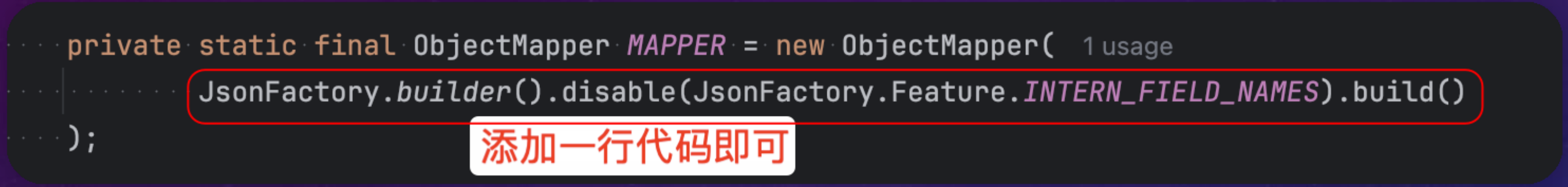


修复方案：？

思考和分享：怎么修复?

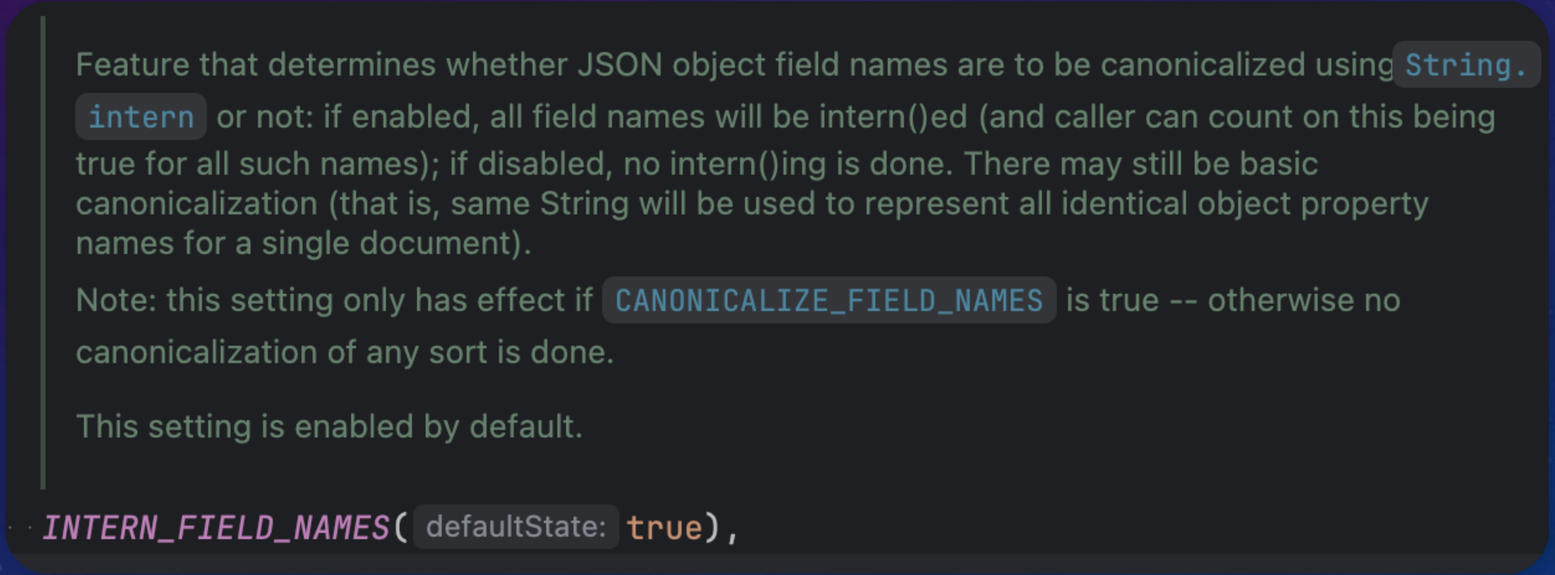
修复方案：禁用 String.intern

修复方案：禁用 Jackson 的 String.intern 功能即可



```
    private static final ObjectMapper MAPPER = new ObjectMapper(  1 usage
            JsonFactory.builder().disable(JsonFactory.Feature.INTERN_FIELD_NAMES).build()
    );
```

添加一行代码即可



Feature that determines whether JSON object field names are to be canonicalized using `String.intern` or not: if enabled, all field names will be intern()ed (and caller can count on this being true for all such names); if disabled, no intern()ing is done. There may still be basic canonicalization (that is, same String will be used to represent all identical object property names for a single document).

Note: this setting only has effect if `CANONICALIZE_FIELD_NAMES` is true -- otherwise no canonicalization of any sort is done.

This setting is enabled by default.

```
INTERN_FIELD_NAMES( defaultState: true),
```

总结&经验教训

- 遇到问题，首先想止损方案：
  - 如果有变更，尝试回滚
  - 如果扩容可行，扩容
  - 临时增加监控策略，人工运维，优先保证服务可用性
  - …
- 问题根因：
  - 业务方将类随机字符串用作了 JSON 的属性名，导致 StringTable 膨胀，影响了 YGC
- 问题特征：
  - 量变引起质变，需要足够长的时间，足够多的数据，才能暴露问题
  - 非常隐蔽，伏笔早就埋下，只在调用方改变请求参数时才发生

总结&经验教训

- 排查过程：
  - 尝试录制线上流量进行回放复现问题，失败
  - 分析了 GC 日志，需要熟悉 GC 日志的格式
  - 使用 jmap dump 内存并避免 FullGC
  - 使用 Eclipse Memory Analyzer 对内存dump 进行了分析
  - 查看了 G1 的源码，获得了关键线索

Q&A