

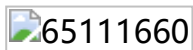
1. 相关知识点;
2. 类不要频繁的加载卸载, 否则可能导致full-gc, 可使用缓存技术。

[8月9日API线上服务频繁fullgc排查](#)

目录

## 一 问题现象

9日中午11点半左右, gh01节点频繁fullgc报警, 间隔1小时左右yf-waimai-sc-api06, yf-waimai-sc-api08也开始报警相同的问题。



65111660

## 二 线上影响

由于非大面积机器爆发, 线上先后4个节点出现问题都已经禁用, 线上服务正常, 晚上重新上线后出问题节点恢复, 暂无实质性影响。

## 三 问题处理

### 线上处理

1. 由于当时午高峰, 出问题的机器只是个别几个节点, 没有出现大面积出问题状况, 故在octo上先禁用了几个出问题的几点
2. 由于上次发版是在8月7号, 故判断重启机器可暂时解决问题
3. 在确认原因前下午准备增大MaxMetaspaceSize阈值为512M重新发版上线
4. 登陆问题主机, `jmap -dump:format=b,file=heap1.hprof pid`, dump堆内存进行分析。

问题分析

### 问题分析

#### 1. 引起fullgc的原因

什么引起了fullgc?

- 登陆问题主机运行`jstat -gc pid 1000`命令显示老年代使用率占比很小, Metaspace使用率占比很大, 故怀疑是Metaspace使用率超过阈值引起的fullgc,

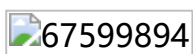
运行jstat -gccause 命令，确实和Metaspace有关。（由于排查问题时较紧急没有保留截图）

Metaspace当时的情况？

- 线上服务设置的最大-XX:MaxMetaspaceSize=256m，当时已经使用了234M左右(具体不记得了)

是否是Metaspace设置过小引起？

- -XX:MaxMetaspaceSize=256m参数在线上运行跑了几个月没有出现过问题
- 看classloading.totalloaded.count发现确实在7日发版前后有较大变化，原来基本是水平比较平缓，发版后有一定斜率上升，故怀疑是上次发版新增内容引起。
  - ClassLoadingMXBean工具类可查看总加载、当前加载、卸载的类数量统计；
  - jstat -class jvmlid：可查看加载类数量、大小，卸载类数量大小；



67599894

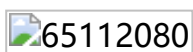
## 2. 是否和上次发版有关

st环境8月3日发版和prod环境新增了什么特性？

- 品牌折叠—闪购列表页
  - 引入了新实验平台，引入新jar，闪购列表页调用使用实验平台相关功能，故此是重点怀疑对象

是否只是线上节点有这个问题？

- 排查了st环境set-gh-waimai-sc-api-staging02节点，即使在大部分情况这个节点是禁用状态发现其也有问题（st机器发版较早，早8月3日测试发版），故怀疑此问题和请求量无关，怀疑和定时任务有关。如下图：



65112080

应用跑着什么定时任务？

- 新引入的实验平台定时任务：**定期拉取实验配置，更新本地实验配置，用于本地分流计算。ExperimentCacheService，每1分钟所有实验场景都调用AviatorEvaluator.compile(group.getExpressions())**如代码所示（注意标红位置）：

- ExperimentCacheService

```
/**
 * Created by fenghezhao on 16/12/20.
 */
@Service
public class ExperimentCacheService implements InitializingBean {
    @Value("${experiment.scene.keys}")
    private String sceneKey;
    @Autowired
    private ExperimentConfigService experimentConfigService;
    @Autowired
    private ExpEngine engine;
    private Set<String> sceneKeys = Sets.newConcurrentHashSet();
    // 轮询
    private ScheduledExecutorService refreshExecutorService =
Executors.newScheduledThreadPool(1);
    // 轮询间隔
    private static final int REFRESH_PERIOD = 1; // minutes
    @Override
    public void afterPropertiesSet() {
        //注册版本比较方法
        AviatorEvaluator.addFunction(new VersionCompareFunction());
        initSceneByProperties();
        // 初始化轮询 -- 全量
        refreshExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                try {
                    pull();
                } catch (Exception e) {
                    logger.error("pull failed!", e);
                }
            }
        }, 0, REFRESH_PERIOD, TimeUnit.MINUTES);
    }
}
```

```

        }
    }
    }, 1, REFRESH_PERIOD, TimeUnit.MINUTES);
}

public void pull() throws Exception {
    for (String key : sceneKeys) {
        //拉取分流实验配置
        pullExpConfig(key);
    }
}

private void pullExpConfig(String sceneKey) {
    WmExperimentConfigRes wmExperimentConfigRes =
experimentConfigService.getExperimentConfigBySceneKey(sceneKey);
    if (wmExperimentConfigRes != null) {
        if (wmExperimentConfigRes.getStatus() == 1) {
            ExpScene expScene =
JSON.parseObject(wmExperimentConfigRes.getConfigJson(), ExpScene.class);
            compileExpression(expScene);
            engine.sceneCache.put(sceneKey, expScene);
        } else if (wmExperimentConfigRes.getStatus() == -1) {
            engine.sceneCache.remove(sceneKey);
        }
    }
}

private void compileExpression(ExpScene scene) {
    if (scene == null) {
        return;
    }
    if (StringUtils.isNotBlank(scene.getExpressions())) {
        Expression sceneExpressions =
AviatorEvaluator.compile(scene.getExpressions());
        scene.setCompiledExpression(sceneExpressions);
    }
    if (CollectionUtils.isEmpty(scene.getGroups())) {
        return;
    }
}

```

```

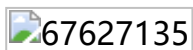
    }
    for (ExpScene.Group group : scene.getGroups()) {
        if (group == null ||
StringUtils.isBlank(group.getExpressions())) {
            continue;
        }

        group.setCompiledExpression(AviatorEvaluator.compile(group.getExpressions()));
    }
}
}

```

没有处理请求的st环境set-gh-waimai-sc-api-staging02节点都在加载着什么class?

- jvm启动参数增加**-verbose:class**参数，如下图所示：



67627135

- aviator是新实验平台引入的轻量级的表达式处理引擎，如上加载的class也指向了是新实验平台引入的问题。

## 问题复现

### 本地写测试代码复现

- 写了如下代码，配置jvm参数：**-XX:MaxMetaspaceSize=10m**（元空间大小）**-XX:+DisableExplicitGC**（禁止显示调用jvm进行gc）**-XX:+PrintGCDetails**（gc详情）**-Xmx30M**（最大堆值）**-verbose:class**，复现了问题：

点击展开内容

```

package com.sankuai.meituan.waimai.sc.api;

import java.util.HashMap;

import java.util.Map;

import com.googlecode.aviator.AviatorEvaluator;

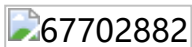
import com.googlecode.aviator.Expression;

```

```

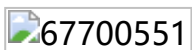
public class Test {
    public static void main(String[] args) throws InterruptedException {
        for (long i = 0; i < Long.MAX_VALUE; i++) {
            String expression = "(a-(b-c)%17)*b/c>100";
            // 编译表达式
            Expression compiledExp = AviatorEvaluator.compile(expression, true);
            Expression compiledExp = AviatorEvaluator.compile(expression);
            Map<String, Object> env = new HashMap<String, Object>();
            env.put("a", 100.3);
            env.put("b", 45);
            env.put("c", 199.100);
            // 执行表达式
            Boolean result = (Boolean) compiledExp.execute(env);
        }
    }
}

```

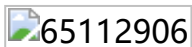


67702882

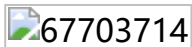
- 如下图所示，每次**AviatorEvaluator.compile**（新实验平台定时任务跑的代码），都会用**AviatorClassLoader.defineClass**方法把字节数组转换为class实例



67700551

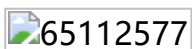


65112906



67703714

- 转换的class实例命名为Script\_xxxxx\_xxxx，如下图：



65112577

综上所述，问题基本原因基本明朗了，定时任务每分钟跑N次的

AviatorEvaluator.compile方法（N是实验场景个数），N次调用都会引起

AviatorClassLoader.defineClass方法把字节数组转换为class实例，导致class实例增长，Metaspace使用率增长引起fullgc。

## 四、问题解决方式

1. 增大MaxMetaspaceSize，也就是9日在没有查到根本问题时，上线采取的方式。

此方式治标不治本

2. 应用方去除新版实验平台，改为老版本实验平台。

a. 处理方式

i. 更改代码，周知相关人员重新在老版本实验平台配置此次上线的实验场景

b. 优缺点

i. 新版实验平台采取的是定时任务拉取实验配置然后本地进行分流计算，而不是像老版本采用RPC的方式，性能上的优化还是很可观的

ii. 切换为旧版需要重新开发，重新配置，且性能不如新版

3. 通知新版实验平台相关RD解决bug及相关建议（**todo: 解决方案；表达式对应一个类？**）

a. 表达式一般配置后不会改变，可以采取Aviator提供缓存的缓存计算表达式的方式，这样相同的表达式只会转换为class一次，代码如下：

```
i. Expression compiledExp =  
    AviatorEvaluator.compile(expression, true);  
  
/**  
    * Compile a text expression to Expression  
    object
```

```

*
* @param expression text expression
* @param cached Whether to cache the
compiled result, make true to cache it.
* @return
*/
public static Expression compile(final
String expression, final boolean cached) {
    if (expression == null ||
expression.trim().length() == 0) {
        throw new
CompileExpressionErrorException("Blank
expression");
    }

    if (cached) {
        FutureTask<Expression> task =
cacheExpressions.get(expression);
        if (task != null) {
            return
getCompiledExpression(expression, task);
        }

        task = new FutureTask<Expression>(new
Callable<Expression>() {
            @Override
            public Expression call() throws
Exception {

```



```

        return innerCompile(expression,
cached);
    }

    });

    FutureTask<Expression> existedTask =
cacheExpressions.putIfAbsent(expression,
task);

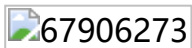
    if (existedTask == null) {
        existedTask = task;
        existedTask.run();
    }

    return getCompiledExpression(expression,
existedTask);

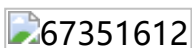
} else {
    return innerCompile(expression, cached);
}

}

```



67906273



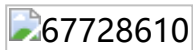
67351612

b. 每分钟同步1次实验配置还是过于频繁，实验场景如果很多的话很占服务资源

- i. 建议更改大默认定时任务同步频次，实验配置延迟15分钟左右生效个人觉得都是可接受的
- ii. 建议把定时任务同步频次抽出配置，开发给应用方，应用方根据自身使用场景配置阈值**
- iii. 建议升级技术方案，定时任务全量同步改为监听变更增量同步的方式**

## 五、其他疑问

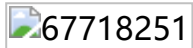
dump文件看Script\_1534067462693\_762 这种class很少，是因为在GC后Unloading class了



67728610

看最后的Metaspace used 4766K, capacity 5425K, 其中class space used 476K, capacity 512K

不太理解class 都卸载了为什么class space capacity只有 512K。。。Metaspace 的 capacity 5425K 但是used 4766K



67718251