背景

公司某个规则引擎系统，主要用来做一些费用计算和业务逻辑核验的功能。不过由于一些**不可描述的历史原因**，导致该系统**没有文档也没有源码**，就连配置方式都是靠口口相传。  
​

  
​

虽然这个系统比较老，但毕竟是商业产品，功能上还是比较完善好用的。该系统在接入业务系统的算费和核验规则后，很快就上线了。  
​

不过上线后，会偶发的出现服务耗时过长的问题。正常情况下，该服务响应耗时也就 **20ms** 以下。但出现问题时，服务的耗时会**增加到几十秒**，每次出现长耗时的时间点也不固定。而且在长耗时期间，所有到达该服务的请求都会出现长耗时，并不只是个别请求才会受影响。  
​

理论上这种问题测试之类的环境应该也有；但由于测试环境重启很频繁，偶发的长耗时可能认为是在重启，就一直没人留意。  
​

虽然啥资料都没有，但问题还是得解决啊，时不时的几十秒耗时谁能顶得住。对接这规则引擎的还是业务的核心系统，超时一分钟得少挣多少保费……

抱着试试看的心态，先研究一下。

排查经过

这种偶发长耗时的问题，排查起来是比较麻烦的。无法稳定复现，也没有规律，就算弄个其他环境模拟也不是很好办；再加上这个系统没文档没源码，更没人懂它的结构和流程，查起来就更费劲了。

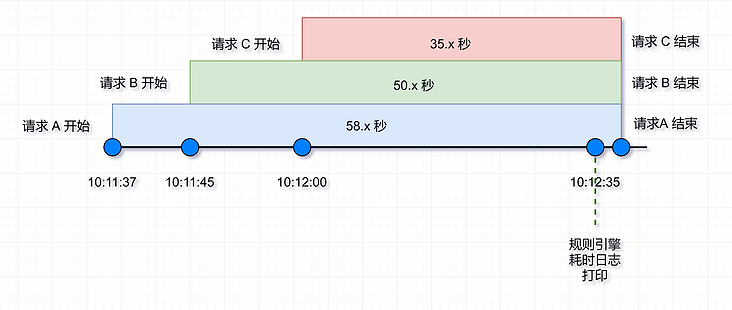
看监控

在查看了监控历史之后，发现每次出现长耗时的时候，CPU & 磁盘 IO 的利用率会升高，但也没高的很离谱。CPU 高的时候不过也才 五六十，而磁盘就更低了，只是稍微有一点波动而已，这点波动会带来几十秒的耗时，可有点说不过去。

看日志

CPU 那块只是稍微有点高，但也不能说明什么问题，还是得找到根本原因。于是我又翻了下那个系统的日志，看看能不能找到什么线索。  
  
卡时间，看日志，折腾了半小时，终于在日志上看到一个关键线索：

2021-05-08 10:12:35.897 INFO **[com.马赛克.rules.res.execution] (default-threads - 65) 规则集 /ansNcreckonRuleApp/1.0/ansncreckonrule/61.0 已在 58 秒后解析完毕。**

58 秒……这行日志的时间点，和我们实际发生耗时的时间点是可以匹配上的，在这个 58 秒范围内，确实有大量的请求耗时，而且都是小于 58 秒或略大于 58 秒的。  
  
除了这个耗时的打印之外，GC 日志也有点可疑。**每次出现这个耗时日志之前，都会有一次 GC 活动日志**，不过暂停时间并不长：

2021-06-07T17:36:44.755+0800:**[GC2021-06-07T17:36:44.756+0800: [DefNew: 879616K->17034K(3531079K), 0.0472190 secs] 3549012K->2686430K(10467840K), 0.0473160 secs] [Times: user=0.05 sys=0.00, real=0.2 secs]**

**更奇怪的是……不是每次 GC 之后，都会有耗时日志**

GC 问题

仔细查看 gc 日志后发现，每次慢响应之前不久，都会有一次 GC，不过不是 FULL GC，而且每次暂停时间也不长，远不及请求的耗时时间。而且 GC 过程中的 CPU 利用率并不高，从数据上看还是比较正常的。  
​

好现在有（一丁点）线索了：

1. 长耗时期间，CPU 利用率有增加
2. 长耗时期间有日志打印，在加载某个东西，时长是可以匹配的
3. 每次长耗时之前，有 GC 活动

猜测可能的原因

毕竟没有源码，也没人懂，想手撕代码都没机会，只能靠猜了……  
​

从以上几个线索来看，GC 活动之后出现 CPU 利用率增加，然后打印了一行加载日志，时间还可以和该系统的长耗时请求对应上。  
​

而且每次长耗时日志之前不久的地方，都会有一次 GC 活动，那么说明这个资源加载的时机和 GC 有关系，GC 会影响资源加载……  
​

想到这里，也大概猜到原因了。很可能是用弱引用(Weak Reference)来维护了这个资源缓存，当 GC 后弱引用的资源被回收，所以需要重新加载（弱引用的详细解释以及测试结果可以参考[《Java 中的强引用/软引用/弱引用/虚引用以及 GC 策略》](https://segmentfault.com/a/1190000020602887)）。

那既然可能是这个原因，如果我找到这个弱引用维护资源缓存的地方，给他改成强引用就能解决问题了！虽然没有源码，但还是可以反编译啊，反编译之后改一下缓存那块的代码，问题不大。  


定位资源加载点

终于找到了可能的原因，可是有一个问题……我怎么知道这行耗时日志在哪打印的？在哪个类里？我连这个服务是用的什么 Web 容器都不知道。

没办法，上 [Arthas](https://arthas.aliyun.com/zh-cn/index.html) 吧，不过肯定不能在生产环境直接测。于是我又新整了一套临时测试环境，用于排查这个问题。  
​

幸好日志里有个 loggerName 前缀com.马赛克.rules.res.execution，通过 [Arthas](https://arthas.aliyun.com/zh-cn/index.html) 的 trace 功能，可以用通配符的形式来 trace 这个包名下的所有类：

*# trace 该包名下的所有类，所有方法，只显示耗时大于 1000ms 的方法*

*# 由于这个输出结果可能会比较多，所以 > 输出到文件，并且后台运行*

trace com.马赛克.rules.res.\* \* '#cost > 1000' > /app/slow\_trace.log &

trace 命令执行了十几秒才返回，一共影响了 169 个类和 1617 个方法，可见通配符匹配多危险……要是生产这样玩我可能会被拉出去祭天。  


结合前面描述的情况，GC 后会有这个耗时问题，那现在来手动触发一次 GC，  
​

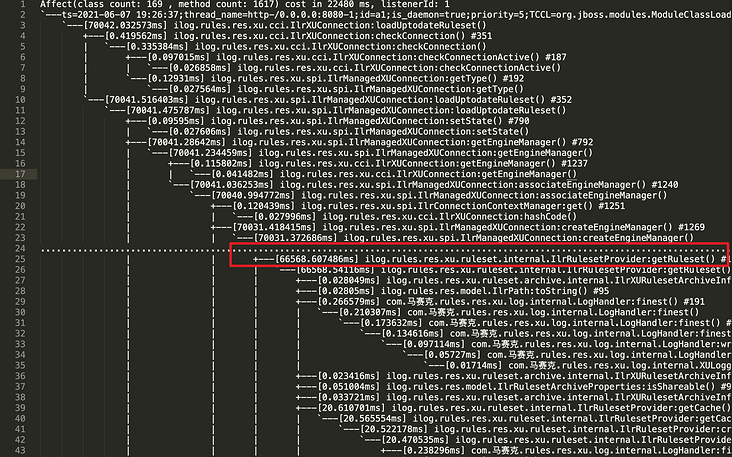
利用 Arthas 的 vmtool 可以直接 forceGC（利用 jmap 或者其他的手段也可以）：

vmtool --action forceGc

forceGC 后，重新测试该系统接口  
​

和上面的情况一样，果然又出现了长耗时，请求返回后打印了相同的耗时日志，只是和生产环境的相比时间更长了（因为我用了 arthas trace 增强）

2020-04-08 12:30:35.897 INFO [com.马赛克.rules.res.execution] (default-threads - 65) 规则集 /ansNcreckonRuleApp/1.0/ansncreckonrule/61.0 已在 70 秒后解析完毕。

同时 Arthas trace 日志写入的那个文件中，也有了内容（链路实在太长，没法贴代码，只能截图了）：  
  
这个 trace 日志一千多行，此处删减了部分内容。  
​

Arthas 的 trace 命令不会进行全链路跟踪，所以这里的**一次 trace 结果**只是当前路径。但已经不错了，问题范围又再一次缩小。  
​

从上图可以看到IlrRulesetProvider:getRuleset这个方法是主要耗时点，那就先来看看这个方法里是个什么玩法。  
​

为了简单，这里临时用 Arthas 的 jad 命令，直接反编译这个类，看看里面的逻辑：

jad ilog.rules.res.xu.ruleset.internal.IlrRulesetProvider

代码有点多，这里删减一些，只保留关键部分：

**public** **final** XURulesetImpl **getRuleset**(IlrXURulesetArchiveInformation archive, IlrXUContext xuCtx, **boolean** waitParsing, Listener listener) **throws** IlrRulesetArchiveInformationNotFoundException, IlrRulesetCreationException, IlrRulesetAlreadyParsingException, IlrRulesetCacheException, XUException {

String canonicalPath = archive.getCanonicalPath().toString();

**this**.logger.finest("IlrRulesetProvider.getRuleset " + canonicalPath + " " + waitParsing);

**if** (!archive.getProperties().isShareable()) {

**return** **this**.factory.createRuleset(archive, xuCtx);

} **else** {

ClassLoader cl = archive.getXOMClassLoader();

XURulesetImpl ruleset;

**while**(**true**) {

**synchronized**(**this**.parsingRulesets) {

*// 字面意思是，从缓存中获取规则集，有的话直接 return 了*

ruleset = (XURulesetImpl)**this**.getCache().getRuleset(canonicalPath, cl);

**if** (ruleset != **null**) {

**return** ruleset;

}

*// 第一个加载的线程，将当前资源添加到 parsingRulesets 同时跳出 while*

**if** (!**this**.parsingRulesets.contains(archive)) {

**this**.parsingRulesets.add(archive);

**break**;

}

**if** (!waitParsing) {

**throw** **new** IlrRulesetAlreadyParsingException("XU.ERROR.10406", (String[])**null**);

}

*// 这里的 wait……应该是防止并发访问，当其他线程也进入该代码块时 wait 等待第一个线程加载完成唤醒*

**try** {

**this**.parsingRulesets.wait();

} **catch** (InterruptedException var20) {

**throw** **new** IlrRulesetCreationException("XU.ERROR.10009", (String[])**null**, var20);

}

}

}

**if** (!**this**.useWorkManager(archive)) {

**this**.logger.finest("IlrRulesetProvider.getRuleset doesn't use the workmanager " + **this**.workManager, (Object[])**null**, xuCtx);

XURulesetImpl var9;

**try** {

*// 创建资源*

ruleset = **this**.factory.createRuleset(archive, xuCtx);

*// 创建完成，添加到缓存*

**this**.getCache().addRuleset(ruleset);

var9 = ruleset;

} **finally** {

**this**.parsingStopped(archive);

}

**return** var9;

}

}

}

虽然删减了很多，但看代码还是有点不清晰，毕竟反编译的代码阅读干扰有点大，这里简单解释下上面的代码逻辑：

1. 先从缓存容器中获取资源
2. 如果取不到就执行创建逻辑
3. 创建完成，再次添加到缓存
4. 在加载时若已有其他线程也执行加载，会主动 wait 等待第一个创建完成的线程唤醒

再结合我们上面的猜测：

很可能是该系统用弱引用(Weak Reference)来维护了这个资源缓存，当 GC 后该资源被回收，所以需要重新加载

那问题就在这个缓存容器上了！只要看看这个 cache 就能知道，肯定有弱引用的代码！  


反编译找代码

jad 反编译在终端里看代码还是太折腾了，不如在 IDE 里直观，所以还是得把代码拉下来分析，不然找个关联类都费劲。  
​

这个系统下有很多 Jar 包，得先找到这些相关的类在哪个 Jar 里，利用 Arthas 的 sc 命令，也非常简单：

sc 查看JVM已加载的类信息

sc -d ilog.rules.res.xu.ruleset.internal.IlrRulesetProvider

**class**-**info** ilog.rules.res.xu.ruleset.internal.IlrRulesetProvider

# 这里是我们关心的信息，该 **class** 所在的 jar

code-source /content/jrules-res-xu-JBOSS61EAP.rar/ra.jar

**name** ilog.rules.res.xu.ruleset.internal.IlrRulesetProvider

isInterface **false**

isEnum **false**

isAnonymousClass **false**

isArray **false**

isLocalClass **false**

isMemberClass **false**

isPrimitive **false**

isSynthetic **false**

simple-**name** IlrRulesetProvider

modifier final,public

annotation

interfaces com.ibm.rules.res.xu.ruleset.internal.RulesetParsingWork$Listener

super-**class** +-java.lang.**Object**

**class**-loader +-ModuleClassLoader **for** Module "deployment.jrules-res-xu-JBOSS61EAP.rar:main" **from** Service Module Loader

+-sun.misc.Launcher$AppClassLoader@2d74e4b3

+-sun.misc.Launcher$ExtClassLoader@5552bb15

classLoaderHash 2d022d73

Affect(**row**-cnt:1) **cost** **in** 79 ms.

找到这个 ra.jar 后，把这个 jar 拖到 IDE 里反编译，不过这个 class 还有些关联的 class 不在这个 ra.jar 中。还是同样的办法，找到关联的 class，然后 sc -d 找到所在的 jar 位置，复制到本地 ide 反编译  
​

反复折腾了几次后，终于把有关联的 4 个 jar 包都弄回本地了，现在可以在 IDE 里开开心心的看代码了。

分析缓存容器的机制

首先是上面那个 createCache 方法，经过分析后得知，cache 的实现类为 IlrRulesetCacheImpl，这个类需要关心的只有两个方法，getRuleset 和 addRuleset：

**public** **void** **addRuleset**(IlrXURuleset executableRuleset) {

**synchronized**(**this**.syncObject) {

*//...*

**this**.entries.add(**new** IlrRulesetCacheEntry(executableRuleset, **this**.maxIdleTimeOutSupport));

*//...*

}

}

**public** IlrXURuleset **getRuleset**(String canonicalRulesetPath, ClassLoader xomClassLoader) {

*// ...*

List<IlrRulesetCacheEntry> cache = **this**.entries;

**synchronized**(**this**.syncObject) {

Iterator iterator = cache.iterator();

**while**(iterator.hasNext()) {

IlrRulesetCacheEntry entry = (IlrRulesetCacheEntry)iterator.next();

IlrXURuleset ruleset = (IlrXURuleset)entry.rulesetReference.get();

**if** (ruleset == **null**) {

iterator.remove();

} **else** **if** (entry.canonicalRulesetPath.equals(canonicalRulesetPath) && (entry.xomClassLoader == xomClassLoader || entry.xomClassLoader != **null** && entry.xomClassLoader.getParent() == xomClassLoader)) {

**return** ruleset;

}

}

}

*// ...*

**return** **null**;

}

看完这两个方法之后，很明显了， entries 才是关键的数据存储集合，看看它是怎么个玩法：

**protected** **transient** List<IlrRulesetCacheEntry> entries = **new** ArrayList();

竟然只是个 ArrayList，继续看看 IlrRulesetCacheEntry这个类：

**public** **class** **IlrRulesetCacheEntry** {

**protected** String canonicalRulesetPath = **null**;

**protected** ClassLoader xomClassLoader = **null**;

**protected** IlrReference<IlrXURuleset> rulesetReference = **null**;

**public** **IlrRulesetCacheEntry**(IlrXURuleset executableRuleset, **boolean** maxIdleTimeOutSupport) {

**this**.canonicalRulesetPath = executableRuleset.getCanonicalRulesetPath();

**this**.xomClassLoader = executableRuleset.getXOMClassLoader();

**long** maxIdleTime = executableRuleset.getRulesetArchiveProperties().getMaxIdleTime();

*// 注意这里是关键，根据 maxIdleTime 的值选择强引用和弱引用*

**if** (maxIdleTime != 0L && (!maxIdleTimeOutSupport || maxIdleTime == -1L)) {

**this**.rulesetReference = **new** IlrWeakReference(executableRuleset);

} **else** {

**this**.rulesetReference = **new** IlrStrongReference(executableRuleset);

}

}

}

代码已经很直白了，根据 maxIdleTime 的不同使用不同的引用策略，不等于 0 就弱引用，等于 0 就强引用；不过还是得看下这俩 Reference 类的代码：

*// 弱引用，继承 WeakReference*

**public** **class** **IlrWeakReference**<**T**> **extends** **WeakReference**<**T**> **implements** **IlrReference**<**T**> {

**public** **IlrWeakReference**(T t) {

**super**(t);

}

}

*// 强引用*

**public** **class** **IlrStrongReference**<**T**> **implements** **IlrReference**<**T**> {

**private** T target;

IlrStrongReference(T target) {

**this**.target = target;

}

**public** T **get**() {

**return** **this**.target;

}

}

这俩类并没有什么特别的地方，和类名的意思相同；IlrWeakReference 继承于 WeakReference，那就是弱引用，当发生 GC 时，引用的对象会被删除。  
​

虽然找到了这个弱引用的地方，但还是需要验证一下，是不是真的使用了这个弱引用

验证是否使用了弱引用

这里使用 Arthas 的 vmtool 命令，来看看缓存中的实时对象：

vmtool --action getInstances --className ilog.rules.res.xu.ruleset.cache.**internal**.IlrRulesetCacheImpl --express 'instances[0].entries.get(0)'

**@IlrRulesetCacheEntry**[

canonicalRulesetPath=**@String**[/ansNcreckonRuleApp/1.0/ansncreckonrule/1.0],

xomClassLoader=**@XOMClassLoader**[com.ibm.rules.res.persistence.**internal**.XOMClassLoader@18794875],

# 这里可以看到，rulesetRef 的实例是 IlrWeakReference

rulesetReference=**@IlrWeakReference**[ilog.rules.res.xu.ruleset.cache.**internal**.IlrWeakReference@dbd2972],

]

从结果上看，石锤了就是弱引用。



但引起弱引用的毕竟是 maxIdleTime，还是需要找到 maxIdleTime 的源头……

寻找 maxIdleTime

在 IlrRulesetCacheEntry 的构造方法里可以看到，maxIdleTime 是从 IlrRulesetArchiveProperties 里获取的：

**long** maxIdleTime = executableRuleset.getRulesetArchiveProperties().getMaxIdleTime();

那就继续看看 IlrRulesetArchiveProperties这个类：

**public** **long** **getMaxIdleTime**() {

*// 从 properties 里获取 key 为 ruleset.maxIdleTime 的 value*

String result = **this**.get("ruleset.maxIdleTime");

**return** result == **null** ? -1L : Long.valueOf(result);

}

**public** String **get**(Object key) {

String result = (String)**this**.properties.get(key);

**return** result == **null** && **this**.defaults != **null** ? (String)**this**.defaults.get(key) : result;

}

getMaxIdleTime 返回的默认值是 -1，也就是说如果没配置这个 maxIdleTime 值，默认也会使用弱引用策略。  
​

到目前为止，问题算是已经精确的定位到了，弱引用的缓存策略导致被 GC 时资源缓存被清空，重新加载资源导致了长耗时。  
​

可是这系统没源码没文档，我上哪改这个 maxIdleTime 去……  
​

不过来都来了，都已经看到 IlrRulesetArchiveProperties 这个类了，不如先看看这个类里到底配置了哪些值，有没有 maxIdleTime

vmtool --action getInstances --className com.ibm.rules.res.xu.ruleset.internal.CRERulesetImpl --express 'instances[0].getRulesetArchiveProperties()'

@IlrRulesetArchivePropertiesImpl[

@String[ruleset.engine]:@String[cre],

@String[ruleset.status]:@String[enabled],

@String[ruleset.bom.enabled]:@String[**true**],

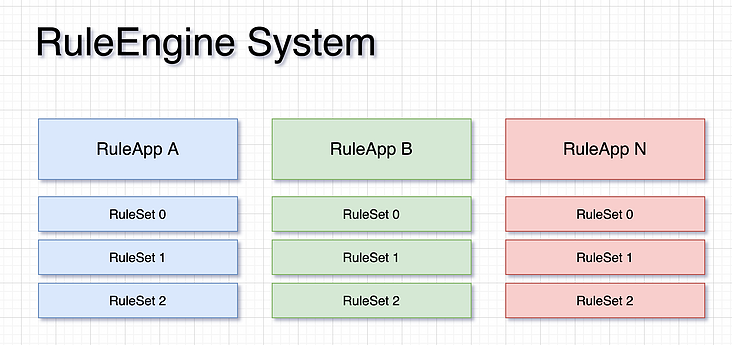
@String[ruleset.managedxom.uris]:@String[resuri://ans-nc-xom.zip/54.0,resuri://ruleapp-core-model-1.5.2.jar/2.0],

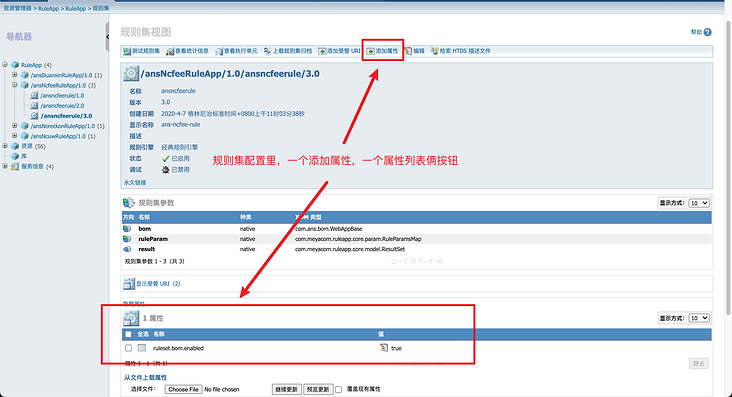
]

从结果上可以看到，properties 里并没有 maxIdleTime 属性，和我们上面的结论是可以匹配的。没有配置 maxIdleTime 属性，默认 -1，所以使用弱引用

寻找 maxIdleTime 的配置方法

反编译的代码虽然看不到注释，但从类名还是可以猜一下的，IlrRulesetArchiveProperties 这个类名应该是“规则集归档属性”的意思。  
​

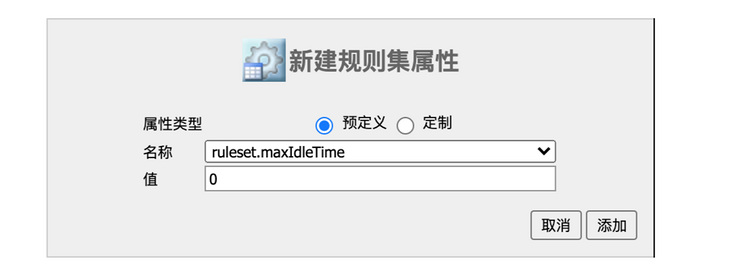
虽然我不懂这个系统的业务规则，但有人懂啊！于是我找来了负责这个系统配置的老哥，找他给我解释了下这个系统的各种概念。  
  
老哥也很友好，直接给我画了一张图  
  
这个规则引擎系统，有一个 App 的概念，就是 saas 平台里常说的那个应用的意思；每个 App 下面可以创建多个规则集，就是图上这个 RuleSet，每个 RuleSet 就是我们的业务规则，比如费用计算公式或者逻辑核验规则，同时每个 RuleSet 还会记录多个每次变更的历史版本。  
​

看到这张图，我也明白了个七七八八，上面定位的 IlrRulesetArchiveProperties 不就是规则集的属性？这个产品有个控制台，应该有这个属性配置的地方吧，不然弄这么个类干啥？规则集都是控制台创建的，那规则集属性应该也可以配置！  


如我所料，控制台的规则集上果然有个属性的概念，而且还可以添加属性。  
​

最重要的是，控制台上ruleset.bom.enabled这个属性，我上面用 arthas vmtool 看的时候也有，那就可以证明这里的配置，应该就是配置 IlrRulesetArchiveProperties 这个类的；只要是给这个规则集加上 maxIdleTime 是不是就可以使用强引用了？  
​

然后我小心翼翼的点击了那个添加属性的按钮……  
​

  
果然有 maxIdleTime 这个选项，选择 maxIdleTime 之后，给它配置为 0。

验证结果

配置完了，还是重启验证一下是不是生效，谁知道这个系统支不支持热刷新呢，改动之后还是重启验证比较稳妥。  
​

先验证该规则集的 properties：

vmtool --action getInstances --className com.ibm.rules.res.xu.ruleset.internal.CRERulesetImpl --express 'instances[0].getRulesetArchiveProperties()'

@IlrRulesetArchivePropertiesImpl[

*# 新增配置*

@String[ruleset.maxIdleTime]:@String[0],

@String[ruleset.engine]:@String[cre],

@String[ruleset.status]:@String[enabled],

@String[ruleset.bom.enabled]:@String[**true**],

@String[ruleset.managedxom.uris]:@String[resuri://ans-nc-xom.zip/54.0,resuri://ruleapp-core-model-1.5.2.jar/2.0],

]

从上面可以看到，我们新增的配置，已经生效了，规则集上已经有了这个 maxIdleTime。  
​

再来看看缓存里的引用，是不是已经变成了强引用：

vmtool --action getInstances --className ilog.rules.res.xu.ruleset.cache.internal.IlrRulesetCacheImpl --express 'instances[0].entries.get(0).rulesetReference'

*# 这里是强引用*

@IlrStrongReference[

target=@CRERulesetImpl[com.ibm.rules.res.xu.ruleset.internal.CRERulesetImpl@28160472],

]

在增加了 maxIdleTime 之后，规则集的缓存就变成了强引用，强引用下就不会再出现因为 GC 被回收的情况了！  
​

用 vmtool 来一遍 fullgc，试试看还会不会重新加载：

vmtool *--action forceGc*

执行了十几遍，也没出出现长耗时问题，问题应该是解决了。  
​

接着把这个配置同步到测试环境，跑了三天，我时不时还上去手动 forceGc 一下，没有再出现过这个长耗时的问题了。

为什么不是每次 GC 后都会出现耗时

本文开头就提到，每次长耗时之前不久都会有一次 GC 活动，但并不是每次 GC 后都会有一次长耗时。  
​

弱引用维护的对象，并不是说在 GC 时就会被清空；只是在 GC 时，**如果弱引用的对象已经没有其他引用了，才会被回收**，比如下面这个例子里：

Map<String,Object> dataMap = **new** HashMap<>();

WeakReference ref = **new** WeakReference(dataMap);

System.gc();

System.out.println(ref.get());

dataMap = **null**;

System.gc();

System.out.println(ref.get());

*//output*

{}

**null**

第一次 gc 时，ref 里的数据不会被清空，而第二次 gc 前弱引用的数据，已经没有其他任何引用了，此时会被清空。  
​

结合这个系统的问题来看，虽然规则集那里使用弱引用缓存，但如果在 GC 时，调用方还持有规则集对象没有释放，那么这个弱引用的规则集缓存也一样不会清空；所以才会出现这个**不是每次 GC 都会导致重新加载， 但每次重新加载却都是因为 GC 的问题**  
​

但我认为没有必要继续跟下去了，持有规则集对象的地方在哪，和这个耗时问题关系并不是很大；修改为强引用之后，就不会再有 GC 回收该对象的情况，那还在意谁持有干嘛呢（其实是我懒，没文档还没源码，找问题找的我头都要秃了）