

Linux CPU负载过高问题排查

1、排查思路

1.1 定位高负载进程

首先登录到服务器使用top命令确认服务器的具体情况，根据具体情况再进行分析判断。

```
load average: 17.94, 20.40, 21.05
```

通过观察load average，以及负载评判标准（8核），可以确认服务器存在负载较高的情况；

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
682	work	20	0	25.6g	20g	19m	S	785.9	66.2	5500:18	java
29705	root	20	0	3347m	219m	14m	S	1.3	0.7	300:30.35	java
4989	root	20	0	799m	12m	4376	S	0.7	0.0	226:18.90	falcon-agent
35	root	20	0	0	0	0	S	0.3	0.0	56:54.82	events/0
4510	work	20	0	15036	1284	932	R	0.3	0.0	0:00.26	top
4606	work	20	0	15132	1300	928	S	0.3	0.0	0:00.21	top
25772	mysql	20	0	2087m	193m	5984	S	0.3	0.6	1004:58	mysqld
1	root	20	0	19356	1072	736	S	0.0	0.0	0:33.48	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.15	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	16:04.88	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	19:01.27	ksoftirqd/0
5	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/0

CPU负载过高异常排查实践与总结CPU负载过高异常排查实践与总结
观察各个进程资源使用情况，可以看出进程id为682的进程，有着较高的CPU占比

1.2 定位具体的异常业务

这里咱们可以使用 pwdx 命令根据 pid 找到业务进程路径，进而定位到负责人和项目：

```
work@online_zeze_master 10.48.10.10 17:32:31
pwdx 682
682: /opt/web/zhuanzhuan_analysis
work@online_zeze_master 10.48.10.10 17:32:36
```

CPU负载过高异常排查实践与总结CPU负载过高异常排查实践与总结
可得出结论：该进程对应的就是数据平台的web服务。

1.3 定位异常线程及具体代码行

传统的方案一般是4步：

1. top oder by with P: 1040 // 首先按进程负载排序找到 maxLoad(pid)
2. top -Hp 进程PID: 1073 // 找到相关负载 线程PID
3. printf “0x%x\n”线程PID: 0x431 // 将线程PID转换为 16进制，为后面查找 jstack 日志做准备
4. jstack 进程PID | vim +/十六进制线程PID - // 例如：jstack 1040|vim +/0x431

但是对于线上问题定位来说，分秒必争，上面的4步还是太繁琐耗时了，之前介绍过[淘宝的oldratlee同学](#)就将上面的流程封装为了一个工具：`show-busy-java-threads.sh`，可以很方便的定位线上的这类问题：

```
[1] Busy(83.2%) thread(26817/0x68c1) stack of java process(26473) under user(work):
"Timer-8" daemon prio=10 tid=0x00007f7d91126800 nid=0x68c1 runnable [0x00007f7db46a9000]
  java.lang.Thread.State: RUNNABLE
    at java.text.SimpleDateFormat.initializeDefaultCentury(SimpleDateFormat.java:894)
    at java.text.SimpleDateFormat.initialize(SimpleDateFormat.java:672)
    at java.text.SimpleDateFormat.<init>(SimpleDateFormat.java:585)
    at java.text.SimpleDateFormat.<init>(SimpleDateFormat.java:568)
    at com.bj58.zhuanzhuan.analysis.util.TimestampUtil.timestampToDate(TimestampUtil.java:31)
    at com.bj58.zhuanzhuan.analysis.util.DateUtil.getTimesBySplitToday(DateUtil.java:745)
    at com.bj58.zhuanzhuan.analysis.service.impl.MonitorNewServiceImpl.queryNewMonitorDatasFromHbaseByDateOptim
(MonitorNewServiceImpl.java:817)
    at com.bj58.zhuanzhuan.analysis.service.impl.MonitorNewServiceImpl.queryNewMonitorDatasFromHbaseOptimize(Mon
orNewServiceImpl.java:848)
    at com.bj58.zhuanzhuan.analysis.thread.ScreenMonitorRunnable.screenMonitor(ScreenMonitorRunnable.java:128)
    at com.bj58.zhuanzhuan.analysis.util.TimeTaskUtils$.run(TimeTaskUtils.java:115)
    at java.util.TimerThread.mainLoop(Timer.java:555)
    at java.util.TimerThread.run(Timer.java:585)

[2] Busy(4.6%) thread(26483/0x6773) stack of java process(26473) under user(work):
"Gang worker#7 (Parallel GC Threads)" prio=10 tid=0x00007f7e18027800 nid=0x6773 runnable
```

CPU负载过高异常排查实践与总结CPU负载过高异常排查实践与总结
可得出结论：是系统中一个时间工具类方法的执行cpu占比较高，定位到具体方法后，查看代码逻辑是否存在性能问题。

※ 如果线上问题比较紧急，可以省略 2.1、2.2 直接执行 2.3，这里从多角度剖析只是为了给大家呈现一个完整的分析思路。

2、根因分析

经过前面的分析与排查，最终定位到一个时间工具类的问题，造成了服务器负载以及cpu使用率的过高。

- 异常方法逻辑：是把时间戳转成对应的具体的日期时间格式；
- 上层调用：计算当天凌晨至当前时间所有秒数，转化成对应的格式放入到set中返回结果；
- 逻辑层：对应的是数据平台实时报表的查询逻辑，实时报表会按照固定的时间间隔来，并且在一次查询中有多次（n次）方法调用。

那么可以得到结论，如果现在时间是当天上午10点，一次查询的计算次数就是 $106060n$ 次 = 36,000n 次计算，而且随着时间增长，越接近午夜单次查询次数会线性增加。由于实时查询、实时报警等模块大量的查询请求都需要多次调用该方法，导致了大量CPU资源的占用与浪费。

3、解决方案

定位到问题之后，首先考虑是要减少计算次数，优化异常方法。排查后发现，在逻辑层使用时，并没有使用该方法返回的set集合中的内容，而是简单的用set的size数值。确认逻辑后，通过新方法简化计算（当前秒数-当天凌晨的秒数），替换调用的方法，解决计算过多的问题。上线后观察服务器负载和cpu使用率，对比异常时间段下降了30倍，恢复至正常状态，至此该问题得已解决。

```
top - 22:05:16 up 623 days, 4:12, 4 users, load average: 0.64, 0.84, 0.91
Tasks: 188 total, 1 running, 187 sleeping, 0 stopped, 0 zombie
Cpu(s): 2.2%us, 0.5%sy, 0.0%ni, 97.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 32749608k total, 14721208k used, 18028400k free, 286612k buffers
Swap: 32767996k total, 0k used, 32767996k free, 8796912k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
11786	work	20	0	25.7g	3.7g	13m	S	22.5	12.0	141:05.94	java
29705	root	20	0	3477m	222m	12m	S	2.0	0.7	635:08.20	java
1081	work	20	0	15036	1284	928	R	0.3	0.0	0:00.06	top
4989	root	20	0	799m	12m	4380	S	0.3	0.0	276:37.45	falcon-agent
13579	zabbix	20	0	18016	1148	960	S	0.3	0.0	402:49.97	zabbix_agentd
1	root	20	0	19356	1052	736	S	0.0	0.0	0:38.83	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.15	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	16:26.41	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	19:17.25	ksoftirqd/0

CPU负载过高异常排查实践与总结CPU负载过高异常排查实践与总结

4、总结

- 在编码的过程中，除了要实现业务的逻辑，也要注重代码性能的优化。一个业务需求，能实现，和能实现的更高效、更优雅其实是两种截然不同的工程师能力和境界的体现，而后者也是工程师的核心竞争力。
- 在代码编写完成之后，多做 review，多思考是不是可以用更好的方式来实现。
- 线上问题不放过任何一个小细节！细节是魔鬼，技术的同学需要有刨根问题的求知欲和追求卓越的精神，只有这样，才能不断的成长和提升。

1人点赞

IT-Linux