


# Darwin Flink Batch 任务堆外内存泄露排查分析

Author:  王蒙

Team: 产品研发和工程架构部-架构-计算-流式计算

Scope: 域内

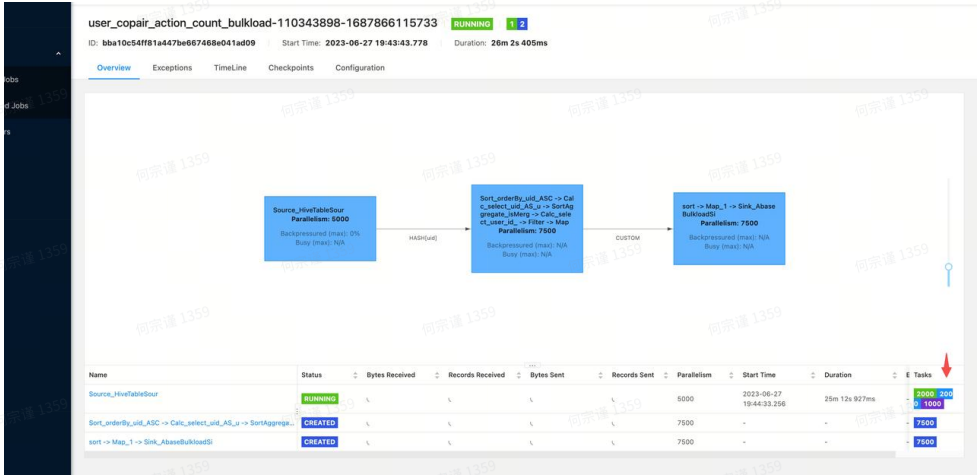
-  总结:
- 问题背景：Darwin Flink Batch 任务频繁因为 TM 堆外内存超限被 kill，影响作业稳定性；
  - 影响范围：**parquet-zstd 读的场景**（跟 zstd codec 的实现有关 [parquet-mr-982](#)）；
  - 问题原因：Hive-Parquet 读取解压时未对 InputStream 及时关闭，导致堆外内存泄露。

## 背景

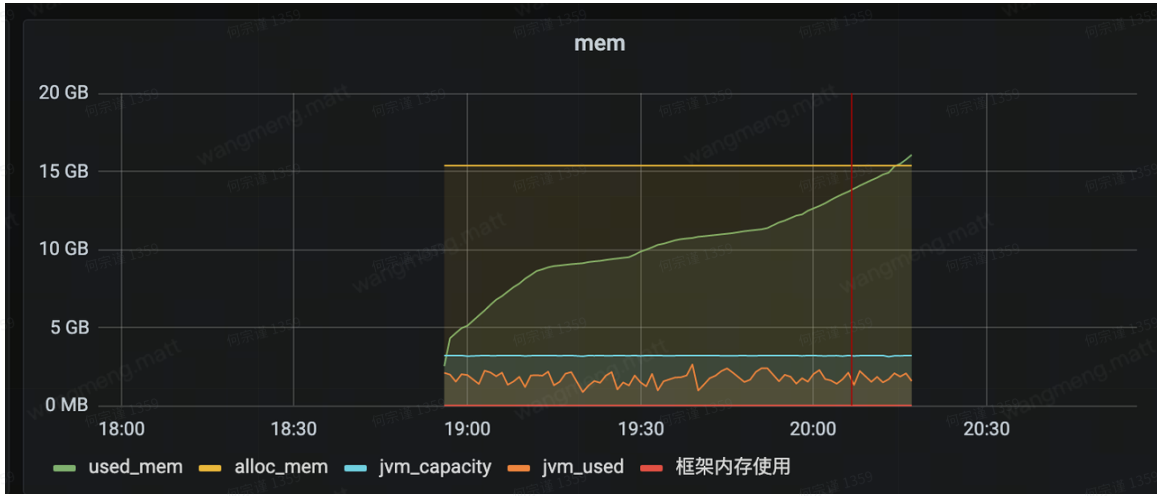
近期 Darwin 业务有一个 Flink Batch SQL 任务在线上测试时，经常遇到**任务的 TM 因为内存超限被 kill 的问题，导致任务失败率较高**（退出码是 22001，内存超限被 kill）。

		...	
	-22001		发生在 Yarn 底层切换到 K8s 的集群，这个退出码是 container 被 Godel 外围组件清理，包括节点硬件故障驱逐，在线内存紧张驱逐等。

作业的拓扑结构如下：



- 作业配置：
  - TM 1core16G，一个 TM 一个 slot；
  - Source 并发 5000，每次最多同时运行 2000 个 Task；
- 作业现象：
  - 一个 TM 平均会执行 2-3 个 Source Task，按照上面的配置，第一个 task 都可以正常跑完，到第二或第三个启动时，TM 就会因为内存超限被 kill，从监控上可以看到 TM 的内存一直在增加，**一个 Task 跑完之后内存并没有释放（确认作业出现堆外内存泄露）。**



## 问题分析

从上面 Container 的内存监控图可以看出，内存的超用是出现在堆外部分（这里的堆外是广义的堆外内存，除了 JVM heap 以外都认为是堆外），任务是也可以稳定性复现，这里通过 Jemalloc profiling（[Flink 堆外内存分析用户文档](#)）能力来分析其堆外内存的使用。

Sys 这边有关于 Jemalloc 的文档，[Jemalloc内存分配与优化实践](#)，Jemalloc 支持 profiling，分析内存非常方便。

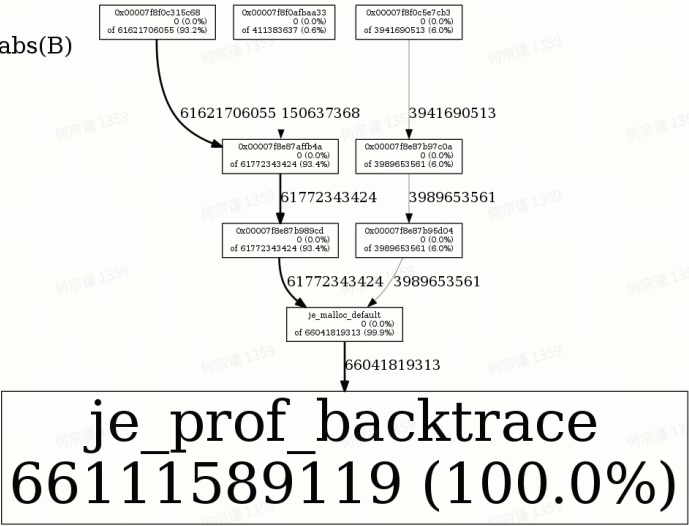
## Jemalloc Profile 分析

打开 Jemalloc 的 profile 之后，通过 jeprof 分析 native 内存的使用情况，信息如下：

```
1  -rw-r--r-- 1 root root 124487 Jun 26 21:36
   jeprof.13.41.i41.heap
2  root@dc05-pff-ff-7cd4-66f9-ab20-246f(container-2cc76e43-e58-
   1686204898999-335888-01-000116):/opt/tiger/yodel/container#
   jeprof --show_bytes /opt/tiger/jdk/jdk1.8/bin/java
   jeprof.13.41.i41.heap
3  Using local file /opt/tiger/jdk/jdk1.8/bin/java.
4  Argument "MSWin32" isn't numeric in numeric eq (==) at
   /usr/local/bin/jeprof line 5124.
5  Argument "linux" isn't numeric in numeric eq (==) at
   /usr/local/bin/jeprof line 5124.
6  Using local file jeprof.13.41.i41.heap.
7  Welcome to jeprof! For help, type 'help'.
8  (jeprof) top
9  Total: 46207943356 B
10 46207943356 100.0% 100.0% 46207943356 100.0% je_prof_backtrace
11 0 0.0% 100.0% 43229250742 93.6% 0x00007f69d527fb4a
12 0 0.0% 100.0% 2635787516 5.7% 0x00007f69d5315d04
13 0 0.0% 100.0% 2635787516 5.7% 0x00007f69d5317c0a
14 0 0.0% 100.0% 43229250742 93.6% 0x00007f69d53189cd
15 0 0.0% 100.0% 400570049 0.9% 0x00007f6a4dfba773
16 0 0.0% 100.0% 2107408 0.0% 0x00007f6a4dfc6759
17 0 0.0% 100.0% 2102276 0.0% 0x00007f6a4dfc7758
18 0 0.0% 100.0% 2097408 0.0% 0x00007f6a4e38903b
19 0 0.0% 100.0% 19022207 0.0% 0x00007f6a4e3cba05
```

使用 Jeprof 程序生成内存透视图：

/opt/tiger/jdk/jdk1.8/bin/java  
Total B: 66111589119  
Focusing on: 66111589119  
Dropped nodes with <= 330557945 abs(B)  
Dropped edges with <= 66111589 B



两个的信息基本一致，这里能看到申请堆外内存，主要是由 **0x00007f69d53189cd** 这个 **进程空间地址**所申请的，但是无法看到详情的线程栈信息。

正常情况下这里是可以直接看到具体线程栈调用这里，这次的 case 有一些特殊（引用是一个临时文件，该文件会被清理），无法看不到详细的调用栈信息。

### 进程 maps 信息分析

与 Sys 通过沟通后，在 `/procs/PID/maps` 文件中，可以看到进程的内存映射信息，这个文件会记录：进程加载的库、堆栈、数据和代码段，

```
1 # 0x00007f69d527fb4a/0x00007f69d53189cd 所在地址范围
2 7f69d5271000-7f69d5340000 r-xp 00000000 fe:20 124651095
   /opt/tiger/yodel/container/tmp/libzstd-
   jni6415142372855756955.so (deleted)
3 7f69d5340000-7f69d5540000 ---p 000cf000 fe:20 124651095
   /opt/tiger/yodel/container/tmp/libzstd-
   jni6415142372855756955.so (deleted)
```

每行的字段信息如下：

内核每进程的 vm_area_struct项	/proc/pid/maps中的项	含义
vm_start	"-"前一列，如 00377000	此段虚拟地址空间起始地址
vm_end	"-"后一列，如 00390000	此段虚拟地址空间结束地址
vm_flags	第三列，如r-xp	此段虚拟地址空间的属性。每种属性用一个字段表示，r表示可读，w表示可写，x表示可执行，p和s共用一个字段，互斥关系，p表示私有段，s表示共享段，如果没有相应权限，则用-'代替
vm_pgoff	第四列，如 00000000	对有名映射，表示此段虚拟内存起始地址在文件中以页为单位的偏移。对匿名映射，它等于0或者 vm_start/PAGE_SIZE
vm_file->f_dentry->d_inode->i_sb->s_dev	第五列，如fd:00	映射文件所属设备号。对匿名映射来说，因为没有文件在磁盘上，所以没有设备号，始终为00:00。对有名映射来说，是映射的文件所在设备的设备号
vm_file->f_dentry->d_inode->i_ino	第六列，如 9176473	映射文件所属节点号。对匿名映射来说，因为没有文件在磁盘上，所以没有节点号，始终为00:00。对有名映射来说，是映射的文件的节点号
	第七列，如/lib/ld-2.5.so	对有名来说，是映射的文件名。对匿名映射来说，是此段虚拟内存存在进程中的角色。[stack]表示在进程中作为栈使用，[heap]表示堆。其余情况则无显示

这里，可以定位到 **0x00007f69d53189cd** 这个地址，指向的是 **/opt/tiger/yodel/container/tmp/libzstd-jni6415142372855756955.so** 的 so 包，这里可以初步定位出来这里的内存主要是 **libzstd-jni** 包的所申请的。

### 内存火焰图分析

参考 [异步分析器](#)，对于 jemalloc 的内存分配进行 profile 分析：

引用是一个临时文件，该文件会被清理



李本超 2024年1月2日  
这个临时文件是怎么判定呢，是在/tmp 目录里面就是临时文件嘛？



王蒙 2024年1月2日  
[@李本超](#) 这个根据地址信息查询的 maps 文件，能看到 `【/opt/tiger/yodel/container/tmp/libzstd-jni6415142372855756955.so (deleted)】` 这个文件路径信息



李本超 2024年1月2日  
所以可以理解为当前这个工具可以完整应对堆外内存分析是吧，找到内存占用大头，就能找到对应的代码？

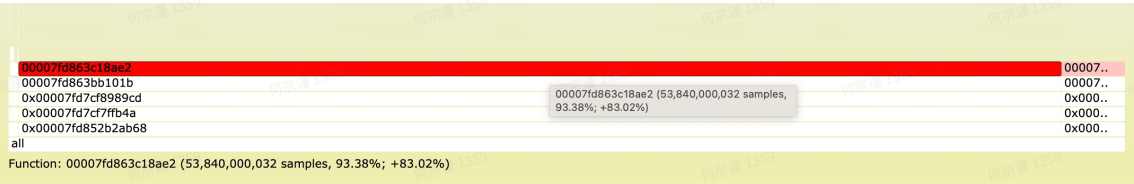


王蒙 2024年1月2日  
[@李本超](#) 我理解这个还得跟火焰图一起交叉来看，火焰图定位具体的方法会更方便一些。这个应该只能看到本地文件信息



📖 [Jemalloc 内存差分火焰图](#) 也可以参考这个，分析不同时间段之后，内存变化情况。在这个 case 中，差分图也是只能看到某个进程空间地址在申请内存，无法看到具体线程栈信息，一定程度影响排查效率。

这里，可以定位到内存申请主要是跟 **Hive-parquet-zstd** 相关。



## Hive-Parquet 分析 -- 问题定位

👤 [胡伟华](#) [云帆](#) [张云帆](#) 在 Parquet 社区找到了一个类似 Case 的 MR —— <https://github.com/apache/parquet-mr/pull/982>，这里改动的点差异如下（第一个图是 flink 使用的 parquet 版本，第二个图是 parquet 最新的代码版本）：

```
public CompressionCodecName getCodecName() { return this.codecName; }

class HeapBytesDecompressor extends BytesDecompressor {
    private final CompressionCodec codec;
    private final Decompressor decompressor;

    HeapBytesDecompressor(CompressionCodecName codecName) {
        this.codec = CodecFactory.this.getCodec(codecName);
        if (this.codec != null) {
            this.decompressor = CodecPool.getDecompressor(this.codec);
        } else {
            this.decompressor = null;
        }
    }

    public BytesInput decompress(BytesInput bytes, int uncompressedSize) throws IOException {
        BytesInput decompressed;
        if (this.codec != null) {
            if (this.decompressor != null) {
                this.decompressor.reset();
            }
            InputStream is = this.codec.createInputStream(bytes.toInputStream(), this.decompressor);
            decompressed = BytesInput.from(is, uncompressedSize);
        } else {
            decompressed = bytes;
        }
        return decompressed;
    }
}
```

### parquet 版本



臧传奇 2023年6月29日  
是否这个也暴露出内部 Parquet 版本过低，好多 Bugfix 没有及时 port 过来啊



王蒙 2023年6月29日  
[@臧传奇](#) 是的，其他的依赖也会有类似问题，这个还比较难避免



```
Decompiled .class file, bytecode version: 52.0 (Java 8)

220 }
221 }
222 }
223 }
224 public BytesInput decompress(BytesInput bytes, int uncompressedSize) throws IOException {
225     BytesInput decompressed;
226     if (this.codec != null) {
227         if (this.decompressor != null) {
228             this.decompressor.reset();
229         }
230     }
231     InputStream is = this.codec.createInputStream(bytes.toInputStream(), this.decompressor);
232     Throwable var5 = null;
233
234     try {
235         byte[] buf = new byte[uncompressedSize];
236         (new DataInputStream(is)).readFully(buf);
237         decompressed = BytesInput.from(buf);
238     } catch (Throwable var14) {
239         var5 = var14;
240         throw var14;
241     } finally {
242         if (is != null) {
243             if (var5 != null) {
244                 try {
245                     is.close();
246                 } catch (Throwable var13) {
247                     var5.addSuppressed(var13);
248                 }
249             } else {
250                 is.close();
251             }
252         }
253     }
254 }
255 } else {
256     decompressed = bytes;
257 }
258 }
259 }
260 }
return decompressed;
```

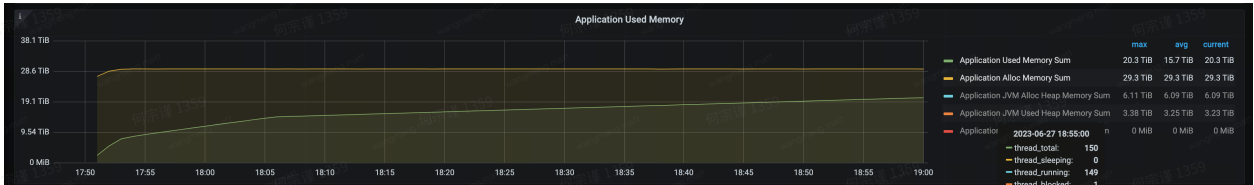
这个方法在内存火焰图上也能清楚的看到：



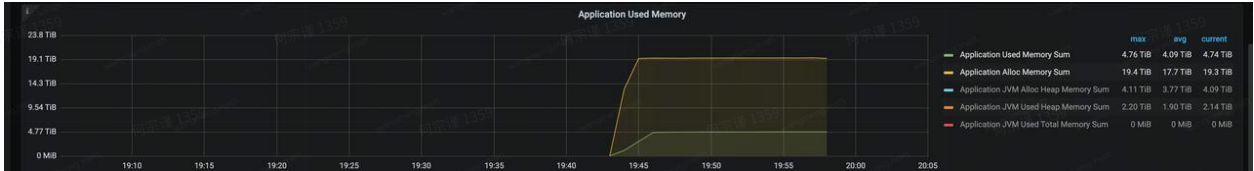
到这里，基本上问题大概率已经明确，经过线上任务验证后，也确认是这个 parquet bug 导致的线上堆外内存泄露。

## 验证效果

修复前，作业内存的使用：



修复后，作业内存的使用：



## 其他堆外内存工具分析

这里，也介绍一些其他的堆外内存排查工具。

## Native Memory Tracking

这个工具可以看到 Java Heap、Class、Thread、Code、GC、Compiler、Internal、Other、Symbol、Native Memory Tracking、Arena Chunk 这几部分的内存使用情况。

Native Memory Tracker 这里看到的内存使用与 jeprof 看到的差异比较大，原因是 Native Memory Tracker 无法 track JNI 调用的内存信息（这个工具在这个 case 中是无效的）。

内存的使



刘首维 2023年6月29日

赞，预期可以很好地提升我们这边 Flink Batch 作业的稳定性

这个只能监控 JVM 原生申请的内存大小，如果是通过 JDK 封装的系统 API 申请的内存，是统计不到的，例如 Java JDK 中的 DirectBuffer 以及 MappedByteBuffer 这两个（当然，对于这两个，我们后面也有其他的办法去看到当前使用的大小。当然xigao dog 啥都不会）。以及如果你自己封装 JNI 调用系统调用去申请内存，都是 Native Memory Tracking 无法涵盖的。

--- 见：[全网最硬核 JVM 内存详解\(上\) | HeapDump性能社区](#)

使用命令如下：

```
1 # 通过下面的参数打开
2 -XX:NativeMemoryTracking=detail
3
4 /opt/tiger/jdk/jdk1.8/bin/jcmd 13 VM.native_memory detail >
  /opt/tiger/yodel/log/memory-detail.log
```

## Pmap 分析

可以参考 [Flink Batch任务使用过多堆外内存分析](#)。

## addr2line

使用方式参考 [Android Native入门](#)，这个工具可以根据 0x00007f69d53189cd 与对应的so包，直接定位到具体的代码行。

```
1 sudo apt-get update
2 sudo apt-get install binutils
3
4 addr2line 0x00007fd852b2ab68 -e
  /opt/tiger/yodel/container/tmp/libzstd-jni6415142372855756955.so -
  f -C -s
```

## 复盘及 Todo

思考	总结
Why? 为什么会发生这个问题?	<ul style="list-style-type: none"><li>Flink hive 依赖包的 bug 导致；</li></ul>
Why? 为什么测试/灰度阶段没有发现?	<ul style="list-style-type: none"><li>特殊场景（hive-zstd read）才会触发，并且数据量小的时候不会触发；</li><li>现在的性能测试流程不会对内存的资源使用进行对比验证；</li></ul>
Why? 为什么系统不能容错?	<ul style="list-style-type: none"><li>内存泄露的 case，系统层没有办法进行自动容错；</li></ul>
Can? 能不能更早发现问题?	<ul style="list-style-type: none"><li>触发条件比较特殊；</li><li>一般情况下，加资源可以 cover，也不会意识到这个是内存泄露的问题；</li></ul>
Can? 解决过程能不能更快?	<ul style="list-style-type: none"><li>线上可能也有不少类似问题，但是加资源可以 cover，目前对线上影响没有那么大；</li><li>这次是在测试环境，大数据量场景，问题就比较严重了；</li><li>堆外分析成本比较高，一定程度也影响问题排查效率；</li></ul>
	<ul style="list-style-type: none"><li>完善测试流程：需要考虑在测试流程中引入对资源使用的对比；</li></ul>

0x00007fd852b2ab68



臧传奇 2023年6月29日

这个地址用法不准确，因为这是一个动态内存地址；应该根据这个地址，减去 zstd-jni.so 的初始地址，得到偏移地址，addr2line 进而使用偏移地址，获得偏移地址指向的源代码；



王蒙 2023年6月29日

@臧传奇 需要减去 7f69d5271000 这个起始地址？



臧传奇 2023年6月29日

@王蒙 对

完善堆外内存排查工具，降低排查成本；



Evan Huang 2024年8月12日

现在有什么工具和文档指示下吗？



王蒙 2024年8月12日

<https://flink.bytedance.net/1645/268150>

最新的、高效的使用方式，已经记录到我们的用户文档里了



王蒙 2024年8月12日

线上镜像默认已经加了 jemalloc 依赖，可以通过动态参数开启

- How? 怎么防止类似的事情发生?
- 完善堆外内存排查工具，降低排查成本；

## ToDo

- ☒

Hive Parquet zstd 堆外内存问题修复；P0

德伟 苏德伟 云帆 张云帆

2023年7月6日 18:00
- ☒

堆外内存排查工具完善（覆盖 Streaming + Batch，Godel + Yarn 场景）；P1

王蒙 马春辉 郦泽坤 曹帝胄
- ☒

引擎默认提供 Jemalloc so 包（Jemalloc 的版本需要跟 sys 同学再一起确认下）；

2023年7月13日 18:00
- ☒

ByteDog 平台能支持 Flink 大数据场景的内存分析（需要跟 sys 同学对齐）；

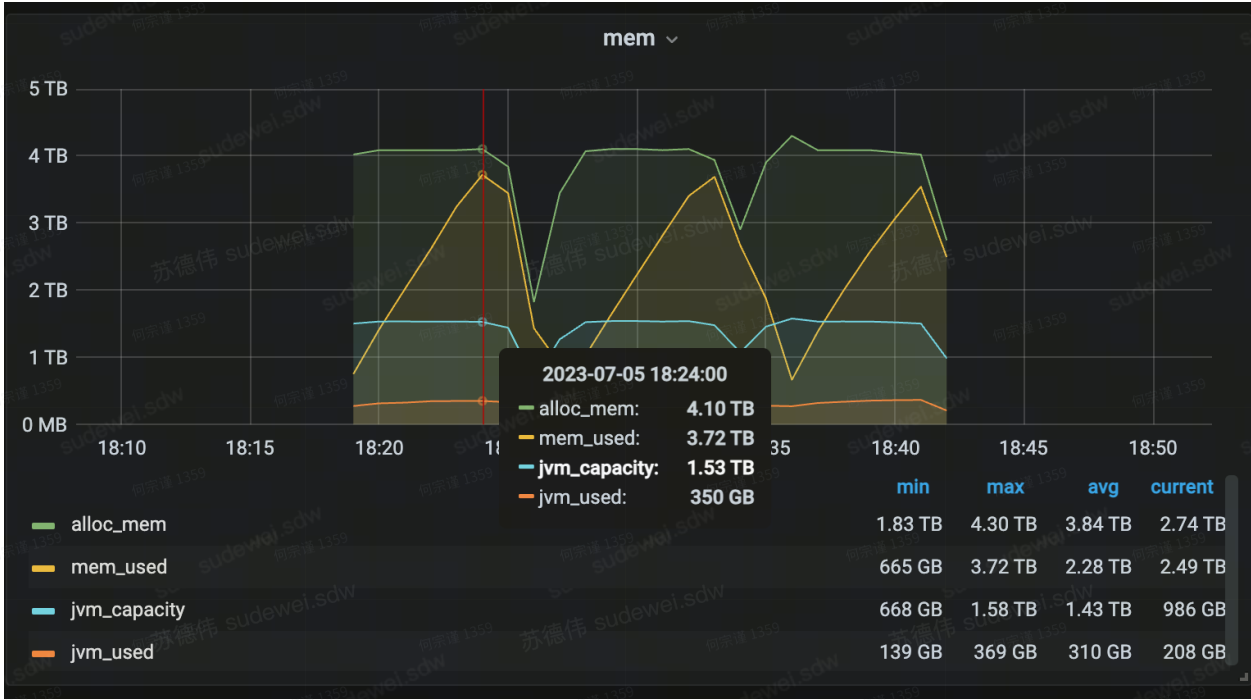
郦泽坤 臧传奇

## 升级parquet版本以修复内存泄漏问题

作业：[https://data.bytedance.net/dorado/development/node/110343898?project=cn\\_6199&version=-1](https://data.bytedance.net/dorado/development/node/110343898?project=cn_6199&version=-1)

当前线上版本：

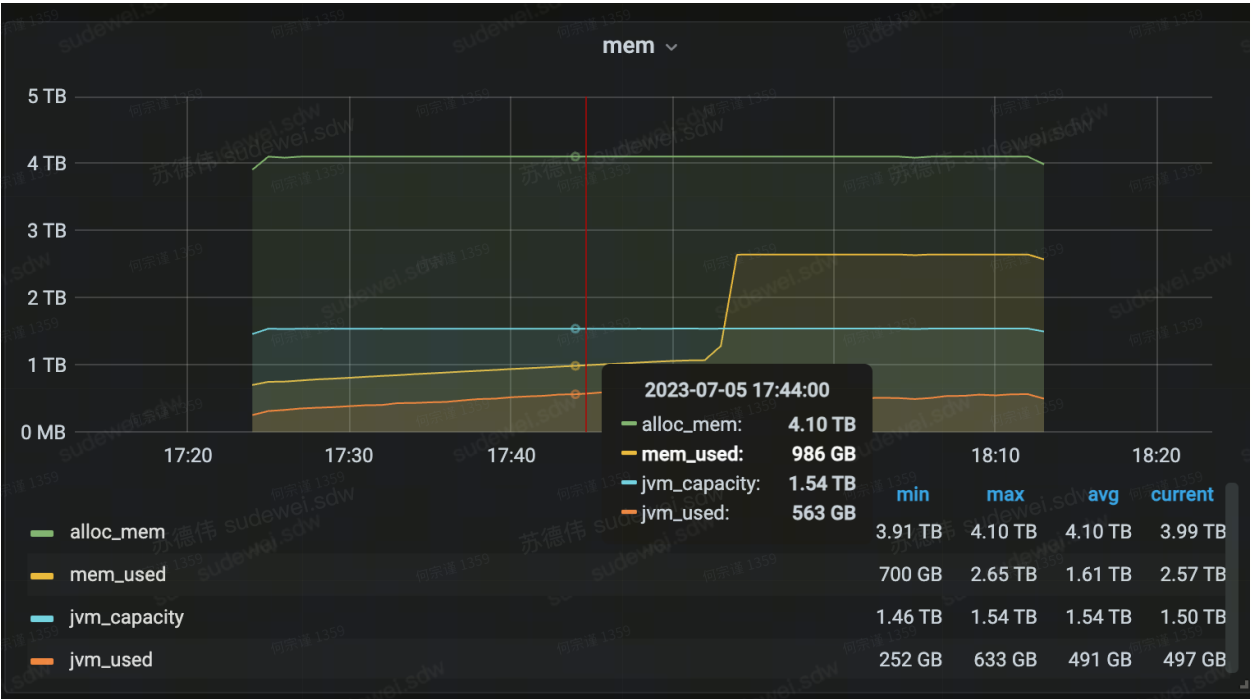
[https://cloud.bytedance.net/megatron/jobs/application\\_1688443034999\\_2787409/cn](https://cloud.bytedance.net/megatron/jobs/application_1688443034999_2787409/cn)



作业直接Fail，无法运行。

stage版本：1.0.0.5583

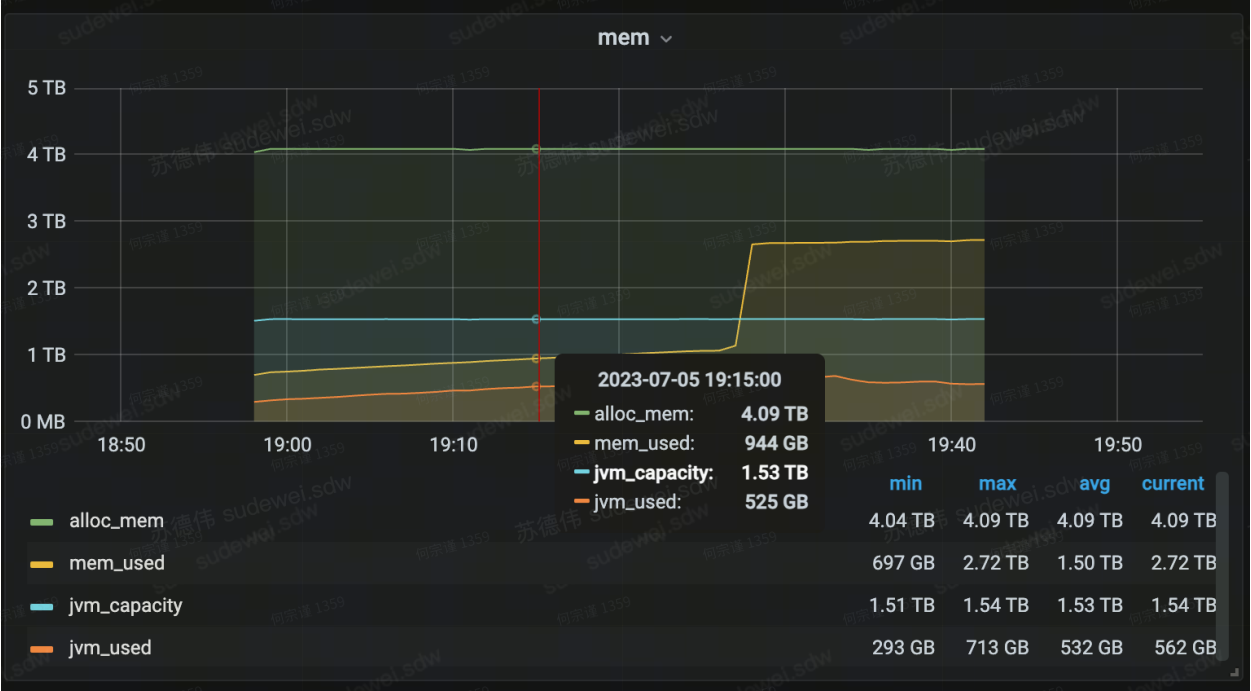
[https://cloud.bytedance.net/megatron/jobs/application\\_1688443034999\\_2709005/cn](https://cloud.bytedance.net/megatron/jobs/application_1688443034999_2709005/cn)



将parquet-hadoop版本升级至1.10.1-bd1.0.33-zstd-1.4，主要就是加上close，可以看到还是有缓慢内存泄漏。

stage版本：1.0.0.5585

[https://cloud.bytedance.net/megatron/jobs/application\\_1688443034999\\_2838124/cn](https://cloud.bytedance.net/megatron/jobs/application_1688443034999_2838124/cn)



几乎和85版本保持一致，可以得知添加close逻辑后修复了一部分问题，但是还是有一些缓慢泄漏的情况存在。



