JProfiler Best Practices: Powerful Performance Diagnostic Tool

Alibaba Cloud
Jul 4 · 11 min read

By Bruce Wu

Background

Performance diagnosis is a problem that software engineers need to frequently face and solve in their daily work. Today, user experience matters the most, and improving application performance can provide significant advantages. Java is one of the most popular programming languages. Its performance diagnosis has been attracting a lot of attention across the industry for a long time. Many factors may cause performance problems in Java applications. Such factors include thread control, disk I/O, database access, network I/O, and garbage collection (GC). You need an excellent performance diagnostic tool to find these problems. This article describes some common Java performance diagnostic tools, and highlights the basic principles and best practices of JProfiler, an excellent representative of these tools. The JProfiler version discussed in this article is `JProfiler10.1.4` .

Brief Introduction to Java Performance Diagnostic Tools

Various performance diagnostic tools are available in the Java world, including simple command line tools, such as jmap and jstat, and comprehensive graphical diagnostic tools, such as JVisualvm and JProfiler. The following section briefly describes each of these tools.

Simple Command Line Tools

The Java Development Kit (JDK) offers many built-in command line tools. They can help you obtain information of the target Java virtual machine (JVM) from different aspects and different layers.

- jinfo — allows you to view and tune various parameters of the target JVM in real-time.
- jstack — allows you to obtain thread stack information of the target Java process, detect deadlocks, and to locate infinite loops.
- jmap — allows you to obtain memory-related information of the target Java process, including the usage of different Java heaps, statistical information of objects in Java heaps, and loaded classes.
- jstat — is a light-weight versatile monitoring tool. It allows you to obtain various information about loaded classes, Just-In-Time (JIT) compilation, garbage collection, and memory usage of the target Java process.
- jcmd — is more comprehensive than jstat. It allows you to obtain various information about the performance statistics, Java Flight Recorder (JFR), memory usage, garbage collection, thread stacking, and JVM runtime of the target Java process.

Comprehensive Graphical Diagnostic Tools

You can use any or any combinations of the above command line tools to help you obtain the basic performance information about the target Java application. However, they have the following drawbacks:

1. They cannot obtain the method-level analytical data, such as the call relationship between different methods, and the frequency and duration that a method is called. They are very important for identifying the performance bottlenecks of your application.
2. You must log on to the host machine of the target Java application to use them, which is not very convenient.
3. The analytical data is generated at a terminal, and the results are not intuitively displayed.

The following are several comprehensive graphical performance diagnostic tools:

JVisualvm

JVisualvm is a built-in visual performance diagnostic tool provided by the JDK. It obtains analytical data of the target JVM, including CPU usage, memory usage, threads, heaps, and stacks, by using various methods such as JMX, jstatd, and Attach API. In addition, it can intuitively display the quantity and size of each object in Java heaps, the number of times that a Java method is called, and the duration that a Java method is executed.

JProfiler

JProfiler is a Java application performance diagnostic tool developed by ej-technologies. It focuses on addressing four important topics:

1. Method calls — The analysis of method calls helps you to understand what your application is doing and find ways to improve its performance.
2. Allocations — Through analyzing objects on the heap, reference chains, and garbage collection, this functionality enables you to fix memory leaks and optimize memory usage.
3. Thread and lock — JProfiler provides multiple analysis views on threads and locks to help you discover multithreading problems.
4. High-level subsystems — Many performance problems occur on a higher semantic level. For example, with Java Database Connectivity (JDBC) calls, you probably want to find out which SQL statement is the slowest. JProfiler supports integrated analysis of these subsystems.

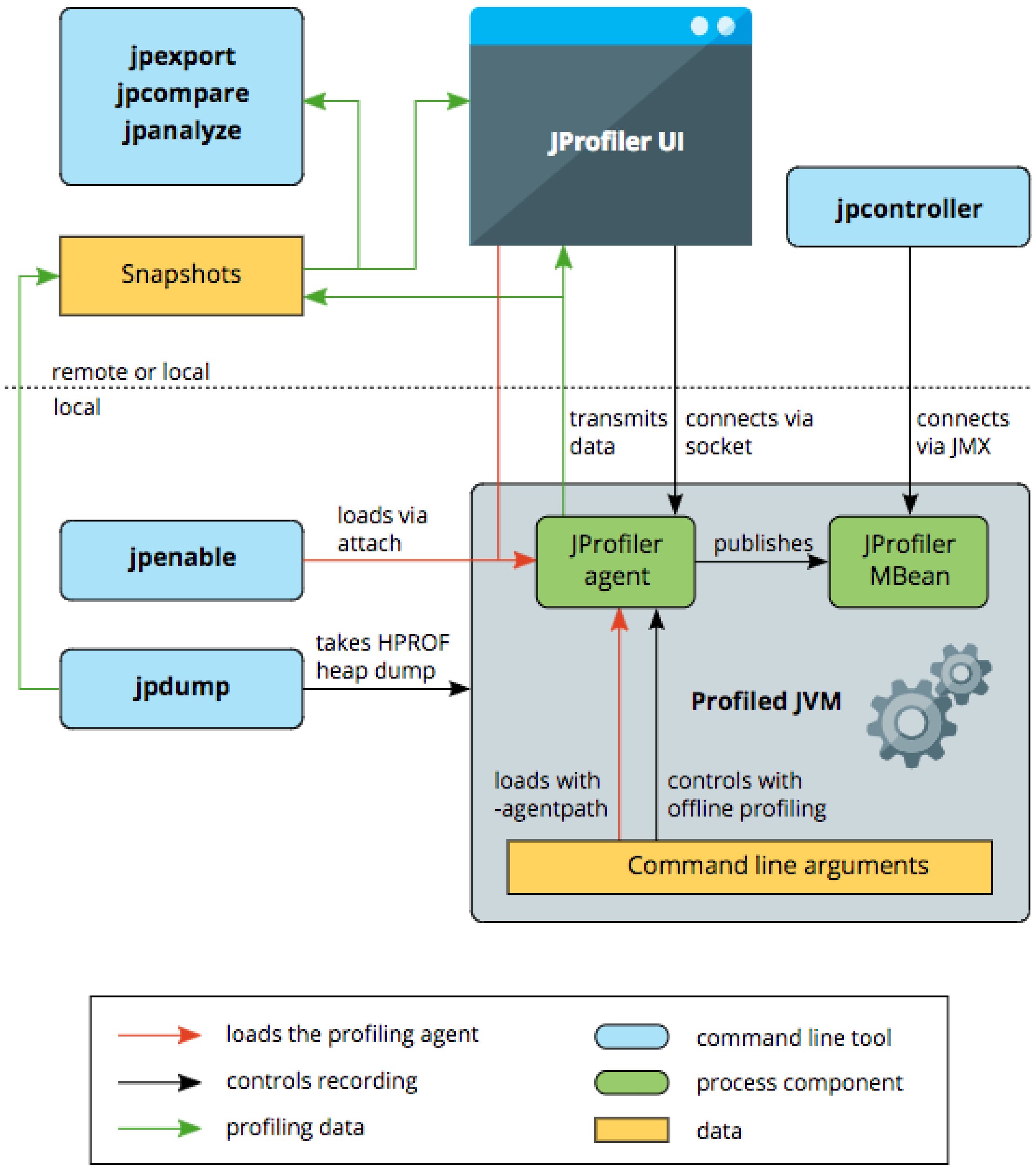
Distributed Application Performance Diagnosis

If you only need to diagnose performance bottlenecks of standalone Java applications, the diagnostic tools described above are enough to meet your needs. However, when standalone modern systems gradually evolve into distributed systems and microservices, the tools above cannot meet the requirements. We need to use the end-to-end tracing function of distributed tracing systems such as Jaeger, ARMS, and SkyWalking. A variety of distributed tracing systems are available in the market, but their implementation mechanisms are similar. They record the tracing information through code tracking, transmit the recorded data to central processing systems through SDKs or agents, and provide query interfaces to display and analyze the results. For more information about the principles of distributed tracing systems, see the article titled OpenTracing Implementation of Jaeger.

Introduction to JProfiler

Core Components

JProfiler consists of a JProfiler agent to collect analytical data from the target JVM, a JProfiler UI to visually analyze data, and command line utilities to provide various functions. The big picture of all important interactions between them is given as follows.



JProfiler Agent

The JProfiler agent is implemented as a native library. You can load it at the startup of a JVM by using the parameter `-agentpath:<path to native library>`, or when the application is running by using the JVM Attach mechanism. After the JProfiler agent is loaded, it sets up the JVM tool interface (JVMTI) environment, to monitor all kinds of events generated by the JVM, such as thread creation and class loading. For example, when it detects a class loading event, the JProfiler agent inserts its own bytecode to these classes to perform its measurements.

JProfiler UI

The JProfiler UI is started separately and connects to the profiling agent through a socket. This means that it is actually irrelevant if the profiled JVM is running on the local machine or on a remote machine — the communication mechanism between the profiling agent and the JProfiler UI is always the same.

From the JProfiler UI, you can instruct the agent to record data, display the profiling data in the UI, and save snapshots to disk.

Command Line Tools

JProfiler provides a series of command line tools to implement different functions.

- jpcontroller — is used to control how the agent collects data. It sends instructions to the agent through the JProfiler MBean registered by the agent.
- jpenable — is used to load the agent to a running JVM.
- jpdump — is used to capture the heap snapshots of a running JVM.
- jpexport & jpcompare — is used to extract data and create HTML reports from previously saved snapshots.

Installation

JProfiler supports diagnosing the performance of both local and remote Java applications. If you need to collect and display analytical data of a remote JVM in real time, perform the following steps:

1. Install JProfiler UI locally.
2. Install JProfiler agent on the remote host machine, and load it to the target JVM.
3. Connect JProfiler UI to the agent.

For more information about the installation steps, see Installing JProfiler and Profiling A JVM.

Best Practices

This section shows you how to use JProfiler to diagnose the performance of Alibaba Cloud LOG Java Producer(the “Producer”), a LogHub class library. If you encounter any performance problems with your application or when you use Producer, you can take similar measures to find out the root cause. If you do not know Producer, we recommend that you read this article first: Alibaba Cloud LOG Java Producer — A powerful tool to migrate logs to the cloud.

For sample code used in this section, see SamplePerformance.java.

JProfiler Settings

Data Collection Mode

JProfiler provides two data collection methods: sampling and instrumentation.

- Sampling — suitable for scenarios that do not require high data collection accuracy. The advantage of this method that it has low impact on system performance. The drawback is that it does not support some features such as method-level statistics.
- Instrumentation — the complete data collection mode that supports high accuracy. The drawback is that it must analyze many classes and it has a relatively heavy impact on the application performance. To reduce the impact, you may want to use it with a filter.

In this example, we need to obtain the method-level statistical information, so we choose the instrumentation method. A filter is configured to ensure that the agent records the CPU data of only two classes, `com.aliyun.openservices.aliyun.log.producer` and `com.aliyun.openservices.log.Client` , under the java package.

Startup Modes of the Application

You can specify different parameters for the JProfiler agent to control the startup mode of the application.

- Wait for connection from the JProfiler GUI — The application is started only when JProfiler GUI establishes connection with the profiling agent, and the profiling settings are completed. With this option you can profile the startup phase of your application. The command that you can use to enable this option: `-agentpath:<path to native library>=port=8849`.

- Start immediately, connect later with the JProfiler GUI — JProfiler GUI establishes connection with the profiling agent when it is necessary, and transmits the profiling settings. This option is flexible, but it cannot profile the startup phase of your application. The command that you can use to enable this option: `-agentpath:<path to native library>=port=8849,nowait`.
- Profile offline, JProfiler cannot connect — You have to configure triggers that record data and save snapshots that can be opened with the JProfiler GUI later on. The command that you can use to enable this option: `-agentpath:<path to native library>=offline,id=xxx,config=/config.xml`.

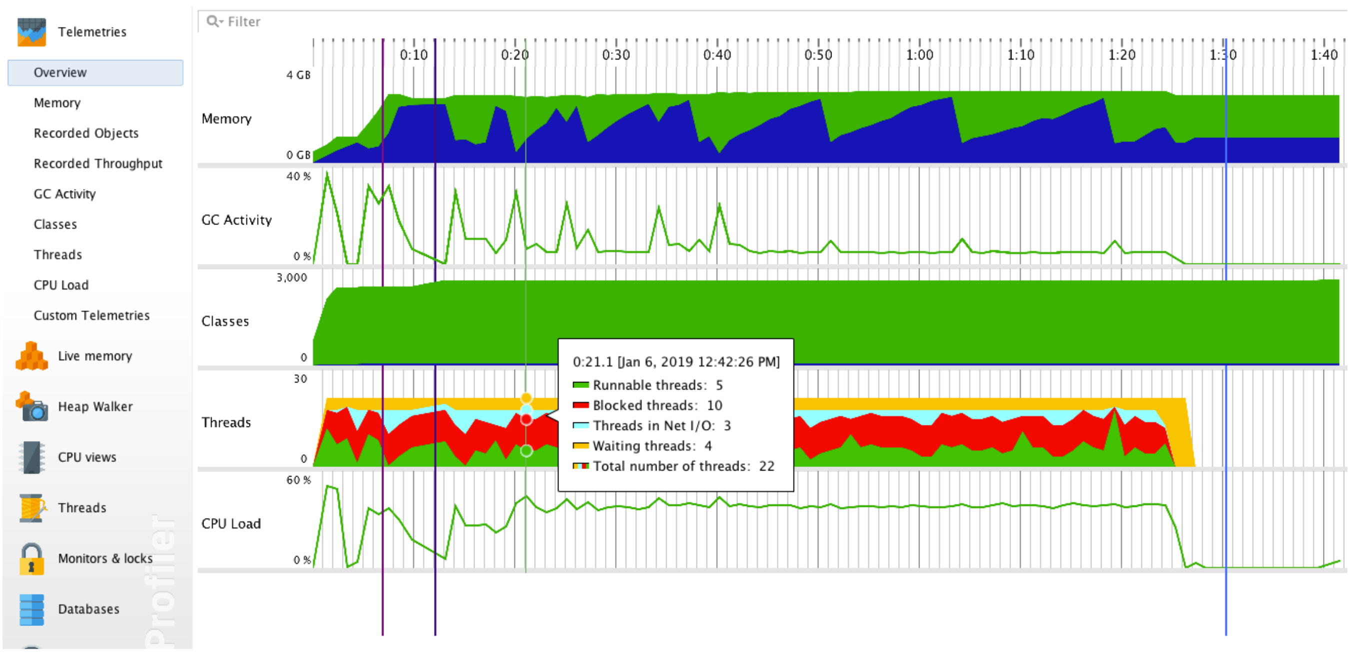
In the testing environment, we need to determine the performance of the application during the startup phase. So we use the default WAIT option here.

Use JProfiler to Diagnose the Application Performance

After completing the profiling settings, you can proceed with the performance diagnosis for Producer.

Overview

On the overview page, we can clearly view graphs (telemetries) about various metrics, such as memory, GC activities, classes, threads, and the CPU load.



0:21.1 [Jan 6, 2019 12:42:26 PM]
🟩 Runnable threads: 5
🟥 Blocked threads: 10
Threads in Net I/O: 3
🟨 Waiting threads: 4
Total number of threads: 22

We can make the following assumptions based on this telemetry:

1. A large number of objects are generated during the application running process. These objects have a very short lifecycle, and most of them can be promptly recycled by the garbage collector. They won't cause memory usage to continually grow.

2. As expected, the number of loaded classes increases rapidly during the startup period, and stabilizes afterward.

3. When the application is running, many threads are blocked. Extra attention must be paid to this issue.

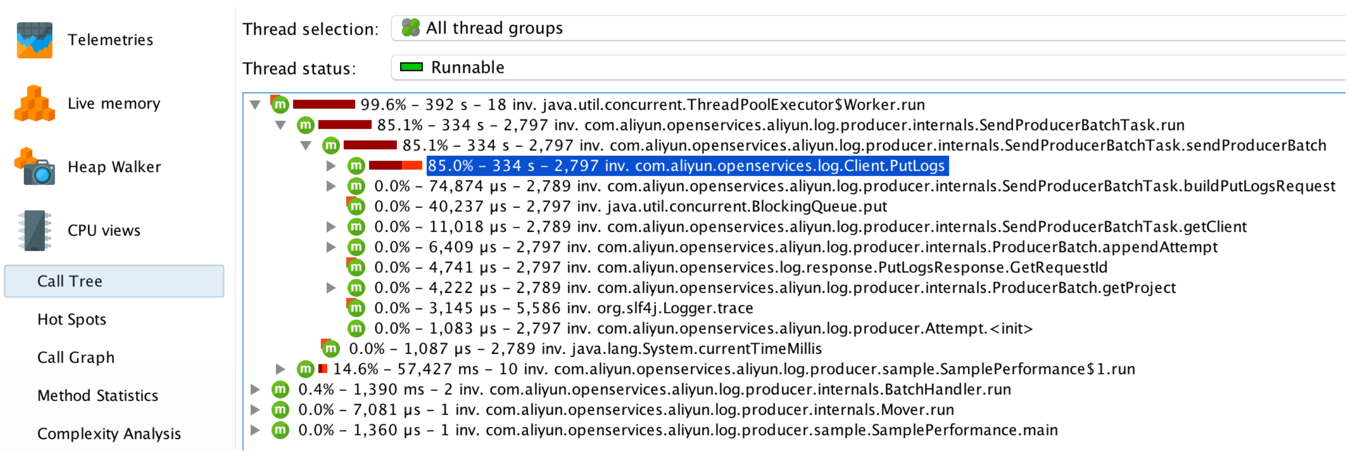
4. At the startup phase of the application, the CPU usage is high. We need to find out why.

CPU Views

The number of executions, execution time, and call relationships of each method in the application are displayed by CPU views. They are helpful in locating the method that has the most significant impact on the performance of your application.

Call Tree

Call tree clearly shows the hierarchical call relationships between different methods by using tree graphs. In addition, JProfiler sorts the sub-methods by their total execution time, which allows you to quickly locate key methods.



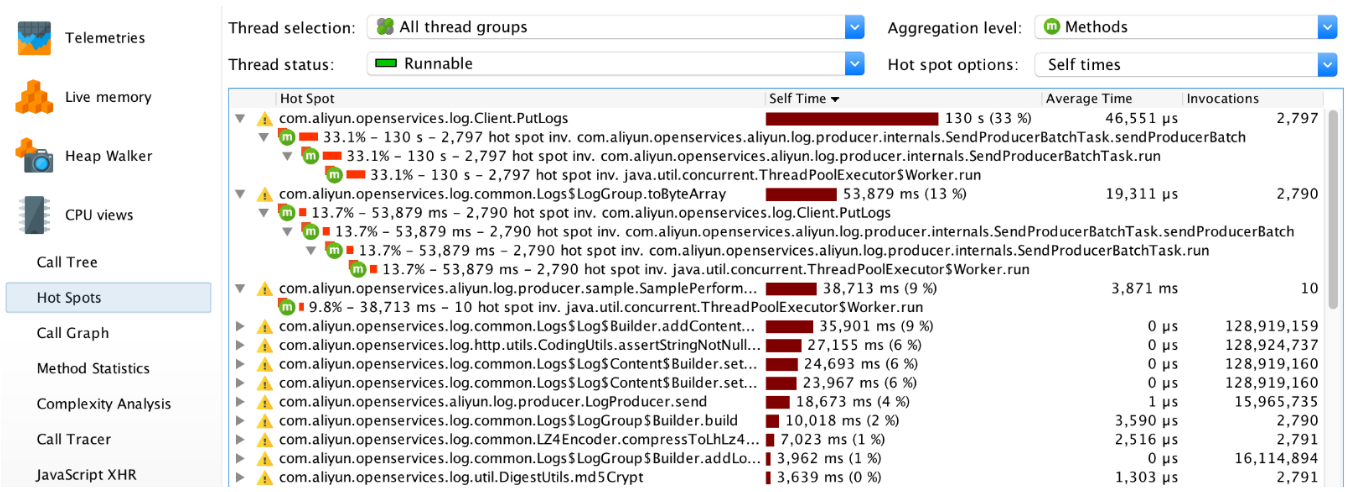
Thread selection: All thread groups
Thread status: Runnable

```
▼ 99.6% - 392 s - 18 inv. java.util.concurrent.ThreadPoolExecutor$Worker.run
  ▼ 85.1% - 334 s - 2,797 inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.run
    ▼ 85.1% - 334 s - 2,797 inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.sendProducerBatch
      ▶ 85.0% - 334 s - 2,797 inv. com.aliyun.openservices.log.Client.PutLogs
      ▶ 0.0% - 74,874 µs - 2,789 inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.buildPutLogsRequest
        0.0% - 40,237 µs - 2,797 inv. java.util.concurrent.BlockingQueue.put
      ▶ 0.0% - 11,018 µs - 2,789 inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.getClient
      ▶ 0.0% - 6,409 µs - 2,797 inv. com.aliyun.openservices.aliyun.log.producer.internals.ProducerBatch.appendAttempt
        0.0% - 4,741 µs - 2,797 inv. com.aliyun.openservices.log.response.PutLogsResponse.GetRequestId
      ▶ 0.0% - 4,222 µs - 2,789 inv. com.aliyun.openservices.aliyun.log.producer.internals.ProducerBatch.getProject
        0.0% - 3,145 µs - 5,586 inv. org.slf4j.Logger.trace
        0.0% - 1,083 µs - 2,797 inv. com.aliyun.openservices.aliyun.log.producer.Attempt.<init>
      0.0% - 1,087 µs - 2,789 inv. java.lang.System.currentTimeMillis
  ▶ 14.6% - 57,427 ms - 10 inv. com.aliyun.openservices.aliyun.log.producer.sample.SamplePerformance$1.run
▶ 0.4% - 1,390 ms - 2 inv. com.aliyun.openservices.aliyun.log.producer.internals.BatchHandler.run
▶ 0.0% - 7,081 µs - 1 inv. com.aliyun.openservices.aliyun.log.producer.internals.Mover.run
▶ 0.0% - 1,360 µs - 1 inv. com.aliyun.openservices.aliyun.log.producer.sample.SamplePerformance.main
```

For Producer, the method `SendProducerBatchTask.run()` takes most of the time. If you continue to look down, you will find that most of the time is taken by executing the method `Client.PutLogs()`.

Hot Spots

If you have many application methods, and many of your sub-methods are executed in a short interval, the Hot Spots view can help you quickly locate performance problems. This view can sort the methods based on various factors, such as their individual execution time, total execution time, average execution time, and number of calls. The individual execution time is equal to the total execution time of the method minus the total execution time of all sub-methods.



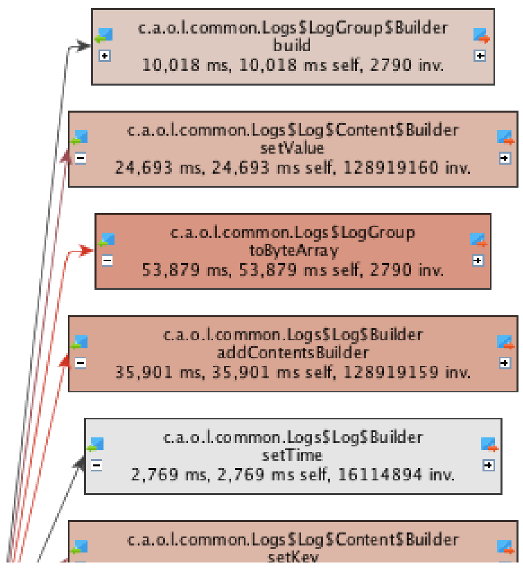
Thread selection: All thread groups
Aggregation level: Methods
Thread status: Runnable
Hot spot options: Self times

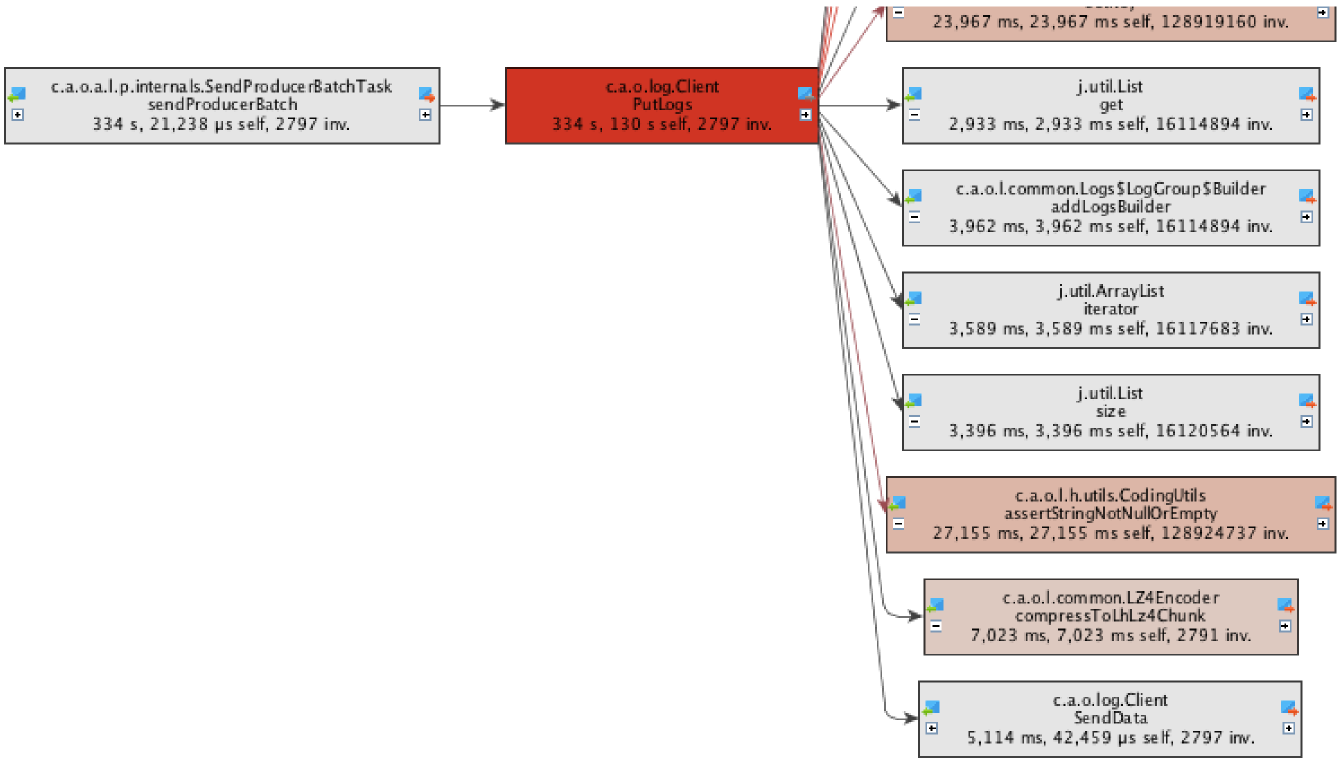
```
Hot Spot                                                  Self Time ▾             Average Time    Invocations
▼ com.aliyun.openservices.log.Client.PutLogs               130 s (33 %)            46,551 µs           2,797
  ▼ 33.1% - 130 s - 2,797 hot spot inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.sendProducerBatch
    ▼ 33.1% - 130 s - 2,797 hot spot inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.run
        33.1% - 130 s - 2,797 hot spot inv. java.util.concurrent.ThreadPoolExecutor$Worker.run
▼ com.aliyun.openservices.log.common.Logs$LogGroup.toByteArray  53,879 ms (13 %)   19,311 µs           2,790
  ▼ 13.7% - 53,879 ms - 2,790 hot spot inv. com.aliyun.openservices.log.Client.PutLogs
    ▼ 13.7% - 53,879 ms - 2,790 hot spot inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.sendProducerBatch
      ▼ 13.7% - 53,879 ms - 2,790 hot spot inv. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.run
          13.7% - 53,879 ms - 2,790 hot spot inv. java.util.concurrent.ThreadPoolExecutor$Worker.run
▼ com.aliyun.openservices.aliyun.log.producer.sample.SamplePerform...  38,713 ms (9 %)  3,871 ms        10
    9.8% - 38,713 ms - 10 hot spot inv. java.util.concurrent.ThreadPoolExecutor$Worker.run
▶ com.aliyun.openservices.log.common.Logs$Log$Builder.addContent...  35,901 ms (9 %)   0 µs     128,919,159
▶ com.aliyun.openservices.log.http.utils.CodingUtils.assertStringNotNull...  27,155 ms (6 %)  0 µs  128,924,737
▶ com.aliyun.openservices.log.common.Logs$Log$Content$Builder.set...  24,693 ms (6 %)  0 µs     128,919,160
▶ com.aliyun.openservices.log.common.Logs$Log$Content$Builder.set...  23,967 ms (6 %)  0 µs     128,919,160
▶ com.aliyun.openservices.aliyun.log.producer.LogProducer.send     18,673 ms (4 %)      1 µs      15,965,735
▶ com.aliyun.openservices.log.common.Logs$LogGroup$Builder.build   10,018 ms (2 %)   3,590 µs          2,790
▶ com.aliyun.openservices.log.common.LZ4Encoder.compressToLhLz4...  7,023 ms (1 %)    2,516 µs          2,791
▶ com.aliyun.openservices.log.common.Logs$LogGroup$Builder.addLo...  3,962 ms (1 %)   0 µs         16,114,894
▶ com.aliyun.openservices.log.util.DigestUtils.md5Crypt            3,639 ms (0 %)    1,303 µs          2,791
```

In this view, you can see that the following three methods: `Client.PutLogs()` , `LogGroup.toByteArray()` , and `SamplePerformance$1.run()` take the most time to be executed individually.

Call Graph

After you find the key methods, the Call Graph view can present you all methods that directly associate with these key methods. This is helpful in finding the solution to the problem and develop the best performance optimization policy.


c.a.o.l.common.Logs$LogGroup$Builder
build
10,018 ms, 10,018 ms self, 2790 inv.
c.a.o.l.common.Logs$Log$Content$Builder
setValue
24,693 ms, 24,693 ms self, 128919160 inv.
c.a.o.l.common.Logs$LogGroup
toByteArray
53,879 ms, 53,879 ms self, 2790 inv.
c.a.o.l.common.Logs$Log$Builder
addContentsBuilder
35,901 ms, 35,901 ms self, 128919159 inv.
c.a.o.l.common.Logs$Log$Builder
setTime
2,769 ms, 2,769 ms self, 16114894 inv.
c.a.o.l.common.Logs$Log$Content$Builder
setKey


23,967 ms, 23,967 ms self, 128919160 inv.
c.a.o.a.l.p.internals.SendProducerBatchTask
sendProducerBatch
334 s, 21,238 µs self, 2797 inv.
c.a.o.log.Client
PutLogs
334 s, 130 s self, 2797 inv.
j.util.List
get
2,933 ms, 2,933 ms self, 16114894 inv.
c.a.o.l.common.Logs$LogGroup$Builder
addLogsBuilder
3,962 ms, 3,962 ms self, 16114894 inv.
j.util.ArrayList
iterator
3,589 ms, 3,589 ms self, 16117683 inv.
j.util.List
size
3,396 ms, 3,396 ms self, 16120564 inv.
c.a.o.l.h.utils.CodingUtils
assertStringNotNullOrEmpty
27,155 ms, 27,155 ms self, 128924737 inv.
c.a.o.l.common.LZ4Encoder
compressToLhLz4Chunk
7,023 ms, 7,023 ms self, 2791 inv.
c.a.o.log.Client
SendData
5,114 ms, 42,459 µs self, 2797 inv.

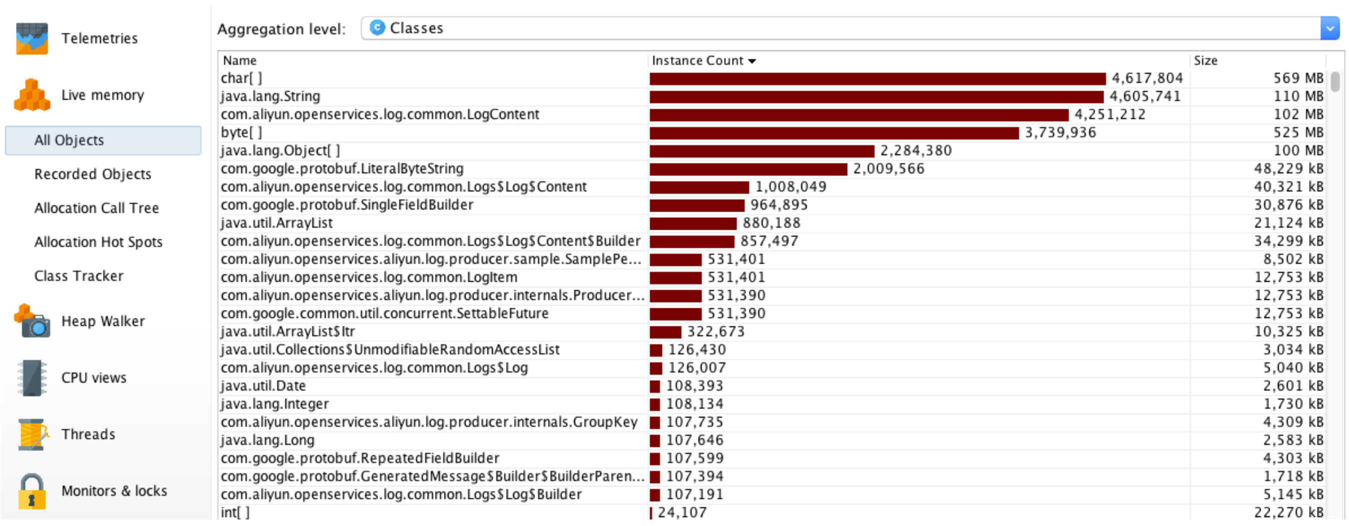
Here, we can see that most of the execution time of the method `Client.PutLogs()` is spent on object serialization. Therefore, the key to performance optimization is to provide a more efficient serialization method.

Live Memory

Live Memory views allow you to know the detailed memory allocation and usage to help you determine if there is a memory leak.

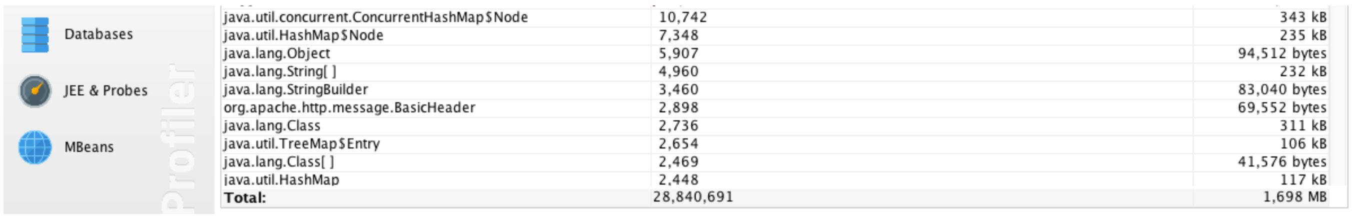
All Objects

The All Objects view shows the number and total size of various objects in the current heap. As you can see from the following figure, a large number of LogContent objects are created during the running process of the application.



Aggregation level: Classes

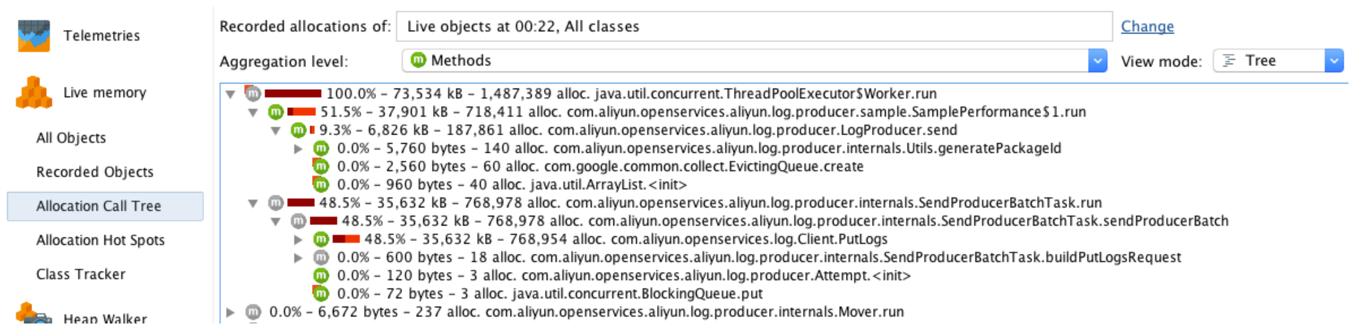
Name	Instance Count	Size
char[ ]	4,617,804	569 MB
java.lang.String	4,605,741	110 MB
com.aliyun.openservices.log.common.LogContent	4,251,212	102 MB
byte[ ]	3,739,936	525 MB
java.lang.Object[ ]	2,284,380	100 MB
com.google.protobuf.LiteralByteString	2,009,566	48,229 kB
com.aliyun.openservices.log.common.Logs$Log$Content	1,008,049	40,321 kB
com.google.protobuf.SingleFieldBuilder	964,895	30,876 kB
java.util.ArrayList	880,188	21,124 kB
com.aliyun.openservices.log.common.Logs$Log$Content$Builder	857,497	34,299 kB
com.aliyun.openservices.aliyun.log.producer.sample.SamplePe...	531,401	8,502 kB
com.aliyun.openservices.log.common.LogItem	531,401	12,753 kB
com.aliyun.openservices.aliyun.log.producer.internals.Producer...	531,390	12,753 kB
com.google.common.util.concurrent.SettableFuture	531,390	12,753 kB
java.util.ArrayList$Itr	322,673	10,325 kB
java.util.Collections$UnmodifiableRandomAccessList	126,430	3,034 kB
com.aliyun.openservices.log.common.Logs$Log	126,007	5,040 kB
java.util.Date	108,393	2,601 kB
java.lang.Integer	108,134	1,730 kB
com.aliyun.openservices.aliyun.log.producer.internals.GroupKey	107,735	4,309 kB
java.lang.Long	107,646	2,583 kB
com.google.protobuf.RepeatedFieldBuilder	107,599	4,303 kB
com.google.protobuf.GeneratedMessage$Builder$BuilderParen...	107,394	1,718 kB
com.aliyun.openservices.log.common.Logs$Log$Builder	107,191	5,145 kB
int[ ]	24,107	22,270 kB



java.util.concurrent.ConcurrentHashMap$Node	10,742	343 kB
java.util.HashMap$Node	7,348	235 kB
java.lang.Object	5,907	94,512 bytes
java.lang.String[ ]	4,960	232 kB
java.lang.StringBuilder	3,460	83,040 bytes
org.apache.http.message.BasicHeader	2,898	69,552 bytes
java.lang.Class	2,736	311 kB
java.util.TreeMap$Entry	2,654	106 kB
java.lang.Class[ ]	2,469	41,576 bytes
java.util.HashMap	2,448	117 kB
Total:	28,840,691	1,698 MB

Allocation Call Tree

The Allocation Call Tree view shows the amount of memory that has been allocated to each method in the form of a tree diagram. As you can see, `SamplePerformance$1.run()` and `SendProducerBatchTask.run()` are large memory consumers.



Recorded allocations of: Live objects at 00:22, All classes

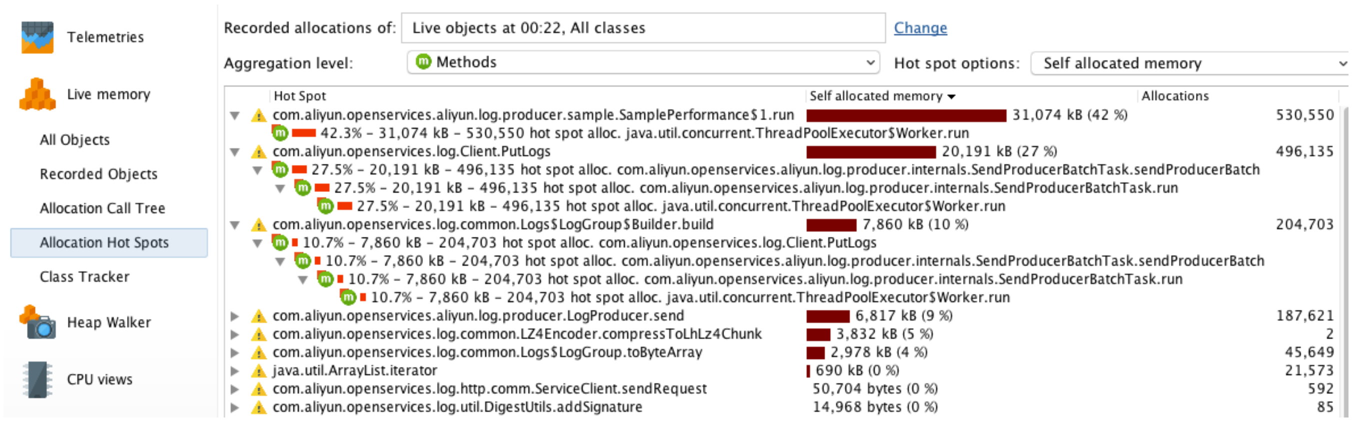
Aggregation level: Methods

View mode: Tree

```
▼ 100.0% - 73,534 kB - 1,487,389 alloc. java.util.concurrent.ThreadPoolExecutor$Worker.run
  ▼ 51.5% - 37,901 kB - 718,411 alloc. com.aliyun.openservices.aliyun.log.producer.sample.SamplePerformance$1.run
    ▼ 9.3% - 6,826 kB - 187,861 alloc. com.aliyun.openservices.aliyun.log.producer.LogProducer.send
      ▶ 0.0% - 5,760 bytes - 140 alloc. com.aliyun.openservices.aliyun.log.producer.internals.Utils.generatePackageId
        0.0% - 2,560 bytes - 60 alloc. com.google.common.collect.EvictingQueue.create
        0.0% - 960 bytes - 40 alloc. java.util.ArrayList.<init>
  ▼ 48.5% - 35,632 kB - 768,978 alloc. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.run
    ▼ 48.5% - 35,632 kB - 768,978 alloc. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.sendProducerBatch
      ▶ 48.5% - 35,632 kB - 768,954 alloc. com.aliyun.openservices.log.Client.PutLogs
      ▶ 0.0% - 600 bytes - 18 alloc. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.buildPutLogsRequest
        0.0% - 120 bytes - 3 alloc. com.aliyun.openservices.aliyun.log.producer.Attempt.<init>
        0.0% - 72 bytes - 3 alloc. java.util.concurrent.BlockingQueue.put
▶ 0.0% - 6,672 bytes - 237 alloc. com.aliyun.openservices.aliyun.log.producer.internals.Mover.run
```

Allocation Hot Spots

If you have many methods, you can quickly find out which method has been assigned the most objects in the Allocation Hot Spots view.



Recorded allocations of: Live objects at 00:22, All classes

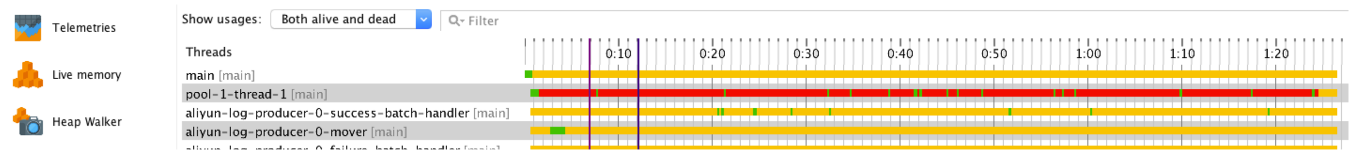
Aggregation level: Methods

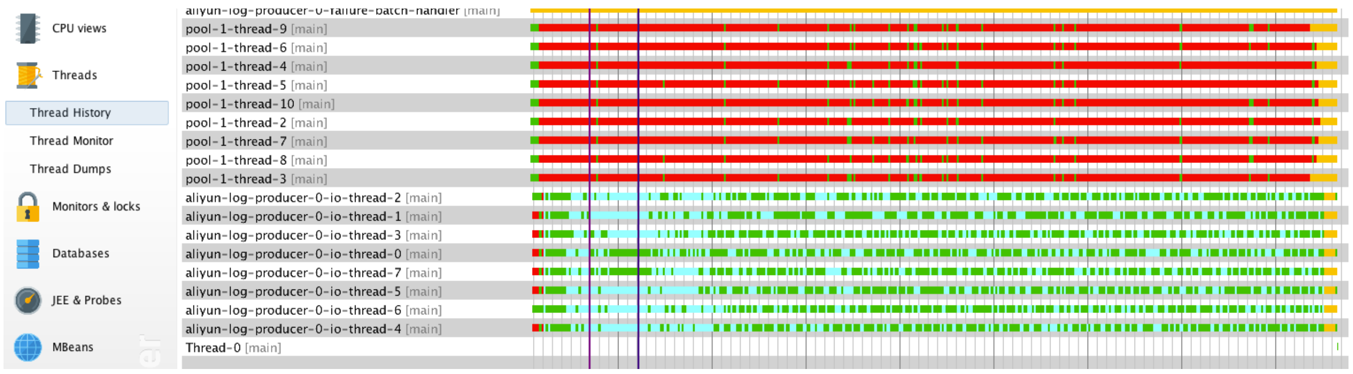
Hot spot options: Self allocated memory

Hot Spot	Self allocated memory	Allocations
▼ com.aliyun.openservices.aliyun.log.producer.sample.SamplePerformance$1.run	31,074 kB (42 %)	530,550
42.3% - 31,074 kB - 530,550 hot spot alloc. java.util.concurrent.ThreadPoolExecutor$Worker.run		
▼ com.aliyun.openservices.log.Client.PutLogs	20,191 kB (27 %)	496,135
▼ 27.5% - 20,191 kB - 496,135 hot spot alloc. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.sendProducerBatch		
▼ 27.5% - 20,191 kB - 496,135 hot spot alloc. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.run		
27.5% - 20,191 kB - 496,135 hot spot alloc. java.util.concurrent.ThreadPoolExecutor$Worker.run		
▼ com.aliyun.openservices.log.common.Logs$LogGroup$Builder.build	7,860 kB (10 %)	204,703
▼ 10.7% - 7,860 kB - 204,703 hot spot alloc. com.aliyun.openservices.log.Client.PutLogs		
▼ 10.7% - 7,860 kB - 204,703 hot spot alloc. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.sendProducerBatch		
▼ 10.7% - 7,860 kB - 204,703 hot spot alloc. com.aliyun.openservices.aliyun.log.producer.internals.SendProducerBatchTask.run		
10.7% - 7,860 kB - 204,703 hot spot alloc. java.util.concurrent.ThreadPoolExecutor$Worker.run		
▶ com.aliyun.openservices.aliyun.log.producer.LogProducer.send	6,817 kB (9 %)	187,621
▶ com.aliyun.openservices.log.common.LZ4Encoder.compressToLhLz4Chunk	3,832 kB (5 %)	2
▶ com.aliyun.openservices.log.common.Logs$LogGroup.toByteArray	2,978 kB (4 %)	45,649
▶ java.util.ArrayList.iterator	690 kB (0 %)	21,573
▶ com.aliyun.openservices.log.http.comm.ServiceClient.sendRequest	50,704 bytes (0 %)	592
▶ com.aliyun.openservices.log.util.DigestUtils.addSignature	14,968 bytes (0 %)	85

Thread History

The Thread History view intuitively shows the status of each thread at different time points.




CPU views
Threads
Thread History
Thread Monitor
Thread Dumps
Monitors & locks
Databases
JEE & Probes
MBeans
aliyun-log-producer-0-failure-batch-handler [main]
pool-1-thread-9 [main]
pool-1-thread-6 [main]
pool-1-thread-4 [main]
pool-1-thread-5 [main]
pool-1-thread-10 [main]
pool-1-thread-2 [main]
pool-1-thread-7 [main]
pool-1-thread-8 [main]
pool-1-thread-3 [main]
aliyun-log-producer-0-io-thread-2 [main]
aliyun-log-producer-0-io-thread-1 [main]
aliyun-log-producer-0-io-thread-3 [main]
aliyun-log-producer-0-io-thread-0 [main]
aliyun-log-producer-0-io-thread-7 [main]
aliyun-log-producer-0-io-thread-5 [main]
aliyun-log-producer-0-io-thread-6 [main]
aliyun-log-producer-0-io-thread-4 [main]
Thread-0 [main]

Tasks run by different threads have different state features.

- The thread `Pool-1-thread-<M>` periodically calls the `Producer.send ()` method to asynchronously send data. They continued to run while the application was starting, but was mostly blocked afterwards. The cause of this phenomenon is that Producer sends data slower than the data generation speed, and the cache size of each Producer instance is limited. After the startup of the application, Producer had enough memory to cache the data that is waiting to be sent, so `pool-1-thread-<M>` remained running for a while. This explains why the application had high CPU usage at startup. As time passed by, the cache memory is fully occupied. `pool-1-thread-<M>` must wait until Producer releases sufficient space. That is why a large number of threads are blocked.

- `aliyun-log-producer-0-mover` detects and sends expired batches to IOThreadPool. The data accumulation speed is fast, a producer batch is sent to IOThreadPool by `pool-1-thread-<M>` immediately after the cached data size reached the upper limit. Therefore, the mover thread remained idle most of the time.

- `aliyun-log-producer-0-io-thread-<N>` sends data from IOThreadPool to your designated logstore, and it takes most of the time on the network I/O status.

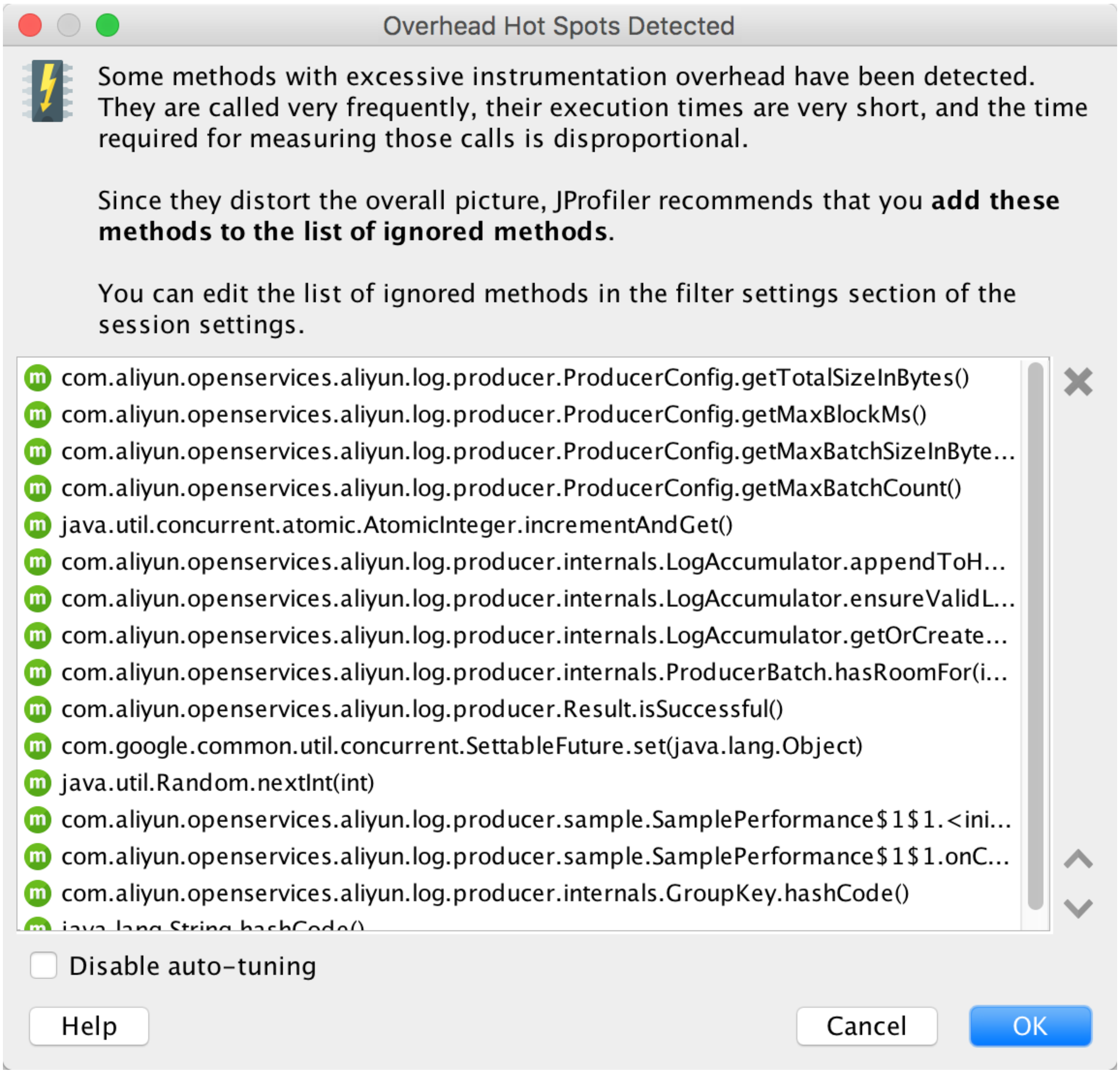
- `aliyun-log-producer-0-success-batch-handlerhandles` batches that were successfully sent to the logstore. The callback is simple and it takes a very short time to execute. Therefore, the SuccessBatchHandler remained idle most of the time.

- `aliyun-log-producer-0-failure-batch-handler` handles batches that failed to be sent to the logstore. In our case, no data has failed to be sent. It remained idle all the time.

According to our analysis, the statuses of these threads are within our expectation.

Overhead Hot Spots Detected

After the application finishes running, JProfiler displays a dialog box to show frequently called methods with very short running time. Next time, you can set JProfiler agent to ignore these methods to reduce the impact caused by JProfiler to the performance of the application.


Overhead Hot Spots Detected

Some methods with excessive instrumentation overhead have been detected. They are called very frequently, their execution times are very short, and the time required for measuring those calls is disproportional.

Since they distort the overall picture, JProfiler recommends that you **add these methods to the list of ignored methods**.

You can edit the list of ignored methods in the filter settings section of the session settings.

```
com.aliyun.openservices.aliyun.log.producer.ProducerConfig.getTotalSizeInBytes()
com.aliyun.openservices.aliyun.log.producer.ProducerConfig.getMaxBlockMs()
com.aliyun.openservices.aliyun.log.producer.ProducerConfig.getMaxBatchSizeInByte...
com.aliyun.openservices.aliyun.log.producer.ProducerConfig.getMaxBatchCount()
java.util.concurrent.atomic.AtomicInteger.incrementAndGet()
com.aliyun.openservices.aliyun.log.producer.internals.LogAccumulator.appendToH...
com.aliyun.openservices.aliyun.log.producer.internals.LogAccumulator.ensureValidL...
com.aliyun.openservices.aliyun.log.producer.internals.LogAccumulator.getOrCreate...
com.aliyun.openservices.aliyun.log.producer.internals.ProducerBatch.hasRoomFor(i...
com.aliyun.openservices.aliyun.log.producer.Result.isSuccessful()
com.google.common.util.concurrent.SettableFuture.set(java.lang.Object)
java.util.Random.nextInt(int)
com.aliyun.openservices.aliyun.log.producer.sample.SamplePerformance$1$1.<ini...
com.aliyun.openservices.aliyun.log.producer.sample.SamplePerformance$1$1.onC...
com.aliyun.openservices.aliyun.log.producer.internals.GroupKey.hashCode()
java.lang.String.hashCode()
```

☐ Disable auto-tuning

Summary

Based on the JProfiler diagnosis, the application does not have any significant performance problems or memory leak. The next step of optimization is to improve the serialization efficiency of objects.

References

- Jprofiler Introduction

Original Source

https://www.alibabacloud.com/blog/jprofiler-best-practices-powerful-performance-diagnostic-tool_594958?spm=a2c41.13091385.0.0

Programming Alibabacloud Java Jprofiler App Development