实战Java虚拟机-高级篇

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

GraalVM

目录

Contents

什么是GraalVM

- GraalVM是Oracle官方推出的一款高性能JDK，使用它享受比OpenJDK或者OracleJDK更好的性能。
- GraalVM的官方网址：https://www.graalvm.org/
- 官方标语：Build faster, smaller, leaner applications。

✓ 更低的CPU、内存使用率

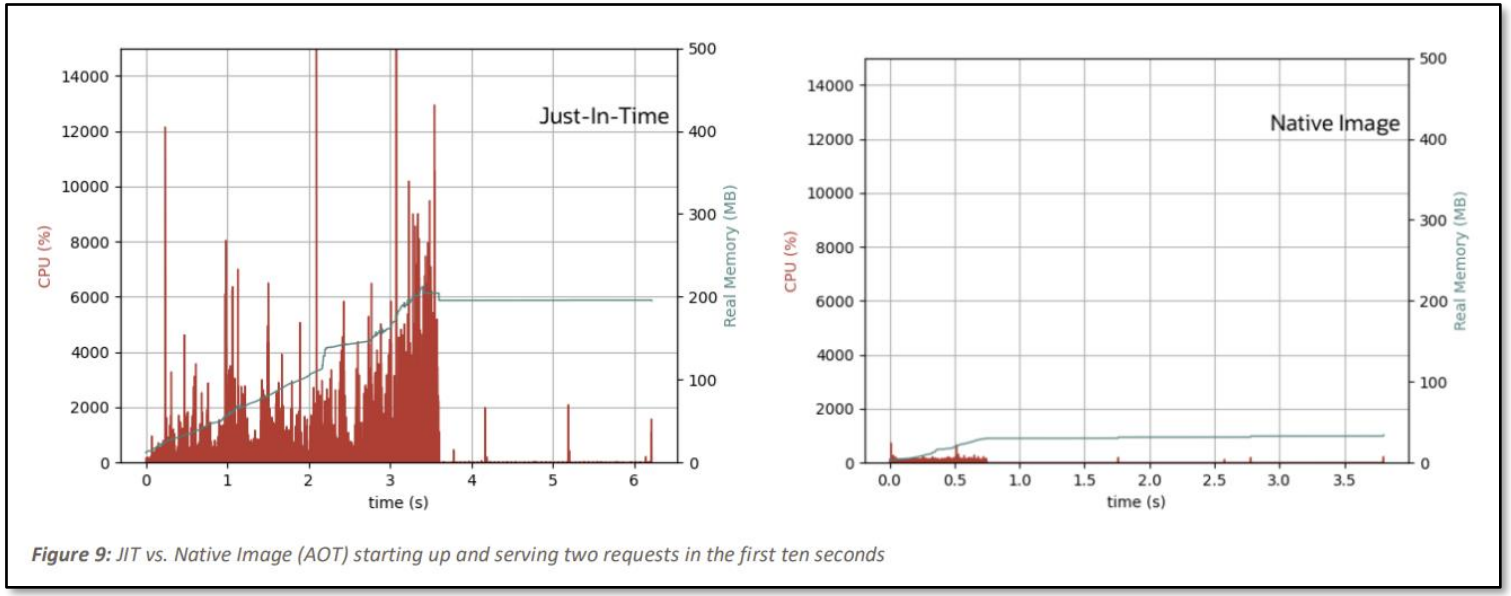
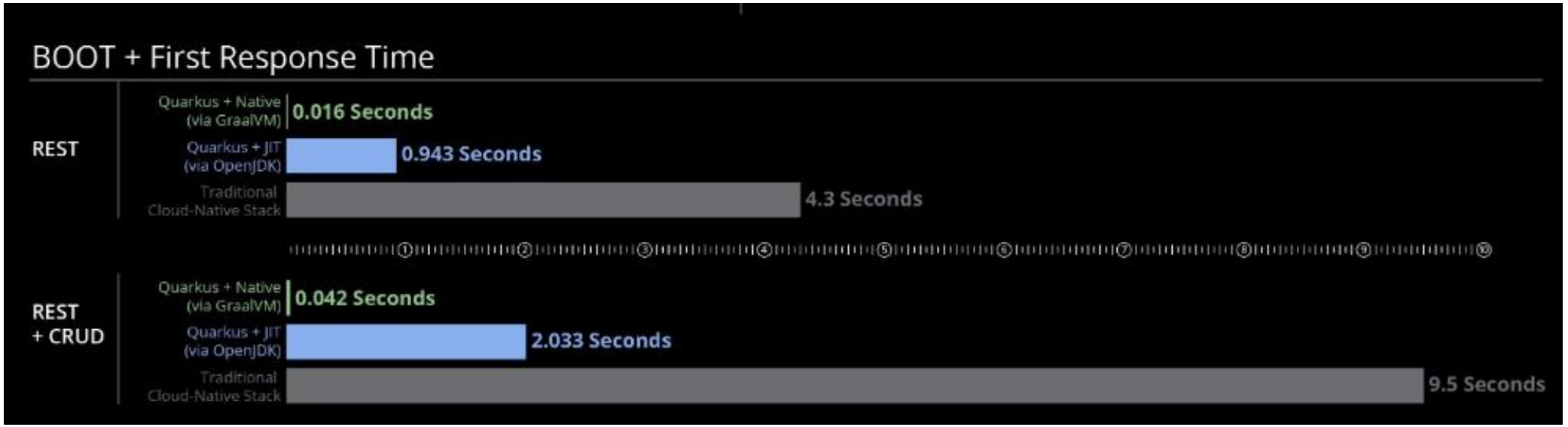


Figure 9: JIT vs. Native Image (AOT) starting up and serving two requests in the first ten seconds

使用GraalVM的本地镜像优化CPU和内存使用率

什么是GraalVM

- 官方标语：Build faster, smaller, leaner applications。
- ✓ 更低的CPU、内存使用率
- ✓ 更快的启动速度，无需预热即可获得最好的性能



使用GraalVM提升启动速度和第一次的接口响应时间

什么是GraalVM

- 官方标语：Build faster, smaller, leaner applications。

✓ 更低的CPU、内存使用率

✓ 更快的启动速度，无需预热即可获得最好的性能

✓ 更好的安全性、更小的可执行文件

✓ 支持多种框架Spring Boot、Micronaut、Helidon 和 Quarkus。

✓ 多家云平台支持。

✓ 通过Truffle框架运行JS、Python、Ruby等其他语言。

GraalVM的版本

- GraalVM分为社区版（Community Edition）和企业版（Enterprise Edition）。企业版相比较社区版，在性能上有更多的优化。

特性	描述	社区版	企业版
收费	是否收费	免费	收费
G1垃圾回收器	使用G1垃圾回收器优化垃圾回收性能	×	√
Profile Guided Optimization（PGO）	运行过程中收集动态数据，进一步优化本地镜像的性能	×	√
高级优化特性	更多优化技术，降低内存和垃圾回收的开销	×	√
高级优化参数	更多的高级优化参数可以设置	×	√

案例

GraalVM社区版环境搭建

需求：

搭建Linux下的GraalVM社区版本环境。

步骤：

1、使用arch查看Linux架构

```
[root@iZ2ze66duhxnyqu79qb2fkZ ~]#  arch
x86_64
```

2、根据架构下载社区版的GraalVM：https://www.graalvm.org/downloads/

3、安装GraalVM，安装方式与安装JDK相同。

4、使用java -version和HelloWorld测试GraalVM。

目录

Contents

GraalVM的两种模式

- JIT（Just-In-Time）模式，即时编译模式

JIT模式的处理方式与Oracle JDK类似，满足两个特点：

✓ Write Once,Run Anywhere -> 一次编写，到处运行。

✓ 预热之后，通过内置的Graal即时编译器优化热点代码，生成比Hotspot JIT更高性能的机器码。

案例

GraalVM 性能测试

需求：

分别在JDK8 、 JDK21 、 GraalVM 21 Graal即时编译器、GraalVM 21 不开启Graal即时编译器运行Jmh性能测试用例，对比其性能。

步骤：

1、在代码文件夹中找到GraalVM的案例代码，将java-simple-stream-benchmark文件夹下的代码使用maven打包成jar包。

2、将jar包上传到服务器，使用不同的JDK进行测试，对比结果。

注意：

-XX:-UseJVMCICompiler参数可以关闭GraalVM中的Graal编译器。

GraalVM的两种模式

- AOT（Ahead-Of-Time）模式，提前编译模式

AOT 编译器通过源代码，为特定平台创建可执行文件。比如，在Windows下编译完成之后，会生成exe文件。通过这种方式，达到启动之后获得最高性能的目的。但是不具备跨平台特性，不同平台使用需要单独编译。

这种模式生成的文件称之为Native Image本地镜像。

Java语言

```
//...public static void main etc

public static int add(int a, int b) {
  return a + b;
}

//...
```

【源代码】

AOT编译

```
0x000000011103cd00: mov 0x8(%rsi),%r10d
0x000000011103cd04: shl $0x3,%r10
0x000000011103cd08: cmp %r10,%rax
0x000000011103cd0b: jne 0x0000000111008b60  ;   {runtime_call}
0x000000011103cd11: data32 xchg %ax,%ax
0x000000011103cd14: nopl 0x0(%rax,%rax,1)
0x000000011103cd1c: data32 data32 xchg %ax,%ax
[Verified Entry Point]
0x000000011103cd20: sub $0x18,%rsp
0x000000011103cd27: mov %rbp,0x10(%rsp)  ;*synchronization entry
                                         ; - SandboxTest::a@-1 (line 5)
```

【windows机器码】本地镜像

Windows

AOT编译

```
0x000000011103cd00: mov 0x8(%rsi),%r10d
0x000000011103cd04: shl $0x3,%r10
0x000000011103cd08: cmp %r10,%rax
0x000000011103cd0b: jne 0x0000000111008b60  ;   {runtime_call}
0x000000011103cd11: data32 xchg %ax,%ax
0x000000011103cd14: nopl 0x0(%rax,%rax,1)
0x000000011103cd1c: data32 data32 xchg %ax,%ax
[Verified Entry Point]
0x000000011103cd20: sub $0x18,%rsp
0x000000011103cd27: mov %rbp,0x10(%rsp)  ;*synchronization entry
                                         ; - SandboxTest::a@-1 (line 5)
```

【linux机器码】本地镜像

Linux

案例

GraalVM AOT模式

需求：

使用GraalVM AOT模式制作本地镜像并运行。

步骤：

1、安装Linux环境本地镜像制作需要的依赖库：

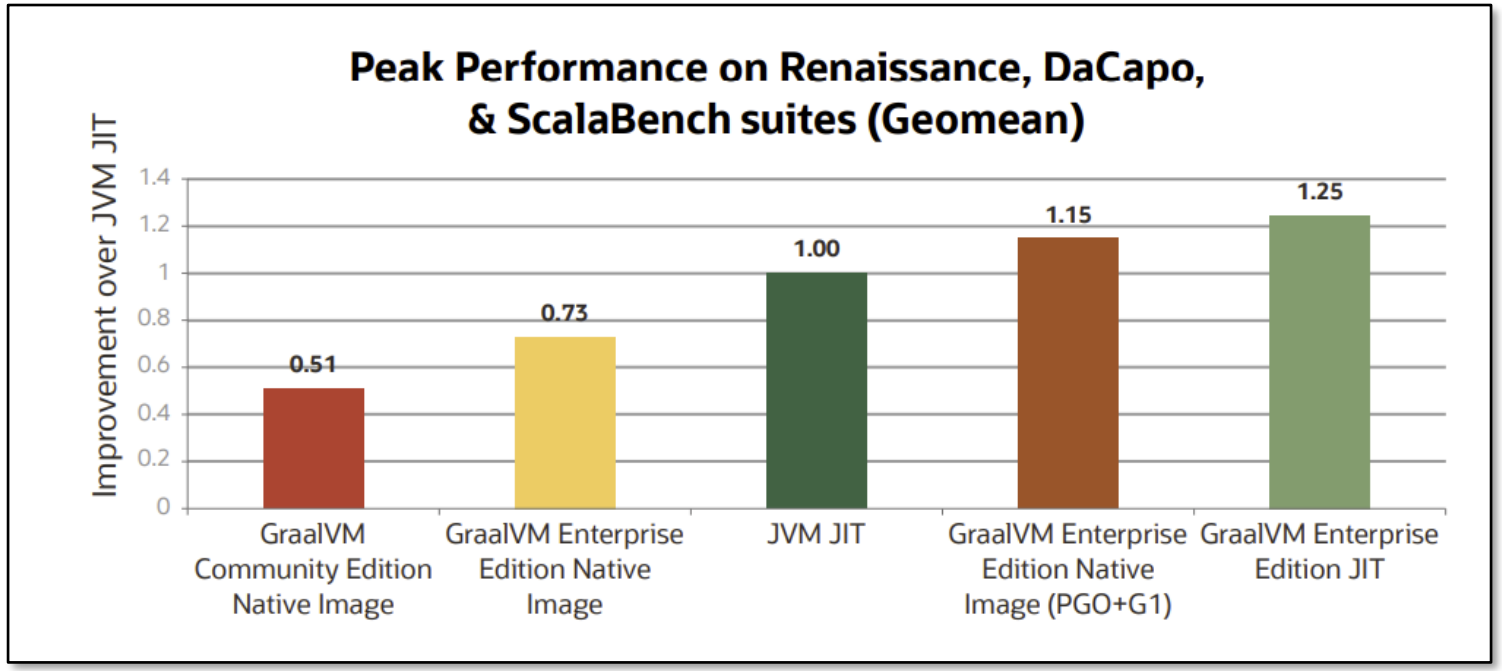
https://www.graalvm.org/latest/reference-manual/native-image/#prerequisites

2、使用 native-image 类名 制作本地镜像。

3、运行本地镜像可执行文件。

GraalVM模式和版本的性能对比

社区版的GraalVM使用本地镜像模式性能不如Hotspot JVM的JIT模式，但是企业版的性能相对会高很多。



性能对比

目录

Contents

GraalVM存在的问题

GraalVM的AOT模式虽然在启动速度、内存和CPU开销上非常有优势，但是使用这种技术会带来几个问题：

1、跨平台问题，在不同平台下运行需要编译多次。编译平台的依赖库等环境要与运行平台保持一致。

2、使用框架之后，编译本地镜像的时间比较长，同时也需要消耗大量的CPU和内存。

3、AOT 编译器在编译时，需要知道运行时所有可访问的所有类。但是Java中有一些技术可以在运行时创建类，例如反射、动态代理等。这些技术在很多框架比如Spring中大量使用，所以框架需要对AOT编译器进行适配解决类似的问题。

解决方案：

1、使用公有云的Docker等容器化平台进行在线编译，确保编译环境和运行环境是一致的，同时解决了编译资源问题。

2、使用SpringBoot3等整合了GraalVM AOT模式的框架版本。

案例

实战案例1：使用SpringBoot3搭建GraalVM环境

需求：

SpringBoot3对GraalVM进行了完整的适配，所以编写GraalVM服务推荐使用SpringBoot3。

步骤：

1、使用 https://start.spring.io/ spring提供的在线生成器构建项目。

2、编写业务代码。

3、执行 mvn -Pnative clean native:compile 命令生成本地镜像。

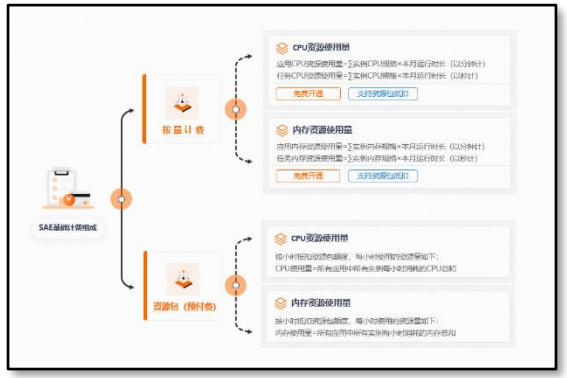
4、运行本地镜像。

思考

问题 什么场景下需要使用GraalVM呢?

1、对性能要求比较高的场景，可以选择使用收费的企业版提升性能。

2、公有云的部分服务是按照CPU和内存使用量进行计费的，使用GraalVM可以有效地降低费用。



阿里云Serverless

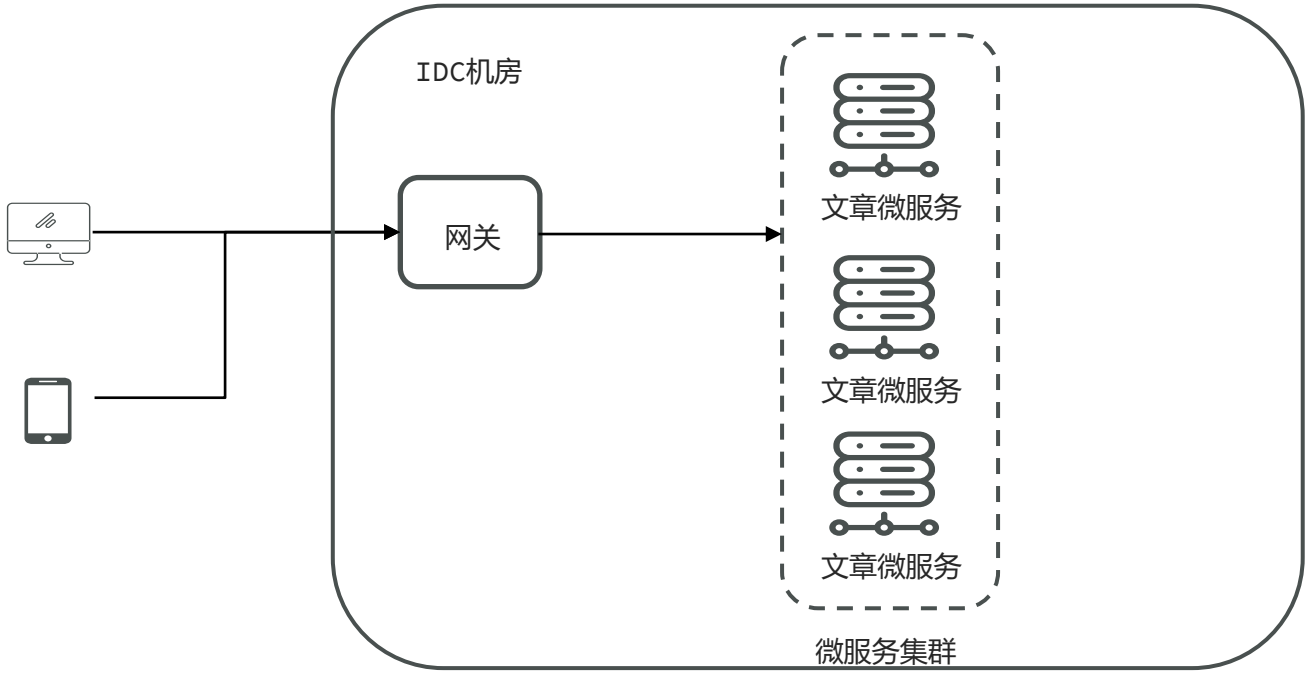
阿里云K8s集群ECI实例

GraalVM企业级应用 - Serverless架构

传统的系统架构中，服务器等基础设施的运维、安全、高可用等工作都需要企业自行完成，存在两个主要问题：

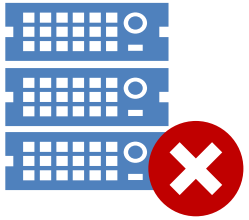
1、开销大，包括了人力的开销、机房建设的开销。

2、资源浪费，面对一些突发的流量冲击，比如秒杀等活动，必须提前规划好容量准备好大量的服务器，这些服务器在其他时候会处于闲置的状态，造成大量的浪费。



Serverless架构

随着虚拟化技术、云原生技术的愈发成熟，云服务商提供了一套称为Serverless无服务器化的架构。企业无需进行服务器的任何配置和部署，完全由云服务商提供。比较典型的有亚马逊AWS、阿里云等。



无需配置、部署服务器

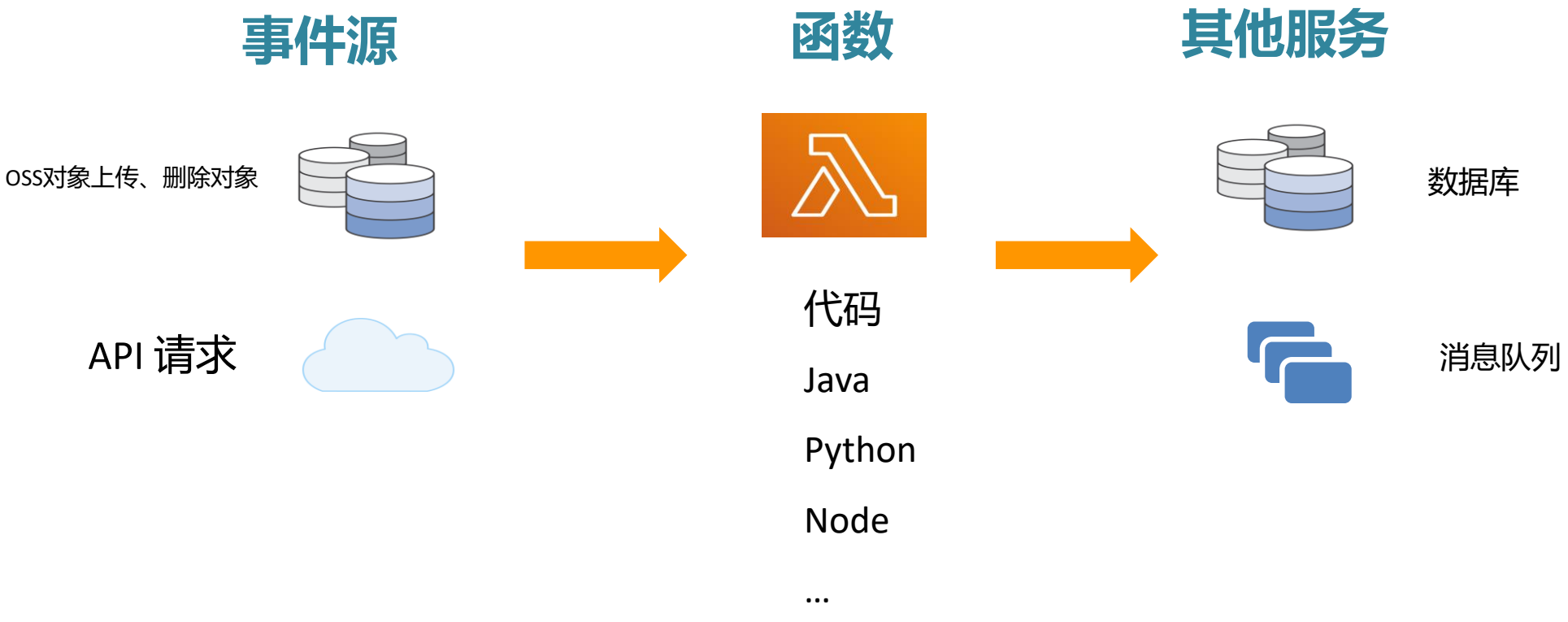
自动扩容

按价值付费

高可用与容错

Serverless架构 – 函数计算

Serverless架构中第一种常见的服务是函数计算（Function as a Service），将一个应用拆分成多个函数，每个函数会以事件驱动的方式触发。典型代表有AWS的Lambda、阿里云的FC。



Serverless架构 – 函数计算

函数计算主要应用场景有如下几种：

① 小程序、API服务中的接口，此类接口的调用频率不高，使用常规的服务器架构容易产生资源浪费，使用Serverless就可以实现按需付费降低成本，同时支持自动伸缩能应对流量的突发情况。

② 大规模任务的处理，比如音视频文件转码、审核等，可以利用事件机制当文件上传之后，自动触发对应的任务。

函数计算的计费标准中包含CPU和内存使用量，所以使用GraalVM AOT模式编译出来的本地镜像可以节省更多的成本。

案例

实战案例2：将程序部署到阿里云函数计算

步骤：

1、在项目中编写Dockerfile文件。

2、使用服务器制作镜像，这一步会消耗大量的CPU和内存资源，同时GraalVM相关的镜像服务器在国外，建议使用阿里云的镜像服务器制作Docker镜像。

3、使用函数计算将Docker镜像转换成函数服务。

4、绑定域名并进行测试。

Serverless架构 – Serverless应用

函数计算的服务资源比较受限，比如AWS的Lambda服务一般无法支持超过15分钟的函数执行，所以云服务商提供了另外一套方案：基于容器的Serverless应用，无需手动配置K8s中的Pod、Service等内容，只需选择镜像就可自动生成应用服务。

同样，Serverless应用的计费标准中包含CPU和内存使用量，所以使用GraalVM AOT模式编译出来的本地镜像可以节省更多的成本。

服务分类	交付模式	弹性效率	计费模式
函数计算	函数	毫秒级	调用次数 CPU内存使用量
Serverless应用	镜像容器	秒级	CPU内存使用量

案例

实战案例3：将程序部署到阿里云Serverless应用

步骤：

1、在项目中编写Dockerfile文件。

2、使用服务器制作镜像，这一步会消耗大量的CPU和内存资源，同时GraalVM相关的镜像服务器在国外，建议使用阿里云的镜像服务器制作Docker镜像。

前两步同实战案例2

3、配置Serverless应用，选择容器镜像、CPU和内存。

4、绑定外网负载均衡并使用Postman进行测试。

目录

Contents

GraalVM的内存参数

由于GraalVM是一款独立的JDK，所以大部分HotSpot中的虚拟机参数都不适用。常用的参数参考：官方手册。

- 社区版只能使用串行垃圾回收器（Serial GC），使用串行垃圾回收器的默认最大 Java 堆大小会设置为物理内存大小的 80%，调整方式为使用 -Xmx最大堆大小。如果希望在编译期就指定该大小，可以在编译时添加参数-R:MaxHeapSize=最大堆大小。
- G1垃圾回收器只能在企业版中使用，开启方式为添加--gc=G1参数，有效降低垃圾回收的延迟。
- 另外提供一个Epsilon GC，开启方式：--gc=epsilon ，它不会产生任何的垃圾回收行为所以没有额外的内存、CPU开销。如果在公有云上运行的程序生命周期短暂不产生大量的对象，可以使用该垃圾回收器，以节省最大的资源。

-XX:+PrintGC -XX:+VerboseGC 参数打印垃圾回收详细信息。

案例

实战案例4：内存快照文件的获取

需求：

获得运行中的内存快照文件，使用MAT进行分析。

步骤：

1、编译程序时，添加 --enable-monitoring=heapdump，参数添加到pom文件的对应插件中。

2、运行中使用 kill -SIGUSR1 进程ID 命令，创建内存快照文件。

3、使用MAT分析内存快照文件。

案例

实战案例5：运行时数据的获取

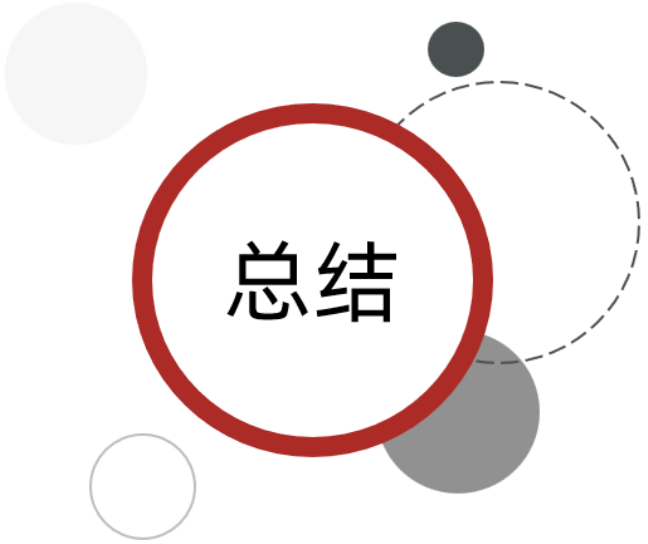
JDK Flight Recorder (JFR) 是一个内置于 JVM 中的工具，可以收集正在运行中的 Java 应用程序的诊断和分析数据，比如线程、异常等内容。GraalVM本地镜像也支持使用JFR生成运行时数据，导出的数据可以使用VisualVM分析。

步骤：

1、编译程序时，添加 --enable-monitoring=jfr，参数添加到pom文件的对应插件中。

2、运行程序，添加 -XX:StartFlightRecording=filename=recording.jfr,duration=10s参数。

3、使用VisualVM分析JFR记录文件。



1、什么是GraalVM?

GraalVM是Oracle官方推出的一款高性能JDK，具备两种模式，JIT模式使用方式与Oracel JDK相同；使用AOT模式制作的本地镜像，具备启动速度快、CPU和内存占用率低的优点，分为免费的社区版和收费的企业版，企业版拥有比社区版更好的性能。

2、什么场景下使用GraalVM?

① 希望拥有更好的性能，使用JIT模式或者升级为企业版。
② 执行时间较短的业务，使用GraalVM生成本地镜像，发布到函数计算云服务。
③ 执行时间较长的业务，比如长时间的计算任务，使用GraalVM生成本地镜像，发布到Serverless容器云服务。

02 新一代的GC

目录

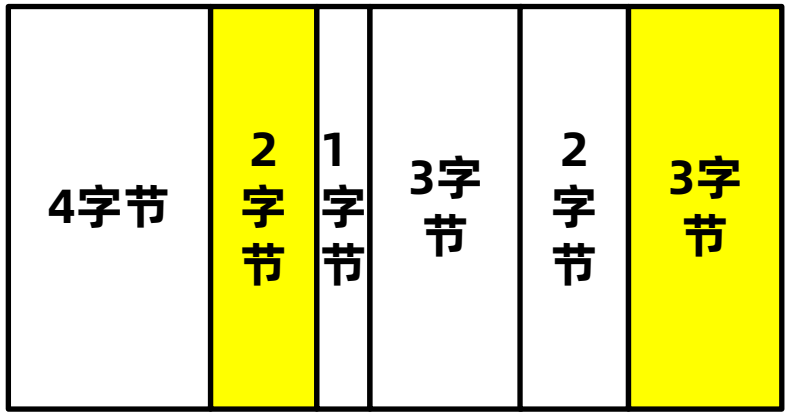
Contents

垃圾回收器的技术演进

STW
并行

	年轻代	老年代	
PS+PO	复制	标记	整理
ParNew+CMS	复制	并行标记	并行清理

可使用



CMS会产生内存碎片，无法分配4个字节空间。
解决方案：
1、产生FULLGC并且进行整理，此时会产生长时间STW。
2、退化成串行回收，产生长时间的STW。

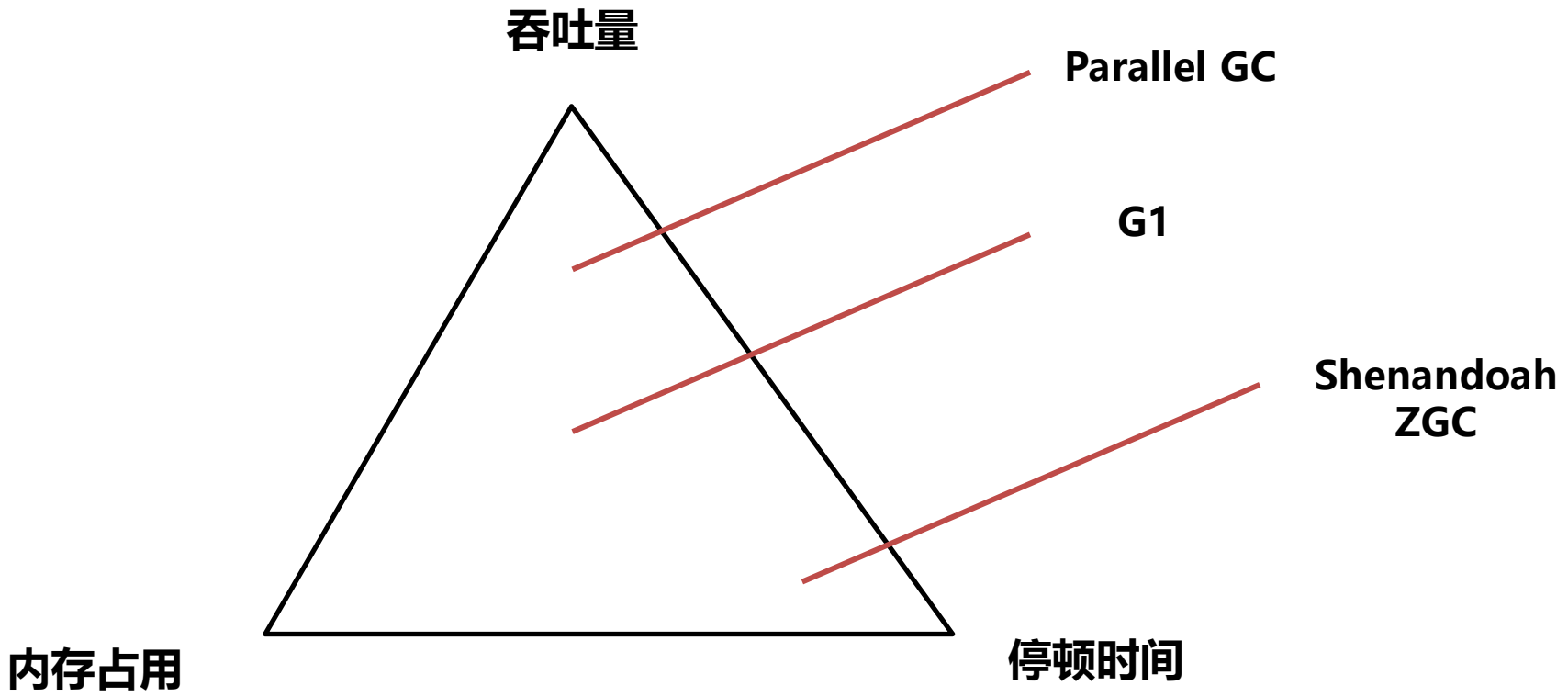
垃圾回收器的技术演进

STW
并行

	年轻代	老年代	
PS+PO	复制	标记	整理
ParNew+CMS	复制	并行标记	并行清理
G1	复制	并行标记	复制-整理
Shenandoah	并行标记	并行复制-整理	
ZGC（JDK21支持分代）	并行标记	并行复制-整理	

垃圾回收器的技术演进

不同的垃圾回收器设计的目标是不同的，如下图所示：

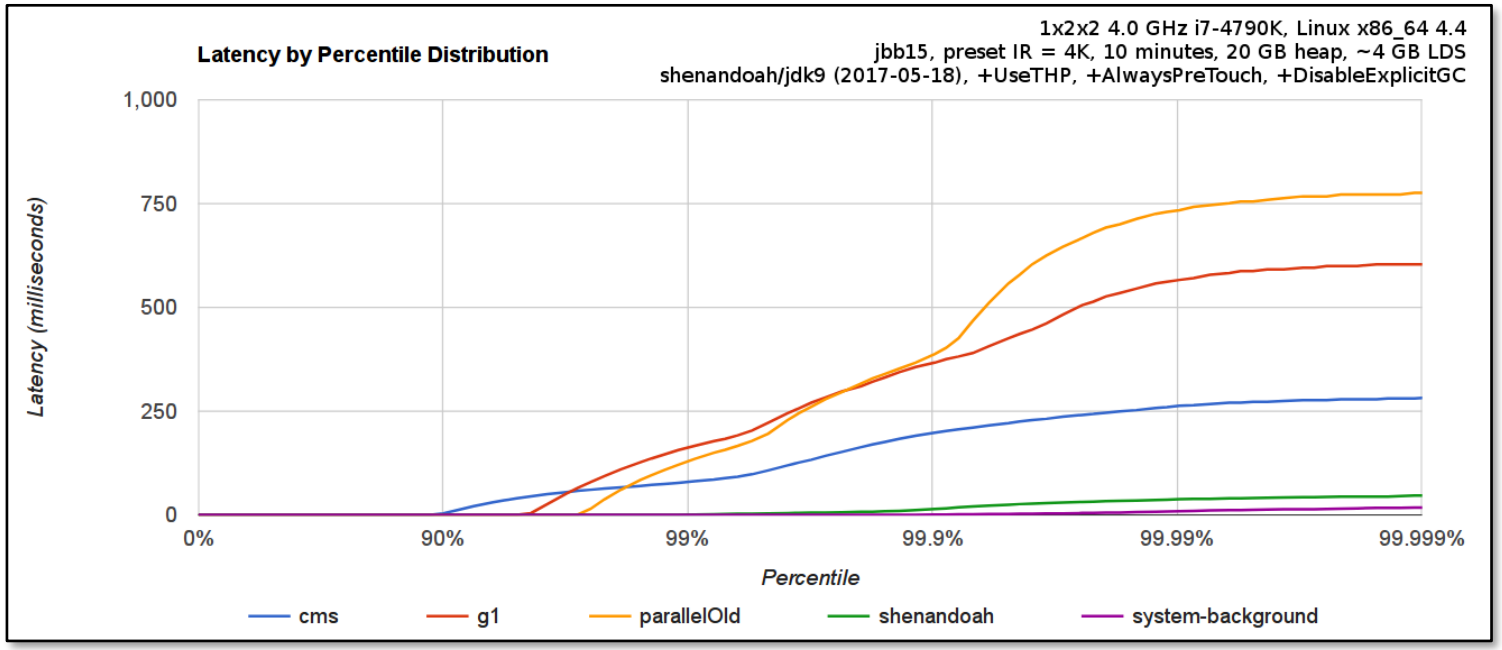


目录

Contents

什么是Shenandoah?

Shenandoah 是由Red Hat开发的一款低延迟的垃圾收集器，Shenandoah 并发执行大部分 GC 工作，包括并发的整理，堆大小对STW的时间基本没有影响。



OpenJDK测试数据

Shenandoah的使用方法

1、下载。Shenandoah只包含在OpenJDK中，默认不包含在内需要单独构建，可以直接下载构建好的。

下载地址：https://builds.shipilev.net/openjdk-jdk-shenandoah/

选择方式如下：

{aarch64, arm32-hflt, mipsel, mips64el, ppc64le, s390x, x86_32, x86_64}：架构，使用arch命令选择对应的的架构。

{server,zero}：虚拟机类型，选择server，包含所有GC的功能。

{release, fastdebug, Slowdebug, optimization}：不同的优化级别，选择release，性能最高。

{gcc*-glibc*, msvc*}：编译器的版本，选择较高的版本性能好一些，如果兼容性有问题（无法启动），选择较低的版本。

Shenandoah的使用方法

2、配置。将OpenJDK配置到环境变量中，使用java –version进行测试。打印出如下内容代表成功。

```
^C[root@iZ2ze66duhxnyqu79qb2fkZ jvm]# java -version
openjdk version "22-testing" 2024-03-19
OpenJDK Runtime Environment (build 22-testing-builds.shipilev.net-openjdk-jdk-shenandoah-b164-20231114)
OpenJDK 64-Bit Server VM (build 22-testing-builds.shipilev.net-openjdk-jdk-shenandoah-b164-20231114, mixed mode, sharing)
```

3、添加参数，运行Java程序。

-XX:+UseShenandoahGC 开启Shenandoah GC

-Xlog:gc 打印GC日志

```
[0.001s][info][gc] Min heap equals to max heap, disabling ShenandoahUncommit
[0.004s][info][gc] Using Shenandoah
[0.004s][info][gc] Heuristics ergonomically sets -XX:+ExplicitGCInvokesConcurrent
[0.004s][info][gc] Heuristics ergonomically sets -XX:+ShenandoahImplicitGCInvokesConcurrent
```

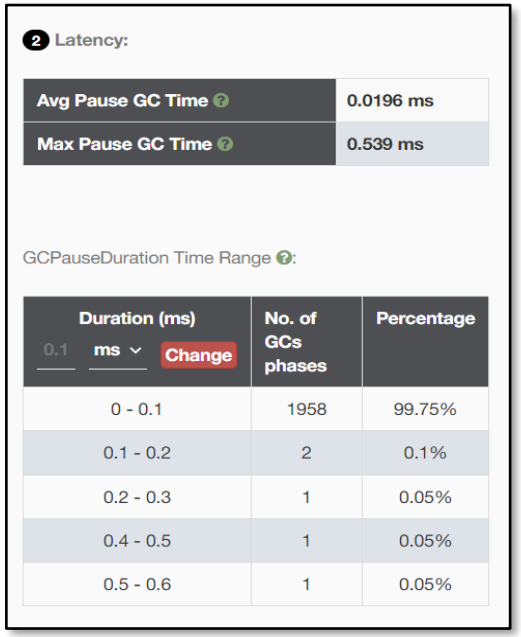
目录

Contents

什么是ZGC?

ZGC 是一种可扩展的低延迟垃圾回收器。ZGC 在垃圾回收过程中，STW的时间不会超过一毫秒，适合需要低延迟的应用。支持几百兆到16TB 的堆大小，堆大小对STW的时间基本没有影响。

ZGC降低了停顿时间，能降低接口的最大耗时，提升用户体验。但是吞吐量不佳，所以如果Java服务比较关注QPS（每秒的查询次数）那么G1是比较不错的选择。



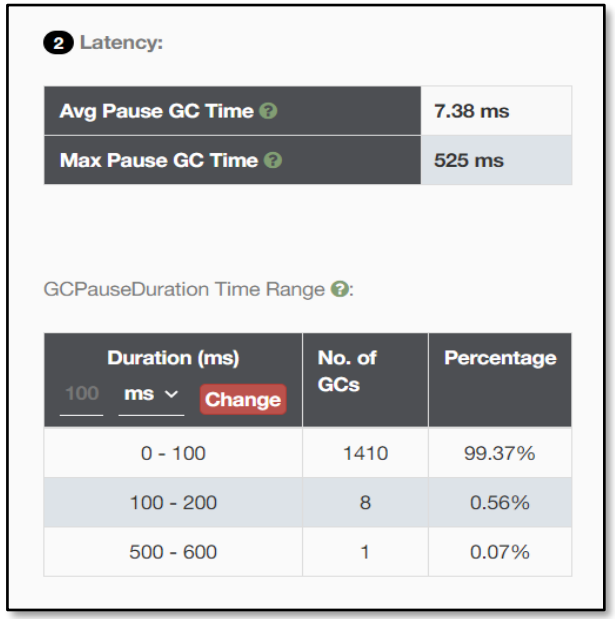
2 Latency:

Avg Pause GC Time	0.0196 ms
Max Pause GC Time	0.539 ms

GCPauseDuration Time Range:

Duration (ms) 0.1 ms Change	No. of GCs phases	Percentage
0 - 0.1	1958	99.75%
0.1 - 0.2	2	0.1%
0.2 - 0.3	1	0.05%
0.4 - 0.5	1	0.05%
0.5 - 0.6	1	0.05%

ZGC的停顿时间



2 Latency:

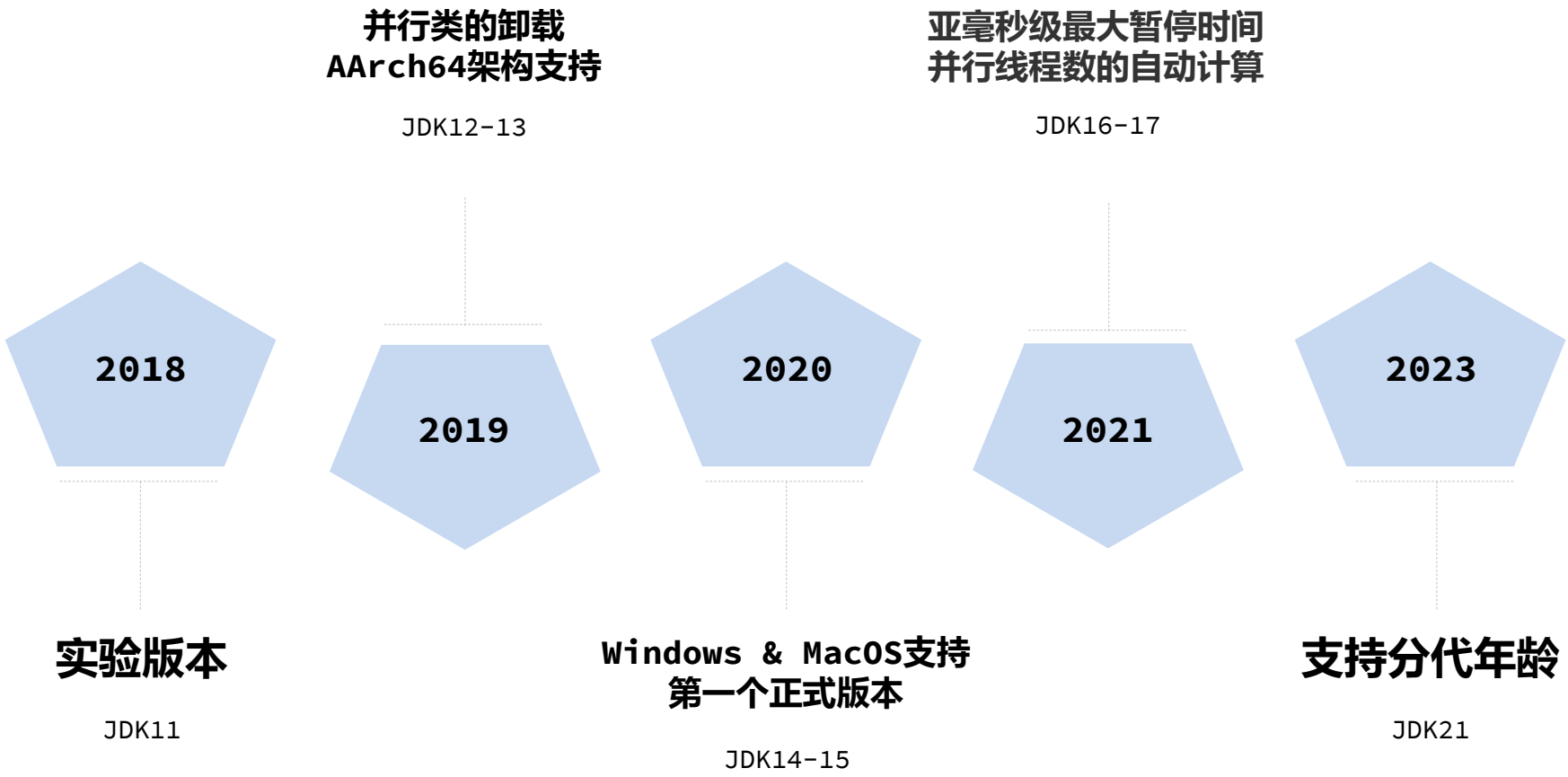
Avg Pause GC Time	7.38 ms
Max Pause GC Time	525 ms

GCPauseDuration Time Range:

Duration (ms) 100 ms Change	No. of GCs	Percentage
0 - 100	1410	99.37%
100 - 200	8	0.56%
500 - 600	1	0.07%

G1的停顿时间

ZGC的版本更迭


2018
实验版本
JDK11
2019
并行类的卸载
AArch64架构支持
JDK12-13
2020
Windows & MacOS支持
第一个正式版本
JDK14-15
2021
亚毫秒级最大暂停时间
并行线程数的自动计算
JDK16-17
2023
支持分代年龄
JDK21

ZGC的使用方法

OracleJDK和OpenJDK中都支持ZGC，阿里的DragonWell龙井JDK也支持ZGC但属于其自行对OpenJDK 11的ZGC进行优化的版本。

建议使用JDK17之后的版本，延迟较低同时无需手动配置并行线程数。

- 分代 ZGC添加如下参数启用 -XX:+UseZGC -XX:+ZGenerational
- 非分代 ZGC通过命令行选项启用 -XX:+UseZGC

ZGC的参数设置

ZGC在设计上做到了自适应，根据运行情况自动调整参数，让用户手动配置的参数最少化。

- 自动设置年轻代大小，无需设置-Xmn参数。
- 自动晋升阈值（复制中存活多少次才搬运到老年代），无需设置-XX:TenuringThreshold。
- JDK17之后支持自动的并行线程数，无需设置-XX:ConcGCThreads。

ZGC的参数设置

需要设置的参数：

-Xmx 值 最大堆内存大小

这是ZGC最重要的一个参数，必须设置。ZGC在运行过程中会使用一部分内存用来处理垃圾回收，所以尽量保证堆中有足够的空间。设置多少值取决于对象分配的速度，根据测试情况来决定。

可以设置的参数：

-XX:SoftMaxHeapSize=值

ZGC会尽量保证堆内存小于该值，这样在内存靠近这个值时会尽早地进行垃圾回收，但是依然有可能会超过该值。例如，-Xmx5g -XX:SoftMaxHeapSize=4g 这个参数设置，ZGC会尽量保证堆内存小于4GB，最多不会超过5GB。

ZGC的调优

ZGC 中可以使用Linux的Huge Page大页技术优化性能，提升吞吐量、降低延迟。

注意：安装过程需要 root 权限，所以ZGC默认没有开启此功能。

操作步骤：

1、计算所需页数，Linux x86架构中大页大小为2MB，根据所需堆内存的大小估算大页数量。比如堆空间需要16G，预留2G（JVM需要额外的一些非堆空间），那么页数就是18G / 2MB = 9216。

2、配置系统的大页池以具有所需的页数（需要root权限）：

```
$ echo 9216 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

3、添加参数-XX:+UseLargePages 启动程序进行测试

目录

Contents

案例

实战案例：内存不足时的垃圾回收测试

需求：

Java服务中存在大量软引用的缓存导致内存不足，测试下g1、Shenandoah、ZGC这三种垃圾回收器在这种场景下的回收情况。

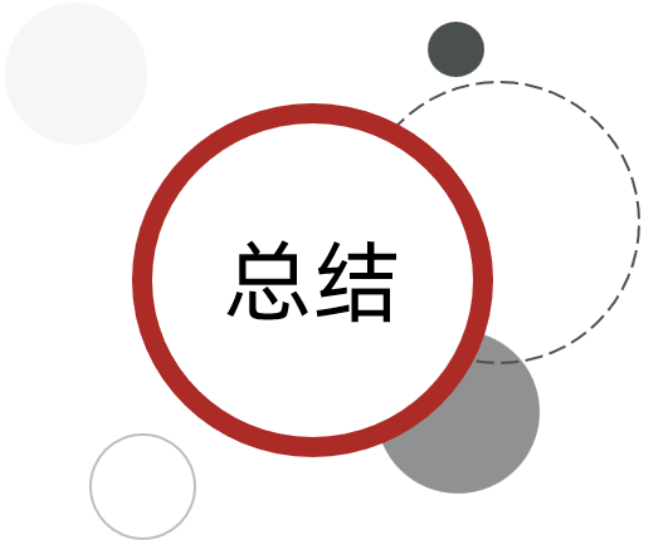
步骤：

1、启动程序，添加不同的虚拟机参数进行测试。

2、使用Apache Benchmark测试工具对本机进行压测。

3、生成GC日志，使用GcEasy进行分析。

4、对比压测之后的结果。



ZGC和Shenandoah设计的目标都是追求较短的停顿时间，他们具体的使用场景如下：

两种垃圾回收器在并行回收时都会使用垃圾回收线程占用CPU资源

① 在内存足够的情况下，ZGC垃圾回收表现的效果会更好，停顿时间更短。

② 在内存不是特别充足的情况下， Shenandoah GC表现更好，并行垃圾回收的时间较短，用户请求的执行效率比较高。

03 揭秘Java工具

Java工具的介绍

在Java的世界中，除了Java编写的业务系统之外，还有一类程序也需要Java程序员参与编写，这类程序就是Java工具。

常见的Java工具有以下几类：

1、诊断类工具，如Arthas、VisualVM等。

2、开发类工具，如Idea、Eclipse。

3、APM应用性能监测工具，如Skywalking、Zipkin等。

4、热部署工具，如Jrebel等。

揭秘Java工具

面试官

Arthas这款工具用到了什么Java技术，有没有了解过？

微服务架构中常用的APM系统，可以监控微服务的性能，你知道他是怎么实现的吗？

我们想实现动态检测Java程序运行情况的系统，你有类似的经历吗？

揭秘Java工具 – 学习路线

01

学习Java工具常用技术
Java Agent

02

了解Arthas、
VisualVM、APM系统等
底层实现

03

实战案例

揭秘Java工具 – 实战案例1

03

实战案例1
简化版Arthas

```
菜单:
1、查看内存使用情况
2、生成堆内存快照
3、打印栈信息
4、打印类加载器
5、打印类源码
6、打印类的参数和耗时
7、退出
```

```
堆内存:
name:G1 Eden Space used:3m max:0m committed:21m
name:G1 Old Gen used:1030m max:1960m committed:1031m
name:G1 Survivor Space used:0m max:0m committed:1m
非堆内存:
name:CodeHeap 'non-nmethods' used:1m max:5m committed:2m
name:Metaspace used:44m max:0m committed:44m
name:CodeHeap 'profiled nmethods' used:8m max:117m committed:9m
name:Compressed Class Space used:5m max:1024m committed:5m
name:CodeHeap 'non-profiled nmethods' used:2m max:117m committed:3m
name:mapped used:0m max:0m
name:direct used:0m max:0m
name:mapped - 'non-volatile memory' used:0m max:0m
```

```
name:http-nio-8882-exec-47 threadId:77 state:WAITING
java.base@22-testing/jdk.internal.misc.Unsafe.park(Native Method)
java.base@22-testing/java.util.concurrent.locks.LockSupport.park(LockSupport.java:371)
java.base@22-testing/java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionNode.block
java.base@22-testing/java.util.concurrent.ForkJoinPool.unmanagedBlock(ForkJoinPool.java:3995)
java.base@22-testing/java.util.concurrent.ForkJoinPool.managedBlock(ForkJoinPool.java:3943)
java.base@22-testing/java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awa
java.base@22-testing/java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:43
org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:141)
org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:33)
org.apache.tomcat.util.threads.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1114)
org.apache.tomcat.util.threads.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1176)
org.apache.tomcat.util.threads.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:659)
org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
```

揭秘Java工具 – 实战案例2

03

实战案例2
性能监控工具

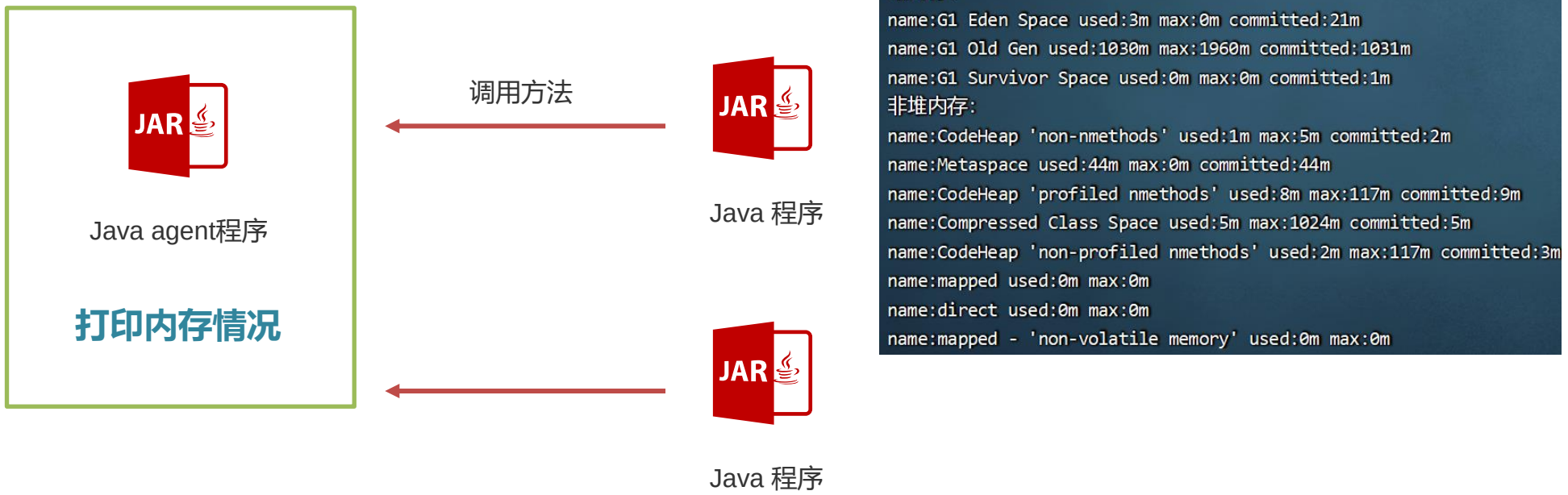
```
<clinit>@com.itheima.jvmoptimize.controller.LeakController 耗时为：0.15458毫秒
<init>@com.itheima.jvmoptimize.controller.LeakController 耗时为：0.06758毫秒
<init>@com.itheima.jvmoptimize.controller.LeakController2 耗时为：0.13528毫秒
<clinit>@com.itheima.jvmoptimize.controller.OOMController 耗时为：0.10762毫秒
<init>@com.itheima.jvmoptimize.controller.OOMController 耗时为：0.05803毫秒
init@com.itheima.jvmoptimize.controller.OOMController 耗时为：0.76496毫秒
<clinit>@com.itheima.jvmoptimize.fullgcdemo.Demo1Controller 耗时为：0.13759毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.Demo1Controller 耗时为：502.50022毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.Demo2Controller 耗时为：46.54233毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.DemoMetaspaceController 耗时为：0.0714毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.Practice 耗时为：0.06785毫秒
<init>@com.itheima.jvmoptimize.performance.PerformanceController 耗时为：0.24018毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.Demo1ArticleController 耗时为：0.07155毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.Demo2ExcelController 耗时为：0.06679毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.DemoQueryController 耗时为：0.05996毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.DemoSqlJointController 耗时为：0.07246毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.DemoThreadLocal 耗时为：0.05967毫秒
<init>@org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController 耗时为：0.38511毫秒
<clinit>@com.itheima.jvmoptimize.controller.LeakController 耗时为：0.15472毫秒
<init>@com.itheima.jvmoptimize.controller.LeakController 耗时为：0.07593毫秒
<init>@com.itheima.jvmoptimize.controller.LeakController2 耗时为：0.16376毫秒
<clinit>@com.itheima.jvmoptimize.controller.OOMController 耗时为：0.12544毫秒
<init>@com.itheima.jvmoptimize.controller.OOMController 耗时为：0.05954毫秒
init@com.itheima.jvmoptimize.controller.OOMController 耗时为：0.93953毫秒
<clinit>@com.itheima.jvmoptimize.fullgcdemo.Demo1Controller 耗时为：0.15222毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.Demo1Controller 耗时为：650.10421毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.Demo2Controller 耗时为：45.36436毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.DemoMetaspaceController 耗时为：0.06811毫秒
<init>@com.itheima.jvmoptimize.fullgcdemo.Practice 耗时为：0.06815毫秒
<init>@com.itheima.jvmoptimize.performance.PerformanceController 耗时为：0.2463毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.Demo1ArticleController 耗时为：0.0721毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.Demo2ExcelController 耗时为：0.06626毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.DemoQueryController 耗时为：0.06877毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.DemoSqlJointController 耗时为：0.0693毫秒
<init>@com.itheima.jvmoptimize.practice.oom.controller.DemoThreadLocal 耗时为：0.0739毫秒
<init>@org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController 耗时为：0.39519毫秒
test@com.itheima.jvmoptimize.fullgcdemo.Demo2Controller 耗时为：86.66299毫秒
```

目录

Contents

Java Agent技术

Java Agent技术是JDK提供的用来编写Java工具的技术，使用这种技术生成一种特殊的jar包，这种jar包可以让Java程序运行其中的代码。



Java Agent技术的两种模式

Java Agent技术实现了让Java程序执行独立的Java Agent程序中的代码，执行方式有两种：

- 静态加载模式
- 动态加载模式

Java Agent技术的两种模式 - 静态加载模式

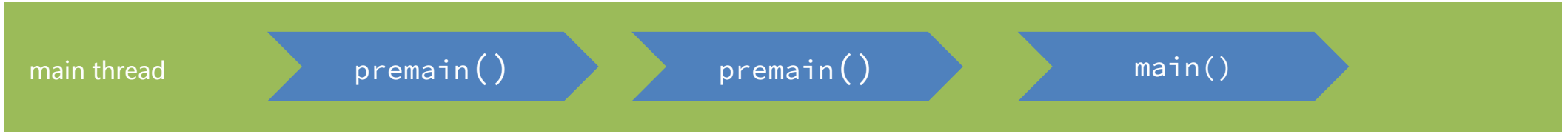
静态加载模式可以在程序启动的一开始就执行我们需要执行的代码，适合用APM等性能监测系统从一开始就监控程序的执行性能。静态加载模式需要在Java Agent的项目中编写一个premain的方法，并打包成jar包。

```
public static void premain(String agentArgs, Instrumentation inst)
```

接下来使用以下命令启动Java程序，此时Java虚拟机将会加载agent中的代码并执行。

```
java -javaagent:./agent.jar -jar test.jar
```

premain方法会在主线程中执行：



Java Agent技术的两种模式 - 动态加载模式

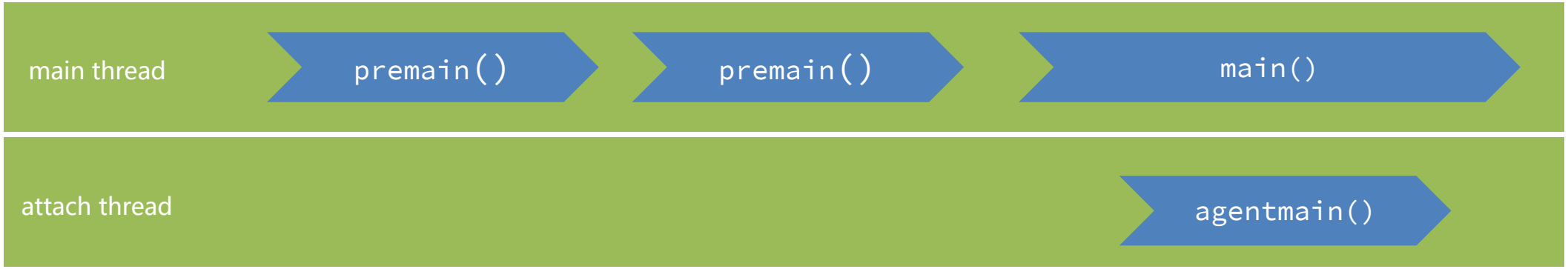
动态加载模式可以随时让java agent代码执行，适用于Arthas等诊断系统。动态加载模式需要在Java Agent的项目中编写一个agentmain的方法，并打包成jar包。

```
public static void agentmain(String agentArgs, Instrumentation inst)
```

接下来使用以下代码就可以让java agent代码在指定的java进程中执行了。

```
VirtualMachine vm = VirtualMachine.attach("24200"); //动态连接到24200进程ID的java程序
vm.loadAgent("itheima-jvm-java-agent-jar-with-dependencies.jar"); //加载java agent
```

agentmain方法会在独立线程中执行：



案例

搭建java agent静态加载模式的环境

步骤：

1、创建maven项目，添加maven-assembly-plugin插件，此插件可以打包出java agent的jar包。

2、编写类和premain方法，premain方法中打印一行信息。

3、编写MANIFEST.MF文件，此文件主要用于描述java agent的配置属性，比如使用哪一个类的premain方法。

4、使用maven-assembly-plugin进行打包。

5、创建spring boot应用，并静态加载上一步打包完的java agent。

案例

搭建java agent动态加载模式的环境

步骤：

1、创建maven项目，添加maven-assembly-plugin插件，此插件可以打包出java agent的jar包。

2、编写类和agentmain方法， agentmain方法中打印一行信息。

3、编写MANIFEST.MF文件，此文件主要用于描述java agent的配置属性，比如使用哪一个类的agentmain方法。

4、使用maven-assembly-plugin进行打包。

5、编写main方法，动态连接到运行中的java程序。

案例

实战案例1：简化版的Arthas

需求：

编写一个简化版的Arthas程序，具备以下几个功能：

1、查看内存使用情况

2、生成堆内存快照

3、打印栈信息

4、打印类加载器

5、打印类的源码

6、打印方法执行的参数和耗时

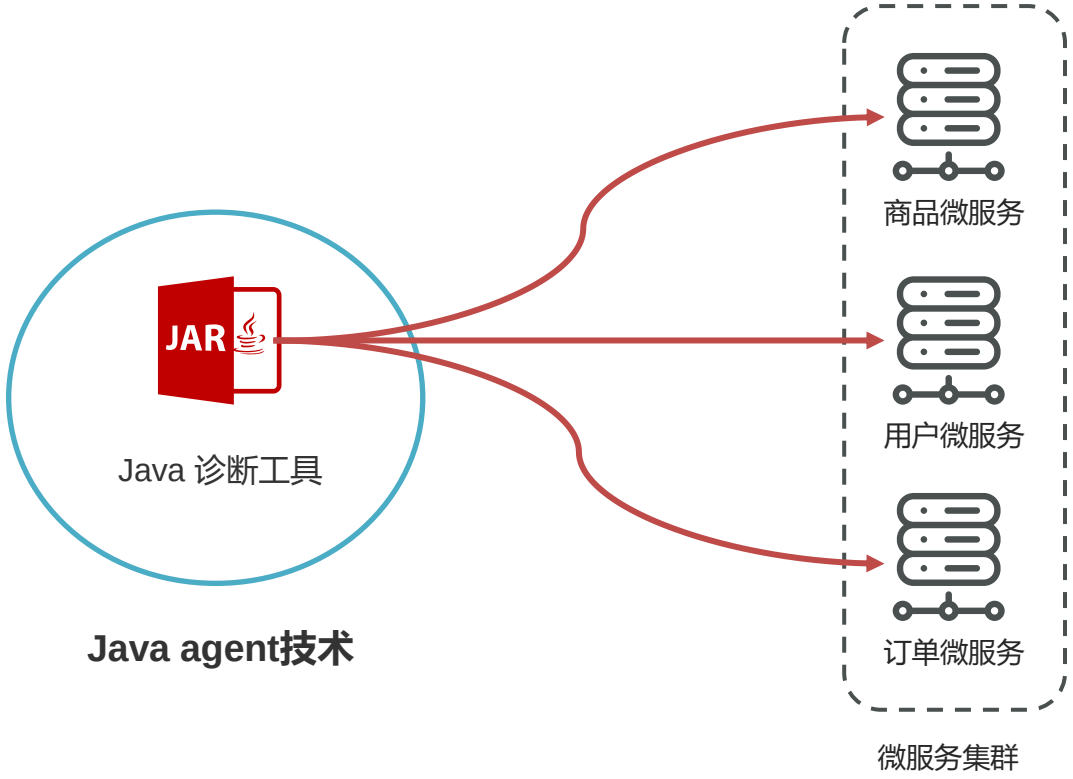
案例

实战案例1：简化版的Arthas

需求：

该程序是一个独立的Jar包，可以应用于任何Java编写的系统中。

具备以下特点：代码无侵入性、操作简单、性能高。



案例

实战案例1：简化版的Arthas

需求1：

编写一个简化版的Arthas程序，具备以下几个功能：

1、查看内存使用情况

2、生成堆内存快照

3、打印栈信息

4、打印类加载器

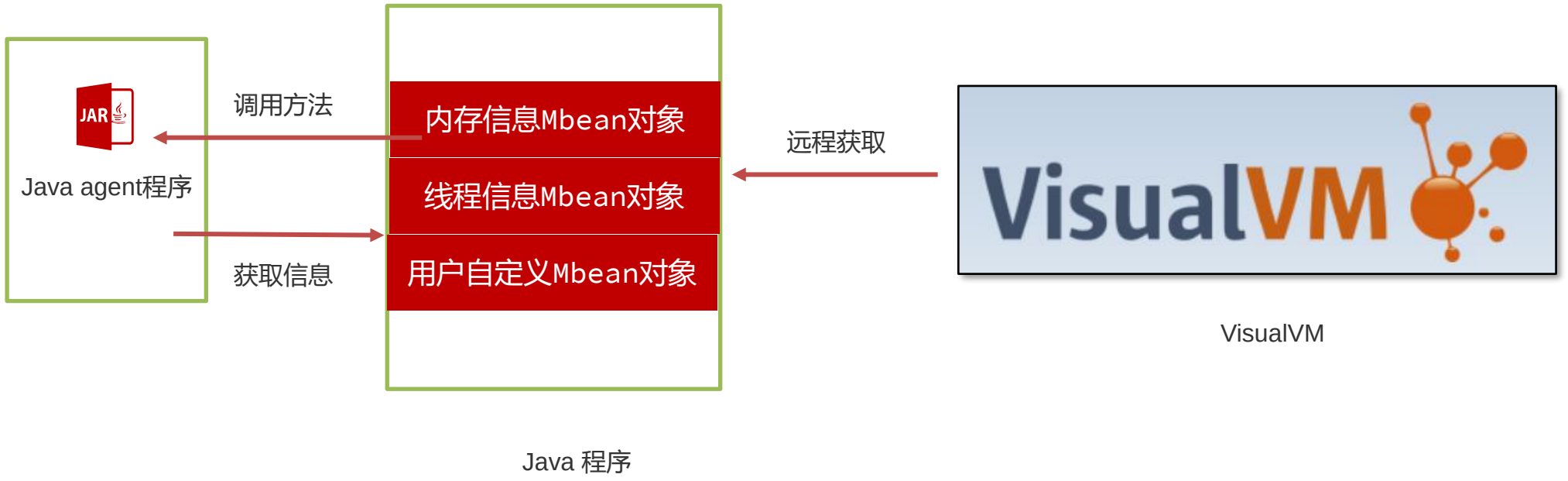
5、打印类的源码

6、打印方法执行的参数和耗时

获取运行时信息 - JMX技术

JDK从1.5开始提供了Java Management Extensions (JMX) 技术，通过Mbean对象的写入和获取，实现：

- 运行时配置的获取和更改
- 应用程序运行信息的获取（线程栈、内存、类信息等）

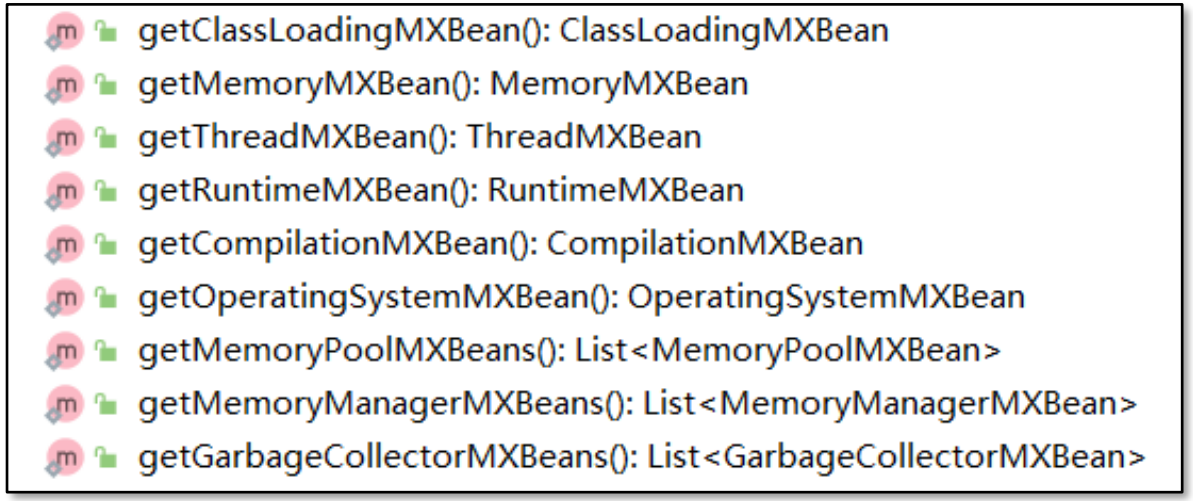


获取运行时信息 - JMX技术

获取JVM默认提供的Mbean可以通过如下的方式，例如获取内存信息：

```
ManagementFactory.getMemoryPoolMXBeans()
```

ManagementFactory提供了一系列的方法获取各种各样的信息：



类加载信息

线程信息

所有内存信息（堆+非堆）

垃圾回收器数量+回收时间

案例

实战案例1：简化版的Arthas

需求2：

编写一个简化版的Arthas程序，具备以下几个功能：

~~1、查看内存使用情况~~

2、查看直接内存使用情况，生成堆内存快照

3、打印栈信息

4、打印类加载器

5、打印类的源码

6、打印方法执行的参数和耗时

获取运行时信息 - JMX技术

更多的信息可以通过ManagementFactory.getPlatformMXBeans获取，比如：

```
Class bufferPoolMXBeanClass = Class.forName("java.lang.management.BufferPoolMXBean");
List<BufferPoolMXBean> bufferPoolMXBeans = ManagementFactory.getPlatformMXBeans(bufferPoolMXBeanClass);
```

通过这种方式，获取到了Java虚拟机中分配的直接内存和内存映射缓冲区的大小。

```
HotSpotDiagnosticMXBean hotSpotDiagnosticMXBean =
        ManagementFactory.getPlatformMXBean(HotSpotDiagnosticMXBean.class);
```

获取到虚拟机诊断用的MXBean，通过这个Bean对象可以生成内存快照。

案例

实战案例1：简化版的Arthas

需求2：

编写一个简化版的Arthas程序，具备以下几个功能：

~~1、查看内存使用情况~~

~~2、查看直接内存使用情况，生成堆内存快照~~

3、打印栈信息

4、打印类加载器

5、打印类的源码

6、打印方法执行的参数和耗时

案例

实战案例1：简化版的Arthas

需求2：

编写一个简化版的Arthas程序，具备以下几个功能：

~~1、查看内存使用情况~~

~~2、查看直接内存使用情况，生成堆内存快照~~

~~3、打印栈信息~~

4、打印类加载器

5、打印类的源码

6、打印方法执行的参数和耗时

获取类和类加载器的信息 - Instumentation对象

Java Agent中可以获得Java虚拟机提供的Instumentation对象：

```
public static void premain(String agentArgs, Instrumentation inst)
```

```
public static void agentmain(String agentArgs, Instrumentation inst)
```

该对象有以下几个作用：

1、redefine，重新设置类的字节码信息。

2、retransform，根据现有类的字节码信息进行增强。

3、获取所有已加载的类信息。

Oracle官方手册： https://docs.oracle.com/javase/17/docs/api/java/lang/instrument/Instrumentation.html

案例

实战案例1：简化版的Arthas

需求2：

编写一个简化版的Arthas程序，具备以下几个功能：

~~1、查看内存使用情况~~

~~2、查看直接内存使用情况，生成堆内存快照~~

~~3、打印栈信息~~

~~4、打印类加载器~~

5、打印类的源码

6、打印方法执行的参数和耗时

打印类的源码

打印类的源码需要分为以下几个步骤

1、获得内存中的类的字节码信息。利用Instrumentation提供的转换器来获取字节码信息。

2、通过反编译工具将字节码信息还原成源代码信息。

定义转换器
实现ClassFileTransformer接口

转换之后的字节码信息

```
default byte[]
transform(  ClassLoader          loader,
            String               className,
            Class<?>             classBeingRedefined,
            ProtectionDomain     protectionDomain,
            byte[]               classfileBuffer)
    throws IllegalClassFormatException {
    return null;
}
```

当前内的字节码信息

这里我们仅仅是去获取字节码信息，所以直接返回原始的字节码信息

打印类的源码

打印类的源码需要分为以下几个步骤

1、获得内存中的类的字节码信息。利用Instrumentation提供的转换器来获取字节码信息。

2、通过反编译工具将字节码信息还原成源代码信息。

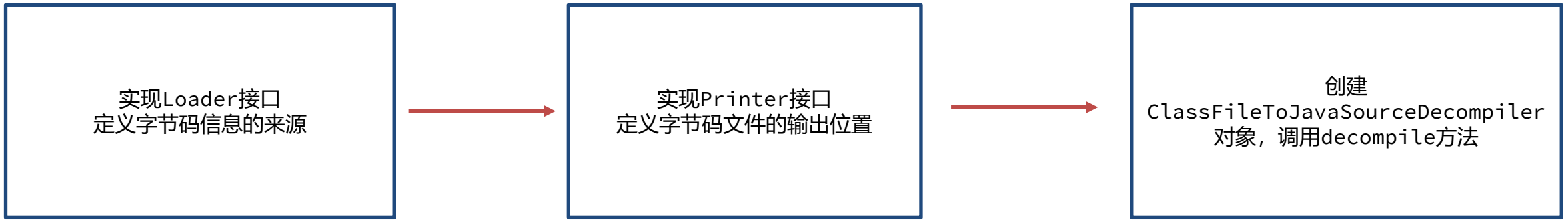
打印类的源码

打印类的源码需要分为以下几个步骤

1、获得内存中的类的字节码信息。利用Instrumentation提供的转换器来获取字节码信息。

2、通过反编译工具将字节码信息还原成源代码信息。

这里我们会使用jd-core依赖库来完成，github地址：https://github.com/java-decompiler/jd-core



案例

实战案例1：简化版的Arthas

需求2：

编写一个简化版的Arthas程序，具备以下几个功能：

~~1、查看内存使用情况~~

~~2、查看直接内存使用情况，生成堆内存快照~~

~~3、打印栈信息~~

~~4、打印类加载器~~

~~5、打印类的源码~~

6、打印方法执行的参数和耗时

问题	Spring AOP是不是也可以实现类似的功能呢?

Spring AOP 确实可以实现统计方法执行时间，打印方法参数等功能，但是使用这种方式存在几个问题：

① 代码有侵入性，AOP代码必须在当前项目中被引入才能完成相应的功能。

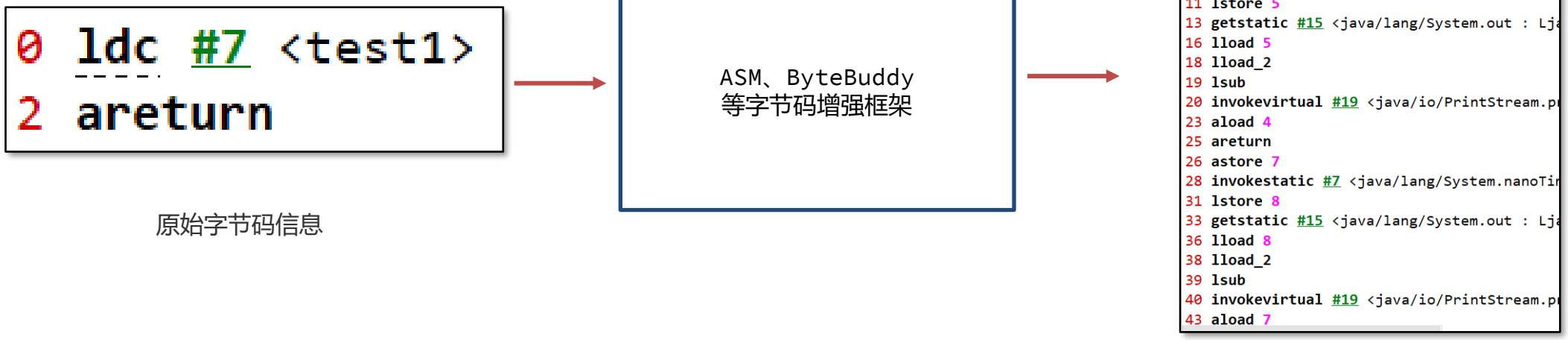
② 无法做到灵活地开启和关闭功能。

③ 与Spring框架强耦合，如果项目没有使用Spring框架就不可以使用。

所以使用Java Agent技术 + 字节码增强技术，就可以解决上述三个问题。

打印方法执行的参数和耗时

打印方法执行的参数和耗时需要对原始类的方法进行增强，可以使用类似于Spring AOP这类面向切面编程的方式，但是考虑到并非每个项目都使用了Spring这些框架，所以我们选择的是最基础的字节码增强框架。字节码增强框架是在当前类的字节码信息中插入一部分字节码指令，从而起到增强的作用。



原始字节码信息

增强后的字节码信息（打印执行时间）

打印方法执行的参数和耗时 - ASM

ASM是一个通用的 Java 字节码操作和分析框架。它可用于直接以二进制形式修改现有类或动态生成类。ASM重点关注性能。让操作尽可能小且尽可能快，所以它非常适合在动态系统中使用。ASM的缺点是代码复杂。

```
@Override
public void visitCode() {
    mv.visitFieldInsn(Opcodes.GETSTATIC, owner: "java/lang/System", name: "out", descriptor: "Ljava/io/PrintStream
    mv.visitLdcInsn( value: "开始执行");
    mv.visitMethodInsn(INVOKEVIRTUAL, owner: "java/io/PrintStream", name: "println", descriptor: "(Ljava/lang/Stri
    super.visitCode();
}

@Override
public void visitInsn(int opcode) {
    if(opcode == ARETURN || opcode == RETURN ) {
        mv.visitFieldInsn(Opcodes.GETSTATIC, owner: "java/lang/System", name: "out", descriptor: "Ljava/io/PrintSt
        mv.visitLdcInsn( value: "结束执行");
        mv.visitMethodInsn(INVOKEVIRTUAL, owner: "java/io/PrintStream", name: "println", descriptor: "(Ljava/lang/
    }
    super.visitInsn(opcode);
}

@Override
public void visitEnd() {
    mv.visitMaxs( maxStack: 20, maxLocals: 50);
    super.visitEnd();
}
```

利用ASM在方法进入和退出时打印两行文字

打印方法执行的参数和耗时 - ASM

ASM的官方网址：https://asm.ow2.io/

操作步骤：

1、引入依赖

```
<dependency>
    <groupId>org.ow2.asm</groupId>
    <artifactId>asm</artifactId>
    <version>9.6</version>
</dependency>
```

2、搭建基础框架，此代码为固定代码

```
ClassWriter cw = new ClassWriter( flags: 0);
// cv forwards all events to cw
ClassVisitor cv = new ClassVisitor(ASM7, cw) {
    @Override
    public MethodVisitor visitMethod(int access, String name, String descriptor, String signature, String[]
        MethodVisitor mv = cv.visitMethod(access, name, descriptor, signature, exceptions);
        return new MyMethodVisitor(this.api,mv);
    }
};
ClassReader cr = new ClassReader(bytes);
cr.accept(cv, parsingOptions: 0);
```

编写类描述如何对类进行增强

打印方法执行的参数和耗时 - ASM

ASM的官方网址：https://asm.ow2.io/

操作步骤：

3、编写一个类描述如何去增强类，类需要继承自MethodVisitor

```
class MyMethodVisitor extends MethodVisitor {

    public MyMethodVisitor(int api, MethodVisitor methodVisitor) { super(api, methodVisitor); }

    @Override
    public void visitCode() {
        mv.visitFieldInsn(Opcodes.GETSTATIC, owner: "java/lang/System", name: "out", descriptor: "Ljava/io/PrintStre
        mv.visitLdcInsn( value: "开始执行");
        mv.visitMethodInsn(INVOKEVIRTUAL, owner: "java/io/PrintStream", name: "println", descriptor: "(Ljava/lang/St
        super.visitCode();
    }

    @Override
    public void visitInsn(int opcode) {
        if(opcode == ARETURN || opcode == RETURN ) {
            mv.visitFieldInsn(Opcodes.GETSTATIC, owner: "java/lang/System", name: "out", descriptor: "Ljava/io/Print
            mv.visitLdcInsn( value: "结束执行");
            mv.visitMethodInsn(INVOKEVIRTUAL, owner: "java/io/PrintStream", name: "println", descriptor: "(Ljava/lan
        }
        super.visitInsn(opcode);
    }

    @Override
    public void visitEnd() {
        mv.visitMaxs( maxStack: 20, maxLocals: 50);
```

方法执行开始

方法产生特定的操作，比如返回

设定方法的最大栈深度和局部变量表的大小

打印方法执行的参数和耗时 - Byte Buddy

Byte Buddy 是一个代码生成和操作库，用于在 Java 应用程序运行时创建和修改 Java 类，而无需编译器的帮助。

Byte Buddy底层基于ASM，提供了非常方便的 API。

```
@Override
public void visitCode() {
    mv.visitFieldInsn(Opcodes.GETSTATIC, owner: "java/lang/System", name: "out", descriptor: "Ljava/io/PrintStream
    mv.visitLdcInsn( value: "开始执行");
    mv.visitMethodInsn(INVOKEVIRTUAL, owner: "java/io/PrintStream", name: "println", descriptor: "(Ljava/lang/Stri
    super.visitCode();
}

@Override
public void visitInsn(int opcode) {
    if(opcode == ARETURN || opcode == RETURN ) {
        mv.visitFieldInsn(Opcodes.GETSTATIC, owner: "java/lang/System", name: "out", descriptor: "Ljava/io/PrintSt
        mv.visitLdcInsn( value: "结束执行");
        mv.visitMethodInsn(INVOKEVIRTUAL, owner: "java/io/PrintStream", name: "println", descriptor: "(Ljava/lang/
    }
    super.visitInsn(opcode);
}

@Override
public void visitEnd() {
    mv.visitMaxs( maxStack: 20, maxLocals: 50);
    super.visitEnd();
}
```

利用ASM在方法进入和退出时打印两行文字

```
@Advice.OnMethodEnter
static void onEnter(){
    System.out.println("方法进入");
}

@Advice.OnMethodExit
static void onExit(){
    System.out.println("方法退出");
}
```

Byte buddy在方法进入和退出时打印两行文字

打印方法执行的参数和耗时 - Byte Buddy

Byte Buddy官网： https://bytebuddy.net/

操作步骤：

1、引入依赖

```
<dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy</artifactId>
    <version>1.14.10</version>
</dependency>
<dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy-agent</artifactId>
    <version>1.14.10</version>
</dependency>
```

2、搭建基础框架，此代码为固定代码

```
new AgentBuilder.Default()
        .disableClassFormatChanges() AgentBuilder
        .with(AgentBuilder.RedefinitionStrategy.RETRANSFORMATION) AgentBuilder.Redef
        .with(
                new AgentBuilder.Listener.WithTransformationsOnly(
                        AgentBuilder.Listener.StreamWriting.toSystemOut())) AgentB
        .type(ElementMatchers.named(clazz.getName())) AgentBuilder.Identified.Narrowab
        .transform((builder, type, classLoader, module, protectionDomain) ->
                builder.visit(Advice.to(MyAdvice.class).on(ElementMatchers.any())
        ) AgentBuilder.Identified.Extendable
        .installOn(inst);
```

匹配特定的类名

编写一个类似于AOP的advice通知

打印方法执行的参数和耗时 - Byte Buddy

操作步骤：

3、编写一个Advice通知描述如何去增强类

```
class MyAdvice {
    @Advice.OnMethodEnter
    static long enter(@Advice.AllArguments Object[] ary) {
        if(ary != null) {
            for(int i =0 ; i < ary.length ; i++){
                System.out.println("Argument: " + i + " is " + ary[i]);
            }
        }
        return System.nanoTime();
    }

    @Advice.OnMethodExit
    static void exit(@Advice.Enter long value) {
        System.out.println("耗时为: " + (System.nanoTime() - value) + "纳秒");
    }
}
```

方法进入时执行

方法退出时执行

案例

实战案例1：简化版的Arthas

需求2：

编写一个简化版的Arthas程序，具备以下几个功能：

~~1、查看内存使用情况~~

~~2、查看直接内存使用情况，生成堆内存快照~~

~~3、打印栈信息~~

~~4、打印类加载器~~

~~5、打印类的源码~~

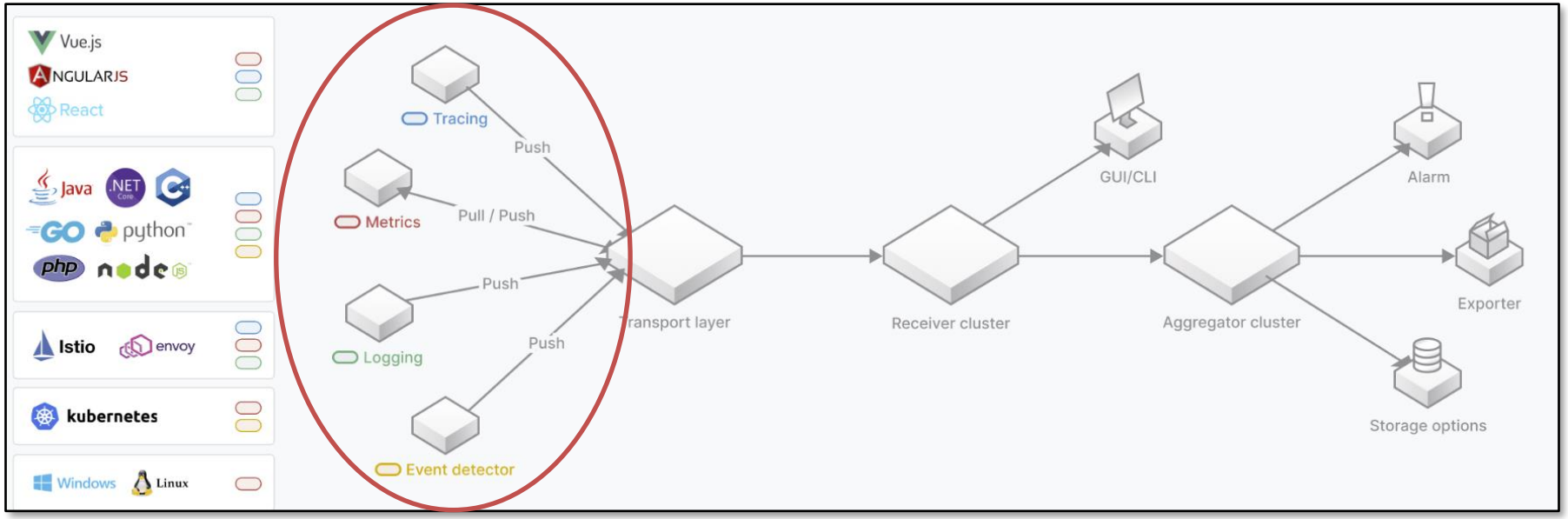
~~6、打印方法执行的参数和耗时~~

最后将整个简化版的arthas进行打包，在服务器上进行测试。使用maven-shade-plugin插件可以将所有依赖打入同一个jar包中并指定入口main方法。

Application performance monitor (APM) 系统

Application performance monitor (APM) 应用程序性能监控系统是采集运行程序的实时数据并使用可视化的方式展示，使用APM可以确保系统可用性，优化服务性能和响应时间，持续改善用户体验。常用的APM系统有Apache Skywalking、Zipkin等。

Skywalking官方网站: https://skywalking.apache.org/



Java程序中的第一步数据采集使用的就是Java Agent技术

案例

实战案例2：APM系统的数据采集

需求：

编写一个简化版的APM数据采集程序，具备以下几个功能：

1、无侵入性获取spring boot应用中，controller层方法的调用时间。

2、将所有调用时间写入文件中。

问题：

Java agent 采用静态加载模式 还是 动态加载模式?

一般程序启动之后就需要持续地进行信息的采集，所以采用静态加载模式。

Java Agent参数的获取

在Java Agent中，可以通过如下的方式传递参数：

```
java -javaagent:./agent.jar=参数 -jar test.jar
```

接下来通过premain参数中的agentArgs字段获取：

```
public static void premain(String agentArgs, Instrumentation inst) {
    System.out.println("agent参数是:" + agentArgs);
```

如果有多个参数，可以使用如下方式：

```
java -javaagent:./agent.jar=param1=value1,param2=value2 -jar test.jar
```

在Java代码中使用字符串解析出对应的key value。

Byte Buddy参数的传递

在Java Agent中如果需要传递参数到Byte Buddy，可以采用如下的方式：

1、绑定Key Value，Key是一个自定义注解，Value是参数的值。

```
new AgentBuilder.Default()
        .disableClassFormatChanges() AgentBuilder
        .with(AgentBuilder.RedefinitionStrategy.RETRANSFORMATION) AgentBuilder.RedefinitionListenable.W
        .with(
                new AgentBuilder.Listener.WithTransformationsOnly(
                        AgentBuilder.Listener.StreamWriting.toSystemOut())) AgentBuilder
        .type(ElementMatchers.isAnnotatedWith(ElementMatchers.named("org.springframework.stereotyp
                .or(ElementMatchers.named("org.springframework.web.bind.annotation.RestController"
        .transform((builder, type, classLoader, module, protectionDomain) ->
                builder.visit(Advice.withCustomMapping().
                        bind(AgentParam.class,agentArgs)
                        .to(TimingAdvice.class).on(ElementMatchers.any()))
        ) AgentBuilder.Identified.Extendable
        .installOn(inst);
```

Byte Buddy参数的传递

2、自定义注解

```
@Retention(RetentionPolicy.RUNTIME)
@java.lang.annotation.Target(ElementType.PARAMETER)
public @interface AgentParam {
    String value();
}
```

3、通过注解注入

```
static void exit(                    默认值
                 @AgentParam("agent.log") String agentParam,
                 @Advice.Origin("#t") String className,
                 @Advice.Origin("#m") String mtdName,
                 @Advice.Enter long value) {
```

总结

Arthas这款工具用到了什么Java技术，有没有了解过？

面试官

回答：

Arthas主要使用了Java Agent技术，这种技术可以让运行中的Java程序执行Agent中编写代码。

Arthas使用了Agent中的动态加载模式，可以选择让某个特定的Java进程加载Agent并执行其中的监控代码。监控方面主要使用的就是JMX提供的一些监控指标，同时使用字节码增强技术，对某些类和某些方法进行增强，从而监控方法的执行耗时、参数等内容。

总结

APM系统是如何获取到Java程序运行中的性能数据的?

面试官

回答：

APM系统比如Skywalking主要使用了Java Agent技术，这种技术可以让运行中的Java程序执行Agent中编写代码。

Skywalking编写了Java Agent，使用了Agent中的静态加载模式，使用字节码增强技术，对某些类和某些方法进行增强，从而监控方法的执行耗时、参数等内容。比如对Controller层方法增强，获取接口调用的时长信息，对数据库连接增强，获取数据库查询的时长、SQL语句等信息。

传智教育旗下高端IT教育品牌