Apusic 金蝶天燕

生成时间：2026-05-19

用户手册

金蝶Apusic Java开发工具包软件

版权声明

本文档所涉及的软件著作权、版权等知识产权已依法进行了注册，由金蝶天燕云计算股份有限公司合法拥有。受《中华人民共和国著作权法》《计算机软件保护条例》《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的版权信息由金蝶天燕云计算股份有限公司合法拥有，受法律的保护，金蝶天燕云计算股份有限公司对本文档可能涉及到的非金蝶天燕云计算股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经金蝶天燕云计算股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将被视为侵权，金蝶天燕云计算股份有限公司有依法追究其责任的权利。

本文档如有更新，不另行通知。对本文档中的问题您可向金蝶天燕云计算股份有限公司告知或查询。未经本公司明确授予的任何权利均予保留。

商标声明

Apusic是深圳市金蝶天燕云计算股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由金蝶天燕合法拥有，受法律保护。未经金蝶天燕的书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯金蝶天燕商标权的，金蝶天燕将依法追究其法律责任。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

目录

1 版本更新说明

本文档根据实际进行更新，最新版本包含历史修改记录。

日期	手册版本	适用产品	更新说明
2024年10	V8.422E01F01	金蝶 Apusic JDK 8.0	首次编写

2 前言

2.1 产品简介

Apusic JDK 是一款高性能、生产环境就绪的OpenJDK发行版本，完全兼容开源 OpenJDK，基于华为毕昇 JDK发展而来，支持多种运行平台，具备更快的云应用启动速度，更好的性能以及提供更为便捷的分析、诊断工具，适合微服务、云原生应用、大数据等实际应用场景，提供最优的 Java 生产环境及解决方案。此外，Apusic JDK 是 Apusic 应用服务器的运行环境，适配了大量 Java 应用程序，解决了业务实际运行中遇到的多个问题，为 Java 应用程序提供一个安全、稳定、可扩展、高性能的 Java 运行环境。

2.2 范围和读者

本手册介绍Apusic JDK产品安装相关的内容，主要适用于实施人员，维护人员等。

2.3 约定和术语

AJDK : 金蝶Apusic JDK（Apusic Java Development Kit，简称：AJDK）

3 产品安装

Apusic JDK 产品支持 Linux 下 aarch64 和 x64 CPU 架构下的安装部署。

3.1 环境要求

软件及操作系统环境要求

组件	要求
操作系统	Linux
CPU	x64、aarch64
内存	4G或以上
硬盘	可用空间10G或以上

3.2 aarch64下安装

1、下载产品包，`apusic-jdk-8u422-linux-aarch64.tar.gz`

2、进入到想要将JDK安装的目录中，并将 `.tar.gz` 压缩包拷贝到当前目录，比如：`/opt/java`

3、将 `.tar.gz` 压缩包解压缩

```
tar -zxvf apusic-jdk-8u422-linux-aarch64.tar.gz
```

4、设置环境变量

编辑 `/etc/profile`，增加如下内容并保存：

```
export JAVA_HOME=/opt/java/apusic-jdk-8u422-linux-aarch64
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

执行 `source /etc/profile` ，使得环境变量生效

5、验证是否安装成功

执行 `java -version` 查看输出的版本是否为apusic jdk对应的版本

3.3 x64下安装

1、下载产品包，比如：`apusic-jdk-8u422-linux-x64.tar.gz`

2、进入到想要将JDK安装的目录中，并将 `.tar.gz` 压缩包拷贝到当前目录，比如：`/opt/java`

3、将 `.tar.gz` 压缩包解压缩

```
tar -zxvf apusic-jdk-8u422-linux-x64.tar.gz
```

4、设置环境变量

编辑`/etc/profile`，增加如下内容并保存：

```
export JAVA_HOME=/opt/java/apusic-jdk-8u422-linux-x64
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

执行`source /etc/profile`，使得环境变量生效

5、验证是否安装成功

执行 `java -version` 查看输出的版本是否为apusic jdk对应的版本

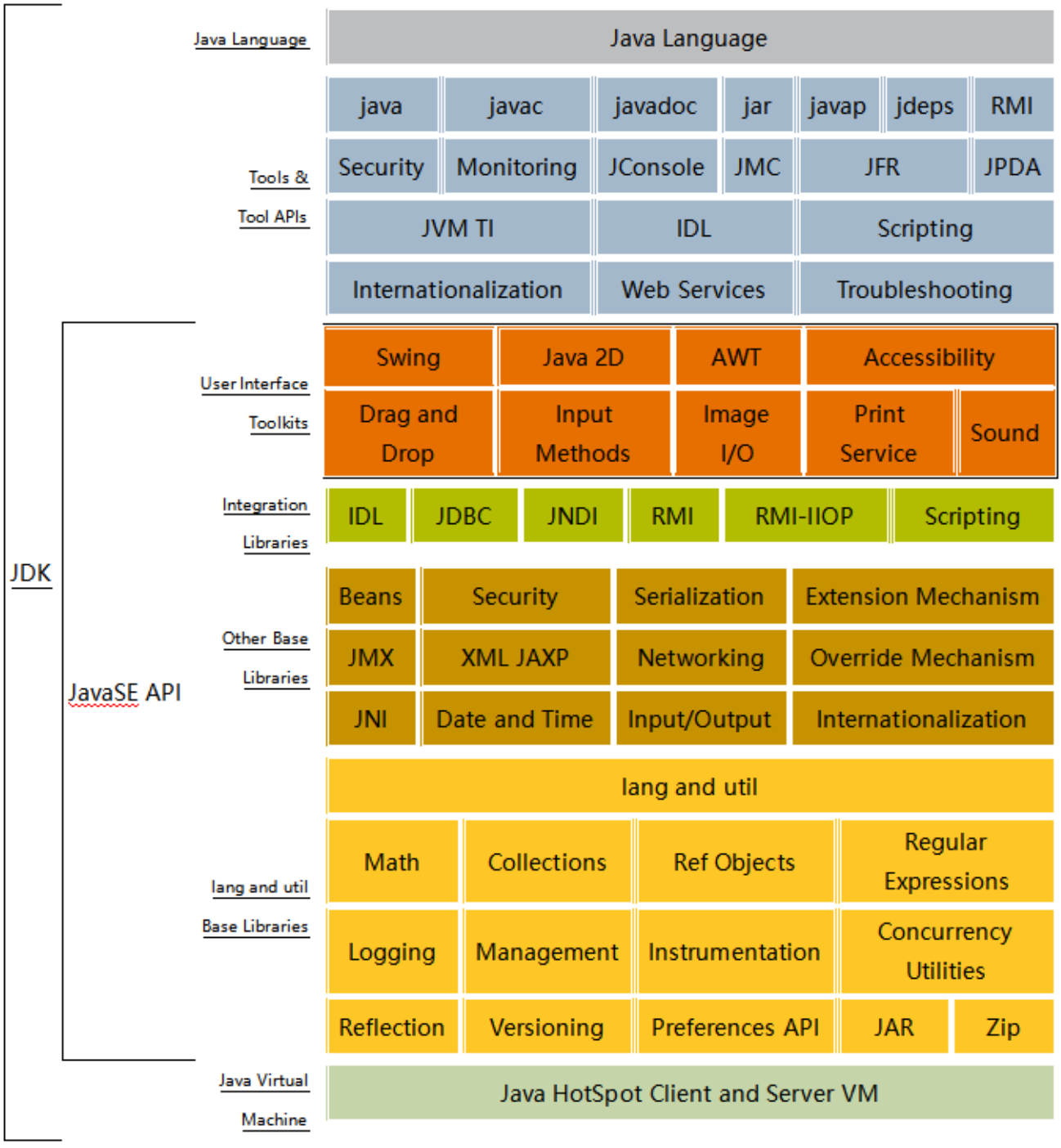
3.4 安装后的工作

如果您想节省磁盘空间，您可以删除 `.tar.gz` 压缩包

4 产品架构

4.1 总体架构

Apusic JDK 功能架构主要包含 Java 语言规范、Java 类库(JavaSE API)、Java 虚拟机 (JVM)、Java 编译器 (javac) 和一系列开发工具构成。



5 技术特性

5.1 支持 JFR

JFR（Java Flight Recorder）是 Java 平台上的一款性能分析工具，它能够捕获和记录JVM运行时的各种数据，帮助开发人员分析应用程序的性能问题并优化程序性能。默认关闭的，用户可以使用以下命令在java应用启动的时候启用JFR：-XX:StartFlightRecording:filename=record.jfr,dumponexit=true

5.2 AppCDS

Java 程序运行初始，类的加载是一个比较耗时的过程，且在每次程序运行中均需要执行一遍。而 CDS（Class Data Sharing）技术，把类加载后的数据保存到文件中。在下次运行时，直接将加载后的类数据从文件中恢复到内存中，不需要再重新执行类的加载过程，从而提高性能。而 AppCDS 在 CDS 的基础上，增加了对应用类的支持。

5.2.1 参数说明

5.2.1.1 -Xshare

- `-Xshare:off` 不打开共享功能，默认为off，但指定此参数后，便可以通过DumpLoadedClassList参数导出ClassList文件
- `-Xshare:dump` 导入共享文件，由SharedClassListFile指定需要共享的类列表，由SharedArchiveFile指定导入的JSA文件
- `-Xshare:on` 使用共享文件，由SharedArchiveFile指定JSA文件
- `-Xshare:auto` 尝试Xshare:on，如果失败，自动修改为Xshare:off

5.2.1.2 -XX:+UseAppCDS

默认CDS(只共享系统类); 打开 `-XX:+UseAppCDS` 即支持AppCDS，除支持共享系统类，还支持应用类；

5.2.1.3 -Xtypecheck 或 -XX:+EnableSplitVerifierForAppCDS

用户可以通过 `-XX:+EnableSplitVerifierForAppCDS` 或者 `-Xtypecheck:on` 选项开启字节码的验证功能,开启之后会使用SplitVerifier对jsa文件中的klass进行验证。该选项默认为不开启，因为大部分情况下在第二步生成jsa文件的过程中字节码已进行过验证。若开启该选项，会对JVM启动时间有轻微延长。

5.2.1.4 -XX:AppCDSLockFile

`-XX:AppCDSLockFile` 适用于并发场景下多进程创建jsa文件的文件锁，用户可按照进程号指定lock 文件： `-XX:AppCDSLockFile=/tmp/appcds-%pid.lock` ，如果用户没有设置该选项，则系统会默认生成一个/tmp/appcds.lock文件，在系统运行结束，VM退出时自动删除。

5.2.2 使用说明

分为三个步骤：生成共享类文件列表(ClassList)；加载ClassList中的类，把这些类的MetaData导致存共享文件；使用共享文件执行Java程序。 下面以HelloWorld程序为例简单说明AppCDS的用法： 预置条件： Java程序HelloWorld.class(其他程序也

行) 下面是每个步骤的命令行，相关参数说明请参见JVM参数说明 步骤一:生成共享类列表文件(hello.lst文件): `java -Xshare:off -XX:+UseAppCDS -XX:DumpLoadedClassList=hello.lst HelloWorld` 步骤二:生成共享文件(根据hello.lst生成hello.jsa): `java -Xshare:dump -XX:+UseAppCDS -XX:SharedClassListFile=hello.lst -XX:SharedArchiveFile=hello.jsa HelloWorld` 步骤三:使用共享文件执行Java程序: `java -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=hello.jsa HelloWorld`

5.2.3 使用限制

1、打开CDS或者AppCDS，不支持修改bootclasspath 例如：通过 -Xbootclasspath 修改 bootclasspath 原因：如果CDS/AppCDS 支持Xbootclasspath 修改 bootclasspath，会导致类加载规则混乱。 即通过jsa运行时，不是取决于当前配置参数，而是取决于生成jsa时的配置 举个简单的例子：patha与pathb都有一个String类。 生成jsa时，配置为patha，则导出到jsa文件是patha的String 使用jsa运行时，配置为pathb，期望的是pathb的String 实际运行时先加载jsa的类信息，即加载patha的String，后面不会再加载pathb的String。跟期望不符合 为了避免这种混乱情况，CDS/AppCDS目前不支持修改bootclasspath

2、CDS/AppCDS 支持共享类的限制 Java version1.4以及之前的类不支持共享 JVM anonymous class不支持共享 运行时动态生成类不支持共享 如果基类不支持共享，则该类也不支持共享

3、CDS/AppCDS 不支持JFR JFR组件启动过程会通过asm 动态创建 anonymous class，与AppCDS冲突

5.2.4 常见问题

1、如果用户classpath中jar的MANIFEST.MF文件中指定了CLASS_PATH, jvm在加载类的时候会把CLASS-PATH指定的jar也纳入到加载路径中。用户需要确保其对应path的正确性。常规情况下，如果对应的path不存在,jvm会抛出一个FileNotFoundException,并忽略它继续执行。但是在dump阶段，如果抛出了FileNotFoundException,jvm内部会重置这个异常为一个dummy异常(VirtualMachineError)，然后中断这个过程，如果此时还有jar没有push到内部的urls中，则会影响后续其他的类的加载和共享。 2、appcds 不支持关闭压缩指针。需要开启。如果设置了Xmx，可能需要调整 `-XX:ObjectAlignmentInBytes=16`,对齐方式值大小基于Xmx的值而定。可以通过： `java -Xmx64g -XX:ObjectAlignmentInBytes=32 -XX:+PrintFlagsFinal -version |grep "UseCompressed"`，确定是否有效。

5.3 快速序列化

序列化是将一个对象序列化为字节流，方便进行传输和保存。OpenJDK 原生的序列化机制耗时较长，序列化的数据太大，反序列化过程查找classmeta太慢。Apusic JDK 优化序列化和反序列化的过程，加快了序列化的速度。

通过JVM参数启用： `-XX:+UnlockExperimentalVMOptions -XX:+UseFastSerializer -DfastSerializerEscapeMode=true`

UseFastSerializer: fastSerializer开关

fastSerializerEscapeMode: 逃生开关。fastSerializer改变了序列化文件格式，因此只能读取优化后的序列化文件，原生的序列化文件会发生读取异常。如果打开逃生开关，则在遇到原生序列化文件时，会从fastSerializer模式切换回原生jdk序列化；

不然则直接抛出异常。此参数需要配合UseFastSerializer进行使用；

enableRMIFastSerializerClass：RMI相关MarshalOutputStream类快速序列化开关。默认为false。由于在启用快速序列化功能的场景下，当开发人员通过现有成熟的基于RMI相关规范软件产品（如VisualVM、JMC等，不便于设置JVM参数）对本地或远程Java应用进行监控等其他操作时，会抛出`java.io.StreamCorruptedException: invalid stream header: DECA0005, and it is a FastSerializer stream`异常，为解决此问题，当前将快速序列化的实现逻辑修改为在启用快速序列化功能的场景下，默认禁用RMI相关MarshalOutputStream类快速序列化功能，当开发人员由于具体的应用场景，需要启用RMI相关MarshalOutputStream类快速序列化功能时，则需要在启动Java应用时，双端除了添加原有的`-XX:+UnlockExperimentalVMOptions -XX:+UseFastSerializer -DfastSerializerEscapeMode=true`参数外，还需要添加`-DenableRMIFastSerializerClass=true`参数，此时RMI相关MarshalOutputStream类也会进行快速序列化处理。

存在的问题

- 序列化对象在读写两端类型不一致：序列化对象在读写两端的定义可能不一致，导致反序列化端无法识别序列化端的对象定义，数据读取时出现异常。
- classmeta信息在运行时发生改变：classmeta在缓存后就不再进行更新。而classmeta如果在运行时发生改变，就可能会触发异常。

5.4 Dynamic CDS特性

Dynamic CDS 技术是 OpenJDK 社区在高版本提出的用于提高Java应用内存使用效率和启动速度的新特性。Apusic JDK 支持该特性，相对于 AppCDS 而言，共享类扩展至自定义类加载器加载的类，扩展了共享类的支持范围，以带来显著的性能提升。

5.4.1 参数说明

base jsa：使用CDS/AppCDS生成的类共享文件，当不指定其路径时，默认为`$JAVA_HOME/jre/lib/aarch64/server/classes.jsa`

top jsa：使用Dynamic CDS生成的类共享文件

5.4.1.1 -Xshare:on

用于使用共享文件，可由SharedArchiveFile指定Base JSA/Top JSA文件

5.4.1.2 -XX:SharedArchiveFile

用于用户首次启动时指定base JSA文件，其是生成Top JSA的基础

5.4.1.3 -XX:ArchiveClassesAtExit

用户指定进程退出时生成Top JSA文件路径。文件名支持按照进程号%p输出，可避免多JVM进程复写一份Top jsa ，导致Top jsa不可用

5.4.1.4 日志开关

-XX:InfoDynamicCDS：打开级别为info的日志，可以通过 `-XX:DynamicCDSLog=PATH` 设置输出路径，默认为标准输出流

-XX:DebugDynamicCDS：打开级别为debug的日志，其余同上

-XX:TraceDynamicCDS：打开级别为trace的日志，其余同上

5.4.2 使用说明

分为两个步骤：将可共享类的MetaData数据dump进文件；使用该文件执行Java程序。

下面以HelloWorld程序为例简单说明Dynamic CDS的用法：

预置条件： Java程序HelloWorld.class

下面是每个步骤的命令行，相关参数说明请参见JVM参数说明

步骤一: 生成top jsa(依赖base jsa)，实验特性开关需要使能:

```
java -XX:+UnlockExperimentalVMOptions -Xshare:on -
XX:SharedArchiveFile=$JAVA_HOME/jre/lib/aarch64/server/classes.jsa -
XX:ArchiveClassesAtExit=top.jsa -XX:+InfoDynamicCDS HelloWorld
```

- 如果因为某些参数改变，导致jdk包中带的default jsa无法使用，请重新用CDS/AppCDS生成base jsa。
- 非正常结束的进程，比如kill -9杀死进程，则无法生成jsa，需要使用中断信号SIGINT，命令为kill -2。或者需要通过命令在进程结束之前生成， `jcmd <pid> GC.dynamic_cds_dump`
- 因为生成jsa过程中会修改运行时数据，无法保证运行时正确，因此dump top jsa结束后直接退出进程。
- 此处可省略该参数 `-XX:SharedArchiveFile` 参数，此时默认使用- `XX:SharedArchiveFile=$JAVA_HOME/jre/lib/aarch64/server/classes.jsa` 文件

步骤二: 使用top jsa运行Java程序:

```
java -Xshare:on -XX:SharedArchiveFile=top.jsa -XX:+InfoDynamicCDS
HelloWorld
```

- 如果在top JSA生成之后base JSA路径发生改变，则top JSA中记录的base JSA路径无效，需要在参数中指定base jsa路径

  `java -Xshare:on -XX:SharedArchiveFile=base.jsa:top.jsa -XX:+InfoDynamicCDS HelloWorld`

5.4.3 使用限制

1、生成、使用 jsa 两阶段参数不一致 （1）restore 阶段使用参数 `-XX:ObjectAlignmentInBytes=32、16` 时，运行时会报错，信息如下：

```
Error occurred during initialization of VM
Unable to use shared archive.
An error has occurred while processing the shared archive file.
The shared archive file's ObjectAlignmentInBytes of 8 does not equal the
current ObjectAlignmentInBytes of 32.
```

错误原因：因为生成base jsa时，ObjectAlignmentInBytes默认为8，读取其时不支持指定其他对齐参数 解决方法：重新使用AppCDS生成base jsa使用，生成时参数加入使用时对应的`-XX:ObjectAlignmentInBytes`的值即可

（2）当dump时的系统环境变量page_size小于resolve时的page_size时，运行时会报错，信息如下：

```
An error has occurred while processing the shared archive file.
Unable to map ReadOnly shared space at required address.
Error occurred during initialization of VM
Unable to use shared archive.
```

错误原因：当jsa文件的page_size小于当前环境的page_size，系统无法为jsa文件恢复分配内存 解决方法：调整dump时的系统环境变量page_size，使其大于等于resolve时的page_size，重新使用AppCDS生成jsa使用即可

2、压缩指针未开启 restore阶段使用参数`-XX:-UseCompressedOops`或`-XX:-UseCompressedClassPointers`时，运行时会报错，信息如下：

```
Error occurred during initialization of VM
Unable to use shared archive.: UseCompressedOops and
UseCompressedClassPointers must be on for UseSharedSpaces.
Class data sharing is inconsistent with other specified options.
```

错误原因：关闭压缩指针会和类共享功能冲突。 解决方法：使用时应当开启压缩指针开关

3、堆地址冲突 restore阶段使用参数`-XX:HeapBaseMinAddress=X1g -Xmx=X2g`（值X1+ X2>=32）的组合，运行时会报错，信息如下：

```
An error has occurred while processing the shared archive file.
Unable to reserve shared space at required address 0x0000000800000000
Error occurred during initialization of VM
Unable to use shared archive.
```

错误原因：jsa 文件的映射地址的堆区间，不能被参数所指定的JVM 其他组件占用。 解决方法：将此组合值X1+ X2改为小于32可正常运行

4、Dynamic CDS不支持的共享类 Java version1.5以及之前的类不支持共享 JVM anonymous class不支持共享 如果基类不支持共享，则该类也不支持共享

5、Dynamic CDS不支持JFR JFR组件启动过程会通过asm 动态创建 anonymous class，与Dynamic CDS冲突

5.4.4 常见问题

1、 Dynamic CDS不支持关闭压缩指针，需要开启。如果设置了Xmx，可能需要调整 `-XX:ObjectAlignmentInBytes=16`，对齐方式值大小基于Xmx的值而定。可以通过： `java -Xmx64g -XX:ObjectAlignmentInBytes=32 -XX:+PrintFlagsFinal -version |grep "UseCompressed"`，确定是否有效。

2、 如果用户classpath中jar的MANIFEST.MF文件中指定了CLASS_PATH, JVM在加载类的时候会把CLASS_PATH指定的jar也纳入到加载路径中。用户需要确保其对应path的正确性。常规情况下，如果对应的path不存在, JVM会抛出一个FileNotFoundException,并忽略它继续执行。但是在dump阶段，如果抛出了FileNotFoundException,JVM内部会重置这个异常为一个dummy异常(VirtualMachineError)，然后中断这个过程，如果此时还有jar没有push到内部的urls中，则会影响后续其他的类的加载和共享。

5.5 G1 垃圾收集器优化

- G1 NUMA-Aware特性：在NUMA架构下，跨NUMA节点操作内存相比本NUMA节点操作内存时延会成倍增加，NUMA-Aware特性目的是让JAVA用户尽可能操作本NUMA节点上内存，从而提升JVM访存速度。

  -XX:+UseG1GC：只在G1GC下生效 -XX:+UseNUMA：开启G1 NUMA-Aware特性，TLAB，Eden区，Survivor区的Region选取将采取就近原则

- G1 Full GC并行化：G1 算法在mark、prepare、adjust、compact等阶段优化成多线程执行方式，有效降低Full GC的STW时间，改善系统最坏情况下的G1 Full GC性能。

  -XX:+UseG1GC：只在G1GC下生效 -XX:+G1ParallelFullGC：开启该特性，需要显示打开，默认是关闭

- G1 GC内存伸缩特性：OpenJDK 8 中 G1 垃圾收集器无法及时将空闲的 Java 堆内存释放给操作系统。G1 仅在 FullGC 才会把空闲的Java堆内存释放给操作系统。但由于 G1 尽可能避免触发 FullGC，因此在许多情况下，除非强制从外部执行 FullGC，否则 G1 不会将空闲的Java堆内存释放给操作系统。Apusic JDK 能够检测应用负载下降和Java堆有空闲内存的情况，并自动减少JVM Java堆占用情况，将空闲内存资源归还给操作系统。

  参数说明

  1. -XX:+UseG1GC，弹性回收必须是G1垃圾回收器开启情况下使用
  2. (默认关闭)-XX:+G1Uncommit开启弹性回收
  3. (默认关闭)-XX:+G1UncommitLog开启弹性回收的日志
  4. (默认值，可以不设置, 范围0-100)-XX:G1PeriodicGCLoadThreshold=10进程负载阈值，负载小于该值，弹性回收正常执行；大于则说明程序负载较高，本次内存回收取消。设置为0可以取消检查负载情况（一般微服务进程该值不会超过5）

5. (默认值，可以不设置)-XX:+G1PeriodicGCProcessLoad上面的阈值使用进程负载，false的时候使用linux系统负载
6. (默认值，可以不设置，范围>0)-XX:G1PeriodicGCInterval=15000弹性回收周期，单位毫秒。15秒执行一次回收，不能设置为0
7. (默认值，可以不设置)-XX:-G1UncommitThreadPriority, 回收线程调度优先级，默认非critical
8. (默认值，可以不设置,范围大于0小于1)-XX:G1UncommitPercent=0.1，回收region的数量，此值指空闲region的比例
9. (默认值，可以不设置,范围>0)-XX:G1UncommitDelay=50，在应用程序第一次gc后该秒开启内存回收功能

5.6 运维工具增强

- jcmd 工具新增 VM.classes 命令，打印所有类的信息。

  查看用法

```
jcmd <pid> help VM.classes
```

  打印类信息KlassAddr、Size、State、Flags、ClassName

```
jcmd <pid> VM.classes
```

  打印InstanceKlass详细信息

```
jcmd <pid> VM.classes -verbose
```

属性名称	描述
KlassAddr	表示Klass虚拟地址
Size	类大小
State	类加载状态
Flags	类的标志
ClassName	类全限定名

  类加载状态

状态	描述
allocated	已分配（但尚未链接）

loaded	已加载并插入到类的层次结构中（但尚未链接）
linked	已成功链接/验证（但尚未初始化）
being_initialized	当前正在运行类初始化
fully_initialized	已初始化（成功的最终状态）
initialization_error	初始化过程中发生错误

类标识

标识	描述
F	具有或继承非空的finalize方法
f	具有final方法
W	方法被重写过
C	被 @Contended annotation标记
R	类被重新定义（redefined）
S	是共享类(BOOLEAN, false)

- **jcmd 工具新增 VM.classloaders 命令，打印类加载器层次结构和类加载器详细信息。**

查看用法

```
jcmd <pid> help VM.classloaders
```

选项	描述
show-classes	打印每个类加载器已加载的类
verbose	打印每个类加载器的 Loader Data 和 Loader Data 虚拟地址

打印类加载器层次结构

```
jcmd <pid> VM.classloaders
```

打印每个类加载器已加载的类

```
jcmd <pid> VM.classloaders show-classes
```

打印每个类加载器的 Loader Data 和 Loader Data 虚拟地址

```
jcmd <pid> VM.classloaders verbose
```

- jcmd 工具新增 VM.metaspace 命令，打印元空间的统计信息。

  查看用法

  ```
  jcmd <pid> help VM.metaspace
  ```

  打印元空间统计信息

  ```
  jcmd <pid> VM.metaspace
  ```

- jcmd 工具新增 System.trim_native_heap 命令，底层调用glibc函数malloc_trim，将glibc不用的内存及时还给操作系统。

  查看用法

  ```
  jcmd <pid> help System.trim_native_heap
  ```

  将glibc不用的内存还给操作系统

  ```
  jcmd <pid> System.trim_native_heap
  ```

- Thread.print 功能增强：增强 jstack 和 jcmd Thread.print 功能，打印线程堆栈时输出更多的信息（cpu，elapsed，allocated，defined_classes）

  cpu 和 elapsed 为本次增强默认打印的信息，allocated 和defined_classes 需要开启对应的选项才会打印，具体选项可以参考功能实现。 cpu 和 elapsed 这两个属性用于检测在 CPU 上花费过多时间或过少时间的线程，一段时间内CPU耗时过高表示某种业务可能存在性能瓶颈，一段时间内CPU耗时未增加的线程可能是线程被阻塞。

  JVM参数

  通过如下方式打印额外的 allocated 和 defined_classes 信息。

  -XX:[+|-]PrintExtendedThreadInfo 进程启动时添加 `-XX:+PrintExtendedThreadInfo` VM参数后，使用jstack和jcmd打印额外线程堆栈时不需要添加 -e选项。同时低版本jdk的jcmd工具也可以打印额外 allocated 和 defined_classes 信息。

  jcmd Thread.print -e 进程启动未添加 `-XX:+PrintExtendedThreadInfo` VM参数，可以通过 `jcmd Thread.print -e` 打印额外的allocated 和 defined_classes 信息。

jstack -e 进程启动未添加 `-XX:+PrintExtendedThreadInfo` VM参数，可以通过 `jstack -e` 打印额外的 allocated 和 defined_classes 信息。

Thread.print 用法说明

```
jcmd <pid> help Thread.print
```

属性	描述
cpu	表示线程在 CPU 上执行的时间 ， 默认打印
elapsed	自线程启动以来经过的挂钟时间，默认打印
allocated	该线程分配的字节数，这可能暗示加载了太多类的线程, 开启选项后打印，如 -XX:+PrintExtendedThreadInfo
defined_classes	该线程定义的类数，这可能暗示加载了太多类的线程,开启选项后打印,如 -XX:+PrintExtendedThreadInfo

jstack命令用法说明

```
jsatck -h
```

1、进程启动时未添加VM参数-XX:+PrintExtendedThreadInfo

（1）jstack不加-e选项打印线程堆栈信息

```
jsatck <pid>
```

```
...
"main" #1 prio=5 os_prio=0 cpu=61.90ms elapsed=23.13s
tid=0x0000ffff8400a000 nid=0x109db waiting on condition
[0x0000ffff8af8e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

（2） jstack加-e选项打印线程堆栈信息

```
jsatck -e <pid>
```

```
...
"main" #1 prio=5 os_prio=0 cpu=61.90ms elapsed=82.92s allocated=423K
defined_classes=405 tid=0x0000ffff8400a000 nid=0x109db waiting on
condition [0x0000ffff8af8e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

(3) jcmd Thread.print不加-e选项打印线程堆栈信息

```
jcmd <pid> Thread.print
```

```
...
"main" #1 prio=5 os_prio=0 cpu=61.90ms elapsed=133.43s
tid=0x0000ffff8400a000 nid=0x109db waiting on condition
[0x0000ffff8af8e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

(4) jcmd Thread.print加-e选项打印线程堆栈信息

```
jcmd <pid> Thread.print -e
```

```
...
"main" #1 prio=5 os_prio=0 cpu=61.90ms elapsed=155.68s
allocated=423K defined_classes=405 tid=0x0000ffff8400a000
nid=0x109db waiting on condition [0x0000ffff8af8e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
```

```
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

2、进程启动时添加VM参数-XX:+PrintExtendedThreadInfo

（1）jstack不加-e选项打印线程堆栈信息

```
jstack <pid>
```

```
...
"main" #1 prio=5 os_prio=0 cpu=62.50ms elapsed=11.50s allocated=423K
defined_classes=405 tid=0x0000ffff9000a000 nid=0x11242 waiting on
condition [0x0000ffff96c1e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

（2）jstack加-e选项打印线程堆栈信息

```
jsatck -e <pid>
```

```
...
"main" #1 prio=5 os_prio=0 cpu=62.50ms elapsed=63.57s allocated=423K
defined_classes=405 tid=0x0000ffff9000a000 nid=0x11242 waiting on
condition [0x0000ffff96c1e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

（3）jcmd Thread.print不加-e选项打印线程堆栈信息

```
jcmd <pid> Thread.print
```

```
...
"main" #1 prio=5 os_prio=0 cpu=62.50ms elapsed=97.13s allocated=423K
defined_classes=405 tid=0x0000ffff9000a000 nid=0x11242 waiting on
condition [0x0000ffff96c1e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

（4）jcmd Thread.print加-e选项打印线程堆栈信息

```
jcmd <pid> Thread.print -e
```

```
...
"main" #1 prio=5 os_prio=0 cpu=62.50ms elapsed=106.97s
allocated=423K defined_classes=405 tid=0x0000ffff9000a000
nid=0x11242 waiting on condition [0x0000ffff96c1e000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at ThreadPrintTest.main(ThreadPrintTest.java:4)
...
```

- **TraceClassLoading 功能增强**：增强TraceClassLoading功能，输出更多的类加载信息（时间、线程id、类加载器以及类加载时的线程堆栈），方便用户排查类加载相关问题。

  新增VM参数

  - *-XX:[+|-]PrintClassLoadingDetails*

  启用PrintClassLoadingDetails后，打印更多的类加载信息（包括时间、线程id和类加载器），开启 TraceClassLoading 生效

  - *-XX:PrintThreadStackOnLoadingClass=ClassName*

  打印指定类加载时的线程堆栈信息，开启 PrintClassLoadingDetails 生效

  输出格式:

```
2022-10-24T17:25:19.746+0800: 111828  [Loaded
org.apache.tomcat.util.net.Acceptor from
file:/home/user/software/tomcat/apache-tomcat-9.0.7/lib/tomcat-
coyote.jar by classloader java.net.URLClassLoader]
"main" #1 prio=5 os_prio=0 tid=0x0000ffffa400b000 nid=0x1b4d4
runnable [0x0000ffffac1bb000]
   java.lang.Thread.State: RUNNABLE
        at java.lang.ClassLoader.defineClass1(Native Method)
        ...
```

内容属性	内容详细信息
时间	2022-10-24T17:25:19.746+0800
线程id	111828
类加载器	java.net.URLClassLoader
线程堆栈	"main" #1 prio=5 os_prio=0 tid=0x0000ffffa400b000 nid=0x1b4d4 runnable [0x0000ffffac1bb000] java.lang.Thread.State: RUNNABLE at java.lang.ClassLoader.defineClass1(Native Method) ...

功能用法:

1、未开启PrintClassLoadingDetails

执行命令:

```
java -XX:+TraceClassLoading TraceClassLoadingTest
```

输出内容:

```
...
[Loaded java.security.UnresolvedPermission from
/home/user/jdk/jre/lib/rt.jar]
[Loaded java.security.BasicPermissionCollection
/home/user/jdk/jre/lib/rt.jar]
```

```
[Loaded TraceClassLoadingTest from file:/home/xiezhaokun/]
[Loaded sun.launcher.LauncherHelper$FXHelper
/home/user/jdk/jre/lib/rt.jar]
[Loaded java.lang.Class$MethodArray /home/user/jdk/jre/lib/rt.jar]
...
```

2、开启PrintClassLoadingDetails

执行命令：

```
java -XX:+TraceClassLoading -XX:+PrintClassLoadingDetails
TraceClassLoadingTest
```

输出内容：

```
...
2022-11-24T10:10:44.912+0800: 55965  [Loaded
java.security.UnresolvedPermission from
/home/user/jdk/jre/lib/rt.jar by classloader bootstrap]
2022-11-24T10:10:44.912+0800: 55965  [Loaded
java.security.BasicPermissionCollection from
/home/user/jdk/jre/lib/rt.jar by classloader bootstrap]
2022-11-24T10:10:44.912+0800: 55965  [Loaded TraceClassLoadingTest
from file:/home/user/ by classloader
sun.misc.Launcher$AppClassLoader]
2022-11-24T10:10:44.912+0800: 55965  [Loaded
sun.launcher.LauncherHelper$FXHelper from
/home/user/jdk/jre/lib/rt.jar by classloader bootstrap]
2022-11-24T10:10:44.912+0800: 55965  [Loaded
java.lang.Class$MethodArray from /home/user/jdk/jre/lib/rt.jar by
classloader bootstrap]
...
```

3、使用 -XX:PrintThreadStackOnLoadingClass 打印打印指定类加载时的线程堆栈信息

执行命令：

```
java -XX:+TraceClassLoading -XX:+PrintClassLoadingDetails -
XX:PrintThreadStackOnLoadingClass=TraceClassLoadingTest
TraceClassLoadingTest
```

输出内容:

```
2022-11-24T10:10:44.912+0800: 55965  [Loaded
java.security.UnresolvedPermission from
/home/user/jdk/jre/lib/rt.jar by classloader bootstrap]
        2022-11-24T10:10:44.912+0800: 55965  [Loaded
java.security.BasicPermissionCollection from
/home/user/jdk/jre/lib/rt.jar by classloader bootstrap]
        2022-11-24T10:10:44.912+0800: 55965  [Loaded
TraceClassLoadingTest from file:/home/user/ by classloader
sun.misc.Launcher$AppClassLoader]
        "main" #1 prio=5 os_prio=0 cpu=70.23ms elapsed=0.07s
tid=0x0000ffffb000a000 nid=0xdc31 runnable [0x0000ffffb5b4d000]
           java.lang.Thread.State: RUNNABLE
                at java.lang.ClassLoader.defineClass1(Native Method)
                at
java.lang.ClassLoader.defineClass(ClassLoader.java:756)
                at
java.security.SecureClassLoader.defineClass(SecureClassLoader.java:14
                at
java.net.URLClassLoader.defineClass(URLClassLoader.java:493)
                at
java.net.URLClassLoader.access$100(URLClassLoader.java:75)
                at
java.net.URLClassLoader$1.run(URLClassLoader.java:389)
                at
java.net.URLClassLoader$1.run(URLClassLoader.java:383)
                at
java.security.AccessController.doPrivileged(Native Method)
                at
```

```
java.net.URLClassLoader.findClass(URLClassLoader.java:382)
                at
java.lang.ClassLoader.loadClass(ClassLoader.java:418)
                locked <0x0000000580b598b8> (a java.lang.Object)
                at
sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:352)
                at
java.lang.ClassLoader.loadClass(ClassLoader.java:351)
                at
sun.launcher.LauncherHelper.checkAndLoadMain(LauncherHelper.java:601)
            2022-11-24T10:10:44.912+0800: 55965  [Loaded
sun.launcher.LauncherHelper$FXHelper from
/home/user/jdk/jre/lib/rt.jar by classloader bootstrap]
            2022-11-24T10:10:44.912+0800: 55965  [Loaded
java.lang.Class$MethodArray from /home/user/jdk/jre/lib/rt.jar by
classloader bootstrap]
            ...
```

- **异步 GC 日志配置**：解决写入GC日志可能会被阻止，进程中维护一个循环固定大小的内存缓冲区，先将GC日志写入到缓冲区，然后通过单独的线程将缓冲区的内容刷新到GC日志文件中。

  新增VM参数：

  - *-XX:[+|-]UseAsyncGCLog*

  开启或关闭异步记录GC日志，使用-Xloggc配置gc日志路径才会生效

  - *-XX:AsyncLogBufferSize=size*

  设置异步缓存区大小, 默认为2M

  - *-XX:[+|-]PrintAsyncGCLog*

  打印异步GC日志的一些调试信息

  启用异步GC日志

```
java -Xloggc:/home/user/gc.log -XX:+UseAsyncGCLog TestAsyncGC
```

  设置日志缓冲区大小

```
java -Xloggc:/home/user/gc.log -XX:+UseAsyncGCLog -
XX:AsyncLogBufferSize=2M TestAsyncGC
```

打印异步日志调试信息

```
java -Xloggc:/home/user/gc.log -XX:+UseAsyncGCLog -
XX:AsyncLogBufferSize=2M -XX:+UnlockDiagnosticVMOptions  -
XX:+PrintAsyncGCLog Test
```

- glibc 内存整理：JVM 通过 glibc来进行内存的分配与释放，但是有时（取决于分配的粒度和许多其他因素）存在即使JVM调用free(3)将内存返回给glibc，glibc依然保留这些已释放的C堆内存而不是返还给操作系统，这将造成不好排查的（非JAVA/JVM层面的）RSS内存占用持续增长。

  新增 VM 参数 GCTrimNativeHeap，默认不开启，开启后虚拟机将会主动触发修剪Glibc内存的操作，修剪的时机有两种：

  - 每次Full GC会进行trim修剪；
  - 启动一个定时器线程，在设置的时间内定时去trim修剪； 需要注意的是，如果使用Serial垃圾收集器，因为只有单线程进行垃圾收集，所以并未额外的去单独开启定时器线程去定时修剪，即在使用Serial垃圾收集器的情况下只会在Full GC的时候进行修剪。

  新增 VM 参数 GCTrimNativeHeapInterval，默认60，单位为秒，在开启GCTrimNativeHeap时的时候生效，用来指定定时器线程定时执行修剪的定时时间。需要注意的是，如果我们不想使用定时修剪的功能，可以将GCTrimNativeHeapInterval的值设置为0，此时就会只保留在Full GC中修剪Glibc内存的行为（适用于JDK8中所有的垃圾收集器）。

  输出的内容格式

```
Trim native heap: RSS+Swap: 613M->610M (-2924K), 0.918ms
Trim native heap: RSS+Swap: 611M->611M (-172K), 0.323ms
Trim native heap: RSS+Swap: 611M->611M (+0B), 0.243ms
```

  用法

  两个参数为实验特性，故开启前需要开启参数-XX:+UnlockExperimentalVMOptions解锁实验属性的参数。 `-XX:+UnlockExperimentalVMOptions -XX:+GCTrimNativeHeap -XX:GCTrimNativeHeapInterval=<seconds>`

  示例： `-XX:+UnlockExperimentalVMOptions -XX:+GCTrimNativeHeap -XX:GCTrimNativeHeapInterval=20`

- jmap **并行扫描**：可指定并行线程数，有效提高jmap堆扫描效率、减少扫描时间。当前JDK8支持G1GC、ParallelGC和CMS,其他GC暂不支持并行扫描。

  jmap –histo后增加parallel可输入并行线程数，如果指定的数目大于系统当前可支持的线程数，则会使用系统当前可支持的线程数(active_workers)： jmap –histo:live,parallel=3 pid ： 指定并行线程数为3 jmap –histo:live,parallel=0 pid ： 使用系统当前可支持的并行线程数 jmap –histo:live,parallel=1 pid ： 使用原有的串行扫描

6 兼容性

广泛兼容国产化操作系统和芯片

6.1 芯片架构

- 鲲鹏
- 飞腾
- 海光
- 海思

6.2 Linux系统

- 国产Linux操作系统：麒麟操作系统、统信UOS、深度Linux、优麒麟、红旗Linux、中标麒麟、华为欧拉、中科方德、凝思等
- RedHat 系列
- CentOS 系列
- Ubuntu 系列
- Suse Linux 系列

全国统一服务热线

4008-555-800

金蝶天燕云计算股份有限公司(简称“金蝶天燕云”)成立于2000年，前身为“金蝶中间件公司”，是金蝶集团旗下新一代软件基础云平台服务商，云计算国家标准制定企业，国家信创产业核心软件企业。金蝶天燕是国家863重点研发计划与核高基重大专项承接企业，也是“两网一站四库十二金”国家重点工程的基础平台提供商，产品广泛应用于政府、军工、金融、能源等关键行业，累计服务客户总数超过10万家。

Apusic 金蝶天燕

云计算国家标准制定企业

金蝶集团旗下基础软件企业

信息技术应用创新核心企业

官网：www.apusic.com