[

Java千百问_08JDK详解（002）_jdk工具集都有什么

,

点击进入_更多_Java千百问

1、jdk工具集都有什么工具

了解java核心框架看这里：java核心框架是什么样的
jdk工具集包含了java开发、编译和运行过程中所使用的工具，主要有以下这些：

调试体系（Debugger Architecture）

具备完善的体系结构和使用规格的调试体系，允许在开发环境中可以用来进行代码调试（调试即逐行执行代码，并且可以监控变量和运行情况）。Java平台调试器体系结构也叫JPDA，它提供了一整套用于调试Java程序的API，以及用于开发Java调试工具的接口和协议。Eclipse等IDE工具均使用了JPDA来开发自己的调试器。

了解更多Java调试体系看这里：[JPDA是什么][3]
[3]:

JVM工具接口（VM Tool Interface）

Java虚拟机工具接口（JVM TI）是一个规范，用来监控状态以及控制在JVM中运行程序的执行。它是用来代替Java虚拟机分析器接口（JVMPI）的，JVMPI已被弃用。

了解更多 JVM工具接口看这里：[JVMTI是什么][4]
[4]:

Java文档工具（Javadoc Tool）

Java文档工具，通过解析源文件和文档注释来产生一组描述代码的HTML页面。Doclet API提供了一种能够为用户检查程序、库源代码层级结构的机制，可以使用这个API生成文档。

了解更多Javadoc Tool看这里：[Javadoc是什么][5]
[5]:

java动态链接（Dynamic Attach）

com.sun.tools.attach包含了一个Oracle扩展Java平台，允许一个应用程序连接到一个正在运行的Java虚拟机。一旦连接，工具代理就能够在目标虚拟机被启动了。

了解更多Dynamic Attach看这里：[AttachAPI是什么][6]
[6]:

java控制器API（JConsole API）

com.sun.tools.jconsole包含了一个Oracle扩展Java平台，提供了一组接口，允许访问JConsole，对线程、内存、类、对象进行分析和统计。

了解更多 JConsoleAPI看这里：[JConsole是什么][7]
[7]:

JDK工具包（JDK Tools & Utilities）

JDK工具包中提供了很多工具和实用的程序。涵盖了基本工具（ javac, java, javadoc, apt, appletviewer, jar, jdb, javah, javap, extcheck），安全工具，国际化工具，RMI工具，IDL， RMI-IIOP工具，部署工具，java插件工具，java Web工具，监控管理工具，故障排查工具等。

了解更多jdk工具包看这里： [JDK工具包有哪些工具][8]
[8]:

]

[

Java千百问_08JDK详解（003）_jdk基础类库都有什么

,

点击进入_更多_Java千百问

1、jdk基础类库都有什么

了解java核心框架看这里：java核心框架是什么样的
jdk基础类库分为两大部分，基础库、扩展基础库，提供了相当大量的功能，基础库具体如下：

1. **lang包、util包（Lang and Util Packages）**
   lang包提供的基本Object类、Class类、String类、基本类型的包装类、基本的数学类等等最基本类。
   util包提供基本的集合框架、jar处理等基础类。
   了解lang包、util包看这里：[lang包、util包有什么功能][3]

2. **math包（Math）**
   java数学包的功能包括浮点库以及任意精度的数据运算。
   了解math包看这里：[java.util.math包有什么功能][4]

3. **监管工具（Monitoring and Management）**
   Java平台提供了全面的监控和管理工具，包括JVM的监管API、监管API日志、jconsole和其他监控工具、 out-of-the-box的监管工具（开箱即用）、Java管理扩展平台（JMX）以及Oracle的平台扩展。主要功能在java.lang.management包中。
   了解java监管工具看这里：[java.lang.management包有什么功能][5]

4. **包版本识别（Package Version Identification）**
   包版本识别特性使得包可以有多个不同的版本，以便在运行时程序和applet可以识别特定的JRE、JVM以及类包的版本。
   了解java版本规范看这里：[Versioning Specification是什么][6]

5. **引用对象（Reference Objects）**
   引用对象可以在一定程度上与垃圾收集器进行交互。程序可以使用引用对象来保持对其他对象的引用，这样后者对象仍然可以被收集器。也可以指定一段时间后通知收集器，来确定给定对象是否被引用（可达性，是否可达）发生了变化。引用对象可以构建简单的缓存，当内存利用率低时来刷新缓存，通过调度这种更灵活的方式进行清理。主要功能在java.lang.ref包中。
   了解引用对象看这里：[java.lang.ref包有什么用][7]

6. **反射（Reflection）**
   反射允许Java代码获取类的字段、方法以及构造函数，并且可以在安全限制的基础上使用被反射的字段、方法和构造函数进行操作。API允许应用程序访问目标对象（基于它运行时的类）的公共成员（字段、方法、构造函数）或者为一个给定的类声明成员。
   了解反射看这里：[java反射是什么][8]

7. **集合框架（Collections Framework）**
   集合是一组对象的合集，集合框架是一个统一管理集合的体系结构，允许不同的集合可以独立的进行操作。Java集合框架减少了编程工作，同时提升了性能。它允许不相关的api之间相互操作，减少设计和学习新API的成本，很大程度上促进了代码的重用。

了解集合框架看这里：[java集合框架是什么][9]

8. **并发工具包（Concurrency Utilities）**
   并发工具包提供了一种强大的、可扩展的高性能多线程框架，以及线程池和阻塞队列等工具。这个包让程序员需要手动完成的并发工作变得简单，最主要的体现就是为集合框架的每一种集合类型提供了并发类。此外，这些包还为并发编程提供了傻瓜式的工具。主要功能再java.util.concurrent包中。
   了解并发工具包看这里：[java.util.concurrent包有什么用][10]

9. **JAR文件（Java Archive Files）**
   JAR（Java Archive）是一个独立于平台的文件格式提高网络传输的速度，并且JAR还支持压缩，降低了文件的大小，进一步提高了传输效率。此外，applet的作者可以通过再JAR文件中加入数字签名，以达到验证来源的目的。而且它是完全可扩展的，对应的API在java.util.jar包中。
   了解JAR看这里：jar是什么
   了解JAR文件工具看这里：[java.util.jar包有什么用][12]

10. **日志工具（Logging）**
    日志API是提供给最终用户一种便于软件维修和分析管理的日志报告（系统管理员、现场服务工程师、软件开发团队）。日志API能够捕捉很多信息，例如安全故障、配置错误、应用程序中的性能瓶颈和bug。主要功能在java.util.logging包中。
    了解java日志工具看这里：[java.util.logging包有什么用][13]

11. **配置工具（Preferences）**
    Preferences API为应用程序提供了一种用来存储和检索用户及系统偏好和配置数据的方法。这些数据被持久化一个实现相关（implementation-dependent）的后备存储器中（操作系统级别）。通过两个单独的树来记录偏好，一个用于用户首选项和一个用于系统首选项。主要功能在java.util.prefs包中。
    了解java偏好配置看这里：[java.util.prefs包有什么用][14]

]

[

Java千百问_08JDK详解（004）_jdk基础扩展类库都有什么

,

1、jdk基础扩展类库都有什么

了解java核心框架看这里：java核心框架是什么样的
jdk基础类库分为两大部分，基础库、扩展基础库，提供了相当大量的功能，扩展基础库具体如下：

1. I/O工具（I/O）
   java.io和java.nio包提供了一组非常丰富的api来管理应用程序的I/O（输入/输出）。包括文件和设备I/O、对象序列化、缓冲区管理、字符集支持等。此外，API还为服务端提供了包括多路复用、非阻塞I/O、内存映射和文件锁等功能。
   了解更多java.io和java.nio包看这里：[io包nio包有什么区别][3]

2. 对象序列化（Object Serialization）
   对象序列化扩展了Java核心输入/输出类所支持的对象。对象序列化将对象转换为流传输，支持编码过的对象、字节流、图像流。序列化用于轻量级的持久化以及通过套接字进行的通信和远程方法调用(RMI)。
   了解java序列化看这里：[Serializable接口有什么用][4]

3. javaNet（Networking）
   提供了网络功能的类，包括寻址、url和uri使用类、连接到服务器的套接字类、网络安全功能等等。主要功能在javax.net包中。
   了解javaNet看这里：[javax.net包有什么用][5]

4. 安全库（Security）
   安全库中提供了很多安全相关的API，如配置访问控制、数字签名、身份验证和授权、密码、网络通信安全等等。主要功能在java.security、javax.security包中。
   了解Security看这里：[security包有什么用][6]

5. 国际化库（Internationalization）
   国际化库中提供了很多可以开发国际化应用程序的API。国际化是指设计一个应用程序，使它在不变更工程的情况下可以适应各种语言和地区。java.util.Locale中提供了部分国际化功能。
   了解国际化看这里： [java.util.Locale包有什么用][7]

6. JavaBeans库（JavaBeans™ Component API）
   JavaBeans库是开发beans的相关类，包含一些基于javabean™架构的组件，用来操作应用程序中的bean（简单来说bean就是只有属性、get和set方法，代表某种资源的类）。
   了解JavaBeans包看这里：[java.beans包有什么用][8]

7. Java管理扩展库（Java Management Extensions，JMX）
   JMX API是一个标准的API，用来监管各种资源，包括应用程序、设备、服务和Java虚拟机。典型用途包括查询和修改应用程序配置、收集关于应用程序行为的统计数据、发送状态变化通知以及错误的条件。JMX API还包括远程访问，因此远程管理程序可以与正在运行的应用程序交互。主要功能在javax.management包中。
   了解 JMX看这里：[Java种JMX是什么][9]

8. **xml工具库（Java API for XMLProcessing，JAXP）**
   Java平台提供了一组丰富的API（JAXP）来处理xml文档和数据。主要功能在javax.xml包中。
   了解JAXP看这里：[Java中JAXP是什么][10]

9. **Java本地方法接口（Java Native Interface，JNI）**
   Java本地方法接口是一个标准的编程接口，用来编写Java本地方法和将一个JVM嵌入到本地应用程序中。本地方法库的主要目标是为不同平台做二进制兼容性，调用其他平台的方法（常见的如C/C++）。通过native关键字声明本地方法，通过System.loadLibrary加载。
   了解JNI看这里：[Java中JNI是什么][11]

10. **可选扩展机制（Extension Mechanism）**
    可选包是可以用它来扩展核心平台Java类（包括任何相关的本地代码）的包。它的扩展机制允许JVM在Java平台中加载可选扩展类。扩展机制还提供了在尚未安装在jdk或jre的环境中通过url中检索可选包的功能。通过标准可扩展的方式来让Java平台上所有应用使用自定义API。
    了解可选扩展机制看这里：[Java中可选扩展机制是什么][12]

11. **标准覆盖机制（Endorsed Standards Override Mechanism）**
    Java平台是在不断更新中发展的，可能会有需要修改Java平台的时候，标准覆盖机制允许在一定范围内覆盖Java平台的原有内容。标准覆盖机制提供了一种方法，后续版本的类和接口如果实现了标准接口或者本身是独立技术，则可能会纳入Java平台。当然不包括java.lang包中的类。
    了解标准覆盖机制看这里：[Java中标准覆盖机制是什么][13]

]

[

Java千百问_08JDK详解（005）_jdk服务集成类库都有什么

,

点击进入_更多_Java千百问

1、jdk服务集成类库都有什么

了解java核心框架看这里：java核心框架是什么样的
1. **数据库链接工具（Java Database Connectivity，JDBC）**
JDBC API提供了使用java访问数据库的通用接口，使用JDBC 3.0或以上，开发人员编写的应用程序可以访问几乎任何数据源，从关系数据库到电子表格甚至文本文件，JDBC技术还提供了很多通用的基础工具和备用接口。
了解JDBC看这里：[JDBC是什么][3]

1. **远程方法调用（Remote Method Invocation，RMI）**
   远程方法调用能够提供分布式应用程序的开发，提供了Java编写程序之间的远程调用与通信。RMI允许运行在一个JVM上的对象调用运行在另一个JVM上对象的方法（两个JVM有可能是不同的主机）。
   了解RMI看这里：[RMI是什么][4]

2. **接口描述语言（Interface description language，IDL，CORBA)**
   IDL是CORBA（Common Object Request Broker Architecture，通用对象请求代理体系结构）中的描述语言，提供标准的相互操作和连通规范。使得在不同平台上运行的对象和用不同语言编写的程序可以相互通信交流。
   它使分布式、支持Web的Java应用可以基于IIOP协议 （Internet Inter-ORB Protocol，互联网内部对象请求代理协议）透明地调用远程服务。
   JavaIDL运行期组件包括一个兼容全对象请求代理Java ORB（Object Request Broker，对象请求代管者），用于基于IIOP协议实现分布式对象之间的通信。
   了解IDL看这里： [IDL是什么][5]
   了解CORBA看这里：[CORBA是什么][6]
   了解IIOP协议看这里：[IIOP协议是什么][7]

3. **RMI-IIOP协议（RMI-IIOP）**
   RMI以Java为核心，可与采用本机方法与现有系统相连接。RMI可采用自然、直接和功能全面的方式为您提供分布式计算技术，而这种技术可帮助您以不断递增和无缝的方式为整个系统添加Java功能。
   IIOP，Internet Inter-ORB Protocol(互联网内部对象请求代理协议)，它是一个用于CORBA 2.0及兼容平台上的协议。用来在CORBA对象请求代理之间交流的协议。Java中使得程序可以和其他语言的CORBA实现实现互操作性的协议。
   RMI-IIOP出现以前，只有RMI和CORBA两种选择来进行分布式程序设计，二者之间不能协作。RMI-IIOP综合了RMI和CORBA的优点，克服了他们的缺点，使得程序员能更方便的编写分布式程序设计，实现分布式计算。
   了解更多RMI-IIOP看这里：[RMI-IIOP是什么][8]

4. **Java脚本引擎（Scripting for the Java Platform）**
   Java脚本API由Java脚本引擎的接口和类组成，提供Java应用程序中使用脚本框架。此API允许在Java应用程序中执

行使用脚本语言编写的程序。脚本语言程序通常由应用程序的终端用户提供。
了解javax.script包看这里：[javax.script包有什么用][9]

5. **Java命名系统接口（Java Naming and Directory Interface™，JNDI）**
   JNDI提供统一的客户端API，在Java编写的应用程序中提供了命名和目录功能。它被设计成独立于任何特定的命名或目录服务来实现，因此各种各样的服务：新的、已部署的都可以通过一致的方式来访问。JNDI架构由一个API和SPI（服务提供者接口）实现，Java应用程序使用这个API来访问各种各样的命名和目录服务，SPI使各种各样的命名和目录服务接入，允许Java应用程序使用JNDI API来访问他们的服务。主要功能在javax.naming包中。
   了解javax.naming包看这里：[javax.naming包有什么用][10]

]

[

Java千百问_08JDK详解（006）_jdk用户界面类库都有什么

,

点击进入_更多_Java千百问

1、jdk用户界面类库都有什么

了解java核心框架看这里：java核心框架是什么样的
1. **输入法框架（Input Method Framework）**
输入法框架包括文本编辑组件与文本输入。文本输入是软件组件，使得用户输入文本而不是简单的在键盘上打字。它们通常用于输入日本、中国或韩国的语言，输入比键盘按键多出数百倍的文本。当然，框架还支持其他语言，以及完全不同的输入机制，如手写和语音识别。
了解更多输入法框架看这里：[java输入法框架是什么][3]

1. **可接入性库（Accessibility）**
   开发人员通过Java可接入性API，可以轻松地创建残疾人都可以访问的Java应用程序。它使Java兼容了屏幕阅读器、语音识别系统、可刷新的布莱叶盲文显示器等辅助技术。主要功能在javax.accessibility包中。
   了解更多可接入性库看这里：[javax.accessibility包有什么用][4]

2. **打印服务（Print Service）**
   Java™打印服务API，允许所有Java平台上的印刷，包括那些需要占用空间小的文件，比如Java ME概要文件。主要功能在javax.print包中。
   了解java打印服务看这里：[javax.print包有什么用][5]

3. **音乐组件（Sound）**
   Java平台包括一个强大的API来获取、处理和播放音频与MIDI(Musical Instrument Digital Interface，数字乐器接口)的数据。这个API支持高效的声音引擎，保证高质量的音频混合以及MIDI合成能力。主要功能在javax.sound包中。
   了解java音乐组件看这里：[javax. sound包有什么用][6]

4. **图片I/O（Image I/O）**
   Java图像I/O API提供了一个可插拔的架构，来处理存储在文件中或者网络中的图像。它为添加特定格式的插件提供了一个框架，在Java I/O中包含几种常见的格式图像，但第三方可以使用这个API来创建自己的、处理特殊格式的插件。主要功能在javax.imageio包中。
   了解java图片I/O看这里：[javax.imageio包有什么用][7]

5. **2D图像组件（Java 2D™ Graphics and Imaging）**
   Java 2D™ API是一个先进的2D图形、图像组件。它包含艺术线条、文本、图像以及综合模型。还包括对alpha通道图像的支持、颜色空间的定义和转换、丰富的display-oriented成像操作符。主要功能在javax.imageio包中。

6. **抽象窗口工具集（ Abstract Windowing Toolkit， AWT）**
   Java™平台的抽象窗口工具包（AWT）提供了创建用户界面组件(如菜单、按钮、文本框、对话框、复选框等）的API，并通过这些组件来处理用户的输入。此外，AWT还能呈现很多简单的图形（如椭圆、多边形），使开发人员能够选择用户界面布局和字体。主要功能在java.awt包中。
   了解更多AWT看这里：[java中AWT是什么][8]

7. **用户界面开发工具集（Swing）**
   Swing是由Java图形用户界面（GUI）组件所构成，可运行在任一支持Java的本地平台上。Swing API是用Java编写的，并没有任何代码依赖于底层操作系统的GUI工具。这允许Swing GUI组件有一个与操作系统不同的"插入"外观。主要功能在javax.swing包中。
   了解更多Swing看这里：[java中Swing是什么][9]

8. **JavaFX**
   JavaFX是用来开发具有高度互动性、丰富用户体验以及功能强大的客户端的RIA（Rich Internet Applications）程序。它拥有UI控制、嵌入式图形堆栈、现代主题、3D图形处理再加上HTML 5支持等多项功能特性。其工具集在设计思路上专注于性能与图形，且能够在嵌入式系统之上顺畅运作。

了解更多JavaFX看这里：[JavaFX是什么][10]

]

[

Java千百问_08JDK详解（007）_javac是什么

,

点击进入_更多_Java千百问

1、javac是什么

javac即Java编程语言编译器，位于jdk/bin目录下，读取使用Java编程语言编写的源文件（.java），并编译成字节码类文件（.class）。编译器会一并编译源代码中的注解（annotations），但是会移除注释。

了解注解注释看这里：[注解、注释有什么区别][2]

javac还可以隐式编译一些没有在命令行中提及的源文件。当编译源文件时，编译器常常需要它还没有识别出的类型的有关信息。对于源文件中使用、扩展或实现的每个类或接口，编译器都需要其类型信息。这包括在源文件中没有明确提及、但通过继承提供信息的类和接口。

了解继承看这里：java类的继承有什么意义

2、javac如何使用

javac工具可以再安装了jdk的操作系统中执行。了解如何安装jdk看这里：如何安装和配置Jdk
打开操作系统的命令终端，输入相应的命令行，就能够使用javac，具体语法如下：

javac [options] [sourcefiles] [@files]

其中，参数可按任意次序排列。参数说明如下：

options
命令行选项，直接执行javac或者javac -help会显示所有options操作，主要有：

1. -classpath
   类路径，设置用户类的路径，它会覆盖CLASSPATH环境变量中的用户类路径。若既未指定CLASSPATH又未指定-classpath，则用户类路径由当前目录构成。多个路径项用分号”;”进行分隔。

2. -sourcepath
   源路径，指定用来查找类、接口定义的源代码路径。它们可以是目录、jar或zip。要注意的是通过类路径查找，如果找到了其源文件，则会自动被重新编译。与用户类路径一样，多个源路径项用分号”;”进行分隔。
   如果使用包，那么目录或归档文件中的本地路径名必须反映包名。例如要引入package是com.test的源文件（com在src文件夹中），则需要指定com上一级目录：
   javac -sourcepath src Test.java

3. -d
   指定存放生成的类文件的位置，若不指定，则类文件会放在源文件目录。需要指定已有的目录，不能自动创建文件夹。

4. -encoding
   设置源文件编码，例如utf-8。若未指定-encoding选项，则使用平台缺省的编码。

5. -g
   生成所有的调试信息，包括局部变量。缺省情况下，只生成行号和源文件信息。

6. -g:none
   不生成任何调试信息。

7. -verbose
   输出有关编译器正在执行的操作的消息。它包括了每个所加载的类和每个所编译的源文件的有关信息。

8. -version
   打印javac版本信息，该版本即jdk的版本。

sourcefiles、@files
一个或多个要编译的源文件。

- 编译一个文件，直接写源文件名，例如：
  执行：javac Test.java，生成编译文件Test.class。

- 编译多个文件，需要创建一个新文件（没有后缀名），写入需要编译的源文件，然后加上前缀@。例如：
  新文件：javaList，其内容：
  Test1.java
  Test2.java
  Test3.java
  执行：javac @javaList，生成编译文件Test1.class、Test2.class、Test3.class。

3、如何通过代码执行javac

对于javac，我们不但可以在操作系统中直接执行，它也提供了若干API，可以使用这些API通过编写代码来完成编译。

具体看这里：[如何使用javacAPI通过代码编译源文件][5]

]

[

Java千百问_08JDK详解（008）_通过代码如何编译java文件

,

点击进入_更多_Java千百问

1、通过代码如何编译java文件

编译器是一个命令行工具（jdk自带的编译工具javac，了解javac看这里：javac是什么），但也可以使用API来调用（一般的IDE都会使用这一组API来封装自己的编译功能）。编译器遵循Java语言规范（The Java Language Specification，JLS）和Java虚拟机规范（The Java Virtual Machine Specification，JVMS）。
在Java 6之后，提供了标准包来操作Java编译器，这就是javax.tools包。我们使用这个包中的API以及其他辅助包可以定制自己的编译器。通过ToolProvider类的源码我们可以看到，javax.tools这个包中的API最终都是通过tools.jar中的com.sun.tools.javac包来调用Java编译器的。

通过代码编译java大体有如下三种方式，灵活运用这几种方式可以DIY属于自己的编译器：

通过JavaCompiler.run()

最简单的用法即使用JavaCompiler类的run方法，前3个参数分别为：输入信息、输出信息、错误信息，如果为null则默认为：System.in、System.out、System.err。最后一个参数为javac后的命令文本，例如传入Test.java，则等同于在终端执行javac Test.java。

例如：

```
public class Test {
 public static void main(String[] args) throws Exception {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        int run = compiler.run(null, null, null, "-version");
        System.out.println("===" + run);
    }
}
```

执行结果（由于输出信息没有指定，默认打印在System.out中）：

```
javac 1.7.0_79
===0
```

通过JavaCompiler.getTask()编译硬盘中代码

使用JavaCompiler.run方法非常简单，但它确不能更有效地得到我们所需要的信息。一般来说我们都会使用StandardJavaFileManager类（jdk 6或以上），这个类可以很好地控制输入、输出，并且可以通过DiagnosticListener得到诊断信息，而DiagnosticCollector类就是listener（监听）的实现。

具体实例如下：

```
public class Test {
public static void main(String[] args) throws Exception {
        Test.compiler();
}

public static void compiler() throws IOException {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        // 建立DiagnosticCollector对象

        DiagnosticCollector diagnostics = new DiagnosticCollector();

        StandardJavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);

        // 建立源文件对象，每个文件被保存在一个从JavaFileObject继承的类中
        Iterable<? extends JavaFileObject> compilationUnits = fileManager.getJavaFileObjectsFromStrings(Arrays
                .asList("/Users/sunjie/Desktop/works/workspace/my-test/src/com/test/Test.java"));

        // options命令行选项
        Iterable<String> options = Arrays.asList("-d",
                "/Users/sunjie/Desktop/works/workspace/my-test/src/com/test/classes");// 指定的路径一定要存在，javac不会自己创建文件夹
        JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, diagnostics, options, null,
                compilationUnits);

        // 编译源程序
        boolean success = task.call();
        fileManager.close();
        System.out.println((success) ? "编译成功" : "编译失败");

        // 打印信息
        for (Object object : diagnostics.getDiagnostics()) {
            Diagnostic diagnostic = (Diagnostic) object;
            System.out.printf("Code: %s%n" + "Kind: %s%n" + "Position: %s%n" + "Start Position: %s%n"
                    + "End Position: %s%n" + "Source: %s%n" + "Message: %s%n", diagnostic.getCode(),
                    diagnostic.getKind(), diagnostic.getPosition(), diagnostic.getStartPosition(),
                    diagnostic.getEndPosition(), diagnostic.getSource(), diagnostic.getMessage(null));
```

```
        }
}
}
```

运行结果如下：

编译成功

在对应路径下会发现com/test/Test.class文件（Test的包是package com.test，所以会自动在对应目录下生成com/test/路径）。

通过JavaCompiler .getTask()编译内存中代码

JavaCompiler不仅可以编译硬盘上的Java文件，而且还可以编译内存中的Java代码，然后使用reflection来运行它们。我们可以编写一个JavaSourceFromString类，通过这个类可以输入Java源代码。

具体实例如下：

```
public class Test {

    public static void main(String[] args) throws Exception {
        Test.compiler2();
    }
public static void compiler2() throws IOException, IllegalAccessException, IllegalArgumentException,
            InvocationTargetException, NoSuchMethodException, SecurityException, ClassNotFoundException {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

        DiagnosticCollector diagnostics = new DiagnosticCollector();

        // 定义一个StringWriter类，用于写Java程序
        StringWriter writer = new StringWriter();
        PrintWriter out = new PrintWriter(writer);
        // 开始写Java程序
        out.println("public class HelloWorld {");
        out.println(" public static void main(String args[]) {");
        out.println(" System.out.println(\"Hello, World\");");
        out.println(" }");
        out.println("}");
        out.close();

        StandardJavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
        // 为这段代码取个名子：HelloWorld
        SimpleJavaFileObject file = (new Test()).new JavaSourceFromString("HelloWorld", writer.toString());
        Iterable compilationUnits = Arrays.asList(file);
        // options命令行选项
        Iterable<String> options = Arrays.asList("-d",
                "/Users/sunjie/Desktop/works/workspace/my-test/src/com/test/classes");// 指定的路径一定要存在，javac不会自己创建文件夹
        JavaCompiler.CompilationTask task = compiler.getTask(null, fileManager, diagnostics, options, null,
                compilationUnits);

        boolean success = task.call();
        System.out.println((success) ? "编译成功" : "编译失败");
    }

    // 用于传递源程序的JavaSourceFromString类
    class JavaSourceFromString extends SimpleJavaFileObject {

        final String code;

        JavaSourceFromString(String name, String code) {
            super(URI.create("string:///" + name.replace('.', '/') + Kind.SOURCE.extension), Kind.SOURCE);
            this.code = code;
        }

        @Override
        public CharSequence getCharContent(boolean ignoreEncodingErrors) {
            return code;
        }
    }
}
```

运行结果如下：

编译成功

在对应路径下会发现HelloWorld.class文件。

]

[

Java千百问_08JDK详解（009）_java程序如何运行

,

点击进入_更多_Java千百问

1、java程序如何运行

java代码首先需要进行编译，编译为.class类文件，然后通过java命令（执行器）去执行。java命令位于jdk/bin目录下。

了解如何编译看这里：javac是什么
它通过启动Java运行时环境（JRE），然后加载指定类（JRE在启动时会搜索以下路径进行类加载：引导类路径，扩展包路径，用户的类路径），调用类的main()方法（了解更多main方法看这里：main方法是什么）。

java命令可以执行一个jar，这个jar必须包含Main-Class文件，用来指定应用程序的入口。了解如何打jar包看这里：不使用IDE如何打jar包
java命令也可以用来启动一个JavaFX应用程序，同样通过main()方法或者javafx.application.Application。通过javafx.application.Application执行，首先执行器构造类的一个实例，然后调用它的init()方法，之后调用start(javafx.stage.Stage)方法。

javaw命令与java命令是相同的，除了一点：javaw没有相关联的控制台窗口。当你不希望一个出现一个命令提示符窗口时，请使用javaw。当然，如果javaw命令发生错误，则会讲所务信息通过一个对话框显示出来。

2、java命令如何使用

java命令可以再安装了jdk的操作系统中执行。了解如何安装jdk看这里：如何安装和配置Jdk
打开操作系统的命令终端，输入相应的命令行，就能够使用java，具体语法如下：

java [options] classname [args]

java [options] -jar filename [args]

javaw [options] classname [args]

javaw [options] -jar filename [args]

其中，参数可按任意次序排列。参数说明如下：

options
命令行选项，直接执行java或者java -help会显示所有options操作，主要有：

1. -classpath
   类路径，设置用户类的路径，它会覆盖CLASSPATH环境变量中的用户类路径。若既未指定CLASSPATH又未指定-classpath，则用户类路径由当前目录构成。多个路径项用分号";"进行分隔。

2. -version
   输出产品版本并退出，一般用于测试jdk是否安装成功以及查看当前环境下使用的jdk版本。

3. -version:<值>
   需要指定的版本才能运行。

4. -verbose:[class|gc|jni]
   启用时详细日志输出。

5. -D<名称>=<值>
   设置系统属性。

6. 其他jvm配置
   可以在这里设置jvm运行参数，例如内存大小-Xms10M -Xmx10M等。
   了解jvm常见配置看这里：jvm常见配置都有哪些

7. -esa | -enablesystemassertions
   启用系统断言

8. -dsa | -disablesystemassertions
   禁用系统断言

classname
所要执行的类名，注意这里是指类名，不是文件名，所以不能加文件后缀名。例如：

java Test

filename
所要执行的jar文件名，这里是文件名，需要文件后缀名。例如：

java -jar Test.jar

args
调用main(String[] args)方法的入参（String类型），多个参数通过空格分割，没有参数不用填写。

例子如下：

```
public class TestMain {

    public static void main(String[] args) throws Exception {
        if (args.length > 0) {
            for (String arg : args) {
                System.out.println("args:" + arg);
            }
        }
        if (args.length <= 0) {
            throw new Exception("Exception");
        }
    }
}
```

首先编译：java Test.java
然后执行：java -Xms10M -Xmx10M Test 0 1 2
结果如下：

args:0
args:1
args:2

]

[

Java千百问_08JDK详解（010）_java、javaw、javaws有什么区别

,

点击进入_更多_Java千百问

1、java、javaw、javaws有什么区别

java、javaw、javaws这三个都是jdk自带的三个工具，都在jdk/bin路径下，这三个工具都是为了启动java应用存在的，具体区别如下：

java
java命令即java应用的执行器，通过它执行的应用日志会再控制台console显示输出与错误信息。

了解更多java命令看这里：java程序如何运行

javaw
javaw与java一样，是java应用的执行器，不同的是它不会再控制台console显示输出与错误信息，取而代之的是使用文本记录这些信息。主要用来启动基于GUI（Graphical User Interface，用户图形接口）的应用程序。

javaws
Java Web Start，是用来启动通过web来描述的项目的。我们需要一个jnlp文件（Java Network Launching Protocol，java网络执行协议文件），通过javaws jnlp来启动应用。jnlp文件中描述了应用的url、应用程序的基本信息（应用名称、版本、供应商、 应用icon等）、必要的系统环境等。

了解jnlp文件看这里：[jnlp文件是什么][3]

它首先通过jnlp文件中的url下载应用并启动，它是一个很有用的获取资源命令，通过中央统一控制，提供更新，可以确保所有的用户都是使用最新的应用。当应用程序被调用时，它被缓存在本地计算机。每次启动时，它会检查是否有更新。

具体使用语法如下：

javaws [run-options]

javaws [control-options]

参数说明如下：

run-options

命令行运行时选项，多个参数允许无顺序。直接执行javaws会显示所有操作，常见的如下：

1. -verbose
   启用详细输出。

2. -offline
   以脱机模式运行应用程序。

3. -system
   仅从系统高速缓存运行应用程序。

jnlp

jnlp文件的路径。

control-options

命令行控制选项，多个参数允许无顺序。常见的如下：

1. -viewer
   在Java控制面板中显示高速缓存查看器，操作系统会自动打开一个查看器。

2. -clearcache
   从高速缓存删除所有未安装的应用程序。

3. -uninstall
   从高速缓存删除所有应用程序。

4. -uninstall jnlp-file
   从高速缓存删除指定应用程序。

5. -import [导入选项] jnlp-file
   将指定应用程序导入高速缓存，导入选项有如下几个：
   -silent：以无提示模式 (不出现用户界面) 导入
   -system：将应用程序导入系统高速缓存
   -codebase ： 从给定的代码库检索资源
   -shortcut：以用户接受提示的方式安装快捷方式
   -association ： 以用户接受提示的方式安装关联

]

[

Java千百问_08JDK详解（011）_jnlp文件是什么

,

点击进入_更多_Java千百问

1、jnlp文件是什么

JNLP（Java Network Launching Protocol，java网络执行协议文件）是java提供的一种可以通过浏览器直接执行java应用程序的途径，它使你可以直接通过一个网页上的url链接打开一个java应用程序。

Java桌面应用程序可以通过JNLP的方式发布。如果版本升级后，不需要再向所有用户发布版本，只需要更新服务器的版本，这就相当于让java应用程序有了web应用的优点。如果你使用JNLP打包一个应用程序，那么它能够：

- 安装并且使用正确版本的JRE（java运行时环境）。
- 从浏览器或者桌面加载应用程序。
- 检测新版本，自动下载最新的版本。
- 为了加速启动速度，会在本机缓存应用程序需要的类。
- 在必要的情况下下载原始的库。
- 以安全的方式使用诸如文件系统这样的本机资源。
- 自动定位和加载外部依赖资源。

我们可以使用javaws命令下载并运行jnlp文件中维护的java应用程序。了解javaws看这里：java、javaw、javaws有什么区别

2、使用jnlp有什么好处

使用JNLP文件发布应用，具有以下优点:

- 其他人不能改变你JNLP文件的内容，例如：添加一个随机的库，或者改变应用程序的信息。
- 允许在应用程序中使用任意的Java虚拟机（JVM）选项和Java系统属性。
- 防止他人在HTML浏览器中直接引用你的applet JAR文件。

3、jnlp文件是什么样的

jnlp是一个标准的文件， jnlp文件中描述了应用的url、应用程序的基本信息（应用名称、版本、供应商、应用icon等）、必要的系统环境等。

模版名称：JNLP-INF/APPLICATION_TEMPLATE.JNLP。名称必须是大写。模版内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="*">
<information>
<title>SampleApp</title>
<vendor>Sample Company</vendor>
<icon href="*"/>
<offline-allowed/>
</information>
<resources>
<java version="1.3+"/>
<jar href="SampleApp.jar"/>
</resources>
<application-desc main-class="com.sample.SampleApp"/>
</jnlp>
```

]

[

Java千百问_02基本使用（012）_如何编写多线程Socket程序

,

点击进入_更多_Java千百问

1、如何编写多线程Socket程序

了解Socket看这里：Socket是什么
多线程Socket与单线程类似，只是使用了多线程的方式来管理连接，主线程负责接收连接，在接到连接后变创建新的线程，每个线程负责与自己的客户端进行通信。

了解单线程Socket看这里：如何编写单多线程Socket程序

与单线程Socket例子相比来说，服务端可以与多个客户端进行通信了，不过多线程频繁的创建与销毁便会带来很大的资源开销，而系统的网络资源等都是有限的。因此一般会引入线程池，可以在某种程度上重用线程，减少线程的创建和销毁的次数以减少开销。

我们的代码也分为客户端和服务端两部分。服务端的代码中包含了使用和不使用线程池的两种方式。

服务端代码：

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SocketThreadPoolDemoServer {

    private int port = 8000;

    private ServerSocket serverSocket;

    private ExecutorService executorService; // 连接池

    private final int POOL_SIZE = 1; // 连接池大小 , 若为 1 时最多支持 2 线程

    public SocketThreadPoolDemoServer() throws Exception {
        serverSocket = new ServerSocket(port);
        executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors() * POOL_SIZE);// 初始化线程池
        System.out.println("waitting connet...");

    }

    /**
     *
     * 接受连接
     *
     * @author sunjie at 2016年6月14日
     *
     */
    public void service() {
        Socket socket = null;
        while (true) {
            try {
                socket = serverSocket.accept();
                executorService.execute(new Handler(socket)); // 使用连接池
                // new Thread(new Handler(socket)).start();// 不使用连接池
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     *
     * 线程类，负责维持与一个客户端的通信
     *
     * @author sunjie at 2016年6月14日
     *
     */
    class Handler implements Runnable {

        private Socket socket = null;
```

```
        public Handler(Socket socket) {
            this.socket = socket;
        }

        @Override
        public void run() {
            System.out.println("new connection accepted:" + socket.getInetAddress() + ":" + socket.getPort());
            try {
                BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream(), "UTF-8"));
                PrintWriter writer = new PrintWriter(socket.getOutputStream());
                String msg = null;
                while ((msg = reader.readLine()) != null) {
                    System.out.println("from " + socket.getInetAddress() + ":" + socket.getPort() + ", receive msg:"
                            + msg);
                    writer.println(msg);
                    writer.flush();
                    if ("close".equals(msg)) {
                        break;
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    if (socket != null) {
                        socket.close();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) throws Exception {
        new SocketThreadPoolDemoServer().service();
    }
}
```

运行服务端代码后，程序会一直进行监听，直到接收到客户端请求为止。结果如下：

waitting connet…

客户端代码（与单线程完全相同）：

```
public class SocketDemoClient {

    private String host = "127.0.0.1";// 要发送给服务端的ip

    private int port = 8000;// 要发送给服务端的端口

    private Socket socket;

    public SocketDemoClient() throws Exception {
        socket = new Socket(host, port);// 构造Socket客户端，并与连接服务端
    }

    public void talk() throws IOException {
        try {
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream(), "UTF-8"));
            PrintWriter writer = new PrintWriter(socket.getOutputStream());
            // 读取本地控制台的消息
            BufferedReader localReader = new BufferedReader(new InputStreamReader(System.in));
            String msg = null;
            while ((msg = localReader.readLine()) != null) {
                writer.println(msg);
                writer.flush();
                System.out.println("send msg:" + reader.readLine());
                if ("close".equals(msg)) {
                    break;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                socket.close();
            }
        }
    }

    public static void main(String[] args) throws Exception {
```

```
        new SocketDemoClient().talk();
    }
}
```

由于我们要测试多个客户端连接同一个服务端，所以我们需要多次运行客户端代码。这里我们运行两次之后（称为客户端1、客户端2），查看服务端的Console，会出现以下结果，说明已经连接成功：

waitting connet…
new connection accepted:/127.0.0.1:59593
new connection accepted:/127.0.0.1:59596

我们在去客户端1的Console中输入我们要发送的消息”维护世界和平”，回车确定后，客户端1的Console出现以下结果，消息已经发出：

send msg:维护世界和平

再去客户端2的Console中输入”好好学习天天向上”，回车确定后，客户端2的Console出现以下结果，消息已经发出：

send msg:好好学习天天向上

在服务端的Console中，我们会看到如下结果，说明两个客户端的消息已经被接受：

waitting connet…
new connection accepted:/127.0.0.1:59593
new connection accepted:/127.0.0.1:59596
from /127.0.0.1:59593, receive msg:维护世界和平
from /127.0.0.1:59596, receive msg:好好学习天天向上

]

[

Java千百问_08JDK详解（013）_JVMTI是什么

,

点击进入_更多_Java千百问

1、JVMTI是什么

JVMTI（JVM Tool Interface）是JPDA体系中的最底层， 由Java虚拟机提供的native编程接口，是JVMPI（Java Virtual Machine Profiler Interface）和 JVMDI（Java Virtual Machine Debug Interface）的更新版本。

了解JPDA体系看这里：JPDA是什么
从它的发展中我们就可以知道，JVMTI提供了调试（debug）和分析（profiler）功能；同时，它还有监听（Monitoring），线程分析（Thread analysis）以及覆盖率分析（Coverage Analysis）等功能。正是由于JVMTI的强大功能，它是实现Java调试器，以及其它Java运行态测试与分析工具的基础。目前已有很多成熟的集成工具提供了JVMTI的实现（例如Sun、IBM以及一些开源项目如Apache Harmony DRLVM），这些工具虽然强大易用，但是在一些特定情况下，开发者常常会有一些特殊的需求，这个时候就需要定制工具来达成目标。

JVMTI是一套本地代码接口，因此我们需要使用C/C++以及JNI。开发时一般采用建立一个Agent（通过C++编写）的方式来使用JVMTI，它可以使用JVMTI函数、设置回调函数、从JVM中得到当前的运行态信息，还可以操作虚拟机的运行态。

2、如何加载JVMTI agent

当我们把Agent编译成一个动态链接库之后，我们可以通过两种方式加载Agent：启动加载模式、活动加载模式。具体如下：

启动加载模式

在Java程序启动时加载它，其实是在java启动时指定加载agent，如下：

```
-agentlib:<agent-lib-name>=<options>
```

注意，这里的路径是环境变量的相对路径，例如 java -agentlib:libagent=opt，java启动时会PATH环境变量定义的路径处装载libagent.so

```
-agentpath:<path-to-agent>=<options>
```

这里是绝对路径，例如 java -agentpath:/home/admin/agentlib/libagent.so=opt

活动加载模式

Java 5之后可以在运行时加载agent，通过com.sun.tools.attach包的API来实现（需要引入${JAVA_HOME}/lib/tools.jar）。使用非常简单，如下：

```
public class TestAgent {
    public static void main(String[] args) throws AttachNotSupportedException, IOException, AgentLoadException,
            AgentInitializationException {
        String pid = "831"; // 想要装载的java进程id
        String agentPath = "/Users/sunjie/Desktop/libagent.so"; // agent.so的路径
        String options = null;// 传入agent的参数
        VirtualMachine virtualMachine = com.sun.tools.attach.VirtualMachine.attach(pid);
        virtualMachine.loadAgentPath(agentPath, options);
        virtualMachine.detach();
    }
}
```

了解AttachAPI看这里：[AttachAPI是什么][3]

3、JVMTI agent是如何工作的

Agent的启动

Agent是在Java虚拟机启动时加载的，这个时间点上：

- 所有的Java类都未被初始化
- 所有的对象实例都未被创建
- 所有的Java代码都没有被执行

但在这个时候，我们已经可以：

- 操作JVMTI的Capability参数
- 使用系统参数

通过启动加载模式加载Agent之后，虚拟机会先寻找一个Agent入口函数：

```
JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *options, void *reserved)
```

如果是运行加载模式，则是：

```
JNIEXPORT jint JNICALL Agent_OnAttach(JavaVM *jvm, char *options, void *reserved);
```

在这个函数中，虚拟机传入了一个JavaVM指针，以及命令行的参数。我们通过*jvm可以获取jvmtiEnv，即可以使用JVMTI函数，当然不同的JVM实现提供的函数细节可能不一样，但是使用的方式一致。如下：

```
jvmtiEnv *jvmti;
(*jvm)->GetEnv(jvm, &jvmti, JVMTI_VERSION_1_1);
```

这里第二个参数为版本信息，不同的JVMTI环境所提供的功能、处理方式可能有所不同，不过它在同一个虚拟机中会保持不变。

Agent的卸载

当Agent完成任务，或者JVM关闭的时候，虚拟机会调用函数来完成最后的清理任务，如下：

```
JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *jvm)
```

4、如何编写JVMTI agent程序

只要有一定的C++基础就可以编写JVMTI agent，具体看这里：[如何编写JVMTI agent程序][4]

]

[

Java千百问_08JDK详解（014）_如何编写JVMTI agent程序

,

点击进入_更多_Java千百问

1、如何编写JVMTI agent程序

了解JPDA看这里：JPDA是什么
了解JVMTI看这里：JVMTI是什么
我们需要使用C++编写agent程序，JVM在不同时机回调下面的接口函数：

```
JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *jvm, char *options, void *reserved)

JNIEXPORT jint JNICALL Agent_OnAttach(JavaVM *jvm, char *options, void *reserved);

JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *jvm)
```

其中第一个是jvm启动时调用，第二个是jvm运行时发出attach时调用，第三个是jvm卸载时调用。

其中*jvm参数传入JavaVM指针，可以用来操作JVM；*options参数是命令行传入的参数；*reserved是一个预留参数。

给出运行加载模式实现的一个简单功能：打印jvm内所有已经装载成功的class。具体如下：

```
/*
*  JVMTI agent
*
*  Created on: 2016-07-02
*      Author: sunjie
*/
#include <jvmti.h>
#include <string>
#include <cstring>
#include <iostream>
#include <list>
#include <map>
#include <set>
#include <stdlib.h>
#include <jni_md.h>

JNIEXPORT jint JNICALL Agent_OnAttach(JavaVM *jvm, char *options,
                void *reserved) {
        jvmtiEnv *jvmti;
        jint result = jvm->GetEnv((void **) &jvmti, JVMTI_VERSION_1_1);
        if (result != JNI_OK) {
                printf("ERROR: Unable to access JVMTI!\n");
        }
        jvmtiError err = (jvmtiError) 0;
        jclass *classes;
        jint count;

        err = jvmti->GetLoadedClasses(&count, &classes);//获取class
        if (err) {
                printf("ERROR: JVMTI GetLoadedClasses failed!\n");
        }
        for (int i = 0; i < count; i++) {
                char *sig;
                jvmti->GetClassSignature(classes[i], &sig, NULL);//获取并打印class签名
                printf("cls sig=%s\n", sig);
        }
        return err;
}

JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm) {
        // nothing to do
}
```

我们将它命名为agent.cpp，下一步就是将它编译为动态链接库，不同操作系统的命令并不相同，这里以mac系统为例，具体如下：

```
g++ -I${JAVA_HOME}/include/ -I${JAVA_HOME}/include/darwin Agent.cpp -fPIC -shared -o libagent.so
```

由于是mac系统，所以引入了${JAVA_HOME}/include/darwin，linux和windows对应的应该是/include/linux、\include\win32。
执行后，我们会看到生成了了libagent.so文件。

2、如何测试agent程序

我们已经有了agent，下面我们测试一下。测试需要有一个目标JVM，所以我们简单模拟一个，具体如下：

```
public class TestMain {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("JVMTI agent Test start");
        int i = 0;
        while (true) {
            Thread.sleep(1000);
            i++;
        }
    }
}
```

死循环的目的在于不能让它结束，方便我们使用JVMTI agent进行操作。运行后结果如下：

JVMTI agent Test start

这时我们需要通过AttachAPI为运行中的JVM加载我们的agent，我们首先获取到目标JVM的pid（通过进程监控获取），然后指定agent动态资源库的路径，具体如下：

了解更多 AttachAPI看这里：[AttachAPI是什么][4]

```
public class TestAgent {
    public static void main(String[] args) throws AttachNotSupportedException, IOException, AgentLoadException,
            AgentInitializationException {
        String pid = "831"; // java进程pid
        String agentPath = "/Users/sunjie/Desktop/libagent.so"; // agent.so的路径
        String options = null;// 传入agent的参数
        VirtualMachine virtualMachine = com.sun.tools.attach.VirtualMachine.attach(pid);
        virtualMachine.loadAgentPath(agentPath, options);
        virtualMachine.detach();
    }
}
```

这里我们需要引入${JAVA_HOME}/lib/tools.jar包，引入com.sun.tools.attach包。运行后，在目标JVM的Console中出现如下结果：

JVMTI agent Test start
cls sig=Ljava/lang/ClassLoader$NativeLibrary;
cls sig=Ljava/util/concurrent/ConcurrentMap;
cls sig=Ljava/lang/Error;
cls sig=[Ljava/lang/Error;
cls sig=Ljava/util/Set;
cls sig=Ljava/util/WeakHashMap;
cls sig=Ljava/lang/ref/Reference;
cls sig=[Ljava/lang/ref/Reference;
cls sig=Ljava/lang/StackOverflowError;
cls sig=Ljava/io/Serializable;
….

成功打印出了所有已经装载成功的class。

顺便提一句，如果采用启动加载模式，则需要在运行TestMain时加上如下参数：

```
java -agentlib:Agent=opt1 TestMain
```

注意，这里传入了*options参数：opt1，不过，我们的agent并没有使用他。

了解更多JVMTI的功能看这里：[JVMTI提供哪些功能][5]

]

[

Java千百问_08JDK详解（015）_JVMTI提供哪些功能

,

点击进入_更多_Java千百问

1、JVMTI提供哪些功能

了解JVMTI看这里：JVMTI是什么
编写JVMTI程序看这里：如何编写JVMTI agent程序
JVMTI 的功能非常丰富，包含了虚拟机中线程、内存堆/栈、类/方法/变量、事件/定时器处理、代码调试等多种功能，这里我们介绍一些常用的功能。

调试功能

调试功能是JVMTI的基本功能之一，这主要包括了设置断点、调试等，在JVMTI里面，设置断点的API本身很简单：

```
jvmtiError SetBreakpoint(jvmtiEnv* env, jmethodID method, jlocation location)
```

jlocation这个数据结构在这里代表的是对应方法方法中一个可执行代码的行数。在断点发生的时候，虚拟机会触发一个事件，我们可以使用在上文中介绍过的方式对事件进行处理。

事件处理和回调函数

使用JVMTI一个基本的方式就是设置回调函数，在某些事件发生的时候触发并作出相应的动作。
因此这一部分的功能非常基本，回调事件包括虚拟机初始化/开始运行/结束、类的加载、方法出入、线程始末等等。如果想对某些事件进行处理，我们首为该事件写一个函数，然后在jvmtiEventCallbacks这个结构中指定相应的函数指针。

比如，我们对线程启动感兴趣，并写了一个HandleThreadStart函数，那么我们需要在Agent_OnLoad函数里加入：

```
jvmtiEventCallbacks eventCallBacks;
memset(&ecbs, 0, sizeof(ecbs)); // 初始化
eventCallBacks.ThreadStart = &HandleThreadStart; // 设置函数指针
jvmti->SetEventCallbacks(eventCallBacks, sizeof(eventCallBacks));
```

在虚拟机运行过程中，一旦有线程开始，虚拟机就会回调HandleThreadStart方法。

设置回调函数的时候，需要注意：虚拟机不保证回调函数同步，比如，好几个线程同时开始运行了，这个HandleThreadStart就会被同时调用几次.

内存控制和对象获取

内存控制是一切运行态的基本功能。 JVMTI除了提供最简单的内存申请和撤销之外（这块内存不受 Java 堆管理，需要自行进行清理工作，不然会造成内存泄漏），也提供了对Java堆的操作。众所周知，Java堆中存储了Java的对象，通过对堆的操作，可以很容易的查找任意的对象，还可以强行执行垃圾收集工作。

JVMTI中没有提供一个直接获取的方式，而是使用一个迭代器（iterater）的方式遍历，由此可见，虚拟机对对象的管理并非是哈希表，而是某种树/图方式：

```
jvmtiError FollowReferences(jvmtiEnv* env,
    jint heap_filter,
    jclass klass,
    jobject initial_object,// 该方式可以指定根节点
    const jvmtiHeapCallbacks* callbacks,// 设置回调函数
    const void* user_data)
```

或者

```
jvmtiError IterateThroughHeap(jvmtiEnv* env,
    jint heap_filter,
    jclass klass,
    const jvmtiHeapCallbacks* callbacks,
    const void* user_data)// 遍历整个 heap
```

在遍历的过程中，我们可以设定一定的条件，例如指定某一个类的对象，并设置一个回调函数，如果条件被满足，回调函数就会被执行。还可以在回调函数中对当前传回的指针进行打标记（tag）操作，在遍历中，只能对满足条件的对象进行tag，然后再使用GetObjectsWithTags函数获取需要的对象。如下：

```
jvmtiError GetObjectsWithTags(jvmtiEnv* env,
    jint tag_count,
    const jlong* tags, // 设定特定的 tag，即我们上面所设置的
    jint* count_ptr,
    jobject** object_result_ptr,
    jlong** tag_result_ptr)
```

如果你仅仅想对特定Java对象操作，应该避免设置其他类型的回调函数，否则会影响效率。多增加一个primitive的回调函数，可能会使整个操作效率下降一个数量级。

线程和锁

在JVMTI中也提供了很多API进行线程的操作，包括查询当前线程状态、暂停、恢复或者终端线程，还可以对线程锁进行操作。我们可以获得特定线程所拥有的锁：

```
jvmtiError GetOwnedMonitorInfo(jvmtiEnv* env,
    jthread thread,
    jint* owned_monitor_count_ptr,
    jobject** owned_monitors_ptr)
```

也可以获得当前线程正在等待的锁：

```
jvmtiError GetCurrentContendedMonitor(jvmtiEnv* env,
    jthread thread,
    jobject* monitor_ptr)
```

我们可以通过以上接口设计自己的算法来判断是否死锁。

JVMTI还提供了一系列的监视器（Monitor）操作，来帮助我们在native环境中实现同步，主要的操作是构建监视器（CreateRawMonitor），获取监视器（RawMonitorEnter），释放监视器（RawMonitorExit），等待和唤醒监视器(RawMonitorWait,RawMonitorNotify)等操作，通过这些简单锁，程序的同步操作可以得到保证。

]

[

Java千百问_01基本概念（016）_32位和64位计算机有什么区别

,

点击进入_更多_Java千百问

1、32位和64位计算机有什么区别

我们通常说的32位、64位计算机是指计算机的CPU位数。当然很早还有8位、16位的CPU，以Intel的80x86系列来说，8位的8080，16位的8086、8088、80186、80286，而32位的CPU最早始于80386，64位就是大家熟悉的EM64T技术以及AMD的x86-64。当然不同的厂商间同位数的CPU内部有很大的区别，但是它们的核心都是一样：CPU处理能力为64位。

这个位数指的是CPU的通用寄存器（GPRs，General-Purpose Registers，寄存器可以简单理解为一个可以暂存指令、数据和地址的空间，CPU运算时的结果都会暂时放在这里）的指令集、寻址能力。

一般来说，相比较32位的CPU来说，64位CPU最为明显的变化就是寄存器和指令指针升级到64位、内存寻址能力提高到64位，还有其他变化例如增加了8个64位的通用寄存器。更高位数的CPU，可以进行更大范围的整数运算，同时可以支持更大的内存。具体如下：

- 从运算来说，32位处理器一次只能处理32位，也就是4个字节的数据，而64位处理器一次就能处理64位，即8个字节的数据。
- 从内存来说，传统32位处理器的寻址空间最大不足4G（理论上有2^32个物理地址），形成了运行效率的瓶颈。而64位的处理器在理论上则可以将近达到1700万个TB（2^64个，大到惊人）。

2、CPU位数大小有什么影响

一个简单的例子可以说明CPU位数的影响，对于16位CPU，指令集只能操作16bit数据和16bit地址。不同CPU的寄存器、指令集不同，要区别对待，这里以8086来说明。

将16bit数据放入寄存器中

例如：

```
MOV AX,1234H          ;向寄存器 AX 传入数据 1234H
MOV AH,56H              ;向寄存器 AX 的高 8 位寄存器 AH 中传入数据 56H
MOV AL,78H    ;向寄存器 AX 的低 8 位寄存器 AL 中传入数据 78H
```

这里要说明的一点，第一句我们向AX寄存器（累加寄存器）中存放了一个16bit的数1234H，但实际是AX由AH、AL两个寄存器组成，所以可以直接操作AH、AL这两个8位寄存器。
如果我们想在一个寄存器中存入一个超过16bit的数，在16位CPU下是不可能的。如果想处理16bit的数，只能借助其他寄存器，分段处理。

获取16bit内存地址中的数据

例如：

```
MOV  BX,1000H
MOV DS, BX  ;向DS段寄存器传入1000H，由于8086不支持直接将数据传入段寄存器，所以只能借助其他寄存器传值。
MOV AX,[1234H]     ;将内存地址1000H:1234H中的值读到AX寄存器中
```

这里要说明的一点，8086的物理地址支持每次传20位的地址，但是由于16位CPU的指令集只能支持16bit，最大的寻址空间理论值为2^16（64K），为了能够支持2^20个地址（1M），所以需要将分为段地址和偏移地址，表现形式如1000H:1234H。

了解CPU物理地址形成看这里：[CPU物理内存地址如何形成][2]

]

[

Java千百问_08JDK详解（017）_Javadoc是什么

,

点击进入_更多_Java千百问

1、Javadoc是什么

javadoc是Sun公司提供的一个技术，它从程序源代码中抽取类、方法、成员等注释，形成一个和源代码配套的API帮助文档。也就是说，只要在编写程序时以一套特定的标签作注释，在程序编写完成后，通过Javadoc就可以形成开发文档了。

了解注释看这里：注释是什么

2、如何使用Javadoc

java为javadoc技术独立出了一个工具，它位于JAVA_HOME/bin/路径中，在终端中执行如下命令：

```
javadoc 文件名.java
```

javadoc工具就会根据指定的文件生成对应的文档，默认生成在同目录，我们看下面一个实例。

首先我们写一个java类，命名为Test.java，并为其添加注释：

```
/**
 *
 * 测试
 *
 * @author sunjie at 2016年7月7日
 *
 */
public class Test {

    /**
     *
     * 主方法
     *
     * @author sunjie at 2016年7月7日
     *
     * @param args
     * @throws UnsupportedEncodingException
     */
    public static void main(String[] args) throws UnsupportedEncodingException {

    }

    /**
     *
     * 文档方法
     *
     * @author sunjie at 2016年7月7日
     *
     * @param testStr
     *            文档参数
     * @return 我是返回值
     */
    public Boolean testJavadoc(String testStr) {
        return Boolean.FALSE;
    }

}
```

我们通过终端访问改文件的路径，执行一下命令：

```
javadoc Test.java
```

我们会看到结果：

```
正在加载源文件Test.java...
正在构造 Javadoc 信息...
标准 Doclet 版本 1.7.0_79
正在构建所有程序包和类的树...
正在生成/com/test/Test.html...
正在生成/com/test/package-frame.html...
正在生成/com/test/package-summary.html...
正在生成/com/test/package-tree.html...
正在生成/constant-values.html...
正在构建所有程序包和类的索引...
正在生成/overview-tree.html...
正在生成/index-all.html...
正在生成/deprecated-list.html...
正在构建所有类的索引...
正在生成/allclasses-frame.html...
正在生成/allclasses-noframe.html...
正在生成/index.html...
正在生成/help-doc.html...
```

这时，我们去Test.java路径，会发现生成了若干html文件，我看找到index.html并打开，开发文档成功生成，如下：

3、javadoc注释标签有哪些

在上面例子中我们可以看到，每一段注释都用了@author，这些叫做注释标签，根javadoc相关的主要有：

1. @author
   对类的说明，标明开发该类模块的作者

2. @version
   对类的说明，标明该类模块的版本

3. @see
   对类、属性、方法的说明，描述相关主题

4. @param
   对方法的说明，对方法中某参数的说明

5. @return
   对方法的说明，对方法返回值的说明

6. @throws
   对方法的说明，对方法可能抛出的异常进行说明

]

[

Java千百问_08JDK详解（018）_JConsole是什么

,

点击进入_更多_Java千百问

1、JConsole是什么

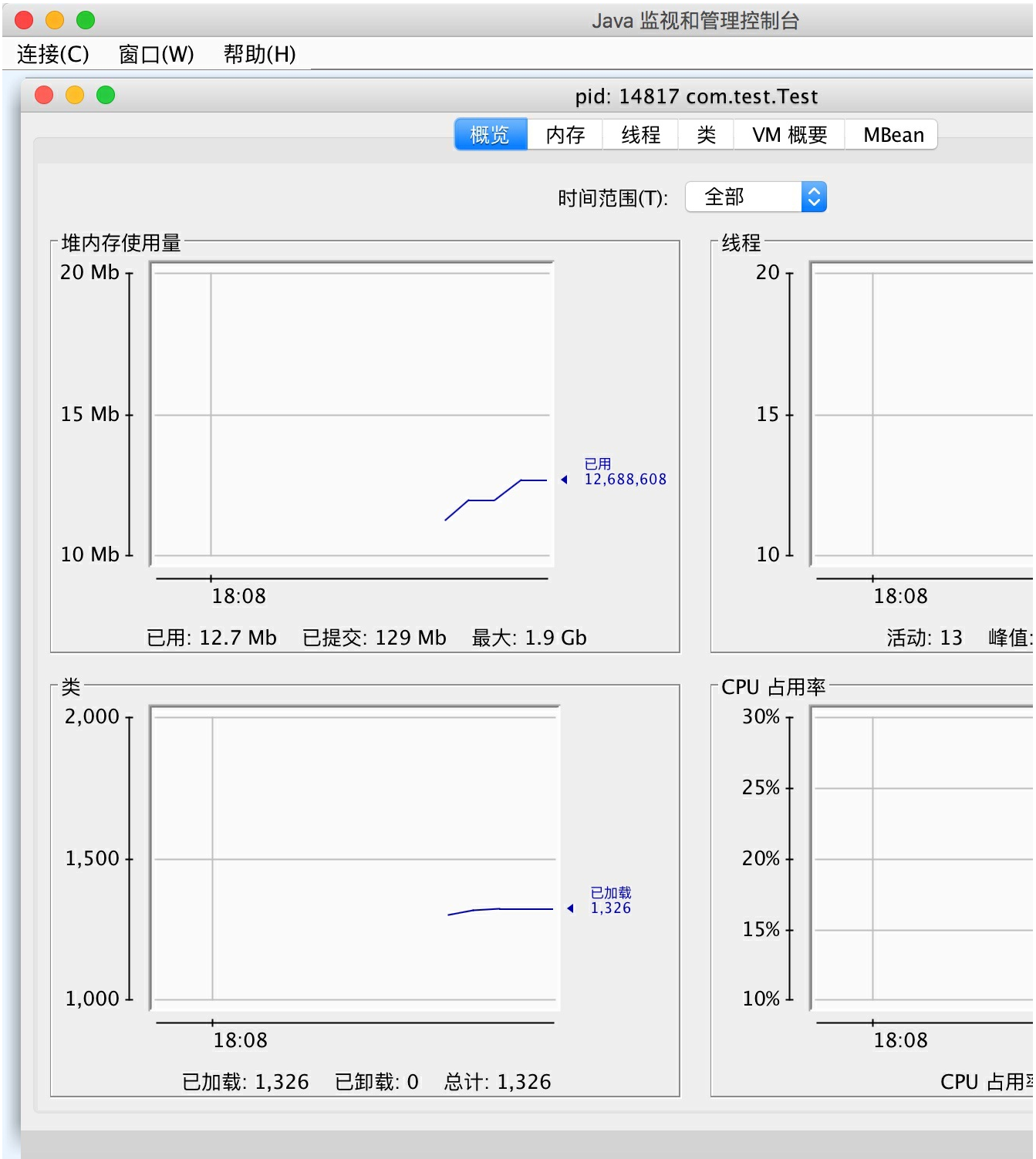
JConsole是一个Java程序性能分析器，可以从命令行或在终端中运行。可以用来监控Java程序的性能以及跟踪代码。

同时在com.sun.tools.jconsole包中提供了一组接口，允许访问JConsole，对线程、内存、类、对象进行分析和统计。

2、如何使用JConsole工具

我们这里主要介绍jconsole工具的用法，在JAVA_HOME/bin路径中，我们可以直接打开他。如下图：

我们可以对正在运行的本地进程或者远程进程进行连接，连接成功后如下图：



首先，我们看到，Jconsole提供了若干图表来展示Java程序的几个关键指标，例如堆内存、线程、类、CPU使用率，我们可以通过切换Tab来查看详情。

通过Jconsole，我们能够清楚的了解Java程序的运行状态，方便我们对他的优化和监控。具体的使用由于使图形化界面，相对比较简单。

了解监控jvm的运行情况看这里：如何监控jvm的运行情况

]

[

Java千百问_08JDK详解（019）_jdk工具包有哪些工具

,

点击进入_更多_Java千百问

jdk工具包有哪些工具

JDK工具包中提供了很多工具和实用的程序。涵盖了基本工具（ javac, java, javadoc, apt, appletviewer, jar, jdb, javah, javap, extcheck），安全工具，国际化工具，RMI工具，IDL， RMI-IIOP工具，部署工具，java插件工具，java Web工具，监控管理工具，故障排查工具等。

这里我们只进行以下简单说明，不一一介绍，具体参见oracle官方文档：http://docs.oracle.com/javase/8/docs/technotes/tools/index.html

基本工具包

包括javac、java、javadoc、apt、appletviewer、jar、jdb、javah、javap、extcheck

了解javac看这里：javac是什么
了解java运行看这里：java程序如何运行

安全工具包

包括keytool, jarsigner, policytool, kinit, klist, ktab

国际化工具包

包括native2ascii

RMI工具包

包括rmic, rmiregistry, rmid, serialver

IDL+RMI-IIOP包

包括tnameserv, idlj, orbd, servertool

部署工具包

包括javapackager, pack200, unpack200

java Web工具包

包括javaws

了解javaws看这里：java、javaw、javaws有什么区别

监控管理工具包

包括jcmd, jconsole, jmc, jvisualvm, jps, jstat, jstatd

故障排查工具包

包括jinfo, jhat, jmap, jsadebugd, jstack

Web Services工具包

包括schemagen, wsgen, wsimport, xjc

脚本工具包

包括jrunscript

]