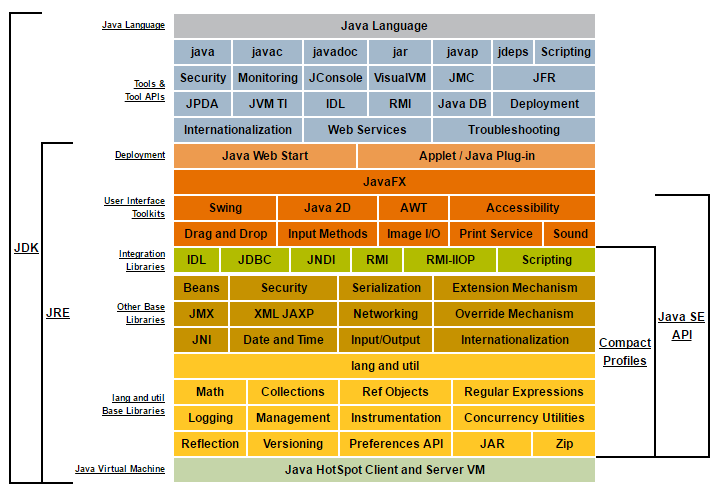
# java技术体系

JCP官方定义Java技术体系包括了以下几个组成部分：

* Java程序设计语言
* 各种硬件平台上的Java虚拟机实现
* Class文件格式
* Java类库API
* 来自商业机构和开源社区的第三方Java类库

**Java开发工具包**（Java Development Kit，JDK）。JDK 包含了 JRE，同时还包括 Java源码的编译器 javac、监控工具 jconsole、分析工具 jvisualvm等。

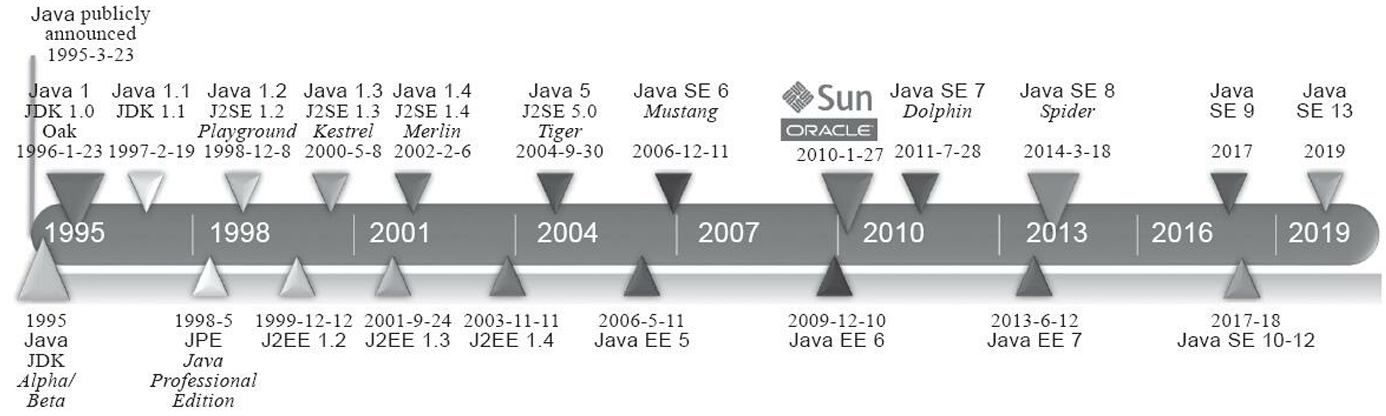
Java运行环境（Java Runtime Environment，简称JRE）是一个软件， JRE可以让计算机系统运行Java应用程序（Java Application）。JRE的内部有一个Java虚拟机（Java Virtual Machine，JVM）以及一些标准类别函数库（Class Library）。



* Java ME（Micro Edition）：支持Java程序运行在移动终端（手机、PDA）上的平台，对Java API 有所精简，并加入了移动终端的针对性支持，这条产品线在JDK 6以前被称为J2ME。有一点请勿混淆，现在在智能手机上非常流行的、主要使用Java语言开发程序的Android并不属于Java ME。
* Java SE（Standard Edition）：支持面向桌面级应用（如Windows下的应用程序）的Java平台，提供了完整的Java核心API，这条产品线在JDK 6以前被称为J2SE。
* Java EE（Enterprise Edition）：支持使用多层架构的企业应用（如ERP、MIS、CRM应用）的 Java平台，除了提供Java SE API外，还对其做了大量有针对性的扩充，并提供了相关的部署支持， 这条产品线在JDK 6以前被称为J2EE，在JDK 10以后被Oracle放弃，捐献给Eclipse基金会管理，此后被称为Jakarta EE。

# Java发展史

1991年4月，由James Gosling博士领导的绿色计划（Green Project）开始启动，此计划最初的目标是开发一种能够在各种消费性电子产品（如机顶盒、冰箱、收音机等）上运行的程序架构。这个计划的产品就是Java语言的前身：Oak（得名于James Gosling办公室外的一棵橡树）。Oak当时在消费品市场上并不算成功，但随着1995年互联网潮流的兴起，Oak迅速找到了最适合自己发展的市场定位并蜕变成为Java语言。



1995年5月23日，Oak语言改名为Java，并且在SunWorld大会上正式发布Java 1.0版本。Java语言第一次提出了“Write Once，Run Anywhere”的口号。

1996年1月23日，JDK 1.0发布，Java语言有了第一个正式版本的运行环境。JDK 1.0提供了一个纯解释执行的Java虚拟机实现（Sun Classic VM）。JDK 1.0版本的代表技术包括：Java虚拟机、Applet、AWT等。

1996年4月，十个最主要的操作系统和计算机供应商声明将在其产品中嵌入Java技术。同年9月，已有大约8.3万个网页应用了Java技术来制作。在1996年5月底，Sun于美国旧金山举行了首届JavaOne大会，从此JavaOne成为全世界数百万Java语言开发者每年一度的技术盛会。

1997年2月19日，Sun公司发布了JDK 1.1，Java里许多最基础的技术支撑点（如JDBC等）都是在JDK 1.1版本中提出的，JDK 1.1版的技术代表有：JAR文件格式、JDBC、JavaBeans、RMI等。Java语言的语法也有了一定的增强，如内部类（Inner Class）和反射（Reflection）都是在这时候出现的。

直到1999年4月8日，JDK 1.1一共发布了1.1.0至1.1.8这9个版本。从1.1.4以后，每个JDK版本都有一个属于自己的名字（工程代号），分别为：JDK 1.1.4-Sparkler（宝石）、JDK 1.1.5-Pumpkin（南瓜）、JDK 1.1.6-Abigail（阿比盖尔，女子名）、JDK 1.1.7-Brutus（布鲁图，古罗马政治家和将军）和JDK 1.1.8-Chelsea（切尔西，城市名）。

1998年12月4日，JDK迎来了一个里程碑式的重要版本：工程代号为Playground（竞技场）的JDK1.2，Sun在这个版本中把Java技术体系拆分为三个方向，分别是面向桌面应用开发的J2SE（Java 2 Platform，Standard Edition）、面向企业级开发的J2EE（Java 2 Platform，Enterprise Edition）和面向手机等移动终端开发的J2ME（Java 2 Platform，Micro Edition）。在这个版本中出现的代表性技术非常多，如EJB、Java Plug-in、Java IDL、Swing等，并且这个版本中Java虚拟机第一次内置了JIT（Just In Time）即时编译器（JDK 1.2中曾并存过三个虚拟机，Classic VM、HotSpot VM和Exact VM，其中Exact VM只在Solaris平台出现过；后面两款虚拟机都是内置了JIT即时编译器的，而之前版本所带的Classic VM只能以外挂的形式使用即时编译器）。在语言和API层面上，Java添加了strictfp关键字，Java类库添加了现在Java编码之中极为常用的一系列Collections集合类等。在1999年3月和7月，分别有JDK 1.2.1和JDK 1.2.2两个小升级版本发布。

1999年4月27日，HotSpot虚拟机诞生。HotSpot最初由一家名为“Longview Techno-logies”的小公司开发，由于HotSpot的优异表现，这家公司在1997年被Sun公司收购。Hot-Spot虚拟机刚发布时是作为JDK 1.2的附加程序提供的，后来它成为JDK 1.3及之后所有JDK版本的默认Java虚拟机。

2000年5月8日，工程代号为Kestrel（美洲红隼）的JDK 1.3发布。相对于JDK 1.2，JDK 1.3的改进主要体现在Java类库上（如数学运算和新的Timer API等），JNDI服务从JDK 1.3开始被作为一项平台级服务提供（以前JNDI仅仅是一项扩展服务），使用CORBA IIOP来实现RMI的通信协议，等等。这个版本还对Java 2D做了很多改进，提供了大量新的Java 2D API，并且新添加了JavaSound类库。JDK1.3有1个修正版本JDK 1.3.1，工程代号为Ladybird（瓢虫），于2001年5月17日发布。

自从JDK 1.3开始，Sun公司维持着稳定的研发节奏：大约每隔两年发布一个JDK的主版本，以动物命名，期间发布的各个修正版本则以昆虫作为工程代号。

2002年2月13日，JDK 1.4发布，工程代号为Merlin（灰背隼）。JDK 1.4是标志着Java真正走向成熟的一个版本，Compaq、Fujitsu、SAS、Symbian、IBM等著名公司都有参与功能规划，甚至实现自己独立发行的JDK 1.4。哪怕是在近二十年后的今天，仍然有一些主流应用能直接运行在JDK 1.4之上，或者继续发布能运行在1.4上的版本。JDK 1.4同样带来了很多新的技术特性，如正则表达式、异常链、NIO、日志类、XML解析器和XSLT转换器，等等。JDK 1.4有两个后续修正版：2002年9月16日发布的工程代号为Grasshopper（蚱蜢）的JDK 1.4.1与2003年6月26日发布的工程代号为Mantis（螳螂）的JDK 1.4.2。

2004年9月30日，JDK 5发布，工程代号为Tiger（老虎）。Sun公司从这个版本开始放弃了谦逊的“JDK 1.x”的命名方式，将产品版本号修改成了“JDK x”。 从JDK 1.2以来，Java在语法层面上的变动一直很小，而JDK 5在Java语法易用性上做出了非常大的改进。如：自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环（foreach循环）等语法特性都是在JDK 5中加入的。在虚拟机和API层面上，这个版本改进了Java的内存模型（Java Memory Model，JMM）、提供了java.util.concurrent并发包等。另外，JDK 5是官方声明可以支持Windows 9x操作系统的最后一个JDK版本。

2006年12月11日，JDK 6发布，工程代号为Mustang（野马）。在这个版本中，Sun公司终结了从JDK 1.2开始已经有八年历史的J2EE、J2SE、J2ME的产品线命名方式，启用Java EE 6、Java SE 6、Java ME 6的新命名来代替。JDK 6的改进包括：提供初步的动态语言支持（通过内置Mozilla JavaScript Rhino引擎实现）、提供编译期注解处理器和微型HTTP服务器API，等等。同时，这个版本对Java虚拟机内部做了大量改进，包括锁与同步、垃圾收集、类加载等方面的实现都有相当多的改动。

在2006年11月13日的JavaOne大会上，Sun公司宣布计划要把Java开源，在随后的一年多时间内，它陆续地将JDK的各个部分在GPL v2（GNU General Public License v2）协议下公开了源码，并建立了OpenJDK组织对这些源码进行独立管理。除了极少量的产权代码（Encumbered Code，这部分代码所有权不属于Sun公司，Sun本身也无权进行开源处理）外，OpenJDK几乎拥有了当时SunJDK 7的全部代码，OpenJDK的质量主管曾经表示在JDK 7中，SunJDK和OpenJDK除了代码文件头的版权注释之外，代码几乎是完全一样的，所以OpenJDK 7与SunJDK 7本质上就是同一套代码库出来的产品。

JDK 6发布以后，由于代码复杂性的增加、Java开源、开发JavaFX、世界经济危机及Oracle对Sun的收购案等原因，Sun公司在发展Java以外的事情上耗费了太多精力和资源，JDK的更新没有能够继续维持两年发布一个主版本的研发速度，这导致了JDK 6的生命周期异常的长，一共发布了211个更新升级补丁，最后的版本为Java SE 6 Update 211，于2018年10月18日发布。

2009年2月19日，工程代号为Dolphin（海豚）的JDK 7完成了其第一个里程碑版本。按照JDK 7最初的功能规划，一共会设置十个里程碑。最后一个里程碑版本原计划定于2010年9月9日结束，但由于各种原因，JDK 7最终无法按计划完成。

从JDK 7最原始的功能清单来看，它本应是一个包含许多重要改进的JDK版本，其中规划的子项目都为Java业界翘首以盼，包括：

* Lambda项目：支持Lambda表达式，支持函数式编程。
* Jigsaw项目：虚拟机层面的模块化支持。
* 动态语言支持：Java是静态语言，为其他运行在Java虚拟机上的动态语言提供支持。
* Garbage-First收集器。
* Coin项目：Java语法细节进化。

令人惋惜的是，在JDK 7开发期间，Sun公司相继在技术竞争和商业竞争中陷入泥潭，公司的股票市值跌至仅有高峰时期的3%，已无力推动JDK 7的研发工作按计划继续进行。为了尽快结束JDK 7长期跳票的问题，Oracle收购Sun公司后随即宣布马上实行“B计划”，大幅裁剪了JDK 7预定目标，以保证JDK 7的正式版能够于2011年7月28日准时发布。“B计划”的主要措施是把不能按时完成的Lambda项目、Jigsaw项目和Coin项目的部分改进延迟到JDK 8之中。最终，JDK 7包含的改进有：提供新的G1收集器（G1在发布时依然处于Experimental状态，直至2012年4月的Update 4中才正式商用）、加强对非Java语言的调用支持（JSR-292，这项特性在到JDK 11还有改动）、可并行的类加载架构等。

Oracle公司接手了JDK开发工作以后，迅速展现出了完全不同于Sun时期的、极具商业化的处事风格。面对Java中使用最广泛而又一直免费的Java SE产品线，Oracle很快定义了一套新的Java SE Support产品计划，把JDK的更新支持作为一项商业服务。JDK 7发布的前80个更新仍然免费面向所有用户提供，但后续的其他更新包，用户只能从“将Java SE升级到Java SE Support”与“将JDK 7升级到最新版本”两个选项里挑一个。JDK 7计划维护至2022年，迄今（面向付费用户）已发布了超过两百个更新补丁，最新版本为JDK 7 Update 221。

对于JDK 7，还有一点值得提起的是，从JDK 7 Update 4起，Java SE的核心功能正式开始为MacOS X操作系统提供支持，并在JDK 7 Update 6中达到所有功能与Mac OS X完全兼容的程度；同时，JDK 7 Update 6还对ARM指令集架构提供了支持。至此，官方提供的JDK可以运行于Windows（不含Windows 9x）、Linux、Solaris和Mac OS X操作系统上，支持ARM、x86、x86-64和SPARC指令集架构，JDK 7也是可以支持Windows XP操作系统的最后一个版本。

2009年4月20日，Oracle宣布正式以74亿美元的价格收购市值曾超过2000亿美元的Sun公司，传奇的Sun Microsystems从此落幕成为历史，Java商标正式划归Oracle所有（Java语言本身并不属于哪间公司所有，它由JCP组织进行管理，尽管在JCP中Sun及后来的Oracle的话语权很大）。由于此前Oracle已经收购了另外一家大型的中间件企业BEA公司，当完成对Sun公司的收购之后，Oracle分别从BEA和Sun手中取得了世界三大商用虚拟机的其中两个：JRockit和HotSpot。当时Oracle宣布要在未来一至两年的时间内，把这两个优秀的Java虚拟机合二为一。两者合并的结果只能说差强人意，JRockit的监控工具Java Mission Control被移植到了HotSpot，作为收费功能提供给购买了Java SE Advanced产品计划的用户，其他功能由于两者架构的差异性明显，HotSpot能够直接借鉴融合的功能寥寥无几。

JDK 8的第一个正式版本原定于2013年9月发布，最终还是跳票到了2014年3月18日，尽管仍然是没有赶上正点，但比起JDK 7那种以年作为计时单位、直接把公司跳崩的研发状况已是大有改善。为了保证日后JDK研发能更顺利地进行，从JDK 8开始，Oracle启用JEP（JDK Enhancement Proposals）来定义和管理纳入新版JDK发布范围的功能特性。JDK 8提供了那些曾在JDK 7中规划过，但最终未能在JDK 7中完成的功能，主要包括：

* JEP 126：对Lambda表达式的支持，这让Java语言拥有了流畅的函数式表达能力。
* JEP 104：内置Nashorn JavaScript引擎的支持。
* JEP 150：新的时间、日期API。
* JEP 122：彻底移除HotSpot的永久代。
* ……

“B计划”中原本说好的会在JDK 8提供的Jigsaw模块化功能再次被延期到了JDK 9，不得不说，即使放到整个Java发展史里看，Jigsaw都能算是天字第一号的大坑。Java的模块化系统本身面临的技术挑战就很艰巨，从微软的DLL技术开始，到Java自己的JAR，再到.NET的Assembly，工程庞大起来都无一例外会陷入“模块地狱”的困境之中，而Jigsaw面临的更大困难是厂商之间以标准话语权为目的，以技术为“找茬”手段的激烈竞争。

原本JDK 9是计划在2016年发布的，但在2016年伊始，Oracle就宣布JDK 9肯定要延期至2017年，后来又连续经过了两次短时间的跳票，最终到2017年9月21日才得以艰难面世。后两次跳票的原因是以IBM和RedHat为首的十三家企业在JCP执行委员会上联手否决了Oracle提出的Jigsaw作为Java模块化规范进入JDK 9发布范围的提案。凭良心说，Java确实有模块化的刚需，不论是JDK自身（例如拆分出Java SE Embedded这样规模较小的产品）抑或是Java应用都需要用到模块化。这方面IBM本身就是各大Java发行厂商中做得最好的，它不仅让自家的JDK实现了高度模块化，还带头成立了OSGi联盟，制订了Java框架层面模块化的事实标准，所以它当然会想把OSGi推到Java规范里去争个“名份”，而不是被Jigsaw革掉“性命”。可是Oracle对此没有丝毫退让，不惜向JCP发去公开信，直言如果提案最后无法通过，那Oracle将摒弃JSR专家组，独立发展带Jigsaw的Java版本，Java顿时面临如Python 2与Python 3那般分裂的危机。

不论如何，经过前后六轮投票，经历桌上桌下的斗争与妥协，Java没有分裂，JDK 9总算是带着Jigsaw最终发布了，除了Jigsaw外，JDK 9还增强了若干工具（JS Shell、JLink、JHSDB等），整顿了HotSpot各个模块各自为战的日志系统，支持HTTP 2客户单API等91个JEP。

JDK 9发布后，Oracle随即宣布Java将会以持续交付的形式和更加敏捷的研发节奏向前推进，以后JDK将会在每年的3月和9月各发布一个大版本，目的就是为避免众多功能特性被集中捆绑到一个JDK版本上而引发交付风险。这次改革确实从根源上解决了跳票问题，但也为Java的用户和发行商带来了颇大的压力，不仅程序员感慨“Java新版本还没开始用就已经过时了”，Oracle自己对着一堆JDK版本分支也在挠头，不知道该如何维护更新，该如何提供技术支持。Oracle的解决方案是顺理成章地终结掉“每个JDK版本最少维护三年”的优良传统，从此以后，每六个JDK大版本中才会被划出一个长期支持（Long Term Support，LTS）版，只有LTS版的JDK能够获得为期三年的支持和更新，普通版的JDK就只有短短六个月的生命周期。JDK 8和JDK 11会是LTS版，再下一个就到2021年发布的JDK 17了。

2018年3月20日，JDK 10如期发布，这版本的主要研发目标是内部重构，诸如统一源仓库、统一垃圾收集器接口、统一即时编译器接口（JVMCI在JDK 9已经有了，这里是引入新的Graal即时编译器）等，这些都将会是对未来Java发展大有裨益的改进，但对普通用户来说JDK 10的新特性就显得乏善可陈，毕竟它只包含了12个JEP，而且其中只有本地类型推断这一个编码端可见的改进。尽管JDK10可见的改进有限，但2018这一年Java圈丝毫不缺乏谈资，相继发生了几件与“金钱”相关的历史性大事件。

首先是2018年3月27日，Android的Java侵权案有了最终判决，法庭裁定Google赔偿Oracle合计88亿美元，要知道2009年Oracle收购Sun也就只花了74亿，收购完成后随即就用Sun的专利把Google告上了法庭，经过Oracle法务部的几轮神操作，一场官司的赔偿让收购Sun公司等同免费。对此事Java技术圈多数吃瓜群众是站在Google这边的，认为Oracle这样做是自绝Java的发展前景，毕竟当年Android刚刚起步的时候可是Sun向Google抛去的橄榄枝，Android的流行也巩固了Java“第一编程语言”的行业地位。摒弃对企业的好恶情感，就事论事，Google采用Java的语法和API类库，开发出来的程序却不能运行在其他Java虚拟机之上，这事情无论怎样都是有违Java技术的精神原旨的，也肯定违反了Java的使用协议。如果说Oracle控告Google“不厚道”，那当年微软用J++做了同样的事情（借用语法和API，但程序不兼容标准Java虚拟机），被Sun告到登报道歉，一边赔款一边割地，声明放弃J++语言和Windows平台上的内置虚拟机，这又该找谁说理去？

按常理说Java刚给Oracle赚了88亿美金，该颇为受宠才对，可Oracle是典型只谈利益不讲情怀的公司，InfoWorld披露的一封Oracle高管邮件表明，Java体系中被认为无法盈利也没有太多战略前景的部分会逐渐被“按计划报废”（Planned Obsolescence）。这事的第一刀落下是在2018年3月，Oracle正式宣告Java EE成为历史名词。虽然Java SE、Java EE和Java ME三条产品线里确实只有Java SE称得上成功，但Java EE毕竟无比辉煌过，现在其中还持有着JDBC、JMS、Servlet等使用极为广泛的基础组件，然而Oracle仍选择把它“扫地出门”，所有权直接赠送给Eclipse基金会，唯一的条件是以后不准再使用“Java”这个商标，所以取而代之的将是Jakarta EE。

2018年10月，JavaOne 2018在旧金山举行，此前没有人想过这会是最后一届JavaOne大会，这个在1996年伴随着Java一同诞生、成长的开发者年度盛会，竟是Oracle下一个裁撤的对象，此外还有Java Mission Control的开发团队，也在2018年6月被Oracle解散。

2018年9月25日，JDK 11发布，这是一个LTS版本的JDK，包含17个JEP，其中有ZGC这样的革命性的垃圾收集器出现，也有把JDK 10中的类型推断加入Lambda语法这种可见的改进，但都比不过它发布时爆出来的谣言轰动：“Java要开始收费啦！”。

随着JDK 11发布，Oracle同时调整了JDK的授权许可证，里面包含了好几个动作。首先，Oracle从JDK 11起把以前的商业特性全部开源给OpenJDK，这样OpenJDK 11和OracleJDK 11的代码和功能，在本质上就是完全相同的（官方原文是Essentially Identical）。然后，Oracle宣布以后将会同时发行两个JDK：一个是以GPLv2+CE协议下由Oracle发行的OpenJDK（也可以称其为Oracle OpenJDK），另一个是在新的OTN协议下发行的传统的OracleJDK，这两个JDK共享绝大部分源码，在功能上是几乎一样的，核心差异是前者可以免费在开发、测试或生产环境中使用，但是只有半年时间的更新支持；后者个人依然可以免费使用，但若在生产环境中商用就必须付费，可以有三年时间的更新支持。如果说由此能得出“Java要收费”的结论，那是纯属标题党，最多只能说Oracle在迫使商业用户要么不断升级JDK的版本，要么就去购买商业支持。这里的商业支持不限定于Oracle公司，如Azul ZingJDK、AdoptOpenJDK等都能提供商业支持。

2019年2月，在JDK 12发布前夕，Oracle果然如之前宣布那样在六个月之后就放弃了对上一个版本OpenJDK的维护，RedHat同时从Oracle手上接过OpenJDK 8和OpenJDK 11的管理权利和维护职责（Red Hat此前已经是OpenJDK 6（自2013年起）和OpenJDK 7（自2015年起）的维护者）。Oracle不愿意在旧版本上继续耗费资源，而RedHat或者说它背后的IBM又乐意扩大自己在Java社区的影响力，这是一笔双赢的交易。RedHat代替Oracle成为JDK历史版本的维护者，应该有利于Java的持续稳定，但从技术发展角度来看，这并不能为Oracle领导Java社区的局面带来根本性的改变，毕竟要添加新的或实验性的功能，仅会针对Java的最新版本，而不会在旧版本上动手。

2019年3月20日，JDK 12发布，只包含8个JEP，其中主要有Switch表达式、Java微测试套件（JMH）等新功能，最引人注目的特性无疑是加入了由RedHat领导开发的Shen-andoah垃圾收集器。Shenandoah作为首个由非Oracle开发的垃圾收集器，其目标又与Oracle在JDK 11中发布的ZGC几乎完全一致，两者天生就存在竞争。Oracle马上用实际行动抵制了这个新收集器，在JDK 11发布时才说应尽可能保证OracleJDK和OpenJDK的兼容一致，转眼就在OracleJDK 12里把Shenandoah的代码通过条件编译强行剔除掉，使其成为历史上唯一进入了OpenJDK发布清单，但在OracleJDK中无法使用的功能。

Oracle收购Sun是Java发展历史上一道明显的分界线。在Sun掌舵的前十几年里，Java获得巨大成功，同时也渐渐显露出来语言演进的缓慢与社区决策的老朽；而在Oracle主导Java后，引起竞争的同时也带来新的活力，Java发展的速度要显著高于Sun时代。Java的未来是继续向前、再攀高峰，还是由盛转衰、锋芒挫缩，你我拭目以待。

Java面临的危机挑战前所未有的艰巨，属于Java的未来也从未如此充满想象与可能。

# Java虚拟机家族

## 虚拟机始祖：Sun Classic/Exact VM

1996年1月23日，Sun发布JDK 1.0，Java语言首次拥有了商用的正式运行环境，这个JDK中所带的虚拟机就是Classic VM。这款虚拟机只能使用纯解释器方式来执行Java代码，如果要使用即时编译器那就必须进行外挂，但是假如外挂了即时编译器的话，即时编译器就会完全接管虚拟机的执行系统，解释器便不能再工作了。

由于解释器和编译器不能配合工作，这就意味着如果要使用编译执行，编译器就不得不对每一个方法、每一行代码都进行编译，而无论它们执行的频率是否具有编译的价值。基于程序响应时间的压力，这些编译器根本不敢应用编译耗时稍高的优化技术，因此这个阶段的虚拟机虽然用了即时编译器输出本地代码，其执行效率也和传统的C/C++程序有很大差距。

Sun的虚拟机团队努力去解决Classic虚拟机所面临的各种问题，提升运行效率，在JDK 1.2时，曾在Solaris平台上发布过一款名为Exact VM的虚拟机，它的编译执行系统已经具备现代高性能虚拟机雏形，如热点探测、两级即时编译器、编译器与解释器混合工作模式等。

Exact VM因它使用准确式内存管理（Exact Memory Management，也可以叫Non-Con-servative/Accurate Memory Management）而得名。准确式内存管理是指虚拟机可以知道内存中某个位置的数据具体是什么类型。譬如内存中有一个32bit的整数123456，虚拟机将有能力分辨出它到底是一个指向了123456的内存地址的引用类型还是一个数值为123456的整数，准确分辨出哪些内存是引用类型，这也是在垃圾收集时准确判断堆上的数据是否还可能被使用的前提。由于使用了准确式内存管理，Exact VM可以抛弃掉以前Classic VM基于句柄（Handle）的对象查找方式（原因是垃圾收集后对象将可能会被移动位置，如果地址为123456的对象移动到654321，在没有明确信息表明内存中哪些数据是引用类型的前提下，那虚拟机肯定是不敢把内存中所有为123456的值改成654321的，所以要使用句柄来保持引用值的稳定），这样每次定位对象都少了一次间接查找的开销，显著提升执行性能。

虽然Exact VM的技术相对Classic VM来说先进了许多，但是它的命运显得十分英雄气短，在商业应用上只存在了很短暂的时间就被外部引进的HotSpot VM所取代，甚至还没有来得及发布Windows和Linux平台下的商用版本。而Classic VM的生命周期则相对要长不少，它在JDK 1.2之前是JDK中唯一的虚拟机，在JDK 1.2时，它与HotSpot VM并存，但默认是使用Classic VM（用户可用java-hotspot参数切换至HotSpot VM），而在JDK 1.3时，HotSpot VM成为默认虚拟机，它仍作为虚拟机的“备用选择”发布（使用java-classic参数切换），直到JDK 1.4的时候，Classic VM才完全退出商用虚拟机的历史舞台，与Exact VM一起进入了Sun Labs Research VM之中。

## 武林盟主：HotSpot VM

相信所有Java程序员都听说过HotSpot虚拟机，它是Sun/OracleJDK和OpenJDK中的默认Java虚拟机，也是目前使用范围最广的Java虚拟机。但不一定所有人都知道的是，这个在今天看起来“血统纯正”的虚拟机在最初并非由Sun公司所开发，而是由一家名为“Longview Technologies”的小公司设计；甚至这个虚拟机最初并非是为Java语言而研发的，它来源于Strongtalk虚拟机，而这款虚拟机中相当多的技术又是来源于一款为支持Self语言实现“达到C语言50%以上的执行效率”的目标而设计的Self虚拟机，最终甚至可以追溯到20世纪80年代中期开发的Berkeley Smalltalk上。Sun公司注意到这款虚拟机在即时编译等多个方面有着优秀的理念和实际成果，在1997年收购了Longview Technologies公司，从而获得了HotSpot虚拟机。

HotSpot既继承了Sun之前两款商用虚拟机的优点（如前面提到的准确式内存管理），也有许多自己新的技术优势，如它名称中的HotSpot指的就是它的热点代码探测技术（这里的描写带有“历史由胜利者书写”的味道，其实HotSpot与Exact虚拟机基本上是同时期的独立产品，HotSpot出现得还稍早一些，一开始HotSpot就是基于准确式内存管理的，而Exact VM之中也有与HotSpot几乎一样的热点探测技术，为了Exact VM和HotSpot VM哪个该成为Sun主要支持的虚拟机，在Sun公司内部还争吵过一场，HotSpot击败Exact并不能算技术上的胜利），HotSpot虚拟机的热点代码探测能力可以通过执行计数器找出最具有编译价值的代码，然后通知即时编译器以方法为单位进行编译。如果一个方法被频繁调用，或方法中有效循环次数很多，将会分别触发标准即时编译和栈上替换编译（On-Stack Replacement，OSR）行为。通过编译器与解释器恰当地协同工作，可以在最优化的程序响应时间与最佳执行性能中取得平衡，而且无须等待本地代码输出才能执行程序，即时编译的时间压力也相对减小，这样有助于引入更复杂的代码优化技术，输出质量更高的本地代码。

2006年，Sun陆续将SunJDK的各个部分在GPLv2协议下开放了源码，形成了Open-JDK项目，其中当然也包括HotSpot虚拟机。HotSpot从此成为Sun/OracleJDK和OpenJDK两个实现极度接近的JDK项目的共同虚拟机。Oracle收购Sun以后，建立了HotRockit项目来把原来BEA JRockit中的优秀特性融合到HotSpot之中。到了2014年的JDK 8时期，里面的HotSpot就已是两者融合的结果，HotSpot在这个过程里移除掉永久代，吸收了JRockit的Java Mission Control监控工具等功能。

得益于Sun/OracleJDK在Java应用中的统治地位，HotSpot理所当然地成为全世界使用最广泛的Java虚拟机，是虚拟机家族中毫无争议的“武林盟主”。

## 小家碧玉：Mobile/Embedded VM

Sun/Oracle公司所研发的虚拟机可不仅包含前面介绍到的服务器、桌面领域的商用虚拟机，面对移动和嵌入式市场，也有专门的Java虚拟机产品。

由于Java ME产品线的发展相对Java SE来说并不那么成功，所以Java ME中的Java虚拟机相比HotSpot要低调得多。Oracle公司在Java ME这条产品线上的虚拟机名为CDC-HI（C Virtual Machine，CVM）和CLDC-HI（Monty VM）。其中CDC/CLDC全称是Connected（Limited）Device Configuration，这是一组在JSR-139及JSR-218规范中进行定义的Java API子集，这组规范希望能够在手机、电子书、PDA等移动设备上建立统一的Java编程接口，CDC-HI VM和CLDC-HI VM就是JSR-139及JSR-218规范的参考实现，后面的HI则是HotSpot Implementation的缩写，但它们并不是由HotSpot直接裁剪而来，只是借鉴过其中一些技术，并没有血缘关系，充其量能叫有所渊源。

Java ME中的Java虚拟机现在处于比较尴尬的位置，所面临的局面远不如服务器和桌面领域乐观，它最大的一块市场——智能手机已被Android和iOS二分天下，现在CDC在智能手机上略微有点声音的产品是Oracle ADF Mobile，原本它提出的卖点是智能手机上的跨平台（“Developing with Java on iOS and Android”），不过用Java在Android上开发应用还要再安装个CDC虚拟机，这事情听着就觉得别扭，有多此一举的嫌疑，在iOS上倒确实还有一些人在用。

而在嵌入式设备上，Java ME Embedded又面临着自家Java SE Embedded（eJDK）的直接竞争和侵蚀，主打高端的CDC-HI经过多年来的扩充，在核心部分其实已经跟Java SE非常接近，能用Java SE的地方大家自然就不愿意用Java ME，所以市场在快速萎缩，Oracle也基本上砍掉了CDC-HI的所有项目，把它们都划归到了Java SE Embedded下。Java SE Embedded里带的Java虚拟机当然还是HotSpot，但这是为了适应嵌入式环境专门定制裁剪的版本，尽可能在支持完整的Java SE功能的前提下向着减少内存消耗的方向优化，譬如只留下了客户端编译器（C1），去掉了服务端编译器（C2）；只保留Serial/Serial Old垃圾收集器，去掉了其他收集器等。

面向更低端设备的CLDC-HI倒是在智能控制器、传感器等领域还算能维持自己的一片市场，现在也还在继续发展，但前途并不乐观。目前CLDC中活得最好的产品反而是原本早该被CLDC-HI淘汰的KVM，国内的老人手机和出口到经济欠发达国家的功能手机（Feature Phone）还在广泛使用这种更加简单、资源消耗也更小的上一代Java ME虚拟机。

## 天下第二：BEA JRockit/IBM J9 VM

BEA System公司的JRockit与IBM公司的IBM J9。

JRockit虚拟机曾经号称是“世界上速度最快的Java虚拟机”（广告词，IBM J9虚拟机也这样宣传过，总体上三大虚拟机的性能是交替上升的），它是BEA在2002年从Appeal Virtual Machines公司收购获得的Java虚拟机。BEA将其发展为一款专门为服务器硬件和服务端应用场景高度优化的虚拟机，由于专注于服务端应用，它可以不太关注于程序启动速度，因此JRockit内部不包含解释器实现，全部代码都靠即时编译器编译后执行。除此之外，JRockit的垃圾收集器和Java Mission Control故障处理套件等部分的实现，在当时众多的Java虚拟机中也处于领先水平。JRockit随着BEA被Oracle收购，现已不再继续发展，永远停留在R28版本，这是JDK 6版JRockit的代号。

IBM J9虚拟机并不是IBM公司唯一的Java虚拟机，不过目前IBM主力发展无疑就是J9。J9这个名字最初只是内部开发代号而已，开始选定的正式名称是“IBM Technology for Java Virtual Machine”，简称IT4J，但这个名字太拗口，接受程度远不如J9。J9虚拟机最初是由IBM Ottawa实验室的一个SmallTalk虚拟机项目扩展而来，当时这个虚拟机有一个Bug是因为8KB常量值定义错误引起，工程师们花了很长时间终于发现并解决了这个错误，此后这个版本的虚拟机就被称为K8，后来由其扩展而来、支持Java语言的虚拟机就被命名为J9。与BEA JRockit只专注于服务端应用不同，IBM J9虚拟机的市场定位与HotSpot比较接近，它是一款在设计上全面考虑服务端、桌面应用，再到嵌入式的多用途虚拟机，开发J9的目的是作为IBM公司各种Java产品的执行平台，在和IBM产品（如IBM WebSphere等）搭配以及在IBM AIX和z/OS这些平台上部署Java应用。

IBM J9直至今天仍旧非常活跃，IBM J9虚拟机的职责分离与模块化做得比HotSpot更优秀，由J9虚拟机中抽象封装出来的核心组件库（包括垃圾收集器、即时编译器、诊断监控子系统等）就单独构成了IBM OMR项目，可以在其他语言平台如Ruby、Python中快速组装成相应的功能。从2016年起，IBM逐步将OMR项目和J9虚拟机进行开源，完全开源后便将它们捐献给了Eclipse基金会管理，并重新命名为Eclipse OMR和OpenJ9。如果为了学习虚拟机技术而去阅读源码，更加模块化的OpenJ9代码其实是比HotSpot更好的选择。如果为了使用Java虚拟机时多一种选择，那可以通过AdoptOpenJDK来获得采用OpenJ9搭配上OpenJDK其他类库组成的完整JDK。

除BEA和IBM公司外，其他一些大公司也号称有自己的专属JDK和虚拟机，但是它们要么是通过从Sun/Oracle公司购买版权的方式获得的（如HP、SAP等），要么是基于OpenJDK项目改进而来的（如阿里巴巴、Twitter等），都并非自己独立开发。

## 软硬合璧：BEA Liquid VM/Azul VM

平时所提及的“高性能Java虚拟机”一般是指HotSpot、JRockit、J9这类在通用硬件平台上运行的商用虚拟机，但其实还有一类与特定硬件平台绑定、软硬件配合工作的专有虚拟机，往往能够实现更高的执行性能，或提供某些特殊的功能特性。这类专有虚拟机的代表是BEA Liquid VM和Azul VM。

Liquid VM也被称为JRockit VE（Virtual Edition，VE），它是BEA公司开发的可以直接运行在自家Hypervisor系统上的JRockit虚拟机的虚拟化版本，Liquid VM不需要操作系统的支持，或者说它自己本身实现了一个专用操作系统的必要功能，如线程调度、文件系统、网络支持等。由虚拟机越过通用操作系统直接控制硬件可以获得很多好处，如在线程调度时，不需要再进行内核态/用户态的切换，这样可以最大限度地发挥硬件的能力，提升Java程序的执行性能。随着JRockit虚拟机终止开发，Liquid VM项目也已经停止了。

Azul VM是Azul Systems公司在HotSpot基础上进行大量改进，运行于Azul Systems公司的专有硬件Vega系统上的Java虚拟机，每个Azul VM实例都可以管理至少数十个CPU和数百GB的内存的硬件资源，并提供在巨大内存范围内停顿时间可控的垃圾收集器（即业内赫赫有名的PGC和C4收集器），为专有硬件优化的线程调度等优秀特性。2010年起，Azul公司的重心逐渐开始从硬件转向软件，发布了自己的Zing虚拟机，可以在通用x86平台上提供接近于Vega系统的性能和一致的功能特性。

随着虚拟机技术的不断发展，Java虚拟机变得越来越强大的同时也越来越复杂，要推动在专有硬件上的Java虚拟机升级发展，难以直接借助开源社区的力量，往往需要耗费更高昂的成本，在商业上的缺陷使得专有虚拟机逐渐没落，Azul Systems公司最终也放弃了Vega产品线，把全部精力投入到Zing和Zulu产品线中。

Zing虚拟机是一个从HotSpot某旧版代码分支基础上独立出来重新开发的高性能Java虚拟机，它可以运行在通用的Linux/x86-64平台上。Azul公司为它编写了新的垃圾收集器，也修改了HotSpot内的许多实现细节，在要求低延迟、快速预热等场景中，Zing VM都要比HotSpot表现得更好。Zing的PGC、C4收集器可以轻易支持TB级别的Java堆内存，而且保证暂停时间仍然可以维持在不超过10毫秒的范围里，HotSpot要一直到JDK 11和JDK 12的ZGC及Shenandoah收集器才达到了相同的目标，而且目前效果仍然远不如C4。Zing的ReadyNow！功能可以利用之前运行时收集到的性能监控数据，引导虚拟机在启动后快速达到稳定的高性能水平，减少启动后从解释执行到即时编译的等待时间。Zing自带的ZVision/ZVRobot功能可以方便用户监控Java虚拟机的运行状态，从找出代码热点到对象分配监控、锁竞争监控等。Zing能让普通用户无须了解垃圾收集等底层调优，就可以使得Java应用享有低延迟、快速预热、易于监控的功能，这是Zing的核心价值和卖点，很多Java应用都可以通过长期努力在应用、框架层面优化来提升性能，但使用Zing的话就可以把精力更多集中在业务方面。

## 挑战者：Apache Harmony/Google Android Dalvik VM

Harmony虚拟机（准确地说是Harmony里的DRLVM）和Dalvik虚拟机只能称作“虚拟机”，而不能称作“Java虚拟机”，但是这两款虚拟机以及背后所代表的技术体系曾经对Java世界产生了非常大的影响和挑战，当时甚至有悲观的人认为成熟的Java生态系统都有分裂和崩溃的可能。

Apache Harmony是一个Apache软件基金会旗下以Apache License协议开源的实际兼容于JDK 5和JDK 6的Java程序运行平台，它含有自己的虚拟机和Java类库API，用户可以在上面运行Eclipse、Tomcat、Maven等常用的Java程序。但是，它并没有通过TCK认证，所以我们不得不用一长串冗长拗口的语言来介绍它，而不能用一句“Apache的JDK”或者“Apache的Java虚拟机”来直接代指。

如果一个公司要宣称自己的运行平台“兼容于Java技术体系”，那该运行平台就必须要通过TCK（Technology Compatibility Kit）的兼容性测试，Apache基金会曾要求当时的Sun公司提供TCK的使用授权，但是一直遭到各种理由的拖延和搪塞，直到Oracle收购了Sun公司之后，双方关系越闹越僵，最终导致Apache基金会愤然退出JCP组织，这是Java社区有史以来最严重的分裂事件之一。

当Sun公司把自家的JDK开源形成OpenJDK项目之后，Apache Harmony开源的优势被极大地抵消，以至于连Harmony项目的最大参与者IBM公司也宣布辞去Harmony项目管理主席的职位，转而参与OpenJDK的开发。虽然Harmony没有真正地被大规模商业运用过，但是它的许多代码（主要是Java类库部分的代码）被吸纳进IBM的JDK 7实现以及Google Android SDK之中，尤其是对Android的发展起了很大推动作用。

Dalvik虚拟机曾经是Android平台的核心组成部分之一，它的名字来源于冰岛一个名为Dalvik的小渔村。Dalvik虚拟机并不是一个Java虚拟机，它没有遵循《Java虚拟机规范》，不能直接执行Java的Class文件，使用寄存器架构而不是Java虚拟机中常见的栈架构。但是它与Java却又有着千丝万缕的联系，它执行的DEX（Dalvik Executable）文件可以通过Class文件转化而来，使用Java语法编写应用程序，可以直接使用绝大部分的Java API等。在Android发展的早期，Dalvik虚拟机随着Android的成功迅速流行，在Android 2.2中开始提供即时编译器实现，执行性能又有了进一步提高。不过到了Android 4.4时代，支持提前编译（Ahead of Time Compilation，AOT）的ART虚拟机迅速崛起，在当时性能还不算特别强大的移动设备上，提前编译要比即时编译更容易获得高性能，所以在Android 5.0里ART就全面代替了Dalvik虚拟机。

## 没有成功，但并非失败：Microsoft JVM及其他

在Java语言诞生的初期（1996年～1998年，以JDK1.2发布之前为分界），它的主要应用之一是在浏览器中运行Java Applets程序，微软为了在Internet Explorer 3浏览器中支持Java Applets应用而开发了自己的Java虚拟机，虽然这款虚拟机只有Windows平台的版本，“一次编译，到处运行”根本无从谈起，但却是当时Windows系统下性能最好的Java虚拟机，它在1997年和1998年连续获得了《PC Magazine》杂志的“编辑选择奖”。但是好景不长，在1997年10月，Sun公司正式以侵犯商标、不正当竞争等罪名控告微软，在随后对微软公司的垄断调查之中，这款虚拟机也曾作为证据之一被呈送法庭。官司的结果是微软向Sun公司（最终微软因垄断赔偿给Sun公司的总金额高达10亿美元）赔偿2000万美金，承诺终止其Java虚拟机的发展，并逐步在产品中移除Java虚拟机相关功能。而最令人感到讽刺的是，到后来在Windows XP SP3中Java虚拟机被完全抹去的时候，Sun公司却又到处登报希望微软不要这样做。Windows XP高级产品经理Jim Cullinan称：“我们花费了三年的时间和Sun公司打官司，当时他们试图阻止我们在Windows中支持Java，现在我们这样做了，可他们又在抱怨，这太具有讽刺意味了。”

# 展望Java技术的未来

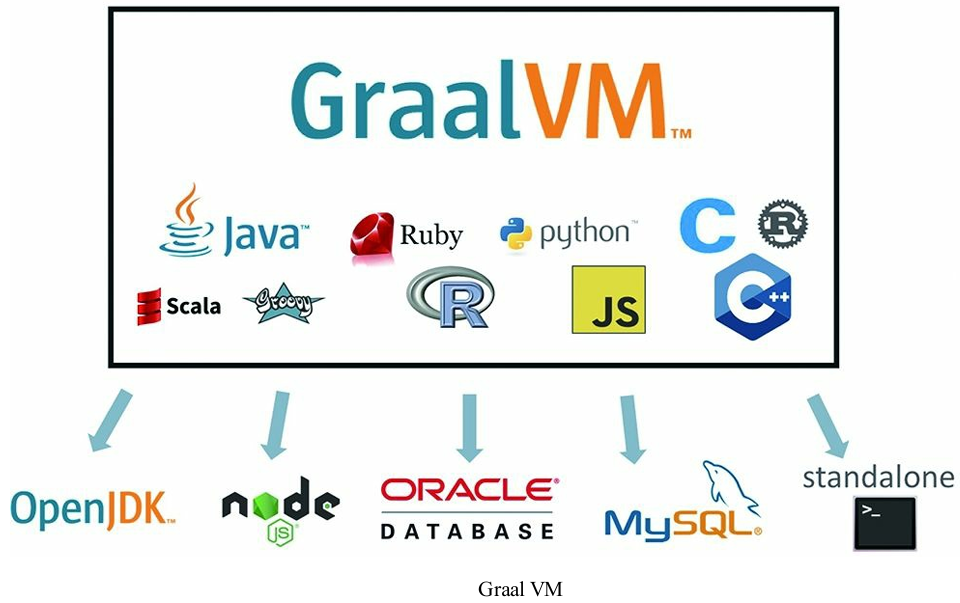
## 无语言倾向

网上每隔一段时间就能见到几条“未来X语言将会取代Java”的新闻，此处“X”可以用Kotlin、Golang、Dart、JavaScript、Python等各种编程语言来代入。这大概就是长期占据编程语言榜单第一位的烦恼，天下第一总避免不了挑战者相伴。

如果Java有拟人化的思维，它应该从来没有惧怕过被哪一门语言所取代，Java“天下第一”的底气不在于语法多么先进好用，而是来自它庞大的用户群和极其成熟的软件生态，这在朝夕之间难以撼动。不过，既然有那么多新、旧编程语言的兴起躁动，说明必然有其需求动力所在，譬如互联网之于JavaScript、人工智能之于Python，微服务风潮之于Golang等。大家都清楚不太可能有哪门语言能在每一个领域都尽占优势，Java已是距离这个目标最接近的选项，但若“天下第一”还要百尺竿头更进一步的话，似乎就只能忘掉Java语言本身，踏入无招胜有招的境界。

2018年4月，Oracle Labs新公开了一项黑科技：Graal VM，从它的口号“Run Programs Faster Anywhere”就能感觉到一颗蓬勃的野心，这句话显然是与1995年Java刚诞生时的“Write Once，Run Anywhere”在遥相呼应。

Graal VM被官方称为“Universal VM”和“Polyglot VM”，这是一个在HotSpot虚拟机基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”的运行平台使用，这里“任何语言”包括了Java、Scala、Groovy、Kotlin等基于Java虚拟机之上的语言，还包括了C、C++、Rust等基于LLVM的语言，同时支持其他像JavaScript、Ruby、Python和R语言等。Graal VM可以无额外开销地混合使用这些编程语言，支持不同语言中混用对方的接口和对象，也能够支持这些语言使用已经编写好的本地库文件。



Graal VM的基本工作原理是将这些语言的源代码（例如JavaScript）或源代码编译后的中间格式（例如LLVM字节码）通过解释器转换为能被Graal VM接受的中间表示（Intermediate Representation，IR），譬如设计一个解释器专门对LLVM输出的字节码进行转换来支持C和C++语言，这个过程称为程序特化（Specialized，也常被称为Partial Evaluation）。Graal VM提供了Truffle工具集来快速构建面向一种新语言的解释器，并用它构建了一个称为Sulong的高性能LLVM字节码解释器。

从更严格的角度来看，Graal VM才是真正意义上与物理计算机相对应的高级语言虚拟机，理由是它与物理硬件的指令集一样，做到了只与机器特性相关而不与某种高级语言特性相关。Oracle Labs的研究总监Thomas Wuerthinger在接受InfoQ采访时谈到：“随着GraalVM 1.0的发布，我们已经证明了拥有高性能的多语言虚拟机是可能的，并且实现这个目标的最佳方式不是通过类似Java虚拟机和微软CLR那样带有语言特性的字节码。”对于一些本来就不以速度见长的语言运行环境，由于Graal VM本身能够对输入的中间表示进行自动优化，在运行时还能进行即时编译优化，因此使用Graal VM实现往往能够获得比原生编译器更优秀的执行效率，譬如Graal.js要优于Node.js，Graal.Python要优于CPtyhon，TruffleRuby要优于Ruby MRI，FastR要优于R语言等。

对Java而言，Graal VM本来就是在HotSpot基础上诞生的，天生就可作为一套完整的符合Java SE 8标准的Java虚拟机来使用。它和标准的HotSpot的差异主要在即时编译器上，其执行效率、编译质量目前与标准版的HotSpot相比也是互有胜负。但现在Oracle Labs和美国大学里面的研究院所做的最新即时编译技术的研究全部都迁移至基于Graal VM之上进行了，其发展潜力令人期待。如果Java语言或者HotSpot虚拟机真的有被取代的一天，那从现在看来Graal VM是希望最大的一个候选项，这场革命很可能会在Java使用者没有明显感觉的情况下悄然而来，Java世界所有的软件生态都没有发生丝毫变化，但天下第一的位置已经悄然更迭。

## 新一代即时编译器

对需要长时间运行的应用来说，由于经过充分预热，热点代码会被HotSpot的探测机制准确定位捕获，并将其编译为物理硬件可直接执行的机器码，在这类应用中Java的运行效率很大程度上取决于即时编译器所输出的代码质量。

HotSpot虚拟机中含有两个即时编译器，分别是编译耗时短但输出代码优化程度较低的客户端编译器（简称为C1）以及编译耗时长但输出代码优化质量也更高的服务端编译器（简称为C2），通常它们会在分层编译机制下与解释器互相配合来共同构成HotSpot虚拟机的执行子系统。

自JDK 10起，HotSpot中又加入了一个全新的即时编译器：Graal编译器，看名字就可以联想到Graal VM。Graal编译器是以C2编译器替代者的身份登场的。C2的历史已经非常长了，可以追溯到Cliff Click大神读博士期间的作品，这个由C++写成的编译器尽管目前依然效果拔群，但已经复杂到连Cliff Click本人都不愿意继续维护的程度。而Graal编译器本身就是由Java语言写成，实现时又刻意与C2采用了同一种名为“Sea-of-Nodes”的高级中间表示（High IR）形式，使其能够更容易借鉴C2的优点。Graal编译器比C2编译器晚了足足二十年面世，有着极其充沛的后发优势，在保持输出相近质量的编译代码的同时，开发效率和扩展性上都要显著优于C2编译器，这决定了C2编译器中优秀的代码优化技术可以轻易地移植到Graal编译器上，但是反过来Graal编译器中行之有效的优化在C2编译器里实现起来则异常艰难。这种情况下，Graal的编译效果短短几年间迅速追平了C2，甚至某些测试项中开始逐渐反超C2编译器。Graal能够做比C2更加复杂的优化，如“部分逃逸分析”（Partial Escape Analysis），也拥有比C2更容易使用激进预测性优化（Aggressive Speculative Optimization）的策略，支持自定义的预测性假设等。

Graal编译器未来的前途可期，作为Java虚拟机执行代码的最新引擎，它的持续改进，会同时为HotSpot与Graal VM注入更快更强的驱动力。

使用-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler参数来启用Graal编译器。

## 向Native迈进

对不需要长时间运行的，或者小型化的应用而言，Java（而不是指Java ME）天生就带有一些劣势，这里并不只是指跑个HelloWorld也需要百多兆的JRE之类的问题，更重要的是指近几年在从大型单体应用架构向小型微服务应用架构发展的技术潮流下，Java表现出来的不适应。

在微服务架构的视角下，应用拆分后，单个微服务很可能就不再需要面对数十、数百GB乃至TB的内存，有了高可用的服务集群，也无须追求单个服务要7×24小时不间断地运行，它们随时可以中断和更新；但相应地，Java的启动时间相对较长，需要预热才能达到最高性能等特点就显得相悖于这样的应用场景。在无服务架构中，矛盾则可能会更加突出，比起服务，一个函数的规模通常会更小，执行时间会更短，当前最热门的无服务运行环境AWS Lambda所允许的最长运行时间仅有15分钟。

一直把软件服务作为重点领域的Java自然不可能对此视而不见，在最新的几个JDK版本的功能清单中，已经陆续推出了跨进程的、可以面向用户程序的类型信息共享（Application Class Data Sharing，AppCDS，允许把加载解析后的类型信息缓存起来，从而提升下次启动速度，原本CDS只支持Java标准库，在JDK 10时的AppCDS开始支持用户的程序代码）、无操作的垃圾收集器（Epsilon，只做内存分配而不做回收的收集器，对于运行完就退出的应用十分合适）等改善措施。而酝酿中的一个更彻底的解决方案，是逐步开始对提前编译（Ahead of Time Compilation，AOT）提供支持。

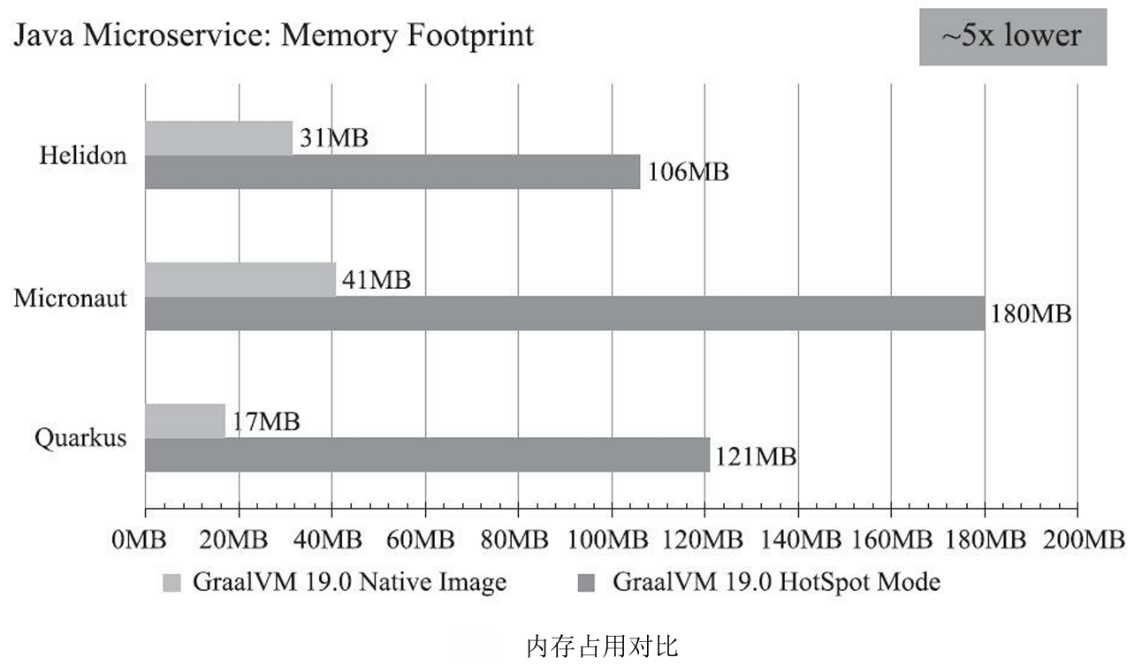
提前编译是相对于即时编译的概念，提前编译能带来的最大好处是Java虚拟机加载这些已经预编译成二进制库之后就能够直接调用，而无须再等待即时编译器在运行时将其编译成二进制机器码。理论上，提前编译可以减少即时编译带来的预热时间，减少Java应用长期给人带来的“第一次运行慢”的不良体验，可以放心地进行很多全程序的分析行为，可以使用时间压力更大的优化措施。

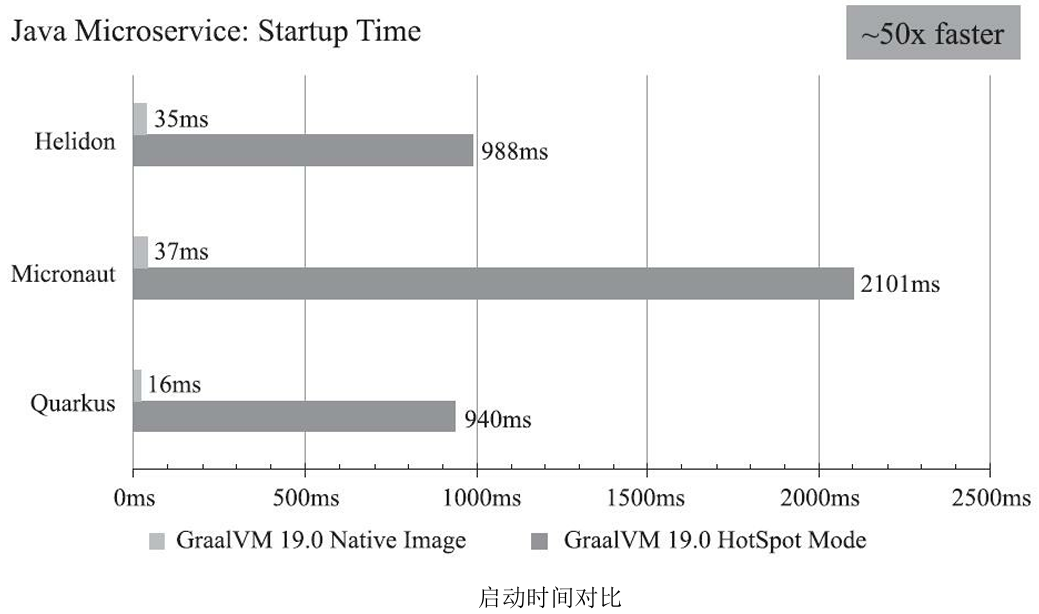
但是提前编译的坏处也很明显，它破坏了Java“一次编写，到处运行”的承诺，必须为每个不同的硬件、操作系统去编译对应的发行包；也显著降低了Java链接过程的动态性，必须要求加载的代码在编译期就是全部已知的，而不能在运行期才确定，否则就只能舍弃掉已经提前编译好的版本，退回到原来的即时编译执行状态。

早在JDK 9时期，Java就提供了实验性的Jaotc命令来进行提前编译，不过多数人试用过后都颇感失望，大家原本期望的是类似于Excelsior JET那样的编译过后能生成本地代码完全脱离Java虚拟机运行的解决方案，但Jaotc其实仅仅是代替即时编译的一部分作用而已，仍需要运行于HotSpot之上。

直到Substrate VM出现，才算是满足了人们心中对Java提前编译的全部期待。Substrate VM是在Graal VM 0.20版本里新出现的一个极小型的运行时环境，包括了独立的异常处理、同步调度、线程管理、内存管理（垃圾收集）和JNI访问等组件，目标是代替HotSpot用来支持提前编译后的程序执行。它还包含了一个本地镜像的构造器（Native Image Generator），用于为用户程序建立基于Substrate VM的本地运行时镜像。这个构造器采用指针分析（Points-To Analysis）技术，从用户提供的程序入口出发，搜索所有可达的代码。在搜索的同时，它还将执行初始化代码，并在最终生成可执行文件时，将已初始化的堆保存至一个堆快照之中。这样一来，Substrate VM就可以直接从目标程序开始运行，而无须重复进行Java虚拟机的初始化过程。但相应地，原理上也决定了Substrate VM必须要求目标程序是完全封闭的，即不能动态加载其他编译器不可知的代码和类库。基于这个假设，Substrate VM才能探索整个编译空间，并通过静态分析推算出所有虚方法调用的目标方法。

Substrate VM带来的好处是能显著降低内存占用及启动时间，由于HotSpot本身就会有一定的内存消耗（通常约几十MB），这对最低也从几GB内存起步的大型单体应用来说并不算什么，但在微服务下就是一笔不可忽视的成本。根据Oracle官方给出的测试数据，运行在Substrate VM上的小规模应用，其内存占用和启动时间与运行在HotSpot上相比有5倍到50倍的下降，具体结果如图





Substrate VM补全了Graal VM“Run Programs Faster Anywhere”愿景蓝图里的最后一块拼图，让Graal VM支持其他语言时不会有重量级的运行负担。譬如运行JavaScript代码，Node.js的V8引擎执行效率非常高，但即使是最简单的HelloWorld，它也要使用约20MB的内存，而运行在Substrate VM上的Graal.js，跑一个HelloWorld则只需要4.2MB内存，且运行速度与V8持平。Substrate VM的轻量特性，使得它十分适合嵌入其他系统，譬如Oracle自家的数据库就已经开始使用这种方式支持用不同的语言代替PL/SQL来编写存储过程。

## 灵活的胖子

即使HotSpot最初设计时考虑得再长远，大概也不会想到这个虚拟机将在未来的二十年内一直保持长盛不衰。这二十年间有无数改进和功能被不断地添加到HotSpot的源代码上，致使它成长为今天这样的庞然大物。

HotSpot的定位是面向各种不同应用场景的全功能Java虚拟机，这是一个极高的要求，仿佛是让一个胖子能拥有敏捷的身手一样的矛盾。如果是持续跟踪近几年OpenJDK的代码变化的人，相信都感觉到了HotSpot开发团队正在持续地重构着HotSpot的架构，让它具有模块化的能力和足够的开放性。

模块化方面原本是HotSpot的弱项，监控、执行、编译、内存管理等多个子系统的代码相互纠缠。而IBM的J9就一直做得就非常好，面向Java ME的J9虚拟机与面向Java EE的J9虚拟机可以是完全由同一套代码库编译出来的产品，只有编译时选择的模块配置有所差别。

现在，HotSpot虚拟机也有了与J9类似的能力，能够在编译时指定一系列特性开关，让编译输出的HotSpot虚拟机可以裁剪成不同的功能，譬如支持哪些编译器，支持哪些收集器，是否支持JFR、AOT、CDS、NMT等都可以选择。能够实现这些功能特性的组合拆分，反映到源代码不仅仅是条件编译，更关键的是接口与实现的分离。

早期（JDK 1.4时代及之前）的HotSpot虚拟机为了提供监控、调试等不会在《Java虚拟机规范》中约定的内部功能和数据，就曾开放过Java虚拟机信息监控接口（Java Virtual Machine Profiler Interface，JVMPI）与Java虚拟机调试接口（Java Virtual Machine Debug Interface，JVMDI）供运维和性能监控、IDE等外部工具使用。到了JDK 5时期，又抽象出了层次更高的Java虚拟机工具接口（Java Virtual Machine Tool Interface，JVMTI）来为所有Java虚拟机相关的工具提供本地编程接口集合，到JDK 6时JVMTI就完全整合代替了JVMPI和JVMDI的作用。

在JDK 9时期，HotSpot虚拟机开放了Java语言级别的编译器接口（Java Virtual Machine Compiler Interface，JVMCI并不是指内部代码上统一C1、C2的编译器接口，而是特指会开放给外部的、使用Java语言实现的编 译器接口），使得在Java虚拟机外部增加、替换即时编译器成为可能，这个改进实现起来并不费劲，但比起之前JVMPI、JVMDI和JVMTI却是更深层次的开放，它为不侵入HotSpot代码而增加或修改HotSpot虚拟机的固有功能逻辑提供了可行性。Graal编译器就是通过这个接口植入到HotSpot之中。

到了JDK 10，HotSpot又重构了Java虚拟机的垃圾收集器接口（Java Virtual Machine Compiler Interface 目前只在HotSpot内部使用，并未对外开放，所以也就没有JVMGI的提法），统一了其内部各款垃圾收集器的公共行为。有了这个接口，才可能存在日后某个版本中的CMS收集器退役，和JDK 12中Shenandoah这样由Oracle以外其他厂商领导开发的垃圾收集器进入HotSpot中的事情。如果未来这个接口完全开放的话，甚至有可能会出现其他独立于HotSpot的垃圾收集器实现。

## 语言语法持续增强

一门语言的功能、语法又是影响语言生产力和效率的重要因素，很多语言特性和语法糖不论有没有，程序也照样能写，但即使只是可有可无的语法糖，也是直接影响语言使用者的幸福感程度的关键指标。JDK 7的Coins项目结束以后，Java社区又创建了另外一个新的语言特性改进项目Amber，JDK 10至13里面提供的新语法改进基本都来自于这个项目，譬如：

* JEP 286：Local-Variable Type Inference，在JDK 10中提供，本地类型变量推断。
* JEP 323：Local-Variable Syntax for Lambda Parameters，在JDK 11中提供，JEP 286的加强，使它可以用在Lambda中。
* JEP 325：Switch Expressions，在JDK 13中提供，实现switch语句的表达式支持。
* JEP 335：Text Blocks，在JDK 13中提供，支持文本块功能，可以节省拼接HTML、SQL等场景里大量的“+”操作。

还有一些是仍然处于草稿状态或者暂未列入发布范围的JEP，可供我们窥探未来Java语法的变化，譬如：

* JEP 301：Enhanced Enums，允许常量类绑定数据类型，携带额外的信息。
* JEP 302：Lambda Leftovers，用下划线来表示Lambda中的匿名参数。
* JEP 305：Pattern Matching for instanceof，用instanceof判断过的类型，在条件分支里面可以不需要做强类型转换就能直接使用。

除语法糖以外，语言的功能也在持续改进之中，以下几个项目是目前比较明确的，也是受到较多关注的功能改进计划：

* Project Loom：现在的Java做并发处理的最小调度单位是线程，Java线程的调度是直接由操作系统内核提供的，会有核心态、用户态的切换开销。而很多其他语言都提供了更加轻量级的、由软件自身进行调度的用户线程（曾经非常早期的Java也有绿色线程），譬如Golang的Groutine、D语言的Fiber等。Loom项目就准备提供一套与目前Thread类API非常接近的Fiber实现。
* Project Valhalla：提供值类型和基本类型的泛型支持，并提供明确的不可变类型和非引用类型的声明。一些语言（如Scala）就有明确的不可变类型声明，而Java中只能在定义类时将全部字段声明为final来间接实现。基本类型的范型支持是指在泛型中引用基本数据类型不需要自动装箱和拆箱，避免性能损耗。
* Project Panama：目的是消弭Java虚拟机与本地代码之间的界线。现在Java代码可以通过JNI来调用本地代码，这点在与硬件交互频繁的场合尤其常用（譬如Android）。但是JNI的调用方式充其量只能说是达到能用的标准而已，使用起来仍相当烦琐，频繁执行的性能开销也非常高昂，Panama项目的目标就是提供更好的方式让Java代码与本地代码进行调用和传输数据。

随着Java每半年更新一次的节奏，新版本的Java中会出现越来越多其他语言里已有的优秀特性，相信博采众长的Java，还能继续保持现在的勃勃生机相当长时间。