JAVA20年

道路与梦想

QCon上海站

2015年10月15-17日

上海光大会展中心
国际大酒店

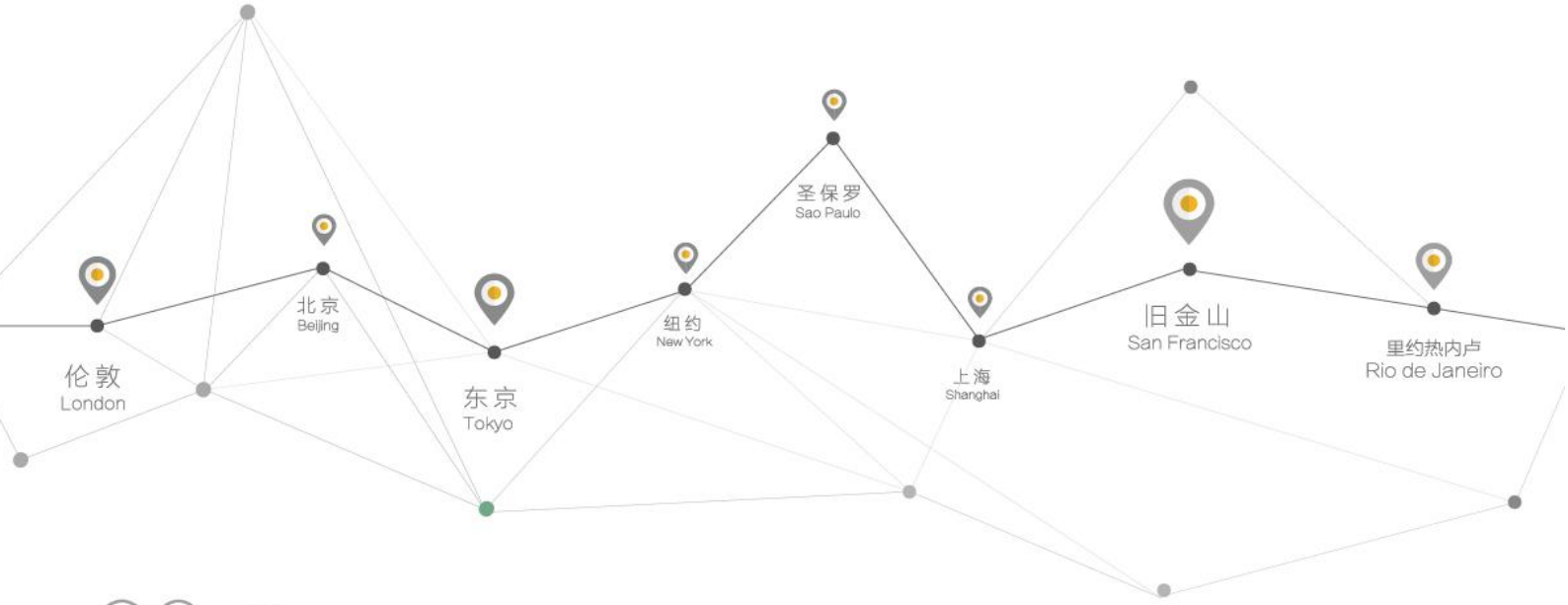
www.qconshanghai.com

Brought by InfoQ

QCon

全球软件开发大会

International Software Development Conference



QCon是由InfoQ主办的全球顶级技术盛会，每年在伦敦、北京、东京、纽约、圣保罗、上海、旧金山，里约热内卢召开。自2007年3月份首次举办以来，已经有超万名高级技术人员参加过QCon大会。QCon内容源于实践并面向社区，演讲嘉宾依据热点话题，面向5年以上的技术团队负责人、架构师、工程总监、高级开发人员分享技术创新和最佳实践。

www.qconferences.com

ArchSummit

International Architect Summit

全球架构师峰会2015

2015.7.17-18　中国·深圳·大梅沙京基海湾大酒店

侧重业务场景，引领技术趋势

HOT

9大热门技术专题

研发体系构建
在线教育
智能硬件
开源与企业发展
互联网金融
电商和零售业的转型
移动化机会
大数据背后的价值
企业云化的痛点与实践

INVITE

邀请制闭门会议

在线教育机遇与挑战
开源与企业发展
企业云化的痛点与实践
互联网基因的金融

议题提交开放，折扣购票启动，详情查阅大会官网

购票热线：010-89880682　会务咨询：arch@cn.infoq.com　大会官网：www.archsummit.com

卷首语一

Java 20 年：道路与梦想

互联网与 Java 的诞生

1995 年春天，那是我第一次亲眼见证并体验互联网的魅力，当时网景公司刚刚发布了 Navigator 浏览器。因为我有 C 语言和 SQL 的软件开发背景，所以我立刻问自己——如何进行互联网编程？应该使用哪种计算机语言？虽然这些问题一时还找不到答案，但有一点我非常清楚：我应当投身于互联网，到那里去学习互联网编程。

同样是在 1995 年，Sun Microsystems 公司正式推出了 Java 1.0.2 版本。我马上就被其简洁的语法及内置的各类网络功能所吸引。当然，我也被其语言扩展性和跨平台能力深深折服。遥想当年，软件工程师们已经被跨平台这一老大难问题困扰了数十年之久，Java 则利用虚拟机解决了这个难题。我希望能在 DLL、DCOM 以及难于使用但又将自己牢牢锁定的其它开发工作之外找到新的编程乐趣。紧接着，我就搬到了硅谷，开始在众多项目中使用 Java 语言。坦率地讲，Java 当时还没能成为主流的开发语言。不过我认为它代表着一种新的趋势。

Sun 公司加大了对 Java 项目的投入，为 Java 迎来了快速发展期。此后不久，Java 1.1（又一主要版本）发布，让 Java 的下载量很快就突破了 100 万。然后微软公司也决定顺应这股潮流，并由此发布了微软的 Java 语言---Visual J++，而这进一步提升了 Java 的人气。不过问题来了，微软加入自己的私有扩展的 Java 语言只能运行在 Windows 平台之上，而非 Linux/Unix 环境下，这导致了一场旷日持久的官司，最后微软败诉。不过微软方面并没有因此而放弃努力，他们转而着手开发.Net 与 Java 分庭抗礼。

Java 的下一个五年

2001 年互联网泡沫破裂，Java 语言的普及速度也开始放缓。在这段低迷时期，有一家公司开始成长并在电子商务领域取得了卓越的成绩，它就是 eBay.com。我有幸能够在 eBay 公司领导技术平台的重新设计并根据 Java 平台对其进行全面调整。值得一提的是，面对 C/C++平台的各种挑战，eBay 仍然成为发展速度最快的电子商务网站。当时的问题在于，随着网站流量的不断增长，Java 语言编写的应用程序能否扛的住？毕竟 Java 当时的性能很有问题。这一切在调优后能否得到改善？Java 在开发效率方面是否有机会压倒其它编程语言？

经过三年的平台重组工作，Java 全面地完成了所有的承诺，之前的问题也烟消云散了。到 2005 年，eBay 已经成为全球规模最大的 Java 业务平台。那时，我们已经全面完成了由 C/C++代码库到 Java 的迁移工作。在此过程中，我们接连经历了从 Java 1.1 到 Java 1.3，再到 Java 1.5 的几次颇具难度的版本升级。JVM GC 算法随着时间的推移而不断改善，Java 在 eBay 的成功已然成为 Java 在第一个发展十年内出色能力与巨大潜力的最有力证明。

在开源领域的巨大成功

2006 年，Sun 公司开源了 Java。在接下来的十年中，整个开源社区在利用 Java 构建开源项目方面获得了巨大的推进与发展助力。InfoQ、JUG、JCP 等社区赞助方在 Java 平台与开发的促进方面起到了积极作用。原本的开发者社区逐步成为 Java 成果的重要载体，并从多个方面推动

着 Java 的发展。开源社区中有越来越多高质量的 Java 框架出现，然后这些框架吸引了更多开发人员到 Java 平台中。社区培育出了一系列以 Java 为基础的工具与框架，使得整个生态丰富多彩。在由 Java 构建而成的项目当中，最为成功的当数 Hadoop 技术。Hadoop 已经在 Java 开发者当中获得了极高人气与信赖。Hadoop 的大数据概念当前正推动着众多行业寻找新的发展模式。时至今日，Java 开源社区中的活跃开发者数量超过 100 万，全世界 Java 开发人员的总数更是突破了 1000 万。

发展的生命周期

如果在诞生后的第一个十年，Java 走过了从婴儿到儿童的历程，那么如今的 Java 已经成为能够独挡一面的成年人，其语言功能已经全面成熟，开发人员拥有丰富而友好的开发环境。在使用 Java 与基于 JVM 的其它语言（如 Groovy、Scala、Clojure、JRuby 以及 Jython 等）时，开发人员的工作效率要远高于使用其它语言——特别是 Java 拥有大量开源框架及工具支持。Java 在开发后台服务方面一直领先于其它语言，这主要是由于 Java 代码拥有良好的可维护性与可管理性。在生产环境中，以 Java 为基础的解决方案拥有运营可追溯性优势以及更出色的社区支持力度。在招聘工程技术人才时，企业往往能够从 Java 开发者群体中更好地找到应聘对象——得益于 Java 的庞大开发者群体。

Java 的未来二十年

时至今日，Java 的身影在设备、云计算以及数据技术领域可谓随处可见。Java 对众多行业的发展产生了深远的影响，例如 ERP、电子商务、移动、社区、金融、游戏乃至一些我们想不到的领域。虽然自 Java 面世以来又有众多其它语言陆续出现，但它们在普及程度上仍然无法与 Java 相提并论。Java 对我的早期从业经历产生了巨大影响，在过去二十年中培养出整整一代开发人员，并将在未来继续为新生代程序员们指明发展方向。

尽管当下仍不断有新语言出现，但毫无疑问，未来二十年，Java 仍将会是最受欢迎的编程语言。如大家所知，Java 不仅仅只是一种主流编程语言，它同时也代表着一整个活跃的生态系统。Java 开发者们将自己的聪明才智投入到这个平台上，而平台则回报给他们工作岗位与相应薪酬。要打理好现有的 Java 解决方案，我们需要 Java。而为了顺利推动未来的业务发展，我们必将打造出更多 Java 应用程序。

随着移动互联网的井喷式发展，市场上出现了非常多的 Android 应用程序。而其发展依靠的正是 Java 的强大力量。随着科技的发展，越来越多的物联网设备将在未来几年中与我们见面。而 Java 也将继续在应用程序及服务的开发当中扮演重要角色，进而通过 Android 等技术方案实现网络设备互通互联。这种趋势目前刚刚起步并拥有可观的发展动力。随着整套开发平台的成熟与改进，Java 必将在未来的技术创新领域找到属于自己的定位。

携程网 CTO——**叶亚明**

携程网高级技术总监——**吴其敏**

卷首语二

Java 20 岁生日快乐

2003 年，我开始使用 Java 编程，一直到今天。12 年的陪伴造就了深厚的感情，今年 Java 20 岁，衷心的对 Java 说一句：Java，20 岁生日快乐，也趁此机会，谈谈对陪伴了自己 12 年的 Java 语言的感受，作为生日礼物献给 Java。

2002 年大学毕业，成为了一名 VB 程序员，但当时公司的另外一款核心的产品是用 Java 编写的，在业余时间自己就抽空自学习了 Java。2003 年，机缘巧合我转为参与那款核心产品的开发，于是变成了一个 Java 程序员，从此便和 Java 结缘。

2003 年到 2007 年，主要是使用 Java 来开发政府类型的软件，相对而言更多的是注重在功能的丰富程度上，在这个阶段主要感受到的是 Java 社区的强大以及 Java 工具/框架的丰富，这两点都促使了开发效率的提升。

2007 年加入阿里巴巴后，规模的原因使得自己更加关注 Java 在高并发、性能和稳定性方面的表现。

早在 2004 年 10 月，Java 的版本号历史性的从 1.4 跳跃为了 5.0，为的是说明这个版本带来了巨大的进步，而在规模领域来看，这个版本带来的最大变化是 juc(java.util.concurrent)包的引入，这个包一举奠定了 Java 在高并发场景中的不俗表现，juc 包各种经典的无锁、锁粒度精妙控制的实现成为了高并发场景 Java 程序员，甚至是其他语言的程序员必学的技巧，凭借此 Doug Lea 也成为了 Java 发展史上重要的人物之一。

高性能的网络通信是工作中另外一个重要的部分，Java 在 BIO、NIO 上的不断前进，以及基于 NIO 的 Mina、Netty、Grizzly 之争使得 Java 在实现网络通信的性能上已经做的非常不错，阿里巴巴在 2008 年开始由一个集中式系统演变为一个大型的分布式系统，Java 在通信上的不俗性能是保障成功完成演变的关键因素之一。

GC 是对于高性能 Java 应用而言一个重要的影响因素，Java 在 GC 上也不断的进行优化和演进，ParallelOldGC、G1GC 的推出对 Java 在 GC 上的影响都带来了更好的控制能力，在大内存领域 G1 GC 的表现也越来越好，这在这个内存变得越来越大的时代显得尤为重要。

在高性能领域方面，Java 持续不断的通过 JVM 来优化程序的执行性能，这对使用这门语言的程序员而言是巨大的帮助，一个同样的 Java 应用，运行在 JDK 7 下会比运行在 JDK 5 的情况下快上接近两倍。

从 2008 年到 2010 年，我们的应用因为 JVM 本身 bug crash 的现象越来越少，运转的状况揭示了 JVM 在稳定性上不断的改进。

综合来看，Java 在高并发、性能、稳定性方面不断改善的表现，使得阿里巴巴这个流量不断增长的网站保持了不错的性价比，社区的成熟、工具/框架的丰富度则为提升我们的开发效率起到了很大的帮助。

展望未来，相信 Java 仍将陪伴自己很多年，在未来的日子里，阿里巴巴将从一个 Java 语言的使用者演变为进步的推动者，帮助 Java 在跟随硬件、软件体系的进步上做的更好，在高并发、高性能领域有长足的进步，使得 Java 应用的运行成本能有明显的降低。

阿里巴巴——**毕玄**

目　录

Java 20 年：转角遇到 Go

作者 郭蕾

1995 年，横空出世的 Java 语言以其颠覆式的特性迅速获得了开发者的关注。跨平台、垃圾回收、面向对象，这在当时都是不可思议的事情，而 Java 却完美地在一门语言中实现了这一特性。可以说，Java 将编程语言设计带领到一个新的高度。20 年后的今天，当年的那些新特性已经不再是什么新鲜词。同时，又会有一些新的语言宣称自己有一些颠覆性的特性，其中 Go 语言就是新语言的一个代表，它部署简单、并发性好，在语言设计上确实优于 Java。为了了解 Java 和 Go 语言的发展现状与趋势，InfoQ 采访了 Go 语言大牛郝林。

InfoQ：今年的 5 月 23 日是 Java 的第 20 岁生日，转眼间，Java 已经走过了 20 年，版本号也已经更新到 Java 8。你怎么看 Java 这门语言？在这 20 年里，有哪些对你印象比较深刻的 Java 事件？

郝林：我觉得 Java 语言一路走来赚足了眼球也惹来了众多非议。就拿它随着 Sun 公司的没落被流转到 Oracle 公司来说吧。我记得当时有一大批 Java 程序员在网上扬言要摒弃 Java 语言，并且一部分人真的这么做了。但事实证明，Oracle 更好地发展了 Java。我认为从 Java 7 开始这门语言相当于迎来了第二春，在发展上增速了不少，各种新鲜特性和类库层出不穷。Java 8 给我印象最深刻的就是对 Lambda 表达式的支持。这使得 Java 真正地对函数式编程提供了支持。这是质的改变。也终将使 Java 语言走得更远。

InfoQ：从版本迭代的角度看，你认为 Java 的发展经历了哪几个阶段？

郝林：我是从 Java 1.3 的末期开始接触它的。所以在我看来 Java 1.3 之前就属于萌芽期吧（虽然那时它已被广泛使用了）。从 1.4 开始，Java 语言有了很多改观，比如 NIO、更多的垃圾回收器、性能上的提升、Java EE 规范的逐步简化，等等。所以我认为从此 Java 进入了第一个高速发展期（也许有上一个但我没赶上）。到了 Java 6 的时候，发展速度其实已经减缓不少了。这也可能是由于 Java 正处于被交接阶段的缘故。不过，我不得不说，Oracle 的调整动作很快，在几乎没有什么断档的情况下，Java 的发展又开始“跑”起来了。这也是我在前一个回答中说的“第二春”。

InfoQ：JVM 的普及促使相关周边语言不断涌现，你怎么看这些 JVM 语言？

郝林：这就是 Java 真正牛的地方。它不单单是一门语言，更是一个平台。到目前为止，JVM 语言已经有很多了，但是发展最好的是 Scala。它解决了一些 Java 在程序开发方面的问题。但是，我认为它的方向有所偏颇。我觉得“简化”往往比“丰富”来得更直接，效果也会更好。相比之下，Clojure 语言就做得很好。但是由于它是一个 Lisp 语言的方言，编码方式和思维方式与 Java 的面向对象思想相去甚远，所以仅仅被一小部分 Java 程序员接受。总之，JVM 语言让 Java 更加流行了。它们虽不完美，但却功不可没。

InfoQ：很多人都在唱衰 Java，您能结合 Java 的发展现状和趋势谈谈 Java 的前景吗？

郝林：任何一个流行的技术都会有人唱衰，更何况 Java 已经发展了 20 年了，中间又经历了种种坎坷。我觉得 Java 9 又会是一个里程碑式的版本。我很期待。我认为在我可预见的未来 Java 不会没落。实际上，Java 语言在企业级软件领域的霸主地位是不可动摇的。在互联网软件领域，它虽然受到了各种开发成本更低的语言（比如 Ruby 和 Python）的不断侵蚀，但是仍然占有一席之地。这正说明了 Java 生命力的顽强。不过，相比于 Java 语言，我更看好 Java 作为一个平台的前景。

InfoQ：你什么时候开始接触 Go 语言的？相比于 Java 语言，它有哪些优势？

郝林：我接触 Go 语言实际上并不算早，大约在 2013 年的上半年。那时候 Go 语言的版本是 1.0，1.1 版本正处于开发期。Go 语言给我的第一印象就是支持多种编程范式、提供了给力的程序构建和发布工具，以及在并发编程方面的极度简化。在当时，我认为 Java 语言的不足恰恰就包括了这几个方面。所以我义无返顾的开始学习并使用 Go 语言。事实证明，Go 语言虽属于新兴语言，但它却是一种革新。另外，与 Java 语言一样，Go 语言的向后兼容做的很好。并且，为了以防万一，它提供了一个命令用于自动地把旧版本的 Go 语言程序源码调整为当前版本的源码。诸如此类的“便捷大法”还有很多。许多在 Java 世界中只能依靠额外的类库或工具才能完成的事情，在 Go 语言看来却是手到擒来。当然，这种实实在在的优势也有诞生时间不同的缘故。正是由于 Java 已经历经了太多，所以在很多方面都很难改变。我觉得这是所有编程语言都应该正视的问题。显然，Go 语言的创造者们已经意识到了这一点。

InfoQ：出色的并发性能是 Go 语言区别于其他语言的一大特色。相比于 Java 的并发编程，它有哪些显著性的优势？

郝林：说到并发，Go 语言给人们的第一印象就是便捷。在这便捷之下，Go 语言权衡了各方面利弊，做了大量的工作，使得我们用极低的开发成本就可以编写出拥有超高运行性能的 Go 语言并发程序。其中最大的亮点就是，Go 语言把“激活”需要并发执行的代码块的操作内置了。我们仅通过一个关键字“go”就可以轻易地完成这项操作。

还记得我们在 Java 中为此需要编写的代码是多么的冗长吗？侵入式的接口实现声明和类继承声明、复杂的匿名内部类，以及困难重重的线程间协调和调度。这些都是不可忽视的程序开发维护成本。我们在编写和修改这样的并发程序时都要保持头脑和思路的绝对清晰，否则就会埋下祸根，搞出不易察觉和定位的 Bug。另一方面，如果透过表象看本质的话，我们就可以看到 Go 语言为了程序员的方便而做的大量的工作。

笼统地讲，Go 语言把对内核线程的使用和调度操作都内置到其运行时系统中了。但是，它远远要比一个线程池复杂得多。Java 线程与内核线程之间关系是 1:1 的。而 Go 语言的 Goroutine（可以看做是 Go 语言中执行并发代码块的实体）与内核线程之间的关系是 M:N 的。这让我们可以使用成千上万个 Goroutine 去执行并发代码块而仅仅耗费极少的内核线程。关于 Go 并发编程更详细的介绍，大家可以参看我著的“图灵原创”图书《Go 并发编程实战》。

InfoQ：Java 和 Go 语言的使用场景是不是不一样？

郝林：Java 语言与 Go 语言在使用场景方面其实有很多相似之处。例如，它们都适用于服务端程序的构建，并且可以很容易地编写出页面模板文件。又例如，它们在桌面软件方面都比较捉襟见肘。有意思的是，就本身而言，Go 语言在适用领域的优势更强，而在不适用领域的劣势也更加明显。优势方面我就不再赘述了，下面说说劣势。比如，用 Java 编写桌面程序起码还有 Swing 和 JavaFX 可选，但是 Go 语言官方至今还没有一个成熟的解决方案。当然，这仍旧与诞生时间有关。另外，我们还可以用 Java 语言编写 Android 应用程序。Go 语言目前虽然已经涉足，但还不完美。不过我在这里爆料一下，我很期待能用 Go 语言编写 iOS 应用程序。实际上，Go 语言在这方面已经有所进展了。总之，两种语言在适用领域方面有所重叠但又有些不同。在很多情况下，我们可以混用这两种语言。

InfoQ：现在的开发语言特别多，Java、Go、PHP、Rust、Python 等，你认为未来语言的发展趋势是怎么样的？

郝林：的确，现在的编程语言层出不穷、多如牛毛。但是编程语言的兴衰是有规律可循的。第一个规律是顺应时代的语言才能有更好的发展。正如 Objective-C 因 iPhone 和 iPad 的诞生而变得火热至极那样。而 Java 也因 Google 公司的“横插一足”而在移动程序开发领域占领了制高点。当今的计算机世界正处于“云”的时代，而从处理器的角度看也正处于多核时代。谁能够更好地把握住这些时代标签，谁就会在发展上更具优势。当然，这里说的“把握住”是需要有真功夫的。只喊不练不起任何作用，而且还会遭人唾弃。第二个规律是能够解决问题的语言就是好语言。对于任何场景都是如此。我相信每个技术团队都会在选择编程语言时进行一番权衡。哪种编程语言能更快更好地解决问题（这也涉及到开发和维护成本），它就肯定会胜出。从这方面看，编程语言并没有好坏之分。它们都必有独特的优势和擅长做的事情，否则就根本不会诞生出来了。而问题的解决能力几乎是发展趋势的唯一评判标准。“多快好省”就是选择编程语言的要诀。这也会从侧面预示一个编程语言的发展趋势。说了这么多，我另一个想要表达的意思是：对于它们的未来，我无法预知。

受访嘉宾介绍

郝林，软件工程师，从事软件开发工作 9 年有余。既搞过企业级软件项目，也堆过互联网软件系统。近期在使用和推广 Go 语言，著有“图灵原创”图书《Go 并发编程实战》，以及在线免费教程《Go 语言第一课》和《Go 命令教程》。

Java 20 年：历史与未来

作者 郭蕾

作为最受欢迎的编程语言之一，Java 已经走过了 20 个年头。从已经落寞的诺基亚到现在火热的电商系统，我们都能看到 Java 语言的身影。从 1995 年的第一个版本到现在的 Java 1.8，我们甚至能从 Java 的版本迭代中看到不同时代编程语言关注的重点。经过了过去 20 年的发展，Java 已经成为如今使用最为广泛的企业级语言。为了庆祝 Java 的第 20 个生日，InfoQ 为此采访了 Java 技术专家彭晨阳（网络 ID：板桥）。

InfoQ：您是哪一年开始接触 Java 的？还记得当时『世界』是怎么看这门语言的吗？

板桥：我大概是 2000 年之前开始接触 Java，当时大家都认为 Java 慢，几乎没有几个人看得上眼，那时使用 Perl/C 实现 CGI 比较快，PHP 很方便。

InfoQ：能回忆下你的职业生涯中与 Java 相关的经历吗？

板桥：2000 年之前使用 Perl 开发过一个类似西祠、西陆社区网站，随着功能日益复杂，维护拓展比较麻烦，打算使用 Java 改造升级。但是 Java 比较复杂，当时有 EJB 等规范，因此误用过 EJB 来做产品，其实 EJB 更适合做企业中可靠性要求比较高的项目。而对于社区项目来说，性能是关键，这个道理后来我从 CAP 定理中才得到答案，当然当时也没有听说过 CAP 理论，这段教训是相当深刻的，EJB 很难掌握，运行起来更慢，最后也以失败告终。

之后研究学习了 Jive 开源 Java 论坛，对其设计模式与缓存两个优点进行了综合学习与应用。有一段时间参与过手机游戏的开发，那时客户端是 J2ME，但是游戏逻辑不加载在客户端，而是将客户端只作为界面展现，类似今天的浏览器+Angular.js 这样富客户端。当然，这个系统对网络要求比较高，但是当时无线网络 3G 还没有推出，后来放弃了，从该项目中我意识到高性能的大型并发系统使用 Tomcat 这样的普通 Web 服务器已经无法承担，于是对异步消息 JMS 等技术产生了兴趣。

之后，陆陆续续参与过一些项目的咨询和设计，大部分都比较普通，无非是 CRUD 增删改查。于是萌生了做一个快速开发框架，在不丢失多层架构的基础上能有 Delphi 等二层架构的开发效率，这大概是 JDON 框架的原型。当然，该框架后来从快速开发为首要目标转移到灵活性为首要目标。

做了不少项目后，需要寻求理论指导，原来的数据库+Java 路数已经不能包打天下，后来逐步开始引入 DDD 领域驱动设计 CQRS 和 EventSourcing。

InfoQ：很多人都在唱衰 Java，您能结合 Java 的发展现状和趋势谈谈 Java 的前景吗？

板桥：Java 发展到今天已经 20 年了，作为一个编程语言确实不简单，想当初人人受怀疑的慢语言到今天通用的健壮语言，真是大智若愚啊。Java 代表的面向对象思想确实给工程领域带来了革命性的变化，当然思想是不断进化发展的，如今人们开始看好函数式编程

语（FP）。尽管 Java 8 也加入了函数语言的特点，但是 OOP 和 FP 两者到底是不同的编程范式，不过掌握 FP 有一定门槛，这也是造成很多人观望的一个原因。

Java 在数据流处理方面还是很有竞争力的，而大数据实时流处理系统是 Java 的新领域，在这个领域有 Apache Kafka、Apache Samza、Apache Storm、Apache Spark 的 Streaming 模块和最新的 Apache Flink。Spark 是基于 JVM 的函数语言 Scala 编写，其余都是 Java 编写。

InfoQ：JVM 的普及促使相关周边语言不断涌现，你怎么看这些 JVM 语言？

板桥：以 Scala 为代表的 JVM 语言发展迅速，Scala 语言特性是首先区分不变性和可变性，当初使用 EJB 时首先要区分是无状态和有状态，这说明思路是一脉相承的。可变性的状态是造成副作用和各种 Bug 的罪魁祸首，可能我们如果只是把可变状态使用数据库实现时没有注意到这种问题。其实这个问题遍布在应用的每个角落，特别是使用类和对象这个 OOP 概念实现时最容易发生。一个类或对象包括字段和方法，如果这个字段值是可变的（可变状态），我们使用这个对象时如果不打开它的类代码是无法得知它有可变状态的，那么就会导致各种副作用发生。

而函数编程由于函数方法是第一公民，没有什么东西挡在它的前面，没有类或对象包裹着它们，因此，它们无法私藏可变状态字段，能够确保应用系统每行代码都是基于不可变的基础之上。

如果说 Scala 之类 JVM 的函数语言适合不断添加功能函数的应用场景，那么 Java 之类的 OOP 语言适合不断增加实体物体的应用场景。前者好动，后者好静。

InfoQ：Java 是如何拥抱云时代的？

板桥：Java 在云时代面临以 Go 语言为主的容器（Docker 等技术）生态圈的挑战。其实 JVM 也是一种容器，但是这种容器特性正在被 Linux 学习与赶超，那么，JVM 的定位就可能比较尴尬。

Docker 之类容器可以在本地笔记本或电脑上运行，然后同样可以部署到云上运行。当在云上运行时，Kubernetes 能够以一种可控的方式升级容器从而实现运行管理一批容器，如同一个大型船队或舰队一样，你可以控制它们的流量访问量，可以指定多少个容器来扩展支撑一个服务的运行，随着访问量提升，你通过增加容器数量能够整个系统的负载能力。

当然，Java 的大型分布式系统越来越多，Java 在云计算与分布式系统中还是扮演主要角色，形成一个大型的生态圈。当我们站在泰山之上，一览众山小，当你在全球拥有多个数据中心时，语言已经变得不那么重要了，关键是架构设计。

InfoQ：Go 语言这两年比较火热，你怎么看这门语言？与 Java 相比，他有哪些优劣？

板桥：Go 语言相对 Java 主要优点是其并发组件模型，Java 的并发比较低级，无非是多线程与锁，想搞清楚 Java 中各种锁的用途，包括数据集合 Collection 的线程安全性与性能差异对比，需要花费大量时间与精力，包括使用经验。而 Go 语言使用了 Channel/CEP 这样的组件简单封装了多线程与锁，将以前 JMS 的 Queue 队列模型架构引入到了语言之中，

两个对象之间交互只要通过 Channel 通道就可以。这种模型保证了并发性，有简化了编程模型，无疑受到很多人的欢迎。

Go 语言当前也受到更加强劲的 Rust 语言挑战，如果说，Go 语言的 Channel 是一种有形的设计，那么，Rust 语言的并发模型达到无形的设计，只要你编写好函数方法，安全性与并发性就无形中得到了解决，不用专门去思考并发，有意识地去使用并发组件模型编程。

InfoQ：现在的开发语言特别多，Java、Go、PHP、Rust、Python 等，你认为未来语言的发展趋势是怎么样的？

板桥：现在的开发语言如雨后春笋，主要原因是 CPU 进入多核并发时代，以及大型架构进入分布式系统，如何使用一种语言从微观的 CPU 多核之间并发到数万台服务器之间的分布式计算处理，这种大一统的愿景促使人们在不断探索。

在 Java 中，我们可以通过框架来实现这点，以我前几年研发 Jdon 框架为例，虽然能够勉强实现并发与分布式，但是这种实现需要很强的知识背景，不利于初学者上手。而要达到普及这个目标，必须从语言入手，让语言初学者在学习掌握语言以后，无形中就会实现了并发与分布式。

Go 语言在这方面比较突出，其并发模型以读写操作为基础。请注意，Java 等语言并发模型没有这么高，它们是以线程为基础，再应用到读写场景中，而我们现实中必须以读写为基础，再应用到具体业务场景中，这里面高低层次：线程–>读写操作–>业务应用，无疑越靠近业务应用的语言越能简化我们的开发，而分布式系统也是基于读写操作，著名的 CAP 定理也隐含了以读写操作为基础的语境。

受访嘉宾介绍

彭晨阳，Jdon.com 创办者和版主。软件开发设计咨询从业 20 年，10 余年 Java 开发经验，拥有 ERP、大型游戏、互动电视三网合一等架构经验。独立咨询顾问，个人擅长复杂系统的软件架构和领域建模。流行新技术思想的传道者，主持解道网站跟踪国际最新软件架构思想和设计技术。首个国内 Java 开源框架项目 Jdon 框架的设计者。

Java 20 年：JVM 虚拟化技术的发展

作者 郭蕾

虚拟化技术已经有了几十年的发展历史，并且在硬件、操作系统层面都已经得到了广泛的应用。虚拟化不但可以显著节省成本，而且还可以提升管理性。同样，虚拟化技术也可以应用在 JVM 中，以提高资源利用率，降低单应用的部署成本。早在 2004 年，Sun 公司就提出过 Java 应用虚拟化的设想，并且还制定过两个 JSR 规范。那现在 JVM 虚拟化技术发展到了哪一步？基于 JVM 的虚拟化技术在实现过程中有哪些难点？为了回答这些问题，InfoQ 采访了 JVM 专家李三红。

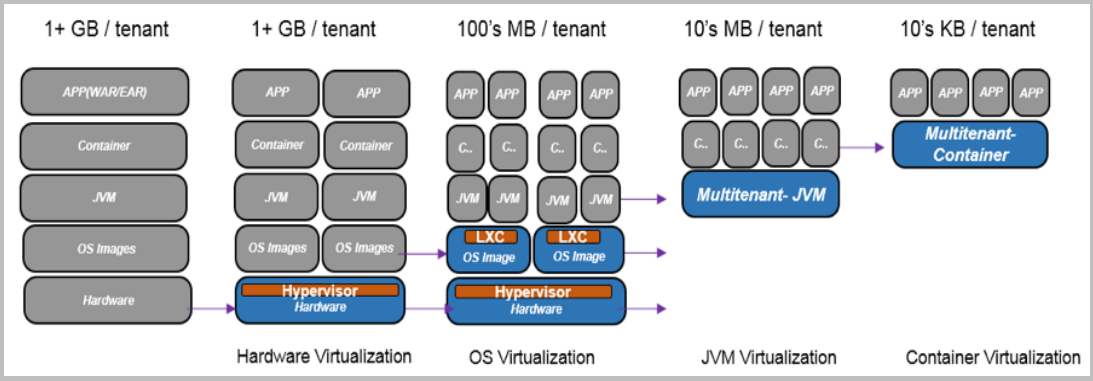
定义以及历史

InfoQ：能介绍下什么是 JVM 的虚拟化吗？JVM 的虚拟化价值在哪里？

李三红：传统的 Java 应用部署模式，一般遵循“硬件->操作系统->JVM->Java 应用”这种自底向上的部署结构，其中 Java EE 应用可以细化为“硬件->操作系统->JVM->JavaEE 容器->JavaEE 应用”的部署结构。这种部署结构往往比较重，操作系统、JVM 和 JavaEE 容器造成的 overhead 很高，而很多时候一个 Java 应用并不需要跑满整个硬件的资源，导致这种模式的资源利用率是比较低的。

而另一方面，硬件虚拟化技术逐渐成熟：VMware Hypervisor、Xen、KVM、Power LPAR 等技术能够帮助我们在同一个硬件上部署多个操作系统实例（每个操作系统实例可以理解为宿主机的租户），而时下流行的 OS Container 技术如 LXC、Docker 等，则是把操作系统虚拟化为多个实例，实现更轻量级的虚拟化。无论哪个层面的虚拟化，其目的都是对资源利用率更加高效的追求。

同样的思路，我们也可以在 JVM 层面，或者容器框架层面做虚拟化，类似于 Hypervisor 或者 OS Container，让虚拟化的 JVM/容器框架可以支持多租户的运行模式，这是比 OS 虚拟化更高一层的做法：



如上图，从左到右来看，随着虚拟化层次的提高，从 Hardware 到 OS，到 JVM，再到容器，单个租户应用的部署成本也在下降，最右边的“多租户”的 JavaEE Container（Multitenant-Container）是在容器框架层面的虚拟化，可以支持更高密度的应用部署，而不是传统的 APP : Container = 1:1 的部署方式。

什么是单个租户应用部署成本？举一个简单的例子，在没有虚拟化之前，传统的部署模式，一个 JavaEE 应用需要独占所有的硬件资源（CPU、MEM、网络等）。Hypervisor 的出现，使得一个共享的硬件资源上可以同时跑多个 OS，这种资源使用上的节约本质上是通过 Over-Commit（即允许租户超量使用物理资源）来达到的，即我们假设跑在同一个虚拟化环境的不同租户不会在同一个时间消费同样的资源。同样的道理可以来理解 JVM/容器框架的虚拟化。

多个 JavaEE 应用可以部署在同一个 JVM/容器内，但逻辑上它们各自会认为是运行在一个独立的 JVM/容器内，从而可以更大程度的提高资源利用率，降低单应用的部署成本。

InfoQ：多租户 JVM 的概念是什么时候出现的？能聊聊它的发展历史吗？

李三红：早在 2004 年，Sun 公司就提出过 JVM 虚拟化这方面的想法，当时 Grzegorz Czajkowski 领导了一个叫做巴塞罗那的研究项目，该项目基于 Java HotSpot 虚拟 1.5 版本开发了 Multi-Tasking Virtual Machine（MVM）。MVM 旨在提高 Java 程序的启动速度，节省内存开销。不过自从 Sun 被甲骨文收购后，我们没有听到关于该项目的任何新的进展。尽管没有看到 MVM 成功产品化，不过它却留下两个重要的 JSR（Java Specification Request）规范：JSR121 和 JSR284。JSR284 目前在 java.net 上有一个实现它的孵化项目 jsr284-ri-tck（ri: Reference Implementation, tck: Test Compatible Kit）。

从 2009 年开始，IBM Java 团队也开始着手研究 JVM 的虚拟化技术，并于 2013 年发布第一个基于 IBM J9 虚拟机的 Bata 版本。IBM Multi-tenant JVM 是 JVM 层面的虚拟化，其思路是把多个标准的 Java 应用运行在同一个 JVM 上，让这些应用共享底层的 GC、JIT、Java 运行时库等基础组件。

Waratek 也在 2012 年的 Red Hat 技术峰会上发布了 CloudVM Beta 版本，支持多租户。与 IBM Multi-tenant JVM 类似，Waratek 允许多个应用运行在同一个 CloudVM 上，每一个应用运行在一个叫 Java Virtual Container（JVC）的容器里。

应用场景以及发展前景

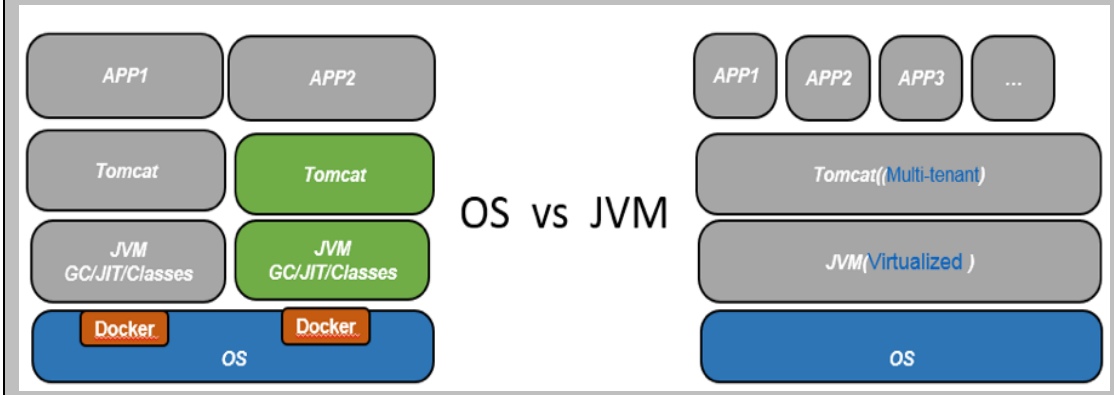
InfoQ：JVM 虚拟化最适合应用在什么场景中？

李三红：虚拟化的本质就是把本来独占的资源共享出来，即在 A 不需要的时候，B 可以拿来用，从而提高资源的利用率。

如果我们能把独占资源的应用，通过 JVM 的虚拟化的能力，同时部署在同一个“多租户”JVM 里，或者是同一个“多租户”容器里， CPU、Heap 这些资源就可以极大程度地共享了，部署的密度也就可以成倍的提高，其结果就是单应用的部署成本成倍的下降。打个简单的比喻，如果按照 1：1 的部署密度，机器的成本是 1000 台，如果我们能把部署的密度提高到 1:10，机器的成本就是 1000/10=100 台。

多租户的 JVM 或者容器，其最大的应用场景就是高密度的应用部署场景。也许有人有疑问，OS 虚拟化也可以提高部署密度， 为什么要选择 JVM 虚拟化？

我们就拿 Docker 作为一个例子来比较一下。比如利用 Docker 可以把运行在同一个实际操作系统上的单个 JVM 进程互相隔离，每个 JVM 进程可以看作是实际操作系统的一个租户，下面是一个简化的例子（先不考虑复杂的 OS 内存交换场景）：



左图表示在 OS 之上，使用 Docker 虚拟出来两个租户，即两个独立的 JVM 进程分别运行在两个隔离的 Docker 容器里。右图则表示 JVM/容器的虚拟化方案。在 Docker 的方案里，由于两个 App1 和 App2 是跨 JVM 进程的，所以 Heap 资源对于它们来说不是共享的，很难在 Heap 内存在 App1 不使用的时候，可以拿来给 App2 使用，这个就是一个限制 Docker 这种方案部署密度的因素，比如你有 8G 内存，你可能会选择部署两个 Docker 容器，每个 4G。而这个因素对于 JVM 虚拟化方案，则不存在这样的限制，因为所有的 App 都是在一个进程里的，Heap 对大家来说都是共享的，完全可以通过合理的安排选择部署更多的 App。

InfoQ：你认为多租户 JVM 的前景如何？为什么现在做多租户 JVM 的厂商这么少？

李三红：在 IBM 的时候，其实我们内部有过很多次的讨论。如果我们从虚拟化的这个角度来看，从硬件，到 OS，再到 JVM，以及容器，JVM 的虚拟化（多租户 JVM）以及容器的虚拟化（多租户容器）是一个技术的发展趋势。但是，现实的情况是，根据我们这些年在这个领域的经验与积累，基于传统的 JVM 实现 JVM 虚拟化，技术上碰到的很多的挑战，其中一些的确难于在短期内找到一个好的解决方案。（关于里面的一些实现细节，读者可以参考我之前的文章，高密度 Java 应用部署的一些实践）。简单列举一个例子，在多租户 JVM 中，每一个租户都有自己独立的 Region，但是如果租户与租户之间出现交叉引用的情况，就会导致 object leakage，这个是我们内部使用的一个技术术语。其结果是，在某些条件下，某个租户退出 Region 不能合理释放。这个问题头痛的地方在于，每次 Java class library 的代码改动，都有可能造成新的 object leakage。类似这样的问题，需要 Java Class Library 开发者和多租户 JVM 之间有一种约定，或者范式来共同约束解决这类问题。

我们再来谈第二个问题。就像你说的，目前做多租户 JVM 的厂商的确很少，事实上本来做商业 JVM 的厂商也就那么几家。IBM J9 是比较早切入这个领域的 JVM，主要是由中国和加拿大的团队一起开发，IBM 也只是做了 JVM 这层的虚拟化，并没有支持到容器级别。不过据我所知，这个项目已经停了，停的原因可能很多，我个人认为主要的原因是：第一，IBM 没有真实的应用场景来检视高密度部署的实际价值。第二，资源和优先

级上的考虑，Java 技术中心有更重要的东西来做。不过到目前为止，IBM 好像还没有在正式场合公布这个东西。

如果我们广义上来谈 JVM 虚拟化，而不要局限于多租户 JVM（即一个 JVM 上跑多个 Java 应用的场景），我个人觉得 JVM 虚拟化技术在高密度部署领域是有非常大的价值，尤其是在云计算大行其道的当今，我想这也是 Waratek 一直所信奉的。这里谈到应用的前景，我想有两个问题值得再深入考虑：第一，在云计算的环境下为 JVM 虚拟化找到高密度部署的应用场景，单个租户应用部署成本是关键的考量因素。第二，基于特定的场景，规避一些技术实现上难点。

Java 语言的发展

InfoQ：一晃眼，Java 就已经 20 岁了，你认为 Java 的发展经历了哪几个阶段？

李三红： James Gosling 带领团队在 1991 年开始了一个叫"Oak"的项目，这个就是 Java 的前身。1995 年，Java1.0 发布。“Write once, run anywhere" 这句 Java 口号想必大家耳熟能详。Java 刚开始出现的时候主要面向 Interactive Television 领域，直至后来几年的发展，SUN 一度想用 Java 来打造桌面的网络操作系统，取代当时如日中天的 Windows。不过不曾想，Java 虽未在 Embedded、Desktop 领域内取得多大的建树，出乎意料地，却在企业级应用领域开花结果，占据了如今几乎统治的地位。失之東隅，却收之桑榆。

Sun 在 2006 年的 Java One 大会上，宣布 Java 技术开源，随后年底的时候在 GPL 协议下发布 HotSpot 以及 javac，这是 Java 发展中的里程碑事件。从 2006 到 2010 年，这期间 SUN 一路磕磕绊绊，尽管手握 Java 这个金矿，但在商业上一直找不到好的盈利点，直到 2010 被 Oracle 收购。2010 年，也是 Java 发展的一个重要的分水岭，Java 面临分家的风险，一方是 Apache Harmony 为代表的，其后是 IBM 的支持，另一方是 OpenJDK 及其背后的 Oracle。最后博弈的结果现在大家都知道了，IBM 转向 OpenJDK，Apache Harmony 也结束了它的历史使命，被 Apache 之后束之高阁。Harmony 为 IBM 在 Java 上赢得的应有的话语权，另外一个副产品，就是给移动端 Android 平台贡献了 Java 核心类库代码。2010 年是 Java 重生的一年。随后 2011 年 OpenJDK 7 发布，OpenJDK 7 是第一个 Java SE 的引用实现。

InfoQ：反过头来看，你认为为什么 Java 能够如此成功？

李三红： 好像有这么一句话，原话不太记得了，一流的公司建生态，二流的公司定标准，三流的公司卖产品。这句话也可以用在 Java 上。Java 的成功，离不开逐步完善，日臻成熟的 Java 技术生态圈，这个是远远超出语言层面优劣的东西。在产品的选型上，无论纵向上，还是横向上，以 Java 技术构建的产品都有着无比的丰富性，Apache 社区的繁荣，可以作为这个故事的佐证。这个也是许多小众语言诸如 Scala、JRuby 愿意选择运行在 JVM 上的原因。

如果单从语言这个层面，谈论 Java 的成功，我就借用 Mark Reinhold 的一句总结，四个词概括 Java 的特征：Readability、Simplicity、Universality 和 Compatibility。这也是 Java 语言能够风行，并被大家所接受的重要因素。

受访嘉宾介绍

李三红，蚂蚁金服 JVM Architect，前 IBM Multi-tenant JVM 项目技术负责人。目前在蚂蚁金服基础技术部，负责 OpenJDK/HotSpot 相关的开发优化工作。十多年的 Java 开发经验，2008 年加入 IBM，参与基于 OSGi 框架的安全方面的开发，2010 年加入 Java 技术中心，参与 IBM Java 虚拟机 J9 的开发，在 Java/JVM 技术的领域拥有多项专利。 在 JavaOne、OSTC、IBM Technical Summit、IBM APN Summit 等会议上担任演讲嘉宾，上海 Java 技术社区 JUG（Java User Group）组织者。

借助开源工具高效完成 Java 应用的运行分析

作者 Joachim Haagen Skeie 译者 李勇

不止一次，我们都萌发过想对运行中程序的底层状况一探究竟的念头。产生这种需求的原因可能是运行缓慢的服务、Java 虚拟机（JVM）崩溃、挂起、死锁、频繁的 JVM 暂停、突然或持续的高 CPU 使用率、甚至于可怕的内存溢出(OOME)。好消息是现在已有许多工具能帮你得到 Java 虚拟机运行过程中的不同参数，这些信息有助于你了解其内部状况，从而诊断上述的各种情况。

在这篇文章中，我将介绍一些优秀的开源工具。其中一些是 JVM 自带的，另一些则是第三方工具。我将从最简单的工具开始介绍，逐渐过渡到一些比较复杂的工具。本文的目的是帮助你找到合适的调试诊断工具，这样当程序出现执行异常、缓慢或根本不能执行时，手头随时有可用的工具。

好了，让我们出发。

如果程序出现不正常的高内存负载、频繁无响应或内存溢出，通常最好的分析切入点是查看内存对象。幸好 JVM 内置了工具“jmap”，让它天生就能完成这种任务。

Jmap（借助 JPM 的一点帮助）

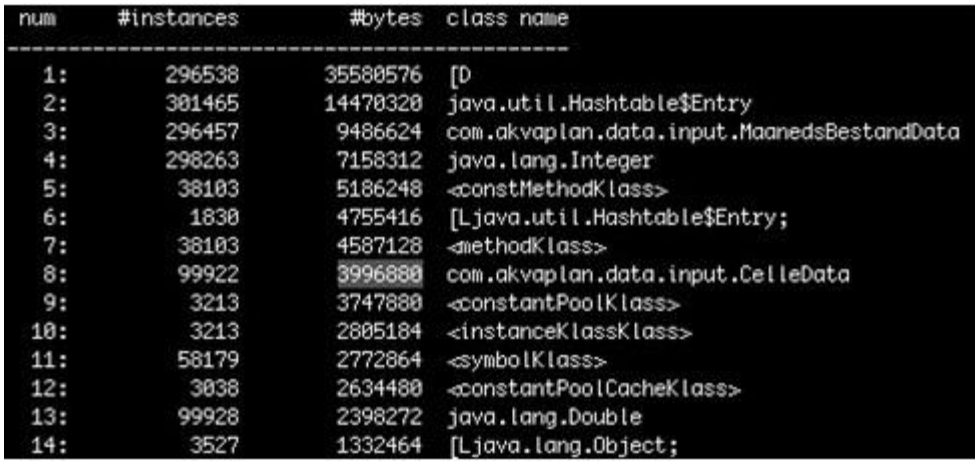
Oracle 将 **jmap** 描述为一种“输出进程、核心文件、远程调试服务器的共享对象内存映射和堆内存细节”的程序。本文将使用 jmap 打印一张内存统计图。

为了运行 jmap，你需要知道被调试程序的 PID（进程标识符）。得到 PID 的简单办法是使用 JVM 提供的 jps，它能列出机器上每一个 JVM 进程及其 PID。jps 输出结果如下图：

图 1：jps 命令的终端输出

为了打印内存统计图，我们需要打开 jmap 控制台程序，并输入程序的 PID 和“-histo:live”选项。如果不添加这个选项，jmap 将完整导出该程序的堆内存,这不是我们想要的结果。所以，如果想得到上图中“eureka.Proxy”程序的内存统计图，我们应该用如下命令来运行 jmap：jmap –histo:live 45417

上述命令输出如下：

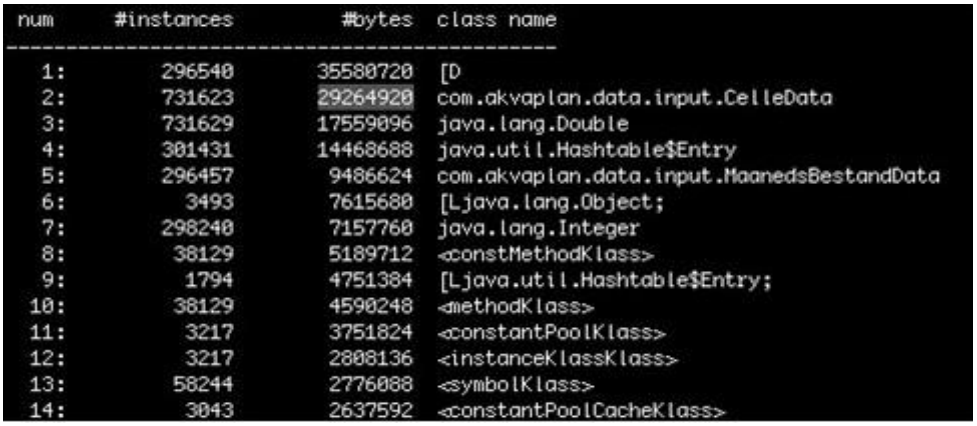


num	#instances	#bytes	class name
1:	296538	35580576	[D
2:	301465	14470320	java.util.Hashtable$Entry
3:	296457	9486624	com.akvaplan.data.input.MaanedsBestandData
4:	298263	7158312	java.lang.Integer
5:	38103	5186248	<constMethodKlass>
6:	1830	4755416	[Ljava.util.Hashtable$Entry;
7:	38103	4587128	<methodKlass>
8:	99922	3996880	com.akvaplan.data.input.CelleData
9:	3213	3747880	<constantPoolKlass>
10:	3213	2805184	<instanceKlassKlass>
11:	58179	2772864	<symbolKlass>
12:	3038	2634480	<constantPoolCacheKlass>
13:	99928	2398272	java.lang.Double
14:	3527	1332464	[Ljava.lang.Object;

图 2：命令 jmap -histo:live 的输出结果显示了堆中现有对象的个数

结果中每行显示了当前堆中每种类类型的信息，包含被分配的实例个数及其消耗的字节数。

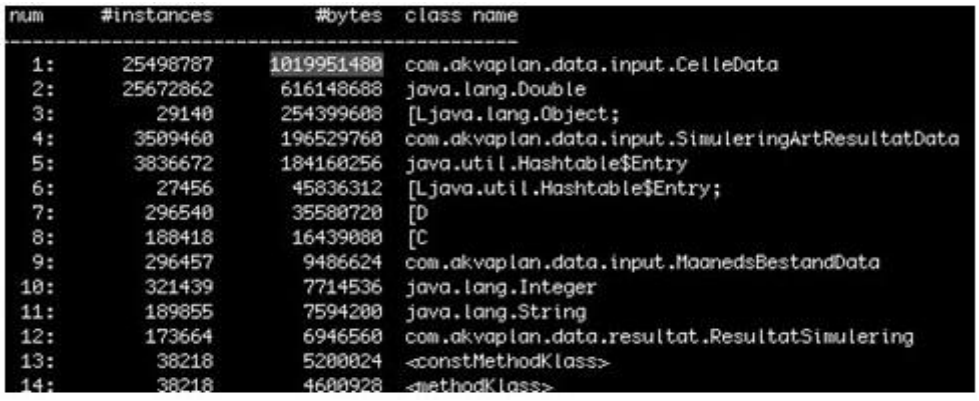
本例中，我请同事有意给程序增加了一处明显的内存泄露。请特别注意位于第 8 行的类，CelleData。将它与下图显示的 4 分钟后截屏进行比较：



num	#instances	#bytes	class name
1:	296540	35580720	[D
2:	731623	29264920	com.akvaplan.data.input.CelleData
3:	731629	17559096	java.lang.Double
4:	301431	14468688	java.util.Hashtable$Entry
5:	296457	9486624	com.akvaplan.data.input.MaanedsBestandData
6:	3493	7615680	[Ljava.lang.Object;
7:	298240	7157760	java.lang.Integer
8:	38129	5189712	<constMethodKlass>
9:	1794	4751384	[Ljava.util.Hashtable$Entry;
10:	38129	4590248	<methodKlass>
11:	3217	3751824	<constantPoolKlass>
12:	3217	2808136	<instanceKlassKlass>
13:	58244	2776088	<symbolKlass>
14:	3043	2637592	<constantPoolCacheKlass>

图 3：jmap 的输出表明 CelleData 类的对象数目增加了

请注意 CelleData 类现在已经变为系统中第二多的类，短短 4 分钟内已经增加了 631,701 个额外实例。等待约一小时后，我们观察到如下结果：



num	#instances	#bytes	class name
1:	25498787	1019951480	com.akvaplan.data.input.CelleData
2:	25672862	616148688	java.lang.Double
3:	29140	254399608	[Ljava.lang.Object;
4:	3509460	196529760	com.akvaplan.data.input.SimuleringArtResultatData
5:	3836672	184160256	java.util.Hashtable$Entry
6:	27456	45836312	[Ljava.util.Hashtable$Entry;
7:	296540	35580720	[D
8:	188418	16439080	[C
9:	296457	9486624	com.akvaplan.data.input.MaanedsBestandData
10:	321439	7714536	java.lang.Integer
11:	189855	7594200	java.lang.String
12:	173664	6946560	com.akvaplan.data.resultat.ResultatSimulering
13:	38218	5200024	<constMethodKlass>
14:	38218	4600928	<methodKlass>

图 4：程序执行 1 小时后 jmap 的输出结果，显示超过 2 千 5 百万个 CelleData 类实例

现在有超过 2 千 5 百万个 CelleData 类实例，占用了超过 1GB 内存！我们可以确认这是一个内存泄露。

这类数据信息的好处是，不仅非常有用而且对于很大的 JVM 堆也能快速反馈结果。我曾经试过检测一个运行频繁并且占用 17GB 堆内存的程序，使用 jmap 能够在 1 分钟内生成程序的性能统计图。

需要注意的是，jmap 不是运行分析工具，在生成统计图时 JVM 可能会暂停，因此当生成统计图时需要确认这种暂停对程序是可接受的。以我的经验，通常在调试一个严重 bug 时需要生成这种统计图，这种情况下，这些 1 分钟的暂停对程序来说是可接受的。这里，我们引出了下一个话题 - 半自动的运行分析工具 VisualVM。

VisualVM

另一个包含于 JVM 中的工具是 VisualVM，它的开发者将它描述为“一种集成了多个 JDK 命令行工具的可视化工具，它能为您提供轻量级的运行分析能力”。这样看来，VisualVM 是另一种你最有可能用到的事后分析工具，一般是错误已出现或性能问题已经用传统方法（客户抱怨大多属于此类）发现。

继续之前的示例程序和它严重的内存泄露问题，在程序执行 30 分钟后，VisualVM 帮我们绘制了如下图表：

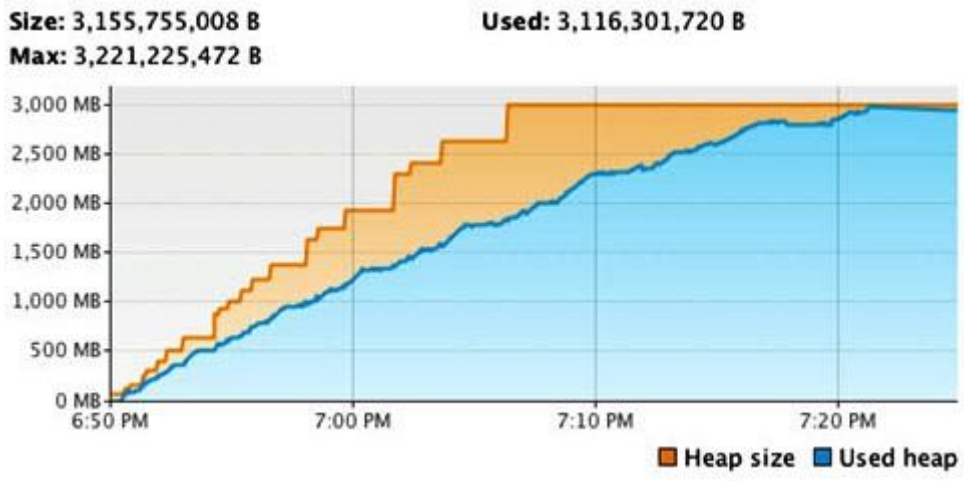


图 5：程序初始运行的 VisualVM 内存图

从这个图表，我们可以清晰地看到截止到 7:00pm，运行仅仅 10 分钟后，程序已经消耗掉超过 1GB 的堆空间。又过了 23 分钟，JVM 已经到了它启动参数–Xmx3g 最大值，导致程序响应缓慢，系统响应缓慢（持续的垃圾回收）和数量惊人的内存溢出错误。

借助 jmap，我们定位了这种内存消耗攀升的原因。修复后，我们让程序重新运行于 VisualVM 的严格监测之下，观察到下面的情况：

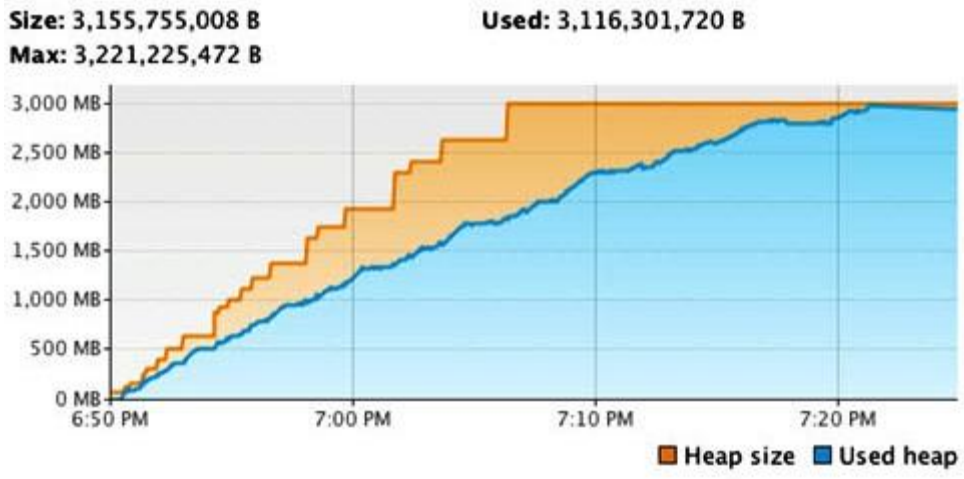


图 6：修复内存泄露问题后的 VisualVM 内存图

如你所见，程序的内存曲线（启动参数仍然为–Xmx3g）有了明显改善。

除了内存图像工具，VisualVM 还提供了一个采样器和一个轻量级的剖析器（Profiler）。

VisualVM 采样器能周期采样程序 CPU 和内存的使用情况。得到的统计数据类似 jmap 的反馈，此外，你还可以通过采样得到方法调用对 CPU 的占用情况。它让你能快速了解周期采样过程中的方法执行次数：

图 7：VisualVM 方法执行时间表

VisualVM 剖析器无需对程序周期采样就可以提供类似采样器的反馈信息，它还可以收集程序在整个正常执行过程中的统计数据（通过操纵程序源代码的字节码）。从剖析器得到的这种统计数据比从采样器而来的更精确和实时。

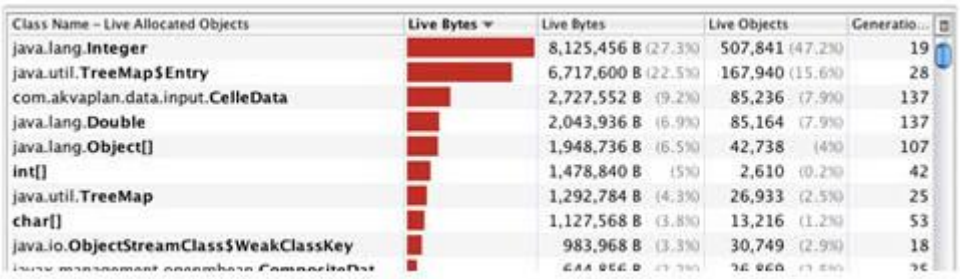


图 8：VisualVM 剖析器的输出

但是，你必须考虑的另一方面是该剖析器属于一种“暴力”分析工具。它的检测方法本质上是重新定义程序执行中的大多数类和方法，结果必然会明显减缓程序执行速度。例如，上述程序运

行部分的常规分析，大约要 35 秒。开启 VisualVM 的内存剖析器后，导致程序完成相同分析要 31 分钟。

我们需要清楚的是 VisualVM 并非功能齐全的剖析器。它无法在你的产品 JVM 上持续运行，不会保存分析数据，无法指定阈值，也不会在超过阈值时发出警报。要想更多的了解功能齐全的剖析器的目标。下面，让我们看看 BTrace，这个功能齐全的开源 java 代理程序。

BTrace

想象一下，如果能收集 JVM 当前的任何信息，那么你感兴趣的信息有哪些？我猜想问题列表会将因人而异，因情形而异。就个人来说，我通常感兴趣的是以下的问题：

- 程序对堆、非堆、永久保存区（Permanent Generation），以及 JVM 包含的不同内存池（新生对象区、长期对象区、存活空间等）的内存使用情况；
- 当前程序的线程数量，以及哪种类型线程正在被使用（单独计数）；
- JVM 的 CUP 负载；
- 系统平均负载/系统 CPU 使用总和；
- 对程序中的某些类和方法，我需要了解它们被调用次数，各自平均执行时间和整体平均时间；
- 对 SQL 调用的调用计数及执行次数；
- 对硬盘和网络操作的调用计数及执行次数。

利用 BTrace 可以采集到所有以上信息，你可以使用 BTrace 脚本定义需要采集的数据。方便的是，BTrace 脚本就是普通 Java 类，包含一些特殊注解来定义 BTrace 在什么地方及如何跟踪你的程序。BTrace 脚本会被 BTrace 编译器-btracec 编译成标准的.class 文件。

BTrace 脚本包含许多部分，正如下图所示。如果需要了解下图脚本的详细内容，请点击该链接或访问 BTrace 项目网站。

由于 BTrace 仅仅是一个代理，记录结果后，它的任务就算完成了。除了文本输出，BTrace 并不具备动态展现被收集信息的功能。缺省情况下，BTrace 脚本输出结果将在 btrace.class 文件所在位置生成一个名为 BTrace 脚本名.class.btrace 的 text 文件。

我们可以通过给 BTrace 设置一个额外参数，让它按某时间间隔循环记录日志。切记，它最多能在 100 个日志文件间循环，当达到*.class.btrace.99，它将覆盖*.class.btrace.00 文件。若让循环间隔在一个合理数字（如，每 7.5 秒）内，你就有充足时间来处理这些输出。只要在 Java 代理的输入参数中加上 fileRollMilliseconds=7500，就可以实现日志循环。

BTrace 一大缺点是它比较原始，难以定义它的输出格式。你也许非常希望有一种更好的方式来处理 BTrace 的输出和数据，比如可以用一种一致的图形用户界面来展示。你可能还需要比较不同时间点的数据和超出阈值能发送警告。一个新的开源工具 **EurekaJ**，就此应运而生。

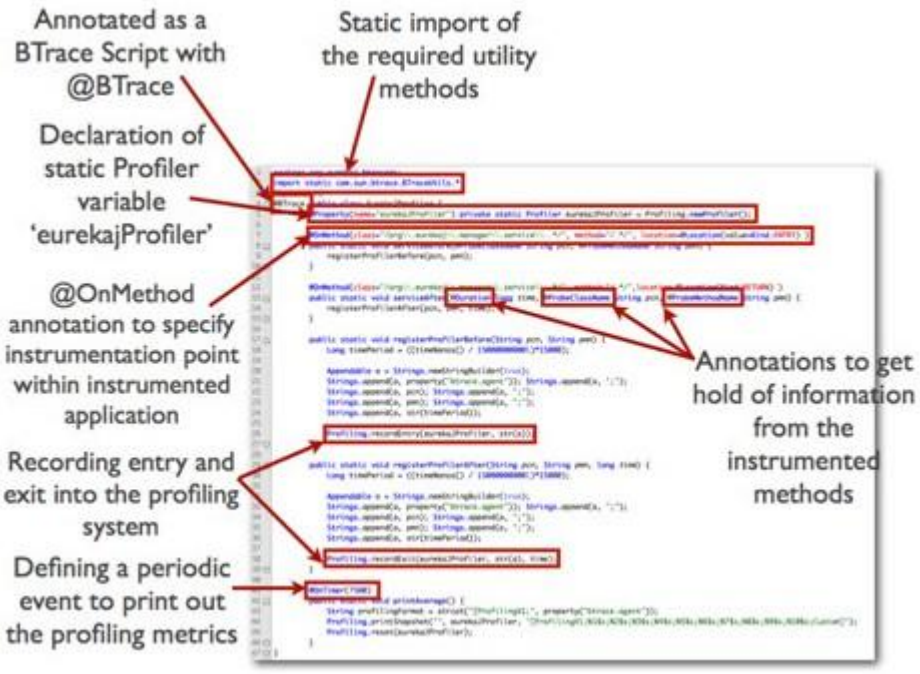


图 9：激活方法分析时必需的 BTrace 脚本

EurekaJ

我最初开发 **EurekaJ** 是在 2008 年。那时，我正在寻找一种具有我需要功能的开源剖析器，但没有找到。于是，我开始开发自己的工具。开发过程中，我涉猎了大量不同的技术并参考了许多架构模型，直到 EurekaJ 第一个版本发布。你可以从项目网站上了解更多的 EurekaJ 历史，查看源代码或下载并试着安装自己的版本。

EurekaJ 提供了两个主要应用：

1. 一个基于 Java 的管理器程序，可以接收传入的统计数据并一致地以可视化视图展现出来一个解析 BTrace 输出的代理程序，将其转化为 JSON 格式并输入到 EurekaJ 管理程序的 REST 接口。
2. EurekaJ 接受两种类型的输入数据格式。EurekaJ 代理期望 BTrace 脚本的输出被格式化为逗号分隔的文件（这点在 BTrace 中可很容易做到），而 EurekaJ 管理程序期望它的输入符合它的 JSON REST 接口格式。这意味着你能通过代理程序或直接经由 REST 接口来传递度量数据。

借助 EurekaJ 管理程序，我们可以在一张图上分组显示多个统计数据、可以定义阈值和给接收者发出警报。我们还可以方便的查看收集到的实时数据或历史数据。

所有收集到的数据排序成一种逻辑树结构，其结构由 BTrace 脚本作者指定。我建议 BTrace 脚本的作者对相关统计数据分组，这样，当它们显示在 EurekaJ 中时会更容易理解和观察。例如，我个人喜欢对统计数据进行如下的逻辑分组：

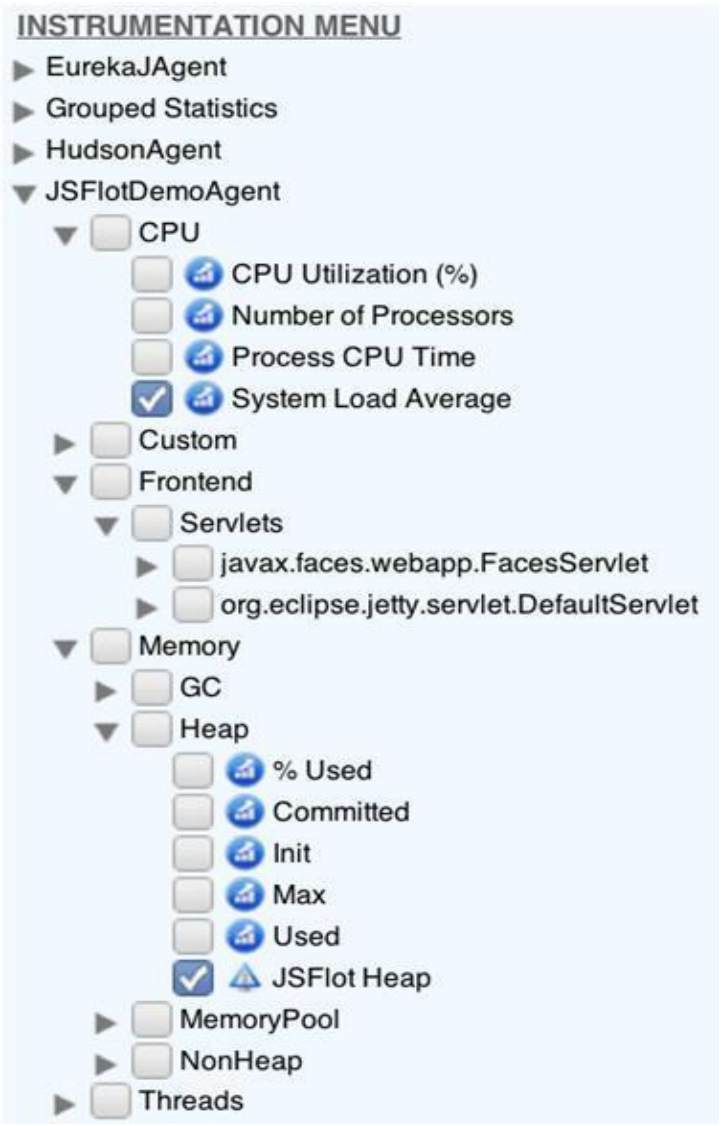


图 10：EurekaJ 演示程序的统计分组示例

图例

一种需要采集的重要信息是程序运行时的平均系统负载。要是你正面对一个运行缓慢的程序，那么缺陷可能并不在程序自身，而是隐藏到应用驻留的主机某处。我曾经在调试运行缓慢的应用时偶尔发现，真正的根源是病毒扫描程序。如果不进行测量分析，这种事情会很难被发现。考虑到这一点，我们需要能够在一张图中显示系统平均负载和进程加载后产生的负载。下图显示了一个运行 EurekaJ 演示程序的 Amazon EC2 虚拟服务器的 2 小时平均负载（该应用登录的用户名和密码都是‘user’）。

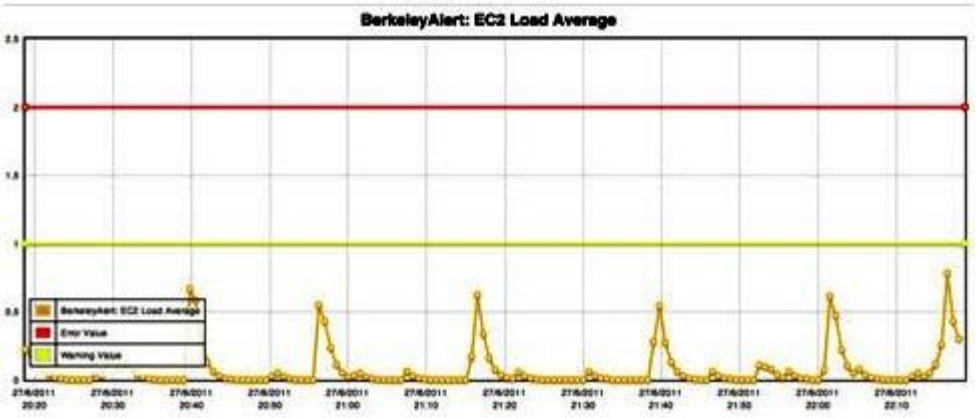


图 11：显示平均系统负载的 EurekaJ 图表

图中，黄色和红色的线条表示警戒阈值。一旦图形超过黄线的次数超过预设的最小警戒次数时，则测量结果到达“警告”状态。类似，若突破红线，测量结果就到达“危险”或“错误”状态。每当发生状态转换，EurekaJ 都会发送一封邮件给之前注册的收件人。

在上面的情形中，好像有周期性的事件每 20 分钟发生一次，从平均负载图上显示的波峰可以看到这一点。首先你要确定的是这个波峰确实由你的程序产生，而非其他原因。我们也可以通过测量进程的 CPU 负载来确认这点。在 EurekaJ 树菜单中，选择两个测量点后，两个图表结果会一起快速成像显示出来，其中一个位于另一个下面。

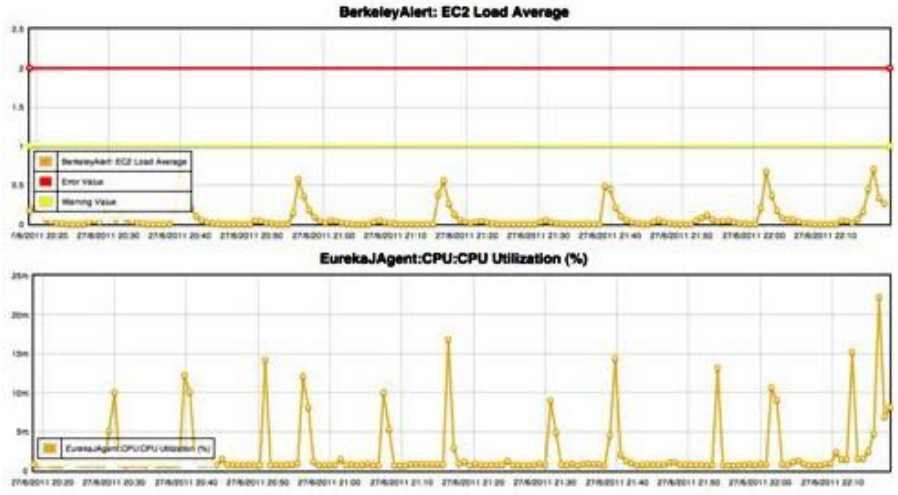


图 12：同时显示多个图表

在上面的例子中，我们清楚地看到进程 CUP 占用和系统负载存在必然的联系。

许多应用需要在程序无响应或不可用时及时发出警告。下图是一个 Confluence（Atlassian 的企业级 Wiki 软件）的例子。这个例子中，程序内存占用快速上升，直到产生程序内存溢出。这时，Confluence 无法处理接收到的请求，同时日志文件记录了各种奇怪的错误。

你可能希望当程序运行导致内存溢出时，程序能立刻抛出一个 OOME（内存溢出错误），然而，事实上 JVM 不会抛出 OOME 直到它发觉垃圾回收过于缓慢。结果，程序没有完全崩溃，又过了 2 小时，Java 仍然没有抛出 OutOfMemoryError，甚至两小时后程序依然在“运行”（意味着 JVM 进程仍然在运行）。显然，这时任何进程监测工具都不能发现程序已经“停止”。

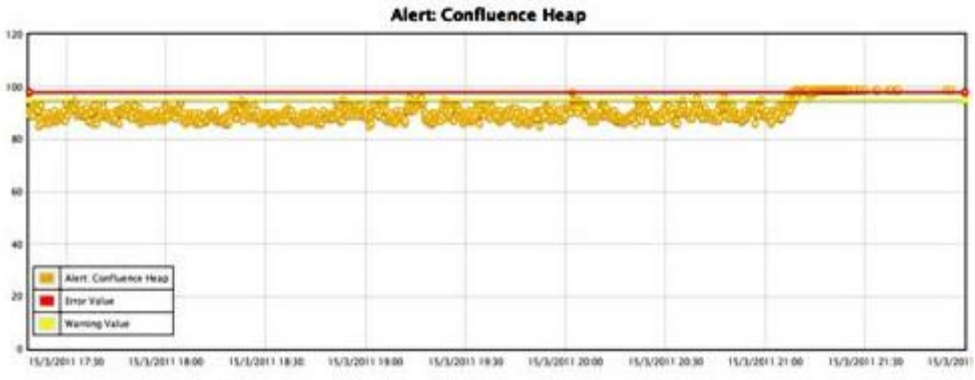


图 13：EurekaJ 堆图显示内存溢出错误的一种可能情形

注意最后几个小时的执行情况，图表揭示了下面的度量指标。

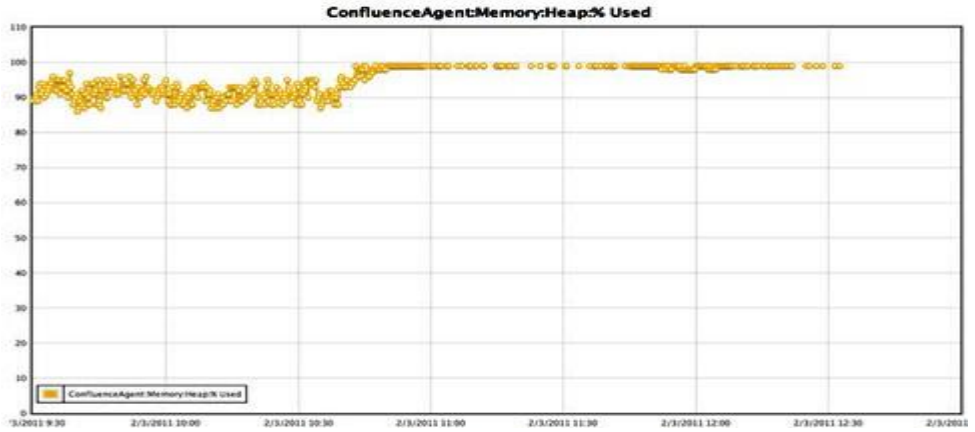


图 14：前面图表放大后的效果

EurekaJ 使我们可以设置程序的堆内存警告，个人建议最好如此。若程序持续占用堆内存超过 95%-98%（取决于堆的大小），几乎可以肯定，程序存在内存问题，要么用–Xmx 参数为程序分配更多的堆，要么优化程序使其使用更少内存。同时，EurekaJ 未来版本计划增加统计数据不足的警报。

最后的图表示例展示了一个包含 4 个不同程序内存使用的图表组。这种类型的图表组可方便用来比较一个程序不同部分的、或甚至不同程序之间、服务器之间的数据。下图的这 4 个程序有不同的内存需求和内存占用模式。

如下图示，不同程序有不同的内存曲线。这些曲线非常依赖一些实际情况，比如使用的架构、缓存数量、用户数、程序负载等。我希望通过下图说明你需要掌握程序在正常和高负载下执行情况的重要性，因为这将直接关系到如何定义报警阈值。

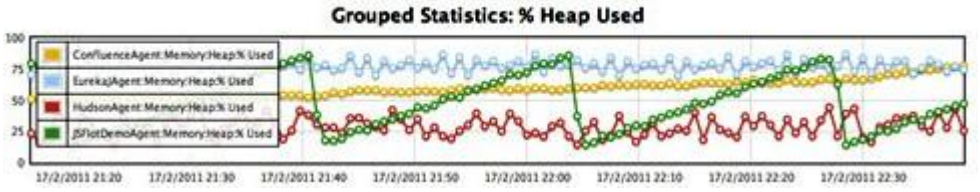


图 15：EurekaJ 图组会将图像彼此叠加在一起

注意性能干扰 – 让非热点区不受影响！

你使用的每一种测量方法似乎都会引起系统性能干扰。一些数据的测量可以被认为“无干扰”（或“忽略不计”），然而另外一些数据的测量可称得上代价昂贵。非常重要的一点是，要知道你需要 BTrace 测量什么。因此，你要将这种分析工具对程序的性能干扰减少到最小。关于这点，请参考下面的 3 条原则：

- 基于“采样”的度量通常可被认为“无影响”。采样 CPU 负载、进程 CPU 负载、内存使用和每 5-10 秒的线程计数，其带来的额外一两个毫秒的影响可被忽略。在我看来，你应该经常收集这类统计数据，它们对你来说不会有什么损耗。
- 对长时间运行的任务的测量也可被认为“无影响”。通常，它仅会对每个被测量方法带来 1700-2500 纳秒的影响。如果你正测量这些对象的执行时间：SQL 查询、网络流量、硬盘读写或一个预期范围在 40 毫秒（磁盘存取）到 1 秒（Servlet 处理）之间的 Servlet 处理过程，那么对这些对象每个增加额外的 2500 纳秒左右的时间也是可接受的。
- 绝对不要对循环内执行的方法进行测量。

寻找程序热点区的一个通用规则是不要影响非热点区域。例如，考虑下面的类：

```
1  package example;
2
3  public class MyDaoClass {
4      private List<Statistics> readStatFromDatabase(String statName, long msFrom, long msTo) throws SQLException {
5          String sql = "select * from Statistics s where s.StatisticName = ? and s.StatisticTimestamp between ? and ?";
6
7          PreparedStatement preparedStatement = derbyEnvironment.getConnection().prepareStatement(sql);
8          preparedStatement.setString(1, statName);
9          preparedStatement.setTimestamp(2, new Timestamp(msFrom));
10         preparedStatement.setTimestamp(3, new Timestamp(msTo));
11
12         List<Statistics> statList = new ArrayList<Statistics>();
13         ResultSet rs = preparedStatement.executeQuery();
14         while (rs.next()) {
15             statList.add(buildNewStat(rs));
16         }
17
18         return statList;
19     }
20
21     public Statistics buildNewStat(ResultSet rs) {
22         Statistics stat = new Statistics();
23         stat.setStatisticID(rs.getLong("StatisticID"));
24         stat.setStatisticName(rs.getString("StatisticName"));
25         stat.setStatisticValue(rs.getDouble("StatisticValue"));
26
27         return stat;
28     }
29 }
```

图 16：需要测量的简单的数据访问接口类

第 5 行的 SQL 执行时间可能使 readStatFromDatabase 方法可能成为一个热点。当查询返回相当多的数据行时，它无疑会成为一个热点，这对 13 行（程序和数据库服务器之间的网络流量）和 14-16 行（结果集中每行所需处理）会造成负面影响。如果从数据库返回结果时间过长，该方法也会成为一个热点（在 13 行）。

方法 buildNewStat 就其本身来说似乎绝不会成为一个热点。即使被多次执行，每次调用都会在几纳秒内完成。另一方面，若给每次调用增加了 2500 纳秒的测量采集干扰，则无论 SQL 何时被执行，都势必会让该方法看起来像个热点。因此，我们要避免测量它。

图 17：显示上面类哪些部分可以被测量，哪些需要避免

建立完整的运行分析

使用 EurekaJ 建立一个完整的运行分析，需要以下几个主要部分：

- 准备需要监测/操纵的程序；
- BTrace - java 代理；
- 告知 BTrace 如何测量的 BTrace 脚本；
- 存储 BTrace 输出的文件系统；
- 将 BTrace 输出传输到 EurekaJ 管理器的 EurekaJ 代理；
- 安装好的 EurekaJ 管理器（本地安装或可通过互联网访问的远程安装）。

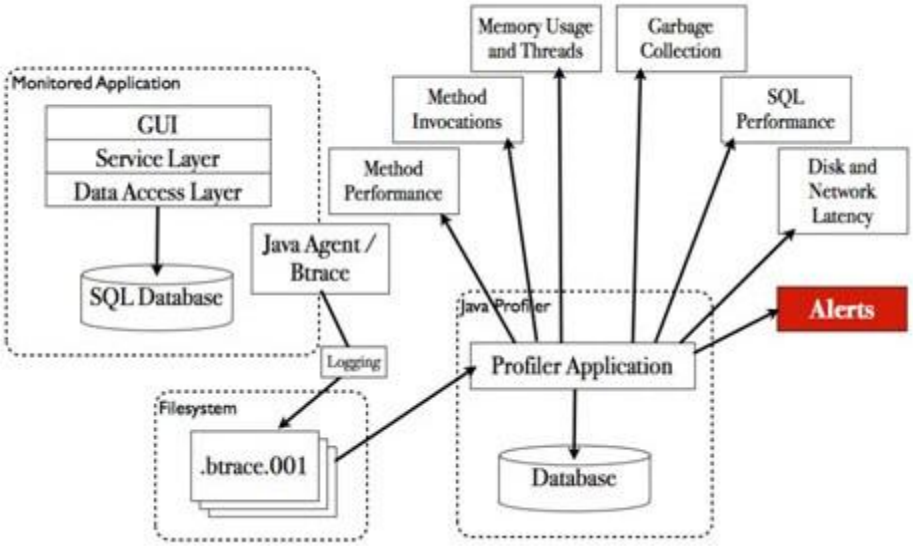


图 18：使用本文所描述工具对程序进行运行分析的概览图

关于这些产品的完整安装手册，请访问 EurekaJ 项目网站。

总结

这篇文章给我们介绍了一些用于程序运行分析的开源工具，它们不仅能帮我们完成对运行中 JVM 的深度分析，而且可以帮助我们对开发、测试和程序部署进行多方位的持续监测。

希望你已经开始了解不断收集度量信息的好处和超过阈值后及时报警能力的重要性。

非常感谢！

参考

[1] jmap 文档
[2] BTrace 脚本概念
[3] BTrace 项目网站
[4] EurekaJ 文档
[5] EurekaJ 项目网站

关于作者

Joachim Haagen Skeie，挪威奥斯陆 Kantega AS 公司的 Java 和 Web 技术的资深顾问，关注程序性能分析和开源软件。你可以通过他的 Twitter 账号来联系。

双重检查锁定与延迟初始化

作者 程晓明

在 Java 程序中，有时候可能需要推迟一些高开销的对象初始化操作，并且只有在使用这些对象时才进行初始化。此时程序员可能会采用延迟初始化。但要正确实现线程安全的延迟初始化需要一些技巧，否则很容易出现问题。比如，下面是非线程安全的延迟初始化对象的示例代码：

```
public class UnsafeLazyInitialization {

private static Instance instance;

public static Instance getInstance() {

    if (instance == null) //1：A 线程执行

    instance = new Instance(); //2：B 线程执行

    return instance;

   }

}
```

在 UnsafeLazyInitialization 中，假设 A 线程执行代码 1 的同时，B 线程执行代码 2。此时，线程 A 可能会看到 instance 引用的对象还没有完成初始化（出现这种情况的原因见后文的“问题的根源”）。

对于 UnsafeLazyInitialization，我们可以对 getInstance()做同步处理来实现线程安全的延迟初始化。示例代码如下：

```
public class SafeLazyInitialization {

    private static Instance instance;



    public synchronized static Instance getInstance() {

        if (instance == null)

            instance = new Instance();

        return instance;

    }

}
```

由于对 getInstance()做了同步处理，synchronized 将导致性能开销。如果 getInstance()被多个线程频繁的调用，将会导致程序执行性能的下降。反之，如果 getInstance()不会被多个线程频繁的调用，那么这个延迟初始化方案将能提供令人满意的性能。

在早期的 JVM 中，synchronized（甚至是无竞争的 synchronized）存在这巨大的性能开销。因此，人们想出了一个“聪明”的技巧：双重检查锁定（double-checked locking）。人们想通过双重检查锁定来降低同步的开销。下面是使用双重检查锁定来实现延迟初始化的示例代码：

```
public class DoubleCheckedLocking {                  //1

    private static Instance instance;                       //2



    public static Instance getInstance() {                  //3

        if (instance == null) {                             //4:第一次检查

            synchronized (DoubleCheckedLocking.class) {  //5:加锁

                if (instance == null)                       //6:第二次检查

                    instance = new Instance();              //7:问题的根源出在这里

            }                                               //8

        }                                                   //9

        return instance;                                    //10

    }                                                       //11

}                                                           //12
```

如上面代码所示，如果第一次检查 instance 不为 null，那么就不需要执行下面的加锁和初始化操作。因此可以大幅降低 synchronized 带来的性能开销。上面代码表面上看起来，似乎两全其美：

- 在多个线程试图在同一时间创建对象时，会通过加锁来保证只有一个线程能创建对象。
- 在对象创建好之后，执行 getInstance()将不需要获取锁，直接返回已创建好的对象。

双重检查锁定看起来似乎很完美，但这是一个错误的优化！在线程执行到第 4 行代码读取到 instance 不为 null 时，instance 引用的对象有可能还没有完成初始化。

问题的根源

前面的双重检查锁定示例代码的第 7 行（instance = new Singleton();）创建一个对象。这一行代码可以分解为如下的三行伪代码：

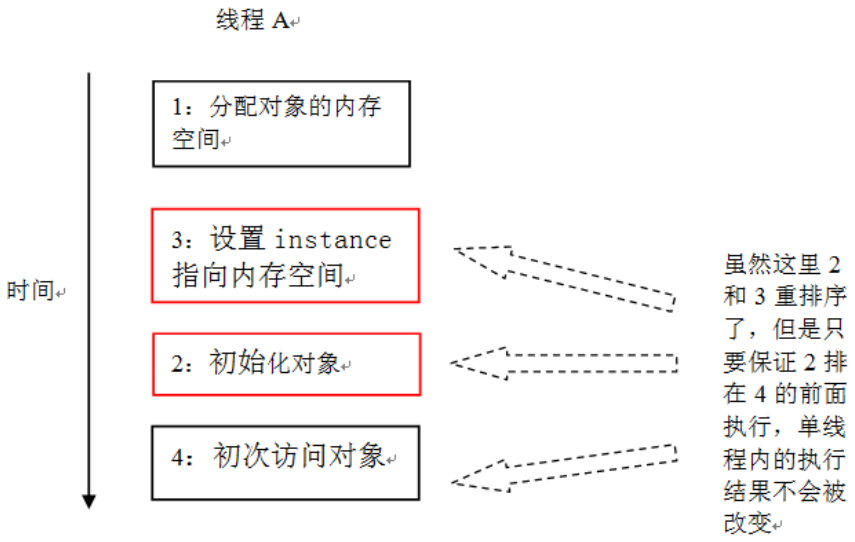
```
memory = allocate();    //1：分配对象的内存空间

ctorInstance(memory);   //2：初始化对象

instance = memory;      //3：设置 instance 指向刚分配的内存地址
```

上面三行伪代码中的 2 和 3 之间，可能会被重排序（在一些 JIT 编译器上，这种重排序是真实发生的，详情见参考文献 1 的“Out-of-order writes”部分）。2 和 3 之间重排序之后的执行时序如下：

```
memory = allocate();    //1：分配对象的内存空间
instance = memory;      //3：设置 instance 指向刚分配的内存地址
                        //注意，此时对象还没有被初始化！
ctorInstance(memory);   //2：初始化对象
```

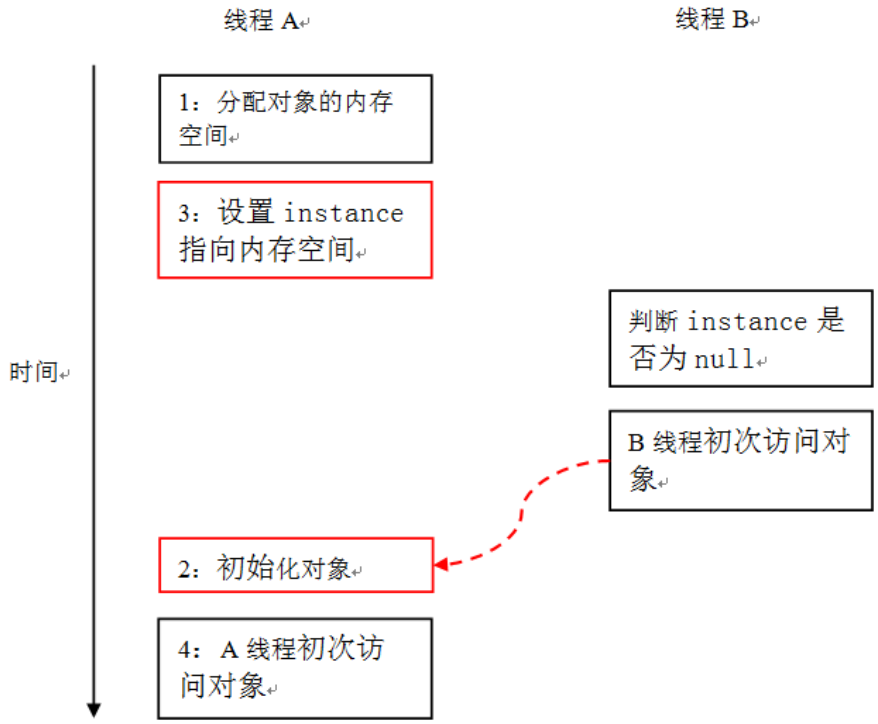
根据 *The Java Language Specification, Java SE 7 Edition*（后文简称为 Java 语言规范），所有线程在执行 Java 程序时必须要遵守 intra-thread semantics。intra-thread semantics 保证重排序不会改变单线程内的程序执行结果。换句话来说，intra-thread semantics 允许那些在单线程内，不会改变单线程程序执行结果的重排序。上面三行伪代码的 2 和 3 之间虽然被重排序了，但这个重排序并不会违反 intra-thread semantics。这个重排序在没有改变单线程程序的执行结果的前提下，可以提高程序的执行性能。

为了更好的理解 intra-thread semantics，请看下面的示意图（假设一个线程 A 在构造对象后，立即访问这个对象）：



如上图所示，只要保证 2 排在 4 的前面，即使 2 和 3 之间重排序了，也不会违反 intra-thread semantics。

下面，再让我们看看多线程并发执行的时候的情况。请看下面的示意图：



由于单线程内要遵守 intra-thread semantics，从而能保证 A 线程的程序执行结果不会被改变。但是当线程 A 和 B 按上图的时序执行时，B 线程将看到一个还没有被初始化的对象。

※注：本文统一用红色的虚箭线标识错误的读操作，用绿色的虚箭线标识正确的读操作。

回到本文的主题，DoubleCheckedLocking 示例代码的第 7 行（instance = new Singleton();）如果发生重排序，另一个并发执行的线程 B 就有可能在第 4 行判断 instance 不为 null。线程 B 接下来将访问 instance 所引用的对象，但此时这个对象可能还没有被 A 线程初始化！下面是这个场景的具体执行时序：

时间	线程 A	线程 B
t1	A1：分配对象的内存空间	
t2	A3：设置 instance 指向内存空间	
t3		B1：判断 instance 是否为空
t4		B2：由于 instance 不为 null，线程 B 将访问 instance 引用的对象
t5	A2：初始化对象	
t6	A4：访问 instance 引用的对象	

这里 A2 和 A3 虽然重排序了，但 Java 内存模型的 intra-thread semantics 将确保 A2 一定会排在 A4 前面执行。因此线程 A 的 intra-thread semantics 没有改变。但 A2 和 A3 的重排序，将导致线程 B 在 B1 处判断出 instance 不为空，线程 B 接下来将访问 instance 引用的对象。此时，线程 B 将会访问到一个还未初始化的对象。

在知晓了问题发生的根源之后，我们可以想出两个办法来实现线程安全的延迟初始化：

1. 不允许 2 和 3 重排序；
2. 允许 2 和 3 重排序，但不允许其他线程“看到”这个重排序。

后文介绍的两个解决方案，分别对应于上面这两点。

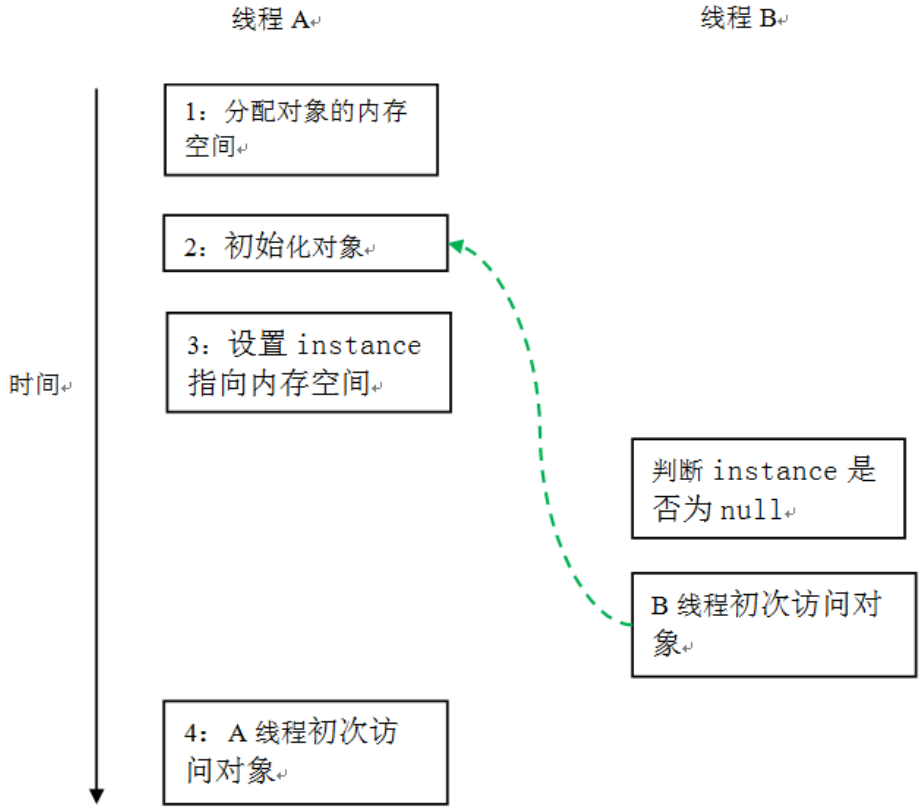
基于 volatile 的双重检查锁定的解决方案

对于前面的基于双重检查锁定来实现延迟初始化的方案（指 DoubleCheckedLocking 示例代码），我们只需要做一点小的修改（把 instance 声明为 volatile 型），就可以实现线程安全的延迟初始化。请看下面的示例代码：

```
public class SafeDoubleCheckedLocking {

    private volatile static Instance instance;



    public static Instance getInstance() {

        if (instance == null) {

            synchronized (SafeDoubleCheckedLocking.class) {

                if (instance == null)

                    instance = new Instance();//instance 为 volatile，现在没问题了

            }

        }

        return instance;

    }

}
```

注意，这个解决方案需要 JDK5 或更高版本（因为从 JDK5 开始使用新的 JSR-133 内存模型规范，这个规范增强了 volatile 的语义）。

当声明对象的引用为 volatile 后，“问题的根源”的三行伪代码中的 2 和 3 之间的重排序，在多线程环境中将会被禁止。上面示例代码将按如下的时序执行：



这个方案本质上是通过禁止上图中的 2 和 3 之间的重排序，来保证线程安全的延迟初始化。

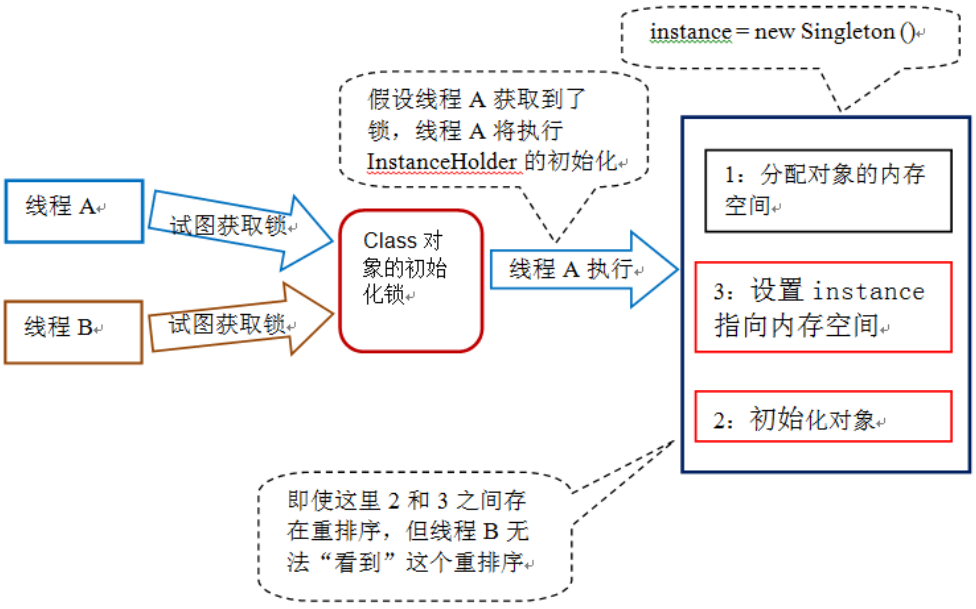
基于类初始化的解决方案

JVM 在类的初始化阶段（即在 Class 被加载后，且被线程使用之前），会执行类的初始化。在执行类的初始化期间，JVM 会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

基于这个特性，可以实现另一种线程安全的延迟初始化方案（这个方案被称之为 Initialization On Demand Holder idiom）：

```
public class InstanceFactory {

    private static class InstanceHolder {

        public static Instance instance = new Instance();

    }

    public static Instance getInstance() {

        return InstanceHolder.instance ;  //这里将导致 InstanceHolder 类被初始化

}

}
```

假设两个线程并发执行 getInstance()，下面是执行的示意图：



这个方案的实质是：允许“问题的根源”的三行伪代码中的 2 和 3 重排序，但不允许非构造线程（这里指线程 B）“看到”这个重排序。

初始化一个类，包括执行这个类的静态初始化和初始化在这个类中声明的静态字段。根据 java 语言规范，在首次发生下列任意一种情况时，一个类或接口类型 T 将被立即初始化：

- T 是一个类，而且一个 T 类型的实例被创建；
- T 是一个类，且 T 中声明的一个静态方法被调用；
- T 中声明的一个静态字段被赋值；
- T 中声明的一个静态字段被使用，而且这个字段不是一个常量字段；
- T 是一个顶级类（top level class，见 Java 语言规范的§7.6），而且一个断言语句嵌套在 T 内部被执行。

在 InstanceFactory 示例代码中，首次执行 getInstance()的线程将导致 InstanceHolder 类被初始化（符合情况 4）。

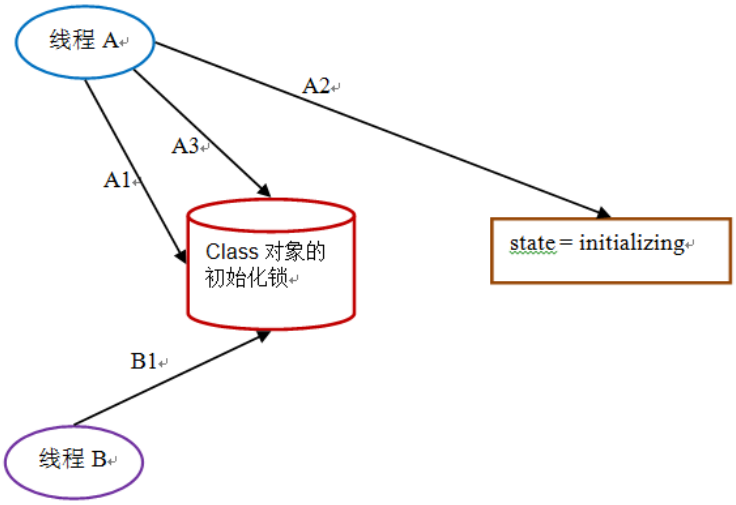
由于 Java 语言是多线程的，多个线程可能在同一时间尝试去初始化同一个类或接口（比如这里多个线程可能在同一时刻调用 getInstance()来初始化 InstanceHolder 类）。因此在 Java 中初始化一个类或者接口时，需要做细致的同步处理。

Java 语言规范规定，对于每一个类或接口 C，都有一个唯一的初始化锁 LC 与之对应。从 C 到 LC 的映射，由 JVM 的具体实现去自由实现。JVM 在类初始化期间会获取这个初始化锁，并且每个线程至少获取一次锁来确保这个类已经被初始化过了（事实上，Java 语言规范允许 JVM 的具体实现在这里做一些优化，见后文的说明）。

对于类或接口的初始化，Java 语言规范制定了精巧而复杂的类初始化处理过程。Java 初始化一个类或接口的处理过程如下（这里对类初始化处理过程的说明，省略了与本文无关的部分；同时为了更好的说明类初始化过程中的同步处理机制，笔者人为的把类初始化的处理过程分为了五个阶段）：

第一阶段：通过在 Class 对象上同步（即获取 Class 对象的初始化锁），来控制类或接口的初始化。这个获取锁的线程会一直等待，直到当前线程能够获取到这个初始化锁。

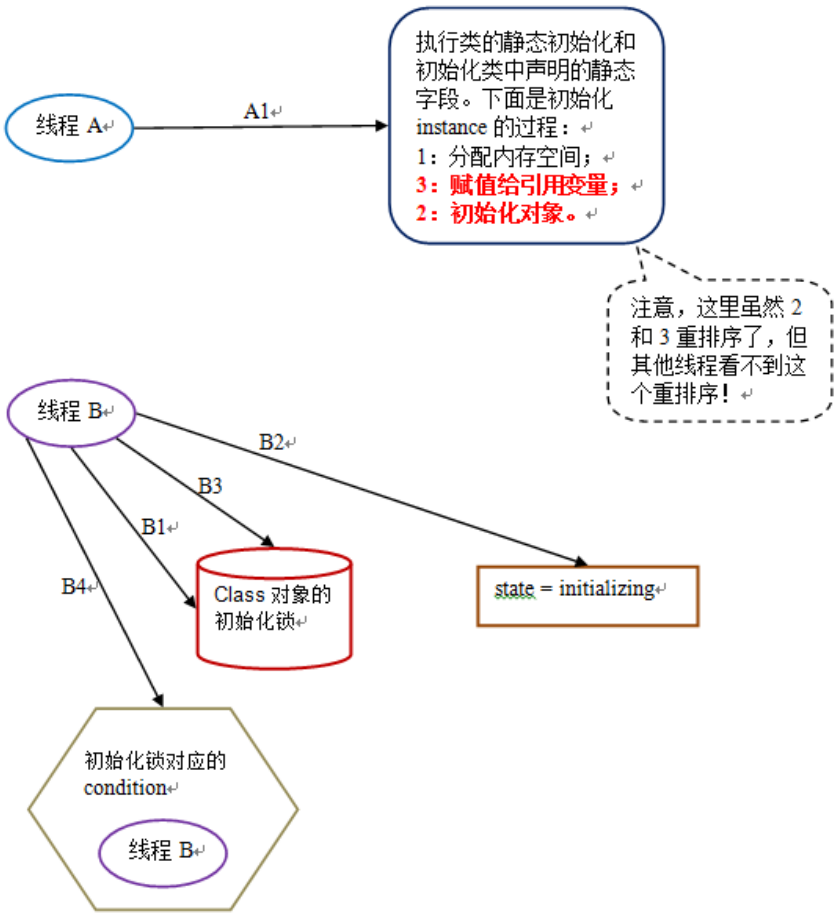
假设 Class 对象当前还没有被初始化（初始化状态 state 此时被标记为 state = noInitialization），且有两个线程 A 和 B 试图同时初始化这个 Class 对象。下面是对应的示意图：



下面是这个示意图的说明：

时间	线程 A	线程 B
t1	A1:尝试获取 Class 对象的初始化锁。这里假设线程 A 获取到了初始化锁	B1:尝试获取 Class 对象的初始化锁，由于线程 A 获取到了锁，线程 B 将一直等待获取初始化锁
t2	A2：线程 A 看到线程还未被初始化（因为读取到 state == noInitialization），线程设置 state = initializing	
t3	A3：线程 A 释放初始化锁	

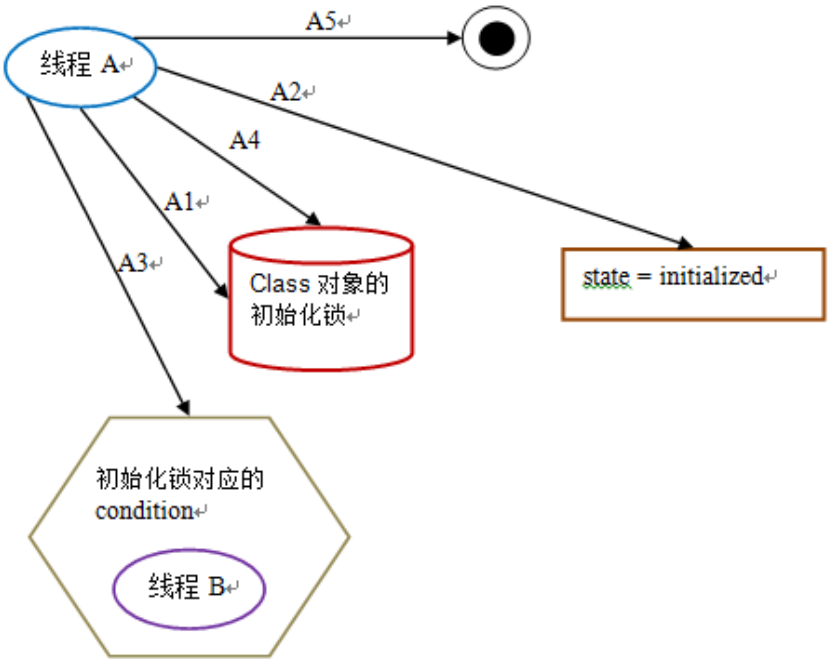
第二阶段：线程 A 执行类的初始化，同时线程 B 在初始化锁对应的 condition 上等待：



下面是这个示意图的说明：

时间	线程 A	线程 B
t1	A1:执行类的静态初始化和初始化类中声明的静态字段	B1：获取到初始化锁
t2		B2：读取到 state == initializing
t3		B3：释放初始化锁
t4		B4：在初始化锁的 condition 中等待

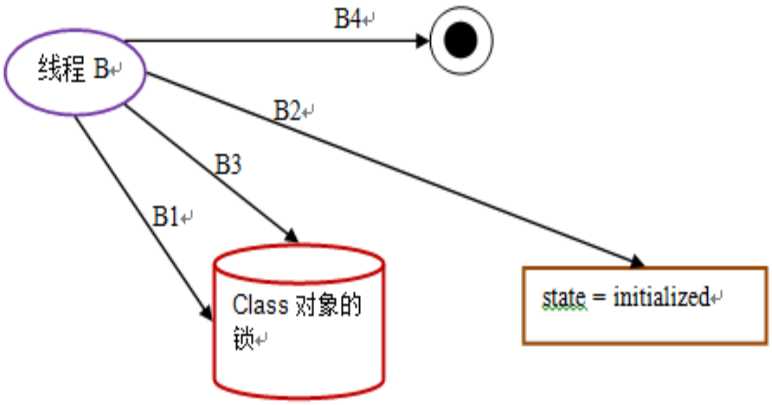
第三阶段：线程 A 设置 state = initialized，然后唤醒在 condition 中等待的所有线程：



下面是这个示意图的说明：

时间	线程 A
t1	A1：获取初始化锁
t2	A2：设置 state = initialized
t3	A3：唤醒在 condition 中等待的所有线程
t4	A4：释放初始化锁
t5	A5：线程 A 的初始化处理过程完成

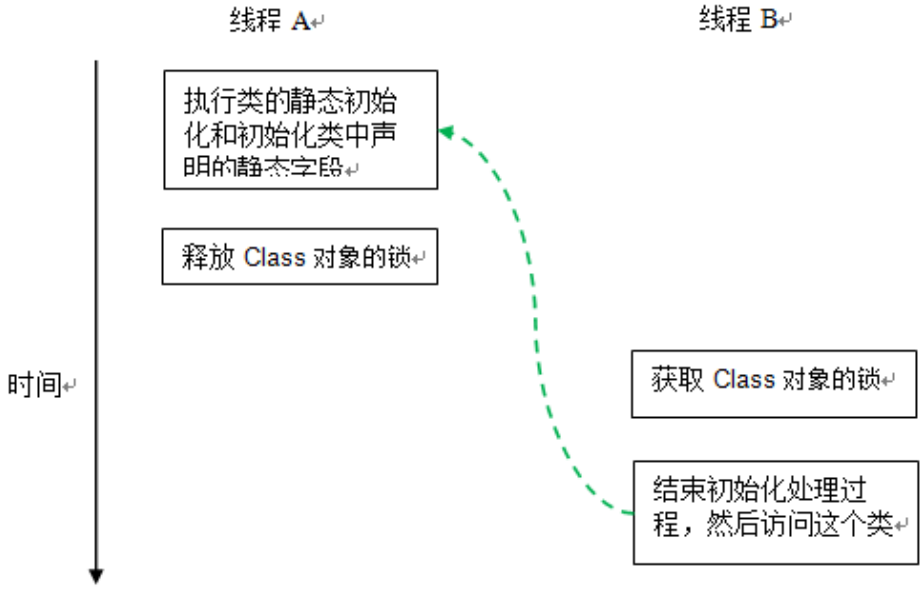
第四阶段：线程 B 结束类的初始化处理：



下面是这个示意图的说明：

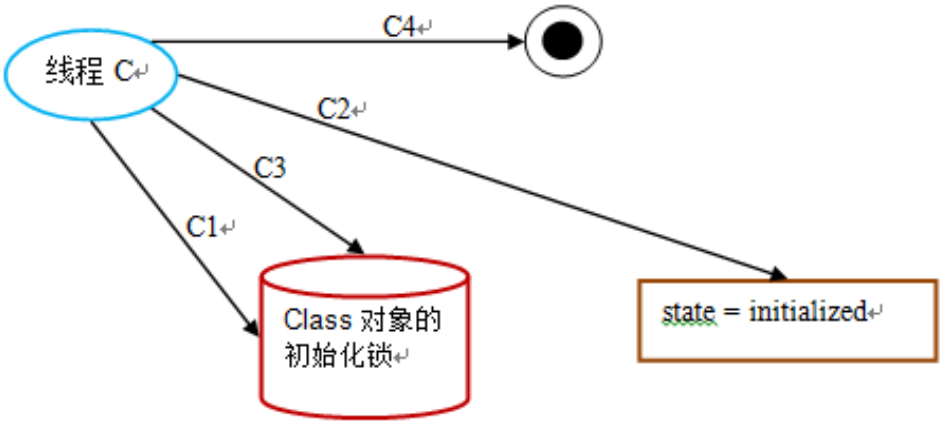
时间	线程 B
t1	B1：获取初始化锁
t2	B2：读取到 state == initialized
t3	B3：释放初始化锁
t4	B4：线程 B 的类初始化处理过程完成

线程 A 在第二阶段的 A1 执行类的初始化，并在第三阶段的 A4 释放初始化锁；线程 B 在第四阶段的 B1 获取同一个初始化锁，并在第四阶段的 B4 之后才开始访问这个类。根据 java 内存模型规范的锁规则，这里将存在如下的 happens-before 关系：



这个 happens-before 关系将保证：线程 A 执行类的初始化时的写入操作（执行类的静态初始化和初始化类中声明的静态字段），线程 B 一定能看到。

第五阶段：线程 C 执行类的初始化的处理：

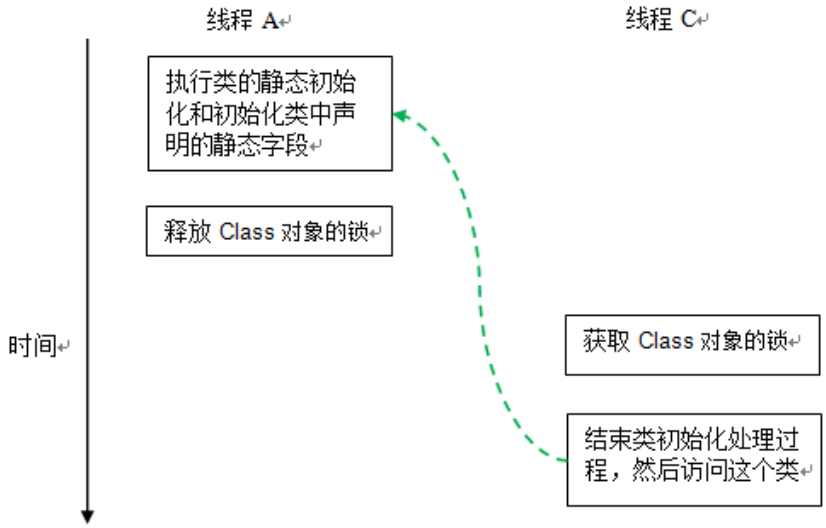


下面是这个示意图的说明：

时间	线程 B
t1	C1：获取初始化锁
t2	C2：读取到 state == initialized
t3	C3：释放初始化锁
t4	C4：线程 C 的类初始化处理过程完成

在第三阶段之后，类已经完成了初始化。因此线程 C 在第五阶段的类初始化处理过程相对简单一些（前面的线程 A 和 B 的类初始化处理过程都经历了两次锁获取-锁释放，而线程 C 的类初始化处理只需要经历一次锁获取-锁释放）。

线程 A 在第二阶段的 A1 执行类的初始化，并在第三阶段的 A4 释放锁；线程 C 在第五阶段的 C1 获取同一个锁，并在在第五阶段的 C4 之后才开始访问这个类。根据 Java 内存模型规范的锁规则，这里将存在如下的 happens-before 关系：



这个 happens-before 关系将保证：线程 A 执行类的初始化时的写入操作，线程 C 一定能看到。

※注 1：这里的 condition 和 state 标记是本文虚构出来的。Java 语言规范并没有硬性规定一定要使用 condition 和 state 标记。JVM 的具体实现只要实现类似功能即可。

※注 2：Java 语言规范允许 Java 的具体实现，优化类的初始化处理过程（对这里的第五阶段做优化），具体细节参见 Java 语言规范的 12.4.2 章。

通过对比基于 volatile 的双重检查锁定的方案和基于类初始化的方案，我们会发现基于类初始化的方案的实现代码更简洁。但基于 volatile 的双重检查锁定的方案有一个额外的优势：除了可以对静态字段实现延迟初始化外，还可以对实例字段实现延迟初始化。

总结

延迟初始化降低了初始化类或创建实例的开销，但增加了访问被延迟初始化的字段的开销。在大多数时候，正常的初始化要优于延迟初始化。如果确实需要对实例字段使用线程安全的延迟初始化，请使用上面介绍的基于 volatile 的延迟初始化的方案；如果确实需要对静态字段使用线程安全的延迟初始化，请使用上面介绍的基于类初始化的方案。

参考文献

- Double-checked locking and the Singleton pattern
- The Java Language Specification, Java SE 7 Edition
- JSR-133: Java Memory Model and Thread Specification
- Java Concurrency in Practice
- Effective Java (2nd Edition)
- JSR 133 (Java Memory Model) FAQ
- The JSR-133 Cookbook for Compiler Writers
- Java theory and practice: Fixing the Java Memory Model, Part 2

作者简介

程晓明，Java 软件工程师，专注于并发编程，现就职于富士通南大。个人邮箱：asst2003@163.com。

Gradle 在大型 Java 项目上的应用

作者 何海洋

在 Java 构建工具的世界里，先有了 Ant，然后有了 Maven。Maven 的 CoC[1]、依赖管理以及项目构建规则重用性等特点，让 Maven 几乎成为 Java 构建工具的事实标准。然而，冗余的依赖管理配置、复杂并且难以扩展的构建生命周期，都成为使用 Maven 的困扰。

Gradle 作为新的构建工具，获得了 2010 Springy 大奖，并入围了 2011 的 Jax 最佳 Java 技术发明奖。它是基于 Groovy 语言的构建工具，既保持了 Maven 的优点，又通过使用 Groovy 定义的 DSL[2]，克服了 Maven 中使用 XML 繁冗以及不灵活等缺点。在 Eugene Dvorkin 撰写的文章《最让人激动的 5 个 Java 项目》中，他是这样介绍 Gradle 的：

“工程自动化是软件项目成功的必要条件，而且它应该实现起来简单、易用、好玩。构建没有千篇一律的方法，所以 Gradle 没有死板的强加方法于我们，尽管你会认为查找和描述方法很重要，然而 Gradle 对于如何描述有着非常好的支持。我不认为工具能够拯救我们，但是 Gradle 能给你所需要的自由，你可以利用 Gradle 构建易描述的、可维护的、简洁的、高性能项目”。

在最近半年里，我在使用 Gradle 作为构建脚本的大型 Java 项目上工作，更深切体会到 Gradle 在项目构建过程中是如此的简单、易用。

1. 多 Module 的项目

Hibernate 项目负责人 Steve Ebersole 在 Hibernate 将构建脚本从 Maven 换成 Gradle 时，专门写了一篇文章“Gradle: why?”，文中提到 Maven 的一个缺点就是：Maven 不支持多 module 的构建。在 Micro-Service[3]架构风格流行的今天，在一个项目里面包含多个 Module 已成为一种趋势。Gradle 天然支持多 module，并且提供了很多手段来简化构建脚本。在 Gradle 中，一个模块就是它的一个子项目（subproject），所以，我使用父项目来描述顶级项目，使用子项目来描述顶级项目下面的模块。

1.1 配置子项目

在多模块的项目中，Gradle 遵循惯例优于配置 （Convention Over Configuration）原则。

在父项目的根目录下寻找 settings.gradle 文件，在该文件中设置想要包括到项目构建中的子项目。在构建的初始化阶段（Initialization），Gradle 会根据 settings.gradle 文件来判断有哪些子项目被 include 到了构建中，并为每一个子项目初始化一个 Project 对象，在构建脚本中通过 project(‘:sub-project-name’)来引用子项目对应的 Project 对象。

通常，多模块项目的目录结构要求将子模块放在父项目的根目录下，但是如果有特殊的目录结构，可以在 settings.gradle 文件中配置。

我所在的项目包括：

- 一个描述核心业务的 core 模块
- 一个遗留的 Enterprise Java Bean（enterprise-beans）模块

- 两个提供不同服务的 Web 项目（cis-war 和 admin-war）
- 一个通过 schema 生成 jaxb 对象的 jaxb 项目以及一个用来用来打 ear 包的 ear 项目
- 一个用于存放项目配置文件相关的 config 子目录。它不是子模块，所以 config 不应该被加到项目的构建中去。

它们都放置在根项目目录下。我们通过如下的 settings.gradle 来设置项目中的子项目：

```
include 'core', 'enterprise-beans', 'cis-war', 'admin-war', 'jaxb', 'ear'
```

我们将需要加入到项目构建中的子项目配置在 settings.gradle 文件中，而没有加入不需要的 config 子目录。

1.2 共享配置

在大型 Java 项目中，子项目之间必然具有相同的配置项。我们在编写代码时，要追求代码重用和代码整洁；而在编写 Gradle 脚本时，同样需要保持代码重用和代码整洁。Gradle 提供了不同的方式使不同的项目能够共享配置。

- allprojects：allprojects 是父 Project 的一个属性，该属性会返回该 Project 对象以及其所有子项目。在父项目的 build.gradle 脚本里，可以通过给 allprojects 传一个包含配置信息的闭包，来配置所有项目（包括父项目）的共同设置。通常可以在这里配置 IDE 的插件，group 和 version 等信息，比如：

```
allprojects {

    apply plugin: 'idea'

    }
```

这样就会给所有的项目（包括当前项目以及其子项目）应用上 idea 插件。

- subprojects：subprojects 和 allprojects 一样，也是父 Project 的一个属性，该属性会返回所有子项目。在父项目的 build.gradle 脚本里，给 subprojects 传一个包含配置信息的闭包，可以配置所有子项目共有的设置，比如共同的插件、repositories、依赖版本以及依赖配置：

```
subprojects {

    apply plugin: 'java'

    repositories {

        mavenCentral()

    }

    ext {

          guavaVersion = ’14.0.1’

          junitVersion = ‘4.10’

   }

    dependencies {
```

```
        compile(
                "com.google.guava:guava:${guavaVersion}"
        )
        testCompile(
                "junit:junit:${junitVersion}"
        )
    }
}
```

这就会给所有子项目设置上 java 的插件、使用 mavenCentral 作为 所有子项目的 repository 以及对 Guava[4]和 JUnit 的项目依赖。此外，这里还在 ext 里配置依赖包的版本，方便以后升级依赖的版本。

- configure：在项目中，并不是所有的子项目都会具有相同的配置，但是会有部分子项目具有相同的配置，比如在我所在的项目里除了 cis-war 和 admin-war 是 web 项目之外，其他子项目都不是。所以需要给这两个子项目添加 war 插件。Gradle 的 configure 可以传入子项目数组，并为这些子项目设置相关配置。在我的项目中使用如下的配置：

```
configure(subprojects.findAll {it.name.contains('war')}) {
    apply plugin: 'war'
    }
```

configure 需要传入一个 Project 对象的数组，通过查找所有项目名包含 war 的子项目，并为其设置 war 插件。

1.3 独享配置

在项目中，除了设置共同配置之外， 每个子项目还会有其独有的配置。比如每个子项目具有不同的依赖以及每个子项目特殊的 task 等。Gradle 提供了两种方式来分别为每个子项目设置独有的配置。

- 在父项目的 build.gradle 文件中通过 project(':sub-project-name')来设置对应的子项目的配置。比如在子项目 core 需要 Hibernate 的依赖，可以在父项目的 build.gradle 文件中添加如下的配置：

```
project(':core') {
      ext{
                hibernateVersion = '4.2.1.Final'
      }
    dependencies {
          compile "org.hibernate:hibernate-core:${hibernateVersion}"
```

```
}

}
```

注意这里子项目名字前面有一个冒号（：）。 通过这种方式，指定对应的子项目，并对其进行配置。

- 我们还可以在每个子项目的目录里建立自己的构建脚本。在上例中，可以在子项目 core 目录下为其建立一个 build.gradle 文件，并在该构建脚本中配置 core 子项目所需的所有配置。例如，在该 build.gradle 文件中添加如下配置：

```
ext{

       hibernateVersion = '4.2.1.Final'

      }

    dependencies {

    compile "org.hibernate:hibernate-core:${hibernateVersion}"

}
```

根据我对 Gradle 的使用经验，对于子项目少，配置简单的小型项目，推荐使用第一种方式配置，这样就可以把所有的配置信息放在同一个 build.gradle 文件里。例如我同事郑晔的开源项目 moco。它只有两个子项目，因而就使用了第一种方式配置，在项目根目录下的 build.gradle 文件中设置项目相关的配置信息。但是，若是对于子项目多，并且配置复杂的大型项目，使用第二种方式对项目进行配置会更好。因为，第二种配置方式将各个项目的配置分别放到单独的 build.gradle 文件中去，可以方便设置和管理每个子项目的配置信息。

1.4 其他共享

在 Gradle 中，除了上面提到的配置信息共享，还可以共享方法以及 Task。可以在根目录的 build.gradle 文件中添加所有子项目都需要的方法，在子项目的 build.gradle 文件中调用在父项目 build.gradle 脚本里定义的方法。例如我定义了这样一个方法，它可以从命令行中获取属性，若没有提供该属性，则使用默认值：

```
def defaultProperty(propertyName, defaultValue) {

    return hasProperty(propertyName) ? project[propertyName] : defaultValue

}
```

注意，这段脚本完全就是一段 Groovy 代码，具有非常好的可读性。

由于在父项目中定义了 defaultProperty 方法，因而在子项目的 build.gradle 文件中，也可以调用该方法。

2. 环境的配置

为了方便地将应用部署到开发、测试以及产品等不同环境上， Gradle 提供了几种不同的方式为不同的环境打包，使得不同的环境可以使用不同的配置文件。此外，它还提供了简单的方法，使得我们能够便捷地初始化数据库 。

2.1 Properties 配置

要为不同的环境提供不一样的配置信息，Maven 选择使用 profile，而 Gradle 则提供了两种方法为构建脚本提供 Properties 配置：

- 第一种方式是使用传统的 properties 文件， 然后在使用 Gradle 时，通过传入不同的参数加载不同的 properties 文件。例如，我们可以在项目中提供 development.properties、test.properties 和 production.properties。在项目运行时，使用-Pprofile=development 来指定加载开发环境的配置。构建脚本中加载 properties 文件的代码如下：

```
ext {

    profile = project['profile']

}

def loadProperties(){

    def props = new Properties()

    new File("${rootProject.projectDir}/config/${profile}.properties")

            .withInputStream {

                stream -> props.load(stream)

            }

    props

}
```

在运行脚本的时候，传入的-Pprofile=development 可以指定使用哪个运行环境的配置文件。代码中使用了 project['profile']从命令行里读取-P 传入的参数，Gradle 会去父项目根目录下的 config 文件夹中需找对应的 properties 文件。

- 另外一种方式就是使用 Groovy 的语法，定义可读性更高的配置文件。比如可以在项目中定义 config.groovy 的配置文件，内容如下：

```
environments {

    development {

        jdbc {

            url = 'development'

            user = 'xxxx'

            password = 'xxxx'
```

```
        }
    }
    test {
        jdbc {
            url = 'test'
            user = 'xxxx'
            password = 'xxxx'
        }
    }
    production {
        jdbc {
            url = 'production'
            user = 'xxxx'
            password = 'xxxx'
        }
    }
}
```

这里定义了三个环境下的不同数据库配置，在构建脚本中使用如下的代码来加载：

```
ext {
    profile = project['profile']
}
def loadGroovy(){
    def configFile = file('config.groovy')
    new ConfigSlurper(profile).parse(configFile.toURL()).toProperties()
}
```

这里在 ConfigSlurper 的构造函数里传入从命令行里取到的-P 的参数。调用 loadGroovy 方法就可以加载项目根目录下的 config.groovy 文件，并作为一个 Map 返回，这样就可以通过 jdbc.url 来获取 url 的值。

从可读性以及代码整洁（配置文件也需要代码整洁）而言，我推荐使用第二种方式来配置，因为这种方法具有清晰的结构。如上面的例子，就可以把数据库相关的信息都放在 jdbc 这个大的

节点下，而不用像 properties 文件这样的扁平结构。但是对于一些已经使用 properties 文件来为不同环境提供配置信息的遗留项目里，使用 properties 文件也没有问题。

2.2 替换

通过不同的方式加载不同环境的配置后，就需要把它们替换到有占位符的配置文件中去。在配置文件中使用@key@来标注要被替换的位置，比如在 config 文件夹中有 jdbc.properties 文件，其内容如下：

```
jdbc.url=@jdbc.url@

jdbc.user=@jdbc.user@

jdbc.password=@jdbc.password@
```

在 Gradle 构建过程中，有一个 processResources 的 Task，可以修改该 Task 的配置，让其在构建过程中替换资源文件中的占位符：

```
processResources {

    from(sourceSets.main.resources.srcDirs) {

        filter(org.apache.tools.ant.filters.ReplaceTokens,

                        tokens: loadGroovyConfig()

)

}

}
```

上面这种做法用来处理子项目 src/main/resources 文件夹下的资源文件，所以需要将这段代码放在子项目的独立配置文件里。

在一些复杂的项目中，经常会把配置文件放置到一个目录进行统一管理。比如在我所在的项目，就专门提供了一个 config 子目录，里面存放了所有的配置信息。在处理这些资源文件时，Gradle 默认提供的 processResources 就不够用了，我们需要在 Gradle 脚本中定义一个 Task 去替换这些包含占位符的配置文件，然后让 package 或者 deploy 的 Task 依赖这个 Task。该 Task 的代码如下：

```
task replace(type: Sync) {

            def configHome = "${project.rootDir}/config"



    from(configHome) {

        include '**/*.properties'

        include '**/*.xml'

        filter org.apache.tools.ant.filters.ReplaceTokens,

tokens: loadGroovyConfig()
```

```
    }

    into "${buildDir}/resources/main"

}
```

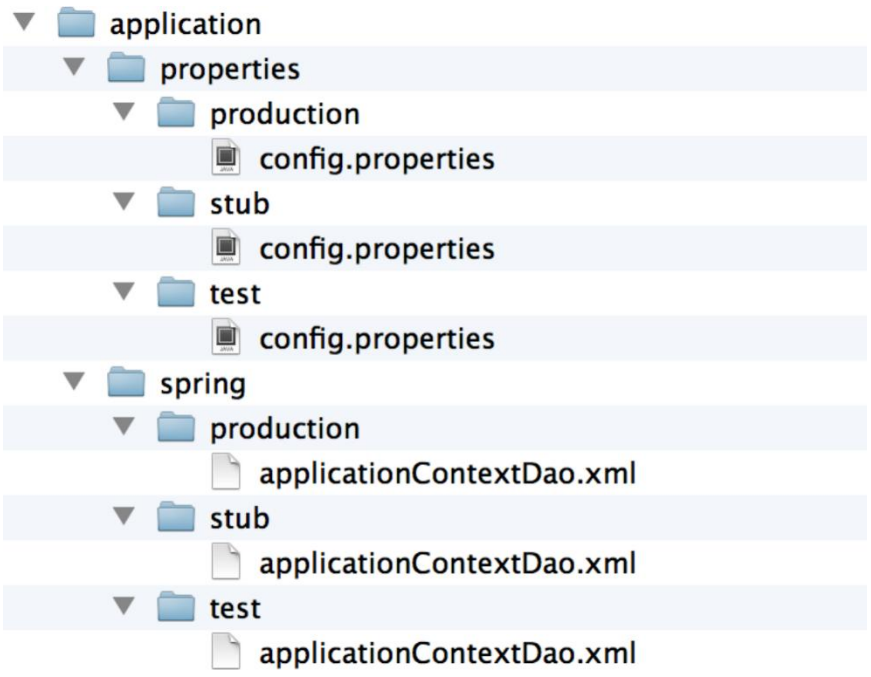
这里定义了一个 Sync 类型的 Task，会将父项目的根目录下的 config 文件夹的所有 properties 和 xml 文件使用从 loadGroovyConfig()方法中加载出来的配置替换，并将替换之后的文件放到 build 文件夹下的 resource/main 目录中。再让打包的 Task 依赖这个 Task，就会把替换之后的配置文件打到包中。

2.3 更复杂的情况

上面介绍了在项目中如何使用 Gradle 处理 properties 和 xml 文件中具有相同配置，但其中的一些值并不相同的情况 。然而，在有些项目中不同的环境配置之间变化的不仅是值，很有可能整个配置文件都不相同；那么，使用上面替换的处理方式就无法满足要求了。

在我所在的项目中，我们需要依赖一个外部的 Web Service。在开发环境上，我们使用了 Stub 来模拟和 Web Service 之间的交互，为开发环境提供测试数据，这些数据都放置在一个 Spring 的配置文件中；而在测试和产品环境上，又要使用对应的测试和产品环境的 Web Service。这时，开发、测试与产品环境的配置完全不同。对于这种复杂的情况，Gradle 可以在构建过程中为不同的环境指定不同的资源文件夹，在不同的资源文件夹中包含不同的配置文件。

例如，在我们项目的 config 目录下包含了 application 文件夹，定义了不同环境所需的不同配置文件，其目录结构如下图所示：



在构建脚本中，根据从命令行读入的-P 参数，使用不同的资源文件夹，其代码如下：

```
sourceSets {

    main {

        resources {
```

```
            srcDir "config/application/spring/${profile}",

                    "config/application/properties/${profile}"

        }

    }

}
```

这样在打包的过程中，就可以使用-P 传入的参数的资源文件夹下面的 properties 和 xml 文件作为项目的配置文件。

2.4 初始化数据库

在项目开发过程中，为了方便为不同环境构建相同的数据库及数据，我们通常需创建数据库的表以及插入一些初始化数据。Gradle 目前没有提供相关的 Task 或者 Plugin，但是我们可以自己创建 Task 去运行 SQL 来初始化各个环境上的数据库。

前面也提到 Gradle 是 Groovy 定义的 DSL，所以我们可以在 Gradle 中使用 Groovy 的代码来执行 SQL 脚本文件。在 Gradle 脚本中，使用 Groovy 加载数据库的 Driver 之后，就可以使用 Groovy 提供的 Sql 类去执行 SQL 来初始化数据库了。代码如下：

```
groovy.sql.Sql oracleSql =

    Sql.newInstance(props.getProperty('database.connection.url'),

                props.getProperty('database.userid'),

                props.getProperty('database.password'),

                props.getProperty('database.connection.driver'))

try {

        new File(script).text.split(";").each {

            logger.info it

            oracleSql.execute(it)

        }

    } catch (Exception e) { }
```

这段代码会初始化执行 SQL 的 groovy.sql.Sql 对象，然后按照分号（;）分割 SQL 脚本文件里的每一条 SQL 并执行。对于一些必须运行成功的 SQL 文件，可以在 catch 块里通过抛出异常来中止数据库的初始化。需要注意的是需要将数据库的 Driver 加载到 ClassPath 里才可以正确地执行。

因为在 Gradle 中包含了 Ant，所以我们除了使用 Groovy 提供的 API 来执行 SQL 之外，还可以使用 Ant 的 sql 任务来执行 SQL 脚本文件。但若非特殊情况，我并不推荐使用 Ant 任务，这部分内容与本文无关，这里不再细述 。

3. 代码质量

代码质量是软件开发质量的一部分，除了人工代码评审之外，在把代码提交到代码库之前，还应该使用自动检查工具来自动检查代码，来保证项目的代码质量。下面介绍一下 Gradle 提供的支持代码检查的插件。

3.1 CheckStyle

CheckStyle 是 SourceForge 下的一个项目，提供了一个帮助 JAVA 开发人员遵守某些编码规范的工具。它能够自动化代码规范检查过程，从而使得开发人员从这项重要却枯燥的任务中解脱出来。Gradle 官方提供了 CheckStyle 的插件，在 Gradle 的构建脚本中只需要应用该插件：

```
apply plugin: 'checkstyle'
```

默认情况下，该插件会找/config/checkstyle/checkstyle.xml 作为 CheckStyle 的配置文件，可以在 checkstyle 插件的配置阶段（Configuration） 设置 CheckStyle 的配置文件：

```
checkstyle{

configFile = file('config/checkstyle/checkstyle-main.xml')

}
```

还可以通过 checkstyle 设置 CheckStyle 插件的其他配置。

3.2 FindBugs

FindBugs 是一个静态分析工具，它检查类或者 JAR 文件，将字节码与一组缺陷模式进行对比以发现可能的问题。Gradle 使用如下的代码为项目的构建脚本添加 FindBugs 的插件：

```
apply plugin: 'findbugs'
```

同样也可以在 FindBugs 的配置阶段（Configuration）设置其相关的属性，比如 Report 的输出目录、检查哪些 sourceSet 等。

3.3 JDepend

在开发 Java 项目时经常会遇到关于包混乱的问题，JDepend 工具可以帮助你在开发过程中随时跟踪每个包的依赖性（引用/被引用），从而设计高维护性的架构，不论是在打包发布还是版本升级都会更加轻松。在构建脚本中加入如下代码即可：

```
apply plugin: 'jdepend'
```

3.4 PMD

PMD 是一种开源分析 Java 代码错误的工具。与其他分析工具不同的是，PMD 通过静态分析获知代码错误，即在不运行 Java 程序的情况下报告错误。PMD 附带了许多可以直接使用的规则，利用这些规则可以找出 Java 源程序的许多问题。此外，用户还可以自己定义规则，检查 Java 代码是否符合某些特定的编码规范。在构建脚本中加入如下代码：

```
apply plugin: 'pmd'
```

3.5 小结

上面提到的几种代码检查插件 apply 到构建脚本之后，可以运行：

```
gradle check
```

来执行代码质量检查。更详细的信息请查阅 Gradle 的官方文档。运行结束后会在对应的项目目录下的 build 文件夹下生成 report。

对于 Gradle 没有提供的代码检查工具，我们可以有两种选择：第一就是自己实现一个 Gradle 插件，第二就是调用 Ant 任务，让 Ant 作为一个媒介去调用在 Ant 中已经有的代码检查工具，比如测试覆盖率的 Cobertura。我们的项目使用了 Ant 来调用 Cobertura，但是为了使用方便，我们将它封装为一个 Gradle 插件，这样就可以在不同的项目里重用。

4. 依赖

几乎每个 Java 项目都会用到开源框架。同时，对于具有多个子模块的项目来说，项目之间也会有所依赖。所以，管理项目中对开源框架和其他模块的依赖是每个项目必须面对的问题。同时，Gradle 也使用 Repository 来管理依赖。

4.1 Jar 包依赖管理

Maven 提出了使用 Repository 来管理 Jar 包，Ant 也提供了使用 Ivy 来管理 jar 包。Gradle 提供了对所有这些 Respository 的支持，可以从 Gradle 的官方文档上了解更详细的信息。

Gradle 沿用 Maven 的依赖管理方法，通过 groupId、name 和 version 到配置的 Repository 里寻找指定的 Jar 包。同样，它也提供了和 Maven 一样的构建生命周期，compile、runtime、testCompile 和 testRuntime 分别对应项目不同阶段的依赖。通过如下方式为构建脚本指定依赖：

```
dependencies {

    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'

    testCompile group:'junit', name: 'junit', version '4.11'

}
```

这里分别指定 group、name 以及 version，但是 Gradle 提供了一种更简单的方式来指定依赖：

```
dependencies {

    compile 'org.hibernate:hibernate-core:3.6.7.Final'

    testCompile 'junit:junit:4.11'

}
```

这样比 Maven 使用 XML 来管理依赖简单多了，但是还可以更简单一点。实际上这里的 compile 和 testCompile 是 Groovy 为 Gradle 提供的方法，可以为其传入多个参数，所以当 compile 有多个 Jar 包依赖的时候，可以同时指定到 compile 里去，代码如下：

```
compile(

            'org.hibernate:hibernate-core:3.6.7.Final',

             'org.springframework:spring-context:3.1.4.RELEASE'

)
```

另外，当在 Respository 无法找到 Jar 包时（如数据库的 driver），就可以将这些 Jar 包放在项目的一个子目录中，然后让项目管理依赖。例如，我们可以在项目的根目录下创建一个 lib 文件夹，用以存放这些 Jar 包。使用如下代码可以将其添加到项目依赖中：

```
dependencies {

    compile(

            'org.hibernate:hibernate-core:3.6.7.Final',

             'org.springframework:spring-context:3.1.4.RELEASE',

            fileTree(dir: "${rootProject.projectDir}/lib", include: '*.jar')

)

}
```

4.2 子项目之间的依赖

对于多模块的项目，项目中的某些模块需要依赖于其他模块，前面提到在初始化阶段，Gradle 为每个模块都创建了一个 Project 对象，并且可以通过模块的名字引用到该对象。在配置模块之间的依赖时，使用这种方式可以告诉 Gradle 当前模块依赖了哪些子模块。例如，在我们的项目中，cis-war 会依赖 core 子项目，就可以在 cis-war 的构建脚本中加上如下代码：

```
dependencies {

    compile(

            'org.hibernate:hibernate-core:3.6.7.Final',

              project(':core')

)

}
```

通过 project(':core')来引用 core 子项目，在构建 cis-war 时，Gradle 会把 core 加到 ClassPath 中。

4.3 构建脚本的依赖

除了项目需要依赖之外，构建脚本本身也可以有自己的依赖。当使用一个非 Gradle 官方提供的插件时，就需要在构建脚本里指定其依赖，当然还需要指定该插件的 Repository。在 Gradle 中，使用 buildscript 块为构建脚本配置依赖。

比如在项目中使用 cucumber-JVM 作为项目 BDD 工具，而 Gradle 官方没有提供它的插件，好在开源社区有人提供 cucumber 的插件。在构建脚本中添加如下代码：

```
buildscript {

    repositories {

        mavenCentral()

    }

    dependencies {
```

```
        classpath "gradle-cucumber-plugin:gradle-cucumber-plugin:0.2"

    }

}

apply plugin: com.excella.gradle.cucumber.CucumberPlugin
```

5. 其他

5.1 apply 其他 Gradle 文件

当一个项目很复杂的时候，Gradle 脚本也会很复杂，除了将子项目的配置移到对应项目的构建脚本之外，还可以可以按照不同的功能将复杂的构建脚本拆分成小的构建脚本，然后在 build.gradle 里使用 apply from，将这些小的构建脚本引入到整体的构建脚本中去。比如在一个项目中既使用了 Jetty，又使用了 Cargo 插件启动 JBoss，就可以把他们分别提到 jetty.gradle 和 jboss.gradle，然后在 build.gradle 里使用如下的代码将他们引入进来：

```
apply from: "jetty.gradle"

apply from: "jboss.gradle"
```

5.2 project 的目录

在脚本文件中，需要访问项目中的各级目录结构。Gradle 为 Project 对象定义了一些属性指向项目的根目录，方便在脚本中引用。

- rootDir：在子项目的脚本文件中可以通过该属性访问到根项目路径。
- rootProject：在子项目中，可以通过该属性获取父项目的 Project 对象。

5.3 使用 Wrapper 指定 Gradle 的版本

为了统一项目中 Gradle 的版本，可以在构建脚本中通过定义一个 wrapper 的 Task，并在该 Task 中指定 Gradle 的版本以及存放 Gradle 的位置。

```
task wrapper(type: Wrapper) {

    gradleVersion = '1.0'

    archiveBase = 'PROJECT'

    archivePath = 'gradle/dists'

}
```

运行 gradle wrapper， 就会在根项目目录下创建一个 wrapper 的文件夹，会包含 wrapper 的 Jar 包和 properties 文件。之后就可以使用 gradlew 来运行 task。第一次使用 gradlew 执行 task 的时候，会在项目根目录下的 gradle/dists 下下载你指定的 Gradle 版本 。这样在项目构建的时候，就会使用该目录下的 Gradle，保证整个团队使用了相同的 Gradle 版本。

5.4 使用 gradle.properties 文件

Gradle 构建脚本会自动找同级目录下的 gradle.properties 文件，在这个文件中可以定义一些 property，以供构建脚本使用。例如，我们要使用的 Repository 需要提供用户名和密码，就可以

将其配置在 gradle.properties 中。这样，每个团队成员都可以修改该配置文件，却不用上传到代码库中对团队其他成员造成影响。可以使用如下的代码定义：

```
username=user

password=password
```

在构建脚本中使用"${username} "就可以访问该文件中定义的相关值。

由于篇幅有限，本文只是我在一个大型 Java 项目上使用 Gradle 的部分经验，并未涵盖所有 Gradle 相关的知识，包括如何编写 Gradle 插件以及 Gradle 对其他语言的构建，读者可以通过阅读 Gradle 的官方文档（比起其他开源软件，Gradle 的另一特点就是文档详细）来了解。另外，Gradle 是基于 Groovy 的构建工具，在使用 Gradle 的时候也需要了解和使用 Groovy。所以，在学习 Gradle 插件的过程中，也能学会 Groovy 相关的用法，可谓一举两得。

参考文献:

[1] CoC: http://en.wikipedia.org/wiki/Convention_over_configuration
[2] DSL: http://en.wikipedia.org/wiki/Domain-specific_language
[3] Micro Service Architecture: http://yobriefca.se/blog/2013/04/29/micro-service-architecture/
[4] Guava: https://code.google.com/p/guava-libraries/

作者介绍

何海洋，Thoughtworks 咨询师，毕业于大连海事大学，有多年软件开发经验，主要从事 Java 项目的开发。目前在 Thoughtworks 公司从事敏捷软件开发。

深入理解 Java 内存模型——锁

作者 程晓明

锁的释放-获取建立的 happens before 关系

锁是 Java 并发编程中最重要的同步机制。锁除了让临界区互斥执行外，还可以让释放锁的线程向获取同一个锁的线程发送消息。

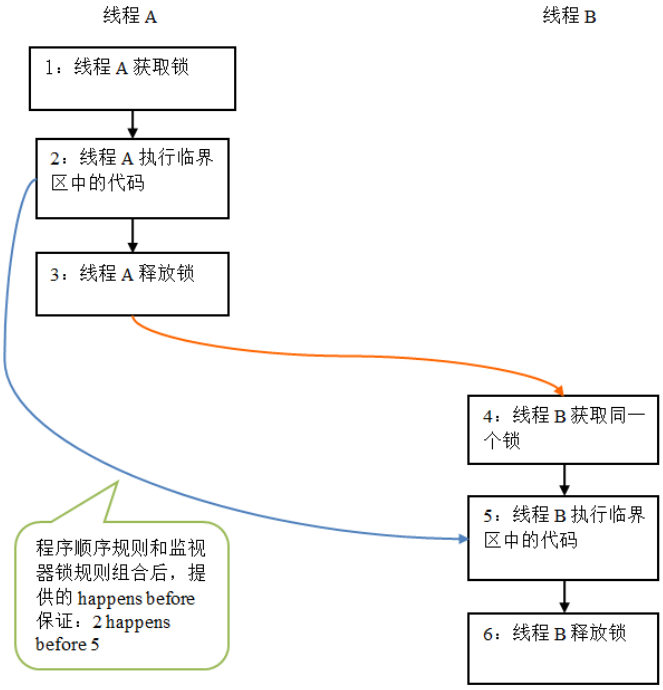
下面是锁释放-获取的示例代码：

```
class MonitorExample {

    int a = 0;



    public synchronized void writer() {  //1

        a++;                              //2

    }                                     //3



    public synchronized void reader() {  //4

        int i = a;                        //5

        ……

    }                                     //6

}
```

假设线程 A 执行 writer()方法，随后线程 B 执行 reader()方法。根据 happens before 规则，这个过程包含的 happens before 关系可以分为两类：

1. 根据程序次序规则，1 happens before 2, 2 happens before 3; 4 happens before 5, 5 happens before 6。
2. 根据监视器锁规则，3 happens before 4。
3. 根据 happens before 的传递性，2 happens before 5。

上述 happens before 关系的图形化表现形式如下：

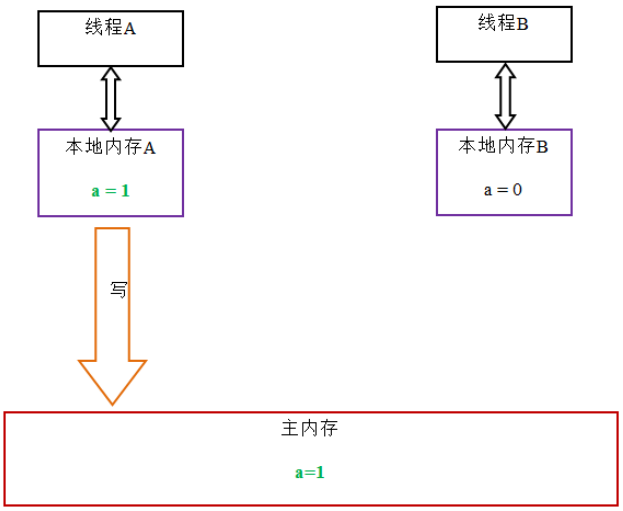


在上图中，每一个箭头链接的两个节点，代表了一个 happens before 关系。黑色箭头表示程序顺序规则；橙色箭头表示监视器锁规则；蓝色箭头表示组合这些规则后提供的 happens before 保证。

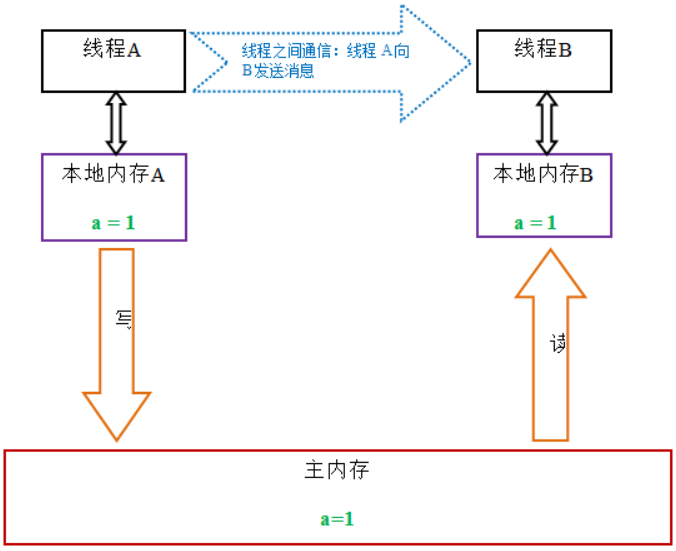
上图表示在线程 A 释放了锁之后，随后线程 B 获取同一个锁。在上图中，2 happens before 5。因此，线程 A 在释放锁之前所有可见的共享变量，在线程 B 获取同一个锁之后，将立刻变得对 B 线程可见。

锁释放和获取的内存语义

当线程释放锁时，JMM 会把该线程对应的本地内存中的共享变量刷新到主内存中。以上面的 MonitorExample 程序为例，A 线程释放锁后，共享数据的状态示意图如下：



当线程获取锁时，JMM 会把该线程对应的本地内存置为无效。从而使得被监视器保护的临界区代码必须要从主内存中去读取共享变量。下面是锁获取的状态示意图：



对比锁释放-获取的内存语义与 volatile 写-读的内存语义，可以看出：锁释放与 volatile 写有相同的内存语义；锁获取与 volatile 读有相同的内存语义。

下面对锁释放和锁获取的内存语义做个总结：

- 线程 A 释放一个锁，实质上是线程 A 向接下来将要获取这个锁的某个线程发出了（线程 A 对共享变量所做修改的）消息。
- 线程 B 获取一个锁，实质上是线程 B 接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。
- 线程 A 释放锁，随后线程 B 获取这个锁，这个过程实质上是线程 A 通过主内存向线程 B 发送消息。

锁内存语义的实现

本文将借助 ReentrantLock 的源代码，来分析锁内存语义的具体实现机制。

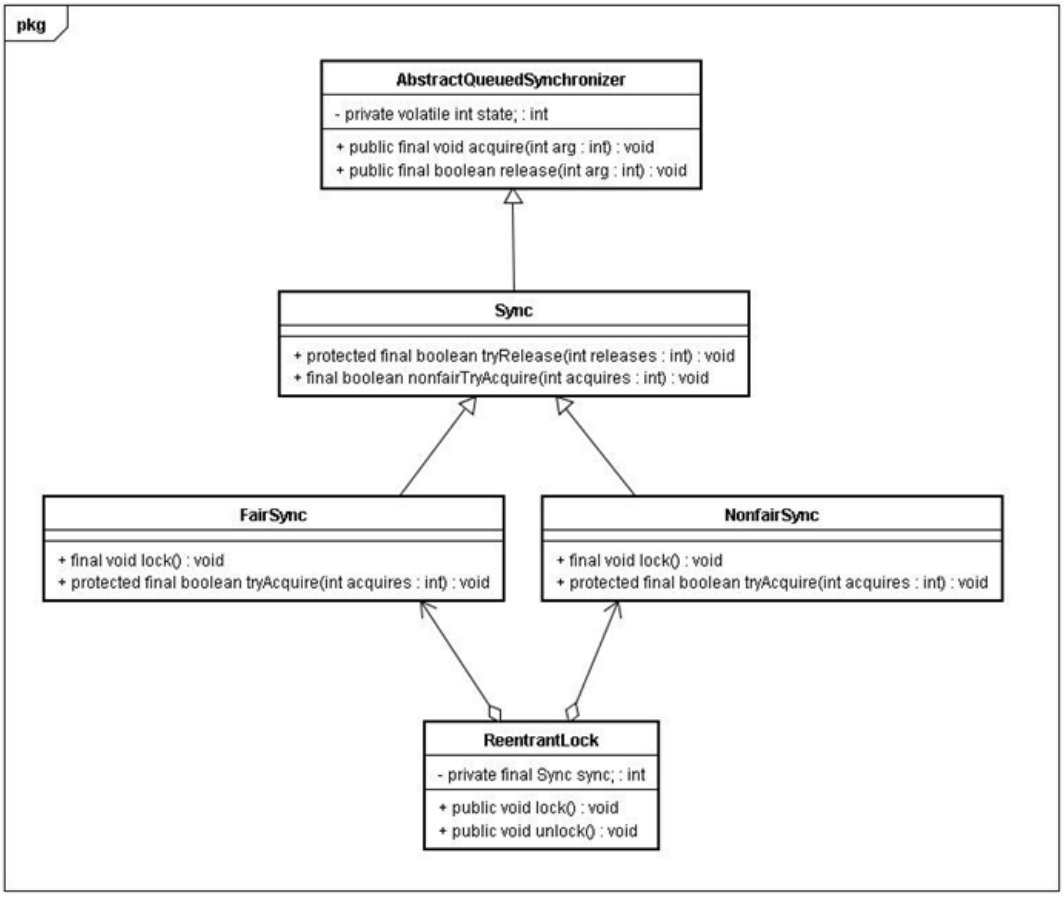
请看下面的示例代码：

```
class ReentrantLockExample {

int a = 0;

ReentrantLock lock = new ReentrantLock();



public void writer() {

    lock.lock();          //获取锁

    try {

        a++;

    } finally {
```

```
            lock.unlock();  //释放锁
        }
    }


public void reader () {
        lock.lock();        //获取锁
        try {
            int i = a;
            ……
        } finally {
            lock.unlock();  //释放锁
        }
    }
}
```

在 ReentrantLock 中，调用 lock()方法获取锁；调用 unlock()方法释放锁。

ReentrantLock 的实现依赖于 java 同步器框架 AbstractQueuedSynchronizer（本文简称之为 AQS）。AQS 使用一个整型的 volatile 变量（命名为 state）来维护同步状态，马上我们会看到，这个 volatile 变量是 ReentrantLock 内存语义实现的关键。 下面是 ReentrantLock 的类图（仅画出与本文相关的部分）：



ReentrantLock 分为公平锁和非公平锁，我们首先分析公平锁。

使用公平锁时，加锁方法 lock()的方法调用轨迹如下：

1. ReentrantLock : lock()
2. FairSync : lock()
3. AbstractQueuedSynchronizer : acquire(int arg)
4. ReentrantLock : tryAcquire(int acquires)

在第 4 步真正开始加锁，下面是该方法的源代码：

```
protected final boolean tryAcquire(int acquires) {

    final Thread current = Thread.currentThread();

    int c = getState();    //获取锁的开始，首先读 volatile 变量 state

    if (c == 0) {

        if (isFirst(current) &&

            compareAndSetState(0, acquires)) {

            setExclusiveOwnerThread(current);

            return true;

        }
```

```
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

从上面源代码中我们可以看出，加锁方法首先读 volatile 变量 state。

在使用公平锁时，解锁方法 unlock()的方法调用轨迹如下：

- ReentrantLock : unlock()
- AbstractQueuedSynchronizer : release(int arg)
- Sync : tryRelease(int releases)

在第 3 步真正开始释放锁，下面是该方法的源代码：

```
protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);            //释放锁的最后，写 volatile 变量 state
    return free;
}
```

从上面的源代码我们可以看出，在释放锁的最后写 volatile 变量 state。

公平锁在释放锁的最后写 volatile 变量 state；在获取锁时首先读这个 volatile 变量。根据 volatile 的 happens-before 规则，释放锁的线程在写 volatile 变量之前可见的共享变量，在获取锁的线程读取同一个 volatile 变量后将立即变的对获取锁的线程可见。

现在我们分析非公平锁的内存语义的实现。

非公平锁的释放和公平锁完全一样，所以这里仅仅分析非公平锁的获取。

使用公平锁时，加锁方法 lock()的方法调用轨迹如下：

- ReentrantLock : lock()
- NonfairSync : lock()
- AbstractQueuedSynchronizer : compareAndSetState(int expect, int update)

在第 3 步真正开始加锁，下面是该方法的源代码：

```
protected final boolean compareAndSetState(int expect, int update) {

    return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

}
```

该方法以原子操作的方式更新 state 变量，本文把 Java 的 compareAndSet()方法调用简称为 CAS。JDK 文档对该方法的说明如下：如果当前状态值等于预期值，则以原子方式将同步状态设置为给定的更新值。此操作具有 volatile 读和写的内存语义。

这里我们分别从编译器和处理器的角度来分析,CAS 如何同时具有 volatile 读和 volatile 写的内存语义。

前文我们提到过，编译器不会对 volatile 读与 volatile 读后面的任意内存操作重排序；编译器不会对 volatile 写与 volatile 写前面的任意内存操作重排序。组合这两个条件，意味着为了同时实现 volatile 读和 volatile 写的内存语义，编译器不能对 CAS 与 CAS 前面和后面的任意内存操作重排序。

下面我们来分析在常见的 intel x86 处理器中，CAS 是如何同时具有 volatile 读和 volatile 写的内存语义的。

下面是 sun.misc.Unsafe 类的 compareAndSwapInt()方法的源代码：

```
public final native boolean compareAndSwapInt(Object o, long offset,

                                               int expected,

                                               int x);
```

可以看到这是个本地方法调用。这个本地方法在 openjdk 中依次调用的 c++代码为：unsafe.cpp，atomic.cpp 和 atomicwindowsx86.inline.hpp。这个本地方法的最终实现在 openjdk 的如下位置：openjdk-7-fcs-src-b147-27jun2011\openjdk\hotspot\src\oscpu\windowsx86\vm\atomicwindowsx86.inline.hpp（对应于 windows 操作系统，X86 处理器）。下面是对应于 intel x86 处理器的源代码的片段：

```
// Adding a lock prefix to an instruction on MP machine

// VC++ doesn't like the lock prefix to be on a single line

// so we can't insert a label after the lock prefix.

// By emitting a lock prefix, we can define a label after it.

#define LOCK_IF_MP(mp) __asm cmp mp, 0  \

                       __asm je L0      \

                       __asm _emit 0xF0 \

                       __asm L0:



inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint
compare_value) {

  // alternative for InterlockedCompareExchange

  int mp = os::is_MP();

  __asm {

    mov edx, dest

    mov ecx, exchange_value

    mov eax, compare_value

    LOCK_IF_MP(mp)

    cmpxchg dword ptr [edx], ecx

  }

}
```

如上面源代码所示，程序会根据当前处理器的类型来决定是否为 cmpxchg 指令添加 lock 前缀。如果程序是在多处理器上运行，就为 cmpxchg 指令加上 lock 前缀（lock cmpxchg）。反之，如果程序是在单处理器上运行，就省略 lock 前缀（单处理器自身会维护单处理器内的顺序一致性，不需要 lock 前缀提供的内存屏障效果）。

intel 的手册对 lock 前缀的说明如下。

1. 确保对内存的读-改-写操作原子执行。在 Pentium 及 Pentium 之前的处理器中，带有 lock 前缀的指令在执行期间会锁住总线，使得其他处理器暂时无法通过总线访问内存。很显然，这会带来昂贵的开销。从 Pentium 4，Intel Xeon 及 P6 处理器开始，intel 在原有总线锁的基础上做了一个很有意义的优化：如果要访问的内存区域（area of memory）在 lock 前缀指令执行期间已经在处理器内部的缓存中被锁定（即包含该内存区域的缓存行当前处于独占或以修改状态），并且该内存区域被完全包含在单个缓存行（cache line）中，那么处理器将直接执行该指令。由于在指令执行期间该缓存行会一直被锁定，其它处理器无法读/写该指

令要访问的内存区域，因此能保证指令执行的原子性。这个操作过程叫做缓存锁定（cache locking），缓存锁定将大大降低 lock 前缀指令的执行开销，但是当多处理器之间的竞争程度很高或者指令访问的内存地址未对齐时，仍然会锁住总线。

2. 禁止该指令与之前和之后的读和写指令重排序。
3. 把写缓冲区中的所有数据刷新到内存中。

上面的第 2 点和第 3 点所具有的内存屏障效果，足以同时实现 volatile 读和 volatile 写的内存语义。

经过上面的这些分析，现在我们终于能明白为什么 JDK 文档说 CAS 同时具有 volatile 读和 volatile 写的内存语义了。

现在对公平锁和非公平锁的内存语义做个总结：

- 公平锁和非公平锁释放时，最后都要写一个 volatile 变量 state。
- 公平锁获取时，首先会去读这个 volatile 变量。
- 非公平锁获取时，首先会用 CAS 更新这个 volatile 变量,这个操作同时具有 volatile 读和 volatile 写的内存语义。

从本文对 ReentrantLock 的分析可以看出，锁释放-获取的内存语义的实现至少有下面两种方式：

1. 利用 volatile 变量的写-读所具有的内存语义。
2. 利用 CAS 所附带的 volatile 读和 volatile 写的内存语义。

concurrent 包的实现

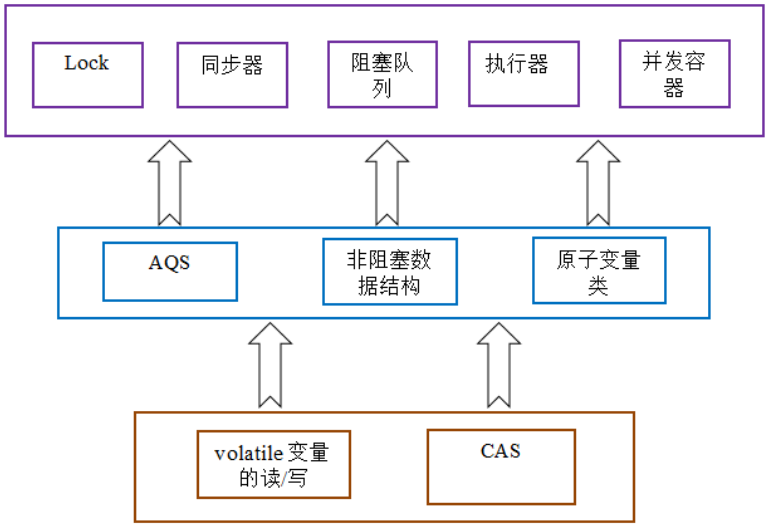
由于 Java 的 CAS 同时具有 volatile 读和 volatile 写的内存语义，因此 Java 线程之间的通信现在有了下面四种方式：

1. A 线程写 volatile 变量，随后 B 线程读这个 volatile 变量。
2. A 线程写 volatile 变量，随后 B 线程用 CAS 更新这个 volatile 变量。
3. A 线程用 CAS 更新一个 volatile 变量，随后 B 线程用 CAS 更新这个 volatile 变量。
4. A 线程用 CAS 更新一个 volatile 变量，随后 B 线程读这个 volatile 变量。

Java 的 CAS 会使用现代处理器上提供的高效机器级别原子指令，这些原子指令以原子方式对内存执行读-改-写操作，这是在多处理器中实现同步的关键（从本质上来说，能够支持原子性读-改-写指令的计算机器，是顺序计算图灵机的异步等价机器，因此任何现代的多处理器都会去支持某种能对内存执行原子性读-改-写操作的原子指令）。同时，volatile 变量的读/写和 CAS 可以实现线程之间的通信。把这些特性整合在一起，就形成了整个 concurrent 包得以实现的基石。如果我们仔细分析 concurrent 包的源代码实现，会发现一个通用化的实现模式：

1. 首先，声明共享变量为 volatile；
2. 然后，使用 CAS 的原子条件更新来实现线程之间的同步；
3. 同时，配合以 volatile 的读/写和 CAS 所具有的 volatile 读和写的内存语义来实现线程之间的通信。

AQS，非阻塞数据结构和原子变量类（java.util.concurrent.atomic 包中的类），这些 concurrent 包中的基础类都是使用这种模式来实现的，而 concurrent 包中的高层类又是依赖于这些基础类来实现的。从整体来看，concurrent 包的实现示意图如下：



参考文献

- Concurrent Programming in Java: Design Principles and Pattern
- JSR 133 (Java Memory Model) FAQ
- JSR-133: Java Memory Model and Thread Specification
- Java Concurrency in Practice
- Java™ Platform, Standard Edition 6 API Specification
- The JSR-133 Cookbook for Compiler Writers
- Intel® 64 and IA-32 ArchitecturesvSoftware Developer's Manual Volume 3A: System Programming Guide, Part 1
- The Art of Multiprocessor Programming

深入分析 ConcurrentHashMap

作者 方腾飞

术语定义

术语	英文	解释
哈希算法	hash algorithm	一种将任意内容的输入转换成相同长度输出的加密方式，其输出被称为哈希值
哈希表	hash table	根据设定的哈希函数 H(key)和处理冲突方法将一组关键字映象到一个有限的地址区间上，并以关键字在地址区间中的象作为记录在表中的存储位置，这种表称为哈希表或散列，所得存储位置称为哈希地址或散列地址

线程不安全的 HashMap

因为多线程环境下，使用 HashMap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，所以在并发情况下不能使用 HashMap，如以下代码：

```
final HashMap<String, String> map = new HashMap<String, String>(2);

Thread t = new Thread(new Runnable() {

    @Override

    public void run() {

        for (int i = 0; i < 10000; i++) {

            new Thread(new Runnable() {

                @Override

                public void run() {

                    map.put(UUID.randomUUID().toString(), "");

                }

            }, "ftf" + i).start();

        }

    }

}, "ftf");
```

```
t.start();

t.join();
```

效率低下的 HashTable 容器

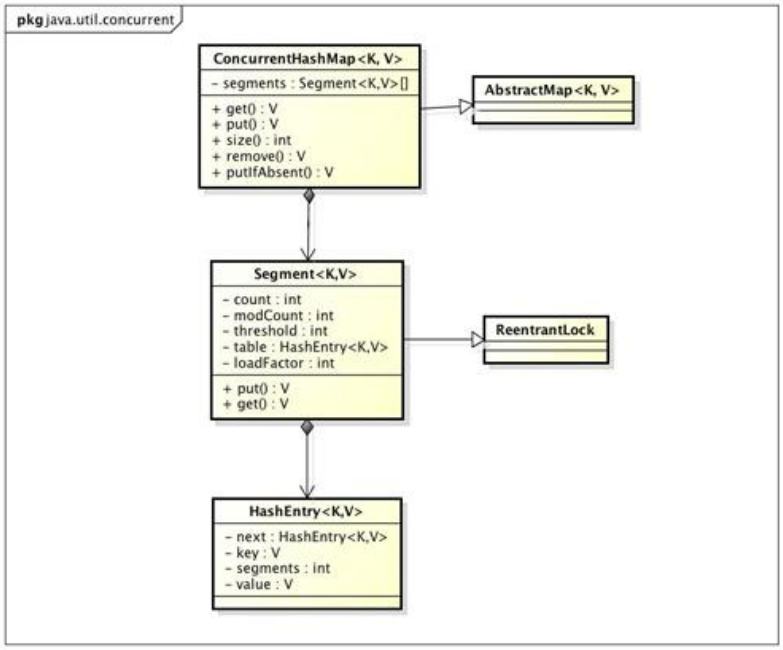
HashTable 容器使用 synchronized 来保证线程安全，但在线程竞争激烈的情况下 HashTable 的效率非常低下。因为当一个线程访问 HashTable 的同步方法时，其他线程访问 HashTable 的同步方法时，可能会进入阻塞或轮询状态。如线程 1 使用 put 进行添加元素，线程 2 不但不能使用 put 方法添加元素，并且也不能使用 get 方法来获取元素，所以竞争越激烈效率越低。

锁分段技术

HashTable 容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问 HashTable 的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是 ConcurrentHashMap 所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

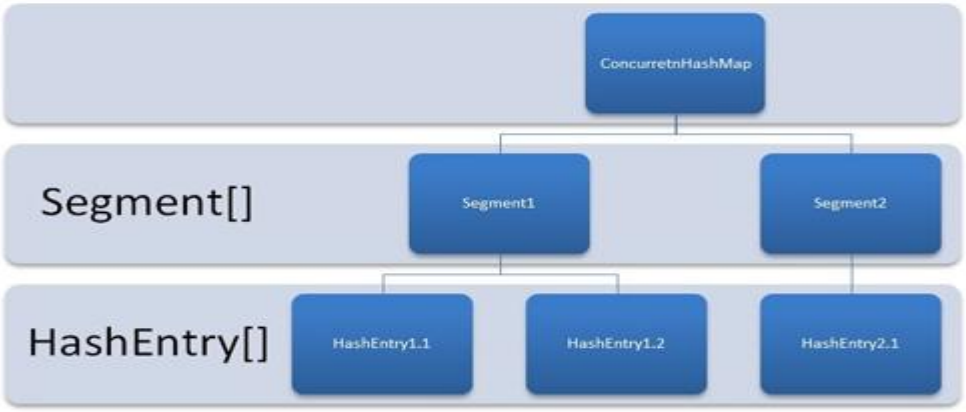
ConcurrentHashMap 的结构

我们通过 ConcurrentHashMap 的类图来分析 ConcurrentHashMap 的结构。



ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。Segment 是一种可重入锁 ReentrantLock，在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构， 一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的

元素， 每个 Segment 守护者一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。



ConcurrentHashMap 的初始化

ConcurrentHashMap 初始化方法是通过 initialCapacity，loadFactor, concurrencyLevel 几个参数来初始化 segments 数组，段偏移量 segmentShift，段掩码 segmentMask 和每个 segment 里的 HashEntry 数组 。

初始化 segments 数组。让我们来看一下初始化 segmentShift，segmentMask 和 segments 数组的源代码。

```
if (concurrencyLevel > MAX_SEGMENTS)

    concurrencyLevel = MAX_SEGMENTS;

// Find power-of-two sizes best matching arguments

int sshift = 0;

int ssize = 1;

while (ssize < concurrencyLevel) {

    ++sshift;

    ssize <<= 1;

}

segmentShift = 32 - sshift;

segmentMask = ssize - 1;

this.segments = Segment.newArray(ssize);
```

由上面的代码可知 segments 数组的长度 ssize 通过 concurrencyLevel 计算得出。为了能通过按位与的哈希算法来定位 segments 数组的索引，必须保证 segments 数组的长度是 2 的 N 次方（power-of-two size），所以必须计算出一个是大于或等于 concurrencyLevel 的最小的 2 的 N 次方值来作为 segments 数组的长度。假如 concurrencyLevel 等于 14，15 或 16，ssize 都会等于 16，即容器

里锁的个数也是 16。注意 concurrencyLevel 的最大大小是 65535，意味着 segments 数组的长度最大为 65536，对应的二进制是 16 位。

初始化 segmentShift 和 segmentMask。这两个全局变量在定位 segment 时的哈希算法里需要使用，sshift 等于 ssize 从 1 向左移位的次数，在默认情况下 concurrencyLevel 等于 16，1 需要向左移位移动 4 次，所以 sshift 等于 4。segmentShift 用于定位参与 hash 运算的位数，segmentShift 等于 32 减 sshift，所以等于 28，这里之所以用 32 是因为 ConcurrentHashMap 里的 hash()方法输出的最大数是 32 位的，后面的测试中我们可以看到这点。segmentMask 是哈希运算的掩码，等于 ssize 减 1，即 15，掩码的二进制各个位的值都是 1。因为 ssize 的最大长度是 65536，所以 segmentShift 最大值是 16，segmentMask 最大值是 65535，对应的二进制是 16 位，每个位都是 1。

初始化每个 Segment。输入参数 initialCapacity 是 ConcurrentHashMap 的初始化容量，loadfactor 是每个 segment 的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个 segment。

```
if (initialCapacity > MAXIMUM_CAPACITY)

    initialCapacity = MAXIMUM_CAPACITY;

int c = initialCapacity / ssize;

if (c * ssize < initialCapacity)

    ++c;

int cap = 1;

while (cap < c)

    cap <<= 1;

for (int i = 0; i < this.segments.length; ++i)

    this.segments[i] = new Segment<K,V>(cap, loadFactor);
```

上面代码中的变量 cap 就是 segment 里 HashEntry 数组的长度，它等于 initialCapacity 除以 ssize 的倍数 c，如果 c 大于 1，就会取大于等于 c 的 2 的 N 次方值，所以 cap 不是 1，就是 2 的 N 次方。segment 的容量 threshold＝(int)cap*loadFactor，默认情况下 initialCapacity 等于 16，loadfactor 等于 0.75，通过运算 cap 等于 1，threshold 等于零。

定位 Segment

既然 ConcurrentHashMap 使用分段锁 Segment 来保护不同段的数据，那么在插入和获取元素的时候，必须先通过哈希算法定位到 Segment。可以看到 ConcurrentHashMap 会首先使用 Wang/Jenkins hash 的变种算法对元素的 hashCode 进行一次再哈希。

```
private static int hash(int h) {

        h += (h << 15) ^ 0xffffcd7d;

        h ^= (h >>> 10);

        h += (h << 3);
```

```
        h ^= (h >>> 6);

        h += (h << 2) + (h << 14);

        return h ^ (h >>> 16);

    }
```

之所以进行再哈希，其目的是为了减少哈希冲突，使元素能够均匀的分布在不同的 Segment 上，从而提高容器的存取效率。假如哈希的质量差到极点，那么所有的元素都在一个 Segment 中，不仅存取元素缓慢，分段锁也会失去意义。我做了一个测试，不通过再哈希而直接执行哈希计算。

```
System.out.println(Integer.parseInt("0001111", 2) & 15);

System.out.println(Integer.parseInt("0011111", 2) & 15);

System.out.println(Integer.parseInt("0111111", 2) & 15);

System.out.println(Integer.parseInt("1111111", 2) & 15);
```

计算后输出的哈希值全是 15，通过这个例子可以发现如果不进行再哈希，哈希冲突会非常严重，因为只要低位一样，无论高位是什么数，其哈希值总是一样。我们再把上面的二进制数据进行再哈希后结果如下，为了方便阅读，不足 32 位的高位补了 0，每隔四位用竖线分割下。

```
0100|0111|0110|0111|1101|1010|0100|1110

1111|0111|0100|0011|0000|0001|1011|1000

0111|0111|0110|1001|0100|0110|0011|1110

1000|0011|0000|0000|1100|1000|0001|1010
```

可以发现每一位的数据都散列开了，通过这种再哈希能让数字的每一位都能参加到哈希运算当中，从而减少哈希冲突。ConcurrentHashMap 通过以下哈希算法定位 segment。

```
final Segment<K,V> segmentFor(int hash) {

        return segments[(hash >>> segmentShift) & segmentMask];

    }
```

默认情况下 segmentShift 为 28，segmentMask 为 15，再哈希后的数最大是 32 位二进制数据，向右无符号移动 28 位，意思是让高 4 位参与到 hash 运算中， (hash >>> segmentShift) & segmentMask 的运算结果分别是 4，15，7 和 8，可以看到 hash 值没有发生冲突。

ConcurrentHashMap 的 get 操作

Segment 的 get 操作实现非常简单和高效。先经过一次再哈希，然后使用这个哈希值通过哈希运算定位到 segment，再通过哈希算法定位到元素，代码如下：

```
public V get(Object key) {
```

```
    int hash = hash(key.hashCode());

    return segmentFor(hash).get(key, hash);

}
```

get 操作的高效之处在于整个 get 过程不需要加锁，除非读到的值是空的才会加锁重读，我们知道 HashTable 容器的 get 方法是需要加锁的，那么 ConcurrentHashMap 的 get 操作是如何做到不加锁的呢？原因是它的 get 方法里将要使用的共享变量都定义成 volatile，如用于统计当前 Segement 大小的 count 字段和用于存储值的 HashEntry 的 value。定义成 volatile 的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在 get 操作里只需要读不需要写共享变量 count 和 value，所以可以不用加锁。之所以不会读到过期的值，是根据 java 内存模型的 happen before 原则，对 volatile 字段的写入操作先于读操作，即使两个线程同时修改和获取 volatile 变量，get 操作也能拿到最新的值，这是用 volatile 替换锁的经典应用场景。

```
transient volatile int count;

volatile V value;
```

在定位元素的代码里我们可以发现定位 HashEntry 和定位 Segment 的哈希算法虽然一样，都与数组的长度减去一相与，但是相与的值不一样，定位 Segment 使用的是元素的 hashcode 通过再哈希后得到的值的高位，而定位 HashEntry 直接使用的是再哈希后的值。其目的是避免两次哈希后的值一样，导致元素虽然在 Segment 里散列开了，但是却没有在 HashEntry 里散列开。

```
hash >>> segmentShift) & segmentMask//定位 Segment 所使用的 hash 算法

int index = hash & (tab.length - 1);// 定位 HashEntry 所使用的 hash 算法
```

ConcurrentHashMap 的 Put 操作

由于 put 方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put 方法首先定位到 Segment，然后在 Segment 里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置然后放在 HashEntry 数组里。

是否需要扩容。在插入元素前会先判断 Segment 里的 HashEntry 数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment 的扩容判断比 HashMap 更恰当，因为 HashMap 是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时 HashMap 就进行了一次无效的扩容。

如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再 hash 后插入到新的数组里。为了高效 ConcurrentHashMap 不会对整个容器进行扩容，而只对某个 segment 进行扩容。

ConcurrentHashMap 的 size 操作

如果我们要统计整个 ConcurrentHashMap 里元素的大小，就必须统计所有 Segment 里元素的大小后求和。Segment 里的全局变量 count 是一个 volatile 变量，那么在多线程场景下，我们是不是直接把所有 Segment 的 count 相加就可以得到整个 ConcurrentHashMap 大小了呢？不是的，虽然相加时可以获取每个 Segment 的 count 的最新值，但是拿到之后可能累加前使用的 count 发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计 size 的时候把所有 Segment 的 put，remove 和 clean 方法全部锁住，但是这种做法显然非常低效。 因为在累加 count 操作过程中，之前累加过的 count 发生变化的几率非常小，所以 ConcurrentHashMap 的做法是先尝试 2 次通过不锁住 Segment 的方式来统计各个 Segment 大小，如果统计的过程中，容器的 count 发生了变化，则再采用加锁的方式来统计所有 Segment 的大小。

那么 ConcurrentHashMap 是如何判断在统计的时候容器是否发生了变化呢？使用 modCount 变量，在 put , remove 和 clean 方法里操作元素前都会将变量 modCount 进行加 1，那么在统计 size 前后比较 modCount 是否发生变化，从而得知容器的大小是否发生变化。

作者介绍

方腾飞，花名清英，淘宝资深开发工程师，关注并发编程，目前在广告技术部从事无线广告联盟的开发和设计工作。个人博客：http://ifeve.com 微博：http://weibo.com/kirals 欢迎通过我的微博进行技术交流。

HotSpot 虚拟机对象探秘

作者 周志明

请读者首先注意本篇的题目中的限定语“HotSpot 虚拟机”，在虚拟机规范中明确写道：“所有在虚拟机规范之中没有明确描述的实现细节，都不应成为虚拟机设计者发挥创造性的牵绊，设计者可以完全自主决定所有规范中不曾描述的虚拟机内部细节，例如：运行时数据区的内存如何布局、选用哪种垃圾收集的算法等”。因此，本篇（整个内存篇中所有的文章）的内容会涉及到虚拟机“自主决定”的实现，我们的讨论将在 HotSpot VM 的范围内展开。同时，我也假定读者已经理解了虚拟机规范中所定义的 JVM 公共内存模型，例如运行时数据区域、栈帧结构等基础知识，如果读者对这些内容有疑问，可以先阅读《Java 虚拟机规范（JavaSE 7 Editon）》第 2 章或《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》的第 2、3 章相关内容。

对象的创建

Java 是一门面向对象的编程语言，Java 程序运行过程中每时每刻都有对象被创建出来。在语言层面上，创建对象通常（例外：克隆、反序列化）仅仅是一个 new 关键字而已，而在虚拟机中，对象（本文中讨论的对象限于普通 Java 对象，不包括数组和 Class 对象等）的创建又是怎样一个过程呢？

虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过的。如果没有，那必须先执行相应的类加载过程。

在类加载通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定（如何确定在下一节对象内存布局时再详细讲解），为对象分配空间的任务具体便等同于一块确定大小的内存从 Java 堆中划分出来，怎么划呢？假设 Java 堆中内存是绝对规整的，所有用过的内存都被放在一边，空闲的内存被放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”（Bump The Pointer）。如果 Java 堆中的内存并不是规整的，已被使用的内存和空闲的内存相互交错，那就没有办法简单的进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”（Free List）。选择哪种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。因此在使用 Serial、ParNew 等带 Compact 过程的收集器时，系统采用的分配算法是指针碰撞，而使用 CMS 这种基于 Mark-Sweep 算法的收集器时（说明一下，CMS 收集器可以通过 UseCMSCompactAtFullCollection 或 CMSFullGCsBeforeCompaction 来整理内存），就通常采用空闲列表。

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存。解决这个问题有两个方案，一种是对分配内存空间的动作进行同步——实际上虚拟机是采用

CAS 配上失败重试的方式保证更新操作的原子性；另外一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块内存，称为本地线程分配缓冲，（TLAB，Thread Local Allocation Buffer），哪个线程要分配内存，就在哪个线程的 TLAB 上分配，只有 TLAB 用完，分配新的 TLAB 时才需要同步锁定。虚拟机是否使用 TLAB，可以通过-XX:+/-UseTLAB 参数来设定。

内存分配完成之后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），如果使用 TLAB 的话，这一个工作也可以提前至 TLAB 分配时进行。这步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象的对象头（Object Header）之中。根据虚拟机当前的运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。关于对象头的具体内容，在下一节再详细介绍。

在上面工作都完成之后，在虚拟机的视角来看，一个新的对象已经产生了。但是在 Java 程序的视角看来，对象创建才刚刚开始——<init>方法还没有执行，所有的字段都为零呢。所以一般来说（由字节码中是否跟随有 invokespecial 指令所决定），new 指令之后会接着就是执行<init>方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

下面代码是 HotSpot 虚拟机 bytecodeInterpreter.cpp 中的代码片段（这个解释器实现很少机会实际使用，大部分平台上都使用模板解释器；当代码通过 JIT 编译器执行时差异就更大了。不过这段代码用于了解 HotSpot 的运作过程是没有什么问题的）。

代码清单 1：HotSpot 解释器代码片段

```
// 确保常量池中存放的是已解释的类

if (!constants->tag_at(index).is_unresolved_klass()) {

    // 断言确保是 klassOop 和 instanceKlassOop（这部分下一节介绍）

    oop entry = (klassOop) *constants->obj_at_addr(index);

    assert(entry->is_klass(), "Should be resolved klass");

    klassOop k_entry = (klassOop) entry;

    assert(k_entry->klass_part()->oop_is_instance(), "Should be instanceKlass");

    instanceKlass* ik = (instanceKlass*) k_entry->klass_part();

    // 确保对象所属类型已经经过初始化阶段

    if ( ik->is_initialized() && ik->can_be_fastpath_allocated() ) {

        // 取对象长度

        size_t obj_size = ik->size_helper();

        oop result = NULL;
```

```
        // 记录是否需要将对象所有字段置零值
        bool need_zero = !ZeroTLAB;
        // 是否在 TLAB 中分配对象
        if (UseTLAB) {
            result = (oop) THREAD->tlab().allocate(obj_size);
        }
        if (result == NULL) {
            need_zero = true;
            // 直接在 eden 中分配对象
            retry:
                HeapWord* compare_to = *Universe::heap()->top_addr();
                HeapWord* new_top = compare_to + obj_size;
                // cmpxchg 是 x86 中的 CAS 指令，这里是一个 C++方法，通过 CAS 方式分配空间，并发失败的话，
转到 retry 中重试直至成功分配为止
                if (new_top <= *Universe::heap()->end_addr()) {
                    if      (Atomic::cmpxchg_ptr(new_top,      Universe::heap()->top_addr(),
compare_to) != compare_to) {
                        goto retry;
                    }
                    result = (oop) compare_to;
                }
        }
        if (result != NULL) {
            // 如果需要，为对象初始化零值
            if (need_zero ) {
                HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
                obj_size -= sizeof(oopDesc) / oopSize;
                if (obj_size > 0 ) {
                    memset(to_zero, 0, obj_size * HeapWordSize);
                }
            }
```

```
            // 根据是否启用偏向锁，设置对象头信息
            if (UseBiasedLocking) {
                result->set_mark(ik->prototype_header());
            } else {
                result->set_mark(markOopDesc::prototype());
            }
            result->set_klass_gap(0);
            result->set_klass(k_entry);
            // 将对象引用入栈，继续执行下一条指令
            SET_STACK_OBJECT(result, 0);
            UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
        }
    }
}
```

对象的内存布局

HotSpot 虚拟机中，对象在内存中存储的布局可以分为三块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

HotSpot 虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC 分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等等，这部分数据的长度在 32 位和 64 位的虚拟机（暂不考虑开启压缩指针的场景）中分别为 32 个和 64 个 Bits，官方称它为“Mark Word”。对象需要存储的运行时数据很多，其实已经超出了 32、64 位 Bitmap 结构所能记录的限度，但是对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息，它会根据对象的状态复用自己的存储空间。例如在 32 位的 HotSpot 虚拟机中对象未被锁定的状态下，Mark Word 的 32 个 Bits 空间中的 25Bits 用于存储对象哈希码（HashCode），4Bits 用于存储对象分代年龄，2Bits 用于存储锁标志位，1Bit 固定为 0，在其他状态（轻量级锁定、重量级锁定、GC 标记、可偏向）下对象的存储内容如下表所示。

表 1 HotSpot 虚拟机对象头 Mark Word

存储内容	标志位	状态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定

指向重量级锁的指针	10	膨胀（重量级锁定）
空，不需要记录信息	11	GC 标记
偏向线程 ID、偏向时间戳、对象分代年龄	01	可偏向

对象头的另外一部分是类型指针，即是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。并不是所有的虚拟机实现都必须在对象数据上保留类型指针，换句话说查找对象的元数据信息并不一定要经过对象本身，这点我们在下一节讨论。另外，如果对象是一个 Java 数组，那在对象头中还必须有一块用于记录数组长度的数据，因为虚拟机可以通过普通 Java 对象的元数据信息确定 Java 对象的大小，但是从数组的元数据中无法确定数组的大小。

以下是 HotSpot 虚拟机 markOop.cpp 中的代码（注释）片段，它描述了 32bits 下 MarkWord 的存储状态：

```
// Bit-format of an object header (most significant first, big endian layout below):

//

// 32 bits:

// --------

// hash:25 ------------>| age:4 biased_lock:1 lock:2 (normal object)

// JavaThread*:23 epoch:2 age:4 biased_lock:1 lock:2 (biased object)

// size:32 ------------------------------------------>| (CMS free block)

// PromotedObject*:29 ---------->| promo_bits:3 ----->| (CMS promoted object)
```

接下来实例数据部分是对象真正存储的有效信息，也既是我们在程序代码里面所定义的各种类型的字段内容，无论是从父类继承下来的，还是在子类中定义的都需要记录袭来。这部分的存储顺序会受到虚拟机分配策略参数（FieldsAllocationStyle）和字段在 Java 源码中定义顺序的影响。HotSpot 虚拟机默认的分配策略为 longs/doubles、ints、shorts/chars、bytes/booleans、oops（Ordinary Object Pointers），从分配策略中可以看出，相同宽度的字段总是被分配到一起。在满足这个前提条件的情况下，在父类中定义的变量会出现在子类之前。如果 CompactFields 参数值为 true（默认为 true），那子类之中较窄的变量也可能会插入到父类变量的空隙之中。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于 HotSpot VM 的自动内存管理系统要求对象起始地址必须是 8 字节的整数倍，换句话说就是对象的大小必须是 8 字节的整数倍。对象头部分正好似 8 字节的倍数（1 倍或者 2 倍），因此当对象实例数据部分没有对齐的话，就需要通过对齐填充来补全。

对象的访问定位

建立对象是为了使用对象，我们的 Java 程序需要通过栈上的 reference 数据来操作堆上的具体对象。由于 reference 类型在 Java 虚拟机规范里面只规定了是一个指向对象的引用，并没有定义这

个引用应该通过什么种方式去定位、访问到堆中的对象的具体位置，对象访问方式也是取决于虚拟机实现而定的。主流的访问方式有使用句柄和直接指针两种。

- 如果使用句柄访问的话，Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据的具体各自的地址信息。如图 1 所示。

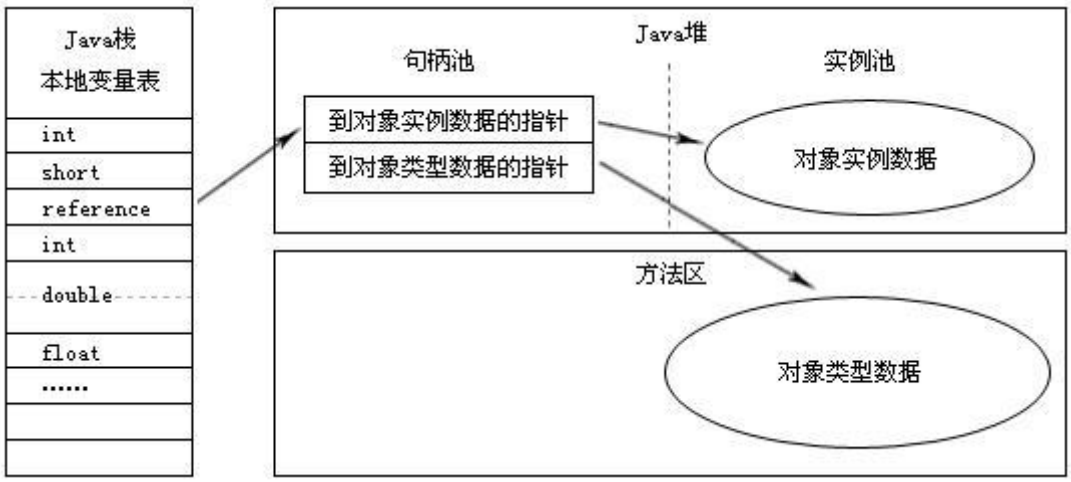


图 1 通过句柄访问对象

- 如果使用直接指针访问的话，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中存储的直接就是对象地址，如图 2 所示。

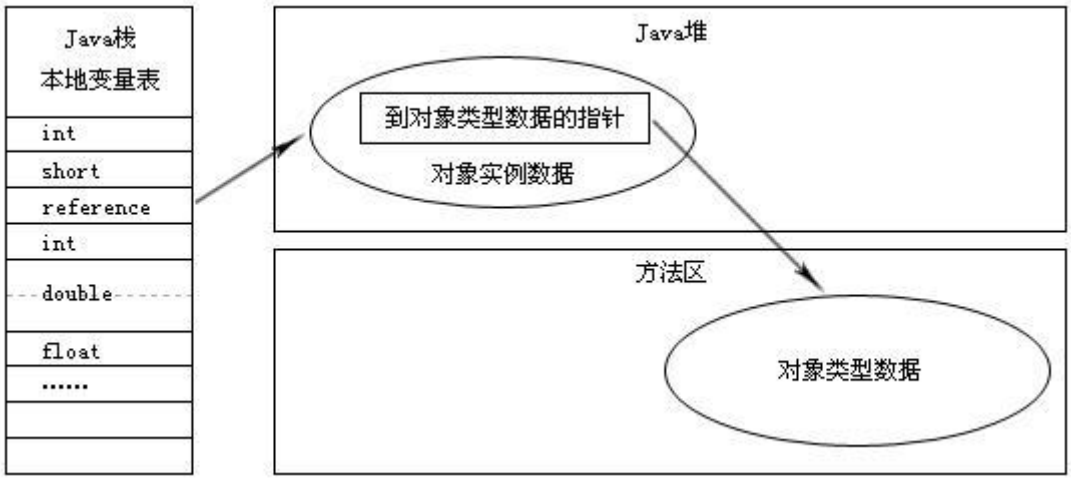


图 2 通过直接指针访问对象

这两种对象访问方式各有优势，使用句柄来访问的最大好处就是 reference 中存储的是稳定句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。

使用直接指针来访问最大的好处就是速度更快，它节省了一次指针定位的时间开销，由于对象访问的在 Java 中非常频繁，因此这类开销积小成多也是一项非常可观的执行成本。从上一部分

讲解的对象内存布局可以看出，就虚拟机 HotSpot 而言，它是使用第二种方式进行对象访问，但在整个软件开发的范围来看，各种语言、框架中使用句柄来访问的情况也十分常见。

参考资料

本文撰写时主要参考了以下资料：

- http://icyfenix.iteye.com/blog/1095132
- http://rednaxelafx.iteye.com/blog/858009
- http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html

Java 字节码忍者禁术

作者 Ben Evans 译者 邵思华

Java 语言本身是由 Java 语言规格说明（JLS）所定义的，而 Java 虚拟机的可执行字节码则是由一个完全独立的标准，即 Java 虚拟机规格说明（通常也被称为 VMSpec）所定义的。

JVM 字节码是通过 javac 对 Java 源代码文件进行编译后生成的，生成的字节码与原本的 Java 语言存在着很大的不同。比方说，在 Java 语言中为人熟知的一些高级特性，在编译过程中会被移除，在字节码中完全不见踪影。

这方面最明显的一个例子莫过于 Java 中的各种循环关键字了（for、while 等），这些关键字在编译过程中会被消除，并替换为字节码中的分支指令。这就意味着在字节码中，每个方法内部的流程控制只包含 if 语句与 jump 指令（用于循环）。

在阅读本文前，我假设读者对于字节码已经有了基本的了解。如果你需要了解一些基本的背景知识，请参考《Java 程序员修炼之道》（Well-Grounded Java Developer）一书（作者为 Evans 与 Verburg，由 Manning 于 2012 年出版），或是来自于 RebelLabs 的这篇报告（下载 PDF 需要注册）。

让我们来看一下这个示例，它对于还不熟悉的 JVM 字节码的新手来说很可能会感到困惑。该示例使用了 javap 工具，它本质上是一个 Java 字节码的反汇编工具，在下载的 JDK 或 JRE 中可以找到它。在这个示例中，我们将讨论一个简单的类，它实现了 Callable 接口：

```
public class ExampleCallable implements Callable {

    public Double call() {

        return 3.1415;

    }

}
```

我们可以通过对 javap 工具进行最简单形式的使用，对这个类进行反汇编后得到以下结果：

```
$ javap kathik/java/bytecode_examples/ExampleCallable.class

Compiled from "ExampleCallable.java"
```

```
public class kathik.java.bytecode_examples.ExampleCallable

       implements java.util.concurrent.Callable {

  public kathik.java.bytecode_examples.ExampleCallable();

  public java.lang.Double call();

  public java.lang.Object call() throws java.lang.Exception;}
```

这个反汇编后的结果看上去似乎是错误的，毕竟我们只写一个 call 方法，而不是两个。而且即使我们尝试手工创建这两个方法，javac 也会提示，代码中有两个具有相同名称和参数的方法，它们仅有返回类型的不同，因此这段代码是无法编译的。然而，这个类确确实实是由上面那个真实的、有效的 Java 源文件所生成的。

这个示例能够清晰地表明在使用 Java 中广为人知的一种限制：不可对返回类型进行重载，其实这只是 Java 语言的一种限制，而不是 JVM 字符码本身的强制要求。javac 确实会在代码中插入一些不存在于原始的类文件中的内容，如果你为此感到担忧，那大可放心，因为这种事每时每刻都在发生！每一位 Java 程序员最先学到的一个知识点就是：“如果你不提供一个构造函数，那么编译器会为你自动添加一个简单的构造函数”。在 javap 的输出中，你也能看到其中有一个构造函数存在，而它并不存在于我们的代码中。

这些额外的方法从某种程度上表明，语言规格说明的需求比 VM 规格说明中的细节更为严格。如果我们能够直接编写字节码，就可以实现许多“不可能”实现的功能，而这种字节码虽然是合法的，却没有任何一个 Java 编译器能够生成它们。

举例来说，我们可以创建出完全不含构造函数的类。Java 语言规格说明中要求每个类至少要包含一个构造函数，而如果我们在代码中没有加入构造函数，javac 会自动加入一个简单的 void 构造函数。但是，如果我们能够直接编写字节码，我们完全可以忽略构造函数。这种类是无法实例化的，即使通过反射也不行。

我们的最后一个例子已经接近成功了，但还是差一口气。在字节码中，我们可以编写一个方法，它将试图调用一个其它类中定义的私有方法。这段字节码是有效的，但如果任何程序打算加载它，它将无法正确地进行链接。这是因为在类型加载器中（classloader）的校验器会检测出这个方法调用的访问控制限制，并且拒绝这个非法访问。

介绍 ASM

如果我们打算在创建的代码中实现这些超越 Java 语言的行为，那就需要完全手动创建这样的一个类文件。由于这个类文件的格式是两进制的，因此可以选择使用某种类库，它能够让我们对某个抽象的数据结构进行操作，随后将其转换为字节码，并通过流方式将其写入磁盘。

具备这种功能的类库有多个选择，但在本文中我们将关注于 ASM。这是一个非常常见的类库，在 Java 8 分发包中有一个以内部 API 的形式提供的版本（其内容稍有不同）。对于用户代码来说，我们选择使用通用的开源类库，而不是 JDK 中提供的版本，毕竟我们不应当依赖于内部 API 来实现所需的功能。

ASM 的核心功能在于，它提供了一种 API，虽然它看上去有些神秘莫测（有时也会显得有些粗糙），但能够以一种直接的方式反映出字节码的数据结构。

我们看到的 Java 运行时是由多年之前的各种设计决策所产生的结果，而在后续各个版本的类文件格式中，我们能够清晰地看到各种新增的内容。

ASM 致力于尽量使构建的类文件接近于真实形态，因此它的基础 API 会分解为一系列相对简单的方法片段（而这些片段正是用于建模的二进制所关注的）。

如果程序员打算完全手动编写类文件，就必需理解类文件的整体结构，而这种结构是会随时改变的。幸运的是，ASM 能够处理多个不同 Java 版本中的类文件格式之间的细微差别，而 Java 平台本身对于可兼容性的高要求也侧面帮助了我们。

一个类文件依次包含以下内容：

- 某个特殊的数字（在传统的 Unix 平台上，Java 中的特殊数字是这个历史悠久的、人见人爱的 0xCAFEBABE）；
- 正在使用中的类文件格式版本号；
- 常量；
- 访问控制标记（例如类的访问范围是 public、protected 还是 package 等）；
- 该类的类型名称；
- 该类的超类；
- 该类所实现的接口；
- 该类拥有的字段（处于超类中的字段上方）；
- 该类拥有的方法（处于超类中的方法上方）；
- 属性（类级别的注解）。

可以用下面这个方法帮助你记忆 JVM 类文件中的主要部分：

ASM 中提供了两个 API，其中最简单的那个依赖于访问者模式。在常见的形式中，ASM 只包含最简单的字段以及 ClassWrite 类（当已经熟悉了 ASM 的使用和直接操作字节码的方式之后，许多开发者会发现 CheckClassAdapter 是一个很实用的起点，作为一个 ClassVisitor，它对代码进行检查的方式，与 Java 的类加载子系统中的校验器的工作方式非常想像。）

让我们看几个简单的类生成的例子，它们都是按照常规的模式创建的：

- 启动一个 ClassVisitor（在我们的示例中就是一个 ClassWriter）；
- 写入头信息；
- 生成必要的方法和构造函数；
- 将 ClassVisitor 转换为字节数组，并写入输出。

示例

```
public class Simple implements ClassGenerator {

 // Helpful constants

 private static final String GEN_CLASS_NAME = "GetterSetter";

 private static final String GEN_CLASS_STR = PKG_STR + GEN_CLASS_NAME;



 @Override

 public byte[] generateClass() {

   ClassWriter cw = new ClassWriter(0);

   CheckClassAdapter cv = new CheckClassAdapter(cw);

   // Visit the class header

   cv.visit(V1_7, ACC_PUBLIC, GEN_CLASS_STR, null, J_L_O, new String[0]);

   generateGetterSetter(cv);

   generateCtor(cv);

   cv.visitEnd();

   return cw.toByteArray();

 }



 private void generateGetterSetter(ClassVisitor cv) {

   // Create the private field myInt of type int. Effectively:

   // private int myInt;

   cv.visitField(ACC_PRIVATE, "myInt", "I", null, 1).visitEnd();
```

```
// Create a public getter method
// public int getMyInt();
MethodVisitor getterVisitor =
   cv.visitMethod(ACC_PUBLIC, "getMyInt", "()I", null, null);
// Get ready to start writing out the bytecode for the method
getterVisitor.visitCode();
// Write ALOAD_0 bytecode (push the this reference onto stack)
getterVisitor.visitVarInsn(ALOAD, 0);
// Write the GETFIELD instruction, which uses the instance on
// the stack (& consumes it) and puts the current value of the
// field onto the top of the stack
getterVisitor.visitFieldInsn(GETFIELD, GEN_CLASS_STR, "myInt", "I");
// Write IRETURN instruction - this returns an int to caller.
// To be valid bytecode, stack must have only one thing on it
// (which must be an int) when the method returns
getterVisitor.visitInsn(IRETURN);
// Indicate the maximum stack depth and local variables this
// method requires
getterVisitor.visitMaxs(1, 1);
// Mark that we've reached the end of writing out the method
getterVisitor.visitEnd();


// Create a setter
// public void setMyInt(int i);
MethodVisitor setterVisitor =
    cv.visitMethod(ACC_PUBLIC, "setMyInt", "(I)V", null, null);
setterVisitor.visitCode();
```

```
    // Load this onto the stack
    setterVisitor.visitVarInsn(ALOAD, 0);
    // Load the method parameter (which is an int) onto the stack
    setterVisitor.visitVarInsn(ILOAD, 1);
    // Write the PUTFIELD instruction, which takes the top two
    // entries on the execution stack (the object instance and
    // the int that was passed as a parameter) and set the field
    // myInt to be the value of the int on top of the stack.
    // Consumes the top two entries from the stack
    setterVisitor.visitFieldInsn(PUTFIELD, GEN_CLASS_STR, "myInt", "I");
    setterVisitor.visitInsn(RETURN);
    setterVisitor.visitMaxs(2, 2);
    setterVisitor.visitEnd();
  }

  private void generateCtor(ClassVisitor cv) {
    // Constructor bodies are methods with special name
    MethodVisitor mv =
        cv.visitMethod(ACC_PUBLIC, INST_CTOR, VOID_SIG, null, null);
    mv.visitCode();
    mv.visitVarInsn(ALOAD, 0);
    // Invoke the superclass constructor (we are basically
    // mimicing the behaviour of the default constructor
    // inserted by javac)
    // Invoking the superclass constructor consumes the entry on the top
    // of the stack.
    mv.visitMethodInsn(INVOKESPECIAL, J_L_O, INST_CTOR, VOID_SIG);
    // The void return instruction
```

```
    mv.visitInsn(RETURN);
    mv.visitMaxs(2, 2);
    mv.visitEnd();
  }

  @Override
  public String getGenClassName() {
    return GEN_CLASS_NAME;
  }
}
```

这段代码使用了一个简单的接口，用一个单一的方法生成类的字节，一个辅助方法以返回生成的类名，以及一些实用的常量：

```
interface ClassGenerator {
public byte[] generateClass();

public String getGenClassName();
// Helpful constants
public static final String PKG_STR = "kathik/java/bytecode_examples/";
public static final String INST_CTOR = "";
public static final String CL_INST_CTOR = "";
public static final String J_L_O = "java/lang/Object";
public static final String VOID_SIG = "()V";
}
```

为了驾驭生成的类，我们需要使用一个 harness 类，它叫做 Main。Main 类提供了一个简单的类加载器，并且提供了一种反射式的方式对生成类中的方法进行回调。为了简便起见，我们将生成的类定入 Maven 的目标文件夹的正确位置，让 IDE 中的 classpath 能够顺利地找到它：

```
public class Main {
public static void main(String[] args) {
    Main m = new Main();
```

```
    ClassGenerator cg = new Simple();
    byte[] b = cg.generateClass();
    try {
      Files.write(Paths.get("target/classes/" + PKG_STR +
        cg.getGenClassName() + ".class"), b, StandardOpenOption.CREATE);
    } catch (IOException ex) {
      Logger.getLogger(Simple.class.getName()).log(Level.SEVERE, null, ex);
    }
    m.callReflexive(cg.getGenClassName(), "getMyInt");
}
```

下面的类提供了一种方法，能够对受保护的defineClass()进行访问，这样一来我们就能够将一个字节数组转换为某个类对象，以便在反射中使用。

```
private static class SimpleClassLoader extends ClassLoader {
 public Class simpleDefineClass(byte[] clazzBytes) {
   return defineClass(null, clazzBytes, 0, clazzBytes.length);
 }
}
private void callReflexive(String typeName, String methodName) {
 byte[] buffy = null;
 try {
   buffy = Files.readAllBytes(Paths.get("target/classes/" + PKG_STR +
     typeName + ".class"));
   if (buffy != null) {
     SimpleClassLoader myCl = new SimpleClassLoader();
     Class newClz = myCl.simpleDefineClass(buffy);
     Object o = newClz.newInstance();
     Method m = newClz.getMethod(methodName, new Class[0]);
     if (o != null && m != null) {
```

```
        Object res = m.invoke(o, new Object[0]);
        System.out.println("Result: " + res);
      }
    }
  } catch (IOException | InstantiationException | IllegalAccessException |
          NoSuchMethodException | SecurityException |
          IllegalArgumentException | InvocationTargetException ex) {
    Logger.getLogger(Simple.class.getName()).log(Level.SEVERE, null, ex);
  }
}
```

有了这个类以后，我们只要通过细微的改动，就可以方便地测试各种不同的类生成器，以此对字节码生成器的各个方面进行探索。

实现无构造函数的类的方式也很相似。举例来说，以下这种方式可以在生成的类中仅包含一个静态字段，以及它的 getter 和 setter（生成器不会调用 generateCtor()方法）：

```
private void generateStaticGetterSetter(ClassVisitor cv) {
// Generate the static field
  cv.visitField(ACC_PRIVATE | ACC_STATIC, "myStaticInt", "I", null,
      1).visitEnd();

  MethodVisitor getterVisitor = cv.visitMethod(ACC_PUBLIC | ACC_STATIC,
                                       "getMyInt", "()I", null, null);
  getterVisitor.visitCode();
  getterVisitor.visitFieldInsn(GETSTATIC, GEN_CLASS_STR, "myStaticInt", "I");

  getterVisitor.visitInsn(IRETURN);
  getterVisitor.visitMaxs(1, 1);
  getterVisitor.visitEnd()
  MethodVisitor setterVisitor = cv.visitMethod(ACC_PUBLIC | ACC_STATIC, "setMyInt",
```

```
                                        "(I)V", null, null);

  setterVisitor.visitCode();

  setterVisitor.visitVarInsn(ILOAD, 0);

  setterVisitor.visitFieldInsn(PUTSTATIC, GEN_CLASS_STR, "myStaticInt", "I");

}

setterVisitor.visitInsn(RETURN);setterVisitor.visitMaxs(2,2);setterVisitor.visitEnd();
```

请留意一下该方法在生成时使用了 ACC_STATIC 标记，此外还请注意方法的参数是位于本地变量列表中的最前面的（这里使用的 ILOAD 0 模式暗示了这一点 —— 而在生成实例方法时，此处应该改为 ILOAD 1，这是因为实例方法中的“this”引用存储在本地变量表中的偏移量为 0）。

通过使用 javap，我们就能够确认在生成的类中确实不包括任何构造函数：

```
$ javap -c kathik/java/bytecode_examples/StaticOnly.class

public class kathik.StaticOnly {

public static int getMyInt(); Code:

0: getstatic    #11                 // Field myStaticInt:I

3: ireturn

public static void setMyInt(int); Code:

0: iload_0

1: putstatic    #11                 // Field myStaticInt:I

4: return

}
```

使用生成的类

目前为止，我们是使用反射的方式调用我们通过 ASM 所生成的类的。这有助于保持这个示例的自包含性，但在很多情况下，我们希望能够将这些代码生成在常规的 Java 文件中。要实现这一点非常简单。以下示例将生成的类保存在 Maven 的目标目录下，写法很简单：

```
$ cd target/classes

$          jar          cvf          gen-asm.jar          kathik/java/bytecode_examples/GetterSetter.class
kathik/java/bytecode_examples/StaticOnly.class

$ mv gen-asm.jar ../../lib/gen-asm.jar
```

这样一来我们就得到了一个 JAR 文件，可以作为依赖项在其它代码中使用。比方说，我们可以这样使用这个 GetterSetter 类：

```
import kathik.java.bytecode_examples.GetterSetter;

public class UseGenCodeExamples {

 public static void main(String[] args) {

   UseGenCodeExamples ugcx = new UseGenCodeExamples();

   ugcx.run();

 }



 private void run() {

   GetterSetter gs = new GetterSetter();

   gs.setMyInt(42);

   System.out.println(gs.getMyInt());

 }

}
```

这段代码在 IDE 中是无法通过编译的（因为 GetterSetter 类没有配置在 classpath 中）。但如果我们直接使用命令行，并且在 classpath 中指向正确的依赖，就可以正确地运行了：

```
$ cd ../../src/main/java/

$ javac -cp ../../../lib/gen-asm.jar kathik/java/bytecode_examples/withgen/UseGenCodeExamples.java

$ java -cp .:../../../lib/gen-asm.jar kathik.java.bytecode_examples.withgen.UseGenCodeExamples

42
```

结论

在本文中，我们通过使用 ASM 类库中所提供的简单 API，学习了完全手动生成类文件的基础知识。我们也为读者展示了 Java 语言和字节码有哪些不同的要求，并且了解到 Java 中的某些规则其实只是语言本身的规范，而不是运行时所强制的要求。我们还看到，一个正确编写的手工类文件可以直接在语言中使用，与通过 javac 生成的文件没有区别。这一点也是 Java 与其它非 Java 语言，例如 Groovy 或 Scala 进行互操作的基础。

这方面的应用还有许多高级技巧，通过本文的学习，读者应该已经掌握了基本的知识，并且能够进一步深入研究 JVM 的运行时，以及如何对它进行各种操作的技术。

作者介绍

Ben Evans 是 Java/JVM 性能分析初创公司 jClarity 的 CEO。在业余时间他是伦敦 Java 社区的领导者之一并且是 Java 社区进程执行委员会的一员。之前的项目经验包括谷歌 IPO 的性能测试，金融交易系统，为 90 年代一些最大的电影编写备受好评的网站，以及其他。

DukeScript：随处运行 Java 的新尝试

作者 Abel Avram 译者 臧秀涛

Jaroslav Tulach 是 NetBeans 的创始人和最初的架构师，Anton (Toni) Epple 则是一位 Java 咨询师和培训师，最近他们凭借 DukeScript 获得了 2014 年的 Duke 选择奖。DukeScript 这门技术希望能将 Java 带到一切客户端、移动终端或桌面，而不需要借助插件。DukeScript 这个名字有些误导性，其实它并不是一门新的脚本语言，相反它只是尝试“将 Java 放到 JavaScript 之中”，进而实现 Java 最初的愿景——“一次编写，到处运行”。

DukeScript 是这样一门技术，它支持使用 Java 和 HTML5 创建跨平台的移动和桌面应用。不同于其他将 Java 应用于服务器端的解决方案，DukeScript 将 Java 应用到了客户端，而且不依赖 Oracle 过去用于运行 Applet 的插件。该技术可以运行于 Android、iOS、桌面浏览器以及任何 HTML5/JavaScript 环境中。

Epple 向 InfoQ 解释了 DukeScript 及相关技术是如何工作的：

> 一个 DukeScript 应用的基本架构其实非常简单，包括 3 个组件：一个是 Java 虚拟机，一个是 HTML 渲染组件，再就是 DukeScript。DukeScript 将 JVM 和 HTML 组件粘合到一起，作为运行在虚拟机中的业务逻辑和用 HTML/JavaScript 编写的 UI 之间的桥梁。
>
> DukeScript 应用运行在 JVM 中，使用 HTML 渲染器显示页面。当页面加载时，DukeScript 会在内部通过 Knockout.js，将该页面的动态元素绑定到数据模型。它与典型的 Knockout.js 应用的差别在于，数据模型由 Java 对象组成，用户可以在 Java 代码中操控这些对象。利用这种方式，业务逻辑可以完全用 Java 编写，与 UI 清晰地分离开来。
>
> 在我们支持的每一个平台上，都要找到一个 JVM 和一个 WebView 组件，并将其衔接到一起。显而易见，真正的困难在于通信，因为每个平台都略有不同。

该技术支持多种场景。在桌面上，可以脱离浏览器，此时 DukeScript 用到了 JavaFX，Epple 介绍说：

> 在桌面上，我们有 Hotspot VM 和 JavaFX WebView，而且后者可以直接与 Java 交互。这也很方便调试应用。当运行在 HotSpot 上时，我们可以使用断点、表达式求值以及 IDE 提供的所有其他优秀功能来调试应用。在 WebView 中，NetBeans 可以检查 DOM 树，显示 CSS，我们可以在应用运行时动态更新页面的 HTML。

Epple 补充说，在两大主流移动平台上，DukeScript 的工作方式类似，不过使用的虚拟机和 WebView 不同：

> 在 Android 上，有 Dalvik 作为虚拟机，android.webkit.WebView 用于渲染 HTML 和执行 JavaScript。在 iOS 上，有 RoboVM（一款通过 LLVM 流水线生成机器代码的 AOT 编译器）和 NSObject.UIResponder.UiView.UIWebView。通过连接这些基本组件，我们可以在这些不同的平台上运行同样的应用。

在桌面浏览器上，Java 代码需要翻译为相应的 JavaScript 片段。这可以通过 Bck2Brwsr（Tulach 编写的一款 JVM）提前编译或即时编译。据 Epple 介绍，对于 JIT 场景，当 Web 页面加载时，Bck2Brwsr 会被加载进来，再由它来加载应用中的 Java 主类并实例化，之后是实例化 Java 数据模型，并实现与 HTML 组件的绑定。当 Java 代码执行时，Bck2Brwsr 将其翻译为 JavaScript，并在浏览器的引擎中运行。Bck2Brwsr 并不是必须的，可以用其他虚拟机替代，比如可以使用 TeaVM。

在 Windows Phone 上，可以使用与 Android 和 iOS 类似的解决方案，以 Bck2Brwsr 作为所选的 JVM，但是目前尚未测试，或许还需要更多工作。

据 Tulach 介绍，Bck2Brwsr 目前有些不足：它没有使用反射，而且“该项目的目标并非来执行现有的任何 Java 库”。它面向的是新的、需要特殊设计的受限环境。Tulach 想在以后增加很多改进，并希望得到社区的帮助：

- 使用 Closure 编译器来生成更紧凑的代码；
- 每个独立的库——ObfuscatePerLibrary；
- 通过 sammy.js 或 crossroads.js 访问多页面；
- 方法和字段支持不同的修饰符；
- 对反射的更多支持（例如，在允许的情况下不要抛出 SecurityException）；
- 没有 private 的方法/字段/构造器/类的访问；
- 可能没有字段的访问；
- 可能需要构造器的访问；
- Java 的调试器（JavaScript 的也可以）；
- 性能基准测试 Sci2000；
- 研究生成对 asm.js 而言友好的代码；
- 为所有 HTML5 元素动态生成 Java 包装器（Honza）。

该框架的另一个重要组件是 HTML APIs via Java 1.0 API（HTML/Java），这是一组用于和 HTML 页面交互的 Java API，最初是为 NetBeans 开发的。默认情况下，该 API 可以通过 JavaFX WebView 在桌面浏览器上与 HTML 交互。该 API 已经与 Knockout 做了集成，后者会提供与数据模型的绑定，所以不需要直接操作 DOM。Tulach 提到，该 API 也可以配合 Controls.js 使用，还可以添加对其他框架的支持（比如 Angular.js 等）。

HTML/Java API 可以用于从 Java 中直接调用 JavaScript，而反向的调用可以借助 JavaScriptBody 注解实现。下列代码片段就是一个例子：

```
@JavaScriptBody(args = {"x", "y"}, body = "return x + y;")

private static native int sum(int x, int y);
```

为简化针对浏览器编写的 Java 代码，并避免“冗长的 JavaBeans 模式”，Tulach 使用了 Model 注解，如下面的例子所示：

```
@Model(className="Person", properties={

   @Property(name = "firstName", type=String.class),

   @Property(name = "lastName", type=String.class)
```

```
    @Property(name = "addresses", type=Address.class, array = true)

})
```

通过 HTTP 或 WebSocket，HTML/Java API 使用 JSON 与服务器通信，这里用到了另一个注解——@OnReceive。关于这一点，Tulach 写到：

> 它会再生成一些样板化代码，因此与服务器的数据交互就只是几行代码的事了。事实上，如果比较原始的 JavaScript 示例代码的大小，就会发现这正是新的 HTML/Java API 所擅长的。用于异步 REST 或 WebSocket 通信的 Java 代码要比对应的 JavaScript 代码短。

HTML/Java API 在设计时力求做到尽可能简单，不依赖其他库，而且可以在不同的 JVM 上执行，包括 HotSpot 和 Bck2Brwsr。

Epple 还扩展了 HTML/Java 库，添加了一个 HTML5 Canvas API，以及一个基于 JavaFX Canvas API 的游戏引擎。

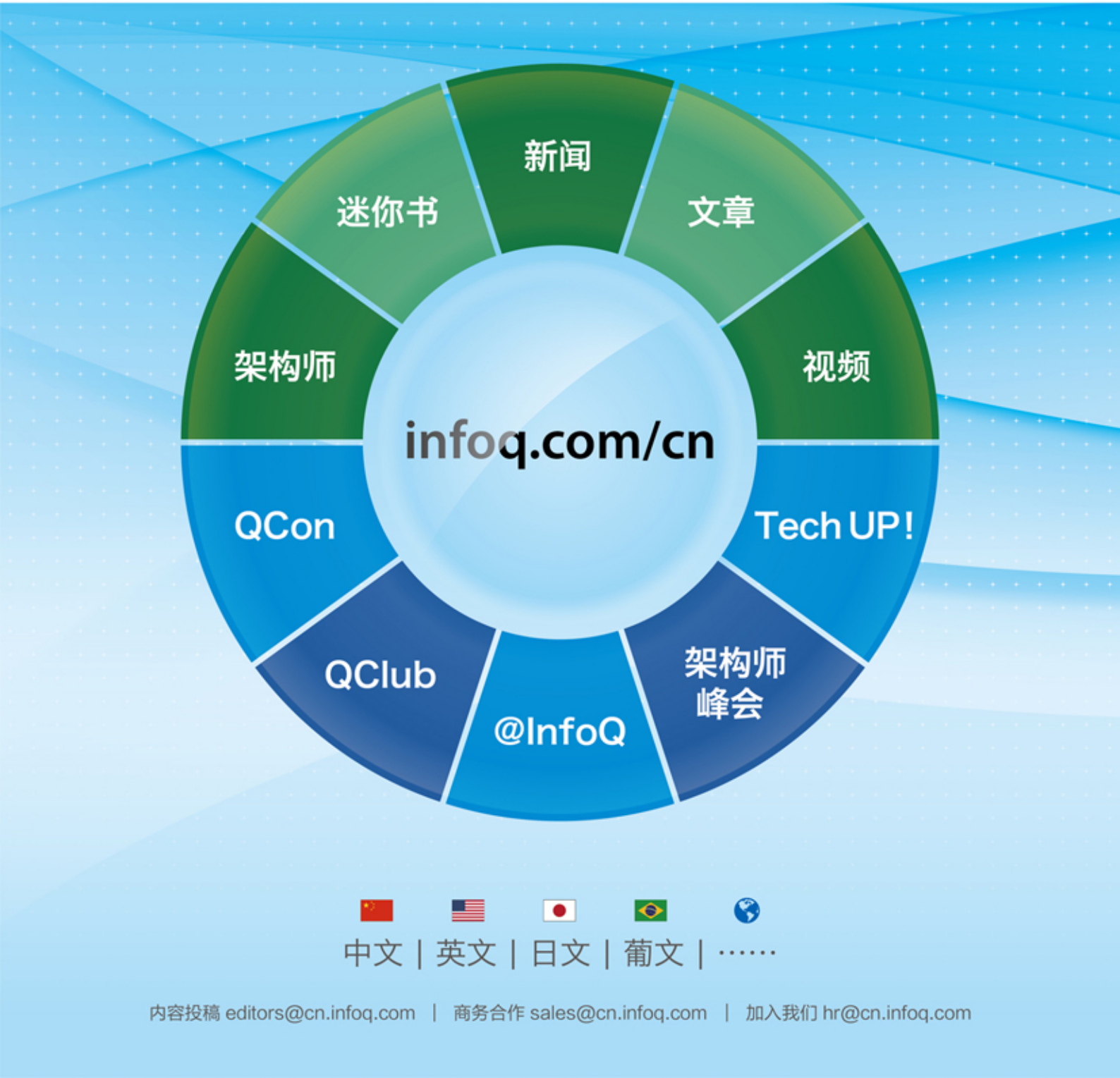
DukeScript 的网站列出了一些例子，其中包括一个简单的 HTML-Java 在线编辑器，这个编辑器还有一个 Angular.js To-Do Demo。

作者介绍

Abel Avram 从 2008 年起，在 InfoQ 参与了很多编辑工作，喜欢撰写移动、HTML、.NET、云计算和企业级架构等主题相关的新闻报道。如果您有兴趣提交新闻或者有价值的文章，可以通过邮件 abel [at] infoq.com 联系他。

InfoQueue

促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 | ……

内容投稿 editors@cn.infoq.com | 商务合作 sales@cn.infoq.com | 加入我们 hr@cn.infoq.com