

开篇词 | 以面试题为切入点，有效提升你的Java内功

2018-04-28 杨晓峰



Java是一门历史悠久的编程语言，可以毫无疑问地说，Java是最主流的编程语言之一。全球有1200万以上Java程序员以及海量的设备，还有无所不能的Java生态圈。

我所知道的诸如阿里巴巴、京东、百度、腾讯、美团、去哪儿等互联网公司，基本都是以Java为首要编程语言的。即使在最新的云计算领域，Java仍然是AWS、Google App Engine等平台上，使用最多的编程语言；甚至是微软Azure云上，Java也以微弱劣势排在前三位。所以，在这些大公司的面试中，基本都会以Java为切入点，考评一个面试者的技术能力。

应聘初级、中级Java工程师，通常只要求扎实的Java和计算机科学基础，掌握主流开源框架的使用；Java高级工程师或者技术专家，则往往全面考察Java IO/NIO、并发、虚拟机等，不仅仅是了解，更要求对底层源代码层面的掌握，并对分布式、安全、性能等领域能力有进一步的要求。

我在Oracle已经工作了近7年，负责过北京Java核心类库、国际化、分发服务等技术团队的组建，面试过从初级到非常资深的Java开发工程师。由于Java组工作任务的特点，我非常注重面试者的计算机科学基础和编程语言的理解深度。我甚至不要求面试者非要精通Java，如果对C/C++等其他语言能够掌握得非常系统和深入，也是符合需求的。

工作多年以及在面试中，我经常能体会到，有些面试者确实是认真努力工作，但坦白说表现出的能力水平却不足以通过面试，通常是两方面原因：

- “知其然不知其所以然”。做了多年技术，开发了很多业务应用，但似乎并未思考过种种技术选择背后的逻辑。坦白说，我并不放心把具有一定深度的任务交给他。更重要的是，我并不确定他未来技术成长的潜力有多大。团队所从事的是公司核心产品，工作于基础技术领域，我们不需要那些“差不多”或“还行”的代码，而是需要达到一定水准的高质量设计与实现。我相信很多其他技术团队的要求会更多、更高。
- 知识碎片化，不成系统。在面试中，面试者似乎无法完整、清晰地描述自己所开发的系统，或者使用的相关技术。平时可能埋头苦干，或者过于死磕某个实现细节，并没有抬头审视这些技术。比如，有的面试者，有一些并发编程经验，但对基本的并发类库掌握却并不扎实，似乎觉得在用的时候进行“面向搜索引擎的编程”就足够了。这种情况下，我没有信心这个面试者有高效解决复杂问题、设计复杂系统的能力。

前人已经掉过的坑，后来的同学就别再“前仆后继”了！

起初，极客时间邀请我写《Java核心技术36讲》专栏，我一开始心里是怀疑其形式和必要性的。经典的书籍一大堆呀，网上也能搜到所谓的“面试宝典”呀，为什么还需要我“指手画脚”？

但随着深入交流，我逐渐被说服了。我发现很多面试者其实是很努力的，只是

- 很难甄别出各种技术的核心与要点，技术书籍这么庞杂，对于经验有限的同学，找到高效归纳自己知识体系的方法不容易。
- 各种“宝典”更专注于问题，解答大多点到即止，甚至有些解答准确性都值得商榷，缺乏系统性的分析与举一反三的讲解。

我在极客时间推出这个专栏，就是为了让更多没有经验或者经验有限的开发者，在准备面试时：

- 少走弯路，利用有限的精力，能够更加高效地准备和学习。
- 提纲挈领，在知识点讲解的同时，为你梳理一个相对完整的Java开发技术能力图谱，将基础夯实。
- Java面试题目千奇百怪，有的面试官甚至会以黑魔法一样的态度，刨根问底JVM底层，似乎不深挖JVM源代码、不谈谈计算机指令，就是不爱学习，这是仁者见仁智者见智的事儿。我会根据自己的经验，围绕Java开发技术的方方面面，精选出5大模块，共36道题目，给出典型的回答，并层层深入剖析。

5大模块分为：

- Java基础：我会围绕Java语言基本特性和机制，由点带面，让你构建牢固的Java技术工底。
- Java进阶：将围绕并发编程、Java虚拟机等领域展开，助你攻坚大厂Java面试的核心阵地。
- Java应用开发扩展：从数据库编程、主流开源框架、分布式开发等，帮你掌握Java开发的十八般兵器。

- Java安全基础：让你理解常见的应用安全问题和处理方法，掌握如何写出符合大厂规范的安全代码。

- Java性能基础：你将掌握相关工具、方法论与基础实践。

这几年我从业系统或产品开发，切换到Java平台自身，接触了更多Java领域的核心技术，我相信我的分享能够提供一些独到的内容，而不是简单的人云亦云。

时移世易，很多大家耳熟能知的问题，其实在现代Java里已经发生了根本性的改变。在技术领域，即使你打算或已经转为技术管理等，扎实的技术功底也是必须的。希望通过我的专栏，不仅可以让你面试成功，还能帮助你未来职业发展更进一步。

万丈高楼平地起，愿我这个Java老兵，能与你一道，逐个击破大厂Java面试考点，直击Java技术核心要点，构建你的Java知识体系。



杨晓峰 Oracle 首席工程师



## 第1讲 | 谈谈你对Java平台的理解?

2018-05-05 杨晓峰



第1讲 | 谈谈你对Java平台的理解?  
杨晓峰  
- 00:00 / 08:03

从你接触Java开发到现在，你对Java最直观的印象是什么呢？是它宣传的“Write once, run anywhere”，还是目前看已经有些过于形式主义的语法呢？你对于Java平台到底了解到什么程度？请你先停下来总结思考一下。

今天我要问你的问题是，[谈谈你对Java平台的理解？“Java是解释执行”，这句话正确吗？](#)

## 典型回答

Java本身是一种面向对象的语言，最显著的特性有两个方面。一是所谓的“书写一次，到处运行”（Write once, run anywhere），能够非常容易地获得跨平台能力；另外就是垃圾收集（GC, Garbage Collection），Java通过垃圾收集器（Garbage Collector）回收分配内存，大部分情况下，程序员不需要自己操心内存的分配和回收。

我们日常会接触到JRE（Java Runtime Environment）或者JDK（Java Development Kit）。JRE，也就是Java运行环境，包含了JVM和Java类库，以及一些模块等。而JDK可以看作是JRE的一个超集，提供了更多工具，比如编译器、各种诊断工具等。

对于“Java是解释执行”这句话，这个说法不太准确。我们开发的Java的源代码，首先通过Java编译成为字节码（bytecode），然后，在运行时，通过Java虚拟机（JVM）内嵌的解释器将字节码转换成为最终的机器码。但是常见的JVM，比如我们大多数情况使用的Oracle JDK提供的Hotspot JVM，都提供了JIT（Just-In-Time）编译器，也就是通常所说的动态编译器，JIT能够在运行时将热点代码编译成机器码，这种情况下部分热点代码就属于编译执行，而不是解释执行了。

## 考点分析

其实这个问题，问得有点笼统。题目本身是非常开放的，往往考察的是多个方面，比如，基础知识理解是否很清楚；是否掌握Java平台主要模块和运行原理等。很多面试者会在这种问题上吃亏，稍微紧张了一下，不知道从何说起，就给出个很简略的回答。

对于这类笼统的问题，你需要尽量表现出自己的思维深入并系统化，Java知识理解得也比较全面，一定要避免让面试官觉得你是个“知其然不知其所以然”的人。毕竟明白基本组成和机制，是日常工作进行问题诊断或者性能调优等很多事情的基础。相信没有招聘方会不喜欢“热爱学习和思考”的面试者。

即使感觉自己回答不是非常完善，也不用担心。我个人觉得这种笼统的问题，有时候回答得稍微片面也很正常，大多数有经验的面试官，不会因为一道题就对面试者轻易地下结论。通常会尽量引导面试者，把他的真实水平展现出来。这种问题就是做个开场热身，面试官经常会根据你的回答扩展相关问题。

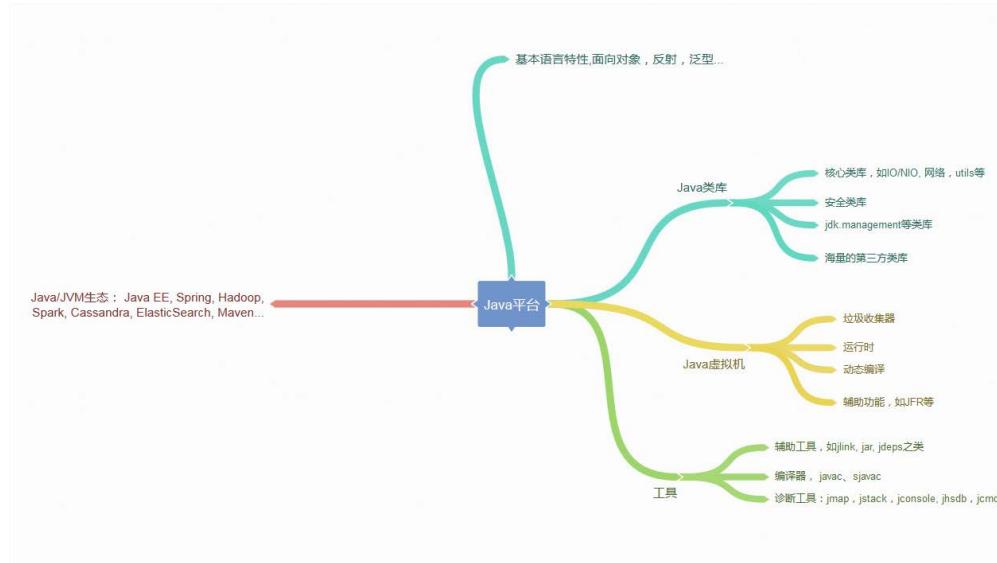
## 知识扩展

回归正题，对于Java平台的理解，可以从很多方面简明扼要地谈一下，例如：Java语言特性，包括泛型、Lambda等语言特性；基础类库，包括集合、IO/NIO、网络、并发、安全等基础类库。对于我们日常工作应用较多的类库，面试前可以系统化总结一下，有助于临场发挥。

或者谈谈JVM的一些基础概念和机制，比如Java的类加载机制，常用版本JDK（如JDK 8）内嵌的Class-Loader，例如Bootstrap、Application和Extension Class-loader；类加载大致过程：加载、验证、链接、初始化（这里参考了周志明的《深入理解Java虚拟机》，非常棒的JVM上手书籍）；自定义Class-Loader等。还有垃圾收集的基本原理，最常见的垃圾收集器，如SerialGC、Parallel GC、CMS、G1等，对于适用于什么样的工作负载最好也心里有数。这些都是可以扩展开的领域，我会在后面的专栏对此进行更系统的介绍。

当然还有JDK包含哪些工具或者Java领域内其他工具等，如编译器、运行时环境、安全工具、诊断和监控工具等。这些基本工具是日常工作效率的保证，对于我们工作在其他语言平台上，同样有所帮助，很多都是触类旁通的。

下图是我总结的一个相对宽泛的蓝图供你参考。



不再扩展了，回到前面问到的解释执行和编译执行的问题。有些面试官喜欢在特定问题上“刨根问底儿”，因为这是进一步了解面试者对知识掌握程度的有效方法，我稍微深入探讨一下。

众所周知，我们通常把Java分为编译期和运行时。这里说的Java的编译与C/C++是有着不同的意义的，`Javac`的编译，编译Java源码生成`.class`文件里面实际是字节码，而不是可以直接执行的机器码。Java通过字节码和Java虚拟机（JVM）这种跨平台的抽象，屏蔽了操作系统和硬件的细节，这也是实现“一次编译，到处执行”的基础。

在运行时，JVM会通过类加载器（Class-Loader）加载字节码，解释或者编译执行。就像我前面提到的，主流Java版本中，如JDK 8实际上是解释和编译混合的一种模式，即所谓的混合模式（-Xmixed）。通常运行在server模式的JVM，会进行上万次调用以收集足够的信息进行高效的编译，client模式这个门限是1500次。Oracle Hotspot JVM内置了两个不同的JIT compiler，C1对应前面说的client模式，适用于对于启动速度敏感的应用，比如普通Java桌面应用；C2对应server模式，它的优化是为长时间运行的服务器端应用设计的。默认是采用所谓的分层编译（TieredCompilation）。这里不再展开更多JIT的细节，没必要一下子钻进去，我会在后面介绍分层编译的内容。

Java虚拟机启动时，可以指定不同的参数对运行模式进行选择。比如，指定`-Xint`，就是告诉JVM只进行解释执行，不对代码进行编译，这种模式抛弃了JIT可能带来的性能优势。毕竟解释器（Interpreter）是逐条读入，逐条解释运行的。与其相对应的，还有一个`-Xcomp`参数，这是告诉JVM关闭解释器，不要进行解释执行，或者叫作最大优化级别。那你可能会问这种模式是不是最高效啊？简单说，还真未必。`-Xcomp`会导致JVM启动变慢非常多，同时有些JIT编译器优化方式，比如分支预测，如果不进行profiling，往往并不能进行有效优化。

除了我们日常最常见的Java使用模式，其实还有一种新的编译方式，即所谓的AOT（Ahead-of-Time Compilation），直接将字节码编译成机器代码，这样就避免了JIT预热等各方面的开销，比如Oracle JDK 9就引入了实验性的AOT特性，并且增加了新的jaotc工具。利用下面的命令把某个类或者某个模块编译成为AOT库。

```
jaotc --output libHelloWorld.so HelloWorld.class
jaotc --output libjava.base.so --module java.base
```

然后，在启动时直接指定就可以了。

```
java -XX:AOTLibrary=./libHelloWorld.so,./libjava.base.so HelloWorld
```

而且，Oracle JDK支持分层编译和AOT协作使用，这两者并不是二选一的关系。如果你有兴趣，可以参考相关文档：<http://openjdk.java.net/jeps/295>。AOT也不仅仅是只有这种方式，业界早就有第三方工具（如GCJ、Excelsior JET）提供相关功能。

另外，JVM作为一个强大的平台，不仅仅只有Java语言可以运行在JVM上，本质上合规的字节码都可以运行，Java语言自身也为这提供了便利，我们可以看到类似Clojure、Scala、Groovy、JRuby、Jython等大量JVM语言，活跃在不同的场景。

今天，我简单介绍了一下Java平台相关的一些内容，目的是提纲挈领地构建一个整体的印象，包括Java语言特性、核心类库与常用第三方类库、Java虚拟机基本原理和相关工具，希望对你有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？知道不如做到，请你也在留言区写写自己对Java平台的理解。我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



Woj

2018-05-05

“一次编译、到处运行”说的是Java语言跨平台的特性，Java的跨平台特性与Java虚拟机的存在密不可分。可在不同的环境中运行。比如说Windows平台和Linux平台都有相应的JDK。安装好JDK后也就有了Java语言的运行环境。其实Java语言本身与其他的编程语言没有特别大的差异，并不是说Java语言可以跨平台，而是在不同的平台上都可以让Java语言运行的环境而已，所以才有了Java一次编译、到处运行这样的效果。

严格的讲，跨平台的语言不止Java一种，但Java是较为成熟的一种。“一次编译、到处运行”这种效果跟编译器有关。编程语言的处理需要编译器和解释器。Java虚拟机和DOS类似，相当于一个供程序员运行的平台。

程序员从源代码到运行的三个阶段：编码——编译——运行——调试。Java在编译阶段则体现了跨平台的特点。编译过程大概是这样的：首先是将Java源代码转化成.CLASS文件字节码，这是第一次编译。.class文件就是可以到处运行的文件。然后Java字节码会被转化为目标机器代码，这是由JVM来执行的，即Java的第二次编译。

到处运行的关键前提就是JVM，因为在第一次编译时JVM起着关键作用。可以在运行Java虚拟机的地方都包含着一个JVM操作系统，从而使JAVA提供了各种不同平台上的虚拟机制，因此实现了“到处运行”的效果。需要强调的一点是，Java并不是编译机制，而是解释机制。Java字节码的设计充分考虑了JIT这一即时编译方式，可以将字节码直接转化成高性能的本地机器码，这同样也是虚拟机的一个构成部分。

作者回复

高手

magic1t4

2018-05-05

我对『Compile once, run anywhere』这个宣传语提出的历史背景非常感兴趣。这个宣传语似乎在暗示C语言有一个缺点：对于每一个不同的平台，源代码都要被编译一次。我不解的地方是，为什么这会是一个问题？不同的平台，可执行的机器码必然是不一样的。源代码自然需要依据不同的平台分别被编译。我觉得真正问题不在编译这一块，而是在C语言源文件这一块。我没有C语言的编程经验，但是似乎C语言程序经常需要调用操作系统层面的API。不同的操作系统，API一般不同。为了支持多平台，C语言程序的源文件需要根据不同平台修改多次。这应该是一个非常大的痛点。我回头查了一下当时的宣传语，原文是『Write once, run anywhere』，焦点似乎并不在编译上，而是在对源文件的修改上。

以上是自己一点不成熟的想法，还请大家指正！

作者回复

汗颜，是我记错了，非常感谢指正

三军

2018-05-05

**Java特性：**  
面向对象（封装、继承、多态）  
平台无关性（JVM运行.class文件）  
语言（泛型、Lambda）  
类库（集合、并发、网络、IO/NIO）  
JRE（Java运行环境、JVM、类库）  
JDI（Java开发工具，包括JRE、javac、诊断工具）

Java是解释运行吗？

不正确！

1. Java源代码经过Javac编译成.class文件
  2. .class文件经JVM解析或编译运行。
- (1) 解析:.class文件经JVM内部的解析器解析执行。  
(2) 编译:存在JIT编译器（Just In Time Compile 即时编译器）把经常运行的代码作为“热点代码”编译与本地平台相关的机器码，并进行各种层次的优化。  
(3) AOT编译器: Java 9提供的直接将所有代码编译成机器码执行。

作者回复

精辟

thinkers

2018-05-05

jre为java提供了必要的运行时环境，jdk为java提供了必要的开发环境！

作者回复

剧透一下，未来jre将退出历史舞台！

Jerry很恨

2018-05-05

2018-05-06

关注了好久，终于盼到了第一讲。

在看到这个题目时，我并没有立马点进来看原文，而是给了自己一些时间进行思考。

首先，个人觉得这个题目非常的抽象和笼统，这个问题没有标准答案，但是有『好』答案，而答案的好坏，完全取决于面试者自身的技术素养和对Java系统的了解。我的理解如下：

宏观角度：

跟C/C++最大的不同点在于，C/C++编程是面向操作系统的，需要开发者极大地关心不同操作系统之间的差异性；而Java平台通过虚拟机屏蔽了操作系统的底层细节，使得开发者无需过多地关心不同操作系统之间的差异性。

通过增加一个间接的中间层来进行“解耦”是计算机领域非常常用的一种“艺术手法”，虚拟机是这样，操作系统是这样，HTTP也是这样。

Java平台已经形成了一个生态系统，在这个生态系统中，有着诸多的研究领域和应用领域：  
 1. 虚拟机、编译技术的研究（例如：GC优化、JIT、AOT等）：对效率的追求是人类的另一个天性之一  
 2. Java语言本身的优化  
 3. 大数据处理  
 4. Java并发编程  
 5. 客户端开发（例如：Android平台）  
 6. ....

微观角度：  
 Java平台中有两大核心：  
 1. Java语言本身。JDK中所提供的核心类库和相关工具  
 2. Java虚拟机以及其他包含的GC

1. Java语言本身。JDK中所提供的核心类库和相关工具。  
 从事Java平台的开发，掌握Java语言、核心类库以及相关工具是必须的。我觉得这是基础中的基础。  
 >> 对语言本身的了解，需要开发者非常熟悉语言的语法结构。而Java又是一种面对对象的语言，这又需要开发者深入了解面对对象的设计理念；  
 >> Java核心类库包含集合类、线程相关类、IO、NIO、J.U.开发包等；  
 >> JDK提供的工具包含：基本的编译工具、虚拟机性能检测相关工具等。

2. Java虚拟机  
 Java语言具有跨平台的特性，也是因为虚拟机的存在。Java源文件被编译成字节码，被虚拟机加载后执行。这里隐含的意思有两层：  
 1) 大部分情况下，编译者只需要关心Java语言本身，而无需特意关心底层细节。包括对内存的分配和回收，也全权交给了GC。  
 2) 对于虚拟机而言，只要是符合规范的字节码，它们都能被加载执行。当然，能正常运行的程序光满足这点是不行的，程序本身需要保证在运行时不出异常。所以，Scala、Kotlin、Jython等语言也可以跑在虚拟机上。

围绕虚拟机的效率问题展开，将涉及到一些优化技术，例如：JIT、AOT。因为如果虚拟机加载字节码后，完全进行解释执行，这势必会影响执行效率。所以，对于这个运行环节，虚拟机会进行一些优化处理，例如JIT技术，会将某些运行特别频繁的代码编译成机器码。而AOT技术，是在运行前，通过工具直接将字节码转换为机器码。

作者回复

◆◆  
 zaiweiwoaini

看评论也能学习知识。  
 作者回复

搬个板凳，哈哈  
 欧阳田

1. JVM的内存模型，堆、栈、方法区；字节码的跨平台性；对象在JVM中的强引用，弱引用，软引用，虚引用，是否可用finalise方法救赎它？；双亲委派进行类加载，什么是双亲呢？双亲就是多条一份文档由我加载，然后你加载，这份文档在JVM中是一样的吗？；多态思想是Java需要最核心的概念，也是面向对象的行为的一个最好诠释；理解方法重载与重写与内存中的执行流程，怎么定位到这个具体方法的。2. 发展流程，JDK5(重写bug)，JDK6(商用最稳定版)，JDK7(switch的字符串支持)，JDK8(函数式编程)。一直在发展进化。3. 理解祖先类Object，它的行为是怎样与现实生活连接起来的。4. 理解23种设计模式，因为它是道与术的结合体。

作者回复

高手  
 刻苦滴涛涛

我理解的java程序执行步骤：  
 首先javac编译器将源代码编译成字节码。  
 然后vm类加载器加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度相对会比较慢。有些方法和代码块是高频率调用的，也就是所谓的热点代码，所以引进JIT技术，提前将这类字节码直接编译成本地机器码。这样类似于缓存技术，运行时再遇到这类代码直接可以执行。而不是先解释后执行。

作者回复

不错，JIT是运行时编译  
 姜亮

写个程序直接执行字节码就是解释执行。写个程序运行时把字节码动态翻译成机器码就是JIT。写个程序把java源代码直接翻译为机器码就是AOT。造个CPU直接执行字节码，字节码就是机器码。  
 作者回复

好主意，当年确实有类似项目  
 曹铮

这种基于运行分析，进行热点代码编译的设计，是因为绝大多数的程序都表现为“小部分的热点耗费了大多数的资源”吧。只有这样才能做到，在某些场景下，一个需要跑在运行时上的语言，可以比直接编译成机器码的语言更“快”。  
 作者回复

对，看到本质  
 -叶追寻

对Java平台的理解，首先想到的是Java的一些特性，比如平台无关性、面向对象、GC机制等，然后会在这几个方面去回答。平台无关性依赖于JVM，将.class文件解释为适用于操作系统的机器码。面向对象则会从封装、继承、多态这些特性去解释，具体内容就不在评论里赘述了。另外Java的内存回收机制，则涉及到Java的内存结构，堆、栈、方法区等，然后围绕什么样的对象可以回收以及回收的执行。以上是我对本题的理解，不足之处还请杨老师指出，希望通过这次学习能把Java系统的总结一下~

作者回复

非常棒，不同语言对平台无关的支持是不同的。Java是最高等级，未来也许会在效率角度出发，进行某种折衷，比如AOT  
 scott

解释执行和编译执行有何区别  
 作者回复

类比一下，一个是同声传译，一个是放录音  
 石头狮子

1. 一次编译，到处运行。JVM 层面封装了系统API，提供不同系统一致的调用行为。减少了为适配不同操作系统，不同架构带来的工作量。  
 2. 垃圾回收，降低了开发过程中需要注意内存回收的难度。降低内存泄露出现的概率。虽然也带来了一些额外开销，但是足以弥补带来的好处。合理的分代策略，提高了内存使用率。  
 3. JIT与其他编译语言相比，降低了编译时间。因为大部分代码是运行时编译，避免了冷代码在编译时也参与编译的问题。  
 提高了代码的执行效率，之前项目中使用过 lua 进行相关开发。由于 lua 是解释性语言，并非使用了 lua-JIT。开发过程中遇到，如果编写的 lua 代码是 JIT 所不支持的会导致代码性能与可编

译的相比十分低下。

作者回复

高手

公众号-Java大后端

今日文章心得: 个人理解的Java平台技术体系包括了以下几个重要组成部分:

Java程序设计语言  
各种硬件平台上的Java虚拟机  
Class文件格式  
Java API类库及相关工具  
来自商业机构和开源社区第三方Java类库

可以把Java程序设计语言、Java虚拟机、Java API类库及相关工具, 这三部分统称为JDK。JDK是用于支持Java程序开发的最小环境; 可以把Java API类库中的Java SE API子集和Java虚拟机统称为JRE, JRE是支持Java程序运行的标准环境。

提起Java, 必然会想起TA跨平台的特性, 但是跨平台重要吗? 重要! 因为可以write once, run anywhere, 这是程序员的终极梦想之一。但是跨平台重要吗? 不重要! 作为程序语言, 会更加关注TA的状态、兼容性、安全性、稳定性, 以及语言自身的与时俱进。要理解Java平台, JVM是必须要迈过去的坎, 将会看到另外的风景。

为什么我们就不能把JVM作为透明的存在呢?

勿在浮沙筑高台, 以JVM的GC为例。既然Java等诸多高级程序语言都已经实现了自动化内存管理, 那我们为什么还要去理解内存管理了? 因为当我们需要排查各种内存溢出、泄漏等底层问题时, 当垃圾收集成为我们开发的软件系统达到更高并发量、更高性能的瓶颈时, 我们就需要对这些“自动化”技术实施必要的监控与调节优化。

作者回复

对, 深入有利于解决更多有难度的工作

非常非常非常普通的中下

没有一个问题是加一个中间层解决不了的, 如果解决不了就加两个

樱花空

以下是在本节课所得到的收获, 结合TJU的内容整理了一下我个人的理解, 若有错误, 还望老师指出。  
Java首先是解释和编译混合的模式。它首先通过javac将源码编译成字节码文件class, 然后在运行的时候通过解释器或者JIT将字节码转换成最终的机器码。

只是用解释器的缺点: 抛弃了JIT可能带来的性能优势。如果代码没有被JIT编译的话, 次次运行时需要重复解析。

只用JIT的优点: 需要后台将代码编译成本地机器码。要花更多的时间, JVM启动会变慢非常多,

增加可执行代码的长度(字节码比JIT编译后的机器码小很多), 这将导致页面调度, 从而降低程序的速度。

有些JIT编译器的优化方式, 比如分支预测, 如果不进行profiling, 往往并不能进行有效优化。

因此, HotSpot采用了懒惰性评估(Lazy Evaluation)的做法, 根据二八定律, 消耗大部分系统资源的只有那一小部分的代码(热点代码), 而这也就是JIT所需要编译的部分。JVM会根据代码每次被执行的情况收集信息并相应地做出一些优化, 因此执行的次数越多, 它的速度就越快。

JDK 9引入了一种新的编译模式AOT(Ahead of Time Compilation), 它是直接将字节码编译成机器码, 这样就避免了JIT预热等各方面的开销。JDK支持分层编译和AOT协作使用。

注: JIT为方法级, 它会缓存编译过的字节码在CodeCache中, 而不需要被重复解释

作者回复

不错, 严格说我说的是oracle jdk和hotspot jvm的行为

吴有为

老师, 您好, 看了文章和大家的评论有点疑问, 程序执行的时候, 类加载器先把class文件加载到内存中, 一般情况下是解释执行, 解释器把class里的内容一行行解释为机器语言然后运行。疑问1: 每次执行class文件都需要解释整个class文件吗? 疑问2: 当new了一个对象的时候是怎么解释这个类的, 是解释整个这个类对应的class? 疑问3: JIT编译的热点代码是指class文件还是class文件的部分内容?

作者回复

我的理解不是以class为单位; JIT是方法级

Alex

评论区都是精华啊

凌

国富论中讲到, 社会的分工细化起到了提高生产力的关键作用。我觉得一次编写到处运行也是社会分工的一种模式, 他使大部分业务程序员注重领域模型的逻辑设计, 不必关心底层的实现, 使软件工程师达到了专业的人做专业的事这个高度。虽然现在掌握一门技术远远不够, 但是对于大部分业务程序员来说, 只有把精力花在最重要的地方比如领域模型的设计, 才会让业务更加流畅完善。所以我觉得JVM机制蕴含了一定的经济学原理。

张立春

任何软件问题都可以通过加一层来解决: 有了系统API就屏蔽了不同硬件的区别, 有了编译器就屏蔽了不同机器语言的区别, 有了JVM就屏蔽了不同操作系统的区别, 有了TCP/IP就屏蔽了不同系统之间通讯的差异, 有了语音识别和翻译就屏蔽了不同语言的差异。也许有一天人工智能可以直接把自然语言翻译成机器码直接生产可用的软件。

jack

Java平台包括java语言, class文件结构, jvm, api类库, 第三方库, 各种编译、监控和诊断工具等。  
Java语言是一种面向对象的高级语言, 通过平台中立的class文件格式和屏蔽底层硬件差异的jvm实现“一次编写, 到处运行”; 通过垃圾收集器管理内存的分配和回收。  
jvm通过使用class文件这种中间表示和具体语言解耦, 使得任何在源码早期编译过程中以class文件为中间表示或者能够转换成class文件的具体语言, 都能运行jvm之上, 也可以使用jvm的各种特性。  
api类库主要包括集合、I/O/NIO、网络、并发等。  
第三方库包括各种商业机构和开源社区的java库, 如spring、mybatis等。  
各种工具如javac、jconsole、jmap、jstack等。

作者回复

到位

佳人如玉巧弄心弦

关于JIT与AOT, 我想KVM更有发言权, 哈哈, 好的东西终将学习与借鉴

作者回复

行家

墨川

老师讲的很精彩受教了，评论区好多高手。赶紧拿个小本本记下来

祥子.Ken

看课程和评论涨知识了，不在此献丑。

分享别人的听课方式，课程已标明了后续的课程名称(问题)，我已先自我测试回答后续的问题，再听后续课程，会印象更深刻一点。

迎iang李

Java新手表示学习了，java的运行机制算是看明白了，但是发现还是有很多词汇不太了解。只有看高手们的文章才能发现自己的短板和不熟悉的领悟。期待后面更精彩，全面深入的讲解。感谢作者和各位大神的精彩评论！

作者回复

交流有利于提高

半日闲

编译型语言：C/C++、Pascal（Delphi）  
编译就是把源代码（高级语言，人类容易读，容易理解）转换成机器码（CPU能理解，能高效的执行）

解释型语言：JavaScript、Perl、Python、Ruby

解释就简单多了，解析源代码，并且直接执行，没有编译过程

编译程序是整体编译完了，再一次性执行。而解释程序是一边解释，一边执行

JAVA语言是一种编译型-解释型语言，同时具备编译特性和解释特性

其所指的编译过程只是将 Java文件编程成平台无关的字节码 class文件。

并不是向C一样编译成可执行的机器语言，在此请读者注意Java中所谓的“编译”和传统的“编译”的区别。

作为编译型语言，JAVA程序要被统一编译成字节码文件——文件后缀是class。此种文件在java中又称为类文件。

java文件不能再计算机上直接执行，它需要被java虚拟机翻译成本机的机器码后才能执行，而java虚拟机的翻译过程则是解释性的。

java字节码文件首先被加载到计算机内存中，然后读出一条指令，翻译一条指令，执行一条指令，该过程被称为java语言的解释执行，是由java虚拟机完成的。

以上说的是Java的解释执行，但是比如我们大多数情况使用的Hotspot JVM，都提供了动态编译器JIT，能够追踪热点代码，然后变成机器指令，这种情况下部分热点代码就属于编译执行，而不是解释执行了

其实甭管它什么解释还是编译，了解了底层的原理就行了

老猫

还在学习Java，给出啥评论，但是看文章和评论，学到不少东西

playapi

内容看了十分钟，评论看了半小时。

大熊

首先javac把源码编译成字节码(.class文件)，然后在程序运行时JVM把需要的.class文件加载内存，解释器逐行把该文件解释成机器码，在同样程序运行过程中部分热点代码可以通过JIT编译成机器码(不是运行)并存在缓存里(运行时编译保证了可移植性)，下次运行可以直接从缓存里取机器码，效率更高。  
AOT和JIT的区别还是不太理解，原文的话是 - AOT直接将字节码编译成机器代码，这样就避免了 JIT 预热等各方面的开销  
其他看不对的地方，麻烦指正

作者回复

jit是运行时才做的，需要预热才知道哪些是热点；  
aot是编译期，静态的，直接编成类似库的东西

有铭

我一直没想明白一个问题，既然jdk是jre的超集，为啥甲骨文提供的jdk安装包要同时安装他们两个。只装jdk不就行了吗，而且这一现象在所有平台均存在

作者回复

未来，可以看看9以后的版本，已经去掉jdk里面的jre，长远看单独发布一个jre的需要变小了，当然我们程序总还要用，使用jdk，或者用jlink订制runtime，也许是更好选择

Jeffrey

更新的好慢啊，大神快点

作者回复

这个节奏不是我定，抱歉

张驰

所有的类运行时解析之后，再次运行时还会重复解析吗？

作者回复

简单说，如果没有被JIT编译，是的

mongo

首先是解释执行？什么是编译执行？我采纳了这位知乎大神的说法 <https://www.zhihu.com/question/21486706/answer/18642540>。对于编译执行？是不是动态代理的实现原理字节码重组后的目标代理类的执行就是属于编译执行？包括反射的实现也是属于编译执行？这个猜想怎么验证？◆◆◆◆

作者回复

简单，Jvm启动时加上 -XX:PrintCompilation,就能把相关信息打出来

Kevin Wang

Java是一个不完全的面向对象编程语言，它基于JVM，经常被人吐槽的有两点：一是慢（其实也不算慢了）；二是代码冗长。同样基于JVM的还有Scala语言，它综合了面向对象与函数式编程，代码非常精简，而且可以无缝使用Java现有的库。

作者回复

启动时间现在有了AOT和Appcds，再加上Jigsaw提供的模块化能力，提高非常大，也许我后面补充一个这方面的文章

过往云烟

java是一次编译，各个平台都会运行的。java执行的时候不只是解释执行，有的时候会把部分代码编译成机器码。所以java是混合编译的。到了jaca9就会出来aot，直接把某个类库编译成二进制文件，这样加快运行。

ls

之前看到过一个理解：编译执行就好像做了一桌子的菜，坐到餐桌就可以吃了，而解释执行就像吃火锅，需要一边等，一边吃，所以效率会慢

对Java理解不好，有个疑问：为什么不能一开始把Java源码翻译成各个平台的机器码？这样到JVM层操作时，再选择JVM所在的平台运行其机器码，这样效率是否高点。

星火

评论区有点强势...

big

我连评论也不敢落下

李军

狭义上讲，JVM不是跨平台的，相当于对不同的操作系统做了适配？

昆少

天啊，评论里满满都是干货。我太想进步了！

成功

看了几个留言，发现点问题。首先C/C++目前也是支持动态编译，只不过是编译成DLL库。如果不在集成环境下编译Java程序，你会发现Javac编译出的.class文件类似.O文件。

倒影

Java是一种面向对象的语言，它最大的特点是跨平台性。一次运行，多处执行。Javac编译器把Java代码编译成.class字节码文件，在运行的时候通过JVM加载字节码文件进行解析针对不同的操作系统编译执行。Java通过JVM屏蔽不同操作系统的差异，让开发者不需要考虑操作系统的不同，只关注于代码，将问题统一解决。这是开发过程中常见的一种统一解决问题的方式，比如Java中的GC（垃圾回收机制），自动回收，在开发的过程中不需要考虑内存回收的问题。Java再比如我们写一个方法，处理某几种类型的问题，只需要根据传入的参数不同来做不同的处理，从而达到一个方法多处可用。

Java是一种面向对象的语言，具有三大特性

1. 封装
2. 继承
3. 多态

Mr. Bean

看文章 很充实 知识很全面 但有一点点晦涩 结果看了评论发现好多大神 心里有一点疑惑慢慢就解决了 继续努力学习

肖智坤

1. 开发  
按照Java平台的语法规则，结合平台提供的三方类库和框架开发程序，  
2. 编译  
开发完成后使用平台提供的javac工具将程序编译成.class文件，  
3. 加载  
平台通过ClassLoader加载编译成功的.class文件到JVM中  
4. 运行  
在运行时，JVM使用JIT将.class文件中的二进制字节码解释或编译（根据使用频率来判断使用解释还是编译）成计算机底层能够读取的机器代码去执行  
5. 回收  
运行期间，通过JVM的垃圾回收，根据参数使用不同的回收方法，将不再使用的内存地址中的数据清空  
6. 监控、调优  
在JVM运行时我们还可以通过平台提供的如：jmap, jconsole等进行监控诊断工具，程序的性能进行监控诊断，进行程序的诊断和调优工作

通过这次课的学习，不知道这样理解Java平台是否正确

坤坤君

评论干货满满！满足！

萧萧

“JIT能在运行时刻将字节码编译成机器码，这样热点代码就是编译运行了”，这个说法有点让人困惑，JVM解释字节码的过程不就是逐条转换为对应的机器码予以执行吗？运行时刻的编译不就是热解释运行吗？这里是说热点代码会被整块编译好了运行，无需逐条解释，提高效率吗

作者回复

JIT会缓存编译过的在codecache里

小六丶

菜鸟一个，看评论就可以学到好多知识！写了挺多年业务代码，看样子是白干了！

作者回复

怎么会，术业有专攻，换个角度总结一下，对业务开发也有帮助

George

Java平台不会强制您时刻关注内存分配（或使用第三方库来完成此工作）。它提供了开箱即用的内存管理功能。当您的Java应用程序在运行时创建一个对象实例时，JVM会自动从堆中为该对象分配内存空间——堆是一个专门留给您的程序使用的内存池。Java垃圾收集器在后台运行，跟踪记录应用程序不再需要哪些对象并从它们回收内存。这种内存处理方法称为隐式内存管理，因为它不需要您编写任何内存处理代码。

作者回复

所以Java程序员适合普罗大众写出质量可靠的应用

晴天

问题是Java平台，所以我可能会更多的回答是JVM部分，Java是解释执行的话我觉得是的，目前并没有用到JDK8。唉，虽然了解一点，但是没用过、就知道点新概念和特性

lijun

2018-05-05

2018-05-12

2018-05-11

2018-05-11

2018-05-11

2018-05-10

2018-05-10

2018-05-09

2018-05-09

2018-05-07

2018-05-05

2018-05-06

2018-05-05

2018-05-05

2018-05-05

我对java平台的理解还是很肤浅，惭愧啊。

作者回复

术业有专攻，加油◆◆

dingwood

老师，请教个有点弱智的问题。jre和jdk区别在哪？原来的理解是jdk为开发环境。环境变量的设置路径全在jdk的相关jar包。jre为运行环境，生产上运行需要jre。但实际开发当中，生产环境一般都重新安装jdk，而不是拷贝jre。jre基本对我这样的开发者绝缘。另外，既然两个环境都能跑JAVA程序，为什么安装环境的时候要弄两套？开发环境用jdk，生产环境也用jdk不就行了？据您说，jre要退出了？缘何？

作者回复

jre缺少一些开发、诊断相关的工具、类库；单独提供jre部分原因是为applet webstart之类功能的需要，现在逐步剔除了

Andy

看了这么多评论，还是没有明白解释执行和编译执行是什么意思，能不能把大道理说的简单点，举个具体的例子啥的，虽然我才学习Java 2个月，但是如果能让学习2个月人都能明白，说明做到了深入浅出不是吗？

pengshao

因为一直都是开发小项目，都是开发完，打包放在服务器上运行程序，这算是编译后执行吧？热更算是JIT吗？基本上项目需要更新都是停服，重新打包后运行，这样的话，是不是就没有利用到JIT这个技术？

作者回复

不是一回事儿，热部署之类不阻挠JIT起作用

张小伟

评论区都是大神。

superpig

C/C++程序员转型中。我理解JVM产生的用意，其实是很早年C的跨平台软件，只能靠预编译宏来实现WORA，那真是地狱一样的代码。此外，C的编译器更不存在跨平台的概念。Java加了一层字节码，Java这个编译器只把代码转换成字节码，这样在代码和编译器这一层，其实就是做了一次归一。Java的跨平台能力，主要来自于JVM，如果一个芯片平台没有JVM支持，那么javac产生的字节码肯定就无法在这个平台上运行了。JVM的思想是屏蔽掉底层硬件的差异，为开发者提供一个纯软件的平台，这样不仅简化程序运行，更简化了编程开发的过程。基于这个思想，微软也推出了自己的跨平台虚拟机.net framework，只不过近年来才真正的跨到Linux上了。既然JVM执行的是字节码，那么只能编译出字节码，就无所谓开发程序所用的高级语言，scala也在JVM上跑就是这个道理。对的，微软的sharp系列语言也是这个道理。这样看来，对上，JVM可以兼容多种高级语言，只要能编译器将这种高级语言编译成字节码就可以；对下，JVM可以兼容多个平台，只要JVM对这个平台进行过适配就可以让程序跑起来。这个思想其实就是现代软件工程里的模块化、层次化的思想，也让我想起了TCP/IP协议族里的层次模型。Everything over IP, IP over everything。在Java的世界里，JVM就是IP了。但是JVM也因此成了瓶颈，执行效率、CPU内存IO管理、进程线程等等，还有C中完全不存在的垃圾回收机制，这些事都需要JVM去做。这也是C程序员吐槽Java效率低的原因之一。不过这样也带来一个好处，就是无论我们需要定位问题、效率优化或是做什么，基本上盯着JVM就可以了。因此要写出高效的Java代码，JVM肯定是绕不过的话题。

Winston

本人将java平台认知分为基础和扩展两个部分（类似于se和ee）。

在基础部分，主要基于java语言本身，java是一门面向对象的语言，具备良好的可移植性(Write once, run anywhere.)，其提供了一系列基础的类库(io/nio/net/util/lang/math等)，能满足多线程并发、(java只提供多线程)、文件读写、网络通信等多种任务需求。同时，java运行于jvm之上，jvm提供了自动的垃圾收集、类加载等功能，一方面提高了java语言的跨平台特性，另一方面，减少使用者申请、释放内存的操作，更易操作。

在扩展方面，基于java语言，开发出许多框架与组件，应用广泛。其中Spring的IOC/AOP特性，ORM框架Mybatis，能够解决网站开发、数据库交互等多方面业务需求。而一些中间件，例如：消息中间件RocketMQ，可以应用于系统模块通信，实现模块解耦。

从中能看出java语言的广泛应用与强大的潜力。

紫豪

看文章是一种学习，看评论更是一种学习。  
每个人分析看待问题的立场不同，得出的感悟也都有独特的语言表达方式。  
很高兴能够与大家一起学习，接触的越多，才发现自己这个工作“多年”的老开发基础是如此的薄弱，希望所有人都能通过杨老师的课程得到一些启发，感谢各位的分享。

feifei

我的理解即正确，又不正确，这个说法不完全错。Java的执行过程，java源文件--> class字节码-->由jvm虚拟机来执行字节码  
执行分为2种解释与编译，在早起的jvm中是解释执行的，后来演变成编译与解释一起运行  
java之所以跨平台并不是因为java语言，是因为jvm虚拟机，在不同的平台上，比如windows与linux环境，都有配套的jvm虚拟机，在不同的平台上安装虚拟机，由虚拟机来执行中间代码即字节码，就实现了跨平台

未来大神

请问JIT为什么不把所有的字节码解释成机器码缓存起来，而要profiling，缓存少部分呢？JVM内存占用不是问题吧

作者回复

我理解这是问题，codcache大小是有限的，编译本身也会有开销

winner\_0715

判断是不是热点代码是由JVM自己判断的吗？实际项目中怎么用JIT呢

wenxueliu

开始为“解释hello world执行流程”就可以将几乎所有知识点串起来了。  
希望能解释下jvm和go的垃圾回收的区别，看到go已经在垃圾回收方面超过jvm了。

作者回复

谢谢建议：

结论可能与实际不符，go gc据我了解可以看做cms的改进版，未必如宣称那么好，要看疗效

李太阳

工作一年多了，听老师一席话，感觉自己的知识真的是太片面了

2018-05-05

2018-07-18

2018-07-18

2018-07-17

2018-07-17

2018-07-18

2018-07-13

2018-07-02

2018-07-01

2018-06-30

2018-06-29

2018-06-30

2018-06-26

2018-06-26

2018-06-21

huffmanT

2018-06-17

JAVA诞生，恰逢internet野蛮生长之时。JAVA的对网络编程支持和易用易理解促使了互联网公司的兴盛。随着时间推移更多更新的语言涌现了出来，如Python, Go等。Java的强大是因为它的生态比较完善各种开箱即用的框架使的Java成为了商业业务应用的一哥。随着Oracle的收编，对商业公司开源和知识产权的忌惮（参考Oracle和谷歌关于Android的官司），会进一步促进了OpenJDK和其它开源语言的发展。对于Coder来说建议多学几门语言，想提升对Java的理解，跟着杨老师就对了。

作者回复

谢谢，如果持续关注了openjdk，你会发现是在逐步更加开放的；关于你说的官司，建议去看看有点深度、更多事实的文章

2018-06-19

Geek\_59a17f  
Java的编译，编译 Java 源码生成“.class”文件里面实际是字节码，而不是可以直接执行的机器码。Java 通过字节码和 Java 虚拟机（JVM）这种跨平台的抽象，屏蔽了操作系统和硬件的细节，这也是实现“一次编译，到处执行”的基础。

在运行时，JVM 会通过类加载器（Class-Loader）加载字节码，解释或者编译执行

待时而发

2018-06-16

看评论也能学习知识，大牛太多了

robbin◆◆

2018-06-16

aot的这种模式我们一般在什么场景来使用。有什么标准规范这类可以参考吗？或者老师可以帮忙讲解下吗？

作者回复

当你的应用对启动时间非常敏感时，可以看Java9以后的Jdk文档

2018-06-12

风起

2018-06-10

我的理解java平台就是帮助开发者做了很多事情，从而减少开发者的重复工作。

1.集合，io, nio, 线程池等这些让我们使用，但是像我这种新手就需要多多学习。  
2.java虚拟机这块，帮助我们做了垃圾收集，因此就要学习里面的垃圾收集策略，内存模型什么的  
3.有开源框架已还有轮子的支持，比如常用的spring mybatis guava这些

改名不换人

2018-06-07

server模式下，JVM会进行上万次调用收集信息以便进行高效的编译，将其中的部门热点代码进行运行时编译。请问这上万次调用是JVM主动发生的，还是应用程序在使用过程中，JVM被动收集的呢？

作者回复

我理解是被动

2018-06-07

吴传卜

2018-06-06

果然评论区和原文一样精彩

Geek\_e848d4

2018-06-05

麻烦写英文的同时把中文也写上，不然看不懂啊◆◆

乖乖

2018-06-03

评论区 从来就不缺 神

刘辉

2018-06-02

请问JIT运行将 热点代码 编译成机器码。什么是热点代码呢？还是JIT会有预热开销。JIT预热那些东西呢？

作者回复

2018-06-03

简单说热点就是调用比较频繁的代码，预热就是收集信息找出热点

红茶君

2018-05-30

刚刚开始学JAVA，只看懂了个大概意思，有好多名词待查……评论区好厉害，涨姿势了。

萨拉赫

2018-05-29

这篇是关于Oracle计划放弃反序列化的文章<https://www.infoworld.com/article/3275924/java/oracle-plans-to-dump-risky-java-serialization.html?from=timeline&isappinstalled=0>

作者回复

2018-05-30

嗯，长久看是，Mark R.也说了没有时间表drop这个功能；后续估计会有其他改进，我不便透露或承诺什么

萨拉赫

2018-05-29

能不能讲讲java序列化/反序列化的问题。以及java很多不安全的问题都与反序列化有关。近期又看到一篇文章说java正打算放弃这个功能。希望您能权威的介绍下，谢谢。

作者回复

2018-05-29

序列化贡献了非常大比例的安全漏洞，  
我得回头查查，不见得是放弃，有取舍，这几个版本一直在为他添加filter

密码123456

2018-05-25

1.jre与jdk的关系。jre是java运行环境，有jvm和类库。而jdk是jre扩展版本，不仅仅有jre的功能还提供一些工具，比如编译器。  
2.jit是动态编译部署的工具

大月

2018-05-24

感觉自己理解的太片面了，还需要不断学习，为找工作打好基础。对于大神所说的都不太理解，我去面壁了，打卡1次

Jerwei

2018-05-24  
大神，我有个小小的疑问。Java代码编译为字节码文件后，JVM得解析器对字节码文件解析执行，是解析成了机器码吗？如果是的话，这个和JIT直接编译成机器码有啥区别，因为都从字节码变成了机器码。JIT在运行时会对热点代码进行编译，怎么指定某些代码是热点代码呢，编译热点代码的过程是提前做的吗？

c@lin@o

2018-05-20  
老师讲的很精彩，让人豁然开朗！能否指点一下是否有好的代码例子，来快速理解提升那？

日光倾城

2018-05-18  
谈谈你对Java平台的理解？Java的底层，包括JVM架构，有类加载子系统，运行时数据区：Java堆、栈、计数器、方法区、本地方法栈，以及执行引擎：JITCompiler、GC还有本地方法库部分。Java核心类库，从Java文件到最后运行出结果的整个过程的理解。上层的话Java生态圈，包括各种框架，第三方类库等。

aka peng ♦♦

2018-05-18  
Android虚拟机由最早的Dalvik VM变成了ART，关键变化就是从JIT变成了AOT。JVM一直不变AOT编译的理由是什么呢？运行时编译的好处是不是就是编译速度快点。对于client端来说，绝对倾向AOT呀。

作者回复

说的有道理，但主流是用Oracle JDK做server应用

njzy\_sb151

2018-05-17  
老师，针对“从你接触Java开发到现在，你对Java最直观的印象是什么”这个问题，我的答案是：易理解，因其是一门面向对象的编程语言，较清晰地表达了客观世界中的对象；简洁方便，因其是一门较规范的编程语言。老师，您好，我想请问一下，若您是面试者，我是求职者，这样的回复，老师您会做何反应？同时，还望老师就我的回复，给一些相关建议，谢谢老师。

作者回复

挺好，不过我的角度和应用开发有区别，可能问“简洁方便”之类，具体体现在什么地方，“规范”是什么意思？类型安全？那和“简洁”可能有冲突吗

Yang

2018-05-17  
高手在民间，看评论也有很多收获

zero

2018-05-15  
有个疑问，什么时候用的是解释执行，什么时候用的是编译执行。类加载的时候算是解释执行，方法调用时是编译执行吗？

杨振效

2018-05-14  
评论区里，依稀见到众多大神的背影

楼的空

2018-05-14  
感谢老师指点，是我疏忽了。因为平时都是使用的Oracle JDK和HotSpot就没有加上限制。佩服老师严谨的态度哈，我还需要多学习学习

邱新海

2018-05-14  
感觉语速再慢一些就更好了。

卡斯瓦德

2018-05-14  
老师你好：

1. 对于AOT还没接触，因为不是Java9，以及最新的Java10都不是长期支持的对于JDK的版本选择你有什么建议么？  
2. 关于刚才看到说.class文件每次new一个对象时都需要重新解释下？哪怕应用没有重启也要重新解释下么？那么反问下如果是JIT的那么还会每次new对象的时候去编译么？

作者回复

2018-05-14  
1. Java11会是长期支持版本；  
2. 最好有真正做JVM的人回答，虽然我认为是，但没人保证每个细节都清楚

十年后的思念

2018-05-13  
杨老师，扩展的知识多来点

作者回复

♦♦

2018-05-12  
今天也在为演唱会门票努力着

2018-05-12  
老师 那AOT编译是直接从源码编译为了机器码吗？字节码不生成了吗 这是否会影响Java通过解释+编译字节码文件达到在不同运行环境“一次编译 处处运行”的特点？

作者回复

2018-05-12  
是的，有代价

飞鸿雪浪

2018-05-12  
JRE最终取消，是不是和GoLang一样，将运行时和代码一同编译成目标机器码，来实现跨平台？

作者回复

2018-05-12  
我可能没说清楚，有了Jigsaw Jlink，自己可以定制最合适的Runtime，另外JRE最初存在部分原因是类似Web Start、Applet的需要，现在也基本进入终场了

2768zxh

2018-05-12  
看了诸位大神的评论，我一个应届毕业生瞬间感觉自己的Java知识面是那么的窄，希望能通过交流得到提升。

我们俩

2018-05-10  
从本文中得到的信息可以分为两点，Java可以一次编译，到处执行，也就是跨平台能力。第二点就是Java有垃圾回收机制，不需要人工操作回收垃圾，Java的JVM可以自动执行回收

Andy

2018-05-10  
还有作业批改，太贴心了，十分感谢！

吴昊很爱你

可否有JavaGC的详解？这个问题一直了解的很粗

2018-05-10

Mason

小白，读专栏有些吃力，能否推荐一本Java入门书籍啊？

作者回复

以前入门是Thinking In Java, core Java

林长健@Damon

我想另一个角度来回答一下问题

Java平台的宗旨是一次编写跨平台运行，其实这也是所有程序员的梦想了，如果有一门编程语言能实现计算机上的任何功能，谁不乐意呢？

然而Java在发展的过程中，初期因为硬件性能瓶颈，Java 的表现性能上比不上C语言类的性能而被诟病。很多场景用Java 是无法忍受的，但是随着摩尔定律的存在，而且Java 这类存面对象的语言又是软件合作非常需要的特性，Java 这类语言的可读性是很高的，很大程度具备了高协作特性，记得软件产业有一次危机就是因为可读性的问题，这说明代码可读性的重要性。

同时Java 也在不停地进步，Android 开发也基于Java ，看到别的同学留言说到即时编译，运行时编译肯定是一些性能影响的，不然也不会有art 虚拟机了。

可以看的出来Java 虚拟机类别很多，但它们都遵循统一Java 规范来，让开发者始终能用同一语言来进行开发。

Java 平台终极思想也是一种容器思维，容器即平台，分层的思维在软件行业真心巴不得弄成千层饼，挺有意思的，也正因为层次多，分工也就越来越多，分工多了才会有今天真的庞大的互联网协作平台，才有了今天的互联网。

作者回复

不错的总结，不过你提到的那个可以去查查，它不遵守规范的，嘿嘿

2018-05-10

吕峰品

山外有山，天外有天，能找到一扇窗是最好不过的事，期待之后的深入讲解，让我们这些泡在业务中的人，能跳出来看看，快哉快哉！

2018-05-09

上面一个朋友说Java内存模型是堆 栈 方法区，说法有误区吧，堆 栈 方法区应该是内存结构，JMM是并发里面的 主存 cpu缓存那块的东西

2018-05-09

作者回复

是的，可能说的时候上下文不同

QuincySx

jre 为什么要退出了，我有注意到 JDK 10 jre 要单独下载了

2018-05-09

Andy

1.JDK与JRE区别：JRE包括jvm虚拟机，工具包，支持模块。JDK是JRE的超集，多了编译工具。

2.Java一次编译到处运行。通常情况下，java编译成为字节码通过虚拟机来进行解释执行。但java也可以通过命令把若干个包或部分程序编译执行。如oracle hotspot中的jin特性可以把java中的包编译执行。也可以通过命令直接把java进行编译。来加快运行速度。

3.JVM有两种启动模式。c1客户端，c2服务端。可以通过java命令参数指定。服装方式启动的慢，他会收集上次服务相关的运行情况。来计算出最佳运行方式。

4.gc有4种，serialized gc.parallel gc ,m1,g1。

作者回复

细节有点不准，1不只编译；2，AOT；3，c1c2是jit compiler名字，如果深入看，不是简单的对应两种启动模式；4 m1->cms？这是hotspot目前提供的，很多其他选择

2018-05-09

Leo

老师，请问解析执行是解析成机器码吗？这和编译成机器码有什么区别？

2018-05-08

作者回复

动态编译会缓存起来，适合重复使用的场景

2018-05-08

YANGFEI

所谓的跨平台、一次编写到处运行，我觉得这个说法并不严谨，它也需要在目标运行机器上安装好jre

2018-05-08

小陈

2018-05-07

老师，AOT的存在，不是又有可能丧失跨平台特性吗？

2018-05-07

Aaron亚伦

评论区的高手真多

2018-05-07

华崧

关于JVM之前看过这样一种说法，老师帮忙看下理解。JVM其实是有3个概念的：规范，实现和实例。个人理解规范就是JVM的各中机制，比如类加载机制，垃圾回收，JMM等等，只要符合规范不同人就可以实现不同的虚拟机。也是因为规范，可以让JVM可以运行Java以外的语言，只要满足字节码规范即可。然后是实现，就如刚才所说，只要符合规范就可以。所以现在是有很多种虚拟机的，除是常用的Hotspot VM,还有JRockit,以及J9 VM等等。内部实现是有所不同的。比如关于年代，Hotspot在jdk8之前是用永久代来实现的，其他虚拟机是没有这个概念的。这就是不同的虚拟机，实现可以是不同的例子。最后是实例，这个是让Java语言跨平台的正常原因。提供了一个运行的软件环境，一个中间层，来屏蔽差异，当然，不同平台上的虚拟机也是不一样的，就算都是Hotspot。美比的话，规范就是接口，实现就是一个实现了接口的类，而实例才是一个最后new出来的对象。

作者回复

不错，文章基本是以最广泛的oracle jdk为基础

2018-05-07

Hesher

平时工作编写的主要是业务代码，对于Java基础类库、底层特性不需要太过关注也很少用得上，但对于线上服务，性能分析和调优需要懂得一些JVM知识，常用的内存和线程堆栈分析工具要掌握用来自救火。

Java发展到现在，越来越好用，生态越来越强大，已经成为了企业中最重要程序语言了。基础类库越发完善，开源框架百家争鸣，即使是JVM也有了一众顶级的上层语言。无论从市场、就业还是技术本身来说，Java平台都值得全面深入地学习。

作者回复

顶你，最大的用户群体，最广泛的社区和厂商，实际开发中语言酷不见得是足够的，很多硬性的能力才是决定因素

2018-05-07

Hidden

最近，每天都在纠结要不要离职，在这家公司一年了，感觉什么提升都没有，除了获得报酬，别的好像并没有什么收获  
作者回复

2018-05-07

问下自己 跳槽为了什么，是不是跳槽就/才能解决，如果是也不用犹豫，否则也可以发现其他选择

2018-05-07

miren6

看到作者一个回复说未来jre将退出舞台，此话怎讲?  
作者回复

2018-05-07

有了jigsaw，还有其他的一些变化，jre单独存在的必要已经没有了

2018-05-07

miren6

看到作者一个回复说未来jre将退出舞台，此话怎讲?  
作者回复

2018-05-07

有了jigsaw，利用Jlink可以定制最合适的runtime，plugin之类也逐渐退役了，jre的存在需求在降低

2018-05-07

陳新Fans

看了老师讲的觉得自己要学的东西还有很多

2018-05-07

Фшэшшух

老师看到您给同学的评论说是Jit在运行时，aot在编译时，我更懵了，老师给我说明一下，我理解的是两者都是在编译时。

2018-05-06

作者回复

Jit是运行时的动态编译，通常说的编译阶段是说javac那种，看语境

2018-05-07

Фшэшшух

JAVA的Jit是运行前将源码编译为字节码文件。但是Jit和aot都可以将字节码编译为机器码，只是Jit将一些高频代码编译，aot将所有代码编译。

2018-05-06

老师这样对嘛？

2018-05-07

那个谁

2018-05-06

语言设计是一种权衡的结果。关键是要理解解释型语言和编译型语言的适应场景和优缺点，如何扬长避短。Jit是一种对热点代码效率的优化，但有时是会反编译，这个情况不知道作者怎么看  
江东去

2018-05-06

可不可以这样理解，JIT编译器就是把运行时JVM解释执行产生的codeCache对应的热点代码，动态编译成机器码去执行。

2018-05-06

龙猫9527

笔记：前端编译器有javac，作用：将.java源文件编译成.class文件；后端编译器有JIT编译器，可以将字节码生成本地机器码；热点代码是指频繁运行的某个方法或者是方法块，为了优化热点代码的执行效率，使用JIT编译器将这些代码编译成机器码。

2018-05-06

汉影

2018-05-06

只用到jdk8，现在jdk10都要出来了。对于jdk版本的使用有什么要特别注意的地方？

2018-05-06

作者回复

看需求，有没有什么特性是至关重要的；也要考虑团队能力，对于大公司，安全往往至关重要，那就要考虑，有没有安全漏洞，有没有可靠支持

2018-05-06

天天向上

2018-05-05

编译执行是指直接将源文件编译成机器码，然后直接执行；解析执行是指解析一条指令，执行一条指令。  
JIT，其用了2个计数器，方法计数器，回边计数器，当两者之和到达一定阈值时，会直接将字节码编译成机器指令。由于JVM内存有限，所以仅仅将热点程序编译成机器码  
杨老师，有2个问题想请教下：

2018-05-05

1，我了解过的编译过程：源文件->汇编指令->二进制(机器码)，那么JAVA的编译执行过程也是会有转成“汇编指令”这一步吗？  
2，aot直接将源文件编译成机器码，那么对于不同的平台，那不是需要对源文件各编译一次，不就做不到编译一次。到处执行了？

2018-05-06

作者回复

1，我的理解Jit是通过IR（中间指令）  
2，是的AOT有代价

2018-05-06

YI

2018-05-05

曾经研究过动态程序分析（dynamic program analysis），读了很多论文，存在很多疑问，今天杨老师的开篇第一讲却让我想明白了很多问题。  
作者回复

2018-05-06

互相交流，三人行必有我师

2018-05-05

『OMG』Moshow

2018-05-05

对了，還要記得提及一個，一次編譯多次運行，也要是對應兼容的jdk版本。例如1.5編譯ok的，1.10不一定ok，jdk10可以的，jdk6不一定可以。  
作者回复

2018-05-06

向后兼容，Java算是最高等级支持，虽然不能100%

2018-05-06

会飞的毛毛虫

2018-05-05

Java平台由于jvm的存在，使得编写、编译的平台无关性，也实现了语言的无关性，使编程语言中出现了一种新语言，叫jvm语言。Java是官方的语言，随着JDK的升级，也在吸收其他语言的

长处。`lambda`, `jshell`.....高手很多，凳子已经准备好了◆◆

作者回复

2018-05-06

的确，日新月异

zhhp

2018-05-05

1、我对Java跨平台的理解是Java不像C/C++之类的语言那样是面向操作系统编程的(线程、内存管理、网络、IO等功能由操作系统来提供)，而是由Java虚拟机提供一个中立的平台，屏蔽掉操作系统之间的差异。Java语言面向这个中立的平台编程，所以能有很好的跨平台能力。

2、请问老师Java的AOT和Android的ART有关系吗？以后Java的运行时有没有可能变得很小巧，然后Java应用可以和运行时一起打包为一个单独的可执行文件？

作者回复

2018-05-06

我说的是OracleJdk的特性，其实虚拟机有各种实现，ART也是使用了一种AOT技术

代码狂徒

2018-05-05

您好，博主，我这边有看到一篇16年的关于JIT的文章，那时候还没有java9吧？上面有提到说AOT是事前编译，属于静态编译，本人理解不是很深入，所以还请博主解释一下，跟您说的这个AOT是一回事吗？

作者回复

2018-05-06

是的，只是oracle jdk9才加入，有的实现早得多

不会游泳的鱼

2018-05-05

Jit一般在什么情况下使用？Jit是JVM的扩展么？

作者回复

2018-05-06

可以看作一个模块，算是高性能JVM的标配，比如hotspot、J9

喻yj晓泥

2018-05-05

JVM虚拟机，从另一个角度来看，可不可以理解为也是一个微型的操作系统，对开发人员来说，这个操作系统提供了系统API(JDK类库)以及一些管理接口用来反应操作系统当前运行时的各项指标，包括内存、线程信息等。不知道可不可以这样去理解？

作者回复

2018-05-05

有道理，不过需要注意的是Java平台的管理相关API大部分的抽象是平台中立的

mao

2018-05-05

看评论真的涨知识，这个课程很赞

作者回复

2018-05-05

藏龙卧虎，专栏只是个大家交流的引子

十三

2018-05-05

瞬间觉得自己面对的是一片大海，作为一个小白，有一点点的兴奋和好奇

学无止境

2018-05-05

希望老师后面多讲讲高并发相关知识。谢谢

作者回复

2018-05-05

必须的

1024

2018-05-05

Java语言是跨平台的，但底下的运行环境却是平台相关的。正是Java虚拟机与字节码造就了“compile once, run anywhere”。也正是这一特性，给人错觉“Java运行慢，不如C/C++”，可以Java真的慢？答案是否定的。快慢，是相对的！Hadoop、Spark等框架恰恰是这一否定的有力支持。

作者回复

2018-05-05

Java自身在不断提高，我回头补充一个专题系统介绍提高启动速度的几大法宝，hadoop稍微不同，本质是打群架……

Libra

2018-05-05

个人观点，Java的跨平台实质是利用不同版本的JVM对底层屏蔽，这样的跨平台只是相对于用户来说的。

作者回复

2018-05-05

对，跑在不同机器的应用不需要重新编译，这也是我为什么说Java跨平台是最高标准，但也不是没有代价；类似Go等跨平台是需要重新编译的，但也带来了好处，比如更轻量级。

各有适用场合，Java有了aot，能力相对更加多样化

gesanri

2018-05-05

文章中提到aot是直接将字节码转换成机器码，如果是这样，那不是跟直接编译执行一样吗？我看周志明的书说的是aot是将Java文件直接编译成机器码，不知道哪种是对的

作者回复

2018-05-05

注意aot是编译期发生的，JIT是运行时，不是一码事

孙晓明

2018-05-05

对“Java平台”应该怎么理解，不应该是语言吗？感觉eclipse之类的是平台。

作者回复

2018-05-05

不同角度，Java不仅是语言，相关JVM、工具、类库组成的平台，才是我们日常使用的东西

sky

2018-05-05

蓝图里不应该少了concurrent包，这个面试必问。

作者回复

2018-05-05

必须的，后面有单独章节

kbrx93

2018-05-05

1. 用解释型或 XX 型语言本身就不准确，设计者眼中哪有什么型，只不过设计出来发现用某种方式运行更好用，所以 XX 型就是呈现给开发者的印象。

2. 比如你设计一个变量，你在设计的时候不会因为这个变量要设计成 static 的，所以它不能干什么，能干什么。相反，你会想，设计成 static 与 非 static 那个更好用，当然是怎么好用怎么来。

3. 理解了这一点，你就会发现，Java 的 JIT 与 AOT 之类的，都是为了好用。

作者回复

2018-05-05

实用主义

朱林朋

2018-05-05

我们还在用java7,不过前面我看新闻，java10已经出来了，9既然提出了aot这种新特性，为啥还能这么快被淘汰，它有啥缺点呢，请老师指点！

作者回复

2018-05-05

9的AOT是试验特性，能力有限，比如只支持Linux x64，java base模块等，快速发展是业界的要求，云计算等推动Java加速，后面可以考虑使用LTS版本

A小星

2018-05-05

java垃圾回收机制！是回收不再使用的堆内存嘛？那内存泄漏，gc不会主动回收这部分内存嘛

作者回复

2018-05-05

基本正确，但不只是堆，还有很多东西在堆外，后面会有详细分析

杨文奇

2018-05-05

解释器是把代码一行一行解释为二进制指令，编译器是把代码一次性编译为二进制指令，对JAVA语言来说，在执行阶段区分编译和解释，但对C语言来说，在执行阶段是直接执行二进制指令，不存在编译和解释，不知道理解对吗？

作者回复

2018-05-05

嗯，C语言直接编译为机器码，并不存在虚拟机，不存在Java的部分概念

说

2018-05-05

Jdk8已经停止维护，是不是可以直接上10了？

作者回复

2018-05-05

不是停止维护，已经免费支持了5年了啊，时间太快了，商业支持还有，是可以考虑新的版本，很多能力有大幅提升

## 第2讲 | Exception和Error有什么区别?

2018-05-08 杨晓峰



第2讲 | Exception和Error有什么区别?  
杨晓峰  
Java  
- 00:00 / 11:14

世界上存在永远不会出错的程序吗？也许这只会出现在程序员的梦中。随着编程语言和软件的诞生，异常情况就如影随形地纠缠着我们，只有正确处理好意外情况，才能保证程序的可靠性。

Java语言在设计之初就提供了相对完善的异常处理机制，这也是Java得以大行其道的原因之一，因为这种机制大大降低了编写和维护可靠程序的门槛。如今，异常处理机制已经成为现代编程语言的标配。

今天我要问你的问题是，请对比Exception和Error，另外，运行时异常与一般异常有什么区别？

## 典型回答

Exception和Error都是继承了Throwable类，在Java中只有Throwable类型的实例才可以被抛出（throw）或者捕获（catch），它是异常处理机制的基本组成类型。

Exception和Error体现了Java平台设计者对不同异常情况的分类。Exception是程序正常运行中，可以预料的意外情况，可能并且应该被捕获，进行相应处理。

Error是指在正常情况下，不大可能出现的情况，绝大部分的Error都会导致程序（比如JVM自身）处于非正常的、不可恢复状态。既然是非正常情况，所以不便也不需要捕获，常见的比如OutOfMemoryError之类，都是Error的子类。

Exception又分为可检查（checked）异常和不检查（unchecked）异常，可检查异常在源代码里必须显式地进行捕获处理，这是编译期检查的一部分。前面我介绍的不可查的Error，是Throwable不是Exception。

不检查异常就是所谓的运行时异常，类似 NullPointerException、ArrayIndexOutOfBoundsException之类，通常是可以编码避免的逻辑错误，具体根据需要来判断是否需要捕获，并不会在编译期强制要求。

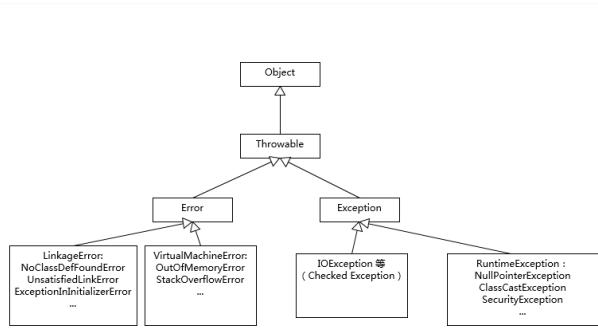
## 考点分析

分析Exception和Error的区别，是从概念角度考察了Java处理机制。总的来说，还处于理解的层面，面试者只要阐述清楚就好了。

我们在日常编程中，如何处理好异常是比较考验功底的，我觉得需要掌握两个方面。

第一，理解Throwable、Exception、Error的设计和分类。比如，掌握那些应用最为广泛的子类，以及如何自定义异常等。

很多面试官会进一步追问一些细节，比如，你了解哪些Error、Exception或者RuntimeException？我画了一个简单的类图，并列出来典型例子，可以给你作为参考，至少做到基本心里有数。



其中有些子类型，最好重点理解一下，比如`NoClassDefFoundError`和`ClassNotFoundException`有什么区别，这也是个经典的入门题目。

第二，理解Java语言中操作`Throwable`的元素和实践。掌握最基本的语法是必须的，如`try-catch-finally`块、`throw`、`throws`关键字等。与此同时，也要懂得如何处理典型场景。

异常处理代码比较繁琐，比如我们需要写很多千篇一律的捕获代码，或者在`finally`里面做一些资源回收工作。随着Java语言的发展，引入了一些更加便利的特性，比如`try-with-resources`和`multiple catch`，具体可以参考下面的代码段。在编译时期，会自动生成相应的处理逻辑，比如，自动按照约定俗成`close`那些扩展了`AutoCloseable`或者`Closeable`的对象。

```

try (BufferedReader br = new BufferedReader(...);
     BufferedWriter writer = new BufferedWriter(...)) // Try-with-resources
// do something
catch (IOException | RuntimeException e) // Multiple catch
// Handle it
}
  
```

#### 知识扩展

前面谈的大多是概念性的东西，下面我来谈些实践中的选择，我会结合一些代码用例进行分析。

先开看第一个吧，下面的代码反映了异常处理中哪些不当之处？

```

try {
    // 业务代码
    // -
    Thread.sleep(1000L);
} catch (Exception e) {
    // Ignore it
}
  
```

这段代码虽然很短，但是已经违反了异常处理的两个基本原则。

第一，尽量不要捕获类似`Exception`这样的通用异常，而是应该捕获特定异常，在这里是`Thread.sleep()`抛出的`InterruptedException`。

这是因为在日常的开发和合作中，我们读代码的机会往往超过写代码，软件工程是门协作的艺术，所以我们有义务让自己的代码能够直观地体现出尽量多的信息，而泛泛的`Exception`之类，恰恰隐藏了我们的目的。另外，我们也要保证程序不会捕获到我们不希望捕获的异常。比如，你可能更希望`RuntimeException`被扩散出来，而不是被捕获。

进一步讲，除非深思熟虑了，否则不要捕获`Throwable`或者`Error`，这样很难保证我们能够正确地处理`OutOfMemoryError`。

第二，不要生吞（swallow）异常。这是异常处理中要特别注意的事情，因为很可能会导致非常难以诊断的诡异情况。

生吞异常，往往是基于假设这段代码可能不会发生，或者感觉忽略异常是无所谓的，但是千万不要在产品代码做这种假设！

如果我们不把异常抛出来，或者也没有输出到日志（Logger）之类，程序可能在后续代码以不可控的方式结束。没人能够轻易判断究竟是哪里抛出了异常，以及是什么原因产生了异常。

再来看看第二段代码

```

try {
    // 业务代码
    // -
} catch (IOException e) {
    e.printStackTrace();
}
  
```

这段代码作为一段实验代码，它是没有任何问题的，但是在产品代码中，通常都不允许这样处理。你先思考一下这是为什么呢？

我们先来看看`printStackTrace()`的文档，开头就是“Prints this throwable and its backtrace to the standard error stream”。问题就在这里，在稍微复杂一点的生产系统中，标准输出（STERRR）不是个合适的输出选项，因为你很难判断出到底输出到哪里去了。

尤其是对于分布式系统，如果发生异常，但是无法找到堆栈轨迹（stacktrace），这纯属是为诊断设置障碍。所以，最好使用产品日志，详细地输出到日志系统里。

我们接下来看下面的代码段，体会一下**Throw early, catch late**原则。

```
public void readPreferences(String fileName){
    //...perform operations...
    InputStream in = new FileInputStream(fileName);
    //...read the preferences file...
}
```

如果fileName是null，那么程序就会抛出`NullPointerException`，但是由于没有第一时间暴露出问题，堆栈信息可能非常令人费解，往往需要相对复杂的定位。这个NPE只是作为例子，实际产品代码中，可能是各种情况，比如获取配置失败之类的。在发现问题的时候，第一时间抛出，能够更加清晰地反映问题。

我们可以修改一下，让问题“**throw early**”，对应的异常信息就非常直观了。

```
public void readPreferences(String filename) {
    Objects.requireNonNull(filename);
    //...perform other operations...
    InputStream in = new FileInputStream(filename);
    //...read the preferences file...
}
```

至于“**catch late**”，其实是我们经常苦恼的问题：捕获异常后，需要怎么处理呢？最差的处理方式，就是我前面提到的“生吞异常”，本质上其实是掩盖问题。如果实在不知道如何处理，可以选择保留原有异常的`cause`信息，直接再抛出或者构建新的异常抛出去。在更高层面，因为有了清晰的（业务）逻辑，往往更清楚合适的处理方式是什么。

有的时候，我们会根据需要自定义异常，这个时候除了保证提供足够的信息，还有两点需要考虑：

- 是否需要定义`Checked Exception`，因为这种类型设计的初衷更是为了从异常情况恢复。作为异常设计者，我们往往有充足信息进行分类。
- 在保证诊断信息足够同时，也要避免包含敏感信息，因为那样可能导致潜在的安全问题。如果我们看Java的标准类库，你可能注意到类似`java.net.ConnectException`，出错信息是类似“`Connection refused (Connection refused)`”，而不包含具体的机器名、IP、端口等，一个重要考量就是信息安全。类似的情况在日志中也有，比如，用户数据一般是不可以输出到日志里面的。

业界有一种争论（甚至可以算是某种程度的共识），Java语言的`Checked Exception`也许是个设计错误，反对者列举了几点：

- `Checked Exception`的假设是我们捕获了异常，然后恢复程序。但是，其实我们大多数情况下，根本就不可能恢复。`Checked Exception`的使用，已经大大偏离了最初的设计目的。
- `Checked Exception`不兼容`functional`编程，如果你写过`Lambda/Stream`代码，相信深有体会。

很多开源项目，已经采纳了这种实践，比如`Spring`、`Hibernate`等，甚至反映在新的编程语言设计中，比如`Scala`等。如果有兴趣，你可以参考：

<http://literatejava.com/exceptions/checked-exceptions-javas-biggest-mistake/>

当然，很多人也觉得没有必要矫枉过正，因为确实有一些异常，比如和环境相关的IO、网络等，其实是存在可恢复性的，而且Java已经通过业界的海量实践，证明了其构建高质量软件的能力。我就不再进一步解读了，感兴趣的同学可以点击[链接](#)，观看Bruce Eckel在2018年全球软件开发大会QCon的分享`Failing at Failure: How and Why We've Been Nonchalantly Moving Away From Exception Handling`。

我们从性能角度来审视一下Java的异常处理机制，这里有两个可能会相对昂贵的地方：

- try-catch代码段会产生额外的性能开销，或者换个角度说，它往往会影响JVM对代码进行优化，所以建议仅捕获必要的代码段，尽量不要一个大的try包住整段的代码；与此同时，利用异常控制代码流程，也不是一个好主意，远比我们通常意义上的条件语句（`If/else`、`switch`）要低效。
- Java每实例化一个`Exception`，都会对当时的栈进行快照，这是一个相对比较重的操作。如果发生的非常频繁，这个开销可就不能被忽略了。

所以，对于部分追求极致性能的底层类库，有种方式是尝试创建不进行栈快照的`Exception`，这本身也存在争议，因为这样做的假设在于，我创建异常时知道未来是否需要堆栈。问题是，实际上可能吗？小范围或许可能，但是在大规模项目中，这么做可能不是个理智的选择。如果需要堆栈，但又没有收集这些信息，在复杂情况下，尤其是类似微服务这种分布式系统，这会大大增加诊断的难度。

当我们的服务出现反应变慢、吞吐量下降的时候，检查发生最频繁的`Exception`也是一种思路。关于诊断后台变慢的问题，我会在后面的Java性能基础模块中系统探讨。

今天，我从一个常见的异常处理概念问题，简单总结了Java异常处理的机制。并结合代码，分析了一些普遍认可的最佳实践，以及业界最新的一些异常使用共识。最后，我分析了异常性能开销，希望对你有所帮助。

## 一课一练

关于今天我们讨论的题目你做到心中有数了吗？可以思考一个问题，对于异常处理编程，不同的编程范式也会影响到异常处理策略，比如，现在非常火热的反应式编程（`Reactive Stream`），因为其本身是异步、基于事件机制的，所以出现异常情况，决不能简单抛出去；另外，由于代码堆栈不再是同步调用那种垂直的结构，这里的异常处理和日志需要更加小心。我们看到的往往是特定`executor`的堆栈，而不是业务方法调用关系。对于这种情况，你有什么好的办法吗？

请你在留言区分享一下你的解决方案，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“**请朋友读**”，把今天的题目分享给好友，或许你能帮到他。



公号-Java大前端

2018-05-08

在Java世界里，异常的出现让我们编写的程序运行起来更加的健壮。同时为程序在调试、运行期间发生的一些意外情况，提供了补救机会；即使遇到一些严重错误而无法弥补，异常也会非常忠实地记录所发生的一切。以下是我对这篇文章心得感悟：

- 1 不要推诿或延迟处理异常，就地解决最好，并且需要实实在在的进行处理，而不是只捕捉，不动作。
- 2 一个函数尽管抛出了多个异常，但是只有一个异常可被传播到调用端。最后被抛出的异常时唯一被调用端接收的异常，其他异常都会被吞没掩盖。如果调用端要知道造成失败的最初原因，程序员就绝不能掩盖任何异常。
- 3 不要在finally代码块中处理返回值。
- 4 按照我们程序员的惯性认知：当遇到return语句的时候，执行函数会立刻返回。但是，在Java语言中，如果存在finally就会有例外。除了return语句，try代码块中的break或continue语句也可能控制权进入finally代码块。
- 5 请勿在try代码块中调用return、break或continue语句。万一无法避免，一定要确保finally的存在不会改变函数的返回值。
- 6 函数返回值有两种类型：值类型与对象引用。对于对象引用，要特别小心，如果在finally代码块中对函数返回的对象成员属性进行了修改，即使不在finally块中显式调用return语句，这个修改也会作用于返回值上。
- 7 勿将异常用于控制流。
- 8 如无必要，勿用异常。

迷途知返

2018-05-17

我比较菜 在听到"NoClassDefFoundError 和 ClassNotFoundException 有什么区别，这也是个经典的入门题目。" 这一段的时候 我以为会讲这两个的区别呢 我觉得这个区别详细讲解 就是干货 文章总结性的语言比较多 并不具体

毛毛熊

2018-05-21

NoClassDefFoundError是一个错误(Error)，而ClassNotFoundException是一个异常，在Java中对于错误和异常的处理是不同的，我们可以从异常中恢复程序但却不应该尝试从错误中恢复程序。  
ClassNotFoundException的产生原因：

Java支持使用Class.forName方法来动态地加载类，任意一个类的类名如果被作为参数传递给这个方法都将导致该类被加载到JVM内存中，如果这个类在类路径中没有被找到，那么此时就会在运行时抛出ClassNotFoundException异常。

ClassNotFoundException的产生原因主要是：  
Java支持使用反射方式在运行时动态地加载类，例如使用Class.forName方法来动态地加载类时，可以将类名作为参数传递给上述方法从而将指定类加载到JVM内存中，如果这个类在类路径中没有被找到，那么此时就会在运行时抛出ClassNotFoundException异常。

解决问题需要确保所需的类连同它的依赖包存在于类路径中，常见问题在于类名书写错误。  
另外一个导致ClassNotFoundException的原因就是：当一个类已经某个类加载器加载到内存中了，此时另一个类加载器又尝试着动态地从同一个包中加载这个类。通过控制动态类加载过程，可以避免上述情况发生。

NoClassDefFoundError产生的原因在于：  
如果JVM或者ClassLoader实例尝试加载（可以通过正常的方法调用，也可能是使用new来创建新的对象）类的时候却找不到类的定义。要查找的类在编译的时候是存在的，运行的时候却找不到了。这个时候就会导致NoClassDefFoundError。

造成这个问题的原因可能是打包过程中漏掉了部分类，或者jar包出现损坏或者篡改。解决这个问题的办法是查找那些在开发期间存在于类路径下但在运行期间却不在类路径下的类。

coder王

2018-05-08

留言中凸显高手。

钱宇祥

2018-05-08

1. 异常：这种情况下的异常，可以通过完善任务重试机制，当执行异常时，保存当前任务信息加入重试队列。重试的策略根据业务需要决定，当达到重试上限依然无法成功，记录任务执行失败，同时发出告警。

2. 日志：类似于消息中间件，处在不同线程之间的同一任务，简单高效一点的做法可能是用traceId/requestId串联。有些日志系统本身支持MDC/NDC功能，可以串联相关联的日志。

作者回复

很好的总结

欧阳田

2018-05-08

1. Error：系统错误，虚拟机出错，我们处理不了，也不需要我们来处理。

2. Exception：可以捕获的异常，且作出处理。也就是要么捕获异常并作出处理，要么继续抛出异常。

3. `RuntimeException`, 经常性出现的错误, 可以捕获, 并作出处理, 可以不捕获, 也可以不用抛出。`ArrayIndexOutOfBoundsException`像这种异常可以不捕获, 为什么呢? 在一个程序里, 使用很多数组, 如果使用一次捕获一次, 则很累。  
 4. 继承某个异常时, 重写方法时, 要么不抛出异常, 要么抛出一模一样的异常。  
 5. 当一个try后跟了很多个catch时, 必须先捕获小的异常再捕获大的异常。  
 6. 假如一个异常发生了, 控制台打印了许多行信息, 是因为程序中进行多层方法调用造成的。关键是看类型和行号。  
 7. 上传下载不能抛异常。上传下载一定要关心。  
 8. 异常不是错误。异常控制代码流程不利于代码简单易读。  
 9. try catch finally执行流程, 与 return, break, continue等混合使用注意代码执行顺序。不是不可以, 而是越是厉害的人, 代码越容易理解。

猿工匠

2018-05-08

每天早上学习与复习一下◆◆◆◆

曹铮

2018-05-08

先说问题外的话, Java的checked exception总是被诟病, 可我是从C#转到Java开发上来的, 中间经历了go, 体验过scala。我觉得Java这种机制并没有什么不好, 不同的语言体验下来, 错误与异常机制真是各有好处和缺点, 而Java我觉得处在中间, 不极端。当然老师提到lambda这确实是个问题, 至于响应式编程, 我可以泛化为异步编程的概念嘛? 一般各种异步编程框架都会对异常的传递和堆栈信息做处理吧? 比如promise/future风格的。本质上大致就是把lambda中的异常捕获并封装, 再进一步延续异步上下文, 或者转同步处理时拿到原始的错误和堆栈信息

作者回复

2018-05-08

是的, 非常棒的总结, 归根结底我们需要一堆人合作构建各种规模的程序, Java异常处理有槽点, 但实践证明了其能力; 类似第二点, 我个人也觉得可以泛化为异步编程的概念, 比如Future Stage之类使用ExecutionException的思路

Alphabet

2018-05-10

老师可以在文章末尾推荐一些基础和进阶的Java学习书籍或是资料吗? 最好是使用较新版本jdk的

莲漪

2018-05-09

非常感谢作者以及评论中的高手们! 我很喜欢作者能够精选评论。

飞云

2018-05-08

能不能讲下怎么捕捉整个项目的全局异常, 说实话前两篇干货都不多, 希望点更实在的干货

作者回复

2018-05-08

谢谢建议, 极客课程设计是尽量偏向通用场景, 我们掉坑里, 往往都不是在高大上的地方, 全局异常Spring MVC的方式就很实用; 对于干货, 你是希望特定场景, 特定问题吗? 说说你的想法

小绵羊拉拉

2018-05-08

看完文章简单认识一些浅层的意思, 但是我关注的比如try catch源码实现, 涉及以及文章中提到try catch产生堆栈快照影响jvm性能等一笔带过觉得不太过瘾。只是对于阿里的面试, 读懂这篇文章还是不够。还希望作者从面试官的角度由浅入深的剖析异常处理, 最后还是谢谢分享

作者回复

2018-05-09

谢谢反馈, 如果不做jvm或非常底层开发, 个人没有看到这些细节的实际意义, 如果非要问可以鄙视他:-)  
 创建Throwable因为要调用native方法fillInStackTrace, 至于try catch finally, jvms第三章有细节, 也可以自己写一段程序, 用javap反编译看看 goto、异常表等等

Jerry银银

2018-05-08

由于反应式编程是异步的, 基于事件的, 所以异常肯定不能直接抛出, 如果直接抛出, 随便一个异常都会引起程序崩溃, 直接影响到对后续事件处理。个人觉得一种处理方式是: 当某个事件发生异常时, 为了不影响对后续事件的处理, 可以对当前发生异常的事件进行拦截处理, 然后将异常信息发送出去。

至于发生异常时, 堆栈信息只是关于特定executor框架中的, 不知道是否可以将之前事件的“上下文”带到executor, 再传递给观察者?  
 (对反应式编程不太了解, 尝试作答^\_^)

James

2018-05-08

个人觉得checked exception / unchecked exception 分别翻译为 检查型异常/非检查型异常 更加好理解。  
 可检查异常容易被理解为可以不检查。

作者回复

2018-05-08

有道理, 谢谢指出

三军

2018-05-10

Java语言规范将派生于Error类或RuntimeException类的所有异常称为未检查(unchecked)异常, 所有其他的异常成为已检查(checked)异常。

编译器将检查是否为所有的已检查异常提供了异常处理器。

这是经典, 要好好理解。

平时我们使用throws往外抛错, 或者try-catch这类异常处理器不就是处理已检查异常吗◆◆

未检查异常就是潜在的, 编译器无需提供异常处理器进行处理。

米斯特.杜

2018-05-08

Java 每实例化一个 Exception, 都会对当时的栈进行快照, 这是一个相对比较重的操作。如果发生的非常频繁, 这个开销可就不能被忽略了。

提问: 为什么要生成快照, 什么时间销毁呢?

whhbq

2018-05-09

关于检查型异常, 因为要强制捕获或者在函数签名声明, 导致要写好多的代码。现在都改为用运行时异常, 根据业务定义好编码和消息, 然后通过全局的异常处理器处理。一些特殊的需要携带更多信息的异常, 会自定义异常类, 当然它也是运行时异常。

张世杰

2018-05-08

老师总结的类图, 对理解Throwable,Exception,Error非常的直观! 但再说掌握的两个方面, 第二方面时候, 仅仅提到懂得如何处理典型场景! 如果能详细描述一下什么样的典型场景, 会对深入理解, 使用Exception,Error非常有帮助!

五年

2018-05-24

老师讲的很好 ◆◆

不过理论讲过之后很容易忘 老师可以开一个github的代码库，将课程的最佳实践还有反例放进去吗

作者回复

有打算，最近出差，黑白颠倒，回去找机会弄下

DavidWhom佳伟

提出面试问题，却没有较好的回答，很难受( :\_- )

Cook

大佬能介绍下，线程间调用导致异常信息丢失的问题吗

作者回复

Tthead 一个UnCaughtExceptionHandler

暴走的◆◆

业务规则检验是抛异常好还是if return好

作者回复

我在文章里提到了，不建议以异常控制业务流程

fangxuan

有种浅尝辄止的感觉，希望老师能再深入一点。另外对于程序中到底是该抛出异常还是默默的处理掉，把我不好，希望老师能给点这方面的最佳实践。函数式编程中遇到checkedexception怎么处理的详细指点一下吗？

13683815260

个人想的有三步：  
1. 完善的异常记录。包括调用的上下文信息，如果在同一个进程中考虑ThreadLocal传递参数。如果分布式，把核心的参数封装传递。  
2. 在基础之上构建tracedid类的调用链跟踪。  
3. 基于回调机制，发生异常时以事件的方式通知调用方。

另 对学习的结果做个小小总结。首先二者继承体系的异同。设计里面也不同，error一般表示不可自主从异常中恢复，Exception意味着可能可以恢复。其中Exception分为两类检查异常，非检查异常。  
最佳实践，try中的代码块不宜过长，捕获时不宜大而全，finally里只释放资源不要有业务逻辑，尤其是修改返回值。用新语法可增强代码的可读性和简洁性。  
业务异常可继承runtimeexception，封装applicationexception。  
finally中的代码始终是执行的，用途为清理资源。  
对于线程池注意runtimeexception导致的线程逃逸现象。

阿修罗哇

对于日志里面我们看到的往往是特定 executor 的堆栈，而不是业务方法调用关系这种情况，我在公司推行的是自定义异常，自定义的异常有一个错误码，这个错误码需要细到某个业务的某个方法的某种错，这样排查问题会很方便，但是写的时候就比较麻烦，文档也比较多  
作者回复

嗯，有些类似trace id的思路，构建树形堆栈也有帮助

风动静泉

\*Exception 又分为可检查 (checked) 异常和不检查 (unchecked) 异常“这句话本身没问题，但是不够全面吧。查了下<<JAVA核心技术 卷 I >> 第9版, pp.474 “JAVA语言规范将派生于Error类或RuntimeException类的所有异常称为未检查(unchecked)异常，所有其他的异常称为已检查(checked)异常。”  
作者回复

没错，看描述的角度和范围，不然让人晕了

拉灯灯

error指的是不可预料的错误，可能会导致程序宕机；而exception指的是在程序运行中可以预见的异常，而异常分为检查异常与一般异常，检查异常需要在程序中显示捕获并处理，一般异常可以通过程序代码来处理，比如数组越界、空指针等；异常处理的两大基本原则：不要捕获泛泛的异常信息，比如直接捕获Exception，这样会在增加代码阅读难度，不要生吞异常，打印异常信息是一个比较重的操作，会导致程序变慢；try catch最好是包括需要检验异常的代码，不要包含过长代码，这样会降低JVM的优化效率；这是学习本节课的部分总结

zero

如果业务中有一个段业务逻辑抛出异常，不能影响后面的业务逻辑的处理，如果不用try catch 吃掉异常，还有什么好的办法处理呢？

happyhacking

提几点我觉得可以改进的地方，  
简单的内容可以简短少一些  
标题和结构感觉不够清晰  
结合实践可以更多一些

我的感受是，一篇读完，读之前模棱两可的，读之后还是不清楚，如果不回头再看一遍的话都不记得有什么内容...

对了，可以贴一些推荐的好文章和资料。总之让读者看到用心呀。

否则和网上搜到的资料水准不就差不多了~

不过本来就是基础的东西，要体现出水平还不能说太深又要实用还是挺难的。加油

somyfyu

捕获了throw或error，为啥就难以保证我们能够正确程序处理 OutOfMemoryError？

作者回复

请问抓住OOM，写什么代码处理，如果内存已经over commit，怎么保证后续逻辑可靠性，不是做不到，不大容易

不吃老鼠的猫

看了整篇文章和留言，大家都提到了不能用异常控制流程，这个我也懂，可是在项目中比如一个service方法，会对请求参数做检验，如果请求参数bean有5个属性需要检验，检验不成功，怎么处理？我目前项目中大都是如果检验不成功，就throw一个RuntimeException，如果不用这种抛异常的方式，用其他什么方式让上层调用放知道呢？如果用返回值，是不是要定义好多返回码？

作者回复

这种处理是对的，这不是通常意义的业务逻辑

石头狮子

- 无论各种异常均可以看成线程无法继续执行的信号(引发线程中断)。是否恢复执行由用户决定。信号的重要程度构成了 `Throwable` 的继承层级。
- 既然是线程无法继续执行就需要打印线程的执行状况，以便分析。
- 既然是线程中断，就可以跨调用 `catch` 与 `throw` 而不用像 `c` 等过程语言要判断每个函数的返回。方便统一 `catch` 处理。

不足之处望指出。

雷霹雳的爸爸

`executor` 出来的异常和外层逻辑的关联信息可以考虑实例化线程池时候自定义 `threadfactory` 保留一部分，比如线程名称前缀在日志里就蛮有用的。而且扩展这个 `factory` 还有一个有用的地方在于可以处理那些未捕获的异常，比如调用的底层代码的运行时异常，老师讲到的那个被很多框架推崇的一切都是运行时异常的哲学往往会让你多线程时候在这个点崴脚，不过呢，这里也有一大招，就是可以“深思熟虑”一下，考虑使用一个反模式，`Runnable` 时候 `try` 一个大的 `Throwable`，然后 `catch` 里面记个 `log` 来避免这种意外中异常被捕获的情况，坏处是老师课程里提到的执行效率的代价，而且直接对工作代码耦合，当然也可以中间加一个油漆工模式隔离下这个 `try` 块，但就没了这个反模式的唯一好处：仅看 `run` 方法上下文就知道这东西是不是已经处理了，之所以这里可以考虑违反通常对异常的最佳实践，主要是要看你处理的问题规模和粒度，这将影响最终你测量出来的性能的差异是不是足够明显，和让你团队中绝大多数人快速理解排查问题哪一个人更在意，本质上讲，如果是团队开发，应该有一致的风格，我倾向于后者，因为这个问题完全可以结合代码审查和静态代码检查工具来做，形成一个统一的团队的代码风格，如果自己干私活...一般可能就不用 `java` 了哈

梁中华

关于捕获全局异常，可以考虑使用 AOP 技术在接口入口层统一捕获，特别是使用类似 `dubbo` 这样的非 `springmvc` 架构的系统非常有用

作者回复

嗯，算是切面编程典型场景

dingsai88

物超所值啊 买的很对

作者回复

感谢认可，很高兴对大家有帮助

BUGS

`Spring cloud sleuth` 不知道是否可以解决调用跟踪问题（包括异常？）

作者回复

有帮助

Hesher

首先说说异常，虽然所有人都不建议一个大 `try-catch` 包住整个方法捕获 `exception`，但我还是不放心啊，怎么破？我的做法是对小的代码段 `try-catch` 捕获指定的异常，然后在最外面套一个大 `try-catch` 防止遗漏的情况，不知道这样就算不算一个折衷的办法，希望晓峰老师和其他同学多多指教。

响应式编程接触比较少，没什么经验，笨办法还是有。异常还是那些异常，如果把握不好捕获的位置和方法，不妨前期多加一些日志，把关键参数输出出来，从错误中反向总结异常处理方式。

作者回复

都抓起来是担心 `RuntimeException` 吗？即使抓起来，业务逻辑怎么走下去呢？  
对于日志，实用主义也不错啊

YANGFEI

`Checked Exception` 不兼容 `functional` 编程，如果你写过 `Lambda/Stream` 代码，相信深有体会。这段话可以详细剖析下吗？

作者回复

比如，写段 `lambda` 的程序试试，调用一个抛出检查异常的方法，现在语言层面并没有流畅处理的机制，往往需要自定义 `functional interface`

Dean

在使用 `RxJava` 或者有重试机制的框架时，许多调用是通过线程池方式运行，那么这时的异常只能由当前执行线程捕获，可以通过写日志的方式记录，并通过 `traceId` 与主线程关联，当然 `traceId` 的传递也是需要格外注意的。

Patrick

老师可不可以做个 `Github repo` 把样例代码可以让大家动手，谢谢老师的课

feifei

对于此场景处理分为几个部分  
1. 发生异常时，将异常信息收集到统一的日志中，不是直接的处理，然后在日志中心进行日志的查看  
2. 对于任务根据业务定义重试机制！  
3. 业务线程要独立与 reactor 线程

这是我的观点，欢迎指正，谢谢

ZK

我就想问，很多程序包括开源的都是比如参数错误，参数非法，类型错误，不进行判断，直接 `throw` 出来异常在外层进行捕获异常后整体返回出去，肯定对性能有影响吧？那么为什么很多还这样做呢？如何取舍？或者优化？

张小小的席大大

感谢老师的分享，感谢下方评论的小伙伴的全面解析

荣

`checked exception / unchecked exception` 是相对编译器 (`javac`) 而言，可检查和不可检查的到。

带着猪散步

可是我要执行一连串流程，每个中断就随时结束返给前端，并写 `mq` 接着重试，这时用异常抛出去更好控制，况且他执行到这有问题了，确实也算异常，这时可以这么搞吧

2018-05-10

2018-05-09

2018-05-09

2018-05-08

2018-05-08

2018-05-08

2018-05-08

2018-05-08

2018-05-08

2018-05-08

2018-07-17

2018-07-16

2018-06-29

2018-06-28

2018-06-27

2018-06-26

2018-06-25

张旭旭

try catch块真的会影响性能吗？我记得Java编程思想里面提到并不会影响性能，请尽情使用

2018-06-22

robbin♦♦

我们在实际开发中因为不能准确保证程序的哪里会有异常情况，一般都会将代码段包起来，同时使用`Throwable`捕获，有时很难做到每段都仔细判断异常。

2018-06-12

关于异步的部分我想是否可以像golang的模式将上下文对象传递

洛叶

PrintStackTrace 在稍微复杂一点的生产系统中，很难判断出到底输出到哪里去了，老师能否举例说明下会输出到什么地方？

2018-06-02

作者回复

默认是标准输出，具体看应用重定向到哪里

2018-06-03

韩峰

老师，关于异常，我有两点疑惑。第一点一个程序一级级的调用，是应该在最上级捕获吗，第二点，目前我公司已有的项目自定义大量的业务异常继承自`runtimeException`,比如库存不足异常，这样会不会额外增加开销，这不是一种不可理的方式

作者回复

这个建议要从业务逻辑角度看哪里负责处理，比如你提到的基础不足，我相信是业务逻辑的一部分，如何处理其实从业务设计角度是有答案的，例如，应用肯定是要给用户一个合理的反馈，而不是简单打个堆栈或不响应。

开销不用过于担心，毕竟也不是netty这种特别性能敏感的框架，还是优先考虑业务需要

2018-05-30

Jerome

如果项目自定义异常应该继承exception还是`runtimeexception`，我现在写的代码两种都有，这两种形式有什么区别 比如错误显示，记录日志。什么场景下用呢？

2018-05-28

徐金铎

从架构，或者不同模块的角度，推荐大家注意一个点，一般类似mybatis这类的框架，都会有关于exception的converter，比如mybatis会先封装`mysqlException`，然后是自己的，再封装向`springException`。类似的思想在微服务调用链路也有体现，上游服务对下游服务的error解析convert，是可以加强代码健壮性的。

2018-05-26

密码123456

2018-05-25

1. 异常的父亲，`Throwable`。
2. 异常的分类，`error`错误，`exception`异常。
3. 对异常分类的使用，`error`是JVM环境运行错误，不可进行捕获。包括，`Throwable`。`exception`是程序上的错误，需要在错误时进行捕获，恢复正常运行的形态。对异常捕获，最好进行异常类型匹配的形式，这样具有日志信息便于查询、排查。
4. 异常的使用是比较消耗性能的，消耗性能的方式有`try`代码快，与生成`exception`的堆栈快照。
5. 注意信息，在生成`exception`错误信息时，不能使用`exception`自带方法进行输出。这种方式，不清楚会输出到什么地方，不好排查。
6. 异常实践。异常分2种进行处理，一种业务异常，一种程序异常。业务异常直接抛出，程序异常，先处理一次，如果处理不了再进行抛出。

aoe

2018-05-24

请问老师，当catch 住异常时  
catch (IOException e) {  
 e.printStackTrace();  
}

1. 正确的打印异常的方式是什么？

例如使用`System.out.println(e.getMessage());`  
2. 日志的级别应该是`warn`、`error`的哪一个更合适？  
谢谢！

aoe

2018-05-24

最大的收获是明白了为什么不建议用异常控制正常业务流程，因为这种条件判断方式是低效的。具体原因：Java 每实例化一个`Exception`，都会对当时的栈进行快照，这是一个相对比较重的操作。如果发生的非常频繁，这个开销就不能被忽略了。

azhansy

2018-05-21

我们读代码的机会往往超过写代码，  
软件工程是门协作的艺术，  
优秀是一种习惯。

周红阳

2018-05-20

使用`Exception`的性能问题是两个方面：1. 创建`exception`,尤其是`fillInStackTrace`开销巨大；2. 部分编译器优化策不能使用。第一个问题解决方法有2个：1. `cache exception`,(或者`Exception`设计为单例)。他的问题是`stackTrace`打出来是错误的。2. 禁止`fillInStackTrace`操作,问题是`stackTrace`完全没有了,不方便定位问题。可以在系统上线稳定后,使用者这个办法。第二个问题可以忽略,可读性好,维护好的代码比性能重要很多,口。`try catch NullPointerException`和使用`if(ref!=null)`，使用`exception`只有`exception`实际发生的时候才会有问题一的性能问题：但是这个`if`判断每次都要执行。异常情况毕竟是极少数，平衡性问题也要考虑到。而且并不是每个代码都有编译器优化我认为：`exception`写的代码更简洁,逻辑更清晰,我喜欢用。

马婷婷

2018-05-19

NoClassDefFoundError 和 ClassNotFoundException 有什么区别

前者是找不到类的定义，类文件还在；后者是找不到`class`文件；前者可能是在类初始化的时候（比如初始化某个变量报错）发生异常；后者可能是找不到引用的类文件，可能是某个包没有导入。前者是运行时异常，后者是可检查异常

日光倾城

2018-05-19

`Exception`与`Error`都继承自`Throwable`，单纯从字面理解，前者是异常，后者是错误。`Exception`是在编码过程中可以预见的异常情况，比如数组越界，文件不存在，网络不通等。`Error`是正常情况下不会发生，一旦发生JVM也无法进行恢复的情况，比如`OutOfMemoryError`、`StackOverflowError`，这些错误情况一旦出现，JVM也无法为力了，只能停止整个进程。运行时的异常也就是非受检异常，比如说NPE，数组越界，这种异常其实是我们可以通过一些判断避免的，比如非空判断，数组长度判断，如果我们没做判断抛出了这类异常，说明我们的程序不严谨。一般异常就是在编码阶段无法预知但是也无法避免的异常情况，比如网络不通之类，所以我们可以提前进行预防，发生这种情况我们如何处理？并尝试从这类异常中恢复，至少能给出良好的提示。

日光倾城

2018-05-19

`Exception`与`Error`都继承自`Throwable`，单纯从字面理解，前者是异常，后者是错误。`Exception`是在编码过程中可以预见的异常情况，比如数组越界，文件不存在，网络不通等。`Error`是正

常情况下不会发生，一旦发生jvm也无法进行恢复的情况，比如`OutOfMemoryError, StackOverflowError`。这些错误情况一旦出现，jvm也无法为力了，只能停止整个进程。  
运行时异常也就是非受检异常，比如`NPE`，数组越界，这种异常其实是我们可以通过一些判断避免的，比如非空判断，数组长度判断。如果我们没做判断抛出了这类异常，说明我们的程序不严谨。一般异常就是我们在编码阶段无法预知但是也无法避免的异常情况，比如网络不通之类，所以我们可以提前进行预防。发生这种情况我们如何处理？并尝试从这类异常中恢复，至少能给出良好的提示。

老男孩

每课都提出课后问题，这种反馈式的学习方法有助于加深理解。而且课后的留言还是主题的一个很好补充。◆◆◆◆

2018-05-18

New Yorker

两点疑问：  
1. checked exception 和 unchecked exception 是 Exception 的子类吗？如果不是，如何自定义一个unchecked exception？  
2. 如何控制一个Exception 在实例化时不进行栈快照？

2018-05-16

作者回复

1. 狭义上是，Error也是unchecked类  
2. 比如cache一个，或者自己实现个，去掉对应逻辑

戴

留言区的一些留言，语言表达能力好差，很难看懂。

2018-05-16

戴

多看看评论，感觉评论比原文有价值。评论里更像一个专题交流会，从中可以看到大家日常工作实践中面临的痛点

2018-05-16

LBF

小白留言：  
捕获不了异常的捕获不是好捕获。  
什么异常都捕获的捕获不是好捕获。  
解决了异常的捕获一定是坏捕获。  
捕获不是俄罗斯转盘，小概率事件会致命！

2018-05-16

先天专注不服  
同样有一个问题，在spring mvc中，就提供了`controllerAdvice`注解，在业务流程中抛出相应的异常，在`controllerAdvice`中通过异常类型，完成相应的message输出。这个本质上也是通过异常完成对流程控制。而且能让业务流程很干净整洁。但是我一直很纠结，按文中说法，这是比较低效的流程控制，请问要如何取舍？

作者回复

区分它是用来作为业务流转，还是利用切面编程做异常处理？  
再说所谓实践，本就是仁者见仁的事

落叶飞逝的恋

只看到理论介绍，和一些实验代码，没看到生产代码实践示范？还比如，如何合理自定义异常的示范？

2018-05-14

末日没有进行曲

2018-05-13

问一下老师如何能更好地消化这些知识呢？感觉看过一遍懂，但是自己说却不能完整表述出来。多看？

2018-05-13

ls

老师您好，我是做Android 客户端的，对于网络请求回来的bean 解释及后续处理，我们的做法也是把一整段给try-catch，即使有异常没有被处理，也不会导致客户端崩溃。像客户端这种频率不是很大的，对性能影响会怎么样？

另外也很希望能讲解下自定义异常的实践，或者如何去设计一个合适的异常处理机制。

作者回复

“过早优化是万恶之源”，当然是保证应用可靠性更重要

2018-05-14

davidimu

2018-05-12

RxJava中有一些异常处理的方法 比如`doOnError` `switchOnError` 可以选择把异常包起来再抛出去来保留上下文

作者回复

2018-05-14

Rxjava我了解有限，如果我没理解错，那是reactive编程注册异常listener的方式，jdk最新版本websocket api也是使用类似机制，虽然细节有区别

云帆

2018-05-12

有些吃力呀

清风

2018-05-12

我有一个问题想请教一下，目前公司里的异常处理方式是直接用try catch 把整个逻辑块都包住，不管是controller层还是service层，这种代码到处可见，虽然觉得这样不太好，但确没有想到更好的解决方案，因为这样确实比较容易排查问题。而且目前使用的是微服务式开发，每个功能可能都由不同人开发，服务调用采用不信任原则也是有道理的，所以想请问一下，目前有关于在这样情况下比较好的异常处理方案？补充一下，公司用的是spring boot框架。

作者回复

保证可靠性是最重要的；这种情况使用了`spring exception handler`吗？

2018-05-14

自在

2018-05-11

我记得栈帧里面异常块与代码块是分离的，如果不出现异常好像是不影响性能的吧...

作者回复

2018-05-11

也会影响优化

2018-05-10

在忽略try-catch和流程控制时的性能消耗下，前者的控制能力更强一些吧？一些业务级异常都是交给顶层的`ExceptionHandler`处理的，写一个整体处理类即可。而用流程控制的话，细节处理上就会相当麻烦。而且像servlet等一些类似server的程序，都会在外包一层try-catch以保证程序在非致命Error下可以正常运行的吧？还望作者解答

淡云天

作者回复

符合catch late原则，业务代码更清楚怎么处理，这个看什么角度，不完全通常意义的流程控制；

“非致命”Error的出现，恰恰说明了异常机制在实践中并没有达到设计目的，所以才出现了争论，这么catch前提是清楚那些Error是应用能处理的

Gerald

2018-05-10

\*尤其是对于分布式系统，如果发生异常，但是无法找到堆栈轨迹（stacktrace），这纯属是为诊断设置障碍。所以，最好使用产品日志，详细地输出到日志系统里。“什么是产品日志呢？”

作者回复

应用里的logger

Gerald

2018-05-10

e.printStackTrace(): 在产品代码里使用有问题，这个不是能理解，IOException 是checked 异常，在catch块里应该怎么处理呢？

小马

2018-05-10

学习总结：1.理解Throwable, Error, Exception 的关系。

2.尽量不要捕获Exception通用异常，这样会使程序保留隐藏的异常，有很大的隐患。

3.不能生异常，就是将一大段代码放在try中，不知道会发生那些异常，就将其全部包起来，问题一保留隐藏异常对程序有隐患问题二：异常捕获处理对jvm的开销很大。更不能在异常处理实现业务逻辑。

学会了规范写异常处理部分的代码，以及这么规范的好处。

YI

2018-05-10

复习了一下Java异常处理知识，有很多启发性的内容，受益匪浅！最后的思考题我也想了下，是否可以构建一个独立的异常分析模块，业务端在提交异步任务时，将其重要的状态信息及traceid发送到异常分析模块，同时异步任务也与traceid捆绑，发生异常时，异步任务只需把捆绑的traceid及异常信息发送给异常分析模块，由异常分析模块来处理业务端与异常的配对。

coulson

2018-05-09

老师，你写的这些我大部分都能看懂，也能理解一部分，可是让我用文字形式写出来，我脑子就一踏糊涂，不知道从哪下手，该怎么办？

作者回复

心动不如行动，个人建议，试着问自己一个特定的问题，然后针对性的总结，别怕不够完美。  
编辑也是一样，不是什么时候都完全想清楚再做的

无意中找到的

2018-05-09

同事老是在catch中，直接用e.printStackTrace()打印堆栈，然后再用log打印一个e.getMessage() 还说在log里面没必要把堆栈打出来，太浪费空间时间了。。。感觉这样明显有问题，可我就是说不过他-\_-||

邓志国

2018-05-09

checked exception至少有一个用，通过编译时提醒你如果出错应该释放资源。否则如果是unchecked，说不定忘记了

Mine

2018-05-09

最近学了一点kotlin，在kotlin里面就是没有checkedException。这里想请教一下老师，在这种一切都是runtime的情况下应该以何种思路处理异常？因为不知道调用的函数会不会抛异常，不知道会抛哪种异常，而都用try catch包起来也不现实，最近一直在困惑这个点.....

作者回复

2018-05-09

我对Kotlin了解有限，初略的理解是：  
二者不完全对比，因为kotlin catch可以返回值，try-catch就变成了表达式，更优雅一些；  
对于函数会不会抛异常，IDE不会提供提示吗

渊

2018-05-09

如何知道我的try block会throw什么exception？

渊

2018-05-09

我怎么知道my try block应该catch什么exception？

作者回复

2018-05-09

try住的代码，抛出什么是明确声明了的，这种情况一般IDE更擅长

wang\_bo

2018-05-09

error开发者能不能去捕获和处理？

作者回复

2018-05-09

绝大部分情况是不，除非确定可以从Error恢复，或者没有别的选择

大胖

2018-05-09

error:jvm或者说是系统级别问题，捕获会导致问题无法定位，而且错误捕获应用程序也没什么作为，不如暴露出来定位错误！  
exception: RuntimeException以外的子类，属于应用程序可以处理的也在编程过程中可以判断的，应该抛出异常进行处理！这样理解对否？

沈老师

2018-05-09

能否选择您项目中一段设计比较完善的异常处理结合业务分析下，这样比较有具体场景的深入了解好的代码是怎么一个思路。谢谢！

板砖

2018-05-09

不要用一个大的try-catch包含整个代码这个怎么理解呢，因为springmvc来说，我现在的做法是在action里一个方法写一个业务，然后就写一个try-catch，如果不用一个大的该怎么写，还是说老师的说法就是一个业务一个try-catch？？

VincentWei

2018-05-08

链路追踪

愣子

反应式编程里出现异常是否可以搞一个类似dead queue的异常队列，把异常放到这个队列里供消费者去消费？

我奋斗去了

2018-05-08

Reactor 目前在Java中还没有使用过，感觉类似nodeJS的回调？期待正解

Rebby

2018-05-08

try 包住整段的代码，远比我们通常意义上的条件语句（if/else、switch）要低效。  
请问这段怎么理解呢？

作者回复

2018-05-08

有的代码被设计为，当发生某种异常时，走某条路径，这种条件判断方式是低效的

Hidden

2018-05-08

分析的太好了，学习完，基础就牢固很多了

作者回复

2018-05-08

很高兴能有帮助，程序开发不是只有那些高大上的东西的，以前有人找我帮助诊断应用上线闪退问题，我发现大部分就是几个NPE、IllegalStateException之类runtime exception低级错误导致的！

刹那间的永恒

2018-05-08

由于是半路出家，所以对于基础这块儿很不靠谱，尤其是之前对于异常处理很不重视，看了这篇文章收获了很多：  
1.catch具体的异常，不是exception；

2.慎用try catch，只包裹可能会抛出异常的代码；

3.慎用return break continue；

4.慎用e.printStackTrace()，都记录到日志（这个还不是很理解）；

5.能处理的异常尽早捕获处理，不能的就抛给上层。

作者回复

2018-05-08

换个角度，如果出了错，我们是希望看看log就能定位问题，还是觉得需要熬夜改改代码多print几行？  
前人（自己）挖坑，含泪也得爬出来

Heart to bodhi

2018-05-08

对于反应式编程 应该用日志记录好每个 executor信息 和输入信息

李志博

2018-05-08

Reactor我记得有onError和doError开头的一些方法

作者回复

2018-05-08

对头，非常不同的编程模式

不会游泳的鱼

2018-05-08

exception做设计的话有什么好的设计方案么？自动测试中捕获异常是必备的

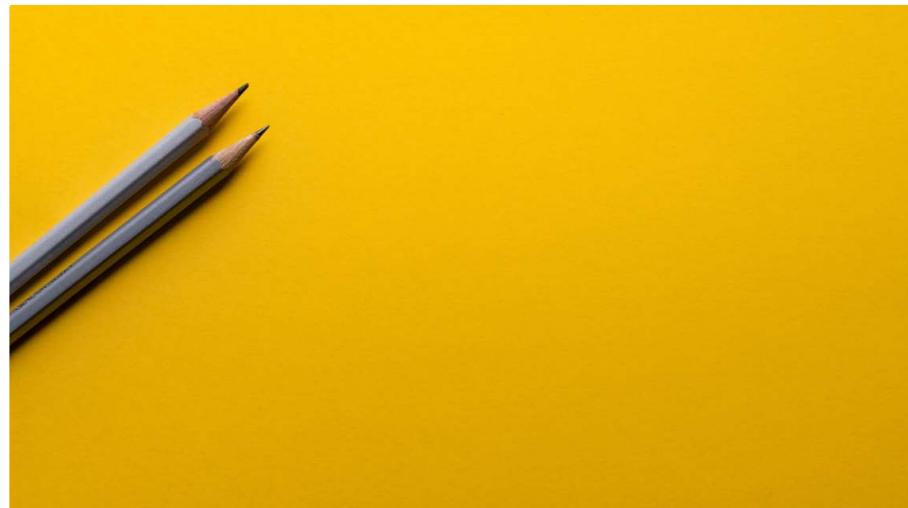
峰

2018-05-08

扩展任务类的实现，把当前的堆栈信息保留进去后提交，这样在任务中捕获异常后就可以做相关处理。

## 第3讲 | 谈谈final、finally、finalize有什么不同？

2018-05-10 杨晓峰



第3讲 | 谈谈final、finally、finalize有什么不同？  
杨晓峰  
00:00 / 11:03

Java语言有很多看起来很相似，但是用途却完全不同的语言要素，这些内容往往容易成为面试官考察你知识掌握程度的切入点。

今天，我要问你的是一个经典的Java基础题目，[谈谈final、finally、finalize有什么不同？](#)

## 典型回答

**final**可以用来修饰类、方法、变量，分别有不同的意义。**final**修饰的**class**代表不可以继承扩展，**final**的变量是不可以修改的，而**final**的方法也是不可以重写的（**override**）。

**finally**则是Java保证重点代码一定要被执行的一种机制。我们可以使用**try-finally**或者**try-catch-finally**来进行类似关闭JDBC连接、保证unlock锁等动作。

**finalize**是基础类**java.lang.Object**的一个方法，它的设计目的是保证对象在被垃圾收集前完成特定资源的回收。**finalize**机制现在已经不推荐使用，并且在JDK 9开始被标记为**deprecated**。

## 考点分析

这是一个非常经典的Java基础问题，我上面的回答主要是从语法和使用实践角度出发的，其实还有很多方面可以深入探讨，面试官还可以考察你对性能、并发、对象生命周期或垃圾收集基本过程等方面的理解。

推荐使用**final**关键字来明确表示我们代码的语义、逻辑意图，这已经被证明在很多场景下是非常好的实践，比如：

- 我们可以将方法或者类声明为**final**，这样就可以明确告知别人，这些行为是不许修改的。

如果你关注过Java核心类库的定义或源码，有没有发现**java.lang**包下面的很多类，相当一部分都被声明成为**final class**？在第三方类库的一些基础类中同样如此，这可以有效避免API使用者更改基础功能，某种程度上，这是保证平台安全的必要手段。

- 使用**final**修饰参数或者变量，也可以清楚地避免意外赋值导致的编程错误，甚至，有人明确推荐将所有方法参数、本地变量、成员变量声明成**final**。

- **final**变量产生了某种程度的不可变（**immutable**）的效果，所以，可以用于保护只读数据，尤其是在并发编程中，因为明确地不能再赋值**final**变量，有利于减少额外的同步开销，也可以省去一些防御性拷贝的必要。

**final**也许会有性能的好处，很多文章或者书籍中都介绍了可在特定场景提高性能，比如，利用**final**可能有助于JVM将方法进行内联，可以改善编译器进行条件编译的能力等等。坦白说，很多类似的结论都是基于假设得出的，比如现代高性能JVM（如HotSpot）判断内联未必依赖**final**的提示，要相信JVM还是非常智能的。类似的，**final**字段对性能的影响，大部分情况下，并没有考虑的必要。

从开发实践的角度，我不想过度强调这一点，这是和JVM的实现很相关的，未经验证比较难以把握。我的建议是，在日常开发中，除非有特别考虑，不然最好不要指望这种小技巧带来的所谓性能好处，程序最好是体现它的语义目的。如果你确实对这方面有兴趣，可以查阅相关资料，我就不再赘述了，不过千万别忘了验证一下。

对于**finally**，明确知道怎么使用就足够了。需要关闭的连接等资源，更推荐使用Java 7中添加的**try-with-resources**语句，因为通常Java平台能够更好地处理异常情况，编码量也要少很多，何乐而不为呢。

另外，我注意到有一些常被考到的**finally**问题（也比较偏门），至少需要了解一下。比如，下面代码会输出什么？

```
try {
    // do something
    System.exit(1);
} finally{
    System.out.println("Print from finally");
}
```

上面**finally**里面的代码可不会被执行的哦，这是一个特例。

对于**finalize**，我们要明确它是不推荐使用的，业界实践一再证明它不是个好的办法，在Java 9中，甚至明确将**Object.finalize()**标记为**deprecated!**！如果没有特别的原因，不要实现**finalize**方法，也不要指望利用它来进行资源回收。

为什么呢？简单说，你无法保证**finalize**什么时候执行，执行的是否符合预期。使用不当会影响性能，导致程序死锁、挂起等。

通常来说，利用上面的提到的**try-with-resources**或者**try-finally**机制，是非常好的回收资源的办法。如果确实需要额外处理，可以考虑Java提供的**Cleaner**机制或者其他替代方法。接下来，我来介绍更多设计考虑和实践细节。

#### 知识点扩展

##### 1. 注意，**final**不是**immutable**！

我在前面介绍了**final**在实践中的益处，需要注意的是，**final**并不等同于**immutable**，比如下面这段代码：

```
final List<String> strList = new ArrayList<>();
strList.add("Hello");
strList.add("world");
List<String> unmodifiableStrList = List.of("hello", "world");
unmodifiableStrList.add("again");
```

**final**只能约束**strList**这个引用不可以被赋值，但是**strList**对象行为不被**final**影响，添加元素等操作是完全正常的。如果我们真的希望对象本身是不可变的，那么需要相应的类支持不可变的行为。在上面这个例子中，[List.of方法](#)创建的本身就是不可变List，最后那句**add**是会在运行时抛出异常的。

**Immutable**在很多场景是非常棒的选择，某种意义上说，Java语言目前并没有原生的不可变支持，如果要实现**immutable**的类，我们需要做到：

- 将**class**自身声明为**final**，这样别人就不能扩展来绕过限制了。
- 将所有成员变量定义为**private**和**final**，并且不要实现**setter**方法。
- 通常构造对象时，成员变量使用深度拷贝来初始化，而不是直接赋值，这是一种防御措施，因为你无法确定输入对象不被其他人修改。
- 如果确实需要实现**getter**方法，或者其他可能会返回内部状态的方法，使用**copy-on-write**原则，创建私有的**copy**。

这些原则是不是在并发编程实践中经常被提到？的确如此。

关于**setter/getter**方法，很多人喜欢直接用IDE一次全部生成，建议最好是你确定有需要时再实现。

##### 2. **finalize**真的那么不堪？

前面简单介绍了**finalize**是一种已经被业界证明了的非常不好的实践，那么为什么会导致那些问题呢？

**finalize**的执行是和垃圾收集关联在一起的，一旦实现了非空的**finalize**方法，就会导致相应对象回收呈现数量级上的变慢，有人专门做过benchmark，大概是40~50倍的下降。

因为，**finalize**被设计成在对象被垃圾收集前调用，这就意味着实现了**finalize**方法的对象是个“特殊公民”，JVM要对它进行额外处理。**finalize**本质上成为了快速回收的阻碍者，可能导致你的对象经过多个垃圾收集周期才能被回收。

有人也许会问，我用**System.runFinalization()**告诉JVM积极一点，是不是就可以了？也许有点用，但是问题在于，这还是不可预测、不能保证的，所以本质上还是不能指望。实践中，因为**finalize**拖慢垃圾收集，导致大量对象堆积，也是一种典型的导致OOM的原因。

从另一个角度，我们要确保回收资源就是因为资源都是有限的，垃圾收集时间的不可预测，可能会极大加剧资源占用。这意味着对于消耗非常高频的资源，千万不要指望**finalize**去承担资源释放的主要职责，最多让**finalize**作为最后的“守门员”，况且它已经暴露了如此多的问题。这也是为什么我推荐，资源用完即显式释放，或者利用资源池来尽量重用。

**finalize**还会掩盖资源回收时的出错信息，我们看下面一段JDK的源代码，截取自**java.lang.ref.Finalizer**

```
private void runFinalizer(JavaLangAccess jla) {
    // ... 省略部分代码
    try {
        Object finalizee = this.get();
        if (finalizee != null && !finalizee instanceof java.lang.Enum) {
            jla.invokeFinalize(finalizee);
            // Clear stack slot containing this variable, to decrease
            // the chances of false retention with a conservative GC
            finalizee = null;
        }
    } catch (Throwable x) {
        super.clear();
    }
}
```

结合我上期专栏介绍的异常处理实践，你认为这段代码会导致什么问题？

是的，你没有看错，这里的**Throwable**是被生吞了！也就意味着一旦出现异常或者出错，你得不到任何有效信息。况且，Java在**finalize**阶段也没有好的方式处理任何信息，不然更加不可预测。

##### 3. 有什么机制可以替换**finalize**吗？

Java平台目前正在逐步使用**java.lang.ref.Cleaner**来替换原有的**finalize**实现。Cleaner的实现利用了幻象引用（**PhantomReference**），这是一种常见的所谓post-mortem清理机制。我会在后面的专栏系统介绍Java的各种引用，利用幻象引用和引用队列，我们可以保证对象被彻底销毁前做一些类似资源回收的工作，比如关闭文件描述符（操作系统有限的资源），它比**finalize**更加轻量、更加可靠。

吸取了**finalize**里的教训，每个Cleaner的操作都是独立的，它有自己的运行线程，所以可以避免意外死锁等问题。

实践中，我们可以为自己的模块构建一个Cleaner，然后实现相应的清理逻辑。下面是JDK自身提供的样例程序：

```

public class CleaningExample implements AutoCloseable {
    // A cleaner, preferably one shared within a library
    private static final Cleaner cleaner = <cleaner>;
    static class State implements Runnable {
        State(...) {
            // initialize State needed for cleaning action
        }
        public void run() {
            // cleanup action accessing State, executed at most once
        }
    }
    private final State;
    private final Cleaner.Cleanable cleanable;
    public CleaningExample() {
        this.state = new State(...);
        this.cleanable = cleaner.register(this, state);
    }
    public void close() {
        cleanable.clean();
    }
}

```

注意，从可预测性的角度来判断，Cleaner或者幻象引用改善的程度仍然是有限的，如果由于种种原因导致幻象引用堆积，同样会出现问题。所以，Cleaner适合作为一种最后的保证手段，而不是完全依赖Cleaner进行资源回收，不然我们就要再做一遍`finalize`的噩梦了。

我也注意到很多第三方库自己直接利用幻象引用定制资源收集，比如广泛使用的MySQL JDBC driver之一的mysql-connector-j，就利用了幻象引用机制。幻象引用也可以进行类似链式依赖关系的动作，比如，进行总量控制的场景，保证只有连接被关闭，相应资源被回收，连接池才能创建新的连接。

另外，这种代码如果稍有不慎添加了对资源的强引用关系，就会导致循环引用关系，前面提到的MySQL JDBC就在特定模式下有这种问题，导致内存泄漏。上面的示例代码中，将`State`定义为`static`，就是为了避免普通的内部类隐含着对外部对象的强引用，因为那样会使外部对象无法进入幻象可达的状态。

今天，我从语法角度分析了`final`、`finally`、`finalize`，并从安全、性能、垃圾收集等方面逐步深入，探讨了实践中的注意事项，希望对你有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？也许你已经注意到了，JDK自身使用的Cleaner机制仍然是有缺陷的，你有什么更好的建议吗？

请你在留言区写你的建议，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



zjh

2018-05-11

一直不懂为什么这三个经常拿来一起比较，本身就一点关系都没有啊，难道仅仅是长的像。我觉得final倒是可以和volatile一起比较下

石头狮子

2018-05-10

列几个 finally 不会被执行的情况:

1. try-catch 异常退出。  
try{  
 system.exit(1);  
}finally{  
 print(abc)  
}

2. 无限循环  
try{  
 while(true){  
 print(abc)  
 }  
}finally{  
 print(abc)

}

3. 线程被杀死  
当执行 `try, finally` 的线程被杀死时, `finally` 也无法执行。

总结  
1. 不要在 `finally` 中使用 `return` 语句。  
2. `finally` 总是执行, 除非程序或者线程被中断。

WolvesLeader

能不能帮我分析一下, 匿名内部类, 访问局部变量时, 局部变量为啥要用`final`来修饰吗?

作者回复

2018-05-10

这个因为Java inner class实际会copy一份, 不是去直接使用局部变量, `final`可以防止出现数据一致性问题

★神峰★

你们都看懂了吗? 我怎么什么都不知道◆◆

有鱼@蔡

1. 你说那异常被吞, 是指没写`e.printStackTrace()`吧? 另外我有个疑惑: `super.clear()` 为什么写在`exception`里, 理论上`super`方法写第一行, 或`finally`里。2. 在一个对象的生命周期里, 其`finalize`方法应该只会被调用1次。3. 强软弱引用大家都知道, 这虚幻引用相比较有什么特别的吗? 请再深入点。4. `final`是不是都在编译后确定位置? 比如`final List`这样的, 内存布局是怎样? 谢谢

皮卡皮卡丘

\*将 `State` 定义为 `static`, 就是为了避免普通的内部类隐含着对外部对象的强引用, 因为那样会使外部对象无法进入幻象可达的状态。“这个该怎么理解呢?

作者回复

2018-05-10

内部类如果不是 `static`, 它本身对外面那个类有引用关系, 这一点其实从构造阶段就能看出来, 你可以写段代码试试, 有强引用就是 `strong reachable` 状态

小哥。

`copy-on-write` 原则, 学习了

sharp

这三个就是卡巴斯基和巴基斯坦的关系, 有个基巴关系。。。

Hesher

见过一些写法是将对象手动赋值为`null`来让GC更快的回收, 不过能起多少作用就不知道了。关于JVM中那几种引用了解不多, 平时可以怎么使用呢?

ls

Java中有说: `finalize` 有一种用途: 在 Java 中调用非 Java 代码, 在非 Java 代码中若调用了C的 `malloc` 来分配内存, 如果不调用 C 的 `free` 函数, 会导致内存泄露。所以需要在 `finalize` 中调用它。

面试中会有问: 为什么 `String` 会设计成不可变? 想听听老师的解释

作者回复

2018-05-11

是的, 很多资源都是需要使用本地方式获取和释放

云学

请问这篇文章中涉及的知识点是Java中最重要的吗? 我感觉有点剑走偏锋, 这种知识了解就好了, 应该有很多知识比这更重要的吧, 虽说面试中可能会问, 但不能以面试为中心, 而要把实际应用中最有用的核心的东西分享出来, 把它讲透彻, 不追求面面俱到, 也不想成为语言专家。我期望通过这个专栏可以获得Java中最核心最实用特性的本质认识, 希望有一种醍醐灌顶的感觉。在阅读java开源框架代码时不再困惑。我有多年的c++开发背景, 希望通过这个专栏对java也有醍醐灌顶的本质认识。

refusecruder

```
杨老师, 关于final不能修改我想请教下, 代码如下, class util {
public final Integer info = 123;
}
```

```
@Test
public void test() throws NoSuchFieldException, IllegalAccessException {
    util util = new util();
    Field field = util.getClass().getDeclaredField("info");
    field.setAccessible(true);
    field.set(util, 789);
    System.out.println(field.get(util));
    System.out.println(util.info);
}
```

这里final修饰的被改了, 如果不加accessible这句会报错, 刚刚试了几个, 似乎是基本数据类型改不了, 封装类型都能改, 请杨老师解答下我的疑惑, 谢谢。

作者回复

2018-05-12

setAccessible是“流氓”, 不问题出在定义为基本数据类型, 会被当作constant, 可以反编译看看

◆◆Children之

用final修饰的class, 这可以有效避免 API 使用者更改基础功能, 某种程度上, 这是保证平台安全的必要手段。这个地方真的很需要个例子去帮助理解。比如大家都知道String类是被final修饰不可被继承, 但假如没有被final修饰, 很好奇会出现什么不安全的后果。

作者回复

2018-05-13

谢谢反馈

公号-Java大前端

1. 定义不可变对象类, 当构造函数传入可变对象引用时, 当getter函数返回可变对象引用时, 容易掉坑。

2. 在不可变对象类的构造函数中, 如果传入值包括了可变对象, 则clone先。

3. 从不可变对象类的getter函数返回前, 如果返回值为可变对象, 则clone先。

4. Java默认的clone方法执行浅拷贝, 对于数组、对象引用只是拷贝地址。浅拷贝在业务实现中可能是一个坑, 需要多加注意。

2018-05-14

2018-05-10

2018-05-14

5 如果步骤2、3中的浅拷贝无法满足不可变对象要求，请实现“深拷贝”。

echo\_陈

2018-05-10

回答上面一个人的问题。  
被final修饰的变量不可变。如果初始化不赋值，后续赋值，就是从null变成你的赋值，违反不可变

Do

2018-05-12

final修饰变量参数的时候，其实理解为内存地址的绑定，这样理解是不是更直观。基本类型指向栈中，引用类型指向堆中。老师后期文章能不能说下java堆栈的区别，还有变量局部变量的生命周期，最好能附上图，加深理解。

作者回复

会有

小绵羊拉拉

2018-05-12

首先，这篇文章比上一讲明显感觉到由浅入深 很不错 有个问题请教一下finalize方法是用来回收对外内存是不是可以这么理解，类似于本地方法申请的能源 new出来的对象实现了这个方法当垃圾回收的时候会将对象放在fqueue等待被执行 不过是异步不知道啥时候被执行 可能被执行的时候对象已经置空导致不安全 可以这么理解吗 新的jdk引入的clear方法 能完全取代虚拟机中finalize方法吗

说重点、

2018-05-10

常被问，string类为什么用final修饰

Colingo

2018-05-10

finally 里面的打印为什么不会被调用？

loveluckystar

2018-05-10

个人理解，finalize本身就是为了提供类似c或c++析构函数产生的，由于java中gc本身就是自动进行的，是不希望被干扰的，(就像System.gc(), 并不一定起作用)所以与其费心研究如何使用这个，不如老老实实把该做的事情做了来的实惠。

作者回复

对，有些特别情况需要额外处理，毕竟无法保证编程都按规范来

feifei

2018-05-10

JDK 自身使用的 Cleaner 机制仍然是有缺陷的，你有什么更好的建议吗？

1. 临时对象，使用完毕后，赋值为null，可以加快对象的回收
2. 公用资源对象，比如数据库连接，使用连接池
3. native调用资源的释放，比如一个进程初始化调用一次，退出调用一次，这类场景可以考虑使用cleaner
4. 对尽量try-finally中完成资源的释放，即使用完毕就释放，最小化的使用，下次使用在申请。
5. 可以使用钩子进行程序的正常退出清理操作。

此为我个人的一点小心得，欢迎老师指正，谢谢

jeff

2018-06-30

回答张勇的问题  
注意你的final值得作用域 生命周期

不吃老鼠的猫

2018-05-17

我有个这样一个场景，在service方法里，处理自己的逻辑，因为是调用外部接口，所以可能会报错，那么我用try catch捕获异常，返回上层结果，但是问题来了，我有个log表，要保存每次请求、响应数据，我捕获到外部异常后，处理了下异常，又抛出去，现在我的处理逻辑是在finally里，来保存所有请求和响应的数据，但觉得有点鸡肋，finally理论上不应该涉及到逻辑代码，求老师指正？

凡

2018-05-10

请问为什么被final修饰的变量需要显示赋值  
梅丹隆

2018-05-10

形容词，副词和名词  
团结兀王二狗他二大爺

2018-07-11

前半部分可以看懂，后面涉及幻象引用就不懂了。完全看懂感觉需要看一下那篇引用的文章。文章整体不错，从多个角度分析一个问题。我想这也是一个工程师应该具备的思维之一，希望自己能够不断提高，与君共勉。

待时而发

2018-07-10

有点懵

2018-06-19

jacy  
加final的state为啥还能赋值呢？

2018-05-28

大熊

2018-05-27

对我这种刚入门的来说 是有点高深了  
徐金铎

2018-05-26

如果是考虑gc回收，推荐大家更关注weakreference, softreference等结合referencequeue来考虑。这样对gc更友好些。  
徐金铎

2018-05-26

Final关键字，还有禁止代码前编译器重排序的作用，可以用做构造溢出的解决方案。

密码123456

2018-05-25

1. **final**修饰的类，不可被继承，修饰的方法不可被重写，修饰的变量不可多次赋值。通过**final**能够得到性能上的优化，但是不明显，如果大量使用可能会干扰代码，不能表达出本来具有的含义。故不使用。  
 2. **finally**。代码中总是会执行的代码段。除了退出虚拟机外。  
 3. **finalize**。在虚拟机回收改对彖前进行调用。此种方式不可取。因为java虚拟机不知道在什么时候才对对象进行回收。

曲高和寡

关于匿名内部类访问局部变量必须**final**的原因，还有一个关键是匿名内部类的生命周期可能比外部类要长，而如果外部类已经被垃圾回收了，那内部类访问的就是一个空变量。**final**可以防止被回收，老师对么？

作者回复

我理解不是

haoz

List.of()方法我的jdk1.8中没有 网上也没有相关资料。是不是老师写错了呢?还望老师多多指教!

作者回复

Java 9引入的，可能忘了介绍；  
 8已经4、5年了，马上9月份发布jdk 11，下一个长期支持版本，建议开始实验新的了

微笑的向日葵

一般来说在框架中做资源创建和回收，都是通过aop

z

有List.of () 方法吗我怎么找不到  
 作者回复

忘了说Java 9新增的，回头修改下

程序员的小浣熊

关于copy\_on\_write实现getter方法可以有例子吗？因为我理解的该原则是懒修改策略，但是不变类不应该不做任何修改么？希望可以解答一下。

作者回复

用词也许有点歧义，就是只暴露copy

微的空

说到**final**的话，现在很多的会谈到**static**。许多代码都会使用到**static final**来同时修饰一个变量（这里称为常量更合适）。从而可以达到一个编译期常量的作用。这使得我们不需要初始化一个类就能够直接访问其成员，对节约资源效率上有不少的提升。  
 所以我觉得老师可以顺带提一下**static**，或者放些补充学习的资料哈。

而以上东西其实可以进一步深挖，这就会关系到老师在第一讲提到过的类加载、验证、链接、初始化。这个过程，介于篇幅原因未能进一步展开，有兴趣的同学可以翻看TJ和深入理解JVM进行学习。

以上是个人理解，若有错误，还望指正。

作者回复

个人认为**static**之类主要还是注重反应语义的需求，过早考虑节省资源太片面

张勇

```
匿名内部类为什么访问外部类局部变量必须是final的? private Animator createAnimatorView(final View view, final int position) {  

MyAnimator animator = new MyAnimator();  

animator.addListener(new AnimatorListener() {  

@Override  

public void onAnimationEnd(Animator arg0) {  

Log.d(TAG, "position=" + position);  

}  

});  

return animator;  

}
```

作者回复

文中介绍了，它其实现是会copy一份，**final**可以避免一致性问题

李润林

**finalize**应该是从c++的析构函数那里继承来的一个东西，随着垃圾收集算法的不断改进，变得完全不可控了。

作者回复

本身使用场景和需求是存在的，只是存在一些弊端，实在不行也得用，毕竟准确性、可靠性是基础

张勇

```
老师咨询下，我下面这段代码String类型的参数用final修饰为啥我每次可以穿不同的参数进去，并且也可以运行成功final不是不可变么， public class MainActivity extends  

AppCompatActivity {  

private static final String TAG="MainActivity";  

@Override  

protected void onCreate(Bundle savedInstanceState) {  

super.onCreate(savedInstanceState);  

setContentView(R.layout.activity_main);  

test("AAAAAAA");  

test("BBBBBBB");  

test("CCCCCCC");  

test("DDDDDDD");  

}  

}  

public void test(final String str){  

Log.d(TAG,str);  

}  

}
```

作者回复

看不出来super内部的逻辑，再说final这东西有太多可以绕过去的方式。比如setaccessible或者配合修改modifier

冯召坤

2018-05-11

分析的不错

老胡

2018-05-11

不可变。最基本是行为不可变，不提供可变的操作。变量私有化，没有增加，修改等方法，final只是保证指针不可变，无法保证内容不可变。

老胡

2018-05-11

Cleaner机制会对jvm回收造成负担，因为gc回收的时候需要检测这个对象十分是Cleaner，然后处理。如果处理过长，十分影响gc的效率。好点方案，容器管理对象，比如spring的scope，或者对象单利等等，gc负担是一个致命问题，所以Cleaner谨慎使用，甚至应该禁止

作者回复

2018-05-11

未必有这么可怕，比finalize有数量级的提高，spring这个不错，但不能解决所有场景

FF

2018-05-11

IDEA 的版本是2018.1.2，throw一个方法没有声明的受检查 ex 也是合规的写法吗？即使 try 里面没有显示 throw

作者回复

2018-05-11

这是rethrow，我理解不存在模棱两可，或者哪位高手有精力去翻spec，谢谢

独裁记忆

2018-05-10

老师，能否之后多开几讲，两三千字的限制怕知识点后面不够或者没法深入。

作者回复

2018-05-10

我也在考虑补充一些，很多地方还没展开就3000多了.....

增煌

2018-05-10

final修饰变量表示不能被修改应该是不能被赋值更合适一点吧？对基本类型来说就是不能修改，对string和object只是引用不能修改，引用的对象还是可变的。

学无止境

2018-05-10

老师，您好，我想请教一下finalize或者cleaner使用场景是什么？谢谢。

life is cool

2018-05-10

老师，final对象变量Jvm是怎么被垃圾回收器回收的呢？

一笑奈何

2018-05-10

先说答案在分析挺好◆◆

曹铮

2018-05-10

之前甚至不知道有cleaner这回事..希望杨老师后面有机会详细说说

作者回复

2018-05-10

好的，2、3千字篇幅限制大，后面补充吧，你可以看看jdk源码用的内部cleaner

FF

2018-05-10

请教杨老师一个异常处理的问题，我们使用eclipse的同学通常这样处理异常：

try{

```
    }catch(Exception ex){  
    throw ex; //这里抛出异常,  
}
```

但方法并没有声明throws Exception，而eclipse通常也能编译执行，这不是违反了基本语法了吗，为何eclipse里面没有任何问题的？

这是什么原因？但这样的代码在使用IntelliJ IDEA的同学那里是完全无法编译执行的，直接就提示语法错误了

很困惑，网上没找到相关答案，望解答，感谢！

作者回复

2018-05-10

IDE有自己的编译实现，如果try里没有显式throw，只是catch那么写，应该是合规的，猜测是idea的bug，你用什么版本？

yao

2018-05-10

@mongo 另外可以起到语义上的作用

mongo

2018-05-10

final修饰引用类型的话，引用值不能被修改。但是引用值指向的内容可以被修改，这样看来修饰引用类型并不是线程安全的。什么场景下会使用final修饰引用类型呢？

作者回复

2018-05-10

引用不希望被修改的时候，仍然有一定保护作用

## 第4讲 | 强引用、软引用、弱引用、幻象引用有什么区别?

2018-05-12 杨晓峰



第4讲 | 强引用、软引用、弱引用、幻象引用有什么区别?  
杨晓峰  
- 00:00 / 10:23

在Java语言中，除了原始数据类型的变量，其他所有都是所谓的引用类型，指向各种不同的对象，理解引用对于掌握Java对象生命周期和JVM内部相关机制非常有帮助。

今天我要问你的是，[强引用、软引用、弱引用、幻象引用有什么区别？具体使用场景是什么？](#)

#### 典型回答

不同的引用类型，主要体现的是对象不同的可达性（reachable）状态和对垃圾收集的影响。

所谓强引用（"Strong" Reference），就是我们最常见的普通对象引用，只要还有强引用指向一个对象，就能表明对象还“活着”，垃圾收集器不会碰这种对象。对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将相应（强）引用赋值为null，就是可以被垃圾收集的了，当然具体回收时机还是要看垃圾收集策略。

软引用（SoftReference），是一种相对强引用弱化一些的引用，可以让对象豁免一些垃圾收集，只有当JVM认为内存不足时，才会去试图回收软引用指向的对象。JVM会确保在抛出`OutOfMemoryError`之前，清理软引用指向的对象。软引用通常用来实现内存敏感的缓存，如果还有空闲内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

弱引用（WeakReference）并不能使对象豁免垃圾收集，仅仅是提供一种访问在弱引用状态下对象的途径。这就可以用来构建一种没有特定约束的关系，比如，维护一种非强制性的映射关系，如果试图获取时对象还在，就使用它，否则重现实例化。它同样是很多缓存实现的选择。

对于幻象引用，有时候也翻译成虚引用，你不能通过它访问对象。幻象引用仅仅是提供了一种确保对象被`finalize`以后，做某些事情的机制，比如，通常用来做所谓的Post-Mortem清理机制，我在专栏上一讲中介绍的Java平台自身Cleaner机制等。也有人利用幻象引用监控对象的创建和销毁。

#### 考点分析

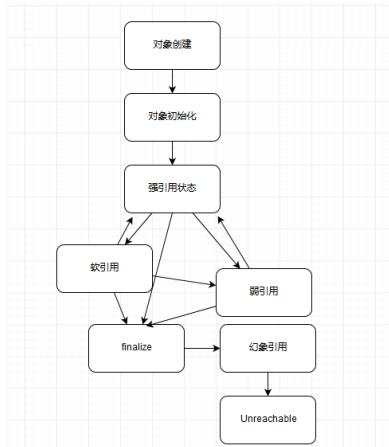
这道面试题，属于既偏门又非常高频的一道题目。说它偏门，是因为在大多数应用开发中，很少直接操作各种不同引用，虽然我们使用的类库、框架可能利用了其机制。它被频繁问到，是因为这是一个综合性的题目，既考察了我们对基础概念的理解，也考察了对底层对象生命周期、垃圾收集机制等的掌握。

充分理解这些引用，对于我们设计可靠的缓存等框架，或者诊断应用OOM等问题，会很有帮助。比如，诊断MySQL connector-J驱动在特定模式下（`useCompression=true`）的内存泄漏问题，就需要我们理解怎么排查幻象引用的堆积问题。

#### 知识扩展

##### 1. 对象可达性状态流转分析

首先，请你看下面流程图，我这里简单总结了对象生命周期和不同可达性状态，以及不同状态可能的改变关系，可能未必100%严谨，来阐述下可达性的变化。



我来解释一下上图的具体状态，这是Java定义的不同可达性级别（reachability level），具体如下：

- 强可达（Strongly Reachable），就是当一个对象可以有一个或多个线程可以不通过各种引用访问到的情况。比如，我们新创建一个对象，那么创建它的线程对它就是强可达。
- 软可达（Softly Reachable），就是当我们只能通过软引用才能访问到对象的状态。
- 弱可达（Weakly Reachable），类似前面提到的，就是无法通过强引用或者软引用访问，只能通过弱引用访问时的状态。这是十分临近finalize状态的时机，当弱引用被清除的时候，就符合finalize的条件了。
- 幻象可达（Phantom Reachable），上面流程图已经很直观了，就是没有强、软、弱引用关联，并且finalize过了，只有幻象引用指向这个对象的时候。
- 当然，还有一个最后的状态，就是不可达（unreachable），意味着对象可以被清除了。

判断对象可达性，是JVM垃圾收集器决定如何处理对象的一部分考虑。

所有引用类型，都是抽象类java.lang.ref.Reference的子类，你可能注意到它提供了get()方法：

**T get()** Returns this reference object's referent.

除了幻象引用（因为get永远返回null），如果对象还没有被销毁，都可以通过get方法获取原有对象。这意味着，利用软引用和弱引用，我们可以将访问到的对象，重新指向强引用，也就是人为的改变了对象的可达性状态！这也是为什么我在上面图里有些地方画了双向箭头。

所以，对于软引用、弱引用之类，垃圾收集器可能会存在二次确认的问题，以保证处于弱引用状态的对象，没有改变为强引用。

但是，你觉得这里有没有可能出现什么问题呢？

不错，如果我们错误的保持了强引用（比如，赋值给了static变量），那么对象可能就没有机会变回类似弱引用的可达性状态了，就会产生内存泄漏。所以，检查弱引用指向对象是否被垃圾收集，也是诊断是否有特定内存泄漏的一个思路，如果我们的框架使用到弱引用又怀疑有内存泄漏，就可以从这个角度检查。

## 2.引用队列（ReferenceQueue）使用

谈到各种引用的编程，就必然要提到引用队列。我们在创建各种引用并关联到响应对象时，可以选择是否需要关联引用队列，JVM会在特定时机将引用enqueue到队列里，我们可以从队列里获取引用（remove方法在这里实际是有获取的意思）进行相关后续逻辑。尤其是幻象引用，get方法只返回null，如果再不指定引用队列，基本就没有意义了。看看下面的示例代码。利用引用队列，我们可以在对象处于相应状态时（对于幻象引用，就是前面说的被finalize了，处于幻象可达状态），执行后期处理逻辑。

```

Object counter = new Object();
ReferenceQueue<Object> refQueue = new ReferenceQueue<Object>();
PhantomReference<Object> p = new PhantomReference<Object>(counter, refQueue);
counter = null;
System.gc();
try {
    // Remove是一个阻塞方法，可以指定timeout，或者选择一直阻塞
    Reference<Object> ref = refQueue.remove(1000L);
    if (ref != null) {
        // do something
    }
} catch (InterruptedException e) {
    // Handle it
}
}
  
```

## 3.显式地影响软引用垃圾收集

前面泛泛提到了引用对垃圾收集的影响，尤其是软引用，到底JVM内部是怎么处理它的，其实并不是非常明确。那么我们能不能使用什么方法来影响软引用的垃圾收集呢？

答案是有的。软引用通常会在最后一次引用后，还能保持一段时间，默认值是根据堆剩余空间计算的（以M bytes为单位）。从Java 1.3.1开始，提供了-XX:SoftRefLRUPolicyMSPerMB参数，我们可以以毫秒（milliseconds）为单位设置。比如，下面这个示例就是设置为3秒（3000毫秒）。

```
-XX:SoftRefLRUPolicyMSPerMB=3000
```

这个剩余空间，其实会受不同JVM模式影响，对于Client模式，比如通常的Windows 32 bit JDK，剩余空间是计算当前堆里空闲的大小，所以更加倾向于回收；而对于server模式JVM，则是根据-Xmx指定的最大值来计算。

本质上，这个行为还是个黑盒，取决于JVM实现，即使是上面提到的参数，在新版的JDK上也未必有效，另外Client模式的JDK已经逐步退出历史舞台。所以在我们应用时，可以参考类似设置，但不要过于依赖它。

#### 4.诊断JVM引用情况

如果你怀疑应用存在引用（或`finalize`）导致的回收问题，可以有很多工具或者选项可供选择，比如HotSpot JVM自身便提供了明确的选项（`PrintReferenceGC`）去获取相关信息。我指定了下面选项去使用JDK 8运行一个样例应用：

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintReferenceGC
```

这是JDK 8使用ParallelGC收集的垃圾收集日志，各种引用数量非常清晰。

```
0.403: [GC (Allocation Failure) 0.871: [SoftReference, 0 refs, 0.0000393 secs]@0.871: [WeakReference, 8 refs, 0.0000138 secs]@0.871: [FinalReference, 4 refs, 0.0000094 secs]@0.871: [PhantomReference, 0 refs, 0 refs, 0.0000085 secs]@0.871: [JNI Weak Reference, 0.0000071 secs][PSYoungGen: 76272K->10728K(141824K)] 128286K->128422K(316928K), 0.4683919 secs] [Times: user=1.17 sys=0.03, real=0.47 secs]
```

注意：JDK 9对JVM和垃圾收集日志进行了广泛的重构，类似`PrintGCTimeStamps`和`PrintReferenceGC`已经不再存在，我在专栏后面的垃圾收集主题里会更加系统的阐述。

#### 5.Reachability Fence

除了我前面介绍的几种基本引用类型，我们也可以通过底层API来达到强引用的效果，这就是所谓的设置reachability fence。

为什么需要这种机制呢？考虑一下这样的场景，按照Java语言规范，如果一个对象没有指向强引用，就符合垃圾收集的标准，有些时候，对象本身并没有强引用，但是也许它的部分属性还在被使用，这样就导致诡异的问题，所以我们需要一个方法，在没有强引用情况下，通知JVM对象是在被使用的。说起来有点绕，我们来看看Java 9中提供的案例。

```
class Resource {
    private static ExternalResource[] externalResourceArray = ...
    int myIndex; Resource(...) {
        myIndex = ...
        externalResourceArray[myIndex] = ...
    ...
}
protected void finalize() {
    externalResourceArray[myIndex] = null;
    ...
}
public void action() {
    try {
        // 需要被保护的代码
        int i = myIndex;
        Resource.update(externalResourceArray[i]);
    } finally {
        // 调用reachabilityFence，明确保留下strongly reachable
        Reference.reachabilityFence(this);
    }
}
private static void update(ExternalResource ext) {
    ext.setStatus = ...
}
```

方法`action()`的执行，依赖于对象的部分属性，所以被特定保护了起来。否则，如果我们在代码中像下面这样调用，那么就可能会出现困扰，因为没有强引用指向我们创建出来的`Resource`对象，JVM对它进行`finalize`操作是完全合法的。

```
new Resource().action()
```

类似的书写结构，在异步编程中似乎是很普遍的，因为异步编程中往往不会用传统的“执行->返回->使用”的结构。

在Java 9之前，实现类似功能相对比较繁琐，有的时候需要采取一些比较隐晦的小技巧。幸好，`java.lang.ref.Reference`给我们提供了新方法，它是JEP 193: Variable Handles的一部分，将Java平台底层的一些能力暴露出来：

```
static void reachabilityFence(Object ref)
```

在JDK源码中，`reachabilityFence`大多使用在`Executors`或者类似新的HTTP/2客户端代码中，大部分都是异步调用的情况。编程中，可以按照上面这个例子，将需要`reachability`保障的代码段用`try-finally`包围起来，在`finally`里明确声明对象强可达。

今天，我总结了Java语言提供的几种引用类型、相应可达状态以及对于JVM工作的意义，并分析了引用队列使用的一些实际情况，最后介绍了在新的编程模式下，如何利用API去保障对象不被以为意外回收，希望对你有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？给你留一道练习题，你能从自己的产品或者第三方类库中找到使用各种引用的案例吗？它们都试图解决什么问题？

请你在留言区写写你的答案，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享出去，或许你能帮到他。



Miaozhe

2018-05-18

接着上个问题：

老师，问个问题：我自己定义一个类，重写finalizer方法后，创建一个对象，被幻想引用，同时该幻想对象使用ReferenceQueue，当我这个对象指向null，被GC回收后，ReferenceQueue中没有改对象，不知道是什么原因？如果我把类中的finalizer方法移除，ReferenceQueue就能获取被释放的对象。

2018-05-17作者回复文章图里阐明了，幻象引用enqueue发生在finalize之后，你查看是不是卡在FinalReference queue里了，那是实现finalization的地方

杨老师，我去查看了，Final reference和Reference发现是Reference Handle线程在监控，但是Debug进不去，还是没有搞清楚原理。

不过，我又发现类中自定义Finalize，如果是空的，正常。如果类中有任何代码，都不能进入Reference Queue，怀疑是对象没有被GC回收。

作者回复

2018-05-18

空的Finalize实现，不会起作用的；

Finalizer是懒家伙，试试system.runfinalization；

公号-Java大前端

2018-05-12

在Java语言中，除了基本数据类型外，其他的都是指向各类对象的对象引用；Java中根据其生命周期的长短，将引用分为4类。

### 1 强引用

特点：我们平常典型编码Object obj = new Object()中的obj就是强引用。通过关键字new创建的对象所关联的引用就是强引用。当JVM内存空间不足，JVM宁愿抛出OutOfMemoryError运行时错误（OOM），使程序异常终止，也不会靠随意回收具有强引用的“存活”对象来解决内存不足的问题。对于一个普通的对象，如果没有其他的引用关系，只要超过了引用的作用域或者显式地将值设为（强）引用赋值为 null，就是可以被垃圾收集的了，具体回收时机还是要看垃圾收集策略。

### 2 软引用

特点：软引用通过SoftReference类实现。软引用的生命周期比强引用短一些。只有当JVM认为内存不足时，才会去试图回收软引用指向的对象；即JVM会确保在抛出OutOfMemoryError之前，清理软引用指向的对象。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。后续，我们可以调用ReferenceQueue的poll()方法来检查是否有它所关心的对象被回收。如果队列为空，将返回一个null，否则该方法返回队列中前面的一个Reference对象。

应用场景：软引用通常用来实现内存敏感的缓存。如果还有空间内存，就可以暂时保留缓存，当内存不足时清理掉，这样就保证了使用缓存的同时，不会耗尽内存。

### 3 弱引用

弱引用通过WeakReference类实现。弱引用的生命周期比软引用短，在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快回收弱引用的对象。弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

应用场景：弱应用同样可用于内存敏感的缓存。

### 4 虚引用

特点：虚引用也叫幻象引用，通过PhantomReference类来实现。无法通过虚引用访问对象的任何属性或函数。幻象引用仅仅是提供了一种确保对象被 finalize 以后，做某些事情的机制。如果一个对象仅持有虚引用，那么它就没有任何引用一样，在任何时候都可能被垃圾回收器回收。虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

ReferenceQueue queue = new ReferenceQueue();

PhantomReference pr = new PhantomReference(object, queue);

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取一些程序行动。

应用场景：可用来跟踪对象被垃圾回收器回收的活动，当一个虚引用关联的对象被垃圾收集器回收之前会收到一条系统通知。

作者回复

2018-05-14

高手

海怪哥哥

2018-05-13

我的理解，Java的这种抽象很有意思。

强引用就像大老婆，关系很稳固。

软引用就像二老婆，随时有失宠的可能，但也有扶正的可能。

弱引用就像情人，关系不稳定，可能跟别人跑了。

幻像引用就是梦中情人，只在梦里出现过。

作者回复

2018-05-14

牛

Jerry恨恨

1. 强引用：项目中到处都是。

2. 软引用：图片缓存框架中，“内存缓存”中的图片是以这种引用来保存，使得JVM在发生OOM之前，可以回收这部分缓存

3. 虚引用：在静态内部类中，经常会使用虚引用。例如，一个类发送网络请求，承担callback的静态内部类，则常以虚引用的方式来保存外部类(宿主类)的引用，当外部类需要被JVM回收时，不会因为网络请求没有及时回来，导致外部类不能被回收，引起内存泄漏

4. 幽灵引用：这种引用的get()方法返回总是null，所以，可以想象，在平常的项目开发肯定用的少。但是根据这种引用的特点，我想可以通过监控这类引用，来进行一些垃圾清理的动作。不过具体的场景，还是希望峰哥举几个稍微详细的实战性的例子？

作者回复

非常不错，高手；

你可以参考Jdk内部cleaner使用，一个方面就贴太多，有凑字数嫌疑了

肖一林

这篇文章只描述了强引用，软引用，弱引用，幻想引用的特征。没有讲他们的概念，更没有讲怎么用...gc roots也没提到。希望能补充完来龙去脉，让没有太多基础的人也能看懂

探索无止境

希望可以配合一些实际的例子来讲解各种引用会更好，不会仅停留在理论理解层面，实际例子更有助于理解！

作者回复

谢谢反馈，我会平衡一下，主要是贴太多代码很容易字数就满了，也不利于录音频

Jane

引用出现的根源是由于GC内存回收的基本原理—GC回收内存本质上是回首对象，而目前比较流行的回收算法是可达性分析算法，从GC Roots开始按照一定的逻辑判断一个对象是否可达，不可达的话就说明这个对象已死（除此之外另外一种常见的算法就是引用计数法）。但是这种算法有个问题是不能解决相互引用的问题。于此Java向用户提供了四种可用的引用：即我们本章讲解的这几种，同时又提供了一种不可使用的引用—FinalReference（这个引用是和结构函数密切相关的）。强引用：开发者可以通过new的方式创建，其它的几种引用Java提供了相应的类：SoftReference、WeakReference、PhantomReference，如果你查看源码你会发现，这三个类实现的核心是Reference与ReferenceQueue（更通俗地说引用队列）两个类，而且这两个类都特别的简单。Reference类是一个链表结构，通过创建一个守护线程来执行对应引用的清除。Cleaner.clean（如果传入的对象是该类的话），以及引用的入队操作（需要在垃圾引用的时候制定一个引用队列），ReferenceQueue这是制定了引用队列的一些具体操作，简单的来说它也是一个链表结构，并提供了一些基本的链表操作。而除了强引用外其它的都是继承于此，通过这样的类约束了引用的相关内容，便于与GC进行交互。这种几种引用的区别如下：

1: 强引用是只有当GC明确判断该引用无效的时候才会回收相关的引用对象，即使抛出OOM警告。

2: 软引用：当GC检测到继续对对象会导致OOM的时候会进行一次垃圾回收，这次回收会讲软引用回收以防抛出异常，根据这样的特点该引用常用来被当作缓存使用。

3: 虚引用：是那些如果引用未被使用，就会在最近的一次GC的时候被回收。例如Java的ThreadLocal与动态代理都是基于这样的一个引用实现的，一般针对那些比较敏感的数据。

4: 幻想引用是针对那些已经执行完析构函数之后，仍然需要在执行一些其它操作的对象：比如资源对象的关闭就可以用到这个引用。

feitian

我觉得录音和文字可以不一样，不要兼顾这两者，录音内容应该远多于文字，就像PPT一样，讲述的人表述的会远多于文字体现出来的东西。所以不用为了录音方便考虑文字内容多少，文字尽量能不靠录音也是完整的，录音的内容会更丰富，但有些不好描述的部分，比如代码要配合文字一起看。

作者回复

好建议，回头和极客反馈下

石头狮子

对各种引用的理解，可以理解为对对象 jvm 堆内存的占用时长。对于对象可达性垃圾回收算法，可达性可以认为回收内存的标志。

1. 强引用：只要对象引用可达，对象使用的内存就一直被占用。

2. 软引用：对对象使用的内存一直占用，直到 jvm 认为有必要回收内存。

3. 弱引用：对对象使用的内存一直占用，直到下一次 gc。

求赞。◆◆◆◆◆

kugool

感觉自己基础还很差 除了强引用 其它几个都不是很明显

爱吃面的蝎子王

希望作者照顾层次化的读者，讲名词概念要有具体解释，并能举例一二帮助理解，不然看完依旧似懂非懂一知半解。

作者回复

谢谢反馈，回头把必要概念加个链接或解释

龙猫猫猫

热评第一讲得比这文章还好

坞摩智

这个确实是日常开发所接触不到的知识点，所以看起来挺费力的

作者回复

未必是直接写，类似基础架构处理mysql oom就会用到了

哈

看完以后，真的很不明白。有一种告诉你了一下概念却没有用具体实际进行比较说明，概念性的东西太抽象看完一点印象都没有...希望作者能用改进一下

程序猿的小浣熊

我觉得可以在github上托管示例代码，说到关键代码的时候，直接链接过去就好。这样就能丰富内容了。

kursk.ye

于是我google到了这篇文章，<http://www.kdgregory.com/index.php?page=java.refobj>，花了几分钟（真的是几天，不是几小时）才基本读完，基本理解这几个reference的概念和作用，从这个角度来讲非常感谢作者。如果不是本文的介绍，我还以为GC还是按照reference counter的原理处理，原来思路早变了。话说回来，《Java Reference Objects》真值得大家好好琢磨，相信可以回答很多人的问题。比如strong reference，soft reference，weak reference怎么互转，如果一个obj 已经 = null,就 obj = reference.get()呗。再有，文章中用weak reference 实现 canonicalizing map改善内存存储效率，减小存储空间的例子，真是非常经典啊。也希望作者以后照顾一下低层次读者，写好技术铺垫和名词定义。顺便问一下大家是怎么

言的，在手机上打那么多字，还有排版是怎么处理的，我是先在电脑上打好字再COPY上来的，大家和我一样吗？

作者回复

非常感谢反馈！

关于引用计数，也有优势，我记得某个国外一线互联网公司调优python，就是只用引用计数，关闭gc

ASCE1885

Android开发面试中这道题大家基本都会，Java后端面试中遇到好些人说没听过的，说到底有部分人还是Java基础不过关，只会用框架。

gayguyogogo

讲些能夯实基础同时在开发中需要用到的

作者回复

好的，谢谢反馈

jimforcode

看得还是比较蒙，希望配合一些例子举例说明

作者回复

谢谢，我尽量兼顾，有不足后面会补充

coder王

Android 中的Glide 图片加载框架的内存缓存就使用到了弱引用缓存机制◆◆

作者回复

图片相当比较大，所以图片缓存是典型场景

wang\_bo

除了强引用，其它的都不懂

有渔@蔡

好文章，就需要这样深的。有个留言问ThreadLocal中，entry的key为软引用，value为实际object.当key被回收后，object会产生内存泄露问题。同请具体解答。谢谢

作者回复

已回复，如果有其他情况可以介绍一下细节，马上飞机去美国，有段时间不能回复，抱歉

呵呵

弱引用，幻影引用还是理解不清楚

作者回复

可以简化下几个方面：对gc是否影响；什么时候enqueue；get返回什么

kyq叶鑫

看到第四讲了，每一讲都看到留言有朋友说看完之后还是懵懵的希望作者多提供实际例子，因为大家都是理工思维，不喜虚的，喜欢直接上干货，建议作者可以从读者背景方面改善文章内容，软硬兼施。

小情绪

多谢杨老师，建议多结合代码讲解会更清晰，毕竟代码是最好的诠释。

作者回复

好的，后面文章平衡下，贴多了代码也有人说凑字数...

孙晓明

看完这篇文章之后，对这四种引用还是一知半解。不知道它们与JVM中heap的新生代，老生代，永代是什么关系？

作者回复

怨我孤陋寡闻了，没有什么直接关系

jutsu

感觉自己理解的慢，每天都会吸收一点进步一点点，谢谢老师

作者回复

加油◆◆

男人，7分熟。

留言板，个个都是人才

George Gong

这几个引用还是不太懂啊！

曹铮

1. 一直不太理解弱引用。文中的“比如”在其他好多地方也这么说——“如果试图获取对象时...否则重新实例化”，但习惯并发编程的人会觉得，假如刚实例化之后，又恰好被回收了呢？

2. 后来看了ThreadLocalMap的Entry代码，我会觉得“弱引用”也许为了一些工具类在设计时又要考虑易用性，又要尽量防止开发者编程不当造成内存泄漏，比如Entry弱引用了ThreadLocal<?>，就不会由于Entry本身一直存在使得对应的ThreadLocal<?>实例一直无法回收？

3. new Resource().action()那里我以前一直以为，对象的方法在运行期间一定会持有this引用，间接使得对象的held可达不会被回收。现在看来是错的。

2018-05-28

2018-05-14

2018-05-13

2018-05-14

2018-05-13

2018-05-14

2018-05-13

2018-05-14

2018-05-12

2018-05-12

2018-05-12

2018-05-12

2018-06-09

2018-05-16

2018-05-17

2018-05-14

2018-05-15

2018-05-14

2018-05-14

2018-05-14

2018-05-12

2018-05-12

请老师解惑和纠错一下万分感谢

作者回复

1. 被缓存的对象在使用时是有强引用的
2. 这种通常是在ThreadLocal或者worker线程音乐了，worker不会停的
3. 按照Java语言规范发生回收是合规的，所以极端情况下会出现field不可用情况

funnyx

2018-05-12

有个问题想请教一下峰哥，就是对对象回收的时机，如果最后这个对象被认为要回收，那么会被添加到F-QUEUE的一个队列中，由一个优先级比较低的线程缓慢执行，在JVM对该队列进行标记之后，如果该对象还没有和引用建立关联，那么该对象应该就被回收了，但是如果被调用了finalize()方法，该对象也不一定会被回收，那么该对象从F-QUEUE被移除之后，后续的垃圾回收该如何进行？因为一个finalize()最多只会被调用一次。

作者回复

我理解，FO移除后，如果有幻象引用，就处于幻象可达状态，走对应逻辑，然后就可以销毁了；如果没有幻象引用，就直接处于不可达状态，可以销毁

正是那朵玫瑰

2018-05-12

老师好，有个疑问想问下，弱引用一旦被回收，就会加入注册引用队列，那么引用队列不一样会占用内存，不等于没有回收么？

北风一叶

2018-07-15

表示只懂强引用

LenX

2018-06-19

老师，在 Wikipedia 上有一个例子，如下：

```
class A {
WeakReference r = new WeakReference(new String("I'm here"));

WeakReference sr = new WeakReference("I'm here");

System.gc();
Thread.sleep(100);

// only r.get() becomes null
System.out.println("after gc:r =" + r.get() + ",static=" + sr.get());
}
```

Java 8 环境运行之后，r.get() 返回 null，但是 sr.get() 不返回 null。

老师，对上面的例子，我有 2 个疑惑。

1. 我对 sr 不返回 null 的理解是：

因为 sr 直接引用的常量池中的字面量 "I'm here"，而常量池对这个字面量本身也有引用，所以无法回收。

这样理解对吗？

2. 在上面的例子中，假如没有 sr，也即代码变为：

```
class A {
WeakReference r = new WeakReference(new String("I'm here"));

System.gc();
Thread.sleep(100);

// only r.get() becomes null
System.out.println("after gc:r =" + r.get());
}
```

在这个类中，在 System.gc 前有 3 个对象，分别是 r、String 对象、常量池中的 "I'm here"。那么，在 gc 之后，常量池中的 "I'm here" 会被垃圾回收吗？

桐。

2018-06-13

老师，可以举例说明一下这些引用的使用场景么？

chris

2018-06-12

这四种引用除了强引用，其他的确实了解甚少，今天一看确实受益匪浅！

vincentjia

2018-06-10

希望能有实例（代码、伪代码），更直观、更理解。如果篇幅所限，是否可以链接到你的博客？

Ethan

2018-06-06

然后为什么 ThreadLocalMap 的 Entry 是弱引用呢？是 remove 后下一次 gc 就会立刻释放缓存吧？那为什么普通的 Map 的 entry 又不是弱引用？普通的 map 就不需要把 entry 设置为弱引用？这个区别在哪？

Ethan

2018-06-06

老师有一个地方我不太理解正确：Cleaner 就是给对象设置一个 PhantomReference，在对象被回收前会被 JVM 放进 pending 队列，ReferenceHandler 会在看到这个幻象引用是 Cleaner 时特殊处理，不加入队列而是调用它的 clean 方法

英耀

2018-06-05

请问一下杨老师，能否稍微详谈一下的 post-mortem 机制？指的是幻象引用+引用队列这一套机制吗？

作者回复

2018-06-06

是的

董飞斌

2018-05-22

希望能够听到老师的原声，别人的朗读感觉有点生硬。还有，对于概念的解释，和代码的展示，可以贴个链接。谢谢老师的分享

岁月如歌

2018-05-22

软引用在内存不足时回收，而弱引用在有内存回收发生时就会被回收掉

岁月如歌

软引用 (SoftReference) , 是一种相对强引用弱化一些的引用, 可以让对象豁免一些垃圾收集, 只有当 JVM 认为内存不足时, 才会去试图回收软引用指向的对象

2018-05-22

张小来

对内存敏感的缓存, 老师可以举个例子吗?

作者回复

2018-05-22

图片

Honey拯救世界

apache commons pool 对象池有提供SoftReferencePool实现

2018-05-22

牛肉味鲜果橙

这个系列的学习买的好值, 既能看到作者对Java深层次的讲解, 又能看能评论专区的各种大神的理解。

2018-05-22

kk

平时开发都是上层应用, 基本上都是调用开源库, 这些稍微底层的技术以及和性能相关的技术实现, 都是调用库来做的, 遇到比较多的就是强引用, 和弱引用。看了个把小时对后面的几个知识点都很陌生, 希望能指导一下, 想要更全面的了解这些知识点, 应该做哪些方面的准备

2018-05-21

Miaozhe

关注

2018-05-18作者回复空的Finalize实现, 不会起作用的;  
Finalizer是懒家伙, 试试system.runfinalization;

2018-05-21

尝试后, 还是不行。  
我测试发现, 自定义的finalize只要有任何变量或对象定义, 都不会进入Reference Queue, 怀疑是, 以为级联依赖的问题, 就是说Finalize方法中, 有未被回收的对象, 那么Finalize所在的对象, 就不会被回收。

ls

按GC层面自己的理解: 强引用是宁愿抛OOM异常也不会回收的对象; 软引用是内存不足时会去回收的对象; 弱引用只要GC线程有扫描到弱引用就会被回收; 而一个对象只有虚引用, 等于没有引用一样, 随时被GC回收;

2018-05-19

关于引用, 最近Android有遇到一个小坑: 自定义了一个函数, 里面 new 了一个实例 A 给 a 引用 (强引用) , 在函数中用 a 去调用底层的函数 (底层开线程, 耗时耗内存) , 没考虑到函数过后, 强引用就已经过了作用域, 当底层吃内存多后触发了 GC 把这个引用和对象回收了; 而B对象引用是在 a 引用链底下, 当顶层引用被回收了, B 也会被回收, B 重写的 finalize 函数又把底层的线程给 kill 了, 导致函数调用了, 但C层的代码在一段时间后 (内存紧张被GC) 就不起作用, 没基础真可怕, 赶紧翻看这讲和上讲的文章补基础知识;

Nemo

try { // 需要被保护的代码  
int i = myIndex; Resource.update(externalResourceArray[i]);  
} finally {  
// 调用 reachabilityFence, 明确保障对象 strongly reachable Reference.reachabilityFence(this);  
}  
老师这段代码finally中指定强可达可以理解, 不过如果action执行结束以后, 这个强引用会被释放掉吗? 在什么时候被释放, 还是也需要显示的调用api释放呢?

2018-05-19

jon

学习了, 可是除了强引用平时在用, 其他引用要怎么用呢?

2018-05-18

arthur

杨老师您好, 我是测试人员, 看懂了这些知识点, 但平时大都只测试脚本, 不会去实现功能, 所以对应用场景不是很了解, 希望老师能多给一些实例, 举一些代码的例子, 方便像我这些基础比较差的同学学习

2018-05-17

Miaozhe

老师, 问个问题: 我自己定义一个类, 重写finalize方法后, 创建一个对象, 被幻想引用, 同时该幻想对象使用ReferenceQueue,  
当我这个对象指向null, 被GC回收后, ReferenceQueue中没有改对象, 不知道是什么原因? 如果我把类中的finalize方法移除, ReferenceQueue就能获取被释放的对象。

2018-05-17

作者回复

文章图里阐明了, 幻象引用enqueue发生在finalize之后, 你查看是不是卡在FinalReference queue里了, 那是实现finalization的地方

2018-05-18

000

除了强引用, 其他的三种平时开发中几乎没使用过

2018-05-17

落叶飞逝的恋

几种引用会怎么出现相互转化

2018-05-17

微笑的向日葵

并没有 第一次听说这个东西

2018-05-16

成

有个关于new Resource().action()的问题, 在action方法里隐式的this不就说明了可达性吗? 望解答

2018-05-16

刀健笑

\*这就可以用来构建一种没有特定约束的关系, 比如, 维护一种非强制性的映射关系, 如果试图获取时对象还在, 就使用它, 否则重现实例", 否则重现实例"是指?

2018-05-16

fly

2018-05-15

我们一般所说的内存泄漏是遍历gcroots, 那泄漏的对象是在gcroots上的, 怎么判定对象泄漏了呢? 泄漏对象和未泄漏对象有不同的标志吗?  
等待老师的解答

Geek\_5b11b8

看了之后，收获不是很大，有没有具体的应用案例或者实际开发中的应用场景

2018-05-15

张勇

创建一个Student的强引用对象stu： Student stu=new Student(小王",3); 创建一个弱引用指向Student SoftReference<Student> softReference=new SoftReference<Student>(stu); 这句话对不对？ 问题1 如果stu不手动的置成stu=null，就算gc快要发生OOM的时候也不会回收这个Student对象因为它还持有一个强引用stu。这句话对不对？ 问题2 如果stu我置换成stu=null，此时只有弱引用指向了stu这个对象这时当内存不足快要发生OOM的时候gc会回收Student占用的这块内存。这句话对不对？ 问题3，如果我从弱引用中获取我这个Student对象，写成 if(softReference.get()!=null) Student student=softReference.get(); 此时这个student是强引用还是弱引用，如果是强引用是不是用完以后需要写成student==null,如果不写student==null这句 gc快要发生OOM的时候也不会回收这个Student对象.下次如果继续使用这个对象是不是用同样的方法if(softReference.get()!null) Student student=softReference.get(); 在获取这个对象

老师我这个例子是针对全局变量的，如果是全局变量我上面的问题帮我解答下，谢谢

张勇

创建一个Student的强引用对象stu： Student stu=new Student(小王",3); 创建一个弱引用指向Student SoftReference<Student> softReference=new SoftReference<Student>(stu); 这句话对不对？ 问题1 如果stu不手动的置成stu=null，就算gc快要发生OOM的时候也不会回收这个Student对象因为它还持有一个强引用stu。这句话对不对？ 问题2 如果stu我置换成stu=null，此时只有弱引用指向了stu这个对象这时当内存不足快要发生OOM的时候gc会回收Student占用的这块内存。这句话对不对？ 问题3，如果我从弱引用中获取我这个Student对象，写成 if(softReference.get()!=null) Student student=softReference.get(); 此时这个student是强引用还是弱引用，如果是强引用是不是用完以后需要写成student==null,如果不写student==null这句 gc快要发生OOM的时候也不会回收这个Student对象.下次如果继续使用这个对象是不是用同样的方法if(softReference.get()!null) Student student=softReference.get(); 在获取这个对象

作者回复

你这几个例子用的都是局部变量？这是有作用域的

xuejian.sun

老师，请教个问题，map中有个普通类A，A中引用了一个loc管理的B，map.remove(A) 后这个A会被GC吗

作者回复

不太清楚问题，只少remove以后A B对象都和map没关系了

张勇

对象从软或者弱引用变成强引用后对对象还会被gc回收吗？

作者回复

取决于什么时候释放，如果一直保持着，就可能是内存泄露的来源了

张勇

若果我一个强引用的对象只是new出来，并没有使用new出来的这个对象，gc也不会回收这个对象吗？ eg：Animal an=new Animal(); an这个对象我在程序中并不使用，是不是gc不会回收这个对象？

作者回复

会，建议去理解下局部变量作用域

Mr. Zhang

还是会配合例子讲解会比较清楚

夏茶

杨老师我有个问题，想请您帮我解答下，谢谢，代码荣耀说了一句“如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中”，对象被垃圾回收了，把软引用加入到队列里，那这里的软引用是对象吗，如果是对象是什么对象，还能不能取到软引用所引用的对象？

作者回复

不是回收，是jvm判断对象处于soft reachable，立即或者过一会儿enqueue到队列里

jimforcode

大神多来点干货啊，底层的东西太抽象了，看完还是没明白

凉白开

threadlocalmap里的key为弱引用 使用过后 应该remove 否则容易出现oom 因为可能key被回收了 它的值还在

作者回复

正解，前面回复过，TtheadLocal是和线程生命周期绑定的，所以遇到线程池就可能出现这种问题，因为worker通常一直活着

王磊

- 1.'对象可达性状态流转分析'章节图示是强软弱幻象引用，而文字说的是强软弱幻象可达，是不是应该都是可达？
2. 另外，我注意到软可达和弱可达是单向的，不能从弱到软，这里能多解释下吗？
3. 说幻象可达是finalize之后的，这个什么意思？我理解只要是幻象引用，它就是幻象可达，和是否执行了finalize()没有关系

请老师解答

作者回复

感谢反馈，

- 1, 2, 不错，画的比较粗糙，注明了不那么严谨，主要是个清楚的示意；
- 3, 不是的，Javaspec清楚要求如此：对象处于某种可达状态，不是说有那种引用，是没有其他更“强”的引用关系； finalize()如果没有重新不会执行，但逻辑顺序不变

Hesher

这篇讲得好，以前看书缺少实际使用经验，理解上总是一知半解最后忘了，这次感觉看完就明白了。还是要多结合实践，才能真正理解这些很少用到的东西。

作者回复

很高兴有帮助

thinkers

除了8种基本数据类型(int,short,byte,long,double,float,boolean,char)，其他都是引用类型！感觉这种引用的分类没有任何实际作用，开发中基本可以忽略点！

作者回复

2018-05-12

除非对于性能完全不敏感，这就是差不多和极致的差别，有兴趣可以看看netty之类底层的优化

疯狂的柴犬

2018-05-12

每天都等着更新，讲的挺细的，waiting下一个

## 第5讲 | String、StringBuffer、StringBuilder有什么区别?

2018-05-15 杨晓峰



今天我会聊聊日常使用的字符串，别看它似乎很简单，但其实字符串几乎在所有编程语言里都是个特殊的存在，因为不管是数量还是体积，字符串都是大多数应用中的重要组成。

今天我要问你的是，[理解Java的字符串，String、StringBuffer、StringBuilder有什么区别？](#)

## 典型回答

`String`是Java语言非常基础和重要的类，提供了构造和管理字符串的各种基本逻辑。它是典型的`Immutable`类，被声明成为`final class`，所有属性也都是`final`的。也由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的`String`对象。由于字符串操作的普遍性，所以相关操作的效率往往对应用性能有明显影响。

`StringBuffer`是为解决上面提到拼接产生太多中间对象的问题而提供的一个类。我们可以用`append`或者`add`方法，把字符串添加到已有序列的末尾或者指定位置。`StringBuffer`本质上是一个线程安全的可修改字符序列，它保证了线程安全，也随之带来了额外的性能开销，所以除非有线程安全的需要，不然还是推荐使用它的后继者，也就是`StringBuilder`。

`StringBuilder`是Java 1.5中新增的，在能力上和`StringBuffer`没有本质区别，但是它去掉了线程安全的部分，有效减小了开销，是绝大部分情况下进行字符串拼接的首选。

## 考点分析

几乎所有的应用开发都离不开操作字符串，理解字符串的设计和实现以及相关工具如拼接类的使用，对写出高质量代码是非常有帮助的。关于这个问题，我前面的回答是一个通常的概率性回答，至少你要知道`String`是`Immutable`的，字符串操作不当可能会产生大量临时字符串，以及线程安全方面的区别。

如果继续深入，面试官可以从各种不同的角度考察，比如可以：

- 通过`String`和相关类，考察基本的线程安全设计与实现，各种基础编程实践。
- 考察JVM对象缓存机制的理解以及如何良好地使用。
- 考察JVM优化Java代码的一些技巧。
- `String`相关类的演进，比如Java 9中实现的巨大变化。
- ...

针对上面这几方面，我会在知识扩展部分与你详细聊聊。

## 知识扩展

## 1.字符串设计和实现考量

我在前面介绍过，`String`是`Immutable`类的典型实现，原生的保证了基础线程安全，因为你无法对它内部数据进行任何修改，这种便利甚至体现在拷贝构造函数中，由于不可变，`Immutable`对象在拷贝时不需要额外复制数据。

我们再来看`StringBuffer`实现的一些细节，它的线程安全是通过把各种修改数据的方法都加上`synchronized`关键字实现的，非常直白。其实，这种简单粗暴的实现方式，非常适合我们常见的线程安全类实现，不必纠结于`synchronized`性能之类的事，有人说“过早优化是万恶之源”，考虑可靠性、正确性和代码可读性才是大多数应用开发最重要的因素。

为了实现修改字符序列的目的，`StringBuffer`和`StringBuilder`底层都是利用可修改的（`char`, JDK 9以后是`byte`）数组，二者都继承了`AbstractStringBuilder`，里面包含了基本操作，区别仅在于最终的方法是否加了`synchronized`。

另外，这个内部数组应该创建多大的呢？如果太小，拼接的时候可能要重新创建足够大的数组；如果太大，又会浪费空间。目前的实现是，构建时初始字符串长度加16（这意味着，如果没有构建对象时输入最初的文字串，那么初始值就是16）。我们如果确定拼接会发生非常多次，而且大概是可预计的，那么就可以指定合适的大小，避免很多次扩容的开销。扩容会产生多重开销，因为要抛弃原有数组，创建新的（可以简单认为是倍数）数组，还要进行`arraycopy`。

前面我讲的这些内容，在具体的代码书写中，应该如何选择呢？

在没有线程安全问题的情况下，全部拼接操作是应该都用StringBuilder实现吗？毕竟这样书写的代码，还是要多敲很多字的，可读性也不理想，下面的对比非常明显。

```
String strByBuilder = new
StringBuilder().append("aa").append("bb").append(
    "cc").append("dd").toString();

String strByConcat = "aa" + "bb" + "cc" + "dd";
```

其实，在通常情况下，没有必要过于担心，要相信Java还是非常智能的。

我们做个实验，把下面一段代码，利用不同版本的JDK编译，然后再反编译，例如：

```
public class StringConcat {
    public static void main(String[] args) {
        String myStr = "aa" + "bb" + "cc" + "dd";
        System.out.println("My String:" + myStr);
    }
}
```

先编译再反编译，比如使用JDK 9：

```
 ${JAVA9_HOME}/bin/javac StringConcat.java
 ${JAVA9_HOME}/bin/javap -v StringConcat.class
```

JDK 8的输出片段是：

```
6: new      #4          // class java/lang/StringBuilder
9: dup
10: invokespecial #5      // Method java/lang/StringBuilder."<init>":()
13: ldc      #6          // String My String;
15: invokevirtual #7      // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
18: aload_1
19: invokevirtual #7      // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
22: invokevirtual #8      // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
```

而在JDK 9中，反编译的结果就非常简单了，片段是：

```
7: invokedynamic #4, 0      // InvokeDynamic #0:makeConcatWithConstants:(Ljava/lang/String;)Ljava/lang/String;
```

你可以看到，在JDK 8中，字符串拼接操作会自动被javac转换为StringBuilder操作，而在JDK 9里面则是因为Java 9为了更加统一字符串操作优化，提供了StringConcatFactory，作为一个统一的入口。javac自动生成的代码，虽然未必是最优化的，但普通场景也足够了，你可以酌情选择。

## 2.字符串缓存

我们粗略统计过，把常见应用进行堆转储（Dump Heap），然后分析对象组成，会发现平均25%的对象是字符串，并且其中约半数是重复的。如果能避免创建重复字符串，可以有效降低内存消耗和对象创建开销。

String在Java 6以后提供了intern()方法，目的是提示JVM把相应字符串缓存起来，以备重复使用。在我们创建字符串对象并调用intern()方法的时候，如果已经有缓存的字符串，就会返回缓存里的实例，否则将其缓存起来。一般来说，JVM会将所有的类似“abc”这样的文本字符串，或者字符串常量之类缓存起来。

看起来很不错是吧？但实际情况估计会让你大跌眼镜。一般使用Java 6这种历史版本，并不推荐大量使用intern，为什么呢？魔鬼存在于细节中，被缓存的字符串是存在所谓PermGen里的，也就是臭名昭著的“永久代”，这个空间是很有限的，也基本不会被FullGC之外的垃圾收集照顾到。所以，如果使用不当，OOM就会光顾。

在后续版本中，这个缓存被放置在堆中，这样就极大地避免了永久代占满的问题，甚至永久代在JDK 8中被MetaSpace（元数据区）替代了。而且，默认缓存大小也在不断地扩大中，从最初的1009，到7u40以后被修改为60013。你可以使用下面的参数直接打印具体数字，可以拿自己的JDK立刻试验一下。

```
-XX:+PrintStringTableStatistics
```

你也可以使用下面的JVM参数手动调整大小，但是绝大部分情况下并不需要调整，除非你确定它的大小已经影响了操作效率。

```
-XX:StringTableSize=N
```

Intern是一种显式地排重机制，但是它也有一定的副作用，因为需要开发者写代码时明确调用，一是不方便，每一个都显式调用是非常麻烦的；另外就是我们很难保证效率，应用开发阶段很难清楚地预计字符串的重复情况，有人认为这是一种污染代码的实践。

幸好在Oracle JDK 8u20之后，推出了一个新的特性，也就是G1 GC下的字符串排重。它是通过将相同数据的字符串指向同一份数据来做到的，是JVM底层的改变，并不需要Java类库做什么修改。

注意这个功能目前是默认关闭的，你需要使用下面参数开启，并且记得指定使用G1 GC：

```
-XX:+UseStringDeduplication
```

前面说到的几个方面，只是Java底层对字符串各种优化的一角，在运行时，字符串的一些基础操作会直接利用JVM内部的Intrinsic机制，往往运行的就是特殊优化的本地代码，而根本就不是Java代码生成的字节码。Intrinsic可以简单理解为是一种利用native方式hard-coded的逻辑，算是一种特别的内联。很多优化还是需要直接使用特定的CPU指令，具体可以看相关[源码](#)，搜索“string”以查找相关Intrinsic定义。当然，你也可以在启动实验应用时，使用下面参数，了解intrinsic发生的状态。

```
-XX:+PrintCompilation -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
//样输出片段
 180  3      java.lang.String::charAt (25 bytes)
@ 1   java.lang.String::isLatin1 (19 bytes)
...
@ 7 java.lang.StringUTF16::getChar (60 bytes) intrinsic
```

可以看出，仅仅是字符串一个实现，就需要Java平台工程师和科学家付出如此大且默默无闻的努力，我们得到的很多便利都是来源于此。

我会在专栏后面的JVM和性能等主题，详细介绍JVM内部优化的一些方法，如果你有兴趣可以再深入学习。即使你不做JVM开发或者暂时还没有使用到特别的性能优化，这些知识也能帮助你增加技术深度。

### 3.String自身的演化

如果你仔细观察过Java的字符串，在历史版本中，它是使用char数组来存数据的，这样非常直接。但是Java中的char是两个bytes大小，拉丁语系语言的字符，根本就不需要太宽的char，这样无区别的实现就造成了一定的浪费。密度是编程语言平台永恒的话题，因为归根结底绝大部分任务是要操作数据的。

其实在Java 6的时候，Oracle JDK就提供了压缩字符串的特性，但是这个特性的实现并不是开源的，而且在实践中也暴露出了一些问题，所以在最新的JDK版本中已经将它移除了。

在Java 9中，我们引入了Compact Strings的设计，对字符串进行了大刀阔斧的改进。将数据存储方式从char数组，改变为一个byte数组加上一个标识编码的所谓coder，并且将相关字符串操作类都进行了修改。另外，所有相关的Intrinsic之类也都进行了重写，以保证没有任何性能损失。

虽然底层实现发生了这么大的改变，但是Java字符串的行为并没有任何大的变化，所以这个特性对于绝大部分应用来说是透明的，绝大部分情况不需要修改已有代码。

当然，在极端情况下，字符串也出现了一些能力退化，比如最大字符串的大小。你可以思考下，原来char数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成byte数组，同样数组长度下，存储能力是退化了一倍的！还好这是存在于理论中的极限，还没有发现现实应用受到影响。

在通用的性能测试和产品实验中，我们能非常明显地看到紧凑字符串带来的优势，即更小的内存占用、更快的操作速度。

今天我从String、StringBuffer和StringBuilder的主要设计和实现特点开始，分析了字符串缓存的intern机制、非代码侵入性的虚拟机层面排重、Java 9中紧凑字符的改进，并且初步接触了JVM的底层优化机制Intrinsic。从实践的角度，不管是Compact Strings还是底层Intrinsic优化，都说明了使用Java基础类库的优势，它们往往能够得到最大程度、最高质量的优化，而且只要升级JDK版本，就能零成本地享受这些益处。

### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？限于篇幅有限，还有很多字符相关的问题没有来得及讨论，比如编码相关的问题。可以思考一下，很多字符串操作，比如getBytes()或String(byte[] bytes)等都是隐含着使用平台默认编码，这是一种好的实践吗？是否有利于避免乱码？

请你在留言区写写你对这个问题的思考，或者分享一下你在操作字符串时掉过的坑，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



Bin

2018-05-16

jdk1.8中，String是标准的不可变类，但其hash值没有用final修饰，其hash值计算是在第一次调用hashCode方法时计算，但方法没有加锁，变量也没用volatile关键字修饰就无法保证其可见性。当有多个线程调用的时候，hash值可能会被计算多次，虽然结果是一样的，但JDK的作者为什么不将其优化一下呢？

作者回复

2018-05-17

这些“优化”在通用场景可能变成持续的成本，volatile read是有明显开销的；如果冲突并不多见，read才是更普遍的，简单的cache是更高效的

公号-Java大前端

2018-05-15

String/StringBuffer/StringBuilder :

今日 心得

## 1 String

### (1) String的创建机制

由于String在Java世界中使用过于频繁，Java为了避免在一个系统中产生大量的String对象，引入了字符串常量池。其运行机制是：创建一个字符串时，首先检查池中是否有值相同的字符串对象。如果有则不需要创建直接从池中刚查找到的对象引用；如果没有则新建字符串对象，返回对象引用，并且将新创建的对象放入池中。但是，通过new方法创建的String对象是不检查字符串池的，而是直接在堆区或栈区创建新的对象，也不会把对象放入池中。上述原则只适用于通过直接量给String对象引用赋值的情况。

举例：String str1 = "123"; //通过直接量赋值方式，放入字符串常量池  
String str2 = new String("123"); //通过new方式赋值方式，不放入字符串常量池

注意：String提供了intern方法。调用该方法时，如果常量池中包括了一个等于此String对象的字符串（由equals方法确定），则返回池中的字符串。否则，将此String对象添加到池中，并且返回此池中对象的引用。

### (2) String的特性

[A] 不可变。是指String对象一旦生成，则不能再对它进行改变。不可变的主要作用在于当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁等待的时间，从而大幅度提高系统性能。不可变模式是一个可以提高多线程程序的性能，降低多线程程序复杂度的设计模式。

[B] 针对常量池的优化。当2个String对象拥有相同的值时，他们只引用常量池中的同一个拷贝。当同一个字符串反复出现时，这个技术可以大幅度节省内存空间。

## 2 StringBuffer/StringBuilder

StringBuffer和StringBuilder都实现了AbstractStringBuilder抽象类，拥有几乎一致对外提供的调用接口；其底层在内存中的存储方式与String相同，都是以一个有序的字符序列（char类型的数据）进行存储。不同点是StringBuffer/StringBuilder对象的值是可以改变的，并且值改变以后，对象引用不会发生改变；两者对象在构造过程中，首先按照默认大小申请一个字符数组，由于会不断加入新数据，当超过默认大小后，会创建一个更大的数组，并将原先的数组内容复制过来，再丢弃旧的数组。因此，对于较大对象的扩容会涉及大量的内存复制操作，如果能够预先评估大小，可提升性能。

唯一需要注意的是，StringBuffer是线程安全的，但是StringBuilder是线程不安全的。可参看Java标准类库的源代码，StringBuffer类中方法定义前面都会有synchronized关键字。为此，StringBuffer的性能要远低于StringBuilder。

## 3 应用场景

[A]在字符串内容不经常发生变化的业务场景优先使用String类。例如：常量声明、少量的字符串拼接操作等。如果有大量的字符串内容拼接，避免使用String与String之间的“+”操作，因为这样会产生大量无用的中间对象，耗费空间且执行效率低下（新建对象、回收对象花费大量时间）。

[B]在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在多线程环境下，建议使用StringBuffer，例如XML解析、HTTP参数解析与封装。

[C]在频繁进行字符串的运算（如拼接、替换、删除等），并且运行在单线程环境下，建议使用StringBuilder，例如SQL语句拼装、JSON封装等。

作者回复

2018-05-15

很到位

Hidden

2018-05-16

公司没有技术氛围，项目也只是功能实现就好，不涉及优化，技术也只是传统技术，想离职，但又怕裸辞后的各种压力

sea陪我看海

2018-05-16

作者我有个疑问，String myStr = "aa" + "bb" + "cc" + "dd"; 应该编译的时候就确定了，不会用到StringBuilder。理由是：

```
String myStr = "aa" + "bb" + "cc" + "dd";
String h = aabbccdd;
Mystr == h 上机实测返回的是true，如果按照你的说法，应该是返回false才对，因为你说拼接用到stringbuilder，那mystr应该是堆地址，h是常亮池地址。
```

KongKong

2018-05-16

编译器为什么把  
String myStr = "aa" + "bb" + "cc" + "dd";  
默认优化成  
String myStr = "aabbccdd";  
这样不是更聪明嘛

愉悦在花香的日子里

2018-05-15

getBytes和String相关的转换时根据业务需要建议指定编码方式，如果不指定则看看JVM参数里有没有指定file.encoding参数，如果JVM没有指定，那使用的默认编码就是运行的操作系统环境的编码了，那这个编码就变得不确定了。常见的编码iso8859-1是单字节编码，UTF-8是变长的编码。

作者回复

2018-05-15

不错，莫依赖于不确定因素

肖一林

2018-05-15

这篇文章写得不错，由浅入深，把来龙去脉写清楚了

作者回复

2018-05-15

谢谢认可

薛好运

2018-05-15

老师，可以讲解这一句话的具体含义吗，谢谢！  
你可以思考下，原来char数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成byte数组，同样数组长度下，存储能力是退化了一倍的！还好这是存在于理论中的极限，还没有发现现实应用受此影响。

作者回复

2018-05-15

已回答，一个char两个byte，注意下各个类型宽度

Jerry银根

2018-05-15

要完全消化一篇文章的所有内容，真得不是一天就能搞定的，可能需要一个月，甚至好几个月。就比如今天的字符串，我觉得这个话题覆盖的面太广：算法角度、易用角度、性能角度、编码传输角度等

但是好在，我获得见识。接下来，花时间慢慢研究呗，连大师们都花了那么多时间研究，我们多花点时间，很正常嘛◆◆

一点学习心得，和大家分享

Jerry银根

2018-05-15

特别喜欢这句话：“仅仅是字符串一个实现，就需要 Java 平台工程师和科学家付出如此大且默默无闻的努力，我们得到的很多便利都是来源于此。”

我想说，同学们，写代码的时候记得感恩哦◆◆

对于字符串的研究，我觉得能很好的理解计算机的本质和训练计算机思维，提升自己解决问题的能力。

小小字符串有着诸多巨人的影子

作者回复

2018-05-15

非常感谢

明翼

2018-06-25

```
回答一下上面一个人的问题，问题是“String s3 = new String("12") + new String("34");
s3.intern();
String s4 = "1234";
System.out.println(s3 == s4); //true
```

求解，为什么在第二段比较中会返回true，从字节码看s3应该就是生成了一个stringbuilder对象完成连接操作后执行了toString，s3不是应该仍然是堆内的对象地址吗？为什么会和常量池中的地址相等？”

我之前也是不明白s3为什么等于s4，查了下资料，说是在jdk1.7之后，如果字符串在堆中有实例，那intern方法就会把这个字符串的引用放在字符串常量池里，所以，String s3 = new String("12") + new String("34");这里在字符串常量池里放了一个字符串“12”，一个字符串“34”，在堆里存放他们的运算结果“1234”，然后把“1234”的引用返回给s3，s3.intern()这段代码运行时，JVM在堆里先到了字符串“1234”，所以就会把它的引用放到字符串常量池里，这个引用和s3相等，String s4 = "1234";这个代码时，会把字符串常量池里“1234”的引用返回给s4，所以s3等于s4的。

个人理解，如有不对，请指正，谢谢◆◆

echo

2018-05-16

String是immutable，在security, Cache, Thread Safe等方面都有很好的体现。

Security: 参考的时候我们很多地方使用String参数，可以保证参数不会被改变，比如数据库连接参数url等，从而保证数据库连接安全。

Cache: 因为创建String前先在Constant Pool里面查看是否已经存在此字符串，如果已经存在，就把该字符串的地址引用赋给字符变量；如果没有，则在Constant Pool创建字符串，并将字符串引用赋给字符串变量。所以存在多个引用指向同一个字符串对象，利用缓存有助于提高内存开销。

Thread Safe: 因为String是immutable，所以它是Automatically thread safe。

问题：我一直不能很好的理解最后一个体现，到底String是如何体现在thread safe，老师能不能用简短的线程代码给讲解？非常感激。

Chris

2018-05-15

new string ("ghhh") .intern () ; 会从堆到常量池是这个作用吗

轩尼诗。

2018-05-31

String s = new String("abc") 创建了几个对象？ 答案1：在字符串常量池中查找有没有“abc”，有则作为参数，就是创建了一个对象；没有则在常量池中创建，然后返回引用，那就是创建了两个对象。答案2：直接在堆中创建一个新的对象。不检查字符串常量池，也不会把对象放入池中。网上正确答案貌似是两个。求指教到底是哪个！ 第一种可以写成String a = "abc"; String s = new String(a)，那么第一种解释说得通。String a = "abc"会在常量池创建"abc"。但是String类的intern()是在字符串常量池中查找该字符串，有就返回，没有就创建。如果第一种解释说得通，那这个方法就废了。

王万云

2018-05-20

看大神的文章真的提高太多了，而且还要看评论，评论区也都是高手云集

王建坤

2018-06-24

讲师你好，有个疑问：“默认缓存大小也在不断地扩大中，从最初的1009，到7u40以后被修改为60013。”  
这里的1009 60013是指字符串的个数？还是内存占用？如果是内存占用的话，那单位是什么？

LenX

2018-06-18

老师，这章学习到了 Java 8 以后，字符串常量池被移到了堆中，那么，如果通过 String.intern() 产生了大量的字符串常量，JVM 会对它们进行垃圾回收吗？

作者回复

2018-06-20

会，具体时机我也没注意

jamie

2018-06-14

编译器为什么不把
String myStr = "aa" + "bb" + "cc" + "dd";
默认优化成
String myStr = "aabbccdd";
这样才是更聪明嘛

这个我反编译试过，已经优化成aabbccdd了

Olm.chenteng

2018-06-11

```
String s1 = new String("do");
s1.intern();
String s2 = "do";

System.out.println(s1 == s2); //false

String s3 = new String("12") + new String("34");
s3.intern();
String s4 = "1234";
System.out.println(s3 == s4); //true
```

求解，为什么在第二段比较中会返回true，从字节码看s3应该就是生成了一个stringbuilder对象完成连接操作后执行了toString，s3不是应该仍然是堆内的对象地址吗？为什么会和常量池中的地址相等？

Io\_

2018-05-28

代码类型说new的时候不检查字符串常量池，那请问老师
String str1 = "abc";
String str2 = new String("abc");
第二行代码到底创建了几个对象呢，网上说1个，因为abc已经在常量池，但是如果new不检查常量池那是不是应该是两个对象呢？

So Leung

那想问下，当new String("a")时，不会在常量池创建对象？

©

String s2=new String("AB"), , 如果，常量池中没有AB,那么会不会去常量池创建，望解答

作者回复

new只是创建新的；另外，有没有想过怎么通过一段程序证明？这样更有助于理解

Miaozhe

杨老师，问个问题：

String str1 = "abc";

String str2 = new String("abc");

理论上，str1是放入字符串常量池，str2是新增一个对象，新开辟一块地址。  
但是通过代码调试，他们的HashCode一样，就说明他们是引用同一个地址。

请帮忙分析一下。

作者回复

据我所知，字符串hashCode是取决于内容，而不是地址；  
对象是否同用 == 判断；

一定要区分， ==, equals, hashCode，搜下或者看看书吧

debugable

java9讲字符串内部使用字节数组保存，取一个中文字符串字符个数是不是就不能用length了？

作者回复

不用担心，我提到了API行为没变化，包括charAt之类全部都是如此

曹铮

思考题里的平台默认编码，平台指的是JVM所在的操作系统，还是指语言平台本身呢？

作者回复

环境编码

嘎哈

char 数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成 byte 数组，同样数组长度下，存储能力是退化了一倍的！  
怎么理解呢？举个例子呗

作者回复

已回复，一个char两个byte，注意下各个类型宽度

jutsu

地铁上看起来啦，从来没有string类型的线程问题，受教了

作者回复

谢谢

呆

String对操作符的覆写是怎样实现的呢，这也算是一类操作符重载吗？

小潘

string的intern方法在fastjson中就出现过bug

风起

在http传进来的string统一就是utf-8 等出去的话会肯定转一下格式。  
然后对于循环外的就用加号， 循环内就用StringBuilder，

过冬

老师，是退化还是进化？

替换成 byte 数组，同样数组长度下，存储能力是退化了一倍的！  
char变byte，数组长度不变，占用空间更小才对啊？

作者回复

退化，占用空间小，最大能表达的信息就少了一半

aoe

学习笔记  
简单的拼接+就行了。只有相对复杂的，比如需要优化buffer大小，才有必要考虑。

在 JDK 8 中，字符串拼接操作会自动被 Java 转换为 StringBuilder 操作。

而在 JDK 9 里面则是因为 Java 9 为了更高的字符串操作优化，提供了 StringConcatFactory，作为一个统一的入口。  
Java 自动生成代码，虽然未必是最优化的，但普通场景也足够了。在 Java 9 中，我们引入了 Compact Strings 的设计，对字符串进行了大刀阔斧的改进。将数据存储方式从 char 数组，  
改变为一个 byte 数组加上一个标准编码的所谓 coder，并且将相关字符串操作类都进行了修改。

在极限情况下，字符串也出现了一些能力退化，比如最大字符串的大小。你可以思考下，原来 char 数组的实现。

字符串的最大长度就是数组本身的长度限制，但是替换成 byte 数组，同样数组长度下，存储能力是退化了一倍的！

还好这是存在于理论中的极限，还没有发现现实应用受此影响。

在通用的性能测试和产品实验中，我们能非常明显地看到紧凑字符串带来的优势，即更小的内存占用、更快的操作速度。

helloworld

2018-05-24

2018-05-23

2018-05-24

2018-05-18

2018-05-19

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-07-17

2018-06-26

2018-06-13

2018-06-07

2018-06-07

2018-06-05

2018-06-04

因为还没有开始学习JVM相关的知识，一涉及到其相关的内容后就感觉似懂非懂了！马上学习JVM相关的知识，希望老师尽早推出！

今天也在为演唱会门票努力着

上面那个问为什么不能把“aa”+“bb”优化成aabbcc的 据我所知 JDK8中已经实现了这种默认优化

作者回复

嗯，会优化，语义上和StringBuilder一样

小墨迹

Jdk1.8

```
String str = "aa";
str = str + "bb";
与
String str = "aa"+ "bb"
反编译结果不一样。求解。谢谢哒。
什么格式的字符串会智能优化
```

作者回复

这的看编译器的策略，具体要问编译器专家了。  
也许相关，9里改为StringConcatFactory，一个原因就是便于一致的优化，不然优化逻辑是碎片、脆弱的

熙

我在思考从字符串常量池检索是否已存在字符串时用的什么算法，貌似没有这块性能问题的？

作者回复

如果我没记错就是哈希表

So Leung

经过验证new String时，不会再常量池中创建对象。

作者回复

嗯，重要的不是结论，而是如何得到结论

◎◎

```
String s1=new String("StringTest");□
System.out.println(s1.intern() ==s1);
//false (JDK 8)
```

```
String s1 = new StringBuilder().append("String").append("Test").toString();□
System.out.println(s1.intern() == s1); //true (JDK 8)
```

```
String s1 = new StringBuilder("StringTest").toString();□
System.out.println(s1.intern() == s1);
//false (JDK 8)
。。老师，append为什么会造成这个差异
```

岁月如歌

stringbuffer的内容可以变化，但是它是线程安全的，stringbulder是非线程安全的，性能更高

张高凯

很不错，深入浅出，涉及到的技术原理都覆盖了，告诉了读者技术方向，想深入的可以继续翻阅更多资料

作者回复

谢谢

df1996

老师你好，为什么我用jdk8试出来aa+bb+cc编译反编译是直接变成了aabbcc，但是我拼接的时候才用stringbuilder

作者回复

这个就是示例，具体不同版本、jdk、平台可能有区别。

9的字节码也仅仅是展示了indy模式的输出，实际有非常多模式，字节码也是不同的，只是接触一下，不过没必要都列出来

刘为红

你回答的“环境编码”不大明白，具体什么地方配置的

刘为红

getBytes()/String(byte[] bytes)使用过，遇到乱码情况，通过使用编码解决，所以这种机制我认为不好，跟平台的编码相关，但我没搞清楚平台的编码是指操作系统的编码？还是java虚拟机的编码

小沙

string为啥要设计成final？

sunlight001

string new对象的时候不是在 创建两个对象吗，一个放在栈中，一个放在常量池中吗？另外stringbuilder比stringbuffer的性能只有微弱的提高，在千万次的测试中，没有数量级的差别，所以我这程序拼接的时候一般都用stringbuffer；

免斯基

印象中win平台默认是gbk编码，mac, Linux是utf8，前一段时间程序乱码问题就是因为开发用win,部署用linux，我觉得还是统一好，现在我们这边全部统一utf8,最起码这种乱码问题避免了

你好，能否讲解一下StringConcatFactory的内在原理，最好能在源码这一层，谢谢。

雪域飞鸿

String的不可变是指用不可变吗？既然里面是一个char类型数组，我能改变数组中的某一个元素吗？  
编码中字符串问题必须注意，否则会引起各种坑.....

xqqsliver

String q="we";q= "z"+q;是两个不同的对象是吗?  
作者回复

思考下，怎么判断两个对象是不是相同？

胖

javac 自动生成的代码，虽然未必是最优化的，但普通场景也足够了。你可以酌情选择。  
所以平常编码，直接用+进行字符串拼接，并不需要显式调用stringbuilder？

作者回复

简单的拼接+就行了，只有相对复杂的，比如需要优化下buffer大小，才有必要考虑

王磊

假如我执行如下操作，并且常量池里确实已缓存过“abc”，那么执行intern()后s会指向常量池的“abc”，堆中的“abc”会被回收，是这样吗？  
String s = new String("abc");  
s.intern()

这样的话就会多产生一个垃圾。不想产生垃圾的话，就创建时直接指向常量池中的“abc”  
String s = "abc";

请老师确实理解是否正确。

作者回复

我理解是

一笑奈何

char 数组的实现，字符串的最大长度就是数组本身的长度限制，但是替换成 byte 数组，同样数组长度下，存储能力是退化了一倍的！同问。不理解这句话。

作者回复

Java一个char是两个byte呀

雪未央

char占两个byte.同样长度的char数组和byte数组容量差两倍，不知道这样理解对不对？

作者回复

正解

冬末未来

用过String最大的坑就是subString方法，在1.6版本导致的内存泄露，老师都没有讲

作者回复

谢谢指出

Leo

老师，jdk6如果不使用intern方法，字符串常量不是存在永久代的常量池中吗？我的理解是intern方法是减少了永久代中字符串常量的数量，至于效率问题，是因为使用intern方法本身，多做了一步操作，所以会增加性能开销。主要考虑时间开销。

作者回复

文字常量那是默认行为，调用intern的不一定都是如此

DavidWhom佳伟

老师，就这样面试题多讲讲挺好的

作者回复

谢谢

wang\_bo

讲的很好，intern方法之前都没怎么接触过

作者回复

谢谢

2018-05-15

2018-05-15

2018-05-15

2018-05-16

2018-05-15

2018-05-16

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

2018-05-15

## 第6讲 | 动态代理是基于什么原理?

2018-05-17 杨晓峰



第6讲 | 动态代理是基于什么原理?  
杨晓峰  
00:00 / 11:24

编程语言通常有各种不同的分类角度，动态类型和静态类型就是其中一种分类角度，简单区分就是语言类型信息是在运行时检查，还是编译期检查。

与其近似的还有一个对比，就是所谓强类型和弱类型，就是不同类型变量赋值时，是否需要显式地（强制）进行类型转换。

那么，如何分类Java语言呢？通常认为，Java是静态的强类型语言，但是因为提供了类似反射等机制，也具备了部分动态类型语言的能力。

言归正传，今天我要问你的是，**谈谈Java反射机制，动态代理是基于什么原理？**

## 典型回答

反射机制是Java语言提供的一种基础功能，赋予程序在运行时自省（introspect，官方用语）的能力。通过反射我们可以直接操作类或者对象，比如获取某个对象的类定义，获取类声明的属性和方法，调用方法或者构造对象，甚至可以运行时修改类定义。

动态代理是一种方便运行时动态构建代理、动态处理方法调用的机制，很多场景都是利用类似机制做到的，比如用来包装RPC调用、面向切面的编程（AOP）。

实现动态代理的方式很多，比如JDK自身提供的动态代理，就是主要利用了上面提到的反射机制。还有其他的实现方式，比如利用传说中更高性能的字节码操作机制，类似ASM、cglib（基于ASM）、Javassist等。

## 考点分析

这个题目给我的第一印象是稍微有点诱导的嫌疑，可能会下意识地以为动态代理就是利用反射机制实现的，这么说也不算错但稍微有些不全面。功能才是目的，实现的方法有很多。总的来说，这道题目考察的是Java语言的另外一种基础机制：反射。它就像是一种魔法，引入运行时自省能力，赋予了Java语言令人意外的活力，通过运行时操作元数据或对象，Java可以灵活地操作运行时才能确定的信息。而动态代理，则是延伸出来的一种广泛应用于产品开发中的技术，很多繁琐的重复编程，都可以被动态代理机制优雅地解决。

从考察知识点的角度，这道题涉及的知识点比较庞杂，所以面试官能够扩展或者深挖的内容非常多，比如：

- 考察你对反射机制的了解和掌握程度。
- 动态代理解决了什么问题，在你业务系统中的应用场景是什么？
- JDK动态代理在设计和实现上与cglib等方式有什么不同，进而如何取舍？

这些考点似乎不是短短一篇文章能够囊括的，我会在知识扩展部分尽量梳理一下。

## 知识扩展

## 1. 反射机制及其演进

对于Java语言的反射机制本身，如果你去看一下java.lang或java.lang.reflect包下的相关抽象，就会有一个很直观的印象了。Class、Field、Method、Constructor等，这些完全就是我们操作类和对象的元数据对应。反射各种典型用例的编程，相信有太多文章或书籍进行过详细的介绍，我就不再赘述了，至少你需要掌握基本场景编程，这里是官方提供的参考文档：<https://docs.oracle.com/javase/tutorial/reflect/index.html>。

关于反射，有一点我需要特意提一下，就是反射提供的AccessibleObject.setAccessible(boolean flag)。它的子类也大都重写了这个方法，这里的所谓accessible可以理解成修饰成员的public、protected、private，这意味着我们可以在运行时修改成员访问限制！

setAccessible的应用场景非常普遍，遍布我们的日常开发、测试、依赖注入等各种框架中。比如，在O/R Mapping框架中，我们为一个Java实体对象，运行时自动生成setter、getter的逻辑，这是加载或者持久化数据非常必要的，框架通常可以利用反射做这个事情，而不需要开发者手动写类似的重复代码。

另一个典型场景就是绕过API访问控制。我们日常开发时可能被迫要调用内部API去做些事情，比如，自定义的高性能NIO框架需要显式地释放DirectBuffer，使用反射绕开限制是一种常见办法。

但是，在Java 9以后，这个方法的使用可能会存在一些争议，因为Jigsaw项目新增的模块化系统，出于强封装性的考虑，对反射访问进行了限制。Jigsaw引入了所谓Open的概念，只有当被反射操作的模块和指定的包对反射调用者模块Open，才能使用setAccessible；否则，被认为是不合法（Illegal）操作。如果我们的实体类是定义在模块里面，我们需要在模块描述符中明确声明：

```
module MyEntities {
    // Open for reflection
    opens com.mycorp to java.persistence;
}
```

因为反射机制使用广泛，根据社区讨论，目前，Java 9仍然保留了兼容Java 8的行为，但是很有可能在未来版本，完全启用前面提到的针对setAccessible的限制，即只有当被反射操作的模块和指定的包对反射调用者模块Open，才能使用setAccessible，我们可以使用下面参数显式设置。

```
--illegal-access={ permit | warn | deny }
```

## 2. 动态代理

前面的问题问到了动态代理，我们一起看看，它到底是解决什么问题？

首先，它是一个代理机制。如果熟悉设计模式中的代理模式，我们会知道，代理可以看作是对调用目标的一个包装，这样我们对目标代码的调用不是直接发生的，而是通过代理完成。其实很多动态代理场景，我认为也可以看作是装饰器（Decorator）模式的应用，我会在后面的专栏设计模式主题予以补充。

通过代理可以让调用者与实现者之间解耦。比如进行RPC调用，框架内部的寻址、序列化、反序列化等，对于调用者往往是没有太大意义的，通过代理，可以提供更加友善的界面。

代理的发展经历了静态到动态的过程，源于静态代理引入的额外工作。类似早期的RMI之类古董技术，还需要rmic之类工具生成静态Stub等各种文件，增加了很多繁琐的准备工作，而这又和我们的业务逻辑没有关系。利用动态代理机制，相应的stub等类，可以在运行时生成，对应的调用操作也是动态完成，极大地提高了我们的生产力。改进后的RMI已经不再需要手动去准备这些了，虽然它仍然是相对古老落后的技术，未来也许会逐步被移除。

这么说可能不够直观，我们可以看JDK动态代理的一个简单例子。下面只是加了一句print，在生产系统中，我们可以轻松扩展类似逻辑进行诊断、限流等。

```
public class MyDynamicProxy {
    public static void main (String[] args) {
        HelloImpl hello = new HelloImpl();
        MyInvocationHandler handler = new MyInvocationHandler(hello);
        // 构造代码略
        Hello proxyHello = (Hello) Proxy.newProxyInstance(HelloImpl.class.getClassLoader(), HelloImpl.class.getInterfaces(), handler);
        // 调用代理方法
        proxyHello.sayHello();
    }
}

interface Hello {
    void sayHello();
}

class HelloImpl implements Hello {
    @Override
    public void sayHello() {
        System.out.println("Hello World");
    }
}

class MyInvocationHandler implements InvocationHandler {
    private Object target;
    public MyInvocationHandler(Object target) {
        this.target = target;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        System.out.println("Invoking sayHello");
        Object result = method.invoke(target, args);
        return result;
    }
}
```

上面的JDK Proxy例子，非常简单地实现了动态代理的构建和代理操作。首先，实现对应的InvocationHandler；然后，以接口Hello为纽带，为被调用目标构建代理对象，进而应用程序就可以使用代理对象间接运行调用目标的逻辑，代理为应用插入额外逻辑（这里是println）提供了便利的入口。

从API设计和实现的角度，这种实现仍然有局限性，因为它是以接口为中心的，相当于添加了一种对于被调用者没有太多意义的限制。我们实例化的是Proxy对象，而不是真正的被调用类型，这在实践中还是可能带来各种不便和能力退化。

如果被调用者没有实现接口，而我们还是希望利用动态代理机制，那么可以考虑其他方式。我们知道Spring AOP支持两种模式的动态代理，JDK Proxy或者cglib，如果我们选择cglib方式，你会发现对接口的依赖被克服了。

cglib动态代理采取的是创建目标类的子类的方式，因为是子类化，我们可以达到近似使用被调用者本身的效果。在Spring编程中，框架通常会处理这种情况，当然我们也可以[显式指定](#)。关于类似方案的实现细节，我就不再详细讨论了。

那我们在开发中怎样选择呢？我来简单对比下两种方式各自优势。

JDK Proxy的优势：

- 最小化依赖关系，减少依赖意味着简化开发和维护，JDK本身的支持，可能比cglib更加可靠。

- 平滑进行JDK版本升级，而字节码类库通常需要进行更新以保证在新版Java上能够使用。

- 代码实现简单。

基于类似cglib框架的优势：

- 有的时候调用目标可能不便实现额外接口，从某种角度看，限定调用者实现接口是有些侵入性的实践，类似cglib动态代理就没有这种限制。

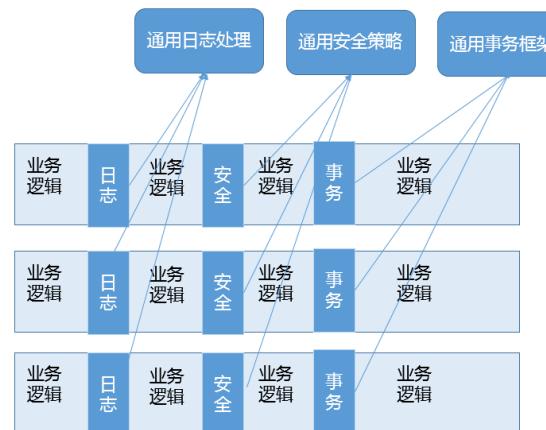
- 只操作我们关心的类，而不必为其他相关类增加工作量。

- 高性能。

另外，从性能角度，我想补充几句。记得有人曾经得出结论说JDK Proxy比cglib或者Javassist慢几十倍。坦白说，不去争论具体的benchmark细节，在主流JDK版本中，JDK Proxy在典型场景可以提供对等的性能水平，数量级的差距基本上不是广泛存在的。而且，反射机制性能在现代JDK中，自身已经得到了极大的改进和优化，同时，JDK很多功能也不完全是反射，同样使用了ASM进行字节码操作。

我们在选型中，性能未必是唯一考量，可靠性、可维护性、编程工作量等往往是更重要的考虑因素，毕竟标准类库和反射编程的门槛要低得多，代码量也是更加可控的，如果我们比较下不同开源项目在动态代理开发上的投入，也能看到这一点。

动态代理应用非常广泛，虽然最初多是因为RPC等使用进入我们的视线，但是动态代理的使用场景远远不仅如此，它完美符合Spring AOP等切面编程。我在后面的专栏还会进一步详细分析AOP的目的和能力，简单来说它可看作是对OOP的一个补充，因为OOP对于跨越不同对象或类的分散、纠缠逻辑表现力不够，比如在不同模块的特定阶段做一些事情，类似日志、用户鉴权、全局性异常处理、性能监控，甚至事务处理等，你可以参考下面这张图。



AOP通过（动态）代理机制可以让开发者从这些繁琐事项中抽身出来，大幅度提高了代码的抽象程度和复用度。从逻辑上来说，我们在软件设计和实现中的类似代理，如Facade、Observer等很多设计目的，都可以通过动态代理优雅地实现。

今天我简要回顾了反射机制，谈了反射在Java语言演进中正在发生的变化，并且进一步探讨了动态代理机制和相关的切面编程，分析了其解决的问题，并探讨了生产实践中的选择考量。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？留一道思考题给你，你在工作中哪些场景使用到了动态代理？相应选择了什么实现技术？选择的依据是什么？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



肖一林

2018-05-17

提一些建议：应该从两条线讲这个问题，一条从代理模式，一条从反射机制。不要老担心篇幅限制讲不清问题，废话砍掉一些，深层次的内在原理多讲些（比如asm），容易自学的扩展知识可以用链接代替。

代理模式：通过代理静默地解决一些业务无关的问题，比如远程、安全、事务、日志、资源关闭……让应用开发者可以只关心他的业务。

静态代理：事先写好代理类，可以手工编写，也可以用工具生成。缺点是每个业务类都要对应一个代理类，非常不灵活。

动态代理：运行时自动生成代理对象。缺点是生成代理对象和调用代理方法都要额外花费时间。

JDK 动态代理：基于 Java 反射机制实现。必须要实现了接口的业务类才能用这种方法生成代理对象。新版本也开始结合ASM机制。

CGLIB 动态代理：基于ASM机制实现。通过生成业务类的子类作为代理类。

Java 发射机制的常见应用：动态代理（AOP, RPC）、提供第三方开发者扩展能力（Servlet容器, JDBC连接）、第三方组件创建对象（DI）……

我水平比较菜，希望多学点东西，希望比免费知识层次更深些，也不光是为了面试，所以提提建议。

作者回复

2018-05-18  
谢谢反馈，类似ASM这种字节码操纵是有单独章节覆盖的，前面基础篇有个整体印象，免得陷入细节；Java内部动态生成还有其他领域，比如Lambda实现机制，个人认为一起分析会连贯一些。

公众号-Java大前端

2018-05-17

反射与动态代理原理

1 关于反射  
反射最大的作用之一就在于我们可以不在编译时知道某个对象的类型，而在运行时通过提供完整的“包名+类名.class”得到。注意：不是在编译时，而是在运行时。

功能：  
• 在运行时能判断任意一个对象所属的类。  
• 在运行时能构造任意一个类的对象。  
• 在运行时判断任意一个类所具有的成员变量和方法。  
• 在运行时调用任意一个对象的方法。

说白了就是，利用Java反射机制我们可以加载一个运行时才得知名称的class，获悉其构造方法，并生成其对象实体，能对其fields设值并唤起其methods。

应用场景：  
反射技术常用于各类通用框架开发中。因为为了保证框架的通用性，需要根据配置文件加载不同的对象或类，并调用不同的方法，这个时候就会用到反射——运行时动态加载需要加载的对象。

特点：  
由于反射会额外消耗一定的系统资源，因此如果不需要动态地创建一个对象，那么就不需要用反射。另外，反射调用方法时可以忽略权限检查，因此可能会破坏封装性而导致安全问题。

2 动态代理  
为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在两者之间起到中介的作用（可类比房屋中介，房东委托中介销售房屋，签订合同等）。  
所谓动态代理，就是实现阶段不用关心代理谁，而是在运行阶段才指定代理哪个对象（不确定性）。如果是自己写代理类的方式就是静态代理（确定性）。

组成部分：  
(动态)代理模式主要涉及三个要素：  
其一：抽象接口类  
其二：被代理类（具体实现抽象接口的类）  
其三：动态代理类：实际调用被代理类的方法和属性的类

实现方式：  
实现动态代理的方式很多，比如 JDK 自身提供的动态代理，就是主要利用了反射机制。还有其他的实现方式，比如利用字节码操作机制，类似 ASM, CGLIB（基于 ASM）, Javassist 等。  
举例，常可采用的JDK提供的动态代理接口InvocationHandler来实现动态代理类。其中invoke方法是该接口必须实现的，它完成对真实方法的调用。通过InvocationHandler接口，所有方法都会由该Handler来进行处理，即所有被代理的方法都由InvocationHandler接管实际的处理任务。此外，我们常可以在invoke方法实现中增加自定义的逻辑实现，实现对被代理类的业务逻辑无侵入。

郭峰

2018-05-17

没涉及到原理，proxy到底是如何实现的，运行时拦截？cglib是编译时对类进行织入？要是更细一些就好了

刘方杰

2018-05-18

哎，阅读越来越困难了，我离核心是不是太远了。

jimforcode

2018-05-17

好像和啥原理没啥关系吧，总结来说就是JDK 自身的反射机制或用第三方库，哪哪看到的都这样说，一笔带过  
作者回复

谢谢反馈，字节码操作、运行时拦截、加载期编织、Java agent等，会和Aop单独介绍，那些内容不是几句话说得完

jimforcode

2018-05-17

比较期待大神讲讲动态代理的原理是什么，对性能会带来什么影响，有没有什么问题需要规避，谢谢

曹铮

2018-05-17

先回答问题：

99%的Java程序员应该都间接使用了AOP，自己项目里直接编写的，比如调用追踪，通用日志，自动重试。反射和AOP真是双剑合璧拔群的技术。从MVC开始约定胜过配置的开发理念大行其道，ORM自动映射，plugin模式，到现在的spring + boot + cloud 的declarative编程大量基于此实现，可以说没有反射和AOP就没有Java的今天。反面就是，自己想进行定制化的改造封装真挺苦逼。

再提个问题：

1. 听到过个说法，反射慢因为一是获取field, Method等要进行元数据的查找，这里有字符串匹配操作。二是Invoke时，要进行一些安全性的检查。这种说法对么？JVM在解释执行的时候就不做一些操作内存操作的检查了么？还有没有其他？  
2. 以前写C#的，里面可以拼表达式树，运行时生成一个函数（不需要有对象），理论上性能是和手写代码一样的，但可以缓存起来。这解决的是手写中间代码太难的问题。请问Java有这种类似的功能嘛？

作者回复

2018-05-18

1. 基本如此；反射是相对保证类型安全的，我觉得要比也是和methodhandle之类对比，那个更是接近jvm内部的黑盒，性能更好。

2. 你是说lambda？也是需要vm生成call site，然后invokedynamic之类调用，所以首次调用开销明显，C#不了解，不过动态生成的感觉都是如此吧；这东西目前没有cache，如果说的是存储在文件系统，未来，嘿嘿.....

这些太零碎，说过了会有单独章节介绍，不然没基础的就晕了，还用不上

ai-wen

2018-05-21

对于我这初学者，读着就懵了

TWO STRINGS

2018-05-17

老师可以在分享结束时推荐一些好的文章，书籍甚至演讲之类的么？

作者回复

2018-05-18

没问题，喜欢底层，去查查JavaOne, FOSDEM, jvmsummit等

灰飞灰猪不会灰飞.烟灭

2018-05-17

cglb是怎么实现对目标对象的拦截的呢？

作者回复

2018-05-18

计划单独介绍

言志

2018-05-17

采用cglb方式的动态代理还有个缺点：不能应用到被代理对象的final方法上。

我在多数据源项目中自动切换数据源功能用到了

云学

2018-06-09

看了好多篇文章，总体感觉是比较累，无论读者是否具有java背景，都应该让他看懂，而不是越看越糊涂，疑问反而更多了

作者回复

2018-06-10

非常感谢，读者基础不同，我尽量兼顾并增加基础的介绍，因为也有反馈希望可以更全面、深入...

有好的建议请不吝赐教

bigfish

2018-07-08

本来资质愚笨，看不懂很多东西的原理，想进来学习一下，jdk动态代理的原理和cglb代理原理等一些原理性的东西（其他章节也是如此），发现听到原理性的东西不多都是一带而过，其实你做的课件对很多点都是一带而过，听到某个名词一下来了兴趣继续一听结束了，我们都深知Java很大很多可以研究的，其实我们想听的很多是一些点的原理，讲完原理在结合实际应用阐述一下，也许我们就会有些豁然开朗的感觉，希望能理解一下！！！

作者回复

2018-07-11

谢谢反馈，后面类加载章节介绍了两者底层机制，照顾下不同基础的读者

蒙奇D路飞

2018-05-28

感觉细节层面缺少具体描述，希望后续对底层原理的描述更细致些~

作者回复

2018-05-28

谢谢反馈，有章节介绍类似字节码操纵之类底层技术，照顾不同基础

雷贤

2018-05-22

何为反射？

反射是指计算机程序在运行时可以访问、检测和修改它本身状态或行为的一种能力。比喻来说，反射就是程序在运行的时候能够“观察”并且修改自己的行为。文章提到了“自省”，它与反射是有区别的。“自省”机制仅指程序在运行时对自身信息（元数据）的检测，而反射机制不仅包括要在运行时对程序自身信息进行检测，还要求程序能进一步根据这些信息改变程序状态或结构。

通过反射的运行时可以访问、检测和修改自身状态的特性，也就出现了动态代理。

一个明白对反射机制的了解（来源于业务能力）。

在一个程序中有两个类A, B，整个程序的运行都由B类来承担，A类的所要做的工作是满足B的要求，如果在没有反射这种机制的条件下实现这个程序就会，既要把B类中的对整个程序运行逻辑进行编写，还要回到A类中对B的工作进行人工手写辅助，这时只要B的任何改动都会影响到A，若有了反射或代理之后 A的代码就可以自己去访问、检测 B，从而自动的修改 A自身的状态来辅助 B。

作者在评论中提到实现后续单独说明，我想到时就可以知道，代码是如何实现程序在运行时的访问、检测和行为的修改了。

1024

2018-05-17

为什么JDK Proxy是基于反射实现的呢？这其中有什么考量呢？为什么不基于其他呢？

作者回复

2018-05-18

我的理解，核心功能需要最小依赖关系，性能也不错

Allen

2018-06-27

文章可以推荐一些关于Java的一些前沿技术的网站，想提高一下技术敏感性，请不吝赐教！

沈琦斌

2018-06-26  
老师，我有个疑问。一些ORM框架比如mybatis，我们在使用过程中已经指定如何进行映射的时候，为什么不直接new一个类对象，然后调用setter方法赋值，而要用动态代理呢？这样设计的好处是什么？水平比较菜，还请谅解，谢谢

作者回复

2018-06-28  
我理解是，不需要依赖目标类的实现，框架使用限制少些，例如，不再要求实现了setter

沈琦斌

2018-06-26  
老师，我有个疑问。一些ORM框架比如mybatis，我们在使用过程中已经指定如何进行映射的时候，为什么不直接new一个类对象，然后调用setter方法赋值，而要用动态代理呢？这样设计的好处是什么？水平比较菜，还请谅解，谢谢

作者回复

已回答

chris

2018-06-13  
之前项目中用到javaassist，在通用业务流程中埋了一个扩展点，通过xml配置具体实现类来做定制业务处理，大概是类加载时会把对应定制代码生成字节码织入进去，老师可以讲讲javaassist原理吗？

作者回复

2018-06-13  
我后面会介绍字节码操纵相关，还有类似Jdk里怎么生成lambda的代码等，不过Jdk是用的asm

Jeffrey

2018-06-12  
感觉还是书看少了，如果看过书，有过业务经验，再来看作者的分享，兴许会有另一种收获。

Geek\_c1b553

2018-05-30  
mysql主从库通过aop动态切换

George

2018-05-25  
一个类的属性copy到另外一个类，使用cglib，先完成属性的拷贝

岁月如歌

2018-05-22  
代理的作用，使调用者和实现者之间解耦

岁月如歌

2018-05-22  
反射的作用：发射使得我们可以在运行的时候再去加载类，创建对象，并能访问到类中的私有属性和方法

岁月如歌

2018-05-22  
JAVA是静态的强类型，也就是说JAVA的类型在编译时确定（静态语言），不同类型之前的变量转换需要强制转换（强类型）

岁月如歌

2018-05-22  
不同类型变量赋值时，是否需要强制转换来区分语言是强类型还是弱类型

岁月如歌

2018-05-22  
语言类型信息在什么时候检查来区分语言是动态类型还是静态类型

iiiDragon

2018-05-21  
老师，多些代码示例。看的时候懂，看完又啥不懂

Aife

2018-05-20  
听不懂啊，怎么办……该从哪里补课

作者回复

2018-05-21  
具体哪些方面？反射还是动态代理？动态代理可以看看AOP的书籍或文章

2018-05-20  
字节码操纵，不用太担心，我这章没有写太多，就是担心基础不一样，搞糊涂了，毕竟真正开发中做这个是小众的，大多数不会直接用

龙猫9527

2018-05-18  
我们使用spring中的aspect来实现权限控制，通过切点表达式和注解以更加细粒度的方式控制。

作者回复

2018-05-18  
AspectJ 我理解是确实能力更全面一些；比如，它可以compile-time, binary, 还有用Javaagent作load-timeweaving; cglib 更是runtime weaving

王磊

2018-05-18  
反射加载的类和常规通过new对象加载的类，在资源消耗上有什么不同？我理解他们的效果是相同的，只是加载方式不同：加载到内存的区域也是相同的，metospace。理解对吗？

作者回复

2018-05-19  
我理解没区别

王磊

2018-05-18  
关于如下的原文，有些疑惑。两种方式的实现是这么泾渭分明吗。例如Jdk代理根据接口生成实现类，不也操作字节码了吗，为什么只强调用cglib；反过来，cglib既然能通过指定类生成子类，加载父类如果不是反射，那用的是什么？希望老师把这里澄清一下。

\*实现动态代理的方式很多，比如 JDK 自身提供的动态代理，就是主要利用了上面提到的反射机制。还有其他的实现方式，比如利用传说中更高性能的字节码操作机制，类似 ASM、cglib（基于

ASM)、Javassist 等。'

作者回复

纠结于细节无助于说明问题，个人认为最好看主要路径实现机制

jutsu

老师的讲解和大神的评论学习很多

2018-05-19

2018-05-17

## 第7讲 | int和Integer有什么区别?

2018-05-19 杨晓峰



第7讲 | int和Integer有什么区别?  
杨晓峰

- 00:00 / 11:04

Java虽然号称是面向对象的语言，但是原始数据类型仍然是重要的组成元素，所以在面试中，经常考察原始数据类型和包装类等Java语言特性。

今天我要问你的是，**int和Integer有什么区别？谈谈Integer的值缓存范围。**

## 典型回答

int是我们常说的整形数字，是Java的8个原始数据类型（Primitive Types，boolean、byte、short、char、int、float、double、long）之一。Java语言虽然号称一切都是对象，但原始数据类型是例外。

Integer是int对应的包装类，它有一个int类型的字段存储数据，并且提供了基本操作，比如数学运算、int和字符串之间转换等。在Java 5中，引入了自动装箱和自动拆箱功能（boxing/unboxing），Java可以根据上下文，自动进行转换，极大地简化了相关编程。

关于Integer的值缓存，这涉及Java 5中另一个改进。构建Integer对象的传统方式是直接调用构造器，直接new一个对象。但是根据实践，我们发现大部分数据操作都是集中在有限的、较小的数值范围，因而，在Java 5中新增了静态工厂方法valueOf，在调用它的时候会利用一个缓存机制，带来了明显的性能改进。按照Javadoc，这个值默认缓存是-128到127之间。

## 考点分析

今天这个问题涵盖了Java里的两个基础要素：原始数据类型、包装类。谈到这里，就可以非常自然地扩展到自动装箱、自动拆箱机制，进而考察封装类的一些设计和实践。坦白说，理解基本原理和用法已经足够日常工作需求了，但是要落实到具体场景，还是有很多问题需要仔细思考才能确定。

面试官可以结合其他方面，来考察面试者的掌握程度和思考逻辑，比如：

- 我在专栏第1讲中介绍的Java使用的不同阶段：编译阶段、运行时，自动装箱/自动拆箱是发生在什么阶段？
- 我在前面提到使用静态工厂方法valueOf会使用到缓存机制，那么自动装箱的时候，缓存机制起作用吗？
- 为什么我们需要原始数据类型，Java的对象似乎也很高效，应用中具体会产生哪些差异？
- 阅读过Integer源码吗？分析下类或某些方法的设计要点。

似乎有太多内容可以探讨，我们一起来分析一下。

## 知识扩展

## 1.理解自动装箱、拆箱

自动装箱实际上算是一种语法糖。什么是语法糖？可以简单理解为Java平台为我们自动进行了一些转换，保证不同的写法在运行时等价，它们发生在编译阶段，也就是生成的字节码是一致的。

像前面提到的整数，javac替我们自动把装箱转换为Integer.valueOf()，把拆箱替换为Integer.intValue()，这似乎这也顺道回答了另一个问题，既然调用的是Integer.valueOf，自然能够得到缓存的好处啊。

如何程序化的验证上面的结论呢？

你可以写一段简单的程序包含下面两句代码，然后反编译一下。当然，这是一种从表现倒推的方法，大多数情况下，我们还是直接参考规范文档会更加可靠，毕竟软件承诺的是遵循规范，而不是保持当前行为。

```
Integer integer = 1;
int unboxing = integer ++;
```

反编译输出:

```
1: invokevirtual #2           // Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
8: invokevirtual #3           // Method
java/lang/Integer.intValue:()I
```

这种缓存机制并不是只有Integer才有，同样存在于其他的一些包装类，比如：

- Boolean，缓存了true/false对应实例，确切说，只会返回两个常量实例Boolean.TRUE/FALSE。
- Short，同样是缓存了-128到127之间的数值。
- Byte，数值有限，所以全部都被缓存。
- Character，缓存范围'\u0000' 到 '\u007F'。

自动装箱/自动拆箱似乎很酷，在编程实践中，有什么需要注意的吗？

原则上，建议避免无意中的装箱、拆箱行为，尤其是在性能敏感的场合，创建10万个Java对象和10万个整数的开销可不是一个数量级的，不管是内存使用还是处理速度，光是对象头的空间占用就已经是数量级的差距了。

我们其实可以把这个观点扩展开，使用原始数据类型、数组甚至本地代码实现等，在性能极度敏感的场景往往具有比较大的优势，用其替换掉包装类、动态数组（如ArrayList）等可以作为性能优化的备选项。一些追求极致性能的产品或者类库，会极力避免创建过多对象。当然，在大多数产品代码里，并没有必要这么做，还是以开发效率优先。以我们经常会使用到的计数器实现为例，下面是一个常见的线程安全计数器实现。

```
class Counter {
    private final AtomicLong counter = new AtomicLong();
    public void increase() {
        counter.incrementAndGet();
    }
}
```

如果利用原始数据类型，可以将其修改为

```
class CompactCounter {
    private volatile long counter;
    private static final AtomicLongFieldUpdater<CompactCounter> updater = AtomicLongFieldUpdater.newUpdater(CompactCounter.class, "counter");
    public void increase() {
        updater.incrementAndGet(this);
    }
}
```

## 2. 源码分析

考察是否阅读过、是否理解JDK源代码可能是部分面试官的关注点，这并不完全是一种苛刻要求，阅读并实践高质量代码也是程序员成长的必经之路，下面我来分析下Integer的源码。

整体看一下Integer的职责，它主要包括各种基础的常量，比如最大值、最小值、位数等；前面提到的各种静态工厂方法valueOf()；获取环境变量数值的方法；各种转换方法，比如转换为不同进制的字符串，如8进制，或者反过来的解析方法等。我们进一步来看一些有意思的地方。

首先，继续深挖缓存，Integer的缓存范围虽然默认是-128到127，但是在特别的应用场景，比如我们明确知道应用会频繁使用更大的数值，这时候应该怎么办呢？

缓存上限值实际是可以根据需要调整的，JVM提供了参数设置：

```
-XX:AutoBoxCacheMax=N
```

这些实现，都体现在[java.lang.Integer](#)源码之中，并实现在IntegerCache的静态初始化块里。

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];
    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        ...
        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }
    ...
}
```

第二，我们在分析字符串的设计实现时，提到过字符串是不可变的，保证了基本的信息安全和并发编程中的线程安全。如果你去看包装类里存储数值的成员变量“value”，你会发现，不管是Integer还是Boolean等，都被声明为“private final”，所以，它们同样是不可变类型！

这种设计是可以理解的，或者说是必须的选择。想象一下这个应用场景，比如Integer提供了getInteger()方法，用于方便地读取系统属性，我们可以用属性来设置服务器某个服务的端口，如果我可以轻易地把获取到的Integer对象改变为其他数值，这会带来产品可靠性方面的严重问题。

第三，Integer等包装类，定义了类似SIZE或者BYTES这样的常量。这反映了什么样的设计考虑呢？如果你使用过其他语言，比如C、C++，类似整数的位数，其实是不确定的，可能在不同的平台，比如32位或者64位平台，存在非常大的不同。那么，在32位JDK或者64位JDK里，数据位数会有不同吗？或者说，这个问题可以扩展为：我使用32位JDK开发编译的程序，运行在64位JDK上，需要做什么特别的移植工作吗？

其实，这种移植对于Java来说相对要简单些，因为原始数据类型是不存在差异的，这些明确定义在[Java语言规范](#)里面，不管是32位还是64位环境，开发者无需担心数据的位数差异。

对于应用移植，虽然存在一些底层实现的差异，比如64位HotSpot JVM里的对象要比32位HotSpot JVM大（具体区别取决于不同JVM实现的选择），但是总体来说，并没有行为差异，应用移植还是可以做到宣称的“一次书写，到处执行”，应用开发者更多需要考虑的是容量、能力等方面差异。

### 3. 原始类型线程安全

前面提到了线程安全设计，你有没有想过，原始数据类型操作是不是线程安全的呢？

这里可能存在不同层面的问题：

- 原始数据类型的变量，显然要使用并发相关手段，才能保证线程安全，这些我会在专栏后面的并发主题详细介绍。如果有线程安全的计算需要，建议考虑使用类似AtomicInteger、AtomicLong这样的线程安全类。
- 特别的是，部分比较宽的数据类型，比如float、double，甚至不能保证更新操作的原子性，可能出现程序读取到只更新了一半数据位的数值！

### 4. Java原始数据类型和引用类型局限性

前面我谈了非常多的技术细节，最后再从Java平台发展的角度来看，原始数据类型、对象的局限性和演进。

对于Java应用开发者，设计复杂而灵活的类型系统似乎已经习以为常了。但是坦白说，毕竟这种类型系统的设计是源于很多年前的技术决定，现在已经逐渐暴露出了一些副作用，例如：

- 原始数据类型和Java泛型并不能配合使用

这是因为Java的泛型某种程度上可以算作伪泛型，它完全是一种编译期的技巧，Java编译期会自动将类型转换为对应的特定类型，这就决定了使用泛型，必须保证相应类型可以转换为Object。

- 无法高效地表达数据，也不便与表达复杂的数据结构，比如vector和tuple

我们知道Java的对象都是引用类型，如果是一个原始数据类型数组，它在内存里是一段连续的内存，而对象数组则不然，数据存储的是引用，对象往往是分散地存储在堆的不同位置。这种设计虽然带来了极大灵活性，但是也导致了数据操作的低效，尤其是无法充分利用现代CPU缓存机制。

Java为对象内建了各种多态、线程安全等方面的支持，但这不是所有场合的需求，尤其是数据处理重要性日益提高，更加高密度的值类型是非常现实的需求。

针对这些方面的增强，目前正在OpenJDK领域紧锣密鼓地进行开发，有兴趣的话你可以关注相关工程：<http://openjdk.java.net/projects/valhalla/>。

今天，我梳理了原始数据类型及其包装类，从源码级别分析了缓存机制等设计和实现细节，并且针对构建极致性能的场景，分析了一些可以借鉴的实践。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？留一道思考题给你，前面提到了从空间角度，Java对象要比原始数据类型开销大的多。你知道对象的内存结构是什么样的吗？比如，对象头的结构。如何计算或者获取某个Java对象的大小？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



cookie\_

对象由三部分组成，对象头，对象实例，对齐填充。

其中对象头一般是十六个字节，包括两部分，第一部分有哈希码，锁状态标志，线程持有的锁，偏向线程id，gc分代年龄等。第二部分是类型指针，也就是对象指向它的美元数据指针，可以理解，对象指向它的类。

对象实例就是对象存储的真正有效信息，也是程序中定义各种类型的字段包括父类继承的和子类定义的，这部分的存储顺序会被虚拟机和代码中定义的顺序影响（这里问一下，这个被虚拟机影响是不是就是重排序？如果是的话，我知道的volatile定义的变量不会被重排序应该就是这里不会受虚拟机影响吧？）。

第三部分对齐填充只是一个类似占位符的作用，因为内存的使用都会被填充为八字节的倍数。

2018-05-19

还是个初学者。以上是我了解，不知道有没有错，希望老师能告知。

Kyle

2018-05-19

节选自《深入理解JAVA虚拟机》：

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3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，第一部份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，这部分数据的长度在32位和64位的虚拟机（未开启压缩指针）中分别为32bit和64bit，官方称它为“Mark Word”。

对象头的另外一部分是类型指针，即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。这点将在2.3.3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却无法确定数组的大小。

接下来的实例数据部分是对象真正存储的有效信息，也是在程序代码中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，都需要记录起来。

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。

公号Java大后端

2018-05-19

1 int和Integer

JDK1.5引入了自动装箱与自动拆箱功能，Java可根据上下文，实现int/Integer,double/Double,boolean/Boolean等基本类型与相应用对象之间的自动转换，为开发过程带来极大便利。

最常用的是通过new方法构建Integer对象。但是，基于大部分数据操作都是集中在有限的、较小的数值范围，在JDK1.5中新增了静态工厂方法 valueOf，其背后实现是将int值从-128到127之间的Integer对象进行缓存，在调用时候直接从缓存中获取，进而提升构建对象的性能。也就是说使用该方法后，如果两个对象的Int值相同且落在缓存值范围内，那么这两个对象就是同一个对象。当值较小且频繁使用时，推荐优先使用整型池方法（时间与空间性能俱佳）。

2 注意事项

[1] 基本类型均具有取值范围，在大数\*大数的时候，有可能会出现越界的情况。

[2] 基本类型转换时，使用声明的方式，例：long result= 1234567890 \* 24 \* 365；结果值一定不会是你所期望的那个值，因为1234567890 \* 24已经超过了int的范围，如果修改为 long result= 1234567890L \* 24 \* 365，就正常了。

[3] 避免基本类型处理货币存储。如果采用double常会带来差距，常采用BigDecimal，整型（如果要精确表示分，可将值扩大100倍转化为整型）解决该问题。

[4] 优先使用基本类型。原则上，建议避免无意中的裝箱、拆箱行为，尤其是在性能敏感的场合。

[5] 如果有线程安全的计算需要，建议考虑使用类型AtomicInteger、AtomicLong 这样的线程安全类。部分比较宽的基本数据类型，比如 float、double，甚至不能保证更新操作的原子性，可能出现程序读取到只更新了一半数据位的数值。

kursk.ye

2018-06-13

这篇文章写得比较零散，整体思路没有串起来，其实我觉得可以从这么一条线索理解这个问题。原始数据类型和 Java 泛型并不能配合使用，也就是Primitive Types 和Generic 不能混用，于是在JAVA就设计了这个auto-boxing/unboxing机制，实际上就是primitive value 与 object之间的隐式转换机制，否则是没有这个机制，开发者就必须每次手动显示转换，那多麻烦是不是？但是primitive value 与 object各自有自己的优势，primitive value 在内存中存的是值，所以找到primitive value 的内存位置，就可以获得值，不像object存的是reference，找到object的内存位置，还要根据reference找下一个内存空间，会产生更多的IO，所以性能比primitive value 差，但是object具备generic的能力，更抽象，解决业务问题编程效率高。于是JAVA设计者的初衷估计是这样的：如果开发者要做计算，就应该使用primitive value如果开发者要处理业务问题，就应该使用object，采用Generic机制，反正JAVA有auto-boxing/unboxing机制，对开发者来讲也不需要注意什么。然后为了弥补object计算能力的不足，还设计了static valueOf()方法提供缓存机制，算是一个弥补。

行者

2018-05-20

1. Mark Word: 标记位 4字节，类似轻量级锁标记位，偏向锁标记位等。

2. Class对象指针: 4字节，指向对象对应class对象的内存地址。

3. 对象实际数据: 对象所有成员变量。

4. 对齐: 对齐填充字节，按照8个字节填充。

Integer占用内存大小， $4 + 4 + 4 + 4 = 16$ 字节。

作者回复

2018-05-22

不错，如果是64位不用压缩指针，对象头会变大，还可能有对齐开销

麦田

2018-05-19

周末了是不是没人看文章了

George

2018-05-25

计算对象大小可通过dump内存之后用memory analyze分析

作者回复

2018-05-25

嗯，也可以利用：

jol, jmap, 或者instrument api (Java agent) 等等

George

2018-05-25

java内存结构

对头：

markword：用于存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁等。这部分数据长度在32位机器和64位机器虚拟机中分别为4字节和8字节；

klass指针：即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属于哪个类的实例；

length：如果是Java数组，对象头必须有一块用于记录数组长度的数据，用4个字节来int来记录数组长度；

实例数据

实例数据是对象真正存储的有效信息，也是程序代码中定义的各种类型的字段内容。无论是从父类继承下来还是在子类中定义的数据，都需要记录下来

堆积填充

对于hotspot最新的自动内存管理系统要求对象的起始地址必须为8字节的整数倍，这就要求当部位8字节的整数倍时，就需要填充数据对其进行填充。原因是访问未对齐的内存，处理器需要做两次内存访问，而对齐的内存访问仅需一次访问

Miaoze

2018-05-21

杨老师，问个问题，如果使用原始类型int定义一个变量在-128和127之间，如int c = 64;会放入Integer 常量缓存吗(IntegerCache)? 编译器是怎么操作的?

作者回复

2018-05-23

不需要，不是对象

两只◆◆

2018-05-19

原始数据类型貌似反射也不行。

Gerald

2018-05-29

为什么我感觉都这么难啊◆◆

作者回复

感谢反馈，具体哪个方面，我可以调整一下，尽量照顾不同基础的朋友

2018-05-29

ZC叶◆◆

想问下 自动装箱和自动拆箱是指类型转换吗？

作者回复

这个...似乎也算。如果你的“转换”是conversion，不是casting

jutsu

2018-05-22

老师的讲解让我想起了科比主导的 细节栏目

2018-05-23

步·壳

2018-05-20

缓存用得很巧妙，值得借鉴

2018-05-19

hansc

2018-05-19

垃圾回收分带年龄，`hashCode`值，锁标记，请问对象逃过垃圾回收的次数记录到哪里呢？

feifei

2018-07-03

JAVA的内存结构分为3部分：

1. 对象头 有两部分`markWord`和`Class`对象指针。markword包括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，

2. 实例数据

3. 对齐填充

获取一个JAVA对象的大小，可以将一个对象进行序列化为二进制的Byte，便可以查看大小，

`Integer value = 10;`

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(value);
// // 读出当前对象的二进制流信息
System.out.println(bos.size());
```

遗忘明天

2018-06-22

long的赋值也不是原子操作吗？

遗忘明天

2018-06-22

long的赋值也不是原子操作吗？

Darren

2018-06-16

老师，原始数据类型的包装类是对象吗？

作者回复

2018-06-16

类是类，实例化后才是对象

Darren

2018-06-16

老师，反编译输出怎么理解的，看不懂语法

作者回复

2018-06-17

具体哪一段，是文章中片段的`invokestatic`之类吗？如果是的话，最准确的可以参考java虚拟机规范，查询相应之类；大多数情况下可以搜索相关分析文章，理解难道会小些

tracer

2018-06-17

integer获取环境变量数值的方法，这个具体是指哪个方法？

作者回复

2018-06-12

`getInteger`, 建议看看文档

不瘦十斤不换名字

2018-06-12

为啥大家都在讨论对象的组成部分◆◆

梁作斌

2018-05-24

不是原子操作的基本类型是 `float`、`double`? 为啥不是 `long`、`double`?

作者回复

2018-05-25

那是举例，不是定义

Slug

2018-05-25

感谢老师放假还在写文章，学到很多，钱花的很值。

Hua

2018-05-19

希望老师多写一些文章这样我就不用看源码了。





第8讲 | 对比Vector、ArrayList、LinkedList有何区别?  
杨晓峰  
0:00 / 12:45

我们在日常的工作中，能够高效地管理和操作数据是非常重要的。由于每个编程语言支持的数据结构不尽相同，比如我最早学习的C语言，需要自己实现很多基础数据结构，管理和操作会比较麻烦。相比之下，Java则要方便得多，针对通用场景的需求，Java提供了强大的集合框架，大大提高了开发者的生产力。

今天我要问的是有关集合框架方面的问题，[对比Vector、ArrayList、LinkedList有何区别？](#)

#### 典型回答

这三者都是实现集合框架中的List，也就是所谓的有序集合，因此具体功能也比较近似，比如都提供按照位置进行定位、添加或者删除的操作，都提供迭代器以遍历其内容等。但因为具体的设计区别，在行为、性能、线程安全等方面，表现又有很大不同。

Vector是Java早期提供的线程安全的动态数组，如果不需要线程安全，并不建议选择，毕竟同步是有额外开销的。Vector内部是使用对象数组来保存数据，可以根据需要自动的增加容量，当数组已满时，会创建新的数组，并拷贝原有数组数据。

ArrayList是应用更加广泛的动态数组实现，它本身不是线程安全的，所以性能要好很多。与Vector近似，ArrayList也是可以根据需要调整容量，不过两者的调整逻辑有所区别，Vector在扩容时会提高1倍，而ArrayList则是增加50%。

LinkedList顾名思义是Java提供的双向链表，所以它不需要像上面两种那样调整容量，它也不是线程安全的。

#### 考点分析

似乎从我接触Java开始，这个问题就一直是经典的面试题，前面我的回答覆盖了三者的一些基本的设计和实现。

一般来说，也可以补充一下不同容器类型适合的场景：

- Vector和ArrayList作为动态数组，其内部元素以数组形式顺序存储的，所以非常适合随机访问的场合。除了尾部插入和删除元素，往往性能会相对较差，比如我们在中间位置插入一个元素，需要移动后续所有元素。
- 而LinkedList进行节点插入、删除却要高效得多，但是随机访问性能则要比动态数组慢。

所以，在应用开发中，如果事先可以估计到，应用操作是偏向于插入、删除，还是随机访问较多，就可以针对性的进行选择。这也是面试最常见的一个考察角度，给定一个场景，选择适合的数据结构，所以对于这种典型选择一定要掌握清楚。

考察Java集合框架，我觉得有很多方面需要掌握：

- Java集合框架的设计结构，至少要有一个整体印象。
- Java提供的主要容器（集合和Map）类型，了解或掌握对应的数据结构、算法，思考具体技术选择。
- 将问题扩展到性能、并发等领域。
- 集合框架的演进与发展。

作为Java专栏，我会在尽量围绕Java相关进行扩展，否则光是罗列集合部分涉及的数据结构就要占用很大篇幅。这并不代表那些不重要，数据结构和算法是基本功，往往也是必考的点，有些公司甚至以考察这些方面而非常知名（甚至是“臭名昭著”）。我这里以需要掌握典型排序算法为例，你至少需要熟知：

- 内部排序，至少掌握基础算法如归并排序、交换排序（冒泡、快排）、选择排序、插入排序等。
- 外部排序，掌握利用内存和外部存储处理超大数据集，至少要理解过程和思路。

考察算法不仅仅是如何简单实现，面试官往往会刨根问底，比如哪些是排序是不稳定的呢（快排、堆排），或者思考稳定性意味着什么；对不同数据集，各种排序的最好或最差情况，从某个角度如何进一步优化（比如空间占用，假设业务场景需要最小辅助空间，这个角度堆排序就比归并优异）等，从简单的了解，到进一步的思考，面试官通常还会观察面试者处

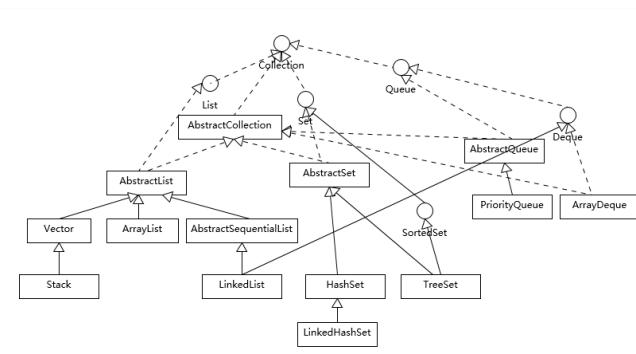
理问题和沟通时的思路。

以上只是一个方面的例子，建议学习相关书籍，如《算法导论》《编程珠玑》等，或相关[教程](#)。对于特定领域，比如推荐系统，建议咨询领域专家。单纯从面试的角度，很多朋友推荐使用一些算法网站如LeetCode等，帮助复习和准备面试，但坦白说我并没有刷过这些算法题，这也是仁者见仁智者见智的事情，招聘时我更倾向于考察面试者自身最擅长的东西，免得招到纯面试高手。

#### 知识扩展

我们先一起来理解集合框架的整体设计，为了有个直观的印象，我画了一个简要的类图。注意，为了避免混淆，我这里没有把java.util.concurrent下面的线程安全容器添加进来；也没有列出Map容器，虽然通常概念上我们也会把Map作为集合框架的一部分，但是它本身并不是真正的集合（Collection）。

所以，我今天主要围绕狭义的集合框架，其他都会在专栏后面的内容进行讲解。



我们可以看到Java的集合框架，Collection接口是所有集合的根，然后扩展开提供了三大类集合，分别是：

- List，也就是我们前面介绍最多的有序集合，它提供了方便的访问、插入、删除等操作。
- Set，Set是不允许重复元素的，这是和List最明显的区别，也就是不存在两个对象equals返回true。我们在日常开发中有很多需要保证元素唯一性的场合。
- Queue/Deque，则是Java提供的标准队列结构的实现，除了集合的基本功能，它还支持类似先入先出（FIFO，First-in-First-Out）或者后入先出（LIFO，Last-In-First-Out）等特定行为。这里不包括BlockingQueue，因为通常是并发编程场合，所以被放置在并发包里。

每种集合的通用逻辑，都被抽象到相应的抽象类之中，比如AbstractList就集中了各种List操作的通用部分。这些集合不是完全孤立的，比如，LinkedList本身，既是List，也是Deque哦。

如果阅读过更多[源码](#)，你会发现，其实，TreeSet代码里实际默认是利用TreeMap实现的，Java类库创建了一个Dummy对象“PRESENT”作为value，然后所有插入的元素其实是以键的形式放入了TreeMap里面；同理，HashSet其实也是以HashMap为基础实现的，原来他们只是Map类的马甲！

就像前面提到过的，我们需要对各种具体集合实现，至少了解基本特征和典型使用场景，以Set的几个实现为例：

- TreeSet支持自然顺序访问，但是添加、删除、包含等操作要相对低效（ $\log(n)$ 时间）。
- HashSet则是利用哈希算法，理想情况下，如果哈希散列正常，可以提供常数时间的添加、删除、包含等操作，但是它不保证有序。
- LinkedHashSet，内部构建了一个记录插入顺序的双向链表，因此提供了按照插入顺序遍历的能力，与此同时，也保证了常数时间的添加、删除、包含等操作，这些操作性能略低于HashSet，因为需要维护链表的开销。
- 在遍历元素时，HashSet性能受自身容量影响，所以初始化时，除非有必要，不然不要将其背后的HashMap容量设置过大。而对于LinkedHashSet，由于其内部链表提供的方便，遍历性能只和元素多少有关系。

我今天介绍的这些集合类，都不是线程安全的，对于Java.util.concurrent里面的线程安全容器，我在专栏后面会去介绍。但是，并不代表这些集合完全不能支持并发编程的场景，在Collections工具类中，提供了一系列的synchronized方法，比如

```
static <T> List<T> synchronizedList(List<T> list)
```

我们完全可以利用类似方法来实现基本的线程安全集合：

```
List list = Collections.synchronizedList(new ArrayList());
```

它的实现，基本就是将每个基本方法，比如get、set、add之类，都通过synchronized添加基本的同步支持，非常简单粗暴，但也非常实用。注意这些方法创建的线程安全集合，都符合迭代fail-fast行为，当发生意外的并发修改时，尽早抛出ConcurrentModificationException异常，以避免不可预计的行为。

另外一个经常会被考察到的问题，就是理解Java提供的默认排序算法，具体是什么排序方式以及设计思路等。

这个问题本身就是有点陷阱的意味，因为需要区分是Arrays.sort()还是Collections.sort()（底层是调用Arrays.sort()）；什么数据类型；多大的数据集，复杂排序是没必要的，Java会直接进行二分插入排序）等。

- 对于原始数据类型，目前使用的是所谓双轴快速排序（Dual-Pivot QuickSort），是一种改进的快速排序算法，早期版本是相对传统的快速排序，你可以阅读[源码](#)。
- 而对于对象数据类型，目前则是使用TimSort，思想上也是一种归并和二分插入排序（binarySort）结合的优化排序算法。TimSort并不是Java的独创，简单说它的思路是查找数据集中已经排好序的分区（这里叫run），然后合并这些分区来达到排序的目的。

另外，Java 8引入了并行排序算法（直接使用parallelSort方法），这是为了充分利用现代多核处理器的计算能力，底层实现基于fork-join框架（专栏后面会对fork-join进行相对详细的介绍），当处理的数据集比较小的时候，差距不明显，甚至还表现差一点；但是，当数据集增长到数万或百万以上时，提高就非常大了，具体还是取决于处理器和系统环境。

排序算法仍然在不断改进，最近双轴快速排序实现的作者提交了一个更进一步的改进，历时多年的研究，目前正在审核和验证阶段。根据作者的性能测试对比，相比于基于归并排序的实现，新改进可以提高随机数据排序速度提高10% ~ 20%，甚至在其他特征的数据集上也有几倍的提高，有兴趣的话你可以参考具体代码和介绍：<http://mail.openjdk.java.net/pipermail/core-libs-dev/2018-January/051000.html>。

在Java 8之中，Java平台支持了Lambda和Stream，相应的Java集合框架也进行了大范围的增强，以支持类似为集合创建相应stream或者parallelStream的方法实现，我们可以非常方便的实现函数式代码。

阅读Java源代码，你会发现，这些API的设计和实现比较独特，它们并不是实现在抽象类里面，而是以默认方法的形式实现在Collection这样的接口里！这是Java 8在语言层面的新特性，允许接口实现默认方法，理论上来说，我们原来实现在类似Collections这种工具类中的方法，大多可以转换到相应的接口上。针对这一点，我在面向对象主题，会专门梳理Java语言面向对象基本机制的演讲。

在Java 9中，Java标准类库提供了一系列的静态工厂方法，比如，List.of()、Set.of()，大大简化了构建小的容器实例的代码量。根据业界实践经验，我们发现相当一部分集合实例都是容量非常有限的，而且在生命周期中并不会进行修改。但是，在原有的Java类库中，我们可能不得不写成：

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
list.add("World");
```

而利用新的容器静态工厂方法，一句代码就够了，并且保证了不可变性。

```
List<String> simpleList = List.of("Hello", "world");
```

更进一步，通过各种of方法创建的实例，还应用了一些我们所谓的最佳实践，比如，它是不可变的，符合我们对线程安全的需求；它因为不需要考虑扩容，所以空间上更加紧凑等。

如果我们去看of方法的源码，你还会发现一个特别有意思的地方：我们知道Java已经支持所谓的可变参数 (varargs)，但是官方类库还是提供了一系列特定参数长度的方法，看起来似乎非常不优雅，为什么呢？这其实是为了最优的性能，JVM在处理变长参数的时候会有明显的额外开销，如果你需要实现性能敏感的API，也可以进行参考。

今天我从Vector、ArrayList、LinkedList开始，逐步分析其设计实现区别、适合的应用场景等，并进一步对集合框架进行了简单的归纳，介绍了集合框架从基础算法到API设计实现的各种改进，希望能对你的日常开发和API设计能够有帮助。

### 一课一练

关于今天我们要讨论的题目你做到心中有数了吗？留一道思考题给你，先思考一个应用场景，比如你需要实现一个云计算任务调度系统，希望可以保证VIP客户的任务被优先处理，你可以利用哪些数据结构或者标准的集合类型呢？更进一步讲，类似场景大多是基于什么数据结构呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



雷霹雳的爸爸

2018-05-22

在这个题目下，自然就会想到优先级队列了，但还需要额外考虑vip再分级，即同等级vip的平权的问题，所以应该考虑除了直接的和vip等级相关的优先级队列优先级规则问题，还得考虑同等级多个客户互相不被单一客户大量任务阻塞的问题，数据结构确实是基础，即便这个思考题考虑的这个场景，待调度数据估计会放在redis里面吧

作者回复

2018-05-23

赞

孙晓刚

2018-05-22

精选第一个对于读写效率问题，我觉得表述有欠缺，或者说不能那么绝对。  
1、并不是所有的增删都会开辟新内存，没有开辟新内存的头部增，效率也是杠杠的。  
2、头部删除也不需要开辟新内存，只是移出最后一个对象。

之前我也是接收了ArrayList的特性随机访问快，增删效率差。直到看到源码才知道，没那么绝对。直接导致结果就是本身适合使用ArrayList的场景会因为这个笼统的说法而选LinkedList

作者回复

2018-05-23

嗯，我文中特意强调了不包括头部

L.B.Q.Y

请教老师个问题，Collection接口的声明是带范型的，其中定义的Object[ ] toArray()方法为什么不是范型方式的？有什么原因吗？

作者回复

按照Javadoc，我觉得这个方法设计目的，就是让调用者精确控制类型；里面声明了，toArray(new Object[0])等同于toArray()

公号-Java大后端

Vector、ArrayList、LinkedList均为线型的数据结构，但是从实现方式与应用场景中又存在差别。

1 底层实现方式

ArrayList内部用数组来实现；LinkedList内部采用双向链表实现；Vector内部用数组实现。

2 读写机制

ArrayList在执行插入元素是超过当前数组预定义的最大值时，数组需要扩容，扩容过程需要调用底层System.arraycopy()方法进行大量的数组复制操作；在删除元素时并不会减少数组的容量（如果需要缩小数组容量，可以调用trimToSize()方法）；在查找元素时要遍历数组，对于非null的元素采取equals的方式寻找。

LinkedList在插入元素时，须创建一个新的Entry对象，并更新相应元素的前后元素的引用；在查找元素时，需遍历链表；在删除元素时，要遍历链表，找到要删除的元素，然后从链表上将此元素删除即可。

Vector与ArrayList仅在插入元素时容量扩充机制不一致。对于Vector，默认创建一个大小为10的Object数组，并将capacityIncrement设置为0。当插入元素数组大小不够时，如果capacityIncrement大于0，则将Object数组的大小扩大为现有size+capacityIncrement；如果capacityIncrement<=0，则将Object数组的大小扩大为现有大小的2倍。

3 读写效率

ArrayList对元素的增加和删除都会引起数组的内存分配空间动态发生变化。因此，对其进行插入和删除速度较慢，但检索速度很快。

LinkedList由于基于链表方式存放数据，增加和删除元素的速度较快，但是检索速度较慢。

4 线程安全性

ArrayList、LinkedList为非线程安全；Vector是基于synchronized实现的线程安全的ArrayList。

需要注意的是：单线程应尽量使用ArrayList，Vector因为同步会有性能损耗；即使在多线程环境下，我们可以利用Collections这个类中为我们提供的synchronizedList(List list)方法返回一个线程安全的同步列表对象。

问题回答

利用PriorityBlockingQueue或Disruptor可实现基于任务优先级为调度策略的执行调度系统。

曹铮

既然是Java的主题，那就用PriorityBlockingQueue吧。

如果是真实场景肯定会考虑高可用能持久化的方案。

其实我觉得应该参考银行窗口，同时三个窗口，就是三个队列，银台就是消费者线程，某一个窗口vip优先，没有vip时也为普通客户服务。要实现，要么有个dispatcher，要么保持vip通道不许普通进入，vip柜台随时从其他队列偷

作者回复

有道理

Miaozhe

杨老师，问个问题，Collection接口下面已细化了List、Set和Queue子接口，未什么又定义了AbstractCollection这个抽象类？具体是什么考虑？以为我发现3个接口的子类都是集成这个抽象类。

作者回复

三个都是Collection，总还是有共同行为的

我奋斗去了

可以使用priority queue，维护两个队列 一个VIP队列 一个普通用户队列。当VIP队列有人的情况优先处理

作者回复

为什么用两个队列，PriorityQueue不是有优先级了

Leo

使用优先级队列实现堆，可以根据优先级进行操作

13683815260

面试的重点HashMap实现原理，扩展什么的，1.7和1.8的区别。还有和hashtable的异同。还有juc下面集合的熟悉程度。

作者回复

下两篇就是

Hesher

我觉得使用queue来实现VIP业务就行了，检查这个队列深度，大于0时就优先先处理。分布式环境下用MQ实现。

13683815260

面试重点HashMap，1.7和1.8然后还有hashtable,还有juc下面的集合

自己深情共白头

之前一直以为Vector不属于集合，只是数组。学习了。针对VIP客户任务优先处理场景，认为采用SortSet进行，按照默认排序即可，数值越小优先级越高，和线程的优先级级别一致

作者回复

和优先队列相比，不那么紧凑，例如treemap用的树比堆要多了节点开销

Allen

多级反馈队列

Miaozhe

2018-05-23

2018-05-23

2018-05-22

2018-05-22

2018-05-23

2018-05-26

2018-05-28

2018-05-22

2018-05-23

2018-05-22

2018-05-22

2018-05-23

2018-05-22

2018-05-22

2018-07-09

2018-07-11

2018-06-30

2018-05-28

再问个问题，为什么ArrayDeque类中的存储对象Object数组，数组的长度必须是2的n次方。

Miaozhe

2018-05-28

今天看了一下PriorityQueue的源码，发现其是使用最小堆结构(二叉堆)，存放在数组中(数组索引对应树的从上到下，从左到右)。采用上面最小，每插入一个数据，就先与根节点比较，如果小于根节，依次换位置，大于根节点，就放在最后一个位置。

Miaozhe

2018-05-28

杨老师，有个问题，TreeSet为什么不支持正序，只支持倒序(DescendingIterator)? Tree本身支持正序列。

zh

2018-05-26

比较片面的说，java集合类底层基本上就是基于数组或者链表来实现的，数组的地址连续性决定了其随机存取速度较快，但是涉及到扩容则比较耗时，而链表则不存在扩容的性能消耗，但随机访问需要遍历地址因此相对数组要慢，所以判断一个集合的特点可以先判断是基于数组还是链表。

L.B.Q.Y

2018-05-24

集合框架的那张图，SortedSet应该是接口，图里面画成类了，TreeSet看起来有两个父类。

作者回复

2018-05-25

感谢指出，已经记录下来去修正一下

Lawt

2018-05-24

写得还挺不错的，担心36期能讲那么多没容吗？持续关注，相信您，期待

悬崖丶

2018-05-23

想问一个小白问题，List接口继承了Collection接口，为什么List接口还要重写一遍Collection接口中的一些方法

作者回复

2018-05-24

覆盖（override），面向对象多态的基本方面

Miaozhe

2018-05-23

最近学习过程中，感觉看原代码比较吃力。一个Hash Map没有看明白，这么对Key的Set赋值，这么对value的Collection赋值。

作者回复

2018-05-24

下一篇就是

呵呵

2018-05-23

阅读速度太快了

webwombat

2018-05-22

那个问题，应该是priority queue吧？操作系统的进程调度一般都是基于优先级队列来实现的。

作者回复

2018-05-23

yes，人家可能进一步提出更多场景继续考

王磊

2018-05-22

使用优先级队列，按照任务按照优先级排好序，这样优先级高的任务被优先处理。



## 第9讲 | 对比Hashtable、HashMap、TreeMap有什么不同?

2018-05-24 杨晓峰



第9讲 | 对比Hashtable、HashMap、TreeMap有什么不同?  
杨晓峰  
- 0:00 / 12:16

Map是广义Java集合框架中的另外一部分，HashMap作为框架中使用频率最高的类型之一，它本身以及相关类型自然也是面试考察的热点。

今天我要问你的是，[对比Hashtable、HashMap、TreeMap有什么不同？](#) 谈谈你对HashMap的掌握。

## 典型回答

Hashtable、HashMap、TreeMap都是最常见的一些Map实现，是以键值对的形式存储和操作数据的容器类型。

Hashtable是早期Java类库提供的一个[哈希表](#)实现，本身是同步的，不支持null键和值，由于同步导致的性能开销，所以已经很少被推荐使用。

HashMap是应用更加广泛的哈希表实现，行为上大致上与HashTable一致，主要区别在于HashMap不是同步的，支持null键和值等。通常情况下，HashMap进行put或者get操作，可以达到常数时间的性能，所以它是绝大部分利用键值对存取场景的首选，比如，实现一个用户ID和用户信息对应的运行时存储结构。

TreeMap则是基于红黑树的一种提供顺序访问的Map，和HashMap不同，它的get、put、remove之类操作都是 $O(\log(n))$ 的时间复杂度，具体顺序可以由指定的Comparator来决定，或者根据键的自然顺序来判断。

## 考点分析

上面的回答，只是对一些基本特征的简单总结，针对Map相关可以扩展的问题很多，从各种数据结构、典型应用场景，到程序设计实现的技术考量，尤其是在Java 8里，HashMap本身发生了非常大的变化，这些都是经常考察的方面。

很多朋友向我反馈，面试官似乎钟爱考察HashMap的设计和实现细节，所以今天我会增加相应的源码解读，主要专注于下面几个方面：

- 理解Map相关类似整体结构，尤其是有序数据结构的一些要点。
- 从源码去分析HashMap的设计和实现要点，理解容量、负载因子等，为什么需要这些参数，如何影响Map的性能，实践中如何取舍等。
- 理解树化改造的相关原理和改进原因。

除了典型的代码分析，还有一些有意思的并发相关问题也经常会被提到，如HashMap在并发环境可能出现[无限循环占用CPU](#)、size不准等诡异的问题。

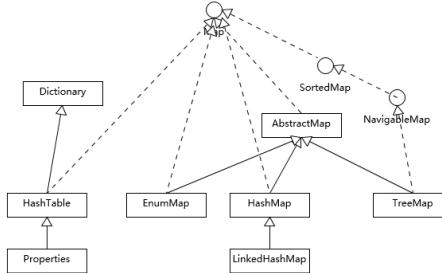
我认为这是一种典型的使用错误，因为HashMap明确声明不是线程安全的数据结构，如果忽略这一点，简单用在多线程场景里，难免会出现问题。

理解导致这种错误的原因，也是深入理解并发程序运行的好办法。对于具体发生了什么，你可以参考这篇很久以前的[分析](#)，里面甚至提供了示意图，我就不再重复别人写好的内容了。

## 知识扩展

## 1.Map整体结构

首先，我们先对Map相关类型有个整体了解，Map虽然通常被包括在Java集合框架里，但是其本身并不是狭义上的集合类型（Collection），具体你可以参考下面这个简单类图。



Hashtable比较特别，作为类似Vector、Stack的早期集合相关类型，它是扩展了Dictionary类的，类结构上与HashMap之类明显不同。

HashMap等其他Map实现则是都扩展了AbstractMap，里面包含了通用方法抽象。不同Map的用途，从类图结构就能体现出来，设计目的已经体现在不同接口上。

大部分使用Map的场景，通常就是放入、访问或者删除，而对顺序没有特别要求，HashMap在这种情况下基本是最好的选择。HashMap的性能表现非常依赖于哈希码的有效性，请务必掌握hashCode和equals的一些基本约定，比如：

- equals相等，hashCode一定要相等。
- 重写了hashCode也要重写equals。
- hashCode需要保持一致性，状态改变返回的哈希值仍然要一致。
- equals的对称、反射、传递等特性。

这方面内容网上有很多资料，我就不再这里详细展开了。

针对有序Map的分析内容比较有限，我再补充一些，虽然LinkedHashMap和TreeMap都可以保证某种顺序，但二者还是非常不同的。

- LinkedHashMap通常提供的是遍历顺序符合插入顺序，它的实现是通过为条目（键值对）维护一个双向链表。注意，通过特定构造函数，我们可以创建反映访问顺序的实例，所谓的put、get、compute等，都算作“访问”。

这种行为适用于一些特定应用场景，例如，我们构建一个空间占用敏感的资源池，希望可以自动将最不常被访问的对象释放掉，这就利用LinkedHashMap提供的机制来实现，参考下面的示例：

```

import java.util.LinkedHashMap;
import java.util.Map;
public class LinkedHashMapSample {
    public static void main(String[] args) {
        LinkedHashMap<String, String> accessOrderedMap = new LinkedHashMap<>(16, 0.75F, true){
            @Override
            protected boolean removeEldestEntry(Map.Entry<String, String> eldest) { // 实现自定义删除策略，否则行为就和普通Map没有区别
                return size() > 3;
            }
        };
        accessOrderedMap.put("Project1", "Valhalla");
        accessOrderedMap.put("Project2", "Banana");
        accessOrderedMap.put("Project3", "Loom");
        accessOrderedMap.forEach((k,v) -> {
            System.out.println(k + ":" + v);
        });
        // 模拟访问
        accessOrderedMap.get("Project2");
        accessOrderedMap.get("Project2");
        accessOrderedMap.get("Project3");
        System.out.println("Iterate over should be not affected:");
        accessOrderedMap.forEach((k,v) -> {
            System.out.println(k + ":" + v);
        });
        // 触发删除
        accessOrderedMap.put("Project4", "Mission Control");
        System.out.println("Oldest entry should be removed:");
        accessOrderedMap.forEach((k,v) -> // 遍历顺序不变
            System.out.println(k + ":" + v));
    }
}
  
```

- 对于TreeMap，它的整体顺序是由键的顺序关系决定的，通过Comparator或Comparable（自然顺序）来决定。

我在上一讲留给你的思考题提到了，构建一个具有优先级的调度系统的问题，其本质就是个典型的优先队列场景，Java标准库提供了基于二叉堆实现的PriorityQueue，它们都是依赖于同一种排序机制，当然也包括TreeMap的 TreeMap。

类似hashCode和equals的约定，为了避免模棱两可的情况，自然顺序同样需要符合一个约定，就是compareTo的返回值需要和equals一致，否则就会出现模棱两可情况。

我们可以分析TreeMap的put方法实现：

```
public V put(K key, V value) {
    Entry<K,V> t = ...
    cmp = k.compareTo(t.key);
    if (cmp < 0)
        t = t.left;
    else if (cmp > 0)
        t = t.right;
    else
        return t.setValue(value);
    // ...
}
```

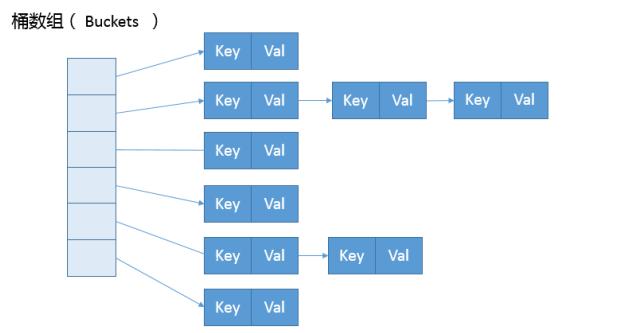
从代码里，你可以看出什么呢？当我不遵守约定时，两个不符合唯一性（equals）要求的对象被当作是同一个（因为，compareTo返回0），这会导致歧义的行为表现。

## 2.HashMap源码分析

前面提到，HashMap设计与实现是个非常高频的面试题，所以我会在这进行相对详细的源码解读，主要围绕：

- HashMap内部实现基本点分析。
- 容量（capacity）和负载系数（load factor）。
- 树化。

首先，我们来一起看看HashMap内部的结构，它可以看作是数组（Node[] table）和链表结合组成的复合结构，数组被分为一个个桶（bucket），通过哈希值决定了键值对在这个数组的地址；哈希值相同的键值对，则以链表形式存储，你可以参考下面的示意图。这里需要注意的是，如果链表大小超过阈值（TREEIFY\_THRESHOLD, 8），图中的链表就会被改造为树形结构。



从非拷贝构造函数的实现来看，这个表格（数组）似乎并没有在最初就初始化好，仅仅设置了一些初始值而已。

```
public HashMap(int initialCapacity, float loadFactor){
    // ...
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

所以，我们深刻怀疑，HashMap也许是按照lazy-load原则，在首次使用时被初始化（拷贝构造函数除外，我这里仅介绍最通用的场景）。既然如此，我们去看看put方法实现，似乎只有一个putVal的调用：

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
```

看来主要的密码似乎藏在putVal里面，到底有什么秘密呢？为了节省空间，我这里只截取了putVal比较关键的几部分。

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
              boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else
```

```

    else {
        // ...
        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for first
            treeifyBin(tab, hash);
        // ...
    }
}

```

从putVal方法最初的几行，我们就可以发现几个有意思的地方：

- 如果表格是null，resize方法会负责初始化它，这从tab = resize()可以看出。
- resize方法兼顾两个职责，创建初始存储表格，或者在容量不满足需求的时候，进行扩容（resize）。
- 在放置新的键值对的过程中，如果发生下面条件，就会发生扩容。

```

if (++size > threshold)
    resize();

```

- 具体键值对在哈希表中的位置（数组index）取决于下面的位运算：

```
i = (n - 1) & hash
```

仔细观察哈希值的源头，我们会发现，它并不是key本身的hashCode，而是来自HashMap内部的另外一个hash方法。注意，为什么这里需要将高位数据移位到低位进行异或运算呢？这是因为有些数据计算出的哈希值差异主要在高位，而HashMap里的哈希寻址是忽略容量以上的高位的，那么这种处理就可以有效避免类似情况下的哈希碰撞。

```

static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>>16);
}

```

- 我前面提到的链表结构（这里叫bin），会在达到一定门限值时，发生树化，我稍后会分析为什么HashMap需要对bin进行处理。

可以看到，putVal方法本身逻辑非常集中，从初始化、扩容到树化，全部都和它有关，推荐你阅读源码的时候，可以参考上面的主要逻辑。

我进一步分析一下身兼多职的resize方法，很多朋友都反馈经常被面试官追问它的源码设计。

```

final Node<K,V>[] resize() {
    // ...
    else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
              oldCap >= DEFAULT_INITIAL_CAPACITY)
        newThr = oldThr << 1; // double there
    // ...
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {
        // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ? (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    Node<K,V>[] newTab = (Node<K,V>[][])new Node[newCap];
    table = n;
    // 移动旧新的数组结构
}

```

依据resize源码，不考虑极端情况（容量理论最大极限由MAXIMUM\_CAPACITY指定，数值为 $1 < 30$ ，也就是 $2^{30}$ 次方），我们可以归纳为：

- 门限值  $\times$  (负载因子)，如果构建HashMap的时候没有指定它们，那么就是依据相应的默认常量值。
- 门限通常是以倍数进行调整 ( $newThr = oldThr << 1$ )，我前面提到，根据putVal中的逻辑，当元素个数超过门限大小时，则调整Map大小。
- 扩容后，需要将老的数组中的元素重新放置到新的数组，这是扩容的一个主要开销来源。

### 3. 容量、负载因子和树化

前面我们快速梳理了一下HashMap从创建到放入键值对的相关逻辑，现在思考一下，为什么我们需要在乎容量和负载因子呢？

这是因为容量和负载系数决定了可用的桶的数量，空桶太多会浪费空间，如果使用的太满则会严重影响操作的性能。极端情况下，假设只有一个桶，那么它就退化成了链表，完全不能提供所谓常数时间存的性能。

既然容量和负载因子这么重要，我们在实践中应该如何选择呢？

如果能够知道HashMap要存取的键值对数量，可以考虑预先设置合适的容量大小。具体数值我们可以根据扩容发生的条件来做简单预估，根据前面的代码分析，我们知道它需要符合计算条件：

负载因子 \* 容量 > 元素数量

所以，预先设置的容量需要满足，大于“预估元素数量/负载因子”，同时它是2的幂数，结论已经非常清晰了。

而对于负载因子，我建议：

- 如果没有特别需求，不要轻易进行更改，因为JDK自身的默认负载因子是非常符合通用场景的需求的。
- 如果确实需要调整，建议不要设置超过0.75的数值，因为会显著增加冲突，降低HashMap的性能。
- 如果使用太小的负载因子，按照上面的公式，预设容量值也进行调整，否则可能会导致更加频繁的扩容，增加无谓的开销，本身访问性能也会受影响。

我们前面提到了树化改造，对应逻辑主要在putVal和treeifyBin中。

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        //树化改造逻辑
    }
}
```

上面是精简过的treeifyBin示意，综合这两个方法，树化改造的逻辑就非常清晰了，可以理解为，当bin的数量大于TREEIFY\_THRESHOLD时：

- 如果容量小于MIN\_TREEIFY\_CAPACITY，只会进行简单的扩容。
- 如果容量大于MIN\_TREEIFY\_CAPACITY，则会进行树化改造。

那么，为什么HashMap要树化呢？

本质上这是个安全问题。因为在元素放置过程中，如果一个对象哈希冲突，都被放置到同一个桶里，则会形成一个链表，我们知道链表查询是线性的，会严重影响存取的性能。

而在现实世界，构造哈希冲突的数据并不是非常复杂的事情，恶意代码就可以利用这些数据大量与服务器端交互，导致服务器端CPU大量占用，这就构成了哈希碰撞拒绝服务攻击，国内一线互联网公司就发生过类似攻击事件。

今天我从Map相关的几种实现对比，对各种Map进行了分析，讲解了有序集合类型容易混淆的地方，并从源码级别分析了HashMap的基本结构，希望对你有所帮助。

一课一练

关于今天我们要讨论的题目你做到心中有数了吗？留一道思考题给你，解决哈希冲突有哪些典型方法呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



天凉好个秋

2018-05-24

解决哈希冲突的常用方法有：

**开放地址法**  
基本思想是：当关键字key的哈希地址p=H(key) 出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p1为基础，产生另一个哈希地址p2，……，直到找出一个不冲突的哈希地址pi，将相应元素存入其中。

**再哈希法**

这种方法是同时构造多个不同的哈希函数：  
Hi=RHi(key) i=1, 2, …, k  
当哈希地址Hi=RHi(key) 出现冲突时，再计算Hi=RH2(key) ……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

**链地址法**

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

## 建立公共溢出区

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

三口先生

2018-05-24

最常用的方法就是线性再散列。即插入元素时，没有发生冲突放在原有的规则下的空槽下，发生冲突时，简单遍历hash表，找到表中下一个空槽，进行元素插入。查找元素时，找到相应的位置的元素，如果不匹配则进行遍历hash表。

然后就是我们非线性再散列，就是冲突时，再hash，核心思想是，如果产生冲突，产生一个新的hash值进行寻址，如果还是冲突，则继续。

上述的方法，主要的缺点在于不能从表中删除元素。

还有就是我们hashmap的思想外部拉链。

公号-Java大后端

## Hashtable、HashMap、TreeMap心得

三者均实现了Map接口，存储的内容是基于key-value的键值对映射，一个映射不能有重复的键，一个键最多只能映射一个值。

## (1) 元素特性

HashTable中的key、value都不能为null；HashMap中的key、value可以为null，很显然只能有一个key为null的键值对，但是允许有多个值为null的键值对；TreeMap中尚未实现Comparator接口时，key不可以为null；当实现Comparator接口时，若未对null情况进行判断，则key不可以为null，反之亦然。

## (2) 序序特性

HashTable、HashMap具有无序特性。TreeMap是利用红黑树来实现的（树中的每个节点的值，都会大于或等于它的左子树的所有节点的值，并且小于或等于它的右子树中的所有节点的值），实现了SortMap接口，能够对保存的记录根据键进行排序。所以一般需要排序的情况下是选择TreeMap来进行，默认为升序排序方式（深度优先搜索），可自定义实现Comparator接口实现排序方式。

## (3) 初始化与增长方式

初始化时：HashTable在不指定容量的情况下默认容量为11，且不要求底层数组的容量一定要为2的整数次幂；HashMap默认容量为16，且要求容量一定为2的整数次幂。

扩容时：Hashtable将容量变为原来的2倍加1；HashMap扩容将容量变为原来的2倍。

## (4) 线程安全性

HashTable其方法函数都是同步的（采用synchronized修饰），不会出现两个线程同时对数据进行操作的情况，因此保证了线程安全性。也正因为如此，在多线程运行环境下效率表现非常低。

因为当一个线程调用HashTable的同步方法时，其他线程也访问同步方法就会进入阻塞状态。比如当一个线程在添加数据时候，另外一个线程即使执行获取其他数据的操作也必须被阻塞，大大降低了程序的运行效率，在新版本中已被废弃，不推荐使用。

HashMap不支持线程的同步，即任一时刻可以有多个线程同时与HashMap：可能会导致数据的一致性。如果需要同步（1）可以用Collections.synchronizedMap方法；（2）使用ConcurrentHashMap类，相较于HashMap类，ConcurrentHashMap基于lock实现锁分段技术，首先将Map存放的数据分成一段一段的存储方式，然后给每一段数据分配一把锁，当一个线程占用锁访问其中一段的数据时，其他段的数据也能被其他线程访问。ConcurrentHashMap不仅保证了多线程运行环境下的数据访问安全性，而且性能上有长足的提升。

## (5) 一段话HashMap

HashMap基于哈希思想，实现对数据的读写。当我们讲键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，让后找到bucket位置来储存值对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。HashMap在每个链表节点中储存键值对对象，当两个键对象的hashcode相同时，它们会储存在同一个bucket位置的链表中，可通过键对象的equals()方法来找到键值对。如果链表大小超过阈值（TREEIFY\_THRESHOLD, 8），链表就会被改造成树形结构。

j.c.

2018-05-24

这是面试必问题。什么时候也能讲讲红黑树的树化具体过程，那个旋转一直没搞懂。另外treeifyBin这个单词的词面意思是什么？

coolboy

2018-06-24

removeEldestEntry这个方法是不是指移除最旧的对象，也就是按照最先被put进来的顺序，而不是指不常访问的对象。

代码狂徒

2018-05-24

针对负载因子，您所指的存太满会影响性能是指什么？毕竟已经开辟了相应内存空间的，没什么不用呢？

作者回复

2018-05-25

冲突可能会增加，影响查询之类性能，当然看具体的需求

xinfangke

2018-05-29

老师 如果 hashmap 中不存在 hash 冲突 是不是就相当于一个数组结构呢 就不存在链表了呢

作者回复

2018-05-29

我理解是

清风

2018-07-05

感觉每个知识点都很重要，但又点到为止，感觉读完不痛不痒，好像学到什么，但细想又没掌握什么，希望能够深入一点！

李飞

2018-06-02

这个内容挺多的，这要补多少天的功课，才能搞定◆◆

作者回复

2018-06-03

加油

鲤鱼

2018-05-29

读到最后链表树化刚准备开始飙车，结果突然跳车。树化讲细点更好

作者回复

2018-05-29

感谢反馈，最近几章篇幅都超标了……只能照顾大多数需求，抱歉

zjh

2018-05-28

受教了，把java集合的源代码掌握了，对java和数据结构的了解都会有很大的提升

代码狂徒

2018-05-24

为什么不是一开始就树化，而是要等到一定程度再树化，链表一开始就是消耗查找性能啊？另外其实不太明白为什么是0.75的负载因子，如果是0.8或者0.9会有什么影响吗？毕竟已经开辟了相关内存空间

作者回复

2018-05-28

回复了，数据少的时候，平均访问长度很小，没必要麻烦；0.75是通用场景建议，取个平衡，具体看你调整它目标是什么了

灰飞灰猪不会灰飞.烟灭

这是1.7的hashmap吧?

Jerry很娘

我一直认为：JAVA集合类是非常好的学习材料。

如果敢说精通JAVA集合类，计算机功底肯定不会太差

amourling

提个意见，文章中请不要出现太多似乎，怀疑之类的必须，该是什么就是什么，不确定的不要拿出来。

Yonei

我感觉树化一个目的是防止hash冲突导致的resize时的死循环，还有就是减少查找遍历路径，毕竟树的查找不用遍历全部，特别像是平衡二叉树的遍历。

作者回复

是的，树性能恶化不会太剧烈

t

分析HashMap源码那一段是基于JDK8的，我对着JDK7看了半天.....发现不对◆◆

沈培斌

老师，感觉最后讲为什么要树化的时候结尾有点突然。既然您说了树化本质上是个安全问题，那么树化以后怎么就解决安全问题了呢，这个我没有理解，谢谢。

作者回复

这个树实现提供可靠的logn访问性能，哈希表好的时候比它强，问题是出在最差情况，退化成链表了

coolboy

杨老师，那个LinkedHashMap例子，我现在改调用两次get("Project1")，但是project4 put后，把Project1给删除了，按照不常访问的情况的话，不应该删除Project1呀，本地jdk1.8

A\_吖我去

所以，并发访问hashmap为啥会死循环，那个链接我访问不到

吴较瘦

我觉得树化的前提有两点，第一点是当前桶内元素个数大于8，第二点是数组的长度大于64。同时满足以上两点时，才会将当前桶内的线性链表转化为以key值排序的红黑树。

影随

您的留言不能直接回复，我是11楼读者。我用的jdk版本是1.8.0。 报错的内容为: both methods have same ensure, yet neither overrides the other。

目前正在试着装10.9貌似已经被废弃了

影随

LinkedHashMapSample 那个示例，为什么

accessOrderedMap  
@Override 的 removeEldestEntry()方法报错?  
只有我这儿报错吗？

作者回复

什么错？Java版本是不是太老了？示例是运行过的，不过我本地最低版本是9

江昆

为什么 HashMap 要树化呢？因为在最坏条件下，链表的查询时间是O(N)， 数的查询时间是O(LOG N)？能请老师解释一下为什么说本质上是因为安全呢？谢谢老师

作者回复

可以构建合适的数据进行哈希碰撞攻击

Honey拯救世界

load factor我觉得叫负载系数好那么一点点，系数有比值的意思，个人看法

Miaozhe

杨老师，我使用你提供的LinkedHashMap Simple样例，.get()方法感觉没有效果，新增后，触发删除，删除的还是第一个插入的数据。

作者回复

没错，第一个就是访问最少的元素，所以按照访问顺序会被删除

沉默的雪人

hashmap的树化，我记得是Jdk1.8的内容吧

作者回复

对

刘琨

不懂红黑树

齐帆

2018-05-24

2018-05-24

2018-07-11

2018-07-05

2018-07-07

2018-07-03

2018-06-27

2018-06-27

2018-06-24

2018-06-20

2018-06-08

2018-06-04

2018-06-05

2018-06-03

2018-06-05

2018-05-30

2018-05-30

2018-05-30

2018-05-30

2018-05-30

2018-05-28

2018-05-26

同问  
请问老师，为什么不是一开始就树化，而是要等到一定程度再树化，链表一开始就是消耗查找性能啊？另外其实不太明白为什么是0.75的负载因子，如果是0.8或者0.9会有什么影响吗？毕竟已经开辟了相关内存空间

作者回复

才回来，倒时差，回复不及时；  
链表短时访问看不出明显差异，但构建简单多了；  
记得回答过了，这不是规定是通常建议，填充多了，冲突可能就大了，再说所谓调优总是有个前提、目标的，看具体情况

2018-05-28

Key

2018-05-24

hashtable为什么不能有key和value为空的情况 一直搞不懂这个

叶易

2018-05-24

hashMap里面再hash，综合高位与低位的特征是个很好的方法，值得借鉴

鹤鸣

2018-05-24

写的好，感谢

作者回复

2018-05-28

非常感谢

Libra

2018-05-24

首先， hashmap的扩容一定是容量不满足要求(cap==0或者cap==阈值) 树化的目的是减少访问时用时。如果沿用1.7的链表设计，在哈希碰撞比较多的情况下会大幅度降低性能。

df1996

2018-05-24

老师，我只是为了测试链表转为红黑树 构造函数设置了长度1，负载因子10，插入8条的时候是正常的8长度链表，put9个的时候就自动扩容了而不是转为红黑树，我在treeifyBin方法中看到当前数组长度小于64就会resize，这是对不合理的负载因子设置的一种保护吗？

极客er

2018-05-24

那如何避免 哈希碰撞拒绝服务攻击 呢？

昵称而已，何必执着

2018-05-24

再hash，线性探测，线性平方探测！

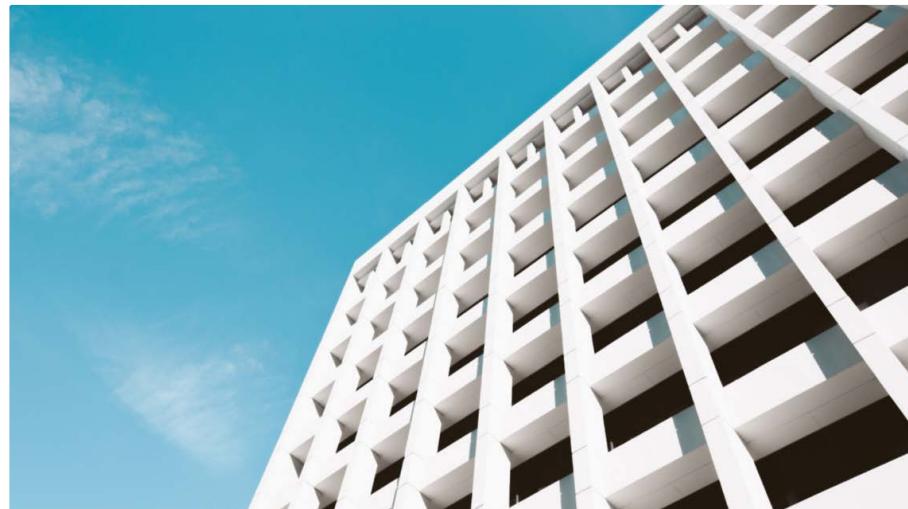
曹铮

2018-05-24

很多hashmap的负载因子都用0.75，这是业内共识？有论文？







第10讲 | 如何保证集合是线程安全的? ConcurrentHashMap如何实现高效地线程安全?  
杨晓峰  
G

- 0:00 / 10:46

我在之前两讲介绍了Java集合框架的典型容器类，它们绝大部分都不是线程安全的，仅有的线程安全实现，比如Vector、Stack，在性能方面也远不尽如人意。幸好Java语言提供了并发包（java.util.concurrent），为高度并发需求提供了更加全面的工具支持。

今天我要问你的是，**如何保证容器是线程安全的? ConcurrentHashMap如何实现高效地线程安全?**

#### 典型回答

Java提供了不同层面的线程安全支持。在传统集合框架内部，除了Hashtable等同步容器，还提供了所谓的同步包装器（Synchronized Wrapper），我们可以调用Collections工具类提供的包装方法，来获取一个同步的包装容器（如Collections.synchronizedMap），但是它们都是利用非常粗粒度的同步方式，在高并发情况下，性能比较低下。

另外，更加普遍的选择是利用并发包提供的线程安全容器类，它提供了：

- 各种并发容器，比如ConcurrentHashMap、CopyOnWriteArrayList。
- 各种线程安全队列（Queue/Deque），如ArrayBlockingQueue、SynchronousQueue。
- 各种有序容器的线程安全版本等。

具体保证线程安全的方式，包括有从简单的synchronize方式，到基于更加精细化的，比如基于分离锁实现的ConcurrentHashMap等并发实现等。具体选择要看开发的场景需求，总体来说，并发包内提供的容器通用场景，远优于早期的简单同步实现。

#### 考点分析

谈到线程安全和并发，可以说是Java面试中必考的考点，我上面给出的回答是一个相对宽泛的总结，而且ConcurrentHashMap等并发容器实现也在不断演进，不能一概而论。

如果要深入思考并回答这个问题及其扩展方面，至少需要：

- 理解基本的线程安全工具。
- 理解传统集合框架并发编程中Map存在的问题，清楚简单同步方式的不足。
- 梳理并发包内，尤其是ConcurrentHashMap采取了哪些方法来提高并发表现。
- 最好能够掌握ConcurrentHashMap自身的演进，目前的很多分析资料还是基于其早期版本。

今天我主要是延续专栏之前两讲的内容，重点解读经常被同时考察的HashMap和ConcurrentHashMap。今天这一讲并不是对并发方面的全面梳理，毕竟这也不是专栏一讲可以介绍完整的，算是个开胃菜吧，类似CAS等更加底层的机制，后面会在Java进阶模块中的并发主题有更加系统的介绍。

#### 知识扩展

##### 1.为什么需要ConcurrentHashMap?

Hashtable本身比较低效，因为它的实现基本就是将put、get、size等各种方法加上“synchronized”。简单来说，这就导致了所有并发操作都要竞争同一把锁，一个线程在进行同步操作时，其他线程只能等待，大大降低了并发操作的效率。

前面已经提过HashMap不是线程安全的，并发情况会导致类似CPU占用100%等问题，那么能不能利用Collections提供的同步包装器来解决问题呢？

看看下面的代码片段，我们发现同步包装器只是利用输入Map构造了另一个同步版本，所有操作虽然不再声明成为synchronized方法，但是还是利用了“this”作为互斥的mutex，没有真正意义上的改进！

```
private static class SynchronizedMap<K,V>
```

```

    implements Map<K,V>, Serializable {
    private final Map<K,V> m; // Backing Map
    final Object mutex; // Object on which to synchronize
    ...
    public int size() {
        synchronized (mutex) {return m.size();}
    }
    ...
}

```

所以，`Hashtable`或者同步包装版本，都只是适合在非高度并发的场景下。

## 2.ConcurrentHashMap分析

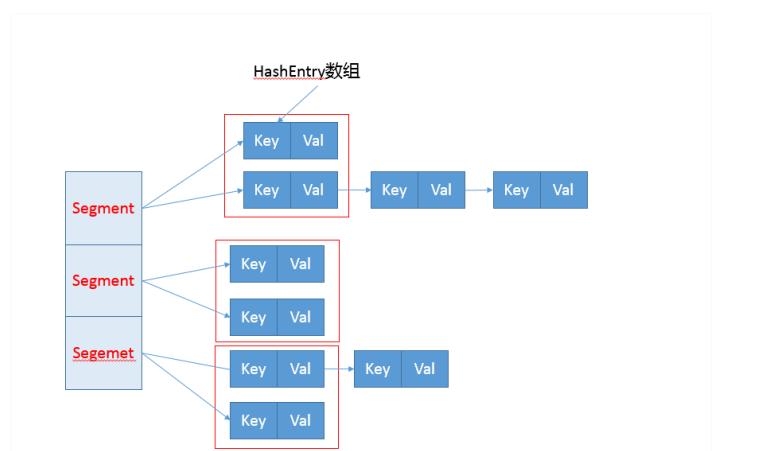
我们再来看看`ConcurrentHashMap`是如何设计实现的，为什么它能大大提高并发效率。

首先，我这里强调，`ConcurrentHashMap`的设计实现其实一直在演化，比如在Java 8中就发生了非常大的变化（Java 7其实也有不少更新），所以，我这里将比较分析结构、实现机制等方面，对比不同版本的主要区别。

早期`ConcurrentHashMap`，其实现是基于：

- 分离锁，也就是将内部进行分段（Segment），里面则是`HashEntry`的数组，和`HashMap`类似，哈希相同的条目也是以链表形式存放。
- `HashEntry`内部使用`volatile`的`value`字段来保证可见性，也利用了不可变对象的机制以改进利用`Unsafe`提供的底层能力，比如`volatile access`，去直接完成部分操作，以最优化性能。毕竟`Unsafe`中的很多操作都是JVM intrinsic优化过的。

你可以参考下面这个早期`ConcurrentHashMap`内部结构的示意图，其核心是利用分段设计，在进行并发操作的时候，只需要锁定相应段，这样就有效避免了类似`Hashtable`整体同步的问题，大大提高了性能。



在构造的时候，`Segment`的数量由所谓的`concurrencyLevel`决定，默认是16，也可以在相应构造函数直接指定。注意，Java需要它是2的幂数值，如果输入是类似15这种非幂值，会被自动调整到16之类2的幂数值。

具体情况，我们一起看看一些`Map`基本操作的[源码](#)，这是JDK 7比较新的`get`代码。针对具体的优化部分，为方便理解，我直接注释在代码段里，`get`操作需要保证的是可见性，所以并没有什么同步逻辑。

```

public V get(Object key) {
    Segment<K,V> s; // manually integrate access methods to reduce overhead
    HashEntry<K,V>[] tab;
    int h = hash(key.hashCode());
    //利用位操作替换普通的数学运算
    long u = (((h >> segmentShift) & segmentMask) << SSHIFT) + SBASE;
    //以Segment为单位进行定位
    //利用Unsafe直接进行volatile access
    if ((s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)) != null &&
        (tab = s.table) != null) {
        //省略
    }
    return null;
}

```

而对于`put`操作，首先是通过二次哈希避免哈希冲突，然后以`Unsafe`调用方式，直接获取相应的`Segment`，然后进行线程安全的`put`操作：

```

public V put(K key, V value) {
    Segment<K,V> s;
    if (value == null)
        throw new NullPointerException();
    // 二次哈希，以保证数据分散性，避免哈希冲突
    int hash = hash(key.hashCode());
    int j = (hash >> segmentShift) & segmentMask;
    if ((s = (Segment<K,V>)UNSAFE.getObject(
        segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment
        s = ensureSegment(j);
    return s.put(key, hash, value, false);
}

```

其核心逻辑实现在下面的内部方法中：

```

final V put(K key, int hash, V value, boolean onlyIfAbsent) {
    // scanAndLockForPut会去查找是否有key相同node
    // 无论如何，确实获取锁
    HashEntry<K,V> node = tryLock() ? null :
        scanAndLockForPut(key, hash, value);
    V oldValue;
    try {
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
            if (e != null) {
                K k;
                // 更新已有value...
            }
            else {
                // 放置HashEntry到特定位置，如果超过阈值，进行rehash
                // ...
            }
        }
    } finally {
        unlock();
    }
    return oldValue;
}

```

所以，从上面的源码清晰的看出，在进行并发写操作时：

- ConcurrentHashMap会获取再入锁，以保证数据一致性。Segment本身就是基于ReentrantLock的扩展实现，所以，在并发修改期间，相应Segment是被锁定的。
- 在最初阶段，进行重复性的扫描，以确定相应key值是否已经在数组里面，进而决定是更新还是放置操作，你可以在代码里看到相应的注释。重复扫描、检测冲突是ConcurrentHashMap的常见技巧。
- 我在专栏上一讲介绍HashMap时，提到了可能发生的扩容问题，在ConcurrentHashMap中同样存在。不过有一个明显区别，就是它进行的不是整体的扩容，而是单独对Segment进行扩容。细节就不介绍了。

另外一个Map的size方法同样需要关注，它的实现涉及分离锁的一个副作用。

试想，如果不进行同步，简单的计算所有Segment的总值，可能会因为并发put，导致结果不准确，但是直接锁定所有Segment进行计算，就会变得非常昂贵。其实，分离锁也限制了Map的初始化等操作。

所以，ConcurrentHashMap的实现是通过重试机制（RETRIES\_BEFORE\_LOCK，指定重试次数2），来试图获得可靠值。如果没有监控到发生变化（通过对比Segment.modCount），就直接返回，否则获取锁进行操作。

下面我来对比一下，在Java 8和之后的版本中，ConcurrentHashMap发生了哪些变化呢？

- 总体结构上，它的内部存储变得和我在专栏上一讲介绍的HashMap结构非常相似，同样是大的桶(bucket)数组，然后内部也是一个个所谓的链表结构(bin)，同步的粒度要更细致一些。
- 其内部仍然有Segment定义，但仅仅是为了保证序列化时的兼容性而已，不再有任何结构上的用处。
- 因为不再使用Segment，初始化操作大大简化，修改为lazy-load形式，这样可以有效避免初始开销，解决了老版本很多人抱怨的这一点。
- 数据存储利用volatile来保证可见性。
- 使用CAS等操作，在特定场景进行无锁并发操作。
- 使用Unsafe、LongAdder之类底层手段，进行极端情况的优化。

先看看现在的数据存储内部实现，我们可以发现Key是final的，因为在生命周期中，一个条目的Key发生变化是不可能的；与此同时val，则声明为volatile，以保证可见性。

```

static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    volatile V val;
    volatile Node<K,V> next;
}

```

```
// -  
}
```

我这里就不再介绍get方法和构造函数了，相对比较简单，直接看并发的put是如何实现的。

```
final V putVal(K key, V value, boolean onlyIfAbsent) { if (key == null || value == null) throw new NullPointerException();  
int hash = spread(key.hashCode());  
int binCount = 0;  
for (Node<K,V>[] tab = table; ;) {  
    Node<K,V> f; int n, i, fh; K fk; V fv;  
    if (tab == null || (n = tab.length) == 0)  
        tab = initTable();  
    else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {  
        // 利用CAS去进行无锁编程安全操作。如果bin是空的  
        if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value)))  
            break;  
    }  
    else if ((fh = f.hash) == MOVED)  
        tab = helpTransfer(tab, f);  
    else if (onlyIfAbsent // 不加锁，进行检查  
            && fh == hash  
            && ((fk = f.key) == key || (fk != null && key.equals(fk)))  
            && (fv = f.val) != null)  
        return fv;  
    else {  
        V oldVal = null;  
        synchronized (f) {  
            // 细粒度的同步修改操作...  
        }  
    }  
    // Bin超过阈值，进行树化  
    if (binCount != 0) {  
        if (binCount >= TREEIFY_THRESHOLD)  
            treeifyBin(tab, i);  
        if (oldVal != null)  
            return oldVal;  
        break;  
    }  
}  
addCount(1L, binCount);  
return null;  
}
```

初始化操作实现在initTable里面，这是一个典型的CAS使用场景，利用volatile的sizeCtl作为互斥手段：如果发现竞争性的初始化，就spin在那里，等待条件恢复；否则利用CAS设置排他标志。如果成功则进行初始化；否则重试。

请参考下面代码：

```
private final Node<K,V>[] initTable() {  
    Node<K,V>[] tab; int sc;  
    while ((tab = table) == null || tab.length == 0) {  
        // 如果发现冲突，进行spin等待  
        if ((sc = sizeCtl) < 0)  
            Thread.yield();  
        // CAS成功返回true，则进入真正的初始化逻辑  
        else if (U.compareAndSetInt(this, SIZECTL, sc, -1)) {  
            try {  
                if ((tab = table) == null || tab.length == 0) {  
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;  
                    @SuppressWarnings("unchecked")  
                    Node<K,V>[] nt = (Node<K,V>[])(new Node<?,?>[n]);  
                    table = tab = nt;  
                    sc = n - (n >>> 2);  
                }  
            } finally {  
                sizeCtl = sc;  
            }  
            break;  
        }  
    }  
    return tab;  
}
```

当bin为空时，同样是没有必要锁定，也是以CAS操作去放置。

你有没有注意到，在同步逻辑上，它使用的是synchronized，而不是通常建议的ReentrantLock之类，这是为什么呢？现代JDK中，synchronized已经被不断优化，可以不再过分

担心性能差异，另外，相比于ReentrantLock，它可以减少内存消耗，这是个非常大的优势。

与此同时，更多细节实现通过使用Unsafe进行了优化，例如tabAt就是直接利用getObjectAcquire，避免间接调用的开销。

```
static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {
    return (Node<K,V>)U.getObjectAcquire(tab, ((long)i << ASHIFT) + ABASE);
}
```

再看看，现在是如何实现size操作的。[阅读代码](#)你会发现，真正的逻辑是在sumCount方法中，那么sumCount做了什么呢？

```
final long sumCount() {
    CounterCell[] as = counterCells; CounterCell a;
    long sum = baseCount;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

我们发现，虽然思路仍然和以前类似，都是分而治之的进行计数，然后求和处理，但实现却基于一个奇怪的CounterCell。难道它的数值，就更加准确吗？数据一致性是怎么保证的？

```
static final class CounterCell {
    volatile long value;
    CounterCell(long x) { value = x; }
}
```

其实，对于CounterCell的操作，是基于java.util.concurrent.atomic.LongAdder进行的，是一种JVM利用空间换取更高效率的方法，利用了[Striped64](#)内部的复杂逻辑。这个东西非常小众，大多数情况下，建议还是使用AtomicLong，足以满足绝大部分应用的性能需求。

今天我从线程安全问题开始，概念性的总结了基本容器工具，分析了早期同步容器的问题，进而分析了Java 7和Java 8中ConcurrentHashMap是如何设计实现的，希望ConcurrentHashMap的并发技巧对你日常开发可以有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？留一个思考题给你，在产品代码中，有没有典型的场景需要使用类似ConcurrentHashMap这样的并发容器呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“[请朋友读](#)”，把今天的题目分享给好友，或许你能帮到他。



徐金泽

2018-05-26

需要注意的一点是，1.8以后的锁的颗粒度，是加在链表头上的，这个是个思路上的突破。

作者回复

2018-05-28

是的

雷霹雳的爸爸

2018-05-26

今天这个纯粹知识盲点，纯赞，源码也得不停看

Sean

最近用ConcurrentHashMap的场景是，由于系统是一个公共服务，全程异步处理。最后一环节需要http rest主动响应接入系统，于是为了定制化需求，利用netty写了一版异步http client。  
真在缓存tcp连接时用到了。

看到下面有位朋友说起了自旋锁和偏向锁。

自旋锁个人理解的是cas的一种应用方式。并发包中的原子类是典型的应用。

偏向锁个人理解的是获取锁的优化。在ReentrantLock中用于实现已获取锁的线程重入问题。

不知道理解的是否有误差。欢迎指正探讨。谢谢

作者回复

正确，互相交错

偏向锁，侧重是低竞争场景的优化，去掉可能不必要的同步

j.c.

2018-05-28

期待unsafe和cas的文章

明翼

2018-05-26

1.7

put加锁

通过分段加锁segment，一个hashmap里有若干个segment，每个segment里有若干个桶，桶里存放K-V形式的链表，put数据时通过key哈希得到该元素要添加到的segment，然后对segment进行加锁，然后在哈希，计算得到给元素要添加到的桶，然后遍历桶中的链表，替换或新增节点到桶中

size

分段计算两次，两次结果相同则返回，否则对所以段加锁重新计算

1.8

put CAS 加锁

1.8中不依赖与segment加锁，segment数量与桶数量一致；

首先判断容器是否为空，为空则进行初始化利用volatile的sizeCtl作为互斥手段，如果发现竞争性的初始化，就暂停在那里，等待条件恢复，否则利用CAS设置排他标志

(U.compareAndSwapInt(this, SIZECTL, sc, -1)) 否则重试。

对key hash计算得到该key存放的桶位置，判断该桶是否为空，为空则利用CAS设置新节点

否则使用synchronize加锁，遍历桶中数据，替换或新增节点到桶中

最后判断是否需要转为红黑树，转换之前判断是否需要扩容

size

利用LongAdd累加计算

coder王

2018-07-04

您说的synchronized被改进很多很多了，那么在我们平常使用中，就用这个synchronized完成一些同步操作是不是OK？♦♦

作者回复

通常的是，前提是JDK版本需要新一点

mongo

2018-05-28

请教老师：putVal方法的第二个if分支，为什么要用tabAt？我的认识里直接数组下标寻址tab[i=(n-1) & hash]也是一个原子操作，不是吗？tabAt里面的getObjectVolatile () 方法跟直接用数组下标tab[i=(n-1) & hash]寻址有什么区别？

作者回复

这个有volatile load语义

hansc

2018-05-26

从1.5有并发包，到1.6对synchronized的改进，到1.7的并发map的分段锁，再到1.8的cas+synchronized。

作者回复

嗯嗯，有的历史我也未必知道

曹铮

2018-05-28

这期内容太难，分寸不好把握

看的concurrent hashmap源码感觉挺困难，网上的博文帮助也不大，尤其是扩容这部分（似乎文章中没提）

求问大佬有什么窍门，或者有什么启发性的paper或文章？

可以泛化成，长期对lock free实现多个状态修改的问题比较困惑，希望得到启发

作者回复

本文尽量梳理了相对比较容易理解的部分；扩容细节我觉得是个加分项，不是每个人都会在乎那么深入；窍门，可以考虑画图辅助理解，我是比较笨的类型，除了死磕，不会太多窍门.....

Xg huang

2018-05-26

这里有个地方想跟老师交流一下想法，从文中“所以，ConcurrentHashMap的实现是通过重试机制（RETRIES\_BEFORE\_LOCK，指定重试次数 2），来试图获得可靠值。如果没有监控到发生重试（通过对比 Segment.modCount），就直接返回，否则获取锁进行操作。”可以看出，在高并发的情况下，“size”方法只是返回“近似值”，而我的问题是：既然只是一个近似值，为什么要用“重试，分段锁”的复杂方法去计算这个值？直接在不加锁的情况下返回segment的size岂不是很简单？我能理解jdk开发者想尽一切努力在高性能地返回最精确的数据，但这个“精确”度无法量化啊，对于调用方来说，这个值依然是不可靠的。所以，在我看来，这种做法收益很小（可能是我也比较懒吧），或者有些设计上的要点我没有领悟出来，希望老师指点一下。

作者回复

这个是在代价可接受情况下，尽量准确，就像含金量90%和99.9%，99.999%，还是有区别的，虽然不是百分百

Levy

2018-06-08

老师你好，tabAt里面的getObjectVolatile () 方法跟直接用数组下标tab[i=(n-1) & hash]寻址有什么区别，这个我也不懂，volatile不是已经保证内存可见性吗？

作者回复

volatile保证的是数组，不是数组元素

Hesher

2018-05-30

2018-05-31

并发包用的很少，这一节内容的前置知识比较多，对于使用经验少的人来说貌似是有点难了。问题很好，正好可以见识一下各种使用场景，不过留言大部分是针对内容的难点提问，而真正回答问题的还没有出现。

作者回复

后面并发部分会详细分析

2018-05-28

行者

老师麻烦讲解自旋锁，偏向锁的特点和区别吧，一直不太清楚。  
作者回复

好，后面有章节

虞飞

老师在课程里讲到同步包装类比较低效，不太适合高并发的场景。那想请教一下老师，在list接口的实现类中，在高并发的场景下，选择哪种实现类比较好？因为ArrayList是线程不安全的，同步包装类又很低效。CopyonwriteArrayList又是以快照的形式来实现的，在频繁写入数据的时候，其实也很低效，那这个类型该怎么选择比较好？

作者回复

目前并发list好像就那一个，我觉得不必拘泥于list，不还有queue之类，看场景需要的真是list吗

mongo

请教老师：putVal方法中，什么情况下会进入else if ((fh=f.hash) == MOVED) 分支？是进行扩容的时候吗？nextTable是做什么用的？  
作者回复

我理解是的，判断是个ForwardingNode，resize正在进行；  
nexttable是扩容时的临时过渡

hansc

osgi环境中多线程启动bundle，扫描注解配置缓存起来。

t

对于我这种菜鸟来说，应该来一期讲讲volatile◆◆

Answer

Unsafe?

shawn

老师，什么只有bin为空的时候才使用cas，其他地方用synchronized 呢？

bazindes

老师的内容讲的丰富深入浅出，希望提高一下朗读人的要求吧。每节课都感觉有读错的。英文读不准就不说了，互斥读成互拆听的实在是别扭。  
作者回复

哈，抱歉，我反馈一下，主播也辛苦，不一定是职业码农

宁宁

在构造的时候，Segment 的数量由所谓的 concurrencyLevel 决定，默认是 16！并发数不是越多越好吗？

作者回复

都是有代价的，取个折衷

李飞

好几个词感觉第一次见，大哭◆◆

作者回复

嗯...哪几个？回头根据反馈补充下

牛在天上飞

老师能解释下可重入锁吗？

作者回复

马上就有章节介绍了

Miaozhe

杨老师，看到ConcurrentHashMap中有定义N CPU,想问问跟CPU什么关系？

作者回复

那个是Runtime.availableProcessors

George

文中说的UNSAFE是什么意思？

作者回复

JDK内部的一个基础类sun.misc.Unsafe，如果公共api能解决的问题，不建议使用它，如果是做底层开发可能会用到

Volong

第一个示例代码报这个错：'< >' cannot be used with anonymous classes

Leiy

我感觉jdk8就相当于把segment分段锁更细粒度了，每个数组元素就是原来一个segment，那并发度就由原来segment数变为数组长度？而且用到了cas乐观锁，所以能支持更高的并发，不知道我这种理解对吗？如果对的话，我就在想，为什么并发大神之前没想到这种，哈哈◆◆。恳请指正。谢谢

作者回复

基本正确，cas只用在部分场景：  
事后看容易啊，说比做容易。◆◆

极客cq

不从实现上只从图形结构上看，ConcurrentHashMap和HashMap一样，不过是将buckets换成了segment，然后加锁方式从整map，下沉到segment (buckets) 上，这样简单理解正确么？

作者回复

有一定道理，但seg和buck范围不完全对等

Kyle

之前用JavaFX做一个客户端IM工具的时候，我将拉来的未被读取的用户聊天信息用ConcurrentHashMap存储（同时异步存储到Sqlite），Key存放userId，Value放未读取的聊天消息列表。  
因为我考虑到存消息和读消息是由两个线程并发处理的，这两个线程共同操作一个ConcurrentHashMap。可能是我没处理好，最后直到我离职了还有消息重复、乱序的问题。请问我这种应用场景有什么问题吗？

Ethan

请问老师以后会不会有讲线程中断的不同处理呢？这一块一直对我来讲都比较抽象而且不好测试

作者回复

会有，我也记录下

lorancechen

rpc调用内部，客户端会记录每次请求的unique id，用于匹配返回的数据应该响应哪个请求。高并发情况下，应该使用concurrenthashmap做这种id到回调的记录。

胖

如果多线程中已经在上层代码使用了读写锁进行访问控制，底层集合是否就可以使用HashMap，而没有必要使用线程安全的容器了？

作者回复

自己做同步当然也是个选择

李军

可不可以用AI让JVM更智能？

作者回复

需要更具体一些，没有一劳永逸的方法，比如，有用机器学习调优jvm运行参数的；  
jvm比较运行场景太多，SE既可以跑在嵌入式，也可以到高性能服务器，缩小范围才好进行

灰飞灰猪不会灰飞，烟灭

jdk7和8的区别感觉就是加锁不一样了，其他的没看懂。

老师，synchronized和lock是不是都是在类字节码中携带自旋锁和偏向锁啊？  
他两底层区别是啥呢？我知道lock里面维护了一个双向链表

作者回复

我列出来的区别不止吧；  
存在锁升级的问题，后面有章节介绍

两只◆◆

有些本地缓存就是基于它实现的。

CUZZ.

有点不懂。。

作者回复

请问具体哪个方面？并发工具后面会有专门分析

2018-05-29

2018-05-29

2018-05-29

2018-05-28

2018-05-28

2018-05-28

2018-05-27

2018-05-28

2018-05-26

2018-05-28

2018-05-26

2018-05-28

2018-05-26

2018-05-26

2018-05-28





## 第11讲 | Java提供了哪些IO方式？NIO如何实现多路复用？

2018-05-29 杨晓峰



第11讲 | Java提供了哪些IO方式？NIO如何实现多路复用？  
杨晓峰  
- 00:00 / 11:41

IO一直是软件开发中的核心部分之一，伴随着海量数据增长和分布式系统的发展，IO扩展能力愈发重要。幸运的是，Java平台IO机制经过不断完善，虽然在某些方面仍有不足，但已经在实践中证明了其构建高扩展性应用的能力。

今天我要问你的问题是，[Java提供了哪些IO方式？NIO如何实现多路复用？](#)

## 典型回答

Java IO方式有很多种，基于不同的IO抽象模型和交互方式，可以进行简单区分。

首先，传统的java.io包，它基于流模型实现，提供了我们最熟知的一些IO功能，比如File抽象、输入输出流等。交互方式是同步、阻塞的方式，也就是说，在读取输入流或者写入输出流时，在读、写动作完成之前，线程会一直阻塞在那里，它们之间的调用是可靠的线性顺序。

java.io包的好处是代码比较简单、直观，缺点则是IO效率和扩展性存在局限性，容易成为应用性能的瓶颈。

很多时候，人们也把java.net下面提供的部分网络API，比如Socket、ServerSocket、HttpURLConnection也归类到同步阻塞IO类库，因为网络通信同样是IO行为。

第二，在Java 1.4中引入了NIO框架（java.nio包），提供了Channel、Selector、Buffer等新的抽象，可以构建多路复用的、同步非阻塞IO程序，同时提供了更接近操作系统底层的高性能数据操作方式。

第三，在Java 7中，NIO有了进一步的改进，也就是NIO 2，引入了异步非阻塞IO方式，也有很多人叫它AIO（Asynchronous IO）。异步IO操作基于事件和回调机制，可以简单理解为，应用操作直接返回，而不会阻塞在那里，当后台处理完成，操作系统会通知相应线程进行后续工作。

## 考点分析

我上面列出的回答是基于一种常见分类方式，即所谓的BIO、NIO、NIO 2（AIO）。

在实际面试中，从传统IO到NIO、NIO 2，其中有很多地方可以扩展开来，考察点涉及方方面面，比如：

- 基础API功能与设计，InputStream/OutputStream和Reader/Writer的关系和区别。
- NIO、NIO 2的基本组成。
- 给定场景，分别用不同模型实现，分析BIO、NIO等模式的设计和实现原理。
- NIO提供的高性能数据操作方式是基于什么原理，如何使用？
- 或者，从开发者的角度来看，你觉得NIO自身实现存在哪些问题？有什么改进的想法吗？

IO的内容比较多，专栏一讲很难能够说清楚。IO不仅仅是多路复用，NIO 2也不仅仅是异步IO，尤其是数据操作部分，会在专栏下一讲详细分析。

## 知识扩展

首先，需要澄清一些基本概念：

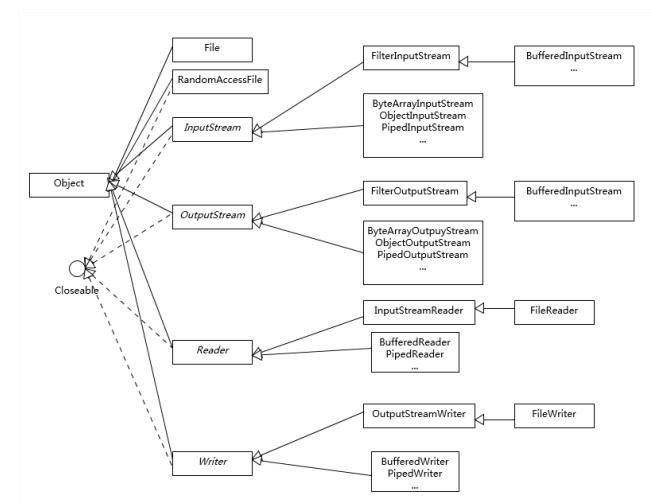
- 区分同步或异步（synchronous/asynchronous）。简单来说，同步是一种可靠的有序运行机制，当我们进行同步操作时，后续的任务是等待当前调用返回，才会进行下一步；而异步则相反，其他任务不需要等待当前调用返回，通常依靠事件、回调等机制来实现任务间次序关系。
- 区分阻塞与非阻塞（blocking/non-blocking）。在进行阻塞操作时，当前线程会处于阻塞状态，无法从事其他任务，只有当条件就绪才能继续，比如ServerSocket新连接建立完毕，或数据读取、写入操作完成；而非阻塞则是不管IO操作是否结束，直接返回，相应操作在后台继续处理。

不能一概而论认为同步或阻塞就是低效，具体还要看应用和系统特征。

对于Java.io，我们都非常熟悉，我这里就从总体上进行一下总结，如果需要学习更加具体的操作，你可以通过[教程](#)等途径完成。总体上，我认为你至少需要理解：

- IO不仅仅是对文件的操作，网络编程中，比如Socket通信，都是典型的IO操作目标。
- 输入流、输出流（InputStream/OutputStream）是用于读取或写入字节的，例如操作图片文件。
- 而Reader/Writer则是用于操作字符，增加了字符编解码等功能，适用于类似从文件中读取或者写入文本信息。本质上计算机操作的都是字节，不管是网络通信还是文件读取，Reader/Writer相当于构建了应用逻辑和原始数据之间的桥梁。
- BufferedOutputStream等带缓冲区的实现，可以避免频繁的磁盘读写，进而提高IO处理效率。这种设计利用了缓冲区，将批量数据进行一次操作，但在使用中千万别忘了flush。
- 参考下面这张类图，很多IO工具类都实现了Closeable接口，因为需要进行资源的释放。比如，打开FileInputStream，它就会获取相应的文件描述符（FileDescriptor），需要利用try-with-resources、try-finally等机制保证FileInputStream被明确关闭，进而相应文件描述符也会失效，否则将导致资源无法被释放。利用专栏前面的内容提到的Cleaner或finalize机制作为资源释放的最后把关，也是必要的。

下面是我整理的一个简化版的类图，阐述了日常开发应用较多的类型和结构关系。



## 1. Java NIO概览

首先，熟悉一下NIO的主要组成部分：

- Buffer，高效的数据容器，除了布尔类型，所有原始数据类型都有相应的Buffer实现。
- Channel，类似在Linux之类操作系统上看到的文件描述符，是NIO中被用来支持批量式IO操作的一种抽象。  
File或者Socket，通常被认为是比较高层次的抽象，而Channel则是更加操作系统底层的一种抽象，这也使得NIO得以充分利用现代操作系统底层机制，获得特定场景的性能优化，例如，DMA（Direct Memory Access）等。不同层次的抽象是相互关联的，我们可以通过Socket获取Channel，反之亦然。
- Selector，是NIO实现多路复用的基础，它提供了一种高效的机制，可以检测到注册在Selector上的多个Channel中，是否有Channel处于就绪状态，进而实现了单线程对多Channel的高效管理。

Selector同样是基于底层操作系统机制，不同模式、不同版本都存在区别，例如，在最新的代码库里，相关实现如下：

Linux上依赖于epoll (<http://hg.openjdk.java.net/jdk/jdk/file/d8327f838b88/src/java.base/linux/classes/sun/nio/ch/EPollSelectorImpl.java>)。

Windows上NIO2（AIO）模式则是依赖于iocp (<http://hg.openjdk.java.net/jdk/jdk/file/d8327f838b88/src/java.base/windows/classes/sun/nio/ch/Iocp.java>)。

- Charset，提供Unicode字符串定义，NIO也提供了相应的编解码器等，例如，通过下面的方式进行字符串到ByteBuffer的转换：

```
Charset.defaultCharset().encode("Hello world!");
```

## 2.NIO能解决什么问题？

下面我通过一个典型场景，来分析为什么需要NIO，为什么需要多路复用。设想，我们需要实现一个服务器应用，只简单要求能够同时服务多个客户端请求即可。

使用java.io和java.net中的同步、阻塞式API，可以简单实现。

```
public class DemoServer extends Thread {
    private ServerSocket serverSocket;
    public int getPort() {
        return serverSocket.getLocalPort();
    }
    public void run() {
        try {
```

```

serverSocket = new ServerSocket(0);
while (true) {
    Socket socket = serverSocket.accept();
    RequestHandler requestHandler = new RequestHandler(socket);
    requestHandler.start();
}
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (serverSocket != null) {
        try {
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}

public static void main(String[] args) throws IOException {
    DemoServer server = new DemoServer();
    server.start();
    try (Socket client = new Socket(InetAddress.getLocalHost(), server.getPort())) {
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
        bufferedReader.lines().forEach(s -> System.out.println(s));
    }
}
}

// 简化实现，不做读取，直接发送字符串
class RequestHandler extends Thread {
    private Socket socket;
    RequestHandler(Socket socket) {
        this.socket = socket;
    }
    @Override
    public void run() {
        try (PrintWriter out = new PrintWriter(socket.getOutputStream())) {
            out.println("Hello world!");
            out.flush();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}

```

其实现要点是：

- 服务器端启动ServerSocket，端口0表示自动绑定一个空闲端口。
- 调用accept方法，阻塞等待客户端连接。
- 利用Socket模拟了一个简单的客户端，只进行连接、读取、打印。
- 当连接建立后，启动一个单独线程负责回复客户端请求。

这样，一个简单的Socket服务器就被实现了。

思考一下，这个解决方案在扩展性方面，可能存在什么潜在问题呢？

大家知道Java语言目前的线程实现是比较重量级的，启动或者销毁一个线程是有明显开销的，每个线程都有单独的线程栈等结构，需要占用非常明显的内存，所以，每一个Client启动一个线程似乎都有些浪费。

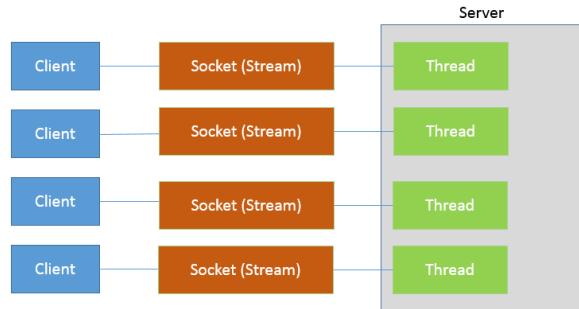
那么，稍微修正一下这个问题，我们引入线程池机制来避免浪费。

```

serverSocket = new ServerSocket(0);
executor = Executors.newFixedThreadPool(8);
while (true) {
    Socket socket = serverSocket.accept();
    RequestHandler requestHandler = new RequestHandler(socket);
    executor.execute(requestHandler);
}

```

这样做似乎好了很多，通过一个固定大小的线程池，来负责管理所有线程，避免频繁创建、销毁线程的开销，这是我们构建并发服务的典型方式。这种工作方式，可以参考下图来理解。



如果连接数并不是非常多，只有最多几百个连接的普通应用，这种模式往往可以工作的很好。但是，如果连接数量急剧上升，这种实现方式就无法很好地工作了，因为线程上下文切换开销会在高并发时变得很明显，这是同步阻塞方式的低扩展性劣势。

NIO引入的多路复用机制，提供了另外一种思路，请参考我下面提供的新的版本。

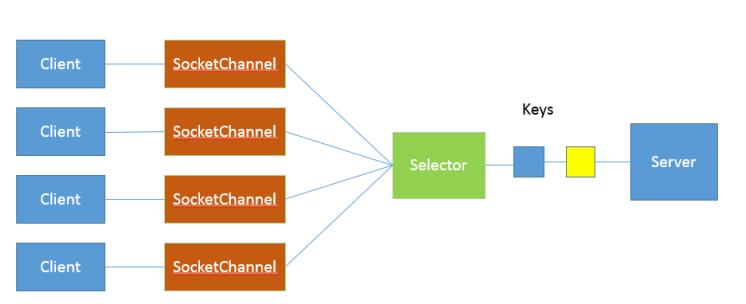
```

public class NIOServer extends Thread {
    public void run() {
        try (Selector selector = Selector.open();
            ServerSocketChannel serverSocket = ServerSocketChannel.open()) { // 创建Selector和Channel
            serverSocket.bind(new InetSocketAddress(InetAddress.getLocalHost(), 8888));
            serverSocket.configureBlocking(false);
            // 注册到Selector，并注明关注点
            serverSocket.register(selector, SelectionKey.OP_ACCEPT);
            while (true) {
                selector.select(); // 阻塞等待就绪的Channel，这是关键点之一
                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                Iterator<SelectionKey> iter = selectedKeys.iterator();
                while (iter.hasNext()) {
                    SelectionKey key = iter.next();
                    // 生产系统中一般会额外进行就绪状态检查
                    sayHelloWorld((ServerSocketChannel) key.channel());
                    iter.remove();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private void sayHelloWorld(ServerSocketChannel server) throws IOException {
        try (SocketChannel client = server.accept()) {
            client.write(Charset.defaultCharset().encode("Hello world!"));
        }
    }
}
// 省略了与前面类似的main
}
  
```

这个非常精简的样例掀开了NIO多路复用的面纱，我们可以分析下主要步骤和元素：

- 首先，通过`Selector.open()`创建一个`Selector`，作为类似调度员的角色。
- 然后，创建一个`ServerSocketChannel`，并且向`Selector`注册，通过指定`SelectionKey.OP_ACCEPT`，告诉调度员，它关注的是新的连接请求。
- 注意，为什么我们要明确配置非阻塞模式呢？这是因为阻塞模式下，注册操作是不允许的，会抛出`IllegalBlockingModeException`异常。
- `Selector`阻塞在`select`操作，当有`Channel`发生接入请求，就会被唤醒。
- 在`sayHelloWorld`方法中，通过`SocketChannel`和`ByteBuffer`进行数据操作，在本例中是发送了一段字符串。

可以看到，在前面两个样例中，IO都是同步阻塞模式，所以需要多线程以实现多任务处理。而NIO则是利用了单线程轮询事件的机制，通过高效地定位就绪的`Channel`，来决定做什么。仅仅`select`阶段是阻塞的，可以有效避免大量客户端连接时，频繁线程切换带来的问题，应用的扩展能力有了非常大的提高。下面这张图对这种实现思路进行了形象地说明。



在Java 7引入的NIO 2中，又增添了一种额外的异步IO模式，利用事件和回调，处理Accept、Read等操作。AIO实现看起来是类似这样子：

```

AsynchronousServerSocketChannel serverSock = AsynchronousServerSocketChannel.open().bind(sockAddr);
serverSock.accept(serverSock, new CompletionHandler<>() { //为异步操作指定CompletionHandler回调函数
    @Override
    public void completed(AsynchronousSocketChannel sockChannel, AsynchronousServerSocketChannel serverSock) {
        serverSock.accept(serverSock, this);
        // 另外一个 write (sock, CompletionHandler<>)
        sayHelloWorld(sockChannel, Charset.defaultCharset().encode
            ("Hello World!"));
    }
    // 省略其他路径处理方法...
});
  
```

鉴于其编程要素（如Future、CompletionHandler等），我们还没有进行准备工作，为避免理解困难，我会在专栏后面相关概念补充后的再进行介绍，尤其是Reactor、Proactor模式等方面将在Netty主题一起分析，这里我先进行概念性的对比：

- 基本抽象很相似，AsynchronousServerSocketChannel对应于上面例子中的ServerSocketChannel；AsynchronousSocketChannel则对应SocketChannel。
- 业务逻辑的关键在于，通过指定CompletionHandler回调接口，在accept/read/write等关键节点，通过事件机制调用，这是非常不同的一种编程思路。

今天我初步对Java提供的IO机制进行了介绍，概要地分析了传统同步IO和NIO的主要组成，并根据典型场景，通过不同的IO模式进行了实现与拆解。专栏下一讲，我还将继续分析Java IO的主题。

-课一练

关于今天我们讨论的题目你做到心中有数了吗？留一道思考题给你，NIO多路复用的局限性是什么呢？你遇到过相关的问题吗？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



I am a psycho

2018-05-29

由于nio实际上是非阻塞io，是一个线程在同步的进行事件处理。当一组channel处理完毕以后，去检查有没有可以处理的channel。这也就是同步+非阻塞。同步，指每个准备好的channel处理是依次进行的。非阻塞，是指线程不会傻傻的等待谁。只有当channel准备好后，才会进行。那么就会有这样一个问题：当每个channel所进行的都是耗时操作时，由于是同步操作，就会积压很多channel任务，从而完成影响。那么就需要对nio进行类似负载均衡的操作，如用线程池去进行管理读写，将channel分给其他的线程去执行，这样既充分利用了每一个线程，

又不至于都堆积在一个线程中，等待执行。杨老师，不知道上述理解是否正确？

雷霹雳的爸爸

批评NIO确实要小心，我觉得主要是三方面，首先是从写BIO过来的同学，需要有一个巨大的观念上的转变，要清楚网络就是并非时刻可读可写，我们用NIO就是在认真的面对这个问题，别把channel当流往死里用，没读出来写不进去的时候，就是该考虑让读线程资源了，第二点是NIO在不同的平台上的实现方式是不一样的，如果你工作用电脑是win，生产是linux，那么建议直接在linux上调试和测试，第三点，概念上的，理解了会在各方面都有益处，NIO在IO操作本身上还是阻塞的，也就是他还是同步IO，AIO读写行为的回调才是异步IO，而这个真正实现，还是看系统底层的，写完之后，我觉得我这一二三有点添油的嫌疑

作者回复

不错，不过，在非常有必要之前，不见得都要底层，毕竟各种抽象，都是为特定领域工程师准备的，JMM等抽象都是为了大家有个清晰的、不同层面的高效交流

逐梦之音

IO的调用可以分为三大块，请求调用，逻辑处理，响应返回处理。常规的BIO在这三个阶段会串行的阻塞的。NIO其实可以理解为将这三个阶段尽可能的去阻塞或者减少阻塞。看了上面的例子，NIO的服务器端在接受客户端请求的时候，是单线程执行的，而BIO是多线程处理的。但是不管咋的，他们服务器端处理具体的客户业务逻辑是都要用多线程的吧？

灰飞灰猪不会灰飞，烟灭

老师 注册管道到select上，应该用队列实现的吧？

开启一个线程大概需要多少内存开销呢，我记得数据库连接大概2M

作者回复

线程看定义stack大小等，32、64位都不一样

Chan

B和N通常是针对数据是否就绪的处理方式来  
sync和async是对阻塞进行更深一层次的阐释，区别在于数据拷贝由用户线程完成还是内核完成，讨论范围一定是两个线程及以上了。

同步阻塞，从数据是否准备就绪到数据拷贝都是由用户线程完成

同步非阻塞，数据是否准备就绪由内核判断，数据拷贝还是用户线程完成

异步非阻塞，数据是否准备就绪到数据拷贝都是内核来完成

所以真正的异步IO一定是非阻塞的。

多路复用IO即使有Reactor通知用户线程也是同步IO范畴，因为数据拷贝期间仍然是用户线程完成。

所以假如我们没有内核支持数据拷贝的情况下，讨论的非阻塞并不是彻底的非阻塞，也就没有引入sync和async讨论的必要了

不知道这样理解是否正确

zjh

看nio代码部分，请求接受和处理都是一个线程在做。这样的话，如果有多个请求过来都是按顺序处理吧，其中一个处理时间比较耗时的话那所有请求都不卡住了吗？如果把nio的处理部分也改成多线程会有什么问题吗

作者回复

这种情况需要考虑耗时操作并发处理，再说处理是费cpu，还是重io，需要不同处理；如果耗时操作非常多，就不符合这种模型的适用场景

残月@诗雨

杨老师，有个问题一直不太明白：BufferedInputStream和普通的InputStream直接read到一个缓冲数组这两种方式有什么区别？

作者回复

我理解是bufferedIS是内部预读，所以两个buffer的意义不一样，前面是减少磁盘之类操作

明翼

看完之后还是不了解nio，感觉看起来越来越吃力了，大半天都啃不了一篇，好多东西都不熟悉，还要自己查资料去了解然后再回过来看，，，老师最好给些学习的资料让我们能找到，就这一篇感觉根本不够

Chan

忘记回答问题了。所以对于多路复用IO，当出现有的IO请求在数据拷贝阶段，会出现由于资源类型过份庞大而导致线程长期阻塞，最后造成性能瓶颈的情况

作者回复

try with resource就相当于在finally里close，一直挂着是因为server在伺服

扁扁圆圆

这里Nio的Selector只注册了一个sever channel，这没有实现多路复用吧，多路复用不是注册了多个channel，处理就绪的吗？而且处理客户端请求也是在同线程内，这还不如上面给的Bio解决方案吧

作者回复

这是简化的例子，少占篇幅

aiwen

到底啥是多路复用？一个线程管理多个链接就是多路复用？

lorancechen

2018-05-29

2018-05-29

2018-05-29

2018-05-29

2018-06-16

2018-05-31

2018-06-06

2018-05-29

2018-05-29

2018-07-05

2018-06-16

2018-06-16

2018-06-05

2018-06-06

2018-06-03

2018-06-02

2018-05-31

我也自己写过一个基于nio2的网络程序，觉得配合future写起来很舒服。  
仓库地址：<https://github.com/LoranceChen/RxSocket> 欢迎相互交流开发经验~

记得在netty中，有一个搁置的netty5.x项目被废弃掉了，原因有一点官方说是性能提升不明显，这是可以理解的，因为linux下是基于epoll，本质还是select操作。

听了课程之后，有一点印象比较深刻，select模式是使用一个线程做监听，而bio每次来一个链接都要做线程切换，所以节省的时间在线程切换上，当然如果是c/c++实现，原理也是一样的。  
想问一个一直困惑的问题，select内部如何实现的呢？  
个人猜测：不考虑内核、应用层的部分，单纯从代码角度考虑，我猜测，当select开始工作时，有一个定时器，比如每10ms去检查一下网络缓冲区中是否有tcp的链接请求包，然后把这些包筛选出来，作为一个集合（即代码中的迭代器）填入java select类的一个集合成员中，然后唤醒select线程，做一个while遍历处理链接请求，这样一次线程调度就可以处理10ms内的所有链接。与bio比，节省的时间在线程上下文切换上。不知道这样理解对不对。  
另外，也希望我能出一个课程，按照上面这种理解底层的方式，讲述select（因为我平常工作在linux机器，所以对select epoll比较感兴趣）如何处理read、write操作的。谢谢~

作者回复

2018-06-01

坦白说，内核epoll之类实现细节目前我的理解也有限

RoverYe

2018-05-30

nio不适合数据量太大交互的场景

ykkk88

2018-05-29

这个nio看起来还是单线程在处理，如果放到多线程池中处理和bio加线程池有啥区别呢

L.B.Q.Y

2018-05-29

NIO多路复用模式，如果对应事件的处理比较耗时，是不是会导致后续事件的响应出现延迟。

作者回复

2018-05-29

所以我理解，适用于大量请求大小有限的场景。（主任务）单线程模型，比如nodejs都有类似情况，

Forrest

2018-05-29

使用线程池可以很好的解决线程复用，避免线程创建带来的开销，效果也很好，一个问题想请教下，当线程池无法满足需要时可以用什么方式解决？

作者回复

2018-05-29

没有看明白问题，抱歉

张凯江

2018-07-18

cpu运算密集型应用。node得诟病

清风

2018-07-03

我认为这个nio 2基于事件编程就跟swing 编程添加监听事件一样，有事件发生了，执行回调函数。不知道理解是否准确。同时针对nio的采用线程不断地轮询，对客户多量大后会有响应延迟，并发数量有限。同时也想知道tomcat是怎么样通过nio 来解决着问题的。一直没有时间看一下他的源码，枉费我工作这么多年。老师可以写一篇这个nio和tomcat的文章！

Allen

2018-06-30

希望能听到更多原理性的东西，而不是在网上能搜到的样例代码

wenxueliu

2018-06-27

哈哈，如果能结合操作系统之中io模型讲也许更好，顺便分析下select和poll和epoll的实现机制，那么就比较完美了。不过话说这样，就这个主题就可以当一门课啦。推荐netty权威指南

Miaozhe

2018-06-06

NIO 2是借助AsynchronousChannelGroup,实现多Channel方案，有具备异步功能，解决NIO 1单线程多通路问题。

不过异步有两种方案：

1.New—一个CompletionHandler对象，在重写方法中处理，Write和Read。

2.通过concurrent.Future对象，建模一个挂起操作，之后可以通过Get获取socketChannel处理Write和Read。

想问问杨老师，两种方式各有什么优劣？

Miaozhe

2018-06-05

问个问题，看到你写的样例中，几处try O.小括号里面写了一些逻辑，这种写法跟放在{}里面有啥区别？望专家回复一下。

作者回复

2018-06-06

前面介绍过，try-with-resources，自动close

yxw

2018-06-03

java.nio.selector主要的问题是效率，当并发连接数达到数万甚至数十万的时候，单线程的selector会是一个瓶颈；另一个问题就是再线上运行过程中经常出现cpu占用100%的情况，原因是由于selector依赖的操作系统底层机制bug 导致的selector假死，需要程序重建selector来解决，这个问题在jdk中似乎并没有很好的解决，netty成为了线上更加可靠的网络框架。不知理解的是否正确，请老师指教。

作者回复

2018-06-05

嗯，有局限性：那个epoll的bug应该在8里修了，netty的改进不止那些，它为了性能改了很多底层，后面会介绍，好多算是hack；另外nio的目的是通用场景的基础API，和终端应用有个距离，核心类库很多都是如此定位，netty这种开源框架更贴近用户场景

lorancechen

2018-05-31

还有一个问题请教，select在单线程下处理监听任务是否会成为瓶颈？能否通过创建多个select实例，并发监听socket事件呢？

作者回复

2018-05-31

Doug Lea曾经推荐过多个Selector，也就是多个reactor，如果你是这意思

Jerry银银

2018-05-30

我是来找茬的◆◆：file 应该的读「fail」

---

请教一个概念性的问题：线程池模式和数据库连接池模式，是不是就是大家通常所说的对象池模式？

funnyx

这个模式和go中的goroutine, channel, select很像，值得研究。

2018-05-29





## 第12讲 | Java有几种文件拷贝方式？哪一种最高效？

2018-05-31 杨晓峰



第12讲 | Java有几种文件拷贝方式？哪一种最高效？

杨晓峰

- 00:00 / 12:38

我在专栏上一讲提到，NIO不止是多路复用，NIO 2也不只是异步IO，今天我们来看看Java IO体系中，其他不可忽略的部分。

今天我要问你的是问题，[Java有几种文件拷贝方式？哪一种最高效？](#)

典型回答

Java有多种比较典型的文件拷贝实现方式，比如：

利用java.io类库，直接为源文件构建一个FileInputStream读取，然后再为目标文件构建一个OutputStream，完成写入工作。

```
public static void copyFileByStream(File source, File dest) throws
    IOException {
    try (InputStream is = new FileInputStream(source);
        OutputStream os = new FileOutputStream(dest)) {
        byte[] buffer = new byte[1024];
        int length;
        while ((length = is.read(buffer)) > 0) {
            os.write(buffer, 0, length);
        }
    }
}
```

或者，利用java.nio类库提供的transferTo或transferFrom方法实现。

```
public static void copyFileByChannel(File source, File dest) throws
    IOException {
    try (FileChannel sourceChannel = new FileInputStream(source)
        .getChannel();
        FileChannel targetChannel = new FileOutputStream(dest).getChannel()) {
        for (long count = sourceChannel.size(); count > 0;) {
            long transferred = sourceChannel.transferTo(
                sourceChannel.position(), count, targetChannel);
            sourceChannel.position(sourceChannel.position() + transferred);
            count -= transferred;
        }
    }
}
```

当然，Java标准类库本身已经提供了几种Files.copy的实现。

对于Copy的效率，这个其实与操作系统和配置等情况相关，总体上来说，NIO transferTo/From的方式可能更快，因为它更能利用现代操作系统底层机制，避免不必要的拷贝和上下文切换。

考点分析

今天这个问题，从面试的角度来看，确实是一个面试考察的点，针对我上面的典型回答，面试官还可能会从实践角度，或者IO底层实现机制等方面进一步提问。这一讲的内容从面试题出发，主要还是为了让你进一步加深对Java IO类库设计和实现的了解。

从实践角度，我前面并没有明确说NIO transfer的方法一定最快，真实情况也确实未必如此。我们可以根据理论分析给出可行的推断，保持合理的怀疑，给出验证结论的思路，有时候面试官考察的就是如何将猜测变成可验证的结论，思考方式远比记住结论重要。

从技术角度展开，下面这些方面值得注意：

- 不同的copy方式，底层机制有什么区别？
- 为什么零拷贝（zero-copy）可能有性能优势？
- Buffer分类与使用。
- Direct Buffer对垃圾收集等方面的影响与实践选择。

接下来，我们一起来分析一下吧。

#### 知识扩展

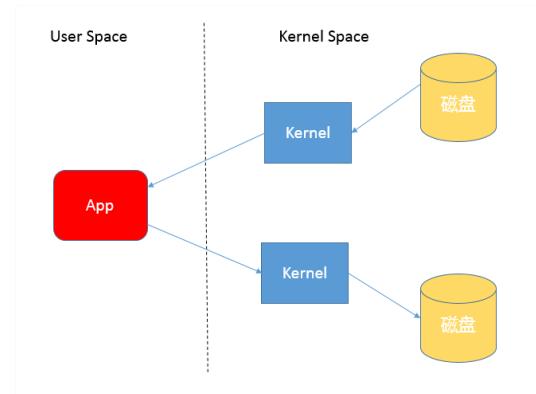
##### 1. 拷贝实现机制分析

先来理解一下，前面实现的不同拷贝方法，本质上有什么明显的区别。

首先，你需要理解用户态空间（User Space）和内核态空间（Kernel Space），这是操作系统层面的基本概念，操作系统内核、硬件驱动等运行在内核态空间，具有相对高的特权；而用户态空间，则是给普通应用和服务使用。你可以参考：[https://en.wikipedia.org/wiki/User\\_space](https://en.wikipedia.org/wiki/User_space)。

当我们使用输入输出流进行读写时，实际上是进行了多次上下文切换，比如应用读取数据时，先在内核态将数据从磁盘读取到内核缓存，再切换到用户态将数据从内核缓存读取到用户缓存。

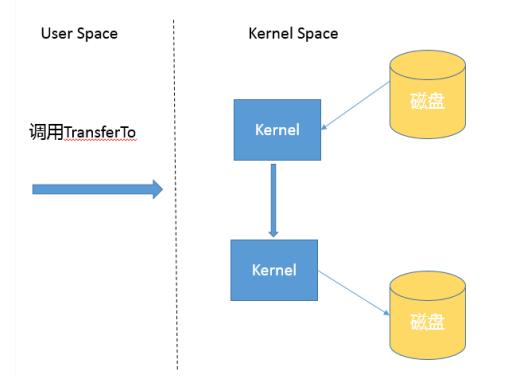
写入操作也是类似，仅仅是步骤相反，你可以参考下面这张图。



所以，这种方式会带来一定的额外开销，可能会降低IO效率。

而基于NIO transferTo的实现方式，在Linux和Unix上，则会使用到零拷贝技术，数据传输并不需要用户态参与，省去了上下文切换的开销和不必要的内存拷贝，进而可能提高应用拷贝性能。注意，transferTo不仅仅是可以用在文件拷贝中，与其类似的，例如读取磁盘文件，然后进行Socket发送，同样可以享受这种机制带来的性能和扩展性提高。

transferTo的传输过程是：



前面我在典型回答中提了第三种方式，即Java标准库也提供了文件拷贝方法（`java.nio.file.Files.copy`）。如果你这样回答，就一定要小心了，因为很少有问题的答案是仅仅调用某个方法。从面试的角度，面试官往往追问：既然你提到了标准库，那么它是怎么实现的呢？有的公司面试官喜欢追问而出名，直到追问到你说不知道。

其实，这个问题的答案还真不是那么直观，因为实际上有几个不同的copy方法。

```
public static Path copy(Path source, Path target, CopyOption... options)
    throws IOException
```

```
public static long copy(InputStream in, Path target, CopyOption... options)
    throws IOException
```

```
public static long copy(Path source, OutputStream out)
    throws IOException
```

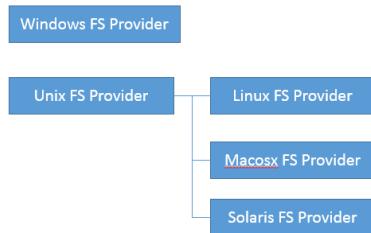
可以看到，`copy`不仅仅是支持文件之间操作，没有人限定输入输出流一定是针对文件的，这是两个很实用的工具方法。

后面两种`copy`实现，能够在方法实现里直接看到使用的是`InputStream.transferTo()`，你可以直接看源码，其内部实现其实是`stream`在用户态的读写；而对于第一种方法的分析过程要相对麻烦一些，可以参考下面片段。简单起见，我只分析同类型文件系统拷贝过程。

```
public static Path copy(Path source, Path target, CopyOption... options)
    throws IOException
{
    FileSystemsProvider provider = provider(source);
    if (provider(target) == provider) {
        // same provider
        provider.copy(source, target, options); //这是本文分析的路径
    } else {
        // different providers
        CopyMoveHelper.copyToForeignTarget(source, target, options);
    }
    return target;
}
```

我把源码分析过程简单记录如下，JDK的源代码中，内部实现和公共API定义也不是能够简单关联上的，NIO部分代码甚至是定义为模板而不是Java源文件，在build过程自动生成源码，下面顺便介绍一下部分JDK代码机制和如何绕过隐藏障碍。

- 首先，直接跟踪，发现`FileSystemProvider`只是个抽象类，阅读它的[源码](#)能够理解到，原来文件系统实际逻辑存在于JDK内部实现里，公共API其实是通过ServiceLoader机制加载一系列文件系统实现，然后提供服务。
- 我们可以在JDK源码里搜索`FileSystemProvider`和`nio`，可以定位到[sun/nio/fs](#)，我们知道NIO底层是和操作系统紧密相关的，所以每个平台都有自己的部分特有文件系统逻辑。



- 省略掉一些细节，最后我们一步步定位到`UnixFileSystemProvider` → `UnixCopyFile.Transfer`，发现这是个本地方法。

- 最后，明确定位到[UnixCopyFile.c](#)，其内部实现清楚说明竟然只是简单的用户态空间拷贝！

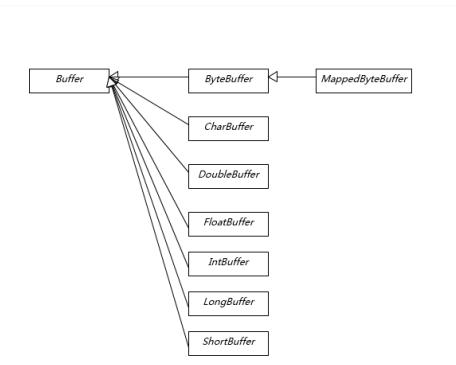
所以，我们明确这个最常见的`copy`方法其实不是利用`transferTo`，而是本地技术实现的用户态拷贝。

前面谈了不少机制和源码，我简单从实践角度总结一下，如何提高类似拷贝等IO操作的性能，有一些宽泛的原则：

- 在程序中，使用缓存等机制，合理减少IO次数（在网络通信中，如TCP传输，window大小也可以看作是类似思路）。
- 使用`transferTo`等机制，减少上下文切换和额外IO操作。
- 尽量减少不必要的转换过程，比如编解码；对对象序列化和反序列化，比如操作文本文件或者网络通信，如果不是过程中需要使用文本信息，可以考虑不要将二进制信息转换成字符串，直接传输二进制信息。

### 3. 掌握NIO Buffer

我在上一讲提到`Buffer`是NIO操作数据的基本工具，Java为每种原始数据类型都提供了相应的`Buffer`实现（布尔除外），所以掌握和使用`Buffer`是十分必要的，尤其是涉及`DirectBuffer`等使用，因为其在垃圾收集等方面特殊性，更要重点掌握。



Buffer有几个基本属性:

- capacity, 它反映这个Buffer到底有多大, 也就是数组的长度。
- position, 要操作的数据起始位置。
- limit, 相当于操作的限额。在读取或者写入时, limit的意义很明显是不一样的。比如, 读取操作时, 很可能将limit设置到所容纳数据的上限; 而在写入时, 则会设置容量或容量以下的可写限度。
- mark, 记录上一次position的位置, 默认是0, 算是一个便利性的考虑, 往往不是必须的。

前面三个是我们日常使用最频繁的, 我简单梳理下Buffer的基本操作:

- 我们创建了一个ByteBuffer, 准备放入数据, capacity当然是缓冲区大小, 而position就是0, limit默认就是capacity的大小。
- 当我们写入几个字节的数据时, position就会跟着水涨船高, 但是它不可能超过limit的大小。
- 如果我们想把前面写入的数据读出来, 需要调用flip方法, 将position设置为0, limit设置为以前的position那里。
- 如果还想从头再读一遍, 可以调用rewind, 让limit不变, position再次设置为0。

更进一步的详细使用, 我建议参考相关[教程](#)。

#### 4.Direct Buffer和垃圾收集

我这里重点介绍两种特别的Buffer。

- Direct Buffer: 如果我们看Buffer的方法定义, 你会发现它定义了isDirect()方法, 返回当前Buffer是否是Direct类型。这是因为Java提供了堆内和堆外(Direct) Buffer, 我们可以以它的allocate或者allocateDirect方法直接创建。
- MappedByteBuffer: 它将文件按照指定大小直接映射为内存区域, 当程序访问这个内存区域时将直接操作这块儿文件数据, 省去了将数据从内核空间向用户空间传输的损耗。我们可以使用[FileChannel.map](#)创建MappedByteBuffer, 它本质上也是种Direct Buffer。

在实际使用中, Java会尽量对Direct Buffer仅做本地IO操作, 对于很多大数据量的IO密集操作, 可能会带来非常大的性能优势, 因为:

- Direct Buffer生命周期内内存地址都不会再发生更改, 进而内核可以安全地对其进行访问, 很多IO操作会很高效。
- 减少了堆内外对象存储的可能额外维护工作, 所以访问效率可能有所提高。

但是请注意, Direct Buffer创建和销毁过程中, 都会比一般的堆内Buffer增加部分开销, 所以通常都建议用于长期使用、数据较大的场景。

使用Direct Buffer, 我们需要清楚它对内存和JVM参数的影响。首先, 因为它不在堆上, 所以Xmx之类参数, 其实并不能影响Direct Buffer等堆外成员所使用的内存额度, 我们可以使用下面参数设置大小:

```
-XX:MaxDirectMemorySize=512M
```

从参数设置和内存问题排查角度来看, 这意味着我们在计算Java可以使用的内存大小的时候, 不能只考虑堆的需要, 还有Direct Buffer等一系列堆外因素。如果出现内存不足, 堆外内存占用也是一种可能性。

另外, 大多数垃圾收集过程中, 都不会主动收集Direct Buffer, 它的垃圾收集过程, 就是基于我在专栏前面所介绍的Cleaner(一个内部实现)和幻象引用(PhantomReference)机制, 其本身不是public类型, 内部实现了一个Deallocator负责销毁的逻辑。对它的销毁往往要拖到full GC的时候, 所以使用不当很容易导致OutOfMemoryError。

对于Direct Buffer的回收, 我有几个建议:

- 在应用程序中, 显式地调用System.gc()来强制触发。
- 另外一种思路是, 在大量使用Direct Buffer的部分框架中, 框架会自己在程序中调用释放方法, Netty就是这么做的, 有兴趣可以参考其实现(PlatformDependent0)。
- 重复使用Direct Buffer。

#### 5.跟踪和诊断Direct Buffer内存占用?

因为通常的垃圾收集日志等记录, 并不包含Direct Buffer等信息, 所以Direct Buffer内存诊断也是个比较头疼的事情。幸好, 在JDK 8之后的版本, 我们可以方便地使用Native Memory Tracking(NMT)特性来进行诊断, 你可以在程序启动时加上下面参数:

-XX:NativeMemoryTracking={summary|detail}

注意，激活NMT通常都会导致JVM出现5%~10%的性能下降，请谨慎考虑。

运行时，可以采用下面命令进行交互式对比：

```
// 打印NMT信息
jcmd <pid> VM.native_memory detail

// 进行baseline，以批分配内存变化
jcmd <pid> VM.native_memory baseline

// 进行baseline，以批分配内存变化
jcmd <pid> VM.native_memory detail,diff
```

我们可以在Internal部分发现Direct Buffer内存使用的信息，这是因为其底层实际是利用unsafe\_allocateMemory。严格说，这不是JVM内部使用的内存，所以在JDK 11以后，其实它是归类在other部分里。

JDK 9的输出片段如下，“+”表示的就是diff命令发现的分配变化：

```
-Internal (reserved=679KB +4KB, committed=679KB +4KB)
  (malloc=615KB +4KB #1571 +4)
  (mmap: reserved=64KB, committed=64KB)
```

注意：JVM的堆外内存远不止Direct Buffer，NMT输出的信息当然也远不止这些，我在专栏后面有综合分析更加具体的内存结构的主题。

今天我分析了Java IO/NIO底层文件操作数据的机制，以及如何实现零拷贝的高性能操作，梳理了Buffer的使用和类型，并针对Direct Buffer的生命周期管理和诊断进行了较详细地分析。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？你可以思考下，如果我们需要在channel读取的过程中，将不同片段写入到相应的Buffer里面（类似二进制消息分拆成消息头、消息体等），可以采用NIO的什么机制做到呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



13576788017

杨老师，想问一下，一般要几年java经验才能达到看懂你文章的地步？？我将近一年经验。。我发现我好几篇都看不懂。。底层完全不懂。。是我太菜了吗。。  
作者回复

2018-05-31

2018-05-31

非常抱歉，具体哪几篇？公司对一年经验的工程师要求应该也不一样的

行者

2018-05-31

可以利用NIO分散-scatter机制来写入不同buffer。  
Code:  
ByteBuffer header = ByteBuffer.allocate(128);  
ByteBuffer body = ByteBuffer.allocate(1024);  
ByteBuffer[] bufferArray = {header, body};  
channel.read(bufferArray);  
注意:该方法适用于请求头长度固定。

作者回复

2018-05-31

赞

石头狮子

若使用非 directbuffer 操作相关 api 的话，jdk 会将其复制为 directbuff，并且在线程内部缓存该directbuffer。jdk 对这个缓存的大小并没有限制。  
之前遇到缓存的 directbuffer 过多，导致oom的情况。后续 jdk 版本加入了对该缓存的限制。  
额外一点是尽量不要使用堆内的 bytebuffer 操作 channel 类 api。

2018-05-31

vash\_ace  
2018-06-01  
其实在初始化 DirectByteBuffer 对象时，如果当前堆内存的条件很苛刻时，会主动调用 System.gc() 强制执行 FGC。所以一般建议在使用 netty 时开启 XX:+DisableExplicitGC  
作者回复

对，检测不够时会尝试调 system.gc，记得老版本有并发分配 bug，会出 oom，netty 文中提到了，它是 hack 到内部自己释放…

乘风破浪

零拷贝是不是可以理解为内核空间与磁盘之间的数据传输，不需要再经过用户态空间？  
作者回复

嗯

闭门车

你好，我看 jdk8 中的源码，看到您说的关于 Files.copy 其中两种是依靠 transferTo 实现的，但是我翻看源码觉得跟您的理解不同，特来求证，源码如下：public static long copy(Path source, OutputStream out) throws IOException {  
 Objects.requireNonNull(out);  
 try (InputStream in = newInputStream(source)) {  
 return copy(in, out);  
 }  
}  
private static long copy(InputStream source, OutputStream sink)  
throws IOException {  
 long nread = 0L;  
 byte[] buf = new byte[BUFFER\_SIZE];  
 int n;  
 while ((n = source.read(buf)) > 0) {  
 sink.write(buf, 0, n);  
 nread += n;  
 }  
 return nread;  
}

灰飞灰猪不会灰飞烟灭

老师，带缓冲区的 io 流比.nio 哪一个性能更好？  
作者回复

嗯，文中提到过，不能一概而论，性能通常是特定场景下的比较才有意义

mongo  
2018-05-31  
杨老师，我也想请教，目前为止您的其他文章都理解的很好，到了上次专栏的 NIO 我理解的不是很好，今天的这篇可以说懵圈了。到了这一步想突破，应该往哪个方向？我自己感觉是因为基于这两点底层原理的上层应用使用的時候观察的不够深入，对原理反应的现象没有深刻感受，就是所谓还没有摸清楚人家长什么样。所以接下来我会认真在使用基于这些原理实现的上层应用过程中不断深挖和观察，比如认真学习 dubbo 框架（底层使用到了 netty，netty 的底层使用了 NIO）来帮助我理解 NIO。在这个过程中促进对 dubbo 的掌握，以此良性循环。不知道方向对不对？老师的习题方法是什么？请老师指点迷津。学习方法不对的话时间成本太可怕。

作者回复

我觉得思路不错，结合实践是非常好的；我自身也仅仅是理论上理解，并没有在大规模实践中踩坑，实践中遇到细节的“坑”其实是宝贝，所以你有更多优势；本文从基础角度出发，也是希望对其原理有个整体印象，至少被面试刨根问底时，可以有所准备，毕竟看问题的角度是不同的

夏洛克的救赎

请问老师您有参与 jdk 的开发吗  
作者回复

是的，目前 lead 的团队主要是 QE 负责

沈老师  
2018-05-31  
上面有两张说明普通 copy 和.nio 下的 transfer，我理解大致意思就是后者省去了切换到用户态的开销，但想问一下前者为什么要设计这样的切换呢？是不是和你 copy 的数据类型有关？还望详释，  
作者回复

抱歉，这得问 Mark R，不清楚历史原因；我理解大部分工作其实是需要用户态的，transfer 是特定场景而非通用场景

玲玲爱学习  
2018-06-28  
堆外缓存是否与内核缓存是同一个东西？

MiaoZhe  
2018-06-07  
杨老师再问个问题，DirectByteBffer 类为什么不能使用？看代码它是继承自 MappedByteBuffer。

CC  
2018-06-07  
我 NIO 从没接触过，很难受，两年开发的  
作者回复

不用担心，是理解有难度吗？

MiaoZhe  
2018-06-07  
再问个问题，在 Java 8 中，对 Byte Buffer 有这样的描述，a byte buffer is either or non-direct。  
我有点晕乎了，在代码调试中 isDirect 是 False。

作者回复

你的bytebuffer怎么分配的? allocate? 试试allocateDirect。  
那段话我理解就是说 不是这个就是那个吧

2018-06-07

Miaozhe

杨老师, 问个问题, Byte Buffer对象什么时候被垃圾回收?

2018-06-07

作者回复

依赖于cleaner, 具体时候不好预测, 在快满时会被调用system.gc触发ref处理, 后续版本有所改进, 至少不会出现, 明明有空间, 但并发分配会oom的问题

2018-06-07

jacky

杨老师, 如果显示调用system.gc会不会导致堆空间充足, 但fullgc频繁的现象呢? 导致应用经常停顿?

2018-05-31

作者回复

肯定有副作用

2018-06-01

I am a psycho

杨老师, 我看1.8的源码中, files.copy共重载了4个方法, 其中有三个调用的都是bio, 有一个是您讲的native调用, 而没有nio的transferTo。请问这是jdk9变成nio的吗?

2018-05-31

作者回复

哦, 现在机场, 我平时工作在最新版本, 现在是jdk11...

2018-06-01

正是那朵玫瑰

2018-05-31

可以具体讲一下MappedByteBuffer的原理和使用技巧么?

2018-05-31

灰飞灰猪不会灰飞-烟灭

2018-05-31

杨老师 我刚刚做个项目, 上传文件到文件服务器, 文件大概10M, 经常上传失败。假如我上传的文件改成1M, 就没这问题了。不知道什么原因, 能提供个思路吗? 谢谢

2018-06-01

作者回复

有个建议, 工程师在沟通故障时, 可以收集下出错信息; 客户端、服务端配置; 网络情况, 比如是否有代理, 等等。

2018-06-01

至于思路, 能否找到程序异常信息之类, 做过哪些尝试将问题缩小范围?

2018-06-01

定位问题通常就是个不断缩小范围, 排除不可能的过程。

2018-06-01

crazyone

2018-05-31

杨老师, Nio transfer 不一定快的场景是否有案例场景说明下? 还有你说MappedByteBuffer本质上也是一个Direct Buffer ,那它设计的目的和意义是什么?

2018-05-31

作者回复

找个普通笔记本试试, stream那个往往更快...

2018-05-31

Cui

2018-05-31

Direct Buffer 生命周期内内存地址都不会再发生更改, 进而内核可以安全地对其进行访问—这里能提高性能的原因 是因为内存地址不变, 没有锁争用吗? 能否详细解答下?

2018-05-31

作者回复

我理解不是锁的问题, 寻址简单, 才好更直接

2018-05-31

LenX

2018-05-31

请教老师:  
 1. 经常看到 Java 进程的 RES 大小远超过设置的 Xmx, 可以认为这就是 Direct Memory 的原因吗? 如果是的话, 可以简单的用堆实际占用的大小减去 RES 就是 Direct Memory 的大小吗?

2018-05-31

2. 可以认为 Direct Memory 不论在什么情况下都不会引起 Full GC, Direct Memory 的回收只是在 Full GC (或调用 System.gc())的时候顺带着回收下, 是吗?

2018-05-31

作者回复

1, 一般不是, 那东西有个默认大小的, metaspace codecach等等都会占用, 后面有章节仔细分析

2018-05-31

2, 不是, 它是利用sun.misc.Cleaner, 实际表现有瑕疵, 经常要更依赖system.gc去触发引用处理, 9和8u有改进, 我会有详解







## 第13讲 | 谈谈接口和抽象类有什么区别?

2018-06-02 杨晓峰



第13讲 | 谈谈接口和抽象类有什么区别?  
杨晓峰  
0:00 / 11:06

Java是非常典型的面向对象语言，曾经有一段时间，程序员整天把面向对象、设计模式挂在嘴边。虽然如今大家对这方面已经不再那么狂热，但是不可否认，掌握面向对象设计原则和技巧，是保证高质量代码的基础之一。

面向对象提供的基本机制，对于提高开发、沟通等各方面效率至关重要。考察面向对象也是面试中的常见一环，下面我来聊聊面向对象设计基础。

今天我要问你的是，[谈谈接口和抽象类有什么区别？](#)

典型回答

接口和抽象类是Java面向对象设计的两个基础机制。

接口是对行为的抽象，它是抽象方法的集合，利用接口可以达到API定义和实现分离的目的。接口，不能实例化；不能包含任何非常量成员，任何field都是隐含着public static final的意义；同时，没有非静态方法实现，也就是说要么是抽象方法，要么是静态方法。Java标准类库中，定义了非常多的接口，比如java.util.List。

抽象类是不能实例化的类，用abstract关键字修饰class，其目的主要是代码重用。除了不能实例化，形式上和一般的Java类并没有太大区别，可以有一个或者多个抽象方法，也可以没有抽象方法。抽象类大多用于抽取相关Java类的共用方法实现或者是共同成员变量，然后通过继承的方式达到代码复用的目的。Java标准库中，比如collection框架，很多通用部分就被抽取成为抽象类，例如java.util.AbstractList。

Java类实现Interface使用implements关键词，继承abstract class则是使用extends关键词，我们可以参考Java标准库中的ArrayList。

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    ...
}
```

考点分析

这是个非常高频的Java面向对象基础问题，看起来非常简单的问题，如果面试官稍微深入一些，你会发现很多有意思的地方，可以从不同角度全面地考察你对基本机制的理解和掌握。比如：

- 对于Java的基本元素的语法是否理解准确。能否定义出语法基本正确的接口、抽象类或者相关继承实现，涉及重载（Overload）、重写（Override）更是有各种不同的题目。
- 在软件设计开发中妥善地使用接口和抽象类。你至少知道典型应用场景，掌握基础类库重要接口的使用；掌握设计方法，能够在review代码的时候看出明显的不利于未来维护的设计。
- 掌握Java语言特性演进。现在很多的框架已经是基于Java 8，并逐渐支持更新版本，掌握相关语法，理解设计目的是很有必要的。

知识扩展

我会从接口、抽象类的一些实践，以及语言变化方面去阐述一些扩展知识点。

Java相比于其他面向对象语言，如C++，设计上有一些基本区别。比如Java不支持多继承。这种限制，在规范了代码实现的同时，也产生了一些局限性，影响着程序设计结构。Java类可以实现多个接口，因为接口是抽象方法的集合，所以这是声明性的，但不能通过扩展多个抽象类来重用逻辑。

在一些情况下存在特定场景，需要抽象出与具体实现、实例化无关的通用逻辑，或者纯调用关系的逻辑，但是使用传统的抽象类会陷入到单继承的窘境。以往常见的做法是，实现由静态方法组成的工具类（Utils），比如java.util.Collections。

设想，为接口添加任何抽象方法，相应的所有实现了这个接口的类，也必须实现新增方法，否则会出现编译错误。对于抽象类，如果我们添加非抽象方法，其子类只会享受到能力扩

展，而不用担心编译出问题。

接口的职责也仅仅限于抽象方法的集合，其实有各种不同的实践。有一类没有任何方法的接口，通常叫作**Marker Interface**，顾名思义，它的目的就是为了声明某些东西，比如我们熟知的**Cloneable**、**Serializable**等。这种用法，也存在于业界其他的Java产品代码中。

从表面看，这似乎和**Annotation**异曲同工，也确实如此，它的好处是简单直接。对于**Annotation**，因为可以指定参数和值，在表达能力上要更强大一些，所以更多人选择使用**Annotation**。

Java 8增加了函数式编程的支持，所以又增加了一类定义，即所谓**functional interface**，简单说就是只有一个抽象方法的接口，通常建议使用@FunctionalInterface Annotation来标记。Lambda表达式本身可以看作是一类**functional interface**，某种程度上这和面向对象可以算是两码事。我们熟知的**Runnable**、**Callable**之类，都是**functional interface**，这里不再多介绍了。有兴趣你可以参考：<https://www.oreilly.com/learning/java-8-functional-interfaces>。

还有一点可能让人感到意外，严格说，Java 8以后，接口也是可以有方法实现的！

从Java 8开始，interface增加了对default method的支持。Java 9以后，甚至可以定义private default method。Default method提供了一种二进制兼容的扩展已有接口的办法。比如，我们熟知的java.util.Collection，它是collection体系的root interface，在Java 8中添加了一系列default method，主要是增加Lambda、Stream相关功能。我在专栏前面提到的类似Collections之类的工具类，很多方法都适合作为default method实现的基础接口里面。

你可以参考下面代码片段：

```
public interface Collection<E> extends Iterable<E> {
    /**
     * Returns a sequential Stream with this collection as its source
     * ...
     */
    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }
}
```

#### 面向对象设计

谈到面向对象，很多人就会想起设计模式，那些是非常经典的问题和设计方法的总结。我今天来夯实一下基础，先来聊聊面向对象设计的基本方面。

我们一定要清楚面向对象的基本要素：封装、继承、多态。

封装的目的是隐藏事务内部的实现细节，以便提高安全性和简化编程。封装提供了合理的边界，避免外部调用者接触到内部的细节。我们在日常开发中，因为无意间暴露了细节导致的难debug太多了，比如在多线程环境暴露内部状态，导致的并发修改问题。从另外一个角度看，封装这种隐藏，也提供了简化的界面，避免太多无意义的细节浪费调用者的精力。

继承是代码复用的基础机制，类似于我们对于马、白马、黑马的归纳总结。但要注意，继承可以看作是非常紧耦合的一种关系，父类代码修改，子类行为也会变动。在实践中，过度滥用继承，可能会起到反效果。

多态，你可能立即会想到重写（override）和重载（overload）、向上转型。简单说，重写是父子类中相同名字和参数的方法，不同的实现；重载则是相同名字的方法，但是不同的参数，本质上这些方法签名是不一样的，为了更好说明，请参考下面的样例代码：

```
public int doSomething() {
    return 0;
}

// 输入参数不同，意味着方法签名不同，重载的体现
public int doSomething(List<String> strs) {
    return 0;
}

// return类型不一样，编译不能通过
public short doSomething() {
    return 0;
}
```

这里你可以思考一个问题，方法名称和参数一致，但是返回值不同，这种情况在Java代码中算是有效的重载吗？答案是不是的，编译都会出错的。

进行面向对象编程，掌握基本的设计原则是必须的，我今天介绍最通用的部分，也就是所谓的S.O.L.I.D原则。

- **单一职责 (Single Responsibility)**，类或者对象最好是只有单一职责，在程序设计中如果发现某个类承担着多种义务，可以考虑进行拆分。
- **开关原则 (Open-Close, Open for extension, close for modification)**，设计要对扩展开放，对修改关闭。换句话说，程序设计应保证平滑的扩展性，尽量避免因为新增同类功能而修改已有实现，这样可以少产出些回归（regression）问题。
- **里氏替换 (Liskov Substitution)**，这是面向对象的基本要素之一，进行继承关系抽象时，凡是可以用父类或者基类的地方，都可以用子类替换。
- **接口分离 (Interface Segregation)**，我们在进行类和接口设计时，如果在一个接口里定义了太多方法，其子类很可能面临两难，就是只有部分方法对它是有意义的，这就破坏了程序的内聚性。  
对于这种情况，可以通过拆分功能单一的多个接口，将行为进行解耦。在未来维护中，如果某个接口设计有变，不会对使用其他接口的子类构成影响。
- **依赖反转 (Dependency Inversion)**，实体应该依赖于抽象而不是实现。也就是说高层次模块，不应该依赖于低层次模块，而是应该基于抽象。实践这一原则是保证产品代码之间适当耦合度的法宝。

#### OOP原则实践中的取舍

值得注意的是，现代语言的发展，很多时候并不是完全遵守前面的原则的，比如，Java 10中引入了本地方法类型推断和var类型。按照，里氏替换原则，我们通常这样定义变量：

```
List<String> list = new ArrayList<>();
```

如果使用var类型，可以简化为

```
var list = new ArrayList<String>();
```

但是，list实际会被推断为“ArrayList < String >”

```
ArrayList<String> list = new ArrayList<String>();
```

理论上，这种语法上的便利，其实是增强了程序对实现的依赖，但是微小的类型泄漏却带来了书写的变量和代码可读性的提高。所以，实践中我们还是要按照得失利弊进行选择，而不是一味得遵循原则。

#### OOP原则在面试题目中的分析

我在以往面试中发现，即使是有多年编程经验的工程师，也还没有真正掌握面向对象设计的基本的原则，如开关原则（Open-Close）。看看下面这段代码，改编自朋友圈盛传的某伟大公司产品代码，你觉得可以利用面向对象设计原则如何改进？

```
public class VIPCenter {
    void serviceVIP(T extend User user) {
        if (user instanceof SlumDogVIP) {
            // 穷X VIP, 活动的限制
            // do somthing
        } else if(user instanceof RealVIP) {
            // do somthing
        }
        // ...
    }
}
```

这段代码的一个问题是，业务逻辑集中在一起，当出现新的用户类型时，比如，大数据发现了我们是肥羊，需要去收获一下，这就需要直接去修改服务方法代码实现，这可能会意外影响不相关的某个用户类型逻辑。

利用开关原则，我们可以尝试改造为下面的代码：

```
public class VIPCenter {
    private Map<User.TYPE, ServiceProvider> providers;
    void serviceVIP(T extend User user) {
        providers.get(user.getType()).service(user);
    }
}
interface ServiceProvider{
    void service(T extend User user) ;
}
class SlumDogVIPServiceProvider implements ServiceProvider{
    void service(T extend User user){
        // do somthing
    }
}
class RealVIPServiceProvider implements ServiceProvider{
    void service(T extend User user) {
        // do something
    }
}
```

上面的示例，将不同对象分类的服务方法进行抽象，把业务逻辑的紧耦合关系拆开，实现代码的隔离保证了方便的扩展。

今天我对Java面向对象技术进行了梳理，对比了抽象类和接口，分析了Java语言在接口层面的演进和相应程序设计实现，最后回顾并实践了面向对象设计的基本原则，希望对你有所帮助。

#### 一课一练

关于接口和抽象类的区别，你做到心中有数了吗？给你布置一个思考题，思考一下自己的产品代码，有没有什么地方违反了基本设计原则？那些一改就崩的代码，是否遵循了开关原则？

请你在留言区写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



公号-Java大后端

2018-06-02

接口vs抽象类vs类

- 1 支持多重继承：接口支持，抽象类不支持，类不支持；
- 2 支持抽象函数：接口语义上支持，抽象类支持，类不支持；
- 3 允许函数实现：接口不允许，抽象类支持，类允许；
- 4 允许实例化：接口不允许，抽象类不允许，类允许；
- 5 允许部分函数实现：接口不允许，抽象类允许，类不允许。
- 6 定义的内容：接口中只能包括public函数以及public static final常量；抽象类与类均无任何限制。
- 7 使用时机：当想要支持多重继承，或是为了定义一种类型请使用接口；当打算提供带有部分实现的“模板”类，而将一些功能需要延迟实现请使用抽象类。

在实际项目开发过程，一方面是业务需求频繁，需要满足开闭原则，也就是小到一个模块，大到一个架构都需要有好的可扩展性；另外一方面软件往往是团队协同开发的过程，由于团队成员水平参差不齐，这方面的坑不少。可以通过前期做好设计评审、code review等手段去提升代码质量。

ωo→无悔

2018-06-02

最后一个例子就是策略模式工厂模式

张立春

2018-06-02

我理解继承的根本目的是为了多态而不是为了复用，如果仅为了复用那就采用松耦合的组合。

Woong

2018-06-02

class SlumDogVIPServiceProvider和RealVIPServiceProvider缺少implementments.

作者回复

2018-06-03

汗，手敲搞出这种低级错误，非常感谢指出

小情绪

2018-06-03

杨老师，对于开头的：接口中没有非静态方法实现，也就是说要么是抽象方法，要么是静态方法。这句话我有疑问，java.util.List中default方法不就是非静态方法的实现吗？还是我理解有误？

作者回复

2018-06-05

前面就是个举例的回答，用来后面分析的，新版Java不准确了

qpm

2018-06-02

hi，老师早上好。我是一家游戏公司的程序员，由于项目非常紧，所以很多技术写的代码，都非常乱。我们的战斗系统中，之前的开发模式是在战斗逻辑里面嵌入并修改一些代码，以达到新技能的开发。这就是典型的以修改来达到需求。通过重构之后，技能的逻辑通过扩展的方式开发出来，可以通过继承技能的抽象类，来完成技能的开发。现在我们这部分的模块从最难处理变成最容易开发的了。

作者回复

2018-06-03

听说过电信代码里有26个if-else.....然后还有俩是重复的，少的时候无所谓，多了就是坑

Seven4X

2018-06-16

对象和抽象类是is a 的关系，对象和接口是 like this的关系。

从接口的命名一般是以able ability后缀表示一种能力。

比如大家都程序员，如果拥有了编写java编程的能力，就可以说你是一个java程序员，同时你还能说你会编写Go，也可以说你是一个go程序员。

程序员就是对编程能力的封装，如果把java编程能力定义为抽象类，那还要实现Go编程能力接口才能同时具有两种语言能力的程序员，这表现的是一个以java编程为主的程序员，同时具有编写go的能力。

如果把java能力和go能力都定义为接口，表现的是，这是一个程序员他同时具有编写java和go的能力。

雷霹雳的爸爸

2018-06-02

问题本身就是典型热身题，但是SOLID则是一块试金石，曾经真的以为是试金石，很多老江湖好像都不知道有这串缩写这么回事，单一职责一说就都是顾名思义，纯粹的同义反复，用自己解释自己都没说差不多吧，里氏替换背后的契约设计基本规则就更甭说了，实际项目中能不违反的几乎凤毛麟角吧，除非对接口调用后就真的没有约束条件，连spring这货都崇尚什么对封装非受控异常，可见这个到底有多么不受人待见，最小知识原则也基本上就是任人践踏吧，要不全变函数接口也就都没必要反复换个名字了吧，依赖倒置还好，好歹有IOC帮衬着，但是helper和各种静态的tool被无数人个性的反复的造，同时拿着依赖查找的实用性来堵DIP的嘴，也就剩OCP，再违反就太不像话了吧，没关系最大的伤害就是无视啊，帅气的蓝精灵命名法的那一串类，依次加个方法还是客气的，直接加个version字段，把if..else请回来，总之这玩意一说我就陷入到吐槽情绪，但问题吐的是solid的槽，还是吐的不把solid当回事的槽，我自己也分不清了，这个问题真的好让人纠结

曹铮

10年前校招就被问抽象类和接口的区别。过了几年被问接口里能不能定义字段。面试官还是蛮爱问这些的…

2018-06-02

bamboo

老师最后举的例子应该就是把简单工厂模式修改为工厂方法模式。原来的违反了开闭选择，工厂方法模式刚好弥补了这个问题，倒是相应的系统中的类个数也对的增加。设计模式没有最优的，只是特定场景下我们选择相对优秀的模式来优化我们的逻辑。不知道是否正确，望老师指点迷津，谢谢老师。◆◆

作者回复

嗯，也要避免过度设计，这个只是举例

2018-06-03

卡斯瓦德

其实interface的default可以通过结合抽象父类来实现吧，抽象父类实现接口方法，但是因为抽象所以不能实例化，而其子类拥有重写权，可以做到default的效果

作者回复

不错，只是：  
抽象类不能多继承；  
default method不会打破现有代码兼容性，lambda需要靠它来无缝增强collection之类API

zc

2018-07-14

\*这里你可以思考一个问题，方法名称和参数一致，但是返回值不同。这种情况在 Java 代码中算是有效的重载吗？答案是不是的，编译都会出错的。“

编译出错是因为重载必须参数不一样，重载与返回值无关。感觉这里的表述有点问题……

示例代码既不属于重写（同名同参同返回）也不属于重载（同名不同参）的范畴。

000

2018-07-14

抽象方法不实现，就是为了给子类用的吗？

仙道

2018-07-12

两个接口里有一个同名方法，然后一个类实现了这两个接口，这怎么办呢

夏洛克的救赎

2018-07-02

但是，list 实际会被推断为“ArrayList < String >”

ArrayList<String> list = new ArrayList<String>();

不是很理解

作者回复

一般我们会写成：  
List<String> list=new ArrayList.....;  
里氏代换，可以参考oo基础那章

张健

2018-06-20

2018-06-14

没解释多态  
方法重载这里，如果用jdk javac编译就会过的

云学

2018-06-13

其实有了函数式编程，绝大部分设计模式是多余的，记住，类的数量不要泛滥！！

Yao

2018-06-12

问个问题，Jdk8 default 是否应该使用？？？

作者回复

看你的需求啊，比如代码需不需要兼容老的jdk版本，有没有需要用的逻辑

lorancechen

2018-06-12

2018-06-09

VIP的例子里面，T extend User这里，直接用User也一样吧

作者回复

是的，说明一下

Jerry很恨

2018-06-10

2018-06-03

根据开闭原则，将VIPCenter改造的方法应该是一种设计模式吧，是什么设计模式呢？

张希功(pokercc)

2018-06-03

2018-06-03

“void serviceVIP(T extend User user) ”  
这是java新出的写法吗？我用java8这样写，编译不通过呢

作者回复

本来是个示意的伪代码…为了尽量精简的说明结构设计，不是为了语法

行者

2018-06-05

2018-06-03

如果把软件开发比做构建一座大楼，使用设计模式进行精心设计，把房屋结构性的支撑规划清楚，这样日后再决定用地板还是地毯，屋顶用什么材料，规划得当能够让我们再后期可以方便的进行替换。

Turing

2018-06-03

那个改进的也可以用策略模式来写

j.c.

同种类型行为是通过不同方法调用还是不同实现类，哪个好点？

2018-06-02



## 第14讲 | 谈谈你知道的设计模式?

2018-06-05 杨晓峰



## 第14讲 | 谈谈你知道的设计模式?

杨晓峰

- 00:00 / 08:23

设计模式是人们为软件开发中相同表征的问题，抽象出的可重复利用的解决方案。在某种程度上，设计模式已经代表了一些特定情况的最佳实践，同时也起到了软件工程师之间沟通的“行话”的作用。理解和掌握典型的设计模式，有利于我们提高沟通、设计的效率和质量。

今天我要问你的是，**谈谈你知道的设计模式？请手动实现单例模式、Spring等框架中使用了哪些模式？**

## 典型回答

大致按照模式的应用目标分类，设计模式可以分为创建型模式、结构型模式和行为型模式。

- **创建型模式**，是对对象创建过程的各种问题和解决方案的总结，包括各种工厂模式（Factory、Abstract Factory）、单例模式（Singleton）、构建器模式（Builder）、原型模式（ProtoType）。
- **结构型模式**，是针对软件设计结构的总结，关注于类、对象继承、组合方式的实践经验。常见的结构型模式，包括桥接模式（Bridge）、适配器模式（Adapter）、装饰者模式（Decorator）、代理模式（Proxy）、组合模式（Composite）、外观模式（Facade）、享元模式（Flyweight）等。
- **行为型模式**，是从类或对象之间交互、职责划分等角度总结的模式。比较常见的行为型模式有策略模式（Strategy）、解释器模式（Interpreter）、命令模式（Command）、观察者模式（Observer）、迭代器模式（Iterator）、模板方法模式（Template Method）、访问者模式（Visitor）。

## 考点分析

这个问题主要是考察你对设计模式的理解和掌握程度，更多相关内容你可以参考：[https://en.wikipedia.org/wiki/Design\\_Patterns\\_](https://en.wikipedia.org/wiki/Design_Patterns_)

我建议可以在回答时适当地举些例子，更加清晰地说明典型模式到底是什么样子，典型使用场景是怎样的。这里举个Java基础类库中的例子供你参考。

首先，[专栏第11讲](#)刚介绍过IO框架，我们知道InputStream是一个抽象类，标准类库中提供了FileInputStream、ByteArrayInputStream等各种不同的子类，分别从不同角度对InputStream进行了功能扩展，这是典型的装饰器模式应用案例。

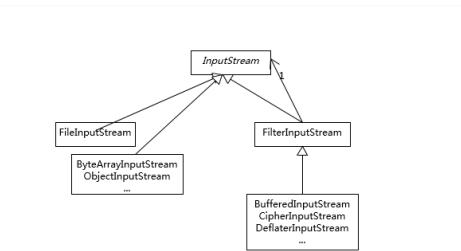
识别装饰器模式，可以通过识别类设计特征来进行判断，也就是其类构造函数以相同的抽象类或者接口为输入参数。

因为装饰器模式本质上是包装同类型实例，我们对目标对象的调用，往往会通过包装类覆盖过的方法，迂回调用被包装的实例，这就可以很自然地实现增加额外逻辑的目的，也就是所谓的“装饰”。

例如，BufferedInputStream经过包装，为输入流过程增加缓存，类似这种装饰器还可以多次嵌套，不断地增加不同层次的功能。

```
public BufferedInputStream(InputStream in)
```

我在下面的类图里，简单总结了InputStream的装饰模式实践。



接下来再看第二个例子。创建型模式尤其是工厂模式，在我们的代码中随处可见，我举个相对不同的API设计实践。比如，JDK最新版本中 HTTP/2 Client API，下面这个创建HttpRequest的过程，就是典型的构建器模式（Builder），通常会被实现成[fluent风格](#)的API，也有人叫它方法链。

```

HttpRequest request = HttpRequest.newBuilder(new URI(uri))
    .header(headerAlice, valueAlice)
    .headers(headerBob, value1Bob,
        headerCarl, valueCarl,
        headerBob, value2Bob)
    .GET()
    .build();
  
```

使用构建器模式，可以比较优雅地解决构建复杂对象的麻烦，这里的“复杂”是指类似需要输入的参数组合较多，如果用构造函数，我们往往需要为每一种可能的输入参数组合实现相应的构造函数，一系列复杂的构造函数会让代码阅读性和可维护性变得很差。

上面的分析也进一步反映了创建型模式的初衷，即，将对象创建过程单独抽象出来，从结构上把对象使用逻辑和创建逻辑相互独立，隐藏对象实例的细节，进而为使用者实现了更加规范、统一的逻辑。

更进一步进行设计模式考察，面试官可能会：

- 希望你写一个典型的设计模式实现。这虽然看似简单，但即使是最简单的单例，也能够综合考察代码基本功。
- 考察典型的设计模式使用，尤其是结合标准库或者主流开源框架，考察你对业界良好实践的掌握程度。

在面试时如果恰好问到你不熟悉的模式，你可以稍微引导一下，比如介绍你在产品中使用了什么自己相对熟悉的模式，试图解决什么问题，它们的优点和缺点等。

下面，我会针对前面两点，结合代码实例进行分析。

#### 知识扩展

我们来实现一个日常非常熟悉的单例设计模式。看起来似乎很简单，那么下面这个样例符合基本需求吗？

```

public class Singleton {
    private static Singleton instance = new Singleton();
    public static Singleton getInstance() {
        return instance;
    }
}
  
```

是不是总感觉缺了点什么？原来，Java会自动为没有明确声明构造函数的类，定义一个public的无参数的构造函数，所以上面的例子并不能保证额外的对象不被创建出来，别人完全可以直接“new Singleton()”，那我们应该怎么处理呢？

不错，可以为单例定义一个private的构造函数（也有建议声明为枚举，这是有争议的，我个人不建议选择相对复杂的枚举，毕竟日常开发不是学术研究）。这样还有什么改进的余地吗？

[专栏第10讲](#)介绍ConcurrentHashMap时，提到过标准类库中很多地方使用懒加载（lazy-load），改善初始内存开销，单例同样适用，下面是修正后的改进版本。

```

public class Singleton {
    private static Singleton instance;
    private Singleton() {
    }
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
  
```

这个单例在单线程环境不存在问题，但是如果处于并发场景，就需要考虑线程安全，最熟悉的就莫过于“双检锁”，其要点在于：

- 这里的volatile能够提供可见性，以及保证getInstance返回的是初始化完全的对象。
- 在同步之前进行null检查，以尽量避免进入相对昂贵的同步块。

- 直接在class级别进行同步，保证线程安全的类方法调用。

```
public class Singleton {
    private static volatile Singleton singleton = null;
    private Singleton() {
    }

    public static Singleton getSingleton() {
        if (singleton == null) { // 尽量避免重复进入同步块
            synchronized (Singleton.class) { // 同步.class, 意味着对同步类方法调用
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

在这段代码中，争论较多的是`volatile`修饰静态变量，当`Singleton`类本身有多个成员变量时，需要保证初始化过程完成后，才能被`get`到。

在现代Java中，内存排序模型（JMM）已经非常完善，通过`volatile`的`write`或者`read`，能保证所谓的`happen-before`，也就是避免常被提到的指令重排。换句话说，构造对象的`store`指令能够被保证一定在`volatile read`之前。

当然，也有一些人推荐利用内部类持有静态对象的方式实现，其理论依据是对象初始化过程中隐含的初始化锁（有兴趣的话你可以参考[Jls-12.4.2](#) 中对LC的说明），这种和前面的双检锁实现都能保证线程安全，不过语法稍显晦涩，未必有特别的优势。

```
public class Singleton {
    private Singleton(){}
    public static Singleton getSingleton(){
        return Holder.singleton;
    }

    private static class Holder {
        private static Singleton singleton = new Singleton();
    }
}
```

所以，可以看出，即使是看似最简单的单例模式，在增加各种高需求之后，同样需要非常多的实现考量。

上面是比较学术的考察，其实实践中未必需要如此复杂，如果我们看Java核心类库自己的单例实现，比如[java.lang.Runtime](#)，你会发现：

- 它并没有使用复杂的双检锁之类。
- 静态实例被声明为`final`，这是被通常实践忽略的，一定程度保证了实例不被篡改（[专栏第6讲](#)介绍过，反射之类可以绕过私有访问限制），也有有限的保证执行顺序的语义。

```
private static final Runtime currentRuntime = new Runtime();
private static Version version;
// ...
public static Runtime getRuntime() {
    return currentRuntime;
}
/** Don't let anyone else instantiate this class */
private Runtime() {}
```

前面说了不少代码实践，下面一起来简要看看主流开源框架，如Spring等如何在API设计中使用设计模式。你至少要有个大体的印象，如：

- [BeanFactory](#)和[ApplicationContext](#)应用了工厂模式。
- 在Bean的创建中，Spring也为不同scope定义的对象，提供了单例和原型等模式实现。
- 我在[专栏第6讲](#)介绍的AOP领域则是使用了代理模式、装饰器模式、适配器模式等。
- 各种事件监听器，是观察者模式的典型应用。
- 类似[JdbcTemplate](#)等则是应用了模板模式。

今天，我与你回顾了设计模式的分类和主要类型，并从Java核心类库、开源框架等不同角度分析了其采用的模式，并结合单例的不同实现，分析了如何实现符合线程安全等需求的单例，希望可以对你的工程实践有所帮助。另外，我想最后补充的是，设计模式也不是银弹，要避免滥用或者过度设计。

## 一课一练

关于设计模式你做到心中有数了吗？你可以思考下，在业务代码中，经常发现大量XXFacade，外观模式是解决什么问题？适用于什么场景？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



sunlight001

2018-06-05

结合流行的开源框架，或者自己的项目学设计模式是很好的办法，生学很容易看不懂学不下去，`xxxFacade`是门面模式，为复杂的逻辑提供简单的借口，设计模式学的时候还能明白，但是用的时候就不知道该怎么用了，我们怎么在项目中使用设计模式呢？

作者回复

不必为了模式而用，优先解决开发、维护中的痛点

2018-06-06

公号-Java大前端

2018-06-05

门面模式形象上来说就是在原系统之前放置了一个新的代理对象，只能通过该对象才能使用该系统，不再允许其它方式访问该系统。该代理对象封装了访问原系统的所有规则和接口方法，提供的API接口较之使用原系统会更加的简单。

举例：`JUnitCore`是`JUnit`类的`Facade`模式的实现类，外部使用该代理对象与`JUnit`进行统一交互，驱动执行测试代码。

使用场景：当我们希望封装或隐藏原系统；当我们使用原系统的功能并希望增加一些新的功能；编写新类的成本远小于所有学会使用或者未来维护原系统所需的成本；

缺点：违反了开闭原则。如有扩展，只能直接修改代理对象。

作者回复

不错

2018-06-06

李昭文TSOS

2018-06-05

为什么我去查`Runtime`的源码，`currentRuntime`没有被`final`修饰呢？

作者回复

什么版本？我这是最新的

2018-06-06

云学

2018-06-09

最开始迷惑设计模式，后来眼中没有模式，其实那本经典的设计模式的书的第一章就非常明确的指出设计模式不是银弹，总感觉Java语言写的程序比C++更重，很多代码都是无用的装饰

李志博

2018-06-05

`Spring` 内部的`asm` 模块 用到了访问者模式

SinO

2018-06-05

有一点理解不太一致，单例模式`double check`中`synchronized`就已经可以提供可见性，`volatile`的作用主要体现在禁指令重排！

作者回复

不冲突，`sync`也不是必然走到

2018-06-06

润兹

2018-06-05

在没用`Facade`之前，为了完成某个功能需要调用各子系统的各方法进行组合才能完成，用了`Facade`之后相当于把多个方法调用聚合成了一个方法，方便用户调用。

田维俊

2018-06-05

公司项目是一个基于`springboot`，`mybatis`开发的`web`后端管理项目。现在的问题是，不同角色登录到系统看到的模块和模块里面的数据是不一样的，有时虽然看到的模块一样，但是由于角色不一样，所以显示的数据是不一样。在这样的情况下，会经常在`service`层方法里面判断角色然后改变`mapper`的数据操作条件或调用`mapper`的不同方法。由于在`service`层频繁的判断角色感觉很不雅，新增角色就要加判断，哎，感觉可以用策略设计模式，可是不知道怎么具体设计。

星空

2018-06-05

外观模式为子系统中一组接口提供一个统一访问的接口，降低了客户端与子系统之间的耦合，简化了系统复杂度。缺点是违反了开闭原则。适用于为一系列复杂的子系统提供一个友好简单的入口，将子系统与客户端解耦。公司基础`paas`平台用到了外观模式，具体是定义一个`ServiceFacade`，然后通过继承众多`xxService`，对外提供`xxService`的服务。

作者回复

业务开发很普遍

2018-06-06

刘杰

2018-07-11

老师，我有个疑问，单例那里可否不用volatile，初始化时  
`Singleton temp=new Singleton();`  
`this.singleton=temp;`  
 这样能保证初始化完成才赋值吗？

Miaozhe

2018-06-08  
 我理解Facade模式，微服务应用场景，如：Nginx对系统子服务进行管理和IP反向代理，提供统一的服务，就是屏蔽外部系统对内部服务的具体实现，以及各微服务的部署虚拟机和URL。  
 再者，容器Docker技术，我认为这是Facade模式，通过镜像把应用相关的组件和配置都预置好，发布这个服务时，直接启动容器，用户不用关心里面的任何细节。

杨老师看看，我分析的对吗？

Walter

2018-06-07  
 外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。它向现有的系统添加一个接口，来隐藏系统的复杂性。  
 这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更容易使用。  
 主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。  
 何时使用：1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个“接待员”即可。2、定义系统的入口。  
 如何解决：客户端不与系统耦合，外观类与系统耦合。  
 关键代码：在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。  
 应用实例：1、去医院看病，可能要去挂号、门诊、划价、取药，让患者或者患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。2、JAVA 的三层开发模式。  
 优点：1、减少系统相互依赖。2、提高灵活性。3、提高了安全性。  
 缺点：不符合开闭原则。如果要修改东西很麻烦，继承重写都不合适。  
 使用场景：1、为复杂的模块或子系统提供外界访问的模块。2、子系统相对独立。3、预防低水平人员带来的风险。  
 注意事项：在层次化结构中，可以使用外观模式定义系统中每一层的入口。

作者回复

不错，业务系统多见

锐

2018-06-06  
 设计模式也不是很弹，要避免滥用或者过度设计。这句话深有体会，为了使用设计模式而使用，有时很头疼

雷霹雳的爸爸

2018-06-06  
 单态这里有一个问题问老师，如果不`double check`的话，仅在声明静态成员的同时即实例化之，那么是否就不用`volatile`关键字修饰这个字段了？为什么？

softpower2018

2018-06-05  
 通过封装的方式，对外屏蔽内部复杂业务逻辑，实现使用方与具体实现的分离。门面模式

yearning

2018-06-05  
**Facade（外观模式）**  
 接口隔离模式，处理组件外部客户程序和组件中各种复杂的子系统高耦合情况，定义一个高层接口，为子系统中的一组接口提供一个一致（稳定）的界面，使得更简单的使用。  
 facade简化整个组件系统的接口，同时子系统的任何变化都不会影响到facade接口。

有一个更简单的称呼，门面模式，打个比方说。你去商店，你只需要告诉店员，你需要什么，至于商店中复杂的采购系统，库存系统，收银系统一概对你不可见。

在经常使用的hibernate，当我们想插入一条用户信息，`facade接口中insert(User user)`，我们只要传递User对象，至于背后的操作对外部调用是不可见。

facade模式是从架构的层次去看整个系统，而是一两个类之间单纯解耦。

Geek\_028e77

2018-06-05  
**facade模式** 主要屏蔽系统内部细节实现，通过facade模式封装统一的接口 提供给外部调用着。有一个优势，当内部系统做变更 优化时，这对外部调用者来说是透明的，一定程度上降低了系统间耦合性...个人理解

wutao

2018-06-05  
**Spring中还用到了策略模式吧**  
 作者回复

当然，列出的只是简要说明





## 第15讲 | synchronized和ReentrantLock有什么区别呢？

2018-06-07 杨晓峰



第15讲 | synchronized和ReentrantLock有什么区别呢？  
- 00:17 / 09:36

从今天开始，我们将进入Java并发学习阶段。软件并发已经成为现代软件开发的基础能力，而Java精心设计的高效并发机制，正是构建大规模应用的基础之一，所以考察并发基本功也成为各个公司面试Java工程师的必选项。

今天我要问你的是，[synchronized和ReentrantLock有什么区别？有人说synchronized最慢，这话靠谱吗？](#)

## 典型回答

synchronized是Java内建的同步机制，所以也有人称其为Intrinsic Locking，它提供了互斥的语义和可见性，当一个线程已经获取当前锁时，其他试图获取的线程只能等待或者阻塞在那里。

在Java 5以前，synchronized是仅有的同步手段，在代码中，synchronized可以用来修饰方法，也可以使用在特定的代码块儿上，本质上synchronized方法等同于把方法全部语句用synchronized块包起来。

ReentrantLock，通常翻译为再入锁，是Java 5提供的锁实现，它的语义和synchronized基本相同。再入锁通过代码直接调用lock()方法获取，代码书写也更加灵活。与此同时，ReentrantLock提供了很多实用的方法，能够实现很多synchronized无法做到的细节控制，比如可以控制fairness，也就是公平性，或者利用定义条件等。但是，编码中也需要注意，必须要注意unlock()方法释放，不然就会一直持有该锁。

synchronized和ReentrantLock的性能不能一概而论，早期版本synchronized在很多场景下性能相差较大，在后续版本进行了较多改进，在低竞争场景中表现可能优于ReentrantLock。

## 考点分析

今天的题目是考察并发编程的常见基础题，我给出的典型回答算是一个相对全面的总结。

对于并发编程，不同公司或者面试官面试风格也不一样，有个别大厂喜欢一直追问你相关机制的扩展或者底层，有的喜欢从实用角度出发，所以你在准备并发编程方面需要一定的耐心。

我认为，锁作为并发的基础工具之一，你至少需要掌握：

- 理解什么是线程安全。
- synchronized、ReentrantLock等机制的基本使用与案例。

更进一步，你还需要：

- 掌握synchronized、ReentrantLock底层实现；理解锁膨胀、降级；理解偏向锁、自旋锁、轻量级锁、重量级锁等概念。
- 掌握并发包中java.util.concurrent.lock各种不同实现和案例分析。

## 知识扩展

专栏前面几期穿插了一些并发的概念，有同学反馈理解起来有点困难，尤其对一些并发相关概念比较陌生，所以在这一讲，我也对会一些基础的概念进行补充。

首先，我们需要理解什么是线程安全。

我建议阅读Brian Goetz等专家撰写的《Java并发编程实战》（Java Concurrency in Practice），虽然可能稍显深奥，但不可否认这是一本非常系统和全面的Java并发编程书籍。按照其中的定义，线程安全是一个多线程环境下正确性的概念，也就是保证多线程环境下共享的、可修改的状态的正确性，这里的状态反映在程序中其实可以看作是数据。

换个角度来看，如果状态不是共享的，或者不是可修改的，也就不存在线程安全问题，进而可以推导出保证线程安全的两个办法：

- 封装：通过封装，我们可以将对象内部状态隐藏、保护起来。

- 不可变：还记得我们在[专栏第3讲](#)强调的final和immutable吗，就是这个道理，Java语言目前还没有真正意义上的原生不可变，但是未来也许会引入。

线程安全需要保证几个基本特性：

- 原子性，简单说就是相关操作不会中途被其他线程干扰，一般通过同步机制实现。
- 可见性，是一个线程修改了某个共享变量，其状态能够立即被其他线程知晓，通常被解释为将线程本地状态反映到内存上，volatile就是负责保证可见性的。
- 有序性，是保证线程内串行语义，避免指令重排等。

可能有点晦涩，那么我们看看下面的代码段，分析一下原子性需求体现在哪里。这个例子通过取两次数值然后进行对比，来模拟两次对共享状态的操作。

你可以编译并执行，可以看到，仅仅是两个线程的低度并发，就非常容易碰到former和latter不相等的情况。这是因为，在两次取值的过程中，其他线程可能已经修改了sharedState。

```
public class ThreadSafeSample {
    public int sharedState;
    public void nonSafeAction() {
        while (sharedState < 100000) {
            int former = sharedState++;
            int latter = sharedState;
            if (former != latter - 1) {
                System.out.printf("Observed data race, former is " +
                    former + ", " + "latter is " + latter);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadSafeSample sample = new ThreadSafeSample();
        Thread threadA = new Thread(){
            public void run(){
                sample.nonSafeAction();
            }
        };
        Thread threadB = new Thread(){
            public void run(){
                sample.nonSafeAction();
            }
        };
        threadA.start();
        threadB.start();
        threadA.join();
        threadB.join();
    }
}
```

下面是在我的电脑上的运行结果：

```
C:\>c:\jdk-9\bin\java ThreadSafeSample
Observed data race, former is 13097, latter is 13099
```

将两次赋值过程用synchronized保护起来，使用this作为互斥单元，就可以避免别的线程并发的去修改sharedState。

```
synchronized (this) {
    int former = sharedState++;
    int latter = sharedState;
    // ...
}
```

如果用javap反编译，可以看到类似片段，利用monitorenter/monitorexit对实现了同步的语义：

```
11: astore_1
12: monitorenter
13: aload_0
14: dup
15: getfield    #2           // Field sharedState:I
18: dup_x1
-
56: monitorexit
```

我会在下一讲，对synchronized和其他锁实现的更多底层细节进行深入分析。

代码中使用synchronized非常便利，如果用来修饰静态方法，其等同于利用下面代码将方法体囊括进来：

```
synchronized (ClassName.class) {}
```

再来看看ReentrantLock。你可能好奇什么是再入？它是表示当一个线程试图获取一个它已经获取的锁时，这个获取动作就自动成功，这是对锁获取粒度的一个概念，也就是锁的持有是以线程为单位而不是基于调用次数。Java锁实现强调再入性是为了和pthread的行为进行区分。

再入锁可以设置公平性（fairness），我们在创建再入锁时选择是否是公平的。

```
ReentrantLock fairLock = new ReentrantLock(true);
```

这里所谓的公平性是指在竞争场景中，当公平性为真时，会倾向于将锁赋予等待时间最长的线程。公平性是减少线程“饥饿”（个别线程长期等待锁，但始终无法获取）情况发生的一个办法。

如果使用synchronized，我们根本无法进行公平性的选择，其永远是不公平的，这也是主流操作系统线程调度的选择。通用场景中，公平性未必有想象中的那么重要，Java默认的调度策略很少会导致“饥饿”发生。与此同时，若要保证公平性则会引入额外开销，自然会导致一定的吞吐量下降。所以，我建议只有当你的程序确实有公平性需要的时候，才有必要指定它。

我们再从日常编码的角度学习下再入锁。为保证锁释放，每一个lock()动作，我建议都立即对应一个try-catch-finally，典型的代码结构如下，这是个良好的习惯。

```
ReentrantLock fairLock = new ReentrantLock(true); // 这里是演示创建公平锁，一般情况下不需要。
try {
    // do something
} finally {
    fairLock.unlock();
}
```

ReentrantLock相比synchronized，因为可以像普通对象一样使用，所以可以利用其提供的各种便利方法，进行精细的同步操作，甚至是实现synchronized难以表达的用例，如：

- 带超时的获取锁尝试。
- 可以判断是否有线程，或者某个特定线程，在排队等待获取锁。
- 可以响应中断请求。
- ...

这里我特别想强调条件变量（java.util.concurrent.Condition），如果说ReentrantLock是synchronized的替代选择，Condition则是将wait、notify、notifyAll等操作转化为相对应的对象，将复杂而晦涩的同步操作转变为直观可控的对象行为。

条件变量最典型的应用场景就是标准类库中的ArrayBlockingQueue等。

我们参考下面的源码，首先，通过再入锁获取条件变量：

```
/** Condition for waiting takes */
private final Condition notEmpty;

/** Condition for waiting puts */
private final Condition notFull;

public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull = lock.newCondition();
}
```

两个条件变量是从同一再入锁创建出来，然后使用在特定操作中，如下面的take方法，判断和等待条件满足：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
        return dequeue();
    } finally {
        lock.unlock();
    }
}
```

当队列为空时，试图take的线程的正确行为应该是等待入队发生，而不是直接返回，这是BlockingQueue的语义，使用条件notEmpty就可以优雅地实现这一逻辑。

那么，怎么保证入队触发后续take操作呢？请看enqueue实现：

```
private void enqueue(E e) {
    // assert lock.isHeldByCurrentThread();
    // assert lock.getHoldCount() == 1;
    // assert items[putIndex] == null;
    final Object[] items = this.items;
```

```

        items[putIndex] = e;
        if (++putIndex == items.length) putIndex = 0;
        count++;
        notEmpty.signal(); // 通知等待的线程，非空条件已经满足
    }
}

```

通过signal/await的组合，完成了条件判断和通知等待线程，非常顺畅就完成了状态流转。注意，signal和await成对调用非常重要，不然假设只有await动作，线程会一直等待直到被打断（interrupt）。

从性能角度，synchronized早期的实现比较低效，对比ReentrantLock，大多数场景性能都相差较大。但是在Java 6中对其进行非常多的改进，可以参考性能对比，在高竞争情况下，ReentrantLock仍然有一定优势。我在下一讲进行详细分析，会更有助于理解性能差异产生的内在原因。在大多数情况下，无需纠结于性能，还是考虑代码书写结构的便利性、可维护性等。

今天，作为专栏进入并发阶段的第一讲，我介绍了什么是线程安全，对比和分析了synchronized和ReentrantLock，并针对条件变量等方面结合案例代码进行了介绍。下一讲，我将对锁的进阶内容进行源码和案例分析。

#### 一课一练

关于今天我们讨论的synchronized和ReentrantLock你做到心中有数了吗？思考一下，你使用过ReentrantLock中的哪些方法呢？分别解决什么问题？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮助到他。



公众号-Java大前端

ReentrantLock是Lock的实现类，是一个互斥的同步器，在多线程高竞争条件下，ReentrantLock比synchronized有更加优异的性能表现。

2018-06-07

1 用法比较  
Lock使用起来比较灵活，但是必须有释放锁的配合动作  
Lock必须手动获取与释放锁，而synchronized不需要手动释放和开启锁  
Lock只适用于代码块锁，而synchronized可用于修饰方法、代码块等。

2 特性比较  
ReentrantLock的优势体现在：  
具备尝试非阻塞地获取锁的特性：当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁  
能被中断地获取锁的特性：与synchronized不同，获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放  
超时地获取锁的特性：在指定的时间范围内获取锁；如果截止时间到了仍然无法获取锁，则返回

3 注意事项  
在使用ReentrantLock类的时候，一定要注意三点：  
在finally中释放锁，目的是保证在获取锁之后，最终能够被释放  
不要将获取锁的过程写在try块内，因为在获取锁时发生了异常，异常抛出的同时，也会导致锁无法被释放。  
ReentrantLock提供了一个newCondition()方法，以便用户在同一锁的情况下可以根据不同的情况执行等待或唤醒的动作。

逐梦之音

2018-06-07

一直在研究JUC方面的。所有的Lock都是基于AQS来实现了。AQS和Condition各自维护了不同的队列，在使用lock和condition的时候，其实就是两个队列的互相移动。如果我们想自定义一个同步器，可以实现AQS。它提供了获取共享锁和互斥锁的方式，都是基于对state操作而言的。ReentrantLock这个是可重入的，其实要弄明白它为锁可重入的呢，咋实现的呢。其实它内部自定义了同步器Sync，这个又实现了AQS，同时又实现了AQS，而后者还提供了一种互斥锁持有的方式。其实就是每次获取锁的时候，看下当前维护的那个线程和当前请求的线程是否一样，一样就可重入了。

作者回复

正解

Kyle

2018-06-07

最近刚看完《Java 并发编程实战》，所以今天看这篇文章觉得丝毫不费力气。开始觉得，极客时间上老师讲的内容毕竟篇幅有限，更多的还是需要我们课后去深入钻研。希望老师以后讲完课也能够适当提供些参考书目，谢谢。

作者回复

后面会对实现做些源码分析，其实还有各种不同的锁...

BY

2018-06-07

要是早看到这篇文章，我上次面试就过了。。

作者回复

加油

灰飞灰猪不会灰飞烟灭

ReentrantLock 加锁的时候通过cas算法，将线程对象放到一个双向链表中，然后每次取出链表中的头节点，看这个节点是否和当前线程相等。是否相等比较的是线程的ID。  
老师我理解的对不对啊？

作者回复

嗯，并发库里都是靠自己的synchronizer

木瓜芒果

杨老师，您好，synchronized在低竞争场景下可能优于reentrantlock，这里的什么程度算是低竞争场景呢？

作者回复

这个精确的标准我还真不知道，我觉得可以这么理解：如果大部分情况，每个线程都不需要真的获取锁，就是低竞争；反之，大部分都要获取锁才能正常工作，就是高竞争

xinfangke

老师 问你个问题 在spring中 如果标注一个方法的事务隔离级别为序列化 而数据库的隔离级别是默认的隔离级别 此时此方法中的更新 插入语句是如何执行的？能保证并发不出错吗

作者回复

这个我没用过，哪位读者熟悉？

sunlight001

老师这里说的低并发和高并发的场景，大致什么数量级的算低并发？我们做管理系统中用到锁的情况基本都算低并发吧

作者回复

还真不知道有没有具体标准，但从逻辑上，低业务量不一定是“低竞争”，可能因为程序设计原因变成了“高竞争”

Daydayup

我用过读写分离锁，读锁保证速度，写锁保证安全问题。再入锁还是挺好用的。老师写的很棒，学到不少知识。感谢  
作者回复

非常感谢

Daydayup

我用过读写分离锁，读锁保证速度，写锁保证安全问题。再入锁还是挺好用的。老师写的很棒，学到不少知识。感谢

李飞

老师，可以问我一个课外题吗。具备怎样的能力才算是java高级开发

Libra

希望后面能讲下lock源码整个的设计思想。

wang\_bo

该怎么系统学习java并发？

Allen

为什么ReentrantLock会如此高效？

张小小的席大da

杨老师 很感谢 在您这了解到很多以前没注意的知识点 时刻关注着您 希望您会一直讲下去

猪哥灰

为了研究java的并发，我先把考研时候的操作系统教材拿出来再仔细研读一下，可见基础之重要性，而不管是什么语言，万变不离其宗

飞鱼

之前有被问到synchronize和ReentrantLock底层实现上的区别，笼统的答了下前者是基于JVM实现的，后者依赖于CPU底层指令的实现，关于这个，请问有更详细的解答吗？

作者回复

synchronized已经介绍了，后面我会介绍AQS, reentrantlock等多种同步结构都是利用它实现的；其实你说的靠计算机之类也对，我想你的意思是cas的实现？

云学

锁是针对数据的，不是针对代码，一个数据一把锁，synchronize似乎违背了这一原则

云学

看完还是觉得c++11的Lockguard比较优雅，难怪耗子哥说学习java是为了更好的用c++

作者回复

互相影响，底层也有很多一致的地方，类似jmm之类也是c++掉的坑，别人吸取了教训

Miaozhe

杨老师，问个问题，看网上有说Condition的await和signal方法，等同于Object的wait和notify，看了一下源码，没有直接的关系。

2018-06-08

2018-06-07

2018-06-08

2018-06-19

2018-06-19

2018-06-08

2018-06-08

2018-06-07

2018-06-07

2018-06-13

2018-06-13

2018-06-13

2018-06-08

2018-06-07

2018-07-08

2018-07-02

2018-06-29

2018-06-17

2018-06-19

2018-06-15

2018-06-14

2018-06-12

ReentrantLock是基于双向链表的对接和CAS实现的，感觉比Object增加了很多逻辑，怎么会比Synchronized效率高？有疑惑。

作者回复

你看到的很对，如果从单个线程做的事来看，也许并没有优势，不管是空间还是时间，但ReentrantLock这种所谓cas，或者叫lock-free，方式的好处，在于高竞争情况的扩展性，而原来那种频繁的上下文切换则会导致吞吐量迅速下降。

Miaoze

2018-06-12

Reentrant Lock的示例的代码好像不完整。

liyacai

2018-06-12

老师，最近在看并发包的源码，发现8与9的源码改了不少，应该以哪个版本为重呢

liyacai

2018-06-08

```
在阅读代码时发现 ReentrantLock中nonfairTryAcquire 中对state做了溢出判断，我想问一下，除了死循环，还有什么情况会导致溢出呢？
if (current == getExclusiveOwnerThread()) {
    int nextc = c + acquires;
    if (nextc < 0) // overflow
        throw new Error("Maximum lock count exceeded");
    setState(nextc);
    return true;
}
```

徐金泽

2018-06-08

补充一点，Syc的静态方法和syc(class)确实是一样的，但是前者是在方法前加syc的flag，后者在反编译后的代码中看不到。所以我查阅了hotspot的文档和代码，确定这个细节处理是有vm做的。两者实际运行，确实是一样的处理。

作者回复

对，字节码不一样，但语义是一样

2018-06-08

java爱好者

2018-06-08

老师，怎么判断一个队列是有界或无界，arrayblockingqueue是有界

作者回复

2018-06-08

这个每个queue的javadoc都明确说明了，有什么疑问？

雷霹雳的爸爸

2018-06-08

真不记得自己用过，诚如老师所讲，其有特殊语义能力，如超时，公平性等，但窃以为别给自己添乱最好，万一忘了unlock，话说他这玩意儿为啥不设计个类似try...with...resource的语法糖？估计就是为了把加锁解锁的语法能分散在不同子进程中撰写使用的考虑？可问题是如果都写成那样了，是否有说明自己程序设计上内聚性不够呢？嗯，再看点源码来找答案吧...话说JCP真是好书，学不学的读读资本论之后就释然了，如果真能顺着它仔细读下来，发现它针对特定要子还会随着内容深入给出不同解读，不禁感叹，也许在利用较低级别的通俗原话时，很有可能是对并发包里面一些现成工具类缺乏了解或者是对真正的并发问题缺乏深入理解造成的。虽然自己看起来也就是个CRUD开发的老佃户命了，还是非常期待老师后面的王冕，毕竟咱还是有赖向地主阶层的心

作者回复

有些场景，就类似blockingQueue，用sync表达要吃力些，甚至有时候是表达不出来的

2018-06-08

一个帅哥

2018-06-08

从书写角度：reentrantlock 需要手动获取和释放锁，而synchronized不需要手动获取和释放。所以，reentrantlock有trylock 和lockinterruptible，所以对锁的操作更灵活。从功能的角度看，reentrantlock支持公平锁和非公平锁 而synchronized 仅支持非公平锁。

kal

2018-06-07

在大部分并发编程资料都是将ReentrantLock翻译为“重入锁”。

谢

2018-06-07

ReentrantLock相比Synchronized，可以实现更多的锁细节。

超时的锁获取  
可以判断是否有线程在等待锁  
可以响应中断请求

灰灰猪不会灰飞烟灭

2018-06-07

老师，signal和await和notify wait有啥区别呢？  
还有lock方式放入双向链表中的node，是不是按照线程对象（对象地址）进行比较的啊？  
就是如何判断当前线程是否获得锁是不是按照线程对象地址啊。谢谢老师

作者回复

有点类似；使用并发库，大部分情况下不再需要调用Object的notify wait之类，简化了很多；  
后面的问题我没太看明白

2018-06-07





周末福利 | 谈谈我对Java学习和面试的看法

2018-06-09 杨晓峰



 周末福利 | 谈谈我对Java学习和面试的看法  
杨晓峰

- 00:21 / 07:18

你好，我是杨晓峰。今天是周末，我们稍微放松一下来聊聊“Java核心技术”之外的内容，正好也借这个机会，兑现一下送出学习奖励礼券的承诺。我在每一讲后面都留下了一道思考题，希望你通过学习，结合自身工作实际，能够认真思考一下这些问题，一方面起到检验学习效果的作用，另一方面可以查漏补缺，思考一下这些平时容易被忽略的面试考察点。我并没有给出这些思考题的答案，希望你通过专栏学习或者查阅其他资料进行独立思考，将自己思考的答案写在留言区与我和其他同学一起交流，这也是提升自己重要的方法之一。

截止到今天，专栏已经更新了15讲，走完了基础模块正式进入进阶模块。现在也正是一个很好的时机停下来回顾一下基础部分的知识，为后面进阶的并发内容打好基础。在这里，我也分享一下我对Java学习和面试的看法，希望对你有所帮助。

首先，**有同学反馈说专栏有的内容看不懂**。我在准备专栏文章的时候对一些同学的基础把握不太准确，后面的文章我进行了调整，将重点技术概念进行讲解，并为其他术语添加链接。

再来说说这种情况，**有人总觉得Java基础知识都已经讲烂了，还有什么可学的？**

对于基础知识的掌握，有的同学经常是“知其然而不知其所以然”，看到几个名词听说过就以为自己掌握了，其实不然。至少，我认为应该能够做到将自己“掌握”的东西，准确地表达出来。

爱因斯坦曾经说过，“如果你不能把它简单地解释出来，那说明你还没有很好地理解它”。了解-掌握-精通，这是我们对事物掌握的一个循序渐进的过程。从自己觉得似乎懂了，到能够说明白，再到能够自然地运用它，甚至触类旁通，这是不断提高的过程。

在专栏学习中，如果有些术语很陌生，那么了解它就达到了学习目的，如果能够理解透彻达到掌握的程度当然更好。乐观点来看，反正都是有收获，也完全不必过分担心。

从学习技巧的角度，每个人都有自己的习惯，我个人喜欢动手实践以及与人进行交流。

- 动手实践是必要一步，如果连上手操作都不肯，你会发现自己的理解很难有深度。
- 在交流的过程中你会发现，很多似是而非的理解，竟然在试图组织语言的时候，突然想明白了，而且别人的观点也验证了自己的判断。技术领域尤其如此，把自己的理解整理成文字，输出、交流是个非常好的提高方法，甚至我认为这是技术工作者成长的必经之路。

#### 再来聊聊针对技术底层，我们是否有必要去阅读源代码？

阅读源代码当然是个好习惯，理解高质量的代码，对于提高我们自己的分析、设计等能力至关重要。

- 根据实践统计，工程师实际工作中，阅读代码的时间其实大大超过写代码的时间，这意味着阅读、总结能力，会直接影响我们的工作效率！这东西有没有捷径呢，也许吧，我的心得是：“无他，但手熟尔”。
- 参考别人的架构、实现，分析其历史上掉过的坑，这是天然的好材料，具体阅读时可以从其修正过的问题等角度入手。
- 现代软件工程，节奏越来越快，需求复杂而多变，越来越凸显出白盒式的重要性。快速定位问题往往需要黑盒结合白盒能力，对内部一无所知，可能就没有思路。与此同时，通用平台、开源框架，不见得能够非常符合自己的业务需求，往往只有深入源代码层面进行定制或者自研，才能实现。我认为这也是软件工程师地位不断提高的原因之一。

那么，**源代码需要理解到什么程度呢**？对于底层技术，这个确实是比较有争议的问题。我个人并不觉得什么东西都要理解底层，懂当然好，但不能代表一切，毕竟知识和能力是有区别的，当然我们也要尊重面试官的要求。我个人认为，不是所有做Java开发的人都需要读JVM源代码，虽然我在专栏中提供了一些底层源代码解读，但也只是希望真的有兴趣、有需要的工程师跟进学习。对于大多数开发人员，了解一些源代码，至少不会在面试问到的时候完全没有准备。

关于阅读源代码和理解底层，我有些建议：

- 带着问题和明确目的去阅读，比如，以debug某个问题的角度，结合实践去验证，让自己能够感到收获，既加深理解，也有实际帮助，激励我们坚持下来。
- 一定要有输出，至少要写下来，整理心得、交流、验证、提高。这和我们日常工作是类似的，千万不要做了好长一段时间后和领导说，没什么结论。

大家大都是工程师，不是科学家，软件开发中需要分清表象、行为（behavior），还是约定（specification）。喜欢源代码、底层是好的，但是一定要区分其到底是实现细节，还是规范的承诺，因为如果我们的程序依赖于表现，很有可能带来未来维护的问题。

我前面提到了白盒方式的重要性，但是，需要慎重决定对内部的依赖，分清是Hack还是Solution。出来混，总是要还的！如果以某种hack方式解决问题，临时性的当然可以，长久会积累并成为升级的障碍，甚至堆积起来愈演愈烈。比如说，我在实验Cassandra的时候，发现它在并发部分引用了Unsafe.monitorEnter() / monitorExit()，这会导致它无法平滑运行在新版的JDK上，因为相应内部API被移除了，比较幸运的是这个东西有公共API可以替代。

最后谈谈我在面试时会看中候选人的哪些素质和能力。

结合我在实际工作中的切身体会，面试时有几个方面我会特别在乎：

- 技术素养好，能够进行深度思考，而不是跳脱地夸夸其谈，所以我喜欢问人家最擅长的东西，如果在最擅长的领域尚且不能仔细思考，怎么能保证在下一份工作中踏实研究呢。当然这种思考，并不是说非要死扣底层和细节，能够看出业务中平凡事情背后的工程意义，同样是不错的。毕竟，除了特别的岗位，大多数任务，如果有良好的技术素养和工作热情，再配合一些经验，基本也就能够保证胜任了。
- 职业精神，是否表现出认真对待每一个任务。我们是职场打拼的专业人士，不是幼儿园被呵护的小朋友。如果有人太挑活儿，团队往往就无法做到基本的公平。有经验的管理角色，大多是把自己的管理精力用在团队的正面建设，而不是把精力浪费在拖团队后腿的人身上，难以协作的人，没有人会喜欢。有人说你的职业高度取决于你“填坑”的能力，我觉得很有道理。现实工作中很少有理想化的完美任务，既目标清晰又有挑战，恰好还是我擅长，这种任务不多见。能够主动地从不清晰中找出清晰，切实地解决问题，是非常重要的能力。
- 是否hands-on，是否主动。我一般不要求当前需要的方面一定是很hands-on，但至少要表现出能够做到。

下面放出中奖名单和精选留言，送出15元学习奖励礼券，希望我的《Java核心技术36讲》不仅能带你走进大厂Java面试场景，还能帮你温故知新基础知识，构建你的Java知识体系。也欢迎你在这里与我交流面试、学习方面的困惑或心得，一起畅所欲言、共同进步。

中奖名单	
昵称	奖励
石头狮子	15元无门槛学习奖励券
Woj	15元无门槛学习奖励券
kursk.ye	15元无门槛学习奖励券
Miaozhe	15元无门槛学习奖励券
肖一林	15元无门槛学习奖励券
曹铮	15元无门槛学习奖励券
雷霹雳的爸爸	15元无门槛学习奖励券
vash_ace	15元无门槛学习奖励券
Walter	15元无门槛学习奖励券
I am a psycho	15元无门槛学习奖励券
majict4	15元无门槛学习奖励券



## 石头狮子

写于 2018/05/05

1. 一次编译，到处运行。jvm 层面封装了系统 API，提供不同系统一致的调用行为。减少了为适配不同操作系统，不同架构的带来的工作量。
2. 垃圾回收，降低了开发过程中需要注意内存回收的难度。降低内存泄露出现的概率。虽然也带来了一些额外开销，但是足以弥补带来的好处。合理的分代策略，提高了内存使用率。
3. jit 与其他编译语言相比，降低了编译时间，因为大部分代码是运行时编译，避免了冷

代码在编译时也参与编译的问题。

提高了代码的执行效率，之前项目中使用过 lua 进行相关开发。由于 lua 是解释性语言，并配合使用了 lua-jit。开发过程中遇到，如果编写的 lua 代码是 jit 所不支持的会导致代码性能与可编译的相比十分低下。

引自：Java核心技术36讲  
第1讲 | 谈谈你对Java平台的理解？



识别二维码打开原文  
「极客时间」App

Woj

写于 2018/05/05

“一次编译、到处运行”说的是 Java 语言跨平台的特性，Java 的跨平台特性与 Java 虚拟机的存在密不可分，可在不同的环境中运行。比如说 Windows 平台和 Linux 平台都有相应的 JDK，安装好 JDK 后也就有了 Java 语

言的运行环境。其实 Java 语言本身与其他的编程语言没有特别大的差异，并不是说 Java 语言可以跨平台，而是在不同的平台都有可以让 Java 语言运行的环境而已，所以才有了 Java 一次编译，到处运行这样的效果。

严格的讲，跨平台的语言不止 Java 一种，但 Java 是较为成熟的一种。“一次编译，到处运行”这种效果跟编译器有关。编程语言的处理需要编译器和解释器。Java 虚拟机和 DOS 类似，相当于一个供程序运行的平台。

程序从源代码到运行的三个阶段：编码——编译——运行——调试。Java 在编译阶段则体现了跨平台的特点。编译过程大概是这样的：首先是将 Java 源代码转化成.CLASS 文件字节码，这是第一次编译。.class 文件就是可以到处运行的文件。然后 Java 字节码会被转化为目标机器代码，这是是由 JVM 来执行的，即 Java 的第二次编译。

“到处运行”的关键和前提就是 JVM。因为在第二次编译中 JVM 起着关键作用。在可以运行 Java 虚拟机的地方都内含着一个 JVM 操作系统。从而使 JAVA 提供了各种不同平台

上的虚拟机制，因此实现了“到处运行”的效果。需要强调的一点是，java 并不是编译机制，而是解释机制。Java 字节码的设计充分考虑了 JIT 这一即时编译方式，可以将字节码直接转化成高性能的本地机器码，这同样是虚拟机的一个构成部分。

引自：Java核心技术36讲

第1讲 | 谈谈你对Java平台的理解？



识别二维码打开原文  
「极客时间」App

kursk.ye

写于 2018/05/24

于是我 google 到了这篇文章,<http://www.kdgregory.com/index.php?page=java.refobj>，花了几 天（真的是几天，不是几小时）才基本读完，基本理解这几个 reference 的概念和作用，从这个角度来讲非常感谢作者。如果不是本文的介绍，我还

以为 GC 还是按照 reference counter 的原理处理，原来思路早变了。话说回来，《Java Reference Objects》真值得大家好好琢磨，相信可以回答很多人的问题，比如 strong reference , soft reference , weak reference 怎么互转，如果一个 obj 已经 = null, 就 obj = reference.get() 呀，再有，文章中用 weak reference 实现 canonicalizing map 改善内存存储效率，减小存储空间的例子，真是非常经典啊。也希望作者以后照顾一下低层次读者，写好技术铺垫和名词定义。顺便问一下大家是怎么留言的，在手机上打那么多字，还有排版是怎么处理的，我是先在电脑上打好字再 COPY 上来的，大家和我一样吗？

| 引自：Java核心技术36讲

| 第4讲 | 强引用、软引用、弱引用、幻象引用有什么区别？



识别二维码打开原文  
「极客时间」App

# Miaozhe

写于 2018/05/18

接着上个问题：

老师，问个问题：我自己定义一个类，重写 finalize 方法后，创建一个对象，被幻想引用，同时该幻想对象使用 ReferenceQueue。

当我这个对象指向 null，被 GC 回收后，ReferenceQueue 中没有改对象，不知道是什么原因？如果我把类中的 finalize 方法移除，ReferenceQueue 就能获取被释放的对象。

2018-05-17 作者回复文章图里阐明了，幻象引用 enqueue 发生在 finalize 之后，你查查是不是卡在 FinalReference queue 里了，那是实现 finalization 的地方

杨老师，我去查看了，Final reference 和 Reference 发现是 Reference Handle 线程在监控，但是 Debug 进出去，还是没有搞清

定你注。

不过，我又发现类中自定义得 `Finalize`，如果是空的，正常。如果类中有任何代码，都不能进入 `Reference Queue`，怀疑是对象没有被 `GC` 回收。

引自：Java核心技术36讲

第4讲 | 强引用、软引用、弱引用、幻象引用有什么区别？



识别二维码打开原文  
「极客时间」App

肖一林

写于 2018/05/17

提一些建议：应该从两条线讲这个问题，一条从代理模式，一条从反射机制。不要老担心篇幅限制讲不清问题，废话砍掉一些，深层次的内在原理多讲些（比如 `asm`），容易自学的扩展知识可以用链接代替

代理模式（通过代理静默地解决一些业务无关

的问题，比如远程、安全、事务、日志、资源关闭……让应用开发者可以只关心他的业务）

静态代理：事先写好代理类，可以手工编写，也可以用工具生成。缺点是每个业务类都要对应一个代理类，非常不灵活。

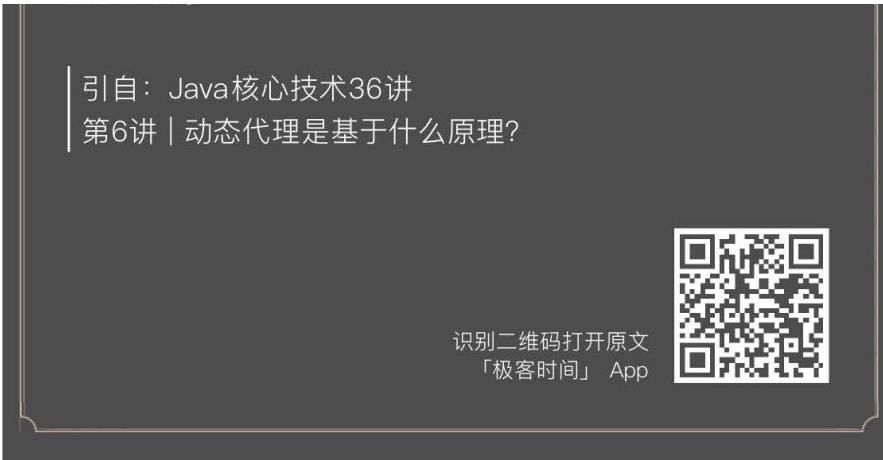
动态代理：运行时自动生成代理对象。缺点是生成代理对象和调用代理方法都要额外花费时间。

JDK 动态代理：基于 Java 反射机制实现，必须要实现了接口的业务类才能用这种方法生成代理对象。新版本也开始结合 ASM 机制。

cglib 动态代理：基于 ASM 机制实现，通过生成业务类的子类作为代理类。

Java 发射机制的常见应用：动态代理（AOP、RPC）、提供第三方开发者扩展能力（Servlet 容器，JDBC 连接）、第三方组件创建对象（DI）……

我水平比较菜，希望多学点东西，希望比免费知识层次更深些，也不光是为了面试，所以提提建议。



曹铮

写于 2018/05/22

既然是 Java 的主题，那就用  
PriorityBlockingQueue 吧。

如果是真实场景肯定会考虑高可用能持久化的  
方案。

其实我觉得应该参考银行窗口，同时三个窗  
口，就是三个队列，银台就是消费者线程，某  
一个窗口 vip 优先，没有 vip 时也为普通客  
户服务。要实现，要么有个 dispatcher，要  
么保持 vip 通道不许普通进入，vip 柜台闲时  
从其他队列偷

引自：Java核心技术36讲  
第8讲 | 对比Vector、ArrayList、LinkedList有何区别？



识别二维码打开原文  
「极客时间」App

# 雷霹雳的爸爸

写于 2018/05/22

在这个题目下，自然就会想到优先级队列了，但还需要额外考虑 vip 再分级，即同等级 vip 的平权的问题，所以应该考虑除了直接的和 vip 等级相关的优先级队列优先级规则问题，还得考虑同等级多个客户互相不被单一客户大量任务阻塞的问题，数据结构确实是基础，即便这个思考题考虑的这个场景，待调度数据估计会放在 redis 里面吧

| 引自：Java核心技术36讲

| 第8讲 | 对比Vector、ArrayList、LinkedList 有何区别？



识别二维码打开原文  
「极客时间」App

vash\_ace

写于 2018/06/01

其实在初始化 DirectByteBuffer 对象时，如果当前堆外内存的条件很苛刻时，会主动调用 System.gc() 强制执行 FGC。所以一般建议在使用 netty 时开启 XX:+DisableExplicitGC

引自：Java核心技术36讲

第12讲 | Java有几种文件拷贝方式？哪一种最高效？



识别二维码打开原文  
「极客时间」App

Walter

写于 2018/06/07

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问

示例，对于一个复杂的子系统，通过一个统一的接口，向客户端提供一个简化的方法。

系统的接口。它向现有的系统添加一个接口，来隐藏系统的复杂性。

这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。

何时使用：1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个“接待员”即可。2、定义系统的入口。

如何解决：客户端不与系统耦合，外观类与系统耦合。

关键代码：在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。

应用实例：1、去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来

处理，就很方便。 2、JAVA 的三层开发模式。

优点： 1、减少系统相互依赖。 2、提高灵活性。 3、提高了安全性。

缺点：不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。

使用场景： 1、为复杂的模块或子系统提供外界访问的模块。 2、子系统相对独立。 3、预防低水平人员带来的风险。

注意事项：在层次化结构中，可以使用外观模式定义系统中每一层的入口。

引自：Java核心技术36讲  
第14讲 | 谈谈你知道的设计模式？

识别二维码打开原文  
「极客时间」App





whhbq

2018-06-09

老师的招人标准，学习了，很实用，看了也很有感触。你能填上别人填不上的坑，就成功了。工作中大多数时候的任务目标并不清晰，特别当你是一个团队的小leader时，没人告诉你后面的方向，要做什么样，很考验能力。

vash\_ace

2018-06-11

感谢杨老师的鼓励，在受宠若惊之余，我觉得这篇“课外阅读”的参考价值不输于任何一篇技术分享。因为文中提到的这些努力或坚持的方向，确实是对个人职业生涯有着巨大的影响和帮助（深测有效）。  
其实道理大家都懂，但很多时候想当架构师做技术大牛的我们就是会以各种理由（项目忙，赶进度，不想加班…）不去对一个bug或一次线上故障做刨根问底的努力，又或者是放弃原本是对的坚持（比如，技术笔记，技术阅读与分享…）。  
那个时候的你所需要的鸡汤或兴奋剂不再是XXX的成功，而是想象一下自己的个人价值。往大了说，你对技术圈做了什么贡献？影响了多少人？影响越大成就感越满足。成就感是个好东西，你越享受就越会上瘾。往俗了说，就是看自己的收入在行业内处于什么样的水平？工资多少不一定能完全体现一个人的真实水平，但至少绝大部分公司和猎头都能根据你上一家公司的收入来定位你属于哪一個level。

公号-Java大前端

2018-06-09

阅读源码的时候。

首先，可通过各种公开的渠道（google、公开文档等）了解代码的总体框架、模块组成及其模块间的关系；  
然后，结合源码的注释进行解读，对不是很明白的部分打断点，调试，甚至可按照自己的想法进行修改后再调试；  
最后，对于重点核心模块进行详细调试，可以把核心类的功能、调用流程等写下来，好记性总是敌不过烂笔头。  
除此之外，个人觉得最重要的是：看源码的时候要有“静气”。

夏洛克的救赎

2018-06-09

看评论都能涨知识，希望评论提供交互功能

2018-06-09

雷霹雳的爸爸

2018-06-09

意外，感谢，更重要的是也复盘下这段学习过程中发现的自己的各种不足，再接再厉！

2018-06-09

iLeGeND

2018-06-09

我们应该面向接口编程，面向规范编程，在单纯的开发中，使dk或者框架，应该以其api文档为参考，如果有问题就看源码，那岂不是面子实现编程了，不同的版本，其实现不见得一样，我们的代码用不能一直改吧

Hidden

2018-06-13

我在阅读源码的时候，只能勉强理解一半，剩下那一半再怎么也理解不了，很是奇怪。

2018-06-12

Zoe.Li

2018-06-11

谢谢杨老师的分享

2018-06-11

Miaozhe

2018-06-11

感谢杨老师分享，这次学习收获很大，特别是认真阅读了HashMap的源码，桶的设计和Hash的位运算正的设计很妙。以前没有看懂，这次参考老师的“死磕”，终于看懂了。

2018-06-10

LenX

2018-06-10

正文中(非留言区)，倒数第二个推荐的读者留言中说：

2018-06-10

“所以一般建议在使用 Netty 时开启 XX: +DisableExplicitGC”。

注意，参数前使用的是 + 号，我觉得不对吧！  
这就表明 System.gc 变成空调用了，这对于 Netty，如果这么做会导致堆外内存不及时回收，反而更容易 OOM。

是这样吗？

2018-06-10

作者回复

2018-06-10

我认为看场景和侧重角度，如果发现cleaner自动回收不符合需求，用system.gc至少可以避免oom；如果应用没这问题，调用它也可能导致应用反应不稳定等问题。  
所以没有一劳永逸的办法或者最佳实践，只能是个思路参考，看实际需求

肖一林

谢谢老师的奖励，每一篇都在看，最近也在组织以前的笔记，放在自己的技术公众号。希望清理技术债务，达到系统学习的目的。结合以前所学，加上老师文章提到的一些底层原理，用自己的方式表达一遍

作者回复

加油，互相提高

zt

话说今天不更新了吗，大神能不能加快下更新的速度，学习完去面试，战线时间太长有点熬人

作者回复

有规定的节奏

iLeGeND

我们应该面向接口编程，面向规范编程，在单纯的开发中，使dk或者框架，应该以其api文档为参考，如果有问题就看源码，那岂不是面子实现编程了，不同的版本，其实现不见得一样，我们的代码用不能一直改吧

2018-06-09

2018-06-10

2018-06-09

2018-06-10

2018-06-09





























第16讲 | synchronized底层如何实现？什么是锁的升级、降级?  
2018-06-12 杨晓峰



第16讲 | synchronized底层如何实现？什么是锁的升级、降级？  
杨晓峰  
00:00 / 11:02

我在[上一讲](#)对比和分析了synchronized和ReentrantLock，算是专栏进入并发编程阶段的热身，相信你已经对线程安全，以及如何使用基本的同步机制有了基础，今天我们将深入了解synchronized底层机制，分析其他锁实现和应用场景。

今天我要问你的问题是，[synchronized底层如何实现？什么是锁的升级、降级？](#)

#### 典型回答

在回答这个问题前，先简单复习一下上一讲的知识点。synchronized代码块是由一对儿monitorenter/monitorexit指令实现的，Monitor对象是同步的基本实现单元。

在Java 6之前，Monitor的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作。

现代的（Oracle）JDK中，JVM对此进行了大刀阔斧地改进，提供了三种不同的Monitor实现，也就是常说的三种不同的锁：偏斜锁（Biased Locking）、轻量级锁和重量级锁，大大改进了其性能。

所谓锁的升级、降级，就是JVM优化synchronized运行的机制，当JVM检测到不同的竞争状况时，会自动切换到适合的锁实现，这种切换就是锁的升级、降级。

当没有竞争出现时，默认会使用偏斜锁。JVM会利用CAS操作（[compare and swap](#)），在对象头上的Mark Word部分设置线程ID，以表示这个对象偏向于当前线程，所以并不涉及真正的互斥锁。这样做的假设是基于在很多应用场景中，大部分对象生命周期中最多会被一个线程锁定，使用偏斜锁可以降低无竞争开销。

如果有另外的线程试图锁定某个已经被偏斜过的对象，JVM就需要撤销（revoke）偏斜锁，并切换到轻量级锁实现。轻量级锁依赖CAS操作Mark Word来试图获取锁，如果重试成功，就使用普通的轻量级锁；否则，进一步升级为重量级锁。

我注意到有的观点认为Java不会进行锁降级。实际上据我所知，锁降级确实是会发生的，当JVM进入安全点（[SafePoint](#)）的时候，会检查是否有闲置的Monitor，然后试图进行降级。

#### 考点分析

今天的题目主要是考察你对Java内置锁实现的掌握，也是并发的经典题目。我在前面给出的典型回答，涵盖了一些基本概念。如果基础不牢，有些概念理解起来就比较晦涩，我建议还是尽量理解和掌握，即使有不懂的也不用担心，在后续学习中还会逐步加深认识。

我个人认为，能够基础性地理解这些概念和机制，其实对于大多数并发编程已经足够了，毕竟大部分工程师未必会进行更底层、更基础的研发，很多时候解决的是知道与否，真正的提高还要靠实践踩坑。

后面我会进一步分析：

- 从源码层面，稍微展开一些synchronized的底层实现，并补充一些上面答案中欠缺的细节，有同学反馈这部分容易被问到。如果你对Java底层源码有兴趣，但还没有找到入手点，这里可以成为一个切入点。

- 理解并发包[java.util.concurrent.lock](#)提供的其他锁实现，毕竟Java可不是只有ReentrantLock一种显式的锁类型，我会结合代码分析其使用。

#### 知识扩展

我在[上一讲](#)提到过synchronized是JVM内部的Intrinsic Lock，所以偏斜锁、轻量级锁、重量级锁的代码实现，并不在核心类库部分，而是在JVM的代码中。

Java代码运行可能是解释模式也可能是编译模式（如果不记得，请复习[专栏第1讲](#)），所以对应的同步逻辑实现，也会分散在不同模块下，比如，解释器版本就是：

[src/hotspot/share/interpreter/InterpreterRuntime.cpp](#)

为了简化便于理解，我这里会专注于通用的基类实现：

[src/hotspot/share/runtime/](#)

另外请注意，链接指向的是最新JDK代码库，所以可能某些实现与历史版本有所不同。

首先，`synchronized`的行为是JVM runtime的一部分，所以我们需要先找到Runtime相关的功能实现。通过在代码中查询类似“monitor\_enter”或“Monitor Enter”，很直观的就可以定位到：

- [sharedRuntime.cpp](#)/hpp，它是解释器和编译器运行时的基类。
- [synchronizer.cpp](#)/hpp，JVM同步相关的各种基础逻辑。

在sharedRuntime.cpp中，下面代码体现了`synchronized`的主要逻辑。

```
Handle h_obj(THREAD, obj);
if (UseBiasedLocking) {
    // Retry fast entry if bias is revoked to avoid unnecessary inflation
    ObjectSynchronizer::fast_enter(h_obj, lock, true, CHECK);
} else {
    ObjectSynchronizer::slow_enter(h_obj, lock, CHECK);
}
```

其实现可以简单进行分解：

- `UseBiasedLocking`是一个检查，因为在JVM启动时，我们可以指定是否开启偏斜锁。

偏斜锁并不适合所有应用场景，撤销操作（revoke）是比较重的行为，只有当存在较多不会真正竞争的`synchronized`块儿时，才能体现出明显改善。实践中对于偏斜锁的一直是有争议的，有人甚至认为，当你需要大量使用并发类库时，往往意味着你不需要偏斜锁。从具体选择来看，我还是建议需要在实践中进行测试，根据结果再决定是否使用。

还有一方面是，偏斜锁会延缓JIT预热的进程，所以很多性能测试中会显式地关闭偏斜锁，命令如下：

```
-XX:-UseBiasedLocking
```

- `fast_enter`是我们熟悉的完整锁获取路径，`slow_enter`则是绕过偏斜锁，直接进入轻量级锁获取逻辑。

那么`fast_enter`是如何实现的呢？同样是通过在代码库搜索，我们可以定位到[synchronizer.cpp](#)。类似`fast_enter`这种实现，解释器或者动态编译器，都是拷贝这段基础逻辑，所以如果我们修改这部分逻辑，要保证一致性。这部分代码是非常敏感的，微小的问题都可能导致死锁或者正确性问题。

```
void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock,
                                    bool attempt_rebias, TRAPS) {
    if (UseBiasedLocking) {
        if (!SafepointSynchronize::is_at_safepoint()) {
            BiasedLocking::Condition cond = BiasedLocking::revoke_and_rebias(obj, attempt_rebias, THREAD);
            if (cond == BiasedLocking::BIAS_REVOKED_AND_REBIASED) {
                return;
            }
        } else {
            assert(!attempt_rebias, "can not rebias toward VM thread");
            BiasedLocking::revoke_at_safepoint(obj);
        }
        assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
    }

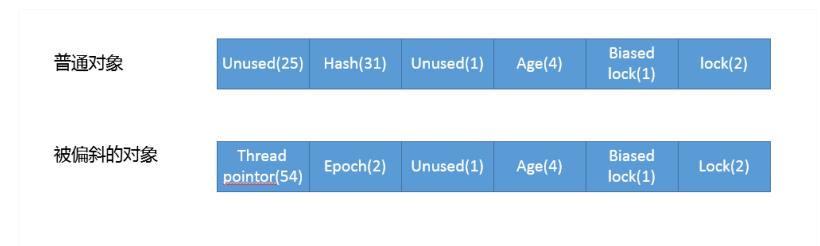
    slow_enter(obj, lock, THREAD);
}
```

我来分析下这段逻辑实现：

- [biasedLocking](#)定义了偏斜锁相关操作，`revoke_and_rebias`是获取偏斜锁的入口方法，`revoke_at_safepoint`则定义了当检测到安全点时的处理逻辑。
- 如果获取偏斜锁失败，则进入`slow_enter`。
- 这个方法里面同样检查是否开启了偏斜锁，但是从代码路径来看，其实如果关闭了偏斜锁，是不会进入这个方法的，所以算是个额外的保障性检查吧。

另外，如果你仔细查看[synchronizer.cpp](#)里，会发现不仅仅是`synchronized`的逻辑，包括从本地代码，也就是JNI，触发的Monitor动作，全都可以在里面找到(`jni_enter/jni_exit`)。

关于[biasedLocking](#)的更多细节我就不展开了，明白它是通过CAS设置Mark Word就完全够用了，对象头中Mark Word的结构，可以参考下图：



顺着锁升级的过程分析下去，偏斜锁到轻量级锁的过程是如何实现的呢？

我们来看看slow\_enter到底做了什么。

```
void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
    markOp mark = obj->mark();
    if (mark->is_neutral()) {
        // 将目前的Mark Word复制到Displaced Header上
        lock->set_displaced_header(mark);
        // 利用CAS设置对象的Mark Word
        if (mark == obj()->cas_set_mark((markOp) lock, mark)) {
            TEVENT(slow_enter: release_dlock);
            return;
        }
        // 检查存在竞争
    } else if (mark->has_locker() &&
               THREAD->is_lock_owned((address)mark->locker())) {
        // 清除
        lock->set_displaced_header(NULL);
        return;
    }

    // 重置Displaced Header
    lock->set_displaced_header(markOpDesc::unused_mark());
    ObjectSynchronizer::inflate(THREAD,
        obj(),
        inflate_cause_monitor_enter)->enter(THREAD);
}
```

请结合我在代码中添加的注释，来理解如何从试图获取轻量级锁，逐步进入锁膨胀的过程。你可以发现这个处理逻辑，和我在这一讲最初介绍的过程是十分吻合的。

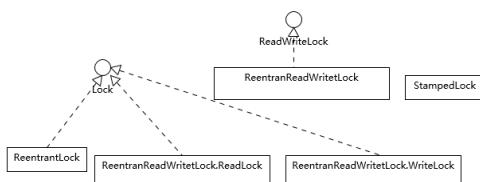
- 设置Displaced Header，然后利用cas\_set\_mark设置对象Mark Word，如果成功就成功获取轻量级锁。

- 否则Displaced Header，然后进入锁膨胀阶段，具体实现在inflate方法中。

今天就不介绍膨胀的细节了，我这里提供了源代码分析的思路和样例，考虑到应用实践，再进一步增加源代码解读意义不大，有兴趣的同学可以参考我提供的[synchronizer.cpp](#)链接，例如：

- `deflate_Idle_monitors`是分析锁降级逻辑的入口，这部分行为还在进行持续改进，因为其逻辑是在安全点内运行，处理不当可能拖长JVM停顿（STW, stop-the-world）的时间。
- `fast_exit`或者`slow_exit`是对应的锁释放逻辑。

前面分析了`synchronized`的底层实现，理解起来有一定难度，下面我们来看一些相对轻松的内容。我在上一讲对比了`synchronized`和`ReentrantLock`，Java核心类库中还有其他一些特别的锁类型，具体请参考下面的图。



你可能注意到了，这些锁竟然不都是实现了`Lock`接口。`ReadWriteLock`是一个单独的接口，它通常是代表了一对儿锁，分别对应只读和写操作，标准类库中提供了再入版本的读写锁实现（`ReentrantReadWriteLock`），对应的语义和`ReentrantLock`比较相似。

`StampedLock`竟然也是个单独的类型，从类图结构可以看出它是不支持再入性的语义的，也就是它不是以持有锁的线程为单位。

为什么我们需要读写锁（`ReadWriteLock`）等其他锁呢？

这是因为，虽然`ReentrantLock`和`synchronized`简单实用，但是行为上有一定局限性，通俗点说就是“太霸道”，要么不占，要么独占。实际应用场景中，有的时候不需要大量竞争的写操作，而是以并发读取为主，如何进一步优化并发操作的粒度呢？

Java并发包提供的读写锁等扩展了锁的能力，它所基于的原理是多个读操作是不需要互斥的，因为读操作并不会更改数据，所以不存在互相干扰。而写操作则会导致并发一致性的問題，所以写线程之间、读写线程之间，需要精心设计的互斥逻辑。

下面是一个基于读写锁实现的数据结构，当数据量较大，并发读多、并发写少的时候，能够比纯同步版本凸显优势。

```
public class RWsample {
    private final Map<String, String> m = new TreeMap<>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();
```

```

public String get(String key) {
    r.lock();
    System.out.println("读锁锁定!");
    try {
        return m.get(key);
    } finally {
        r.unlock();
    }
}

public String put(String key, String entry) {
    w.lock();
    System.out.println("写锁锁定!");
    try {
        return m.put(key, entry);
    } finally {
        w.unlock();
    }
}
// ...
}

```

在运行过程中，如果读锁试图锁定时，写锁是被某个线程持有，读锁将无法获得，而只好等待对方操作结束，这样就可以自动保证不会读取到有争议的数据。

读写锁看起来比synchronized的粒度似乎细一些，但在实际应用中，其表现也并不尽如人意，主要还是因为相对比较大的开销。

所以，JDK在后期引入了StampedLock，在提供类似读写锁的同时，还支持优化读模式。优化读基于假设，大多数情况下读操作并不会和写操作冲突，其逻辑是先试着修改，然后通过validate方法确认是否进入了写模式，如果没有进入，就成功避免了开销；如果进入，则尝试获取读锁。请参考我下面的样例代码。

```

public class StampedSample {
    private final StampedLock sl = new StampedLock();

    void mutate() {
        long stamp = sl.writeLock();
        try {
            write();
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    Data access() {
        long stamp = sl.tryOptimisticRead();
        Data data = read();
        if (!sl.validate(stamp)) {
            stamp = sl.readLock();
            try {
                data = read();
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return data;
    }
}

```

注意，这里的writeLock和unlockWrite一定要保证成对调用。

你可能很好奇这些显式锁的实现机制，Java并发包内的各种同步工具，不仅仅是各种Lock，其他的如Semaphore、CountDownLatch，甚至是早期的FutureTask等，都是基于一种AQS框架。

今天，我全面分析了synchronized相关实现和内部运行机制，简单介绍了并发包中提供的其他显式锁，并结合样例代码介绍了其使用方法，希望对你有所帮助。

#### 一课一练

关于今天我们讨论的你做到心中有数了吗？思考一个问题，你知道“自旋锁”是做什么的吗？它的使用场景是什么？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



公号-Java大后端

自旋锁: 竞争锁的失败的线程, 并不会真的在操作系统层面挂起等待。而是JVM会让线程做几个空循环(基于预测在不久的将来就能获得), 在经过若干次循环后, 如果可以获得锁, 那么进入临界区; 如果还不能获得锁, 才会真实的将线程在操作系统层面进行挂起。

适用场景: 自旋锁可以减少线程的阻塞, 这对于锁竞争不激烈, 且占用时间非常短的代码块来说, 有较大的性能提升, 因为自旋的消耗会小于线程阻塞挂起操作的消耗。  
如果锁的竞争激烈, 或者持有锁的线程需要长时间占用锁执行同步块, 就不适合使用自旋锁了, 因为自旋锁在获取锁前一直都是占用cpu做无用功, 线程自旋的消耗大于线程阻塞挂起操作的消耗, 造成cpu的浪费。

作者回复

不错, 自旋是种乐观情况的优化

2018-06-12

yearning

2018-06-12

这次原理真的看了很久, 一直鼓劲自己, 看不懂就是说明自己有突破。

下面看了并发编程对于自旋锁的了解, 同时更深刻理解同步锁的性能。

自旋锁采用让当前线程不停循环体去执行实现, 当循环条件被其他线程改变时, 才能进入临界区。

由于自旋锁只是将当前线程不停执行循环体, 不进行线程状态的改变, 所以响应会更快。但当线程不停增加时, 性能下降明显。  
线程竞争不激烈, 并且保持锁的时段段。适合使用自旋锁。

为什么会提出自旋锁, 因为互斥锁, 在线程的睡眠和唤醒都是复杂且昂贵的操作, 需要大量的CPU指令。如果互斥仅仅被锁住是一小段时间, 用来进行线程休眠和唤醒的操作时间比睡眠时间还长, 更有可能比不上不断自旋锁上轮询的时间长。

当然自旋锁被持有的时间更长, 其他尝试获取自旋锁的线程会一直轮询自旋锁的状态。这将十分浪费CPU。

在单核CPU上, 自旋锁是无用, 因为当自旋锁尝试获取锁不成功会一直尝试, 这会一直占用CPU, 其他线程不可能运行,  
同时由于其他线程无法运行, 所以当前线程无法释放锁。

混合型互斥锁, 在多核系统上起初表现的像自旋锁一样, 如果一个线程不能获取互斥锁, 它不会马上被切换为休眠状态, 在一段时间依然无法获取锁, 进行睡眠状态。

混合型自旋锁, 起初表现的和正常自旋锁一样, 如果无法获取互斥锁, 它也许会放弃该线程的执行, 并允许其他线程执行。

切记, 自旋锁只有在多核CPU上有效果, 单核毫无效果, 只是浪费时间。

以上基本参考来源于:  
[http://ifeve.com/java\\_lock\\_see1/](http://ifeve.com/java_lock_see1/)  
<http://ifeve.com/practice-of-using-spinlock-instead-of-mutex/>

作者回复

很不错总结

2018-06-12

黑子

2018-06-12

自旋锁 for(:)结合cas确保线程获取锁

作者回复

差不多

2018-06-12

sunlight001

2018-06-12

自旋锁是尝试获取锁的线程不会立即阻塞, 采用循环的方式去获取锁, 好处是减少了上下文切换, 缺点是消耗cpu

作者回复

不错

2018-06-12

灰飞灰猪不会灰飞, 烟灭

2018-06-12

老师 AQS就不涉及用户态和内核态的切换了 对吧?

作者回复

我理解是, cas是基于特定指令

2018-06-12

jacy

看了大家对自旋锁的评论，我的收获如下：  
 1. 基于乐观情况下推荐使用 即锁竞争不强，锁等待时间不长的情况下推荐使用  
 2. 单cpu无效。因为基于cas的锁询问会占用cpu，导致无法做线程切换  
 3. 锁的竞争不产生上下文切换，如果可估计到睡眠的时间很长，用互斥锁更好

作者回复

不错

Miaozhe

杨老师，看到有回复说自旋锁在单核CPU上是无用，感觉这个理论不准确，因为Java多线程在很早时候单核CPU的PC上就能运行，计算机原理中也介绍，控制器会轮巡各个进程或线程。而且多线程是运行在JVM上，跟物理机没有很直接的关系吧？

作者回复

已回复，我也认为单核无用

齐帆

老师后面会详细讲 AQS 吗

作者回复

有的

肖一林

重量级锁还是互斥锁吗？自旋锁应该是线程拿不到锁的时候，采取重试的办法，适合重试次数不多的场景，如果重试次数过多还是会被系统挂起，这种情况下还不如没有自旋锁。

作者回复

是的

宾语

挺不错

刘杰

偏斜锁和轻量级锁的区别不是很清晰

有福

自旋锁基本形式就是通过while循环加上cpu指令级别保证的cas原子操作来判断某一个共享变量内存的值是否被其他线程修改，如果没有修改那么就认为获取到了锁。  
 这个变量需要设置为volatile，否则容易被指令重排引发bug。

之前实际应用的场景就是开发多核处理器下的core to core的高性能无锁队列。

由于是一直while循环，所以cpu在检查锁状态的时候基本上是100%，所以自旋锁基本上是用来判断某个状态是否发生，也就是用来同步的，而不是用来互斥的。

前前个坏的

可是我还是不清楚偏斜锁和轻量级锁的区别

zeusoul

老师，你好，我想问下，Java 的CAS和volatile对于服务器集群是否支持。

Cui

基于AQS的锁是属于哪种级别的锁？

Cui

老师你好 心中一直有个疑问：synchronize和AQS的LockSupport同样起到阻塞线程的作用，这两者的区别是什么？能不能从实现原理和使用效果的角度说说？

作者回复

LockSupport park是waiting，另一个是blocked；具体底层，马上一篇有说明

Miaozhe

杨老师，StampedSample这个例子，access方法是不是写错了？

```
long stamp = sl.tryOptimisticRead();
Data data = read();
```

应该是在先判断tryOptimisticRead的结果，如果获取了锁，才进入read()吧？因为没有获取锁的读，可能是脏读。  
 自己代码调试，发现即使tryOptimistic的结果为0，也会向下执行read()。

作者回复

我记得validate会返回false，如果输入stamp是0，所以程序并没有漏洞

Miaozhe

关于自旋锁不适合单核CPU的问题，下来查找了一下资料：  
 1. JVM在操作系统中是作为一个进程存在，但是OS一般都将线程作为最小调度单位，进程是资源分配的最小单位。这就是说进程是不活动的，只是作为线程的容器，那么Java的线程是在JVM进程中，也被CPU调度。  
 2. 单核CPU使用多线程时，一个线程被CPU执行，其它处于等待轮巡状态。  
 3. 为什么多线程跑在单核CPU上也很快呢？是由于这种线程还有其它IO操作(File, Socket)，可以跟CPU运算并行。  
 4. 结论，根据前面3点的分析，与自旋锁的优点冲突。线程竞争不激烈，占用锁时间短。

作者回复

自旋是基于乐观假设，就是等待中锁被释放了，单核cpu就自己占着cpu，别人没机会让

I.am DZX

2018-06-19

2018-06-19

2018-06-13

2018-06-13

2018-06-12

2018-06-12

2018-06-12

2018-07-15

2018-07-12

2018-07-06

2018-07-03

2018-06-23

2018-06-22

2018-06-22

2018-06-26

2018-06-15

2018-06-17

2018-06-13

2018-06-13

请问自旋锁和非公平获取锁是不是有点冲突了

作者回复

我理解非公平是不保证，另外自旋抢到的线程不见得就是等的久的

TWO STRINGS

StampedLock那里乐观读锁好像是说写操作不需要等待读操作完成，而不是“读操作并不需要等待写完成”吧

作者回复

非常感谢，这话写的是有问题

食指可多

以前写过自旋锁的实现，当某个线程调用自旋锁实例的lock方法时，使用cas进行设置，`cas(lockThread, null, currentThread)`，也就是当前无锁定时当前线程会成功，失败则循环尝试直到成功。利用cas保证操作的原子性，成员变量`lockThread`设置为`volatile`保证并发线程间可见性。所以从机制上可以看到，若是在高并发场景，成功拿到锁之外的所有线程会继续努力尝试持有锁，造成CPU资源的浪费。如评论中其它同学所说适合在低并发场景使用。

作者回复

是的

Geek\_e61ae8

老师讲到读写锁，这里涉及到读并发高，当我更改要加载的数据，这时需要写，读到内存后准备切换，但是一直获取不了写锁。这种采用自己boolean值来控制，让读sleep等待，或者直接返回不进锁（已经获取读锁的线程等处理结束）。写获取锁后更新，替换boolean值。另一种采用公平锁。老师觉得建议那种？

作者回复

我建议用StampedLock或读写锁

Geek\_e61ae8

这块老师讲了读写锁，如果读并发高，当配置更改，触发了写，但是又获取不了锁，这种情况可以采用boolean值自己控制当写完，替换内存时，让读的线程等待。（已经获取锁的等处理完）没处理的等待。这种是建议加上公平锁好，还是说自己控制好

作者回复

没看懂，是说让读线程不停的检查boolean值等待吗？自己控制要达到的目的是什么呢

凡旗

杨老师，操作系统的互斥锁要怎么理解

作者回复

这个还是请看操作系统相关代码或资料，原理上mutex和只有0、1值的semaphore是近似的，但现代操作系统怎么实现我真没研究过，谁有空儿补充下？

Miaozhe

杨老师，偏斜锁有什么作用？还是没有看明白，如果只是被一个线程获取，那么锁还有什么意义？

另外，如果我有两个线程明确定义调用同一个对象的Synchronized块，JVM默认肯定先使用偏斜锁，之后在升级到轻量级锁，必须经过撤销Revoke吗？编译的时候不会自动优化？

作者回复

我理解偏斜锁就是为了优化那些没有并发却写了同步逻辑的代码：javac编译时能判断的是有限的；一旦有另外线程想获取，就会revoke，而且撤销明显

张玮(大圣)

自旋锁类似和忙等待一个套路

作者回复

嗯，我理解是一个意思

tysen

简单来说就是while，一直cas直到成功吧。

作者回复

差不多，也要考虑退出自旋的情况

雷雷雷的爸爸

今天老师讲这个真够我喝一壶的。而且老师总结的角度启发性很大，最近也再读JCIP，对比起来很有意思，对于自旋锁这个理解，我一直还是肤浅的，顾名思义比较多，就是在那里面兜几个圈子——写个循环——试几次，好处是减少线程切换导致的开销，一般也需要有底层有CAS能力的构件支持一下，比如用Atomic开头那些类，当然也未必，比如说nio读不出来东西的时候，也先尝试几次，总之就是暂时不把cpu让度出去，先在占着坑来几次，大概可能这么个意思吧

作者回复

差不多，算是种乐观主义的“优化”

浩

自旋就是空转，什么都不干，就在循环等待锁，相当于缓冲一段时间，看能否获得锁，如果此次自旋获得锁，那么下次，会比此次更长时间自旋，增大获得锁的概率，否则，减少自旋次数。

作者回复

基本正确





## 第17讲 | 一个线程两次调用start()方法会出现什么情况?

2018-06-14 杨晓峰



第17讲 | 一个线程两次调用start()方法会出现什么情况?  
杨晓峰  
- 00:18 / 10:01

今天我们来深入聊聊线程，相信大家对于线程这个概念都不陌生，它是Java并发的基础元素，理解、操纵、诊断线程是Java工程师的必修课，但是你真的掌握线程了吗？

今天我要问你的问题是，[一个线程两次调用start\(\)方法会出现什么情况？谈谈线程的生命周期和状态转移。](#)

## 典型回答

Java的线程是不允许启动两次的，第二次调用必然会抛出IllegalThreadStateException，这是一种运行时异常，多次调用start被认为是编程错误。

关于线程生命周期的不同状态，在Java 5以后，线程状态被明确定义在其公共内部枚举类型java.lang.Thread.State中，分别是：

- 新建（NEW），表示线程被创建出来还没真正启动的状态，可以认为它是个Java内部状态。
- 就绪（RUNNABLE），表示该线程已经在JVM中执行，当然由于执行需要计算资源，它可能是正在运行，也可能还在等待系统分配给它CPU片段，在就绪队列里面排队。
- 在其他一些分析中，会额外区分一种状态RUNNING，但是从Java API的角度，并不能表示出来。
- 阻塞（BLOCKED），这个状态和我们前面两讲介绍的同步非常相关，阻塞表示线程在等待Monitor lock。比如，线程试图通过synchronized去获取某个锁，但是其他线程已经独占了，那么当前线程就会处于阻塞状态。
- 等待（WAITING），表示正在等待其他线程采取某些操作。一个常见的场景是类似生产者消费者模式，发现任务条件尚未满足，就让当前消费者线程等待（wait），另外的生产者线程去准备任务数据，然后通过类似notify等动作，通知消费线程可以继续工作了。ThreadJoin()也会令线程进入等待状态。
- 计时等待（TIMED\_WAIT），其进入条件和等待状态类似，但是调用的是存在超时条件的方法，比如wait或join等方法的指定超时版本，如下面示例：

```
public final native void wait(long timeout) throws InterruptedException;
```

- 终止（TERMINATED），不管是意外退出还是正常执行结束，线程已经完成使命，终止运行，也有人把这个状态叫作死亡。

在第二次调用start()方法的时候，线程可能处于终止或者其他（非NEW）状态，但是不论如何，都是不可以再次启动的。

## 考点分析

今天的问题可以算是个常见的面试热身题目，前面的给出的典型回答，算是对基本状态和简单流转的一个介绍，如果觉得还不够直观，我在下面分析会对比一个状态图进行介绍。总的来说，理解线程对于我们日常开发或者诊断分析，都是不可或缺的基础。

面试官可能会以此为契机，从各种不同角度考察你对线程的掌握：

- 相对理论一些的面试官可能会问你线程到底是什么以及Java底层实现方式。
- 线程状态的切换，以及和锁等并发工具类的互动。
- 线程编程时容易踩的坑与建议等。

可以看出，仅仅是一个线程，就有非常多的内容需要掌握。我们选择重点内容，开始进入详细分析。

## 知识扩展

首先，我们来整体看一下线程是什么？

从操作系统的角度，可以简单认为，线程是系统调度的最小单元，一个进程可以包含多个线程，作为任务的真正运作者，有自己的栈（Stack）、寄存器（Register）、本地存储（Thread Local）等，但是会和进程中其他线程共享文件描述符、虚拟地址空间等。

在具体实现中，线程还分为内核线程、用户线程，Java的线程实现其实是与虚拟机相关的。对于我们最熟悉的Sun/Oracle JDK，其线程也经历了一个演进过程，基本上在Java 1.2之后，JDK已经抛弃了所谓的Green Thread，也就是用户调度的线程，现在的模型是一对一映射到操作系统内核线程。

如果我们来看Thread的源码，你会发现其基本操作逻辑大都是以JNI形式调用的本地代码。

```
private native void start0();
private native void setPriority0(int newPriority);
private native void interrupt0();
```

这种实现有利有弊，总体上来说，Java语言得益于精细节度的线程和相关的并发操作，其构建高扩展性的大型应用的能力已经毋庸置疑。但是，其复杂性也提高了并发编程的门槛，近几年的Go语言等提供了协程（coroutine），大大提高了构建并发应用的效率。于此同时，Java也在Loom项目中，孕育新的类似轻量级用户线程（Fiber）等机制，也许在不久的将来就可以在新版JDK中使用到它。

下面，我来分析下线程的基本操作。如何创建线程想必你已经非常熟悉了，请看下面的例子：

```
Runnable task = () -> {System.out.println("Hello World!");};
Thread myThread = new Thread(task);
myThread.start();
myThread.join();
```

我们可以直接扩展Thread类，然后实例化。但在本例中，我选取了另外一种方式，就是实现一个Runnable，将代码逻辑放在Runnable中，然后构建Thread并启动（start），等待结束（join）。

Runnable的好处是，不会受Java不支持类多继承的限制，重用代码实现，当我们需要重复执行相应逻辑时优点明显。而且，也能更好的与现代Java并发库中的Executor之类框架结合使用，比如将上面start和join的逻辑完全写成下面的结构：

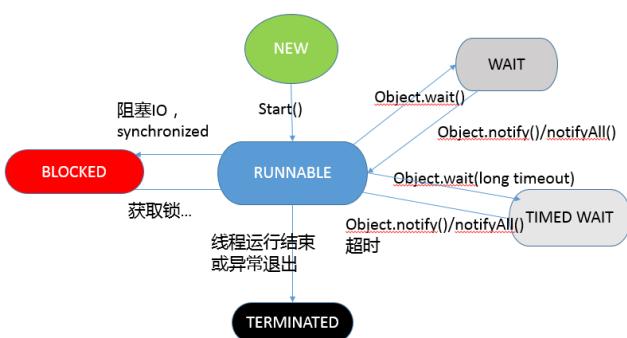
```
Future future = Executors.newFixedThreadPool(1)
.submit(task)
.get();
```

这样我们就不必操心线程的创建和管理，也能利用Future等机制更好地处理执行结果。线程生命周期通常和业务之间没有本质联系，混淆实现需求和业务需求，就会降低开发的效率。

从线程生命周期的状态开始展开，那么在Java编程中，有哪些因素可能影响线程的状态呢？主要有：

- 线程自身的方法，除了start，还有多个join方法，等待线程结束；yield是告诉调度器，主动让出CPU；另外，就是一些已经被标记为过时的resume、stop、suspend之类，据我所知，在JDK最新版本中，destroy/stop方法将被直接移除。
- 基类Object提供了一些基础的wait/notify/notifyAll方法。如果我们持有某个对象的Monitor锁，调用wait会让当前线程处于等待状态，直到其他线程notify或者notifyAll。所以，本质上是提供了Monitor的获取和释放的能力，是基本的线程间通信方式。
- 并发类库中的工具，比如CountDownLatch.await()会让当前线程进入等待状态，直到latch被基数为0，这可以看作是线程间通信的Signal。

我这里画了一个状态和方法之间的对应图：



Thread和Object的方法，听起来简单，但是实际应用中被证明非常晦涩、易错，这也是为什么Java后来又引入了并发包。总的来说，有了并发包，大多数情况下，我们已经不再需要去调用wait/notify之类的方法了。

前面谈了不少理论，下面谈谈线程API使用，我会侧重于平时工作学习中，容易被忽略的一些方面。

先来看看守护线程（Daemon Thread），有的时候应用中需要一个长期驻留的服务程序，但是不希望其影响应用退出，就可以将其设置为守护线程。如果JVM发现只有守护线程存在时，将结束进程，具体可以参考下面代码段。注意，必须在线程启动之前设置。

```
Thread daemonThread = new Thread();
daemonThread.setDaemon(true);
daemonThread.start();
```

再来看看[Spurious wakeup](#)。尤其是在多核CPU的系统中，线程等待存在一种可能，就是在没有任何线程广播或者发出信号的情况下，线程就被唤醒，如果处理不当就可能出现诡异的并发问题，所以我们在等待条件过程中，建议采用下面模式来书写。

```
// 推荐
while (isCondition()) {
    waitForACondition(...);
}

// 不推荐，可能引入bug
if (isCondition()) {
    waitForACondition(...);
}
```

Thread.onSpinWait(), 这是Java 9中引入的特性。我在[专栏第16讲](#)给你留的思考题中，提到“自旋锁”(spin-wait, busy-waiting)，也可以认为其不算是一种锁，而是一种针对短期等待的性能优化技术。“onSpinWait()”没有任何行为上的保证，而是对JVM的一个暗示，JVM可能会利用CPU的pause指令进一步提高性能，性能特别敏感的应用可以关注。

再有就是借用[ThreadLocal](#)，这是Java提供的一种保存线程私有信息的机制，因为其在整个线程生命周期内有效，所以可以方便地在一个线程关联的不同业务模块之间传递信息，比如事务ID、Cookie等上下文相关信息。

它的实现结构，可以参考[源码](#)，数据存储于线程相关的ThreadLocalMap，其内部条目是弱引用，如下面片段。

```
static class ThreadLocalMap {
    static class Entry extends WeakReference<ThreadLocal> {
        /** The value associated with this ThreadLocal. */
        Object value;
        Entry(ThreadLocal k, Object v) {
            super(k);
            value = v;
        }
    }
    ...
}
```

当Key为null时，该条目就变成“废弃条目”，相关“value”的回收，往往依赖于几个关键点，即set、remove、rehash。

下面是set的示例，我进行了精简和注释：

```
private void set(ThreadLocal<> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);

    for (Entry e = tab[i]; ...) {
        //-
        if (k == null) {
            // 替换废弃条目
            replaceStaleEntry(key, value, i);
            return;
        }
    }

    tab[i] = new Entry(key, value);
    int sz = ++size;
    // 扫描并清理发现的废弃条目，并检查容量是否超限
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash(); // 清理废弃条目，如果仍然超限，则扩容（加倍）
}
```

具体的清理逻辑是实现在cleanSomeSlots和expungeStaleEntry之中，如果你有兴趣可以自行阅读。

结合[专栏第4讲](#)介绍的引用类型，我们会发现一个特别的地方，通常弱引用都会和引用队列配合清理机制使用，但是ThreadLocal是个例外，它并没有这么做。

这意味着，废弃项目的回收依赖于显式地触发，否则就要等待线程结束，进而回收相应ThreadLocalMap！这就是很多OOM的来源，所以通常都会建议，应用一定要自己负责remove，并且不要和线程池配合，因为worker线程往往是不会退出的。

今天，我介绍了线程基础，分析了生命周期中的状态和各种方法之间的对应关系，这也有助于我们更好地理解synchronized和锁的影响，并介绍了一些需要注意的操作，希望对你有所帮助。

### -一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天我准备了一个有意思的问题，写一个最简单的打印HelloWorld的程序，说说看，运行这个应用，Java至少会创建几个线程呢？然后思考一下，如何明确验证你的结论，真实情况很可能令你大跌眼镜哦。

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



风动静泉

一课一练：  
使用了两种方式获取当前程序的线程数。  
1. 使用线程管理器MXBean  
2. 直接通过线程组的activeCount  
第二种需要注意不断向上找父线程组，否则只能获取当前线程组，结果是1

结论：  
使用以上两种方式获取的线程总数都是5个。

main  
Attach Listener  
Signal Dispatcher  
Finalizer  
Reference Handler

此外，如果使用的IDE是IDEA直接运行会多一个Monitor Ctrl-break线程，这个是IDE的原因。debug模式下不会有这个线程。

作者回复

不错

qpm

做了一个test分析老师的问题，观察到的情况如下：  
JVM 启动 Hello World的线程分析  
环境：

macOS + jdk8  
检测获得  
Thread[Reference Handler,10,system]  
Thread[Finalizer,8,system]  
Thread[main,5,main]  
Thread[Signal Dispatcher,9,system]  
Hello World!

其他：  
Reference Handler：处理引用对象本身的垃圾回收  
Finalizer：处理用户的Finalizer方法  
Signal Dispatcher：外部vm命令的转发器

在jdk内部中  
还有一个Attach Listener的线程  
是负责接收外部命令的，如jmap、jstack

作者回复

不错

行者

“我们会发现一个特别的地方，通常幻象引用都会和引用队列配合清理机制使用，但是 ThreadLocal 是个例外，它并没有这么做。”  
老师，Entry继承的是WeakReference，这个是弱引用吧。

```
main:  
System.out.println("hello world");  
ThreadGroup group = Thread.currentThread().getThreadGroup();  
ThreadGroup topGroup = group;  
while (group != null) {  
    topGroup = group;  
    group = group.getParent();  
}  
int nowThreads = topGroup.activeCount();  
Thread[] lstThreads = new Thread[nowThreads];  
topGroup.enumerate(lstThreads);  
for (int i = 0; i < nowThreads; i++) {  
    System.out.println("线程number:" + i + " = " + lstThreads[i].getName());  
}  
out:  
线程number: 0 = Reference Handler // 计算对象是否可达?  
线程number: 1 = Finalizer // 回收对象时触发的finalize方法?  
线程number: 2 = Signal Dispatcher // 线程调度员  
线程number: 3 = main  
线程number: 4 = Monitor Ctrl-Break // 监控器，锁相关
```

作者回复

前面是翻译串了，已经修正；后面大家用了很多方法，基本都可以，主要目的是结合前面的介绍加深理解

爱折腾的老挝鸡

**theadlocal**里面的值如果是线程池的线程里面设置的，当任务完成，线程归还线程池时，这个**threadlocal**里面的值是不是不会被回收？

作者回复

嗯，线程池一般不建议和**thread local**配合…

三木子

现在觉得踩坑是一种很好学习方法

作者回复

同意

tyson

1. 站在应用程序方面，只创建了一个线程。
2. 站在jvm方面，肯定还有gc等其余线程。

总结：

1. 线程是系统调度的最小单元，应该是进程吧。线程是操作系统的资源，在运行的时候会打开文件描述符等。
2. `resume`, `stop`, `suspend`等已经被废弃了
3. 线程的等待和唤醒，建议使用`reentrantlock`的condition `wait/notify`方法
4. 可以使用线程的`join`方法、`countdownlatch`, `cyclicbarrier`、`future`等进行线程的等待

作者回复

不错

锐

通常弱引用都会和引用队列配合清理机制使用，但是 `ThreadLocal` 是个例外，它并没有这么做。

这意味着，废弃项目的回收依赖于显式地触发，否则就要等待线程结束，进而回收相应 `ThreadLocalMap`！这就是很多 OOM 的来源

这个平时还真没注意

作者回复

嗯，为了生命周期的需求

sunlight001

**threadlocal**在放入值之后，在get出来之后，需要做remove操作，我这么理解对么？以前写的程序都没remove◆◆

作者回复

不用了，明确移除是好习惯

Eason

“比如，线程试图通过 `synchronized` 去获取某个锁，但是其他线程已经独占了，那么当前线程就会处于阻塞状态”这个例子换一个理解，感觉也是在等待其他线程做某些操作。在“阻塞”中也是在“等待”中？？

作者回复

`wait`和`blocked`是不同的

黄启航

杨老师您好，我有个疑问：

文章最后说“弱引用都会和引用队列配合清理工作，但是`Threadlocal`是个例外，它并没有这么做。这意味着，废弃项目的回收依赖显示地触发，否则就要等待线程的结束”。

我的疑问：既然没有利用引用队列来实现自动清除，那`ThreadLocalMap`内部的Entry继承`WeakReference`有何用意？能起到什么作用？

tracer

有讲解那五个线程的资料吗？

作者回复

不知道，源码…

jacy

`Threadlocal`进行线程隔离，线程拥有自己的数据空间，`synchronize`进行线程同步。  
另外想问老师，虚假唤醒的深层次原因是啥呢？

TonyEasy

老师，我有一点疑问，在线程池复用线程时，对同一线程调用多次`start()`方法，为何不报错呢？

作者回复

工作线程一般不退出的

TonyEasy

老师，我有一点疑问，在线程池重用线程时是不是对同一个线程调用了多次`start()`方法呢？

作者回复

不是的，工作线程一般不退出的，复用的是类似`Runnable`这种

扫地僧的功夫梦

调用`notify()`/`notifyAll()`方法线程是变为阻塞状态吧，因为线程还没获取到锁。

作者回复

已回复，不是的

三木子

2018-06-17

看了17讲回来留言`threadlocal`

mongo

2018-06-19

杨老师请教你，关于高并发和线程池，我刚刚入门，工作中没有涉及过这一块。我阅读了`oracle java tutorial high level concurrency`章节，阅读并粗略理解了《并发编程实践》这本书，想进一步清晰我的理解，我现在苦于在实践练习方面不知道怎么进行。老师有什么具体可行的思路指点一下吗？留言圈里有好多大神，在这里同时也请教其他的朋友。谢谢老师，谢谢大家。

作者回复

下面章节就会覆盖这部分，我谈下自己的思路：大部分工程师是没有机会在工作中，全面使用并发的那些东西的，尤其是反馈读者中初学者不少；所以，我建议有个整体性体系有个了解，分清大体都有什么，然后可以选些实践场景，去实现用例代码。面试中大体也就够了，毕竟项目经验不是教程能解决的

肖一林

2018-06-15

`threadlocal`和线程池结合的问题真的没考虑过

作者回复

线程池里的线程生命周期长

Miaozhe

2018-06-17

问个问题，`NIO 2`的异步是不是利用协程的原理设计的？它实际运行的是多线程吗？

作者回复

我理解不是一回事，`openjdk`目前没有协程，`Loom`过程在做相关事情

tyson

2018-06-14

- 1、站在应用程序方面，只创建了一个线程。
- 2、站在jvm方面，肯定还有gc等其余线程。

总结：

1. 线程是系统调度的最小单元，应该是进程吧。线程是操作系统的资源，在运行的时候会打开文件描述符等。
2. `resume`, `stop`, `suspend`等已经被废弃了
3. 线程的等待和唤醒，建议使用`reentrantlock`.`condition wait/notify`方法
4. 可以使用线程的`join`方法、`countdownlatch`、`cyclicbarrier`、`future`等进行线程的等待

作者回复

正解

雷霹雳的爸爸

2018-06-14

老师今天这课后题，又打脸了平时工作不仔细的地方，我首先想到的是好歹得`sleep`一下或打个断点用类似`visualvm`的工具看下，或者`top`之类数一下，赶着出门，回来试下

作者回复

主要是为了加深理解，这种也就是老学究关心，哈哈

高杰

2018-06-14

有几个弱引用，虚引用的地方，音频和文字对不上。把我搞晕了。  
应该有2个线程，还有jvm的gc线程？还有第三个线程吗？

作者回复

翻译修正了，谢谢指出；你试试用比如最简单的`jstack`查看下，不止这些哦

甘建新

2018-06-14

线程得内存分配是怎么样的呢？

作者回复

后边虚拟机那边介绍

hanmashashou

2018-06-14

`weak` 应该不是幻象引用吧

作者回复

汗颜，已修正

灰飞灰猪不会灰飞.烟灭

2018-06-14

老师 `future`模式是怎么异步返回结果的呢？是不是把每个线程的运行结果放到`queue`中，然后轮询`queue`返回结果？

作者回复

是说`FutureTask`的实现吗？我记得是有区别的

食指可爱多

2018-06-15

我了解确定线程有：任务线程，Main线程，垃圾回收线程，还有些线程没细心关注名字和用途，惭愧了。可以在业务线程中等待，然后在命令行用`jstack`看当前jvm的线程堆栈。

作者回复

其他就包括我们前面章节说过的`finalizer`，各种`cleaner`等，还有事件处理等

2018-06-14



## 第18讲 | 什么情况下Java程序会产生死锁？如何定位、修复？

2018-06-16 杨晓峰



第18讲 | 什么情况下Java程序会产生死锁？如何定位、修复？  
杨晓峰  
00:20 / 09:37

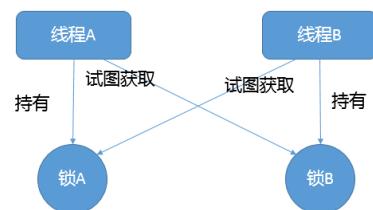
今天，我会介绍一些日常开发中类似线程死锁等问题的排查经验，并选择一两个我自己修复过或者诊断过的核心类库死锁问题作为例子，希望不仅能在面试时，包括在日常工作中也能对你有所帮助。

今天我要问你的是，[什么情况下Java程序会产生死锁？如何定位、修复？](#)

## 典型回答

死锁是一种特定的程序状态，在实体之间，由于循环依赖导致彼此一直处于等待之中，没有任何个体可以继续前进。死锁不仅仅是在线程之间会发生，存在资源独占的进程之间同样也可能出现死锁。通常来说，我们大多是聚焦在多线程场景中的死锁，指两个或多个线程之间，由于互相持有对方需要的锁，而永久处于阻塞的状态。

你可以利用下面的示例图理解基本的死锁问题：



定位死锁最常见的方式就是利用jstack等工具获取线程栈，然后定位互相之间的依赖关系，进而找到死锁。如果是比较明显的死锁，往往jstack等就能直接定位，类似JConsole甚至可以在图形界面进行有限的死锁检测。

如果程序运行时发生了死锁，绝大多数情况下都是无法在线解决的，只能重启、修正程序本身问题。所以，代码开发阶段互相审查，或者利用工具进行预防性排查，往往也是很重要的。

## 考点分析

今天的问题偏向于实用场景，大部分死锁本身并不难定位，掌握基本思路和工具使用，理解线程相关的基本概念，比如各种线程状态和同步、锁、Latch等并发工具，就已经足够解决大多数问题了。

针对死锁，面试官可以深入考察：

- 抛开字面上的概念，让面试者写一个可能死锁的程序，顺便也考察下基本的线程编程。
- 诊断死锁有哪些工具，如果是分布式环境，可能更关心能否用API实现吗？
- 后期诊断死锁还是挺痛苦的，经常加班，如何在编程中尽量避免一些典型场景的死锁，有其他工具辅助吗？

## 知识扩展

在分析开始之前，先以一个基本的死锁程序为例，我在这里只用了两个嵌套的synchronized去获取锁，具体如下：

```
public class DeadLockSample extends Thread {
    private String first;
    private String second;
    public DeadLockSample(String name, String first, String second) {
        super(name);
        this.first = first;
        this.second = second;
    }

    public void run() {
        synchronized (first) {
            System.out.println(this.getName() + " obtained: " + first);
            try {
                Thread.sleep(1000L);
                synchronized (second) {
                    System.out.println(this.getName() + " obtained: " + second);
                }
            } catch (InterruptedException e) {
                // Do nothing
            }
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    String lockA = "lockA";
    String lockB = "lockB";
    DeadLockSample t1 = new DeadLockSample("Thread1", lockA, lockB);
    DeadLockSample t2 = new DeadLockSample("Thread2", lockB, lockA);
    t1.start();
    t2.start();
    t1.join();
    t2.join();
}
}
```

这个程序编译执行后，几乎每次都可以重现死锁，请看下面截取的输出。另外，这里有个比较有意思的地方，为什么我先调用Thread1的start，但是Thread2却先打印出来了呢？这是因为线程调度依赖于（操作系统）调度器，虽然你可以通过优先级之类进行影响，但是具体情况是不确定的。

```
C:\>e:\jdk-9\bin\java DeadLockSample
Thread2 obtained: lockB
Thread1 obtained: lockA
```

下面来模拟问题定位，我就选取最常见的jstack，其他一些类似JConsole等图形化的工具，请自行查找。

首先，可以使用jps或者系统的ps命令、任务管理器等工具，确定进程ID。

其次，调用jstack获取线程栈：

```
$JAVA_HOME\bin\jstack your_pid
```

然后，分析得到的输出，具体片段如下：

```
"Thread2" #13 prio=5 os_prio=0 tid=0x00000000006d5b000 nid=0x1ff90 waiting for monitor
entry [0x000000002c34f00]
java.lang.Thread.State: BLOCKED (on object monitor)
at DeadLockSample.run(DeadLockSample.java:17)
- waiting to lock <0x00000006d18f4f70> (a java.lang.String)
- locked <0x00000006d18f4fa0> (a java.lang.String)

"Thread1" #12 prio=5 os_prio=0 tid=0x00000000006d5a800 nid=0x1558 waiting for monitor
entry [0x000000002c44f00]
java.lang.Thread.State: BLOCKED (on object monitor)
at DeadLockSample.run(DeadLockSample.java:17)
- waiting to lock <0x00000006d18f4fa0> (a java.lang.String)
- locked <0x00000006d18f4f70> (a java.lang.String)
...

```

最后，结合代码分析线程栈信息。上面这个输出非常明显，找到处于BLOCKED状态的线程，按照试图获取（waiting）的锁ID（请看我标记为相同颜色的数字）查找，很快就定位问题。jstack本身也会把类似的简单死锁抽取出，直接打印出来。

在实际应用中，类死锁情况未必有如此清晰的输出，但是总体上可以理解为：

区分线程状态 -> 查看等待目标 -> 对比Monitor等待状态

所以，理解线程基本状态和并发相关元素是定位问题的关键，然后配合程序调用栈结构，基本就可以定位到具体的问题代码。

如果我们是开发自己的管理工具，需要用更加程序化的方式扫描服务进程、定位死锁，可以考虑使用Java提供的标准管理API，[ThreadMXBean](#)，其直接就提供了findDeadlockedThreads()方法用于定位。为方便说明，我修改了DeadLockSample，请看下面的代码片段。

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

```

ThreadMXBean mbean = ManagementFactory.getThreadMXBean();
Runnable d1Check = new Runnable() {

    @Override
    public void run() {
        long[] threadIds = mbean.findDeadlockedThreads();
        if (threadIds != null) {
            ThreadInfo[] threadInfos = mbean.getThreadInfo(threadIds);
            System.out.println("Detected deadlock threads:");
            for (ThreadInfo threadInfo : threadInfos) {
                System.out.println(threadInfo.getThreadName());
            }
        }
    }
};

ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
// 等待5秒，然后每10秒进行一次死锁扫描
scheduler.scheduleAtFixedRate(d1Check, 5L, 10L, TimeUnit.SECONDS);
// 死锁样例代码...
}

```

重新编译执行，你就能看到死锁被定位到的输出。在实际应用中，就可以据此收集进一步的信息，然后进行预警等后续处理。但是要注意的是，对线程进行快照本身是一个相对重量级的操作，还是要慎重选择频度和时机。

如何在编程中尽量预防死锁呢？

首先，我们来总结一下前面例子中死锁的产生包含哪些基本元素。基本上死锁的发生是因为：

- 互斥条件，类似Java中Monitor都是独占的，要么是我用，要么是你用。
- 互斥条件是长期持有的，在使用结束之前，自己不会释放，也不能被其他线程抢占。
- 循环依赖关系，两个或者多个个体之间出现了锁的链条环。

所以，我们可以据此分析可能的避免死锁的思路和方法。

第一种方法

如果可能的话，尽量避免使用多个锁，并且只有需要时才持有锁。否则，即使是非常精通并发编程的工程师，也难免会掉进坑里，嵌套的synchronized或者lock非常容易出问题。

我举个[例子](#)，Java NIO的实现代码向来以锁多著称，一个原因是，其本身模型就非常复杂，某种程度上是不得不如此；另外是在设计时，考虑到既要支持阻塞模式，又要支持非阻塞模式。直接结果就是，一些基本操作如connect，需要操作三个锁以上。在最近的一个JDK改进中，就发生了死锁现象。

我将其简化为下面的伪代码，问题是暴露在HTTP/2客户端中，这是个非常现代的反应式风格的API，非常推荐学习使用。

```

/// Thread HttpClient-6-SelectorManager:
readLock.lock();
writeLock.lock();
// 持有readLock/writeLock，调用close()需要获得closeLock
close();
// Thread HttpClient-6-Worker-2 持有closeLock
implCloseSelectableChannel(); //想获得readLock

```

在close发生时，HttpClient-6-SelectorManager线程持有readLock/writeLock，试图获得closeLock；与此同时，另一个HttpClient-6-Worker-2线程，持有closeLock，试图获得readLock，这就不可避免地进入了死锁。

这里比较难懂的地方在于，closeLock的持有状态（就是我标记为绿色的部分）并没有在线程栈中显示出来，请参考我在下图中标记的部分。

```
"HttpClient-6-SelectorManager" #35 daemon prio=5 os_prio=0 tid=0x00007f966c0d8000 nid=0x3d4 waiting on condition [0x00007f96c8176000]
java.lang.Thread.State: WAITING (parking)
at jdk.internal.misc.Unsafe.park(java.base@11~internal/java.base@11-internal/jdk.internal.misc.Unsafe.park)
- parking to wait for <0x00000000e0cbf660> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
at java.util.concurrent.LockSupport.park(java.base@11~internal/java.base@11-internal/java.util.concurrent.locks.LockSupport.java:194)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(java.base@11~internal/java.util.concurrent.locks.AbstractQueuedSynchronizer.java:917)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(java.base@11~internal/java.util.concurrent.locks.AbstractQueuedSynchronizer.java:917)
at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@11~internal/java.util.concurrent.locks.AbstractQueuedSynchronizer.java:1240)
at java.util.concurrent.locks.ReentrantLock.lock(java.base@11~internal/java.util.concurrent.locks.ReentrantLock.java:267)
at sun.nio.ch.SocketChannelImpl.implCloseSelectableChannel(java.base@11~internal/sun.nio.ch.SocketChannelImpl.java:839)
at java.nio.channels.spi.AbstractSelectableChannel.implCloseChannel(java.base@11~internal/java.nio.channels.spi.AbstractSelectableChannel.java:241)
at java.nio.channels.spi.AbstractInterruptibleChannel.close(java.base@11~internal/java.nio.channels.spi.AbstractInterruptibleChannel.java:112)
locked <0x00000000e0cbf660> (a java.lang.Object)
at jdk.incubator.http.PlainHttpConnection.close(jdk.incubator.http.PlainHttpConnection.java:189)
- locked <0x00000000e0cbf378> (a jdk.incubator.http.PlainHttpConnection)
at jdk.incubator.http.HttpExchange.cancelImpl(jdk.incubator.httpclient@11~internal/jdk.incubator.http.HttpExchange.java:379)
- locked <0x00000000e0cbf128> (a java.lang.Object)
at jdk.incubator.http.HttpExchange.cancel(jdk.incubator.httpclient@11~internal/jdk.incubator.http.HttpExchange.java:366)
at jdk.incubator.http.HttpExchange.cancel(jdk.incubator.httpclient@11~internal/jdk.incubator.http.HttpExchange.java:174)
at jdk.incubator.http.MultiExchange.cancel(jdk.incubator.httpclient@11~internal/jdk.incubator.http.MultiExchange.java:212)
at jdk.incubator.http.MultiExchange$TimeEvent.handle(jdk.incubator.httpclient@11~internal/jdk.incubator.http.MultiExchange.java:354)
at jdk.incubator.http.HttpClientImpl.purgeTimeoutsAndReturnNextDeadline(jdk.incubator.httpclient@11~internal/java.util.concurrent.TimeUnit.java:101)
at jdk.incubator.http.HttpClientImpl.access$300(jdk.incubator.httpclient@11~internal/java.util.concurrent.TimeUnit.java:76)
at jdk.incubator.http.HttpClientImpl$SelectorManager.run(jdk.incubator.httpclient@11~internal/java.util.concurrent.TimeUnit.java:698)

"HttpClient-6-Worker-2" #38 daemon prio=5 os_prio=0 tid=0x00007f9660008000 nid=0x3d8 waiting for monitor entry [0x00007f965aceb000]
java.lang.Thread.State: BLOCKED (on object monitor)
at java.nio.channels.spi.AbstractSelectableChannel.close(java.base@11~internal/java.nio.channels.spi.AbstractSelectableChannel.java:109)
- waiting to lock <0x00000000e0cbf660> (a java.lang.Object)
at sun.nio.ch.SocketChannelImpl.connect(java.base@11~internal/sun.nio.ch.SocketChannelImpl.java:663)
at jdk.incubator.http.PlainHttpConnection.lambda$connectAsync$0(jdk.incubator.httpclient@11~internal/jdk.incubator.http.PlainHttpConnection.java:110)

```

更加具体来说, 请查看[SocketChannelImpl](#)的663行, 对比`implCloseSelectableChannel()`方法实现和[AbstractInterruptibleChannel.close\(\)](#)在109行的代码, 这里就不展示代码了。

所以, 从程序设计的角度反思, 如果我们赋予一段程序太多的职责, 出现“既要...又要...”的情况时, 可能就需要我们审视下设计思路或目的是否合理了。对于类库, 因为其基础、共享的定位, 比应用开发往往更加令人苦恼, 需要仔细斟酌之间的平衡。

## 第二种方法

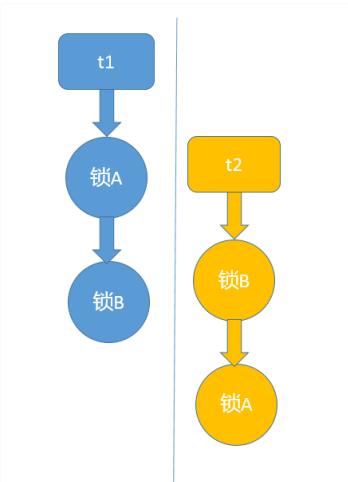
如果必须使用多个锁, 尽量设计好锁的获取顺序, 这个说起来简单, 做起来不容易, 你可以参看著名的[银行家算法](#)。

一般的情况, 我建议可以采取些简单的辅助手段, 比如:

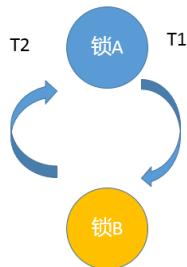
- 将对象(方法)和锁之间的关系, 用图形化的方式表示分别抽取出来, 以今天最初讲的死锁为例, 因为是调用了同一个线程所以更加简单。



- 然后根据对象之间组合、调用的关系对比和组合, 考虑可能调用时序。



- 按照可能时序合并，发现可能死锁的场景。



#### 第三种方法

使用带超时的方法，为程序带来更多可控性。

类似Object.wait(...)或者CountDownLatch.await(...), 都支持所谓的timed\_wait, 我们完全可以就不假定该锁一定会获得, 指定超时时间, 并为无法得到锁时准备退出逻辑。

并发Lock实现, 如ReentrantLock还支持非阻塞式的获取锁操作tryLock(), 这是一个插队行为(barging), 并不在乎等待的公平性, 如果执行时对象恰好没有被独占, 则直接获取锁。有时, 我们希望条件允许就尝试插队, 不然就按照现有公平性规则等待, 一般采用下面的方法:

```

if (lock.tryLock() || lock.tryLock(timeout, unit)) {
    ...
}
    
```

#### 第四种方法

业界也有一些其他方面的尝试, 比如通过静态代码分析(如FindBugs)去查找固定的模式, 进而定位可能的死锁或者竞争情况。实践证明这种方法也有一定作用, 请参考[相关文档](#)。

除了典型应用中的死锁场景, 其实还有一些更令人头疼的死锁, 比如类加载过程发生的死锁, 尤其是在框架大量使用自定义类加载时, 因为往往不是在应用本身的代码库中, jstack等工具也不见得能够显示全部锁信息, 所以处理起来比较棘手。对此, Java有[官方文档](#)进行了详细解释, 并针对特定情况提供了相应JVM参数和基本原则。

今天, 我从样例程序出发, 介绍了死锁产生原因, 并帮你熟悉了排查死锁基本工具的使用和典型思路, 最后结合实例介绍了实际场景中的死锁分析方法与预防措施, 希望对你有所帮助。

#### -一课一练

关于今天我们讨论的题目你做到心中有数了吗? 今天的思考题是, 有时候并不是阻塞导致的死锁, 只是某个线程进入了死循环, 导致其他线程一直等待, 这种问题如何诊断呢?

请你在留言区写写你对这个问题的思考, 我会选出经过认真思考的留言, 送给你一份学习奖励礼券, 欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢? 你可以“请朋友读”, 把今天的题目分享给好友, 或许你能帮到他。



石头狮子

1. 死锁的另一个好朋友就是饥饿。死锁和饥饿都是线程活跃性问题。  
实践中死锁可以使用jvm自带的工具进行排查。

2. 课后题指出的死循环死锁可以认为是自旋锁死锁的一种，其他线程因为等待不到具体的信号提示，导致线程一直饥饿。  
这种情况下可以查看线程cpu使用情况，排查出使用cpu时间片最高的线程，再打出该线程的堆栈信息，排查代码。

3. 基于互斥量的锁如果发生死锁往往cpu使用率较低，实践中也可以从这一方面进行排查。

作者回复

很好的总结

I am a psycho

当是死循环引起的其他线程阻塞，会导致cpu飙升，可以先看下cpu的使用率。

作者回复

对，比如Linux上，可以使用top命令配合grep Java之类，找到忙的pid，然后，转换成16进制，就是jstack输出中的格式，再定位代码

黑子

任务线程规范命名，详细记录逻辑运行日志。jstack查看线程状态。

作者回复

不错

西卿十六夜

老师，面试遇到过一个很刁钻的问题。如何在jvm不重启的情况下杀死一个线程，在stop被移除后，如果线程存在死锁那是否意味着必须要修复代码再重启虚拟机呢？

作者回复

不知道有什么好办法，也许用我例子哪个API去找到死锁线程，想办法把死锁条件打开；但我觉得这东西不靠谱，假设真的解除死锁，你还能保证程序正确性吗，这不会是个通用解决方案

另外，即使以前有stop方法，blocked状态的线程也是关不了的吧，它不响应你的请求的

tracer

看了下jconsole检测死锁功能的源码，果然也是用ThreadMXBean获取死锁线程并分组，然后打印相关线程信息的。

curlev3

回答老师的问题

可以通过linux下top命令查看cpu使用率较高的java进程，进而用top -H p pid查看该java进程中cpu使用率较高的线程。再用jstack命令查看线程具体调用情况，排查问题。

作者回复

非常不错

coolboy

杨老师，问个小白问题，java的线程状态有BLOCK、WAITING状态，使用java的内置关键字synchronized时，会出现BLOCK状态。但如果用java的reentrantLock时，也会出现BLOCK状态的吗，不应该只有WAITING状态的？

作者回复

是

Miaozhe

杨老师，Sorry。接着上了问题，是我的进程PID搞错了，应该用Javax，我用或eclipse的PID了。

jacy

尽然可以用ThreadMXBean来抓线程死锁信息，受教了。  
循环死锁，会导致cpu某线程的cpu时间片占用率相当高，可以结合操作系统工具分析出线程号，然后用jstack分析线程  
作者回复

不错

Miaozhe

杨老师，我Win7系统，Java 8上运行Dead Lock Simple例子，通过Jstack获取的Thread 1和Thread 2的线程状态，都是Runnable,但是Waiting on Condition[0x 00000000]。但是，我通过Thread Group打印出来，两个线程状态都是Block。  
晕乎了。。。。

作者回复

我不能重现，你是Jdk8update多少？synchronized正常理解就是Blocked

残阳

以前做排查的时候看thread dump，一般都会直接按一些关键字搜索。比如wait, lock之类，然后再找重复的内存地址。看完这篇文章之后感觉对死锁的理解更深刻。

作者回复

谢谢，地址也很重要

肖一林

一课一练：  
最典型的场景是nio的Selector类，这个类内部有三个集合，并且对这些集合做了同步。如果多个线程同时操作一个Selector，就很容易发生死锁。它的select方法会一直拿着锁，并且循环等待事件发生。如果有其他线程在修改它内部的集合数据，就死锁了。

同样用stack可以发现问题，找出被阻塞的线程，看它等待哪个锁，再找到持有这把锁的线程，这个线程一搬处于运行状态

作者回复

不错，selected key 和 cancelled key的集合不是线程安全的，我记得标准文档就建议

肖一林

初学nio的时候确实动不动就发生死锁。现在好像也没有特别好的教程，都是一些java.io的教程。很多教程跟不上技术的迭代。也可能是因为直接io编程在项目实践中偏少。

另外，这个小程序的图片不能放大看，不知道是微信的原因还是小程序的原因。老师看到了帮忙反馈一下。

作者回复

nio确实教程少，书籍也不好找 Java IO, NIO, NIO2好像也没引进；如果想系统学习，我建议买本《netty实战》，Java自己的nio定位偏重于基础性API，与终端应用需求有点鸿沟









## 第19讲 | Java并发包提供了哪些并发工具类?

2018-06-19 杨晓峰



第19讲 | Java并发包提供了哪些并发工具类?  
杨晓峰  
00:35 / 10:32

通过前面的学习，我们一起回顾了线程、锁等各种并发编程的基本元素，也逐步涉及了Java并发包中的部分内容，相信经过前面的热身，我们能够更快地理解Java并发包。

今天我要问你的是，[Java并发包提供了哪些并发工具类？](#)

#### 典型回答

我们通常所说的并发包也就是java.util.concurrent及其子包，集中了Java并发的各种基础工具类，具体主要包括几个方面：

- 提供了比synchronized更加高级的各种同步结构，包括CountDownLatch、CyclicBarrier、Semaphore等，可以实现更加丰富的多线程操作，比如利用Semaphore作为资源控制器，限制同时进行工作的线程数量。
- 各种线程安全的容器，比如最常见的ConcurrentHashMap、有序的ConcurrentSkipListMap，或者通过类似快照机制，实现线程安全的动态数组CopyOnWriteArrayList等。
- 各种并发队列实现，如各种BlockingQueue实现，比较典型的ArrayBlockingQueue、SynchronousQueue或针对特定场景的PriorityBlockingQueue等。
- 强大的Executor框架，可以创建各种不同类型的线程池，调度任务运行等，绝大部分情况下，不再需要自己从头实现线程池和任务调度器。

#### 考点分析

这个题目主要考察你对并发包了解程度，以及是否有实际使用经验。我们进行多线程编程，无非是达到几个目的：

- 利用多线程提高程序的扩展能力，以达到业务对吞吐量的要求。
- 协调线程间调度、交互，以完成业务逻辑。
- 线程间传递数据和状态，这同样是实现业务逻辑的需要。

所以，这道题目只能算作简单的开始，往往面试官还会进一步考察如何利用并发包实现某个特定的用例，分析实现的优缺点等。

如果你在这方面的基础比较薄弱，我的建议是：

- 从总体上，把握住几个主要组成部分（前面回答中已经简要介绍）。
- 理解具体设计、实现能力和。
- 再深入掌握一些比较典型工具类的适用场景、用法甚至是原理，并熟练写出典型的代码用例。

掌握这些通常就够了，毕竟并发包提供了方方面面的工具，其实很少有机会能在应用中全面使用过，扎实地掌握核心功能就非常不错了。真正特别深入的经验，还是得靠在实际场景中踩坑来获得。

#### 知识扩展

首先，我们来看看并发包提供的丰富同步结构。前面几讲已经分析过各种不同的显式锁，今天我将专注于

- [CountDownLatch](#)，允许一个或多个线程等待某些操作完成。
- [CyclicBarrier](#)，一种辅助性的同步结构，允许多个线程等待到达某个屏障。
- [Semaphore](#)，Java版本的信号量实现。

Java提供了经典信号量 ([Semaphore](#)) 的实现，它通过控制一定数量的允许 (permit) 的方式，来达到限制通用资源访问的目的。你可以想象一下这个场景，在车站、机场等出

租车时，当很多空出租车就位时，为防止过度拥挤，调度员指挥排队等待坐车的队伍一次进来5个人上车，等这5个人坐车出发，再放进去下一批，这和Semaphore的工作原理有些类似。

你可以试试使用Semaphore来模拟实现这个调度过程：

```
import java.util.concurrent.Semaphore;
public class UsualSemaphoreSample {
    public static void main(String[] args) throws InterruptedException {
        System.out.println("Action...GO!");
        Semaphore semaphore = new Semaphore(5);
        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new SemaphoreWorker(semaphore));
            t.start();
        }
    }
}
class SemaphoreWorker implements Runnable {
    private String name;
    private Semaphore semaphore;
    public SemaphoreWorker(Semaphore semaphore) {
        this.semaphore = semaphore;
    }
    @Override
    public void run() {
        try {
            log("is waiting for a permit!");
            semaphore.acquire();
            log("acquired a permit!");
            log("executed!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            log("released a permit!");
            semaphore.release();
        }
    }
    private void log(String msg){
        if (name == null) {
            name = Thread.currentThread().getName();
        }
        System.out.println(name + " " + msg);
    }
}
```

这段代码是比较典型的Semaphore示例，其逻辑是，线程试图获得工作允许，得到许可则进行任务，然后释放许可，这时等待许可的其他线程，就可获得许可进入工作状态，直到全部处理结束。编译运行，我们就能看到Semaphore的允许机制对工作线程的限制。

但是，从具体节奏来看，其实并不符合我们前面场景的需求，因为本例中Semaphore的用法实际是保证，一直有5个人可以试图乘车，如果有1个人出发了，立即就有排队的人获得许可，而这并不完全符合我们前面的要求。

那么，我再修改一下，演示个非典型的Semaphore用法。

```
import java.util.concurrent.Semaphore;
public class AbnormalSemaphoreSample {
    public static void main(String[] args) throws InterruptedException {
        Semaphore semaphore = new Semaphore(0);
        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new MyWorker(semaphore));
            t.start();
        }
        System.out.println("Action...GO!");
        semaphore.release(5);
        System.out.println("Wait for permits off");
        while (semaphore.availablePermits()!=0) {
            Thread.sleep(100L);
        }
        System.out.println("Action...GO again!");
        semaphore.release(5);
    }
}
class MyWorker implements Runnable {
    private Semaphore semaphore;
    public MyWorker(Semaphore semaphore) {
        this.semaphore = semaphore;
    }
    @Override
    public void run() {
        try {
```

```
        semaphore.acquire();
        System.out.println("Executed!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

注意，上面的代码，更侧重的是演示Semaphore的功能以及局限性。其实有很多线程编程中的反实践，比如使用了sleep来协调任务执行，而且使用轮询调用availablePermits来检测信号量获取情况，这都是很无效并且脆弱的，通常只是用在测试或者诊断场景。

总的来说，我们可以看出Semaphore就是个计数器，其基本逻辑基于acquire/release，并没有太复杂的同步逻辑。

如果Semaphore的数值被初始化为1，那么一个线程就可以通过acquire进入互斥状态，本质上和互斥锁是非常相似的。但是区别也非常明显，比如互斥锁是有持有者的，而对于Semaphore这种计数器结构，虽然有类似功能，但其实不存在真正意义的持有者，除非我们进行扩展包装。

下面，来看看CountDownLatch和CyclicBarrier，它们的行为有一定的相似度，经常会被考察二者有什么区别，我来简单总结一下。

- CountDownLatch是不可以重置的，所以无法重用；而CyclicBarrier则没有这种限制，可以重用。
  - CountDownLatch的基本操作组合是countDown/await。调用await的线程阻塞等待countDown足够的次数，不管你是在一个线程还是多个线程里countDown，只要次数足够即可。所以就像Brain Goetz说过的，CountDownLatch操作的是事件。
  - CyclicBarrier的基本操作组合，则就是await，当所有的伙伴（parties）都调用了await，才会继续进行任务，并自动进行重置。注意，正常情况下，CyclicBarrier的重置都是自动发生的，如果我们调用reset方法，但还有线程在等待，就会导致等待线程被打扰，抛出BrokenBarrierException异常。CyclicBarrier侧重点是线程，而不是调用事件，它的典型应用场景是用线程等待并发线程结束。

如果用CountDownLatch去实现上面的排队场景，该怎么做呢？假设有10个人排队，我们将其分成5个人一批，通过CountDownLatch来协调批次，你可以试试下面的示例代码。

```
import java.util.concurrent.CountDownLatch;
public class LatchSample {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(6);
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new FirstBatchWorker(latch));
            t.start();
        }
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new SecondBatchWorker(latch));
            t.start();
        }
        // 注意这里也是演示目的的逻辑，并不是推荐的协调方式
        while (latch.getCount() != 1) {
            Thread.sleep(100L);
        }
        System.out.println("Wait for first batch finish");
        latch.countDown();
    }
}
class FirstBatchWorker implements Runnable {
    private CountDownLatch latch;
    public FirstBatchWorker(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run() {
        System.out.println("First batch executed!");
        latch.countDown();
    }
}
class SecondBatchWorker implements Runnable {
    private CountDownLatch latch;
    public SecondBatchWorker(CountDownLatch latch) {
        this.latch = latch;
    }
    @Override
    public void run() {
        try {
            latch.await();
            System.out.println("Second batch executed!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

CountDownLatch的调度方式相对简单，后一批次的线程进行await，等待前一批countDown足够多次。这个例子也从侧面体现出了它的局限性，虽然它也能够支持10个人排队的情况，但是因为不能重用，如果要支持更多人排队，就不必依赖一个CountDownLatch进行了。其编译运行输出如下：

```
C:\>e:\jdk-9\bin>java LatchSample
First batch executed!
Wait for first batch finish
Second batch executed!
```

在实际应用中的条件依赖，往往没有这么别扭，CountDownLatch用于线程间等待操作结束是非常简单普遍的用法。通过countDown/await组合进行通信是很高效的，通常不建议使用例子里那个循环等待方式。

如果用CyclicBarrier来表达这个场景呢？我们知道CyclicBarrier其实反映的是线程并行运行时的协调，在下面的示例里，从逻辑上，5个工作线程其实更像是代表了5个可以就绪的空车，而不再是5个乘客，对比前面CountDownLatch的例子更有助于我们区别它们的抽象模型，请看下面的示例代码：

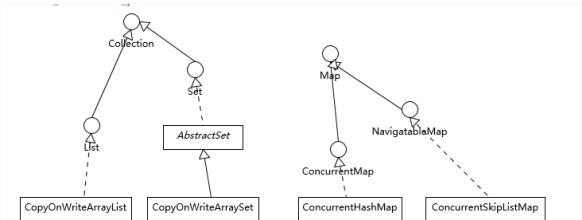
```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierSample {
    public static void main(String[] args) throws InterruptedException {
        CyclicBarrier barrier = new CyclicBarrier(5, new Runnable() {
            @Override
            public void run() {
                System.out.println("Action...GO again!");
            }
        });
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new CyclicWorker(barrier));
            t.start();
        }
    }
    static class CyclicWorker implements Runnable {
        private CyclicBarrier barrier;
        public CyclicWorker(CyclicBarrier barrier) {
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                for (int i=0; i<3 ; i++){
                    System.out.println("Executed!");
                    barrier.await();
                }
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

为了让输出更能表达运行时序，我使用了CyclicBarrier特有的barrierAction，当屏障被触发时，Java会自动调度该动作。因为CyclicBarrier会自动进行重置，所以这个逻辑其实可以非常自然的支持更多排队人数。其编译输出如下：

```
C:\>e:\jdk-9\bin>java CyclicBarrierSample
Executed!
Executed!
Executed!
Executed!
Executed!
Action...GO again!
Executed!
Executed!
Executed!
Executed!
Action...GO again!
Executed!
Executed!
Executed!
Executed!
Action...GO again!
```

Java并发库还提供了Phaser，功能与CountDownLatch很接近，但是它允许线程动态地注册到Phaser上面，而CountDownLatch显然是不能动态设置的。Phaser的设计初衷是，实现多个线程类似步骤、阶段场景的协调，线程注册等待屏障条件触发，进而协调彼此行动，具体请参考这个[例子](#)。

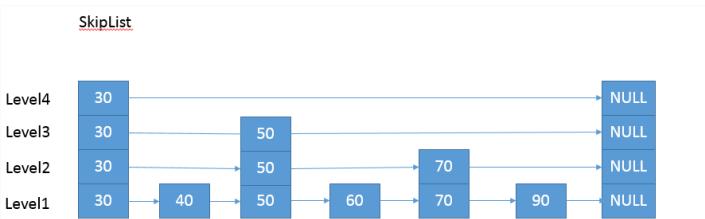
接下来，我来梳理下并发包里提供的线程安全Map、List和Set。首先，请参考下面的类图。



你可以看到，总体上种类和结构还是比较简单的，如果我们的应用侧重于Map放入或者获取的速度，而不在乎顺序，大多推荐使用ConcurrentHashMap，反之则使用ConcurrentSkipListMap；如果我们需要对大量数据进行非常频繁地修改，ConcurrentSkipListMap也可能表现出优势。

我在前面的专栏，谈到了普通无顺序场景选择HashMap，有顺序场景则可以选择类似TreeMap等，但是为什么并发容器里面没有ConcurrentTreeMap呢？

这是因为TreeMap要实现高效的线程安全是非常困难的，它的实现基于复杂的红黑树。为保证访问效率，当我们插入或删除节点时，会移动节点进行平衡操作，这导致在并发场景中难以进行合理粒度的同步。而SkipList结构则要相对简单很多，通过层次结构提高访问速度，虽然不够紧凑，空间使用有一定提高( $O(n\log n)$ )，但是在增删元素时线程安全的开销要好很多。为了方便你理解SkipList的内部结构，我画了一个示意图。



关于两个CopyOnWrite容器，其实CopyOnWriteArrayList是通过包装了CopyOnWriteArraySet来实现的，所以在学习时，我们可以专注于理解一种。

首先，CopyOnWrite到底是什么意思呢？它的原理是，任何修改操作，如add、set、remove，都会拷贝原数组，修改后替换原来的数组，通过这种防御性的方式，实现另类的线程安全。请看下面的代码片段，我进行注释的地方，可以清晰地理解其逻辑。

```

public boolean add(E e) {
    synchronized (lock) {
        Object[] elements = getArray();
        int len = elements.length;
        // 拷贝
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;
        // 替换
        setArray(newElements);
        return true;
    }
}
final void setArray(Object[] a) {
    array = a;
}
  
```

所以这种数据结构，相对比较适合读多写少的操作，不然修改的开销还是非常明显的。

今天我对Java并发包进行了总结，并且结合实例分析了各种同步结构和部分线程安全容器，希望对你有所帮助。

#### -课一练

关于今天我们讨论的题目你做到心中有数了吗？留给你的思考题是，你使用过类似CountDownLatch的同步结构解决实际问题吗？谈谈你的使用场景和心得。

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



天秤座的选择

做android的，一个页面有A,B,C三个网络请求，其中请求C需要请求A和请求B的返回数据作为参数，用过CountdownLatch解决。

2018-06-20

三个石头

2018-06-19

你用的Semaphore第二个例子，构造函数中为啥为0，信号量不是非负整数吗？

石头狮子

2018-06-20

列举实践中两个应用并发工具的场景：

1. 请求熔断器，使用 Semaphore 涔断某些请求线程，待系统恢复以后再逐步释放信号量。
2. Worker 搜索停止标志。使用 countdownlatch 标记 Worker 找到的结果个数，达到结果后其他线程不再继续执行。

夏天◆◆

2018-06-20

以前使用countdownlatch进行并发异常的模拟，来修改bug，具体是在发生异常的错误堆栈上进行await，在某些条件处或触发点进行countdown，来尽可能模拟触发异常时的场景，很多可以必现，修改之后没有问题，才算解决一个并发异常

TWO STRINGS

2018-06-20

ArrayBlockingQueue使用了两个condition来分别控制put和take的阻塞与唤醒，但是我在想好像只用一个condition也可以，因为put和take只会有一个是处于阻塞等待状态。所以设计成两个condition的原因是什么呢？只是为了提高可读性么？

Daydayup

2018-06-22

CountDownLatch最近还真用上了。我的需求是每个对象一个线程，分别在每个线程里计算各自的数据，最终等到所有线程计算完毕，我还需要将每个有共通的对象进行合并，所以用它很合适。

作者回复

2018-06-22

合适的场景

Leiy

2018-06-19

对于CopyOnWriteArrayList，适用于读多写少的场景，这个比较好理解，但是在实际使用时候，读写比占多少时候，可以使用？心里还是没数，这个怎么去衡量？

扫地僧的功夫梦

2018-06-19

17讲的问题，留言有点晚，老师可能不会看，想得到老师的回复：调用notify()或notifyAll()方法后线程是处于阻塞状态吧，因为线程还没获取到锁。

作者回复

2018-06-19

是说调用notify的那个线程的状态吗？

不是的，这里有很多方面。

阻塞一般发生在进入同步块儿时。

notify并不会让出当前的monitor。

可以用wait释放锁，但是进入waiting状态。

不建议靠记忆去学习，类似问题我建议思考一下：能不能用一段程序验证，需不需要利用什么工具；别忘了从Javadoc得到初步信息

授人以鱼不如授人以渔，提供答案更重要，最好不要怀疑我这里的每个结论，自己写代码去玩玩

xinxin◆◆

2018-07-07

老师为什么我用ConcurrentHashMap执行remove操作的时候cpu总是跳得很高，hashmap就还好没那么夸张。。现在为了线程安全还是用ConcurrentHashMap，但执行remove操作的线程一多经常就卡死了。

作者回复

2018-07-12

你是什么版本jdk？

杨文奇

2018-07-04

Android中多线程上传多张图片到阿里云，将每个线程返回的图片url地址，组合成一个数组传给服务端

jacy

2018-06-25

感觉CountDownLatch有点像c++中的条件锁，想问一下老师，可否给点从c++转java的建议。

作者回复

2018-06-26

这个...经验谈不上，我也没这经验；或者你可以对比二者的区别，加深理解；  
为什么转？希望达到什么目标？

MiaoZhe

2018-06-22

问个问题，CyclicBarrier初始化的Parties值是5，最后Await的是6，这种情况会是什么样子？

洛措

2018-06-22

“提供了比 synchronized 更加高级的各种同步结构，包括 CountDownLatch、CyclicBarrier、Semaphore 等，可以实现更加丰富的多线程操作，比如利用 Semaphore 作为资源控制器，限制同时进行工作的线程数量。”，发现一处拼写错误，在这一段中，Semaphore 拼写错了，应该是 Semaphore。

步亮

2018-06-21

SemaphoreWorker类应该为static

作者回复

2018-06-22

哈哈，那不是inner class

zjh

2018-06-21

感觉再分布式的情况下，单体应用中需要多个线程并行的情况可能会被分散在多个应用里面，可能很少会用到CountDownLatch和cyclicbarrier，semaphore倒是比较适合用在分布式的场景下，用来做一些限流。

作者回复

2018-06-22

不错

虞飞

2018-06-20

copyonwriteArrayList循环插入大量数据时（比如100万个）效率很低，因为它形成了100万次快照，那从理论上来说，有没有可能形成一次快照插100万条数据呢？

扫地僧的功夫梦

2018-06-20

谢谢老师的回复，还是notify()/notifyAll()问题，我想说的是被唤醒的线程再重新获取锁之前应该是阻塞状态吧。

洛措

2018-06-20

在写爬虫时，使用过 Semaphore，来控制最多爬同一个域名下的 url 数量。

Jerry很恨

2018-06-20

对于Java 并发包提供了哪些并发工具类，我是这么理解的：  
1. 执行任务，需要对应的执行框架（Executors）；

2. 多个任务被同时执行时，需要协调，这就需要Lock、闭锁、栅栏、信号量、阻塞队列；  
3. Java程序中充满了对象，在并发场景中当然避免不了遇到同种类型的N个对象，而对象需要被存储，这需要高效的线程安全的容器类

Jerry很恨

2018-06-20

客户端开发中遇到的并发场景会不会有点少？如果对并发领域有浓厚的兴趣，可以尝试超哪个方向转型（又或许不需要转型，只需要找一个方向来进行实践）？

老师课后留的问题，我在工作中没有遇到过。我在JAVA并发编程实践中了解到的一种场景是：需要测试N个线程并发执行某个任务时需要的时间。









第20讲 | 并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别?

2018-06-21 杨晓峰



第20讲 | 并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别?  
杨晓峰  
- 01:45 / 08:36

在上一讲中，我分析了Java并发包中的部分内容，今天我来介绍一下线程安全队列。Java标准库提供了非常多的线程安全队列，很容易混淆。

今天我要问你的是，[并发包中的ConcurrentLinkedQueue和LinkedBlockingQueue有什么区别？](#)

#### 典型回答

有时候我们把并发包下面的所有容器都习惯叫作并发容器，但是严格来讲，类似ConcurrentLinkedQueue这种“Concurrent\*\*\*”容器，才是真正代表并发。

关于问题中它们的区别：

- Concurrent类型基于lock-free，在常见的多线程访问场景，一般可以提供较高吞吐量。
- 而LinkedBlockingQueue内部则是基于锁，并提供了BlockingQueue的等待性方法。

不知道你有没有注意到，java.util.concurrent包提供的容器（Queue、List、Set）、Map，从命名上可以大概区分为Concurrent、CopyOnWrite和Blocking\*等三类，同样是线程安全容器，可以简单认为：

- Concurrent类型没有类似CopyOnWrite之类容器相对较重的修改开销。
- 但是，凡事都是有代价的，Concurrent往往提供了较低的遍历一致性。你可以这样理解所谓的弱一致性，例如，当利用迭代器遍历时，如果容器发生修改，迭代器仍然可以继续进行遍历。
- 与弱一致性对应的，就是我介绍过的同步容器常见的行为“fast-fail”，也就是检测到容器在遍历过程中发生了修改，则抛出ConcurrentModificationException，不再继续遍历。
- 弱一致性的另外一个体现是，size等操作准确性是有限的，未必是100%准确。
- 与此同时，读取的性能具有一定的不确定性。

#### 考点分析

今天的问题是又一个引子，考察你是否了解并发包内部不同容器实现的设计目的和实现区别。

队列是非常重要的数据结构，我们日常开发中很多线程间数据传递都要依赖于它，Executor框架提供的各种线程池，同样无法离开队列。面试官可以从不同角度考察，比如：

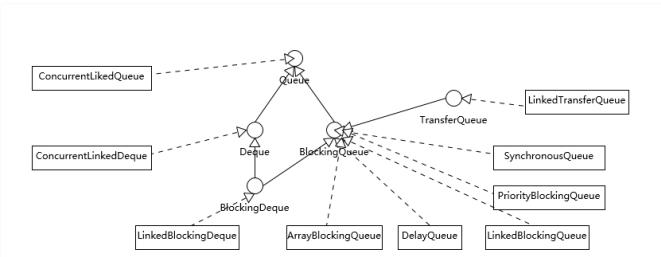
- 哪些队列是有界的，哪些是无界的？（很多同学反馈了这个问题）
- 针对特定场景需求，如何选择合适的队列实现？
- 从源码的角度，常见的线程安全队列是如何实现的，并进行了哪些改进以提高性能表现？

为了能更好地理解这一讲，你需要掌握一些基本的队列本身和数据结构方面知识，如果这方面知识比较薄弱，《数据结构与算法分析》是一本比较全面的参考书，专栏还是尽量专注于Java领域的特性。

#### 知识扩展

线程安全队列一览

我在[专栏第8讲](#)中介绍过，常见的集合中如LinkedList是个Deque，只不过不是线程安全的。下面这张图是Java并发类库提供的各种各样的线程安全队列实现，注意，图中并未将非线程安全部分包含进来。



我们可以从不同的角度进行分类。从基本的数据结构的角度分析，有两个特别的[Deque](#)实现，`ConcurrentLinkedDeque`和`LinkedBlockingDeque`。[Deque](#)的侧重点是支持对队列头尾都进行插入和删除，所以提供了特定的方法，如：

- 尾部插入时需要的[addLast\(e\)](#)、[offerLast\(e\)](#)。
- 尾部删除所需要的[removeLast\(\)](#)、[pollLast\(\)](#)。

从上面这些角度，能够理解`ConcurrentLinkedDeque`和`LinkedBlockingQueue`的主要功能区别，也就足够日常开发的需要了。但是如果我们深入一些，通常会更加关注下面这些方面。

从行为特征来看，绝大部分Queue都是实现了`BlockingQueue`接口。在常规队列操作基础上，`Blocking`意味着其提供了特定的等待性操作，获取时（take）等待元素进队，或者插入时（put）等待队列出现空位。

```

/**
 * 获取并移除队列头结点。如果必要，其会等待直到队列出现元素
 *
 */
E take() throws InterruptedException;

/**
 * 插入元素。如果队列已满，则等待直到队列出现空闲空间
 *
 */
void put(E e) throws InterruptedException;
  
```

另一个`BlockingQueue`经常被考察的点，就是是否有界（Bounded、Unbounded），这一点也往往会影响我们在应用开发中的选择，我这里简单总结一下。

- `ArrayBlockingQueue`是最典型的有界队列，其内部以final的数组保存数据，数组的大小就决定了队列的边界，所以我们在创建`ArrayBlockingQueue`时，都要指定容量，如

```
public ArrayBlockingQueue(int capacity, boolean fair)
```

- `LinkedBlockingQueue`，容易被误解为无边界，但其实其行为和内部代码都是基于有界的逻辑实现的，只不过如果我们没有在创建队列时就指定容量，那么其容量限制就自动被设置为`Integer.MAX_VALUE`，成为了无界队列。
- `SynchronousQueue`，这是一个非常奇葩的队列实现，每个删除操作都要等待插入操作，反之每个插入操作也要等待删除动作。那么这个队列的容量是多少呢？是1吗？其实不是的，其内部容量是0。
- `PriorityBlockingQueue`是无边界的优先队列，虽然严格意义上讲，其大小总归是要受系统资源影响。
- `DelayedQueue`和`LinkedTransferQueue`同样是无边界的队列。对于无边界的队列，有一个自然的结果，就是`put`操作永远也不会发生其他`BlockingQueue`的那种等待情况。

如果我们分析不同队列的底层实现，`BlockingQueue`基本都是基于锁实现，一起来看看典型的`LinkedBlockingQueue`。

```

/** Lock held by take, poll, etc */
private final ReentrantLock takeLock = new ReentrantLock();

/** Wait queue for waiting takes */
private final Condition notEmpty = takeLock.newCondition();

/** Lock held by put, offer, etc */
private final ReentrantLock putLock = new ReentrantLock();

/** Wait queue for waiting puts */
private final Condition notFull = putLock.newCondition();
  
```

我在介绍`ReentrantLock`的条件变量用法的时候分析过`ArrayBlockingQueue`，不知道你有没有注意到，其条件变量与`LinkedBlockingQueue`版本的实现是有区别的。`notEmpty`、`notFull`都是同一个再入锁的条件变量，而`LinkedBlockingQueue`则改进了锁操作的粒度，头、尾操作使用不同的锁，所以在通用场景下，它的吞吐量相对要更好一些。

下面的`take`方法与`ArrayBlockingQueue`中的实现，也是有不同的，由于其内部结构是链表，需要自己维护元素数量值，请参考下面的代码。

```
public E take() throws InterruptedException {
    final E x;
```

```

final int c;
final AtomicInteger count = this.count;
final ReentrantLock takeLock = this.takeLock;
takeLock.lockInterruptibly();
try {
    while (count.get() == 0) {
        notEmpty.await();
    }
    x = dequeue();
    c = count.getAndDecrement();
    if (c > 1)
        notEmpty.signal();
} finally {
    takeLock.unlock();
}
if (c == capacity)
    signalNotFull();
return x;
}

```

类似ConcurrentLinkedQueue等，则是基于CAS的无锁技术，不需要在每个操作时使用锁，所以扩展性表现要更加优异。

相对比较另类的SynchronousQueue，在Java 6中，其实现发生了非常大的变化，利用CAS替换掉了原本基于锁的逻辑，同步开销比较小。它是Executors.newCachedThreadPool()的默认队列。

#### 队列使用场景与典型用例

在实际开发中，我提到过Queue被广泛使用在生产者-消费者场景，比如利用BlockingQueue来实现，由于其提供的等待机制，我们可以少操心很多协调工作，你可以参考下面样例代码：

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class ConsumerProducer {
    public static final String EXIT_MSG = "Good bye!";
    public static void main(String[] args) {
        // 使用较小队列，以更好地在输出中展示其影响
        BlockingQueue<String> queue = new ArrayBlockingQueue<String>(3);
        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);
        new Thread(producer).start();
        new Thread(consumer).start();
    }

    static class Producer implements Runnable {
        private BlockingQueue<String> queue;
        public Producer(BlockingQueue<String> q) {
            this.queue = q;
        }

        @Override
        public void run() {
            for (int i = 0; i < 20; i++) {
                try{
                    Thread.sleep(5L);
                    String msg = "Message" + i;
                    System.out.println("Produced new item: " + msg);
                    queue.put(msg);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            try {
                System.out.println("Time to say good bye!");
                queue.put(EXIT_MSG);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    static class Consumer implements Runnable{
        private BlockingQueue<String> queue;
        public Consumer(BlockingQueue<String> q){
            this.queue=q;
        }
    }
}

```

```

@Override
public void run() {
    try{
        String msg;
        while(EXIT_MSG.equalsIgnoreCase( msg = queue.take())){
            System.out.println("Consumed item: " + msg);
            Thread.sleep(10L);
        }
        System.out.println("Got exit message, bye!");
    }catch(InterruptedException e) {
        e.printStackTrace();
    }
}
}
}

```

上面是一个典型的生产者-消费者样例，如果使用非Blocking的队列，那么我们就要自己去实现轮询、条件判断（如检查poll返回值是否null）等逻辑，如果没有特别的场景要求，Blocking实现起来代码更加简单、直观。

前面介绍了各种队列实现，在日常的应用开发中，如何进行选择呢？

以LinkedBlockingQueue、ArrayBlockingQueue和SynchronousQueue为例，我们一起来分析一下，根据需求可以从很多方面考量：

- 考虑应用场景中对队列边界的要求。ArrayBlockingQueue是有明确的容量限制的，而LinkedBlockingQueue则取决于我们是否在创建时指定，SynchronousQueue则干脆不能缓存任何元素。
- 从空间利用角度，数组结构的ArrayBlockingQueue要比LinkedBlockingQueue紧凑，因为其不需要创建所谓节点，但是其初始分配阶段就需要一段连续的空间，所以初始内存需求更大。
- 通用场景中，LinkedBlockingQueue的吞吐量一般优于ArrayBlockingQueue，因为它实现了更加细粒度的锁操作。
- ArrayBlockingQueue实现比较简单，性能更好预测，属于表现稳定的“选手”。
- 如果我们需要实现的是两个线程之间接力性（handoff）的场景，按照[专栏上一讲](#)的例子，你可能会选择CountDownLatch，但是SynchronousQueue也是完美符合这种场景的，而且线程间协调和数据传输统一起来，代码更加规范。
- 可能令人意外的是，很多时候SynchronousQueue的性能表现，往往大大超过其他实现，尤其是在队列元素较小的场景。

今天我分析了Java中让人眼花缭乱的各种线程安全队列，试图从几个角度，让每个队列的特点更加明确，进而希望减少你在日常工作中使用时的困扰。

#### -课-练

关于今天我们讨论的题目你做到心中有数了吗？今天的内容侧重于Java自身的角度，面试官也可能从算法的角度来考察，所以今天留给你的思考题是：指定某种结构，比如栈，用它实现一个BlockingQueue，实现思路是怎样的呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



sunlight001

2018-06-21

这个看着很吃力啊，都没接触过◆◆

无能可称

2018-06-27

@Jerry银银。用两个栈可以实现fifo的队列

crazyone

2018-06-22

从上面这些角度，能够理解 ConcurrentLinkedDeque 和 LinkedBlockingQueue 的主要功能区别。这段应该是“ConcurrentLinkedDeque 和 LinkedBlockingQueue 的主要功能区别”

灰飞灰猪不会灰飞.烟灭

老师 线程池中如果线程已经运行结束则删除该线程。如何判断线程已经运行结束了呢？源码中我看按照线程的状态，我不清楚这些状态值哪来的。java代码有判断线程状态的方法吗？谢谢老师

作者回复

所谓结束是指terminated? 正常的线程池移除工作线程，要么线程意外退出，比如任务抛异常，要么线程闲置，又规定了闲置时间；线程池中线程是把额外封装的，本来下章写了，内容篇幅超移到后面了，慢慢来；有，建议学会看文档，自己找答案

石头狮子

实现课后题过程中把握以下几个维度，

1. 数据操作的颗粒度。
2. 计数、遍历方式。
3. 数据结构空、满时线程的等待方式，有锁或无锁方式。
4. 使用离散还是连续的存储结构。

石头狮子

实现课后题过程中把握以下几个维度，

1. 数据操作的颗粒度。
2. 计数、遍历方式。
3. 数据结构空、满时线程的等待方式，有锁或无锁方式。
4. 使用离散还是连续的存储结构。

汉影

用栈实现BlockingQueue，我的理解是：栈是LIFO，BlockingQueue是FIFO，因此需要两个栈。take时先把栈A全部入栈到栈B，然后栈B出栈得到目标元素；put时把栈B全部入栈到栈A，然后栈A再入栈目标元素。相当于倒序一下。

不知道理解对不对，请老师指出。

爱新觉罗老流氓

杨老师，“与弱一致性对应的，就是我介绍过的同步容器常见的行为“fast-fail”，也就是检测到容器在遍历过程中发生了修改，则抛出 ConcurrentModificationException，不再继续遍历。”

这一段落里，快速失败的英文在doc上是“fail-fast”，在ArrayList源码中文档可以搜到。

还有，同步容器不应该是“fail-safe”吗？

作者回复

谢谢指出，我查查是不是我记反了

Invoker.C

求老师解答一个困扰已久的问题，就是初始化arrayblockingqueue的时候，capacity的大小如何评估和设置？望解答

作者回复

不清楚你的硬件、业务特点，一个大概原则是尽量让进和出的速率一致，不然出慢，进就block，反过来也不好；实际操作上，你试试找时机检查remainCapacity，就可以判断进出速率的对比

Jerry银银

用栈来实现BlockingQueue，换句话是说，用先进后出的数据结构来实现先进先出的数据结构，怎么感觉听起来不那么对劲呢？请指点

hansc

老师，copyonwrite 涉及到用户太切换到内核态吗？

猕猴桃 盛哥

```
{
  "test": [
    [
      89,
      90,
      [
        [
          1093,
          709
        ],
        [
          1056,
          709
        ]
      ]
    ]
  ]
}
```

测试题：这个json用java对象怎么表示？

夏洛克的救赎

老师你好，问个题外问题，在jdk10源码 string类中，成员变量coder起到什么作用？如何理解？

作者回复

编码，区分拉丁和非拉丁语系







第21讲 | Java并发类库提供的线程池有哪几种？分别有什么特点？

2018-06-23 杨晓峰



第21讲 | Java并发类库提供的线程池有哪几种？分别有什么特点？  
杨晓峰  
0:00 / 12:30

我在[专栏17讲](#)中介绍过线程是不能够重复启动的，创建或销毁线程存在一定的开销，所以利用线程池技术来提高系统资源利用效率，并简化线程管理，已经是非常成熟的选择。

今天我要问你的是，Java并发类库提供的线程池有哪几种？分别有什么特点？

#### 典型回答

通常开发者都是利用Executors提供的通用线程池创建方法，去创建不同配置的线程池，主要区别在于不同的ExecutorService类型或者不同的初始参数。

Executors目前提供了5种不同的线程池创建配置：

- `newCachedThreadPool()`，它是一种用来处理大量短时间工作任务的线程池，具有几个鲜明特点：它会试图缓存线程并重用，当无缓存线程可用时，就会创建新的工作线程；如果线程闲置的时间超过60秒，则被终止并移出缓存；长时间闲置时，这种线程池，不会消耗什么资源。其内部使用`SynchronousQueue`作为工作队列。
- `newFixedThreadPool(int nThreads)`，重用指定数目(`nThreads`)的线程，其背后使用的是无界的工作队列，任何时候最多有`nThreads`个工作线程是活动的。这意味着，如果任务数量超过了活动队列数目，将在工作队列中等待空闲线程出现；如果有工作线程退出，将会有新的工作线程被创建，以补足指定的数目`nThreads`。
- `newSingleThreadExecutor()`，它的特点在于工作线程数目被限制为1，操作一个无界的工作队列，所以它保证了所有任务的都是被顺序执行，最多会有一个任务处于活动状态，并且不允许使用者改动线程池实例，因此可以避免其改变线程数目。
- `newSingleThreadScheduledExecutor()`和`newScheduledThreadPool(int corePoolSize)`，创建的是个`ScheduledExecutorService`，可以进行定时或周期性的工作调度，区别在于单一工作线程还是多个工作线程。
- `newWorkStealingPool(int parallelism)`，这是一个经常被人忽略的线程池，Java 8才加入这个创建方法，其内部会构建`ForkJoinPool`，利用`Work-Stealing`算法，并行地处理任务，不保证处理顺序。

#### 考点分析

Java并发包中的`Executor`框架无疑是并发编程中的重点，今天的题目考察的是对几种标准线程池的了解，我提供的是一个针对最常见的应用方式的回答。

在大多数应用场景下，使用`Executors`提供的5个静态工厂方法就足够了，但是仍然可能需要直接利用`ThreadPoolExecutor`等构造函数创建，这就要求你对线程构造方式有进一步的了解，你需要明白线程池的设计和结构。

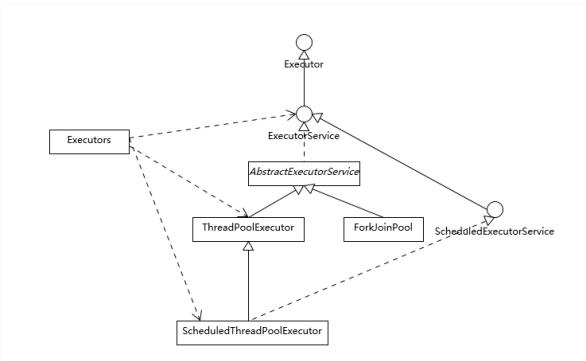
另外，线程池这个定义就是个容易让人误解的术语，因为`ExecutorService`除了通常意义上“池”的功能，还提供了更全面的线程管理、任务提交等方法。

`Executor`框架可不仅仅是线程池，我觉得至少下面几点值得深入学习：

- 掌握`Executor`框架的主要内容，至少要了解组成与职责，掌握基本开发用例中的使用。
- 对线程池和相关并发工具类型的理解，甚至是源码层面的掌握。
- 实践中有哪些常见问题，基本的诊断思路是怎样的。
- 如何根据自身应用特点合理使用线程池。

#### 知识扩展

首先，我们来看看`Executor`框架的基本组成，请参考下面的类图。



我们从整体上把握一下各个类型的主要设计目的：

- Executor是一个基础的接口，其初衷是将任务提交和任务执行细节解耦，这一点可以体会其定义的唯一方法。

```
void execute(Runnable command);
```

Executor的设计是源于Java早期线程API使用的教训，开发者在实现应用逻辑时，被太多线程创建、调度等不相关细节所打扰。就像我们进行HTTP通信，如果还需要自己操作TCP握手，开发效率低下，质量也难以保证。

- ExecutorService则更加完善，不仅提供service的管理功能，比如shutdown等方法，也提供了更加全面的提交任务机制，如返回Future而不是void的submit方法。

```
<T> Future<T> submit(Callable<T> task);
```

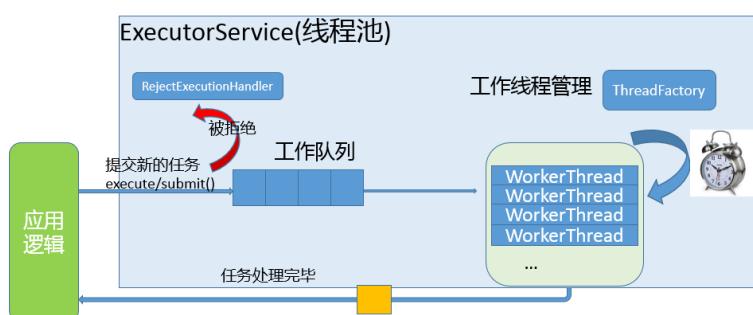
注意，这个例子输入的可是Callable，它解决了Runnable无法返回结果的困扰。

• Java标准类库提供了几种基础实现，比如ThreadPoolExecutor、ScheduledThreadPoolExecutor、ForkJoinPool，这些线程池的设计特点在于其高度的可调节性和灵活性，以尽量满足复杂多变的实际应用场景，我会进一步分析其构建部分的源码，剖析这种灵活性的源头。

- Executors则从简化使用的角度，为我们提供了各种方便的静态工厂方法。

下面我就从源码角度，分析线程池的设计与实现，我将主要围绕最基础的ThreadPoolExecutor源码。ScheduledThreadPoolExecutor是ThreadPoolExecutor的扩展，主要是增加了调度逻辑，如想深入了解，你可以参考相关[教程](#)。而ForkJoinPool则是为ForkJoinTask定制的线程池，与通常意义的线程池有所不同。

这部分内容比较晦涩，罗列概念也不利于你去理解，所以我会配合一些示意图来说明。在现实应用中，理解应用与线程池的交互和线程池的内部工作过程，你可以参考下图。



简单理解一下：

- 工作队列负责存储用户提交的各个任务，这个工作队列，可以是容量为0的SynchronousQueue（使用newCachedThreadPool），也可以是像固定大小线程池（newFixedThreadPool）那样使用LinkedBlockingQueue。

```
private final BlockingQueue<Runnable> workQueue;
```

- 内部的“线程池”，这是指保持工作线程的集合，线程池需要在运行过程中管理线程创建、销毁。例如，对于带缓存的线程池，当任务压力较大时，线程池会创建新的工作线程，当业务压力退去，线程池会在闲置一段时间（默认60秒）后结束线程。

```
private final HashSet<Worker> workers = new HashSet<>();
```

线程池的工作线程被抽象为静态内部类Worker，基于[AQS](#)实现。

- ThreadFactory提供上面所需要的创建线程逻辑。
  - 如果任务提交时被拒绝，比如线程池已经处于SHUTDOWN状态，需要为其提供处理逻辑，Java标准库提供了类似[ThreadPoolExecutor.AbortPolicy](#)等默认实现，也可以按照实际需求自定义。
- 从上面的分析，就可以看出线程池的几个基本组成部分，一起都体现在线程池的构造函数中，从字面我们就可以大概猜测到其用意：
- corePoolSize，所谓的核心线程数，可以大致理解为长期驻留的线程数目（除非设置了allowCoreThreadTimeOut）。对于不同的线程池，这个值可能会有很大区别，比如newFixedThreadPool会将其设置为nThreads，而对于newCachedThreadPool则是0。
  - maximumPoolSize，顾名思义，就是线程不够时能够创建的最大线程数。同样进行对比，对于newFixedThreadPool，当然就是nThreads，因为其要求是固定大小，而newCachedThreadPool则是Integer.MAX\_VALUE。
  - keepAliveTime和TimeUnit，这两个参数指定了额外的线程能够闲置多久，显然有些线程池不需要它。
  - workQueue，工作队列，必须是BlockingQueue。

通过配置不同的参数，我们就可以创建出行为大相径庭的线程池，这就是线程池高度灵活性的基础。

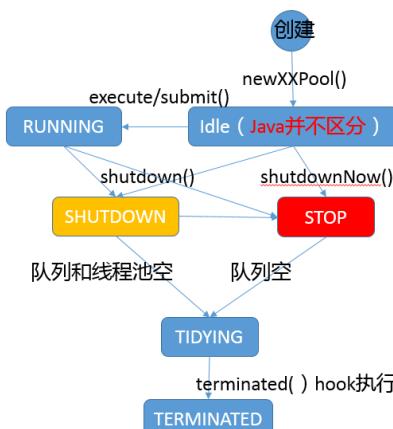
```
public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

进一步分析，线程池既然有生命周期，它的状态是如何表征的呢？

这里有一个非常有意思的设计，ctl变量被赋予了双重角色，通过高低位的不同，既表示线程池状态，又表示工作线程数目，这是一个典型的高效优化。试想，实际系统中，虽然我们可以指定线程极限为Integer.MAX\_VALUE，但是因为资源限制，这只是个理论值，所以完全可以将空闲位赋予其他意义。

```
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
// 真正决定了工作线程数的理论上限
private static final int COUNT_BITS = Integer.SIZE - 3;
private static final int COUNT_MASK = (1 << COUNT_BITS) - 1;
// 线程池状态，存储在数字的高位
private static final int RUNNING = -1 << COUNT_BITS;
-
// Packing and unpacking ctl
private static int runStateOf(int c) { return c & ~COUNT_MASK; }
private static int workerCountOf(int c) { return c & COUNT_MASK; }
private static int ctlOf(int rs, int wc) { return rs | wc; }
```

为了让你能对线程生命周期有个更加清晰的印象，我这里画了一个简单的状态流转图，对线程池的可能状态和其内部方法之间进行了对应，如果有不理解的方法，请参考Javadoc。注意，实际Java代码中并不存在所谓Idle状态，我添加它仅仅是便于理解。



前面都是对线程池属性和构建等方面分析，下面我选择典型的execute方法，来看看其是如何工作的，具体逻辑请参考我添加的注释，配合代码更加容易理解。

```

public void execute(Runnable command) {
    ...
    int c = ctl.get();
    // 检查工作线程数目，低于corePoolSize则添加Worker
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // isRunning就是检查线程池是否被shutdown
    // 工作队列可能是有界的，offer是比较友好的入队方式
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        ...
        // 再次进行的特性检查
        if (!isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    // 尝试添加一个worker, 如果失败以为着已经饱和或者被shutdown了
    else if (!addWorker(command, false))
        reject(command);
}

```

### 线程池实践

线程池虽然为提供了非常强大、方便的功能，但是也不是很弹，使用不当同样会导致问题。我这里介绍些典型情况，经过前面的分析，很多方面可以自然的推导出来。

- 避免任务堆积。前面我说过newFixedThreadPool是创建指定数目的线程，但是其工作队列是无界的，如果工作线程数目太少，导致处理跟不上入队的速度，这就很有可能占用大量的系统内存，甚至是出现OOM。诊断时，你可以使用Jmap之类的工具，查看是否有大量的任务对象入队。
- 避免过度扩展线程。我们通常在处理大量短任务时，使用缓存的线程池，比如在最新的HTTP/2 client API中，目前的默认实现就是如此。我们在创建线程池的时候，并不能准确预计任务压力有多大，数据特征是什么样子（大部分请求是1K、100K还是1M以上？），所以很难明确设定一个线程数目。
- 另外，如果线程数目不断增长（可以使用jstack等工具检查），也需要警惕另外一种可能性，就是线程泄漏，这种情况往往是因为任务逻辑有问题，导致工作线程迟迟不能被释放。建议你排查下线程栈，很有可能多个线程都是卡在近似的代码处。
- 避免死锁等同步问题。对于死锁的场景和排查，你可以复习[专栏第18讲](#)。
- 尽量避免在使用线程池时操作ThreadLocal，同样是[专栏第17讲](#)已经分析过的，通过今天的线程池学习，应该更能理解其原因，工作线程的生命周期通常都会超过任务的生命周期。

### 线程池大小的选择策略

上面我已经介绍过，线程池大小不合适，太多会太少，都会导致麻烦，所以我们需要去考虑一个合适的线程池大小。虽然不能完全确定，但是有一些相对普适的规则和思路。

- 如果我们的任务主要是进行计算，那么就意味着CPU的处理能力是稀缺的资源，我们能够通过大量增加线程数提高计算能力吗？往往是不能的，如果线程太多，反倒可能导致大量的上下文切换开销。所以，这种情况下，通常建议按照CPU核的数目N或者N+1。
- 如果是需要较多等待的任务，例如I/O操作比较多，可以参考Brain Goetz推荐的计算方法：

线程数 = CPU核数 × (1 + 平均等待时间/平均工作时间)

这些时间并不能精准预计，需要根据采样或者概要分析等方式进行计算，然后在实际中验证和调整。

- 上面是仅仅考虑了CPU等限制，实际还可能受各种系统资源限制影响，例如我最近就在Mac OS X上遇到了大负载时[ephemeral端口受限](#)的情况。当然，我是通过扩大可用端口范围解决的，如果我们不能调整资源的容量，那么就只能限制工作线程的数目了。这里的资源可以是文件句柄、内存等。

另外，在实际工作中，不要把解决问题的思路全部指望到调整线程池上，很多时候架构上的改变更能解决问题，比如利用背压机制的[Reactive Stream](#)、合理的拆分等。

今天，我从Java创建的几种线程池开始，对Executor框架的主要组成、线程池结构与生命周期等方面进行了讲解和分析，希望对你有所帮助。

### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是从逻辑上理解，线程池创建和生命周期。请谈一谈，如果利用newSingleThreadExecutor()创建一个线程池，corePoolSize、maxPoolSize等都是什么数值？ThreadFactory可能在线程池生命周期中被使用多少次？怎么验证自己的判断？

请你在留言区写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“[请朋友读](#)”，把今天的题目分享给好友，或许你能帮到他。



曹铮

2018-06-23

疑问，为什么当初sun的线程池模式要设计成队列满了才能创建非核心线程？类比其他类似池的功能实现，很多都是设置最小数最大数，达到最大数才向等待队列里加入，比如有的连接池实现。

作者回复

Doug Lea这个实现基本是工业标准了，除非特定场景需求

2018-06-26

三木子

2018-06-23

我觉得还有一点很重要，就是放在线程池中的线程要捕获异常，如果直接抛出异常，每次都会创建线程，也就等于线程池没有发挥作用，如果大并发下一直创建线程可能会导致JVM挂掉。最近遇到的一个坑

作者回复

任务出异常是要避免

2018-06-26

沈琦斌

2018-06-28

老师，我想问的是cache的线程池大小是1，每次还要新创建，那和我自己创建而不用线程池有什么区别？

作者回复

你是说cachedthreadpool？那个大小是浮动的，不是1；如果说single，executorservice毕竟还提供了工作队列，生命周期管理，工作线程维护等很多事，还是要高效

2018-06-29

I am a psycho

2018-06-23

通过看源码可以得知，core和max都是1，而且通过FinalizableDelegatedExecutorService进行了包装，保证线程池无法修改。同时shutdown方法通过调用interruptIdleWorkers方法，去停掉所有工作的线程，而shutdownNow方法是直接粗暴的停掉所有线程。无论是shutdown还是shutdownNow都不会进行等待，都会直接将线程池状态设置成shutdown或者stop，如果需要等待，需要调用awaitTermination方法。查了一下threadfactory的使用，只找到了在worker创建的时候，用来初始化了线程。

作者回复

不错，很棒的总结：

我问threadFactory次数，其实是问worker都在什么情况下会被创建，比如，比较特别的，任务抛异常时；随便自定义一个threadfactory，模拟提交任务就能体会到

2018-06-26

菜鸡互啄

2018-06-24

特别想咨询老师一个问题。我目前工作两年，平时公司业务很繁忙，加班特别多，自己的时间很少，所以觉得迷茫，不知道现在这个阶段应该利用有限的时间着重去学习哪些方面的知识。是应该对基础知识继续深究下去还是应该去看一些框架源码或架构方面的知识。

欣

2018-07-04

杨老师，我照着文章翻看源码，下面那块是不是不太对？

Executors 目前提供了 5 种不同的线程池创建配置：

newSingleThreadExecutor，它创建的是个 FinalizableDelegatedExecutorService

newSingleThreadScheduledExecutor 创建的是 ScheduledThreadPoolExecutor

作者回复

2018-07-04

谢谢指出

Cc养鱼人

2018-06-28

听了一段时间课程，质量很高。我的需求是android JavaVM

作者回复

2018-06-29

android我并没有特别的经验，尽管很多方面是通用的

One day

2018-06-25

老师提的问题，能不能给个答案

作者回复

2018-06-26

已回复

王磊

core和max应该都是1。验证的方法是自己写一个Threadlocal，里面有相应创建线程的日志，然后把它传入创建线程池。

作者回复

2018-06-25

core和Max源码或者逻辑分析都很清楚；而创建线程次数理论上是不确定的，比如任务执行中抛异常，就要重新创建worker

灰飞灰猪不会灰飞，烟灭

老师 放入队列中的线程是直接调用start方法还是把队列中的线程放入线程工厂，让线程工厂执行？

另外，怎么判断一个线程是否执行完成呢？（只有执行完成才返回结果）谢谢老师

作者回复

2018-06-26

工厂是创建线程，执行完成通常是说任务，而不是线程，任务才是我们关心的；可以用Future

2018-06-26







第22讲 | AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？

2018-06-26 杨晓峰



第22讲 | AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？

杨晓峰 - 00:26 / 11:03

在今天这一讲中，我来分析一下并发包内部的组成，一起来看看各种同步结构、线程池等，是基于什么原理来设计和实现的。

今天我要问你的是，[AtomicInteger底层实现原理是什么？如何在自己的产品代码中应用CAS操作？](#)

典型回答

AtomicInteger是对int类型的一个封装，提供原生性的访问和更新操作，其原生性操作的实现是基于CAS ([compare-and-swap](#)) 技术。

所谓CAS，表示的是一些列操作的集合，获取当前数值，进行一些运算，利用CAS指令试图进行更新。如果当前数值未变，代表没有其他线程进行并发修改，则成功更新。否则，可能出现在不同的选择，要么进行重试，要么就返回一个成功或者失败的结果。

从AtomicInteger的内部属性可以看出，它依赖于Unsafe提供的一些底层能力，进行底层操作；以volatile的value字段，记录数值，以保证可见性。

```
private static final jdk.internal.misc.Unsafe U = jdk.internal.misc.Unsafe.getUnsafe();
private static final long VALUE = U.objectFieldOffset(AtomicInteger.class, "value");
private volatile int value;
```

具体的原子操作细节，可以参考任意一个原子更新方法，比如下面的getAndIncrement。

Unsafe会利用value字段的内存地址偏移，直接完成操作。

```
public final int getAndIncrement() {
    return U.getAndAddInt(this, VALUE, 1);
}
```

因为getAndIncrement需要返回数值，所以需要添加失败重试逻辑。

```
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}
```

而类似compareAndSet这种返回boolean类型的函数，因为其返回值表现的就是成功与否，所以不需要重试。

```
public final boolean compareAndSet(int expectedValue, int newValue)
```

CAS是Java并发中所谓lock-free机制的基础。

考点分析

Java

CAS

Java

今天的问题有点偏向于并发机制的底层了，虽然我们在开发中未必会涉及的实现层面，但是理解其机制，掌握如何在中运用该技术，还是十分有必要的，尤其是这也是个并发编程的面试热点。

有的同学反馈面试官会问CAS更加底层是如何实现的，这依赖于CPU提供的特定指令，具体根据体系结构的不同还存在着明显区别。比如，x86 CPU提供`cmpxchg`指令；而在精简指令集的体系架构中，则通常是靠一对儿指令（如“load and reserve”和“store conditional”）实现的，在大多数处理器上CAS都是个非常轻量级的操作，这也是其优势所在。

大部分情况下，掌握到这个程度也就够用了，我认为没有必要让每个Java工程师都去了解到指令级别，我们进行抽象、分工就是为了让不同层面的开发者在开发中，可以尽量屏蔽不相关的细节。

如果我作为面试官，很有可能深入考察这些方向：

- 在什么场景下，可以采用CAS技术，调用Unsafe毕竟不是大多数场景的最好选择，有没有更加推荐的方式呢？毕竟我们掌握一个技术，cool不是目的，更不是为了应付面试，我们还是希望能在实际产品中有价值。
- 对ReentrantLock、CyclicBarrier等并发结构底层的理解。

#### 知识扩展

关于CAS的使用，你可以设想这样一个场景：在数据库产品中，为保证索引的一致性，一个常见的选择是，保证只有一个线程能够排他性地修改一个索引分区，如何在数据库抽象层面实现呢？

可以考虑为索引分区对象添加一个逻辑上的锁，例如，以当前独占的线程ID作为锁的数值，然后通过原子操作设置lock数值，来实现加锁和释放锁，伪代码如下：

```
public class AtomicBTreePartition {
    private volatile long lock;
    public void acquireLock(){}
    public void releaseLock(){}
}
```

那么在Java代码中，我们怎么实现锁操作呢？Unsafe似乎不是个好的选择，例如，我就注意到类似Cassandra等产品，因为Java 9中移除了Unsafe.moniterEnter() / moniterExit()，导致无法平滑升级到新的JDK版本。目前Java提供了两种公共API，可以实现这种CAS操作，比如使用java.util.concurrent.atomic.AtomicLongFieldUpdater，它是基于反射机制创建，我们需要保证类型和字段名称正确。

```
private static final AtomicLongFieldUpdater<AtomicBTreePartition> lockFieldUpdater =
    AtomicLongFieldUpdater.newUpdater(AtomicBTreePartition.class, "lock");

private void acquireLock(){
    long t = Thread.currentThread().getId();
    while (!lockFieldUpdater.compareAndSet(this, 0L, t)){
        // 等待一会儿，数据库操作可能比较慢
        -
    }
}
```

[Atomic包](#)提供了最常用的原子性数据类型，甚至是引用、数组等相关原子类型和更新操作工具，是很多线程安全程序的首选。

我在专栏第七讲中曾介绍使用原子数据类型和Atomic\*FieldUpdater，创建更加紧凑的计数器实现，以替代AtomicLong。优化永远是针对特定需求、特定目的，我这里的侧重点是介绍可能的思路，具体还是要看需求。如果仅仅创建一两个对象，其实完全没有必要进行前面的优化。但是如果对象成千上万或者更多，就要考虑紧凑性的影响了。而atomic包提供的[LongAdder](#)，在高度竞争环境下，可能就是比AtomicLong更佳的选择，尽管它的本质是空间换时间。

回归正题，如果是Java 9以后，我们完全可以采用另外一种方式实现，也就是Variable Handle API，这是源自于[JEP 193](#)，提供了各种粒度的原子或者有序性的操作等。我将前面的代码修改为如下实现：

```
private static final VarHandle HANDLE = MethodHandles.lookup().findStaticVarHandle(
    AtomicBTreePartition.class, "lock");

private void acquireLock(){
    long t = Thread.currentThread().getId();
    while (!HANDLE.compareAndSet(this, 0L, t)){
        // 等待一会儿，数据库操作可能比较慢
        -
    }
}
```

过程非常直观，首先，获取相应的变量句柄，然后直接调用其提供的CAS方法。

一般来说，我们进行的类似CAS操作，可以并且推荐使用Variable Handle API去实现，其提供了精细化的公共底层API。我这里强调公共，是因为其API不会像内部API那样，发生不可预测的修改，这一点提供了对于未来产品维护和升级的基础保障，坦白说，很多额外工作量，都是源于我们使用了Hack而非Solution的方式解决问题。

CAS也并不是没有副作用，试想，其常用的失败重试机制，隐含着一个假设，即竞争情况是短暂的。大多数应用场景中，确实大部分重试只会发生一次就获得了成功，但是总是有意外情况，所以在有需要的时候，还是要考虑限制自旋的次数，以免过度消耗CPU。

另外一个就是著名的ABA问题，这是通常只在lock-free算法下暴露的问题。我前面说过CAS是在更新时比较前值，如果对方只是恰好相同，例如期间发生了A -> B -> A的更新，仅仅判断数值是A，可能会导致不合理的修改操作。针对这种情况，Java提供了AtomicStampedReference工具类，通过为引用建立类似版本号（stamp）的方式，来保证CAS的正确性，具体用法请参考这里的[链接](#)。

前面介绍了CAS的场景与实现，幸运的是，大多数情况下，Java开发者并不需要直接利用CAS代码去实现线程安全容器等，更多是通过并发包等间接享受到lock-free机制在扩展性上的好处。

下面我来介绍一下AbstractQueuedSynchronizer（AQS），其是Java并发包中，实现各种同步结构和部分其他组成单元（如线程池中的Worker）的基础。

学习，如果上来就去看它的一系列方法（下图所示），很有可能把自己看晕，这种似懂非懂的状态也没有太大的实践意义。

我建议的思路是，尽量简化一下，理解为什么需要AQS，如何使用AQS，至少要做什么，再进一步结合JDK源代码中的实践，理解AQS的原理与应用。

**Doug Lea**曾经介绍过AQS的设计初衷。从原理上，一种同步结构往往是可以利用其他的结构实现的。例如我在专栏第19讲中提到过可以使用Semaphore实现互斥锁。但是，对某种同步结构的倾向，会导致复杂、晦涩的实现逻辑，所以，他选择了将基础的同步相关操作抽象在AbstractQueuedSynchronizer中，利用AQS为我们构建同步结构提供了范本。

AQS内部数据和方法，可以简单拆分为：

- 一个volatile的整数成员表征状态，同时提供了setState和getState方法

```
private volatile int state;
```

- 一个先入先出（FIFO）的等待线程队列，以实现多线程间竞争和等待，这是AQS机制的核心之一。

- 各种基于CAS的基础操作方法，以及各种期望具体同步结构去实现的acquire/release方法。

利用AQS实现一个同步结构，至少要实现两个基本类型的方法，分别是acquire操作，获取资源的独占权；还有就是release操作，释放对某个资源的独占。

以ReentrantLock为例，它内部通过扩展AQS实现了Sync类型，以AQS的state来反映锁的持有情况。

```
private final Sync sync;
abstract static class Sync extends AbstractQueuedSynchronizer { ... }
```

下面是ReentrantLock对应acquire和release操作，如果是CountDownLatch则可以看作是await()/countDown()，具体实现也有区别。

```
public void lock() {
    sync.acquire(1);
}
public void unlock() {
    sync.release(1);
}
```

排除掉一些细节，整体地分析acquire方法逻辑，其直接实现是在AQS内部，调用了tryAcquire和acquireQueued，这两个需要搞清楚的基本部分。

```
public final void acquire(int arg) {
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

首先，我们来看看tryAcquire。在ReentrantLock中，tryAcquire逻辑实现在NonfairSync和FairSync中，分别提供了进一步的不公平或公平性方法，而AQS内部tryAcquire仅仅是个接近未实现的方法（直接抛异常），这是留给实现者自己定义的操作。

我们可以看到公平性在ReentrantLock构建时如何指定的，具体如下：

```
public ReentrantLock() {
    sync = new NonfairSync(); // 默认是非公平的
}
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

以非公平的tryAcquire为例，其内部实现了如何配合状态与CAS获取锁，注意，对比公平版本的tryAcquire，它在锁无人占有时，并不检查是否有其他等待者，这里体现了非公平的语义。

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState(); // 获取当前AQS内部状态量
    if (c == 0) { // 0表示无人占有，则直接用CAS修改状态位
        if (!compareAndSetState(0, acquires)) { // 不检测排队情况，直接争抢
            setExclusiveOwnerThread(current); // 设置当前线程为独占锁
            return true;
        }
    } else if (current == getExclusiveOwnerThread()) { // 即使状态不是0，也可能当前线程是锁持有者，因为这是再入锁
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

接下来我再来分析acquireQueued，如果前面的tryAcquire失败，代表着锁争抢失败，进入排队竞争阶段。这里就是我们所说的，利用FIFO队列，实现线程间对锁的竞争的部分，算是AQS的核心逻辑。

当前线程会被包装成为一个排他模式的节点（EXCLUSIVE），通过addWaiter方法添加到队列中。acquireQueued的逻辑，简要来说，就是如果当前节点的前面是头节点，则试图获取锁，一切顺利则成为新的头节点；否则，有必要则等待，具体处理逻辑请参考我添加的注释。

```
final boolean acquireQueued(final Node node, int arg) {
    boolean interrupted = false;
    try {
        for (;;) { // 循环
            final Node p = node.predecessor(); // 获取前一个节点
            if (p == head && tryAcquire(arg)) { // 如果前一个节点是头结点，表示当前节点合适去tryAcquire
                setHead(node); // acquire成功，则设置新的头节点
                p.next = null; // 将前面节点对当前节点的引用清空
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node)) // 检查是否失败后需要park
                interrupted |= parkAndCheckInterrupt();
        }
    } catch (Throwable t) {
        cancelAcquire(node); // 出现异常，取消
        if (interrupted)
            selfInterrupt();
        throw t;
    }
}
```

到这里线程试图获取锁的过程基本展现出来了，tryAcquire是按照特定场景需要开发者去实现的部分，而线程间竞争则是AQS通过Waiter队列与acquireQueued提供的，在release方法中，同样会对队列进行对应操作。

今天我介绍了Atomic数据类型的底层技术CAS，并通过实例演示了如何在产品代码中利用CAS，最后介绍了并发包的基础技术AQS，希望对你有所帮助。

### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天布置一个源码阅读作业，AQS中Node的waitStatus有什么作用？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



wenxueliu

2018-06-26

建议：

1. 希望能有堆外内存的主题，范型部分希望能与cpp比较讲解。
2. 一些主题如果已经有公开的比较好的资料，可以提供链接，对重点强调即可。希望能看到更多公开资料所没有的信息，这也是老鸟们付费的初衷。

同意的点赞

Cui

2018-06-26

老师，看了AQS的实现原理后，我再回顾了您之前关于Synchronized的文章，心中有些疑问：

1. synchronized在JVM中是会进行锁升级和降级的，并且是基于CAS来掌握竞争的情况，在竞争不多的情况下利用CAS的轻量级操作来减少开销。
2. 而AQS也是基于CAS操作队列的，位于队列头的节点优先获得锁，其他的节点会被LockSupport.park()起来（这个好像依赖的是操作系统的互斥锁，应该也是个重量级操作）。

我覺得这两种方式都是基于CAS操作的，只是操作的对象不同（一个是Mark Word，一个是队列节点），当竞争较多时，还是不可避免地会使用到操作系统的互斥锁。然而，我再测试这两者的时候，在无竞争的情况下，两者性能相当，但是，当竞争起来后，AQS的性能明显比synchronized要好（测试案例是8个线程并发对一个int递增，每个线程递增1000万次，AQS的耗时大概要少30%），这是为什么呢？

作者回复

2018-06-28

Locksupport的实现据说速度快，我也没具体对比过；不过jdk9里，monitor相关操作也加快了，可以看看jep143

二木◆◆

2018-06-28

一直很好奇，为何CAS指令在发现内容未变的时候就能判断没有其他线程修改呢？可能被修改后的值与比较的值一样呀

OneThin

能否出一节讲一下unsafe，感觉这个才是最基础的。另外unsafe为什么叫unsafe呢

2018-07-16

王胖小子

2018-07-09

CAS有部分实现是解决ABA问题，可以讲一下ABA问题是如何解决的，除了version外，还有没有其他的方式

卡斯瓦德

2018-07-05

老师请教个问题，acquireQueued的源代码中，使用for(;;)做了个自旋锁吧，作者为什么不用while(true)，这种方式呢，是因为开销不一样吗？

作者回复

2018-07-07

也许，这个我不知道具体原因，看上去while会比for多一个变量

黄明恩

2018-06-28

老师可否分析下Object.wait和notify的原理

爱新觉罗老流氓

2018-06-27

ReentrantLock的非公平锁，其实只有一次非公平的机会！那一次就是在lock方法中，非公平锁的实现有if else分支，在if时就进行一次cas state，成功的线程去执行任务代码去了。那么失败的线程就会进入else逻辑，就是AQS#acquired，从这里开始非公平锁和公平锁就完全一样了：只是公平锁被欺负了一次，它的lock方法是直接调acquired方法。

为什么只有这一次呢？先看AQS#acquired，第一个逻辑是tryAcquired，公平锁和非公平锁实现略有区别。但记住，在这个时刻下，即使你看到公平锁tryAcquired实现中多一个hasQueuedPredecessors判断，无关紧要，重要的是这个时刻，还没有执行后面的addWaiter逻辑，根本没有入队，那么公平锁进入这个hasXXX方法，当然也是马上出来，执行后面的cas state，跟不公平锁没有不同……

如果，AQS#acquired的第一个tryAcquired失败了，都会进入acquiredQueued，此方法中有个强制的逻辑，就是无限for循环中的 final Node p = node.predecessors(); 在这个逻辑下，公平锁也要乖乖排队……

以上只是分析了lock方法，带超时的tryLock方法还没有具体看代码。如果我的lock分析有误，欢迎指出批评！

三口先生

2018-06-26

大于0取消状态，小于0有效状态，表示等待状态四种cancelled, signal, condition, propagate

作者回复

2018-06-28

不错

antipas

2018-06-26

看AQS源码过程中产生了新问题，它对线程的挂起唤醒是通过locksupport实现的，那么它与wait/notify又有何不同，使用场景有何不同。我的理解是使用 wait/notify需要synchronized锁，而lock/unlock需要条件触发

作者回复

2018-06-28

这是两种方式，wait基于monitor，一般用并发库就不用Object.wait、notify之类了

I.am.DZX

2018-06-26

CANCELLED 1 因为超时或中断设置为此状态，标志节点不可用

SIGNAL -1 处于此状态的节点释放资源时会唤醒后面的节点

CONDITION -2 处于条件队列里，等待条件成立(signal.signalall) 条件成立后会置入获取资源的队列里

PROPAGATE -3 共享模式下使用，头节点获取资源时将后面节点设置为此状态，如果头节点获取资源后还有足够的资源，则后面节点会尝试获取，这个状态主要是为了共享状态下队列里足够多的节点同时获取资源

0 初始状态

作者回复

2018-06-28

好

三木子

2018-06-26

最近遇到配置tomcat连接池，导致cpu过高问题，最后发现配置连接池数过大导致上下文切换次数过多，也就是线程池中任务数过少，空闲的线程过多，我想问为什么会导致上下文切换过多？

TonyEasy

2018-06-26

老师，说实话这一期的对我来说有点难度了，钦佩老师对知识理解的深入，请问老师可以指点下java学习的路线图吗，或者您分享下您自己的学习路线。

作者回复

2018-06-28

大家基础不一样，以后被问到不生疏也好；关于路线，不知道你的兴趣和规划是什么，通常来说Java只是技能树中的一项，项目经验，领域知识，综合起来才能要到高价







第23讲 | 请介绍类加载过程，什么是双亲委派模型？

2018-06-28 杨晓峰



第23讲 | 请介绍类加载过程，什么是双亲委派模型？  
杨晓峰  
00:05 / 13:58

Java通过引入字节码和JVM机制，提供了强大的跨平台能力。理解Java的类加载机制是深入Java开发的必要条件，也是个面试考察热点。

今天我要问你的是问题：[请介绍类加载过程，什么是双亲委派模型？](#)

#### 典型回答

一般来说，我们把Java的类加载过程分为三个主要步骤：加载、链接、初始化，具体行为在[Java虚拟机规范](#)里有非常详细的定义。

首先是加载阶段（Loading），它是Java将字节码数据从不同的数据源读取到JVM中，并映射为JVM认可的数据结构（Class对象），这里的数据源可能是各种各样的形态，如jar文件、class文件，甚至是网络数据源等；如果输入数据不是ClassFile的结构，则会抛出ClassFormatError。

加载阶段是用户参与的阶段，我们可以自定义类加载器，去实现自己的类加载过程。

第二阶段是链接（Linking），这是核心的步骤，简单说是把原始的类定义信息平滑地转化入JVM运行的过程中。这里可进一步细分为三个步骤：

- 验证（Verification），这是虚拟机安全的重要保障，JVM需要核验字节码是否符合Java虚拟机规范的，否则就被认为是VerifyError，这样就防止了恶意信息或者不合规的信息危害JVM的运行，验证阶段有可能触发更多class的加载。
- 准备（Preparation），创建类或接口中的静态变量，并初始化静态变量的初始值。但这里的“初始化”和下面的显式初始化阶段是有区别的，侧重点在于分配所需要的内存空间，不会去执行更进一步的JVM指令。
- 解析（Resolution），在这一步会将常量池中的符号引用（symbolic reference）替换为直接引用。在[Java虚拟机规范](#)中，详细介绍了类、接口、方法和字段等各个方面的解析。

最后是初始化阶段（Initialization），这一步真正去执行类初始化的代码逻辑，包括静态字段赋值的动作，以及执行类定义中的静态初始化块内的逻辑，编译器在编译阶段就会把这部分逻辑整理好，父类型的初始化逻辑优先于当前类型的逻辑。

再来谈谈双亲委派模型，简单说就是当类加载器（Class-Loader）试图加载某个类型的时候，除非父加载器找不到相应类型，否则尽量将这个任务代理给当前加载器的父加载器去做。使用委派模型的目的是避免重复加载Java类型。

#### 考点分析

今天的问题是关于JVM类加载方面的基础问题，我前面给出的回答参考了[Java虚拟机规范](#)中的主要条款。如果你在面试中回答这个问题，在这个基础上还可以举例说明。

我们来看一个经典的延伸问题，准备阶段谈到静态变量，那么对于常量和不同静态变量有什么区别？

需要明确的是，没有人能够精确的理解和记忆所有信息，如果碰到这种问题，有直接答案当然最好；没有的话，就说说自己的思路。

我们定义下面这样的类型，分别提供了普通静态变量、静态常量，常量又考虑到原始类型和引用类型可能有区别。

```
public class CLPreparation {
    public static int a = 100;
    public static final int INT_CONSTANT = 1000;
    public static final Integer INTEGER_CONSTANT = Integer.valueOf(10000);
}
```

编译并反编译一下：

```
Java CLPreparation.java
Javap -v CLPreparation.class
```

可以在字节码中看到这样的额外初始化逻辑：

```
0: bipush    100
2: putstatic #2          // Field a:I
5: sipush    10000
8: invokestatic #3       // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
11: putstatic #4         // Field INTEGER_CONSTANT:Ljava/lang/Integer;
```

这能让我们更清楚，普通原始类型静态变量和引用类型（即使是常量），是需要额外调用putstatic等JVM指令的，这些是在显式初始化阶段执行，而不是准备阶段调用；而原始类型常量，则不需要这样的步骤。

关于类加载过程的更多细节，有非常多的优秀资料进行介绍，你可以参考大名鼎鼎的《深入理解Java虚拟机》，一本非常好的入门书籍。我的建议是不要仅看教程，最好能够想出代码实例去验证自己对某个方面的理解和判断，这样不仅能加深理解，还能够在未来的应用开发中使用到。

其实，类加载机制的范围实在太大，我从开发和部署的不同角度，各选取了一个典型扩展问题供你参考：

- 如果要真正理解双亲委派模型，需要理解Java中类加载器的架构和职责，至少要懂具体有哪些内建的类加载器，这些是我上面的回答里没有提到的；以及如何自定义类加载器？
- 从应用角度，解决某些类加载问题，例如我的Java程序启动较慢，有没有办法尽量减小Java类加载的开销？

另外，需要注意的是，在Java 9中，Jigsaw项目为Java提供了原生的模块化支持，内建的类加载器结构和机制发生了明显变化。我会对此进行讲解，希望能够避免一些未来升级中可能产生的问题。

#### 知识扩展

首先，从架构角度，一起来看看Java 8以前各种类加载器的结构，下面是三种Oracle JDK内建的类加载器。

- 启动类加载器（Bootstrap Class-Loader），加载 jre/lib下面的jar文件，如rt.jar。它是个超级公民，即使是在开启了Security Manager的时候，JDK仍赋予了它加载的程序AllPermission。

对于做底层开发的工程师，有的时候可能不得不去试图修改JDK的基础代码，也就是通常意义上的核心类库，我们可以使用下面的命令行参数。

```
# 指定新的bootclasspath，替换java.*包的内部实现
java -Xbootclasspath:<your_boot_classpath> your_App

# a意味着append，将指定目录添加到bootclasspath后面
java -Xbootclasspath/a:<your_dir> your_App

# p意味着prepend，将指定目录添加到bootclasspath前面
java -Xbootclasspath/p:<your_dir> your_App
```

用法其实很易懂，例如，使用最常见的“/p”，既然是前置，就有机会替换个别基础类的实现。

我们一般可以使用下面方法获取父加载器，但是在通常的JDK/JRE实现中，扩展类加载器getParent()都只能返回null。

```
public final ClassLoader getParent()
```

- 扩展类加载器（Extension or Ext Class-Loader），负责加载我们放到jre/lib/ext目录下面的jar包，这就是所谓的extension机制。该目录也可以通过设置“java.ext.dirs”来覆盖。

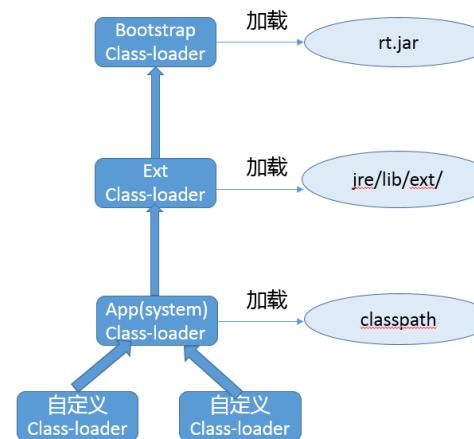
```
java -Djava.ext.dirs=your_ext_dir HelloWorld
```

- 应用类加载器（Application or App Class-Loader），就是加载我们最熟悉的classpath的内容。这里有一个容易混淆的概念，系统（System）类加载器，通常来说，其默认就是JDK内建的应用类加载器，但是它同样是可能修改的，比如：

```
java -Djava.system.class.loader=com.yourcorp.YourClassLoader HelloWorld
```

如果我们指定了这个参数，JDK内建的应用类加载器就会成为定制加载器的父亲，这种方式通常用在类似需要改变双亲委派模式的场景。

具体请参考下图：

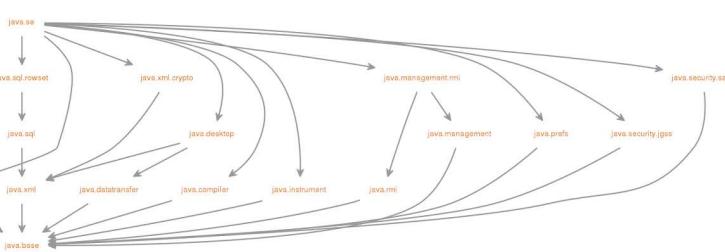


至于前面被问到的双亲委派模型，参考这个结构图更容易理解。试想，如果不同类加载器都自己加载需要的某个类型，那么就会出现多次重复加载，完全是种浪费。

通常类加载机制有三个基本特征：

- 双亲委派模型。但是不是所有类加载器都遵守这个模型，有的时候，启动加载器所加载的类型，是可能要加载用户代码的，比如JDK内部的ServiceProvider/[ServiceLoader](#)机制，用户可以在标准API框架上，提供自己的实现，JDK也需要提供些默认的参考实现。例如，Java中JNDI、JDBC、文件系统、Cipher等很多方面，都是利用的这种机制，这种情况就不会用双亲委派模型去加载，而是利用所谓的上下文加载器。
- 可见性。子类加载器可以访问父加载器加载的类型，但是反过来是不允许的，不然，因为缺少必要的隔离，我们就没有办法利用类加载器去实现容器的逻辑。
- 单一性。由于父加载器的类型对于子加载器是可见的，所以父加载器中加载过的类型，就不会在子加载器中重复加载。但是注意，类加载器“邻居”间，同一类型仍然可以被加载多次，因为互相并不可见。

在JDK 9中，由于Jigsaw项目引入了Java平台模块化系统 (JPMS)，Java SE的源代码被划分为一系列模块。



类加载器，类文件容器等都发生了非常大的变化，我这里总结一下：

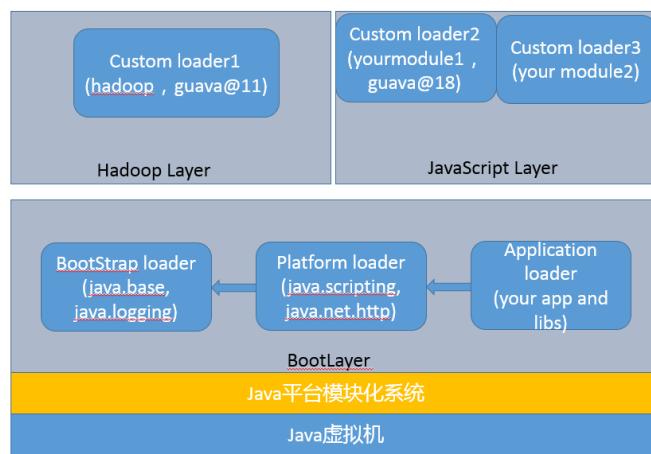
- 前面提到的-Xbootclasspath参数不可用了。API已经被划分到具体的模块，所以上文中，利用-Xbootclasspath/p替换某个Java核心类型代码，实际上变成了对相应的模块进行的修补，可以采用下面的解决方案：

首先，确认要修改的类文件已经编译好，并按照对应模块（假设是java.base）结构存放，然后，给模块打补丁：

```
java -patch-module java.base=your_patch yourApp
```

- 扩展类加载器被重命名为平台类加载器（Platform Class-Loader），而且extension机制则被移除。也就意味着，如果我们指定java.ext.dirs环境变量，或者lib/ext目录存在，JVM将直接返回错误！建议解决办法就是将其放入classpath里。
- 部分不需要AllPermission的Java基础模块，被降级到平台类加载器中，相应的权限也被更精细粒度地限制起来。
- rt.jar和tools.jar同样是被移除了！JDK的核心类库以及相关资源，被存储在JImage文件中，并通过新的JRT文件系统访问，而不是原有的JAR文件系统。虽然看起来很惊人，但幸好对于大部分软件的兼容性影响，其实是有限的，更直接地影响是IDE等软件，通常只要升级到新版本就可以了。
- 增加了Layer的抽象。JVM启动默认创建BootLayer，开发者也可以自己去定义和实例化Layer，可以更加方便的实现类似容器一般的逻辑抽象。

结合了Layer，目前的JVM内部结构就变成了下面的层次，内建类加载器都在BootLayer中，其他Layer内部有自定义的类加载器，不同版本模块可以同时工作在不同的Layer。



谈到类加载器，绕不过的一个话题是自定义类加载器，常见的场景有：

- 实现类似进程内隔离，类加载器实际上用作不同的命名空间，以提供类似容器、模块化的效果。例如，两个模块依赖于某个类库的不同版本，如果分别被不同的容器加载，就可以互不干扰。这个方面的集成者是[Java EE](#)和[OSGI](#)、[JPM](#)等框架。
- 应用需要从不同的数据源获取类定义信息，例如网络数据源，而不是本地文件系统。
- 或者是需要自己操纵字节码，动态修改或者生成类型。

我们可以总体上简单理解自定义类加载过程：

- 通过指定名称，找到其二进制实现，这里往往就是自定义类加载器会“定制”的部分，例如，在特定数据源根据名字获取字节码，或者修改或生成字节码。
- 然后，创建Class对象，并完成类加载过程。二进制信息到Class对象的转换，通常就依赖[defineClass](#)，我们无需自己实现，它是final方法。有了Class对象，后续完成加载过程就顺理成章了。

具体实现我建议参考这个[示例](#)。

我在[专栏第1讲](#)中，就提到了由于字节码是平台无关抽象，而不是机器码，所以Java需要类加载和解释、编译，这些都导致Java启动变慢。谈了这么多类加载，有没有什么通用办法，不需要代码和其他工作量，就可以降低类加载的开销呢？

这个，可以有。

- 在第1讲中提到的AOT，相当于直接编译成机器码，降低的其实主要是解释和编译开销。但是其目前还是个试验特性，支持的平台也有限，比如，JDK 9仅支持Linux x64，所以局限性太大，先暂且不谈。
- 还有就是较少人知道的AppCDS（Application Class-Data Sharing），CDS在Java 5中被引进，但仅限于Bootstrap Class-loader，在8u40中实现了AppCDS，支持其他的类加载器，在目前2018年初发布的JDK 10中已经开源。

简单来说，AppCDS基本原理和工作过程是：

首先，JVM将类信息加载，解析成为元数据，并根据是否需要修改，将其分为Read-Only部分和Read-Write部分。然后，将这些元数据直接存储在文件系统中，作为所谓Shared Archive。命令很简单：

```
Java -Xshare:dump -XX:+UseAppCDS -XX:SharedArchiveFile=<jsa> \
-XX:SharedClassListFile=<classlist> -XX:SharedArchiveConfigFile=<config_file>
```

第二，在应用程序启动时，指定归档文件，并开启AppCDS。

```
Java -Xshare:on -XX:+UseAppCDS -XX:SharedArchiveFile=<jsa> yourApp
```

通过上面的命令，JVM会通过内存映射技术，直接映射到相应的地址空间，免除了类加载、解析等各种开销。

AppCDS改善启动速度非常明显，传统的Java EE应用，一般可以提高20%~30%以上；实验中使用Spark KMeans负载，20个slave，可以提高11%的启动速度。

与此同时，降低内存footprint，因为同一环境的Java进程间可以共享部分数据结构。前面谈到的两个实验，平均可以减少10%以上的内存消耗。

当然，也不是没有局限性，如果恰好大量使用了运行时动态类加载，它的帮助就有限了。

今天我梳理了一下类加载的过程，并针对Java新版中类加载机制发生的变化，进行了相对全面的总结，最后介绍了一个改善类加载速度的特性，希望对你有所帮助。

### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，谈谈什么是Jar Hell问题？你有遇到过类似情况吗，如何解决呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



Allen

2018-06-28

还能再讲讲ORM映射的细节吗？端个小板凳准备听◆◆

作者回复

有计划提到，但暂时没计划单独讲，如果需要的朋友多，也许后期加餐吧

2018-06-29

Allen

2018-06-28

希望能在后面讲解一哈IOC和AOP原理，期待

jacy

2018-07-03

老师能否讲讲一般什么场景下需要自定义加载，有什么好处，为什么不用其他方式解决，比如jar hell。此问题可以通过其他方式直接解决。有的评论提到自定义类加载，我并不认为是比较好的解决方案。

PeterBO15

2018-06-29

Jar hell jar包冲突，对于大项目或没有maven的项目是比较麻烦的。1 应用无法启动 2 编译时没问题，运行时报错。解决方法：1 改为maven 项目，使包的管理和依赖可视化2 在1的基础上，解决明显的包编译冲突 3 根据运行时报错找到冲突的包，或者要排除的包

作者回复

不错

2018-06-29

Daniel

2018-06-28

感谢，这块是我需要恶补的地方，虽然很多看不懂，但是多看几遍还是有用处的。佩服老师！

作者回复

哪一块儿，抱歉，看来需要再改进一下，感谢认可

Walter

2018-06-28

当一个类或一个资源文件存在多个jar中，就会出现jar hell问题。

可以通过以下代码来诊断方案：

```
try {
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    String resourceName = "net/sf/cglib/proxy/MethodInterceptor.class";
    Enumeration<URL> urls = classLoader.getResources(resourceName);
    while(urls.hasMoreElements()){
        System.out.println(urls.nextElement());
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

输出结果：jar:file:/D:/workspace/Test/lib/cglib-3.2.4.jar!/net/sf/cglib/proxy/MethodInterceptor.class

三木子

2018-06-28

就是一个类存在不同的jar包中，解决办法就是自己加载手动通过classloader加载类

DesertSnow

2018-06-28

老大666

helloworld

2018-07-17

一知半解

balance

2018-07-12

深奥

小白001

那两个war依赖相同的jar包，运行在同一个jvm，类会重复加载吗？

作者回复

2018-07-07

回复过了，我认为应该看具体server设计，一般共享jar或者单独应该都能做得到

2018-07-12

小白001

两个war依赖相同的jar包，部署在同一个tomcat，类会重复加载吗？

作者回复

2018-07-05

没有深入研究Tomcat类加载，逻辑上，这种不应该是可选吗？要么共享，要么各自一份，各有利弊的样子

2018-07-07

wupengfei

请教老师，使用委派模型具体是怎么避免重复加载java类型的？

作者回复

2018-07-04

同一加载器不能重复加载某个类型，既然都尽量委派给父加载器，除非它加载不了，不然就只有一份

2018-07-05

A\_前我去

老师老师，如果我写了一个java.lang.String类，怎么进行加载的，怎么跟原来的类进行区分的？

作者回复

2018-07-03

试验下吧，除非你修改jdk本身实现，不然加载不了

2018-07-05

三口先生

Jar冲突的问题，一般源于一个资源文件出现在多个jar里面啦。优先加载自己指定路径下的jar文件，如果加载不到，交给夫类

2018-06-28

Kyle

2018-06-28

Jar Hell问题：当一个类或一个资源文件存在多个jar中，就会出现Jar Hell问题。这不就是我们平常用的jar包冲突？

网上的解决方案都是通过写一段类加载代码将冲突的类、jar包打印出来。平常我自己的话，会利用Eclipse、IDEA里的显示jar包加载结构的插件来检查出冲突的jar。







第24讲 | 有哪些方法可以在运行时动态生成一个Java类?

2018-06-30 杨晓峰



第24讲 | 有哪些方法可以在运行时动态生成一个Java类?  
杨晓峰  
- 0:00 / 08:29

在开始今天的学习前，我建议你先复习一下[专栏第6讲](#)有关动态代理的内容。作为Java基础模块中的内容，考虑到不同基础的同学以及一个循序渐进的学习过程，我当时并没有在源码层面介绍动态代理的实现技术，仅进行了相应的技术比较。但是，有了[上一讲](#)的类加载的学习基础后，我想是时候该进行深入分析了。

今天我要问你的是，[有哪些方法可以在运行时动态生成一个Java类?](#)

#### 典型回答

我们可以从常见的Java类来源分析，通常的开发过程是，开发者编写Java代码，调用javac编译成class文件，然后通过类加载机制载入JVM，就成为应用运行时可以使用的Java类了。

从上面过程得到启发，其中一个直接的方式是从源码入手，可以利用Java程序生成一段源码，然后保存到文件等，下面就只需要解决编译问题了。

有一种笨办法，直接用ProcessBuilder之类启动javac进程，并指定上面生成的文件作为输入，进行编译。最后，再利用类加载器，在运行时加载即可。

前面的方法，本质上还是在当前程序之外编译的，那么还有没有不这么low的办法呢？

你可以考虑使用Java Compiler API，这是JDK提供的标准API，里面提供了与javac对等的编译器功能，具体请参考[java.compiler](#)相关文档。

进一步思考，我们一直围绕Java源码编译成为JVM可以理解的字节码，换句话说，只要是符合JVM规范的字节码，不管它是如何生成的，是不是都可以被JVM加载呢？我们能不能直接生成相应的字节码，然后交给类加载器去加载呢？

当然也可以，不过直接去写字节码难度太大，通常我们可以利用Java字节码操纵工具和类库来实现，比如在[专栏第6讲](#)中提到的[ASM](#)、[Javassist](#)、cglib等。

#### 考点分析

虽然曾经被视为黑魔法，但在当前复杂多变的开发环境中，在运行时动态生成逻辑并不是什么罕见的场景。重新审视我们谈到的动态代理，本质上不就是在特定的时机，去修改已有类型实现，或者创建新的类型。

明白了基本思路后，我还是围绕类加载机制进行展开，面试过程中面试官很可能从技术原理或实践的角度考察：

- 字节码和类加载到底是怎么无缝进行转换的？发生在整个类加载过程的哪一步？
- 如何利用字节码操纵技术，实现基本的动态代理逻辑？
- 除了动态代理，字节码操纵技术还有那些应用场景？

#### 知识扩展

首先，我们来理解一下，类从字节码到Class对象的转换，在类加载过程中，这一步是通过下面的方法提供的功能，或者defineClass的其他本地对等实现。

```
protected final Class<> defineClass(String name, byte[] b, int off, int len,
                                      ProtectionDomain protectionDomain)
protected final Class<> defineClass(String name, java.nio.ByteBuffer b,
                                      ProtectionDomain protectionDomain)
```

我这里只选取了最基础的两个典型的defineClass实现，Java重载了几个不同的方法。

可以看出，只要能够生成出规范的字节码，不管是作为byte数组的形式，还是放到ByteBuffer里，都可以平滑地完成字节码到Java对象的转换过程。

JDK提供的defineClass方法，最终都是本地代码实现的。

```

static native Class<?> defineClass1(ClassLoader loader, String name, byte[] b, int off, int len,
                                      ProtectionDomain pd, String source);

static native Class<?> defineClass2(ClassLoader loader, String name, java.nio.ByteBuffer b,
                                      int off, int len, ProtectionDomain pd,
                                      String source);

```

更进一步，我们来看看JDK dynamic proxy的[实现代码](#)。你会发现，对应逻辑是实现在ProxyBuilder这个静态内部类中，ProxyGenerator生成字节码，并以byte数组的形式保存，然后通过调用Unsafe提供的defineClass入口。

```

byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
    proxyName, interfaces.toArray(EMPTY_CLASS_ARRAY), accessFlags);
try {
    Class<?> pc = UNSAFE.defineClass(proxyName, proxyClassFile,
        0, proxyClassFile.length,
        loader, null);
    reverseProxyCache.sub(pc).putIfAbsent(loader, Boolean.TRUE);
    return pc;
} catch (ClassFormatError e) {
    // 如果出现ClassFormatError，很可能输入参数有问题，比如 ProxyGenerator#bug
}

```

前面理顺了二进制的字节码信息到Class对象的转换过程，似乎我们还没有分析如何生成自己需要的字节码，接下来一起来看看相关的字节码操纵逻辑。

JDK内部动态代理的逻辑，可以参考[java.lang.reflect.ProxyGenerator](#)的内部实现。我觉得可以认为这是种另类的字节码操纵技术，其利用了[DataOutputStream](#)提供的能力，配合hard-coded的各种JVM指令实现方法，生成所需的字节码数组。你可以参考下面的示例代码。

```

private void codeLocalLoadStore(int lvar, int opcode, int opcode_0,
                               DataOutputStream out)
throws IOException
{
    assert lvar >= 0 && lvar <= 0xFFFF;
    // 根据变量数值，以不同格式 dump操作码
    if (lvar <= 3) {
        out.writeByte(opcode_0 + lvar);
    } else if (lvar <= 0xFF) {
        out.writeByte(opcode);
        out.writeByte(lvar & 0xFF);
    } else {
        // 使用宽指令修饰符，如果变量索引不能用无符号byte
        out.writeByte(opcode_wide);
        out.writeByte(opcode);
        out.writeShort(lvar & 0xFFFF);
    }
}

```

这种实现方式的好处是没有太多依赖关系，简单实用，但是前提是你需要懂各种[JVM指令](#)，知道怎么处理那些偏移地址等，实际门槛非常高，所以并不适合大多数的普通开发场景。

幸好，Java社区专家提供了各种从底层到更高抽象水平的字节码操作类库，我们不需要什么都自己从头做。JDK内部就集成了ASM类库，虽然并未作为公共API暴露出来，但是它广泛建议在，如[java.lang.instrumentation](#) API底层实现，或者[Lambda Call Site](#)生成的内部逻辑中，这些代码的实现我就不在这里展开了，如果你确实有兴趣或有需要，可以参考类似LamdaForm的字节码生成逻辑：[java.lang.invoke.InvokeBytecodeGenerator](#)。

从相对实用的角度思考一下，实现一个简单的动态代理，都要做什么？如何使用字节码操纵技术，走通这个过程呢？

对于一个普通的Java动态代理，其实实现过程可以简化成为：

- 提供一个基础的接口，作为被调用类型（com.mycorp.HelloImpl）和代理类之间的统一入口，如com.mycorp.Hello。
- 实现[InvocationHandler](#)，对代理对象方法的调用，会被分派到其invoke方法来真正实现动作。
- 通过Proxy类，调用其newProxyInstance方法，生成一个实现了相应基础接口的代理类实例，可以看下面的方法签名。

```

public static Object newProxyInstance(ClassLoader loader,
                                     Class<?>[] interfaces,
                                     InvocationHandler h)

```

我们分析一下，动态代码生成是具体发生在什么阶段呢？

不错，就是在newProxyInstance生成代理类实例的时候。我选取了JDK自己采用的ASM作为示例，一起来看看用ASM实现的简要过程，请参考下面的示例代码片段。

第一步，生成对应的类，其实和我们去写Java代码很类似，只不过改为用ASM方法和指定参数，代替了我们书写的源码。

```

ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);

cw.visit(V1_8,           // 指定Java版本
        ACC_PUBLIC,      // 说明是public类型
        "com/mycorp/HelloProxy", // 指定包和类的名称

```

```
null, // 签名, null表示不是泛型
"java/lang/Object", // 指定父类
new String[]{ "com/mycorp>Hello" }); // 指定需要实现的接口
```

更进一步，我们可以按照需要为代理对象实例，生成需要的方法和逻辑。

```
MethodVisitor mv = cw.visitMethod(
    ACC_PUBLIC, // 声明公共方法
    "sayHello", // 方法名称
    "()Ljava/lang/Object;", // 描述符
    null, // 签名, null表示不是泛型
    null); // 可能抛出的异常, 如果有, 则指定字符串数组

mv.visitCode();
// 省略代码编辑实现细节
cw.visitEnd(); // 结束类字节码生成
```

上面的代码虽然有些晦涩，但总体还是能多少理解其用意。不同的`visitX`方法提供了创建类型、创建各种方法等逻辑。ASM API，广泛的使用了[Visitor](#)模式，如果你熟悉这个模式，就会知道它所针对的场景是将算法和对象结构解耦，非常适合字节码操纵的场合，因为我们大部分情况都是依赖于特定结构修改或者添加新的方法、变量或者类型等。

按照前面的分析，字节码操作最后大都是生成`byte`数组，`ClassWriter`提供了一个简便的方法。

```
cw.toByteArray();
```

然后，就可以进入我们熟知的类加载过程了，我就不再赘述了，如果你对ASM的具体用法感兴趣，可以参考这个[教程](#)。

最后一个问题是，字节码操纵技术，除了动态代理，还可以应用在什么地方？

这个技术似乎离我们日常开发遥远，但其实已经深入到各个方面，也许很多你现在正在使用的框架、工具就应用该技术，下面是我能想到的几个常见领域。

- 各种Mock框架
- ORM框架
- IOC容器
- 部分Profiler工具，或者运行时诊断工具等
- 生成形式化代码的工具

甚至可以认为，字节码操纵技术是工具和基础框架必不可少的部分，大大减少了开发者的负担。

今天我们探讨了更加深入的类加载和字节码操作方面技术。为了理解底层的原理，我选取的例子是比较偏底层的、能力全面的类库，如果实际项目中需要进行基础的字节码操作，可以考虑使用更高层次视角的类库，例如[Byte Buddy](#)等。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？试想，假如我们有这样一个需求，需要添加某个功能，例如对某类型资源如网络通信的消耗进行统计，重点要求是，不开启时必须是\*\*零开销，而不是低开销，\*\*可以利用我们今天谈到的或者相关的技术实现吗？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



三口先生

2018-06-30

将资源消耗的这个实例，用动态代理的方式创建这个实例动态代理对象，在动态代理的`invoke`中添加新的需求。开始使用代理对象，不开启则使用原来的方法，因为动态代理是在运行时创建，所以是零消耗。

作者回复

2018-06-30

不错

omegamiao

2018-07-17

请问老师，使用classloader动态加载的外部jar包，应该如何正确的卸载？已经加载到systemclassloader.....通过反射urlclassloader的addurl方法加进去的  
作者回复

2018-07-18

加载到system classloader... 我理解卸载不了

胡居居

2018-07-05

所有用到java反射的地方，底层都是采用了字节码操纵技术，老师，这么理解对吗？

作者回复

2018-07-07

不是，Proxy这里是个特例，因为需要生成新的class

tyson

2018-07-03

可以考虑用javaagent+字节码处理拦截方法进行统计：对httpclient中的方法进行拦截，增加header或者转发等进行统计。开启和关闭只要增加一个javaagent启动参数就行  
作者回复

2018-07-04

是的，我自己也是用Javaagent方案

rivers

2018-07-02

希望作者能详细的讲下，javaassist等其他代理模式的使用，砍掉了很多内容，尽量考虑下水平有限的读者

四阿哥

2018-06-30

老师，您这个专栏完结了还会不会出其他专栏，你的每一篇我起码要听三四遍，我都是咬文嚼字的听，非常有用，非常好的内心法

作者回复

2018-06-30

非常感谢，老学究感到很欣慰，希望能对未来实践有帮助，专栏形式更多的是解决知识点的问题，后续专栏还没开始思考

antipas

2018-06-30

无痕埋点原理就是这样。像注解类框架也用到了比如retrofit

作者回复

2018-06-30

是的

hansc

2018-06-30

请问老师，ioc容器哪里使用到了asm？

作者回复

2018-06-30

不见得是asm，而是字节码操纵





## 第25讲 | 谈谈JVM内存区域的划分，哪些区域可能发生OutOfMemoryError?

2018-07-03 杨晓峰



第25讲 | 谈谈JVM内存区域的划分，哪些区域可能发生OutOfMemoryError?  
杨晓峰  
00:00 / 10:42

今天，我将从内存管理的角度，进一步探索Java虚拟机（JVM）。垃圾收集机制为我们打理了很多繁琐的工作，大大提高了开发的效率，但是，垃圾收集也不是万能的，懂得JVM内部的内存结构、工作机制，是设计高扩展性应用和诊断运行时问题的基础，也是Java工程师进阶的必备能力。

今天我要问你的是，[谈谈JVM内存区域的划分，哪些区域可能发生OutOfMemoryError?](#)

## 典型回答

通常可以把JVM内存区域分为下面几个方面，其中，有的区域是以线程为单位，而有的区域则是整个JVM进程唯一的。

首先，程序计数器（PC, Program Counter Register）。在JVM规范中，每个线程都有它自己的程序计数器，并且任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的Java方法的JVM指令地址；或者，如果是在执行本地方法，则是未指定值（undefined）。

第二，Java虚拟机栈（Java Virtual Machine Stack），早期也叫Java栈。每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的Java方法调用。

前面谈程序计数器时，提到了当前方法；同理，在一个时间点，对应的只会有一个活动的栈帧，通常叫作当前帧，方法所在的类叫作当前类。如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，成为新的当前帧，一直到它返回结果或者执行结束。JVM直接对Java栈的操作只有两个，就是对栈帧的压栈和出栈。

栈帧中存储着局部变量表、操作数（operand）栈、动态链接、方法正常退出或者异常退出的定义等。

第三，堆（Heap），它是Java内存管理的核心区域，用来放置Java对象实例，几乎所有创建的Java对象实例都是被直接分配在堆上。堆被所有的线程共享，在虚拟机启动时，我们指定的“Xmx”之类参数就是用来指定最大堆空间等指标。

理所当然，堆也是垃圾收集器重点照顾的区域，所以堆内空间还会被不同的垃圾收集器进行进一步的细分，最有名的就是新生代、老年代的划分。

第四，方法区（Method Area）。这也是所有线程共享的一块内存区域，用于存储所谓的元（Meta）数据，例如类结构信息，以及对应的运行时常量池、字段、方法代码等。

由于早期的Hotspot JVM实现，很多人习惯于将方法区称为永久代（Permanent Generation）。Oracle JDK 8中将永久代移除，同时增加了元数据区（Metaspace）。

第五，运行时常量池（Run-Time Constant Pool），这是方法区的一部分。如果仔细分析过反编译的类文件结构，你能看到版本号、字段、方法、超类、接口等各种信息，还有一项信息就是常量池。Java的常量池可以存放各种常量信息，不管是编译期生成的各种字面量，还是需要在运行时决定的符号引用，所以它比一般语言的符号表存储的信息更加宽泛。

第六，本地方方法栈（Native Method Stack）。它和Java虚拟机栈是非常相似的，支持对本地方法的调用，也是每个线程都会创建一个。在Oracle Hotspot JVM中，本地方方法栈和Java虚拟机栈是在同一块儿区域，这完全取决于技术实现的决定，并未在规范中强制。

## 考点分析

这是个JVM领域的基础题目，我给出的答案依据的是[JVM规范](#)中运行时数据区定义，这也和大多数书籍和资料解读的角度类似。

JVM内部的概念庞杂，对于初学者比较晦涩，我的建议是在工作之余，还是要去阅读经典书籍，比如我推荐过多次的《深入理解Java虚拟机》。

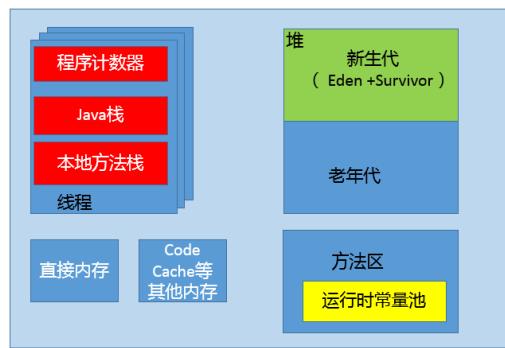
今天这一讲作为Java虚拟机内存管理的开篇，我会侧重于：

- 分析广义上的JVM内存结构或者说Java进程内存结构。
- 谈到Java内存模型，不可避免的要涉及OutOfMemory（OOM）问题。那么在Java里面存在哪些种OOM的可能性，分别对应哪个内存区域的异常状况呢？

注意，具体JVM的内存结构，其实取决于其实现，不同厂商的JVM，或者同一厂商发布的不同版本，都可能存在一定差异。我在下面的分析中，还会介绍Oracle Hotspot JVM的部分设计变化。

## 知识扩展

首先,为了让你有个更加直观、清晰的印象,我画了一个简单的内存结构图,里面展示了我前面提到的堆、线程栈等区域,并从数量上说明了什么是线程私有,例如,程序计数器、Java栈等,以及什么是Java进程唯一。另外,还额外划分出了直接内存等区域。



这张图反映了实际中Java进程内存占用,与规范中定义的JVM运行时数据区之间的差别,它可以看作是运行时数据区的一个超集。毕竟理论上的视角和现实中的视角是有区别的,规范侧重的是通用的、无差别的部分,而对于应用开发者来说,只要是Java进程在运行时会占用,都会影响到我们的工程实践。

我这里简要介绍两点区别:

- 直接内存(Direct Memory)区域,它就是我在[专栏第12讲](#)中谈到的Direct Buffer所直接分配的内存,也是个容易出现问题的地方。尽管,在JVM工程师的眼中,并不认为它是JVM内部内存的一部分,也并未体现JVM内存模型中。
- JVM本身是个本地程序,还需要其他的内存去完成各种基本任务,比如,JIT Compiler在运行时对热点方法进行编译,就会将编译后的方法储存在Code Cache里面;GC等功能需要运行在本地线程之中,类似部分都需要占用内存空间。这些是实现JVM JIT等功能的需要,但规范中并不涉及。

如果深入到JVM的实现细节,你会发现一些结论似乎有些模棱两可,比如:

- Java对象是不是都创建在堆上的呢?

我注意到有一些观点,认为通过[逃逸分析](#),JVM会在栈上分配那些不能逃逸的对象,这在理论上是可行的,但是取决于JVM设计者的选择。据我所知,Oracle Hotspot JVM中并未这么做,这一点在逃逸分析相关的[文档](#)里已经说明,所以可以明确所有的对象实例都是创建在堆上。

- 目前很多书籍还是基于JDK 7以前的版本,JDK已经发生了很大变化,Intern字符串的缓存和静态变量曾经都被分配在永久代上,而永久代已经被元数据区取代。但是,Intern字符串缓存和静态变量并不是被转移到元数据区,而是直接在堆上分配,所以这一点同样符合前面一点的结论:对象实例都是分配在堆上。

接下来,我们来看看什么是OOM问题,它可能在哪些内存区域发生?

首先,OOM如果通俗点儿说,就是JVM内存不够用了,javadoc中对[OutOfMemoryError](#)的解释是,没有空闲内存,并且垃圾收集器也无法提供更多内存。

这里面隐含着一层意思是,在抛出[OutOfMemoryError](#)之前,通常垃圾收集器会被触发,尽其所能去清理出空间,例如:

- 我在[专栏第4讲](#)的引用机制分析中,已经提到了JVM会去尝试回收软引用指向的对象。
- 在[java.nio.Bits.reserveMemory\(\)](#)方法中,我们能清楚地看到,System.gc()会被调用,以清理空间,这也是为什么在大量使用NIO的Direct Buffer之类时,通常建议不要加下面的参数,毕竟是个最后的尝试,有可能避免一定的内存不足问题。

```
-XX:+DisableExplicitGC
```

当然,也不是在任何情况下垃圾收集器都会被触发的,比如,我们去分配一个超大对象,类似一个超大数组超过堆的最大值,JVM可以判断出垃圾收集并不能解决这个问题,所以直接抛出[OutOfMemoryError](#)。

从我前面分析的数据区的角度,除了程序计数器,其他区域都有可能会因为可能的空间不足发生[OutOfMemoryError](#),简单总结如下:

- 堆内存不足是最常见的OOM原因之一,抛出的错误信息是“`java.lang.OutOfMemoryError: Java heap space`”,原因可能千奇百怪,例如,可能存在内存泄漏问题;也很有可能就是堆的大小不合理,比如我们要处理比较可观的数据量,但是没有显式指定JVM堆大小或者指定数值偏小;或者出现JVM处理引用不及时,导致堆积起来,内存无法释放等。
- 而对于Java虚拟机栈和本地方法栈,这里要稍微复杂一点。如果我们写一段程序不断的进行递归调用,而且没有退出条件,就会导致不断地进行压栈。类似这种情况,JVM实际会抛出`StackOverflowError`;当然,如果JVM试图去扩展栈空间的时候失败,则会抛出`OutOfMemoryError`。
- 对于老版本的Oracle JDK,因为永久代的大小是有限的,并且JVM对永久代垃圾回收(如,常量池回收、卸载不再需要的类型)非常不积极,所以当我们不断添加新类型的时候,永久代出现[OutOfMemoryError](#)也非常常见,尤其是在运行时存在大量动态类型生成的场合;类似Intern字符串缓存占用太多空间,也会导致OOM问题。对应的异常信息,会标记出来和永久代相关:“`java.lang.OutOfMemoryError: PermGen space`”。
- 随着元数据区的引入,方法区内存已经不再那么窘迫,所以相应的OOM有所改观,出现OOM,异常信息则变成了:“`java.lang.OutOfMemoryError: Metaspace`”。
- 直接内存不足,也会导致OOM,这个已经在[专栏第11讲](#)介绍过。

今天是JVM内存部分的第一讲,算是我们先进行了热身准备,我介绍了主要的内存区域,以及在不同版本Hotspot JVM内部的变化,并且分析了各区域是否可能产生`OutOfMemoryError`,以及OME发生的典型情况。

### 一课一练

关于今天我们讨论的题目你做到心中有数了吗?今天的思考题是,我在试图分配一个100M bytes大数据的时候发生了OME,但是GC日志显示,明明堆上还有远不止100M的空间,你觉得可能问题的原因是什么?想要弄清楚这个问题,还需要什么信息呢?

请你在留言区写写你对这个问题的思考,我会选出经过认真思考的留言,送给你一份学习奖励礼券,欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



I am a psycho

2018-07-03

如果仅从JVM的角度来看，要看下新生代和老年代的垃圾回收机制是什么。如果新生代是serial，会默认使用copying算法，利用两块eden和survivor来进行处理。但是默认当遇到超大对象时，会直接将超大对象放置到老年代中，而不用走正常对象的存活次数记录。因为要放置的是一个byte数组，那么必然需要申请连续的空间。当空间不足时，会进行gc操作。这里又需要看老年代的gc机制是哪一的。如果是serial old，那么会采用mark compact，会进行整理，从而整理出连续空间。如果还不够，说明是老年代的空间不够，所谓的堆内存大于100m是新+老共同的结果。如果采用的是cms(concurrent mark sweep)，那么只会标记清理，并不会压缩，所以内存会碎片化，同时可能出现内存垃圾。如果是cms的话，即使老年代的空间大于100m，也会出现没有连续的空间供对象使用。

作者回复

非常不错的总结

2018-07-04

石头狮子

2018-07-03

1. 新生代大小过小。无法分配足够的内存。同时也老年代过小，导致提升失败。这时系统认为没有足够的空间存放该100M数据。  
2. 堆可以抽象的看成计算资源。堆看成存储资源。计算资源不共享，不会发生线程安全问题。堆资源共享，容易发生线程安全问题。

3. JAVA 封装了不同系统的线程模型，结果是在 java 内部有实现了一个通用的 java线程库。所以就需要用内存来保存线程信息。

鸡肉饭饭

2018-07-03

我们拿JDK7来说，有可能的原因是JVM的剩余内存有100M，但是它是分在不同年齡的内存区域。

因此应当单独的去看每一块eden, survivor, old的大小。（通过Survivor Ratio知道s和e的比例大小，通过MaxNewSize知道young和old的比例）看看这三块区域是否有超过100M的内存大小。如果没有，就是因为没有一个区域能够再存储一个100M的对象。

如果有，就可以通过工具查看下，每一块e s o每一块区域剩下的内存空间，如果没有一块内存大小超过100M，便是因为这个原因导致数组分配失败。

LenX

2018-07-03

从不同的垃圾收集器角度来看：

首先，数组的分配是需要连续的内存空间的（据说，有个别非主流JVM支持数组用不连续的内存空间分配◆◆）。所以：

1) 对于使用年轻代和老年代来管理内存的垃圾收集器，堆大于 100M，表示的是新生代和老年代加起来总和大于100M，而新生代和老年代各自并没有大于 100M 的连续内存空间。

进一步，又由于数组一般直接进入老年代（会跳过对对象的年龄的判断），所以，是否可以认为老年代中没有连续大于 100M 的空间呢。

2) 对于 G1 这种按 region 来管理内存的垃圾收集器，可能的情况是没有多个连续的 region，它们的内存总和大于 100M。

当然，不管是哪种垃圾收集器以及收集算法，当内存空间不足时，都会触发 GC，只不过，可能 GC 之后，还是没有连续大于 100M 的内存空间，于是 OOM 了。

作者回复

2018-07-04

很好的视角，g1 region之类确实有影响，另外g1还是有年代的概念的

tyson

2018-07-03

堆内存100M 包含了新生代(eden+s0+o1)和老年代，大对象一般分配在老年代，那么最有可能在分配过程中老年代的空间不足。

作者回复

2018-07-04

不错，可能性很多，其实和gc的选择也有关，例如g1 region比较小

爱吃芒果的董先森

2018-07-03

因为给数组分配的是连续地址，而显示的是总的地址，不管是不是连续的。

作者回复

2018-07-04

也对，最好综合考虑堆内存结构、gc区别等，后续会讲解

boom

2018-07-03

小白请教一个不相关的问题 ~ java 内存模型跟 jvm内存模型的区别与联系是啥呢 ~

作者回复

怎么叫都有，看上下文，jsr133 jmm是解决多线程环境一致性，或者可以看做memory ordering model

2018-07-04

师琳博  
100m的byte数组，一个byte对应一个引用，这样需要100m个的引用，所以需要的堆空间也不会低于100m，而对象的引用是在栈中分配的，（栈和堆加起来估计不低于200m）况且还是数组，对应的那么多引用还需要分配连续的内存空间。堆空间够的话，个人认为可能是栈空间不足造成的

sunlight001  
堆上有空间的划分，新生代和老年代，有可能新生代的空间不够，看到的是老年代的空间，个人猜测◆◆

一个坏人  
2018-07-18  
老师好，请教一个问题。JMM模型中各种内存分区是逻辑分区的。JVM会根据参数计算每一块分区的起始地址、结束地址？如果会，什么时候执行这一操作呢？每一块区域有规定的顺序么？

代码狂徒  
2018-07-10  
老师，您是说方法区就是有永久代？那也就是说方法区在jdk8中已经不存在了？元数据区跟方法区有什么区别呢？那您的图是jdk7的图，有B得图吗？求解

作者回复  
2018-07-11  
不是，方法区只是个逻辑概念，永久带和元数据区是具体设计、实现的选择；以前放到永久带，而且永久带内部还有类似Intern字符串之类内容；元数据区具体内容和永久带也有区别，文章介绍了；那个图只是个简化示例，8去掉永久带就是了，具体到比较复杂的gc比如g1，就不是这个结构，请看后面讲

markin  
2018-07-08  
老师，能否跟我们介绍一下您平时获取资料的渠道，比如apache的一些开源项目，官网上就有很丰富的文档。但是我们获取jvm相关文档的渠道少之又少，无非就是博客或者书籍，这些都比较简单，并且可能参杂着很多难以识别的错误观点。授人以鱼不如授人以渔，先谢谢老师了。

作者回复  
2018-07-12  
Oracle官网也提供了很多好的文档：  
虚拟机规范 <https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>  
诊断指南 <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/index.html>  
调优指南 <https://docs.oracle.com/javase/10/gctuning/>  
OpenJDK网站，或者那些感兴趣的邮件列表 <http://mail.openjdk.java.net/mailman/listinfo>  
YouTube上查查javaone、JVM summit之类  
回头有必要整理个书单之类

但这些东西太多了，自己把握一下

三木子  
2018-07-05  
老师，关于这篇文章留的问题你可以给你的答案吗？

作者回复  
2018-07-07  
嗯，参考我的回复，下一讲中有更多细节，具体堆内结构还是会划分，例如tlab, eden等，可以简单理解，对象分配是试图tlab，太大就eden，还不行就oldgen，所以我们需要的是相应区域有连续空闲

一凡  
2018-07-04  
老师，请问后续的章节里有对lambda的讲解么，这方面看网上的资料实在是很难理解

作者回复  
2018-07-05  
暂时没计划，也许补充一章，Java内容太多，而且据我所知国内公司使用并不多

Steven<sup>0.08</sup>  
2018-07-04  
数组是连续分配的，gc表明有多余100m，但有可能满足不了连续100m的空间，故会报OOM

豌米豆发  
2018-07-03  
可能一，新生代没有足够的连续空间，且不能直接在老年代分配。比如E+S0+S1>100MB，但E<100MB，S0<100MB。  
可能二，大对象直接进入老年代，但老年代也没有足够的连续空间。参数+XX:PretenureSizeThreshold。  
可能三，线程数量太多，导致物理内存不足。  
可能四，直接内存使用太多，导致物理内存不足。

作者回复  
2018-07-04  
不错，下一章会有更多内存结构细节

三口先生  
2018-07-03  
堆内存比例设置不合理

作者回复  
2018-07-04  
也对，回答比较简洁，哈哈

三木子  
2018-07-03  
我觉得是新生代的空间不足，因为新生代的实际可用大小占新生代总大小的90%，这是由于新生代有Eden+s1+s0组织，实际只用了eden + 1个s

作者回复  
2018-07-04  
可能之一

A张邦卓  
2018-07-03  
堆中没有100M的连续地址了

作者回复  
2018-07-04  
差不多，细节再思考下

四阿哥  
2018-07-04

后续会到java内存模型吗？之前看了一下jsr133感觉晦涩难懂。

作者回复

会

yotsuba1022

關於課後練習，由於Heap是有分代的，所以可能當前的100M已經超過eden area的大小了，所以儘管heap size比100M要大，還是會無法分配內存。這樣理解對嗎？

作者回复

有合理性，具体情况有几种可能，大对象是可能直接分配在老年代的

wgl

堆中没有物理地址连续的100M空间了。

Kyle

还有就是，大对象会直接分配在老年代。有没有可能是老年代大小不够。

Kyle

堆里是存放数组实例，栈区里存放数组引用。是不是因为栈区满了。

未完的歌

有可能是内存碎片化问题，或者是大对象的内存分配策略问题。

需要了解一下积极垃圾收集算法，例如Mark Sweep就会造成内存碎片化问题，另外内存分配策略也是一个关注点。

作者回复

是的

2018-07-03

2018-07-04

2018-07-03

2018-07-04

2018-07-03

2018-07-03

2018-07-03

2018-07-03

2018-07-04

## 第26讲 | 如何监控和诊断JVM堆内和堆外内存使用?

2018-07-05 杨晓峰



第26讲 | 如何监控和诊断JVM堆内和堆外内存使用?  
杨晓峰  
- 00:00 / 12:36

上一讲我介绍了JVM内存区域的划分，总结了相关的一些概念，今天我将结合JVM参数、工具等方面，进一步分析JVM内存结构，包括外部资料相对较少的堆外部分。

今天我要问你的问题是，[如何监控和诊断JVM堆内和堆外内存使用?](#)

## 典型回答

了解JVM内存的方法有很多，具体能力范围也有区别，简单总结如下：

- 可以使用综合性的图形化工具，如JConsole、VisualVM（注意，从Oracle JDK 9开始，VisualVM已经不再包含在JDK安装包中）等。这些工具具体使用起来相对比较直观，直接连接到Java进程，然后就可以在图形化界面里掌握内存使用情况。

以JConsole为例，其内存页面可以显示常见的堆内存和各种堆外部分使用状态。

- 也可以使用命令行工具进行运行时查询，如jstat和jmap等工具都提供了一些选项，可以查看堆、方法区等使用数据。
- 或者，也可以使用jmap等提供的命令，生成堆转储（Heap Dump）文件，然后利用jhat或Eclipse MAT等堆转储分析工具进行详细分析。
- 如果你使用的是Tomcat、Weblogic等Java EE服务器，这些服务器同样提供了内存管理相关的功能。
- 另外，从某种程度上来说，GC日志等输出，同样包含着丰富的信息。

这里有一个相对特殊的部分，就是是堆外内存中的直接内存，前面的工具基本不适用，可以使用JDK自带的Native Memory Tracking（NMT）特性，它会从JVM本地内存分配的角度进行解读。

## 考点分析

今天选取的问题是Java内存管理相关的基础实践，对于普通的内存问题，掌握上面我给出的典型工具和方法就足够了。这个问题也可以理解为考察两个基本方面能力，第一，你是否真的理解了JVM的内部结构；第二，具体到特定内存区域，应该使用什么工具或者特性去定位，可以用什么参数调整。

对于JConsole等工具的使用细节，我在专栏里不再赘述。如果你还没有接触过，你可以参考[JConsole官方教程](#)。我这里特别推荐[Java Mission Control](#)（JMC），这是一个非常强大的工具，不仅仅能够使用JMX进行普通的管理、监控任务，还可以配合[Java Flight Recorder](#)（JFR）技术，以非常低的开销，收集和分析JVM底层的Profiling和事件等信息。目前，Oracle已经将其开源，如果你有兴趣请可以查看OpenJDK的[Mission Control](#)项目。

关于内存监控与诊断，我会在知识扩展部分结合JVM参数和特性，尽量从庞杂的概念和JVM参数选项中，梳理出相对清晰的框架：

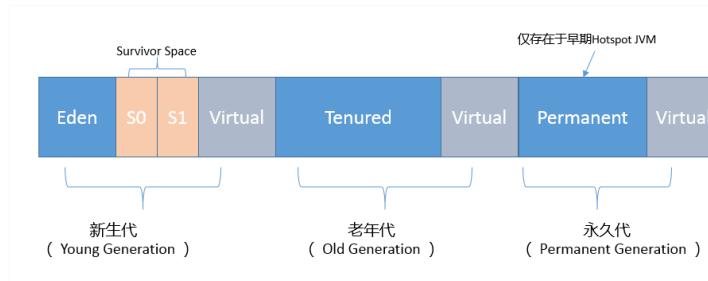
- 细化对各部分内存区域的理解，堆内结构是怎样的？如何通过参数调整？
- 堆外内存到底包括哪些部分？具体大小受哪些因素影响？

## 知识点扩展

今天的分析，我会结合相关JVM参数和工具，进行对比以加深你对内存区域更细粒度的理解。

首先，堆内部是什么结构？

对于堆内存，我在上一讲介绍了最常见的新生代和老年代的划分，其内部结构随着JVM的发展和新GC方式的引入，可以有不同角度的理解。下图就是年代视角的堆结构示意图。



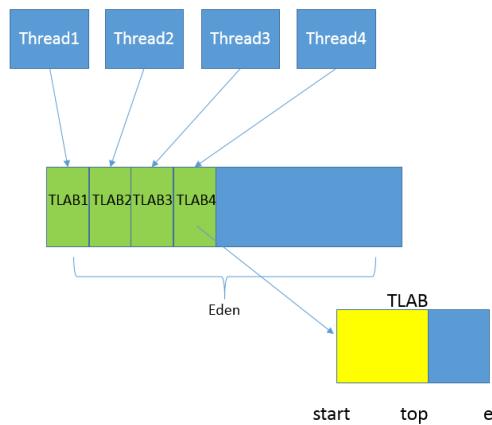
你可以看到，按照通常的GC年代方式划分，Java堆内分为：

### 1.新生代

新生代是大部分对象创建和销毁的区域，在通常的Java应用中，绝大部分对象生命周期都是很短暂的。其内部又分为Eden区域，作为对象初始分配的区域；两个Survivor，有时候也叫from, to区域，被用来放置从Minor GC中保留下来的对象。

- JVM会随意选取一个Survivor区域作为“to”，然后会在GC过程中进行区域间拷贝，也就是将Eden中存活下来的对象和from区域的对象，拷贝到这个“to”区域。这种设计主要是为了防止内存的碎片化，并进一步清理无用对象。

从内存模型而不是垃圾收集的角度，对Eden区域继续进行划分，Hotspot JVM还有一个概念叫做Thread Local Allocation Buffer (TLAB)，据我所知所有OpenJDK衍生出来的JVM都提供了TLAB的设计。这是JVM为每个线程分配的一个私有缓存区域，否则，多线程同时分配内存时，为避免操作同一地址，可能需要使用加锁等机制，进而影响分配速度。你可以参考下面的示意图。从图中可以看出，TLAB仍然在堆上，它是分配在Eden区域内的。其内部结构比较直观易懂，start, end就是起始地址，top（指针）则表示已经分配到哪里了。所以我们分配新对象，JVM就会移动top，当top和end相遇时，即表示该缓存已满，JVM会试图再从Eden里分配一块儿。



### 2.老年代

放置长生命周期的对象，通常都是从Survivor区域拷贝过来的对象。当然，也有特殊情况，我们知道普通的对象会被分配在TLAB上；如果对象较大，JVM会试图直接分配在Eden其他位置上；如果对象太大，完全无法在新生代找到足够长的连续空闲空间，JVM就会直接分配到老年代。

### 3.永久代

这部分就是早期Hotspot JVM的方法区实现方式了，储存Java类元数据、常量池、Intern字符串缓存，在JDK 8之后就不存在永久代这块儿了。

那么，我们如何利用JVM参数，直接影响堆和内部区域的大小呢？我来简单总结一下：

- 最大堆体积

`-Xmx value`

- 初始的最小堆体积

`-Xms value`

- 老年代和新生代的比例

`-XX:NewRatio=value`

默认情况下，这个数值是3，意味着老年代是新生代的3倍大；换句话说，新生代是堆大小的 $1/4$ 。

- 当然，也可以不用比例的方式调整新生代的大小，直接指定下面的参数，设定具体的内存大小数值。

```
-XX:NewSize=value
```

- Eden和Survivor的大小是按照比例设置的，如果SurvivorRatio是8，那么Survivor区域就是Eden的1/8大小，也就是新生代的1/10，因为YoungGen=Eden + 2\*Survivor，JVM参数格式是

```
-XX:SurvivorRatio=value
```

- TLAB当然也可以调整，JVM实现了复杂的适应策略，如果你有兴趣可以参考这篇[说明](#)。

不知道你有没有注意到，我在年代视角的堆结构示意图也就是第一张图中，还标记出了Virtual区域，这是块儿什么区域呢？

在JVM内部，如果Xms小于Xmx，堆的大小并不会直接扩展到其上限，也就是说保留的空间 (reserved) 大于实际能够使用的空间 (committed)。当内存需求不断增长的时候，JVM会逐渐扩展新生代等区域的大小，所以Virtual区域代表的就是暂时不可用 (uncommitted) 的空间。

第二，分析完堆内空间，我们一起来看看JVM堆外内存到底包括什么？

在JMC或JConsole的内存管理界面，会统计部分非堆内存，但提供的信息相对有限，下图就是JMC活动内存池的截图。

活动内存池						
池名称	类型	已用	最大值	使用情况	已用峰值	最大值峰值
G1 Old Gen	HEAP	0 B	3.97 GiB	0 %	0 B	3.97 GiB
G1 Survivor Space	HEAP	0 B			0 B	
G1 Eden Space	HEAP	50 MiB			50 MiB	
Metaspace	NON_HEAP	13.5 MiB	13.5 MiB		13.5 MiB	
CodeHeap 'profiled nmethods'	NON_HEAP	3.44 MiB	117 MiB	2.94 %	3.44 MiB	117 MiB
CodeHeap 'non-nmethods'	NON_HEAP	1.23 MiB	5.56 MiB	22.1 %	1.28 MiB	5.56 MiB
Compressed Class Space	NON_HEAP	1.41 MiB	1 GiB	0.137 %	1.41 MiB	1 GiB
CodeHeap 'non-profiled nmethods'	NON_HEAP	1.29 MiB	117 MiB	1.1 %	1.29 MiB	117 MiB

接下来我会依赖NMT特性对JVM进行分析，它所提供的详细分类信息，非常有助于理解JVM内部实现。

首先来做些准备工作，开启NMT并选择summary模式。

```
-XX:NativeMemoryTracking=summary
```

为了方便获取和对比NMT输出，选择在应用退出时打印NMT统计信息

```
-XX:+UnlockDiagnosticVMOptions -XX:+PrintNMTStatistics
```

然后，执行一个简单的在标准输出打印HelloWorld的程序，就可以得到下面的输出

```
C:\>e:\jdk-9\bin\java -XX:NativeMemoryTracking=summary -XX:+UnlockDiagnosticVMOptions -XX:+PrintNMTStatistics HelloWorld
Hello World!
Native Memory Tracking:
Total: reserved=5707663KB, committed=356683KB
  - Java Heap <reserved=4167680KB, committed=262144KB>
    <nmap: reserved=4167680KB, committed=262144KB>

  - Class <reserved=1056893KB, committed=4989KB>
    <classes #54>
    <malloc:125KB #115>
    <nmap: reserved=1056768KB, committed=4864KB>

  - Thread <reserved=24676KB, committed=24676KB>
    <thread #25>
    <stack: reserved=24576KB, committed=24576KB>
    <malloc:22KB #133>
    <arena=28KB #48>

  - Code <reserved=247788KB, committed=7592KB>
    <malloc:44KB #48>
    <nmap: reserved=247744KB, committed=7548KB>

  - GC <reserved=193897KB, committed=53237KB>
    <malloc:9653KB #1794>
    <nmap: reserved=188736KB, committed=43584KB>

  - Compiler <reserved=124KB, committed=134KB>
    <malloc:3KB #43>
    <arena=131KB #3>

  - Internal <reserved=657KB, committed=657KB>
    <malloc:593KB #1538>
    <nmap: reserved=64KB, committed=64KB>

  - Symbol <reserved=1299KB, committed=1999KB>
    <malloc:123KB #1322>
    <arena=75KB #1>

Native Memory Tracking <reserved=125KB, committed=125KB>
  <malloc:5KB #59>
  <tracking overhead=120KB>

Arena Chunk <reserved=1007KB, committed=1007KB>
  <malloc:1007KB>

Logging <reserved=3KB, committed=3KB>
  <malloc:3KB #136>
```

我来仔细分析一下，NMT所表征的JVM本地内存使用：

- 第一部分非常明显是Java堆，我已经分析过使用什么参数调整，不再赘述。
- 第二部分是Class内存占用，它所统计的就是Java类元数据所占用的空间，JVM可以通过类似下面的参数调整其大小：

```
-XX:MaxMetaspaceSize=value
```

对于本例，因为HelloWorld没有什么用户类库，所以其内存占用主要是启动类加载器（Bootstrap）加载的核心类库。你可以使用下面的小技巧，调整启动类加载器元数据区，这主要是为了对比以加深理解，也许只有在hack JDK时才有实际意义。

```
-XX:InitialBootClassLoaderMetaspaceSize=30720
```

- 下面是Thread，这里既包括Java线程，如程序主线程、Cleaner线程等，也包括GC等本地线程。你有没有注意到，即使是一个HelloWorld程序，这个线程数量竟然还有25。似乎有很多浪费，设想我们要用Java作为Serverless运行时，每个function是非常短暂的，如何降低线程数量呢？
- 如果你充分理解了专栏讲解的内容，对JVM内部有了充分理解，思路就很清晰了：
- JDK 9默认GC是C1，虽然它在较大堆场景表现良好，但本身就会比传统的Parallel GC或者Serial GC之类复杂太多，所以要么降低其并行线程数目，要么直接切换GC类型；  
JIT编译默认是开启了TieredCompilation的，将其关闭，那么JIT也会变得简单，相应本地线程也会减少。

我们来对比一下，这是默认参数情况的输出：

```
Thread <reserved=24676KB, committed=24676KB>
  <thread #25>
    <stack: reserved=24576KB, committed=24576KB>
    <nalloc=22KB #123>
    <arena=28KB #49>
```

下面是替换了默认GC，并关闭TieredCompilation的命令行

```
C:\>jdk-9\bin\java -XX:NativeMemoryTracking=summary
-XX:+UnlockDiagnosticVMOptions -XX:+PrintNMTStatistics -XX:-TieredCompilation
-XX:+UseParallelGC HelloWorld
```

得到的统计信息如下，线程数目从25降到了17，消耗的内存也下降了大概1/3。

```
Thread <reserved=16452KB, committed=16452KB>
  <thread #17>
    <stack: reserved=16384KB, committed=16384KB>
    <nalloc=49KB #86>
    <arena=19KB #32>
```

- 接下来是Code统计信息，显然这是CodeCache相关内存，也就是JIT compiler存储编译热点方法等信息的地方，JVM提供了一系列参数可以限制其初始值和最大值，例如：

```
-XX:InitialCodeCacheSize=value
```

```
-XX:ReservedCodeCacheSize=value
```

你可以设置下列JVM参数，也可以只设置其中一个，进一步判断不同参数对CodeCache大小的影响。

`-XX:-TieredCompilation -XX:+UseParallelGC -XX:InitialCodeCacheSize=4096`

```
Code <reserved=49562KB, committed=614KB>
  <nalloc=26KB #313>
  <nmap: reserved=49536KB, committed=588KB>
```

很明显，CodeCache空间下降非常大，这是因为我们关闭了复杂的TieredCompilation，而且还限制了其初始大小。

- 下面就是GC部分了，就像我前面介绍的，G1等垃圾收集器其本身的设施和数据结构就非常复杂和庞大，例如Remembered Set通常都会占用20%~30%的堆空间。如果我把GC明确修改为相对简单的Serial GC，会有什么效果呢？

使用命令：

```
-XX:+UseSerialGC
```

```
Thread <reserved=12340KB, committed=12340KB>
  <thread #13>
    <stack: reserved=12288KB, committed=12288KB>
    <nalloc=3KB #62>
    <arena=14KB #24>

Code <reserved=49562KB, committed=614KB>
  <nalloc=26KB #313>
  <nmap: reserved=49536KB, committed=588KB>

GC <reserved=13639KB, committed=911KB>
  <nalloc=7KB #78>
  <nmap: reserved=13632KB, committed=904KB>
```

可见，不仅总线程数大大降低（25 → 13），而且GC设施本身的内存开销就少了非常多。据我所知，AWS Lambda中Java运行时就是使用的Serial GC，可以大大降低单个function的启动和运行开销。

- Compiler部分，就是JIT的开销，显然关闭TieredCompilation会降低内存使用。

- 其他一些部分占比都非常低，通常也不会出现内存使用问题，请参考[官方文档](#)。唯一的例外就是Internal（JDK 11以后在Other部分）部分，其统计信息包含着Direct Buffer的直接内存，这其实是堆外内存中比较敏感的部分，很多堆外内存OOM就发生在这一部分。请参考专栏第12讲的处理步骤。原则上Direct Buffer是不推荐频繁创建或销毁的，如果你怀疑直接内存区域有问题，通常可以通过类似Instrument构造函数等手段，排查可能的问题。

JVM内部结构就介绍到这里，主要目的是为了加深理解，很多方面只有在定制或调优JVM运行时才能真正涉及，随着微服务和Serverless等技术的兴起，JDK确实存在着为新特征的工作负载进行定制的需求。

今天我结合JVM参数和特性，系统地分析了JVM堆内和堆外内存结构，相信你一定对JVM内存结构有了比较深入的了解，在定制Java运行时或者处理OOM等问题的时候，思路也会更

加清晰。JVM问题千奇百怪，如果你能快速将问题缩小，大致就能清楚问题可能出在哪里，例如如果定位到问题可能是堆内存泄漏，往往就已经有非常清晰的[思路和工具](#)可以去解决了。

### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，如果用程序的方式而不是工具，对Java内存使用进行监控，有哪些技术可以做到？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“[请朋友读](#)”，把今天的题目分享给好友，或许你能帮到他。



Loading...

今天阿里面试官问了我一个问题，我想了很久没想通，希望得到解答。为什么在标记垃圾的时候，需要stop the world

2018-07-05

作者回复

对方问得有点含糊，不知道是否故意的，以cms为例，它有不同的mark：initial mark, conc mark, remark；conc时候不需要stw；其他需要短暂stw，这样引用关系才不变，另外效率也高

2018-07-06

lxm

2018-07-05

可以利用ManagementFactory对内存使用情况进行粗略评估 由于是进程中执行 数据不是很准  
作者回复

2018-07-07

是的，利用JMX MXbean公开出来的api

北溟鱼汤

2018-07-05

java.lang.Runtime类有freeMemory()、totalMemory()等方法可以获取到jvm内存情况，看了一下是本地方法。  
作者回复

2018-07-07

是的

lxm

2018-07-05

除了利用ManagementFactory获取内存cpu使用情况还有引用sigar进行监控  
作者回复

2018-07-07

是的

小卡向前冲

2018-07-11

使用javaagent可以获得对象的大小，但是引入时有点麻烦，查到的资料都说需要将代码放到jar包中，然后在启动时加上 -javaagent:jar包名。  
放不知道这个算不算。

ethan

2018-07-09

jar包发生冲突，如何定位是哪些jar包发生问题  
作者回复

2018-07-11

出错信息应该包含具体类的名字等信息；mvn依赖树

qpm

2018-07-07

班门弄斧，为老师补充一些关于Eden，两个Survivor的细节。

1、大部分对象创建都是在Eden的，除了个别大对象外。

2、Minor GC开始前，to-survivor是空的，from-survivor是由对象的。

3、Minor GC后，Eden的存活对象都copy到to-survivor中，from-survivor的存活对象也复制to-survivor中。其中所有对象的年龄+1

4、from-survivor清空，成为新的to-survivor，带有对象的to-survivor变成新的from-survivor。重复回到步骤2

这是我看这边文章也有的疑问，通过查阅资料理解的，希望可以帮助到其他同学

作者回复

2018-07-07

非常感谢，下一章有配图详解，受制于一章的篇幅限制

张玮(大圣)

当然，也有特殊情况，我们知道普通的对象会被分配在 TLAB 上；如果对象较大，JVM 会试图直接分配在 Eden 其他位置上；如果对象太大，完全无法在新生代找到足够长的连续空闲空间，JVM 就会直接分配到老年带。

这里的较大具体会分配到eden的那个位置呢，请杨兄指教下

作者回复

看具体选择，比如G1 gc，会有单独的region放大对象，甚至有可能是占有不只一个region；所以，文章是个提醒，具体还是要看自己的需要去深入

L.B.Q.Y

jmx可以做到通过代码而不是工具去监控，其实jdk安装包的工具也是对jmx的一个薄层的封装。

作者回复

是的

三木子

还有一个就是jstat，可以实时查看gc信息，这个也还是没有工具直观，

作者回复

是的，文中简单提了下

Hidden

新对象都会创建在eden 和 from 区域，当发生minor gc时 把这两个区域的存活对象复制到 to区域，然后清理eden 和 from 区域，是这样理解吧

作者回复

有点区别，新对象大多是在eden，from是minor gc活下来copy的

三木子

除了工具就是命令方式了，用过命令有vmstat，这属于linux的，主要监控cpu和内存使用情况，这里是服务器总体内存，所以这个命令不是非常直观。

作者回复

也是个办法；JMX之类内建的方式更直观一些

2018-07-07

2018-07-07

2018-07-05

2018-07-07

2018-07-05

2018-07-07

2018-07-05

2018-07-07

2018-07-05

2018-07-07









第27讲 | Java常见的垃圾收集器有哪些?  
杨晓峰  
- 00:22 / 12:25

垃圾收集机制是Java的招牌能力，极大地提高了开发效率。如今，垃圾收集几乎成为现代语言的标配，即使经过如此长时间的发展，Java的垃圾收集机制仍然在不断的演进中，不同大小的设备、不同特征的应用场景，对垃圾收集提出了新的挑战，这当然也是面试的热点。

今天我要问你的是，[Java常见的垃圾收集器有哪些？](#)

#### 典型回答

实际上，垃圾收集器（GC，Garbage Collector）是和具体JVM实现紧密相关的，不同厂商（IBM、Oracle），不同版本的JVM，提供的选择也不同。接下来，我来谈谈最主流的Oracle JDK。

- Serial GC，它是最古老的垃圾收集器，“Serial”体现在其收集工作是单线程的，并且在进行垃圾收集过程中，会进入臭名昭著的“Stop-The-World”状态。当然，其单线程设计也意味着精简的GC实现，无需维护复杂的数据结构，初始化也简单，所以一直是Client模式下JVM的默认选项。

从年代的角度，通常将其老年代实现单独称作Serial Old，它采用了标记-整理（Mark-Compact）算法，区别于新生代的复制算法。

Serial GC的对应JVM参数是：

```
-XX:+UseSerialGC
```

- ParNew GC，很明显是个新生代GC实现，它实际是Serial GC的多线程版本，最常见的应用场景是配合老年代的CMS GC工作，下面是对应参数

```
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC
```

- CMS（Concurrent Mark Sweep）GC，基于标记-清除（Mark-Sweep）算法，设计目标是尽量减少停顿时间，这一点对于Web等反应时间敏感的应用非常重要，一直到今天，仍然有很多系统使用CMS GC，但是，CMS采用的标记-清除算法，存在着内存碎片化问题，所以难以避免在长时间运行等情况下发生full GC，导致恶劣的停顿。另外，既然强调了并发（Concurrent），CMS会占用更多CPU资源，并和用户线程争抢。

- Parallel GC，在早期JDK 8等版本中，它是server模式JVM的默认GC选择，也被称作是吞吐量优先的GC。它的算法和Serial GC比较相似，尽管实现要复杂的多，其特点是新生代和老年代GC都是并行进行的，在常见的服务器环境中更加高效。

开启选项是：

```
-XX:+UseParallelGC
```

另外，Parallel GC引入了开发者友好的配置项，我们可以直接设置暂停时间或吞吐量等目标，JVM会自动进行适应性调整，例如下面参数：

```
-XX:MaxGCPauseMillis=value
-XX:GCTimeRatio=N // GC时间和用户时间比例 = 1 / (N+1)
```

- G1 GC这是一种兼顾吞吐量和停顿时间的GC实现，是Oracle JDK 9以后的默认GC选项。G1可以直观的设定停顿时间的目标，相比于CMS GC，G1未必能做到CMS在最好情况下的延时停顿，但是最差情况要好很多。

G1 GC仍然存在着年代的概念，但是其内存结构并不是简单的条带式划分，而是类似棋盘的一个个region。Region之间是复制算法，但整体上实际可看作是标记-整理（Mark-Compact）算法，可以有效地避免内存碎片，尤其是当Java堆非常大的时候，G1的优势更加明显。

G1吞吐量和停顿表现都非常不错，并且仍然在不断地完善，与此同时CMS已经在JDK 9中被标记为废弃（deprecated），所以G1 GC值得你深入掌握。

## 考点分析

今天的问题是考察你对GC的了解，GC是Java程序员的面试常见题目，但是并不是每个人都有机会或者必要对JVM、GC进行深入了解，我前面的总结是为不熟悉这部分内容的同学提供一个整体的印象。

对于垃圾收集，面试官可以循序渐进从理论、实践各种角度深入，也未必是要求面试者什么都懂。但如果你懂得原理，一定会成为面试中的加分项。在今天的讲解中，我侧重介绍比较通用、基础性的部分：

- 垃圾收集的算法有哪些？如何判断一个对象是否可以回收？
- 垃圾收集器工作的基本流程。

另外，Java一直处于非常迅速的发展之中，在最新的JDK实现中，还有多种新的GC，我会在最后补充，除了前面提到的垃圾收集器，看看还有哪些值得关注的选择。

## 知识扩展

垃圾收集的原理和基础概念

第一，自动垃圾收集的前提是清楚哪些内存可以被释放。这一点可以结合我前面对Java类加载和内存结构的分析，来思考一下。

主要就是两个方面，最主要部分就是对象实例，都是存储在堆上的；还有就是方法区中的元数据等信息，例如类型不再使用，卸载该Java类似乎是很合理的。

对于对象实例收集，主要是两种基本算法，[引用计数](#)和可达性分析。

- 引用计数算法，顾名思义，就是为对象添加一个引用计数，用于记录对象被引用的情况，如果计数为0，即表示对象可回收。这是很多语言的资源回收选择，例如因人工智能而更加火热的Python，它更是同时支持引用计数和垃圾收集机制，具体哪种最优是要看场景的。业界有大规模实践中仅保留引用计数机制，以提高吞吐量的尝试。Java并没有选择引用计数，是因为其存在一个基本的难题，也就是很难处理循环引用关系。

- 另外就是Java选择的可达性分析，Java的各种引用关系，在某种程度上，将可达性问题还进一步复杂化，具体请参考[专栏第4讲](#)，这种类型的垃圾收集通常叫作追踪性垃圾收集([Tracing Garbage Collection](#))。其原理简单来说，就是将对象及其引用关系看作一个图，选定活动的对象作为GC Roots，然后跟踪引用链条，如果一个对象和GC Roots之间不可达，也就是不存在引用链条，那么即可认为是可回收对象。JVM会把虚拟机栈和本地方法栈中正在引用的对象、静态属性引用的对象和常量，作为GC Roots。

方法区无元数据的回收比较复杂，我简单梳理一下。还记得我对类加载器的分类吧，一般来说初始化类加载器加载的类型是不会进行类卸载(unload)的，而普通的类型的卸载，往往是要求相应自定义类加载器本身被回收，所以大量使用动态类型的场合，需要防止元数据区(或者早期的永久代)不会OOM。在8u40以后的JDK中，下面参数已经是默认的：

```
-XX:+ClassUnloadingWithConcurrentMark
```

第二，常见的垃圾收集算法，我认为总体上有个了解，理解相应的原理和优缺点，就已经足够了，其主要分为三类：

- 复制(Copying)算法，我前面讲到的新世代GC，基本都是基于复制算法，过程就如[专栏上一讲](#)所介绍的，将活着的对象复制到to区域，拷贝过程中将对象顺序放置，就可以避免内存碎片化。  
这么做的代价是，既然要进行复制，既要提前预留内存空间，有一定的浪费；另外，对于G1这种分拆成为大量region的GC，复制而不是移动，意味着GC需要维护region之间对象引用关系，这个开销也不小，不管是内存占用或者时间开销。
- 标记-清除(Mark-Sweep)算法，首先进行标记工作，标识出所有要回收的对象，然后进行清除。这么做除了标记、清除过程效率有限，另外就是不可避免的出现碎片化问题，这就导致其不适合特别大的堆；否则，一旦出现Full GC，暂停时间可能根本无法接受。
- 标记-整理(Mark-Compact)，类似于标记-清除，但为避免内存碎片化，它会在清理过程中将对象移动，以确保移动后的对象占用连续的内存空间。

注意，这些只是基本的算法思路，实际GC实现过程要复杂的多，目前还在发展中的前沿GC都是复合算法，并且并行和并发兼备。

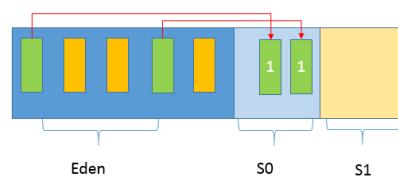
如果对这方面的算法有兴趣，可以参考一本比较有意思的书《垃圾回收的算法与实现》，虽然其内容并不是围绕Java垃圾收集，但是对通用算法讲解比较形象。

## 垃圾收集过程的理解

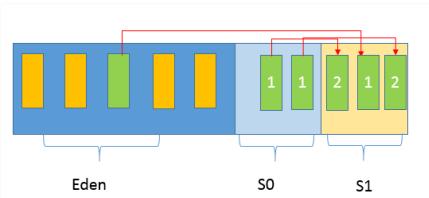
我在[专栏上一讲](#)对堆结构进行了比较详细的划分，在垃圾收集的过程，对应到Eden、Survivor、Tenured等区域会发生什么变化呢？

这实际上取决于具体的GC方式，先来熟悉一下通常的垃圾收集流程，我画了一系列示意图，希望能有助于你理解清楚这个过程。

第一，Java应用不断创建对象，通常都是分配在Eden区域，当其空间占用达到一定阈值时，触发minor GC。仍然被引用的对象(绿色方块)存活下来，被复制到JVM选择的Survivor区域，而没有被引用的对象(黄色方块)则被回收。注意，我给存活对象标记了“数字1”，这是为了表明对象的存活时间。

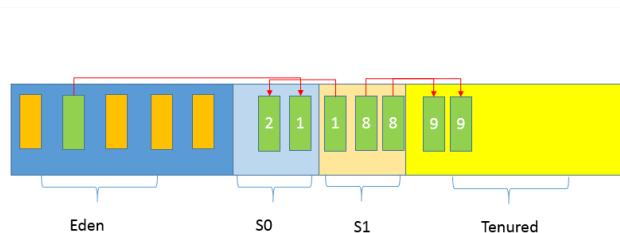


第二，经过一次Minor GC，Eden就会空闲下来，直到再次达到Minor GC触发条件，这时候，另外一个Survivor区域则会成为to区域，Eden区域的存活对象和From区域对象，都会被复制到to区域，并且存活的年龄计数会被加1。



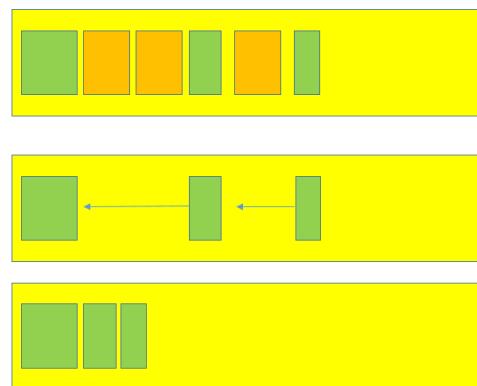
第三，类似第二步的过程会很多次，直到有对象年龄计数达到阈值，这时候就会发生所谓的晋升（Promotion）过程，如下图所示，超过阈值的对象会被晋升到老年代。这个阈值是可以通过参数指定：

-XX:MaxTenuringThreshold=<N>



后面就是老年代GC，具体取决于选择的GC选项，对应不同的算法。下面是一个简单标记-整理算法过程示意图，老年代中的无用对象被清除后，GC会将对象进行整理，以防止内存碎片化。

### 老年代 标记-压缩



通常我们把老年代GC叫作Major GC，将对整个堆进行的清理叫作Full GC，但是这个也没有那么绝对，因为不同的老年代GC算法其实表现差异很大，例如CMS，“concurrent”就体现在清理工作是与工作线程一起并发运行的。

#### GC的新发展

GC仍然处于飞速发展之中，目前的默认选项G1 GC在不断的进行改进，很多我们原来认为的缺点，例如串行的Full GC、Card Table扫描的低效等，都已经被大幅改进，例如，JDK 10以后，Full GC已经是并行运行，在很多场景下，其表现还略优于Parallel GC的并行Full GC实现。

即使是Serial GC，虽然比较古老，但是简单的设计和实现未必就是过时的，它本身的开销，不管是GC相关数据结构的开销，还是线程的开销，都是非常小的，所以随着云计算的兴起，在Serverless等新的应用场景下，Serial GC找到了新的舞台。

比较不幸的是CMS GC，因为其算法的理论缺陷等原因，虽然现在还有非常大的用户群体，但是已经被标记为废弃，如果没有组织主动承担CMS的维护，很有可能会在未来版本移除。

如果你有关注目前尚处于开发中的JDK 11，你会发现，JDK又增加了两种全新的GC方式，分别是：

- [Epsilon GC](#)，简单说就是个不做垃圾收集的GC，似乎有点奇怪，有的情况下，例如在进行性能测试的时候，可能需要明确判断GC本身产生了多大的开销，这就是其典型应用场景。

- [ZGC](#)，这是Oracle开源出来的一个超级GC实现，具备令人惊讶的扩展能力，比如支持T bytes级别的堆大小，并且保证绝大部分情况下，延迟都不会超过10 ms。虽然目前还处于实验阶段，仅支持Linux 64位的平台，但其已经表现出的能力和潜力都非常令人期待。

当然，其他厂商也提供了各种独具一格的GC实现，例如比较有名的低延迟GC，[Zing](#)和[Shenandoah](#)等，有兴趣请参考我提供的链接。

今天，作为GC系列的第一讲，我从整体上梳理了目前的主流GC实现，包括基本原理和算法，并结合我前面介绍过的内存结构，对简要的垃圾收集过程进行了介绍，希望能够对你的相关实践有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天谈了一堆的理论，思考一个实践中的问题，你通常使用什么参数去打开GC日志呢？还会额外添加哪些选项？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



公众号-Java大前端

2018-07-08

JVM提供的收集器较多，特征不一，适用于不同的业务场景：

Serial收集器：串行运行；作用于新生代，复制算法，响应速度优先；适用于单CPU环境下的client模式。  
ParNew收集器：并行运行；作用于新生代，复制算法；响应速度优先；多CPU环境Server模式下与CMS配合使用。  
Parallel Scavenge收集器：并行运行；作用于新生代，复制算法；吞吐量优先；适用于后台运算而不需要太多交互的场景。

Serial Old收集器：串行运行；作用于老年代，标记-整理算法；响应速度优先；单CPU环境下Client模式。  
Parallel Old收集器：并行运行；作用于老年代，标记-整理算法；吞吐量优先；适用于后台运算而不需要太多交互的场景。  
CMS收集器：并发运行；作用于老年代，标记-清除算法；响应速度优先；适用于互联网或B/S业务。

G1收集器：并发运行；可作用于新生代或者老年代；标记-整理算法+复制算法；响应速度优先；面向服务端应用。

Evan

2018-07-18

直接分配到老年代的对象在年轻代有空间了会移动回来吗？

作者回复

2018-07-18

不会

陈华应

2018-07-17

老师，Oracle的jvm的CMSSC，本身能够解决内存碎片化的问题吗？

作者回复

2018-07-18

我理解是目前实现不能完全避免，cms又不再发展了

张南南

2018-07-11

JDK8的话，互联网B/S项目，追求高响应和底停顿，请问是用CMS好还是G1好呢，或者有其他更好的选择

作者回复

2018-07-11

没有绝对，我建议综合考虑：  
G1理论上比cms更容易调，但你更熟悉哪个？实际用cms的挺多，也许更多经验；  
如果都不熟，先看g1能否达到你的延迟、吞吐目标；  
还有基础配置，如堆大小，比较大，比如16g以上，建议优先g1

雪狼

2018-07-09

ZGC如此强大，非常期待！

咨询大师，Java未来有没有计划让手动内存回收辅助自动内存回收以提高回收效率？既默认情况下自动内存回收完全没问题，但在极致情况下允许开发者介入甚至完全接管内存回收过程（类似与C和C++）以提高程序执行效率？

作者回复

2018-07-11

现在有一些手段影响gc，或者用直接内存再显式释放，更近一步我不知道了

null

2018-07-09

老师,请问一下,当Survivor满了而且Survivor中的对象还没有达到进老年带的年龄后怎么处理,是会增加Survivor的大小吗还是直接将Survivor中的对象放到老年带呢

作者回复

发生promotion, 放到老年带:  
maxtenuringthreshold是个上限值

三木子

用过-XX:+PrintGCDetails, 打印比较详细  
作者回复

这个jdk9已经deprecated了哦

Gotta

老师, python支持那里好像有笔误, 应该是同时支持引用计数和可达性等垃圾收集机制。其二, 标记清楚算法不适合大堆, 请问这里的大堆有什么可以量化的标准吗? 比如多大的堆才是大堆?  
作者回复

有道理, 意思是一样, 只是一些搞python的同学, 喜欢把引用计数以外的才称做gc, 大小没有那么绝对, 调优永远是针对特定场景、特定需求, 不存在一劳永逸的指标, 一般建议30G以上慎用cms, 但你看Cassandra的官方指南, 建议用在16g以下  
作者回复





## 第28讲 | 谈谈你的GC调优思路?

2018-07-10 杨晓峰



我发现，目前不少外部资料对G1的介绍大多还停留在JDK 7或更早期的实现，很多结论已经存在较大偏差，甚至一些过去的GC选项已经不再推荐使用。所以，今天我会选取新版JDK中的默认G1 GC作为重点进行详解，并且我会从调优实践的角度，分析典型场景和调优思路。下面我们一起更新下这方面的知识。

今天我要问你的是问题：[谈谈你的GC调优思路？](#)

## 典型回答

谈到调优，这一定是针对特定场景、特定目的事情。对于GC调优来说，首先就需要清楚调优的目标是什么？从性能的角度看，通常关注三个方面：内存占用（footprint）、延时（latency）和吞吐量（throughput），大多数情况下调优会侧重于其中一个或者两个方面的目标，很少有情况可以兼顾三个不同的角度。当然，除了上面通常的三个方面，也可能需要考虑其他GC相关的场景，例如，OOM也可能与不合理的GC相关参数有关；或者，应用启动速度方面的需求，GC也会是个考虑的方面。

基本的调优思路可以总结为：

- 理解应用需求和问题，确定调优目标。假设，我们开发了一个应用服务，但发现偶尔会出现性能抖动，出现较长的服务停顿。评估用户可接受的响应时间和业务量，将目标简化为，希望GC暂停尽量控制在200ms以内，并且保证一定标准的吞吐量。
- 掌握JVM和GC的状态，定位具体的问题，确定真的有GC调优的必要。具体有很多方法，比如，通过jstat等工具查看GC等相关状态，可以开启GC日志，或者是利用操作系统提供的诊断工具等。例如，通过追踪GC日志，就可以查找是不是GC在特定时间发生了长时间的暂停，进而导致了应用响应不及时。
- 这里需要思考，选择的GC类型是否符合我们的应用特征，如果是，具体问题表现在哪里，是Minor GC过长，还是Mixed GC等出现异常停顿情况；如果不是，考虑切换到什么类型，如CMS和G1都是更侧重于低延迟的GC选项。
- 通过分析确定具体调整的参数或者软硬件配置。
- 验证是否达到调优目标，如果达到目标，即可以考虑结束调优；否则，重复完成分析、调整、验证这个过程。

## 考点分析

今天考察的GC调优问题是JVM调优的一个基础方面，很多JVM调优需求，最终都会落实在GC调优上或者与其相关，我提供的是一个常见的思路。

真正快速定位和解决具体问题，还是需要对JVM和GC知识的掌握，以及实际调优经验的总结，有的时候甚至是源自经验积累的直觉判断。面试官可能会继续问项目中遇到的真实问题，如果你能清楚、简要地介绍其上下文，然后将诊断思路和调优实践过程表述出来，会是个很好的加分项。

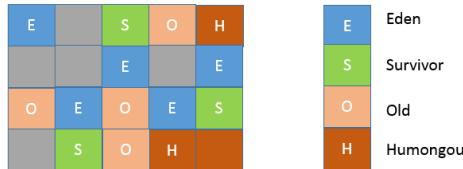
专栏虽然无法提供具体的项目经验，但是可以帮助你掌握常见的调优思路和手段，这不管是面试还是在实际工作中都是很有帮助的。另外，我会还会从下面不同角度进行补充：

- 上一讲中我已经谈到，涉及具体的GC类型，JVM的实际表现要更加复杂。目前，G1已经成为新版JDK的默认选择，所以值得你去深入理解。
- 因为G1 GC一直处在快速发展之中，我会侧重它的演进变化，尤其是行为和配置相关的变化。并且，同样是因为JVM的快速发展，即使是收集GC日志等方面也发生了较大改进，这也是为什么我在上一讲留给你的思考题是有关日志相关选项，看完讲解相信你会很惊讶。
- 从GC调优实践的角度，理解通用问题的调优思路和手段。

## 知识扩展

首先，先来整体了解一下G1 GC的内部结构和主要机制。

从内存区域的角度，G1同样存在着年代的概念，但是与我前面介绍的内存结构很不一样，其内部是类似棋盘状的一个个region组成，请参考下面的示意图。



region的大小是一致的，数值是在1M到32M字节之间的一个2的幂值数，JVM会尽量划分2048个左右、同等大小的region，这点可以从源码[heapRegionBounds.hpp](#)中看到。当然这个数字既可以手动调整，G1也会根据堆大小自动进行调整。

在G1实现中，年代是个逻辑概念，具体体现在一部分region是作为Eden，一部分作为Survivor，除了意料之中的Old region，G1会将超过region 50%大小的对象（在应用中，通常是byte或char数组）归类为Humongous对象，并放置在相应的region中。逻辑上，Humongous region算是老年代的一部分，因为复制这样的大对象是很昂贵的操作，并不适合新生代GC的复制算法。

你可以思考下region设计有什么副作用？

例如，region大小和大对象很难保证一致，这会导致空间的浪费。不知道你有没有注意到，我的示意图中有的区域是Humongous颜色，但没有用名称标记，这是为了表示，特别大的对象是可能占用超过一个region的。并且，region大小不合适，会令你在分配大对象时更难找到连续空间，这是一个长久存在的情况，请参考[OpenJDK社区的讨论](#)。这本质也可以看作是JVM的bug，尽管解决办法也非常简单，直接设置较大的region大小，参数如下：

```
-XX:G1HeapRegionSize=N, 例如16M
```

从GC算法的角度，G1选择的是复合算法，可以简化理解为：

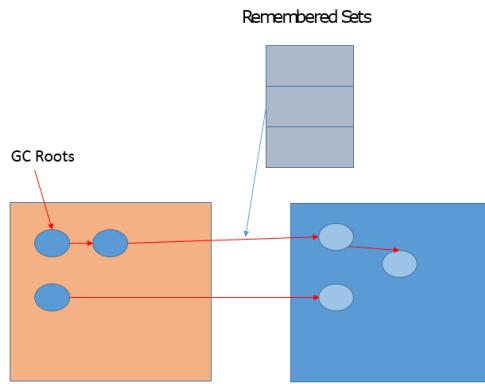
- 在新生代，G1采用的仍然是并行的复制算法，所以同样会发生Stop-The-World的暂停。
  - 在老年代，大部分情况下都是并发标记，而整理（Compact）则是和新生代GC时捎带进行，并且不是整体性的整理，而是增量进行的。
- 我在[上一讲](#)曾经介绍过，习惯上人们喜欢把新生代GC（Young GC）叫作Minor GC，老年代GC叫作Major GC，区别于整体性的Full GC。但是现代GC中，这种概念已经不再准确，对于G1来说：
- Minor GC仍然存在，虽然具体过程会有区别，会涉及Remembered Set等相关处理。
  - 老年代回收，则是依靠Mixed GC。并发标记结束后，JVM就有足够的信息进行垃圾收集，Mixed GC不仅同时会清理Eden、Survivor区域，而且还会清理部分Old区域。可以通过设置下面的参数，指定触发阈值，并且设定最多被包含在一次Mixed GC中的region比例。

```
-XX:G1MixedGCLiveThresholdPercent  
-XX:G1OldCSetRegionThresholdPercent
```

从G1内部运行的角度，下面的示意图描述了G1正常运行时的状态流转变化，当然，在发生逃逸失败等情况下，就会触发Full GC。



G1相关概念非常多，有一个重点就是Remembered Set，用于记录和维护region之间对象的引用关系。为什么需要这么做呢？试想，新生代GC是复制算法，也就是说，类似对象从Eden或者Survivor到to区域的“移动”，其实是“复制”，本质上是一个新的对象。在这个过程中，必须保证老年代到新生代的跨区引用仍然有效。下面的示意图说明了相关设计。



G1的很多开销都是源自Remembered Set，例如，它通常约占用Heap大小的20%或更高，这可是非常可观的比例。并且，我们进行对象复制的时候，因为需要扫描和更改Card Table的信息，这个速度影响了复制的速度，进而影响暂停时间。

描述G1内部的资料很多，我就不重复了，如果你想了解更多内部结构和算法等，我建议参考一些具体的[入门书](#)，书籍方面我推荐Charlie Hunt等撰写的《Java Performance Companion》。

接下来，我介绍下大家可能还不了解的G1行为变化，它们在一定程度上解决了专栏其他讲中提到的部分困扰，如类型卸载不及时的问题。

- 上面提到了Humongous对象的分配和回收，这是很多内存问题的来源，Humongous region作为老年代的一部分，通常认为它会在并发标记结束后才进行回收，但是在新版G1中，Humongous对象回收采取了更加激进的策略。  
我们知道G1记录了老年代Region间对象引用，Humongous对象数量有限，所以能够快速的知道是否有老年代对象引用它。如果没有，能够阻止它被回收的唯一可能，就是新生代是否有对象引用了它，但这个信息是可以在Young GC时就知道的，所以完全可以在Young GC中就进行Humongous对象的回收，不用像其他老年代对象那样，等待并发标记结束。
- 我在[专栏第5讲](#)提到了在8u20以后字符串排重的特性，在垃圾收集过程中，G1会把新创建的字符串对象放入队列中，然后在Young GC之后，并发地（不会STW）将内部数据（char数组，JDK 9以后是byte数组）一致的字符串进行排重，也就是将其引用同一个数组。你可以使用下面参数激活：

```
-XX:+UseStringDeduplication
```

注意，这种排重虽然可以节省不少内存空间，但这种并发操作会占用一些CPU资源，也会导致Young GC稍微变慢。

- 类型卸载是个长期困扰一些Java应用的问题，在[专栏第25讲](#)中，我介绍了一个类只有当加载它的自定义类加载器被回收后，才能被卸载。元数据区替换了永久代之后有所改善，但还是可能出现问题。

G1的类型卸载有什么改进吗？很多资料中都谈到，G1只有在发生Full GC时才进行类型卸载，但这显然不是我们想要的。你可以加上下面的参数查看类型卸载：

```
-XX:+TraceClassUnloading
```

幸好现代的G1已经不是如此了，8u40以后，G1增加并默认开启下面的选项：

```
-XX:+ClassUnloadingWithConcurrentMark
```

也就是说，在并发标记阶段结束后，JVM即进行类型卸载。

- 我们知道老年代对象回收，基本要等待并发标记结束。这意味着，如果并发标记结束不及时，导致堆已满，但老年代空间还没完成回收，就会触发Full GC，所以触发并发标记的时机很重要。早期的G1调优中，通常会设置下面参数，但是很难给出一个普适的数值，往往要根据实际运行结果调整

```
-XX:InitiatingHeapOccupancyPercent
```

在JDK 9之后的G1实现中，这种调整需求会少很多，因为JVM只会将该参数作为初始值，会在运行时进行采样，获取统计数据，然后据此动态调整并发标记启动时机。对应的JVM参数如下，默认已经开启：

```
-XX:+G1UseAdaptiveIOP
```

- 在现有的资料中，大多指出G1的Full GC是最劲的单线程串行GC。其实，如果采用的是最新的JDK，你会发现Full GC也是并行进行的了，在通用场景中的表现还优于Parallel GC的Full GC实现。

当然，还有很多其他的改变，比如更快的Card Table扫描等，这里不再展开介绍，因为它们并不带来行为的变化，基本不影响调优选择。

前面介绍了G1的内部机制，并且穿插了部分调优建议，下面从整体上给出一些调优的建议。

首先，建议尽量升级到较新的JDK版本，从上面介绍的改进就可以看到，很多人常常讨论的问题，其实升级JDK就可以解决了。

第二，掌握GC调优信息收集途径。掌握尽量全面、详细、准确的信息，是各种调优的基础，不仅仅是GC调优。我们来看看打开GC日志，这似乎是很简单的事情，可是你确定真的掌握了吗？

除了常用的两个选项,

```
-XX:+PrintGCDetails  
-XX:+PrintGCDateStamps
```

还有一些非常有用的日志选项，很多特定问题的诊断都是要依赖这些选项：

```
-XX:+PrintAdaptiveSizePolicy // 打印G1 Ergonomics相关信息
```

我们知道GC内部一些行为是适应性的触发的，利用PrintAdaptiveSizePolicy，我们就可以知道为什么JVM做出了可能我们不希望发生的一些动作。例如，G1调优的一个基本建议就是避免进行大量的Humongous对象分配，如果Ergonomics信息说明发生了这一点，那么就可以考虑要么增大堆的大小，要么直接将region大小提高。

如果是怀疑出现引用清理不及时的情况，则可以打开下面选项，掌握到底是哪里出现了堆积。

```
-XX:+PrintReferenceGC
```

另外，建议开启选项下面的选项进行并行引用处理。

```
-XX:+ParallelRefProcEnabled
```

需要注意的一点是，JDK 9中JVM和GC日志机构进行了重构，其实我前面提到的PrintGCDetails已经被标记为废弃，而PrintGCDateStamps已经被移除，指定它会导致JVM无法启动。可以使用下面的命令查询新的配置参数。

```
java -Xlog:help
```

最后，来看一些通用实践，理解了我前面介绍的内部结构和机制，很多结论就一目了然了，例如：

- 如果发现Young GC非常耗时，这很可能就是因为新生代太大了，我们可以考虑减小新生代的最小比例。

```
-XX:G1NewSizePercent
```

降低其最大值同样对降低Young GC延迟有帮助。

```
-XX:G1MaxNewSizePercent
```

如果我们直接为G1设置较小的延迟目标值，也会起到减小新生代的效果，虽然会影响吞吐量。

- 如果是Mixed GC延迟较长，我们应该怎么做呢？

还记得前面说的，部分Old region会被包含进Mixed GC，减少一次处理的region个数，就是个直接的选择之一。

我在上面已经介绍了G1OldCSetRegionThresholdPercent控制其最大值，还可以利用下面参数提高Mixed GC的个数，当前默认值是8，Mixed GC数量增多，意味着每次被包含的region减少。

```
-XX:G1MixedGCountTarget
```

今天的内容算是抛砖引玉，更多内容你可以参考[G1调优指南](#)等，远不是几句话可以囊括的。需要注意的是，也要避免过度调优，G1对大堆非常友好，其运行机制也需要浪费一定的空间，有时候稍微多给堆一些空间，比进行苛刻的调优更加实用。

今天我梳理了基本的GC调优思路，并对G1内部结构以及最新的行为变化进行了详解。总的来说，G1的调优相对简单、直观，因为可以直接设定暂停时间等目标，并且其内部引入了各种智能的自适应机制，希望这一切的努力，能够让你在日常应用开发时更加高效。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，定位Full GC发生的原因，有哪些方式？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



涛哥迷妹

感谢了这么晚还在回复,点赞了

2018-07-18

涛哥迷妹

如果dump堆太大,我觉得可以先通过jmap -heap看下是哪个区占用特别多,再看下对内存占用前20到30的大对象,然后结合jstat gcusage 查看下引起Gc原因,目前想到这些思路请问还有更好的工具? jconsole jdb远程连接查看占用?

2018-07-18

涛哥迷妹

FullGc可通过Gc日志或者添加fullgc前后堆dump 查看引起fullgc原因 CPU飙升可以看看jstack  
作者回复

2018-07-17

不错思路,如果堆太大dump不现实呢;除了gc日志,还有没有其他工具?

2018-07-18

涛哥迷妹

请问Remembered Set和cardtable关系是怎么样的?他们之间是如何协作一起完成g1 gc的?  
作者回复

2018-07-16

我理解card table是remembered set的一种实现

2018-07-18

涛哥迷妹

Remembered Set和cardtable的关系是什么?他们什么时候使用能说明下吗

2018-07-16

涛哥迷妹

Remembered Set和cardtable 是什么关系什么时候使用能说明下这个过程吗

2018-07-16

三口先生

jmap指令加不加live,分析是否是内存泄漏或者是请求的内存处理不过来等原因。  
作者回复

2018-07-11

嗯,不过,加live会触发full gc吧

2018-07-11

潇洒的毅小峰

老师要快点更新啊感觉到26更不完,校招马上开始了呢,学的不亦乐乎

2018-07-11

作者回复

汗

小刚

老师讲的很到位,看得不是很明白,得多看几遍。

2018-07-11









## 第29讲 | Java内存模型中的happen-before是什么？

2018-07-12 杨晓峰



第29讲 | Java内存模型中的happen-before是什么?  
杨晓峰  
- 00:19 / 10:17

Java语言在设计之初就引入了线程的概念，以充分利用现代处理器的计算能力，这既带来了强大、灵活的多线程机制，也带来了线程安全等令人混淆的问题，而Java内存模型（Java Memory Model，JMM）为我们提供了一个在纷乱之中达成一致的指导准则。

今天我要问你的是，[Java内存模型中的happen-before是什么？](#)

## 典型回答

Happen-before关系，是Java内存模型中保证多线程操作可见性的机制，也是对早期语言规范中含糊的可见性概念的一个精确定义。

它的具体表现形式，包括但不限于我们直觉中的synchronized、volatile、lock操作顺序等方面，例如：

- 线程内执行的每个操作，都保证happen-before后面的操作，这就保证了基本的程序顺序规则，这是开发者在书写程序时的基本约定。
- 对于volatile变量，对它的写操作，保证happen-before在随后对该变量的读取操作。
- 对于一个锁的解锁操作，保证happen-before加锁操作。
- 对象构建完成，保证happen-before于finalizer的开始动作。
- 甚至是类似线程内部操作的完成，保证happen-before其他Thread.join()的线程等。

这些happen-before关系是存在着传递性的，如果满足a happen-before b和b happen-before c，那么a happen-before c也成立。

前面我一直用happen-before，而不是简单说前后，是因为它不仅仅是对执行时间的保证，也包括对内存读、写操作顺序的保证。仅仅是时钟顺序上的先后，并不能保证线程交互的可见性。

## 考点分析

今天的问题是一个常见的考察Java内存模型基本概念的问题，我前面给出的回答尽量选择了和日常开发相关的规则。

JMM是面试的热点，可以看作是深入理解Java并发编程、编译器和JVM内部机制的必要条件，但这同时也是个容易让初学者无所适从的主题。对于学习JMM，我有一些个人建议：

- 明确目的，克制住技术的诱惑。除非你是编译器或者JVM工程师，否则我建议不要一头扎进各种CPU体系结构，纠结于不同的缓存、流水线、执行单元等。这些东西虽然很酷，但其复杂性是超乎想象的，很可能增加学习难度，也未必有实践价值。
- 克制住对“秘籍”的诱惑。有些时候，某些编程方式看起来能起到特定效果，但分不清是实现差异导致的“表现”，还是“规范”要求的行为，就不要依赖于这种“表现”去编程，尽量遵循语言规范进行，这样我们的应用行为才能更加可靠、可预计。

在这一讲中，兼顾面试和编程实践，我会结合例子梳理下面两点：

- 为什么需要JMM，它试图解决什么问题？
- JMM是如何解决可见性等各种问题的？类似volatile，体现在具体用例中有什么效果？

注意，专栏中Java内存模型就是特指JSR-133中重新定义的JMM规范。在特定的上下文里，也许会与JVM（Java）内存结构等混淆，并不存在绝对的对错，但一定要清楚面试官的本意，有的面试官也会特意考察是否清楚这两种概念的区别。

## 知识扩展

为什么需要JMM，它试图解决什么问题？

Java是最早尝试提供内存模型的语言，这是简化多线程编程、保证程序可移植性的一个飞跃。早期类似C、C++等语言，并不存在内存模型的概念（C++ 11中也引入了标准内存模

型），其行为依赖于处理器本身的内存一致性模型，但不同的处理器可能差异很大，所以一段C++程序在处理器A上运行正常，并不能保证其在处理器B上也是一致的。

即使如此，最初的Java语言规范仍然是存在着缺陷的，当时的目標是，希望Java程序可以充分利用现代硬件的计算能力，同时保持“书写一次，到处执行”的能力。

但是，显然问题的复杂度被低估了，随着Java被运行在越来越多的平台上，人们发现，过于泛泛的内存模型定义，存在很多模棱两可之处，对synchronized或volatile等，类似指令重排序时的行为，并没有提供清晰规范。这里说的指令重排序，既可以是编译器优化行为，也可能是源自于现代处理器的乱序执行等。

换句话说：

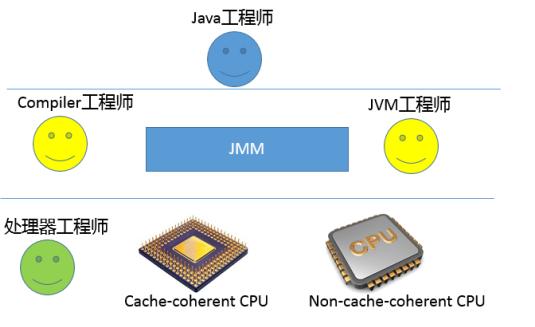
- 既不能保证一些多线程程序的正确性，例如最著名的就是双检锁（Double-Checked Locking，DCL）的失效问题，具体可以参考我在[第14讲](#)对单例模式的说明，双检锁可能导致未完整初始化的对象被访问，理论上这叫并发编程中的安全发布（Safe Publication）失败。
- 也不能保证同一段程序在不同的处理器架构上表现一致，例如有的处理器支持缓存一致性，有的不支持，各自都有自己的内存排序模型。

所以，Java迫切需要一个完善的JMM，能够让普通Java开发者和编译器、JVM工程师，能够清晰地达成共识。换句话说，可以相对简单并准确地判断出，多线程程序什么样的执行序列是符合规范的。

所以：

- 对于编译器、JVM开发者，关注点可能是如何使用类似内存屏障（Memory-Barrier）之类技术，保证执行结果符合JMM的推断。
- 对于Java应用开发者，则可能更加关注volatile、synchronized等语义，如何利用类似happen-before的规则，写出可靠的多线程应用，而不是利用一些“秘籍”去糊弄编译器、JVM。

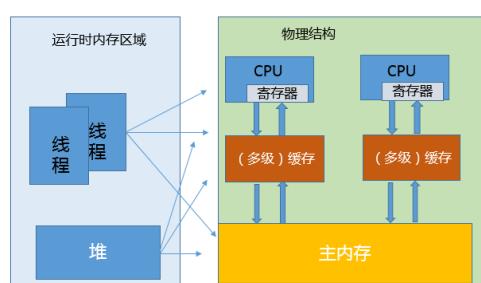
我画了一个简单的角色层次图，不同工程师分工合作，其实所处的层面是有区别的。JMM为Java工程师隔离了不同处理器内存排序的区别，这也是为什么我通常不建议过早深入处理器体系结构，某种意义上来说，这样本就违背了JMM的初衷。



JMM是怎么解决可见性等问题的呢？

在这里，我有必要简要介绍一下典型的问题场景。

我在[第25讲](#)里介绍了JVM内部的运行时数据区，但是真正程序执行，实际是要跑在具体的处理器内核上。你可以简单理解为，把本地变量等数据从内存加载到缓存、寄存器，然后运算结束写回主内存。你可以从下面示意图，看这两种模型的对应。



看上去很美好，但是当多线程共享变量时，情况就复杂了。试想，如果处理器对某个共享变量进行了修改，可能只是体现在该内核的缓存里，这是个本地状态，而运行在其他内核上的线程，可能还是加载的旧状态，这很可能导致一致性的问题。从理论上来说，多线程共享引入了复杂的数据依赖性，不管编译器、处理器怎么做重排序，都必须尊重数据依赖性的要求，否则就打破了正确性！这就是JMM所要解决的问题。

JMM内部的实现通常是依赖于所谓的内存屏障，通过禁止某些重排序的方式，提供内存可见性保证，也就是实现了各种happen-before规则。与此同时，更多复杂度在于，需要尽量确保各种编译器、各种体系结构的处理器，都能够提供一致的行为。

我以volatile为例，看看如何利用内存屏障实现JMM定义的可见性？

对于一个volatile变量：

- 对该变量的写操作之后，编译器会插入一个写屏障。
- 对该变量的读操作之前，编译器会插入一个读屏障。

内存屏障能够在类似变量读、写操作之后，保证其他线程对`volatile`变量的修改对当前线程可见，或者本地修改对其他线程提供可见性。换句话说，线程写入，写屏障会通过类似强迫刷出处理器缓存的方式，让其他线程能够拿到最新数值。

如果你对更多内存屏障的细节感兴趣，或者想了解不同体系结构的处理器模型，建议参考JSR-133[相关文档](#)，我个人认为这些都是和特定硬件相关的，内存屏障之类只是实现JMM规范的技术手段，并不是规范的要求。

从应用开发者的角度，JMM提供的可见性，体现在类似`volatile`上，具体行为是什么样呢？

我这里循序渐进的举两个例子。

首先，前几天有同学问我一个问题，请看下面的代码片段，希望达到的效果是，当`condition`被赋值为`false`时，线程A能够从循环中退出。

```
// Thread A
while (condition) {
}

// Thread B
condition = false;
```

这里就需要`condition`被定义为`volatile`变量，不然其数值变化，往往并不能被线程A感知，进而无法退出。当然，也可以在`while`中，添加能够直接或间接起到类似效果的代码。

第二，我想举Brian Goetz提供的一个经典用例，使用`volatile`作为守护对象，实现某种程度上轻量级的同步，请看代码片段：

```
Map configOptions;
char[] configText;
volatile boolean initialized = false;

// Thread A
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;

// Thread B
while (!initialized)
    sleep();
// use configOptions
```

JSR-133重新定义的JMM模型，能够保证线程B获取的`configOptions`是更新后的数值。

也就是说`volatile`变量的可见性发生了增强，能够起到守护其上下文的作用。线程A对`volatile`变量的赋值，会强制将该变量自己和当时其他变量的状态都刷出缓存，为线程B提供可见性。当然，这也是以一定的性能开销作为代价的，但毕竟带来了更加简单的多线程行为。

我们经常会说`volatile`比`synchronized`之类更加轻量，但轻量也仅仅是相对的。`volatile`的读、写仍然要比普通的读写要开销更大，所以如果你是在性能高度敏感的场景，除非你确定需要它的语义，不然慎用。

今天，我从`happen-before`关系开始，帮你理解了什么是Java内存模型。为了更方便理解，我作了简化，从不同工程师的角色划分等角度，阐述了问题的由来，以及JMM是如何通过类似内存屏障等技术实现的。最后，我以`volatile`为例，分析了可见性在多线程场景中的典型用例。

#### -课-练

关于今天我们讨论的题目你做到心中有数了吗？今天留给你的思考题是，给定一段代码，如何验证所有符合JMM执行可能？有什么工具可以辅助吗？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



公号-Java大后端

可从四个维度去理解JMM

- 1 从JVM运行时视角来看，JVM内存可分为JVM栈、本地方法栈、PC计数器、方法区、堆；其中前三区是线程所私有的，后两者则是所有线程共有的
- 2 从JVM内存功能视角来看，JVM可分为堆内存、非堆内存与其他。其中堆内存对应于上述的堆区，非堆内存对应于上述的JVM栈、本地方法栈、PC计数器、方法区；其他则对应于直接内存
- 3 从线程运行视角来看，JVM可分为主内存与线程工作内存。Java内存模型规定了所有的变量都存储在主内存中，每个线程的工作内存保存了被该线程使用到的变量，这些变量是主内存的副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量
- 4 从垃圾回收视角来看，JVM中的堆区=新生代+老年带。新生代主要用于存放新创建的对象与存活时长小的对象，新生代=E+S1+S2；老年带则用于存放存活时间长的对象

三木子

刚看完文章，在看了下《深入JAVA虚拟机》的java内存模型章节，又加深点印象。这本书真不能像小说一样读！

不瘦十斤不换名字

jmm可以从两个方面理解，第一个方面是jmm规范了一个抽象的内存结构，jmm运行时内存进行简化抽象得到了主内存和本地内存两块内存区域，在线程运行时，从主内存中加载数据到本地内存，在本地内存中完成计算后，在刷新到主内存。第二个方面是jmm可以理解我一组保证数据内存可见性和程序正确性的规则，由于这种内存模型存在很明显的数据一致性问题，再加上编译器的指令重排序和cpu乱序执行优化，使问题更加复杂了，而jmm就是通过类似内存屏障等手段保证了内存可见性问题以及在多线程环境下乱序优化和指令重排序带来的线程安全问题。从使用者的角度理解，jmm实际上平衡了java程序员对简单性的渴望和jvm工程师cpu工程师对性能的追求的平衡，面向底层时，jmm在保证正确性的同时最大限度的放宽了对指令重排序和乱序执行优化的限制，面向上层jmm通过内存屏障实现了volatile和synchronized等内存语义，使程序员可以简单方便的应用这些特性来保证程序的正确性。

不瘦十斤不换名字

jmm可以从两个方面理解，一是抽象内存结构，jmm把内存结构抽象成主内存和线程本地内存两种，在计算时，从主内存中加载数据，在本地内存计算，然后在刷新到主内存，但这种模型有明显的一致性问题，二是jmm可以理解我一组保住内存可见性及成正确性的规范，因为这种模型存在明显的致性问题，同时，由于java编译器指令重排序优化和cpu乱序执行优化的存在，使问题变得更加复杂，所以jmm基于内存屏障提供了类似sa if serial以及happens before的保障，从使用者的角度理解，jmm平衡了jvm工程师以及cpu工程师在性能上的需求和java程序员在简单性上的渴望，所以jmm在保证正确性的同时会最大限度的放宽对指令重排和乱序执行的限制，对于java程序员，jmm提供了如volatile和synchronized这样的顶层机制为程序员提供简单的编程模型。（参考老师的这篇文章及java并发编程艺术理解）

mao

java虚拟机规范中对对普通变量的赋值是不是也强制刷新到主内存

刘杰

您又说到了单例模式中的那个其他线程访问尚未初始完成对象的问题，忍不住再问下，是否可以先用一个局部变量初始化对象，再把局部变量赋值给类成员？这样可以解决吗？

作者回复

不清楚，这个我没想到规范中哪一条可以保证：  
建议看看比较权威的文章 [https://shipilev.net/blog/2014/safe-public-construction/#\\_safe\\_initialization](https://shipilev.net/blog/2014/safe-public-construction/#_safe_initialization)

3W1H

老师的例子里面的thread a,b的逻辑是在一个方法里面吗？

作者回复

和几个方法没关系

Seven4X

老师◆◆我有个问题：  
有一个全局的ConcurrentHashMap<String, Set<Foo>>  
Key是Foo的一个字符串属性  
然后有一个方法

通过Foo.getStr() 以此为key判断是否存在map 中如果不存在就创建一个set添加到map  
现在这个方法并发情况下第一次创建Set时会出现替换Set的问题，我想如何通过volatile解决？原谅学生愚钝.







第30讲 | Java程序运行在Docker等容器环境有哪些新问题?  
杨晓峰  
00:20 / 10:24

如今, Docker等容器早已不是新生事物, 正在逐步成为日常开发、部署环境的一部分。Java能否无缝地运行在容器环境, 是否符合微服务、Serverless等新的软件架构和场景, 在一定程度上也会影响未来的技术栈选择。当然, Java对Docker等容器环境的支持也在不断增强, 自然地, Java在容器场景的实践也逐渐在面试中被涉及。我希望通过专栏今天这一讲, 能够帮你做到胸有成竹。

今天我要问你的是, [Java程序运行在Docker等容器环境有哪些新问题?](#)

#### 典型回答

对于Java来说, Docker毕竟是一个较新的环境, 例如, 其内存、CPU等资源限制是通过CGroup (Control Group) 实现的, 早期的JDK版本 (8u131之前) 并不能识别这些限制, 进而会导致一些基础问题:

- 如果未配置合适的JVM堆和元数据区、直接内存等参数, Java就有可能试图使用超过容器限制的内存, 最终被容器OOM kill, 或者自身发生OOM。
- 错误判断了可获取的CPU资源, 例如, Docker限制了CPU的核数, JVM就可能设置不合适的GC并行线程数等。

从应用打包、发布等角度出发, JDK本身就比较大, 生成的镜像就更为臃肿, 当我们的镜像非常多的时候, 镜像的存储等开销就比较明显了。

如果考虑到微服务、Serverless等新的架构和场景, Java自身的大小、内存占用、启动速度, 都存在一定局限性, 因为Java早期的优化大多是针对长时间运行的大型服务器端应用。

#### 考点分析

今天的问题是个针对特定场景和知识点的问题, 我给出的回答简单总结了目前业界实践中发现的一些问题。

如果我是面试官, 针对这种问题, 如果你确实没有太多Java在Docker环境的使用经验, 直接说不知道, 也算是可以接受的, 毕竟没有人能够掌握所有知识点嘛。

但我们要清楚, 有经验的面试官, 一般不会以纯粹偏僻的知识点作为面试考察的目的, 更多是考察思考问题的思路和解决问题的方法。所以, 如果有基础的话, 可以从操作系统、容器原理、JVM内部机制、软件开发实践等角度, 展示系统性分析新问题、新场景的能力。毕竟, 变化才是世界永远的主题, 能够在新变化中找出共性与关键, 是优秀工程师的必备能力。

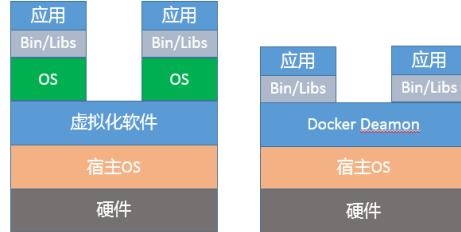
今天我会围绕下面几个方面展开:

- 面试官可能会进一步问到, 有没有想过为什么类似Docker这种容器环境, 会有点“欺负”Java? 从JVM内部机制来说, 问题出现在哪里?
- 我注意到有种论调说“没人在容器环境用Java”, 不去争论这个观点正确与否, 我会从工程实践出发, 梳理问题原因和相关解决方案, 并探讨下新场景下的最佳实践。

#### 知识扩展

首先, 我们先来搞清楚Java在容器环境的局限性来源, Docker到底有什么特别?

虽然看起来Docker之类容器和虚拟机非常相似, 例如, 它也有自己的shell, 能独立安装软件包, 运行时与其他容器互不干扰。但是, 如果深入分析你会发现, Docker并不是一种完全的虚拟化技术, 而更是一种轻量级的隔离技术。



上面的示意图，展示了Docker与虚拟机的区别。从技术角度，基于namespace，Docker为每个容器提供了单独的命名空间，对网络、PID、用户、IPC通信、文件系统挂载点等实现了隔离。对于CPU、内存、磁盘IO等计算资源，则是通过CGroup进行管理。如果你想了解更多Docker的细节，请参考相关[技术文档](#)。

Docker仅在类似Linux内核之上实现了有限的隔离和虚拟化，并不是像传统虚拟化软件那样，独立运行一个新的操作系统。如果是虚拟化的操作系统，不管是Java还是其他程序，只要调用的是同一个系统API，都可以透明地获取所需的信息，基本不需要额外的兼容性改变。

容器虽然省略了虚拟操作系统的开销，实现了轻量级的目标，但也带来了额外复杂性，它限制对于应用不是透明的，需要用户理解Docker的新行为。所以，有专家曾经说过，“幸运的是Docker没有完全隐藏底层信息，但是不幸的是Docker没有隐藏底层信息！”

对于Java平台来说，这些未隐藏的底层信息带来了很多意外的困难，主要体现在以下几个方面：

- 第一，容器环境对于计算资源的管理方式是全新的，CGroup作为相对比较新的技术，历史版本的Java显然并不能自然地理解相应的资源限制。

- 第二，namespace对于容器内的应用细节增加了一些微妙的差异，比如cmd、jstack等工具会依赖于“/proc//”下面提供的部分信息，但是Docker的设计改变了这部分信息的原有结构，我们需要对原有工具进行[修改](#)以适应这种变化。

- 从JVM运行机制的角度，为什么这些“沟通障碍”会导致OOM等问题呢？

你可以思考一下，这个问题实际是反映了JVM如何根据系统资源（内存、CPU等）情况，在启动时设置默认参数。

这就是所谓的[Ergonomics](#)机制，例如：

- JVM会大概根据检测到的内存大小，设置最初启动时的堆大小为系统内存的1/64；并将堆最大值，设置为系统内存的1/4。
- 而JVM检测到系统的CPU核数，则直接影响到了Parallel GC的并行线程数目和JIT compiler线程数目，甚至是应用中ForkJoinPool等机制的并行等级。

这些默认参数，是根据通用场景选择的初始值。但是由于容器环境的差异，Java的判断很可能是基于错误信息而做出的。这就类似，我以为住的是独栋别墅，实际上却只有一个房间是我住的。

更加严重的是，JVM的一些原有诊断或备用机制也会受到影响。为保证服务的可用性，一种常见的选择是依赖“-XX:OnOutOfMemoryError”功能，通过调用处理脚本的形式来做一些补救措施，比如自动重启服务等。但是，这种机制是基于fork实现的，当Java进程已经过度提交内存时，fork新的进程往往已经不可能正常运行了。

根据前面的总结，似乎问题非常棘手，那我们在实践中，如何解决这些问题呢？

首先，如果你能够升级到最新的JDK版本，这个问题就迎刃而解了。

- 针对这种情况，JDK 9中引入了一些实验性的参数，以方便Docker和Java“沟通”，例如针对内存限制，可以使用下面的参数设置：

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap
```

注意，这两个参数是顺序敏感的，并且只支持Linux环境。而对于CPU核心数限定，Java已经被修正为可以正确理解“-cpuset-cpus”等设置，无需单独设置参数。

- 如果你可以切换到JDK 10或者更新的版本，问题就更加简单了。Java对容器（Docker）的支持已经比较完善，默认就会自适应各种资源限制和实现差异。前面提到的实验性参数“UseCGroupMemoryLimitForHeap”已经被标记为废弃。

与此同时，新增了参数用以明确指定CPU核心的数目。

```
-XX:ActiveProcessorCount=N
```

如果实践中发现问题，也可以使用“-XX:-UseContainerSupport”，关闭Java的容器支持特性，这可以作为一种防御性机制，避免新特性破坏原有基础功能。当然，也欢迎你向OpenJDK社区反馈问题。

- 幸运的是，JDK 9中的实验性改进已经被移植到Oracle JDK 8u131之中，你可以直接下载相应[镜像](#)，并配置“UseCGroupMemoryLimitForHeap”，后续很有可能还会进一步将JDK 10中相关的增强，应用到JDK 8最新的更新中。

但是，如果我暂时只能使用老版本的JDK怎么办？

我这里有几个建议：

- 明确设置堆、元数据区内存区域大小，保证Java进程的总大小可控。

例如，我们可能在环境中，这样限制容器内存：

```
$ docker run -it --rm --name yourcontainer -p 8080:8080 -m 800M repo/your-java-container:openjdk
```

那么，就可以额外配置下面的环境变量，直接指定JVM堆大小。

```
-e JAVA_OPTIONS='-Xmx300m'
```

- 明确配置GC和JIT并行线程数目，以避免二者占用过多计算资源。

```
-XX:ParallelGCThreads  
-XX:CICompilerCount
```

除了我前面介绍的OOM等问题，在很多场景中还发现Java在Docker环境中，似乎会意外使用Swap。具体原因待查，但很有可能也是因为Ergonomics机制失效导致的，我建议配置下面参数，明确告知JVM系统内存限额。

```
-XX:MaxRAM= cat /sys/fs/cgroup/memory/memory.limit_in_bytes
```

也可以指定Docker运行参数，例如：

```
--memory-swappiness=0
```

这是受操作系统Swappiness机制影响，当内存消耗达到一定门限，操作系统会试图将不活跃的进程换出（Swap out），上面的参数有显式关闭Swap的作用。所以可以看到，Java在Docker中的使用，从操作系统、内核到JVM自身机制，需要综合运用我们所掌握的知识。

回顾我在专栏第25讲JVM内存区域的介绍，JVM内存消耗远不止包括堆，很多时候仅仅设置Xmx是不够的，MaxRAM也有助于JVM合理分配其他内存区域。如果应用需要设置更多Java启动参数，但又不确定什么数值合理，可以试试一些社区提供的工具，但要注意通用工具的局限性。

更进一步来说，对于容器镜像大小的问题，如果你使用的是JDK 9以后的版本，完全可以用Jlink工具定制最小依赖的Java运行环境，将JDK裁剪为几十M的大小，这样运行起来并不困难。

今天我从Docker环境中Java可能出现的问题开始，分析了为什么容器环境对应用并不透明，以及这种偏差干扰了JVM的相关机制。最后，我从实践出发，介绍了主要问题的解决思路，希望对你在实际开发时有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，针对我提到的微服务和Serverless等场景Java表现出的不足，有哪些方法可以改善Java的表现？

请你在留言区写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



卡斯瓦德

2018-07-14

1. 老师听说docker里面只能用open-jdk使用oracle-jdk是有法律风险的，现在还是这样么？ 2. jdk8设置了-xmx值小于docker设定的值就好，我们使用了docker-compose貌似这个只有设定内存使用上限，但是不超过这个值一般没问题。 3. 至于swap：没有遇见过，能能讲何时会出现么，好预案下。 4. 说个docker遇到相关问题是jdbc驱动，貌似mysql5.14以前的驱动对docker不友好，如果select count (\*) from table 这个值超过5000就会拿不到结果，而实际mysql-server端已经执行完毕且sleep了

作者回复

2018-07-14

1，法律问题我不知道不评价。个人建议看清事实，莫被人pr；2，是，出问题是极端情况，大部分场景并不复杂；3，具体我只注意到有人反应问题，但没有细节；回到一些常见实践，例如用G1，如果吞吐量不达标，通常调优堆大小设置为尽量大但又swap不发生，不然会影响吞吐量；4，很感谢提供这个案例，了解具体情况吗









第31讲 | 你了解Java应用开发中的注入攻击吗?  
杨晓峰  
0:00 / 09:55

安全是软件开发领域永远的主题之一，随着新技术浪潮的兴起，安全的重要性愈发凸显出来，对于金融等行业，甚至可以说安全是企业的生命线。不论是移动设备、普通PC、小型机，还是大规模分布式系统，以及各种主流操作系统，Java作为软件开发的基础平台之一，可以说是无处不在，自然也就成为安全攻击的首要目标之一。

今天我要问你的是，[你了解Java应用开发中的注入攻击吗？](#)

典型回答

注入式（Inject）攻击是一类非常常见的攻击方式。其基本特征是程序允许攻击者将不可信的动态内容注入到程序中，并将其执行，这就可能完全改变最初预计的执行过程，产生恶意效果。

下面是几种主要的注入式攻击途径，原则上提供动态执行能力的语言特性，都需要提防发生注入攻击的可能。

首先，就是最常见的SQL注入攻击。一个典型的场景就是Web系统的用户登录功能，根据用户输入的用户名和密码，我们需要去后端数据库核实信息。

假设应用逻辑是，后端程序利用界面输入动态生成类似下面的SQL，然后让JDBC执行。

```
Select * from user_info where username = "input_usr_name" and password = "input_pwd"
```

但是，如果我输入的input\_pwd是类似下面的文本，

```
" or ""="
```

那么，拼接出的SQL字符串就变成了下面的条件，OR的存在导致输入什么名字都是复合条件的。

```
Select * from user_info where username = "input_usr_name" and password = "" or "" = ""
```

这里只是举个简单的例子，它是利用了期望输入和可能输入之间的偏差。上面例子中，期望用户输入一个数值，但实际输入的则是SQL语句片段。类似场景可以利用注入的不同SQL语句，进行各种不同目的的攻击，甚至还可以加上“;delete xxx”之类语句，如果数据库权限控制不合理，攻击效果就可能是灾难性的。

第二，操作系統命令注入。Java语言提供了类似Runtime.exec(...)的API，可以用来执行特定命令，假设我们构建了一个应用，以输入文本作为参数，执行下面的命令：

```
ls -la input_file_name
```

但是如果用户输入是 “input\_file\_name;rm -rf /\*”，这就有可能出现问题了。当然，这只是个举例，Java标准类库本身进行了非常多的改进，所以类似这种编程错误，未必可以真的完成攻击，但其反映的一类场景是真实存在的。

第三，XML注入攻击。Java核心类库提供了全面的XML处理、转换等各种API，而XML自身是可以包含动态内容的，例如XPATH，如果使用不当，可能导致访问恶意内容。

还有类似LDAP等允许动态内容的协议，都是可能利用特定命令，构造注入式攻击的，包括XSS（Cross-site Scripting）攻击，虽然并不和Java直接相关，但也可能在JSP等动态页面中发生。

考点分析

今天的问题是安全领域的入门题目，我简单介绍了最常见的几种注入场景作为示例。安全本身是个非常大的主题，在面试中，面试官可能会考察安全问题，但如果不是特定安全专家

岗位，了解基础的安全实践就可以满足要求了。

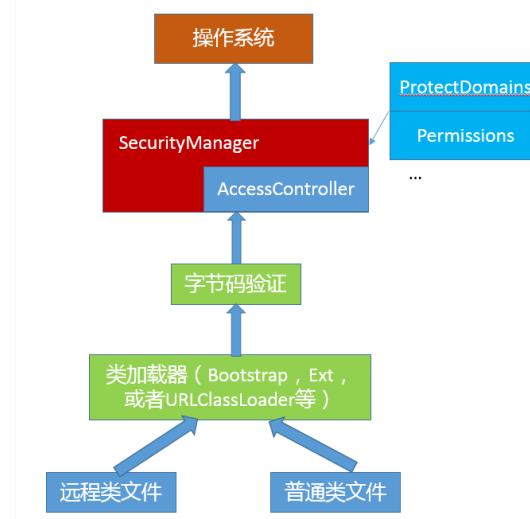
Java工程师未必都要成为安全专家，但了解基础的安全领域常识，有利于发现和规避日常开发中的风险。今天我会侧重和Java开发相关的安全内容，希望可以起到一个抛砖引玉的作用，让你对Java开发安全领域有个整体印象。

- 谈到Java应用安全，主要涉及哪些安全机制？
- 到底什么是安全漏洞？对于前面提到的SQL注入等典型攻击，我们在开发中怎么避免？

#### 知识扩展

首先，一起来看看哪些Java API和工具构成了Java安全基础。很多方面我在专栏前面的讲解中已经有所涉及，可以简单归为三个主要组成部分：

- 第一，运行时安全机制。可以简单认为，就是限制Java运行时的行为，不要做越权或者不靠谱的事情，具体来看：
- 在类加载过程中，进行字节码验证，以防止不合规的代码影响JVM运行或者载入其他恶意代码。
  - 类加载器本身也可以对代码之间进行隔离，例如，应用无法获取启动类加载器（Bootstrap Class-Loader）对象实例，不同的类加载器也可以起到容器的作用，隔离模块之间不必要的可见性等。目前，Java Applet、RMI等特性已经或逐渐退出历史舞台。类加载等机制总体上反倒在不断简化。
  - 利用SecurityManager机制和相关的组件，限制代码的运行时行为能力，其中，你可以定制policy文件和各种粒度的权限定义，限制代码的作用域和权限，例如对文件系统的操作权限，或者监听某个网络端口的权限等。我画了一个简单的示意图，对运行时安全的不同层次进行了整理。



可以看到，Java的安全模型是以代码为中心的，贯穿了从类加载，如URLClassLoader加载网络上的Java类等，到应用程序运行时权限检查等全过程。

- 另外，从原则上来说，Java的GC等资源回收管理机制，都可以看作是运行时安全的一部分，如果相应机制失效，就会导致JVM出现OOM等错误，可看作是另类的拒绝服务。

第二，Java提供的安全框架API，这是构建安全通信等应用的基础。例如：

- 加密、解密API。
- 授权、鉴权API。
- 安全通信相关的类库，比如基本HTTPS通信协议相关标准实现，如[TLS 1.3](#)；或者附属的类似证书撤销状态判断（[OSCP](#)）等协议实现。

注意，这一部分API内部实现是和厂商相关的，不同JDK厂商往往定制自己的加密算法实现。

第三，就是JDK集成的各种安全工具，例如：

- [keytool](#)，这是个强大的工具，可以管理安全场景中不可或缺的秘钥、证书等，并且可以管理Java程序使用的keystore文件。
- [jarsigner](#)，用于对jar文件进行签名或者验证。

在应用实践中，如果对安全要求非常高，建议打开SecurityManager，

```
-Djava.security.manager
```

请注意其开销，通常只要开启SecurityManager，就会导致10% ~ 15%的性能下降，在JDK 9以后，这个开销有所改善。

理解了基础Java安全机制，接下来我们一起来探讨安全漏洞（[Vulnerability](#)）。

按照传统的定义，任何可以用来绕过系统安全策略限制的程序瑕疵，都可以算作安全漏洞。具体原因可能非常多，设计或实现中的疏漏、配置错误等，任何不慎都有可能导致安全漏洞出现，例如恶意代码绕过了Java沙箱的限制，获取了特权等。如果你想了解更多安全漏洞的信息，可以从[通用安全漏洞库](#)（CVE）等途径获取，了解安全漏洞评价标准。

但是，要达到攻击的目的，未必都需要绕过权限限制。比如利用哈希碰撞发起拒绝服务攻击（DOS, Denial-Of-Service attack），常见的场景是，攻击者可以事先构造大量相同哈希值的数据，然后以JSON数据的形式发送给服务器端，服务器端在将其构建为Java对象过程中，通常以Hashtable或HashMap等形式存储，哈希碰撞将导致哈希表发生严重退化，算法复杂度可能上升一个数量级（HashMap后续进行了改进，我在[专栏第9讲](#)介绍了树化机制），进而耗费大量CPU资源。

像这种攻击方式，无关乎权限，可以看作是程序实现的瑕疵，给了攻击者以低成本进行进攻的机会。

我在开头提到的各种注入式攻击，可以有不同角度、不同层面的解决方法，例如针对SQL注入：

- 在数据输入阶段，填补期望输入和可能输入之间的鸿沟。可以进行输入校验，限定什么类型的输入是合法的，例如，不允许输入标点符号等特殊字符，或者特定结构的输入。
- 在Java应用进行数据库访问时，如果不用完全动态的SQL，而是利用PreparedStatement，可以有效防范SQL注入。不管是SQL注入，还是OS命令注入，程序利用字符串拼接生成运行逻辑都是个可能的风险点！
- 在数据库层面，如果对查询、修改等权限进行了合理限制，就可以在一定程度上避免被注入删除等高破坏性的代码。

在安全领域，有一句准则：安全倾向于“明显没有漏洞”，而不是“没有明显漏洞”。所以，为了更加安全可靠的服务，我们最好是采取整体性的安全设计和综合性的防范手段，而不是头痛医头、脚痛医脚的修修补补，更不能心存侥幸。

一个比较普通的建议是，尽量使用较新版本的JDK，并使用推荐的安全机制和标准。如果你有看过JDK release notes，例如[8u141](#)，你会发现JDK更新会修复已知的安全漏洞，并且会对安全机制等进行增强。但现实情况是，相当一部分应用还在使用很古老的不安全版本JDK进行开发，并且很多信息处理的也很随意，或者通过明文传输、存储，这些都存在暴露安全隐患的可能。

今天我首先介绍了典型的注入攻击，然后整理了Java内部的安全机制，并探讨了到底什么是安全漏洞和典型的表现形式，以及如何防范SQL注入攻击等，希望对你有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，你知道Man-In-The-Middle（MITM）攻击吗？有哪些常见的表现形式？如何防范呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

**7月19日也就是本周四晚上8点半，我会做客极客Live，做一期主题为“1小时搞定Java面试”的直播分享，我会聊聊Java面试那些事儿，感兴趣的同学不要错过哦。**

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



齐帆

2018-07-17

期待杨晓峰老师直播！

鸡肉饭饭

2018-07-17

杨老师，您好，被一个安全问题困扰许久。就是开发者是否能够通过一定的手段修改jdk中的String类，并将修改后的String类进行替换，对于这个问题，应当从哪里开始寻找答案？谢谢  
作者回复

2018-07-18

你是说，类似自己build一个jdk吗？但即使改写里面的方法也未必生效，因为有的方法是用的intrinsic的内部实现

13683815260

2018-07-17

纠结String的同学可以试一下，String应该是加载不了的。

咖啡猫口里的咖啡猫◆◆

2018-07-17

我来回答，，鸡肉饭饭的，，，数据存在immutable, mutable, 两种，java没有原生immutable支持，string如果是new就是相对意义的immutable, java基本类型和string是有高效缓存池范围，OK？

羊羊羊

2018-07-17

也不是很懂，根据自己的理解讲一下，部分可能是错误的。中间人攻击原理大概是用户在正常上网的时候，同网段的恶意用户对其进行欺骗。恶意用户向局域网广播：我是路由器，然后正常使用（电脑无防备）收到以后认为恶意用户就是路由器，然后向恶意用户发送数据包，恶意用户可以截获数据包，再向路由器发送正常用户的的数据包，路由器将返回的数据包在给恶意用户，恶意用户在给正常用户，恶意用户就形成了中间人的效果，可以向返回的数据包注入html代码，达到劫持用户网站的效果。不过现在大部分的网站都是https且双向认证，比较难获取到用户发送数据包中的账号密码。

作者回复

2018-07-18

不错，如果从Java API的角度看，也存在很多可能，即使是https，在连接没完整建立前，最初的通信并不是安全的，例如，过程中发生proxy authentication之类，其实还是http







## 第32讲 | 如何写出安全的Java代码?

2018-07-19 杨晓峰



第32讲 | 如何写出安全的Java代码?  
杨晓峰  
- 00:19 / 10:31

在上一讲中，我们已经初步接触了Java安全，今天我们将一起探讨更多Java开发中可能影响到安全的场合。很多安全问题，在特定的上下文，存在着不同的定义，尽管本质是相似或一致的，这是由于Java平台自身的特性所带来特有的问题。今天这一讲我将侧重于Java开发者的角度谈代码安全，而不是讲广义的安全风险。

今天我要问你的是问题，[如何写出安全的Java代码](#)？

## 典型回答

这个问题可能有点宽泛，我们可以用特定类型的安全风险为例，如拒绝服务（DoS）攻击，分析Java开发者需要重点考虑的点。

DoS是一种常见的网络攻击，有人也称其为“洪水攻击”。最常见的表现是，利用大量机器发送请求，将目标网站的带宽或者其他资源耗尽，导致其无法响应正常用户的请求。

我认为，从Java语言的角度，更加需要重视的是程序级别的攻击，也就是利用Java、JVM或应用程序的瑕疵，进行低成本的DoS攻击，这也是想要写出安全的Java代码所必须考虑的。例如：

- 如果使用的是早期的JDK和Applet等技术，攻击者构建合法但恶劣的程序就相对容易，例如，将其线程优先级设置为最高，做一些看起来无害但空耗资源的事情。幸运的是类似技术已经逐步退出历史舞台，在JDK 9以后，相关模块就已经被移除。
- 上一讲中提到的哈希碰撞攻击，就是个典型的例子，对方可以轻易消耗系统有限的CPU和线程资源。从这个角度思考，类似加密、解密、图形处理等计算密集型任务，都要防范被恶意滥用，以免攻击者通过直接调用或者间接触发方式，消耗系统资源。
- 利用Java构建类似上传文件或者其他接受输入的服务，需要对消耗系统内存或存储的上限有所控制，因为我们不能将系统安全依赖于用户的合理使用。其中特别注意的是涉及解压缩功能时，就需要防范Zip bomb等特定攻击。
- 另外，Java程序中需要明确释放的资源有很多种，比如文件描述符、数据库连接，甚至是再入锁，任何情况下都应该保证资源释放成功，否则即使平时能够正常运行，也可能被攻击者利用而耗尽某些资源，这也算是可能的DoS攻击来源。

所以可以看出，实现安全的Java代码，需要从功能设计到实现细节，都充分考虑可能的安全影响。

## 考点分析

关于今天的问题，以典型的DoS攻击作为切入点，将问题聚焦在Java开发中，我介绍了Java应用设计、实现的注意事项，后面还会介绍更加全面的实践。

其实安全问题实际就是软件的缺陷，软件安全并不存在一劳永逸的秘籍，既离不开设计、架构中的风险分析，也离不开编码、测试等阶段的安全实践手段。对于面试官来说，考察安全问题，除了对特定安全领域知识的考察，更多的是要看面试者的Java编程基本功和知识的积累。

所以，我会在后面会循序渐进探讨Java安全编程，这里面没有什么黑科技，只有规范的开发标准，很多安全问题其实是态度问题，取决于你是否真的认真对待它。

- 我将以一些典型的代码片段为出发点，分析一些非常容易被忽略的安全风险，并介绍安全问题频发的热点场景，如Java序列化和反序列化。
- 从软件生命周期的角度，探讨设计、开发、测试、部署等不同阶段，有哪些常见的安全策略或工具。

## 知识扩展

首先，我们一起来看一段不起眼的条件判断代码，这里可能有什么问题吗？

```
// a, b, c都是int类型的数值
if (a + b < c) {
// -
}
```

你可能会纳闷，这是再常见不过的一个条件判断了，能有什么安全隐患？

这里的隐患是数值类型需要防范溢出，否则这不仅仅可能会带来逻辑错误，在特定情况下可能导致严重的安全漏洞。

从语言特性来说，Java和JVM提供了很多基础性的改进，相比于传统的C、C++等语言，对于数组越界等处理要完善的多，原生的避免了缓冲区溢出等攻击方式，提高了软件的安全性。但这并不代表完全杜绝了问题，Java程序可能调用本地代码，也就是JNI技术，错误的数值可能导致C/C++层面的数据越界等问题，这是很危险的。

所以，上面的条件判断，需要判断其数值范围，例如，写成类似下面结构。

```
if (a < - b)
```

再来看一个例子，请看下面的一段异常处理代码：

```
try {
    // 业务代码
} catch (Exception e) {
    throw new RuntimeException(hostname + port + " doesn't response");
}
```

这段代码将敏感信息包含在异常消息中，试想，如果是一个Web应用，异常也没有良好的包装起来，很有可能就把内部信息暴露给终端客户。古人曾经告诫我们“言多必失”是很有道理的，虽然真本意不是指软件安全，但尽量少暴露信息，也是保证安全的基本原则之一。即使我们并不认为某个信息有安全风险，我的建议也是如果没有必要，不要暴露出来。

这种暴露还可能通过其他方式发生，比如某著名的编程技术网站，就被曝光过所有用户名和密码。这些信息都是明文存储，传输过程也未必进行加密，类似这种情况，暴露只是个时间早晚的问题。

对于安全标准特别高的系统，甚至可能要求敏感信息被使用后，要立即明确在内存中销毁，以免被探测；或者避免在发生core dump时，意外暴露。

第三，Java提供了序列化等创新的特性，广泛使用在远程调用等方面，但也带来了复杂的安全问题。直到今天，序列化仍然是个安全问题频发的场景。

针对序列化，通常建议：

- 敏感信息不要被序列化！在编码中，建议使用transient关键字将其保护起来。
- 反序列化中，建议在readObject中实现与对象构件过程相同的安全检查和数据检查。

另外，在JDK 9中，Java引入了过滤器机制，以保证反序列化过程中数据都要经过基本验证才可以使用。其原理是通过黑名单和白名单，限定安全或者不安全的类型，并且你可以进行定制，然后通过环境变量灵活进行配置，更加具体的使用你可以参考[ObjectInputFilter](#)。

通过前面的介绍，你可能注意到，很多安全问题都是源于非常基本的编程细节，类似`Immutable`、封装等设计，都存在着安全性的考虑。从实践的角度，让每个人都了解和掌握这些原则，有必要但并不太现实，有没有什么工程实践手段，可以帮助我们排查安全隐患呢？

#### 开发和测试阶段

在实际开发中，各种功能点五花八门，未必能考虑的全面。我建议没有必要所有都需要自己去从头实现，尽量使用广泛验证过的工具、类库，不管是来自于JDK自身，还是Apache等第三方组织，都在社区的反馈下持续地完善代码安全。

开发过程中应用代码规约标准，是避免安全问题的有效手段。我特别推荐来自孤尽的《阿里巴巴Java开发手册》，以及其配套工具，充分总结了业界在Java等领域的实践经验，将规约实践系统性地引入国内的软件开发，可以有效提高代码质量。

当然，凡事都是有代价的，规约会增加一定的开发成本，可能对迭代的节奏产生一定影响，所以对于不同阶段、不同需求的团队，可以根据自己的情况对规约进行适应性的调整。

落实到实际开发流程中，以OpenJDK团队为例，我们应用了几个不同角度的实践：

- 在早期设计阶段，就由安全专家组对新特性进行风险评估。
- 开发过程中，尤其是code review阶段，应用OpenJDK自身定制的代码规范。
- 利用多种静态分析工具如[FindBugs](#)、[Parfait](#)等，帮助早期发现潜在安全风险，并对相应问题采取零容忍态度，强制要求解决。
- 甚至OpenJDK会默认将任何（编译等）警告，都当作错误对待，并体现在CI流程中。
- 在代码check-in等关键环节，利用hook机制去调用规则检查工具，以保证不合规代码不能进入OpenJDK代码库。

关于静态分析工具的选择，我们选取的原则是“足够好”。没有什么工具能够发现所有问题，所以在保证功能的前提下，影响更大的是分析效率，换句话说是代码分析的噪音高低。不管分析有多么的完备，如果太多误报，就会导致有用信息被噪音覆盖，也不利于后续其他程序化的处理，反倒不利于排查问题。

以上这些是为了保证JDK作为基础平台的苛刻质量要求，在实际产品中，你需要斟酌具体什么程度的要求是合理的。

#### 部署阶段

JDK自身的也是个软件，难免会存在实现瑕疵，我们平时看到JDK更新的安全漏洞补丁，其实就是在修补这些漏洞。我最近还注意到，某大厂后台被曝出了使用的JDK版本存在序列化相关的漏洞。类似这种情况，大多数都是因为使用的JDK是较低版本，算是可以通过部署解决的问题。

如果是安全敏感型产品，建议关注JDK在加解密方面的[路线图](#)，同样的标准也应用于其他语言和平台，很多早期认为非常安全的算法，已经被攻破，及时地升级基础软件是安全的必要条件。

攻击和防守是对称的，只要有一个严重漏洞，对于攻击者就足够了，所以，不能对黑盒形式的部署心存侥幸，这并不能保证系统的安全，攻击者可以利用对软件设计的猜测，结合一系列手段，探测出漏洞。

今天我以DoS等典型攻击方式为例，分析了其在Java平台上的特定表现，并从更多安全编码的细节帮你体会安全问题的普遍性，最后我介绍了软件开发周期中的安全实践，希望能对你的工作有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？你在开发中遇到过Java特定的安全问题吗？是怎么解决的呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

别忘了今晚8点半我会做客“极客Live”，和你一起聊聊Java面试那些事儿。在“极客时间”App内点击“极客Live”即可加入直播，今晚我们不见不散。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



baoning

2018-07-19

期待

笑笑冰怪

2018-07-19

期待

羊羊羊

2018-07-21

安全倾向于“明显没有漏洞”，而不是“没有明显漏洞”。这段话很喜欢，老师能不能在详细的剖析下这句话。

sunmeilin

2018-07-21

错过了老师的live课，好可惜，不知道后面还有没有？

LenX

2018-07-20

在密码使用的场景中，比如用户注册/登录。

我使用 char[] 数组存储和验证密码，并在使用结束后，通过随机的字符覆盖掉 char[]。

如果使用 String 存储密码，由于它的不可变性，它的缺陷是会一直驻留在堆中，直到未来被垃圾回收。

小粉蒸

2018-07-20

请问没赶上直播的怎么办，有录音之类的吗？

Leiy

2018-07-20

老师，你好，那个c是负数，b是正数，c-b也可能溢出吧？

作者回复

2018-07-20

对，所以我强调判断数值范围，具体看情况选择

王大为

2018-07-20

谢谢老师的回复，我编译的openjdk8,netbeans7.4,按照周志明老师的JVM书籍上说的编译的。请问oracle jdk也可以编译下来断点调试吗，您那边用的什么IDE调试的啊？

互联网一两年

2018-07-19

老师，能否就spring 源码 解读一下呢？非常期待

王大为

2018-07-19

老师，您好，我编译了openjdk，导入到netbeans中，怎么打不了断点调试jdk模块，请您指导一下

作者回复

2018-07-20

netbeans...基本没什么用过，openjdk什么版本？netbeans呢？oracle官方的jdk有这个问题吗？

另外build的时候有没有加上enable-debug之类参数？

三木子

2018-07-19

最近项目在改安全问题，主要遇到的有：SQL注入，IO流没有关闭，使用不安全Random，重定向Url合法性没做检验，上传文件前后端没做文件大小类型校验。通过反射方式访问私有方法。日志包含敏感信息，没有禁止除GET,POST以外的HTTP请求等等

作者回复

2018-07-20

JDK新版中random升级很多，例如基于DRBG的实现，值得尝试

老韩

2018-07-19

hook有直接可用的或者简单修改就能用的推荐吗

作者回复

openjdk用的是自身定制的...但我理解主流代码检查工具，都可以试试定制个适合做check-in守门的版本，想清楚限定什么，一般是个子集

2018-07-20



第33讲 | 后台服务出现明显“变慢”，谈谈你的诊断思路？

2018-07-21 杨晓峰



第33讲 | 后台服务出现明显“变慢”，谈谈你的诊断思路？  
杨晓峰  
- 00:05 / 10:18

在日常工作中，应用或者系统出现性能问题往往是不可避免的，除了在有一定规模的IT企业或者专注于特定性能领域的企业，可能大多数工程师并不会成为专职的性能工程师，但是掌握基本的性能知识和技能，往往是日常工作的需要，并且也是工程师进阶的必要条件之一，能否定位和解决性能问题也是对你知识、技能和能力的检验。

今天我要问你的是，后台服务出现明显“变慢”，谈谈你的诊断思路？

#### 典型回答

首先，需要对这个问题进行更加清晰的定义：

- 服务是突然变慢还是长时间运行后观察到变慢？类似问题是否重复出现？
- “慢”的定义是什么，我能够理解是系统对其他方面的请求的反应延时变长吗？

第二，理清问题的症状，这更便于定位具体的原因，有以下一些思路：

- 问题可能来自于Java服务自身，也可能仅仅是受系统里其他服务的影响。初始判断可以先确认是否出现了意外的程序错误，例如检查应用本身的错误日志。  
对于分布式系统，很多公司都会实现更加系统的日志、性能等监控系统。一些Java诊断工具也可以用于这个诊断，例如通过JFR ([Java Flight Recorder](#))，监控应用是否大量出现了某种类型的异常。  
如果有，那么异常可能就是个突破点。  
如果没有，可以先检查系统级别的资源等情况，监控CPU、内存等资源是否被其他进程大量占用，并且这种占用是否符合系统正常运行状况。
- 监控Java服务自身，例如GC日志里面是否观察到Full GC等恶劣情况出现，或者是否Minor GC在变长等；利用jstat等工具，获取内存使用的统计信息也是个常用手段；利用jstack等工具检查是否出现死锁等。
- 如果还不能确定具体问题，对应用进行Profiling也是个办法，但因为它会对系统产生侵入性，如果不是非常必要，大多数情况下并不建议在生产系统进行。
- 定位了程序错误或者JVM配置的问题后，就可以采取相应的补救措施，然后验证是否解决，否则还需要重复上面部分过程。

#### 考点分析

今天我选择的是一个常见的并且比较贴近实际应用的性能相关问题，我提供的回答包括两部分。

- 在正面回答之前，先探讨更加精确的问题定义是什么。有时候面试官并没有表达清楚，有必要确认自己的理解正确，然后再深入回答。
- 从系统、应用的不同角度、不同层次，逐步将问题域尽量缩小，隔离出真实原因。具体步骤未必千篇一律，在处理过较多这种问题之后，经验会令你的直觉分外敏感。

大多数工程师也许并没有全面的性能问题诊断机会，如果被问到也不必过于紧张，你可以向面试官展示诊断问题的思考方式，展现自己的知识和综合运用的能力。接触到一个陌生的问题，通过沟通，能够条理清晰地将排查方案逐步确定下来，也是能力的体现。

面试官可能会针对某个角度的诊断深入询问，兼顾工作和面试的需求，我会针对下面一些方面进行介绍。目的是让你对性能分析有个整体的印象，在遇到特定领域问题时，即使不知道具体细节的工具和手段，至少也可以找到探索、查询的方向。

- 我将介绍业界常见的性能分析方法论。
- 从系统分析到JVM、应用性能分析，把握整体思路和主要工具。对于线程状态、JVM内存使用等很多方面，我在专栏前面已经陆陆续续介绍了很多，今天这一讲也可以看作是聚焦性能角度的一个小结。

如果你有兴趣进行系统性的学习，我建议参考Charlie Hunt编撰的《Java Performance》或者Scott Oaks的《Java Performance: The Definitive Guide》。另外，如果不希望出现理解偏差，最好是阅读英文版。

#### 知识扩展

首先，我们来了解一下业界最广泛的性能分析方法论。

根据系统架构不同，分布式系统和大型单体应用也存在着思路的区别，例如，分布式系统的性能瓶颈可能更加集中。传统意义上的性能调优大多是对单体应用的调优，专栏的侧重点也是如此，Charlie Hunt曾将其方法论总结为两类：

- 自上而下。从应用的顶层，逐步深入到具体的不同模块，或者更进一步的技术细节单元，找到可能的问题和解决办法。这是最常见的性能分析思路，也是大多数工程师的选择。
- 自下而上。从类似CPU这种硬件底层，判断类似Cache-Miss之类的问题和调优机会，出发点是指令级别优化。这往往是专业的性能工程师才能掌握的技能，并且需要专业工具配合，大多数是移植到新的平台上，或需要提供极致性能时才会进行。

例如，将大数据应用移植到SPARC体系结构的硬件上，需要对比和尽量释放性能潜力，但又希望尽量不改源代码。

我所给出的回答，首先是试图排除功能性错误，然后就是典型的自上而下分析思路。

第二，我们一起来看看自上而下分析中，各个阶段的常见工具和思路。需要注意的是，具体的工具在不同的操作系统上可能区别非常大。

系统性能分析中，CPU、内存和IO是主要关注项。

对于CPU，如果是常见的Linux，可以先用top命令查看负载状况。下图是我截取的一个状态。

```
top - 21:54:51 up 66 days, 12:57, 2 users, load average: 1.00, 1.01, 1.05
Tasks: 160 total, 2 running, 158 sleeping, 0 stopped, 0 zombie
%Cpu(s): 89.4 us, 10.6 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 4047236 total, 3993748 used, 53488 free, 594500 buffers
KiB Swap: 15304700 total, 673860 used, 14630840 free. 1810128 cached Mem

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  2182 root      20   0 3093180 370220 7740 S 99.7 9.1 27630:00 java
  1 root      20   0 119892 4928 3140 S 0.0 0.1 1:18.88 systemd
```

可以看到，其平均负载（load average）的三个值（分别是1分钟、5分钟、15分钟）非常低，并且暂时看并没有升高迹象。如果这些数值非常高（例如，超过50%、60%），并且短期平均值高于长期平均值，则表明负载很重；如果还有升高的趋势，那么就要非常警惕了。

进一步的排查有很多思路，例如，我在专栏第18讲曾经问过，怎么找到最耗费CPU的Java线程，简要介绍步骤：

- 利用top命令获取相应pid，“-H”代表thread模式，你可以配合grep命令更精准定位。

```
top -H
```

• 然后转换成为16进制。

```
printf "%x" your_pid
```

• 最后利用jstack获取的线程栈，对比相应的ID即可。

当然，还有更加通用的诊断方向，利用vmstat之类，查看上下文切换的数量，比如下面就是指定时间间隔为1，收集10次。

```
vmstat -1 -10
```

输出如下：

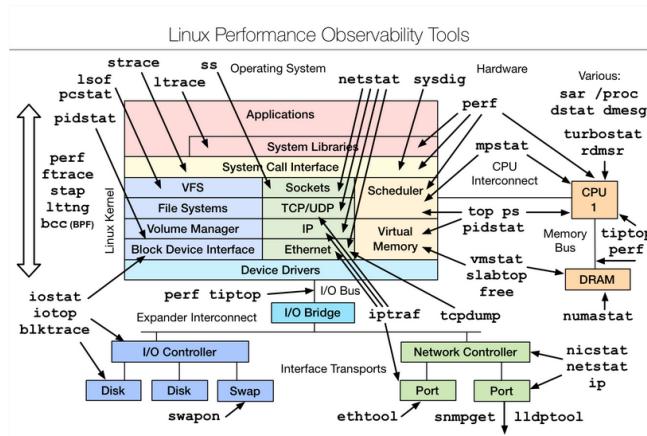
```
procs --memory-- --swap-- --io-- --system-- --cpu--
  b swapd free  buff cache si so bi bo in cs us sy id wa st
  0 0 40564 40564 595524 1813860 0 1 9 30 6 3 41 11 48 0 0
  0 0 673860 40568 595524 1813860 0 0 0 0 440 463 89 11 0 0 0
  0 0 673860 40568 595524 1813860 0 0 0 0 435 680 90 10 0 0 0
  0 0 673860 40568 595524 1813860 0 0 0 0 384 395 92 8 0 0 0
  0 0 673860 40568 595524 1813860 0 0 0 0 391 409 88 12 0 0 0
  0 0 673860 40568 595524 1813860 0 0 0 0 374 390 90 10 0 0 0
  0 0 673860 40568 595524 1813860 0 0 0 0 382 406 88 12 0 0 0
  0 0 673860 40568 595524 1813860 0 0 0 0 379 424 92 8 0 0 0
```

如果每秒上下文（cs，[context switch](#)）切换很高，并且比系统中断很多（in，[system interrupt](#)），就表明很有可能是因为不合理的多线程调度所导致。当然还需要利用[pidstat](#)等手段，进行更加具体的定位，我就不再进一步展开了。

除了CPU，内存和IO是重要的注意事项，比如：

- 利用free之类查看内存使用。
- 或者，进一步判断swap使用情况，top命令输出中Virt作为虚拟内存使用量，就是物理内存（Res）和swap求和，所以可以反推swap使用。显然，JVM是不希望发生大量的swap使用的。
- 对于IO问题，既可能发生在磁盘IO，也可能是网络IO。例如，利用iostat等命令有助于判断磁盘的健康状况。我曾经帮助诊断过Java服务部署在国内的某云厂商机器上，其原因就是IO表现较差，拖累了整体性能，解决办法就是申请替换了机器。

讲到这里，如果你对系统性能非常感兴趣，我建议参考[Brendan Gregg](#)提供的完整图谱，我所介绍的只能算是九牛一毛。但我还是建议尽量结合实际需求，免得迷失在其中。



对于JVM层面的性能分析，我们已经介绍过非常多了：

- 利用JMC、JConsole等工具进行运行时监控。
- 利用各种工具，在运行时进行堆转储分析，或者获取各种角度的统计数据（如jstat -gcutil分析GC、内存分带等）。
- GC日志等手段，诊断Full GC、Minor GC，或者引用堆积等。

这里并不存在放之四海而皆准的办法，具体问题可能非常不同，还要看你是否能充分利用这些工具，从种种迹象之中，逐步判断出问题所在。

对于应用Profiling，简单来说就是利用一些侵入性的手段，收集程序运行时的细节，以定位性能问题瓶颈。所谓的细节，就是例如内存的使用情况，最频繁调用的方法是什么，或者上下文切换的情况等。

我在前面给出的典型回答里提到，一般不建议生产系统进行Profiling，大多数是在性能测试阶段进行。但是，当生产系统确实存在这种需求时，也不是没有选择。我建议使用JFR配合JMC来做Profiling，因为它是从Hotspot JVM内部收集底层信息，并经过了大量优化，性能开销非常低，通常是低于2%的；并且如此强大的工具，也已经被Oracle开源出来！

所以，JFR/JMC完全具备了生产系统Profiling的能力，目前也确实在真正大规模部署的云产品上使用过相关技术，快速地定位了问题。

它的使用也非常方便，你不需要重新启动系统或者提前增加配置。例如，你可以在运行时启动JFR记录，并将这段时间的信息写入文件：

```
Jcmd <pid> JFR.start duration=120s filename=myrecording.jfr
```

然后，使用JMC打开“.jfr”文件就可以进行分析了，方法、异常、线程、IO等应有尽有，其功能非常强大。如果你想了解更多细节，可以参考相关[指南](#)。

今天我从一个典型性能问题出发，从症状表现到具体的系统分析、JVM分析，系统性地整理了常见性能分析的思路；并且在知识扩展部分，从方法论和实际操作的角度，让你将理论和实际结合，相信一定可以对你有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，Profiling工具获取数据的主要方式有哪些？各有什么优缺点。

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



Yano

一直以来看老师的专栏没有留言过，今天特意来留言。我看了13讲（当时只出到13讲）面试就轻松通过了~老师每一篇文章，都让我非常收益，赞一个~！

2018-07-21

One day

之前看到目录上还有讲spring和数据库等等，后面还会讲吧！

2018-07-21

Seven4X

百度搜索 profiling是什么

2018-07-21





第34讲 | 有人说“Lambda能让Java程序慢30倍”，你怎么看？

2018-07-24 杨晓峰



在上一讲中，我介绍了Java性能问题分析的一些基本思路。但在实际工作中，我们不能仅仅等待性能出现问题再去试图解决，而是需要定量的、可对比的方法，去评估Java应用性能，来判断其是否能够符合业务支撑目标。今天这一讲，我会介绍从Java开发者角度，如何从代码级别判断应用的性能表现，重点理解最广泛使用的基准测试（Benchmark）。

今天我要问你的是问题：[有人说“Lambda能让Java程序慢30倍”，你怎么看？](#)

为了让你清楚地了解这个背景，请参考下面的代码片段。在实际运行中，基于Lambda/Stream的版本（lambdaMaxInteger），比传统的for-each版本（forEachLoopMaxInteger）慢很多。

```
// 一个大的ArrayList, 内部是随机的整形数据
volatile List<Integer> integers = ...;

// 基准测试1
public int forEachLoopMaxInteger() {
    int max = Integer.MIN_VALUE;
    for (Integer n : integers) {
        max = Integer.max(max, n);
    }
    return max;
}

// 基准测试2
public int lambdaMaxInteger() {
    return integers.stream().reduce(Integer.MIN_VALUE, (a, b) -> Integer.max(a, b));
}
```

#### 典型回答

我认为，“Lambda能让Java程序慢30倍”这个争论实际反映了几个方面：

第一，基准测试是一个非常有效的通用手段，让我们以直观、量化的方式，判断程序在特定条件下的性能表现。

第二，基准测试必须明确定义自身的范围和目标，否则很可能产生误导的结果。前面代码片段本身的逻辑就有瑕疵，更多的开销是源于自动装箱、拆箱（auto-boxing/unboxing），而不是源自Lambda和Stream，所以得出的初始结论是没有说服力的。

第三，虽然Lambda/Stream为Java提供了强大的函数式编程能力，但是也需要正视其局限性：

- 一般来说，我们可以认为Lambda/Stream提供了与传统方式接近对等的性能，但是如果对于性能非常敏感，就不能完全忽视它在特定场景的性能差异了，例如：初始化的开销。Lambda并不算是语法糖，而是一种新的工作机制，在首次调用时，JVM需要为其构建CallSite实例。这意味着，如果Java应用启动过程引入了很多Lambda语句，会导致启动过程变慢。其实现特点决定了JVM对它的优化可能与传统方式存在差异。

- 增加了程序诊断等方面的复杂性，程序栈要复杂很多，Fluent风格本身也不算是对于调试非常好的结构，并且在可检查异常的处理方面也存在着局限性等。

#### 考点分析

今天的题目是源自于一篇有争议的[文章](#)，原文后来更正为“如果Stream使用不当，会让你的代码慢5倍”。针对这个问题我给出的回答，并没有纠结于所谓的“快”与“慢”，而是从工程实践的角度指出了基准测试本身存在的问题，以及Lambda自身的局限性。

从知识点的角度，这个问题考察了我在[专栏第7讲](#)中介绍过的自动装箱/拆箱机制对性能的影响，并且考察了Java 8中引入的Lambda特性的相关知识。除了这些知识点，面试官还能更加深入探讨如何用基准测试之类的方法，将含糊的观点变成可验证的结论。

对于Java语言的很多特性，经常有很多似是而非的“秘籍”，我们有必要去伪存真，以定量、定性的方式探究真相，探讨更加易于推广的实践。找到结论的能力，比结论本身更重要，因此在今天这一讲中，我们来探讨一下：

- 基准测试的基础要素，以及如何利用主流框架构建简单的基准测试。
- 进一步分析，针对保证基准测试的有效性，如何避免偏离测试目的，如何保证基准测试的正确性。

#### 知识扩展

首先，我们先来整体了解一下基准测试的主要目的和特征，专栏里我就不重复那些[书面的定义](#)了。

性能往往是特定情景下的评价，泛泛地说性能“好”或者“快”，往往是具有误导性的。通过引入基准测试，我们可以定义性能对比的明确条件、具体的指标，进而保证得到定量的、可重复的对比数据，这是工程中的实际需要。

不同的基准测试其具体内容和范围也存在很大的不同。如果是专业的性能工程师，更加熟悉的可能是类似[SPEC](#)提供的工业标准的系统级测试；而对于大多数Java开发者，更熟悉的是范围相对较小、关注点更加细节的微基准测试（Micro-Benchmark）。我在文章开头提的问题，就是典型的微基准测试，也是我今天的侧重点。

什么时候需要开发微基准测试呢？

我认为，当需要对一个大型软件的某小部分的性能进行评估时，就可以考虑微基准测试。换句话说，微基准测试大多是API级别的验证，或者与其他简单用例场景的对比，例如：

- 你在开发共享类库，为其他模块提供某种服务的API等。
- 你的API对于性能，如延迟、吞吐量有着严格的要求，例如，实现了定制的HTTP客户端API，需要明确它对HTTP服务器进行大量GET请求时的吞吐能力，或者需要对比其他API，保证至少对等甚至更高的性能标准。

所以微基准测试更是偏基础、底层平台开发者的需求，当然，也是那些追求极致性能的前沿工程师的最爱。

如何构建自己的微基准测试，选择什么样的框架比较好？

目前应用最为广泛的框架之一就是[JMH](#)，OpenJDK自身也大量地使用JMH进行性能对比，如果你是做Java API级别的性能对比，JMH往往是你的首选。

JMH是由Hotspot JVM团队专家开发的，除了支持完整的基准测试过程，包括预热、运行、统计和报告等，还支持Java和其他JVM语言。更重要的是，它针对Hotspot JVM提供了各种特性，以保证基准测试的正确性，整体准确性大大优于其他框架，并且，JMH还提供了用近乎白盒的方式进行Profiling等工作的能力。

使用JMH也非常简单，你可以直接将其依赖加入Maven工程，如下图：

```
<dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
        <version>${jmh.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-generator-moxy</artifactId>
        <version>${jmh.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

也可以，利用类似下面的命令，直接生成一个Maven项目。

```
$ mvn archetype:generate \
-DinteractiveMode=false \
-DarchetypeGroupId=org.openjdk.jmh \
-DarchetypeArtifactId=jmh-java-benchmark-archetype \
-DgroupId=org.sample \
-DartifactId=te8 \
-Dversion=1.0
```

JMH利用注解（Annotation），定义具体的测试方法，以及基准测试的详细配置。例如，至少要加上@Benchmark以标识它是个基准测试方法，而BenchmarkMode则指定了基准测试模式，例如下面例子指定了吞吐量（Throughput）模式，还可以根据需要指定平均时间（AverageTime）等其他模式。

```
@Benchmark
@BenchmarkMode(Mode.Throughput)
public void te8Method() {
    // Put your benchmark code here.
}
```

当我们实现了具体的测试后，就可以利用下面的Maven命令构建。

```
mvn clean install
```

运行基准测试与运行不同的Java应用没有明显区别。

```
java -jar target/benchmarks.jar
```

更加具体的上手步骤，请参考相关[文章](#)。JMH处处透着浓浓的工程师味道，并没有纠结于完善的文档，而是提供了非常棒的[GitHub仓库](#)，所以你需要习惯于直接从代码中学习。

[指南](#)[样例代码](#)

如何保证微基准测试的正确性，有哪些坑需要规避？

首先，构建微基准测试，需要从JVM层面理解代码，尤其是具体的性能开销，不管是CPU还是内存分配。这有两个方面的考虑。第一，需要保证我们写出的基准测试符合测试目的，确保验证的是我们要覆盖的功能点，这一讲的问题就是个典型例子；第二，通常对于微基准测试，我们通常希望代码片段确实是有限的，例如，执行时间如果需要很多毫秒(ms)，甚至是秒级，那么这个有效性就要存疑了，也不便于诊断问题所在。

更加重要的是，由于微基准测试基本上都是体量较小的API层面测试，最大的威胁来自于过度“聪明”的JVM！Brain Goetz曾经很早就指出了微基准测试中的[典型问题](#)。

由于我们执行的是非常有限的代码片段，必须要保证JVM优化过程不影响原始测试目的，下面几个方面需要重点关注：

- 保证代码经过了足够并且合适的预热。我在[专栏第1讲](#)中提到过，默认情况下，在server模式下，JIT会在一段代码执行10000次后，将其编译为本地代码，client模式则是1500次以后，我们需要排除代码执行初期的噪音，保证真正采样到的统计数据符合其稳定运行状态。  
通常建议使用下面的参数来判断预热工作到底是经过了多久。

```
-XX:+PrintCompilation
```

我这里建议另外加上一个参数，否则JVM将默认开启后台编译，也就是在其他线程进行，可能导致输出的信息有些混淆。

```
-Xbatch
```

与此同时，也要保证预热阶段的代码路径和采集阶段的代码路径是一致的，并且可以观察PrintCompilation输出是否在后期运行中仍然有零星的编译语句出现。

- 防止JVM进行无效代码消除（Dead Code Elimination），例如下面的代码片段中，由于我们并没有使用计算结果mul，那么JVM就可能直接判断无效代码，根本就不执行它。

```
public void testMethod() {
    int left = 10;
    int right = 100;
    int mul = left * right;
}
```

如果你发现代码统计数据发生了数量级程度上的提高，需要警惕是否出现了无效代码消除的问题。

解决办法也很直接，尽量保证方法有返回值，而不是void方法，或者使用JMH提供的[BlackHole](#)设施，在方法中添加下面语句。

```
public void testMethod(Blackhole blackhole) {
    // ...
    blackhole.consume(mul);
}
```

- 防止发生常量折叠（Constant Folding）。JVM如果发现计算过程是依赖于常量或者事实上的常量，就可能会直接计算其结果，所以基准测试并不能真实反映代码执行的性能。JMH提供了[State](#)机制来解决这个问题，将本地变量修改为State对象信息，请参考下面示例。

```
@State(Scope.Thread)
public static class MyState {
    public int left = 10;
    public int right = 100;
}

public void testMethod(MyState state, Blackhole blackhole) {
    int left = state.left;
    int right = state.right;
    int mul = left * right;
    blackhole.consume(mul);
}
```

- 另外JMH还会对State对象进行额外的处理，以尽量消除共享（[False Sharing](#)）的影响，标记@State，JMH会自动进行补齐。

- 如果你希望确定方法内联（Inlining）对性能的影响，可以考虑打开下面的选项。

```
-XX:+PrintInlining
```

从上面的总结，可以看出来微基准测试是一个需要高度了解Java、JVM底层机制的技术，是个非常好的深入理解程序背后效果的工具，但是也反映了我们需要审慎对待微基准测试，不被可能的假象蒙蔽。

我今天介绍的内容是相对常见并易于把握的，对于微基准测试，GC等基层机制同样会影响其统计数据。我在前面提到，微基准测试通常希望执行时间和内存分配速率都控制在有限范围内，而在这个过程中发生GC，很可能导致数据出现偏差，所以Serial GC是个值得考虑的选项。另外，JDK 11引入了[Epsilon GC](#)，可以考虑使用这种什么也不做的GC方式，从最大可能性去排除相关影响。

今天我从一个争议性的程序开始，探讨了如何从开发者角度而不是性能工程师角度，利用（微）基准测试验证你在性能上的判断，并且介绍了其基础构建方式和需要重点规避的风险点。

**一课一练**

关于今天我们讨论的题目你做到心中有数了吗？我们在项目中需要评估系统的容量，以计划和保证其业务支撑能力，谈谈你的思路是怎么样的？常用手段有哪些？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



明翼

2018-07-24

杨老师，我今天去面试的时候，java基础知识答的还可以，就是面试官扩展着问，问到spring和springcloud框架时就有些懵了，请问我现在应该如何学习，spring一直都是懂得不是很多，仅限于会用，杨老师能推荐一些相关的学习资料吗

作者回复

我后面会有篇Spring相关，但毕竟能覆盖很有限。我自己的理解角度和业务开发可能也不太一样。  
如果是需要整体上的对比和介绍，类似知乎、博客之类有整理完善的，请查找一下；  
但是，真正的深入还是需要系统性学习，相关书籍，阅读源码，上手实践

Geek\_7120fb

2018-07-25

专栏写的好，像流水和故事。大概技术像作者的开发人员可以毕业了

yushing

2018-07-25

请问杨老师，无效代码消除后，mul的计算不执行了，那left和right也就没有使用了，是不是left和right的赋值语句也会被判断为无效代码不执行了呢？







## 第35讲 | JVM优化Java代码时都做了什么？

2018-07-26 杨晓峰 &amp; 郑雨迪



我在专栏上一讲介绍了微基准测试和相关的注意事项，其核心就是避免JVM运行中对Java代码的优化导致失真。所以，系统地理解Java代码运行过程，有利于在实践中进行更进一步的调优。

今天我要问你的是，[JVM优化Java代码时都做了什么？](#)

与以往我来给出典型回答的方式不同，今天我邀请了隔壁专栏[《深入拆解Java虚拟机》](#)的作者，同样是来自Oracle的郑雨迪博士，让他以JVM专家的身份去思考并回答这个问题。

来自JVM专栏作者郑雨迪博士的回答

JVM在对代码执行的优化可分为运行时（runtime）优化和即时编译器（JIT）优化。运行时优化主要是解释执行和动态编译通用的一些机制，比如说锁机制（如偏斜锁）、内存分配机制（如TLAB）等。除此之外，还有一些专门用于优化解释执行效率的，比如说模版解释器、内联缓存（inline cache，用于优化虚方法调用的动态绑定）。

JVM的即时编译器优化是指将热点代码以方法为单位转换成机器码，直接运行在底层硬件之上。它采用了多种优化方式，包括静态编译器可以使用的如方法内联、逃逸分析，也包括基于程序运行profile的投机性优化（speculative/optimistic optimization）。这个怎么理解呢？比如我有一条 instanceof 指令，在编译之前的执行过程中，测试对象的类一直是同一个，那么即时编译器会假设编译之后的执行过程中还是会是这个类，并且根据这个类直接返回 instanceof 的结果。如果出现了其他类，那么就抛弃这段编译后的机器码，并且切换回解释执行。

当然，JVM的优化方式仅仅作用在运行应用代码的时候。如果应用代码本身阻塞了，比如说并发时等待另一线程的结果，这就不在JVM的优化范畴啦。

**考点分析**

感谢郑雨迪博士从JVM的角度给出的回答。今天这道面试题在专栏里有不少同学问我，也是会在面试时被面试官刨根问底的一个知识点，郑博士的回答已经非常全面和深入啦。

大多数Java工程师并不是JVM工程师，知识点总归是要落地的，面试官很有可能会从实践的角度探讨，例如，如何在生产实践中，与JIT等JVM模块进行交互，落实到如何真正进行实际调优。

在今天这一讲，我会从Java工程师日常的角度出发，侧重于：

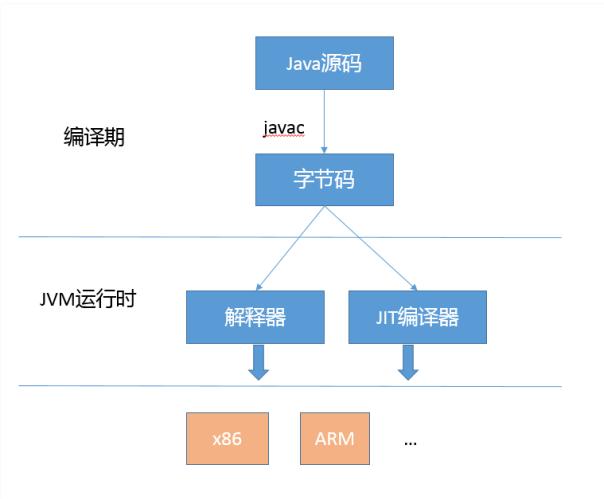
- 从整体去了解Java代码编译、执行的过程，目的是对基本机制和流程有个直观的认识，以保证能够理解调优选择背后的逻辑。
- 从生产系统调优的角度，谈谈将JIT的知识落实到实际工作中的可能思路。这里包括两部分：如何收集JIT相关的信息，以及具体的调优手段。

**知识扩展**

首先，我们从整体的角度来看看Java代码的整个生命周期，你可以参考我提供的示意图。

- 从整体去了解Java代码编译、执行的过程，目的是对基本机制和流程有个直观的认识，以保证能够理解调优选择背后的逻辑。

- 从生产系统调优的角度，谈谈将JIT的知识落实到实际工作中的可能思路。这里包括两部分：如何收集JIT相关的信息，以及具体的调优手段。

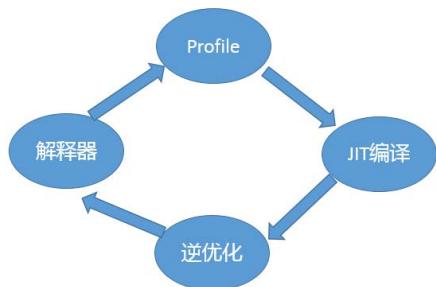


我在[专栏第1讲](#)就已经提到过，Java通过引入字节码这种中间表达方式，屏蔽了不同硬件的差异，由JVM负责完成从字节码到机器码的转化。

通常所说的编译期，是指javac等编译器或者相关API等将源码转换为字节码的过程，这个阶段也会进行少量类似常量折叠之类的优化，只要利用反编译工具，就可以直接查看细节。

javac优化与JVM内部优化也存在关联，毕竟它负责了字节码的生成。例如，Java 9中的字符串拼接，会被javac替换成对StringConcatFactory的调用，进而为JVM进行字符串拼接优化提供了统一的入口。在实际场景中，还可以通过不同的[编译选项](#)来干预这个过程。

今天我要讲的重点是JVM运行时的优化，在通常情况下，编译器和解释器是共同起作用的，具体流程可以参考下面的示意图。



JVM会根据统计信息，动态决定什么方法被编译，什么方法解释执行，即使是已经编译过的代码，也可能在不同的运行阶段不再是热点，JVM有必要将这种代码从Code Cache中移除出去，毕竟其大小是有限的。

就如郑博士所回答的，解释器和编译器也会进行一些通用优化，例如：

- 锁优化，你可以参考我在[专栏第16讲](#)提供的解释器运行时的源码分析。
- Intrinsic机制，或者叫作内建方法，就是针对特别重要的基础方法，JDK团队直接提供定制的实现，利用汇编或者编译器的中间表达方式编写，然后JVM会直接在运行时进行替换。

这么做的理由有很多，例如，不同体系结构的CPU在指令等层面存在着差异，定制才能充分发挥出硬件的能力。我们日常使用的典型字符串操作、数组拷贝等基础方法，Hotspot都提供了内建实现。

而即时编译器（JIT），则是更多优化工作的承担者。JIT对Java编译的基本单元是整个方法，通过对方法调用的计数统计，甄别出热点方法，编译为本地代码。另外一个优化场景，则是最针对所谓热点循环代码，利用通常说的线上替换技术（OSR, On-Stack Replacement，更加细节请参考[R太的文章](#)），如果方法本身的调用频度还不够编译标准，但是内部有大量的循环之类，则还是会有进一步优化的价值。

从理论上来看，JIT可以看作就是基于两个计数器实现，方法计数器和回边计数器提供给JVM统计数据，以定位到热点代码。实际中的JIT机制要复杂得多，郑博士提到了[逃逸分析](#)、[循环展开](#)、方法内联等，包括前面提到的Intrinsic等通用机制同样会在JIT阶段发生。

第二，有哪些手段可以探查这些优化的具体发生情况呢？

专栏中已经陆续介绍了一些，我来简单总结一下并补充部分细节。

- 打印编译发生的细节。

```
-XX:+PrintCompilation
```

- 输出更多编译的细节。

```
-XX:UnlockDiagnosticVMOptions -XX:+LogCompilation -XX:LogFile=<your_file_path>
```

JVM会生成一个xml形式的文件，另外，LogFile选项是可选的，不指定则会输出到

```
hotspot_pid<pid>.log
```

具体格式可以参考Ben Evans提供的[JitWatch](#)工具和[分析指南](#)。

```
<thread_logfile thread='9960' filename='C:\Users\xiaofeya\AppData\Local\Temp\hs_c9960_pid3936.log' />
<writer thread='12944' />
<task_queued compile_id='1' method='java.lang.StringUTF16 getChar ([B)C' bytes='60' count='97408' iicount='97408' stamp='0.328'
comment='tiered' hot_count='97408' />
<task_queued compile_id='2' method='java.lang.StringLatin1 hashCode ([B)I' bytes='42' count='116' backedge_count='2048' iicount='116'
level='3' stamp='0.328' comment='tiered' hot_count='116' />
<task_queued compile_id='3' method='java.lang.String isLatin1 ()Z' bytes='19' count='1408' iicount='1408' level='3' stamp='0.328'
comment='tiered' hot_count='1408' />
```

- 打印内联的发生，可利用下面的诊断选项，也需要明确解锁。

```
-XX:+PrintInlining
```

- 如何知晓Code Cache的使用状态呢？

很多工具都已经提供了具体的统计信息，比如，JMC、JConsole之类，我也介绍过使用NMT监控其使用。

第三，我们作为应用开发者，有哪些可以触手可及的调优角度和手段呢？

- 调整热点代码门限值

我曾经介绍过JIT的默认门限，server模式默认10000次，client是1500次。门限大小也存在着调优的可能，可以使用下面的参数调整；与此同时，该参数还可以变相起到降低预热时间的作用。

```
-XX:CompileThreshold=N
```

很多人可能会产生疑问，既然是热点，不是早晚会达到门限次数吗？这个还真未必，因为JVM会周期性的对计数的数值进行衰减操作，导致调用计数器永远不能达到门限值，除了可以利用CompileThreshold适当调整大小，还有一个办法就是关闭计数器衰减。

```
-XX:-UseCounterDecay
```

如果你是利用debug版本的JDK，还可以利用下面的参数进行试验，但是生产版本是不支持这个选项的。

```
-XX:CounterHalfLifetime
```

- 调整Code Cache大小

我们知道JIT编译的代码是存储在Code Cache中的，需要注意的是Code Cache是存在大小限制的，而且不会动态调整。这意味着，如果Code Cache太小，可能只有一小部分代码可以被JIT编译，其他的代码则没有选择，只能解释执行。所以，一个潜在的调优点就是调整其大小限制。

```
-XX:ReservedCodeCacheSize=<SIZE>
```

当然，也可以调整其初始大小。

```
-XX:InitialCodeCacheSize=<SIZE>
```

注意，在相对较新版本的Java中，由于分层编译（Tiered-Compilation）的存在，Code Cache的空间需求大大增加，其本身默认大小也被提高了。

- 调整编译器线程数，或者选择适当的编译器模式

JVM的编译器线程数目与我们选择的模式有关，选择client模式默认只有一个编译线程，而server模式则默认是两个，如果是当前最普遍的分层编译模式，则会根据CPU内核数目计算C1和C2的数值，你可以通过下面的参数指定的编译器线程数。

```
-XX:C1CompilerCount=N
```

在强劲的多处理器环境中，增大编译线程数，可能更加充分的利用CPU资源，让预热等过程更加快速；但是，反之也可能导致编译线程争抢过多资源，尤其是当系统非常繁忙时。例如，系统部署了多个Java应用实例的时候，那么减小编译线程数目，则是可以考虑的。

生产实践中，也有人推荐在服务器上关闭分层编译，直接使用server编译器，虽然会导致稍慢的预热速度，但是可能在特定工作负载上会有微小的吞吐量提高。

- 其他一些相对边界比较混淆的所谓“优化”

比如，减少进入安全点。严格说，它远不只是发生在动态编译的时候，GC阶段发生的更加频繁，你可以利用下面选项诊断安全点的影响。

```
-XX:+PrintSafepointStatistics -XX:+PrintGCAfterStoppedTime
```

注意，在JDK 9之后，PrintGCApplicationStoppedTime已经被移除了，你需要使用“-Xlog:safepoint”之类方式来指定。

很多优化阶段都可能和安全点相关，例如：

- 在JIT过程中，逆优化等场景会需要插入安全点。
- 常规的锁优化阶段也可能发生。比如，偏斜锁的设计目的是为了避免无竞争时的同步开销，但是当真的发生竞争时，撤销偏斜锁会触发安全点，是很重的操作。所以，在并发场景中偏斜锁的价值其实是被质疑的，经常会明确建议关闭偏斜锁。

-XX:+UseBiasedLocking

主要的优化手段就介绍到这里，这些方法都是普通Java开发者就可以利用的。如果你想对JVM优化手段有更深入的了解，建议你订阅JVM专家郑雨迪博士的专栏。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？请思考一个问题，如何程序化验证final关键字是否会影响性能？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

[点击下方图片进入JVM专栏](#)



BY

2018-07-26

profile是啥意思。。。。







周末福利 | 一份Java工程师必读书单  
2018-07-28 杨晓峰



你好，我是杨晓峰。今天这期周末福利，[我整理了几本自己在学习和使用Java时用过的参考书](#)，把它们分享与你。在专栏里，有不少同学留言让我推荐一些参考书。另外，我认为，书是是个好的系统化知识来源，但更多提高还是来源于实践、阅读源码、技术交流等，毕竟书籍也很难完全跟上技术和架构的变革，另外我也尽量缩减了书单的长度。

关于夯实Java编程基础，我推荐Bruce Eckel的《Java编程思想》（Thinking in Java），非常有名的经典书籍。这本书的特点是，不仅仅介绍Java编程的基础知识点，也会思考编程中的各种选择与判断，包括穿插设计模式的使用，作者从理论到实践意义从不同的角度进行探讨，构建稳固的Java编程知识体系。



当然这本书也有不足之处，毕竟每个人的基础不太一样，如果你完全没有Java编程基础，也可以考虑其他的参考书，例如《Java核心技术》。

另外，这两本书的部分内容已经多少有点过时了，尤其是《Java编程思想》。例如，目前很少会需要学习Java桌面图形类库等，较新的语法和API当然也没有包含，我的建议是尽管忽略过时内容，适当补充Java新技术的学习。

提到经典，自然也不少了《Effective Java》，这本书的英文第三版已经在国内上市，涵盖了Java 7到Java 9的各种新特性。严格来说，这本书不算是一本基础书籍，但当你有一定基础后，还是非常建议通读一下的。关于这本书的阅读，我的建议是边学习边回顾，在吸收书中的经验时，多去想想自己在实际应用中是如何处理的。虽然《Effective Java》的具体章节可能是从某个点出发，但可以说都是对Java、JVM、面向对象等各种知识的综合运用，对于设计和实现高质量的代码很有帮助。



《Head First设计模式》这本书就不用我再费笔墨去介绍了吧，能把设计模式用这种轻松的形式展现本身已经十分不易了，章节之间的联系让你可以反复加深印象，加上生动的表达方式和丰富的习题更容易沉浸其中。



这本书非常适合对面向对象和设计模式基础有限的同学。设计模式不是银弹，实践中也莫要为了模式而模式，掌握典型模式，能够举一反三就很好了，就当作是程序员之间沟通的“方言”。

谈过了Java基础，接下来聊聊并发和虚拟机的参考书。

《Java并发编程实战》，作者全是响当当的人物，比如Brian Goetz，我多次在专栏里引用他的观点，众多强力作者也保证了书的质量。抛开作者光环，这本书的内容全部建立在理论之上，先讲道理再谈实践，可以真正让你知其然也知其所以然。这本书更加侧重并发编程中有哪些问题，如何来深刻地理解和定义问题，如何利用可靠的手段指导工程实践，并没有过分纠结于并发类库的源码层面。



这本书我的学习建议是，尽量充分利用其中提供的样例代码，结合自己的业务实践去深入学习，毕竟这本书的内容有些偏理论，可能并不适合你快速掌握所谓并发“核心”技术。

关于JVM的学习，不用我多说了吧，看过专栏的同学肯定都知道，我经常推荐周志明的《深入理解Java虚拟机》，可以说是国内最好的JVM书籍之一。



我这里并没有单独推荐类似GC算法等书籍，它们对于大多数Java工程师的价值也许有限。

关于性能优化，我推荐Charlie Hunt和Binu John所著的《Java性能优化权威指南》(Java Performance)，也是我上次在直播时向大家推荐的。Java之父James Gosling，也力荐这本参考书。



但这本书也存在着不足，里面过于偏重Solaris等商业操作系统和相关工具，我建议你在阅读的时候，尽量体会其思路和原理，更加侧重于Linux等主流开放平台。

还有一些如开源软件和互联网架构相关的图书可以作为扩展阅读，你可以参考下面这几本。

《Spring实战》



Spring

Spring

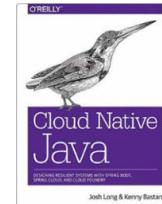
可以说 相关框架已经成为业务开发的事实标准，系统性地掌握 框架的设计和实践，是必需的技能之一。

#### 《Netty实战》



Netty在性能、可扩展性等方面的突出表现，已经得到充分验证，作为基础的通信框架，已经广泛应用在各种互联网架构、游戏等领域，甚至可以说，如果没有仔细分析过Netty，对NIO等方面的理解很可能还在很肤浅的阶段。

#### 《Cloud Native Java》



Java应用程序架构处于飞快的演进之中，微服务等新的架构应用越来越广泛，即使未必是使用Spring Boot、Spring Cloud等框架，但是系统的学习其设计思想和实践技术，绝对是有必要的。当然如果你在实践中使用Dubbo等框架，也可以选择相关书籍。

前沿领域的变化非常快，很多风靡一时的开源软件，在实践中逐渐被证明存在各种弊端，或者厂商停止维护。所以这部分的学习，我建议不要盲目追新，最好是关注于分布式设计中的问题和解决的思路，做到触类旁通，并且注重书籍之外的学习渠道。

下面两本并不算是Java书籍，但Java程序员进阶少不了对互联网主流架构的学习，了解分布式架构、缓存、消息中间件等令人眼花缭乱的技术，对于有志于成为架构师的Java工程师来说非常有帮助。

#### 《大型分布式网站架构设计与实践》



这本书总结了作者在构建安全、可稳定性、高扩展性、高并发的分布式网站方面的心得。

#### 《深入分布式缓存：从原理到实践》



这本书融合了原理、架构和一线互联网公司的案例实践，值得参考。

下面给入选精选留言的同学送出15元学习奖励礼券。专栏即将进入尾声，希望所有订阅的同学能够坚持到底，也欢迎大家留言分享自己学习或面试的心得体会。



# 中奖名单

昵称 奖励

jacy	15元无门槛学习奖励券
风动静泉	15元无门槛学习奖励券
欣	15元无门槛学习奖励券
铁拳阿牛	15元无门槛学习奖励券
ixm	15元无门槛学习奖励券
三口先生	15元无门槛学习奖励券
鹅米豆发	15元无门槛学习奖励券
三木子	15元无门槛学习奖励券
LenX	15元无门槛学习奖励券
Yano	15元无门槛学习奖励券
杨东yy	15元无门槛学习奖励券
盼盼	15元无门槛学习奖励券

WINNING LIST  
GOOD LUCK



iLeGeND

2018-07-28

我竟然都看过

忆水寒

2018-07-28

关于设计模式，我推荐一本书《设计模式之禅》，我觉得比《Head first设计模式》要好很多。

孤路齐飞

2018-07-28

关于看书，我时常不能做到深入。由于工作经验尚浅，对书中所表述的内容不是很懂，但是经过一段时间的积淀，往往能对书中某些部分产生共鸣，所以看书还是要看的，关键在于坚持，慢工出细活。不懂就多看几遍，厚积才能薄发^0^，感谢杨老师分享~

arryontheway

2018-07-28

effective java买了第二版还没看呢，第三版就出来了..

null

2018-07-28

看的书越多，自己却越迷茫。总在问自己：为什么我懂得那么少

LenX

2018-07-28

读专栏，做笔记，一篇没落下，感谢老师分享。

清风

2018-07-28

感谢老师的分享

齐帆

2018-07-29

《Java8 实战》

小树

2018-07-29

谢谢老师的引导

王全江

2018-07-28

是需要一点一点的啃，慢慢磨练

tracer

2018-07-28

关于设计模式我也向大家推荐一本图解设计模式，真的不错

爱芒果的董先生

2018-07-28

有些问题感觉深追下去就是和操作系统相关了。linux 内核简单的一些工作方式，tcp，ip协议的实现等。突然感觉真的有好多需要了解的。

zt

2018-07-28

看留言的数量发现坚持到最后的人不多◆◆，有些困惑，看书的时候理解的差不多，但到实际的时候找不到切入点，没法融入到实践中，老师能给指点下吗？

j.c.

2018-07-28

什么时候才能看完

揭晓明

2018-07-28

刚刚才加入专栏，加入太晚，希望能好好消化大牛的分享！

fullstack徐

2018-07-28

试一下

Darcy

听了老师的课收益良多，希望老师能补充一下，泛型和类型擦除等知识点，谢谢。♦♦

2018-07-28

Darcy

听了老师的课收益良多，希望老师能补充一下，泛型和类型擦除等知识点，谢谢。♦♦

2018-07-28

三木子

设计模式这本武功秘籍，有些这招式看过几遍。遇到强敌准备使招时，感觉又忘记了招式。悟性太差啊

2018-07-28

陈华应

感谢老师分享，赞一个

2018-07-28

陈道恒

有好几本，我也买了，的确都是好书，剩下的也会慢慢买齐。谢谢杨老师分享。

2018-07-28







第36讲 | 谈谈MySQL支持的事务隔离级别，以及悲观锁和乐观锁的原理和应用场景？

2018-07-31 杨晓峰



在日常开发中，尤其是业务开发，少不了利用Java对数据库进行基本的增删改查等数据操作，这也是Java工程师的必备技能之一。做好数据操作，不仅仅需要对Java语言相关框架的掌握，更需要对各种数据库自身体系结构的理解。今天这一讲，作为补充Java面试考察知识点的完整性，关于数据库的应用和细节还需要在实践中深入学习。

今天我要问你的问题是，**谈谈MySQL支持的事务隔离级别，以及悲观锁和乐观锁的原理和应用场景？**

**典型回答**

所谓隔离级别（[Isolation Level](#)），就是在数据库事务中，为保证并发读写的正确性而提出的定义，它并不是MySQL专有的概念，而是源于[ANSI/ISO](#)制定的[SQL-92](#)标准。

每种关系型数据库都提供了各自特色的隔离级别实现，虽然在通常的[定义](#)中是以锁为实现单元，但实际的实现千差万别。以最常见的MySQL InnoDB引擎为例，它是基于[MVCC](#)（Multi-Versioning Concurrency Control）和锁的复合实现，按照隔离程度从低到高，MySQL事务隔离级别分为四个不同层次：

- 读未提交（Read uncommitted），就是一个事务能够看到其他事务尚未提交的修改，这是最低的隔离水平，允许[脏读](#)出现。
- 读已提交（Read committed），事务能够看到的数据都是其他事务已经提交的修改，也就是保证不会看到任何中间性状态，当然脏读也不会出现。读已提交仍然是比较低级别的隔离，并不保证再次读取时能够获取同样的数据，也就是允许其他事务并发修改数据，允许不可重复读和幻象读（Phantom Read）出现。
- 可重复读（Repeatable reads），保证同一个事务中多次读取的数据是一致的，这是MySQL InnoDB引擎的默认隔离级别，但是和一些其他数据库实现不同的是，可以简单认为MySQL在可重复读级别不会出现幻象读。
- 串行化（Serializable），并发事务之间是串行化的，通常意味着读取需要获取共享读锁，更新需要获取排他写锁，如果SQL使用WHERE语句，还会获取区间锁（MySQL以GAP锁形式实现，可重复读级别中默认也会使用），这是最高的隔离级别。

至于悲观锁和乐观锁，也并不是MySQL或者数据库独有的概念，而是并发编程的基本概念。主要区别在于，操作共享数据时，“悲观锁”即认为数据出现冲突的可能性更大，而“乐观锁”则是认为大部分情况不会出现冲突，进而决定是否采取排他性措施。

反映到MySQL数据库应用开发中，悲观锁一般就是利用类似SELECT ... FOR UPDATE这样的语句，对数据加锁，避免其他事务意外修改数据。乐观锁则与Java并发包中的AtomicFieldUpdater类似，也是利用CAS机制，并不会对数据加锁，而是通过对比数据的时间戳或者版本号，来实现乐观锁需要的版本判断。

我认为前面提到的MVCC，其本质就可以看作是种乐观锁机制，而排他性的读写锁、双阶段锁等则是悲观锁的实现。

有关它们的应用场景，你可以构建一下简化的火车余票查询和购票系统。同时查询的人可能很多，虽然具体座位票只能是卖给一个人，但余票可能很多，而且也并不能预测哪个查询者会购票，这个时候就更适合用乐观锁。

**考点分析**

今天的问题来源于实际面试，这两部分问题反映了面试官试图考察面试者在日常应用开发中，是否学习或者思考过数据库内部的机制，是否了解并发相关的基础概念和实践。

我从普通数据库应用开发者的角度，提供了一个相对简化的答案，面试官很有可能进一步从实例的角度展开，例如设计一个典型场景重现脏读、幻象读，或者从数据库设计的角度，可以用哪些手段避免类似情况。我建议你在准备面试时，可以在典型的数据库上试验一下，验证自己的观点。

其他可以考察的点也有很多，在准备这个问题时你也可以对比Java语言的并发机制，进行深入理解，例如，随着隔离级别从低到高，竞争性（Contention）逐渐增强，随之而来的代价同样是性能和扩展性的下降。

数据库衍生出很多不同的职责方向：

- 数据库管理员（DBA），这是一个单独的专业领域。
- 数据库应用工程师，很多业务开发者就是这种定位，综合利用数据库和其他编程语言等技能，开发业务应用。
- 数据库工程师，更加侧重于开发数据库、数据库中间件等基础软件。

后面两者与Java开发更加相关，但是需要的知识和技能是不同的，所以面试的考察角度也有区别，今天我会分析下对相关知识学习和准备面试的看法。

另外，在数据库相关领域，Java工程师最常接触到的就是O/R Mapping框架或者类似的数据库交互类库，我会选取最广泛使用的框架进行对比和分析。

#### 知识扩展

首先，我来谈谈对数据库相关领域学习的看法，从最广泛的应用开发者角度，至少需要掌握：

- 数据库设计基础，包括数据库设计中的几个基本范式，各种数据库的基础概念，例如表、视图、索引、外键、序列号生成器等，清楚如何将现实中业务实体和其依赖关系映射到数据库结构中，掌握典型实体数据应该使用什么样的数据库数据类型等。
- 每种数据库的设计和实现多少会存在差异，所以至少要精通你使用过的数据库的设计要点。我今天开篇谈到的MySQL事务隔离级别，就区别于其他数据库，进一步了解MVCC、Locking等机制对于处理进阶问题非常有帮助；还需要了解，不同索引类型的使用，甚至是底层数据结构和算法等。
- 常见的SQL语句，掌握基础的SQL调优技巧，至少要了解基本思路是怎样的，例如SQL怎样写才能更好利用索引、知道如何分析[SQL执行计划](#)等。
- 更进一步，至少需要了解针对高并发等特定场景中的解决方案，例如读写分离、分库分表，或者如何利用缓存机制等，目前的数据存储也远不止传统的关系型数据库了。



上面的示意图简单总结了我对数据库领域的理解，希望可以给你进行准备时提供一些借鉴。当然在准备面试时并不是一味找一堆书闷头苦读，我还是建议从实际工作中使用的数据库出发，侧重于结合实践，完善和深化自己的知识体系。

接下来我们还是回到Java本身，目前最为通用的Java和数据库交互技术就是JDBC，最常见的开源框架基本都是构建在JDBC之上，包括我们熟悉的[JPA](#)/[Hibernate](#)、[MyBatis](#)、Spring JDBC Template等，各自都有独特的设计特点。

Hibernate是最负盛名的O/R Mapping框架之一，它也是一个JPA Provider。顾名思义，它是以对象为中心的，其强项更体现在数据库到Java对象的映射，可以很方便地在Java对象层面体现外键约束等相对复杂的关系，提供了强大的持久化功能。内部大量使用了[Lazy-load](#)等技术提高效率。并且，为了屏蔽数据库的差异，降低维护开销，Hibernate提供了类SQL的HQL，可以自动生成某种数据库特定的SQL语句。

Hibernate应用非常广泛，但是过度强调持久化和隔离数据库底层细节，也导致了很多弊端，例如HQL需要额外的学习，未必比深入学习SQL语言更高效；减弱程序员对SQL的直接控制，还可能导致其他代价。本来一句SQL的事情，可能被Hibernate生成几条，隐藏的内部细节也阻碍了进一步的优化。

而MyBatis虽然仍然提供了一些映射的功能，但更加以SQL为中心，开发者可以侧重于SQL和存储过程，非常简单、直接。如果我们的应用需要大量高性能的或者复杂的SELECT语句等，“半自动”的MyBatis就会比Hibernate更加实用。

而Spring JDBC Template也是更加接近于SQL层面，Spring本身也可以集成Hibernate等O/R Mapping框架。

关于这些具体开源框架的学习，我的建议是：

- 从整体上把握主流框架的架构和设计理念，掌握主要流程，例如SQL解析生成、SQL执行到结果映射等处理过程到底发生了什么。
- 掌握映射等部分的细节定义和原理，根据我在准备专栏时整理的面试题目，发现很多题目都是偏向于映射定义的细节。
- 另外，对比不同框架的设计和实现，既有利于你加深理解，也是面试考察的热点方向之一。

今天我从数据库应用开发者的角度，分析了MySQL数据库的部分内部机制，并且补充了我对数据库相关面试准备和知识学习的建议，最后对主流O/R Mapping等框架进行了简单的对比。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，从架构设计的角度，可以将MyBatis分为哪几层？每层都有哪些主要模块？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



lizishushu

2018-07-31

mybatis架构自下而上分为基础支撑层、数据处理层、API接口层这三层。

基础支撑层，主要是用来做连接管理、事务管理、配置加载、缓存管理等最基础组件，为上层提供最基础的支撑。  
数据处理层，主要是用来做参数映射、SQL解析、SQL执行、结果映射等处理，可以理解为请求到达，完成一次数据库操作的流程。  
API接口层，主要对外提供API，提供诸如数据的增删改查、获取配置等接口。

L.B.Q.Y

2018-07-31

mysql (innodb) 的可重复读隔离级别下，为什么可以认为不会出现幻像读呢？

anji

2018-07-31

0.sql工厂-主要设定数据库连接信息  
1.接口层-主要有Mapper接口用于对外提供具体的SQL执行方法  
2.xml文件-有具体的SQL实现语句，以及数据库字段对应Java类字段的映射关系。每个Mapper对应每个数据库表

郭国梁

2018-07-31

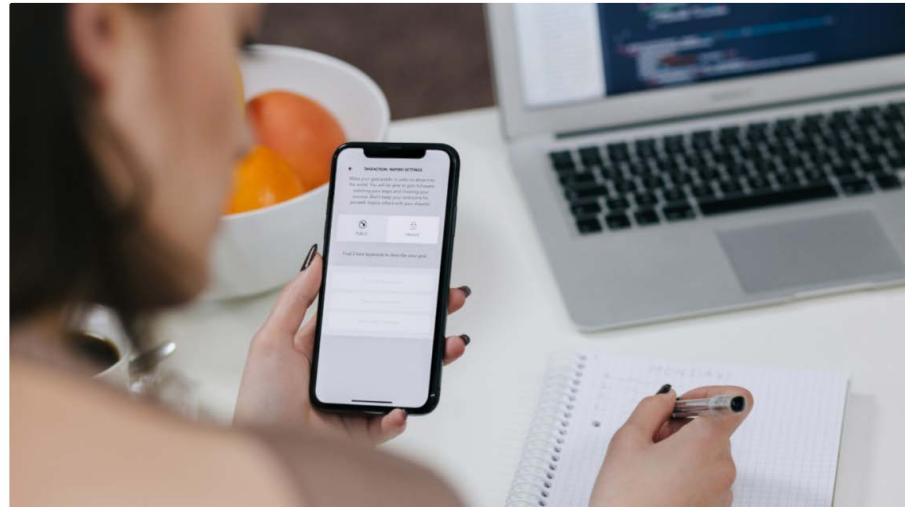
乐观锁 悲观锁 脏读 幻读 不可重复读 CAS MVCC 隔离级别 锁队列 2PC TCC





## 第37讲 | 谈谈Spring Bean的生命周期和作用域？

2018-08-02 杨晓峰



第37讲 | 谈谈Spring Bean的生命周期和作用域？

杨晓峰

- 00:15 / 09:59

在企业应用软件开发中，Java是毫无争议的主流语言，开放的Java EE规范和强大的开源框架功不可没，其中Spring毫无疑问已经成为企业软件开发的事实标准之一。今天这一讲，我将补充Spring相关的典型面试问题，并谈谈其部分设计细节。

今天我要问你的问题是，[谈谈Spring Bean的生命周期和作用域？](#)

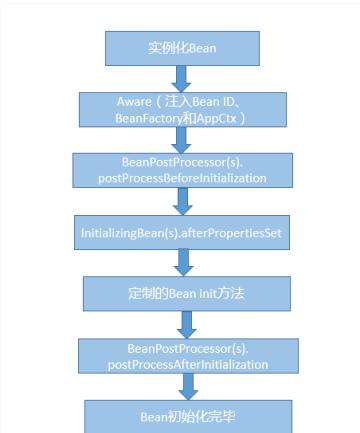
典型回答

Spring Bean生命周期比较复杂，可以分为创建和销毁两个过程。

首先，创建Bean会经过一系列的步骤，主要包括：

- 实例化Bean对象。
- 设置Bean属性。
- 如果我们通过各种Aware接口声明了依赖关系，则会注入Bean对容器基础设施层面的依赖。具体包括BeanNameAware、BeanFactoryAware和ApplicationContextAware，分别会注入Bean ID、Bean Factory或者ApplicationContext。
- 调用BeanPostProcessor的前置初始化方法postProcessBeforeInitialization。
- 如果实现了InitializingBean接口，则会调用afterPropertiesSet方法。
- 调用Bean自身定义的init方法。
- 调用BeanPostProcessor的后置初始化方法postProcessAfterInitialization。
- 创建过程完毕。

你可以参考下面示意图理解这个具体过程和先后顺序。



第二，Spring Bean的销毁过程会依次调用DisposableBean的destroy方法和Bean自身定制的destroy方法。

Spring Bean有五个作用域，其中最基础的有下面两种：

- Singleton，这是Spring的默认作用域，也就是为每个IOC容器创建唯一的一个Bean实例。
- Prototype，针对每个getBean请求，容器都会单独创建一个Bean实例。

从Bean的特点来看，Prototype适合有状态的Bean，而Singleton则更适合无状态的情况。另外，使用Prototype作用域需要经过仔细思考，毕竟频繁创建和销毁Bean是有明显开销的。

如果是Web容器，则支持另外三种作用域：

- Request，为每个HTTP请求创建单独的Bean实例。
- Session，很显然Bean实例的作用域是Session范围。
- GlobalSession，用于Portlet容器，因为每个Portlet有单独的Session，GlobalSession提供一个全局性的HTTP Session。

#### 考点分析

今天我选取的是一个入门性质的高频Spring面试题目，我认为相比于记忆题目典型回答里的细节步骤，理解和思考Bean生命周期所体现出来的Spring设计和机制更有意义。

你能看到，Bean的生命周期是完全被容器所管理的，从属性设置到各种依赖关系，都是容器负责注入，并进行各个阶段其他事宜的处理。Spring容器为应用开发者定义了清晰的生命周期沟通界面。

如果从具体API设计和使用技巧来看，还记得我在[专栏第13讲](#)提到过的Marker Interface吗，Aware接口就是个典型应用例子，Bean可以实现各种不同Aware的子接口，为容器以Callback形式注入依赖对象提供了统一入口。

言归正传，还是回到Spring的学习和面试。关于Spring，也许一整本书都无法完整涵盖其内容，专栏里我会有限地补充：

- Spring的基础机制。
- Spring框架的涵盖范围。
- Spring AOP自身设计的一些细节，前面[第24讲](#)偏重于底层实现原理，这样还不够全面，毕竟不管是动态代理还是字节码操纵，都还只是基础，更需要Spring层面对切面编程的支持。

#### 知识扩展

首先，我们先来看看Spring的基础机制，至少你需要理解下面两个基本方面。

- 控制反转（Inversion of Control），或者也叫依赖注入（Dependency Injection），广泛应用于Spring框架之中，可以有效地改善了模块之间的紧耦合问题。

从Bean创建过程可以看到，它的依赖关系都是由容器负责注入，具体实现方式包括带参数的构造函数、setter方法或者AutoWired方式实现。

- AOP，我们已经在前面接触过这种切面编程机制，Spring框架中的事务、安全、日志等功能都依赖于AOP技术，下面我会进一步介绍。

#### 第二，Spring到底是指什么？

我前面谈到的Spring，其实是狭义的[Spring Framework](#)，其内部包含了依赖注入、事件机制等核心模块，也包括事务、O/R Mapping等功能组成的数据访问模块，以及Spring MVC等Web框架和其他基础组件。

广义上的Spring已经成为了一个庞大的生态系统，例如：

- Spring Boot，通过整合通用实践，更加自动、智能的依赖管理等，Spring Boot提供了各种典型应用领域的快速开发基础，所以它是以应用为中心的一个框架集合。
- Spring Cloud，可以看作是在Spring Boot基础上发展出的更加高层次的框架，它提供了构建分布式系统的通用模式，包含服务发现和服务注册、分布式配置管理、负载均衡、分布式诊断等各种子系统，可以简化微服务系统的构建。
- 当然，还有针对特定领域的Spring Security、Spring Data等。

上面的介绍比较笼统，针对这么多内容，如果将目标定得太过宽泛，可能就迷失在Spring生态之中，我建议还是深入你当前使用的模块，如Spring MVC。并且，从整体上把握主要前沿框架（如Spring Cloud）的应用范围和内部设计，至少要了解主要组件和具体用途，毕竟如何构建微服务等，已经逐渐成为Java应用开发面试的热点之一。

第三，我们来探讨一下更多有关Spring AOP自身设计和实现的细节。

先问一下自己，我们为什么需要切面编程呢？

切面编程落实到软件工程其实是为了更好地模块化，而不仅仅是为了减少重复代码。通过AOP等机制，我们可以把横跨多个不同模块的代码抽离出来，让模块本身变得更加内聚，进而业务开发者可以更加专注于业务逻辑本身。从迭代能力上来看，我们可以通过切面的方式进行修改或者新增功能，这种能力不管是在问题诊断还是产品能力扩展中，都非常有用。

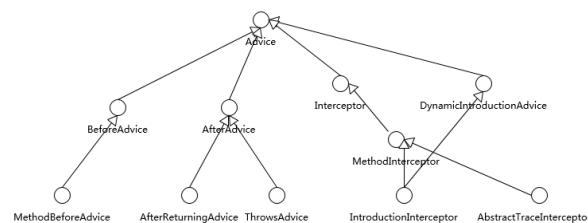
在之前的分析中，我们已经分析了AOP Proxy的实现原理，简单回顾一下，它底层是基于JDK动态代理或者cglib字节码操纵等技术，运行时动态生成被调用类型的子类等，并实例化代理对象，实际的方法调用会被代理给相应的代理对象。但是，这并没有解释具体在AOP设计层面，什么是切面，如何定义切入点和切面行为呢？

Spring AOP引入了其他几个关键概念：

- Aspect，通常叫作方面，它是跨不同Java类层面的横切性逻辑。在实现形式上，既可以是XML文件中配置的普通类，也可以在类代码中用“@Aspect”注解去声明。在运行时，Spring框架会创建类似Advisor来指代它，其内部会包括切入的时机（Pointcut）和切入的动作（Advice）。
- Join Point，它是Aspect可以切入的特定点，在Spring里面只有方法可以作为Join Point。
- Advice，它定义了切面中能够采取的动作。如果你去看Spring源码，就会发现Advice、Join Point并没有定义在Spring自己的命名空间里，这是因为它们是源自[AOP联盟](#)，可以看作是Java工程师在AOP层面沟通的通用规范。

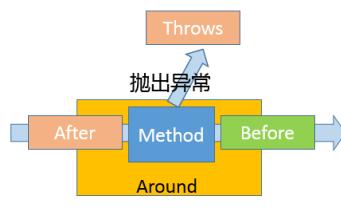
Java核心类库中同样存在类似代码，例如Java 9中引入的Flow API就是Reactive Stream规范的最小子集，通过这种方式，可以保证不同产品直接的无缝沟通，促进了良好实践的推广。

具体的Spring Advice结构请参考下面的示意图。



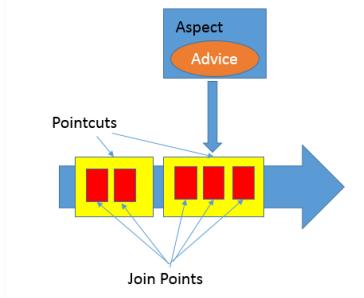
其中，`BeforeAdvice`和`AfterAdvice`包括它们的子接口是最简单的实现。而`Interceptor`则是所谓的拦截器，用于拦截住方法（也包括构造器）调用事件，进而采取相应动作，所以`Interceptor`是覆盖住整个方法调用过程的`Advice`。通常将拦截器类型的`Advice`叫作`Around`，在代码中可以使用“`@Around`”来标记，或者在配置中使用“`<aop:around>`”。

如果从时序上来看，则可以参考下图，理解具体发生的时机。



- Pointcut，它负责具体定义Aspect被应用在哪些Join Point，可以通过指定具体的类名和方法名来实现，或者也可以使用正则表达式来定义条件。

你可以参看下面的示意图，来进一步理解上面这些抽象在逻辑上的意义。



- Join Point仅仅是可利用的机会。

- Pointcut是解决了切面编程中的Where问题，让程序可以知道哪些机会点可以应用某个切面动作。

- 而Advice则是明确了切面编程中的What，也就是做什么；同时通过指定Before、After或者Around，定义了When，也就是什么时候做。

在准备面试时，如果在实践中使用过AOP是最好的，否则你可以选择一个典型的AOP实例，理解具体的实现语法细节，因为在面试考察中也许会问到这些技术细节。

如果你有兴趣深入内部，最好可以结合Bean生命周期，理解Spring如何解析AOP相关的注解或者配置项，何时何地使用到动态代理等机制。为了避免被复杂的源码弄晕，我建议你可以从比较精简的测试用例作为一个切入点，如[CglibProxyTests](#)。

另外，Spring框架本身功能点非常多，AOP并不是它所支持的唯一切面技术，它只能利用动态代理进行运行时编译，而不能进行编译期的静态编译或者类加载期编译。例如，在Java平台上，我们可以使用Java Agent技术，在类加载过程中对字节码进行操纵，比如修改或者替换方法实现等。在Spring体系中，如何做到类似功能呢？你可以使用AspectJ，它具有更加全面的能力，当然使用也更加复杂。

今天我从一个常见的Spring面试题开始，浅谈了Spring的基础机制，探讨了Spring生态范围，并且补充分析了部分AOP的设计细节，希望对你有所帮助。

#### 一课一线

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，请介绍一下Spring声明式事务的实现机制，可以考虑将具体过程画图。

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



null

2018-08-02

老师，IOC为什么可以实现解耦吖？

在引入IOC容器之前，对象A依赖于对象B，则需要A主动去创建对象B，控制权都在A。

在引入IOC容器之后，当对象A运行到需要对象B的时候，IOC容器会主动创建一个对象B注入到对象A，控制权在容器。

控制权发生了反转，为什么能降低系统耦合，或者说降低什么之间的耦合？（自己的理解：应该不是降低对象间的耦合，因为不管由A还是容器创建B对象，A都是耦合B的。感觉自己理解的方向偏了。）

谢谢！

作者回复

2018-08-02

IOC容器负责打理这些事情。同样的依赖关系，一个是a自己负责，一个是ioc容器负责，相当于ab之间的直接联系，变成了间接的。再配合OO，更换实现只需要修改配置

汉斯·冯·拉特

2018-08-03

想不到博主对spring也有深入了解。声明式事务是通过beanPostProcessor来实现的，springioc会用beanPostProcessor的某个方法（具体方法名忘记了，这里假设为方法A）返回结果为getBean的结果。所以spring的事务模块在方法A中，用代理的方式，在目标方法前后加入一些与事务有关的代码，方法A的返回值就是这个代理类。欢迎拍砖！

齐帆

2018-08-02

Advice 的时序图的before,after画反了吗

木子李

2018-08-02

希望大大可以多出几篇关于spring的文章，谢谢

yao\_jn

2018-08-02

读老师的文章收益很大，希望老师再对框架多讲一些，还有底层原理，毕竟很多时候看源码很费力，提点下会好很多！

铁拳阿牛

2018-08-02

可以按照课程去些demo到一个github项目里，配合章节理论，这样有理论有代码可能对课程，和对学员更有帮助！不过对老师的成本也提高了。

王

2018-08-02

能否介绍一下热加载，还有目前第三方软件，class，jar都可以热加载。







第38讲 | 对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？

2018-08-04 杨晓峰



第38讲 | 对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？  
杨晓峰  
00:18 / 09:27

今天我会对NIO进行一些补充，在[专栏第11讲](#)中，我们初步接触了Java提供的几种IO机制，作为语言基础类库，Java自身的NIO设计更偏底层，这本无可厚非，但是对于一线的应用开发者，其复杂性、扩展性等方面，就存在一定的局限了。在基础NIO之上，Netty构建了更加易用、高性能的网络框架，广泛应用于互联网、游戏、电信等各种领域。

今天我要问你的问题是，[对比Java标准NIO类库，你知道Netty是如何实现更高性能的吗？](#)

#### 典型回答

单独从性能角度，Netty在基础的NIO等类库之上进行了很多改进，例如：

- 更加优雅的Reactor模式实现、灵活的线程模型、利用EventLoop等创新性的机制，可以非常高效地管理成百上千的Channel。
- 充分利用了Java的Zero-Copy机制，并且从多种角度，“斤斤计较”般的降低内存分配和回收的开销。例如，使用池化的Direct Buffer等技术，在提高IO性能的同时，减少了对象的创建和销毁；利用反射等技术直接操纵SelectionKey，使用数组而不是Java容器等。
- 使用更多本地代码。例如，直接利用JNI调用Open SSL等方式，获得比Java内建SSL引擎更好的性能。
- 在通信协议、序列化等其他角度的优化。

总的来说，Netty并没有Java核心类库那些强烈的通用性、跨平台等各种负担，针对性能等特定目标以及Linux等特定环境，采取了一些极致的优化手段。

#### 考点分析

这是一个比较开放的问题，我给出的回答是个概要性的举例说明。面试官很可能利用这种开放问题作为引子，针对你回答的一个或者多个点，深入探讨你在不同层次上的理解程度。

在面试准备中，兼顾整体性的同时，不要忘记选定个别重点进行深入理解掌握，最好是进行源码层面的深入阅读和实验。如果你希望了解更多从性能角度Netty在编码层面的手段，可以参考Norman在Devoxx上的[分享](#)，其中的很多技巧对于实现极致性能的API有一定借鉴意义，但在一般的业务开发中要谨慎采用。

虽然提到Netty，人们会自然地想到高性能，但是Netty本身的优势不仅仅只有这一个方面。

下面我会侧重两个方面：

- 对Netty进行整体介绍，帮你了解其基本组成。
- 从一个简单的例子开始，对比在[第11讲](#)中基于IO、NIO等标准API的实例，分析它的技术要点，给你提供一个进一步深入学习的思路。

#### 知识扩展

首先，我们从整体了解一下Netty。按照官方定义，它是一个异步的、基于事件Client/Server的网络框架，目标是提供一种简单、快速构建网络应用的方式，同时保证高吞吐量、低延时、高可靠性。

从设计思路和目的上，Netty与Java自身的NIO框架相比有哪些不同呢？

我们知道Java的标准类库，由于其基础性、通用性的定位，往往过于关注技术模型上的抽象，而不是从一线应用开发者的角度去思考。我曾提到过，引入并发包的一个重要原因就是，应用开发者使用Thread API比较痛苦，需要操心的不仅仅是业务逻辑，而且还要自己负责将其映射到Thread模型上。Java NIO的设计也有类似的特点，开发者需要深入掌握线程、IO、网络等相关概念，学习路径很长，很容易导致代码复杂、晦涩，即使是有经验的工程师，也难以快速地写出高可靠的实现。

Netty的设计强调了“Separation Of Concerns”，通过精巧设计的事件机制，将业务逻辑和无关技术逻辑进行隔离，并通过各种方便的抽象，一定程度上填补了基础平台和业务并发之间的鸿沟，更有利于在应用开发中普及业界的最佳实践。

另外，Netty > java.nio + java.net!

从API能力范围来看，Netty完全是Java NIO框架的一个大大的超集，你可以参考Netty官方的模块划分。



除了核心的事件机制等，Netty还额外提供了很多功能，例如：

- 从网络协议的角度，Netty除了支持传输层的UDP、TCP、[SCTP](#)协议，也支持HTTP(s)、WebSocket等多种应用层协议，它并不是单一协议的API。
- 在应用中，需要将数据从Java对象转换成为各种应用协议的数据格式，或者进行反向的转换，Netty为此提供了一系列扩展的编码解码框架，与应用开发场景无缝衔接，并且性能良好。
- 它扩展了Java NIO Buffer，提供了自己的ByteBuf实现，并且深度支持Direct Buffer等技术，甚至hack了Java内部对Direct Buffer的分配和销毁等。同时，Netty也提供了更加完善的Scatter/Gather机制实现。

可以看到，Netty的能力范围大大超过了Java核心类库中的NIO等API，可以说它是一个从应用视角出发的产物。

当然，对于基础API设计，Netty也有自己独到的见解，未来Java NIO API也可能据此进行一定的改进，如果你有兴趣可以参考[JDK-8187540](#)。

接下来，我们一起来看一个入门的代码实例，看看Netty应用到底是什么样子。

与[第11讲](#)类似，同样是以简化的Echo Server为例，下图是Netty官方提供的Server部分，完整用例请点击[链接](#)。

```
public final class EchoServer {
    static final boolean SSL = System.getProperty("ssl") != null;
    static final int PORT = Integer.parseInt(System.getProperty("port", "8007"));

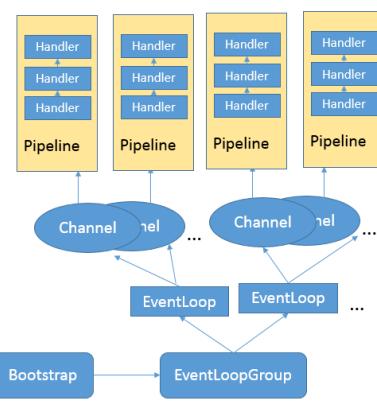
    public static void main(String[] args) throws Exception {
        // Configure SSL.
        final SslContext sslCtx;
        if (SSL) {
            SslContextBuilder sslCtxtBuilder = SslContextBuilder.forServer(ssc.certificate(), ssc.privateKey()).build();
            sslCtx = sslCtxtBuilder;
        } else {
            sslCtx = null;
        }

        // Create a server.
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        final EchoServerHandler serverHandler = new EchoServerHandler();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
              .channel(NioServerSocketChannel.class)
              .option(ChannelOption.SO_BACKLOG, 100)
              .handler(new LoggingHandler(LogLevel.INFO))
              .childHandler(new ChannelInitializer<SocketChannel>() {
                  @Override
                  public void initChannel(SocketChannel ch) throws Exception {
                      ChannelPipeline p = ch.pipeline();
                      if (sslCtx != null) {
                          p.addLast(sslCtx.newHandler(ch.alloc()));
                      }
                      p.addLast(new LoggingHandler(LogLevel.INFO));
                      p.addLast(serverHandler);
                  }
              });
            // Start the server.
            ChannelFuture f = b.bind(PORT).sync();
            // Wait until the server socket is closed.
            f.channel().closeFuture().sync();
        } finally {
            // Shut down all event loops to terminate all threads.
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

上面的例子，虽然代码很短，但已经足够体现出Netty的几个核心概念，请注意我用红框标记出的部分：

- ServerBootstrap**，服务器端程序的入口，这是Netty为简化网络程序配置和关闭等生命周期管理，所引入的Bootstrapping机制。我们通常要做的创建Channel、绑定端口、注册Handler等，都可以通过这个统一的入口，以Fluent API等形式完成，相对简化了API使用。与之相对应，[Bootstrap](#)则是Client端的通常入口。
- Channel**，作为一个基于NIO的扩展框架，Channel和Selector等概念仍然是Netty的基础组件，但是针对应用开发具体需求，提供了相对易用的抽象。
- EventLoop**，这是Netty处理事件的核心机制。例子中使用了EventLoopGroup。我们在NIO中通常要做的几件事情，如注册感兴趣的事件、调度相应的Handler等，都是EventLoop负责。
- ChannelFuture**，这是Netty实现异步IO的基础之一，保证了同一个Channel操作的调用顺序。Netty扩展了Java标准的Future，提供了针对自己场景的特有Future定义。
- ChannelHandler，这是应用开发者放置业务逻辑的主要地方，也是我上面提到的“Separation Of Concerns”原则的体现。
- ChannelPipeline**，它是ChannelHandler链条的容器，每个Channel在创建后，自动会被分配一个ChannelPipeline。在上面的示例中，我们通过ServerBootstrap注册了ChannelInitializer，并且实现了initChannel方法，而在该方法中则承担了向ChannelPipeline安装其他Handler的任务。

你可以参考下面的简化示意图，忽略Inbound/OutBound Handler的细节，理解这几个基本单元之间的操作流程和对应关系。



对比Java标准NIO的代码，Netty提供的相对高层次的封装，减少了对Selector等细节的操纵，而EventLoop、Pipeline等机制则简化了编程模型，开发者不用担心并发等问题，在一定程度上简化了应用代码的开发。最难能可贵的是，这一切并没有以可靠性、可扩展性为代价，反而将其大幅度提高。

我在[专栏周末福利](#)中已经推荐了Norman Maurer等编写的《Netty实战》（Netty In Action），如果你想系统学习Netty，它会是个很好的入门参考。针对Netty的一些实现原理，很可能成为面试中的考点，例如：

- Reactor模式和Netty线程模型。
- Pipelining、EventLoop等部分的设计实现细节。
- Netty的内存管理机制、[引用计数](#)等特别手段。
- 有的时候面试官也喜欢对比Java标准NIO API，例如，你是否知道Java NIO早期版本中的Epoll[空转问题](#)，以及Netty的解决方式等。

对于这些知识点，公开的深入解读已经有很多了，在学习时希望你不要一开始就被复杂的细节弄晕，可以结合实例，逐步、有针对性的进行学习。我的一个建议是，可以试着画出相应的示意图，非常有助于理解并能清晰阐述自己的看法。

今天，从Netty性能的问题开始，我概要地介绍了Netty框架，并且以Echo Server为例，对比了Netty和Java NIO在设计上的不同。但这些都仅仅是冰山的一角，全面掌握还需要下非常多的功夫。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，Netty的线程模型是什么样的？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



亿水寒

2018-08-06

Netty采用Reactor线程模型。这里面主要有三种Reactor线程模型，分别是单线程模式、主从Reactor模式、多Reactor线程模式。其都可以通过初试和EventLoopGroup进行设置。其主要区别在于，单Reactor模式就是一个线程，既进程处理连接，也处理IO。类似于我们传统的OIO编程。主从Reactor模式，其实就是将监听连接和处理IO的分开在不同的线程完成。最后，主从Reactor线程模型，为了解决多Reactor模型下单一线程性能不足的问题。改为了多线程池进行处理。官方默认的是采用这种主从Reactor模型。其线程数默认为CPU内核的2倍。杨老师，不知道我说的对不对？

作者回复

2018-08-07

对的

---

Levy

netty线程模型一般分为监听线程和I/O处理线程，也即bossGroup和workerGroup，属于多Reactor模型

作者回复

2018-08-04

是的，不同版本模型有点区别，但逻辑上都还是区分

2018-08-05

---

zt

2018-08-06

老师，能给说下mina和netty的相同和不同吗？

作者回复

2018-08-07

mina没仔细研究过…

咖啡猫口里的咖啡猫◆◆

2018-08-05

为什么我看开源项目把netty做网络层，模型都是1-N，为什么N-N会浪费资源

---

L

2018-08-05

杨老师，可以的话麻烦推荐几个能获取到最新Java开发资料，或者编程领域相关的外文网站，博客等，就是想关注一些最新的技术信息

Allen

2018-08-05

请问为啥不推荐netty权威指南？◆◆

Allen

2018-08-05

有三种模型。1-1。1-n n-n. 并且可以根据实际情况自动进行调整，可谓是线程模型的终极版本，简直是太酷了





第39讲 | 谈谈常用的分布式ID的设计方案？Snowflake是否受冬令时切换影响？

2018-08-07 杨晓峰



第39讲 | 谈谈常用的分布式ID的设计方案？Snowflake是否受冬令时切换影响？

杨晓峰

- 00:16 / 10:03

专栏的绝大部分主题都侧重于Java语言和虚拟机，基本都是单机模式下的问题，今天我会补充一个分布式相关的问题。严格来说，分布式并不算是Java领域，而是一个单独的大主题，但确实也会在Java技术岗位面试中被涉及。在准备面试时，如果有丰富的分布式系统经验当然好；如果没有，你可以选择典型问题和基础技术进行适当准备。关于分布式，我自身的实战经验也非常有限，专栏里就谈谈从理论出发的一些思考。

今天我要问你的是，[谈谈常用的分布式ID的设计方案？Snowflake是否受冬令时切换影响？](#)

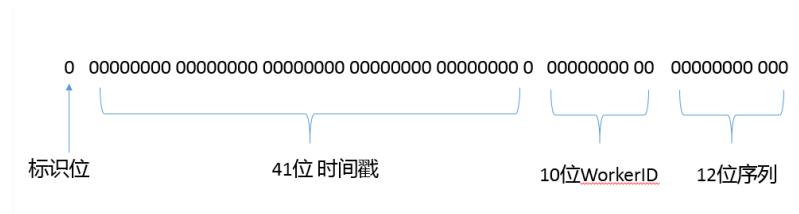
#### 典型回答

首先，我们需要明确通常的分布式ID定义，基本的要求包括：

- 全局唯一，区别于单点系统的唯一，全局是要求分布式系统内唯一。
- 有序性，通常都需要保证生成的ID是有序递增的。例如，在数据库存储等场景中，有序ID便于确定数据位置，往往更加高效。

目前业界的方案很多，典型方案包括：

- 基于数据库自增序列的实现。这种方式优缺点都非常明显，好处是简单易用，但是在扩展性和可靠性等方面存在局限性。
- 基于Twitter早期开源的[Snowflake](#)的实现，以及相关改动方案。这是目前应用相对比较广泛的一种方式，其结构定义你可以参考下面的示意图。



整体长度通常是64 ( $1 + 41 + 10 + 12 = 64$ ) 位，适合使用Java语言中的long类型来存储。

头部是1位的正负标识位。

紧跟着的高位部分包含41位时间戳，通常使用System.currentTimeMillis()。

后面是10位的WorkerID，标准定义是5位数据中心 + 5位机器ID，组成了机器编号，以区分不同的集群节点。

最后的12位就是单位毫秒内可生成的序列号数目的理论极限。

Snowflake的[官方版本](#)是基于Scala语言，Java等其他语言的[参考实现](#)有很多，是一种非常简单实用的方式，具体位数的定义是可以根据分布式系统的真实场景进行修改的，并不一定严格按照示意图中的设计。

- Redis、Zookeeper、MongoDB等中间件，也都有各种唯一ID解决方案。其中一些设计也可以算作是Snowflake方案的变种。例如，MongoDB的[ObjectId](#)提供了一个12 byte (96位) 的ID定义，其中32位用于记录以秒为单位的时间，机器ID则为24位，16位用作进程ID，24位随机起始的计数序列。
- 国内的一些大厂开源了其自身的部分分布式ID实现，InfoQ就曾经介绍过微信的[seqsvr](#)。它采取了相对复杂的两层架构，并根据社交应用的数据特点进行了针对性设计，具体请参

考相关[代码实现](#)。另外，[百度](#)、美团等也都有开源或者分享了不同的分布式ID实现，都可以进行参考。

关于第二个问题，Snowflake是否受冬令时切换影响？

我认为没有影响，你可以从Snowflake的具体算法实现寻找答案。我们知道Snowflake算法的Java实现，大都是依赖于System.currentTimeMillis()，这个数值代表什么呢？从Javadoc可以看出，它是返回当前时间和1970年1月1号UTC时间相差的毫秒数，这个数值与夏/冬令时并没有关系，所以并不受其影响。

#### 考点分析

今天的问题不仅源自面试的热门考点，并且也存在着广泛的应用场景，我前面给出的回答只是一个比较精简的典型方案介绍。我建议你针对特定的方案进行深入分析，以保证在面试官可能会深入追问时能有充分准备；如果恰好在现有系统使用分布式ID，理解其设计细节是很有必要的。

涉及分布式，很多单机模式下的简单问题突然就变得复杂了，这是分布式天然的复杂性，需要从不同角度去理解适用场景、架构和细节算法，我会从下面的角度进行适当解读：

- 我们的业务到底需要什么样的分布式ID，除了唯一和有序，还有哪些必须要考虑的要素？
- 在实际场景中，针对典型的方案，有哪些可能的局限性或者问题，可以采取什么办法解决呢？

#### 知识扩展

如果试图深入回答这个问题，首先需要明确业务场景的需求要点，我们到底需要一个什么样的分布式ID？

除了唯一和有序，考虑到分布式系统的功能需要，通常还会额外希望分布式ID保证：

- 有意义，或者说包含更多信息，例如时间、业务等信息。这一点和有序性要求存在一定关联，如果ID中包含时间，本身就能保证一定程度的有序，虽然并不能绝对保证。ID中包含额外信息，在分布式数据存储等场合中，有助于进一步优化数据访问的效率。
- 高可用性，这是分布式系统的必然要求。前面谈到的方案中，有的是真正意义上的分布式，有的还是传统主从的思路，这一点没有绝对的对错，取决于我们业务对扩展性、性能等方面的要求。
- 紧凑性，ID的大小可能受到实际应用的制约，例如数据库存储往往对长ID不友好，太长的ID会降低MySQL等数据库索引的性能，编程语言在处理时也可能受数据类型长度限制。

在具体的生产环境中，还有可能提出对QPS等方面的具体要求，尤其是在国内一线互联网公司的业务规模下，更是需要考虑峰值业务场景的数量级层次需求。

#### 第二，主流方案的优缺点分析。

对于数据库自增方案，除了实现简单，它生成的ID还能够保证固定步长的递增，使用很方便。

但是，因为每获取一个ID就会触发数据库的写请求，是一个代价高昂的操作，构建高扩展性、高性能解决方案比较复杂，性能上限明显，更不要谈扩容等场景的难度了。与此同时，保证数据库方案的高可用性也存在挑战，数据库可能发生宕机，即使采取主从热备等各种措施，也可能出现ID重复等问题。

实际大厂往往是构建了多层的组合架构，例如美团公开的数据库方案[Leaf-Segment](#)，引入了起到缓存等作用的Leaf层，对数据库操作则是通过数据库中间件提供的批量操作，这样既能保证性能、扩展性，也能保证高可用。但是，这种方案对基础架构层面的要求很多，未必适合普通业务规模的需求。

与其相比，Snowflake方案的好处是算法简单，依赖也非常少，生成的序列可预测，性能也非常好，比如Twitter的峰值超过10万/s。

但是，它也存在一定的不足，例如：

- 时钟偏斜问题（Clock Skew）。我们知道普通的计算机系统时钟并不能保证长久的一致性，可能发生时钟回拨等问题，这就会导致时间戳不准确，进而产生重复ID。

针对这一点，Twitter曾经在文档中建议开启[NTP](#)，毕竟Snowflake对时间存在依赖，但是也有人提议关闭NTP。我个人认为还是应该开启NTP，只是可以考虑将stepback设置为0，以禁止回调。

从设计和具体编码的角度，还有一个很有效的措施就是缓存历史时间戳，然后在序列生成之前进行检验，如果出现当前时间落后于历史时间的不合理情况，可以采取相应的动作，要么重试、等待时钟重新一致，或者就直接提示服务不可用。

- 另外，序列号的可预测性是把双刃剑，虽然简化了一些工程问题，但很多业务场景并不适合可预测的ID。如果你用它作为安全令牌之类，则是非常危险的，很容易被黑客猜测并利用。
- ID设计阶段需要谨慎考虑暴露出的信息。例如，[Erlang版本的flake实现](#)基于MAC地址计算WorkerID，在安全敏感的领域往往是不可以这样使用的。
- 从理论上来说，类似Snowflake的方案由于时间数据位数的限制，存在与[2038年问题](#)相似的理论极限。虽然目前的系统设计考虑十年后的问题还早，但是理解这些可能的极限是有必要的，也许会成为面试的过程中的考察点。

如果更加深入到时钟和分布式系统时序的问题，还有与分布式ID相关但又有所区别的问题，比如在分布式系统中，不同机器的时间很可能是不一致的，如何保证事件的有序性？Lamport在1978年的论文（[Time, Clocks, and the Ordering of Events in a Distributed System](#)）中就有很深入的阐述，有兴趣的同学可以去查找相应的翻译和解读。

最后，我再补充一些当前分布式领域的面试热点，例如：

- 分布式事务，包括其产生原因、业务背景、主流的解决方案等。
- 理解[CAP](#)、[BASE](#)等理论，懂得从最终一致性等角度来思考问题，理解[Paxos](#)、[Raft](#)等一致性算法。
- 理解典型的分布式锁实现，例如最常见的[Redis分布式锁](#)。
- 负载均衡等分布式领域的典型算法，至少要了解主要方案的原理。

这些方面目前已经有相对比较深入的分析，尤其是来自一线大厂的实践经验。另外，在[左耳听风专栏的“程序员进阶攻略”](#)里，提供了非常全面的分布式学习资料，感兴趣的同学可以参考。

今天我简要梳理了当前典型的分布式ID生成方案，并探讨了ID设计的一些考量，尤其是应用相对广泛的Snowflake的不足之处，希望对你有所帮助。

#### 一课一练

关于今天我们讨论的题目你做到心中有数了吗？今天的思考题是，从理论上来看，Snowflake这种基于时间的算法，从形式上天然地限制了ID的并发生成数量，如果在极端情况下，短时间需要更多ID，有什么办法解决呢？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



杨晓峰 Oracle 首席工程师

涛哥

能讲下下时钟偏斜和时钟回拨吗，不是理解

有铭

为啥最后一段是12的长度而不是别的数

作者回复

我提到了，各部分不是固定的，看业务需求，例如，集群小，位数可以设计短点儿，seq就可以更多位，时间也未必非要41位

黄琨

缩减workID长度，增加序列号长度

2018-08-07

2018-08-07

2018-08-07

2018-08-07





结束语 | 技术没有终点  
2018-08-09 杨晓峰

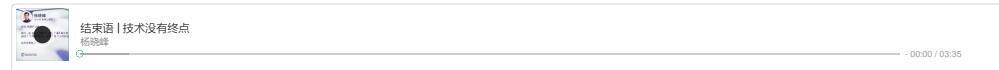
杨晓峰  
Oracle 首席工程师

你好,我是杨晓峰。

我们一起度过了**100天**,学习了**42篇文章**,  
阅读了**135,042字**,收听了**6个小时的音频**。

技术没有终点.....

**极客时间**



James Governor曾经说过：“这么多年最大的体会就是Java is Dead is Dead！”。我也是同样的感受，Java已经一再被证明它在业界中不可替代的作用。选择Java语言的开发者是幸运的，经历了如此漫长的发展后，我们依然有幸可以见证“廉颇老矣”的Java再次加速成长“焕发青春”，因此非常值得你深入进去。

时间过得飞快，专栏已经到了尾声，非常感谢你的支持。这一路走来，真心体会到专栏创作的不易，经常在深夜反复斟酌文章的难易程度，踌躇如何让内容既不失深入性，又要通俗易懂，尽量覆盖更多知识点。同时，也有深深的不舍，你的留言和反馈，迸发出了知识的火花，也让我进一步感受到了专栏的价值所在。

回想最初我在专栏内容设计时，希望更侧重于Java语言和虚拟机的基础领域，因为这些内容在飞速变化的世界中更加具备长久价值。

在专栏里，我与你一起重温了Java语言和虚拟机那些“黑魔法”，并通过探讨其背后的故事，尽我所能帮你达到“知其所以然”和体系化的目标。通过专栏的学习，相信面试官在考察并发、JVM等内部结构和机制的时候，你一定能做到胸有成竹。如果我的专栏，对你在日常软件设计或者问题诊断时还能有所帮助，就再好不过了。

在这里，同为工程师的我，想和同样走在这条道路上的同学，聊聊我对工程师职业发展的看法，希望对你有参考意义。

Easy is cheap！在平时工作中，技术人员免不了要构建一个广泛的知识体系，但终归是要克制住诱惑，将某个领域做到精深。水桶装水量取决于最短板，但是大多数情况下，我们在工作中获取的回报，更多来源于自身的长处，甚至某种程度上还决定了我们是拥有自己选择的自由，还是疲于奔命，毕竟我们每个人的体力、精力是有现实的上限的。

在工作岗位上，从初级到高级工程师成长的过程中，最基本的一个变化就是，我们的角色会逐渐发生从how到what的转换。工作初期，我们更多是承担被指派的任务，重点是搞清楚怎么做，但是当我们逐步成长起来，更多的是要看清楚什么是最重要的。

还好在极客时间的专栏里，我并不是你的老师，而是和你在一起交流的工程师，交流促进思考，沟通产生价值，技术人永远不要羞于表达自己的观点，请你坚持独立思考。

对于刚刚才订阅专栏的新同学来说，可能你打开的第一篇文章就是结束语，不过不用担心，我会一直在这里解答你学习过程中的疑惑，我给你的专栏学习建议是：注重实践和项目推动，确保结果输出，仅仅把专栏看作是个参照物，找到自己的技术道路。

对于从开始就伴随专栏一路走来的老同学，我想对你说：技术没有终点，感谢你的一路陪伴，也希望我们共同的努力能够带来丰富的收获。

最后，如果你对Java领域的前沿技术感兴趣，也欢迎关注我的公众号“xiaofoya”，我会不定期更新，嗯，非常不定期。

讲到这里，专栏的结束并不代表你将止步于此，而是应该通过前面的学习，把专栏的结束当作新的开始，不管过去你是否从中掌握了新的知识或是提升了视野，希望你能以一种全新的状态重新出发，继续勇攀技术的高峰。



**杨晓峰**  
Oracle 首席工程师

“

不知道在学习过程中，你有哪些体会和评价？  
这里有一份专栏调查问卷，邀请你填写。

在8月11日前提交，  
极客时间赠送给你专属优惠券。

我们一起继续成长！

[去提交](#)



**极客时间**  
带给开发者·能升技术认知

# Java核心技术36讲

— Oracle 首席工程师 —

带你修炼 Java 内功 —

**杨晓峰** Oracle 首席工程师



杨晓峰

2018-08-10

非常感谢大家的支持，深深感受到大家对技术的热忱和坚持，祝福并相信每个人都可以在工作生活中取得不断的进步！

摩羯行僧

2018-08-09

感谢老师的辛勤付出和精彩讲解。这一路走来，跟随老师的专栏又一次对Java知识体系有了更深的理解和掌握。许多以前有点迷惑或一知半解的地方都得到了深入的解答，有了更明确的认知。只是没想到这么快就要和老师说再见了，真有点不舍。不过就像老师说的，技术没有终点，以后的路还得需要自己去坚持学习。专栏中还有许多章节和知识点我还需要回顾和实践，以达到真正的精通和掌握。最后也祝老师身体健康，生活愉快！

三木子

2018-08-09

这个专栏对于我最大的收获就是明白要在一个领域精耕细作。技术没有终点，我才刚上车。

臣醉飞扬

2018-08-09

好像还有消息队列，redis，微服务和SOA区别等几个系列没讲呢？

零点的钟声

2018-08-09

通过阅读作者的文章不仅让我的基础更加牢固，而且也为我解惑了工作中没有去仔细思考的疑惑！真所谓传道授业解惑也，在这说声：先生，辛苦您了！并且希望自己在以后的工作中能够多学，多想，做到技术上的知行合一！貌似有点难？毕竟我是一个想成为架构师的男人！

作者回复

2018-08-11

非常感谢

忆水寒

杨老师的文章真是不错，系统的帮我们复习了java相关知识。期待杨老师出下一期课程。也希望在这个平台认识更多优秀的朋友，可以关注我的公众号：码农的修炼之道。我们跟着杨老师的脚步一起进步。

作者回复

2018-08-11

非常感谢，已关注

侯树成

2018-08-09

感谢晓峰老师，我是个8年多的JAVA工程师，做过两年J2EE应用服务器的开发。自己也写了快三年公众号。一开始订阅的时候心想可能专栏会写的阳春白雪不接地气，单纯是想学习技术文章的写作风格。但您一篇篇的更新时，我能感受到内容的循循善诱，能感受到为了考虑读者接受程度的斟酌。有一次看回复里，您看读者留言时表示「老学究」很欣慰，心有戚戚焉。专栏真的很棒，没想到这么快就结束了，很不舍。再次感谢，祝您工作顺利。

作者回复

2018-08-11

非常感谢，互相交流

ebony

2018-08-09

老师您好，看您说选择java是幸运的，这一点我深有同感。我想问下net转java的一些建议。

晏传兵

2018-08-13

感谢老师付出！

唐诗三百首

2018-08-12

我在极客的第一次学费交给了你，谢谢老师，希望老师多多分享知识

就你最近表现

2018-08-12

杨老师，我是一位刚刚毕业的应届生，也是第一次在您的专栏中留言，首先感谢您对知识的无私奉献，确实给我这个刚从事工作的新手提供了很大的帮助，技术没有终点，请问一下杨老师，之后您还会继续新的专栏吗，我很期待

Lyle

2018-08-10

感谢大牛的分享！！！

作者回复

2018-08-11

非常感谢

李韶文TSOS

2018-08-10

老师专栏中的JDK版本是多少？有些专栏中代码，我自己去照着写，IDE总是报错

作者回复

2018-08-11

我基本都是dk11了

rivers

2018-08-10

感谢杨老师的每一篇文章，让我对java的知识结构理解的更加系统，感谢老师的殷勤付出，谢谢

作者回复

2018-08-11

非常感谢

疯狂的栗子

2018-08-10

谢谢杨老师，我是一年JAVA小菜鸟，平常都是写写业务代码，很少去考究底层代码的实现，我觉得每一讲都对我有莫大帮助，仅靠工作接触去学好JAVA是不可能的，平常的学习也非常重要的

作者回复

2018-08-11

感谢支持，互相交流提高

三一口二

2018-08-10

谢谢！确实收获很大！

作者回复

2018-08-11

非常感谢

资小军

2018-08-09

坚持看完每一篇文章和评论，感受颇深。也发现了自己的短板。再次感谢前辈的分享与付出。学无止境。共勉

作者回复

2018-08-11

共勉

杨逸林

2018-08-09

感谢◆◆

作者回复

2018-08-11

共勉

ayou

2018-08-09

感谢老师滴辛苦付出！

作者回复

2018-08-11

共勉

Allen

2018-08-09

一路走来，您对我影响最大的一句话就是程序应该以表达他的语意为主，并将其用于我写代码的实践中，无悔入专栏，谢谢老师◆◆

作者回复

2018-08-11

共勉

零点的钟声

2018-08-09

通过阅读作者的文章，使我的基础更加牢固，不仅让我Java有了更深的理解，而且也解决了一些在开发过程中没有去研究的疑惑。总之受益匪浅，学而不思则罔，思而不学则怠！希望自己以后能够多想，多学，多做！我是要成为架构师的男人！我

作者回复

2018-08-11

非常感谢

勇

2018-08-09

谢谢老师

作者回复

2018-08-11

共勉

jack

2018-08-09

技术没有终点，一直保持求知心态！

作者回复

2018-08-11

共勉

任鹏斌

2018-08-09

谢谢老师，专栏已结束学习在路上，继续加油

作者回复

2018-08-11

非常感谢

syz

2018-08-09

谢谢老师 我是做安卓的 听老师的课是为了回顾java基础 只是好难呀◆◆ 只听明白其中三五节课但也超值了 谢谢老师

作者回复

2018-08-11

啊...那些部分，我再改进一下

爱吃芒果的董先森

2018-08-09

stay hungry stay foolish。送给一起走在技术路上的我们。

作者回复

2018-08-11

共勉

Allen

2018-08-09

谢谢老师

作者回复

2018-08-11

非常感谢

陈华应

2018-08-09

谢谢老师，找到自己的方向，学习，实践，思考，努力行程自己的体系！

作者回复

2018-08-11

不错，互相交流提高

淡定@年华

2018-08-09

谢谢老师