关注性能：改进您的开发过程 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/252/2021_2022__E5_85_B3_E6_B3_A8_E6_80_A7_E8_c104_252377.htm 性能是 Java 平台屡屡受到指责的一个方面。然而，Java 平台的巨大成功也使得对性能问题作一番严肃的调查研究颇有必要。在这个新专栏中，无畏的优化大师 Jack Shirazi 和 Kirk Pepperdine，分别是 JavaPerformanceTuning.com 的董事和 CTO，他们在整个 Internet 上推行性能大讨论，展开他们所碰到的问题并加以澄清。本月，他们来到 JavaRanch，讨论有关编译速度、异常以及堆长度调优等方面的话题。上个月，我们在 JavaRanch 的 Big Moose Saloon 板块上花了大量的时间，以便查看 JavaRanch 的生手会提出什么样的性能方面的疑问。后来发现，大部分问题都是关于 J2SE 和开发过程的提出的问题主要是关于 Java 语言、核心类以及如何改进他们的开发过程。编译速度您是否曾发现您的编译阶段很慢？是不是 javac 所花的时间太长？那么试试 Jikes 编译器吧，在创建 .class 文件时，它会加入额外的“ 动力 ”。这就是新兴的 Jikes，它拥有完整的 Java 源支持。（可能会引起 VerifyError，不支持所有的 Javac 选项，字节码可能不像所说的那么好，而且性能也可能受到影响。因此，在使用之前，请务必阅读使用手册。）所以说，在 JavaRanch 上对 Jikes 的讨论不像我们自制的广告那么直接，但是有的读者也明确指出，Jikes Java 编译器是设计用来加快编译速度的。知道这一点很有用，尤其是对于那些需要编译很多文件的项目更是如此。不过要清楚，虽然 Jikes 有助于加快开发进程，但是对于最后的编译，最好还是使用与在生产中

将要使用的 JVM 一起提供的那个编译器。不同的 JVM 版本会有不同的情况，所以当使用来自不同 JVM 的编译器时就可能引发问题。异常开销很大是的，异常开销很大。那么，这是不是就意味着您不该使用异常？当然不是。但是，何时应该使用异常，何时又不应该使用异常呢？不幸的是，答案不是一下子就说得清的。我们要说的是，您不必放弃已经学到的好的 try-catch 编程习惯，但是使用异常时可能会遇到麻烦，创建异常就是一个例子。当创建一个异常时，需要收集一个栈跟踪（stack track），这个栈跟踪用于描述异常是在何处创建的。还记得当代码中抛出一个意料之外的异常时，您所看到的输出来的栈跟踪吗？像下面这个：Exception in thread "main" my.corp.DidntExpectThisException at T.noExceptionsHere(T.Java:13) at T.main(T.Java:7)构建这些栈跟踪时需要为运行时栈做一份快照，正是这一部分开销很大。运行时栈不是为有效的异常创建而设计的，而是设计用来让运行时尽可能快地运行。入栈，出栈，入栈，出栈。让这样的工作顺利完成，而没有任何不必要的延迟。但是，当需要创建一个 Exception 时，JVM 不得不说：“ 先别动，我想就您现在的样子存一份快照，所以暂时停止入栈和出栈操作，笑着等我拍完快照吧。 ”栈跟踪不只包含运行时栈中的一两个元素，而是包含这个栈中的每一个元素，从栈顶到栈底，还有行号和一切应有的东西。如果在一个深度为20的栈中创建了异常，那么就别指望只记录顶部的几个栈元素了您得完完整整地记录下所有20个元素。从 main 或 Thread.run （在栈底）到栈顶，记录整个栈。因此，创建异常这一部分开销很大。从技术上讲，栈跟踪快照是在本地方法

Throwable.fillInStackTrace() 中发生的，这个方法又是从 Throwable contructor 那里调用的。但是这并没有什么影响如果您创建一个 Exception，就得付出代价。好在捕获异常开销不大，因此可以使用 try-catch 将核心内容包起来。您也可以在方法定义中定义 throws 子句，这样对性能不会造成什么损失，例如：public Blah myMethod(Foo x) throws SomeBarException {....从技术上讲，您甚至可以随意地抛出异常，而不用花费很大的代价。招致性能损失的并不是 throw 操作尽管在没有预先创建异常的情况下就抛出异常是有点不寻常。真正要花代价的是创建异常。try { doThings(). if (true) throw new SomeException(). // cos my program runs too fast} catch(SomeException e) { doMoreThings().}幸运的是，好的编程习惯已教会我们，不应该不管三七二十一就抛出异常。异常是为异常的情况而设计的，使用时也应该牢记这一原则。但是，万一您不想遵从好的编程习惯，Java 语言就会让您知道，那样做可以让您的程序运行得更快，从而鼓励您去那样做。最大堆长度在我们访问过的所有讨论组中，有关 JVM 堆的问题不断冒出。在 JavaRanch 上有一次讨论就是以“ 最大堆长度设置应该是怎样的？ ”这一基本问题开始的。在深入研究之前，让我们先复习一下 Java 运行时中内存管理的基础知识。JVM 有一片它自己管理的内存空间。对象存活（或消亡）所在的那部分空间就叫做堆空间。对象在堆空间中创建，又由 JVM 垃圾收集器在不同的时机围绕着堆空间对其进行迁移。例如，当对堆进行碎片整理（或者紧缩）时，便需要移动对象。对象在堆中也会消亡。一个死去的对象也就是应用程序再也不能访问的对象。JVM 垃圾收集器寻找这些死去的对

象，并回收这些对象所占用的空间，以便让这些空间能为新的对象所用。如果垃圾收集器无法进一步通过回收死去的对象来释放出空间，那么就说这个堆已满。一个已满的堆会引发问题。如果堆是满的，而应用程序又试图创建更多的对象，JVM 就会向底层操作系统请求更多的内存。如果 JVM 得不到更多的内存，那么分配一个新对象的这一操作就会抛出 OutOfMemoryError 异常。除非应用程序极其完善，否则那就意味着该应用程序要崩溃。那么，对此我们能做点什么呢？大多数 JVM 都有一个可选的参数，可用于指定堆所能达到的最大长度。如果堆已经达到了这个长度，JVM 就不能再向操作系统请求更多的内存。在 Sun 和 IBM 最近提供的 JVM 中，该参数可通过 -Xmx 选项指定。更老版本的 JVM 使用的是一个 -mx 选项，现在大多数 JVM 还能理解这个选项。应用服务器拥有它们自己的配置参数，可用于指定最大堆长度，这些参数通常是通过 -Xmx 参数指定的。如果没有显式地使用 -Xmx 参数，JVM 有一个默认的最大堆长度，当然这个默认值是特定于供应商和版本的。Sun 1.4 JVM 提供的最大堆长度的默认值是 64 兆字节。那么，为了达到最佳性能，最大堆长度应该为多少呢？您可能会认为“越大越好”，因为这样的话就可以避开 out-of-memory 错误，并且可以尽量多地为应用程序分配所需的内存。然而，事实证明，如果堆太大的话可能会产生大问题，这是由操作系统的工作方式所致的。现代操作系统有两种内存模式，一种是实（real）内存，一种是虚拟（virtual）内存。虚拟内存可以制造出一种假象，让人认为拥有比实内存更多的内存，这是通过使用交换文件（swap file）中的磁盘空间补充实内存来办到的，在这里交换文件充当的

是一种额外（overflow）内存。操作系统可以调出当前使用不多的页，将它们放在磁盘中，直到需要时才重新调回内存，这样便腾出了实内存（暂时地）以供他用。通过这种方式，可用的内存便表现得比实内存更大，从而允许更多或者更大的进程得以运行。相应的代价就是那些在磁盘中的页在需要时不得不重新调回内存，这样就降慢了速度。毕竟磁盘的速度比起内存来要慢得多。如果您允许堆比系统的实内存（您机器上的物理内存）还要大的话，那么这个堆就要分页。分页本身没什么问题毕竟，只是那些不经常使用的页才要被分派到磁盘中。但是，当遇到垃圾收集的时候，由于要对整个堆进行扫描，所有那些很少使用的页又要返回到实内存中，而其他的页则需要被移出实内存，送到磁盘上去，以便为那些老的页腾出空间。这是一个恶性循环，因为被移出到磁盘的页本身在堆中很可能使用得不多，作为垃圾收集的一部分，垃圾收集器要扫描这些页。其结果就是，比起真正要做的有用的事来，您需要花费更多的时间来将页移进和移出内存。 垃圾收集常常是一个应用程序的瓶颈所在。但是，如果您还要让堆大到令操作系统不得不频繁地使用分页技术以便 JVM 能执行垃圾收集，那么其结果就是一次又一次缓慢的调页动作，从而让应用程序慢如蠕动。因此，务必确保最大堆长度小于可用的系统 RAM，要为需要同时运行的其他进程考虑，尽量防止这种调页灾难的发生。