02 | 如何制定性能调优策略？

2019-05-23 刘超

你好，我是刘超。

上一讲，我在介绍性能调优重要性的时候，提到了性能测试。面对日渐复杂的系统，制定合理的性能测试，可以提前发现性能瓶颈，然后有针对性地制定调优策略。总结一下就是“测试-分析-调优”三步走。

今天，我们就在这个基础上，好好聊一聊“如何制定系统的性能调优策略”。

性能测试攻略

性能测试是提前发现性能瓶颈，保障系统性能稳定的必要措施。下面我先给你介绍两种常用的测试方法，帮助你从点到面地测试系统性能。

1.微基准性能测试

微基准性能测试可以精准定位到某个模块或者某个方法的性能问题，特别适合做一个功能模块或者一个方法在不同实现方式下的性能对比。例如，对比一个方法使用同步实现和非同步实现的性能。

2.宏基准性能测试

宏基准性能测试是一个综合测试，需要考虑到测试环境、测试场景和测试目标。

首先看测试环境，我们需要模拟线上的真实环境。

然后看测试场景。我们需要确定在测试某个接口时，是否有其他业务接口同时也在平行运行，造成干扰。如果有，请重视，因为你一旦忽视了这种干扰，测试结果就会出现偏差。

最后看测试目标。我们的性能测试是要有目标的，这里可以通过吞吐量以及响应时间来衡量系统是否达标。不达标，就进行优化；达标，就继续加大测试的并发数，探底接口的 TPS（最大每秒事务处理量），这样做，可以深入了解到接口的性能。除了测试接口的吞吐量和响应时间以外，我们还需要循环测试可能导致性能问题的接口，观察各个服务器的 CPU、内存以及 I/O 使用率的变化。

以上就是两种测试方法的详解。其中值得注意的是，性能测试存在干扰因子，会使测试结果不准确。所以，**我们在做性能测试时，还要注意一些问题。**

1.热身问题

当我们做性能测试时，我们的系统会运行得越来越快，后面的访问速度要比我们第一次访问的速度快上几倍。这是怎么回事呢？

在 Java 编程语言和环境中，.java 文件编译成为 .class 文件后，机器还是无法直接运行 .class 文件中的字节码，需要通过解释器将字节码转换成本地机器码才能运行。为了节约内存和执行效率，代码最初被执行时，解释器会率先解释执行这段代码。

随着代码被执行的次数增多，当虚拟机发现某个方法或代码块运行得特别频繁时，就会把这些代码认定为热点代码（Hot Spot Code）。为了提高热点代码的执行效率，在运行时，虚拟机将会通过即时编译器（JIT compiler，just-in-time compiler）把这些代码编译成与本地平台相关的机器码，并进行各层次的优化，然后存储在内存中，之后每次运行代码时，直接从内存中获取即可。

所以在刚开始运行的阶段，虚拟机会花费很长的时间来全面优化代码，后面就能以最高性能执行了。

这就是热身过程，如果在进行性能测试时，热身时间过长，就会导致第一次访问速度过慢，你就可以考虑先优化，再进行测试。

2.性能测试结果不稳定

我们在做性能测试时发现，每次测试处理的数据集都是一样的，但测试结果却有差异。这是因为测试时，伴随着很多不稳定因素，比如机器其他进程的影响、网络波动以及每个阶段 JVM 垃圾回收的不同等等。

我们可以通过多次测试，将测试结果求平均，或者统计一个曲线图，只要保证我们的平均值是在合理范围之内，而且波动不是很大，这种情况下，性能测试就是通过的。

3.多JVM情况下的影响

如果我们的服务器有多个 Java 应用服务，部署在不同的 Tomcat 下，这就意味着我们的服务器

会有多个 JVM。任意一个 JVM 都拥有整个系统的资源使用权。如果一台机器上只部署单独的一个 JVM，在做性能测试时，测试结果很好，或者你调优的效果很好，但在一台机器多个 JVM 的情况下就不一定了。所以我们应该尽量避免线上环境中一台机器部署多个 JVM 的情况。

合理分析结果，制定调优策略

这里我将“三步走”中的分析和调优结合在一起讲。

我们在完成性能测试之后，需要输出一份性能测试报告，帮我们分析系统性能测试的情况。其中测试结果需要包含测试接口的平均、最大和最小吞吐量，响应时间，服务器的 CPU、内存、I/O、网络 IO 使用率，JVM 的 GC 频率等。

通过观察这些调优标准，可以发现性能瓶颈，我们再通过自下而上的方式分析查找问题。首先从操作系统层面，查看系统的 CPU、内存、I/O、网络的使用率是否存在异常，再通过命令查找异常日志，最后通过分析日志，找到导致瓶颈的原因；还可以从 Java 应用的 JVM 层面，查看 JVM 的垃圾回收频率以及内存分配情况是否存在异常，分析日志，找到导致瓶颈的原因。

如果系统和 JVM 层面都没有出现异常情况，我们可以查看应用服务业务层是否存在性能瓶颈，例如 Java 编程的问题、读写数据瓶颈等等。

分析查找问题是一个复杂而又细致的过程，某个性能问题可能是一个原因导致的，也可能是几个原因共同导致的结果。我们分析查找问题可以采用自下而上的方式，而我们解决系统性能问题，则可以采用自上而下的方式逐级优化。下面我来介绍下从应用层到操作系统层的几种调优策略。

1.优化代码

应用层的问题代码往往会因为耗尽系统资源而暴露出来。例如，我们某段代码导致内存溢出，往往是将 JVM 中的内存用完了，这个时候系统的内存资源消耗殆尽了，同时也会引发 JVM 频繁地发生垃圾回收，导致 CPU 100% 以上居高不下，这个时候又消耗了系统的 CPU 资源。

还有一些是非问题代码导致的性能问题，这种往往是比较难发现的，需要依靠我们的经验来优化。例如，我们经常使用的 LinkedList 集合，如果使用 for 循环遍历该容器，将大大降低读的效率，但这种效率的降低很难导致系统性能参数异常。

这时有经验的同学，就会改用 Iterator（迭代器）迭代循环该集合，这是因为 LinkedList 是链表实现的，如果使用 for 循环获取元素，在每次循环获取元素时，都会去遍历一次 List，这样会降低读的效率。

2.优化设计

面向对象有很多设计模式，可以帮助我们优化业务层以及中间件层的代码设计。优化后，不仅可以精简代码，还能提高整体性能。例如，单例模式在频繁调用创建对象的场景中，可以共享一个创建对象，这样可以减少频繁地创建和销毁对象所带来的性能消耗。

3.优化算法

好的算法可以帮助我们大大地提升系统性能。例如，在不同的场景中，使用合适的查找算法可以降低时间复杂度。

4.时间换空间

有时候系统对查询时的速度并没有很高的要求，反而对存储空间要求苛刻，这个时候我们可以考虑用时间来换取空间。

例如，我在 03 讲就会详解的用 String 对象的 intern 方法，可以将重复率比较高的数据集存储在常量池，重复使用一个相同的对象，这样可以大大节省内存存储空间。但由于常量池使用的是 HashMap数据结构类型，如果我们存储数据过多，查询的性能就会下降。所以在这种对存储容量要求比较苛刻，而对查询速度不作要求的场景，我们就可以考虑用时间换空间。

5.空间换时间

这种方法是使用存储空间来提升访问速度。现在很多系统都是使用的 MySQL 数据库，较为常见的分表分库是典型的使用空间换时间的案例。

因为 MySQL 单表在存储千万数据以上时，读写性能会明显下降，这个时候我们需要将表数据通过某个字段 Hash 值或者其他方式分拆，系统查询数据时，会根据条件的 Hash 值判断找到对应的表，因为表数据量减小了，查询性能也就提升了。

6.参数调优

以上都是业务层代码的优化，除此之外，JVM、Web 容器以及操作系统的优化也是非常关键的。

根据自己的业务场景，合理地设置 JVM 的内存空间以及垃圾回收算法可以提升系统性能。例如，如果我们业务中会创建大量的大对象，我们可以通过设置，将这些大对象直接放进老年代。这样可以减少年轻代频繁发生小的垃圾回收（Minor GC），减少 CPU 占用时间，提升系统性能。

Web 容器线程池的设置以及 Linux 操作系统的内核参数设置不合理也有可能导致系统性能瓶颈，根据自己的业务场景优化这两部分，可以提升系统性能。

兜底策略，确保系统稳定性

上边讲到的所有的性能调优策略，都是提高系统性能的手段，但在互联网飞速发展的时代，产品的用户量是瞬息万变的，无论我们的系统优化得有多好，还是会存在承受极限，所以为了保证系统的稳定性，我们还需要采用一些兜底策略。

什么是兜底策略？

第一，限流，对系统的入口设置最大访问限制。这里可以参考性能测试中探底接口的 TPS 。同

时采取熔断措施，友好地返回没有成功的请求。

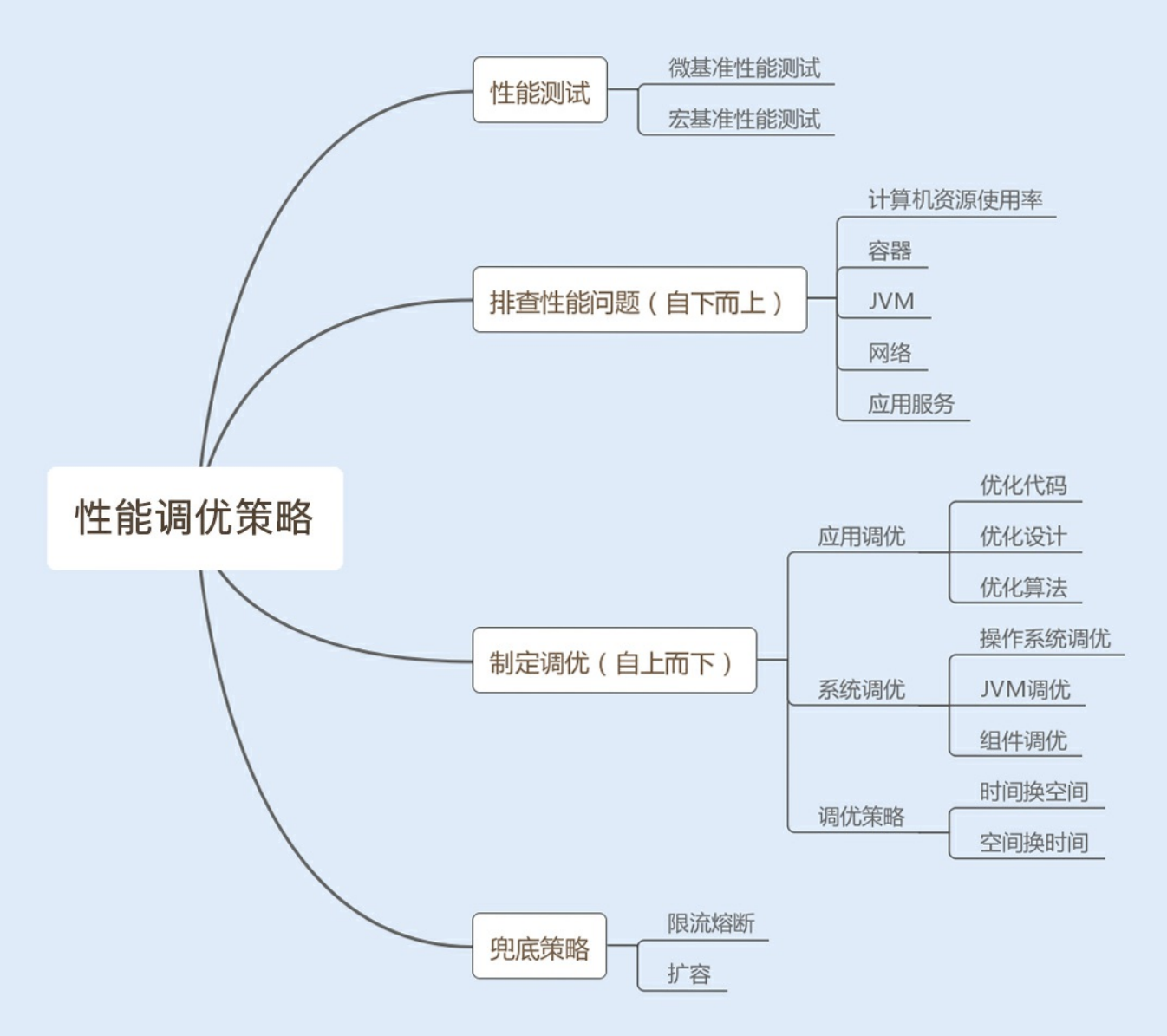
第二，实现智能化横向扩容。智能化横向扩容可以保证当访问量超过某一个阈值时，系统可以根据需求自动横向新增服务。

第三，提前扩容。这种方法通常应用于高并发系统，例如，瞬时抢购业务系统。这是因为横向扩容无法满足大量发生在瞬间的请求，即使成功了，抢购也结束了。

目前很多公司使用 Docker 容器来部署应用服务。这是因为 Docker 容器是使用 Kubernetes 作为容器管理系统，而 Kubernetes 可以实现智能化横向扩容和提前扩容 Docker 服务。

总结

学完这讲，你应该对性能测试以及性能调优有所认识了。我们再通过一张图来回顾下今天的内容。



我们将性能测试分为微基准性能测试和宏基准性能测试，前者可以精准地调优小单元的业务功能，后者可以结合内外因素，综合模拟线上环境来测试系统性能。两种方法结合，可以更立体地

测试系统性能。

测试结果可以帮助我们制定性能调优策略，调优方法很多，这里就不一一赘述了。但有一个共同点就是，调优策略千变万化，但思路和核心都是一样的，都是从业务调优到编程调优，再到系统调优。

最后，给你提个醒，任何调优都需要结合场景明确已知问题和性能目标，不能为了调优而调优，以免引入新的Bug，带来风险和弊端。

思考题

假设你现在负责一个电商系统，马上就有新品上线了，还要有抢购活动，那么你会将哪些功能做微基准性能测试，哪些功能做宏基准性能测试呢？

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

精选留言

何何何何何少侠 17

1. 新品上线需要对系统基础功能、尤其是上线涉及改动、有耦合的业务做宏基准测试，如：用户服务、商品服务、订单服务、支付服务、优惠券服务等。从而保证支撑抢购活动的服务正常运行

2. 针对抢购活动，如：秒杀 团购等促销。需要做微基准测试以验证服务是否达到预期。测试过程中需要留意诸如 qps、内存、cpu、网络带宽、线程堆栈等指标是否达标。不仅考虑单机性能，更要拓展到集群时性能的阈值能达到多少从而给出更加准确的性能测试评估报告

3. 多说一句：此外还要考虑服务的质量，要测试出抢购活动的瓶颈在哪儿从而应对即将到来的大促活动，以方便开发、运维团队制定更好的如服务限流、降级、动态伸缩等方案。

2019-05-24

作者回复

回答的很全面，赞一个

2019-05-24

业余草

4

总结的很好，期待后面的实战内容！！！

2019-05-23

昨夜星辰

4

新上线的系统作宏基础测试，抢购活动作微基本测试

2019-05-23

木偶笨笨

3

感觉论题有一点过于发散，讲到限流熔断这些内容了，我理解限流熔断实际是架构师的事情，是不是另开一课再讲。这门课focus在调优方法、工具、技巧，以及相关理论比如jvm、多线程原理是不是会更合适。

2019-05-23

作者回复

感谢你的建议。我相信很多同学跟你有一样的想法，那就是赶紧学会使用性能排查工具，性能如何监测分析，如何解决性能问题。

由于不同的性能问题，性能排查以及调优都是不固定的，所以在后面的一些章节中，会有一些结合实际场景来进行性能排查的实战。

在大家了解一些理论性的知识点以及基础之后，也有专门一讲来讲述性能监测工具、调优工具的使用，所以大家保持耐心，切记心急吃不了热豆腐。

在这里我们强调了即使我们性能测试做的再好，兜底策略是一定要做的，兜底也是性能调优的一部分。试想下，我们的性能调优做的再好，系统同样存在极限，当系统达到极限，系统肯定出现性能瓶颈。

在学习成长的过程中，我们切忌将知识点局限于某个层级，或者将自己局限于某一种语言。例

如线程池的大小设置，其实也是一种限流的方式，所以限流熔断并不只是局限于架构这块的内容。

我们要做性能调优最重要的目的是什么？在我看来就是为了避免发生线上事故，如果发生线上事故，也是要避免线上大面积事故。所以性能调优做的再好，系统也是存在极限的，兜底策略是系统的保护伞，特别在高并发的系统中，降级/熔断/限流成为保证系统性能稳定性的重要环节。

2019-05-23

SlamDunk 2

如果我们的服务器有多个 Java 应用服务，部署在不同的 Tomcat 下，这就意味着我们的服务器会有多个 JVM。
不同tomcat也可以使用同一个jrm下的同一个jvm呀，为什么这里要说会有多个jvm呢？

2019-06-07

-W.LI- 2

抢购秒杀，感觉架构层面的优化比较多吧，尽量缩短链路，缩短响应时间，没有依赖的服务串行优化为并行。或者本地持久化后保证最终一致性。查询商品详情，下单支付这些接口宏观测试，内部的比较占用系统资源的关键代码(占用IO资源，逻辑复杂消耗CPU资源等)做微测试。还有就是需要做限流兜底，读服务采用合理的缓存策略等。

2019-05-24

CharlesCai 2

期待作者的新内容！朗读者的声音好听又专业！提一个小功能，网页版能不能实现一下标记或做笔记的功能。

2019-05-23

编辑回复

接收成功！谢谢你的建议。

2019-05-24

阿厚 1

多少别人一天没有解决的问题，被我用一部分一部分注释代码，半小时解决了。

2019-06-04

作者回复

如果能用排除法去解决问题，是一个比较好的方式。不过很多线上事故，在线下是无法重现的，这个方式就比较难派上用场了。

2019-06-05

zhangtnty 1

老师好, 我理解文中题目中抢购的不同实现方式是微观调优，综合考虑上线后流量峰值等可为宏观调优。
老师在文中提到的降级和限流是日常关键的一环，老师把它说成兜底，我常理解为"保命" 。也希望老师对于降级和限流可以展开分享一篇。各种调优最终都会有极限。

2019-05-24

作者回复

同学你好，你理解的很到位，兜底就是保命，但高于保命，我们不仅仅需要保证系统不挂掉，还要保证流量范围内的请求是正常的。微基准性能测试可以理解为对某块代码进行测试，包括对不同实现方式的性能测试比较。

后面我会在实战中讲到限流、降级的实现和使用，由于这个属于优化的辅助功能，不做具体实现方式的讲解。如果对相关知识感兴趣，可以留言保持沟通。

2019-05-24

建国 1

老师我又来了，两个问题，1.您在这节中介绍的那么多的知识点在后面的课程中都会逐个讲解到吧 2.有没有nginx调优呢，因为我们给客户部署时发现，用阿里云的SLB和自己搭建的nginx，某个接口响应时间差10+倍

2019-05-24

作者回复

你好 建国，欢迎多提问。我先回答你的第一个问题，前面两讲中，一方面，是让你对性能调优有一个全面的认识: 调优的目的是什么，有没有指标可衡量，如何发现性能问题，发现后，我们有什么策略可以调优; 另一方面，我多次强调了基础知识以及调优的思维方式的重要性。所以接下来我将从基础讲起，再到高级篇，学会高性能编程的同时，总结出一惯的调优思维方式。从中很多章节中会有结合实际场景使用到一些测试工具以及性能调优工具。除了这些，我还会在最后用实战的方式来为你讲解实际业务场景下的调优。

从这个专栏的目录来看，没有专题专门讲nginx的调优，nginx如果只是作为转发，由于nginx是基于事件驱动模型实现的web请求转发，使用异步处理方式来避免阻塞，对性能损耗应该不大。如果用lua脚本做了一些逻辑判断，或者限流等等，这个是有损的，会带来很大的损耗。

2019-05-24

丿北纬91度灬 1

老师您好，思考题中，新产品中的抢购活动，针对抢购的商品数量、支付等内容进行微基性能测试，对于商品数量、支付这些比较关键的代码，多线程高并发下商品数量的读写，数据同步，支付的安全等需要精准的测试，而宏基准性能测试更是偏向于整体的业务逻辑，针对整个新产品的整体功能，例如秒杀活动的从开始抢购到成功支付，或者开始抢购到未抢购到商品等流程进行宏基准性能测试，我这样理解对嘛老师

2019-05-23

作者回复

这位同学，你理解的很好。微基准测试我在这里纠正一点，包括进入抢购页面、提交订单、支付调起，再细一些包括排队等待功能、库存扣减的分布式锁功能、幂等性校验等。

2019-05-24

进步慢是一种罪 1

抢购活动（秒杀）作为微基准测试，商品详情页浏览，支付，支付后的通知等做宏基准测试。

2019-05-23

ANYI

1

hi，老师，入职新公司，直接派去客户现场调优，有一份压测报工，知道是哪些场景性能有问题，但对于业务不熟，只有一堆代码；该如何快速进入；

2019-05-23

作者回复

你好 ANYI，建议可以先对一个一个小模块进行性能测试和调优。先对一些代码性问题进行优化，例如之前有同学提到的，合并多次请求，减少多次数据库操作，优化sql（优化join以及索引），优化Java基础代码（集合的合理使用，序列化的优化）等等，先完成这些基础性优化。

在这基础之上，我们再去针对一些业务进行优化，例如业务存在高耦合，我们可以解耦业务，使用一些好的设计方法。通过这种方式逐步了解整个系统的业务以及架构。

代码层级优化之后，我们可以考虑调优JVM、容器以及操作系统，我相信代码层的优化可以满足大部分的性能优化需求，其他的性能调优则是满足一些特殊的场景下的高性能需求。

2019-05-23

etdick

1

老师，现在的微服务架构，一台物理机部署了多个微服务。每个服务相当于一个JVM。如何调优？

2019-05-23

作者回复

你好 etdick。

首先，在做性能测试时，我们应该单独部署测试每个微服务的性能，尽量排除服务之间的干扰，先完成单个服务的性能调优；

其次，模拟线上环境下多个服务部署，根据实际业务来模拟多个服务的高峰值的性能测试，如果服务与服务之间存在性能上的互相干扰，且属于不同的业务，我们应该考虑实际生产环境中，两个业务场景是不是存在相同的峰值期，若是，则需考虑分开不同服务器部署或根据需求进行服务降级。

除此之外，我们还可以设置JVM参数来调优各个JVM的内存分配以及垃圾回收。我们知道两个JVM会互相产生影响的主要原因是对CPU的使用情况，而垃圾回收频率是抢占CPU的主要因素。我们可以调优内存分配降低垃圾回收频率，或设置合适的垃圾回收器。由于不同场景具体的分配调优方式不同，我们将会在之后的内容中讲解到。

2019-05-23

Jxin

0

请教大佬个问题。函数内部打印日志，日志的文本为中文长篇描述。是不是每次调用该函数都会有new这个日志文本的开销。毕竟从字节码来看，方法内部的字符串不会被纳入常量池。

2019-06-03

作者回复

是的，但这种打印日志的字符串一般很少被长时间引用，打完日志对象很快会被回收。

2019-06-03

Geek_ebda96

0

老师，这句话

这就是热身过程，如果在进行性能测试时，热身时间过长，就会导致第一次访问速度过慢，你就可以考虑先优化，再进行测试。

指的优化是优化JVM的一些参数，还是指优化代码呢？如果是优化代码，热身时间过长，有没有例子能够说明一些，第一次查询数据先放入缓存这个算吗？

2019-06-02

作者回复

可以通过设置CompileThreshold参数降低执行方法次数阈值来提前预热代码，也可以通过调用WarmUpContextListener.invoke方法指定需要预热的方法，当然也可以在启动时提前写个循环或多线程调用该方法。

我们还可以使用一些工具来预热，例如之前有同学提到的JMH。

2019-06-03

Maxwell

0

老师，最近段时间经常报端口被大量CLOSE_WAIT，重启后过半天又重现，以前未出现过，一般有哪些排查方式

2019-06-01

Geek_ca1254

0

有一个问题，老司机，现在大部分的应该是没法保证测试环境的机器和线上环境的机器配置是一致的。从而测试做出的性能测试报告其实是不准确的。是不是可以有一个什么内存与性能存在一个平衡点的比例公式去衡量？

2019-05-30

作者回复

你好，这个很难去衡量这个比例，内存可以，但cpu的性能未必是正比。

2019-05-30

ABC

0

javap -c 的输出如下:

```
Compiled from "HelloWorld.java"
public class HelloWorld {
public HelloWorld();
```

Code:
0: aload_0
1: invokespecial #1 // Method java/lang/Object."<init>":()V
4: return

public static void main(java.lang.String[]);
Code:
0: new #2 // class java/util/LinkedList
3: dup
4: invokespecial #3 // Method java/util/LinkedList."<init>":()V
7: astore_1
8: aload_1
// ... 省略add方法的字节码
85: iconst_0
86: istore_2
87: iload_2
88: aload_1
89: invokevirtual #16 // Method java/util/LinkedList.size:()I
92: if_icmpge 117
95: aload_1
96: iload_2
97: invokevirtual #17 // Method java/util/LinkedList.get:(I)Ljava/lang/Object;
100: checkcast #18 // class java/lang/String
103: astore_3
104: getstatic #19 // Field java/lang/System.out:Ljava/io/PrintStream;
107: aload_3
108: invokevirtual #20 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
111: iinc 2, 1
114: goto 87
117: getstatic #19 // Field java/lang/System.out:Ljava/io/PrintStream;
120: ldc #21 // String ok
122: invokevirtual #20 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
125: return
}

2019-05-29

ABC

0

老师你好，我看了kevin的例子，自己也去写了一下，没明白使用for(;;)循环是怎么每次都遍历了的呢？

代码如下:
import java.util.*;
public class HelloWorld {

```
public static void main(String[] args) {
LinkedList<String> list = new LinkedList();
list.add("1");
list.add("2");
list.add("3");
list.add("4");
list.add("5");
list.add("6");
list.add("7");
list.add("8");
list.add("9");
list.add("10");
list.add("11");
for (int i=0;i<list.size();i++ ) {
String item=list.get(i);
System.out.println(item);
}


System.out.println("ok");
}
}
```

2019-05-29

作者回复

你好，明天的第五讲中，会详细讲到，请留意一下

2019-05-29