



下载APP



01 | 性能建模设计：如何满足软件设计中的性能需求？

2021-05-17 尉刚强

性能优化高手课

[进入课程 >](#)**讲述：尉刚强**

时长 16:08 大小 14.78M



你好，我是尉刚强。今天是课程的第一讲，我想先和你一起来学习下基于性能的性能建模设计方法。

基于性能对软件进行建模和设计的目的是什么呢，其实是为了保证软件产品最终交付的性能，跟一开始的设计预期相匹配。然而，在实际的软件建模和设计过程中，很多人其实都忽视了性能的评估分析，导致生成的软件性能差，被客户频繁投诉，甚至有可能导致产品失败，给公司带来严重的后果。

所以这节课，我们就来看看如何在软件设计阶段做好性能的评估分析，通过一定的方法前识别出软件设计中潜在的性能问题，并指导优化设计，从而更好地满足软件设计中的性能需求。



学会了这个方法之后，你不仅可以提前获取产品的性能预估表现，还可以用它来指导软硬件资源的选型设计，甚至在一些场景下，如果客户对产品要求的性能目标不合理，你也可以利用这个方法推动他调整性能目标。

那么具体是什么方法呢？答案就是**软件执行模型**和**系统执行模型**这两种对系统建模的方法思路。

软件执行模型是一种静态分析模型，一般不需要考虑多用户和资源竞争等动态情况，我们可以用它来分析评估系统的理想响应时间；而系统执行模型则是需要重点考虑多用户、资源竞争等情况的动态分析模型，我们可以利用它来分析和评估系统吞吐量。虽然这两个模型的关注点不同，但我们可以借此识别出软件设计中存在的一些性能问题。

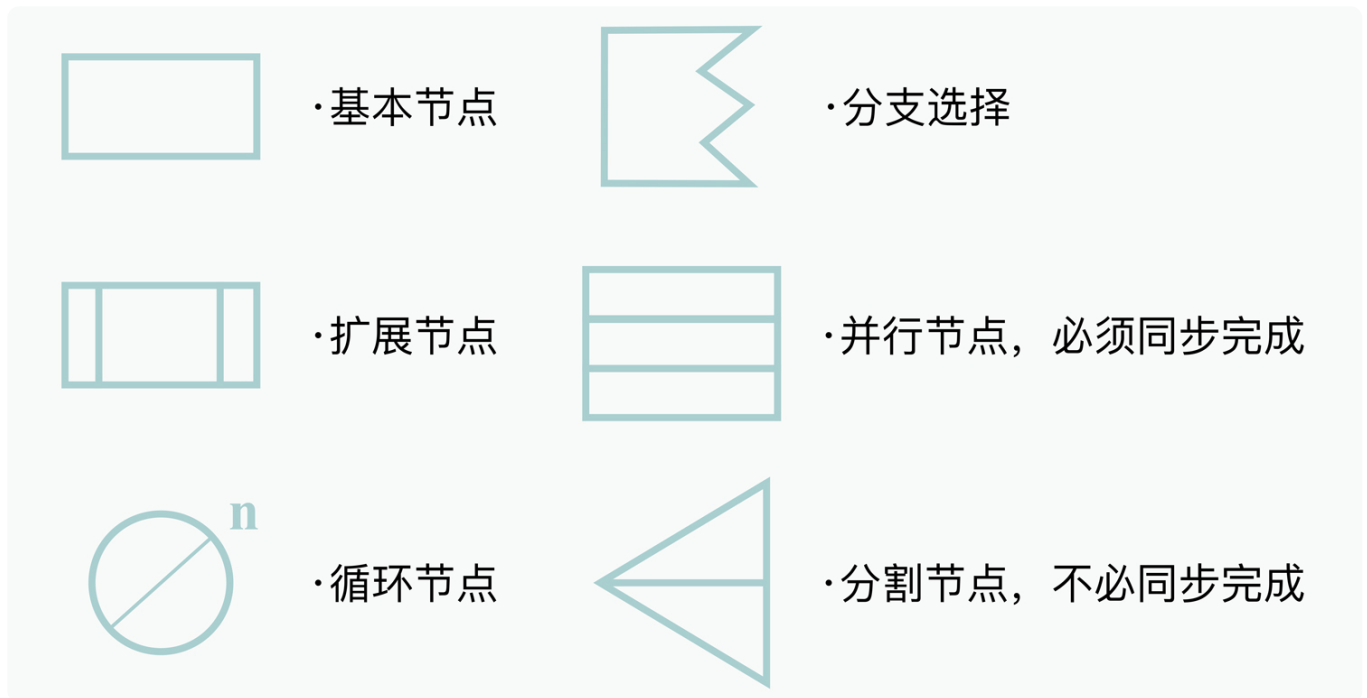
所以接下来，我们就从软件执行模型开始，来看看如何在软件建模与设计的过程中，最大化地满足性能需求吧。

软件执行模型

前面我说过，软件执行模型是一种静态分析模型，我们在分析过程中主要关注执行步骤和流程即可。而传统的 UML 时序图承载的与性能不相关的额外信息比较多，所以这里呢，你可以选择使用**执行图**来表示软件执行模型。

执行图是一种对软件执行过程、步骤的可视化描述手段，你可以通过推理来计算这些步骤流程的开销，从而帮助你预估软件的性能表象。

不过在使用执行图评估和分析性能之前，我们还需要知道执行图的大概结构。它主要是由节点和箭头组成的，这里我列举了一些常见的节点类型，你可以先熟悉下：

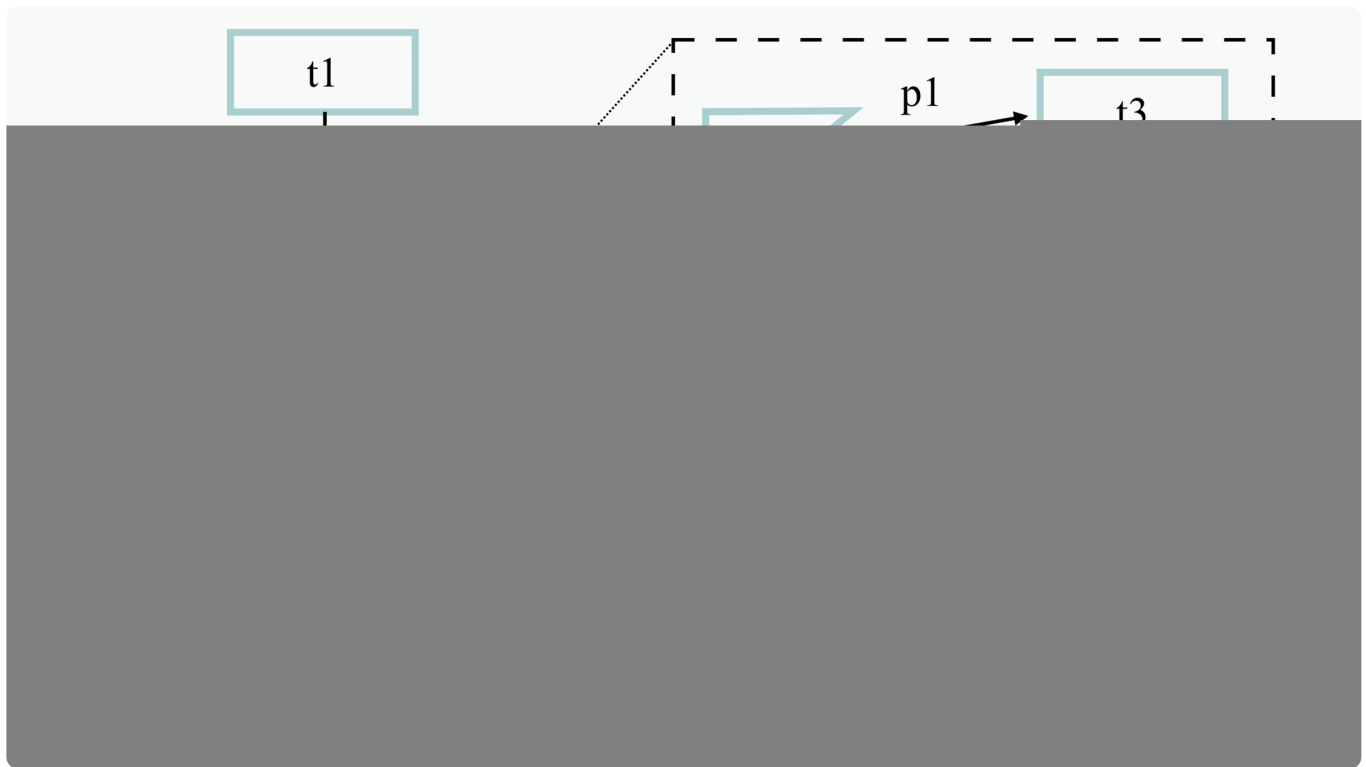


其中，扩展节点代表的是还需要进一步被细化的节点，它可以用另外单独的执行子图做进一步的描述；并行节点代表着多个并行执行的节点单元，只有当所有的节点都执行完毕之后，才能执行后续的操作；分割节点代表着有多个异步执行的节点单元，它并不需要等待所有节点执行结束，就可以开始后续的操作。

好了，理解这些节点类型的语义之后，我们接下来看看如何用这些节点类型来表示一个执行图。

简单的软件执行模型是什么样子的？

下面给出的是一个简单的执行图，节点中的数字代表的是一个权重，你可以用它来表示 CPU 执行时长、数据库操作次数、磁盘操作次数等，这里我们先假设它代表的是 CPU 执行时长，以便于理解接下来的计算过程。



现在，我们具体来看看这个执行图的操作步骤：

第一步是基本节点，执行开销为 $t1$ 。

第二步是循环节点，其中 n 代表的是循环次数，同时你可以看到这里的循环体是一个扩展节点。扩展节点可以使用另外一个执行子图来表示，上图中的扩展执行子图是由一个分支选择节点组成的，其中包含的两个分支节点开销分别为 $t3$ 和 $t4$ ，而这个分支选择节点本身开销为 $t2$ 。

第三步是并行节点，它包含了三个基本节点，执行开销分别为 $t5$, $t6$, $t7$ 。

最后一步也是基本节点，开销为 $t8$ 。

然后我们就可以根据这个执行图，估算出采用这种软件设计后的平均处理时延为：

$t1+t8+\max(t5, t6, t7)+n((p1 \times t3)+(p2 \times t4)+t2)$ 。同样的，你还可以估算出最短和最长的处理时延，分别如下：

最短处理时延： $t1+t8+\max(t5, t6, t7)+n \times (\min(t3, t4)+t2)$ 。

最长处理时延： $t1+t8+\max(t5, t6, t7)+n \times (\max(t3, t4)+t2)$ 。

这里你可能有一个疑问，**在软件设计阶段怎么能估算出这些值呢？**

是这样的，首先你不能期望获取准确的评估值，因为在开始软件设计的阶段，你能获取的前期测量信息不仅有限，而且可能不是非常准确的。

但是，你可以根据软件设计去估算出一些值，比如根据业务逻辑分析数据库的操作次数，再结合数据库性能指标，来估算开销等。通过这样的估算，你就已经能够提前识别出一些性能设计上的问题了。

如何利用软件执行模型分析评估系统性能？

OK，现在我们已经了解了对于软件设计来说，执行图是一种能够有效且方便地分析评估处理时延的手段。那么接下来，我就通过一个真实的人工智能对话引擎的软件设计案例，来带你详细了解下，使用执行图分析性能并引导软件设计的过程。

注意：这是一个被大幅简化后的真实案例，其中介绍的相关测量评估数据并不是真实的，只是为了阐述问题而已。

下图是这个对话业务的语义树模型，这个语义树模型代表的是人类对话场景的一个模拟过程，就像两个人在聊天过程中，一方可能会接着对方最近几句中的其中一句进行回复。语义树中的每个节点代表着一个语义，然后智能对话引擎就可以通过一个智能计算模型，来判别这个语义是否匹配。



其中，黄色节点代表着已经识别的用户对话上下文语义，红色虚线箭头代表用户对话发生的顺序，而绿色节点则是接下来用户对话可能发生的语义。当然了，在一个实际的对话执行引擎中，包含的功能其实有很多，这里我们先就一个简化后的智能对话引擎，来查看下它的执行图：

从图上你可以看到，对话引擎在执行过程中，首先要进行语音识别，将用户的语音转换为文字，然后转换为词向量。紧接着就需要遍历执行所有智能语义模型，来计算匹配度，然后根据一定的算法选择出最佳匹配的语义。最后还需要查询相关的数据库，构造用户的响应文字信息，并转换成语音发送给用户。

到这里，你就可以使用前面介绍的模型求解的方法，来预估下该智能对话引擎的响应反馈时延。

在这个对话执行引擎中，语音识别、词向量转换、构造回复（多次查询或修改数据库）、文字转语音都是相对比较确定的，粗略估算分析可以控制在 50ms 以内。而中间循环使用各个节点的语义计算模型，需要的时间开销会比较大。

假设你经过初步的测量，单个的语义分析模型的计算，需要的时间开销约为 10ms，而针对一些复杂的对话业务场景，系统中待识别的语义计算模型数（假设为 n ），可能有几百个。所以，这部分处理时间可能已经超过了 1 秒钟，无法满足用户的响应时延性能要求。

那么，为了更好地解决这个问题，你就可以在设计时采用并发架构，将语义计算模型任务拆分到多个核上来运行。这样，调整后的软件执行图就如下图所示：

这里我们可以看到，调整后系统会同时启动 6 个并发任务来执行语义计算模型，由于每个语义计算模型的执行时间比较接近，所以在静态分析的过程中，理论上可以将中间循环的处理时间提升近 6 倍，从而就很好地满足了软件的性能需求。

但是，在系统真实的运行过程中，真的可以达到理论上的性能提升效果吗？

其实不一定，如果这个软件系统是运行在一个仅有两个 CPU 硬件核的机器上，那么程序中虽然启动了 6 个并发任务，其最大的加速比也只能到 2，所以不可能达到理论上 6 倍处理

时延的提升。

由此你肯定也就发现了，**不考虑软硬件资源使用状况的软件执行模型，是存在一定局限性的**，所以我们还需要通过一定的手段，来减少性能评估与真实运行时的偏差，而这就是我接下来要给你介绍系统执行模型的原因。

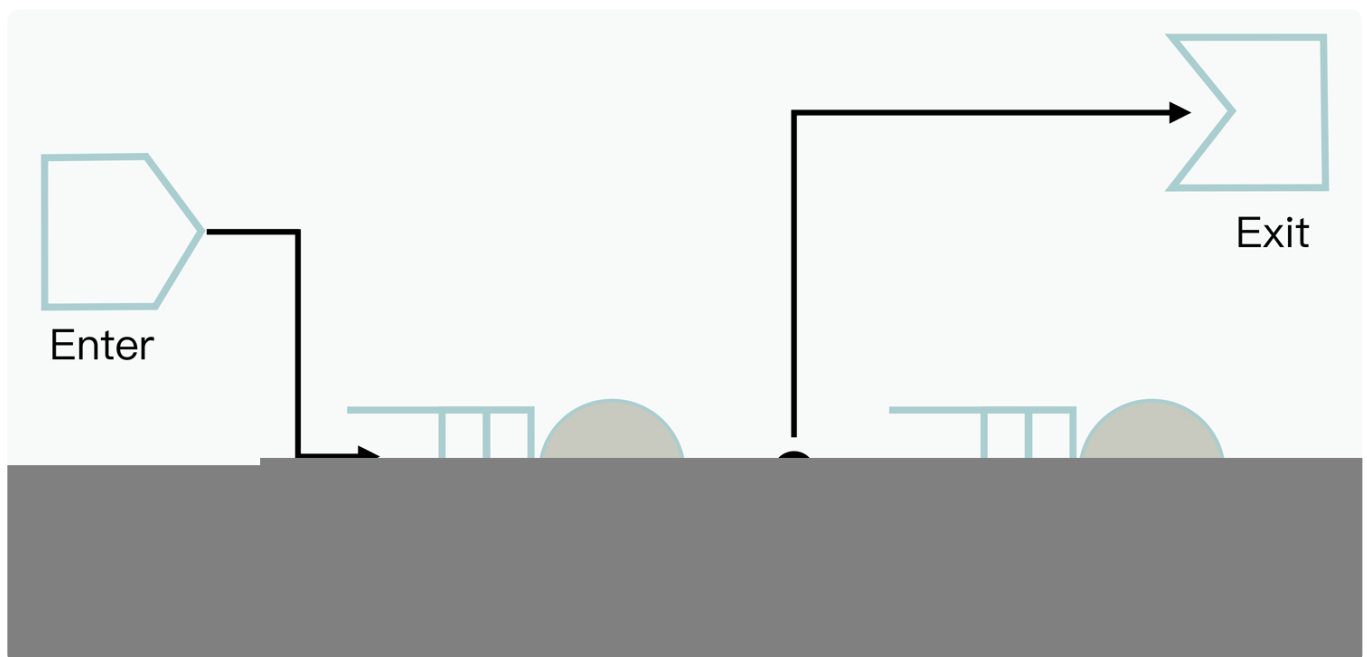
好，下面我们就具体来看看吧。

系统执行模型

我们开发的所有软件都是运行在一系列硬件资源上的，比如 CPU、内存、磁盘、网络等，而系统执行模型作为一种动态分析模型，实际上就是**针对多用户和硬件资源竞争场景下的动态建模过程**。

这也就是说，我们在使用系统执行模型对软件运行态进行建模的过程中，其实可以将系统中的关键资源抽象成一个队列服务器模型。其中，服务器代表具体的硬件资源，而队列则代表处于排队状态的用户作业。这样，针对存在排队处理的业务逻辑，我们在数据中就可以使用 **QNM**（Queuing Network Model，排队网络模型）进行分析。

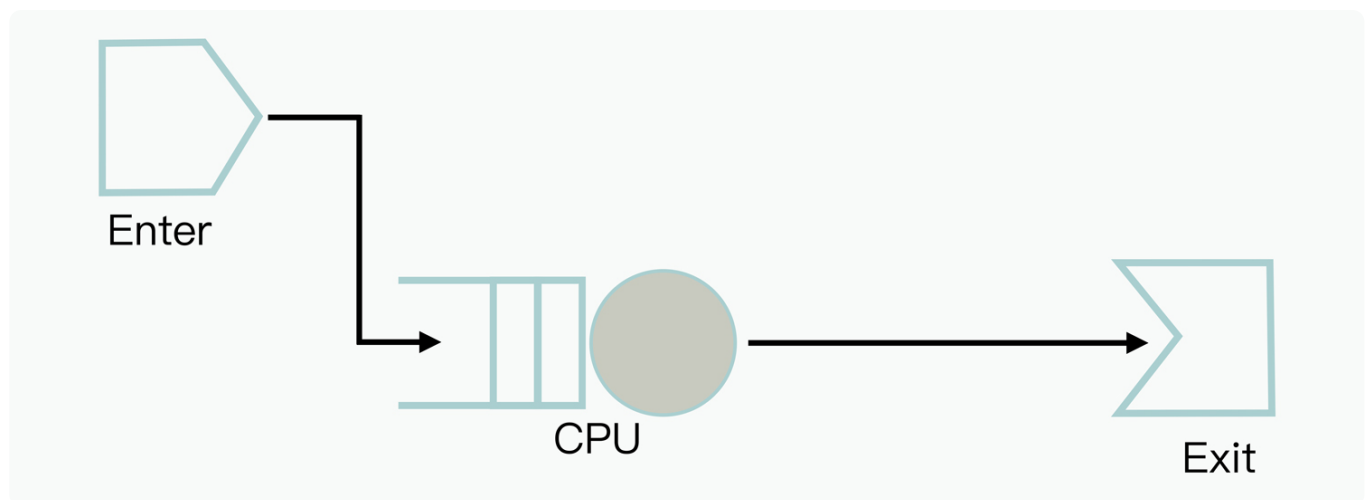
QNM 是针对排队问题的一种数学建模分析方法，我们可以借助这个模型，来帮助模拟与分析系统的运行态性能，从而可以有针对性地对软件设计进行调整和优化。下面展示的就是一个比较简单的 QNM 拓扑图：



在这个拓扑图中，描述的过程是当用户作业到达后，首先排队访问 CPU，紧接着的流程是一个选择逻辑（图上的黑点），这里有两个分支，一个是直接退出，另一个是接着排队访问磁盘，然后再继续访问 CPU 资源。QNM 模型支持的元素类型比较多，比如还有延迟等待、普通队列等，不过在互联网服务场景下，一般不需要构造特别复杂的 QNM 模型，所以这里我就不去深入介绍了。

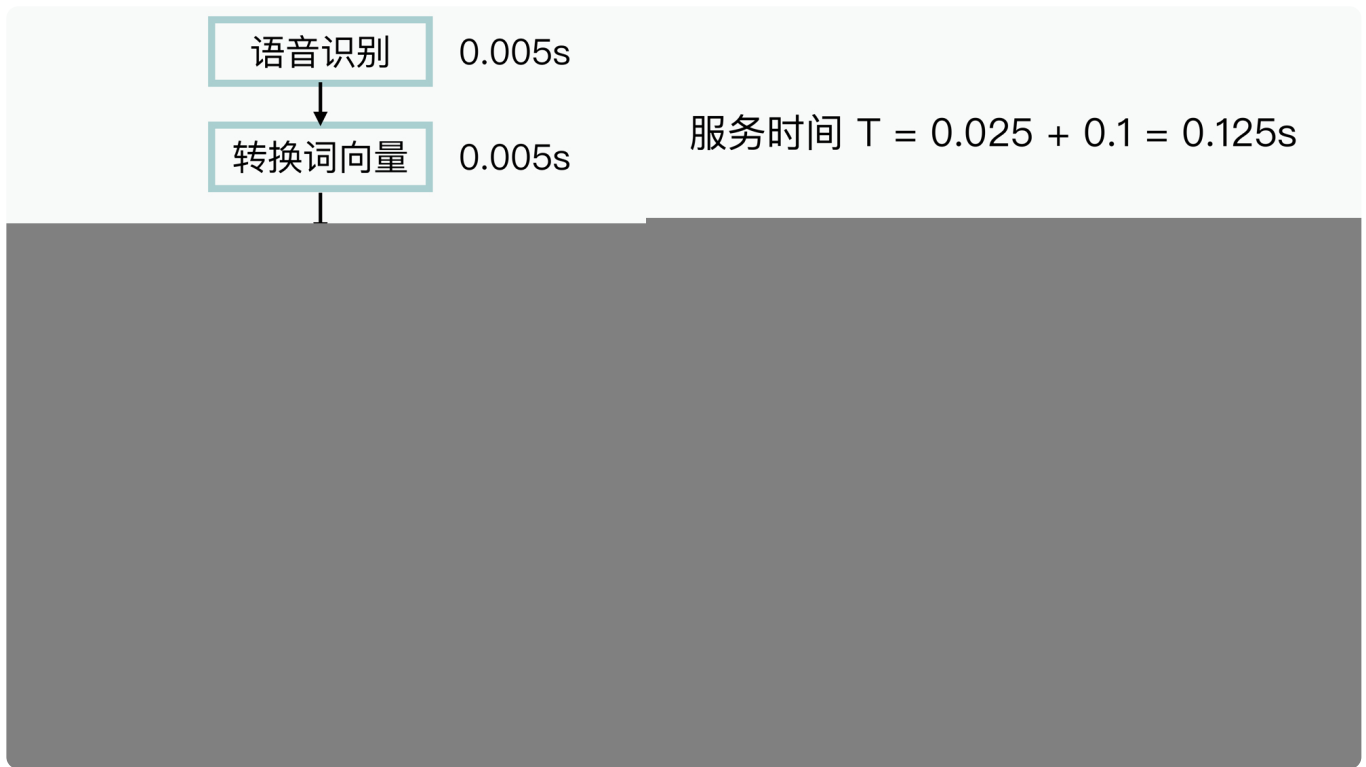
这里你需要注意的是，在进行系统执行建模时，除了处理 CPU、磁盘等硬件资源可以使用排队服务来建模外，一些外部依赖的数据库、第三方服务等，也满足排队与服务器模型的特性，因此同样可以使用这种方式来建模。

那么现在，我们回到前面介绍的智能对话引擎案例当中。因为这个系统的核心关键资源是 CPU，所以针对这个 CPU 资源，我们建立的 QNM 模型如下：



可以发现，这是一个最简单的、只考虑 CPU 资源排队竞争的 QNM 模型，当系统接收到业务请求后，经过 CPU 的排队并处理之后，就结束退出了。

接下来，我们就可以基于之前的执行图，然后基于测量或估算的方法，来计算获取的单个对话任务的服务时间，如下图所示：



那么，在这个对话引擎的执行过程中，我们就可以基于一些前期测量数据，预估出每个节点的执行时间。这里假设用户平均需要识别的智能语义模型数目 n 为 10，然后我们可以根据图中的语义逻辑，来推算出单个用户回复需要占用的 CPU 服务时间为 0.125s。

不过这里还有一个问题：在真实的业务负载场景下，该引擎的对话响应时延，是否也可以达到理论值 0.125s 呢？其实很多时候都是不能的，这是因为当系统中同时存在很多个对话请求时，会因为竞争使用 CPU 资源而存在排队等待的情况，从而就会增加对话响应时延。

所以接下来，我就借助系统执行模型，来带你分析评估下这个智能引擎的动态运行性能，这样你就会更清晰地了解到，考虑资源竞争的系统执行建模分析，与软件执行静态建模之间存在的差异。另外，我还会对比该系统在不同 CPU 核数的场景下，其平均的响应时间表现是怎样的，这样你就会明白，**不同软件与硬件选型也可以直接反映在性能评估的结果值当中。**

这里首先你要知道的是，**系统执行模型所做的性能评估分析，通常只能分析系统处于稳态情况下的性能表现**，毕竟在非稳态的场景下，我们很难可以准确地评估性能，而且分析的意义也不大。所以，针对这个对话执行引擎来说，我们可以先假设用户对话到达速率在某个恒定速率，然后再来分析对话响应时延。

补充：在该对话引擎中，我们设定所有的用户对话请求处理都是可并行的，所以可以使用 QNM 来进行分析。

好，下面我们就来看看这个具体的计算过程，如下图所示，其中系统到达速率 5 作业 /s，代表的含义是每秒钟会有 5 个用户对话请求到达：

系统到达速率 $\lambda = 5$ 作业/s，服务时间 $T = 0.125s$



图中左边的数学公式，是 QNM 模型的通用数学分析公式（具体的公式证明你可以参考下 [QNM 的相关资料或论文](#)，这里我就不展开介绍公式的原理了），这样接下来，你就可以计算出不同 CPU 核数下的平均响应时延值。

不过，查看上面的计算结果，你可能会发现两个比较奇怪的现象：

1. 单核场景下，系统响应时间 0.454 远大于 CPU 执行时间 0.125。这其实是因为当系统处于动态运行过程中，有可能会由于多个任务竞争使用 CPU 资源，从而引发了排队的时延问题。
2. 在增加 CPU 核数后，响应时间的提升速度明显大于并发提升的速度。这其实是因为缓解了排队现象，从而导致响应时延变少。

所以到这里，我们应该能够发现，相比软件执行模型，使用系统执行模型来分析和评估系统的响应时间，才能够帮助我们更准确地分析动态负载场景下的性能表现，从而支持在软件设计的调整和优化。

小结

今天这一讲，我重点给你介绍了软件执行模型和系统执行模型两种对系统建模的方法思路。这里你需要明确一点，就是在使用软件执行模型来对系统性能进行静态分析时，会相对容易一些，但它只能分析单个用户场景下的理论性能表现，而使用系统执行模型则可以帮你动态分析多用户和资源竞争场景下的性能表现。

而且在课程中，我并没有非常深入地去讲解这两种模型的所有建模细节，这是因为在软件设计阶段，你首先应该有基于性能进行建模与分析的意识，然后才是去学习如何正确使用这些性能建模的方法。

所以，我希望你在学习了今天的课程之后，能够在实际的软件设计过程中，提前识别出性能关键点，并寻找到合适的性能建模方法来实现性能评估。另外，你还可以在软件生命周期中，去持续矫正这个系统性能模型，从而达成持续地支撑后续各种设计与实现优化的目标。

思考题

在互联网云服务场景中，当应用服务器实例 CPU 负荷，超过一定门限（如 80%）就会触发弹性扩容，而不会等到满负荷时候才触发，这是为什么呢？

欢迎在留言区分享你的答案和思考。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [开篇词 | 与我一起修炼，成为一名软件性能大师吧！](#)

下一篇 [02 | 并行设计（上）：如何利用并行设计挖掘性能极限？](#)

精选留言 (3)


写留言



没想好 
2021-05-20

扩容本身也需要时间，满负荷触发扩容会导致扩容这段时间的数据丢失，所以往前调一点，哪怕在扩容时新来的数据也有地方放
展开 ▾



danger boy 
2021-05-19

非常喜欢这个课程，生动有趣
展开 ▾

作者回复: 谢谢



王博
2021-05-17

讲的不错，有收获，赞！
展开 ▾

作者回复: 谢谢

