18 | Benchmark测试（下）：如何做好宏基准测试？

2021-06-26 尉刚强

《性能优化高手课》 课程介绍 >

讲述：尉刚强
时长 17:57 大小 16.45M

你好，我是尉刚强。在上节课，我给你介绍了如何才能做好微基准性能测试，而这节课的主要关注点则是如何才能做好宏基准级别的性能测试。

现在我们已经知道，**宏基准性能测试的目标是获取软件系统级的性能基线水平，以此支撑系统基于性能去弹性扩展、部署运维等，或是指导系统设计层面的性能优化。**比如说，针对互联网在线数据产品或在线视频直播产品，当为某个跨年晚会提供服务时，我们就可以根据性能基线水平提前扩容，部署对应规模的服务集群，应对流量峰值，以此避免由于瞬间峰值而导致整个业务瘫痪的尴尬情况。

然而，对大型软件系统进行宏基准测试，其实是一件非常复杂的软件工程活动，我举几个例子你就明白了。

比如，针对互联网服务，其用户服务很多样、复杂，所以很难模拟用户行为；而针对一些大型嵌入式设备，在业务场景中需要与很多具体设备进行连网，但我们知道，测试设备是非常昂贵的资源。

所以在这堂课上，我会来帮你分析系统级别的性能测试，都面临着哪些问题与挑战，以及要如何使用比较低的成本并正确获取系统的关键性能指标，从而更好地支撑你的宏基准性能测试能力。

系统级性能基准测试挑战分析

那么首先，我就来带你分析下对于系统级的性能基准测试来说，目前都存在哪些问题和挑战，以此帮助你理解做好宏基准测试方法论的背后原因。

挑战一：全量系统规模大，不易复制

随着业务与技术的逐步演进，从最初的单体架构演进至 SOA 架构，再到现在的微服务架构，软件的系统架构越来越复杂；而软件从原来的单机物理部署演进至集群部署，再到后来的云部署，部署形态也逐渐多样化；另外，随着分布式业务的增多，分布式协同也更加复杂，这就导致了系统运行状态更加多变且不可预测。

因此，在真实运行的产品系统之外，构造一个同等规模的全量被测系统的成本会非常大！

挑战二：引入多机 Cache 机制，仿真业务性能难

除此之外，在互联网业务场景中，当系统性能不能满足用户的需求时，我们确实还可以通过弹性计算来扩展系统的性能。但是在系统应用弹性扩展的过程中，一方面，弹性扩展引入的负载均衡机制增加了系统测试的复杂度；另一方面，弹性扩展能力也比较容易把系统一些潜在的性能瓶颈隐藏起来，就导致我们不能在项目前期及时发现问题。

所以为了优化软件的服务性能，在系统中的很多环节都会引入增加 Cache 机制。比如，在业务中经常使用分布级 Cache（如 Redis、MemCache 等），对数据库请求结果进行缓存；或使用内存级 Cache（如 Caffeine 等）缓冲一些临时数据；甚至在系统网关服务或 Nginx 上也可以设计缓存机制，来缓存部分页面请求数据等。

可是，对于系统的基准性能测试结果来说，**引入的多级 Cache 机制，也更容易导致其不能准确反映真实的系统性能。**

挑战三：引入安全机制，导致仿真用户难

在互联网业务中，系统的安全性越来越重要。因此，我们在软件中也会引入很多复杂的安全交互机制。可是，在采用安全交互机制后，支撑保护系统不容易被攻击的同时，也使得构造性能测试场景和数据的难度大大增加。

挑战四：业务场景多样分布，测试难复制

最后，在系统运行的过程中，由于接入的客户端种类越来越多，处理的请求也更加多样。所以单从这个维度上来讲，去仿真软件产品在线业务的请求分布会太过复杂，甚至是不现实的。

总之，基于上述的分析，你可以发现，从全系统级的端到端实施性能基准测试，需要处理解决的问题非常多。所以，直接通过产品端到端的性能测试成本会太大，而且测试效率也会比较低。

所以，**我认为更高效的做法是**：通过软件系统架构分析，将系统级性能基准指标拆分成规模较小的子系统或服务上的性能测试，然后再通过小规模的性能基准测试结果组合，分析系统级的关键性能，从而实现使用比较低的成本获取更核心的性能指标的目的。

而为了支撑这种性能测试方法，你就需要深入理解产品业务的系统架构，并学会系统科学地分解业务性能指标。

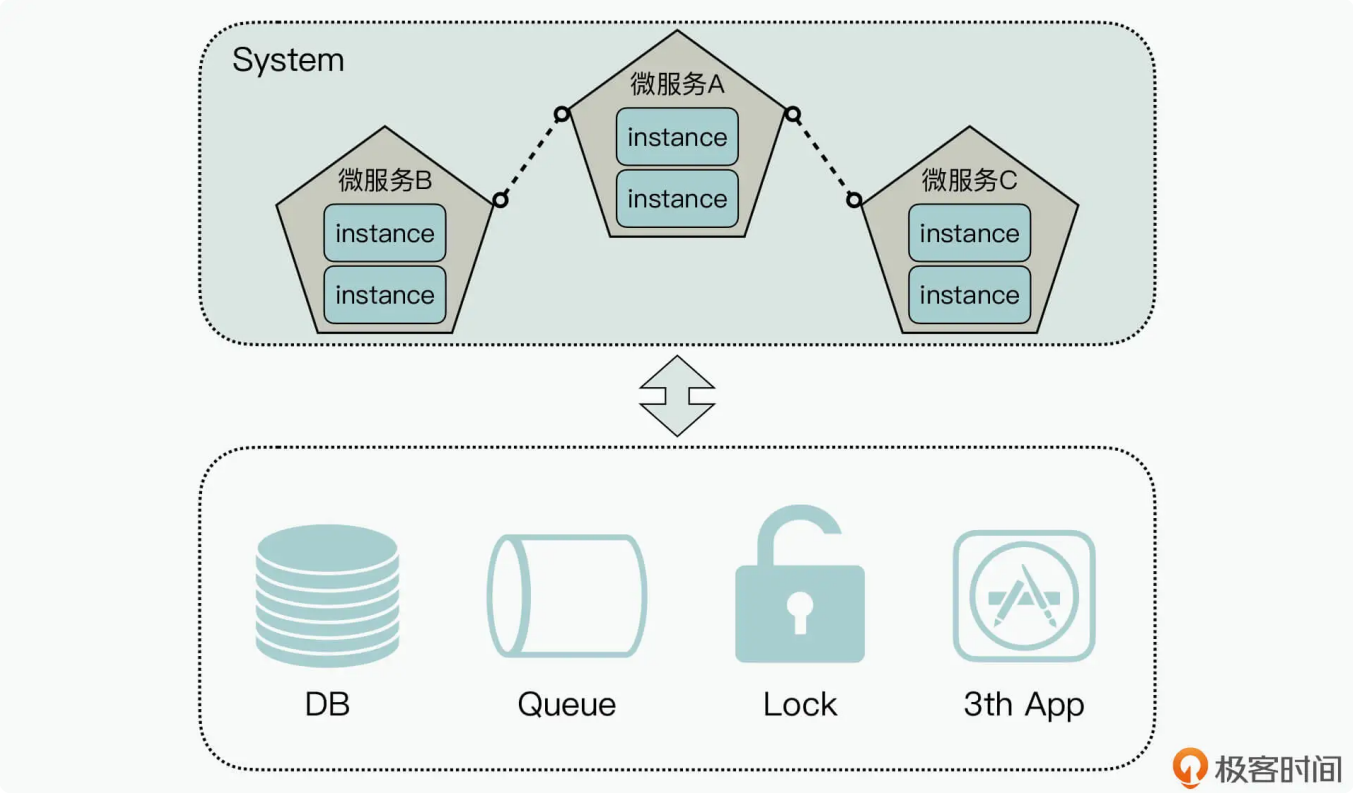
两年前，我参与了一个互联网 SaaS 服务的性能优化项目，在启动优化前需要先从头开始获取系统级的各种性能基线，而我正是通过这种方式去分解性能基准测试，从而实现了用比较短的时间获取到所关注的核心性能基线数据。

另外，当时我参与优化的 SaaS 服务主要是采用微服务架构，而微服务架构也是现在互联网业务的典型架构之一。所以接下来，我会基于当时项目中的性能基准测试经验，剥离掉具体的业务逻辑，在此基础上给你分享下针对微服务架构，进行性能基线测试的分析与设

计过程。如果你在以后的项目开发中需要做宏基准性能测试，你也可参考借鉴其中的做法。

基于软件架构设计基准性能测试过程

好，我们先来了解下这个互联网产品的微服务架构，它的简化示意图如下所示：



首先你会发现，整个产品的业务功能可以通过多个微服务来实现，如微服务 A、微服务 B、微服务 C，它们之间可以基于标准接口（如 Rest 等）进行通信；同时，你还能看到每个微服务中都包含了多个实例，每个微服务实例数是不相等的，这是根据业务的性能需求弹性扩展出来的；最后，如图的底部所示，所有微服务会共同依赖一些公共的基础设施，包括数据库、消息队列、分布式协同锁、第三方应用或 SaaS 服务，等等。

而由于对整个系统进行性能基准测试的成本比较大，所以接下来，我们还需要针对这个系统进行性能基准测试分解。

那么具体的分解步骤是怎样的呢？下面我来给你具体讲解一下。

第一步，我将多个微服务组合后的**系统级性能测试，划分为了多个微服务级的性能测试**。因为对微服务架构而言，单个微服务级的性能基准更具有指导意义。

第二步，在对单个微服务进行性能基准测试时，我又将注意力集中到了最重要的两点。

1. 单个微服务实例的性能基线。

首先，单个微服务实例在部署时，使用的硬件资源规格是相对确定的，因而方便我们进行基准化性能测试；

其次，由于单个微服务实例的性能基准指标具有良好的指导意义，因此当系统面对突发业务时，我们可以更好地基于单个服务性能来规划弹性策略；

最后，对单个微服务实例进行性能基线测试，也会更加节省资源，成本会更低。

2. 微服务多个实例间的并发性能基线。

造成并发性能瓶颈的，可能是来自于数据库访问，也可能是来自于分布式互斥锁。所以，针对数据库访问性能瓶颈，我们可以通过单独数据库级性能测试来检测；而针对分布式互斥锁等，我们需要单独构造性能模拟测试场景，来进行测试分析，或者也可以通过一些数学形式化分析手段来计算获取。

但是，当把单个微服务作为被测系统的时候，你可能会发现它与周边的微服务交互太复杂，因而不好构造测试场景或数据。所以这个时候，我建议你也可以**选择组合几个耦合度比较大的微服务一起进行性能基准测试**（如果业务的微服务架构设计不是太差的话，相信你可以找到这样的被测微服务或者微服务组）。

第三步，针对公共基础设施的性能测试可以被独立处理，具体还可以分为以下几种情况：

不需要性能测试的基础设施，比如部分数据库，很多数据库的性能基线水平可以通过其他渠道获取，并且判断在短时期内不会对系统造成性能瓶颈，可以先不用测试。

需要精准性能测试的基础设施，比如部分第三方系统或者服务的性能，直接影响到关键业务流程，但是性能基线没有任何参考。

通过形式化验证分析的基础设施，比如互斥并发上限瓶颈等，可以通过形式化或数学分析去验证，尽量避免直接去测试。

如此一来，基于以上分析，我们通过将很多微服务组成的系统级性能测试，分解成了多个微服务或微服务组、公共基础设施的性能测试，然后针对微服务级的性能基准测试，再进

一步分解，识别出关键的性能测试目标，从而实现了以大拆小的效果，这样就可以通过尽量小的代价，去获取系统的一些关键性能基线水平。

但是这里要注意，在针对被测系统进行性能基准测试时，如果我们还不了解系统的运行模型，就很容易走进一个死胡同，造成测试不仅费力费时，而且结果还不准确。

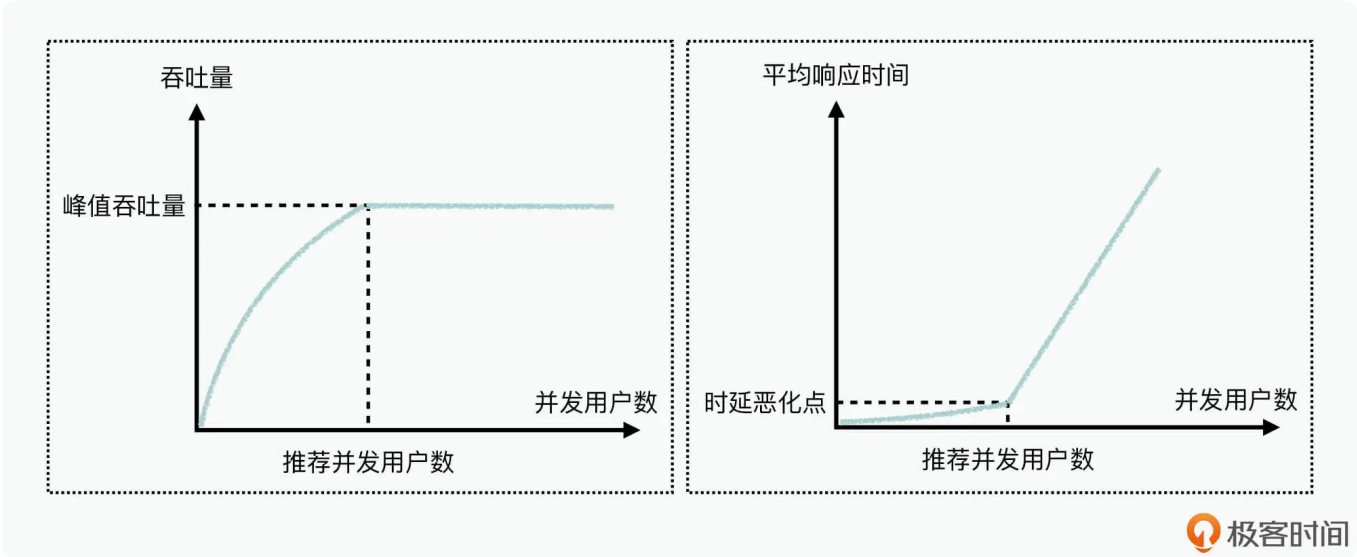
那么既然如此，有没有什么办法可以帮助我们理解系统的运行模型呢？当然是有的，接下来我要介绍的利特尔法，就可以很好地帮助你去理解软件系统的运行状态。

利特尔法则

利特尔法则（Little's law）是由麻省理工大学斯隆商学院的教授约翰·利尔特（John Litte）提出，并在 1961 年就已经被证明的一个数学模型，使用这个数学模型来分析现在互联网服务的请求处理负载，也非常适用。

那么首先，我们就来了解下利特尔法则的表述：稳定系统下的处理能力 L，等于系统稳态请求到达速率 γ 乘以单位请求处理时间 W，书写后的公式是：$L = \gamma * W$。

为了更好地理解这个数学模型，这里我们可以把它应用到互联网的请求负载处理中，用 γ 代表并发用户数、W 代表平均响应时间、L 代表系统的吞吐量（TPS），然后基于这个数学模型可以推导出如下所示的图例：



上面的两张图，分别展示了利特尔法则所揭示的软件系统的两个非常重要的性能规律，接下来就分别来看下这两个规律是什么吧。

首先，如图的左侧所示，它展示的关于并发用户数与峰值吞吐量线性关系的第一个性能规律，也就是说当被测系统达到性能饱和状态之前，吞吐量会随着并发用户数逐步上升，但是当系统达到饱和状态后，系统的吞吐量会达到峰值，不能再继续提升。

所以，如果把这个法则应用到微服务请求的性能基准测试中，其直接表现就是：**当系统处于饱和态后，微服务的 TPS（Transaction Per Second，性能测试指标）将不会再提升。**

接下来，图的右侧部分，它展示的是并发用户数与平均响应时间的第二个性能规律，也就是当被测系统请求达到饱和状态前，请求的平均响应时延是比较稳定的，但随着并发用户数的进一步提升，当被测系统达到饱和状态后，由于吞吐量不变，请求开始排队处理，因而单个请求处理时延将会急速上升，从而直接影响到客户感受。

也就是说，由于我们在对被测系统进行性能测试时，主要目标就是寻找这个系统饱和的边界。因此，我们通常只需要关注其中一个图即可，然后根据利特尔法则，就可以计算出另一个性能指标。

不过，在真实的系统服务中，可能会因为业务间的请求处理而存在一定的干扰，导致与理论的曲线有略微的差别，但是一般情况下对最终的测试结果影响不太大。所以这里你要重点提防的就是，在具体的性能测试过程中，如果测试方法使用不恰当，或者真实并发用户数不真实，都会很容易导致测试获取的性能基线不准确。

那么现在你可能要问了，针对宏基准性能测试而言，测试本身也是一个复杂的软件活动，怎么才能保证性能测试结果正确性呢？

我认为，最好的办法就是**在性能测试之前，首先验证性能基准测试的正确性**。下面我就带你具体了解一下，这样做的好处和具体操作步骤。

验证性能测试的正确性

为了验证性能测试结果的正确性，需要先去验证请求与响应数据的准确性。我会推荐使用脚本或工具，抓取现网中真实的业务请求与响应消息，来作为性能测试输入数据。

不过，这样做可能在一些场景下会比较困难，所以你也可以使用脚本来构建业务请求，但要尽量追求接近真实的业务请求。

那么，是不是性能测试使用的请求与响应数据都校验合格后，就能保证性能测试的正确性了呢？

还不行！为了验证性能测试结果的准确性，你还需要获取被测系统上的一些监控数据，具体包括两点：**系统运行态的资源状态监控信息、软件内部实现态的监控状态。**

其中，系统运行态的资源状态监控信息是需要优先保证的，这样做主要有几个好处：

首先，通过检测资源状态信息，可以用来验证被测系统是否真实达到瓶颈，而不是由于测试工具的性能瓶颈造成的假象。

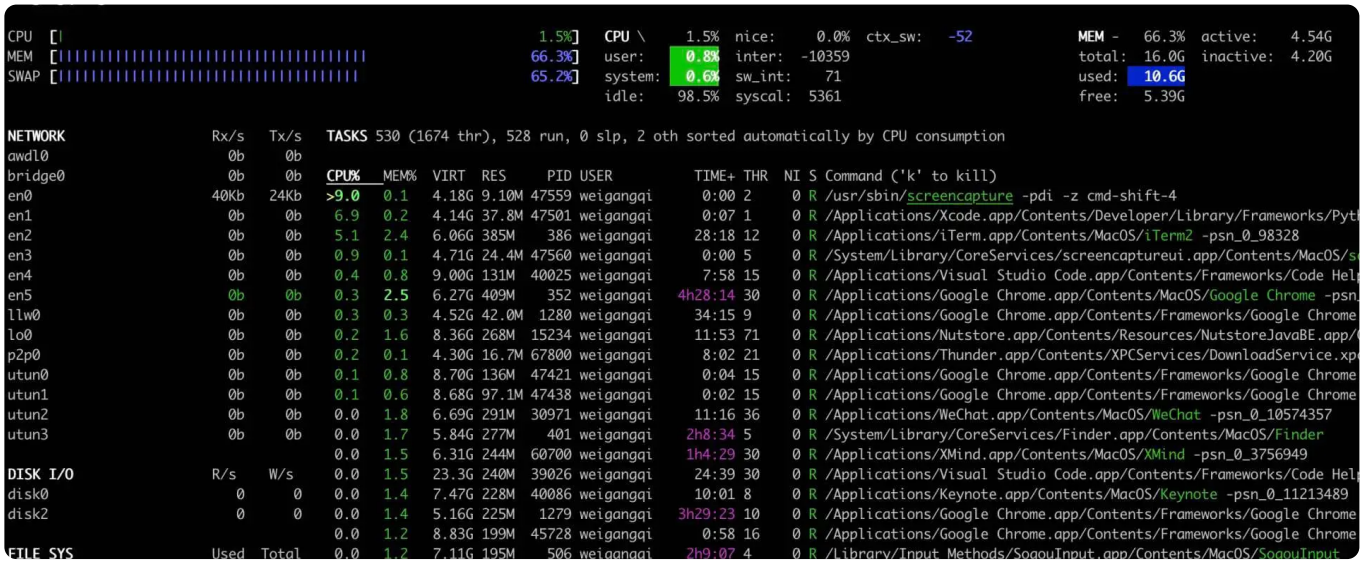
其次，还可以验证被测系统的资源使用瓶颈状态是否与预期一致，用来判断峰值性能的合理性。

最后，获取资源状态监控数据还可以用于支撑软件架构性能的进一步调优。

实际上，系统运行态的资源级可监控信息非常多，这里我推荐你两个工具，来协助监控被测系统运行态的资源使用状态。

第一个工具：Glances。这是一个由 Python 语言开发的工具，它能够报告统计 CPU、内存、网络、磁盘和进程使用状态，可以避免你去记忆操作系统提供的各种复杂的监控工具与命令。

下面是一个使用 Glances 获取监控状态的截图，你能看到各种资源使用率。



但是要注意，使用这个工具获取的是全量状态信息，你还需要识别被测系统或者服务所占用的资源。在一些场景下，你可能需要定制对被测系统的特殊监控信息，而这个时候你就可以使用**第二个工具：库—psutil**。

使用这个库可以通过代码定制实现监控逻辑，并借助 Git 库管理起来。比如，你使用下面的 Python 代码就可以很方便地获取系统监控信息：

复制代码

```
import psutil
psutil.cpu_times() //获取cpu 信息
psutil.virtual_memory() //获取内存信息
psutil.disk_partitions() //获取磁盘信息
```

当然，psutil 支持的功能是比较多的，详细的 psutil 使用说明，你可以参考官方说明。

小结

十多年前，我在针对无线系统产品级进行性能测试时，使用的每套性能测试设备价值都需要好几百万，所以在当时就尝试过将一些性能测试拆分成更小的子系统级的测试，从而减少了对产品级性能测试设备的依赖，取得了非常好的效果。

那么，在互联网业务领域中，当系统级性能测试也非常复杂的时候，你也可以通过对软件架构的分析，将其拆分成服务级与基础设施上的性能基准测试，然后再借助各种工具来完成更低成本的性能基准测试。

思考题

在你的产品性能基准测试中，有哪些性能基线水平可以通过一部分的测试数据，再加上一些数学分析推导来获取呢？

欢迎在留言区分享你的观点和看法。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。

赞 0 提建议

上一篇 17 | Benchmark测试（上）：如何做好微基准测试？

下一篇 19 | 性能测试工具：如何选择最合适的性能测试工具？

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。