大咖助场 | 李玥：高并发场景下如何优化微服务的性能？

2020-06-26 李玥

系统性能调优必知必会

进入课程 >

讲述：张浩

时长 12:16　大小 11.25M

你好，我是李玥。相信这里有部分同学对我是比较熟悉的，我在极客时间开了两门课，分别是《消息队列高手课》和《后端存储实战课》。今天很荣幸受邀来到陶辉老师的专栏做一期分享。

陶辉老师的这门课程，其中的知识点都是非常“硬核”，因为涉及到计算机操作系统底层的这些运行机制，确实非常抽象。我也看到有些同学在留言区提到，希望能通过一些例子来帮助大家更好地消化一下这些知识。那么这期分享呢，我就来帮陶辉老师做一次科普，帮助同学们把“基础设施优化”这一部分中讲到的一些抽象的概念和方法，用举例子的方式来梳理一遍。总结下的话，就是帮你理清这些问题：

线程到底是如何在 CPU 中执行的？

线程上下文切换为什么会影响性能?

为什么说异步比同步的性能好?

BIO、NIO、AIO 到底有什么区别?

为什么线程数越多反而性能越差?

今天的课程，从一个选择题开始。假设我们有一个服务，服务的业务逻辑和我们每天在做的业务都差不多，根据传入的参数去数据库查询数据，然后执行一些简单的业务逻辑后返回。我们的服务需要能支撑 10,000TPS 的请求数量，那么数据库的连接池设置成多大合适呢?

我给你二个选项:

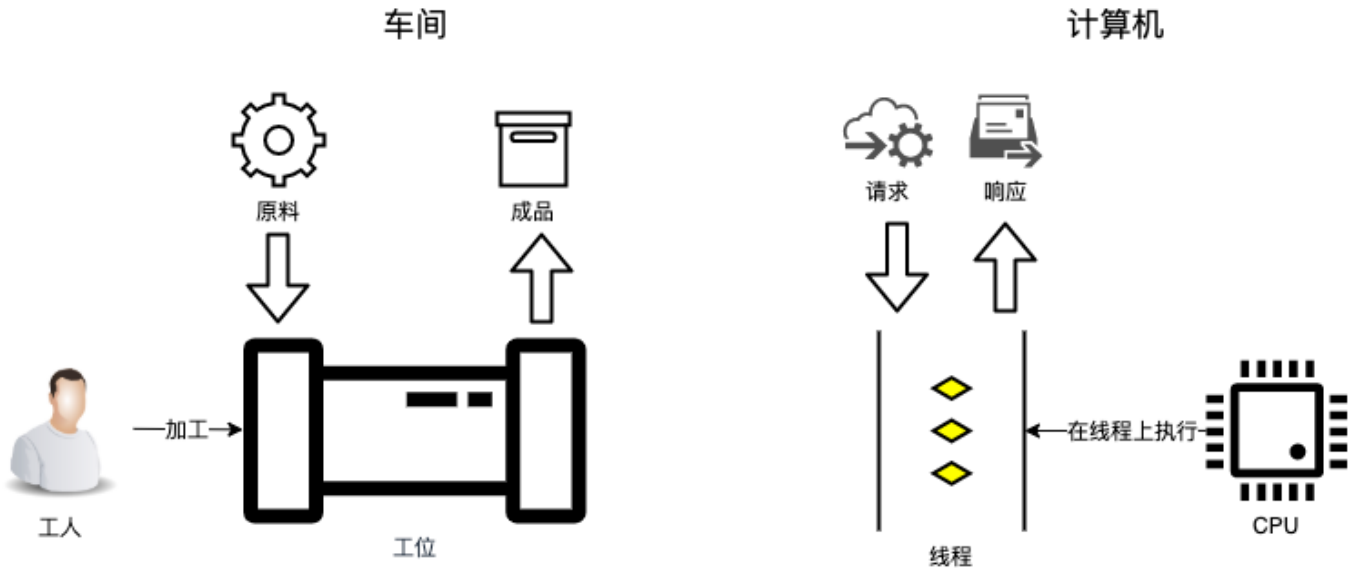
A. 32

B. 2048

我们直接公布答案，选项 A 是性能更好的选择。连接池的大小直接影响的是，同时请求到数据库服务器的并发数量。那我们直觉的第一印象可能是，并发越多总体性能应该越好才对，事实真的是这样吗？下面我们通过一个例子来探究一下这个问题的答案。

说有一个工厂，要新上一个车间，车间里面设置了 8 条流水生产线，每个流水线设置 1 个工位，那需要安排多少个工人才能达到最佳的效率呢？显然是需要 8 个工人是吧？工人少了生产线闲置，工人多了也没有工位让他们去工作，工人闲置，8 个工人对 8 条流水线是效率最优解。这里面的车间，就可以类比为一台计算机，工位就是线程，工人就是 CPU 的核心。通过这个类比，我们就可以得出这样一个结论：**一个 8 核的 CPU，8 个线程的情况下效率是最高的。** 这时，每个 CPU 核心正好对应一个线程。

这是一个非常理想的情况，它有一个前提就是，流水线上的工人（CPU 核心）一直有事情做，没有任何等待。而现实情况下，我们绝大部分的计算程序都做不到像工厂流水线那么高效。我们开发的程序几乎是**请求 / 响应**的模型，对应到车间的例子，生产模式不太像流水线，更像是来料加工。工人在工位上等着，来了一件原料，工人开始加工，加工完成后，成品被送走，然后再等待下一件，周而复始。对应到计算机程序中，原料就是请求，工人在工位上加工原料的过程，相当于 CPU 在线程上执行业务逻辑的过程，成品就是响应，或者说是请求的返回值。你可以对照下面这个图来理解上面我们讲的这个例子，以及对应到计算机程序中的概念。



来料加工这种情况下，只有 8 个工位并不能保证 8 个工人一直满负荷的工作。因为，工人每加工完成一件产品之后，需要等待成品被送出去，下一件原料被送进来，才能开始继续工作。在同一个工位上加工每件产品之间的等待是不可避免的，那怎么才能最大化工人的效率，尽量减少工人等待呢？很简单，增加一些工位就可以了。工人在 A 工位加工完成一件产品之后，不在 A 工位等着，马上去另外一个原料已经就绪的 B 工位继续工作，这样只要工位设置得足够多，就可以保证 8 个工人一直满负荷工作。

那同样是 8 个工人满负荷工作，多工位来料加工这种方式，和上面提到的 8 条流水线作业的方式，哪种效率更高呢？还是流水线的效率高，是不是？原因是，虽然在这两种方式下，工人们都在满负荷工作，但是，来料加工这种方式下，工人在不同的工位之间切换，也是需要一点点时间的，相比于流水线作业，这部分工时相当于被浪费掉了。

工人在工位间切换，对应到计算机执行程序的过程，就是 CPU 在不同的线程之间切换，称为**线程上下文切换**。一次线程上下文切换的时间耗时非常短，大约只有几百个纳秒（ns）。一般来说我们并不需要太关注这个短到不可感知的切换时间，但是，在多线程高并发的场景下，如果没有很好的优化，就有可能出现，CPU 在大量线程间频繁地发生切换，累积下来，这个切换时间就很可观了，严重的话就会拖慢服务的总体性能。

我们再来思考另外一个问题：设置多少个工位最合适呢？工位数量不足时，工人不能满负荷工作，工位数量太多了也不行，工人需要频繁地切换工位，浪费时间。这里面一定存在一个最优工位数，可以让所有工人正好不需要等待且满负荷工作。最优工位数取决于工人的加工速度、等待原料的时长等因素。如果这些参数是确定的，那我们确定这个最佳工位数就不太难了。一般来说，工位的数量设置成工人数量的两三倍就差不多了，如果等待的时间比较长，可能需要五六倍，大致是这样一个数量级。把这个结论对应到计算机系统中就是，**对于**

一个请求 / 响应模型的服务，并发线程数设置为 CPU 核数 N 倍时性能最佳，N 的大致的经验值范围是[2, 10]。

有了这个结论，再回过头来看我们课程开始提到的那个数据库连接池问题。数据库服务符合“请求 / 响应模型”，所以它的并发数量并不是越多越好，根据我们上面得出的结论，大约是 CPU 核数的几倍时达到最佳性能。这个问题来自于数据库连接池 HikariCP 的一篇 Wiki: About Pool Sizing，里面有详细的性能测试数据和计算最佳连接池数量的公式，强烈推荐你课后去看一下。

为什么说异步比同步的性能好?

然后我们再来思考这样一个问题。我们开发的很多业务服务实际的情况是，并发线程数越多总体性能越好，几百甚至上千个线程才达到最佳性能。这并不符合我们上面说的那个结论啊？什么原因?

原因是这样的，我们上面这个结论它有一个适用范围，它的适用范围是，像数据库服务这样，只依赖于本地计算资源的服务。

如果说，我们的业务服务，它在处理请求过程中，还需要去调用其他服务，这种情况就不适用于我们上面所说的结论。这里面的其它服务包括数据库服务或者是下游的业务服务等等。不适用的原因是，我们线程在执行业务逻辑过程中，很大一部分时间都花在等待外部服务上了，在这个等待的过程中，几乎不需要 CPU 参与。换句话说，每个线程需要的 CPU 时间是非常少的，这样的情况下，一个 CPU 核心需要非常多的线程才能把它“喂饱”，这就是为什么这些业务服务需要非常多的线程数，才能达到最佳性能的原因。

我们刚刚讲过，线程数过多很容易导致 CPU 频繁的在这些线程之间切换，虽然 CPU 看起来已经在满负荷运行了，但 CPU 并没有把所有的时间都用在执行我们的业务逻辑上，其中一部分 CPU 时间浪费在线程上下文切换上了。怎么来优化这种情况呢？要想让 CPU 高效地执行业务逻辑，最佳方式就是我们开头提到的流水线，用和 CPU 核数相同的线程数，通过源源不断地供给请求，让 CPU 一直不停地执行业务逻辑。**所以优化的关键点是，减少线程的数量**，把线程数量控制在和 CPU 核数相同的数量级这样一个范围。

要减少线程数量，有这样两个问题需要解决。

第一个问题是，如何用少量的线程来处理大量并发请求呢？我们可以用一个请求队列，和一组数量固定的执行线程，来解决这个问题。线程的数量就等于 CPU 的核数。接收到的请求先放入请求队列，然后分配给执行线程去处理。这样基本上能达到，让每个 CPU 的核心相对固定到一个线程上，不停地执行业务逻辑这样一个效果。

第二个问题是，执行线程在需要调用外部服务的时候，如何避免线程等待外部服务，同时还要保证及时处理返回的响应呢？我们希望的情况是，执行线程需要调用外部服务的时候，把请求发送出去之后，不要去等待响应，而是去继续处理下一个请求。等外部请求的响应回来之后，能有一个通知，来触发执行线程再执行后续的业务逻辑，直到给客户端返回响应。这其实就是我们通常所说的**异步 IO 模型（AIO，Asynchronous I/O）**，这个模型的关键就是，线程不去等待 Socket 通道上的数据，而是待数据到达时，由操作系统来发起一个通知，触发业务线程来处理。Linux 内核从 2.6 开始才加入了 AIO 的支持，到目前为止 AIO 还没有被广泛使用。

使用更广泛的是 **IO 多路复用模型（IO Multiplexing）**，IO 多路复用本质上还是一种同步 IO 模型。但是，它允许一个线程同时等待多个 Socket 通道，任意一个通道上有数据到来，就解除等待去处理。IO 多路复用没有 AIO 那么理想化，但也只是多了一个线程用于等待响应，相比 AIO 来说，效果也差不了多少，在内核 AIO 支持还不完善的时代，是一个非常务实且高效的网络 IO 模型。

很多编程语言中，都有一些网络 IO 框架，封装了这些 IO 模型，来帮我们解决这个问题，比如 Java 语言中的 BIO、NIO、AIO 分别对应了同步 IO 模型、IO 多路复用模型和异步 IO 模型。

解决了上面这两个问题之后，我们用很少量的线程就可以处理大量的并发请求。这种情况下，负责返回响应的线程和接收请求的线程，不再是同一个线程，这其实就是我们所说的**异步模型**。你可以看到，**异步模型并不会让程序的业务逻辑执行得更快，但是它可以非常有效地避免线程等待，大幅减少 CPU 在线程上下文切换上浪费的时间。**这样，在同样的计算机配置下，异步模型相比同步模型，可以更高效地利用计算机资源，从而拥有更好的总体的吞吐能力。

小结

以上就是本节课的全部内容了，我们来简单地做个小结。

理论上，线程数量设置为 CPU 核数，并且线程没有等待的情况下，CPU 几乎不会发生线程上下文切换，这个时候程序的执行效率是最高的。实际情况下，对于一个请求 / 响应模型的服务，并发线程数设置为 CPU 核数 N 倍时性能最佳。这个 N 取决于业务逻辑的执行时间、线程等待时间等因素，N 的大致的经验值范围是[2, 10]。

使用异步模型编写微服务，配合异步 IO 或者 IO 多路复用，可以有效地避免线程等待，用少量的线程处理大量并发请求，大幅减少线程上下文切换的开销，从而达到提升服务总体性能的效果。

思考题

最后留给你一道思考题。IO 多路复用，它只是一种 IO 模型，实际上有多种实现。在 Linux 中，有 select、poll、epoll 三种实现方式，课后请你去查阅一下资料，看看这三种实现方式有什么区别？

感谢阅读，如果今天的内容让你有所收获，欢迎把它分享给你的朋友。

上一篇　21 | AKF立方体：怎样通过可扩展性来提高性能？

下一篇　22 | NWR算法：如何修改读写模型以提升性能？

精选留言 (4)

写留言

胡鹏
2020-06-29

感觉说得太绝对了，现网DB链接池配置32不现实，业务侧mysql访问组件一个进程就会建立几个链接，多起几个进程mysql就受不了了

展开

。。。
2020-06-28

如果是应用服务器有20多台，数据库是一台master3台salve的话，这样的数据库连接池怎么设置呢？希望老师能回答

展开

1

凉人。
2020-06-26

老师你好，我有两个问题想请教一下。
1 多工位解决等待问题，这部分应该解决的是io读写的时间吧？
2 异步io，在发送信号之前是在做下一个请求么？如果是在处理下个请求，那么信号唤醒，这里会涉及线程的切换么？

展开

2

點點點，点颠
2020-06-26

先回答一下思考题：
　首先是poll和 select 函数对比， poll 函数和 select 不一样的地方就是，在 select 里面，文件描述符的个数已经随着 fd_set 的实现而固定，没有办法对此进行配置；而在 poll 函数里，我们可以控制 pollfd 结构的数组大小，这意味着我们可以突破原来 select 函数最大描述符的限制，在这种情况下，应用程序调用者需要分配 pollfd 数组并通知 poll 函数...

展开

2