知道吗，你的Java web应用其实是使用线程池来处理请求的。这一实现细节被许多人忽略，但是你迟早都需要理解线程池如何使用，以及如何正确地根据应用调整线程池配置。这篇文章的目的是为了解释线程模型——什么是线程池、以及怎样正确地配置线程池。

单线程模型

让我们从一些基础的线程模型开始，然后再随着线程模型的演变进行更深一步的学习。你使用的任何应用服务器或框架，如Tomcat、Dropwizard、Jetty等，它们的基本原理其实是相同的。Web服务器的最底层实际上是一个socket。这个socket监听并接受到达的TCP连接。一旦一个连接被建立，就可以通过这个新建立的连接读取、解析信息，然后将这些信息包装成一个HTTP请求。这个HTTP请求还将被移交至web应用程序，来完成请求的动作。

我们将通过一个简单的服务器程序来展示线程在其中所起到的作用。这个服务器程序展示了大部分应用服务器的底层实现细节。让我们以一个简单的单线程web服务器程序开始，它的代码像下面这样：

```
ServerSocket listener = new ServerSocket(8080);
try {
    while (true) {
        Socket socket = listener.accept();
        try {
            handleRequest(socket);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
} finally {
    listener.close();
}
```

这段代码在8080端口上创建了一个ServerSocket，紧接着通过循环来监听和接受新到达的连接。一旦连接建立，会将socket传递给handleRequest方法。这个方法可能会读取该HTTP请求，处理这个请求，然后写回一个响应。在这个简单的例子中，handleRequest方法从socket中读取简单的一行数据，然后返回一个简短的HTTP响应。但是，handleRequest有可能需要处理一些更复杂的任务，例如读数据库或者执行其它一些IO操作。

```
final static String response =
    "HTTP/1.0 200 OKrn" +
    "Content-type: text/plainrn" +
    "rn" +
    "Hello Worldrn";

public static void handleRequest(Socket socket) throws IOException {
    // Read the input stream, and return "200 OK"
    try {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));

        log.info(in.readLine());

        OutputStream out = socket.getOutputStream();
        out.write(response.getBytes(StandardCharsets.UTF_8));
    } finally {
        socket.close();
    }
}
```

因为只有一个线程处理所有的socket，因此只有在完全处理好一个请求后，才能再接受下一个请求。在实际的应用中，handleRequest方法可能需要经过100毫秒才能返回，那么这个服务器程序在一秒中，只能按顺序处理10个请求。

多线程模型

尽管handleRequest可能会被IO操作阻塞，CPU却可能是空闲的，它可以处理其它更多请求，但这对单线程模型来说是不能实现的。因此，通过创建多个线程，可以使服务器程序实现并发操作：

```
public static class HandleRequestRunnable implements Runnable {
    final Socket socket;

    public HandleRequestRunnable(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        try {
            handleRequest(socket);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// Main loop here
ServerSocket listener = new ServerSocket(8080);
try {
    while (true) {
        Socket socket = listener.accept();
        new Thread( new HandleRequestRunnable(socket) ).start();
    }
} finally {
    listener.close();
}
```

上面这段代码中，accept()方法仍然是在一个单线程循环中被调用。但是当TCP连接建立，socket创建时，服务器就创建一个新的线程。这个新生的线程将执行和单线程模型中一样的handleRequest方法。

新线程的建立使调用accept方法的线程能够处理更多的TCP连接，这样服务器就能并发地处理请求了。这一技术被称为“thread per request”（一个线程处理一个请求），也是现在最流行的服务器技术。值得注意的是，还有一些其它的服务器技术，如NGINX和Node.js采用的事件驱动异步模型，它们都没有使用线程池。因此，它们都不在本文的讨论范围内。

“thread per request”方式里创建新线程（稍后销毁这个线程）的操作是昂贵的，因为Java虚拟机和操作系统都需要为这一操作分配资源。另外，在上面那段代码的中，可以创建的线程数量是不受限制的。这么做的隐患很大，因为它可能导致服务器资源迅速枯竭。

资源枯竭

每个线程都需要一定的内存空间来作为自己的栈空间。在最近的64位虚拟机版本中，默认的栈空间是1024KB。如果server收到很多请求，或者handleRequest方法的执行时间变得比较长，就会造成服务器产生很多并发线程。如果要维护1000个线程，仅就栈空间而言，虚拟机就必须耗费1GB的RAM空间。另外，为处理请求，每个线程都会在堆上产生许多对象，这就有可能导致虚拟机的堆空间被迅速占满，给虚拟机的垃圾收集器带来很大压力，造成频繁的垃圾回收，最终导致OutOfMemoryErrors。

线程消耗的不仅是RAM资源，这些线程还可能消耗其它有限的资源，例如文件句柄、数据库连接等。过多地消耗这类资源可能导致一些其它的错误或造成系统崩溃。因此，要防止系统资源被线程耗尽，就必须对服务器产生的线程数量做出限制。

通过使用-Xss参数来调整每个线程的栈空间，可以在一定程度上解决资源枯竭的问题，但它绝

不是灵丹妙药。一个小的栈空间可以使得每个线程占用的内存减小，但这样可能会造成StackOverflowErrors栈溢出错误。栈空间的调整方式不尽相同，但是对许多应用来说，1024KB过于浪费了，而256KB或512KB会更加合适。Java所允许的最小栈空间的大小是160KB。

线程池

可以通过一个简单的线程池来避免持续地创建新线程，限制最大线程数量。线程池跟踪着所有线程，在线程数量达到上限前，它会创建新的线程，当有空闲线程时，它会使用空闲线程。

```
ServerSocket listener = new ServerSocket(8080);
ExecutorService executor = Executors.newFixedThreadPool(4);
try {
    while (true) {
        Socket socket = listener.accept();
        executor.submit( new HandleRequestRunnable(socket) );
    }
} finally {
    listener.close();
}
```

上面这段代码使用了ExecutorService类来提交任务(Runnable)。提交的任务将会被线程池中的线程执行，而不是通过新创建的线程执行。在这个例子中，所有的请求都通过一个线程数量固定为4的线程池来完成。这个线程池限制了并发执行的请求数量，从而限制了系统资源的使用。

除了newFixedThreadPool方法创建的线程池外，Executors类还提供了newCachedThreadPool 方法来创建线程池。这种线程池同样有无法限制线程数量的问题，但是它会优先使用线程池中已创建的空闲线程来处理请求。这种类型的线程池特别适用于执行短期任务的请求，因为它们不会长时间的阻塞外部资源。

ThreadPoolExecutors 类也可以直接创建，这样就可以对它进行一些个性化的配置。例如可以配置线程池内最小线程数和最大线程数，也可以配置线程创建和销毁的策略。稍后，本文将介绍这样的例子。

工作队列

对于线程数量固定的线程池，善于观察的读者可能会提出这样的一个疑问：当线程池中的线程都在工作时，一个新的请求到达，会发生什么呢？当线程池中的线程都在工作时，ThreadPoolExecutor可能会使用一个队列来组织新到达的请求，直到线程池中有空闲的线程可以使用。Executors.nexFixedThreadPool方法会默认创建一个没有长度限制的LinkedList。这个LinkedList也可能会产生系统资源耗尽的问题，虽然这个过程会比较缓慢，因为队列中的请求所占用的资源比线程占用的资源要少得多。但是在我们的例子中，队列中的每个请求都保持着一个socket，而每一个socket都需要打开一个文件句柄，操作系统对同时打开的文件句柄数量是有限制的，所以队列中保持socket并不是一个好的方式，除非必须这么做。因此，限制工作队列的长度也是有意义的。

```
public static ExecutorService newBoundedFixedThreadPool(int nThreads, int capacity) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(capacity),
        new ThreadPoolExecutor.DiscardPolicy());
}

public static void boundedThreadPoolServerSocket() throws IOException {
    ServerSocket listener = new ServerSocket(8080);
    ExecutorService executor = newBoundedFixedThreadPool(4, 16);
    try {
        while (true) {
            Socket socket = listener.accept();
            executor.submit( new HandleRequestRunnable(socket) );
```

```
            }
        } finally {
            listener.close();
        }
    }
```

我们再一次创建一个线程池，这一次我们没有使用Executors.newFixedThreadPool方法，而是自定义了一个ThreadPoolExecutor，在构造方法中传递了一个大小限制为16个元素的LinkedBlockingQueue。同样的，类ArrayBlockingQueue也可以被用来限制队列的长度。

如果所有的线程都在执行任务，而且工作队列也被请求填满了，此时对于新到达请求的处理方式，取决于ThreadPoolExecutor构造方法的最后一个参数。在我们这个例子中，我们使用的是DiscardPolicy，这个参数会让线程池丢弃新到达的请求。还有一些其它的处理策略，例如AbortPolicy会让Executor抛出一个异常，CallerRunsPolicy会使任务在它的调用端线程池中执行。CallerRunsPolicy策略提供了一个简单的方式来限制任务提交的速度。但是这样做可能是有害的，因为它会阻塞一个原本不应被阻塞的线程。

一个好的默认策略应该是Discard或Abort，它们都会使线程池丢弃新到达的任务。这样服务器就能容易地向客户端响应一个错误，例如HTTP的503错误“Service unavailable”。有的人可能会认为，队列的长度应该是允许增长的，这样所有的任务最终都能被执行。但是用户是不愿意长时间等待的，而且若任务到达的速度超过任务处理的速度，队列将会无限地增长。队列是被用来缓冲突然爆发的请求，或者处理短期任务的，通常情况下，队列应该是空的。

多少线程合适呢？

现在，我们知道了如何创建一个线程池。但是有一个更困难的问题，线程池里应该创建多少个线程呢？我们已经知道了线程池中的最大线程数量应该被限制，才不会导致系统资源耗尽。这些系统资源包括了内存（堆栈）、打开的文件句柄、打开的TCP连接、打开的数据库连接以及其它有限的系统资源。相反的，如果线程执行的是CPU密集型任务而不是IO密集型任务，服务器的物理内核数就应该被视为是有限的资源，这样创建的线程数就不应该超过系统的内核数。

系统应创建多少线程取决于这个应用执行的任务。开发人员应使用现实的请求来对系统进行负载测试，测试不同的线程池大小配置对系统的影响。每次测试都增加线程池的大小，直到系统达到崩溃的临界点。这个方法使你可以发现线程池线程数量的上限。超过这个上限，系统的资源将耗尽。在某些情况下，可以谨慎地增加系统的资源，例如分配更多的RAM空间给JVM，或者调整操作系统使其支持同时打开更多的文件句柄。然而，在某些情况下创建的线程数量会达到我们测试出的理论上限，这非常值得我们注意。稍后还会看到这方面的内容。

利特尔法则

$$L = \lambda W \quad or \quad \lambda = \frac{L}{W} \quad or \quad W = \frac{L}{\lambda}$$

排队论，特别的，Little’s Law，可以用来帮助我们理解线程池的一些特性。简单地说，利特尔法则解释了这三种变量的关系：L—系统里的请求数量、λ—请求到达的速率和W—每个请求的处理时间。例如，如果每秒10个请求到达，处理一个请求需要1秒，那么系统在每个时刻都有10个请求在处理。如果处理每个请求的时间翻倍，那么系统每时刻需要处理的请求数也翻倍为20，因此需要20个线程。

任务的执行时间对于系统中正在处理的请求数量有着很大的影响，一些后端资源的迟延，例如数据库，通常会使得请求的处理时间被延长，从而导致线程池中的线程被迅速用尽。因此，理论上测出的线程数上限对于这种情况就不是很合适，这个上限值还应该考虑到线程的执行时间，并结合理论上的上限值。

例如，假设JVM最多能同时处理的请求数为1000。如果我们预计每个请求需要耗费的时间不超过30秒，那么，在最坏的情况下我们每秒能同时处理的请求数不会超过33 ⅓个。但是，如果一切都很顺利，每个请求只需使用500ms就可以完成，那么通过1000个线程应用每秒就可以处理2000个请求。当系统突然出现短暂的任务执行迟延的问题时，通过使用一个队列来减缓这一问题是可行的。

为什么线程数配置不当会带来麻烦？

如果线程池的线程数量过少，我们就无法充分利用系统资源，这使得用户需要花费很长时间来等待请求的响应。但是，如果允许创建过多的线程，系统的资源又会被耗尽，这会对系统造成更大的破坏。

不仅仅是本地的资源被耗尽，其它一些应用也会受到影响。例如，许多应用都使用同一个后端数据库进行查询等操作。数据库有并发连接数量的限制。如果一个应用不加限制地占用了所有数据库连接，其它获取数据库连接的应用都将被阻塞。这将导致许多应用运行中断。

更糟的是，资源耗尽还会引发一些连锁故障。设想这样一个场景，一个应用有许多个实例，这些实例都运行在一个负载均衡器之后。如果一个实例因为过多的请求而占用了过多内存，JVM就需要花更多的时间进行垃圾收集工作，那么JVM处理请求的时间就减少了。这样一来，这个应用实例处理请求的能力降低了，系统中的其它实例就必须处理更多的请求。其它的实例也会因为请求数过多以及线程池大小没有限制的原因产生资源枯竭等问题。这些实例用尽了内存资源，导致虚拟机进行频繁地内存收集操作。这样的恶性循环会在这些实例中产生，直到整个系统奔溃。

我见过许多没有进行负载测试的应用，这些应用能够创建任意多的线程。通常情况下，这些应用只要很少数量的线程就能处理好以一定速率到达的请求。但是，如果应用需要使用其它的一些远程服务来处理用户请求，而这个远程服务的处理能力突然降低了，这将增加【大】W的值（应用处理请求的平均时间）。这样，线程池的线程就会被迅速用尽。如果对应用进行线程数量的负载测试，那么资源枯竭问题就会在测试中显现出来。

多少个线程池合适？

对于微服务架构和面向服务的架构（SOA）来说，它们通常需要请求一些后端服务。线程池的配置非常容易导致程序失败，因此必须谨慎地配置线程池。如果远程服务的性能下降，系统中的线程数量就会迅速达到线程池的上限，其它后续到达的服务就会被丢弃。这些后续的请求也许并不是要使用性能出现故障的服务，但是它们都只能被丢弃了。

针对不同的后端服务请求，设置不同的线程池可以解决这一问题。在这个模式中，仍然使用同一个线程池来处理用户的请求，但是当用户的请求需要调用一个远程服务时，这个任务就被传递给一个指定的后端线程池。这样处理用户请求的主线程池就不会因为调用后端服务而产生很大的负担。当后端服务出现故障时，只有调用这个服务的线程池才会受到影响。

使用多个线程池还有一个好处，就是它能帮助避免出现死锁问题。如果每个空闲线程都因为一个尚未处理完毕的请求阻塞，就会发生死锁，没有一个线程可以继续往下执行。如果使用多个线程池，理解好每个线程池应负责的工作，那么死锁的问题就能在一定程度上避免。

截止时间和一些最佳实践

一个最佳实践是给需要远程调用的请求规定一个截止时间。如果远程服务在规定的时间内没有响应，就丢弃这个请求。这样的技术也可以用在线程池中，如果线程处理某个请求的时间超过了规定时间，那么这个线程就应被停止，为新到达的请求腾出资源，这样也就给W（处理请求的平均时间）规定了上限。虽然这样的做法看起来有些浪费，但是如果一个用户（特别是当用

户在使用浏览器时），在等待请求的响应，那么30秒以后，浏览器无论如何也会放弃这个请求，或者更有可能的是：用户不会耐心地等待这个请求响应，而是进行其它操作去了。

快速失败也是一个可以用来处理后端请求的线程池方案。如果后端服务失效了，线程池中的线程数会迅速到达上限，这些线程都在等待没有响应的后端服务。如果使用快速失败机制，当后端服务被标记为失效时，所有的后续请求都会迅速失败，而不是进行不必要的等待。当然，它也需要一种机制来判断后端何时恢复为可用的。

最后，如果一个请求需要独立地调用多个后端服务，那么这个请求就应能并行地调用这些后端服务，而不是顺序地进行。这样就能降低请求的等待时间，但这是以增加线程数为代价的。

幸运的是，有一个非常好的库hystrix，这个库封装了许多很好的线程策略，然后以非常简单和友好的方式将这些借口暴露出来。

性能

我希望这篇文章能改进你对线程池的理解。一个合适的线程池配置需要理解应用的需求，还需要考虑这几个因素，系统允许的最大线程数、处理用户请求所需的时间。好的线程池配置不仅可以避免系统出现连锁故障，还能帮助计划和提供服务。

即使你的应用没有直接地使用一个线程池，它们也间接地通过应用服务器或其它更高级的抽象形式使用了线程池。Tomcat、JBoss、Undertow、Dropwizard 都提供了多种可配置的线程池（这些线程池正是你编写的Servlet运行的地方）。