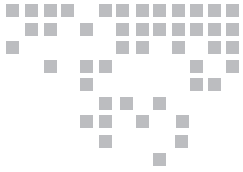

第 1 章 Chapter 1

并发编程的挑战

并发编程的目的是为了让程序运行得更快，但是，并不是启动更多的线程就能让程序最大限度地并发执行。在进行并发编程时，如果希望通过多线程执行任务让程序运行得更快，会面临非常多的挑战，比如上下文切换的问题、死锁的问题，以及受限于硬件和软件的资源限制问题，本章会介绍几种并发编程的挑战以及解决方案。

1.1 上下文切换

即使是单核处理器也支持多线程执行代码，CPU 通过给每个线程分配 CPU 时间片来实现这个机制。时间片是 CPU 分配给各个线程的时间，因为时间片非常短，所以 CPU 通过不停地切换线程执行，让我们感觉多个线程是同时执行的，时间片一般是几十毫秒（ms）。

CPU 通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

这就像我们同时读两本书，当我们在读一本英文的技术书时，发现某个单词不认识，于是便打开中英文字典，但是在放下英文技术书之前，大脑必须先记住这本书读到了多少页的第多少行，等查完单词之后，能够继续读这本书。这样的切换是会影响读书效率的，同样上下文切换也会影响多线程的执行速度。

1.1.1 多线程一定快吗

下面的代码演示串行和并发执行并累加操作的时间，请分析：下面的代码并发执行一定

比串行执行快吗？

```
public class ConcurrencyTest {

        private static final long count = 10000l;

        public static void main(String[] args) throws InterruptedException {
                concurrency();
                serial();
        }

        private static void concurrency() throws InterruptedException {
                long start = System.currentTimeMillis();
                Thread thread = new Thread(new Runnable() {
                        @Override
                        public void run() {
                                int a = 0;
                                for (long i = 0; i < count; i++) {
                                        a += 5;
                                }
                        }
                });
                thread.start();
                int b = 0;
                for (long i = 0; i < count; i++) {
                        b--;
                }
                long time = System.currentTimeMillis() - start;
                thread.join();
                System.out.println("concurrency :" + time+"ms,b="+b);
        }

        private static void serial() {
                long start = System.currentTimeMillis();
                int a = 0;
                for (long i = 0; i < count; i++) {
                        a += 5;
                }
                int b = 0;
                for (long i = 0; i < count; i++) {
                        b--;
                }
                long time = System.currentTimeMillis() - start;
                System.out.println("serial:" + time+"ms,b="+b+",a="+a);
        }

}
```

上述问题的答案是“不一定”，测试结果如表 1-1 所示。

表 1-1　测试结果

循环次数	串行执行耗时 /ms	并发执行耗时	并发比串行快多少
1 亿	130	77	约 1 倍
1 千万	18	9	约 1 倍
1 百万	5	5	差不多
10 万	4	3	慢
1 万	0	1	慢

从表 1-1 可以发现，当并发执行累加操作不超过百万次时，速度会比串行执行累加操作要慢。那么，为什么并发执行的速度会比串行慢呢？这是因为线程有创建和上下文切换的开销。

1.1.2　测试上下文切换次数和时长

下面我们来看看有什么工具可以度量上下文切换带来的消耗。

- 使用 Lmbench3 ㊀可以测量上下文切换的时长。
- 使用 vmstat 可以测量上下文切换的次数。

下面是利用 vmstat 测量上下文切换次数的示例。

```
$ vmstat 1
 procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
  r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
  0  0      0 127876 398928 2297092    0    0     0     4    2    2  0  0 99  0  0
  0  0      0 127868 398928 2297092    0    0     0     0  595 1171  0  1 99  0  0
  0  0      0 127868 398928 2297092    0    0     0     0  590 1180  1  0 100 0  0
  0  0      0 127868 398928 2297092    0    0     0     0  567 1135  0  1 99  0  0
```

CS（Content Switch）表示上下文切换的次数，从上面的测试结果中我们可以看到，上下文每 1 秒切换 1000 多次。

1.1.3　如何减少上下文切换

减少上下文切换的方法有无锁并发编程、CAS 算法、使用最少线程和使用协程。

- 无锁并发编程。多线程竞争锁时，会引起上下文切换，所以多线程处理数据时，可以用一些办法来避免使用锁，如将数据的 ID 按照 Hash 算法取模分段，不同的线程处理不同段的数据。
- CAS 算法。Java 的 Atomic 包使用 CAS 算法来更新数据，而不需要加锁。
- 使用最少线程。避免创建不需要的线程，比如任务很少，但是创建了很多线程来处理，这样会造成大量线程都处于等待状态。
- 协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

㊀ Lmbench3 是一个性能分析工具。

1.1.4 减少上下文切换实战

本节将通过减少线上大量 WAITING 的线程，来减少上下文切换次数。

第一步：用 jstack 命令 dump 线程信息，看看 pid 为 3117 的进程里的线程都在做什么。

```
sudo -u admin /opt/ifeve/java/bin/jstack 31177 > /home/tengfei.fangtf/dump17
```

第二步：统计所有线程分别处于什么状态，发现 300 多个线程处于 WAITING（onobject-monitor）状态。

```
[tengfei.fangtf@ifeve ~]$ grep java.lang.Thread.State dump17 | awk '{print $2$3$4$5}'
    | sort | uniq -c
 39 RUNNABLE
 21 TIMED_WAITING(onobjectmonitor)
 6 TIMED_WAITING(parking)
 51 TIMED_WAITING(sleeping)
 305 WAITING(onobjectmonitor)
 3 WAITING(parking)
```

第三步：打开 dump 文件查看处于 WAITING（onobjectmonitor）的线程在做什么。发现这些线程基本全是 JBOSS 的工作线程，在 await。说明 JBOSS 线程池里线程接收到的任务太少，大量线程都闲着。

```
"http-0.0.0.0-7001-97" daemon prio=10 tid=0x000000004f6a8000 nid=0x555e in
    Object.wait() [0x0000000052423000]
 java.lang.Thread.State: WAITING (on object monitor)
 at java.lang.Object.wait(Native Method)
 - waiting on <0x00000007969b2280> (a org.apache.tomcat.util.net.AprEndpoint$Worker)
 at java.lang.Object.wait(Object.java:485)
 at org.apache.tomcat.util.net.AprEndpoint$Worker.await(AprEndpoint.java:1464)
 - locked <0x00000007969b2280> (a org.apache.tomcat.util.net.AprEndpoint$Worker)
 at org.apache.tomcat.util.net.AprEndpoint$Worker.run(AprEndpoint.java:1489)
 at java.lang.Thread.run(Thread.java:662)
```

第四步：减少 JBOSS 的工作线程数，找到 JBOSS 的线程池配置信息，将 maxThreads 降到 100。

```
<maxThreads="250" maxHttpHeaderSize="8192"
 emptySessionPath="false" minSpareThreads="40" maxSpareThreads="75"
     maxPostSize="512000" protocol="HTTP/1.1"
 enableLookups="false" redirectPort="8443" acceptCount="200" bufferSize="16384"
 connectionTimeout="15000" disableUploadTimeout="false" useBodyEncodingForURI=
     "true">
```

第五步：重启 JBOSS，再 dump 线程信息，然后统计 WAITING（onobjectmonitor）的线程，发现减少了 175 个。WAITING 的线程少了，系统上下文切换的次数就会少，因为每一次从 WAITTING 到 RUNNABLE 都会进行一次上下文的切换。读者也可以使用 vmstat 命令测试一下。

```
[tengfei.fangtf@ifeve ~]$ grep java.lang.Thread.State dump17 | awk '{print $2$3$4$5}'
    | sort | uniq -c
 44 RUNNABLE
 22 TIMED_WAITING(onobjectmonitor)
 9 TIMED_WAITING(parking)
 36 TIMED_WAITING(sleeping)
 130 WAITING(onobjectmonitor)
1  WAITING(parking)
```

1.2　死锁

锁是个非常有用的工具，运用场景非常多，因为它使用起来非常简单，而且易于理解。但同时它也会带来一些困扰，那就是可能会引起死锁，一旦产生死锁，就会造成系统功能不可用。让我们先来看一段代码，这段代码会引起死锁，使线程 t1 和线程 t2 互相等待对方释放锁。

```
public class DeadLockDemo {

    privat static String A = "A";
    private static String B = "B";

    public static void main(String[] args) {

            new DeadLockDemo().deadLock();
    }

    private void deadLock() {
            Thread t1 = new Thread(new Runnable() {
                    @Override
                    publicvoid run() {
                            synchronized (A) {
                                    try { Thread.currentThread().sleep(2000);
                                    } catch (InterruptedException e) {
                                            e.printStackTrace();
                                    }
                                    synchronized (B) {
                                            System.out.println("1");
                                    }
                            }
                    }
            });

            Thread t2 = new Thread(new Runnable() {
                    @Override
                    publicvoid run() {
                            synchronized (B) {
                                    synchronized (A) {
```

```
                                        System.out.println("2");
                                    }
                                }
                            }
                        });

                        t1.start();
                        t2.start();
                    }

                }
```

这段代码只是演示死锁的场景，在现实中你可能不会写出这样的代码。但是，在一些更为复杂的场景中，你可能会遇到这样的问题，比如 t1 拿到锁之后，因为一些异常情况没有释放锁（死循环）。又或者是 t1 拿到一个数据库锁，释放锁的时候抛出了异常，没释放掉。

一旦出现死锁，业务是可感知的，因为不能继续提供服务了，那么只能通过 dump 线程查看到底是哪个线程出现了问题，以下线程信息告诉我们是 DeadLockDemo 类的第 42 行和第 31 行引起的死锁。

```
"Thread-2" prio=5 tid=7fc0458d1000 nid=0x116c1c000 waiting for monitor entry [116c1b000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at com.ifeve.book.forkjoin.DeadLockDemo$2.run(DeadLockDemo.java:42)
        - waiting to lock <7fb2f3ec0> (a java.lang.String)
        - locked <7fb2f3ef8> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:695)

"Thread-1" prio=5 tid=7fc0430f6800 nid=0x116b19000 waiting for monitor entry [116b18000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at com.ifeve.book.forkjoin.DeadLockDemo$1.run(DeadLockDemo.java:31)
        - waiting to lock <7fb2f3ef8> (a java.lang.String)
        - locked <7fb2f3ec0> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:695)
```

现在我们介绍避免死锁的几个常见方法。

- 避免一个线程同时获取多个锁。
- 避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。
- 尝试使用定时锁，使用 lock.tryLock（timeout）来替代使用内部锁机制。
- 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。

1.3 资源限制的挑战

（1）什么是资源限制

资源限制是指在进行并发编程时，程序的执行速度受限于计算机硬件资源或软件资源。例如，服务器的带宽只有 2Mb/s，某个资源的下载速度是 1Mb/s 每秒，系统启动 10 个线程下

载资源，下载速度不会变成 10Mb/s，所以在进行并发编程时，要考虑这些资源的限制。硬件资源限制有带宽的上传 / 下载速度、硬盘读写速度和 CPU 的处理速度。软件资源限制有数据库的连接数和 socket 连接数等。

（2）资源限制引发的问题

在并发编程中，将代码执行速度加快的原则是将代码中串行执行的部分变成并发执行，但是如果将某段串行的代码并发执行，因为受限于资源，仍然在串行执行，这时候程序不仅不会加快执行，反而会更慢，因为增加了上下文切换和资源调度的时间。例如，之前看到一段程序使用多线程在办公网并发地下载和处理数据时，导致 CPU 利用率达到 100%，几个小时都不能运行完成任务，后来修改成单线程，一个小时就执行完成了。

（3）如何解决资源限制的问题

对于硬件资源限制，可以考虑使用集群并行执行程序。既然单机的资源有限制，那么就让程序在多机上运行。比如使用 ODPS、Hadoop 或者自己搭建服务器集群，不同的机器处理不同的数据。可以通过“数据 ID% 机器数”，计算得到一个机器编号，然后由对应编号的机器处理这笔数据。

对于软件资源限制，可以考虑使用资源池将资源复用。比如使用连接池将数据库和 Socket 连接复用，或者在调用对方 webservice 接口获取数据时，只建立一个连接。

（4）在资源限制情况下进行并发编程

如何在资源限制的情况下，让程序执行得更快呢？方法就是，根据不同的资源限制调整程序的并发度，比如下载文件程序依赖于两个资源——带宽和硬盘读写速度。有数据库操作时，涉及数据库连接数，如果 SQL 语句执行非常快，而线程的数量比数据库连接数大很多，则某些线程会被阻塞，等待数据库连接。

1.4　本章小结

本章介绍了在进行并发编程时，大家可能会遇到的几个挑战，并给出了一些解决建议。有的并发程序写得不严谨，在并发下如果出现问题，定位起来会比较耗时和棘手。所以，对于 Java 开发工程师而言，笔者强烈建议多使用 JDK 并发包提供的并发容器和工具类来解决并发问题，因为这些类都已经通过了充分的测试和优化，均可解决了本章提到的几个挑战。