

第4章 Java并发编程基础

Java从诞生开始就明智地选择了内置对多线程的支持,这使得Java语言相比同一时期的其他语言具有明显的优势。线程作为操作系统调度的最小单元,多个线程能够同时执行,这将显著提升程序性能,在多核环境中表现得更加明显。但是,过多地创建线程和对线程的不当管理也容易造成问题。本章将着重介绍Java并发编程的基础知识,从启动一个线程到线程间不同的通信方式,最后通过简单的线程池示例以及应用(简单的Web服务器)来串联本章所介绍的内容。

4.1 线程简介

4.1.1 什么是线程

现代操作系统在运行一个程序时，会为其创建一个进程。例如，启动一个Java程序，操作系统就会创建一个Java进程。现代操作系统调度的最小单元是线程，也叫轻量级进程(Light Weight Process)，在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。处理器在这些线程上高速切换，让使用者感觉到这些线程在同时执行。

一个Java程序从main()方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上Java程序天生就是多线程程序，因为执行main()方法的是一个名称为main的线程。下面使用JMX来查看一个普通的Java程序包含哪些线程，如代码清单4-1所示。

代码清单4-1 MultiThread.java

```
public class MultiThread{
    public static void main(String[] args) {
        // 获取Java线程管理MXBean
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
        // 不需要获取同步的monitor和synchronizer信息, 仅获取线程和线程堆栈信息
        ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false, false);
        // 遍历线程信息, 仅打印线程ID和线程名称信息
        for (ThreadInfo threadInfo : threadInfos) {
            System.out.println("[ " + threadInfo.getThreadId() + " ] " + threadInfo.
                getThreadName());
        }
    }
}
```

输出如下所示(输出内容可能不同)。

```
[4] Signal Dispatcher // 分发处理发送给JVM信号的线程
[3] Finalizer          // 调用对象finalize方法的线程
[2] Reference Handler  // 清除Reference的线程
[1] main               // main线程, 用户程序入口
```

可以看到, 一个Java程序的运行不仅仅是main()方法的运行, 而是main线程和多个其他线程的同时运行。

4.1.2 为什么要使用多线程

执行一个简单的“Hello,World!”, 却启动了那么多的“无关”线程, 是不是把简单的问题复杂化了? 当然不是, 因为正确使用多线程, 总是能够给开发人员带来显著的好处, 而使用多线程的原因主要有以下几点。

(1)更多的处理器核心

随着处理器上的核心数量越来越多, 以及超线程技术的广泛运用, 现在大多数计算机都比以往更加擅长并行计算, 而处理器性能的提升方式, 也从更高的主频向更多的核心发展。如何利用好处理器上的多个核心也成了现在的主要问题。

线程是大多数操作系统调度的基本单元, 一个程序作为一个进程来运行, 程序运行过程中能够创建多个线程, 而一个线程在一个时刻只能运行在一个处理器核心上。试想一下, 一个单线程程序在运行时只能使用一个处理器核心, 那么再多的处理器核心加入也无法显著提升该程序的执行效率。相反, 如果该程序使用多线程技术, 将计算逻辑分配到多个处理器核心上, 就会显著减少程序的处理时间, 并且随着更多处理器核心的加入而变得更有效率。

(2)更快的响应时间

有时我们会编写一些较为复杂的代码(这里的复杂不是说复杂的算法, 而是复杂的业务逻辑), 例如, 一笔订单的创建, 它包括插入订单数据、生成订单快照、发送邮件通知卖家和记录货品销售数量等。用户从单击“订购”按钮开始, 就要等待这些操作全部完成才能看到订购成功的结果。但是这么多业务操作, 如何能够让其更快地完成呢?

在上面的场景中, 可以使用多线程技术, 即将数据一致性不强的操作派发给其他线程处理(也可以使用消息队列), 如生成订单快照、发送邮件等。这样做的好处是响应用户请求的线程能够尽可能快地处理完成, 缩短了响应时间, 提升了用户体验。

(3) 更好的编程模型

Java为多线程编程提供了良好、考究并且一致的编程模型，使开发人员能够更加专注于问题的解决，即为所遇到的问题建立合适的模型，而不是绞尽脑汁地考虑如何将其多线程化。一旦开发人员建立好了模型，稍做修改总是能够方便地映射到Java提供的多线程编程模型上。

4.1.3 线程优先级

现代操作系统基本采用时分的形式调度运行的线程，操作系统会分出一个个时间片，线程会分配到若干时间片，当线程的时间片用完了就会发生线程调度，并等待着下次分配。线程分配到的时间片多少也就决定了线程使用处理器资源的多少，而线程优先级就是决定线程需要多或者少分配一些处理器资源的线程属性。

在Java线程中，通过一个整型成员变量priority来控制优先级，优先级的范围从1~10，在线程构建的时候可以通过setPriority(int)方法来修改优先级，默认优先级是5，优先级高的线程分配时间片的数量要多于优先级低的线程。设置线程优先级时，针对频繁阻塞(休眠或者I/O操作)的线程需要设置较高优先级，而偏重计算(需要较多CPU时间或者偏运算)的线程则设置较低的优先级，确保处理器不会被独占。在不同的JVM以及操作系统上，线程规划会存在差异，有些操作系统甚至会忽略对线程优先级的设定，示例如代码清单4-2所示。

代码清单4-2 Priority.java


```
public class Priority {
    private static volatile boolean notStart = true;
    private static volatile boolean notEnd = true;
    public static void main(String[] args) throws Exception {
        List<Job> jobs = new ArrayList<Job>();
        for (int i = 0; i < 10; i++) {
            int priority = i < 5 ? Thread.MIN_PRIORITY : Thread.MAX_PRIORITY;
            Job job = new Job(priority);
            jobs.add(job);
            Thread thread = new Thread(job, "Thread:" + i);
            thread.setPriority(priority);
            thread.start();
        }
        notStart = false;
        TimeUnit.SECONDS.sleep(10);
        notEnd = false;
        for (Job job : jobs) {
            System.out.println("Job Priority : " + job.priority + ",
                               Count : " + job.jobCount);
        }
    }
    static class Job implements Runnable {
        private int priority;
        private long jobCount;
    }
}
```

```
public Job(int priority) {
    this.priority = priority;
}
public void run() {
    while (notStart) {
        Thread.yield();
    }
    while (notEnd) {
        Thread.yield();
        jobCount++;
    }
}
}
```

运行该示例，在笔者机器上对应的输出如下。

```
Job Priority : 1, Count : 1259592
Job Priority : 1, Count : 1260717
Job Priority : 1, Count : 1264510
Job Priority : 1, Count : 1251897
Job Priority : 1, Count : 1264060
Job Priority : 10, Count : 1256938
Job Priority : 10, Count : 1267663
Job Priority : 10, Count : 1260637
Job Priority : 10, Count : 1261705
Job Priority : 10, Count : 1259967
```

从输出可以看到线程优先级没有生效，优先级1和优先级10的Job计数的结果非常相近，没有明显差距。这表示程序正确性不能依赖线程的优先级高低。

 **注意** 线程优先级不能作为程序正确性的依赖，因为操作系统可以完全不用理会Java线程对于优先级的设定。笔者的环境为:Mac OS X 10.10, Java版本为1.7.0_71，经过笔者验证该环境下所有Java线程优先级均为5(通过jstack查看)，对线程优先级的设置会被忽略。另外，尝试在Ubuntu 14.04环境下运行该示例，输出结果也表示该环境忽略了线程优先级的设置。

4.1.4 线程的状态

Java线程在运行的生命周期中可能处于表4-1所示的6种不同的状态，在给定的一个时刻，线程只能处于其中的一个状态。

表4-1 Java线程的状态

状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

下面我们使用jstack工具(可以选择打开终端，键入jstack或者到JDK安装目录的bin目录下执行命令)，尝试查看示例代码运行时的线程信息，更加深入地理解线程状态，示例如代码清单4-3所示。

代码清单4-3 ThreadState.java

```
public class ThreadState {
    public static void main(String[] args) {
        new Thread(new TimeWaiting (), "TimeWaitingThread").start();
        new Thread(new Waiting(), "WaitingThread").start();
        // 使用两个Blocked线程，一个获取锁成功，另一个被阻塞
        new Thread(new Blocked(), "BlockedThread-1").start();
        new Thread(new Blocked(), "BlockedThread-2").start();
    }
    // 该线程不断地进行睡眠
    static class TimeWaiting implements Runnable {
        @Override
        public void run() {
            while (true) {
                SleepUtils.second(100);
            }
        }
    }
    // 该线程在Waiting.class实例上等待
    static class Waiting implements Runnable {
```



```

@Override
public void run() {
    while (true) {
        synchronized (Waiting.class) {
            try {
                Waiting.class.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
// 该线程在Blocked.class实例上加锁后, 不会释放该锁
static class Blocked implements Runnable {
    public void run() {
        synchronized (Blocked.class) {
            while (true) {
                SleepUtils.second(100);
            }
        }
    }
}
}

```

上述示例中使用的SleepUtils如代码清单4-4所示。

代码清单4-4 SleepUtils.java

```

public class SleepUtils {
    public static final void second(long seconds) {
        try {
            TimeUnit.SECONDS.sleep(seconds);
        } catch (InterruptedException e) {
        }
    }
}

```

运行该示例, 打开终端或者命令提示符, 键入“jps”, 输出如下。

```

611
935 Jps
929 ThreadState
270

```

可以看到运行示例对应的进程ID是929, 接着再键入“jstack 929”(这里的进程ID需要和读

者自己键入jps得出的ID一致)，部分输出如下所示。

```
// BlockedThread-2线程阻塞在获取Blocked.class示例的锁上
"BlockedThread-2" prio=5 tid=0x00007feacb05d000 nid=0x5d03 waiting for monitor
entry [0x000000010fd58000]
    java.lang.Thread.State: BLOCKED (on object monitor)
// BlockedThread-1线程获取到了Blocked.class的锁
"BlockedThread-1" prio=5 tid=0x00007feacb05a000 nid=0x5b03 waiting on condition
[0x000000010fc55000]
    java.lang.Thread.State: TIMED_WAITING (sleeping)
// WaitingThread线程在Waiting实例上等待
"WaitingThread" prio=5 tid=0x00007feacb059800 nid=0x5903 in Object.wait()
[0x000000010fb52000]
    java.lang.Thread.State: WAITING (on object monitor)
// TimeWaitingThread线程处于超时等待
"TimeWaitingThread" prio=5 tid=0x00007feacb058800 nid=0x5703 waiting on condition
[0x000000010fa4f000]
    java.lang.Thread.State: TIMED_WAITING (sleeping)
```

通过示例，我们了解到Java程序运行中线程状态的具体含义。线程在自身的生命周期中，并不是固定地处于某个状态，而是随着代码的执行在不同的状态之间进行切换，Java线程状态变迁如图4-1示。

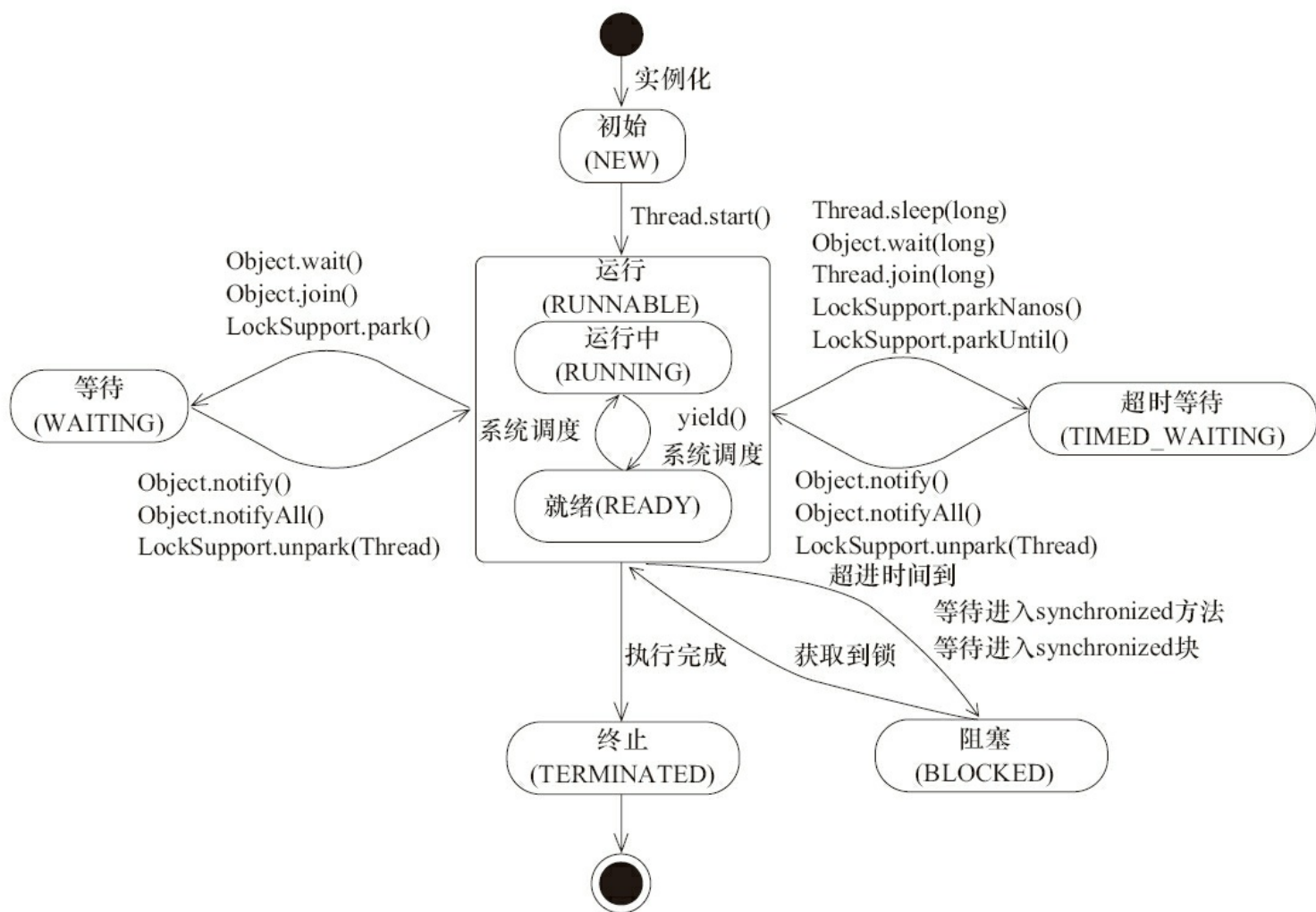


图4-1 Java线程状态变迁

由图4-1中可以看到，线程创建之后，调用start()方法开始运行。当线程执行wait()方法之后，线程进入等待状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而超时等待状态相当于在等待状态的基础上增加了超时限制，也就是超时时间到达时将会返回到运行状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到阻塞状态。线程在执行Runnable的run()方法之后将会进入到终止状态。

注意 Java将操作系统中的运行和就绪两个状态合并称为运行状态。阻塞状态是线程阻塞在进入synchronized关键字修饰的方法或代码块(获取锁)时的状态，但是阻塞在java.concurrent包中Lock接口的线程状态却是等待状态，因为java.concurrent包中Lock接口对于阻塞的实现均使用了LockSupport类中的相关方法。

4.1.5 Daemon线程

Daemon线程是一种支持型线程,因为它主要被用作程序中后台调度以及支持性工作。这意味着,当一个Java虚拟机中不存在非Daemon线程的时候,Java虚拟机将会退出。可以通过调用Thread.setDaemon(true)将线程设置为Daemon线程。



注意 Daemon属性需要在启动线程之前设置,不能在启动线程之后设置。

Daemon线程被用作完成支持性工作,但是在Java虚拟机退出时Daemon线程中的finally块并不一定会执行,示例如代码清单4-5所示。

代码清单4-5 Daemon.java

```
public class Daemon {
    public static void main(String[] args) {
        Thread thread = new Thread(new DaemonRunner(), "DaemonRunner");
        thread.setDaemon(true);
        thread.start();
    }

    static class DaemonRunner implements Runnable {
        @Override
        public void run() {
            try {
                SleepUtils.second(10);
            } finally {
                System.out.println("DaemonThread finally run.");
            }
        }
    }
}
```

运行Daemon程序,可以看到在终端或者命令提示符上没有任何输出。main线程(非Daemon线程)在启动了线程DaemonRunner之后随着main方法执行完毕而终止,而此时Java虚拟机中已经没有非Daemon线程,虚拟机需要退出。Java虚拟机中的所有Daemon线程都需要立即终止,因此DaemonRunner立即终止,但是DaemonRunner中的finally块并没有执行。



注意 在构建Daemon线程时,不能依靠finally块中的内容来确保执行关闭或清理资源

的逻辑。

4.2 启动和终止线程

在前面章节的示例中通过调用线程的`start()`方法进行启动, 随着`run()`方法的执行完毕, 线程也随之终止, 大家对此一定不会陌生, 下面将详细介绍线程的启动和终止。

4.2.1 构造线程

在运行线程之前首先要构造一个线程对象，线程对象在构造的时候需要提供线程所需要的属性，如线程所属的线程组、线程优先级、是否是Daemon线程等信息。代码清单4-6所示的代码摘自java.lang.Thread中对线程进行初始化的部分。


代码清单4-6 Thread.java

```
private void init(ThreadGroup g, Runnable target, String name, long stackSize,
    AccessControlContext acc) {
    if (name == null) {
        throw new NullPointerException("name cannot be null");
    }
    // 当前线程就是该线程的父线程
    Thread parent = currentThread();
    this.group = g;
    // 将daemon、priority属性设置为父线程的对应属性
    this.daemon = parent.isDaemon();
    this.priority = parent.getPriority();
    this.name = name.toCharArray();
    this.target = target;
    setPriority(priority);
    // 将父线程的InheritableThreadLocal复制过来
    if (parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals = ThreadLocal.createInheritedMap(parent.
            inheritableThreadLocals);
    // 分配一个线程ID
    tid = nextThreadID();
}
```

在上述过程中，一个新构造的线程对象是由其parent线程来进行空间分配的，而child线程继承了parent是否为Daemon、优先级和加载资源的contextClassLoader以及可继承的ThreadLocal，同时还会分配一个唯一的ID来标识这个child线程。至此，一个能够运行的线程对象就初始化好了，在堆内存中等待着运行。

4.2.2 启动线程

线程对象在初始化完成之后，调用start()方法就可以启动这个线程。线程start()方法的含义是：当前线程（即parent线程）同步告知Java虚拟机，只要线程规划器空闲，应立即启动调用start()方法的线程。

 **注意** 启动一个线程前，最好为这个线程设置线程名称，因为这样在使用jstack分析程序或者进行问题排查时，就会给开发人员提供一些提示，自定义的线程最好能够起个名字。

4.2.3 理解中断

中断可以理解为线程的一个标识位属性，它表示一个运行中的线程是否被其他线程进行了中断操作。中断好比其他线程对该线程打了个招呼，其他线程通过调用该线程的interrupt()方法对其进行中断操作。

线程通过检查自身是否被中断来进行响应，线程通过方法isInterrupted()来进行判断是否被中断，也可以调用静态方法Thread.interrupted()对当前线程的中断标识位进行复位。如果该线程已经处于终结状态，即使该线程被中断过，在调用该线程对象的isInterrupted()时依旧会返回false。

从Java的API中可以看到，许多声明抛出InterruptedException的方法（例如Thread.sleep(long millis)方法）这些方法在抛出InterruptedException之前，Java虚拟机会先将该线程的中断标识位清除，然后抛出InterruptedException，此时调用isInterrupted()方法将会返回false。

在代码清单4-7所示的例子中，首先创建了两个线程，SleepThread和BusyThread，前者不停地睡眠，后者一直运行，然后对这两个线程分别进行中断操作，观察二者的中断标识位。

代码清单4-7 Interrupted.java

```
public class Interrupted {
    public static void main(String[] args) throws Exception {
        // sleepThread不停的尝试睡眠
        Thread sleepThread = new Thread(new SleepRunner(), "SleepThread");
        sleepThread.setDaemon(true);
        // busyThread不停的运行
        Thread busyThread = new Thread(new BusyRunner(), "BusyThread");
        busyThread.setDaemon(true);
        sleepThread.start();
        busyThread.start();
        // 休眠5秒，让sleepThread和busyThread充分运行
        TimeUnit.SECONDS.sleep(5);
        sleepThread.interrupt();
        busyThread.interrupt();
        System.out.println("SleepThread interrupted is " + sleepThread.isInterrupted());
        System.out.println("BusyThread interrupted is " + busyThread.isInterrupted());
        // 防止sleepThread和busyThread立刻退出
    }
}
```

```
        SleepUtils.second(2);
    }
    static class SleepRunner implements Runnable {
        @Override
        public void run() {
            while (true) {
                SleepUtils.second(10);
            }
        }
    }
    static class BusyRunner implements Runnable {
        @Override
        public void run() {
            while (true) {
            }
        }
    }
}
```

输出如下。

```
SleepThread interrupted is false
BusyThread interrupted is true
```

从结果可以看出，抛出InterruptedException的线程SleepThread，其中断标识位被清除了，而一直忙碌运作的线程BusyThread，中断标识位没有被清除。

4.2.4 过期的suspend()、resume()和stop()

大家对于CD机肯定不会陌生，如果把它播放音乐比作一个线程的运作，那么对音乐播放做出的暂停、恢复和停止操作对应在线程Thread的API就是suspend()、resume()和stop()。

在代码清单4-8所示的例子中，创建了一个线程PrintThread，它以1秒的频率进行打印，而主线程对其进行暂停、恢复和停止操作。

代码清单4-8 Deprecated.java

```
public class Deprecated {
    public static void main(String[] args) throws Exception {
        DateFormat format = new SimpleDateFormat("HH:mm:ss");
        Thread printThread = new Thread(new Runner(), "PrintThread");
        printThread.setDaemon(true);
        printThread.start();
        TimeUnit.SECONDS.sleep(3);
        // 将PrintThread进行暂停, 输出内容工作停止
        printThread.suspend();
        System.out.println("main suspend PrintThread at " + format.format(new Date()));
        TimeUnit.SECONDS.sleep(3);
        // 将PrintThread进行恢复, 输出内容继续
        printThread.resume();
        System.out.println("main resume PrintThread at " + format.format(new Date()));
        TimeUnit.SECONDS.sleep(3);
        // 将PrintThread进行终止, 输出内容停止
        printThread.stop();
        System.out.println("main stop PrintThread at " + format.format(new Date()));
        TimeUnit.SECONDS.sleep(3);
    }
    static class Runner implements Runnable {
        @Override
        public void run() {
            DateFormat format = new SimpleDateFormat("HH:mm:ss");
            while (true) {
                System.out.println(Thread.currentThread().getName() + " Run at " +
                    format.format(new Date()));
                SleepUtils.second(1);
            }
        }
    }
}
```

输出如下(输出内容中的时间与示例执行的具体时间相关)。

```
PrintThread Run at 17:34:36
PrintThread Run at 17:34:37
PrintThread Run at 17:34:38
main suspend PrintThread at 17:34:39
main resume PrintThread at 17:34:42
PrintThread Run at 17:34:42
PrintThread Run at 17:34:43
PrintThread Run at 17:34:44
main stop PrintThread at 17:34:45
```

在执行过程中，PrintThread运行了3秒，随后被暂停，3秒后恢复，最后经过3秒被终止。

通过示例的输出可以看到，suspend()、resume()和stop()方法完成了线程的暂停、恢复和终止工作，而且非常“人性化”。但是这些API是过期的，也就是不建议使用的。

不建议使用的原因主要有：以suspend()方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。同样，stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。



注意 正因为suspend()、resume()和stop()方法带来的副作用，这些方法才被标注为不建议使用的过期方法，而暂停和恢复操作可以用后面提到的等待/通知机制来替代。

4.2.5 安全地终止线程

在4.2.3节中提到的中断状态是线程的一个标识位，而中断操作是一种简便的线程间交互方式，而这种交互方式最适合用来取消或停止任务。除了中断以外，还可以利用一个boolean变量来控制是否需要停止任务并终止该线程。

在代码清单4-9所示的例子中，创建了一个线程CountThread，它不断地进行变量累加，而主线程尝试对其进行中断操作和停止操作。

代码清单4-9 Shutdown.java

```
public class Shutdown {
    public static void main(String[] args) throws Exception {
        Runner one = new Runner();
        Thread countThread = new Thread(one, "CountThread");
        countThread.start();
        // 睡眠1秒, main线程对CountThread进行中断, 使CountThread能够感知中断而结束
        TimeUnit.SECONDS.sleep(1);
        countThread.interrupt();
        Runner two = new Runner();
        countThread = new Thread(two, "CountThread");
        countThread.start();
        // 睡眠1秒, main线程对Runner two进行取消, 使CountThread能够感知on为false而结束
        TimeUnit.SECONDS.sleep(1);
        two.cancel();
    }
    private static class Runner implements Runnable {
        private long i;
        private volatile boolean on = true;
        @Override
        public void run() {
            while (on && !Thread.currentThread().isInterrupted()) {
                i++;
            }
            System.out.println("Count i = " + i);
        }
        public void cancel() {
            on = false;
        }
    }
}
```

输出结果如下所示(输出内容可能不同)。

```
Count i = 543487324
Count i = 540898082
```

示例在执行过程中，main线程通过中断操作和cancel()方法均可使CountThread得以终止。这种通过标识位或者中断操作的方式能够使线程在终止时有机会去清理资源，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅。

4.3 线程间通信

线程开始运行，拥有自己的栈空间，就如同一个脚本一样，按照既定的代码一步一步地执行，直到终止。但是，每个运行中的线程，如果仅仅是孤立地运行，那么没有一点儿价值，或者说价值很少，如果多个线程能够相互配合完成工作，这将会带来巨大的价值。

4.3.1 volatile和synchronized关键字

Java支持多个线程同时访问一个对象或者对象的成员变量, 由于每个线程可以拥有这个变量的拷贝(虽然对象以及成员变量分配的内存是在共享内存中的, 但是每个执行的线程还是可以拥有一份拷贝, 这样做的目的是加速程序的执行, 这是现代多核处理器的一个显著特性), 所以程序在执行过程中, 一个线程看到的变量并不一定是最新的。

关键字volatile可以用来修饰字段(成员变量), 就是告知程序任何对该变量的访问均需要从共享内存中获取, 而对它的改变必须同步刷新回共享内存, 它能保证所有线程对变量访问的可见性。

举个例子, 定义一个表示程序是否运行的成员变量boolean on=true, 那么另一个线程可能对它执行关闭动作(on=false), 这里涉及多个线程对变量的访问, 因此需要将其定义成为volatile boolean on=true, 这样其他线程对它进行改变时, 可以让所有线程感知到变化, 因为所有对on变量的访问和修改都需要以共享内存为准。但是, 过多地使用volatile是不必要的, 因为它会降低程序执行的效率。

关键字synchronized可以修饰方法或者以同步块的形式来进行使用, 它主要确保多个线程在同一个时刻, 只能有一个线程处于方法或者同步块中, 它保证了线程对变量访问的可见性和排他性。

在代码清单4-10所示的例子中, 使用了同步块和同步方法, 通过使用javap工具查看生成的class文件信息来分析synchronized关键字的实现细节, 示例如下。

代码清单4-10 Synchronized.java

```
public class Synchronized {  
    public static void main(String[] args) {  
        // 对Synchronized Class对象进行加锁  
        synchronized (Synchronized.class) {  
        }  
    }  
}
```



```
        // 静态同步方法, 对Synchronized Class对象进行加锁
        m();
    }
    public static synchronized void m() {
    }
}
```

在Synchronized.class同级目录执行javap-v Synchronized.class, 部分相关输出如下所示:

```
public static void main(java.lang.String[]);
    // 方法修饰符, 表示: public static flags: ACC_PUBLIC, ACC_STATIC
    Code:
        stack=2, locals=1, args_size=1
        0: ldc         #1    // class com/murdock/books/multithread/book/Synchronized
        2: dup
        3: monitorenter    // monitorenter:监视器进入, 获取锁
        4: monitorexit    // monitorexit:监视器退出, 释放锁
        5: invokestatic    #16 // Method m:()V
        8: return
    public static synchronized void m();
    // 方法修饰符, 表示: public static synchronized
    flags: ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
    Code:
        stack=0, locals=0, args_size=0
        0: return
```

上面class信息中, 对于同步块的实现使用了monitorenter和monitorexit指令, 而同步方法则是依靠方法修饰符上的ACC_SYNCHRONIZED来完成的。无论采用哪种方式, 其本质是对一个对象的监视器(monitor)进行获取, 而这个获取过程是排他的, 也就是同一时刻只能有一个线程获取到由synchronized所保护对象的监视器。

任意一个对象都拥有自己的监视器, 当这个对象由同步块或者这个对象的同步方法调用时, 执行方法的线程必须先获取到该对象的监视器才能进入同步块或者同步方法, 而没有获取到监视器(执行该方法)的线程将会被阻塞在同步块和同步方法的入口处, 进入BLOCKED状态。

图4-2描述了对象、对象的监视器、同步队列和执行线程之间的关系。

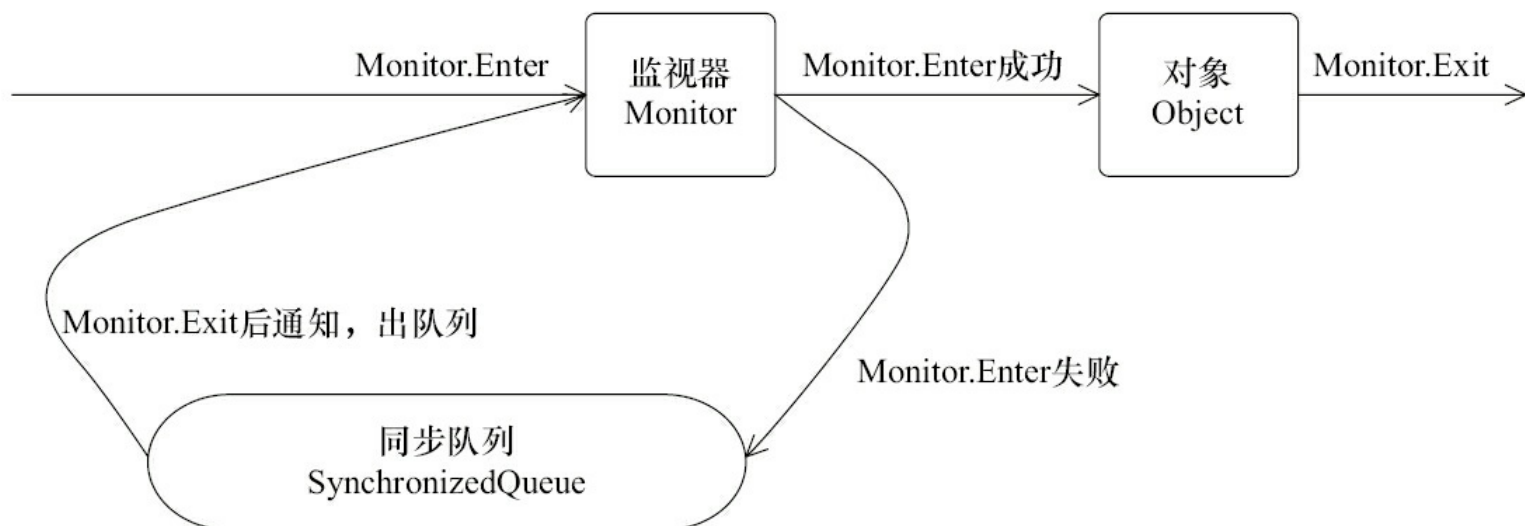


图4-2 对象、监视器、同步队列和执行线程之间的关系

从图4-2中可以看到，任意线程对Object(Object由synchronized保护)的访问，首先要获得Object的监视器。如果获取失败，线程进入同步队列，线程状态变为BLOCKED。当访问Object的前驱(获得了锁的线程)释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。

4.3.2 等待/通知机制

一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作，整个过程开始于一个线程，而最终执行又是另一个线程。前者是生产者，后者就是消费者，这种模式隔离了“做什么”(what)和“怎么做”(How)，在功能层面上实现了解耦，体系结构上具备了良好的伸缩性，但是在Java语言中如何实现类似的功能呢？

简单的办法是让消费者线程不断地循环检查变量是否符合预期，如下面代码所示，在while循环中设置不满足的条件，如果条件满足则退出while循环，从而完成消费者的工作。

```
while (value != desire) {
    Thread.sleep(1000);
}
doSomething();
```

上面这段伪代码在条件不满足时就睡眠一段时间，这样做的目的是防止过快的“无效”尝试，这种方式看似能够解实现所需的功能，但是却存在如下问题。

- 1)难以确保及时性。在睡眠时，基本不消耗处理器资源，但是如果睡得过久，就不能及时发现条件已经变化，也就是及时性难以保证。
- 2)难以降低开销。如果降低睡眠的时间，比如休眠1毫秒，这样消费者能更加迅速地发现条件变化，但是却可能消耗更多的处理器资源，造成了无端的浪费。

以上两个问题，看似矛盾难以调和，但是Java通过内置的等待/通知机制能够很好地解决这个矛盾并实现所需的功能。

等待/通知的相关方法是任意Java对象都具备的，因为这些方法被定义在所有对象的超类java.lang.Object上，方法和描述如表4-2所示。

表4-2 等待/通知的相关方法

方法名称	描 述
notify()	通知一个在对象上等待的线程，使其从 wait() 方法返回，而返回的前提是该线程获取到了对象的锁
notifyAll()	通知所有等待在该对象上的线程
wait()	调用该方法的线程进入 WAITING 状态，只有等待另外线程的通知或被中断才会返回，需要注意，调用 wait() 方法后，会释放对象的锁
wait(long)	超时等待一段时间，这里的参数时间是毫秒，也就是等待长达 n 毫秒，如果没有通知就超时返回
wait(long, int)	对于超时时间更细粒度的控制，可以达到纳秒

等待/通知机制，是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()或者notifyAll()方法，线程A收到通知后从对象O的wait()方法返回，进而执行后续操作。上述两个线程通过对象O来完成交互，而对象上的wait()和notify/notifyAll()的关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

在代码清单4-11所示的例子中，创建了两个线程——WaitThread和NotifyThread，前者检查flag值是否为false，如果符合要求，进行后续操作，否则在lock上等待，后者在睡眠了一段时间后对lock进行通知，示例如下所示。

代码清单4-11 WaitNotify.java

```
public class WaitNotify {
    static boolean flag = true;
    static Object lock = new Object();
    public static void main(String[] args) throws Exception {
        Thread waitThread = new Thread(new Wait(), "WaitThread");
        waitThread.start();
        TimeUnit.SECONDS.sleep(1);
        Thread notifyThread = new Thread(new Notify(), "NotifyThread");
        notifyThread.start();
    }
    static class Wait implements Runnable {
        public void run() {
            // 加锁, 拥有lock的Monitor
            synchronized (lock) {
                // 当条件不满足时, 继续wait, 同时释放了lock的锁
                while (flag) {
                    try {
                        System.out.println(Thread.currentThread() + " flag is true. wa
                        @ " + new SimpleDateFormat("HH:mm:ss").format(new Date()));
                        lock.wait();
                    } catch (InterruptedException e) {
                    }
                }
            }
            // 条件满足时, 完成工作
        }
    }
}
```

```

        System.out.println(Thread.currentThread() + " flag is false. running
        @ " + new SimpleDateFormat("HH:mm:ss").format(new Date()));
    }
}
}
static class Notify implements Runnable {
    public void run() {
        // 加锁, 拥有lock的Monitor
        synchronized (lock) {
            // 获取lock的锁, 然后进行通知, 通知时不会释放lock的锁,
            // 直到当前线程释放了lock后, WaitThread才能从wait方法中返回
            System.out.println(Thread.currentThread() + " hold lock. notify @ " +
            new SimpleDateFormat("HH:mm:ss").format(new Date()));
            lock.notifyAll();
            flag = false;
            SleepUtils.second(5);
        }
        // 再次加锁
        synchronized (lock) {
            System.out.println(Thread.currentThread() + " hold lock again. sleep
            @ " + new SimpleDateFormat("HH:mm:ss").format(new Date()));
            SleepUtils.second(5);
        }
    }
}
}
}

```

输出如下(输出内容可能不同, 主要区别在时间上)。

```

Thread[WaitThread,5,main] flag is true. wait @ 22:23:03
Thread[NotifyThread,5,main] hold lock. notify @ 22:23:04
Thread[NotifyThread,5,main] hold lock again. sleep @ 22:23:09
Thread[WaitThread,5,main] flag is false. running @ 22:23:14

```

上述第3行和第4行输出的顺序可能会互换, 而上述例子主要说明了调用wait()、notify()以及notifyAll()时需要注意的细节, 如下。

1) 使用wait()、notify()和notifyAll()时需要先对调用对象加锁。

2) 调用wait()方法后, 线程状态由RUNNING变为WAITING, 并将当前线程放置到对象的等待队列。

3) notify()或notifyAll()方法调用后, 等待线程依旧不会从wait()返回, 需要调用notify()或notifyAll()的线程释放锁之后, 等待线程才有机会从wait()返回。

4) notify()方法将等待队列中的一个等待线程从等待队列中移到同步队列中, 而notifyAll()方法则是将等待队列中所有的线程全部移到同步队列, 被移动的线程状态由WAITING变为BLOCKED。

5) 从wait()方法返回的前提是获得了调用对象的锁。

从上述细节中可以看到, 等待/通知机制依托于同步机制, 其目的就是确保等待线程从wait()方法返回时能够感知到通知线程对变量做出的修改。

图4-3描述了上述示例的过程。

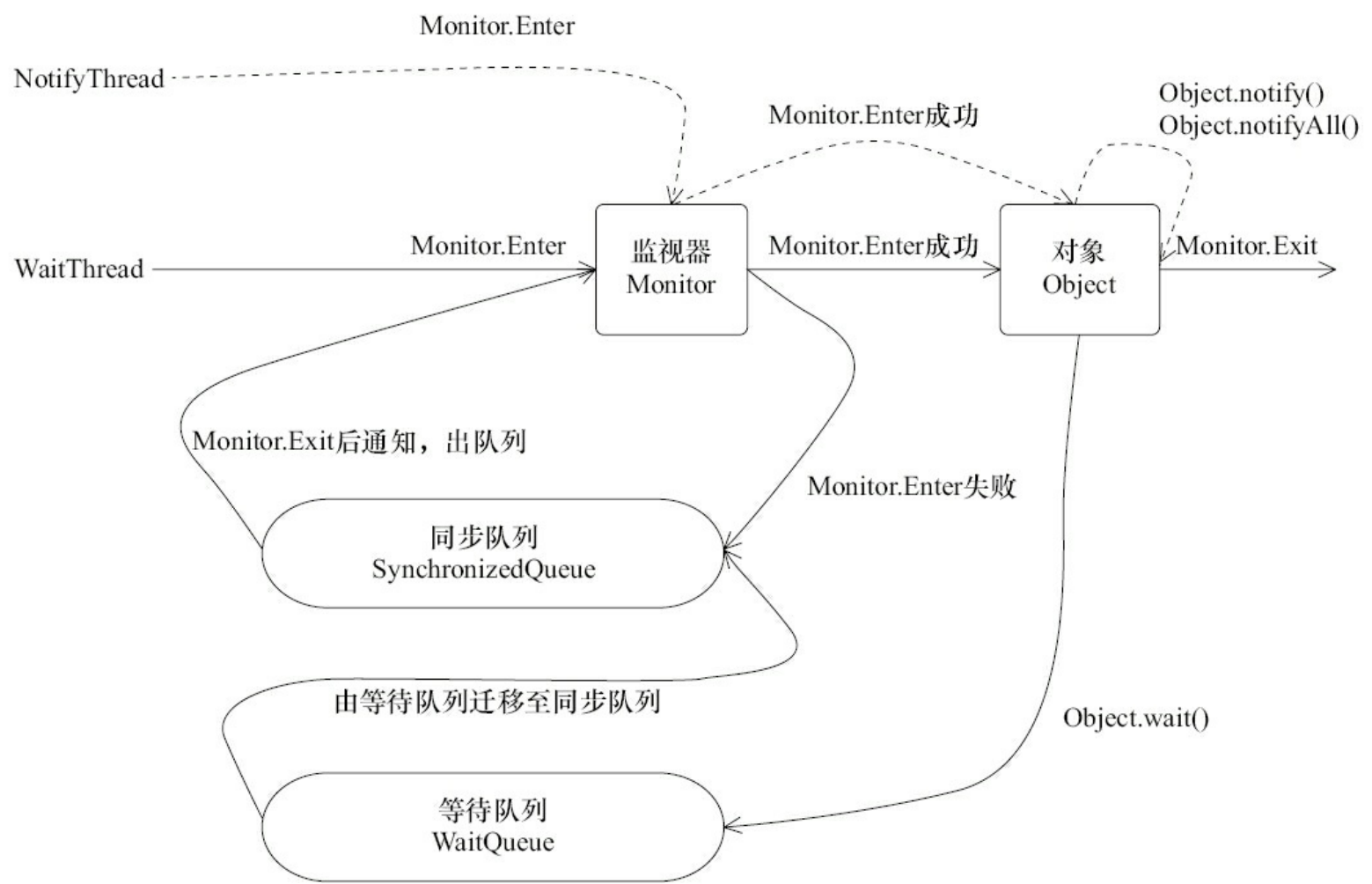


图4-3 WaitNotify.java运行过程

在图4-3中, WaitThread首先获取了对象的锁, 然后调用对象的wait()方法, 从而放弃了锁并进入了对象的等待队列WaitQueue中, 进入等待状态。由于WaitThread释放了对象的锁,

NotifyThread随后获取了对象的锁，并调用对象的notify()方法，将WaitThread从WaitQueue移到SynchronizedQueue中，此时WaitThread的状态变为阻塞状态。NotifyThread释放了锁之后，WaitThread再次获取到锁并从wait()方法返回继续执行。

4.3.3 等待/通知的经典范式

从4.3.2节中的WaitNotify示例中可以提炼出等待/通知的经典范式，该范式分为两部分，分别针对等待方(消费者)和通知方(生产者)。

等待方遵循如下原则。

- 1) 获取对象的锁。
- 2) 如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。
- 3) 条件满足则执行对应的逻辑。

对应的伪代码如下。

```
synchronized(对象) {  
    while(条件不满足) {  
        对象.wait();  
    }  
    对应的处理逻辑  
}
```

通知方遵循如下原则。

- 1) 获得对象的锁。
- 2) 改变条件。
- 3) 通知所有等待在对象上的线程。

对应的伪代码如下。

```
synchronized(对象) {  
    改变条件  
    对象.notifyAll();  
}
```

4.3.4 管道输入/输出流

管道输入/输出流和普通的文件输入/输出流或者网络输入/输出流不同之处在于，它主要用于线程之间的数据传输，而传输的媒介为内存。

管道输入/输出流主要包括了如下4种具体实现：PipedOutputStream、PipedInputStream、PipedReader和PipedWriter，前两种面向字节，而后两种面向字符。

在代码清单4-12所示的例子中，创建了printThread，它用来接受main线程的输入，任何main线程的输入均通过PipedWriter写入，而printThread在另一端通过PipedReader将内容读出并打印。

代码清单4-12 Piped.java

```
public class Piped {
    public static void main(String[] args) throws Exception {
        PipedWriter out = new PipedWriter();
        PipedReader in = new PipedReader();
        // 将输出流和输入流进行连接, 否则在使用时会抛出IOException
        out.connect(in);
        Thread printThread = new Thread(new Print(in), "PrintThread");
        printThread.start();
        int receive = 0;
        try {
            while ((receive = System.in.read()) != -1) {
                out.write(receive);
            }
        } finally {
            out.close();
        }
    }
    static class Print implements Runnable {
        private PipedReader in;
        public Print(PipedReader in) {
            this.in = in;
        }
        public void run() {
            int receive = 0;
            try {
                while ((receive = in.read()) != -1) {
                    System.out.print((char) receive);
                }
            } catch (IOException ex) {
```

```
}  
    }  
}  
}
```

运行该示例，输入一组字符串，可以看到被printThread进行了原样输出。

```
Repeat my words.  
Repeat my words.
```

对于Piped类型的流，必须先要进行绑定，也就是调用connect()方法，如果没有将输入/输出流绑定起来，对于该流的访问将会抛出异常。

4.3.5 Thread.join()的使用

如果一个线程A执行了thread.join()语句,其含义是:当前线程A等待thread线程终止之后才从thread.join()返回。线程Thread除了提供join()方法之外,还提供了join(long millis)和join(long millis,int nanos)两个具备超时特性的方法。这两个超时方法表示,如果线程thread在给定的超时时间里没有终止,那么将会从该超时方法中返回。

在代码清单4-13所示的例子中,创建了10个线程,编号0~9,每个线程调用前一个线程的join()方法,也就是线程0结束了,线程1才能从join()方法中返回,而线程0需要等待main线程结束。

代码清单4-13 Join.java

```
public class Join {
    public static void main(String[] args) throws Exception {
        Thread previous = Thread.currentThread();
        for (int i = 0; i < 10; i++) {
            // 每个线程拥有前一个线程的引用,需要等待前一个线程终止,才能从等待中返回
            Thread thread = new Thread(new Domino(previous), String.valueOf(i));
            thread.start();
            previous = thread;
        }
        TimeUnit.SECONDS.sleep(5);
        System.out.println(Thread.currentThread().getName() + " terminate.");
    }
    static class Domino implements Runnable {
        private Thread thread;
        public Domino(Thread thread) {
            this.thread = thread;
        }
        public void run() {
            try {
                thread.join();
            } catch (InterruptedException e) {
            }
            System.out.println(Thread.currentThread().getName() + " terminate.");
        }
    }
}
```

输出如下。

```
main terminate.  
0 terminate.  
1 terminate.  
2 terminate.  
3 terminate.  
4 terminate.  
5 terminate.  
6 terminate.  
7 terminate.  
8 terminate.  
9 terminate.
```

从上述输出可以看到，每个线程终止的前提是前驱线程的终止，每个线程等待前驱线程终止后，才从join()方法返回，这里涉及了等待/通知机制（等待前驱线程结束，接收前驱线程结束通知）。

代码清单4-14是JDK中Thread.join()方法的源码(进行了部分调整)。

代码清单4-14 Thread.java

```
// 加锁当前线程对象  
public final synchronized void join() throws InterruptedException {  
    // 条件不满足, 继续等待  
    while (isAlive()) {  
        wait(0);  
    }  
    // 条件符合, 方法返回  
}
```

当线程终止时，会调用线程自身的notifyAll()方法，会通知所有等待在该线程对象上的线程。可以看到join()方法的逻辑结构与4.3.3节中描述的等待/通知经典范式一致，即加锁、循环和处理逻辑3个步骤。

4.3.6 ThreadLocal的使用

ThreadLocal, 即线程变量, 是一个以ThreadLocal对象为键、任意对象为值的存储结构。这个结构被附带在线程上, 也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值。

可以通过set(T)方法来设置一个值, 在当前线程下再通过get()方法获取到原先设置的值。

在代码清单4-15所示的例子中, 构建了一个常用的Profiler类, 它具有begin()和end()两个方法, 而end()方法返回从begin()方法调用开始到end()方法被调用时的时间差, 单位是毫秒。

代码清单4-15 Profiler.java

```
public class Profiler {
    // 第一次get()方法调用时会进行初始化(如果set方法没有调用), 每个线程会调用一次
    private static final ThreadLocal<Long> TIME_THREADLOCAL = new ThreadLocal<Long>() {
        protected Long initialValue() {
            return System.currentTimeMillis();
        }
    };
    public static final void begin() {
        TIME_THREADLOCAL.set(System.currentTimeMillis());
    }
    public static final long end() {
        return System.currentTimeMillis() - TIME_THREADLOCAL.get();
    }
    public static void main(String[] args) throws Exception {
        Profiler.begin();
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Cost: " + Profiler.end() + " mills");
    }
}
```

输出结果如下所示。

```
Cost: 1001 mills
```

Profiler可以被复用在方法调用耗时统计的功能上, 在方法的入口前执行begin()方法, 在

方法调用后执行end()方法，好处是两个方法的调用不用在一个方法或者类中，比如在AOP(面向方面编程)中，可以在方法调用前的切入点执行begin()方法，而在方法调用后的切入点执行end()方法，这样依旧可以获得方法的执行耗时。

4.4 线程应用实例

4.4.1 等待超时模式

开发人员经常会遇到这样的方法调用场景：调用一个方法时等待一段时间（一般来说是给
定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，
超时返回默认结果。

前面的章节介绍了等待/通知的经典范式，即加锁、条件循环和处理逻辑3个步骤，而这种
范式无法做到超时等待。而超时等待的加入，只需要对经典范式做出非常小的改动，改动内容
如下所示。

假设超时时间段是 T ，那么可以推断出在当前时间 $\text{now}+T$ 之后就会超时。

定义如下变量。

·等待持续时间： $\text{REMAINING}=T$ 。

·超时时间： $\text{FUTURE}=\text{now}+T$ 。

这时仅需要 $\text{wait}(\text{REMAINING})$ 即可，在 $\text{wait}(\text{REMAINING})$ 返回之后会将执行：

$\text{REMAINING}=\text{FUTURE}-\text{now}$ 。如果 REMAINING 小于等于0，表示已经超时，直接退出，否则将
继续执行 $\text{wait}(\text{REMAINING})$ 。

上述描述等待超时模式的伪代码如下。

```
// 对当前对象加锁
public synchronized Object get(long mills) throws InterruptedException {
    long future = System.currentTimeMillis() + mills;
    long remaining = mills;
    // 当超时大于0并且result返回值不满足要求
    while ((result == null) && remaining > 0) {
        wait(remaining);
    }
}
```



```
        remaining = future - System.currentTimeMillis();  
    }  
    return result;  
}
```

可以看出，等待超时模式就是在等待/通知范式基础上增加了超时控制，这使得该模式相比原有范式更具有灵活性，因为即使方法执行时间过长，也不会“永久”阻塞调用者，而是会按照调用者的要求“按时”返回。

4.4.2 一个简单的数据库连接池示例

我们使用等待超时模式来构造一个简单的数据库连接池，在示例中模拟从连接池中获取、使用和释放连接的过程，而客户端获取连接的过程被设定为等待超时的模式，也就是在1000毫秒内如果无法获取到可用连接，将会返回给客户端一个null。设定连接池的大小为10个，然后通过调节客户端的线程数来模拟无法获取连接的场景。

首先看一下连接池的定义。它通过构造函数初始化连接的最大上限，通过一个双向队列来维护连接，调用方需要先调用fetchConnection(long)方法来指定在多少毫秒内超时获取连接，当连接使用完成后，需要调用releaseConnection(Connection)方法将连接放回线程池，示例如代码清单4-16所示。

代码清单4-16 ConnectionPool.java

```
public class ConnectionPool {
    private LinkedList<Connection> pool = new LinkedList<Connection>();
    public ConnectionPool(int initialSize) {
        if (initialSize > 0) {
            for (int i = 0; i < initialSize; i++) {
                pool.addLast(ConnectionDriver.createConnection());
            }
        }
    }
    public void releaseConnection(Connection connection) {
        if (connection != null) {
            synchronized (pool) {
                // 连接释放后需要进行通知, 这样其他消费者能够感知到连接池中已经归还了一个连接
                pool.addLast(connection);
                pool.notifyAll();
            }
        }
    }
    // 在mills内无法获取到连接, 将会返回null
    public Connection fetchConnection(long mills) throws InterruptedException {
        synchronized (pool) {
            // 完全超时
            if (mills <= 0) {
                while (pool.isEmpty()) {
                    pool.wait();
                }
                return pool.removeFirst();
            } else {
```

```

        long future = System.currentTimeMillis() + mills;
        long remaining = mills;
        while (pool.isEmpty() && remaining > 0) {
            pool.wait(remaining);
            remaining = future - System.currentTimeMillis();
        }
        Connection result = null;
        if (!pool.isEmpty()) {
            result = pool.removeFirst();
        }
        return result;
    }
}
}
}

```

由于java.sql.Connection是一个接口，最终的实现是由数据库驱动提供方来实现的，考虑到只是个示例，我们通过动态代理构造了一个Connection，该Connection的代理实现仅仅是在commit()方法调用时休眠100毫秒，示例如代码清单4-17所示。

代码清单4-17 ConnectionDriver.java

```

public class ConnectionDriver {
    static class ConnectionHandler implements InvocationHandler {
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            if (method.getName().equals("commit")) {
                TimeUnit.MILLISECONDS.sleep(100);
            }
            return null;
        }
    }
    // 创建一个Connection的代理, 在commit时休眠100毫秒
    public static final Connection createConnection() {
        return (Connection) Proxy.newProxyInstance(ConnectionDriver.class.getClassLoader(),
            new Class<>[] { Connection.class }, new ConnectionHandler());
    }
}

```

下面通过一个示例来测试简易数据库连接池的工作情况，模拟客户端ConnectionRunner获取、使用、最后释放连接的过程，当它使用时连接将会增加获取到连接的数量，反之，将会增加未获取到连接的数量，示例如代码清单4-18所示。

代码清单4-18 ConnectionPoolTest.java

```

public class ConnectionPoolTest {
    static ConnectionPool pool = new ConnectionPool(10);
    // 保证所有ConnectionRunner能够同时开始
    static CountdownLatch start = new CountdownLatch(1);
    // main线程将会等待所有ConnectionRunner结束后才能继续执行
    static CountdownLatch end;
    public static void main(String[] args) throws Exception {
        // 线程数量, 可以修改线程数量进行观察
        int threadCount = 10;
        end = new CountdownLatch(threadCount);
        int count = 20;
        AtomicInteger got = new AtomicInteger();
        AtomicInteger notGot = new AtomicInteger();
        for (int i = 0; i < threadCount; i++) {
            Thread thread = new Thread(new ConnetionRunner(count, got, notGot),
                "ConnectionRunnerThread");
            thread.start();
        }
        start.countDown();
        end.await();
        System.out.println("total invoke: " + (threadCount * count));
        System.out.println("got connection: " + got);
        System.out.println("not got connection " + notGot);
    }
    static class ConnetionRunner implements Runnable {
        int count;
        AtomicInteger got;
        AtomicInteger notGot;
        public ConnetionRunner(int count, AtomicInteger got, AtomicInteger notGot) {
            this.count = count;
            this.got = got;
            this.notGot = notGot;
        }
        public void run() {
            try {
                start.await();
            } catch (Exception ex) {
            }
            while (count > 0) {
                try {
                    // 从线程池中获取连接, 如果1000ms内无法获取到, 将会返回null
                    // 分别统计连接获取的数量got和未获取到的数量notGot
                    Connection connection = pool.fetchConnection(1000);
                    if (connection != null) {
                        try {
                            connection.createStatement();
                            connection.commit();
                        } finally {
                            pool.releaseConnection(connection);
                            got.incrementAndGet();
                        }
                    } else {
                        notGot.incrementAndGet();
                    }
                } catch (Exception ex) {
                } finally {
                    count--;
                }
            }
        }
    }
}

```

```
        }
        end.countDown();
    }
}
```

上述示例中使用了CountDownLatch来确保ConnectionRunnerThread能够同时开始执行，并且在全部结束之后，才使main线程从等待状态中返回。当前设定的场景是10个线程同时运行获取连接池(10个连接)中的连接，通过调节线程数量来观察未获取到连接的情况。线程数、总获取次数、获取到的数量、未获取到的数量以及未获取到的比率，如表4-3所示(笔者机器CPU: i7-3635QM, 内存为8GB, 实际输出可能与此表不同)。

表4-3 线程数量与连接获取的关系

线程数量	总获取次数	获取到次数	未获取到次数	未获取到比率
10	200	200	0	0%
20	400	387	13	3.25%
30	600	542	58	9.67%
40	800	700	100	12.5%
50	1 000	828	172	17.2%

从表中的数据统计可以看出，在资源一定的情况下(连接池中的10个连接)，随着客户端线程的逐步增加，客户端出现超时无法获取连接的比率不断升高。虽然客户端线程在这种超时获取的模式下会出现连接无法获取的情况，但是它能够保证客户端线程不会一直挂在连接获取的操作上，而是“按时”返回，并告知客户端连接获取出现问题，是系统的一种自我保护机制。数据库连接池的设计也可以复用到其他的资源获取的场景，针对昂贵资源(比如数据库连接)的获取都应该加以超时限制。

4.4.3 线程池技术及其示例

对于服务端的程序，经常面对的是客户端传入的短小(执行时间短、工作内容较为单一)任务，需要服务端快速处理并返回结果。如果服务端每次接受到一个任务，创建一个线程，然后进行执行，这在原型阶段是个不错的选择，但是面对成千上万的任务递交进服务器时，如果还是采用一个任务一个线程的方式，那么将会创建数以万记的线程，这不是一个好的选择。因为这会使操作系统频繁的进行线程上下文切换，无故增加系统的负载，而线程的创建和消亡都是需要耗费系统资源的，也无疑浪费了系统资源。

线程池技术能够很好地解决这个问题，它预先创建了若干数量的线程，并且不能由用户直接对线程的创建进行控制，在这个前提下重复使用固定或较为固定数目的线程来完成任务的执行。这样做的好处是，一方面，消除了频繁创建和消亡线程的系统资源开销，另一方面，面对过量任务的提交能够平缓的劣化。

下面先看一个简单的线程池接口定义，示例如代码清单4-19所示。

代码清单4-19 ThreadPool.java

```
public interface ThreadPool<Job extends Runnable> {  
    // 执行一个Job, 这个Job需要实现Runnable  
    void execute(Job job);  
    // 关闭线程池  
    void shutdown();  
    // 增加工作者线程  
    void addWorkers(int num);  
    // 减少工作者线程  
    void removeWorker(int num);  
    // 得到正在等待执行的任务数量  
    int getJobSize();  
}
```

客户端可以通过execute(Job)方法将Job提交入线程池执行，而客户端自身不用等待Job的执行完成。除了execute(Job)方法以外，线程池接口提供了增大/减少工作者线程以及关闭线程池的方法。这里工作者线程代表着一个重复执行Job的线程，而每个由客户端提交的Job都将进

入到一个工作队列中等待工作者线程的处理。

接下来是线程池接口的默认实现，示例如代码清单4-20所示。

代码清单4-20 DefaultThreadPool.java

```
public class DefaultThreadPool<Job extends Runnable> implements ThreadPool<Job> {
    // 线程池最大限制数
    private static final int MAX_WORKER_NUMBERS = 10;
    // 线程池默认的数量
    private static final int DEFAULT_WORKER_NUMBERS = 5;
    // 线程池最小的数量
    private static final int MIN_WORKER_NUMBERS = 1;
    // 这是一个工作列表, 将会向里面插入工作
    private final LinkedList<Job> jobs = new LinkedList<Job>();
    // 工作者列表
    private final List<Worker> workers = Collections.synchronizedList(new
    ArrayList<Worker>());
    // 工作者线程的数量
    private int workerNum = DEFAULT_WORKER_NUMBERS;
    // 线程编号生成
    private AtomicLong threadNum = new AtomicLong();
    public DefaultThreadPool() {
        initializeWokers(DEFAULT_WORKER_NUMBERS);
    }
    public DefaultThreadPool(int num) {
        workerNum = num > MAX_WORKER_NUMBERS ? MAX_WORKER_NUMBERS : num < MIN_WORKER_
        NUMBERS ? MIN_WORKER_NUMBERS : num;
        initializeWokers(workerNum);
    }
    public void execute(Job job) {
        if (job != null) {
            // 添加一个工作, 然后进行通知
            synchronized (jobs) {
                jobs.addLast(job);
                jobs.notify();
            }
        }
    }
    public void shutdown() {
        for (Worker worker : workers) {
            worker.shutdown();
        }
    }
    public void addWorkers(int num) {
        synchronized (jobs) {
            // 限制新增的Worker数量不能超过最大值
            if (num + this.workerNum > MAX_WORKER_NUMBERS) {
                num = MAX_WORKER_NUMBERS - this.workerNum;
            }
            initializeWokers(num);
            this.workerNum += num;
        }
    }
}
```

```

    }
}
public void removeWorker(int num) {
    synchronized (jobs) {
        if (num >= this.workerNum) {
            throw new IllegalArgumentException("beyond workNum");
        }
        // 按照给定的数量停止Worker
        int count = 0;
        while (count < num) {
            Worker worker = workers.get(count)
            if (workers.remove(worker)) {
                worker.shutdown();
                count++;
            }
        }
        this.workerNum -= count;
    }
}
public int getJobSize() {
    return jobs.size();
}
// 初始化线程工作者
private void initializeWokers(int num) {
    for (int i = 0; i < num; i++) {
        Worker worker = new Worker();
        workers.add(worker);
        Thread thread = new Thread(worker, "ThreadPool-Worker-" + threadNum.
            incrementAndGet());
        thread.start();
    }
}
// 工作者, 负责消费任务
class Worker implements Runnable {
    // 是否工作
    private volatile boolean    running    = true;
    public void run() {
        while (running) {
            Job job = null;
            synchronized (jobs) {
                // 如果工作者列表是空的, 那么就wait
                while (jobs.isEmpty()) {
                    try {
                        jobs.wait();
                    } catch (InterruptedException ex) {
                        // 感知到外部对WorkerThread的中断操作, 返回
                        Thread.currentThread().interrupt();
                        return;
                    }
                }
                // 取出一个Job
                job = jobs.removeFirst();
            }
            if (job != null) {
                try {
                    job.run();
                } catch (Exception ex) {
                    // 忽略Job执行中的Exception
                }
            }
        }
    }
}

```



```
        }  
    }  
}  
}  
public void shutdown() {  
    running = false;  
}  
}
```

从线程池的实现可以看到, 当客户端调用execute(Job)方法时, 会不断地向任务列表jobs中添加Job, 而每个工作者线程会不断地从jobs上取出一个Job进行执行, 当jobs为空时, 工作者线程进入等待状态。

添加一个Job后, 对工作队列jobs调用了其notify()方法, 而不是notifyAll()方法, 因为能够确定有工作者线程被唤醒, 这时使用notify()方法将会比notifyAll()方法获得更小的开销(避免将等待队列中的线程全部移动到阻塞队列中)。

可以看到, 线程池的本质就是使用了一个线程安全的工作队列连接工作者线程和客户端线程, 客户端线程将任务放入工作队列后便返回, 而工作者线程则不断地从工作队列上取出工作并执行。当工作队列为空时, 所有的工作者线程均等待在工作队列上, 当有客户端提交了一个任务之后会通知任意一个工作者线程, 随着大量的任务被提交, 更多的工作者线程会被唤醒。

4.4.4 一个基于线程池技术的简单Web服务器

目前的浏览器都支持多线程访问，比如说在请求一个HTML页面的时候，页面中包含的图片资源、样式资源会被浏览器发起并发的获取，这样用户就不会遇到一直等到一个图片完全下载完成才能继续查看文字内容的尴尬情况。

如果Web服务器是单线程的，多线程的浏览器也没有用武之地，因为服务端还是一个请求一个请求的顺序处理。因此，大部分Web服务器都是支持并发访问的。常用的Java Web服务器，如Tomcat、Jetty，在其处理请求的过程中都使用到了线程池技术。

下面通过使用前一节中的线程池来构造一个简单的Web服务器，这个Web服务器用来处理HTTP请求，目前只能处理简单的文本和JPG图片内容。这个Web服务器使用main线程不断地接受客户端Socket的连接，将连接以及请求提交给线程池处理，这样使得Web服务器能够同时处理多个客户端请求，示例如代码清单4-21所示。

代码清单4-21 SimpleHttpServer.java

```
public class SimpleHttpServer {
    // 处理HttpRequest的线程池
    static ThreadPool<HttpRequestHandler> threadPool = new DefaultThreadPool
        <HttpRequestHandler>(1);
    // SimpleHttpServer的根路径
    static String basePath;
    static ServerSocket serverSocket;
    // 服务监听端口
    static int port = 8080;
    public static void setPort(int port) {
        if (port > 0) {
            SimpleHttpServer.port = port;
        }
    }
    public static void setBasePath(String basePath) {
        if (basePath != null && new File(basePath).exists() && new File(basePath).
            isDirectory()) {
            SimpleHttpServer.basePath = basePath;
        }
    }
    // 启动SimpleHttpServer
    public static void start() throws Exception {
        serverSocket = new ServerSocket(port);
```

```

Socket socket = null;
while ((socket = serverSocket.accept()) != null) {
    // 接收一个客户端Socket, 生成一个HttpRequestHandler, 放入线程池执行
    threadPool.execute(new HttpRequestHandler(socket));
}
serverSocket.close();
}

static class HttpRequestHandler implements Runnable {
    private Socket    socket;
    public HttpRequestHandler(Socket socket) {
        this.socket = socket;
    }
    @Override
    public void run() {
        String line = null;
        BufferedReader br = null;
        BufferedReader reader = null;
        PrintWriter out = null;
        InputStream in = null;
        try {
            reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            String header = reader.readLine();
            // 由相对路径计算出绝对路径
            String filePath = basePath + header.split(" ")[1];
            out = new PrintWriter(socket.getOutputStream());
            // 如果请求资源的后缀为jpg或者ico, 则读取资源并输出
            if (filePath.endsWith(".jpg") || filePath.endsWith(".ico")) {
                in = new FileInputStream(filePath);
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                int i = 0;
                while ((i = in.read()) != -1) {
                    baos.write(i);
                }
                byte[] array = baos.toByteArray();
                out.println("HTTP/1.1 200 OK");
                out.println("Server: Molly");
                out.println("Content-Type: image/jpeg");
                out.println("Content-Length: " + array.length);
                out.println("");
                socket.getOutputStream().write(array, 0, array.length);
            } else {
                br = new BufferedReader(new InputStreamReader(new
                    FileInputStream(filePath)));
                out = new PrintWriter(socket.getOutputStream());
                out.println("HTTP/1.1 200 OK");
                out.println("Server: Molly");
                out.println("Content-Type: text/html; charset=UTF-8");
                out.println("");
                while ((line = br.readLine()) != null) {
                    out.println(line);
                }
            }
            out.flush();
        } catch (Exception ex) {
            out.println("HTTP/1.1 500");
            out.println("");
            out.flush();
        } finally {

```

```

        close(br, in, reader, out, socket);
    }
}
// 关闭流或者Socket
private static void close(Closeable... closeables) {
    if (closeables != null) {
        for (Closeable closeable : closeables) {
            try {
                closeable.close();
            } catch (Exception ex) {
            }
        }
    }
}
}
}

```

该Web服务器处理用户请求的时序图如，图44所示。

在图4-4中，SimpleHttpServer在建立了与客户端的连接之后，并不会处理客户端的请求，而是将其包装成HttpRequestHandler并交由线程池处理。在线程池中的Worker处理客户端请求的同时，SimpleHttpServer能够继续完成后续客户端连接的建立，不会阻塞后续客户端的请求。

接下来，通过一个测试对比来认识线程池技术带来服务器吞吐量的提高。我们准备了一个简单的HTML页面，内容如代码清单4-22所示。

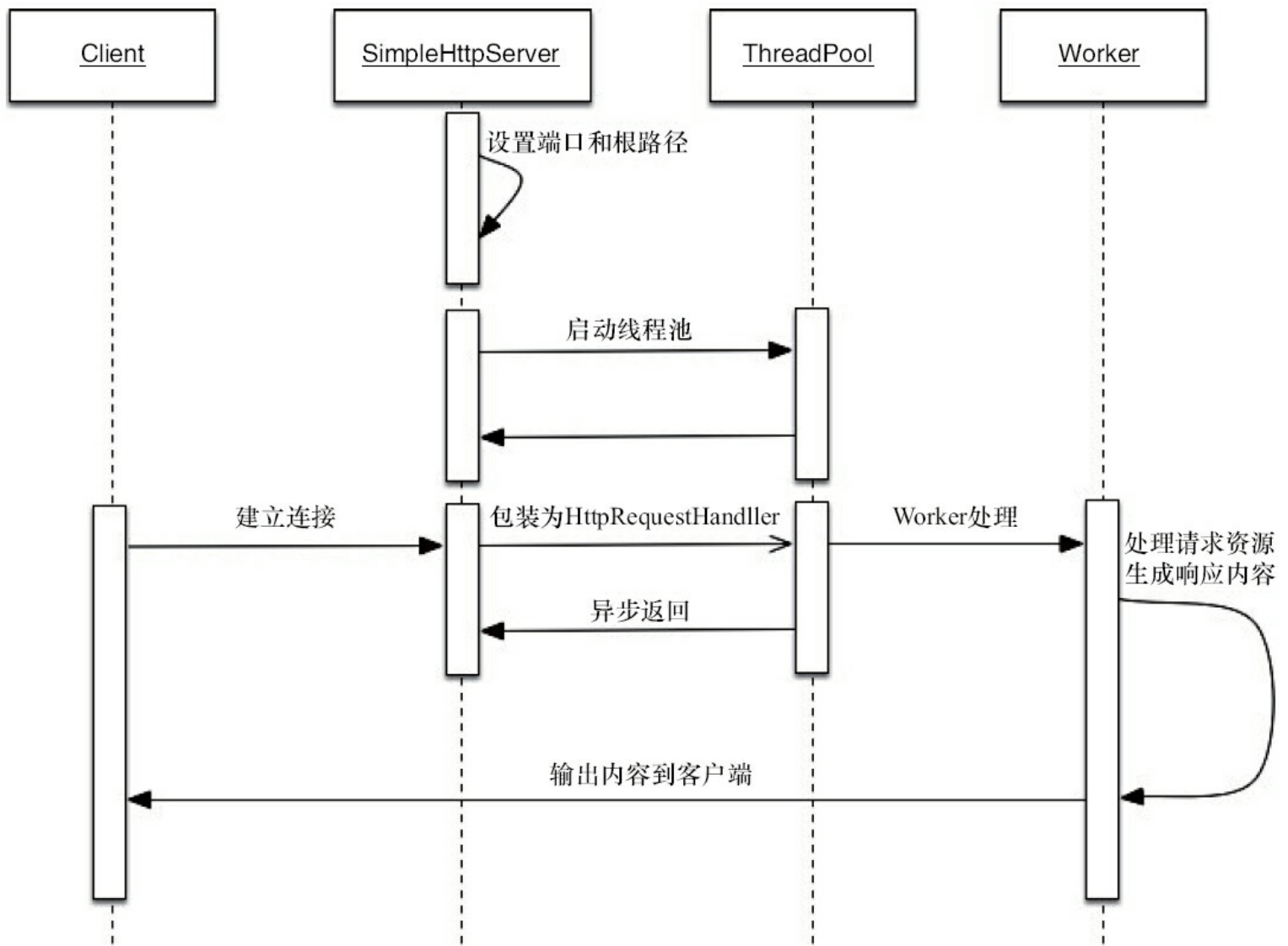


图4-4 SimpleHttpServer时序图

代码清单4-22 Index.html

```
<html>
  <head>
    <title>测试页面</title>
  </head>
  <body >
    <h1>第一张图片</h1>
    
    <h1>第二张图片</h1>
    
    <h1>第三张图片</h1>
    
  </body>
</html>
```

将SimpleHttpServer的根目录设定到该HTML页面所在目录，并启动SimpleHttpServer，通过Apache HTTP server benchmarking tool (版本2.3) 来测试不同线程数下，SimpleHttpServer的吞吐量表现。

测试场景是5000次请求，分10个线程并发执行，测试内容主要考察响应时间(越小越好)和每秒查询的数量(越高越好)，测试结果如表4-4所示(笔者机器CPU:i7-3635QM, 内存为8GB, 实际输出可能与此表不同)。

表4-4 测试结果

线程池线程数量	1	5	10
响应时间 (ms)	0.352	0.246	0.163
每秒查询的数量	3 076	4 065	6 123
测试完成时间 (s)	1.625	1.230	0.816

可以看到，随着线程池中线程数量的增加，SimpleHttpServer的吞吐量不断增大，响应时间不断变小，线程池的作用非常明显。

但是，线程池中线程数量并不是越多越好，具体的数量需要评估每个任务的处理时间，以及当前计算机的处理器能力和数量。使用的线程过少，无法发挥处理器的性能；使用的线程过多，将会增加系统的无故开销，起到相反的作用。

4.5 本章小结

本章从介绍多线程技术带来的好处开始, 讲述了如何启动和终止线程以及线程的状态, 详细阐述了多线程之间进行通信的基本方式和等待/通知经典范式。在线程应用示例中, 使用了等待超时、数据库连接池以及简单线程池3个不同的示例巩固本章前面章节所介绍的Java多线程基础知识。最后通过一个简单的Web服务器将上述知识点串联起来, 加深我们对这些知识点的理解。