1 什么是线程

多任务（multitasking）是操作系统的一种能力，看起来可以在同一时刻运行多个程序。操作系统会为每个进程分配CPU时间片，给人并行处理的感觉。

多线程程序在更低一层扩展了多任务的概念：单个程序看起来在同时完成多个任务。每个任务在一个**线程**（thread）中执行，线程是控制线程的简称。如果一个程序可以同时运行多个线程，则称这个程序是**多线程**的（multithreaded）。

多进程和多线程的本质区别在于，每个进程都有自己的一整套变量，而线程则共享数据。共享变量使线程之间的通信比进程之间的通信更有效、更容易。此外，在有些操作系统中，与进程相比，线程更"轻量级"，创建、撤销一个线程比启动新进程的开销要小得多。

创建和启动线程的简单过程为：

1. 将要执行的代码放在一个类的`run`方法中，这个类要实现`Runnable`接口。`Runnable`接口只有一个方法：

```
public interface Runnable
{
    void run();
}
```

2. 用这个`Runnable`构造一个`Thread`对象：

```
/* java.lang.Thread */
Thread(Runnable target) // 构造一个新线程，调用指定目标的 run 方法
```

3. 启动线程：

```
/* java.lang.Thread */
void start() // 启动这个线程，从而调用 run 方法。这个方法会立即返回，新线程会并发运行
```

下面的示例程序使用了两个线程：

```
public class ThreadTest
{
    public static final int DELAY = 10;
    public static final int STEPS = 100;
    public static final double MAX_AMOUNT = 1000;

    public static void main(String[] args)
    {
        Bank bank = new Bank(4, 100000); // 创建 4 个账户，每个账户初始金额为 100000
        Runnable task1 = () ->
        {
            try
            {
                for (int i = 0; i < STEPS; i++)
                {
                    double amount = MAX_AMOUNT * Math.random();
                    bank.transfer(0, 1, amount); // 从 0 向 1 转账
                    Thread.sleep((int) (DELAY * Math.random()));
                }
            }
            catch (InterruptedException e) {}
        };

        Runnable task2 = () ->
        {
```

```
            try
            {
                for (int i = 0; i < STEPS; i++)
                {
                    double amount = MAX_AMOUNT * Math.random();
                    bank.transfer(2, 3, amount); // 从 2 向 3 转账
                    Thread.sleep((int) (DELAY * Math.random()));
                }
            }
            catch (InterruptedException e) {}
        };

        new Thread(task1).start();
        new Thread(task2).start();
    }
}
```

还可以通过建立`Thread`类的子类来定义线程。在子类中重写`run`方法，然后构造这个子类的对象，调用`start`方法来启动线程。例如：

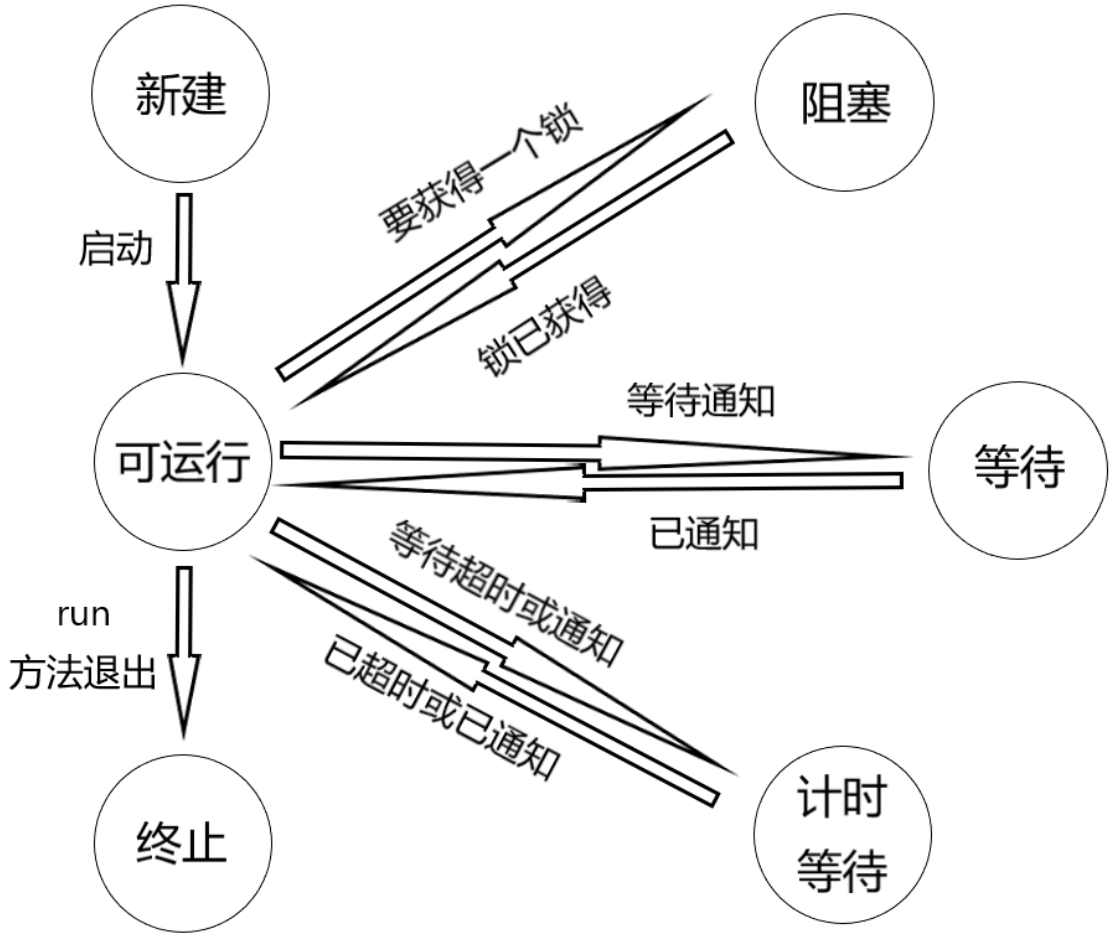
```
class MyThread extends Thread
{
    public void run()
    {
        // 任务代码
    }
}

new MyThread().start();
```

不要调用`Thread`类或`Runnable`对象的`run`方法，直接调用`run`方法只会在同一个线程中执行这个任务，而没有启动新的线程。

2 线程状态

线程有 6 种状态：New（新建）、Runnable（可运行）、Blocked（阻塞）、Waiting（等待）、Timed waiting（计时等待）、Terminated（终止）。这 6 种状态的关系如下图所示：



要确定一个线程的当前状态，只需要调用`getState`方法：

```
/* java.lang.Thread */
Thread.State getState()
    // 得到这个线程的状态，取值为 NEW、RUNNABLE、BLOCKED、WAITING、TIMED_WAITING 或
TERMINATED
```

下面分别对每一种状态进行解释。

2.1 新建线程

当用`new`操作符创建一个新线程时，这个线程还没有开始运行，它的状态是新建（new）。当一个线程处于新建状态时，程序还没有开始运行线程中的代码。在线程运行之前还有一些基础工作要做。

2.2 可运行线程

一旦调用`start`方法，线程就处于可运行（runnable）状态。一个可运行的线程可能正在运行也可能没有运行，要由操作系统为线程提供具体的运行时间。Java 规范没有将“正在运行”作为一个单独的状态，一个正在运行的线程仍然处于可运行状态。

一旦一个线程开始运行，它不一定始终保持运行。事实上，运行中的线程有时需要暂停，让其他线程有机会运行。线程调度的细节依赖于操作系统提供的服务。抢占式调度系统给每一个可运行线程一个时间片来执行任务，当时间片用完时，操作系统会剥夺该线程的运行权，并给另一个线程一个机会来运行。

现在所有的桌面以及服务器操作系统都使用抢占式调度。但是像手机这样的小型设备可能使用协作式调度。在这样的设备中，一个线程只有在调用 `yield` 方法或者被阻塞或等待时才失去控制权。

```
/* java.util.Thread */
static void yield() // 使当前正在执行的线程向另一个线程交出运行权
```

在有多个处理器的机器上，每一个处理器运行一个线程，可以有多个线程并行运行。如果线程的数目多于处理器的数目，调度器还是需要分配时间片。

2.3 阻塞和等待线程

当线程处于阻塞或等待状态时，它暂时是不活动的，不运行任何代码，而且消耗最少的资源。要由线程调度器重新激活这个线程，具体细节取决于它是怎样到达非活动状态的。

- 当一个线程试图获取一个内部的对象锁，而这个锁目前被其他线程占有，该线程就会被阻塞。当所有其他线程都释放了这个锁，并且线程调度器允许该线程持有这个锁时，它将变成非阻塞状态。
- 当线程等待另一个线程通知调度器出现一个条件时，这个线程会进入等待状态。调用 `Object.wait` 方法或 `Thread.join` 方法，或者是等待 `java.util.concurrent` 库中的 `Lock` 或 `Condition` 时，就会出现这种情况。
- 有几个方法有超时参数，调用这些方法会让线程进入计时等待状态。这一状态将一直保持到超时期满或者接收到适当的通知。带有超时参数的方法有 `Thread.sleep` 和计时版的 `Object.wait`、`Thread.join`、`Lock.tryLock` 以及 `Condition.await`。

```
/* java.lang.Thread */
void join()
    // 等待终止指定的线程
void join(long millis)
    // 等待指定的线程终止或者等待经过指定的毫秒数
static void sleep(long millis)
    // 休眠指定的毫秒数
```

当一个线程阻塞或等待时，可以调度另一个线程运行。当一个线程被重新激活，调度器检查它是否具有比当前运行线程更高的优先级，如果优先级更高，调度器会剥夺某个当前运行线程的运行权，选择一个新线程运行。

2.4 终止线程

线程会由于以下两个原因之一而终止：

- `run` 方法正常退出，线程自然终止。
- 因为一个没有捕获的异常终止了 `run` 方法，使线程意外终止。

3 线程属性

3.1 中断线程

`interrupt` 方法可以用来请求终止一个线程。当对一个线程调用 `interrupt` 方法时，就会设置线程的中断状态。这是每个线程都有的 `boolean` 标志，每个线程都应该不时地检查这个标志，以判断线程是否被中断。

```
/* java.lang.Thread */
void interrupt()
    // 向线程发送中断请求，将线程的中断状态设置为 true
    // 如果当前该线程被一个 sleep 调用阻塞，则抛出一个 InterruptedException 异常
```

要想得出是否设置了中断状态，首先调用静态的`Thread.currentThread`方法获得当前线程，然后调用`isInterrupted`方法。

```
/* java.lang.Thread */
static Thread currentThread()
    // 返回当前正在执行的线程的 Thread 对象
boolean isInterrupted()
    // 测试线程是否被中断
```

如果线程被阻塞，就无法检查中断状态。当在一个被`sleep`或`wait`调用阻塞的线程上调用`interrupt`方法时，那个阻塞调用将被一个`InterruptedException`异常中断。

中断一个线程只是要引起它的注意，被中断的线程可以决定如何响应中断。某些线程非常重要，所以应该处理这个异常，然后再继续执行。但是，更普遍的情况是，线程只希望将中断解释为一个终止请求，这种线程的`run`方法具有如下形式：

```
Runnable r = () ->
{
    try
    {
        ...
        while (!Thread.currentThread().isInterrupted() && ...)
        {
            ...
        }
    }
    catch (InterruptedException e)
    {
        ...
    }
    finally
    {
        ...
    }
};
```

如果在每次工作迭代之后都调用`sleep`方法（或者其他可中断方法），`isInterrupted`检查既没有必要也没有用处。如果设置了中断状态，此时倘若调用`sleep`方法，它不会休眠，而会清除中断状态并抛出`InterruptedException`异常。因此，如果在循环中调用了`sleep`方法，不要检测中断状态，而应当捕获`InterruptedException`异常，如下所示：

```
Runnable r = () ->
{
    try
    {
        ...
        while (...)
        {
            ...
```

```
            Thread.sleep(delay);
        }
    }
    catch (InterruptedException e)
    {
        ...
    }
    finally
    {
        ...
    }
};
```

不要抑制`InterruptedException`异常。例如，下面的做法是错误的：

```
void mySubTask()
{
    ...
    try { sleep(delay); }
    catch (InterruptedException e) {} // 错误
    ...
}
```

如果想不出在`catch`子句中可以做什么有意义的工作，仍然有两种合理的选择：

- 在`catch`子句中调用`Thread.currentThread().interrupt()`来设置中断状态，这样调用者就可以检测中断状态。

```
void mySubTask()
{
    ...
    try { sleep(delay); }
    catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    ...
}
```

- 或者，更好的选择是，用`throws InterruptedException`标记方法，去掉`try`语句块，这样调用者就可以捕获这个异常。

```
void mySubTask() throws InterruptedException
{
    ...
    sleep(delay);
    ...
}
```

有另一个检测中断状态的方法`interrupted`，它是一个静态方法，测试当前线程是否被中断，同时将当前线程的中断状态重置为`false`。

```
/* java.lang.Thread */
static boolean interrupted() // 测试当前线程是否被中断，同时将当前线程的中断状态重置为
false
```

3.2 守护线程

可以通过调用 `setDaemon` 方法将一个线程转换为守护线程（daemon thread）：

```
/* java.lang.Thread */
void setDaemon(boolean isDaemon) // 标识该线程为守护线程或用户线程。这一方法必须在线程启动之前调用
```

守护线程的唯一用途是为其他线程提供服务。例如，计时器线程定时发送“计时器嘀嗒”信号给其他线程，以及清空过时缓存项的线程，它们都是守护线程。当只剩下守护线程时，虚拟机就会退出。

3.3 线程名

默认情况下，线程有容易记的名字，如 Thread-2。可以用 `setName` 方法为线程设置任何名字：

```
/* java.lang.Thread */
void setName(String name)
```

3.4 未捕获异常的处理器

线程的 `run` 方法不能抛出任何检查型异常。但是，非检查型异常可能会导致线程终止，在这种情况下，线程会死亡。在线程死亡之前，异常会传递到一个用于处理未捕获异常的处理器。这个处理器必须属于一个实现了 `Thread.UncaughtExceptionHandler` 接口的类，这个接口只有一个方法：

```
/* java.lang.Thread.UncaughtExceptionHandler */
void uncaughtException(Thread t, Throwable e)
    // 当线程因一个未捕获异常而终止时，要记录一个定制报告
```

可以用 `setUncaughtExceptionHandler` 方法为任何线程安装一个处理器，也可以用 `Thread` 类的静态方法 `setDefaultUncaughtExceptionHandler` 为所有线程安装一个默认的处理器。

```
/* java.lang.Thread */
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)
    // 设置未捕获异常的处理器。如果没有安装处理器，则将线程组对象作为处理器
Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()
    // 获取未捕获异常的处理器
static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)
    // 设置未捕获异常的默认处理器
static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()
    // 获取未捕获异常的默认处理器
```

如果没有安装默认处理器，默认处理器则为 `null`。但是，如果没有为单个线程安装处理器，那么处理器就是该线程的 `ThreadGroup` 对象。线程组是可以一起管理的线程的集合。默认情况下，创建的所有线程都属于同一个线程组，但是也可以建立其他的组。

`ThreadGroup` 类实现了 `Thread.UncaughtExceptionHandler` 接口，它的 `uncaughtException` 方法执行以下操作：

1. 如果该线程组有父线程组，那么调用父线程组的 `uncaughtException` 方法。
2. 否则，如果 `Thread.getDefaultUncaughtExceptionHandler` 方法返回一个非 `null` 的处理器，则调用该处理器。
3. 否则，如果 `Throwable` 是 `ThreadDeath` 的一个实例，什么都不做。
4. 否则，将线程的名字以及 `Throwable` 的栈轨迹输出到 `System.err`。

3.5 线程优先级

在 Java 中，每个线程有一个优先级。默认情况下，一个线程会继承构造它的那个线程的优先级。可以用 `setPriority` 方法设置线程的优先级：

```
/* java.lang.Thread */
void setPriority(int newPriority) // 设置线程的优先级
```

`Thread` 类中定义了 3 个线程优先级字段：

```
/* java.lang.Thread */
static int MIN_PRIORITY // 最小优先级，值为 1
static int NORM_PRIORITY // 默认优先级，值为 5
static int MAX_PRIORITY // 最大优先级，值为 10
```

可以用 `setPriority` 方法将优先级设置为 `MIN_PRIORITY` 与 `MAX_PRIORITY` 之间的任何值。

每当线程调度器有机会选择新线程时，它首先选择具有较高优先级的线程。但是，线程优先级高度依赖于系统。当虚拟机依赖于宿主机平台的线程实现时，Java 线程的优先级会映射到宿主机平台的优先级。平台的线程优先级别可能比上述的 10 个级别多，也可能更少。例如，Windows 有 7 个优先级别。

在没有使用操作系统线程的 Java 早期版本中，线程优先级可能很有用。不过现在不要使用线程优先级了。

4 同步

在大多数实际的多线程应用中，两个或两个以上的线程需要共享对同一数据的存取。如果两个线程存取同一个对象，并且每个线程分别调用了一个修改该对象状态的方法，这两个线程就会相互覆盖，对象的最终状态取决于线程访问数据的次序，可能会导致对象被破坏。这种情况通常称为**竞态条件**。

4.1 竞态条件的一个例子

```
import java.util.*;

class Bank
{
    private final double[] accounts;

    /**
     * 构造Bank对象
     * @param n 账户数量
     * @param initialBalance 每个账户的初始余额
     */
    public Bank(int n, double initialBalance)
    {
        accounts = new double[n];
        Arrays.fill(accounts, initialBalance);
    }

    /**
     * 从一个账户向另一个账户转账
     * @param from 要转账的账户
     * @param to 要转移到的账户
     * @param amount 转账金额
     */
```

```
    public void transfer(int from, int to, double amount)
    {
        if (accounts[from] < amount) return;
        System.out.print(Thread.currentThread());
        accounts[from] -= amount;
        System.out.printf(" %10.2f from %d to %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
    }

    /**
     * 得到所有账户余额的总和
     * @return 总余额
     */
    public double getTotalBalance()
    {
        double sum = 0;
        for (double a : accounts)
        {
            sum += a;
        }
        return sum;
    }

    /**
     * 得到账户数量
     * @return 账户数量
     */
    public int size()
    {
        return accounts.length;
    }
}

public class UnsynchBankTest
{
    public static final int NACCOUNTS = 100;
    public static final double INITIAL_BALANCE = 1000;
    public static final double MAX_AMOUNT = 1000;
    public static final int DELAY = 10;

    public static void main(String[] args)
    {
        Bank bank = new Bank(NACCOUNTS, INITIAL_BALANCE);
        for (int i = 0; i < NACCOUNTS; i++)
        {
            int fromAccount = i;
            Runnable r = () ->
            {
                try
                {
                    while (true)
                    {
                        int toAccount = (int) (bank.size() * Math.random());
                        double amount = MAX_AMOUNT * Math.random();
                        bank.transfer(fromAccount, toAccount, amount);
                        Thread.sleep((int) (DELAY * Math.random()));
                    }
```

```
                }
                catch (InterruptedException e)
                {
                }
            };
            Thread t = new Thread(r);
            t.start();
        }
    }
}
```

4.2 竞态条件详解

上面的程序中有多个线程会更新银行账户余额，当两个线程试图同时更新同一个账户时，就会出现问题。例如下面这条语句：

```
accounts[to] += amount;
```

这条语句可能如下处理：

1. 将 `accounts[to]` 加载到寄存器。
2. 添加 `amount`。
3. 将结果写回 `accounts[to]`。

假定第1个线程执行步骤1和2，然后它的运行权被抢占。再假设第2个线程被唤醒，更新 `account` 数组中的同一个元素。然后，第1个线程被唤醒并完成第3步。这个动作会抹去第2个线程所做的更新，这样总金额就不再正确了。

真正的问题是 `transfer` 方法可能会在执行到中间时被中断。如果能够确保线程失去控制之前方法已经运行完成，那么银行账户对象的状态就不会被破坏。

4.3 锁对象

Java 5引入了 `ReentrantLock` 类，用它保护代码块的基本结构如下：

```
myLock.lock(); // myLock 是一个 ReentrantLock 对象
try
{
    // 临界区
}
finally
{
    myLock.unlock();
}
```

```
/* java.util.concurrent.locks.Lock */
void lock() // 获得这个锁。如果锁当前被另一个线程占有，则阻塞
void unlock() // 释放这个锁
```

临界区指的是一个访问共用资源的程序片段，这些共用资源无法同时被多个线程访问。上述结构确保任何时刻只有一个线程进入临界区。一旦一个线程锁定了锁对象，其他任何线程都无法通过 `lock` 方法。当其他线程调用 `lock` 时，它们会暂停，直到第一个线程释放这个锁对象。

要把 `unlock` 操作放在 `finally` 子句中，这一点至关重要。如果在临界区的代码抛出一个异常，锁必须释放，否则其他线程将永远阻塞。

使用锁时，不能使用 try-with-resources 语句。

下面使用一个锁来保护`Bank`类的`transfer`方法：

```
public class Bank
{
    private ReentrantLock bankLock = new ReentrantLock();
    ...
    public void transfer(int from, int to, double amount)
    {
        bankLock.lock();
        try
        {
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
        }
        finally
        {
            bankLock.unlock();
        }
    }
}
```

假设一个线程调用了`transfer`，但是在执行结束前被抢占。再假设第二个线程也调用了`transfer`，由于第二个线程不能获得锁，将在调用`lock`方法时被阻塞，它会暂停，必须等待第一个线程执行完`transfer`方法。当第一个线程释放锁时，第二个线程才能开始运行。

注意每个`Bank`对象都有自己的`ReentrantLock`对象。如果两个线程试图访问同一个`Bank`对象，那么锁可以用来保证串行化访问。不过，如果两个线程访问不同的`Bank`对象，每个线程会得到不同的锁对象，两个线程都不会阻塞。这是正常的，因为线程在操纵不同的`Bank`对象时，线程之间不会相互影响。

`ReentrantLock`称为重入（reentrant）锁，因为线程可以反复获得已拥有的锁。锁有一个持有计数来跟踪对`lock`方法的嵌套调用。线程每一次调用`lock`后都要调用`unlock`来释放锁，由于这个特性，被一个锁保护的代码可以调用另一个使用相同锁的代码。例如，`transfer`方法调用`getTotalBalance`方法，这也会封锁`bankLock`对象，此时`bankLock`对象的持有计数为2。当`getTotalBalance`方法退出时，持有计数变为1。当`transfer`方法退出的时候，持有计数变为0，线程释放锁。

要注意确保临界区中的代码不要因为抛出异常而跳出临界区。如果在临界区代码结束之前抛出了异常，`finally`子句将释放锁，但是对象可能处于被破坏的状态。

4.4 条件对象

通常，线程进入临界区后却发现只有满足了某个条件之后它才能执行。可以使用一个条件对象来管理那些已经获得了一个锁却不能做有用工作的线程。

在上面的银行示例程序中，如果一个账户没有足够的资金转账，我们不希望从这样的账户转出资金。注意不能使用类似下面的代码：

```
if (bank.getBalance(from) >= amount)
{
    bank.transfer(from, to, amount);
}
```

在通过条件测试之后、调用`transfer`方法之前，当前线程有可能被中断。在线程再次运行前，账户余额可能已经低于提款金额。必须确保在检查余额与转账活动之间没有其他线程修改余额，为此，可以用一个锁来保护这个测试和转账操作：

```
public void transfer(int from, int to, double amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // 等待
            ...
        }
        // 转账
        ...
    }
    finally
    {
        bankLock.unlock();
    }
}
```

当账户中没有足够的资金时，我们要等待，直到另一个线程向账户增加了资金。但是，这个线程刚刚获得了对`bankLock`的排他性访问权，因此别的线程没有存款的机会。这里就要引入条件对象。

一个锁对象可以有一个或多个相关联的条件对象，可以用`newCondition`方法获得一个条件对象：

```
/* java.util.concurrent.locks.Lock */
Condition newCondition() // 返回一个与这个锁相关联的条件对象
```

习惯上会给每个条件对象一个合适的名字来反映它表示的条件。例如，在这里我们建立了一个条件对象来表示"资金充足"条件：

```
class Bank
{
    private Condition sufficientFunds;
    ...
    public Bank()
    {
        ...
        sufficientFunds = bankLock.newCondition();
    }
}
```

如果`transfer`方法发现资金不足，应该调用`await`方法：

```
/* java.util.concurrent.locks.Condition */
void await() // 将该线程放在这个条件的等待集中
```

等待获得锁的线程和已经调用了`await`方法的线程存在本质上的不同。一旦一个线程调用了`await`方法，它就进入这个条件的等待集，当前线程暂停并放弃锁，这就允许另一个线程运行。当锁可用时，该线程并不会变为可运行状态，仍保持非活动状态，直到另一个线程在同一条件上调用`signalAll`方法。

当另一个线程完成转账时，它应该调用`signalAll`方法：

```
/* java.util.concurrent.locks.Condition */
void signalAll() // 解除该条件等待集中所有线程的阻塞状态
```

这个调用会重新激活等待这个条件的所有线程。当这些线程从等待集中移出时，它们再次成为可运行的线程，调度器最终将它们再次激活。同时，它们会尝试重新进入该对象。一旦锁可用，它们中的某个线程将从`await`调用返回，得到这个锁，并从之前暂停的地方继续执行。此时，线程应当再次测试条件，不能保证现在一定满足条件。因此，`await`调用应该放在如下形式的循环中：

```
while (accounts[from] < amount)
{
    sufficientFunds.await();
}
```

最终需要有某个其他线程调用`signalAll`方法，这一点至关重要。当一个线程调用`await`方法时，它没有办法重新自行激活，只能寄希望于其他线程。如果没有其他线程来重新激活等待的线程，它就永远不再运行了，这将导致死锁现象。如果所有其他线程都被阻塞，最后一个活动线程调用了`await`方法但没有先解除另外某个线程的阻塞，现在这个线程也会阻塞，此时没有线程可以解除其他线程的阻塞状态，程序会永远挂起。

从经验上讲，只要一个对象的状态有变化，而且可能有利于等待的线程，就可以调用`signalAll`方法。例如，当一个账户余额发生改变时，就应该再给等待的线程一个机会来检查余额。

利用条件对象，`transfer`方法可以改写如下：

```
public void transfer(int from, int to, double amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            sufficientFunds.await();
        }
        // 转账
        ...
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

注意`signalAll`调用不会立即激活一个等待的线程，它只是解除等待线程的阻塞，使这些线程可以在当前线程释放锁之后竞争访问对象。

另一个方法`signal`只是随机选择等待集中的一个线程，并解除这个线程的阻塞状态。这比解除所有线程的阻塞更高效，但也存在危险。如果随机选择的线程仍然不能运行，它就会再次阻塞，如果没有其他线程再次调用`signal`，系统就会进入死锁。

```
/* java.util.concurrent.locks.Condition */
void signal() // 从该条件的等待集中随机选择一个线程，解除其等待状态
```

只有当线程拥有一个条件的锁时，它才能在这个条件上调用`await`、`signalAll`或`signal`方法。

利用锁和条件对象对`Bank`类进行修改后，就不会出现错误了。

```
import java.util.*;
import java.util.concurrent.locks.*;

public class Bank
{
    private final double[] accounts;
    private Lock bankLock;
    private Condition sufficientFunds;

    /**
     * 构造Bank对象
     * @param n 账户数量
     * @param initialBalance 每个账户的初始余额
     */
    public Bank(int n, double initialBalance)
    {
        accounts = new double[n];
        Arrays.fill(accounts, initialBalance);
        bankLock = new ReentrantLock();
        sufficientFunds = bankLock.newCondition();
    }

    /**
     * 从一个账户向另一个账户转账
     * @param from 要转账的账户
     * @param to 要转移到的账户
     * @param amount 转账金额
     */
    public void transfer(int from, int to, double amount) throws
InterruptedException
    {
        bankLock.lock();
        try
        {
            while (accounts[from] < amount)
            {
                sufficientFunds.await();
            }
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
            sufficientFunds.signalAll();
        }
        finally
        {
            bankLock.unlock();
        }
    }

    /**
     * 得到所有账户余额的总和
```

```
     * @return 总余额
     */
    public double getTotalBalance()
    {
        bankLock.lock();
        try
        {
            double sum = 0;
            for (double a : accounts)
            {
                sum += a;
            }
            return sum;
        }
        finally
        {
            bankLock.unlock();
        }
    }

    /**
     * 得到账户数量
     * @return 账户数量
     */
    public int size()
    {
        return accounts.length;
    }
}
```

4.5 synchronized关键字

总结以下锁和条件的要点：

- 锁用来保护代码片段，一次只能有一个线程执行被保护的代码。
- 锁可以管理试图进入被保护代码段的线程。
- 一个锁可以有一个或多个相关联的条件对象。
- 每个条件对象管理那些已经进入被保护代码段但还不能运行的线程。

`Lock` 和 `Condition` 接口可以充分控制锁定。不过，大多数情况下，并不需要那样控制，完全可以使用 Java 语言内置的一种机制。从1.0版开始，Java 中的每个对象都有一个内部锁，如果一个方法声明时有 `synchronized` 关键字，那么对象的锁将保护整个方法。也就是说，要调用方法，线程必须获得内部对象锁。

下面两段代码是等价的：

```
public synchronized void method()
{
    // 方法体
}

public void method()
{
    this.intrinsicLock.lock(); // intrinsicLock 表示内部锁
    try
    {
        // 方法体
```

```
    }
    finally { this.intrinsicLock.unlock(); }
}
```

内部对象锁只有一个关联条件。与条件对象类似，内部对象锁的条件也有相对应的`wait`、`notifyAll`和`notify`方法：

```
/* java.lang.Object */
final void wait() throws InterruptedException
    // 导致一个线程进入等待状态，直到它得到通知。该方法只能在一个同步方法或同步块中调用
    // 如果当前线程不是对象锁的所有者，该方法会抛出一个 IllegalMonitorStateException 异常
final void wait(long millis) throws InterruptedException
final void wait(long millis, int nanos) throws InterruptedException
    // 导致一个线程进入等待状态，直到它得到通知或者经过了指定的时间。这些方法只能在一个同步方法或同步块中调用
    // 如果当前线程不是对象锁的所有者，这些方法会抛出一个 IllegalMonitorStateException 异常
    // 纳秒数不能超过 1000000
final void notifyAll()
    // 解除在这个对象上调用 wait 方法的那些线程的阻塞状态。该方法只能在一个同步方法或同步块中调用
    // 如果当前线程不是对象锁的所有者，该方法会抛出一个 IllegalMonitorStateException 异常
final void notify()
    // 随机选择一个在这个对象上调用 wait 方法的线程，解除其阻塞状态。该方法只能在一个同步方法或同步块中调用
    // 如果当前线程不是对象锁的所有者，该方法会抛出一个 IllegalMonitorStateException 异常
```

将静态方法声明为同步也是合法的。如果调用这样一个方法，它会获得相关类对象的内部锁。例如，如果`Bank`类有一个静态同步方法，那么当调用这个方法时，`Bank.class`对象的锁会锁定，此时其他线程不能调用这个类的任何同步静态方法。

内部锁和条件存在一些限制：

- 不能中断一个正在尝试获得锁的线程。
- 不能指定尝试获得锁时的超时时间。
- 每个锁仅有一个条件可能是不够的。

使用`synchronized`关键字改写`Bank`类如下：

```
import java.util.*;

public class Bank
{
    private final double[] accounts;

    /**
     * 构造Bank对象
     * @param n 账户数量
     * @param initialBalance 每个账户的初始余额
     */
    public Bank(int n, double initialBalance)
    {
        accounts = new double[n];
        Arrays.fill(accounts, initialBalance);
    }
```

```
    /**
     * 从一个账户向另一个账户转账
     * @param from 要转账的账户
     * @param to 要转移到的账户
     * @param amount 转账金额
     */
    public synchronized void transfer(int from, int to, double amount)
        throws InterruptedException
    {
        while (accounts[from] < amount)
        {
            wait();
        }
        System.out.print(Thread.currentThread());
        accounts[from] -= amount;
        System.out.printf(" %10.2f from %d to %d", amount, from, to);
        accounts[to] += amount;
        System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
        notifyAll();
    }

    /**
     * 得到所有账户余额的总和
     * @return 总余额
     */
    public synchronized double getTotalBalance()
    {
        ...
    }

    /**
     * 得到账户数量
     * @return 账户数量
     */
    public int size()
    {
        return accounts.length;
    }
}
```

在代码中应该使用`Lock/Condition`还是`synchronized`关键字？下面是建议策略：

- 最好既不使用`Lock/Condition`也不使用`synchronized`关键字。在许多情况下，可以使用`java.util.concurrent`包中的某种机制，它会处理所有的锁定。
- 如果`synchronized`关键字适合你的程序，那么尽量使用这种做法，这样可以减少代码量，还能减少出错的概率。
- 如果特别需要`Lock/Condition`结构提供的额外能力，则使用`Lock/Condition`。

4.6 同步块

同步块的形式如下：

```
synchronized (obj)
{
    // 临界区
}
```

当线程进入同步块时，它会获得`obj`的锁。

有时我们会发现一些“专用”锁，例如：

```
public class Bank
{
    private double[] accounts;
    private Object lock = new Object();
    ...
    public void transfer(int from, int to, double amount)
    {
        synchronized (lock)
        {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
        ...
    }
}
```

在这里，创建`lock`对象只是为了使用这个对象拥有的锁。

有时程序员使用一个对象的锁来实现额外的原子操作，这种做法称为**客户端锁定**。例如，考虑`Vector`类，它的方法是同步的。假设将银行余额存储在一个`Vector<Double>`中，下面是`transfer`方法的一个原生实现：

```
public void transfer(Vector<Double> accounts, int from, int to, double amount)
//错误
{
    accounts.set(from, accounts.get(from) - amount);
    accounts.set(to, accounts.get(to) + amount);
    System.out.println(...);
}
```

`Vector`类的`get`和`set`方法是同步的，但是，这对于我们并没有帮助。在第一次`get`调用完成之后，一个线程完全可能在`transfer`方法中被抢占，然后另一个线程可能会在相同的位置存储不同的值。解决办法是截获`Vector`对象的锁：

```
public void transfer(Vector<Double> accounts, int from, int to, double amount)
{
    synchronized (accounts)
    {
        accounts.set(from, accounts.get(from) - amount);
        accounts.set(to, accounts.get(to) + amount);
    }
    System.out.println(...);
}
```

这个方法是可行的，但是完全依赖于这样一个事实：`Vector`类会对自己的所有更改方法使用内部锁。不过，`Vector`类的文档没有给出这样的承诺，你必须仔细研究源代码，而且还要希望将来的版本不会引入非同步的更改方法。可以看到，客户端锁定是非常脆弱的，通常不推荐使用。

4.7 监视器概念（管程）

锁和条件是实现线程同步的强大工具，但是它们不是面向对象的。研究人员努力寻找方法，希望不要求程序员考虑显式锁就可以保证多线程的安全性，最成功的解决方案之一是**监视器**（monitor）。20世纪70年代，Per Brinch Hansen和Tony Hoare最早提出这一概念 。用 Java 的术语来讲，监视器具有如下特性：

- 监视器是只包含私有字段的类。
- 监视器类的每个对象有一个关联的锁。
- 所有方法由这个锁锁定。换句话说，如果客户端调用`obj.method()`，那么`obj`对象的锁在方法调用开始时自动获得，并且当方法返回时自动释放。因为所有的字段是私有的，这样的安排可以确保一个线程处理字段时，没有其他线程能够访问这些字段。
- 锁可以有任意多个相关联的条件。

Java 设计者以不太严格的方式采用了监视器概念。Java 中的每个对象都有一个内部锁和一个内部条件。如果一个方法用`synchronized`关键字声明，那么它表现得就像是一个监视器方法，可以通过`wait`、`notifyAll`、`notify`方法来访问条件变量。不过，Java 对象在以下3个方面不同于监视器，这削弱了线程的安全性：

- 字段不要求是`private`。
- 方法不要求是`synchronized`。
- 内部锁对客户是可用的。

4.8 volatile字段

`volatile`关键字为实例字段的同步访问提供了一种免锁机制。如果声明一个字段为`volatile`，那么编译器和虚拟机就知道该字段可能被另一个线程并发更新。

例如，假设一个对象有一个`boolean`标记`done`，它的值由一个线程设置，而由另一个线程查询，这时可以使用锁：

```
private boolean done;
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
```

或许使用内部对象锁不是一个好主意。如果另一个线程已经对该对象加锁，`isDone`和`setDone`方法可能会阻塞。如果是这个问题，可以只为这个变量使用一个单独的锁，但是这会很麻烦。在这种情况下，将字段声明为`volatile`就很合适：

```
private volatile boolean done;
public boolean isDone() { return done; }
public void setDone() { done = true; }
```

编译器会插入适当的代码，以确保如果一个线程对`done`变量做了修改，这个修改对读取这个变量的所有其他线程都可见。

`volatile`变量不能提供原子性，不能保证读取、翻转和写入不被中断。例如，下面的方法不能确保翻转字段中的值：

```
public void flipDone() { done = !done; } // 没有原子性
```

4.9 final字段

前面已经了解到，除非使用锁或`volatile`修饰符，否则无法从多个线程安全地读取一个字段。除此之外，还有一种情况可以安全地访问一个共享字段，即这个字段为`final`时，因为`final`字段只能读取不能修改。例如，考虑以下声明：

```
final HashMap<String, Double> accounts = new HashMap<String, Double>();
```

其他线程会在构造器完成构造之后才看到这个变量。如果不用`final`，就不能保证其他线程看到的是`accounts`更新后的值。

4.10 原子性

假设对共享变量除了赋值之外并不做其他操作，那么可以将这些共享变量声明为`volatile`。

`java.util.concurrent.atomic`包中有很多类使用了很高效的机器级指令来保证其他操作的原子性。例如，`AtomicInteger`类提供了方法`incrementAndGet`和`decrementAndGet`，它们分别以原子方式将一个整数进行自增或自减，并返回更改后的值。也就是说，获得值、修改值并设置然后生成新值的操作不会中断，可以保证即使是多个线程并发地访问同一个实例，也会计算并返回正确的值。

```
/* java.util.concurrent.atomic.AtomicInteger */
AtomicInteger()
    // 创建一个原子化整数，初值为 0
AtomicInteger(int initialValue)
    // 用指定的初值创建一个原子化整数
int incrementAndGet()
    // 以原子方式将当前值加 1
int decrementAndGet()
    // 以原子方式将当前值减 1
boolean compareAndSet(int expect, int update)
    // 如果当前值等于 expect，就将当前值改为 update
int updateAndGet(IntUnaryOperator updateFunction)
    // 用给定函数的计算结果原子地更新当前值，并返回更新后的值
int getAndUpdate(IntUnaryOperator updateFunction)
    // 用给定函数的计算结果原子地更新当前值，并返回更新前的值
int accumulateAndGet(int x, IntBinaryOperator updateFunction)
    // 对当前值和第一个参数应用二元操作，用这一操作的结果更新当前值，并返回更新后的值
int getAndAccumulate(int x, IntBinaryOperator updateFunction)
    // 对当前值和第一个参数应用二元操作，用这一操作的结果更新当前值，并返回更新前的值
```

提供原子操作的类还有`AtomicIntegerArray`、`AtomicIntegerFieldUpdater`、`AtomicLong`、`AtomicLongArray`、`AtomicLongFieldUpdater`、`AtomicReference`、`AtomicReferenceArray`、`AtomicReferenceFieldUpdater`等。

如果有大量线程要访问相同的原子值，性能会大幅下降。`LongAdder`和`LongAccumulator`类解决了这个问题。`LongAdder`类包括多个变量（加数），其总和为当前值。可以有多个线程更新不同的加数，线程个数增加时会自动提供新的加数。通常情况下，只有当所有工作都完成之后才需要总和的值，对于这种情况，这种方法会很高效，性能会有显著的提升。`LongAdder`类的重要方法如下：

```
/* java.util.concurrent.atomic.LongAdder */
void increment() // 自增 1
void decrement() // 自减 1
void add(long x) // 增加一个给定值
long sum() // 返回总和
```

`LongAccumulator`类将这种思想推广到任意的累加操作。在构造器中，可以提供这个操作以及它的零元素：

```
/* java.util.concurrent.atomic.LongAccumulator */
LongAccumulator(LongBinaryOperator accumulatorFunction, long identity)
    // 用给定的累加操作和零元素构造一个实例
```

一般来说，这个操作必须满足结合律和交换律。这说明，最终结果必须不依赖于以什么顺序结合这些中间值。

在内部，这个累加器包含变量 a_1，a_2，...，a_n，每个变量初始化为零元素。调用`accumulate`方法并提供值 v 时，其中一个变量会以原子方式更新为 $a_i = a_i \ op \ v$，这里 op 是中缀形式的累加操作。`get`方法的结果是 $a_1 \ op \ a_2 \ op \ ... \ op \ a_n$。

```
/* java.util.concurrent.atomic.LongAccumulator */
void accumulate(long x)
    // 用给定的值更新累加器
long get()
    // 返回累加器的当前值
    // 在没有并发更新的情况下调用会返回正确结果，但是在调用时发生的并发更新可能不会被合并
```

类似地，`DoubleAdder`和`DoubleAccumulator`也采用同样的方式，只不过处理的是`double`值。

4.11 死锁

考虑下面的情况：

1. 账户1：200元
2. 账户2：300元
3. 线程1：从账户1转300元到账户2
4. 线程2：从账户2转400元到账户1

线程1和线程2都被阻塞，因为账户1和账户2中的余额都不足以进行转账，两个线程都无法执行下去。

有可能会因为每一个线程要等待更多的钱款存入而导致所有线程都被阻塞，这样的状态称为死锁。

还有一种做法会导致死锁。让第 i 个线程负责向第 i 个账户存钱，而不是从第 i 个账户取钱，这样一来，有可能所有线程都集中到一个账户上，每一个线程都试图从这个账户中取出大于该账户余额的钱。

还有一种很容易导致死锁的情况：将`Bank`类中的`signalAll`方法换成`signal`方法。`signalAll`方法会通知所有等待增加资金的线程，而`signal`方法只解除一个线程的阻塞。如果该线程不能继续运行，所有的线程都会阻塞。

Java 编程语言中没有任何东西可以避免或打破这种死锁，必须仔细设计程序，确保不会出现死锁。

4.12 线程局部变量

有时可能要避免共享变量，使用`ThreadLocal`辅助类为各个线程提供各自的实例。`withInitial`方法用于构造线程局部变量：

```
/* java.lang.ThreadLocal<T> */
static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier)
    // 创建一个线程局部变量，其初始值通过调用给定的 supplier 生成
```

例如，`SimpleDateFormat`类不是线程安全的，要为每个线程构造一个实例，可以使用以下代码：

```
public static final ThreadLocal<SimpleDateFormat> dateFormat
    = ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyy-MM-dd"));
```

其他重要方法如下：

```
/* java.lang.ThreadLocal<T> */
T get()
    // 返回此线程局部变量在当前线程中的值。如果是首次调用 get 方法，会首先调用 initialValue 方法进行初始化
void set(T t)
    // 为当前线程设置一个新值
void remove()
    // 删除对应当前线程的值
protected T initialValue()
    // 返回当前线程变量的初始值。在线程中第一次调用 get 方法时会调用此方法
    // 如果在调用 get 之前调用了 set 方法，就不会调用这个方法
    // 如果在调用 remove 之后接着 get 方法，也会调用这个方法
    // 默认的实现会返回 null。如果想要得到非 null 的初始值，就要继承 ThreadLocal 类，并在子类中重写这个方法。通常使用匿名内部类
```

在多个线程中生成随机数也存在类似的问题。`java.util.Random`类是线程安全的，但是如果多个线程需要等待一个共享的随机数生成器，这会很低效。Java 7 提供了一个`ThreadLocalRandom`类，其中的静态方法`current`会返回特定于当前线程的`ThreadLocalRandom`实例：

```
/* java.util.concurrent.ThreadLocalRandom */
static ThreadLocalRandom current()
```

5 线程安全的集合

5.1 阻塞队列

很多线程问题可以使用一个或多个队列以优雅而安全的方式来描述。生产者线程向队列插入元素，消费者线程获取元素。使用队列，可以安全地从一个线程向另一个线程传递数据。

在协调多个线程之间的合作时，阻塞队列是一个有用的工具。当试图向队列添加元素而队列已满，或者从队列移出元素而队列为空的时候，阻塞队列将导致线程阻塞。工作线程可以周期性地将中间结果存储在阻塞队列中，其他工作线程移除中间结果并进一步进行修改。队列会自动平衡负载，如果第一组线程运行得比第二组满，第二组在等待结果时会阻塞；如果第一组线程运行得更快，队列会填满，直到第二组赶上来。

下面给出了`BlockingQueue`接口中的重要方法：

```
/* java.util.concurrent.BlockingQueue<E> */
boolean add(E e)
    // 向队列添加一个元素。如果队列满，则抛出 IllegalStateException 异常
E element()
    // 返回队头元素，但不删除。如果队列为空，则抛出 NoSuchElementException 异常
E remove()
    // 删除并返回队头元素。如果队列为空，则抛出 NoSuchElementException 异常

boolean offer(E e)
    // 向队列添加一个元素。如果添加成功，则返回 true；如果队列满，则返回 false
boolean offer(E e, long timeout, Timeunit unit)
    // 向队列添加一个元素。如果超时，则返回 false
    // timeout 指定时间限制，unit 指定时间单位
E peek()
    // 返回队头元素，但不删除。如果队列为空，则返回 null
```

```
E poll()
    // 删除并返回队头元素。如果队列为空，则返回 null
E poll(long timeout, Timeunit unit)
    // 删除并返回队头元素。如果超时，则返回 null

void put(E e)
    // 向队尾添加一个元素。如果队列满，则阻塞
E take()
    // 删除并返回队头元素。如果队列为空，则阻塞
```

注意，`poll`和`peek`方法返回`null`来指示失败，因此向阻塞队列中插入`null`值是非法的。

阻塞队列方法分成3类，这取决于当队列满或空时它们完成的动作。如果使用阻塞队列作为线程管理工具，将要用到`put`和`take`方法，它们可以使线程阻塞；当试图向满队列添加元素或者从空队列得到队头元素时，`add`、`remove`、`element`方法会抛出异常，而`offer`、`poll`、`peek`方法返回特殊值作为错误提示。

`java.util.concurrent`包提供了阻塞队列的几个变体。`LinkedBlockingQueue`用链表实现，实现了`BlockingQueue`接口，在默认情况下的容量没有上界，也可以选择指定一个最大容量。

```
/* java.util.concurrent.LinkedBlockingQueue<E> */
LinkedBlockingQueue()
    // 构造一个容量无上限的阻塞队列
LinkedBlockingQueue(int capacity)
    // 构造一个有指定容量的阻塞队列
```

`LinkedBlockingDeque`是一个双端队列，用链表实现。

```
/* java.util.concurrent.LinkedBlockingDeque<E> */
LinkedBlockingDeque()
    // 构造一个容量无上限的阻塞双端队列
LinkedBlockingDeque(int capacity)
    // 构造一个有指定容量的阻塞双端队列
void putFirst(E element)
    // 向队头添加元素。如果队列满则阻塞
void putLast(E element)
    // 向队尾添加元素。如果队列满则阻塞
E takeFirst()
    // 移除并返回队头元素。如果队列为空则阻塞
E takeLast()
    // 移除并返回队尾元素。如果队列为空则阻塞
boolean offerFirst(E element, long time, TimeUnit unit)
    // 向队头添加一个元素。如果超时，则返回 false
boolean offerLast(E element, long time, TimeUnit unit)
    // 向队尾添加一个元素。如果超时，则返回 false
E pollFirst(long time, TimeUnit unit)
    // 移除并返回队头元素，必要时阻塞，直至元素可用或超时。失败时返回 null
E pollLast(long time, TimeUnit unit)
    // 移除并返回队尾元素，必要时阻塞，直至元素可用或超时。失败时返回 null
```

`ArrayBlockingQueue`用循环数组实现，在构造时需要指定容量，并且有一个可选的参数来指定是否需要公平性。如果设置了公平参数，那么等待时间最长的线程会优先得到处理。通常公平性会降低性能，只有在确实非常需要时才使用公平参数。

```
/* java.util.concurrent.ArrayBlockingQueue<E> */
ArrayBlockingQueue(int capacity)
ArrayBlockingQueue(int capacity, boolean fair)
    // 构造一个有指定容量和公平性设置的阻塞队列
```

`PriorityBlockingQueue`是一个优先队列，用堆实现，元素按照它们的优先级顺序移除。这个队列没有容量上限。如果队列是空的，获取元素的操作会阻塞。

```
/* java.util.concurrent.PriorityBlockingQueue<E> */
PriorityBlockingQueue()
PriorityBlockingQueue(int initialCapacity)
PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)
    // 构造一个无上限阻塞优先队列，默认初始容量为 11
    // 如果没有指定比较器，则元素必须实现Comparable接口
```

`DelayQueue`是一个没有容量上限的阻塞队列，它的元素必须实现`Delayed`接口。

```
interface Delayed extends Comparable<Delayed>
{
    long getDelay(TimeUnit unit); // 得到对象的剩余延迟，用给定的时间单位度量。负值表示延迟已经结束
}
```

`Delayed`接口中还有一个从`Comparable<Delayed>`接口继承而来的`compareTo`方法。`DelayQueue`使用`compareTo`方法对元素进行排序。只有延迟结束的元素才能从`DelayQueue`中移除。

```
/* java.util.concurrent.DelayQueue<E extends Delayed> */
DelayQueue() // 构造一个包含Delayed元素的无上限阻塞队列
```

Java 7 增加了一个`TransferQueue`接口，允许生产者线程等待，直到消费者准备就绪可以接收元素。

```
/* java.util.concurrent.TransferQueue<E> */
void transfer(E element)
    // 传输一个值，这个调用将阻塞，直到另一个线程将元素删除
boolean tryTransfer(E element, long time, TimeUnit unit)
    // 尝试在给定的超时时间内传输元素，这个调用将阻塞，直到另一个线程将元素删除。调用成功时返回true
```

`LinkedTransferQueue`实现了`TransferQueue`接口。

下面的示例程序展示了如何使用阻塞队列来控制一组线程。程序在一个目录及其所有子目录下搜索所有文件，打印出包含指定关键字的行。

```
import java.io.*;
import java.nio.charset.*;
import java.nio.file.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class BlockingQueueTest
{
    private static final int FILE_QUEUE_SIZE = 10;
```

```
    private static final int SEARCH_THREADS = 100;
    private static final Path DUMMY = Path.of("");
    private static BlockingQueue<Path> queue = new ArrayBlockQueue<>
(FILE_QUEUE_SIZE);

    public static void main(String[] args)
    {
        try(Scanner in = new Scanner(System.in))
        {
            System.out.print("Enter base directory (e.g. /opt/jdk-9-src): ");
            String directory = in.nextLine();
            System.out.print("Enter keyword (e.g. volatile): ");
            String keyword = in.nextLine();

            // 生产者线程，枚举所有子目录下的所有文件，并把它们放到阻塞队列中
            Runnable enumerator = () ->
            {
                try
                {
                    enumerate(Path.of(directory));
                    queue.put(DUMMY);
                }
                catch (IOException e)
                {
                    e.printStackTrace();
                }
                catch (InterruptedException e)
                {}
            };
            new Thread(enumerator).strat();

            // 消费者线程，从阻塞队列中取出一个文件，查找关键字
            for (int i = 1; i <= SEARCH_THREADS; i++)
            {
                Runnable searcher = () ->
                {
                    try
                    {
                        boolean done = false;
                        while (!done)
                        {
                            Path file = queue.take();
                            if (file == DUMMY)
                            {
                                queue.put(file);
                                done = true;
                            }
                            else
                            {
                                search(file, keyword);
                            }
                        }
                    }
                    catch (IOException e)
                    {
                        e.printStackTrace();
                    }
                    catch (InterruptedException e)
```

```
                    {}
                };
                new Thread(searcher).start();
            }
        }
    }

    /**
     * 递归地枚举给定目录下的所有文件和子目录，并把它们放到一个阻塞队列中
     */
    public static void enumerate(Path directory) throws IOException,
InterruptedException
    {
        try (Stream<Path> children = Files.list(directory))
        {
            for (Path child : children.collect(Collectors.toList()))
            {
                if (Files.isDirectory(child))
                {
                    enumerate(child);
                }
                else
                {
                    queue.put(child);
                }
            }
        }
    }

    /**
     * 在文件中查询给定的关键词，输出包含指定关键字的行
     */
    public static void search(Path file, String keyword) throws IOException
    {
        try (Scanner in = new Scanner(file, StandardCharsets.UTF_8))
        {
            int lineNumber = 0;
            while (in.hasNextLine())
            {
                lineNumber++;
                String line = in.nextLine();
                if (line.contains(keyword))
                {
                    System.out.printf("%s:%d:%s\n", file, lineNumber, line);
                }
            }
        }
    }
}
```

5.2 高效的映射、集和队列

`java.util.concurrent`包提供了映射、有序集和队列的高效实现：`ConcurrentHashMap`、`ConcurrentSkipListMap`、`ConcurrentSkipListSet`、`ConcurrentLinkedQueue`。

这些集合使用复杂的算法，通过允许并发地访问数据结构的不同部分尽可能减少竞争。

与大多数集合不同，这些类的`size`方法不一定在常量时间内完成操作。确定这些集合的当前大小通常需要遍历。

这些集合返回弱一致性的迭代器，这意味着迭代器不一定能反映出它们构造之后的所有更改，但是，它们不会将同一个值返回两次，也不会抛出`ConcurrentModificationException`异常。与之形成对照的是，对于`java.util`包中的集合，如果集合在迭代器构造之后发生改变，集合的迭代器将抛出一个`ConcurrentModificationException`异常。

并发散列映射`ConcurrentHashMap`可以高效地支持大量阅读器和一定数量的书写器。默认情况下认为可以有至多16个同时运行的书写器线程，同一时间如果多于16个，其他线程将暂时阻塞。

```
/* java.util.concurrent.ConcurrentLinkedQueue<E> */
ConcurrentLinkedQueue<E>()
    // 构造一个可以被多线程安全访问的无上限非阻塞的队列

/* java.util.concurrent.ConcurrentSkipListSet<E> */
ConcurrentSkipListSet<E>()
    // 构造一个可以被多线程安全访问的有序集，要求元素实现 Comparable 接口
ConcurrentSkipListSet<E>(Comparator<? super E> comp)
    // 构造一个可以被多线程安全访问的有序集，排序方式由参数指定

/* java.util.concurrent.ConcurrentHashMap<K, V> */
ConcurrentHashMap<K, V>()
ConcurrentHashMap<K, V>(int initialCapacity)
ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int
concurrencyLevel)
    // 构造一个可以被多线程安全访问的散列映射表。默认的初始容量为 16
    // 装载因子的默认值为 0.75。如果每个桶的平均负载超过装载因子，表的大小会重新调整
    // 并发级别是估计的并发书写器的线程数

/* java.util.concurrent.ConcurrentSkipListMap<K, V> */
ConcurrentSkipListMap<K, V>()
    // 构造一个可以被多线程安全访问的有序映像，要求键实现 Comparable 接口
ConcurrentSkipListMap<K, V>(Comparator<? super K> comp)
    // 构造一个可以被多线程安全访问的有序映像，排序方式由参数指定
```

5.3 映射条目的原子更新

`ConcurrentHashMap`只能保证数据结构不被多线程访问破坏，但是不能保证操作的原子性。例如，假设要统计单词的频率，使用`ConcurrentHashMap<String, Long>`，下面的使计数值自增的代码不是线程安全的：

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // ERROR--可能出错
```

在老版本的 Java 中，要实现原子更新，必须使用`replace`方法：

```
/* java.util.concurrent.ConcurrentHashMap<K, V> */
boolean replace(K key, V oldValue, V newValue)
    // 当 key 对应的值等于 oldValue 时，以原子方式将值更新为 newValue ，并返回 true；否则
返回 false
```

使用`replace`方法对上述例子进行原子更新的代码如下：

```
do
{
    Long oldValue = map.get(word);
    Long newValue = oldValue == null ? 1 : oldValue + 1;
}
while (!map.replace(word, oldValue, newValue));
```

第二种方法是使用`ConcurrentHashMap<String, AtomicLong>`，原子更新代码如下：

```
map.putIfAbsent(word, new AtomicLong()); // 每次自增都要构造一个 AtomicLong，冗余
map.get(word).incrementAndGet();
```

如今，Java API 提供了一些新方法，可以更方便地完成原子更新。

```
/* java.util.concurrent.ConcurrentHashMap<K, V> */
V compute(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction)
    // 接收键和相关联的值，用给定的二元函数计算新值。如果没有相关联的值，则使用 null 进行计算
V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)
    // 如果键没有相关联的值，就用给定的函数计算值，如果计算出的值非 null，就将其加入映射中
V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction)
    // 如果键有相关联的值，就用给定的二元函数计算新值，并将旧值修改为新值
V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
remappingFunction)
    // 如果键没有相关联的值，就将值设置为给定的初始值；否则，对初始值和当前值应用给定的二元函数
计算新值
```

例如，用`compute`方法实现上述计数自增的代码如下：

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```

下面的示例程序使用并发散列映射来统计一个目录树的 Java 文件中的所有单词及其数目。

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class CHMDemo
{
    public static ConccurentHashMap<String, Long> map = new ConcurrentHashMap<>
();

    /**
     * 将给定文件中的所有单词添加到并发散列映射中
     */
    public static void process(Path file)
    {
        try (Scanner in = new Scanner(file))
        {
            while (in.hasNext())
            {
```

```
                String word = in.next();
                map.merge(word, 1L, Long::sum);
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }

    /**
     * 返回给定目录的所有后代
     * @param rootDir 根目录
     * @return 根目录的所有后代组成的集
     */
    public static Set<Path> descendants(Path rootDir) throws IOException
    {
        try (Stream<Path> entries = Files.walk(rootDir))
        {
            return entries.collect(Collectors.toSet());
        }
    }

    public static void main(String[] args)
        throws InterruptedException, ExecutionException, IOException
    {
        int processors = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(processors);
        Path pathToRoot = Path.of(".");
        for (Path p : descendants(pathToRoot))
        {
            if (p.getFileName().toString().endsWith(".java"))
            {
                executor.execute(() -> process(p));
            }
        }
        executor.shutdown();
        executor.awaitTermination(10, TimeUnit.MINUTES);
        map.forEach((k, v) ->
            {
                if (v >= 10)
                    System.out.println(k + " occurs " + v + " times");
            });
    }
}
```

5.4 对并发散列映射的批操作

Java API 为并发散列映射提供了批操作，即使有其他线程在处理映射，这些操作也能安全地执行。批操作会遍历映射，处理遍历过程中找到的元素。这里不会冻结映射的当前快照。

有3种不同的操作：

- `search`（搜索）：为每个键或值应用一个函数，直到函数生成一个非`null`的结果，此时搜索终止，返回这个函数的结果。
- `reduce`（规约）：使用一个累加函数组合所有键或值。
- `forEach`：为所有键或值应用一个函数。

每个操作都有4个版本：

- *opeartion*Keys：处理键。
- *opeartion*Values：处理值。
- *opeartion*：处理键和值。
- *opeartion*Entries：处理`Map.Entry`对象。

对于上述各操作，需要指定一个参数化阈值，如果映射包含的元素多于这个阈值，就会并行完成批操作。如果希望批操作在一个线程中运行，可以使用阈值`Long.MAX_VALUE`；如果希望用尽可能多的线程运行批操作，可以使用阈值1。

`search`方法有以下版本：

```
/* java.util.concurrent.ConcurrentHashMap<K, V> */
<U> U searchKeys(long threshold, Function<? super K, ? extends U>
searchFunction)
<U> U searchValues(long threshold, Function<? super V, ? extends U>
searchFunction)
<U> U search(long threshold, BiFunction<? super K, ? super V, ? extends U>
searchFunction)
<U> U searchEntries(long threshold, Function<Map.Entry<K, V>, ? extends U>
searchFunction)
```

例如，要找出第一个出现次数超过1000的单词，代码如下：

```
String result = map.search(threshold, (k, v) -> v > 1000 ? k : null);
```

`forEach`方法有两种形式。第一种形式只对各个映射条目应用一个消费者函数，例如：

```
map.forEach(threshold, (k, v) -> System.out.println(k + " -> " + v));
```

第二种形式还有一个额外的转换器函数作为参数，要先应用这个转换器函数，其结果会传递到消费者函数，例如：

```
map.forEach(threshold,
    (k, v) -> k + " -> " + v, // 转换器函数
    System.out::println); // 消费者函数
```

转换器可以用作过滤器。只要转换器返回`null`，这个值就会被跳过。例如：

```
map.forEach(threshold,
    (k, v) -> v > 1000 ? k + " -> " + v : null, // 小于 1000 的值被过滤掉
    System.out::println); // 消费者函数
```

`reduce`方法用一个累加函数组合其输入。例如，可以如下计算所有值的总和：

```
map.reduceValues(threshold, Long::sum);
```

与`forEach`类似，也可以向`reduce`方法提供转换器函数，这个转换器函数也可以作为过滤器。例如：

```
map.reduceValues(threshold,
    v -> v > 1000 ? 1L : null, // 小于 1000 的值被过滤掉
    Long::sum); // 累加函数
```

如果映射值为空，或者所有条目都被过滤掉，`reduce`方法会返回`null`。如果只有一个元素，则返回其转换结果，不会应用累加器。

`reduce`方法对于`int`、`long`、`double`类型还有相应的特殊化操作，分别有后缀`ToInt`、`ToLong`和`ToDouble`。需要把输入转换为一个基本类型值，并指定一个默认值和一个累加器函数。映射为空时返回默认值。例如：

```
map.reduceValuesToLong(threshold,
    Long::longValue, // 转换为基本类型
    0, // 默认值
    Long::sum); // 基本类型累加器
```

注意，特殊化版本的`reduce`操作在累加结束后，还要与默认值累加。因此，默认值必须是累加器的零元素。

5.5 并发集视图

静态`ConcurrentHashMap.newKeySet`方法会生成一个键集`Set<K>`。这实际上是`ConcurrentHashMap<K, Boolean>`的一个包装器，所有映射值都为`Boolean.TRUE`，不过因为只是要把它用作一个集，所以并不关心映射值。

```
/* java.util.concurrent.ConcurrentHashMap<K, V> */
static <K> ConcurrentHashMap.KeySetView<K, Boolean> newKeySet()
static <K> ConcurrentHashMap.KeySetView<K, Boolean> newKeySet(int
initialCapacity)
```

`keySet`方法可以生成特定映射的键集，这个集是可更改的。如果删除集的元素，键以及相应的值也会从映射中删除。向键集增加元素没有意义，因为没有相应的值可以增加。

```
/* java.util.concurrent.ConcurrentHashMap<K, V> */
ConcurrentHashMap.KeySetView<K, V> keySet()
ConcurrentHashMap.KeySetView<K, V> keySet(V mappedValue) // 向键集增加元素时，用参数
指定的值作为新键的映射值
```

5.6 写数组的拷贝

`CopyOnWriteArrayList`和`CopyOnWriteArraySet`是线程安全的集合，其中所有更改器会建立底层数组的一个副本。如果迭代访问集合的线程数超过更改集合的线程数，这样的安排是很有用的。当构造一个迭代器的时候，它包含当前数组的一个引用。如果这个数组后来被更改了，迭代器仍然引用旧数组，但是集合的数组已经替换，所以原来的迭代器可以访问一致的（但可能过时的）视图，而且不存在任何同步开销。

5.7 并行数组算法

`Arrays`类提供了大量并行化操作。`Arrays.parallelSort`静态方法可以对一个基本类型值或对象的数组排序。

```
/* java.util.Arrays */
static void parallelSort(byte[] a)
static void parallelSort(char[] a)
static void parallelSort(double[] a)
static void parallelSort(float[] a)
static void parallelSort(int[] a)
static void parallelSort(long[] a)
static void parallelSort(short[] a)
static void parallelSort(byte[] a)
static <T extends Comparable<? super T>> void parallelSort(T[] a)
```

对对象排序时，可以提供一个`Comparator`：

```
/* java.util.Arrays */
static <T> void parallelSort(T[] a, Comparator<? super T> cmp)
```

对于所有方法都可以提供一个范围的边界：

```
/* java.util.Arrays */
static void parallelSort(byte[] a, int fromIndex, int toIndex)
static void parallelSort(char[] a, int fromIndex, int toIndex)
static void parallelSort(double[] a, int fromIndex, int toIndex)
static void parallelSort(float[] a, int fromIndex, int toIndex)
static void parallelSort(int[] a, int fromIndex, int toIndex)
static void parallelSort(long[] a, int fromIndex, int toIndex)
static void parallelSort(short[] a, int fromIndex, int toIndex)
static void parallelSort(byte[] a, int fromIndex, int toIndex)
static <T extends Comparable<? super T>> void parallelSort(T[] a, int fromIndex,
int toIndex)
static <T> void parallelSort(T[] a, int fromIndex, int toIndex, Comparator<?
super T> cmp)
```

`parallelSetAll`方法用由一个函数计算得到的值填充数组。这个函数接收元素索引，然后计算相应位置上的值。

```
/* java.util.Arrays */
static void parallelSetAll(double[] array, IntToDoubleFunction generator)
static void parallelSetAll(int[] array, IntUnaryOperator generator)
static void parallelSetAll(long[] array, IntToLongFunction generator)
static <T> void parallelSetAll(T[] array, IntFunction<? extends T> generator)
```

`parallelPrefix`方法用一个给定结合操作的相应前缀的累加结果替换各个数组元素。

```
/* java.util.Arrays */
static void parallelPrefix(double[] array, DoubleBinaryOperator op)
static void parallelPrefix(double[] array, int fromIndex, int Toindex,
DoubleBinaryOperator op)
static void parallelPrefix(int[] array, IntBinaryOperator op)
static void parallelPrefix(int[] array, int fromIndex, int Toindex,
IntBinaryOperator op)
static void parallelPrefix(long[] array, LongBinaryOperator op)
static void parallelPrefix(long[] array, int fromIndex, int Toindex,
LongBinaryOperator op)
static <T> void parallelPrefix(T[] array, BinaryOperator<T> op)
static <T> void parallelPrefix(T[] array, int fromIndex, int Toindex,
BinaryOperator<T> op)
```

设`arr`是一个数组，则调用`parallelPrefix`方法时，对于每一个下标`i`，用给定的操作对`arr[0],arr[1],...arr[i]`进行计算，计算结果作为`arr[i]`的新值。例如：

```
int[] arr = [1, 2, 3, 4];
Arrays.parallelPrefix(arr, (x, y) -> x * y);
```

调用`parallelPrefix`方法之后，数组变成`[1, 1*2, 1*2*3, 1*2*3*4]`。

5.8 同步包装器

任何集合类都可以通过使用同步包装器变成线程安全的。

```
/* java.util.Collections */
static <E> Collection<E> synchronizedCollection(Collection<E> c)
static <E> List<E> synchronizedList(List<E> c)
static <E> Set<E> synchronizedSet(Set<E> c)
static <E> SortedSet<E> synchronizedSortedSet(SortedSet<E> c)
static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)
```

结果集合的方法使用锁加以保护，可以提供线程安全的访问。

要确保没有任何线程通过原始的非同步方法访问数据结构。要确保这一点，最容易的方法是不保存原始集合对象的任何引用，构造原始集合并立即传递给包装器，例如：

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>);
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>);
```

如果希望迭代访问一个集合，同时另一个线程有机会更改这个集合，就需要使用客户端锁定，例如：

```
synchronized (synchHashMap)
{
    Iterator<K> iter = synchHashMap.keySet().iterator();
    while (iter.hasNext()) ...;
}
```

注意：在迭代过程中，如果另一个线程更改了集合，迭代器会失效，抛出`ConcurrentModificationException`异常。

通常最好使用`java.util.concurrent`包中定义的集合，而不是同步包装器。但是，经常更改的数组列表是一个例外，同步的`ArrayList`要胜过`CopyOnWriteArrayList`。

6 任务和线程池

线程的创建和撤销会产生一定的开销。如果程序中有大量生命期很短的线程，那么不应该把每个任务映射到一个单独的线程，而应该使用线程池。线程池中包含许多准备运行的线程，为线程池提供一个`Runnable`，就会有一个线程调用`run`方法。当`run`方法退出时，这个线程不会死亡，而是留在池中准备为下一个请求提供服务。

6.1 Callable和Future

`Runnable`接口封装一个异步运行的任务，可以把它想象成一个没有参数和返回值的方法。`Callable`与`Runnable`类似，但是有返回值，也会抛出异常。`Callable`接口是一个参数化的类型，类型参数是返回值的类型，只有一个方法`call`。

```
public interface Callable<V>
{
    V call() throws Exception; // 运行一个将产生结果的任务
}
```

`Future`保存异步结算的结果。可以启动一个计算，将`Future`对象交给某个线程，这个`Future`对象的所有者在结果计算好后就可以得到结果。`Future<V>`接口有以下方法：

```
/* java.util.concurrent.Future<V> */
V get()
    // 获取结果，调用时会阻塞，直到计算完成。如果运行该任务的线程被中断，则抛出 Interrupted 异常
V get(long time, TimeUnit unit)
    // 获取结果，调用时会阻塞，直到计算完成或超时。如果超时，会抛出 TimeoutException 异常
    // 如果运行该任务的线程被中断，则抛出 Interrupted 异常
boolean cancel(boolean mayInterrupt)
    // 如果任务还没有开始，任务会被取消并且不再开始；如果任务已经开始，并且参数值为 true，任务会被中断
    // 如果成功执行了取消操作，则返回 true
boolean isCancelled()
    // 如果任务在完成前被取消，则返回 true
boolean isDone()
    // 如果任务结束，无论是正常完成、中途取消还是发生异常，都返回 true
```

执行`Callable`的一种方法是使用`FutureTask`类，它实现了`Future`和`Runnable`接口，可以构造一个线程来执行任务。例如：

```
Callable<Integer> task = ...;
FutureTask<Integer> futureTask = new FutureTask<Integer>(task);
Thread t = new Thread(futureTask);
t.start();
Integer result = task.get();
```

```
/* java.util.concurrent.FutureTask<V> */
FutureTask(Callable<V> task)
    // 构造一个 FutureTask 对象，它将在运行时执行给定的 Callable
FutureTask(Runnable task, V result)
    // 构造一个 FutureTask 对象，它将在运行时执行给定的 Runnable，get 方法在任务成功完成时返回指定的 result 对象
```

6.2 执行器

`Executors`类有许多静态工厂方法，用来构造线程池：

```
/* java.util.concurrent.Executors */
static ExecutorService newCachedThreadPool()
    // 返回一个缓存线程池。如果有空闲线程可用，就使用现有空闲线程执行任务；如果没有可用的空闲线程，则创建一个新线程
    // 如果线程已空闲 60 秒则终止该线程
static ExecutorService newFixedThreadPool(int threads)
    // 返回一个固定线程池，池中包含固定数目的线程。空闲线程会一直保留
    // 如果提交的任务多于空闲线程数，就把未得到服务的任务放到队列中，当其他任务完成后再运行排队的任务
static ExecutorService newSingleThreadExecutor()
    // 返回一个执行器，在一个单独的线程中顺序地执行任务
```

这3个方法返回`ThreadPoolExecutor`类的对象，这个类实现了`ExecutorService`接口。

如果线程生存期很短，或者大量时间都在阻塞，可以使用缓存线程池。如果线程工作量很大并且并不阻塞，就应该使用固定线程池。单线程执行器可以测量不使用并发的情况下应用的运行速度会慢多少，对于性能分析很有帮助。

可以用下面的方法之一将`Runnable`或`Callable`对象提交给`ExecutorService`：

```
/* java.util.concurrent.ExecutorService */
<T> Future<T> submit(Callable<T> task)
Future<?> submit(Runnable task) // get 方法在任务完成时返回 null
<T> Future<T> submit(Runnable task, T result) // get 方法在任务完成时返回指定的 result 对象
```

线程池会在方便的时候尽早执行提交的任务。调用`submit`方法时，会返回一个`Future`对象，可用来得到结果或取消任务。

使用完线程池时，调用`shutdown`方法，启动线程池的关闭序列，线程池完成已提交的任务但不再接受新的任务。当所有任务都完成时，线程池中的线程死亡。

```
/* java.util.concurrent.ExecutorService */
void shutdown()
```

另一种方法是调用`shutdownNow`方法，尝试停止所有正在执行的任务，取消所有尚未开始的任务，并返回正在等待执行的任务列表。

```
/* java.util.concurrent.ExecutorService */
List<Runnable> shutdownNow()
```

`ScheduledExecutorService`接口为调度执行或重复执行任务提供了一些方法，这是对支持建立线程池的`java.util.Timer`的泛化。`Executors`类的`newScheduledThreadPool`和`newSingleThreadScheduledExecutor`方法返回实现`ScheduledExecutorService`接口的对象。

```
/* java.util.concurrent.Executors */
static ScheduledExecutorService newScheduledThreadPool(int threads)
    // 返回一个线程池，使用给定数目的线程调度任务
static ScheduledExecutorService newSingleThreadScheduledExecutor()
    // 返回一个执行器，在一个单独的线程中调度任务

/* java.util.concurrent.ScheduledExecutorService */
ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)
ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)
    // 调度在指定的时间之后执行任务
ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long
period, TimeUnit unit)
    // 调度在初始延迟之后，周期性地运行给定的任务，周期长度是 period 个单位
ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long
delay, TimeUnit unit)
    // 调度在初始延迟之后，周期性地运行给定的任务，在一次调度完成和下次调度开始之间有长度为
delay 个单位的延迟
```

6.3 控制任务组

`invokeAny`方法提交一个`Callable`集合中的所有对象，并返回某个已完成任务的结果。我们不知道返回的究竟是哪个任务的结果，往往是最快完成的那个任务。对于搜索问题，如果愿意接受任何一种答案，就可以使用这个方法。例如，如果需要对一个大整数进行因数分解，可以提交很多任务，每个任务尝试对不同范围内的数进行分解，只要其中一个任务得到了答案，计算就可以终止了。

```
/* java.util.concurrent.ExecutorService */
<T> T  invokeAny(Collection<Callable<T>> tasks)
    // 执行给定的任务，返回其中一个任务的结果
<T> T  invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)
    // 执行给定的任务，返回其中一个任务的结果。如果超时，抛出 TimeoutException 异常
```

`invokeAll`方法提交一个`Callable`集合中的所有对象，这个方法会阻塞，直到所有任务都完成，并返回表示所有任务答案的`Future`对象列表。

```
/* java.util.concurrent.ExecutorService */
<T> List<Future<T>>  invokeAll(Collection<Callable<T>> tasks)
    // 执行给定的任务，返回所有任务的结果
<T> List<Future<T>>  invokeAll(Collection<Callable<T>> tasks, long timeout,
TimeUnit unit)
    // 执行给定的任务，返回所有任务的结果。如果超时，抛出 TimeoutException 异常
```

得到计算结果后，可以对`Future`对象列表进行迭代处理。例如：

```
void solve(Executor executor, Collcetion<Callable<T>> tasks)
{
    List<Future<T>> results = executor.invokeAll(tasks);
    for (Future<T> result : results)
    {
        T res = result.get();
        // 处理计算结果
    }
}
```

如果要按计算出结果的顺序处理结果，可以使用`ExecutorCompletionService`类。

```
/* java.util.concurrent.ExecutorCompletionService<V> */
ExecutorCompletionService(Executor e)
    // 构造一个 ExecutorCompletionService 对象，收集给定执行器的结果
Future<V> submit(Callable<V> task)
Future<V> submit(Runnable task, V result)
    // 提交一个任务给底层的执行器
Future<V> take()
    // 移除并返回下一个已完成的结果。如果没有可用的已完成结果，则阻塞
Future<V> poll()
    // 移除并返回下一个已完成的结果。如果没有可用的已完成结果，则返回 null
Future<V> poll(long time, TimeUnit unit)
    // 移除并返回下一个已完成的结果。如果没有可用的已完成结果，则等待一段时间；如果在等待时间内没有可完成结果，则返回 null
```

首先用执行器构造一个`ExecutorCompletionService`对象，将任务提交给这个对象。该对象会管理`Future`对象的一个阻塞队列，其中包含所提交任务的结果。使用`ExecutorCompletionService`类处理计算结果的示例代码如下：

```
void solve(Executor executor, Collcetion<Callable<T>> tasks)
{
    ExecutorCompletionService service = new ExecutorCompletionService(executor);
    for (Callable<T> task : tasks)
    {
        service.submit(task);
    }
    int size = tasks.size();
    for (int i = 0; i < size; i++)
    {
        result = service.take().get();
        // 处理计算结果
    }
}
```

下面的示例程序展示了如何使用`Callable`和执行器：

```
import java.io.*;
import java.nio.file.*;
import java.time.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class ExecutorDemo
```

```
{
    /**
     * 计算给定单词在文件中出现的次数
     */
    public static long occurrences(String word, Path path)
    {
        try (Scanner in = new Scanner(path))
        {
            int count = 0;
            while (in.hasNext())
            {
                if (in.next().equals(word)) count++;
            }
            return count;
        }
        catch (IOException e)
        {
            return 0;
        }
    }

    /**
     * 返回给定目录的所有子目录
     */
    public static Set<Path> descendants(Path rootDir) throws IOException
    {
        try (Stream<Path> entries = Files.walk(rootDir))
        {
            return
entries.filter(Files.isRegularFile).collect(Collectors.toSet());
        }
    }

    /**
     * 声明一个在文件中搜索单词的任务
     */
    public static Callable<Path> searchForTask(String word, Path path)
    {
        return () ->
        {
            try (Scanner in = new Scanner(path))
            {
                while (in.hasNext())
                {
                    if (in.next().equals(word)) return path;
                    if (Thread.currentThread.isInterrupted)
                    {
                        System.out.println("Search in " + path + " canceled.");
                        return null;
                    }
                }
                throw new NoSuchElementException();
            }
        };
    }

    public static void main(String[] args)
        throws InterruptedException, ExecutionException, IOException
```

```
    {
        try (Scanner in = new Scanner(System.in))
        {
            System.out.print("Enter base directory (e.g. /opt/jdk-9-src): ");
            String start = in.nextLine();
            System.out.print("Enter keyword (e.g. volatile): ");
            String word = in.nextLine();

            ////////////////////////////////////////////////////////
            // 第一部分，统计一个目录树中给定单词出现的次数
            ////////////////////////////////////////////////////////
            Set<Path> files = descendants(Path.of(start));
            ArrayList<Callable<Long>> tasks = new ArrayList<Callable<Long>>();
            for (Path file : files)
            {
                Callable<Long> task = () -> occurrences(word, file);
                tasks.add(task);
            }
            ExecutorService executor = Executors.newCachedThreadPool();

            Instant stratTime = Instant.now();
            List<Future<Long>> results = executor.invokeAll(tasks);
            long total = 0;
            for (Future<Long> result : results)
            {
                total += result.get();
            }
            Instant endTime = Insant.now();
            System.out.println("Occurrences of " + word + ": " + total);
            System.out.println("Time elapsed: " + Duration.between(startTime,
endTime).toMillis() + " ms");

            ////////////////////////////////////////////////////////
            // 第二部分，搜索包含给定单词的某个文件
            ////////////////////////////////////////////////////////
            ArrayList<Callable<Path>> searchTasks = new
ArrayList<Callable<Path>>();
            for (Path file : files)
            {
                searchTasks.add(searchForTask(word, file));
            }
            Path found = executor.invokeAny(searchTasks);
            System.out.println(word + " occurs in: " + found);

            // 打印线程池的最大大小
            if (executor instanceof ThreadPoolExecutor)
            {
                System.out.println("Largest pool size: " + ((ThreadPoolExecutor)
executor).getLargestPoolSize());
            }
            executor.shutdown();
        }
    }
}
```

6.4 fork-join框架

Java 7 引入了 fork-join 框架，用来支持多核处理器。这类任务可以分解为若干子任务，在多个处理器上并发执行，然后合并这些结果，其基本结构如下所示：

```
if（任务规模足够小）
    直接求解
else
{
    将任务分解为子任务
    递归求解子任务
    合并结果
}
```

要使用 fork-join 框架解决这类问题，需要提供一个扩展 `RecursiveTask<T>` 的类（如果计算会生成一个类型为 `T` 的结果）或者提供一个扩展 `RecursiveAction` 的类（如果不生成任何结果），覆盖 `conpute` 方法定义计算过程。

```
/* java.util.concurrent.RecursiveAction */
protected abstract void compute() // 此任务执行的主要计算

/* java.util.concurrent.RecursiveTask<T> */
protected abstract T compute() // 此任务执行的主要计算
```

`ForkJoinTask<V>` 类是 fork-join 任务的抽象基类，`RecursiveTask<T>` 和 `RecursiveAction` 都是它的子类。一个 `ForkJoinTask` 对象是一个类似线程的实体，但是比普通线程更加轻量级。`ForkJoinTask<V>` 类定义了对任务所做的操作。

```
/* java.util.concurrent.ForkJoinTask<V> */
public final V join()
    // 当计算完成时，返回计算结果。异常完成会导致 RuntimeException 或 Error
public final V invoke()
    // 开始执行此任务，在必要时等待其完成，并返回其结果
public static void invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2)
public static void invokeAll(ForkJoinTask<?>... tasks)
public static <T extends ForkJoinTask<?>> Collection<T> invokeAll(Collection<T>
tasks)
    // 接收任务并阻塞，直到所有任务完成后返回
```

fork-join 任务要提交到 `ForkJoinPool` 中执行

```
/* java.util.concurrent.ForkJoinPool */
public <T> T invoke(ForkJoinTask<T> task)
    // 执行给定的任务，返回计算结果
public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
    // 执行给定集合中的所有任务，当所有任务都完成后，返回 Future 对象的列表
```

在下面的示例程序中，使用 fork-join 框架统计数组中满足某个特定属性的元素个数。

```
import java.util.concurrent.*;
import java.util.function.*;

class Counter extends RecursiveTask<Integer>
{
    public static final int THRESHOLD = 1000;
    private double[] values;
```

```
    private int from;
    private int to;
    private DoublePredicate filter;

    public Counter(double[] values, int from, int to, DoublePredicate filter)
    {
        this.values = values;
        this.from = from;
        this.to = to;
        this.filter = filter;
    }

    protected Integer compute()
    {
        if (to - from < THRESHOLD)
        {
            int count = 0;
            for (int i = from, i < to; i++)
            {
                if (filter.test(values[i])) count++;
            }
            return count;
        }
        else
        {
            int mid = (from + to) / 2;
            Counter first = new Counter(values, from, mid, filter);
            Counter second = new Counter(values, mid, to, filter);
            invokeAll(first, second);
            return first.join() + second.join();
        }
    }
}

public class ForkJoinTest
{
    public static void main(String[] args)
    {
        final int SIZE = 10000000;
        double[] numbers = new double[SIZE];
        for (int i = 0; i < SIZE; i++)
        {
            numbers[i] = Math.random();
        }
        Counter counter = new Counter(numbers, 0, numbers.length, x -> x > 0.5);
        ForkJoinPool pool = new ForkJoinPool();
        pool.invole(counter);
        System.out.println(counter.join());
    }
}
```

在后台，fork-join 框架使用了一种有效的智能方法来平衡可用线程的工作负载，这种方法称为**工作密取**。每个工作线程都有一个双端队列来完成任务，一个工作线程将子任务压入其双端队列的队头。当工作线程空闲时，它会从另一个双端队列的队尾密取一个任务。由于打的子任务都在队尾，这种密取很少出现。

7 异步计算

7.1 可完成Future

当有一个`Future`对象时，需要调用`get`方法来获得值，这个方法会阻塞，直到值可用。`CompletableFuture`类实现了`Future`接口，它提供了获得结果的另一种机制，可以不必阻塞等待。

使用`thenAccept`方法注册一个回调，当结果可用时，就会在某个线程中利用该结果调用这个回调。通过这种方式，无须阻塞就可以在结果可用时对结果进行处理。

```
/* java.util.concurrent.CompletableFuture<T> */
public CompletableFuture<Void> thenAccept(Consumer<? super T> action)
    // 当前 CompletableFuture 正常完成时，将结果作为给定 action 的参数并执行 action
```

要想异步运行任务并得到`CompletableFuture`对象，不要把它直接提交给`ExecutorService`，而应当使用静态方法`CompletableFuture.supplyAsync`。

```
/* java.util.concurrent.CompletableFuture<T> */
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
    // 将 supplier 指定的任务提交给 ForkJoinPool.commonPool() 异步完成，并返回一个
CompletableFuture 对象
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor
executor)
    // 将 supplier 指定的任务提交给指定的 executor 异步完成，并返回一个
CompletableFuture 对象
```

注意`supplyAsync`方法的第一个参数是`Supplier<U>`，它描述了无参数且返回值类型为 U 的函数，不过`Supplier`函数不能抛出检查型异常。

`CompletableFuture`可以在两种情况下完成：得到一个结果，或者有一个未捕获的异常。要处理这两种情况，可以使用`whenComplete`方法，为运行结果和异常提供处理函数。

```
/* java.util.concurrent.CompletableFuture<T> */
CompletableFuture<T> whenComplete(BiConsumer<? super T, ? super Throwable>
action)
    // 当前任务完成时，对结果（如果没有就为 null）和异常（如果没有就为 null）调用给定的
action，并返回一个新的 CompletableFuture 对象
    // 当 action 返回时，新的 CompletableFuture 对象完成
    // 如果 action 本身遇到异常，则返回的 CompletableFuture 对象会异常地完成此异常，除非此
对象也异常地完成
```

用`supplyAsync`方法创建`CompletableFuture`对象时，任务完成时会隐式地设置完成值。要想显式地设置结果，可以使用`complete`方法，这样可以提供更大的灵活性：

```
/* java.util.concurrent.CompletableFuture<T> */
boolean complete(T value)
    // 当运算未完成时，将结果值设置为给定值
    // 如果此调用将当前 CompletableFuture 对象转换为完成状态，则返回 true；否则返回 false
```

对于异常完成的`CompletableFuture`对象，可以使用`completeExecptionally`方法设置抛出的异常：

```
/* java.util.concurrent.CompletableFuture<T> */
boolean completeExecptionally(Throwable ex)
    // 当运算未完成时，将结果值设置为给定的异常
    // 如果此调用将当前 CompletableFuture 对象转换为完成状态，则返回 true；否则返回 false
```

可以在多个线程中对同一个`CompletableFuture`对象安全地调用`complete`或`completeExecptionally`方法，如果`CompletableFuture`对象已完成，这些调用没有任何作用。

使用`completedFuture`静态方法可以获得具有给定值的已完成的`CompletableFuture`对象：

```
/* java.util.concurrent.CompletableFuture<T> */
public static <U> CompletableFuture<U> completedFuture(U value)
    // 返回一个具有给定值的已完成的 CompletableFuture 对象
```

与普通的`Future`不同，调用`cancel`方法时，`CompletableFuture`的计算不会中断，只会把这个`CompletableFuture`对象设置为以异常方式完成。

```
/* java.util.concurrent.CompletableFuture<T> */
boolean cancel(boolean mayInterruptIfRunning)
    // 如果任务没有完成，则用 CancellationException 异常完成此 CompletableFuture 对象
    // 尚未完成的依赖 CompletableFuture 对象也将异常完成，用此 CancellationException 导致的 CompletionException 完成
```

7.2 组合可完成Future

非阻塞调用通过回调来实现，程序员为任务完成之后要执行的动作注册一个回调。如果下一个动作也是异步的，在它之后的下一个动作就会在一个不同的回调中。对于一个由多个异步操作组成的控制流，需要多个回调，程序员会按照执行顺序依次考虑这些异步操作，但实际上程序逻辑会分散在不同的回调中。要在一组回调中实现这样的控制流，或者要理解所实现的控制流，会很有难度。`CompletableFuture`类提供了一种机制来解决这个问题，可以将异步任务组合为一个处理管线。

下表列出了处理单个`CompletableFuture`对象的方法：

方法	参数	描述
`thenApply`	`T -> U`（表示一个函数式接口，参数类型为 T，返回类型为 U，下同）	对结果应用一个函数
`thenAccept`	`T -> void`	对结果应用一个函数，不过结果为 `void`
`thenCompose`	`T -> CompletableFuture<U>`	对结果调用函数并执行返回的 future
`handle`	`(T, Throwable) -> U`	处理结果或错误，生成一个新结果
`whenComplete`	`(T, Throwable) -> void`	处理结果或错误，不生成新结果
`exceptionally`	`Throwable -> T`	处理错误，生成一个结果
`completeOnTimeout`	`T, long, TimeUnit`	如果超时，生成给定值作为结果
`orTimeout`	`long, TimeUnit`	如果超时，生成一个 `TimeoutException` 异常
`thenRun`	`Runnable`	执行 `Runnable`，结果为 `void`

上述每个方法还有两个带有 `Async` 后缀的形式，与7.1中的 `supplyAsync` 方法类似，其中一个使用共享 `ForkJoinPool`，另一个通过 `Executor` 参数指定执行器。

上表中有 `void` 结果的方法通常都在处理管线的最后使用。

下表列出了组合多个 `CompletableFuture` 对象的方法：

方法	参数	描述
`thenCombine`	`CompletableFuture<U>, (T, U) -> V`	执行两个动作并用给定函数组合结果
`thenAcceptBoth`	`CompletableFuture<U>, (T, U) -> void`	执行两个动作并用给定函数组合结果，不过结果为 `void`
`runAfterBoth`	`CompletableFuture<?>, Runnable`	两个都完成后执行 `Runnable`
`applyToEither`	`CompletableFuture<T>, T -> V`	得到其中一个的结果时，调用给定的函数
`acceptEither`	`CompletableFuture<T>, T -> void`	得到其中一个的结果时，调用给定的函数，不过结果为 `void`
`runAfterEither`	`CompletableFuture<?>, Runnable`	其中一个完成后执行 `Runnable`
`static allOf`	`CompletableFuture<?>...`	所有给定 future 都完成后完成，结果为 `void`
`static anyOf`	`CompletableFuture<?>...`	任意给定 future 完成后则完成，结果为 `void`

下面给出一个完整的程序，它会读取一个 Web 页面，扫描页面得到其中的图像，并保存在本地。

```
import java.awt.image.*;
import java.io.*;
import java.net.*;
import java.nio.charset.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.regex.*;

import javax.imageio.*;

public class CompletableFutureDemo
{
    private static final Pattern IMG_PATTERN = Pattern.compile(
        "[<]\\s*[iI][mM][gG]\\s*[^>]*[sS][rR][cC]\\s*[=]\\s*['\"]([^'\"]*)['\"]
[^>]*[>]");
    private ExecutorService executor = Executors.newCachedThreadPool();
    private URL urlToProcess;

    public CompletableFuture<String> readPage(URL url)
    {
        return CompletableFuture.supplyAsync(() ->
            {
                try
                {
                    String contents = new
String(url.openStream().readAllBytes(), StandardCharsets.UTF_8);
                    System.out.println("Read page from " + url);
                    return contents;
```

```
                }
                catch (IOException e)
                {
                    throw new UncheckedIOException(e);
                }
            }, executor);
    }

    public List<URL> getImageURLs(String webpage)
    {
        try
        {
            ArrayList<URL> result = new ArrayList<URL>();
            Matcher matcher = IMG_PATTERN.matcher(webpage);
            while (matcher.find())
            {
                URL url = new URL(urlToProcess, matcher.group(1));
                result.add(url);
            }
            System.out.println("Found URLs: " + result);
            return result;
        }
        catch (IOException e)
        {
            throw new UncheckedIOException(e);
        }
    }

    public CompletableFuture<List<BufferedImage>> getImages(List<URL> urls)
    {
        return CompletableFuture.supplyAsync(() ->
            {
                try
                {
                    ArrayList<BufferedImage> result = new
ArrayList<BufferedImage>();
                    for (URL url : urls)
                    {
                        result.add(ImageIO.read(url));
                        System.out.println("Loaded " + url);
                    }
                    return result;
                }
                catch (IOException e)
                {
                    throw new UncheckedIOException(e);
                }
            }, executor);
    }

    public void saveImages(List<BufferedImage> images)
    {
        System.out.println("Saving " + images.size() + " images");
        try
        {
            for (int i = 0; i < images.size(); i++)
            {
                String filename = "/tmp/image" + (i + 1) + ".png";
```

```
                ImageIO.write(images.get(i), "PNG", new File(filename));
            }
        }
        catch (IOException e)
        {
            throw new UncheckedIOException(e);
        }
        executor.shutdown();
    }

    public void run(URL url) throws IOException, InterruptedException
    {
        CompletableFuture.completedFuture(url)
            .thenComposeAsync(this::readPage, executor)
            .thenApply(this::getImageURLs)
            .thenCompose(this::getImages)
            .thenAccept(this::saveImages);
    }

    public static void main(String[] args) throws IOException,
InterruptedException
    {
        new CompletableFutureDemo().run(new
URL("http://horstmann.com/index.html"));
    }
}
```

8 进程

8.1 建立一个进程

用`ProcessBuilder`类建立并配置进程。首先使用构造器构造`ProcessBuilder`对象。

```
/* java.lang.ProcessBuilder */
ProcessBuilder(String... command)
ProcessBuilder(List<String> command)
    // 用给定的命令和参数构造一个 ProcessBuilder 对象
```

每个进程都有一个工作目录，用来解析相对目录名。默认情况下，进程的工作目录与虚拟机相同，通常是启动 Java 程序的那个目录。可以用`directory`方法改变工作目录。

```
/* java.lang.ProcessBuilder */
ProcessBuilder directory(File directory) // 设置进程的工作目录
```

接下来，要指定如何处理进程的标准输入、输出和错误流。默认情况下，它们分别是一个管道，可以通过`Process`对象访问。

```
/* java.lang.Process */
abstract OutputStream getOutputStream() // 得到一个输出流，用于写入进程的输入流
abstract InputStream getInputStream() // 得到一个输入流，用于读取进程的输出流
abstract InputStream getErrorStream() // 得到一个输入流，用于读取进程的错误流
```

进程的输入流是 JVM 的输出流，我们会写入这个流，而我们写入的内容会成为进程的输入。与之相反，进程的输出流和错误流是 JVM 的输入流，因为我们会读取进程写入输出流和错误流的内容，对程序员来说是输入流。

使用`redirectInput`、`redirectionOutput`、`redirectError`方法重定向进程的输入流、输出流和错误流。

```
/* java.lang.ProcessBuilder */
ProcessBuilder redirectInput(File file)
ProcessBuilder redirectOutput(File file)
ProcessBuilder redirectError(File file)
    // 将进程的标准输入、输出和错误流重定向到给定的文件
ProcessBuilder redirectInput(ProcessBuilder.Redirect source)
ProcessBuilder redirectOutput(ProcessBuilder.Redirect destination)
ProcessBuilder redirectError(ProcessBuilder.Redirect destination)
    // 重定向进程的标准输入、输出和错误流，参数值可以是以下几种：
    /* ProcessBuilder.Redirect.PIPE -- 默认行为，通过 Process 对象访问
     * ProcessBuilder.Redirect.INHERIT -- 虚拟机的流
     * ProcessBuilder.Redirect.from(file) -- 从给定的文件读
     * ProcessBuilder.Redirect.to(file) -- 向给定的文件写。如果给定文件已存在，则该文件的内容将被覆盖
     * ProcessBuilder.Redirect.appendTo(file) -- 向给定的文件尾部追加内容
     */
```

可以使用`redirectErrorStream`方法合并输出流和错误流，合并之后就不能调用`ProcessBuilder.redirectError`方法和`Process.getErrorStream`方法。

```
/* java.lang.ProcessBuilder */
ProcessBuilder redirectErrorStream(boolean redirectErrorStream)
    // 如果参数为 true，则合并输出流和错误流
```

要想修改进程的环境变量，可以调用`environment`方法获取环境变量映射，之后通过对映射的操作设置环境变量。

```
/* java.lang.ProcessBuilder */
Map<String, String> environment()
```

如果希望利用管道将一个进程的输出作为另一个进程的输入，Java 9 提供了一个`startPipeline`方法。传入一个`ProcessBuilder`列表，将它们连接成一个管道，前一个进程的输出作为后一个进程的输入，从最后一个进程读取结果。

```
/* java.lang.ProcessBuilder */
static List<Process> startPipeline(List<ProcessBuilder> builders)
    // 启动一个进程管线，将各个进程的标准输出连接到下一个进程的标准输入
```

8.2 运行一个进程

配置了`ProcessBuilder`对象之后，要调用`start`方法启动进程。

```
/* java.lang.ProcessBuilder */
Process start() // 启动进程，并生成它的 Process 对象
```

要等待进程完成可以调用无参的`waitFor`方法：

```
/* java.lang.Process */
abstract int waitFor()
    // 等待进程完成并生成退出值
```

如果不想无限期地等待，可以使用有时限的`waitFor`方法，不过这个方法不返回进程的退出值。要想得到退出值，还需要调用`exitValue`方法。

```
/* java.lang.Process */
boolean waitFor(long timeout, TimeUnit unit)
    // 等待进程完成，不过不能超出给定的时间。如果进程退出，返回 true
abstract int exitValue()
    // 返回进程的退出值
```

在程序运行中，可以调用`isAlive`方法检查进程是否存活。

```
/* java.lang.Process */
boolean isAlive() // 检查这个进程是否存活
```

要杀死进程，可以使用`destory`或`destoryForcibly`方法，这两个方法的区别取决于平台。例如，在UNIX 平台上，`destory`方法会以`SIGTERM`终止进程，`destoryForcibly`方法会以`SIGKILL`终止进程。

```
/* java.lang.Process */
abstract void destory()
Process destoryForcibly()
    // 终止这个进程。可能正常终止，也可能强制终止
```

`supportsNormalTermination`方法检查进程是否可以正常终止。

```
/* java.lang.Process */
boolean supportsNormalTermination()
    // 如果可以正常终止，则返回 true；否则，返回 false，此时必须强制销毁
```

`onExit`方法可以在进程完成时接收到一个异步通知，并返回一个`CompletableFuture<Process>`，可以用它执行进程结束后的处理工作。

```
/* java.lang.Process */
CompletableFuture<Process> onExit() // 生成一个 CompletableFuture，在进程退出时执行
```

8.3 进程句柄

要获得程序启动的进程的更多信息，或者想了解计算机上正在运行的任何其他进程，可以使用`ProcessHandle`接口。有4种方式得到`ProcessHandle`对象：

1. 给定一个`Process`对象，调用`toHandle`方法。
2. 给定一个`long`类型的操作系统进程 ID，`ProcessHandle.of(id)`可以生成这个进程的句柄。
3. `ProcessHandle.current()`返回运行这个 Java 虚拟机的进程的句柄。
4. `ProcessHandle.allProcesses()`可以生成对当前进程可见的所有操作系统进程的句柄。

```
/* java.lang.Process */
ProcessHandle toHandle() // 生成描述这个进程的 ProcessHandle

/* java.lang.ProcessHandle */
static Optional<ProcessHandle> of(long pid) // 生成有给定 PID 的进程的句柄
static ProcessHandle current() // 生成虚拟机进程的句柄
static Stream<ProcessHandle> allProcesses() // 生成所有进程的句柄
```

给定一个进程句柄，可以得到它的进程 ID、父进程、子进程和后代进程。

```
/* java.lang.ProcessHandle */
long pid() // 生成这个进程的 PID
Optional<ProcessHandle> parent() // 返回父进程的句柄
Stream<ProcessHandle> children() // 生成子进程的句柄
Stream<ProcessHandle> descendants() // 生成后代进程的句柄
```

注意：`allProcesses`、`children`和`descendants`方法返回的`Stream<ProcessHandle>`实例只是当时的快照，流中的进程在访问时可能已经终止，也可能启动了其他进程，而新启动的进程不在流中。

`info`方法可以生成一个`ProcessHandle.Info`对象，它提供了一些方法来获得进程的有关信息。

```
/* java.lang.ProcessHandle */
ProcessHandle.Info info() // 生成这个进程的详细信息

/* java.lang.ProcessHandle.Info */
Optional<String[]> arguments() // 返回进程参数的字符串数组
Optional<String> command() // 返回进程的可执行路径名
Optional<String> commandLine() // 返回进程的命令行
Optional<Instant> startInstant() // 返回进程的开始时间
Optional<Duration> toalCpuDuration() // 返回进程使用的CPU时间
Optional<String> user() // 返回进程的使用者
```

与`Process`类一样，`ProcessHandle`接口也有`isAlive`、`supportsNormalTermination`、`destory`、`destoryForcibly`和`onExit`方法，但是没有`waitFor`方法。