

Java 多线程编程详解

我们不妨设想，为了创建一个新的线程，我们需要做些什么？很显然，我们必须指明这个线程所要执行的代码，而这就是在 Java 中实现多线程我们所需要做的一切！

真是神奇！Java 是如何做到这一点的？通过类！作为一个完全面向对象的语言，Java 提供了类 `java.lang.Thread` 来方便多线程编程，这个类提供了大量的方法来方便我们控制自己的各个线程，我们以后的讨论都将围绕这个类进行。

那么如何提供给 Java 我们要线程执行的代码呢？让我们来看看 `Thread` 类。`Thread` 类最重要的方法是 `run()`，它为 `Thread` 类的方法 `start()` 所调用，提供我们的线程所要执行的代码。为了指定我们自己的代码，只需要覆盖它！

方法一：继承 `Thread` 类，覆盖方法 `run()`

我们在创建的 `Thread` 类的子类中重写 `run()`，加入线程所要执行的代码即可。

下面是一个例子：

```
public class MyThread extends Thread {
    int count= 1, number;
    public MyThread(int num) {
        number = num;
        System.out.println("创建线程 " + number);
    }
    public void run() {
        while(true) {
            System.out.println("线程 " + number + ":计数 " + count);
            if(++count== 6) return;
        }
    }
}

public static void main(String args[]) {
    for(int i = 0; i < 5; i++) new MyThread(i+1).start();
}
```

这种方法简单明了，符合大家的习惯，但是，它也有一个很大的缺点，那就是如果我们的类已经从一个类继承（如小程序必须继承自 `Applet` 类），则无法再继承 `Thread` 类，这时如果我们又不想建立一个新的类，应该怎么办呢？

我们不妨来探索一种新的方法：我们不创建 `Thread` 类的子类，而是直接使用它，那么我们只能将我们的方法作为参数传递给 `Thread` 类的实例，有点类似回调函数。但是 Java 没有指针，我们只能传递一个包含这个方法的类的实例。那么如何限制这个类必须包含这一方法呢？当然是使用接口！（虽然抽象类也可满足，但是需要继承，而我们之所以要采用这种新方法，不就是为了避免继承带来的限制吗？）

Java 提供了接口 `java.lang.Runnable` 来支持这种方法。

方法二：实现 Runnable 接口

Runnable 接口只有一个方法 run()，我们声明自己的类实现 Runnable 接口并提供这一方法，将我们的线程代码写入其中，就完成了这一部分的任务。

但是 Runnable 接口并没有任何对线程的支持，我们还必须创建 Thread 类的实例，这一点通过 Thread 类的构造函数

public Thread(Runnable target); 来实现。

下面是一个例子：

```
public class MyThread implements Runnable {
    int count= 1, number;
    public MyThread(int num) {
        number = num;
        System.out.println("创建线程 " + number);
    }
    public void run() {
        while(true) {
            System.out.println("线程 " + number + ":计数 " + count);
            if(++count== 6) return;
        }
    }
    public static void main(String args[]) {
        for(int i = 0; i < 5; i++) new Thread(new MyThread(i+1)).start();
    }
}
```

严格地说，创建 Thread 子类的实例也是可行的，但是必须注意的是，该子类必须没有覆盖 Thread 类的 run 方法，否则该线程执行的将是子类的 run 方法，而不是我们用以实现 Runnable 接口的类的 run 方法，对此大家不妨试验一下。

使用 Runnable 接口来实现多线程使得我们能够在类中包容所有的代码，有利于封装，它的缺点在于，我们只能使用一套代码，若想创建多个线程并使各个线程执行不同的代码，则仍必须额外创建类，如果这样的话，在大多数情况下也许还不如直接用多个类分别继承 Thread 来得紧凑。

综上所述，两种方法各有千秋，大家可以灵活运用。

下面让我们一起来研究一下多线程使用中的一些问题。

三：线程的七种状态

下面为线程中的 7 中非常重要的状态：（有的书上也只有认为前五种状态：而将“锁池”和“等待池”都看成是“阻塞”状态的特殊情况：这种认识也是正确的，但是将“锁池”和“等待池”单独分离出来有利于对程序的理解）

1. 初始状态，线程创建，线程对象调用 start()方法。

2. 可运行状态，也就是等待 Cpu 资源，等待运行的状态。
3. 运行状态，获得了 cpu 资源，正在运行状态。
4. 阻塞状态，也就是让出 cpu 资源，进入一种等待状态，而且不是可运行状态，有三种情况会进入阻塞状态。
 - 1)如等待输入（输入设备进行处理，而 CUP 不处理），则放入阻塞，直到输入完毕，阻塞结束后会进入可运行状态。
 - 2)线程休眠，线程对象调用 sleep()方法，阻塞结束后会进入可运行状态。
 - 3)线程对象 2 调用线程对象 1 的 join()方法，那么线程对象 2 进入阻塞状态，直到线程对象 1 中止。
5. 中止状态，也就是执行结束。
6. 锁池状态
7. 等待队列

四：线程的优先级

线程的优先级代表该线程的重要程度，当有多个线程同时处于可执行状态并等待获得 CPU 时间时，线程调度系统根据各个线程的优先级来决定给谁分配 CPU 时间，优先级高的线程有更大的机会获得 CPU 时间，优先级低的线程也不是没有机会，只是机会要小一些罢了。

你可以调用 Thread 类的方法 getPriority() 和 setPriority()来存取线程的优先级，线程的优先级介于 1(MIN_PRIORITY) 和 10(MAX_PRIORITY) 之间，缺省是 5(NORM_PRIORITY)。

五：线程的同步

由于同一进程的多个线程共享同一片存储空间，在带来方便的同时，也带来了访问冲突这个严重的问题。Java 语言提供了专门机制以解决这种冲突，有效避免了同一个数据对象被多个线程同时访问。

由于我们可以通过 private 关键字来保证数据对象只能被方法访问，所以我们只需针对方法提出一套机制，这套机制就是 synchronized 关键字，它包括两种用法：synchronized 方法和 synchronized 块。

1. synchronized 方法：通过在方法声明中加入 synchronized 关键字来声明 synchronized 方法。如：

```
public synchronized void accessVal(int newVal);
```

synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）。

在 Java 中，不光是类实例，每一个类也对应一把锁，这样我们也可将类的静态成员函数声明为 synchronized，以控制其对类的静态成员变量的访问。

synchronized 方法的缺陷：若将一个大的方法声明为 synchronized 将会大大影响效率，

典型地，若将线程类的方法 `run()` 声明为 `synchronized`，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 `synchronized` 方法的调用都永远不会成功。当然我们可以通过将访问类成员变量的代码放到专门的方法中，将其声明为 `synchronized`，并在主方法中调用来解决这一问题，但是 Java 为我们提供了更好的解决办法，那就是 `synchronized` 块。

2. `synchronized` 块：通过 `synchronized` 关键字来声明 `synchronized` 块。语法如下：

```
synchronized(syncObject) {  
    //允许访问控制的代码  
}
```

`synchronized` 块是这样一个代码块，其中的代码必须获得对象 `syncObject`（如前所述，可以是类实例或类）的锁方能执行，具体机制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。

六：线程的阻塞

为了解决对共享存储区的访问冲突，Java 引入了同步机制，现在让我们来考察多个线程对共享资源的访问，显然同步机制已经不够了，因为在任意时刻所要求的资源不一定已经准备好了被访问，反过来，同一时刻准备好了的资源也可能不止一个。为了解决这种情况下的访问控制问题，Java 引入了对阻塞机制的支持。

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析：

1. `sleep()` 方法：`sleep()` 允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新进入可执行状态。

典型地，`sleep()` 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

2. `suspend()` 和 `resume()` 方法：两个方法配套使用，`suspend()`使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 `resume()` 被调用，才能使得线程重新进入可执行状态。典型地，`suspend()` 和 `resume()` 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 `resume()` 使其恢复。

3. `yield()` 方法：`yield()` 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 `yield()` 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。

4. `wait()` 和 `notify()` 方法：两个方法配套使用，`wait()` 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 `notify()` 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 `notify()` 被调用。

初看起来它们与 `suspend()` 和 `resume()` 方法对没有什么分别，但是事实上它们是截然

不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。

上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 `Thread` 类，但是这一对却直接隶属于 `Object` 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 `wait()` 方法导致线程阻塞，并且该对象上的锁被释放。而调用任意对象的 `notify()` 方法则导致因调用该对象的 `wait()` 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 `synchronized` 方法或块中调用，理由也很简单，只有在 `synchronized` 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 `synchronized` 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现 `IllegalMonitorStateException` 异常。

`wait()` 和 `notify()` 方法的上述特性决定了它们经常和 `synchronized` 方法或块一起使用，将它们和操作系统的进程间通信机制作一个比较就会发现它们的相似性：`synchronized` 方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 `block` 和 `wakeup` 原语（这一对方法均声明为 `synchronized`）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

关于 `wait()` 和 `notify()` 方法最后再说明两点：

第一：调用 `notify()` 方法导致解除阻塞的线程是从因调用该对象的 `wait()` 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 `notify()`，还有一个方法 `notifyAll()` 也可起到类似作用，唯一的区别在于，调用 `notifyAll()` 方法将把因调用该对象的 `wait()` 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，`suspend()` 方法和不指定超时期限的 `wait()` 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 `wait()` 和 `notify()` 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，更容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

七：守护线程

守护线程是一类特殊的线程，它和普通线程的区别在于它并不是应用程序的核心部分，当一个应用程序的所有非守护线程终止运行时，即使仍然有守护线程在运行，应用程序也将终止，反之，只要有一个非守护线程在运行，应用程序就不会终止。守护线程一般被用于在后台为其它线程提供服务。

可以通过调用方法 `isDaemon()` 来判断一个线程是否是守护线程，也可以调用方法 `setDaemon()` 来将一个线程设为守护线程。

八：线程组

线程组是一个 Java 特有的概念，在 Java 中，线程组是类 `ThreadGroup` 的对象，每个线程都隶属于唯一一个线程组，这个线程组在线程创建时指定并在线程的整个生命期内都不能更改。你可以通过调用包含 `ThreadGroup` 类型参数的 `Thread` 类构造函数来指定线程属的线程组，若没有指定，则线程缺省地隶属于名为 `system` 的系统线程组。

在 Java 中，除了预建的系统线程组外，所有线程组都必须显式创建。

在 Java 中，除系统线程组外的每个线程组又隶属于另一个线程组，你可以在创建线程组时指定其所隶属的线程组，若没有指定，则缺省地隶属于系统线程组。这样，所有线程组组成了一棵以系统线程组为根的树。

Java 允许我们对一个线程组中的所有线程同时进行操作，比如我们可以通过调用线程组的相应方法来设置其中所有线程的优先级，也可以启动或阻塞其中的所有线程。

Java 的线程组机制的另一个重要作用是线程安全。线程组机制允许我们通过分组来区分有不同安全特性的线程，对不同组的线程进行不同的处理，还可以通过线程组的分层结构来支持不对等安全措施采用。Java 的 `ThreadGroup` 类提供了大量的方法来方便我们对线程组树中的每一个线程组以及线程组中的每一个线程进行操作。

九：总结

在这一文章中，我们一起学习了 Java 多线程编程的方方面面，包括创建线程，以及对多个线程进行调度、管理。我们深刻认识到了多线程编程的复杂性，以及线程切换开销带来的多线程程序的低效性，这也促使我们认真地思考一个问题：我们是否需要多线程？何时需要多线程？

多线程的核心在于多个代码块并发执行，本质特点在于各代码块之间的代码是乱序执行的。我们的程序是否需要多线程，就是要看这是否也是它的内在特点。

假如我们的程序根本不要求多个代码块并发执行，那自然不需要使用多线程；假如我们的程序虽然要求多个代码块并发执行，但是却不要乱序，则我们完全可以用一个循环来简单高效地实现，也不需要多线程；只有当它完全符合多线程的特点时，多线程机制对线程间通信和线程管理的强大支持才能有用武之地，这时使用多线程才是值得的。