线程问题

1.定义

线程：线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程 中是可以有多个线程的，这个应用程序也可以称之为多线程程序。

进程：是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多 个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创 建、运行到消亡的过程

一个程序运行后至少有一个进程，一个进程中可以包含多个线程

2.创建多线程的方式.

1)继承Thread类创建线程

通过继承Thread类来创建并启动多线程的一般步骤如下

1 d定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的任务，run()方法也称为线程执行体。

2 创建Thread子类的实例，也就是创建了线程对象

3 启动线程，即调用线程的start()方法

2）实现Runnable接口创建线程

1】定义Runnable接口的实现类，一样要重写run()方法，这个run（）方法和Thread中的run()方法一样是线程的执行体

2】创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象

3】第三部依然是通过调用线程对象的start()方法来启动线程

3）使用Callable的call()方法和Future创建线程

和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()方法功能要强大
call()方法可以有返回值
call()方法可以声明抛出异常
Java5提供了Future接口来代表Callable接口里call()方法的返回值，并且为Future接口提供了一个实现类FutureTask，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Thread类的target。在Future接口里定义了几个公共方法来控制它关联的Callable任务。
执行步骤：
1】创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例（从java8开始可以直接使用Lambda表达式创建Callable对象）。
2】使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值
3】使用FutureTask对象作为Thread对象的target创建并启动线程（因为FutureTask实现了Runnable接口）
4】调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

4）使用线程池

合理利用线程池能够带来三个好处：

1. 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。
2. 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。

3. 提高线程的可管理性。可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内
存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java通过Executors中的静态方法提供四种常用线程池，分别为：

```
1.newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
2. newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
3.newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
4. newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO，LIFO，优先级)执行
创建方式：
ExecutorService service = Executors.newFixedThreadPool(2);
```

线程池API

```
Java里面线程池的顶级接口是 java.util.concurrent.Executor ，但是严格意义上讲 Executor 并不是一个线程
池，而只是一个执行线程的工具。真正的线程池接口是
java.util.concurrent.ExecutorService

 Executor接口<---继承--ExecutorService接口<--实现----
AbstractExecutorService<----实现-------ThreadPoolExecutor
```

使用线程池中线程对象的步骤：

```
public class ThreadPoolDemo {
public static void main(String[] args) {
// 创建线程池对象
ExecutorService service = Executors.newFixedThreadPool(2);//包含2个线程对象
// 创建Runnable实例对象
MyRunnable r = new MyRunnable();
//自己创建线程对象的方式
```

```
// Thread t = new Thread(r);
// t.start(); ---> 调用MyRunnable中的run()
// 从线程池中获取线程对象,然后调用MyRunnable中的run()
service.submit(r);
// 再获取个线程对象，调用MyRunnable中的run()
service.submit(r);
service.submit(r);
// 注意：submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。
// 将使用完的线程又归还到了线程池中
// 关闭线程池
//service.shutdown();
}
}
```

3.*Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runable接口的话，则很容易的实现资源共享。 总结： 实现Runnable接口比继承Thread类所具有的优势：

1. 适合多个相同的程序代码的线程去共享同一个资源。
2. 可以避免java中的单继承的局限性。
3. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。
4. 线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类

run()方法是多线程程序的一个执行目标。所有的多线代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。
在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread
对象的start()方法来运行多线程代码。
实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是继承Thread类还是实现
Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的.
Runnable对象仅仅作为Thread对象的target，Runnable实现类里包含的run()方法仅作为线程执行体。
而实际的线程对象依然是Thread实例，只是该Thread线程负责执行其target的run()方法。

4.Callable和Thread,Runnable三者的区别

实现Runnable和实现Callable接口的方式基本相同，不过是后者执行call()方法有返回值，后者线程执行体run()方法无返回值，因此可以把这两种方式归为一种这种方式与继承Thread类的方法之间的差别如下：

1、线程只是实现Runnable或实现Callable接口，还可以继承其他类。

2、这种方式下，多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。

3、但是编程稍微复杂，如果需要访问当前线程，必须调用Thread.currentThread()方法。

4、继承Thread类的线程类不能再继承其他父类（Java单继承决定）。

5.线程之间的通信

1.为什么要处理线程间通信?

1．多个线程并发执行时，在默认情况下CPU是**随机切换线程**的，当我们需要多个线程来共同完成一件任务，并且我们希望他们有规律的执行，那么多线程之间需要一些协调通信，以此来帮我们达到多线程共同操作一份数据。
2.当然如果我们没有使用线程通信来使用多线程共同操作同一份数据的话，虽然可以实现，但是在很大程度会造成多线程之间对同一共享变量的争夺，那样的话势必为造成很多错误和损失!
3.所以，我们才引出了线程之间的通信，`多线程之间的通信能够避免对同一共享变量的争夺.

2.如何保证线程间通信有效利用资源： *等待唤醒机制*。

1.

2. wait：线程不再活动，不再参与调度，进入 wait set 中，因此不会浪费 CPU 资源，也不会去竞争锁了，这时 的线程状态即是 WAITING。它还要等着别的线程执行一个特别的动作，也即是“通知（notify）”在这个对象 上等待的线程从wait set 中释放出来，重新进入到调度(ready queue）中
3. notify：则选取所通知对象的 wait set 中的一个线程释放；例如，餐馆有空位置后，等候就餐最久的顾客最先入座。
4. notifyAll：则释放所通知对象的 wait set 上的全部线程。

调用wait和notify方法需要注意的细节

1. ```
   1．wait方法与notify方法必须要由同一个锁对象调用。因为：对应的锁对象可以通过notify唤醒使用同一个锁对
   象调用的wait方法后的线程。
   2．wait方法与notify方法是属于Object类的方法的。因为：锁对象可以是任意对象，而任意对象的所属类都是继
   承了Object类的。
   3．wait方法与notify方法必须要在同步代码块或者是同步函数中使用。因为：必须要通过锁对象调用这2个方
   法。

6.线程间的安全问题

1.什么情况下回发生线程安全问题

当我们使用多个线程访问同一资源的时候，且多个线程中对资源有写的操作，就容易出现线程安全问题

2.解决线程安全问题的方法(同步机制)

2.1 同步代码块

```
格式：synchronize（锁对象）{

                    需要被同步的代码
            }
```

锁对象只是一个概念,可以想象为在对象上标记了一个锁.

1. 锁对象 可以是任意类型。
2. 多个线程对象 要使用同一把锁。
3. 一个线程在同步代码块中sleep了，并不会释放锁对象； `注意:在任何时候,最多允许一个线程拥有同步锁,谁拿到锁就进入代码块,其他的线程只能在外等着``(BLOCKED)`。

2.2 同步方法。

使用synchronized修饰的方法,就叫做同步方法,保证A线程执行该方法的 时候,其他线程只能在方法外等着.

```
public synchronized void method(){
可能会产生线程安全问题的代码
}
```

`锁对象是谁?` `对于非static方法,同步锁就是this。` `对于static方法,我们使用当前方法所在类的字节码对象(类名.class)。`

2.3 锁机制。

Lock锁也称同步锁，加锁与释放锁方法化了，如下：

```
public void lock() :加同步锁。
public void unlock() :释放同步锁
```

3.Lock和synchronized的选择

总结来说，Lock和synchronized有以下几点不同：

1）Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；
2）synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；
3）Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；
4）通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。
5）Lock可以提高多个线程进行读操作的效率。

4.synchronized的缺陷

当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：
1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；
2）线程执行发生异常，此时JVM会让线程自动释放锁。

那么如果这个获取锁的线程由于要等待IO或者其他原因（比如调用sleep方法）被阻塞了，但是又没有释放锁，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

因此就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间或者能够响应中断），通过Lock就可以办到。

7 线程状态

1.NEW(新建) :

线程刚被创建，但是并未启动。还没调用start方

2.Runnable(可运行)

线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。

3.Blocked(锁阻塞)

当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。

4.Waiting(无限等待)

一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入Waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够唤醒

5.TimedWaiting(计时等待)

同waiting状态，有几个方法有超时参数，调用他们将进入Timed Waiting状态。这一状态将一直保持到超时期满或者接收到唤醒通知。带有超时参数的常用方法有Thread.sleep 、Object.wait。

6.Teminated(被终止)

因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。