1、实现多线程的两种方法

实现多线程有两种方法：继承 Thread 和实现 Runnable 接口。

继承 Thread：
以卖票为例：

```
public class MyThread extends Thread {

    private static int COUNT = 5;

    private int ticket = COUNT;

    private String name;

    public MyThread(String s){

        name = s;

    }

    @Override

        public void run() {

        for (int i = 0; i < COUNT; i++){

            if(ticket > 0){

                System.out.println(name + "-->" + ticket--);

            }

        }

    }
```

测试使用：

```
MyThread thread1 = new MyThread("thread1");

        MyThread thread2 = new MyThread("thread2");
```

```
        thread1.start();

        thread2.start();
```

输出：

```
thread1-->5

thread2-->5

thread1-->4

thread2-->4

thread1-->3

thread2-->3

thread1-->2

thread2-->2

thread1-->1

thread2-->1
```

可以看到，这种方式每个线程自己拥有了一份票的数量，没有实现票的数量共享。下面看实现 Runnable 的方式：

实现 Runnable 接口：

```
public class MyRunnable implements Runnable {

    private static int COUNT = 5;

    private int ticket = COUNT;



    @Override

    public void run() {
```

```
        for(int i = 0; i < COUNT; i++){
            if(ticket > 0){
                System.out.println("ticket-->" + ticket--);
            }
        }
    }
}
```

测试使用：

```
 MyRunnable runnable = new MyRunnable();
        new Thread(runnable).start();
        new Thread(runnable).start();
```

输出：

```
ticket-->5
ticket-->3
ticket-->2
ticket-->1
ticket-->4
```

可以看到，实现 Runnable 的方式可以实现同一资源的共享。
实际工作中，一般使用实现 Runnable 接口的方式，是因为：

- 支持多个线程去处理同一资源，同时，线程代码和数据有效分离，体现了面向对象的思想；

- 避免了 Java 的单继承性，如果使用继承 Thread 的方式，那这个扩展类就不能再去继承其他类。

拓展：
Thread 的 start()和 run()方法区别：

start()方法用于启动一个线程，使其处于就绪状态，得到了 CPU 就会执行，而直接调用 run()方法，就相当于是普通的方法调用，会在主线程中直接运行，此时没有开启一个线程。

下列方法中哪个是执行线程的方法？ （）

- A、run()
- B、start()
- C、sleep()
- D、suspend()

正确答案：A

- run()方法用来执行线程体中具体的内容
- start()方法用来启动线程对象，使其进入就绪状态
- sleep()方法用来使线程进入睡眠状态
- suspend()方法用来使线程挂起，要通过 resume()方法使其重新启动

2、访问控制修饰符（新补充）

关于访问控制修饰符，在第一篇总结中已有详细的介绍。但最近在使用 String 类的一个方法 compareTo()的时候，对 private 修饰符有了新的理解。String 类的 compareTo 方法是用来比较两个字符串的字典序的，其源码如下：

```
public int compareTo(String anotherString) {

    int len1 = value.length;

    int len2 = anotherString.value.length;

    //重点是这里！！！
```

```
    int lim = Math.min(len1, len2);

    char v1[] = value;

    char v2[] = anotherString.value;

    //重点是这里！！！

    int k = 0;

    while (k < lim) {

        char c1 = v1[k];

        char c2 = v2[k];

        if (c1 != c2) {

            return c1 - c2;

        }

        k++;

    }

    return len1 - len2;

}
```

上面代码逻辑很好理解，我在看到它里面直接使用 anotherString.value 来获取 String 的字符数组的时候很奇怪，因为 value 是被定义为 private 的，只能在类的内部使用，不能在外部通过类对象.变量名的方式访问。我们平常都是通过 String 类的 toCharArray()方法来获取 String 的字符数组的，看到上面的这种使用方法，我赶紧在别的地方测试了一下，发现的确是不能直接通过 xx.value 的方法来获取字符数组。

正如前面所说的，value 是被定义为 private 的，只能在类的内部使用，不能在外部通过类对象.变量名的方式访问。因为 compareTo 方法就是 String 类的内部成员方法，compareTo 方法的参数传递的就是 String 对象过来，此时使用“类对象.变量名”的方式是在该类的内部使用，因此可以直接访问到该类的私有成员。自己再模仿 String 类来测试一下，发现果然如此。。

问题很细节，但是没有一下想通，说明还是对 private 的修饰符理解不够到位，前面自认为只要是 private 修饰的，就不能通过“类对象.变量名”的方式访问，其实还是需要看在哪里面使用。

3、线程同步的方法

当我们有多个线程要访问同一个变量或对象时，而这些线程中既有对改变量的读也有写操作时，就会导致变量值出现不可预知的情况。如下一个取钱和存钱的场景：

没有加入同步控制的情形：

```
public class BankCount {

    private int count = 0;

    //余额

    public void addMoney(int money){

        //存钱

        count += money;

        System.out.println(System.currentTimeMillis() + "存入: " + money);

        System.out.println("账户余额: " + count);

    }

    public void getMoney(int money){

        //取钱

        if(count - money < 0){

            System.out.println("余额不足");

            System.out.println("账户余额: " + count);

            return;

        }
```

```
        count -= money;
        System.out.println(System.currentTimeMillis() + "取出: " + money);
        System.out.println("账户余额: " + count);
    }
}
```

测试类:

```
public class BankTest {
    public static void main(String[] args) {
        final BankCount bankCount = new BankCount();
        new Thread(new Runnable() {
            //取钱线程
            @Override
                    public void run() {
                while(true){
                    bankCount.getMoney(200);
                    try {
                        Thread.sleep(1000);
                    }
                    catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
```

```
            }
        }
        ).start();
        new Thread(new Runnable() {
            //存钱线程
            @Override
                        public void run() {
                while(true){
                    bankCount.addMoney(200);
                    try {
                        Thread.sleep(1000);
                    }
                    catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
        ).start();
    }
}
```

部分打印结果如下：

```
余额不足

账户余额：0

1462265808958 存入：200

账户余额：200

1462265809959 存入：200

账户余额：200

1462265809959 取出：200

账户余额：200

1462265810959 取出：200

账户余额：200

1462265810959 存入：200

账户余额：200

1462265811959 存入：200

账户余额：200
```

可以看到，此时有两个线程共同使用操作了 bankCount 对象中的 count 变量，使得 count 变量结果不符合预期。因此需要进行同步控制，同步控制的方法有以下几种：

（1）使用 synchronized 关键字同步方法

每一个 Java 对象都有一个内置锁，使用 synchronized 关键字修饰的方法，会使用 Java 的内置锁作为锁对象，来保护该方法。每个线程在调用该方法前，都需要获得内置锁，如果该锁已被别的线程持有，当前线程就进入阻塞状态。

修改 BankCount 类中的两个方法，如下：

```
public synchronized void addMoney(int money){
```

```
    //存钱
    count += money;
    System.out.println(System.currentTimeMillis() + "存入：" + money);
    System.out.println("账户余额：" + count);
}
public synchronized void getMoney(int money){
    //取钱
    if(count - money < 0){
        System.out.println("余额不足");
        System.out.println("账户余额：" + count);
        return;
    }
    count -= money;
    System.out.println(System.currentTimeMillis() + "取出：" + money);
    System.out.println("账户余额：" + count);
}
```

运行测试打印如下结果：

```
余额不足
账户余额：0
1462266451171 存入：200
账户余额：200
1462266452171 取出：200
```

```
账户余额：0

1462266452171 存入：200

账户余额：200

1462266453171 存入：200

账户余额：400

1462266453171 取出：200

账户余额：200

1462266454171 存入：200

账户余额：400

1462266454171 取出：200

账户余额：200

1462266455171 取出：200

账户余额：0
```

可以看到，打印结果符合我们的预期。

另外，如果我们使用 synchronized 关键字来修饰 static 方法，此时调用该方法将会锁住整个类。（关于类锁、对象锁下面有介绍）

（2）使用 synchronzied 关键字同步代码块

使用 synchronized 关键字修饰的代码块，会使用对象的内置锁作为锁对象，实现代码块的同步。

改造 BankCount 类的两个方法：

```
public void addMoney(int money){

    //存钱
```

```
    synchronized(this){
        count += money;
        System.out.println(System.currentTimeMillis() + "存入: " + money);
        System.out.println("账户余额: " + count);
    }
}
public void getMoney(int money){
    //取钱
    synchronized(this){
        if(count - money < 0){
            System.out.println("余额不足");
            System.out.println("账户余额: " + count);
            return;
        }
        count -= money;
        System.out.println(System.currentTimeMillis() + "取出: " + money);
        System.out.println("账户余额: " + count);
    }
}
```

（注：这里改造后的两个方法中因为 synchronized 包含了方法体的整个代码语句，效率上与在方法名前加 synchronized 的第一种同步方法差不多，因为里面涉及到了打印 money 还是需要同步的字段，所以全部包含起来，仅仅是为了说明 synchronized 作用...）

打印结果：

```
余额不足

账户余额：0

1462277436178 存入：200

账户余额：200

1462277437192 存入：200

账户余额：400

1462277437192 取出：200

账户余额：200

1462277438207 取出：200

账户余额：0

1462277438207 存入：200

账户余额：200

1462277439222 存入：200

账户余额：400

1462277439222 取出：200

账户余额：200
```

可以看到，执行结果也符合我们的预期。

synchronized 同步方法和同步代码块的选择：

同步是一种比较消耗性能的操作，应该尽量减少同步的内容，因此尽量使用同步代码块的方式来进行同步操作，同步那些需要同步的语句（这些语句一般都访问了一些共享变量）。但是像我们上面举得这个例子，就不得不同步方法的整个代码块，因为方法中的代码每条语句都涉及了共享变量，因此此时就可以直接使用 synchronized 同步方法的方式。

（3）使用重入锁（ReentrantLock）实现线程同步

重入性：是指同一个线程多次试图获取它占有的锁，请求会成功，当释放锁的时候，直到重入次数为 0，锁才释放完毕。

ReentrantLock 是接口 Lock 的一个具体实现类，和 synchronized 关键字具有相同的功能，并具有更高级的一些功能。如下使用：

```
public class BankCount {

    private Lock lock = new ReentrantLock();

    //获取可重入锁

    private int count = 0;

    //余额

    public void addMoney(int money){

        //存钱

        lock.lock();

        try {

            count += money;

            System.out.println(System.currentTimeMillis() + "存入：" + money);

            System.out.println("账户余额：" + count);

        }

        finally{

            lock.unlock();

        }

    }

    public void getMoney(int money){

        //取钱
```

```
        lock.lock();
        try {
            if(count - money < 0){
                System.out.println("余额不足");
                System.out.println("账户余额: " + count);
                return;
            }
            count -= money;
            System.out.println(System.currentTimeMillis() + "取出: " + money);
            System.out.println("账户余额: " + count);
        }
        finally{
            lock.unlock();
        }
    }
}
```

部分打印结果：

```
1462282419217 存入：200
账户余额：200
1462282420217 取出：200
账户余额：0
1462282420217 存入：200
```

```
账户余额：200

1462282421217 存入：200

账户余额：400

1462282421217 取出：200

账户余额：200

1462282422217 存入：200

账户余额：400

1462282422217 取出：200

账户余额：200

1462282423217 取出：200

账户余额：0
```

同样结果符合预期，说明使用 ReentrantLock 也是可以实现同步效果的。使用 ReentrantLock 时，lock()和 unlock()需要成对出现，否则会出现死锁，一般 unlock 都是放在 finally 中执行。

synchronized 和 ReentrantLock 的区别和使用选择：

1、使用 synchronized 获得的锁存在一定缺陷：

不能中断一个正在试图获得锁的线程
试图获得锁时不能像 ReentrantLock 中的 trylock 那样设定超时时间，当一个线程获得了对象锁后，其他线程访问这个同步方法时，必须等待或阻塞，如果那个线程发生了死循环，对象锁就永远不会释放；
每个锁只有单一的条件，不像 condition 那样可以设置多个

2、尽管 synchronized 存在上述的一些缺陷，在选择上还是以 synchronized 优先：

如果 synchronized 关键字适合程序，尽量使用它，可以减少代码出错的几率和代码数量；（减少出错几率是因为在执行完 synchronized 包含完的最后一句语句后，锁会自动释放，不需要像 ReentrantLock 一样手动写 unlock 方法；）
如果特别需要 Lock/Condition 结构提供的独有特性时，才使用他们；（比如设定一个线程长时间不能获取锁时设定超时时间或自我中断等功能。）

许多情况下可以使用 java.util.concurrent 包中的一种机制，它会为你处理所有的加锁情况；（比如当我们在多线程环境下使用 HashMap 时，可以使用 ConcurrentHashMap 来处理多线程并发）。

下面两种同步方式都是直接针对共享变量来设置的：

（4）对共享变量使用 volatile 实现线程同步

- volatile 关键字为变量的访问提供了一种免锁机制
- 使用 volatile 修饰域相当于告诉虚拟机该域可能会被其他线程更新
- 因此每次使用该变量就要重新计算，直接从内存中获取，而不是使用寄存器中的值
- volatile 不会提供任何原子操作，它也不能用来修饰 final 类型的变量。

修改 BankCount 类如下：

```
public class BankCount {

    private volatile int count = 0;

    //余额

    public void addMoney(int money){

        //存钱

        count += money;

        System.out.println(System.currentTimeMillis() + "存入：" + money);

        System.out.println("账户余额：" + count);

    }

    public void getMoney(int money){

        //取钱

        if(count - money < 0){

            System.out.println("余额不足");
```

```
            System.out.println("账户余额: " + count);
            return;
        }
        count -= money;
        System.out.println(System.currentTimeMillis() + "取出: " + money);
        System.out.println("账户余额: " + count);
    }
}
```

部分打印结果:

```
余额不足
账户余额: 200
1462286786371 存入: 200
账户余额: 200
1462286787371 存入: 200
账户余额: 200
1462286787371 取出: 200
账户余额: 200
1462286788371 取出: 200
1462286788371 存入: 200
账户余额: 200
账户余额: 200
1462286789371 存入: 200
```

```
账户余额：200
```

可以看到，使用 volitale 修饰变量，并不能保证线程的同步。volitale 相当于一种“轻量级的 synchronized”，但是它不能代替 synchronized，volitale 的使用有较强的限制，它要求该变量状态真正独立于程序内其他内容时才能使用 volatile。volitle 的原理是每次线程要访问 volatile 修饰的变量时都是从内存中读取，而不是从缓存当中读取，以此来保证同步（这种原理方式正如上面例子看到的一样，多线程的条件下很多情况下还是会存在很大问题的）。因此，我们尽量不会去使用 volitale。

（5）ThreadLocal 实现同步局部变量

使用 ThreadLocal 管理变量，则每一个使用该变量的线程都获得该变量的副本，副本之间相互独立，这样每一个线程都可以随意修改自己的变量副本，而不会对其他线程产生影响。

ThreadLocal 的主要方法有：

- initialValue()：返回当前线程赋予当前线程拷贝的局部线程变量的初始值。一般在定义 ThreadLocal 类的时候会重写该方法，返回初始值；
- get()：返回当前线程拷贝的局部线程变量的值；
- set(T value)：为当前线程拷贝的局部线程变量设置一个特定的值；
- remove()：移除当前线程赋予局部线程变量的值

如下使用：

```
public class BankCount {

    private static ThreadLocal<Integer> count = new ThreadLocal<Integer>(){

        protected Integer initialValue() {

            return 0;

        }

        ;

    }
```

```
    ;

    //余额

    public void addMoney(int money){

        //存钱

        count.set(count.get() + money);

        System.out.println(System.currentTimeMillis() + "存入: " + money);

        System.out.println("账户余额: " + count.get());

    }

    public void getMoney(int money){

        //取钱

        if(count.get() - money < 0){

            System.out.println("余额不足");

            System.out.println("账户余额: " + count.get());

            return;

        }

        count.set(count.get() - money);

        System.out.println(System.currentTimeMillis() + "取出: " + money);

        System.out.println("账户余额: " + count.get());

    }

}
```

部分打印结果:

```
余额不足
```

```
1462289139008 存入：200

账户余额：0

账户余额：200

余额不足

账户余额：0

1462289140008 存入：200

账户余额：400

余额不足

账户余额：0

1462289141008 存入：200

账户余额：600

余额不足

账户余额：0
```

从打印结果可以看到，测试类中的两个线程分别拥有了一份 count 拷贝，即取钱线程和存钱线程都有一个 count 初始值为 0 的变量，因此可以一直存钱但是不能取钱。

ThreadLocal 使用时机：

由于 ThreadLocal 管理的局部变量对于每个线程都会产生一份单独的拷贝，因此 ThreadLocal 适合用来管理与线程相关的关联状态，典型的管理局部变量是 private static 类型的，比如用户 ID、事物 ID，我们的服务器应用框架对于每一个请求都是用一个单独的线程中处理，所以事物 ID 对每一个线程是唯一的，此时用 ThreadLocal 来管理这个事物 ID，就可以从每个线程中获取事物 ID 了。

ThreadLocal 和前面几种同步机制的比较：

- hreadLocal 和其它所有的同步机制都是为了解决多线程中的对同一变量的访问冲突，在普通的同步机制中，是通过对象加锁来实现多个线程对同一变量

的安全访问的。这时该变量是多个线程共享的，使用这种同步机制需要很细致地分析在什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放该对象的锁等等很多。所有这些都是因为多个线程共享了资源造成的。

- ThreadLocal 就从另一个角度来解决多线程的并发访问，ThreadLocal 会为每一个线程维护一个和该线程绑定的变量的副本，从而隔离了多个线程的数据，每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal 提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的整个变量封装进 ThreadLocal，或者把该对象的特定于线程的状态封装进 ThreadLocal。
- ThreadLocal 并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之间进行通信的有效方式；而 ThreadLocal 是隔离多个线程的数据共享，从根本上就不在多个线程之间共享资源（变量），这样当然不需要对多个线程进行同步了。所以，如果你需要进行多个线程之间进行通信，则使用同步机制；如果需要隔离多个线程之间的共享冲突，可以使用 ThreadLocal，这将极大地简化你的程序，使程序更加易读、简洁。

4、锁的等级：方法锁、对象锁、类锁

Java 中每个对象实例都可以作为一个实现同步的锁，也即对象锁（或内置锁），当使用 synchronized 修饰普通方法时，也叫方法锁（对于方法锁这个概念我觉得只是一种叫法，因为此时用来锁住方法的可能是对象锁也可能是类锁），当我们用 synchronized 修饰 static 方法时，此时的锁是类锁。

对象锁的实现方法：

- 用 synchronized 修饰普通方法（非 static）；
- 用 synchronized(this){...}的形式包括代码块；

上面两种方式获得的锁是同一个锁对象，即当前的实例对象锁。（当然，也可以使用其他传过来的实例对象作为锁对象），如下实例：

```
public class BankCount {

    public synchronized void addMoney(int money){
```

```
        //存钱
        synchronized(this){
            //同步代码块
            int i = 5;
            while(i-- > 0){
                System.out.println(Thread.currentThread().getName() + ">存入: " + mo
ney);
                try {
                    Thread.sleep(500);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    public synchronized void getMoney(int money){
        //取钱
        int i = 5;
        while(i-- > 0){
            System.out.println(Thread.currentThread().getName() + ">取钱: " + money)
;
            try {
                Thread.sleep(500);
```

```
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

测试类：

```
public class BankTest {
    public static void main(String[] args) {
        final BankCount bankCount = new BankCount();
        new Thread(new Runnable() {
            //取钱线程
            @Override
                        public void run() {
                bankCount.getMoney(200);
            }
        }
        ,"取钱线程").start();
        new Thread(new Runnable() {
            //存钱线程
            @Override
```

```
                    public void run() {
                bankCount.addMoney(200);
            }
        }
        ,"存钱线程").start();
    }
}
```

打印结果如下：

```
取钱线程>取钱：200
取钱线程>取钱：200
取钱线程>取钱：200
取钱线程>取钱：200
取钱线程>取钱：200
存钱线程>存入：200
存钱线程>存入：200
存钱线程>存入：200
存钱线程>存入：200
存钱线程>存入：200
```

打印结果表明，synchronized 修饰的普通方法和代码块获得的是同一把锁，才会使得一个线程执行一个线程等待的执行结果。

类锁的实现方法：

- 使用 synchronized 修饰 static 方法

- 使用 synchronized(类名.class){...}的形式包含代码块

因为 static 的方法是属于类的，因此 synchronized 修饰的 static 方法获取到的肯定是类锁，一个类可以有很多对象，但是这个类只会有一个.class 的二进制文件，因此这两种方式获得的也是同一种类锁。

如下修改一下上面代码的两个方法：

```
public void addMoney(int money){

    //存钱

    synchronized(BankCount.class){

        int i = 5;

        while(i-- > 0){

            System.out.println(Thread.currentThread().getName() + ">存入：" + money)
;

            try {

                Thread.sleep(500);

            }

            catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

public static synchronized void getMoney(int money){

    //取钱

    int i = 5;
```

```
    while(i-- > 0){
        System.out.println(Thread.currentThread().getName() + ">取钱：" + money);
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

打印结果和上面一样。说明这两种方式获得的锁是同一种类锁。
类锁和对象锁是两种不同的锁对象，如果将 addMoney 方法改为普通的对象锁方式，继续测试，可以看到打印结果是交替进行的。

注：

- 一个线程获得了对象锁或者类锁，其他线程还是可以访问其他非同步方法，获得了锁只是阻止了其他线程访问使用相同锁的方法、代码块；
- 一个获得了对象锁的线程，可以在该同步方法中继续去访问其他相同锁对象的同步方法，而不需要重新申请锁。

后面一篇，将总结线程池 ThreadPool、生产者消费者问题及实现、sleep 和 wait 方法区别。