Java 基础提升篇：synchronized 同步块和 volatile 同步变量

初遇

Java 语言包含两种内在的同步机制：同步块（或方法）和 volatile 变量。这两种机制的提出都是为了实现代码线程的安全性。其中 Volatile 变量的同步性较差（但有时它更简单并且开销更低），而且其使用也更容易出错。

synchronized 同步块

Java 中的同步块用 synchronized 标记。同步块在 Java 中是同步在某个对象上。所有同步在一个对象上的同步块在同时只能被一个线程进入并执行操作。所有其他等待进入该同步块的线程将被阻塞，直到执行该同步块中的线程退出。

有四种不同的同步块：

实例方法：一个实例一个线程。

静态方法：一个类只能由一个线程同时执行。

实例方法中的同步块

静态方法中的同步块

在多线程下最好是使用这种：

```
public class MyClass {
 public static void log2(String msg1, String msg2){
      synchronized(MyClass.class){
         log.writeln(msg1);
         log.writeln(msg2);
      }
   }
}
```

上述同步块都同步在不同对象上。实际需要哪种同步块视具体情况而定。

实例方法同步

下面是一个同步的实例方法：

```
public synchronized void add(int value){
    this.count += value;
}
```

注意在方法声明中同步（synchronized ）关键字。这告诉 Java 该方法是同步的。

Java 实例方法同步是同步在拥有该方法的对象上。这样，每个实例其方法同步都同步在不同的对象上，即该方法所属的实例。只有一个线程能够在实例方法同步块中运行。如果有多个实例存在，那么**一个线程一次可以在一个实例同步块中执行操作。**一个实例一个线程。

静态方法同步

静态方法同步和实例方法同步方法一样，也使用 synchronized 关键字。Java 静态方法同步如下示例：

```
public static synchronized void add(int value){
 count += value;
}
```

同样，这里 synchronized 关键字告诉 Java 这个方法是同步的。

静态方法的同步是指同步在该方法所在的类对象上。因为在 Java 虚拟机中一个类只能对应一个类对象，所以同时只允许一个线程执行同一个类中的静态同步方法。

对于不同类中的静态同步方法，一个线程可以执行每个类中的静态同步方法而无需等待。**不管类中的哪个静态同步方法被调用，一个类只能由一个线程同时执行。**

实例方法中的同步块

有时你不需要同步整个方法，而是同步方法中的一部分。Java 可以对方法的一部分进行同步。在非同步的 Java 方法中的同步块的例子如下所示：

```
public void add(int value){
    synchronized(this){
       this.count += value;
    }
}
```

示例使用 Java 同步块构造器来标记一块代码是同步的。该代码在执行时和同步方法一样。

注意 **Java 同步块构造器用括号将对象括起来。**在上例中，使用了“this”，即为调用 add 方法的实例本身。**在同步构造器中用括号括起来的对象叫做监视器对象。**上述代码使用监视

器对象同步，同步实例方法使用调用方法本身的实例作为监视器对象。

一次只有一个线程能够在同步于同一个监视器对象的 Java 方法内执行。

下面两个例子都同步他们所调用的实例对象上，因此他们在同步的执行效果上是等效的。

```
public class MyClass {
   public synchronized void log1(String msg1, String msg2){
      log.writeln(msg1);
      log.writeln(msg2);
   }
   public void log2(String msg1, String msg2){
      synchronized(this){
         log.writeln(msg1);
         log.writeln(msg2);
      }
   }
}
```

在上例中，每次只有一个线程能够在两个同步块中任意一个方法内执行。

如果第二个同步块不是同步在 this 实例对象上，那么两个方法可以被线程同时执行。

静态方法中的同步块

和上面类似，下面是两个静态方法同步的例子。这些方法同步在该方法所属的类对象上。

```
public class MyClass {
    public static synchronized void log1(String msg1, String msg2){
       log.writeln(msg1);
       log.writeln(msg2);
    }
    public static void log2(String msg1, String msg2){
       synchronized(MyClass.class){
          log.writeln(msg1);
          log.writeln(msg2);
       }
    }
}
```

这两个方法不允许同时被线程访问。

如果第二个同步块不是同步在 MyClass.class 这个对象上，那么这两个方法可以同时被线程访问。

Java 同步实例

在下面例子中，启动了两个线程，都调用 Counter 类同一个实例的 add 方法。因为同步在该方法所属的实例上，所以同时只能有一个线程访问该方法。

```
public class Counter{
    long count = 0;
    public synchronized void add(long value){
      this.count += value;
    }
  }
  public class CounterThread extends Thread{
    protected Counter counter = null;
    public CounterThread(Counter counter){
       this.counter = counter;
    }
    public void run() {
   for(int i=0; i<10; i++){
         counter.add(i);
       }
    }
  }
  public class Example {
    public static void main(String[] args){
      Counter counter = new Counter();
      Thread  threadA = new CounterThread(counter);
      Thread  threadB = new CounterThread(counter);
      threadA.start();
      threadB.start();
    }
}
```

创建了两个线程。他们的构造器引用同一个 Counter 实例。Counter.add 方法是同步在实例上，是因为 add 方法是实例方法并且被标记上 synchronized 关键字。因此每次只允许一个线程调用该方法。另外一个线程必须要等到第一个线程退出 add()方法时，才能继续执行方法。

如果两个线程引用了两个不同的 Counter 实例，那么他们可以同时调用 add()方法。这些方法调用了不同的对象，因此这些方法也就同步在不同的对象上。这些方法调用将不会被阻塞。如下面这个例子所示：

```
public class Example {
   public static void main(String[] args){
     Counter counterA = new Counter();
     Counter counterB = new Counter();
     Thread  threadA = new CounterThread(counterA);
     Thread  threadB = new CounterThread(counterB);
     threadA.start();
     threadB.start();
   }
}
```

注意这两个线程，threadA 和 threadB，不再引用同一个 counter 实例。CounterA 和 counterB 的 add 方法同步在他们所属的对象上。调用 counterA 的 add 方法将不会阻塞调用 counterB 的 add 方法。

volatile 同步变量

线程为了提高效率，将某成员变量(如 A)拷贝了一份（如 B），线程中对 A 的访问其实访问的是 B。只在某些动作时才进行 A 和 B 的同步。因此存在 A 和 B 不一致的情况。volatile 就是用来避免这种情况的。volatile 告诉 jvm，它所修饰的变量不保留拷贝，直接访问主内存中的（也就是上面说的 A）。

理解 volatile 特性的一个好方法是：把对 volatile 变量的单个读/写，看成是使用同一个监视器锁对这些单个读/写操作做了同步。下面我们通过具体的示例来说明。

```
class VolatileFeaturesExample {
    volatile long vl = 0L;  //使用 volatile 声明 64 位的 long 型变量
    public void set(long l) {
        vl = l;   //单个 volatile 变量的写
    }
    public void getAndIncrement () {
        vl++;    //复合（多个）volatile 变量的读/写
    }
    public long get() {
        return vl;   //单个 volatile 变量的读
    }
}
```

假设有多个线程分别调用上面程序的三个方法，这个程序在语意上和下面程序等价：

```
class VolatileFeaturesExample {
```

```
    long vl = 0L;                  // 64 位的 long 型普通变量
    public synchronized void set(long l) { //对单个的普通 变量的写用同一个监视器同步
        vl = l;
    }
    public void getAndIncrement () { //普通方法调用
        long temp = get();           //调用已同步的读方法
        temp += 1L;                  //普通写操作
        set(temp);                   //调用已同步的写方法
    }
    public synchronized long get() {
    //对单个的普通变量的读用同一个监视器同步
        return vl;
    }
}
```

如上面示例程序所示，对一个 volatile 变量的单个读/写操作，与对一个普通变量的读/写操作使用同一个监视器锁来同步，它们之间的执行效果相同。

监视器锁的 happens-before 规则保证释放监视器和获取监视器的两个线程之间的内存可见性，这意味着对一个 volatile 变量的读，总是能看到（任意线程）对这个 volatile 变量最后的写入。

监视器锁的语义决定了临界区代码的执行具有原子性。这意味着即使是 64 位的 long 型和 double 型变量，**只要它是 volatile 变量，对该变量的读写就将具有原子性。如果是多个 volatile 操作或类似于 volatile++这种复合操作，这些操作整体上不具有原子性。**

简而言之，volatile 变量自身具有下列特性：

- **可见性**：对一个 volatile 变量的读，总是能看到（任意线程）对这个 volatile 变量最后的写入。
- **原子性**：对任意单个 volatile 变量的读/写具有原子性，但类似于 volatile++这种复合操作不具有原子性。