# HenCoder Plus 第 21 课 讲义

# Java 多线程和线程同步

## 进程和线程

- 进程和线程
  - 操作系统中运行多个软件
  - 一个运行中的软件可能包含多个进程
  - 一个运行中的进程可能包含多个线程
- CPU 线程和操作系统线程
  - CPU 线程
    - 多核 CPU 的每个核各自独立运行，因此每个核一个线程
    - 「四核八线程」：CPU 硬件方在硬件级别对 CPU 进行了一核多线程的支持（本质上依然是每个核一个线程）
  - 操作系统线程：操作系统利用时间分片的方式，把 CPU 的运行拆分给多条运行逻辑，即为操作系统的线程
  - 单核 CPU 也可以运行多线程操作系统
- 线程是什么：按代码顺序执行下来，执行完毕就结束的一条线
  - UI 线程为什么不会结束？因为它在初始化完毕后会执行死循环，循环的内容是刷新界面

## 多线程的使用

- Thread 和 Runnable
  - Thread

```
Thread thread = new Thread() {
    @Override
    public void run() {
        System.out.println("Thread
started!");
    }
};
thread.start();
```

  - Runnable

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread with
Runnable started!");
    }
};
Thread thread = new Thread(runnable);
thread.start();
```

- ThreadFactory

```
ThreadFactory factory = new ThreadFactory() {
    int count = 0;

    @Override
    public Thread newThread(Runnable r) {
        count ++;
        return new Thread(r, "Thread-" + count);
    }
};

Runnable runnable = new Runnable() {
    @Override
    public void run() {

 System.out.println(Thread.currentThread().getNa
me() + " started!");
    }
};

Thread thread = factory.newThread(runnable);
thread.start();
Thread thread1 = factory.newThread(runnable);
```

```
thread1.start();
```

- Executor 和线程池
  - 常用：`newCachedThreadPool()`

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Thread with
Runnable started!");
    }
};

Executor executor =
Executors.newCachedThreadPool();
executor.execute(runnable);
executor.execute(runnable);
executor.execute(runnable);
```

  - 短时批量处理：`newFixedThreadPool()`

```
ExecutorService executor =
Executors.newFixedThreadPool(20);
for (Bitmap bitmap : bitmaps) {

 executor.execute(bitmapProcessor(bitmap));
}
executor.shutdown();
```

- Callable 和 Future

```
Callable<String> callable = new Callable<String>
() {
    @Override
    public String call() {
```

```
            try {
                Thread.sleep(1500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Done!";
        }
};

ExecutorService executor =
Executors.newCachedThreadPool();
Future<String> future =
executor.submit(callable);
try {
    String result = future.get();
    System.out.println("result: " + result);
} catch (InterruptedException |
ExecutionException e) {
    e.printStackTrace();
}
```

## 线程同步与线程安全

- synchronized
  - synchronized 方法

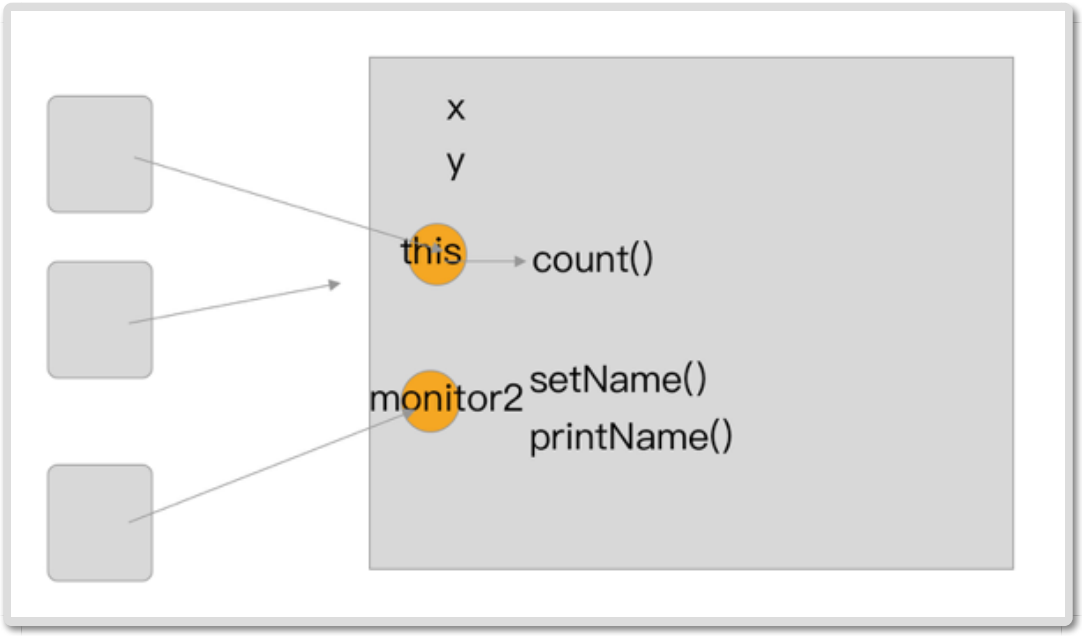
```
private synchronized void count(int newValue)
{
    x = newValue;
    y = newValue;
    if (x != y) {
        System.out.println("x: " + x + ", y:"
+ y);
    }
}
```

- synchronized 代码块

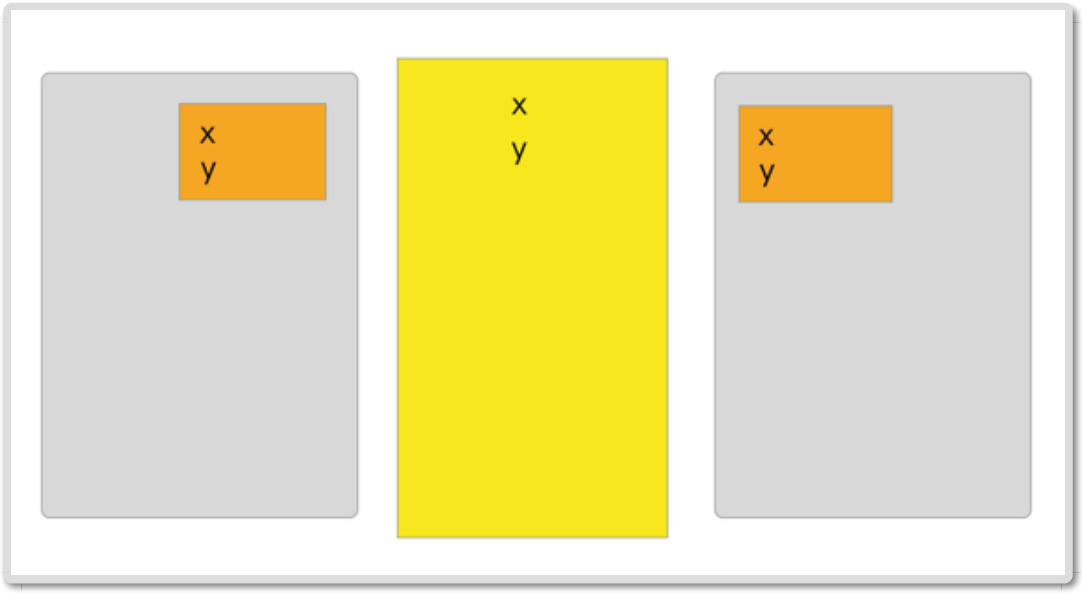
```
private void count(int newValue) {
    synchronized (this) {
        x = newValue;
        y = newValue;
        if (x != y) {
            System.out.println("x: " + x + ",
y:" + y);
        }
    }
}
```

```
synchronized (monitor1) {
    synchronized (monitor2) {
        name = x + "-" + y;
    }
}
```

- synchronized 的本质
  - 保证方法内部或代码块内部资源（数据）的**互斥访问**。即同一时间、由同一个 Monitor 监视的代码，最多只能有一个线程在访问




- 保证线程之间对监视资源的**数据同步**。即，任何线程在获取到 Monitor 后的第一时间，会先将共享内存中的数据复制到自己的缓存中；任何线程在释放 Monitor 的第一时间，会先将缓存中的数据复制到共享内存中。




- volatile
  - 保证加了 volatile 关键字的字段的操作具有**同步性**，以及对 `long` 和 `double` 的操作的原子性（long double 原子性这个简单说一下就行）。因此 volatile 可以看做是简化版的 synchronized。
  - volatile 只对基本类型（byte、char、short、int、long、float、double、boolean）的赋值操作和对象的引用赋值操作有效，你要修改 User.name 是不能保证同步的。
  - volatile 依然解决不了 ++ 的原子性问题。
- `java.util.concurrent.atomic` 包：
  - 下面有 `AtomicInteger` `AtomicBoolean` 等类，作用和 volatile 基本一致，可以看做是通用版的 volatile。

```
AtomicInteger atomicInteger = new
AtomicInteger(0);

...

atomicInteger.getAndIncrement();
```

- Lock / ReentrantReadWriteLock
  - 同样是「加锁」机制。但使用方式更灵活，同时也更麻烦一些。

```
Lock lock = new ReentrantLock();

...

lock.lock();
try {
    x++;
} finally {
    lock.unlock();
}
```

> finally 的作用：保证在方法提前结束或出现 Exception 的时候，依然能正常释放锁。

  - 一般并不会只是使用 `Lock`，而是会使用更复杂的锁，例如 `ReadWriteLock`：

```
ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();
Lock readLock = lock.readLock();
Lock writeLock = lock.writeLock();

private int x = 0;

private void count() {
    writeLock.lock();
```

```
        try {
            x++;
        } finally {
            writeLock.unlock();
        }
    }

    private void print(int time) {
        readLock.lock();
        try {
            for (int i = 0; i < time; i++) {
                System.out.print(x + " ");
            }
            System.out.println();
        } finally {
            readLock.unlock();
        }
    }
```

- 线程安全问题的本质：

  在多个线程访问共同的资源时，在某**一个线程**对资源进行**写操作的中途**（写入已经开始，但还没结束），**其他线程**对这个**写了一半的资源**进行了**读操作**，或者基于这个**写了一半的资源**进行了**写操作**，导致出现**数据错误**。

- 锁机制的本质：

  通过对共享资源进行访问限制，让同一时间只有一个线程可以访问资源，保证了数据的准确性。

- 不论是线程安全问题，还是针对线程安全问题所衍生出的锁机制，它们的核心都在于共享的**资源**，而不是某个方法或者某几行代码。