Java 多线程编程核心技术：线程安全和锁Synchronized概念

一、进程与线程的概念

（1）在传统的操作系统中，程序并不能独立运行，作为资源分配和独立运行的基本单位都是进程。

在未配置 OS 的系统中，程序的执行方式是顺序执行，即必须在一个程序执行完后，才允许另一个程序执行；在多道程序环境下，则允许多个程序并发执行。程序的这两种执行方式间有着显著的不同。也正是程序并发执行时的这种特征，才导致了在操作系统中引入进程的概念。

自从在 20 世纪 60 年代人们提出了进程的概念后，在 OS 中一直都是以进程作为能拥有资源和独立运行的基本单位的。直到 20 世纪 80 年代中期，人们又提出了比进程更小的能独立运行的基本单位——线程(Threads)，试图用它来提高系统内程序并发执行的程度，从而可进一步提高系统的吞吐量。特别是在进入 20 世纪 90 年代后，多处理机系统得到迅速发展，线程能比进程更好地提高程序的并行执行程度，充分地发挥多处理机的优越性，因而在近几年所推出的多处理机 OS 中也都引入了线程，以改善 OS 的性能。

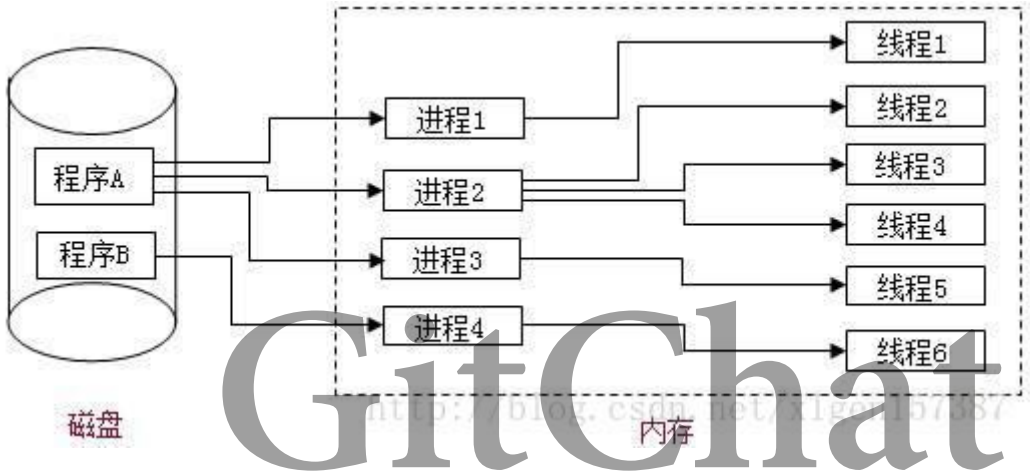
——以上摘自《计算机操作系统-汤小丹等编著-3 版》

通过上述的大致了解，基本知道线程和进程是干什么的了，那么我们下边给进程和线程总结一下概念：

（3）**进程**（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

（4）**线程**，有时被称为**轻量级进程**(Lightweight Process，LWP），是程序执行流的最小单元。线程是程序中一个单一的顺序控制流程。进程内一个相对独立的、可调度的执行单元，是系统独立调度和分派CPU的基本单位指运行中的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为多线程。

（5）进程和线程的关系：



（6）线程和进程各自有什么区别和优劣

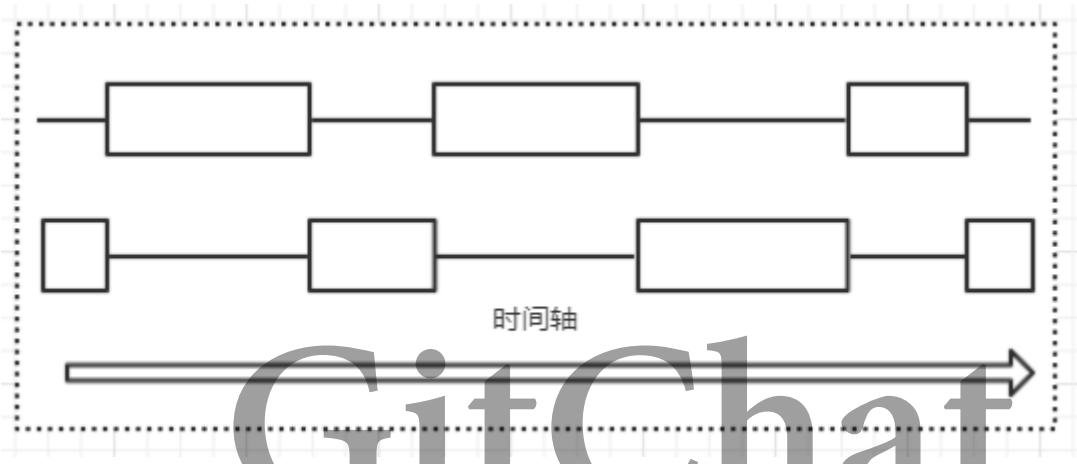
- 进程是资源分配的最小单位，线程是程序执行的最小单位。
- 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多，线程的上下文切换的性能消耗要小于进程

对于一次方法的调用来说，同步方法调用一旦开始，就必须等待该方法的调用返回，后续的方法才可以继续执行；异步的话，方法调用一旦开始，就可以立即返回，调用者可以执行后续的方法，这里的异步方法通常会在另一个线程里真实的执行，而不会妨碍当前线程的执行。

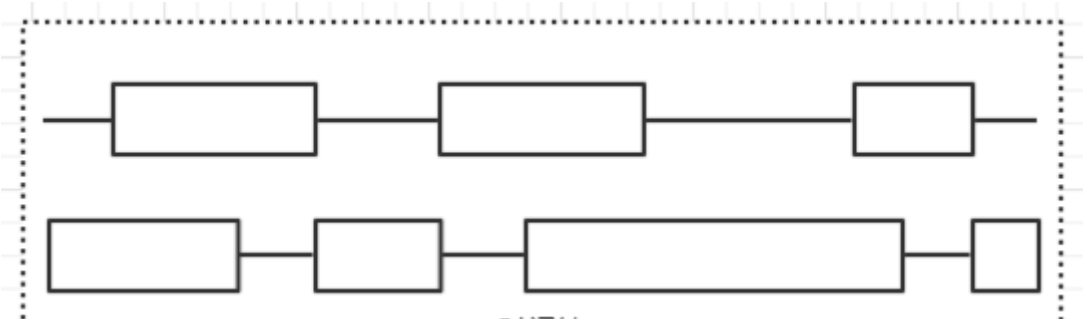
三、并行与并发

并发和并行是两个相对容易比较混淆的概念。他都可以表示在同一时间范围内有两个或多个任务同时在执行，但其在任务调度的时候还是有区别的，首先看下图：

并发任务执行过程：



并行任务执行过程：



（1）继承Thread，重写run（）方法

```
public class MyThread extends Thread {

    @Override
    public void run() {
        while (true) {

System.out.println(this.currentThread().getName());
        }
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); //线程启动的正确方式
    }
}
```

输出结果：

```
Thread-0
Thread-0
Thread-0
...
```


另外，要明白启动线程的是start（）方法而不是run（）方法，如果用run（）方法，那么他就是一个普通的方法执行了。

（2）实现Runable接口

```
public class MyRunnable implements Runnable {

    @Override
```

Thread类本身实现了Runnable接口，并且持有run方法，但Thread类的run方法主体是空的，Thread类的run方法通常是由子类的run方法重写。

五、线程安全

线程安全概念：当多个线程访问某一个类（对象或方法）时，这个类始终能表现出正确的行为，那么这个类（对象或方法）就是线程安全的。

线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问，直到该线程读取完，释放了锁，其他线程才可使用。这样的话就不会出现数据不一致或者数据被污染的情况。 线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据以至于所得到的数据是脏数据。这里的加锁机制常见的如：synchronized

六、synchronized修饰符

（1）synchronized：可以在任意对象及方法上加锁，而加锁的这段代码称为**互斥区**或**临界区**。

（2）不使用synchronized实例（代码A）：

```
public class MyThread extends Thread {

    private int count = 5;

    @Override
    public void run() {
        count--;
        System.out.println(this.currentThread().getName() + "
count:" + count);
```

```
        }
    }
```

输出的一种结果如下：

```
thread3 count:2
thread4 count:1
thread1 count:2
thread2 count:3
thread5 count:0
```

可以看到，上述的结果是不正确的，这是因为，多个线程同时操作 run（） 方法，对count进行修改，进而造成错误。

（3）使用synchronized实例（代码B）：

```
public class MyThread extends Thread {

    private int count = 5;

    @Override
    public synchronized void run() {
        count--;
        System.out.println(this.currentThread().getName() + "
count:" + count);
    }

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        Thread thread1 = new Thread(myThread, "thread1");
        Thread thread2 = new Thread(myThread, "thread2");
        Thread thread3 = new Thread(myThread, "thread3");
        Thread thread4 = new Thread(myThread, "thread4");
        Thread thread5 = new Thread(myThread, "thread5");
```

```
thread5 count:1
thread4 count:0
```

可以看出代码A和代码B的区别就是在 run（） 方法上加上了synchronized修饰。

说明如下：

当多个线程访问MyThread 的run方法的时候，如果使用了synchronized修饰，那个多线程就会以排队的方式进行处理（这里排队是按照CPU分配的先后顺序而定的），一个线程想要执行synchronized修饰的方法里的代码，首先是尝试获得锁，如果拿到锁，执行synchronized代码体的内容，如果拿不到锁的话，这个线程就会不断的尝试获得这把锁，直到拿到为止，而且多个线程同时去竞争这把锁，也就是会出现锁竞争的问题。

七、一个对象有一把锁！多个线程多个锁！

何为，一个对象一把锁，多个线程多个锁！首先看一下下边的实例代码（代码C）：

```
public class MultiThread {

    private int num = 200;

    public synchronized void printNum(String threadName,
String tag) {
        if (tag.equals("a")) {
            num = num - 100;
            System.out.println(threadName + " tag a,set num
over!");
        } else {
            num = num - 200;
            System.out.println(threadName + " tag b,set num
over!");
        }
        System.out.println(threadName + " tag " + tag + "
```

```
            Thread.sleep(5000);
            System.out.println("等待5秒，确保thread1已经执行完毕！");

            new Thread(new Runnable() {
                public void run() {
                    multiThread2.printNum("thread2", "b");
                }
            }).start();
        }
    }
```

输出结果：

```
thread1 tag a,set num over!
thread1 tag a, num = 100
等待5秒，确保thread1已经执行完毕!
thread2 tag b,set num over!
thread2 tag b, num = 0
```

可以看出，有两个对象： multiThread1 和 multiThread2 ，如果多个对象使用同一把锁的话，那么上述执行的结果就应该是： thread2 tag b, num = -100 ，因此，是每一个对象拥有该对象的锁的。

关键字 synchronized 取得的锁都是对象锁，而不是把一段代码或方法当做锁，所以上述实例代码C中哪个线程先执行 synchronized 关键字的方法，那个线程就持有该方法所属对象的锁，两个对象，线程获得的就是两个不同对象的不同的锁，他们互补影响的。

那么，我们在正常的场景的时候，肯定是有一种情况的就是，所有的对象会对一个变量count进行操作，那么如何实现哪？很简单就是加static，我们知道，用static修改的方法或者变量，在该类的所有对象是具有相同的引用的，这样的话，无论实例化多少对象，调用的都是一个方法，代码如下（代码D）：

```
            }
            System.out.println(threadName + " tag " + tag + ",
num = " + num);
        }

        public static void main(String[] args) throws
InterruptedException {
            final MultiThread multiThread1 = new MultiThread();
            final MultiThread multiThread2 = new MultiThread();

            new Thread(new Runnable() {
                public void run() {
                    multiThread1.printNum("thread1", "a");
                }
            }).start();

            Thread.sleep(5000);
            System.out.println("等待5秒，确保thread1已经执行完毕！");

            new Thread(new Runnable() {
                public void run() {
                    multiThread2.printNum("thread2", "b");
                }
            }).start();
        }
    }
```


输出结果：

```
thread1 tag a,set num over!
thread1 tag a, num = 100
等待5秒，确保thread1已经执行完毕!
thread2 tag b,set num over!
thread2 tag b, num = -100
```

同步的目的就是为了线程的安全，其实对于线程的安全，需要满足两个最基本的特性：原子性和可见性;

（2）异步：asynchronized

异步的概念就是独立，相互之间不受到任何制约，两者之间没有任何关系，这里的异步可以理解为多个线程之间不会竞争共享资源。

（3）示例代码：

```
public class MyObject {

    public void method() {
        System.out.println(Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        final MyObject myObject = new MyObject();

        Thread t1 = new Thread(new Runnable() {
            public void run() {
                myObject.method();
            }
        }, "t1");

        Thread t2 = new Thread(new Runnable() {
            public void run() {
                myObject.method();
            }
        }, "t2");

        t1.start();
        t2.start();
    }
}
```