Java - 多线程

作者：someone33881
原文链接：https://ld246.com/article/1545316775019
来源网站：链滴

    本文主要是记录在学习
ava多线程过程中的一些知识点备忘！

知识点规整：

一、java多线程基础概念

● 进程 VS 线程：基本概念，什么是多线程?

● 如何使用多线程：继承Thread类，实现Runnable接口，使用线程池

● 实例变量和线程安全：不共享数据的情况，共享数据的情况

● 常用方法：currentThread(),getId(),getName(),getPriority(),isAlive(),sleep(long millis),interrupt(),interrupted()和isInterrupted(),setName(String name),isDaemon(),setDaemon(boolean on),join(),yield(),setPriority(int newPriority)

● 如何停止一个线程：使用interrupt()方法；使用return停止线程；已弃用的[（1）stop()不安全；（）stop（Throwable obj）不安全；（3）suspend()+resume()死锁];

● 线程的优先级

● java多线程分类：多线程分类，如何设置守护?

二、synchronized关键字 同步方法

● 变量安全性：如果两个线程同时操作对象中的实例变量，则会出现“非线程安全”，解决办法就是方法前加上synchronized关键词即可

● 多个对象多个锁

● synchronized方法与锁对象：synchronzied取得的锁都是对象锁，而不是一段代码或方法当做锁

● 脏读：发生脏读的情况是在读取实例变量时，此值已经被其他线程更改过

● synchronized锁重入：“可重入锁”概念是“自己可以再次获取自己的内部锁”，另外可重入锁也持在父子类继承的环境中

● 同步不具有继承性

三、synchronized关键字 同步语句块

● synchronized声明方法的缺点

● synchronized(this)同步代码块的使用

● synchronized(object)代码块间使用

● synchronized代码块间的同步性：其他线程执行对象synchronized同步方法和synchronized(this)码块时呈现同步效果；如果两个线程使用了同一个“对象监视器”，运行结果同步，否则不同步

● 静态同步synchronized方法与synchronized(class)代码块：synchronized关键字加到static方法和snchronized(class)代码块上都是给Class类上锁，而synchronized关键字加到非static静态方法上都给对象上锁

● 数据类型String的常量池属性:在JVM中具有String常量池缓存的功能

20181222-2

1、synchronized声明方法的缺点

答曰：使用synchronized关键字声明方法是有很大弊端的，比如两个线程A和B，一个线程A调用同步法获得锁，那么另一个线程B就需要等待A执行完，但是如果说A执行的是一个很费时间的任务的话这就会很耗时

如下是一个暴露synchronized声明方法的缺点示例，以及如何通过synchronzied同步语句块去解决样的问题

```
public class Task {

    private String getData1;
    private String getData2;

    public synchronized void doLongTimeTask() {
        try {
            System.out.println("begin task");
            Thread.sleep(3000);
            getData1 = "长时间处理任务后从远程返回的值1 threadName="
                    + Thread.currentThread().getName();
            getData2 = "长时间处理任务后从远程返回的值2 threadName="
                    + Thread.currentThread().getName();
            System.out.println(getData1);
            System.out.println(getData2);
            System.out.println("end task");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

public class CommonUtils {

    public static long beginTime1;
    public static long endTime1;
```

```
    public static long beginTime2;
    public static long endTime2;
}


public class Aokay1Thread extends Thread {
    private Task task;
    public Aokay1Thread(Task task) {
        super();
        this.task = task;
    }
    @Override
    public void run() {
        super.run();
        CommonUtils.beginTime1 = System.currentTimeMillis();
        task.doLongTimeTask();
        CommonUtils.endTime1 = System.currentTimeMillis();
    }
}


public class Aokay2Thread extends Thread {
    private Task task;
    public Aokay2Thread(Task task) {
        super();
        this.task = task;
    }
    @Override
    public void run() {
        super.run();
        CommonUtils.beginTime2 = System.currentTimeMillis();
        task.doLongTimeTask();
        CommonUtils.endTime2 = System.currentTimeMillis();
    }
}


public class Run {

    public static void main(String[] args) {
        Task task = new Task();

        Aokay1Thread at1 = new Aokay1Thread(task);
        at1.start();

        Aokay2Thread at2 = new Aokay2Thread(task);
        at2.start();

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
        long beginTime = CommonUtils.beginTime1;
        if (CommonUtils.beginTime2 < CommonUtils.beginTime1) {
            beginTime = CommonUtils.beginTime2;
        }

        long endTime = CommonUtils.endTime1;
        if (CommonUtils.endTime2 > CommonUtils.endTime1) {
            endTime = CommonUtils.endTime2;
        }

        System.out.println("耗时：" + ((endTime - beginTime) / 1000));
    }
}
```

从运行时间上看（耗时约6秒，即第二个线程等待第一个线程执行完之后才执行），synchronized方问题很明显；可以使用synchronized同步块来解决这个问题，需要注意的是synchronized同步块的用方式，使用不当并不会带来效率上的提升

2、synchronized(this)同步代码块的使用

将上述的中的Task类代码进行如下修改：

```
public class Task {

    private String getData1;
    private String getData2;

    public void doLongTimeTask() {
        try {
            System.out.println("begin task");
            Thread.sleep(3000);

            String privateGetData1 = "长时间处理任务后从远程返回的值1 threadName="
                    + Thread.currentThread().getName();
            String privateGetData2 = "长时间处理任务后从远程返回的值2 threadName="
                    + Thread.currentThread().getName();

            //这个地方是核心的不同的地方！！！
            synchronized (this) {
                getData1 = privateGetData1;
                getData2 = privateGetData2;
            }

            System.out.println(getData1);
            System.out.println(getData2);
            System.out.println("end task");
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

从运行时间上看（耗时约3秒，即第二个线程并不需要等待第一个线程执行完，就可以执行）

=》当一个线程访问一个对象的synchronized同步代码块时，另一个线程仍然可以访问对象非synchronized同步代码块

3、synchronized(object)代码块间使用

也即，两个或多个线程是否使用同一个“对象监视器”，如果是使用同一个对象监视器则结果是同步，否则运行结果就不是同步的了！代码示例如下：

```
public class AokayObject {
}

public class Service {

    public void testMethod1(AokayObject object) {
        synchronized (object) {
            try {
                System.out.println("testMethod1 ___getLock time="
                        + System.currentTimeMillis() + " run ThreadName="
                        + Thread.currentThread().getName());
                Thread.sleep(2000);
                System.out.println("testMethod1 releaseLock time="
                        + System.currentTimeMillis() + " run ThreadName="
                        + Thread.currentThread().getName());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Aokay1Thread extends Thread {

    private Service service;
    private AokayObject object;

    public Aokay1Thread(Service service, AokayObject object) {
        super();
        this.service = service;
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        service.testMethod1(object);
    }
}

public class Aokay2Thread extends Thread {
    private Service service;
    private AokayObject object;

    public Aokay2Thread(Service service, AokayObject object) {
        super();
```

```
        this.service = service;
        this.object = object;
    }

    @Override
    public void run() {
        super.run();
        service.testMethod1(object);
    }
}

public class Run1 {

    public static void main(String[] args) {
        Service service = new Service();
        //下面两个线程均使用同一个的该“对象监视器” =>结果是同步的
        AokayObject object = new AokayObject();

        Aokay1Thread a = new Aokay1Thread(service, object);
        a.setName("a");
        a.start();

        Aokay2Thread b = new Aokay2Thread(service, object);
        b.setName("b");
        b.start();
    }
}

public class Run2 {

    public static void main(String[] args) {
        Service service = new Service();
        //下面两个线程各自使用的不同的对象的“对象监视器” =>结果是不同步的
        AokayObject object1 = new AokayObject();
        AokayObject object2 = new AokayObject();

        Aokay1Thread a = new Aokay1Thread(service, object1);
        a.setName("a");
        a.start();

        Aokay2Thread b = new Aokay2Thread(service, object2);
        b.setName("b");
        b.start();
    }
}
```

4、synchronized代码块间的同步性

当一个对象访问synchronized(this)代码块时，其他线程对同一个对象中所有其他synchronized(this)代码块代码的访问都将被阻塞，这说明“synchronized(this)代码块使用的是同一个对象监视器”，即synchronized(this)代码块也是锁定当前对象的

=>

- 其他线程执行对象中synchronized同步方法和synchronized(this)代码块时呈现同步效果
- 如果两个线程使用了同一个“对象监视器”则运行结果同步，否则不同步

5、静态同步synchronized方法与synchronized(class)代码块

synchronized关键字加到static静态方法和synchronized(class)代码块都是给Class类上锁的，而synchronized关键字加到非static静态方法上则是给对象上锁的

```
public class Service {

    public static void printA() {
        synchronized (Service.class) {
            try {
                System.out.println(
                        "线程名称为：" + Thread.currentThread().getName() + "在" + System.currentTimeMillis() + "进入printA");
                Thread.sleep(3000);//该时间差用以标记是否同步
                System.out.println(
                        "线程名称为：" + Thread.currentThread().getName() + "在" + System.currentTimeMillis() + "离开printA");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    synchronized public static void printB() {
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在" + System.currentTimeMillis() + "进入printB");
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在" + System.currentTimeMillis() + "离开printB");
    }

    synchronized public void printC() {
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在" + System.currentTimeMillis() + "进入printC");
        System.out.println("线程名称为：" + Thread.currentThread().getName() + "在" + System.currentTimeMillis() + "离开printC");
    }
}


public class Aokay1Thread extends Thread {
    private Service service;
    public Aokay1Thread(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printA();
    }
}
```

```
public class Aokay2Thread extends Thread {
    private Service service;
    public Aokay2Thread(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printB();
    }
}

public class Aokay3Thread extends Thread {
    private Service service;
    public Aokay3Thread(Service service) {
        super();
        this.service = service;
    }
    @Override
    public void run() {
        service.printC();
    }
}

public class Run {
    public static void main(String[] args) {
        Service service = new Service();

        Aokay1Thread a = new Aokay1Thread(service);
        a.setName("A");
        a.start();

        Aokay2Thread b = new Aokay2Thread(service);
        b.setName("B");
        b.start();

        Aokay3Thread c = new Aokay3Thread(service);
        c.setName("C");
        c.start();
    }
}
```

=》运行是第一条先打印的A，然后是两条C,然后再是一条A和两条B

=》静态同步synchronized方法与synchronized(class)代码块持有的锁一样，都是Class锁，Class锁对象的所有实例均起作用！synchronized关键字加到非static静态方法上持有的是对象锁

=》线程A,B和线程C持有的锁不一样，所以A和B运行同步，但是和C运行不同步

6、数据类型String的常量池属性

在JVM中具有String常量池缓存的功能

```
String s1 = "a";
String s2 = "a";
```

```
System.out.println(s1==s2);//true
```

上面打印结果是true，为什么?

答曰：字符串常量池中的字符串只存在一份！即执行完第一行代码后，常量池中已存在“a”，那么s不会再在常量池中申请新的空间，而是直接把已经存在的字符串内存地址返回给s2

=>因此数据类型String的常量池属性，所有synchronized(string)在使用时某些情况会出现一些问题比如两个线程运行

```
synchronized("abc"){}

//和

synchronized("abc"){}
```

修饰的方法时，这两个线程就会持有相同的锁，导致某一时刻只有一个线程能够运行！所有尽量不要用synchronized(string)，而是使用synchronized(object)

20181222-1

1、synchronized关键字

synchronized关键字也被称为重量级锁，在JDK1.6之后进行了主要包括“减少获得锁和释放锁带来性能消耗而引入的偏向锁和轻量级锁”，以及其他各种优化之后变得在某些情况下并不是重了

2、变量安全性

“非线程安全”问题存在于“实例变量”中，如果是方法内部的私有变量，则不存在“非线程安全问”，所以得到结果也就是“线程安全”的了

如果两个线程同时操作对象中的实例变量，则会出现“非线程安全”，解决办法就是在方法前加上synhronized关键字即可!

（可见20181220-8示例）

3、多个对象多个锁

```
public class HasSelfPrivateNum {

    private int num = 0;

    synchronized public void addI(String username) {
        try {
            if (username.equals("a")) {
                num = 100;
                System.out.println("a set over!");
                //如果去掉hread.sleep(2000)，那么运行结果就会显示为同步的效果
                Thread.sleep(2000);
            } else {
                num = 200;
                System.out.println("b set over!");
            }
            System.out.println(username + " num=" + num);
        } catch (InterruptedException e) {
```

```
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}


public class Aokay1Thread extends Thread {
    //定义了一个属于Aokay1Thread的私有对象HasSelfPrivateNum
    private HasSelfPrivateNum numRef;

    public Aokay1Thread(HasSelfPrivateNum numRef) {
        super();
        this.numRef = numRef;
    }

    @Override
    public void run() {
        super.run();
        numRef.addI("a");
    }

}


public class Aokay2Thread extends Thread {
    //定义了一个属于Aokay2Thread的私有对象HasSelfPrivateNum
    private HasSelfPrivateNum numRef;

    public Aokay2Thread(HasSelfPrivateNum numRef) {
        super();
        this.numRef = numRef;
    }

    @Override
    public void run() {
        super.run();
        numRef.addI("b");
    }

}


public class Run {

    public static void main(String[] args) {
        //创建了两个HasSelfPrivateNum对象
        HasSelfPrivateNum numRef1 = new HasSelfPrivateNum();
        HasSelfPrivateNum numRef2 = new HasSelfPrivateNum();

        Aokay1Thread at1 = new Aokay1Thread(numRef1);
        //开启线程at1且在该线程的run中的锁是对象numRef1的对象锁
        at1.start();
```

```
        Aokay2Thread at2 = new Aokay2Thread(numRef2);
        //开启线程at2且在该线程的run中的锁是对象numRef2的对象锁
        at2.start();

    }

}
```

执行结果是：

a set over

b set over

b num=200

a num=100

两个线程Aokay1Thread和Aokay2Thread分别访问的是同一个类HasSelfPrivateNum的不同实例的同名称的同步方法，但是效果却是异步执行。为什么会是这样的呢？？

因为synchronized取得的锁都是对象锁即例子中HasSelfPrivateNum类某个实例对象的锁，而不是一段代码或方法当做锁！在实际运行中，哪个线程先执行带synchronized关键字的方法，则哪个线程持有该方法所属对象的锁Lock,那么其他线程只能呈等待状态，前提是多个线程访问的是同一个对象本例子很显然上面是两个对象）

在上述例子中，是创建了两个HasSelfPrivateNum类对象，因此就会产生两个锁！当Aokay1Thread程run方法中的引用numRef执行到addI方法中的Thread.sleep(2000)语句是，Aokay2Thread就会机执行，所以才会导致上述执行结果（注意：a num=100会在停顿了2秒才输出）

4、synchronzied方法与锁对象

由3可知，synchronized取得的锁是对象锁，而不是把一段代码或方法当做锁；如果多个线程访问的同一个对象，哪个线程先执行到带synchronized关键字的方法，则哪个线程就持有该方法所属对象的Lock，其他的线程只能呈等待状态！如果多个线程访问的是多个对象则不一定吗，因为多个对象会产多个锁！

那么，当多个线程访问的是同一个对象中的非synchronized类型的方法会出现什么效果呢?

答曰：会异步调用非synchronized类型方法，解决办法就是在该方法前加上synchronzied关键字

5、脏读

脏读发生的情况就是在读取实例变量时，该值已经被其他线程更改过了！

```
public class PublicVar {

    public String username = "A";
    public String password = "AA";

    synchronized public void setValue(String username, String password) {
        try {
            this.username = username;
            Thread.sleep(5000);
```

```
            this.password = password;

            System.out.println("setValue method thread name="
                    + Thread.currentThread().getName() + " username="
                    + username + " password=" + password);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    //该方法前加上synchronized关键字就同步了
    public void getValue() {
        System.out.println("getValue method thread name="
                + Thread.currentThread().getName() + " username=" + username
                + " password=" + password);
    }
}

public class Aokay1Thread extends Thread {

    private PublicVar publicVar;

    public Aokay1Thread(PublicVar publicVar) {
        super();
        this.publicVar = publicVar;
    }

    @Override
    public void run() {
        super.run();
        publicVar.setValue("B", "BB");
    }
}

public class Run {

    public static void main(String[] args) {
        try {
            PublicVar publicVarRef = new PublicVar();
            Aokay1Thread at1 = new Aokay1Thread(publicVarRef);
            at1.start();

            Thread.sleep(200);//打印结果受此值大小影响

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

运行结果是：

setValue method thread name=main username=B password=AA

getValue method thread name=Thread-0 username=B password=BB

解决办法：在getValue方法前加上synchronized关键字即可

运行结果是：

setValue method thread name=Thread-0 username=B password=BB

getValue method thread name=main username=B password=BB

6、synchronized锁重入

可重入锁：自己可以再次获取自己的内部锁！

比如一个线程获得了某个对象的锁，此时这个对象锁还没释放，当其再次想要获取这个对象的锁的时也还是可以获取的，如果不可锁重入的话，就会造成死锁

```
public class Service {

    synchronized public void service1() {
        System.out.println("service1");
        service2();
    }

    synchronized public void service2() {
        System.out.println("service2");
        service3();
    }

    synchronized public void service3() {
        System.out.println("service3");
    }

}


public class AokayThread extends Thread {
    @Override
    public void run() {
        Service service = new Service();
        service.service1();
    }

}

public class Run {
    public static void main(String[] args) {
        AokayThread at = new AokayThread();
        at.start();
    }
}
```

运行结果：

service1

service2

service3

此外，可重入锁也支持在父子类继承的环境中

```
public class Main {

    public int i = 10;

    synchronized public void operateIMainMethod() {
        try {
            i--;
            System.out.println("main print i=" + i);
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

public class Sub extends Main {

    synchronized public void operateISubMethod() {
        try {
            while (i > 0) {
                i--;
                System.out.println("sub print i=" + i);
                Thread.sleep(100);
                this.operateIMainMethod();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class AokayThread extends Thread {
    @Override
    public void run() {
        Sub sub = new Sub();
        sub.operateISubMethod();
    }
}

public class Run {
    public static void main(String[] args) {
        AokayThread at = new AokayThread();
        at.start();
    }
}
```

运行结果：sub print i=9 main print i=8 ..... sub print i=1 main print i=0

=>当存在父子类继承关系时，子类是完全可以通过“可重入锁”调用父类的同步方法!

另外，当出现异常时，其所持有的锁会自动释放

7、同步不具有继承性

如果父类有一个带有synchronized关键字的方法，子类继承并重写了这个方法

但是同步不能继承，所以还是需要在子类方法中添加synchronized关键字

20181221

1、多线中的常用方法

- currentThread()：返回对当前正在执行的线程对象的引用
- getId()：返回此线程的标识符
- getName()：返回此线程的名称
- getPriority()：返回此线程的优先级
- isAlive()：测试这个线程是否还处于活动状态，所谓活动状态就是指线程已经启动且尚未终止，线处于正在运行或准备运行的状态
- sleep(long millis)：使当前正在执行的线程以指定的毫秒数“休眠”（暂时停止执行），具体取决系统定时器和调度程序的精度和准确性
- interrupt()：中断这个线程
- interrupted()和isInterrupted()：？？？interrupted为测试当前线程是否已经是中断状态，执行后有将状态标志清除为false的功能；isInterrupted为测试线程Thread对相关是否已经是中断状态，但清除状态标志
- setName(String name)：将此线程的名称更改为参数name的值
- isDaemon()：测试这个线程是否是守护线程
- setDaemon(boolean on)：将此线程标记为daemon线程或用户线程
- join()：在很多情况下，主线程生成并启动了子线程，如果子线程里要进行大量的耗时的运算，主程往往将于子线程之前结束，但是如果主线程处理完其他的事务后，需要用到子线程的处理结果，也是“主线程需要等待子线程执行完成之后再结束，这个时候就需要用到join()方法了”；；join的作用是“等待该线程终止”，这里的该线程指的就是主线程等待子线程的终止，也就是在子线程调用了joi方法后面的代码，只有等到子线程结束了才能执行
- yield()：作用是放弃当前的CPU资源，将它让给其他的任务去占用CPU时间！注意：放弃的时间不定，可能一会就会重新获得CPU时间片
- setPriority(int newPriority)：更改此线程的优先级

2、interrupt()和return两种方式停止线程

？？？代码编写？？

3、线程的优先级

每个线程都具有各自的优先级，线程的优先级可以在程序中表明该线程的重要性，如果有很多线程处就绪状态，系统会根据优先级来决定首先使哪个线程进入运行状态；但这并不意味着低优先级的线程不到运行，而只是它运行的几率比较小，如垃圾回收机制线程的优先级就比较低（所以很多垃圾得不及时的回收处理）

线程优先级具有继承特性：比如A线程启动B线程，则B线程的优先级和A是一样的

线程优先级具有随机特性：即线程优先级高的不一定每一次都先执行完

Thread类中包含的成员变量代表了线程的某些优先级，如**Thread.MIN_PRIORITY（常数1），Thread.NORM_PRIORITY（常数5），Thread.MAX_PRIORITY（常数10）**；其中每个线程的优先级都在**Thread.MIN_PRIORITY（常数1）到Thread.MAX_PRIORITY（常数10）之间，在默认情况下优先级都是Thread.NORM_PRIORITY（常数5）**

4、java多线程分类

- 用户线程：运行在前台，执行具体的任务，如程序的主线程、连接网络的子线程等都是用户线程
- 守护线程：

（1）运行在后台，为其他前台线程服务，也即守护线程是JVM中非守护线程的“佣人”

（2）特点：一旦所有的用户线程均结束运行，则守护线程会随着JVM一起结束工作

（3）应用：数据库连接池中的检测线程，JVM虚拟机启动后的检测线程

（4）最常见的守护线程：垃圾回收线程

- 如何设置守护线程？通过调用Thread类的setDaemon(true)方法设置当前线程为守护线程，需要注的是：（a）setDaemon(true)必须在start()方法前执行，否则会抛出IllegalThreadStateException常；（b）在守护线程中产生的新线程也是守护线程；（c）不是所有的任务都可以分配给守护线程来行的，比如读写操作或者计算逻辑
- ？？守护线程相关的示例代码

20181220

1、进程 VS 线程

进程是程序的一次执行过程，是系统运行程序的基本单位（因此，进程是动态的）；系统中运行一个序即是一个进程从创建、运行到消亡的过程

线程是比进程更小的执行单位；一个进程在其执行过程中可以产生多个线程；与进程不同的是，同类多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程或者多个线程之间切换工作，负担要比进程小得多，因此线程也被称为轻量级进程

多线程：多个线程同时运行（多核CPU）或交替运行（单核CPU，即顺序地交替执行）

多线程的必要性：开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力和性能

为什么提倡多线程而不是多进程？：因为线程间的切换和调度的成本要远远小于进程间的切换和调度

2、同步 VS 异步

- 同步和异步均用来形容一次方法调用
- 同步：同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为
- 异步：异步方法调用更像是一个消息传递，一旦开始，方法调用就会立即返回，调用者可以继续后的操作
- 关于异步的经典常用实现方式：使用消息队列；在不使用消息队列时，用户的请求数据是直接写入据库的，因此在高并发情况下数据库压力剧增，导致响应速度变慢；而在使用消息队列之后，用户请

数据发送给消息队列之后立即返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数库！

● 由于消息队列服务器处理速度快于数据库，且消息队列比数据库有更好的伸缩性，因此响应速度会幅改善！

3、并发Concurrency VS 并行Paralleism

并发与并行均可以表示两个或多个任务一起执行，但是偏重点不同：并行是真正意义上的的“同时执”，而并发对于单个CPU是多线程交替运行（并发），对于多个CPU是可以同时运行（并行）；即多时的并发=并行

4、高并发HC(High Concurrency)

高并发是互联网分布式系统架构设计中必须考虑的一个因素，通常是指设计保证系统能够同时并行处很多请求；高并发的一些常用指标有：响应时间ResponseTime、吞吐量Throughput、每秒查询率QS（Query Per Second）、并发用户数等

5、临界区

临界区即是用来表示一种公共资源或者说是共享数据，可以被多个线程使用；但是，每一次只能有一线程可以使用它，一旦临界区资源被占用，其他线程想要使用这个资源，就必须等待；在并行程序中临界区资源是保护的对象

6、阻塞 VS 非阻塞

非阻塞是指在不能立刻得到结果之前，该函数不会阻塞当前线程，而是会立刻返回；阻塞与之相反、

7、创建多线程的方式

（1）继承Thread类 【该方式实际开发过程中不使用】

```
public class AokayThread extends Thread {
  @Override
  public void run(){
    //super.run();
    System.out.println("This
 is AokayThread");
  }
}

public class Aokay {
  public static void main(String[] args){
    AokayThread at = new AokayThread();
    at.start();
    System.out.println("Main主线程运行结束！！！");
  }
}
```

=>定义一个Thread的继承类AokayThread=>重写run()方法=>在另一个线程如主线程Main中new个该继承类，并使用start()方法开启该线程（执行该线程run方法中的内容）

=>线程AokayThread在主线程Main中是一个子任务，且CPU是以不确定的方式或者说是随机的时间调用该线程中的run方法的

（2）实现Runnable接口

相比较于(1)更加推荐实现Runnable接口方式开发多线程，这是因为java继承是单继承但是是可以实多个接口的

```
public class AokayRunnable implements Runnable {
  @Override
  public void run(){
    System.out.println("This is AokayRunnable");
  }
}

public class Aokay {
  public static void main(String[] args) {
    Runnable ar = new AokayRunnale();
    Thread t = new Thread(ar);
    t.start();
    System.out.println("Main主线程运行结束！！！");
  }
}
```

=>定义一个Runnable接口的实现类AokayRunnable=>重写run()方法=>在另一个线程如主线程Mai中首先new一个该实现类，然后再用该实现类new一个Thread类出来，最后使用Thread对象的start(方法开启该线程（执行该线程run方法中的内容）

（3）使用线程池

相比较于（1）和（2），使用线程池是最为推荐的一种方式，在阿里巴巴java开发手册中就明确强调“线程资源必须通过线程池提供，不允许在应用中自行显示创建线程”

具体示例代码将在线程池部分详细介绍

8、实例变量和线程安全

线程类中的实例变量针对其他线程可以有共享和不共享之分，例子如下：

● 不共享数据的情况：多个线程之间不共享变量，线程安全的情况

```
public class AokayThread extends Thread {

  priavte int count = 5;

  public AokayThread(String name){
    super();
    this.setName(name);
  }

  @Override
  public void run(){
    super.run();
    while(count > 0){
      count--;
      System.out.println("由：" + AokayThread.currentThread().getName() + "计算，count=" +
ount);
    }
```

```
  }

  public static void main(String[] args){
    //区别主要是此处构造了三个Thread子类AokayThread的对象
    AokayThread a = new AokayThread("A");
    AokayThread b = new AokayThread("B");
    AokayThread c = new AokayThread("C");

    //直接用各个类AokayThrea的对象启动各个线程
    a.start();
    b.start();
    c.start();
  }
}
```

根据运行结果，是每个线程均是有一个属于自己的实例变量count，它们之间互不影响！

- 共享数据的情况：多个线程之间共享变量，线程不安全的情况

```
public class Aokay2Thread extends Thread{

  private int count = 5;

  @Override
  public void run(){
    super.run();
    //while(count > 0){
      count--;
      System.out.println("由：" + Aokay2Thread.currentThread().getName() + "计算，count=" +
count);
    //}
  }

  public static void main(String[] args){
    //区别主要是此处仅构造了一个Thread子类Aokay2Thread的对象
    Aokay2Thread a2t = new Aokay2Thread();
    //再用Aokay2Thread对象a2t去构造五个Thread类对象
    Thread a = new Thread(a2t,"A");
    Thread b = new Thread(a2t,"B");
    Thread c = new Thread(a2t,"C");
    Thread d = new Thread(a2t,"D");
    Thread e = new Thread(a2t,"E");

    //然后用各个Thread类对象去启动各个线程
    a.start();
    b.start();
    c.start();
    d.start();
    e.start();
  }
}
```

根据运行结果，发现这里面可能会出现了错误，明明想要的是依次递减的结果，为什么呢？这是因为

大多数jvm中,count--分为以下三个步骤：（1）取得原有count值；（2）计算count-1；（3）对cout进行赋值

==》因此，多个线程同时访问就会出现问题（？？具体如何分析一步步解释运行结果，可根据具体果进行猜测逻辑？？）

==》如何解决共享数据的线程安全问题？一是利用synchronized关键字（保证任意时刻只能有一个程执行该方法），二是利用AtomicInteger(即JUC中的Atomic原子类)！！注意，不能用volatile关键，因为volatile关键字不能保证复合操作的原子性。