Java 多线程总结

一、 实现线程的方式及其常用方法与属性

1.1 进程与线程的概念及线程的有点

进程可以说是操作系统的基础，是程序的一次运行，进程是操作系统进行资源分配和调度的最小单元。

线程可以理解为进程的一个或多个子任务，如果一个进程只有一个线程，可以理解为单任务进程，单任务的特点就行排队执行，也就是同步。使用多线程的目的就是在线程安全的情况下进行异步执行，尽可能的提高 CPU 及系统资源的利用率，这也是其优点。

1.2 实现线程的两种方式

Java 实现多线程常用的两种方式有继承 Thread 类、实现 Runnable 接口，实现 Runnable 接口相对来说具有优势，突破了 Java 单继承的局限性，维持了程序的健壮性，尤其是在多个线程需要造作同一资源时，实现接口的方式是首选。在将多线程交由线程池管理的情况下也必须是实现接口的方式。

实现多线程还有其他方式，给自己留个疑问后续在深究一下。

1.3 Java 多线程中常用方法

currentThread():指明代码段正在被哪个线程调用；

isAlive():判断当前线程是否处于活动状态，线程处于就绪状态或运行状态为活动状态；

sleep(long millisecond):指定毫秒数让当前正在执行的线程暂停执行，当线程持有锁的情况下，执行此方法可达到暂停执行作用但不释放锁，因此在有锁的情况下慎用此方法；

停止线程的方法：

1. 使用退出标志，当线程执行完 run()方法中程序时线程终止。
2. 使用 stop()方法强行终止线程，不推荐使用，容易出现脏数据。
3. 使用 interrupt()方法中断线程。

使用退出标志即为常用的 while（flag）{}死循环，当 flag 变为 false 时，程序正常执行完毕，即线程终止。

使用 stop()方法可以达到退出线程的效果，由于在调用此方法时会释放锁，这就有可能使得其他线程拿到脏数据，造成数据不同步，因此此方式已过时不推荐使用。

interrupt()方法配合抛异常或 return，都能达到终止线程得效果。当检测到线程处于中断状态时，抛出异常或 return 即可。在睡眠状态调用此方法中断线程会

抛出异常。给出检测线程中断状态得两个接口方法：

this.interrupted():测试当前线程是否处于中断状态，当前线程时指运行此方法的线程；

this.isInterrupted():测试线程是否已中断，同样用 this 指定时与上一个方法意义不同，用线程对象指定时意义相同，且此方法具有清除线程中断状态的作用，当线程调用 interrupt()方法后处于中断状态时，调用此方法可激活。

线程暂停及恢复：

suspend()：线程暂停

resume()：恢复已暂停线程

此二方法已过时不推荐使用，由于 suspend()具有独占的特性，当拥有公共资源时调用此方法不释放会造成后续线程无限时间等待，例如在 synchronized 代码块内调用此方法时并不会释放锁；使用此方法还易造成数据不同步。

yield():让出 CPU 使其重新调度

1.4 线程的常用属性

线程的优先级：

Java 中线程的优先级分为 10 个等级，1-10，通常使用 3 个常量来设置线程的 priority 属性，三个常量分别为：

MIN_PRIORITY=1

NORM_PRIORITY=5

MAX_PRIORITY=10

线程的优先级具有继承性，比如说 A 线程中启动 B 线程，那么 B 线程的优先级与 A 线程的优先级是一样的。

线程的优先级具有一定的规则性，并不是说优先级高的线程就一定会首先被执行，且线程优先级高的线程也并不一定是先执行完，也就是说 CPU 只是尽量将执行资源让给优先级比较高的线程。

守护线程：

守护线程顾名思义可理解为陪伴线程，当进程中所有非守护线程都执行完毕了，则守护线程自动销毁。典型的守护线程有垃圾回收线程。

线程对象 thread 通过调用 setDaemon(true)设置当前线程为守护线程。

二、 对象及变量的并发访问

2.1 synchronized 关键字

多个线程共同访问 1 个对象中的实例变量就有可能造成非线程安全，为了解决此问题引入 synchronized 关键字，此关键字可作用在变量、方法、代码块。

该关键字取得的锁都是对象的锁，而不是把一段代码或方法当作锁，哪个线程执行带此关键字的方法或代码块，或访问带此关键字的变量，哪个线程就持有该方法、代码块、变量所属的对象锁。

只有共享资源的读写访问才需要同步化，如果不是共享资源没有同步的必要。

关键字 synchronized 具有锁重入的特性，在一个 synchronized 方法或代码块的内部调用本类的其他 synchronized 方法或代码块时是永远可以得到的。

出现异常时，锁自动释放。

锁不具备继承性，例如父类 A 拥有同步方法 a，其子类 B 重写方法 a，但在方法声明时未添加 synchronized 关键字，则 B 类中的方法 a 并不具有同步性。

同步方法与同步代码块的差异性：

同步方法是对当前对象进行加锁，而同步代码块是对任一对象进行加锁，同步方法会使方法内所有操作流程进行排队机制，排队就会效率降低，而同步代码块只针对涉及到线程安全的地方进行加锁，减少互斥访问的代码块，从而在保证线程安全的前提下尽可能的提升程序运行效率。

静态同步 synchronized 方法：

静态同步方法是对当前的*.java 文件对应的类进行加锁，在同一个类中既有静态同步方法，又有非静态同步方法，其分别持有的是不同的锁，非静态同步方法持有的锁是对象的锁。

Synchronized(class)代码块的作用与 synchronized static 方法的作用是一样的，都是锁在*.java 文件上。

多线程的死锁：

当有不同的线程都等待在根本不可能释放的锁上时就会造成死锁，因此在设计同步访问时必须要避免此问题。

锁对象发生改变不会影响同步效果，只要对象不对，即使对象的属性发生改变，运行的结果还是同步的。

2.2 volatile 关键字

引入线程堆栈、公共堆栈的概念，JVM 为提高程序运行效率，在线程运行时，会将程序片段加入到当前工作的工作内存中即为线程的私有堆栈，运行期间只从私有堆栈中读取数据，当多个线程访问公共资源时，每个线程将公共资源引入到自己线程的私有堆栈中，当线程执行完毕后将值同步到公共堆栈中，这就会造成公共资源值不同步的结果，引入 volatile 关键字强制使线程每次从公共堆栈中读取共享资源值，此关键字增加了共享资源在多个变量之间的可见性。但是此关键字的缺点是不支持原子性。

volatile 与 synchronized 的比较：

1. volatile 是线程的轻量级实现，性能要高于 synchronized 关键字；volatile 只能修饰变量，而 synchronized 可修饰变量、方法、代码块，随着 JDK 新版本的发布 synchronized 关键字的效率在逐步提高。
2. 多线程访问 volatile 不会发生阻塞，而 synchronized 会发生阻塞，这正是其效率高的原因，也是其不支持原子性的根源。
3. volatile 能保证数据的可见性，但不能保证原子性；而 synchronized 既能保证原子性，也能通过锁机制间接保证数据的可见性。

线程安全主要包含原子性及可见性两个方面，synchronized 还包含有互斥性。

volatile 关键字使用场景，当实例变量发生变化时，并且多个线程需要获得最新的值使用，此时声明带有此关键字的变量，当有 synchronized 关键字出现时 volatile 关键字是多余的。

三、 线程通信

首先说为什么要进行线程间的通信，线程是进程中子任务，多个线程之间彼此互相独立，通过线程之间的通信增加其交互性，在提高 CPU 的同时，还能够对多个线程进行有效的把控与监督。

3.1 等待/通知机制。

等待通知的经典案例就是生产者消费者模型

Java 中用 wait()方法使当前线程进入等待状态，并且在 wait()所在代码行处停止，直到接到通知或被中断为止，调用此方法前线程必须获得该对象的对象锁，因此只能在同步方法或同步代码块中调用此方法，在执行 wait()方法后释放锁。

Java 中用 notify()方法实现通知，调用前线程同样需要获得对象锁，该方法用来通知那些可能等待在该对象的对象锁上的线程，如果有多个线程则由线程规划器挑选其中一个呈 wait 状态的线程。带参数的 wait(long)方法的功能是等待某一时间内是否有线程对锁进行唤醒，如果超过这个时间则自动唤醒。执行 notify()方法之后，当前线程不会马上释放该对象锁，要等到执行 notify()方法的线程将程序执行完，也就是说退出 synchronized 代码块之后，当前线程才会释放锁。notify()方法可以使等待队列中的其中一个线程唤醒，也就是进入可执行状态，notifyAll()方法，使等待在某一对象锁的线程全部唤醒进入可执行状态。

中断呈 wait 状态的线程会抛出异常，即当线程调用 wait()后未被唤醒时调用 interrupt()方法会抛出线程中断异常 InterruptedException。

生产者消费者模型中，一生产一消费的模型可正常执行，多消费者多生产者时容易出现假死，所谓假死就是所有线程进入到 wait 状态，原因就是模型采用 notify()方法唤醒某一个等待在锁上的线程，而唤醒的线程有可能是同类，也是就说生产者唤醒的可能仍是生产者，这就导致进入假死状态，这也是 notify()方法的弊端，在这里 notifyAll()方法可避免此假死状态，因为其唤醒了所有等待在相同对象锁上的线程，包括同类以及异类。虽然 notifyAll()方法解决了此问题，但在效率上不可忽视，线程切换的开销不可忽略，后面我们会提到只唤醒异类线程的问题。

3.2 通过管道实现线程间的通信

Java 提供一种特殊的流——管道流(pipeStream)来实现线程间的通信，JDK 中提供 4 个类可以实现线程间的通信：

PipeInputStream 和 PipeOutputStream

PipedReader 和 PipedWriter

流操作与常用 IO 流无差别，只需要将输出流与输入流建立连接即可，如 inputStream.connect(outputStream)，或者 outputStream.connect(inputStream)都可以，字符管道流与此类似

3.3 联合线程的使用

使用 join()方法来实现线程的联合,例如在线程 b 中调用 a.join()方法,则 b 线程必须等待线程 a 执行完毕后才销毁。

使用场景就是母线程需要子线程执行完毕时才结束。

使用联合线程可有效的控制已知线程的执行顺序，联合线程具有使线程排队的功能，类似同步的运行效果，但与 synchronized 有本质的区别，join()方法在内部调用 wait()方法进行等待而 synchronized 是使用对象监视器的原理，中断正在联合中的线程会抛出异常，如上述例子中，线程 a 未执行完毕，此时 b 线程调用 interrupt()方法会抛出异常。

join(long)方法设置等待时间，超过设置时间母线程继续执行。

join(long)与 sleep(long)区别：

此二方法在某些情况的使用上可达到相同的效果，主要区别来自其同步的原理不同，join(long)方法内部采用的 wait(剩余时间)，当调用此方法时就会释放当前对象的锁，而 sleep(long)方法不释放锁。

3.4 ThreadLocal 类的使用

多个线程共享一个变量值可以使用 public static 变量的形式，ThreadLocal 可以实现多个线程共用一个 ThreadLocal 对象但每一个线程都有自己的共享变量。可以理解为 ThreadLocal 对象是一个线程仓库，每个线程需要放入自己的共享变量时，仓库为其分配一个独立的车间，各车间之间互不影响，这是 ThreadLocal 的隔离性。

例如：threadLocal 是一个 ThreadLocal<T>对象

线程 A 第一次调用 threadLocal.get() 时为空，线程 A 可通过 threadLocal.set(object)进行仓储，此时线程 B 第一次调用 threadLocal.get()时也为空，因为 A、B 线程分配了不同的车间，此时 B 也可进行仓储，而两线程仓储之后在分别进行取值也都不影响。

也可通过继承 InheritableThreadLocal 类并重写其初始化值得方法，使仓库的每个车间都不为空但也都仍然彼此独立。也可以重写其 childValue(Object parentValue)方法对其仓储的值进行修改。

四、 Lock 的使用

首先说一下为什么已经有 synchronized 关键字还要引入 Lock 接口这种神奇的东西呢，记得上面提到的假死，提到 notify/notifyAll 唤醒所有等待在相同锁上的线程，如果是唤醒同类线程，那么无疑多了一次线程切换增加了系统开销，而 Lock 接口的出现，可以手动获取和释放指定锁，比如我们生产者消费者模型中，声明有生产锁和消费锁两种锁，每生产完一个物品时唤醒消费锁，每消费完一个物品时唤醒生产锁，保证系统切换的线程是异类线程，从而提高系统性能。Lock 接口有两个实现类。

4.1 ReentrantLock

声明锁：
Lock lock=new ReentrantLock();
获取锁：
lock.lock();
释放锁：
lock.unlock();
其中获取锁与释放锁之间的代码区即为同步区。
使用 Condition 实现等待通知机制：
Condition condition=lock.newCondition();
等待：
condition.await();
通知：
condition.siganl()/signalAll();
同样执行等待通知的操作都必须获得锁，也就是执行 lock.lock()方法。
公平锁与非公平锁：
Lock 锁分为公平锁与非公平锁，公平锁的意思就是 CPU 根据线程进入就绪状态的顺序调度线程，反之为非公平锁。
公平锁声明方式：
Lock lock=new ReentrantLock(true);//false 表示非公平锁
Lock 锁常用的接口方法：
int getHoldCount();//返回等待在此锁上的线程数
int getQueueLength();//返回正在等待获取此锁的线程估计数
int getWaitQueueLength(Condition condition);//返回等待与此锁相关的给定条件的线程估计数
boolean hasQueueThread(Thread thread);//查询指定线程是否正在等待获取此锁
boolean hasQueueThreads();//查询是否有线程正在等待获取此锁
boolean hasWaiters(Condition condition);//查询是否有线程正在等待与此锁有关的 condition 条件
boolean isFair();//判断是否为公平锁
boolean isHeldByCurrentLock();//查询当前线程是否获取了此锁定
boolean isLock();//查询是否有线程持有此锁
void lockInterruptibly();//如果当前线程未被中断则获取锁，如果已经中断则抛出异常
boolean tryLock();//如果当前锁未被其他线程保持则由当前线程保持并返回 true，否则返回 false
Boolean tryLock(long timeout,TimeUnit unit);//timeout 时间长度 ，unit 指定 timeout 类型 TimeUnit.SECONDS 等等，表示如果在指定时间内获取到锁就返回 true，否则返回 false
void awaitUninterruptibly();//通过 condition.awaitUninterruptibly()调用，造成当前线程一直处于等待状态，直到 condition 条件被唤醒，在等待过程中如果线程被中断不会抛出异常，这是与线程直接调用 interrupt()方法的区别

boolean awaitUntil(Date deadline);//指定 condition 条件等待到某一时刻，但可通过 signal()方法提前唤醒

4.2 ReenTrantReadWriteLock

ReentrantLock 的锁具有强互斥作用，就是同一时间内只有一个线程可以执行同步区代码，ReenTrantReadWriteLock 的出现改善了此效率低下的问题，此锁称作读写锁：

Lock lock=new ReenTrantReadWriteLock();

读锁获取与释放：

lock.readLock().lock();

lock.readLock().unlock();

写锁获取与释放：

lock.writeLock().lock();

lock.writeLock().unlock();

其中读与读之间不互斥，读与写互斥，写与写互斥

有关 synchronized 实现同步的地方 lock 接口都可以代替实现，且 lock 有一些更为方便的接口方法，而在并发中大量的类使用 lock 作为同步的处理方式。

五、 定时器 Timer

为什么把定时器 Timer 归类的线程中，原因是 TimerTask 是个抽象类，实现其需要重写 run()方法，而恰好可以将定时执行的任务置于 run()方法内部，通过 timer.schedule(..)调用。

Timer 核心的地方就是有多个重载方法方便使用：

经过 delay(ms)后开始进行调度，仅仅调度一次

public void schedule(TimerTask task, long delay)

在指定的时间点 time 上调度一次

public void schedule(TimerTask task, Date time)

在 delay（ms）后开始调度，而后以周期 period（ms）调度

public void schedule(TimerTask task, long delay, long period)

在指定时间 firstTime 时间调度，而后以周期 period（ms）调度

public void schedule(TimerTask task, Date firstTime, long period)

timer.scheduleAtFixedRate(…)同样也有多个重载，与 timer.schedule(…)在功能上无差别，唯一的区别就是前者具有追赶性，比如第一次开始执行的时间是 9 点 10 分 10 秒，而现在的时间是 9 点 11 分 10 秒，前者会挤时间追赶执行丢掉时间区间内的任务，而后者则不会。

六、 单例模式下的多线程

单例模式就是在整个进程中有且仅有一个实例化对象，多个线程之间共享此对象。

单例模式有两种方式，分别为饿汉模式和懒汉模式。

饿汉模式就是在任何线程调用之前已经实例化了唯一的对象，在整个系统保证是单例的，缺点是如果没有线程调用那这个对象的实例化就是多余的，类的加载机制不受人为控制增加了系统的开销。

懒汉模式就是在线程调用时判断对象是否已经实例化，若已经实例化则直接返回，若还未实例化则实例化后返回，换句话说可以控制类的加载机制，这样在没有线程调用时可节省系统开销，缺点是在多线程环境下，容易造成非单例的情况。

解决懒汉模式下几种保证单例模式的方案：

在获取单例对象的方法加入 synchronized 关键字，无疑可以解决，互斥访问效率必然低下。

使用 DCL 双重检测模式，具体代码截图如下：

```
1  package singleton;
2
3  public class MySingleton {
4
5      private static MySingleton mySingleton;
6
7      //私有构造 限制为单例模式
8      private MySingleton() {
9
10     }
11
12     public static MySingleton getInstance() {
13         if(mySingleton!=null) {
14             return mySingleton;
15         }else {
16             synchronized (MySingleton.class) {
17                 if(mySingleton==null) {
18                     mySingleton=new MySingleton();
19                 }
20             }
21             return mySingleton;
22         }
23     }
24
25 }
26
```

使用静态内置类实现单例模式，代码截图如下：

```java
package singleton;

/**
 * 使用静态内部类实现单例模式
 *
 * @author Together
 *
 */
public class MySingletonStaticInnerClass {

    private static class SingletonHolder {
        private static final MySingletonStaticInnerClass MY_SINGLETON_STATIC_INNER_CLASS
        =new  MySingletonStaticInnerClass();
    }

    private  MySingletonStaticInnerClass() {
    }

    public static MySingletonStaticInnerClass getInstance() {
        return SingletonHolder.MY_SINGLETON_STATIC_INNER_CLASS;
    }
}

```

该方式下的单例由静态内部类 SingletonHolder 的饿汉模式保证，由于内部类只有外部类的 getInstance()调用，因此内部类被加载的时机也就是第一次调用 getInstance()的时候，从内部看是一个饿汉模式，从外部看又的确是懒汉模式。

使用枚举特性实现单例模式，代码截图如下：

```java
package singleton;

/**
 * 使用枚举实现单例模式
 *
 * @author Together
 *
 */
public enum MySingletonEnum {

    singleton;
    private Temp temp;//单例对象
    private  MySingletonEnum() {
        temp=new Temp();
        System.out.println(temp.hashCode());
    }
    public Temp getTemp() {
        return temp;
    }
}
```

利用枚举在使用时才调用其构造方法的原理，从而控制了单例对象的加载时机，有效实现了懒汉模式下的单例。关于枚举的特性可以单独写篇文章进行

论述。由于使用枚举实现单例模式代码简洁、自动序列化机制、线程安全等等诸多优点，此方式成为实现懒汉模式下的单例的最佳选择。

Mr 至简写于 2018.04.18