Java并发编程

1.并发编程的引入

1.1上下文切换

线程允许在同一个进程中同时存在多个程序控制流，**线程共享进程范围内的资源**，例如内存句柄和文件句柄， 与此同时，每个线程拥有各自的**程序计数器PC**（指令序列中CPU正在执行的位置）、**栈**（记录执行历史）和存 入其中的**局部变量**（CPU寄存器），同一进程的多个线程可以同时调度到多个CPU上运行。 并发指在一段时间 间隔内多个任务执行，CPU调度采用**时间片轮转**的方式，以线程为基本调度单位，进行时间片的分配（一般几 十ms），当前线程任务执行一个时间片之后会切换到下一个线程任务执行，但切换之前会进行状态的保存， 当下一次时间片分配到该线程任务时加载这个任务的状态，即上下文切换。

- 上下文切换的原因：1）当前线程任务时间片用完；2）中断，CPU接收中断请求，在当前线程任务和发起中断 请求之间进行上下文切换，软中断IO阻塞、竞争失败等使得线程挂起， `Thread.sleep()`，`Object.wait()`， `Thread.join()`，`Thread.yield()`，`LockSupport.park()`
- 减少线程上下文切换：1）无锁并发编程，多线程竞争锁会引起上下文切换，避免使用锁，改变竞态条件；2）CAS算法 ，如 `AtomicInteger` 等原子类采用CAS更新数据，线程无需阻塞；3）使用最少线程，避免创建过多线程； 4）协程，一种程序

1.2并发编程多线程的优势

- 多核多CPU系统，并发程序通过提高处理器资源利用率来提升**系统吞吐率**
- 日渐复杂的业务请求，将每一个类似socket连接请求分配给一个线程进行处理，互不影响

2.并发带来的问题

2.1安全性问题

在没有同步控制的情况下，多个线程的执行顺序是不可预测的， UnsafeSequence 导致结果不同， 线程交替执行使得结果和预期的不尽相同。

竞态条件：计算的正确性取决于多个线程的交替执行时序，不同时序会造成不同结果，最为常 见的是先检查后执行（check-then-act），基于一种可能失效的观察结果来做出判断后者执行某个计算。

2.2活跃性问题

由于是线程进行切换，并且存在**竞争**，当某个操作或者线程无法继续执行就发生了活跃性问题。

- **死锁**：两个（及以上）的线程在执行过程中，由于资源竞争而造成的一种阻塞等待的现象，若无外力推动 会陷入无限等待，产生了死锁。死锁条件：1）互斥资源；2）请求和保持，阻塞对获取资源保持不放； 3）不剥夺，只有使用完毕才释放资源；4）循环等待，死锁发生时，所等待线程陷入循环等待；

```
 //死锁示例
public class DeadLock {
    public static String obj1 = "obj1";
    public static String obj2 = "obj2";
    public static void main(String[] args) {
        Thread a = new Lock1();
        Thread b = new Thread(new Lock2());
        a.start();
        b.start();
    }
}
//以同样顺序获取锁导致死锁
class Lock1 extends Thread {
    @Override
    public void run() {
        try {
            System.out.println("lock1......");
            synchronized (DeadLock.obj1) {
                System.out.println("lock " + DeadLock.obj1);
                Thread.sleep(3000);

                synchronized (DeadLock.obj2) {
                    System.out.println("lock " + DeadLock.obj2);
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
class Lock2 implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println("lock1......");
            synchronized (DeadLock.obj2) {
                System.out.println("lock " + DeadLock.obj2);
                Thread.sleep(3000);

                synchronized (DeadLock.obj1) {
                    System.out.println("lock " + DeadLock.obj1);
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- **饥饿**：如果一个线程因为 CPU 时间全部被其他线程抢走而得不到 CPU 运行时间，这种状态被称之 为"饥饿"。饥饿原因：1）高优先级线程占用所有CPU时间；2）线程阻塞等待某一同步块，但其他线程持续的 对同步块进行访问；3）线程本身等待一个本身也处于永久等待的对象；
- **活锁**：两个线程之间互相谦让，让其他线程优先使用资源，互相谦让。

2.3性能问题

体现在多个方面，服务时间长、响应不及时、吞吐率过低、资源消耗高，为解决安全性问题实施的策略可能 导致性能问题，如加锁导致性能低下。

3.Java内存模型

为了保证共享内存的正确性（**可见性、有序性、原子性**），内存模型定义了共享内存系统中多线程程序读写 操作行为的规范。通过这些规则来规范对内存的读写操作，从而保证指令执行的正确性。它与处理器有关、 与缓存有关、与并发有关、与编译器也有关。他解决了CPU多级缓存、处理器优化、指令重排等导致的内存 访问问题，保证了并发场景下的一致性、原子性和有序性。

Java的内存模型是解决多线程场景下并发问题的一个重要规范，定义了共享内存中多线程程序读写操作行为的规范， Java内存模型规定所有变量存储在主内存中，每个线程还有自己的工作内存，线程的工作内存中保存该线程中 用到的变量的主内存副本拷贝，线程对变量的所有操作必须在工作内存中进行，而不能直接读写主内存，不同 线程之间无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据 同步进行。

3.1原子性

指一个操作中cpu不可中途暂停然后再调度，要么执行完成要不就不执行；处理器优化会导致原子性问题，体现在 并发编程之中是，线程A进行非原子性操作，中途线程B读取信息，那么读取的就是错误信息。

```
 i++; //非原子性操作
i = 9; //假设赋值时32位高低位分别赋值，低位赋值后中断，另一线程读取，读到的错误数据
x = 10; //原子操作
y = x; //--非原子性操作
x++; // --
x = x + 1; // --
```

Java中对基本数据类型的变量的读取和赋值操作都是原子性操作，不可被中断，但实现更大范围的赋值， 就不是原子性操作，需要通过 synchronized、Lock 保证任意时刻只有一个线程执行该代码块。

3.2可见性

当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程应该立即能看到修改的值； 缓存一致性问题会导致可见性问题的发生。

当程序在运行过程中，会将运算需要的数据（变量）从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据 和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。CPU的三级缓存：L1>L2>L3>内存（速度），每级缓存80%命中；当CPU要读取一个数据时，首先从一级缓存中查找，如果没有找到再从二级缓存中查找， 如果还是没有就从三级缓存或内存中查找。

在多核cpu多线程的情况下，每个核至少一个L1缓存，多线程访问进程中某个共享内存时，且多线程在不同的核上执行，则每个核会在缓存中保留一份 共享内存的缓冲，此时多线程进行写操作，会造成**缓存不一致问题。**

```
//线程A
int i = 0;
i = 10;

//线程B
j = i;
```

此种情况赋值未及时可见，Java提供 volatile 关键字来保证可见性，当共享变量被 volatile 修饰时，会保证被 修改的值立即更新到主内存，其他线程需要读取时会去内存中读取新值。 synchronized、Lock 也能保证可见性， 因为每一时刻只能有一个线程获取锁然后执行同步代码，并且释放锁之前会对变量的修改刷新到主内存，保证可见性。

3.3有序性

程序执行顺序按照代码的先后顺序执行； flag = true; a = 1 会重排序造成程序逻辑错误；指令重排序会导致有序性问题.

指令重排序，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语 句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

指令重排置在程序执行过程中，为性能考虑，编译器和CPU可能会对指令重新排序，一条汇编指令执行可以分为很多步骤。 对没有依赖的指令进行重排序，as-if-serial规则。

IF（取指）、ID（译码和取操作数）、EX（ALU计算单元）、MEM（存储器访问）、WB（写回寄存器）。

指令执行是流水线执行，后面的指令执行可能需要前面指令执行完成才可，会有等待过程，可进行指令重排序；

```
a = b + c; d = e - f;
* LW Rb b-IF ID EX MEM WB
* LW Rc c----IF ID EX MEM WB
* ADD Ra,Rb,Rc-IF ID ? MEM WB // ?需要等待前面指令的完成
* SW a Ra //等待Rc的取值ID
* LW Re e //此处仍要等待可通过指令重排放到前面执行
```

Java内存模型中，允许编译器和处理器对指令重排序，指令重排不会对单线程程序影响，会影响多线程并发执行的正确性。

```
//线程A
a = 1; //1
flag = true; //2

//线程B
if (flag) { //3
    int i = a*a; //4
    ...
}
//重排序：1/2重排序 最终线程B的i = 0；
//3/4重排序 由于if (flag) 和temp = a*a没有依赖性关系，会重排序使得temp = 0 = i;
```

*happens-before*如果一个操作的执行结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系， 两个操作可以是一个线程内也可以是不同线程之间。

- 程序顺序规则：一个线程中的每一个操作先于该线程后续任意操作
- 监视器锁规则：锁的解锁先于发生加锁
- volatile变量规则：volatile变量的写先于读
- 传递性：A先于B，B先于C，那么A先于C
- 线程启动、中断、停止：start()方法先于它每一个动作，interrupt()先于被中断线程的代码，线程所有操作先于线程终结Thread.join()
- 对象：对象构造函数先于finalize()方法

*as-if-serial*语义规则，对于存在数据依赖关系的操作禁止重排序

```
//as-if-serial规则1.2没有数据依赖关系可以重排序
int a = 1; //1
int b = 2; //2
int c = a*b; //3
```

A happens-before，JMM并不要求A一定要在B之前执行，JMM仅仅要求前一个操作对后一个操作可见，且前一个 操作顺序排在第二个操作之前。如果A的执行结果不需要对操作B可见；而且重排序A,B之后结果并不影响最终结果， 此种情况下JMM允许重排序。

3.创建线程方式（6种）

1.继承 THREAD 类，重写 RUN() 方法为线程的业务逻辑， START() 启动线程

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " run()方法正在
执行...");
    }
}
public class TheadTest {

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();
        System.out.println(Thread.currentThread().getName() + " main()方法执
行结束");
    }

}
```

2.实现 RUNNABLE 接口

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " run()方法执行
中...");
    }
}
```

3.匿名内部类

```
new Thread(()->print("11"));
```

4.实现 CALLABLE<V> 接口带返回值

- 创建实现`Callable`接口的类`myCallable`
- 以`myCallable`为参数创建`FutureTask`对象
- 将`FutureTask`作为参数创建`Thread`对象
- 调用线程对象的`start()`方法

```
public class MyCallable implements Callable<Integer> {

    @Override
    public Integer call() {
        System.out.println(Thread.currentThread().getName() + " call()方法执
行中...");
```

```
        return 1;
    }

}
public class CallableTest {

    public static void main(String[] args) {
        FutureTask<Integer> futureTask = new FutureTask<Integer>(new MyCalla
ble());
        Thread thread = new Thread(futureTask);
        thread.start();

        try {
            Thread.sleep(1000);
            System.out.println("返回结果 " + futureTask.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + " main()方法执
行完成");
    }

}
```

5.定时器 TIMER

`Timer timer = new Timer(); timer.schedule(new TimerTask(), ,);`

6.线程池 EXECUTORS 工具类

newFixedThreadPool, newCachedThreadPool, newSingleThreadExecutor, newScheduledThreadPool 四种线程池，都实现 了 ExecutorService 接口。

```
public class MyRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " run()方法执行
中...");
    }

}

public class SingleThreadExecutorTest {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newSingleThreadExecutor
();
        MyRunnable runnableTest = new MyRunnable();
```

```
            for (int i = 0; i < 5; i++) {
                executorService.execute(runnableTest);
            }

            System.out.println("线程任务开始执行");
            executorService.shutdown();
        }

    }
```

`Callable，Runnable` 接口的区别：

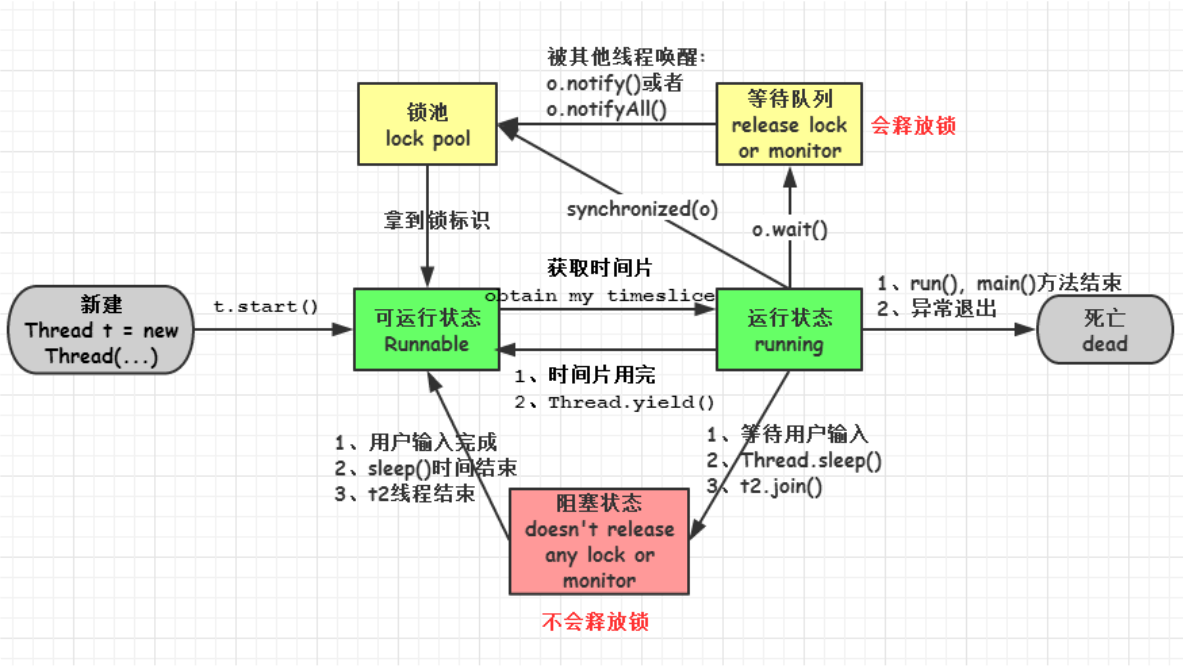
- 相同点：都是接口，都可编写多线程，都采用`Thread.run()`启动
- 不同点： `Callable` 接口 `call()` 有返回值，是个泛型，和 `Future，FutureTask` 配合获取执行结果； 可以获取异常信息。

为什么我们调用 `start()` 方法时会执行 `run()` 方法，为什么我们不能直接调用 `run()` 方法?

`run()` 方法只是线程体中一个普通方法，当线程运行时才会执行次业务方法， `start()` 方法启动线程， 使得线程由创建进入就绪状态，竞争`CPU`资源，当获取时间片时就可以开始执行进入运行状态。

4.线程状态

新建 new 、就绪(start())、运行(获取时间片cpu资源)、超时等待(sleep(),join()自动唤醒)、等待(wait()需要notify()唤醒)、阻塞(synchronized, blockIO获取锁资源)、结束 采用线程调度（分时或者抢占式）。



5.方法

- wait()：线程阻塞，释放所持有对象锁
- sleep()：睡眠，进入超时等待状态，不释放锁
- notify()：唤醒一个等待线程，随机
- notifyAll()：唤醒所有等待线程，进行竞争
- Thread.join()方法：在主线程中调用线程A的join方法进行同步，等待A线程执行完毕才会进行主线程

sleep(),wait() 区别：前者Thread方法后者Object方法，前者不释放锁，wait常用于线程通信，而sleep用于暂停执行，进入状态不同。

万物皆对象，对象皆为锁

6.线程安全性问题的条件

- 多线程环境
- 多个线程共享一个资源，竞争
- 对资源进行非原子操作

7.synchronized（当前对象锁）重量级锁

修饰静态方法锁.class文件--monitor,monitorexit

```
 //双重校验锁实现对象单例（线程安全）
//在多个线程试图在同一时间创建对象时，会通过加锁来保证只有一个线程能创建对象。
//在对象创建好之后，执行getInstance()将不需要获取锁，直接返回已创建好的对象。
public class Singleton {

    private volatile static Singleton uniqueInstance; //volatile禁止指令重排序

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
       //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

可重入

重入锁是指一个线程(Thread.currentThread() == pThread)获取到该锁之后，该线程可以继续获得该锁。底层原理维护一个计 数器，当线程获取该锁时，计数器加一，再次获得该锁时继续加一，释放锁时，计数器 减一，当计数器值为0时，表明该锁未被任何线程所持有，其它线程可以竞争获取锁。

自旋锁

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线 程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。 既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

```
public class SpinLock {
    private AtomicReference<Thread> owner =new AtomicReference<>();
    public void lock(){
```

```
            Thread current = Thread.currentThread();
            while(!owner.compareAndSet(null, current)){ //自旋操作
            }
        }
        public void unlock (){
            Thread current = Thread.currentThread();
            owner.compareAndSet(current, null);
        }
    }
```

8.volatile

可见性（线程A知道线程B修改了变量）

volatile,synchronized,wait/notify（通信）,while轮询

SYNCHRONIZED、VOLATILE、CAS 比较

- synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。
- volatile 提供多线程共享变量可见性和禁止指令重排序优化。
- CAS 是基于冲突检测的乐观锁（非阻塞）