高质量并发详解

并发是java高级程序员必须要深入研究的话题，从Synchronized到Lock，JDK本身提供了很多优秀的并发类和锁控制器，灵活使用这些类，可以写出优秀的并发程序，而这些类基本上都是在java.util.concurrent包中的，本文将会从具体的例子出发，一步一步带领大家进入java高质量并发的世界。

本文的例子可以参考https://github.com/ddean2009/learn-java-concurrency/

第一章 java.util.concurrent简介

java.util.concurrent包提供了很多有用的类，方便我们进行并发程序的开发。本文将会做一个总体的简单介绍。

主要的组件

java.util.concurrent包含了很多内容， 本文将会挑选其中常用的一些类来进行大概的说明：

- Executor
- ExecutorService
- ScheduledExecutorService
- Future
- CountDownLatch
- CyclicBarrier
- Semaphore
- ThreadFactory

Executor

Executor是一个接口，它定义了一个execute方法，这个方法接收一个Runnable，并在其中调用Runnable的run方法。

我们看一个Executor的实现：

```
public class Invoker implements Executor {
    @Override
    public void execute(Runnable r) {
        r.run();
    }
}
```

现在我们可以直接调用该类中的方法：

```
    public void execute() {
        Executor executor = new Invoker();
        executor.execute( () -> {
            log.info("{}", Thread.currentThread().toString());
        });
    }
```

> 注意，Executor并不一定要求执行的任务是异步的。

ExecutorService

如果我们真正的需要使用多线程的话，那么就需要用到ExecutorService了。

ExecutorService管理了一个内存的队列，并定时提交可用的线程。

我们首先定义一个Runnable类：

```
public class Task implements Runnable {
    @Override
    public void run() {
        // task details
    }
}
```

我们可以通过Executors来方便的创建ExecutorService：

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

上面创建了一个ThreadPool， 我们也可以创建单线程的ExecutorService：

```
ExecutorService executor =Executors.newSingleThreadExecutor();
```

我们这样提交task：

```
public void execute() {
    executor.submit(new Task());
}
```

因为ExecutorService维持了一个队列，所以它不会自动关闭， 我们需要调用executor.shutdown() 或者executor.shutdownNow()来关闭它。

如果想要判断ExecutorService中的线程在收到shutdown请求后是否全部执行完毕，可以调用如下的方法：

```
try {
            executor.awaitTermination( 5l, TimeUnit.SECONDS );
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

ScheduledExecutorService

ScheduledExecutorService和ExecutorService很类似，但是它可以周期性的执行任务。

我们这样创建ScheduledExecutorService：

```
ScheduledExecutorService executorService
                = Executors.newSingleThreadScheduledExecutor();
```

executorService的schedule方法，可以传入Runnable也可以传入Callable：

```
Future<String> future = executorService.schedule(() -> {
        // ...
        return "Hello world";
    }, 1, TimeUnit.SECONDS);

    ScheduledFuture<?> scheduledFuture = executorService.schedule(() -> {
        // ...
    }, 1, TimeUnit.SECONDS);
```

还有两个比较相近的方法：

```
scheduleAtFixedRate( Runnable command, long initialDelay, long period, TimeUnit unit )

scheduleWithFixedDelay( Runnable command, long initialDelay, long delay, TimeUnit unit
)
```

两者的区别是前者的period是以任务开始时间来计算的，后者是以任务结束时间来计算。

Future

Future用来获取异步执行的结果。可以调用cancel(boolean mayInterruptIfRunning) 方法来取消线程的执行。

我们看下怎么得到一个Future对象：

```
public void invoke() {
    ExecutorService executorService = Executors.newFixedThreadPool(10);

    Future<String> future = executorService.submit(() -> {
        // ...
        Thread.sleep(10000l);
        return "Hello world";
    });
}
```

我们看下怎么获取Future的结果：

```
if (future.isDone() && !future.isCancelled()) {
    try {
        str = future.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

future还可以接受一个时间参数，超过指定的时间，将会报TimeoutException。

```
try {
    future.get(10, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException | TimeoutException e) {
    e.printStackTrace();
}
```

CountDownLatch

CountDownLatch是一个并发中很有用的类，CountDownLatch会初始化一个counter，通过这个counter变量，来控制资源的访问。我们会在后面的文章详细介绍。

CyclicBarrier

CyclicBarrier和CountDownLatch很类似。CyclicBarrier主要用于多个线程互相等待的情况，可以通过调用await()方法等待，知道达到要等的数量。

```
public class Task implements Runnable {

    private CyclicBarrier barrier;

    public Task(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    @Override
```

```
    public void run() {
        try {
            LOG.info(Thread.currentThread().getName() +
              " is waiting");
            barrier.await();
            LOG.info(Thread.currentThread().getName() +
              " is released");
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

}
```

```
public void start() {

    CyclicBarrier cyclicBarrier = new CyclicBarrier(3, () -> {
        // ...
        LOG.info("All previous tasks are completed");
    });

    Thread t1 = new Thread(new Task(cyclicBarrier), "T1");
    Thread t2 = new Thread(new Task(cyclicBarrier), "T2");
    Thread t3 = new Thread(new Task(cyclicBarrier), "T3");

    if (!cyclicBarrier.isBroken()) {
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Semaphore

Semaphore包含了一定数量的许可证，通过获取许可证，从而获得对资源的访问权限。通过 tryAcquire()来获取许可，如果获取成功，许可证的数量将会减少。

一旦线程release()许可，许可的数量将会增加。

我们看下怎么使用：

```
static Semaphore semaphore = new Semaphore(10);

public void execute() throws InterruptedException {

    LOG.info("Available permit : " + semaphore.availablePermits());
    LOG.info("Number of threads waiting to acquire: " +
      semaphore.getQueueLength());
```

```
    if (semaphore.tryAcquire()) {
        try {
            // ...
        }
        finally {
            semaphore.release();
        }
    }

}
```

ThreadFactory

ThreadFactory可以很方便的用来创建线程：

```
public class ThreadFactoryUsage implements ThreadFactory {
    private int threadId;
    private String name;

    public ThreadFactoryUsage(String name) {
        threadId = 1;
        this.name = name;
    }

    @Override
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r, name + "-Thread_" + threadId);
        log.info("created new thread with id : " + threadId +
                " and name : " + t.getName());
        threadId++;
        return t;
    }
}
```

第二章 java并发中的Synchronized关键词

如果在多线程的环境中，我们经常会遇到资源竞争的情况，比如多个线程要去同时修改同一个共享变量，这时候，就需要对资源的访问方法进行一定的处理，保证同一时间只有一个线程访问。

java提供了synchronized关键字，方便我们实现上述操作。

为什么要同步

我们举个例子，我们创建一个类，提供了一个setSum的方法：

```
public class SynchronizedMethods {

    private int sum = 0;

    public void calculate() {
        setSum(getSum() + 1);
    }
}
```

如果我们在多线程的环境中调用这个calculate方法：

```
    @Test
    public void givenMultiThread_whenNonSyncMethod() throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(3);
        SynchronizedMethods summation = new SynchronizedMethods();

        IntStream.range(0, 1000)
                .forEach(count -> service.submit(summation::calculate));
        service.shutdown();
        service.awaitTermination(1000, TimeUnit.MILLISECONDS);

        assertEquals(1000, summation.getSum());
    }
```

按照上面的方法，我们预计要返回1000， 但是实际上基本不可能得到1000这个值，因为在多线程环境中，对同一个资源进行同时操作带来的不利影响。

那我们怎么才能够建线程安全的环境呢?

Synchronized关键词

java提供了多种线程安全的方法，本文主要讲解Synchronized关键词，Synchronized关键词可以有很多种形式：

- Instance methods
- Static methods
- Code blocks

当我们使用synchronized时，java会在相应的对象上加锁，从而在同一个对象等待锁的方法都必须顺序执行，从而保证了线程的安全。

Synchronized Instance Methods

Synchronized关键词可以放在实例方法的前面：

```
    public synchronized void synchronisedCalculate() {
        setSum(getSum() + 1);
    }
```

看下调用结果：

```
@Test
public void givenMultiThread_whenMethodSync() {
    ExecutorService service = Executors.newFixedThreadPool(3);
    SynchronizedMethods method = new SynchronizedMethods();

    IntStream.range(0, 1000)
      .forEach(count -> service.submit(method::synchronisedCalculate));
    service.awaitTermination(1000, TimeUnit.MILLISECONDS);

    assertEquals(1000, method.getSum());
}
```

这里synchronized将会锁住该方法的实例对象，多个线程中只有获得该实例对象锁的线程才能够执行。

Synchronized Static Methods

Synchronized关键词也可以用在static方法前面：

```
    public static synchronized void syncStaticCalculate() {
        staticSum = staticSum + 1;
    }
```

Synchronized放在static方法前面和实例方法前面锁住的对象不同。放在static方法前面锁住的对象是这个Class本身，因为一个Class在JVM中只会存在一个，所以不管有多少该Class的实例，在同一时刻只会有一个线程可以执行该放方法。

```
    @Test
    public void givenMultiThread_whenStaticSyncMethod() throws InterruptedException {
        ExecutorService service = Executors.newCachedThreadPool();

        IntStream.range(0, 1000)
                .forEach(count ->
                        service.submit(SynchronizedMethods::syncStaticCalculate));
        service.shutdown();
        service.awaitTermination(100, TimeUnit.MILLISECONDS);

        assertEquals(1000, SynchronizedMethods.staticSum);
    }
```

Synchronized Blocks

有时候，我们可能不需要Synchronize整个方法，而是同步其中的一部分，这时候，我们可以使用Synchronized Blocks：

```
    public void performSynchronizedTask() {
        synchronized (this) {
            setSum(getSum() + 1);
        }
    }
```

我们看下怎么测试：

```
    @Test
    public void givenMultiThread_whenBlockSync() throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(3);
        SynchronizedMethods synchronizedBlocks = new SynchronizedMethods();

        IntStream.range(0, 1000)
                .forEach(count ->
                        service.submit(synchronizedBlocks::performSynchronizedTask));
        service.shutdown();
        service.awaitTermination(100, TimeUnit.MILLISECONDS);

        assertEquals(1000, synchronizedBlocks.getSum());
    }
```

上面我们同步的是实例，如果在静态方法中，我们也可以同步class：

```
    public static void performStaticSyncTask(){
        synchronized (SynchronizedMethods.class) {
            staticSum = staticSum + 1;
        }
    }
```

我们看下怎么测试：

```
    @Test
    public void givenMultiThread_whenStaticSyncBlock() throws InterruptedException {
        ExecutorService service = Executors.newCachedThreadPool();

        IntStream.range(0, 1000)
                .forEach(count ->
                        service.submit(SynchronizedMethods::performStaticSyncTask));
        service.shutdown();
        service.awaitTermination(100, TimeUnit.MILLISECONDS);

        assertEquals(1000, SynchronizedMethods.staticSum);
    }
```

第三章 java中的Volatile关键字使用

在本文中，我们会介绍java中的一个关键字volatile。 volatile的中文意思是易挥发的，不稳定的。那么在java中使用是什么意思呢?

我们知道，在java中，每个线程都会有个自己的内存空间，我们称之为working memory。这个空间会缓存一些变量的信息，从而提升程序的性能。当执行完某个操作之后，thread会将更新后的变量更新到主缓存中，以供其他线程读写。

因为变量存在working memory和main memory两个地方，那么就有可能出现不一致的情况。 那么我们就可以使用Volatile关键字来强制将变量直接写到main memory，从而保证了不同线程读写到的是同一个变量。

什么时候使用volatile

那么我们什么时候使用volatile呢？当一个线程需要立刻读取到另外一个线程修改的变量值的时候，我们就可以使用volatile。我们来举个例子：

```
public class VolatileWithoutUsage {
    private  int count = 0;

    public void incrementCount() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

这个类定义了一个incrementCount()方法，会去更新count值，我们接下来在多线程环境中去测试这个方法：

```
    @Test
    public void testWithoutVolatile() throws InterruptedException {
        ExecutorService service= Executors.newFixedThreadPool(3);
        VolatileWithoutUsage volatileWithoutUsage=new VolatileWithoutUsage();

        IntStream.range(0,1000).forEach(count -
>service.submit(volatileWithoutUsage::incrementCount) );
        service.shutdown();
        service.awaitTermination(1000, TimeUnit.MILLISECONDS);
        assertEquals(1000,volatileWithoutUsage.getCount() );
    }
```

运行一下，我们会发现结果是不等于1000的。

```
java.lang.AssertionError:
Expected :1000
Actual   :999
```

这是因为多线程去更新同一个变量，我们在上篇文章也提到了，这种情况可以通过加Synchronized关键字来解决。

那么是不是我们加上Volatile关键字后就可以解决这个问题了呢?

```
public class VolatileFalseUsage {
    private volatile int count = 0;

    public void incrementCount() {
        count++;
    }
    public int getCount() {
        return count;
    }

}
```

上面的类中，我们加上了关键字Volatile，我们再测试一下：

```
    @Test
    public void testWithVolatileFalseUsage() throws InterruptedException {
        ExecutorService service= Executors.newFixedThreadPool(3);
        VolatileFalseUsage volatileFalseUsage=new VolatileFalseUsage();

        IntStream.range(0,1000).forEach(count -
>service.submit(volatileFalseUsage::incrementCount) );
        service.shutdown();
        service.awaitTermination(5000, TimeUnit.MILLISECONDS);
        assertEquals(1000,volatileFalseUsage.getCount() );
    }
```

运行一下，我们会发现结果还是错误的：

```
java.lang.AssertionError:
Expected :1000
Actual   :992
~~
```

为什么呢？ 我们先来看下count++的操作，count++可以分解为三步操作，1. 读取count的值，2.给count加1，3.将count写回内存。添加Volatile关键词只能够保证count的变化立马可见，而不能保证1，2，3这三个步骤的总体原子性。 要实现总体的原子性还是需要用到类似Synchronized的关键字。

下面看下正确的用法：

~~~java
public class VolatileTrueUsage {
    private volatile int count = 0;

    public void setCount(int number) {
        count=number;
    }
```
~~~

```
    public int getCount() {
        return count;
    }
}
```

```
    @Test
    public void testWithVolatileTrueUsage() throws InterruptedException {
        VolatileTrueUsage volatileTrueUsage=new VolatileTrueUsage();
        Thread threadA = new Thread(()->volatileTrueUsage.setCount(10));
        threadA.start();
        Thread.sleep(100);

        Thread reader = new Thread(() -> {
            int valueReadByThread = volatileTrueUsage.getCount();
            assertEquals(10, valueReadByThread);
        });
        reader.start();
    }
```

Happens-Before

从java5之后，volatile提供了一个Happens-Before的功能。Happens-Before 是指当volatile进行写回主内存的操作时，会将之前的非volatile的操作一并写回主内存。

```
public class VolatileHappenBeforeUsage {

    int a = 0;
    volatile boolean flag = false;

    public void writer() {
        a = 1;               // 1 线程A修改共享变量
        flag = true;         // 2 线程A写volatile变量
    }
}
```

上面的例子中，a是一个非volatile变量，flag是一个volatile变量，但是由于happens-before的特性，a 将会表现的和volatile一样。

第四章 wait和sleep的区别

在本篇文章中，我们将会讨论一下java中wait()和sleep()方法的区别。并讨论一下怎么使用这两个方法。

Wait和sleep的区别

wait() 是Object中定义的native方法：

```
public final native void wait(long timeout) throws InterruptedException;
```

所以每一个类的实例都可以调用这个方法。wait()只能在synchronized block中调用。它会释放synchronized时加在object上的锁。

sleep()是定义Thread中的native静态类方法：

```
public static native void sleep(long millis) throws InterruptedException;
```

所以Thread.sleep()可以在任何情况下调用。Thread.sleep()将会暂停当前线程，并且不会释放任何锁资源。

我们先看一下一个简单的wait使用：

```
@Slf4j
public class WaitUsage {

    private static Object LOCK = new Object();

    public static void WaitExample() throws InterruptedException {
        synchronized (LOCK) {
            LOCK.wait(1000);
            log.info("Object '" + LOCK + "' is woken after" +
                    " waiting for 1 second");
        }
    }
}
```

再看一下sleep的使用：

```
@Slf4j
public class SleepUsage {

    public static void sleepExample() throws InterruptedException {
        Thread.sleep(1000);
        log.info(
                "Thread '" + Thread.currentThread().getName() +
                        "' is woken after sleeping for 1 second");
    }
}
```

唤醒wait和sleep

sleep()方法自带sleep时间，时间过后，Thread会自动被唤醒。
或者可以通过调用interrupt()方法来中断。

相比而言wait的唤醒会比较复杂，我们需要调用notify() 和 notifyAll()方法来唤醒等待在特定wait object上的线程。

notify()会根据线程调度的机制选择一个线程来唤醒，而notifyAll()会唤醒所有等待的线程，由这些线程重新争夺资源锁。

wait,notity通常用在生产者和消费者情形，我们看下怎么使用：

```
@Slf4j
public class WaitNotifyUsage {

    private int count =0;

    public void produceMessage() throws InterruptedException {

        while(true) {
            synchronized (this) {
                while (count == 5) {
                    log.info("count == 5 , wait ....");
                    wait();
                }
                count++;
                log.info("produce count {}", count);
                notify();
            }
        }
    }

    public void consumeMessage() throws InterruptedException {

        while (true) {
            synchronized (this) {
                while (count == 0) {
                    log.info("count == 0, wait ...");
                    wait();
                }
                log.info("consume count {}", count);
                count--;
                notify();
            }
        }
    }
}
```

看下怎么调用：

```
   @Test
    public void testWaitNotifyUsage() throws InterruptedException{
        WaitNotifyUsage waitNotifyUsage=new WaitNotifyUsage();

        ExecutorService executorService=Executors.newFixedThreadPool(4);
        executorService.submit(()-> {
            try {
                waitNotifyUsage.produceMessage();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        });

        executorService.submit(()-> {
            try {
                waitNotifyUsage.consumeMessage();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread.sleep(50000);
    }
```

第五章 java中Future的使用

Future是java 1.5引入的一个interface，可以方便的用于异步结果的获取。 本文将会通过具体的例子讲解如何使用Future。

创建Future

正如上面所说，Future代表的是异步执行的结果，意思是当异步执行结束之后，返回的结果将会保存在Future中。

那么我们什么时候会用到Future呢？ 一般来说，当我们执行一个长时间运行的任务时，使用Future就可以让我们暂时去处理其他的任务，等长任务执行完毕再返回其结果。

经常会使用到Future的场景有：1. 计算密集场景。2. 处理大数据量。3. 远程方法调用等。

接下来我们将会使用ExecutorService来创建一个Future。

```
    <T> Future<T> submit(Callable<T> task);
```

上面是ExecutorService中定义的一个submit方法，它接收一个Callable参数，并返回一个Future。

我们用一个线程来计算一个平方运算：

```
    private ExecutorService executor
            = Executors.newSingleThreadExecutor();

    public Future<Integer> calculate(Integer input) {
        return executor.submit(() -> {
            System.out.println("Calculating..."+ input);
            Thread.sleep(1000);
            return input * input;
        });
    }
```

submit需要接受一个Callable参数，Callable需要实现一个call方法，并返回结果。这里我们使用lamaba表达式来简化这一个流程。

从Future获取结果

上面我们创建好了Future，接下来我们看一下怎么获取到Future的值。

```
        FutureUsage futureUsage=new FutureUsage();
         Future<Integer> futureOne = futureUsage.calculate(20);
         while(!futureOne.isDone()) {
             System.out.println("Calculating...");
             Thread.sleep(300);
         }
         Integer result = futureOne.get();
```

首先我们通过Future.isDone() 来判断这个异步操作是否执行完毕，如果完毕我们就可以直接调用futureOne.get()来获得Futre的结果。

这里futureOne.get()是一个阻塞操作，会一直等待异步执行完毕才返回结果。

如果我们不想等待，future提供了一个带时间的方法：

```
Integer result = futureOne.get(500, TimeUnit.MILLISECONDS);
```

如果在等待时间结束的时候，Future还有返回，则会抛出一个TimeoutException。

取消Future

如果我们提交了一个异步程序，但是想取消它， 则可以这样：

```
uture<Integer> futureTwo = futureUsage.calculate(4);

        boolean canceled = futureTwo.cancel(true);
```

Future.cancel(boolean) 传入一个boolean参数，来选择是否中断正在运行的task。

如果我们cancel之后，再次调用get()方法，则会抛出CancellationException。

多线程环境中运行

如果有两个计算任务，先看下在单线程下运行的结果。

```
        Future<Integer> future1 = futureUsage.calculate(10);
        Future<Integer> future2 = futureUsage.calculate(100);

        while (!(future1.isDone() && future2.isDone())) {
            System.out.println(
                    String.format(
                            "future1 is %s and future2 is %s",
                            future1.isDone() ? "done" : "not done",
                            future2.isDone() ? "done" : "not done"
                    )
```

```
            );
            Thread.sleep(300);
        }

        Integer result1 = future1.get();
        Integer result2 = future2.get();

        System.out.println(result1 + " and " + result2);
```

因为我们通过Executors.newSingleThreadExecutor（）来创建的单线程池。所以运行结果如下：

```
Calculating...10
future1 is not done and future2 is not done
future1 is not done and future2 is not done
future1 is not done and future2 is not done
future1 is not done and future2 is not done
Calculating...100
future1 is done and future2 is not done
future1 is done and future2 is not done
future1 is done and future2 is not done
100 and 10000
```

如果我们使用Executors.newFixedThreadPool(2)来创建一个多线程池，则可以得到如下的结果：

```
calculating...10
calculating...100
future1 is not done and future2 is not done
future1 is not done and future2 is not done
future1 is not done and future2 is not done
future1 is not done and future2 is not done
100 and 10000
```

第六章 java并发中ExecutorService的使用

ExecutorService是java中的一个异步执行的框架，通过使用ExecutorService可以方便的创建多线程执行环境。

本文将会详细的讲解ExecutorService的具体使用。

创建ExecutorService

通常来说有两种方法来创建ExecutorService。

第一种方式是使用Executors中的工厂类方法，例如：

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

除了newFixedThreadPool方法之外，Executors还包含了很多创建ExecutorService的方法。

第二种方法是直接创建一个ExecutorService， 因为ExecutorService是一个interface，我们需要实例化ExecutorService的一个实现。

这里我们使用ThreadPoolExecutor来举例：

```
ExecutorService executorService =
            new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<Runnable>());
```

为ExecutorService分配Tasks

ExecutorService可以执行Runnable和Callable的task。其中Runnable是没有返回值的，而Callable是有返回值的。我们分别看一下两种情况的使用：

```
Runnable runnableTask = () -> {
    try {
        TimeUnit.MILLISECONDS.sleep(300);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Callable<String> callableTask = () -> {
    TimeUnit.MILLISECONDS.sleep(300);
    return "Task's execution";
};
```

将task分配给ExecutorService，可以通过调用xecute(), submit(), invokeAny(), invokeAll()这几个方法来实现。

execute() 返回值是void，他用来提交一个Runnable task。

```
executorService.execute(runnableTask);
```

submit() 返回值是Future，它可以提交Runnable task, 也可以提交Callable task。 提交Runnable的有两个方法：

```
<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);
```

第一个方法在返回传入的result。第二个方法返回null。

再看一下callable的使用：

```
Future<String> future =
  executorService.submit(callableTask);
```

invokeAny() 将一个task列表传递给executorService，并返回其中的一个成功返回的结果。

```
String result = executorService.invokeAny(callableTasks);
```

invokeAll() 将一个task列表传递给executorService，并返回所有成功执行的结果：

```
List<Future<String>> futures = executorService.invokeAll(callableTasks);
```

关闭ExecutorService

如果ExecutorService中的任务运行完毕之后，ExecutorService不会自动关闭。它会等待接收新的任务。如果需要关闭ExecutorService， 我们需要调用shutdown() 或者 shutdownNow() 方法。

shutdown() 会立即销毁ExecutorService，它会让ExecutorServic停止接收新的任务，并等待现有任务全部执行完毕再销毁。

```
executorService.shutdown();
```

shutdownNow()并不保证所有的任务都被执行完毕，它会返回一个未执行任务的列表：

```
List<Runnable> notExecutedTasks = executorService.shutdownNow();
```

oracle推荐的最佳关闭方法是和awaitTermination一起使用：

```
executorService.shutdown();
        try {
            if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
        }
```

先停止接收任务，然后再等待一定的时间让所有的任务都执行完毕，如果超过了给定的时间，则立刻结束任务。

Future

submit() 和 invokeAll() 都会返回Future对象。之前的文章我们已经详细讲过了Future。 这里就只列举一下怎么使用：

```
 Future<String> future = executorService.submit(callableTask);
String result = null;
try {
    result = future.get();
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

ScheduledExecutorService

ScheduledExecutorService为我们提供了定时执行任务的机制。

我们这样创建ScheduledExecutorService：

```
ScheduledExecutorService executorService
                = Executors.newSingleThreadScheduledExecutor();
```

executorService的schedule方法，可以传入Runnable也可以传入Callable：

```
Future<String> future = executorService.schedule(() -> {
        // ...
        return "Hello world";
    }, 1, TimeUnit.SECONDS);

    ScheduledFuture<?> scheduledFuture = executorService.schedule(() -> {
        // ...
    }, 1, TimeUnit.SECONDS);
```

还有两个比较相近的方法：

```
scheduleAtFixedRate( Runnable command, long initialDelay, long period, TimeUnit unit )

scheduleWithFixedDelay( Runnable command, long initialDelay, long delay, TimeUnit unit
)
```

两者的区别是前者的period是以任务开始时间来计算的，后者是以任务结束时间来计算。

ExecutorService和 Fork/Join

java 7 引入了Fork/Join框架。 那么两者的区别是什么呢?

ExecutorService可以由用户来自己控制生成的线程，提供了对线程更加细粒度的控制。而Fork/Join则是为了让任务更加快速的执行完毕。

第七章 java中Runnable和Callable的区别

在java的多线程开发中Runnable一直以来都是多线程的核心，而Callable是java1.5添加进来的一个增强版本。

本文我们会详细探讨Runnable和Callable的区别。

运行机制

首先看下Runnable和Callable的接口定义：

```
@FunctionalInterface
public interface Runnable {
```

```
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

```
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Runnable需要实现run（）方法，Callable需要实现call（）方法。

我们都知道要自定义一个Thread有两种方法，一是继承Thread，而是实现Runnable接口，这是因为Thread本身就是一个Runnable的实现：

```
class Thread implements Runnable {
    /* Make sure registerNatives is the first thing <clinit> does. */
    private static native void registerNatives();
    static {
        registerNatives();
    }
    ...
```

所以Runnable可以通过Runnable和之前我们介绍的ExecutorService 来执行，而Callable则只能通过ExecutorService 来执行。

返回值的不同

根据上面两个接口的定义，Runnable是不返还值的，而Callable可以返回值。

如果我们都通过ExecutorService来提交，看看有什么不同：

- 使用runnable

```
    public void executeTask() {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future future = executorService.submit(()->log.info("in runnable!!!!"));
        executorService.shutdown();
    }
```

- 使用callable

```
    public void executeTask() {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future future = executorService.submit(()->{
            log.info("in callable!!!!");
            return "callable";
        });
        executorService.shutdown();
    }
```

虽然我们都返回了Future，但是runnable的情况下Future将不包含任何值。

Exception处理

Runnable的run（）方法定义没有抛出任何异常，所以任何的Checked Exception都需要在run（）实现方法中自行处理。

Callable的Call（）方法抛出了throws Exception，所以可以在call（）方法的外部，捕捉到Checked Exception。我们看下Callable中异常的处理。

```
public void executeTaskWithException(){
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future future = executorService.submit(()->{
            log.info("in callable!!!!");
            throw new CustomerException("a customer Exception");
        });
        try {
            Object object= future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
            e.getCause();
        }
        executorService.shutdown();
    }
```

上面的例子中，我们在Callable中抛出了一个自定义的CustomerException。

这个异常会被包含在返回的Future中。当我们调用future.get()方法时，就会抛出ExecutionException，通过e.getCause()，就可以获取到包含在里面的具体异常信息。

第八章 ThreadLocal的使用

ThreadLocal主要用来为当前线程存储数据，这个数据只有当前线程可以访问。

在定义ThreadLocal的时候，我们可以同时定义存储在ThreadLocal中的特定类型的对象。

```
ThreadLocal<Integer> threadLocalValue = new ThreadLocal<>();
```

上面我们定义了一个存储Integer的ThreadLocal对象。

要存储和获取ThreadLocal中的对象也非常简单，使用get（）和set（）即可：

```
threadLocalValue.set(1);
Integer result = threadLocalValue.get();
```

我可以将ThreadLocal看成是一个map，而当前的线程就是map中的key。

除了new一个ThreadLocal对象，我们还可以通过：

```
    public static <S> ThreadLocal<S> withInitial(Supplier<? extends S> supplier) {
        return new SuppliedThreadLocal<>(supplier);
    }
```

ThreadLocal提供的静态方法withInitial来初始化一个ThreadLocal。

```
ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 1);
```

withInitial需要一个Supplier对象，通过调用Supplier的get()方法获取到初始值。

要想删除ThreadLocal中的存储数据，可以调用：

```
threadLocal.remove();
```

下面我通过两个例子的对比，来看一下使用ThreadLocal的好处。

在实际的应用中，我们通常会需要为不同的用户请求存储不同的用户信息，一般来说我们需要构建一个全局的Map，来根据不同的用户ID，来存储不同的用户信息，方便在后面获取。

在Map中存储用户数据

我们先看下如果使用全局的Map该怎么用：

```
public class SharedMapWithUserContext implements Runnable {

    public static Map<Integer, Context> userContextPerUserId
            = new ConcurrentHashMap<>();
    private Integer userId;
    private UserRepository userRepository = new UserRepository();
```

```
    public SharedMapWithUserContext(int i) {
        this.userId=i;
    }

    @Override
    public void run() {
        String userName = userRepository.getUserNameForUserId(userId);
        userContextPerUserId.put(userId, new Context(userName));
    }
}
```

这里我们定义了一个static的Map来存取用户信息。

再看一下怎么使用：

```
    @Test
    public void testWithMap(){
        SharedMapWithUserContext firstUser = new SharedMapWithUserContext(1);
        SharedMapWithUserContext secondUser = new SharedMapWithUserContext(2);
        new Thread(firstUser).start();
        new Thread(secondUser).start();
        assertEquals(SharedMapWithUserContext.userContextPerUserId.size(), 2);
    }
```

在ThreadLocal中存储用户数据

如果我们要在ThreadLocal中使用可以这样：

```
public class ThreadLocalWithUserContext implements Runnable {

    private static ThreadLocal<Context> userContext
            = new ThreadLocal<>();
    private Integer userId;
    private UserRepository userRepository = new UserRepository();

    public ThreadLocalWithUserContext(int i) {
        this.userId=i;
    }

    @Override
    public void run() {
        String userName = userRepository.getUserNameForUserId(userId);
        userContext.set(new Context(userName));
        System.out.println("thread context for given userId: "
                + userId + " is: " + userContext.get());
    }

}
```

测试代码如下：

```
public class ThreadLocalWithUserContextTest {

    @Test
    public void testWithThreadLocal(){
        ThreadLocalWithUserContext firstUser
                = new ThreadLocalWithUserContext(1);
        ThreadLocalWithUserContext secondUser
                = new ThreadLocalWithUserContext(2);
        new Thread(firstUser).start();
        new Thread(secondUser).start();
    }
}
```

运行之后，我们可以得到下面的结果：

```
thread context for given userId: 1 is: com.flydean.Context@411734d4
thread context for given userId: 2 is: com.flydean.Context@1e9b6cc
```

不同的用户信息被存储在不同的线程环境中。

> 注意，我们使用ThreadLocal的时候，一定是我们可以自由的控制所创建的线程。如果在ExecutorService环境下，就最好不要使用ThreadLocal，因为在ExecutorService中，线程是不可控的。

第九章 java中线程的生命周期

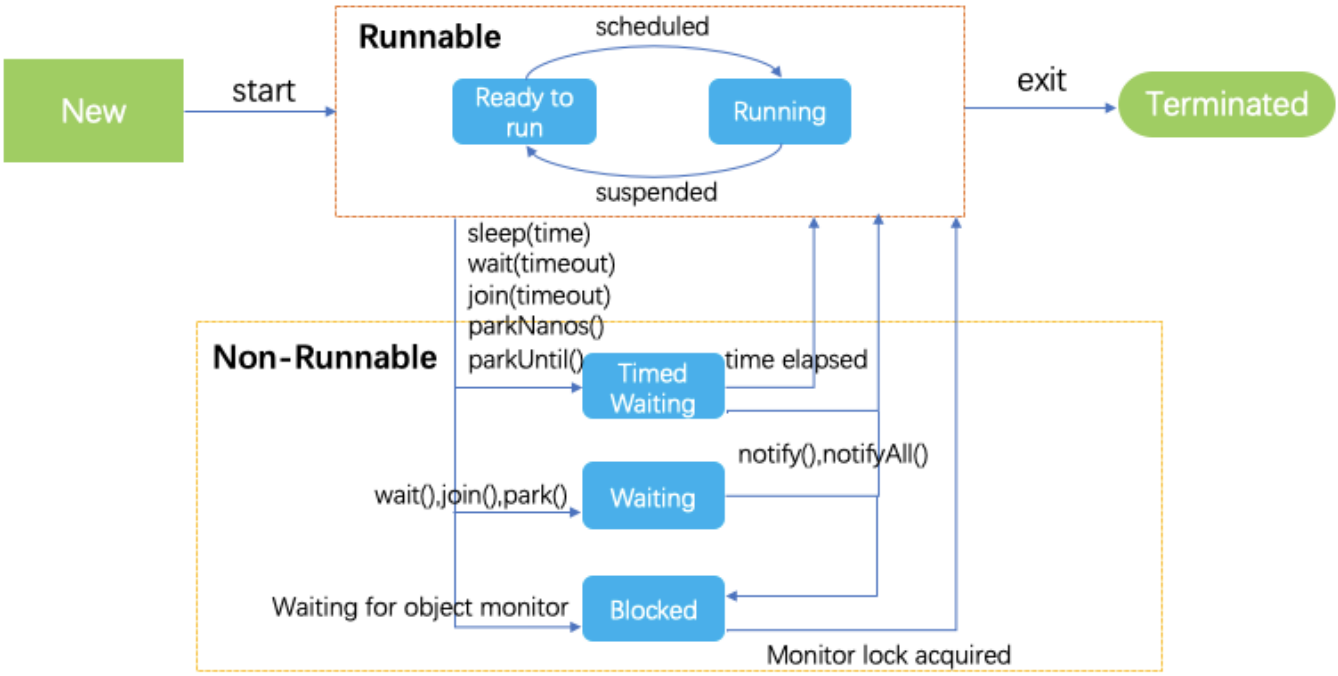
线程是java中绕不过去的一个话题， 今天本文将会详细讲解java中线程的生命周期，希望可以给大家一些启发。

java中Thread的状态

java中Thread有6种状态，分别是：

1. NEW - 新创建的Thread，还没有开始执行
2. RUNNABLE - 可运行状态的Thread，包括准备运行和正在运行的。
3. BLOCKED - 正在等待资源锁的线程
4. WAITING - 正在无限期等待其他线程来执行某个特定操作
5. TIMED_WAITING - 在一定的时间内等待其他线程来执行某个特定操作
6. TERMINATED - 线程执行完毕

我们可以用一个图来直观的表示：



JDK代码中的定义如下：

```
public enum State {
        /**
         * Thread state for a thread which has not yet started.
         */
        NEW,

        /**
         * Thread state for a runnable thread.  A thread in the runnable
         * state is executing in the Java virtual machine but it may
         * be waiting for other resources from the operating system
         * such as processor.
         */
        RUNNABLE,

        /**
         * Thread state for a thread blocked waiting for a monitor lock.
         * A thread in the blocked state is waiting for a monitor lock
         * to enter a synchronized block/method or
         * reenter a synchronized block/method after calling
         * {@link Object#wait() Object.wait}.
         */
        BLOCKED,

        /**
         * Thread state for a waiting thread.
         * A thread is in the waiting state due to calling one of the
         * following methods:
```

```
         * <ul>
         *   <li>{@link Object#wait() Object.wait} with no timeout</li>
         *   <li>{@link #join() Thread.join} with no timeout</li>
         *   <li>{@link LockSupport#park() LockSupport.park}</li>
         * </ul>
         *
         * <p>A thread in the waiting state is waiting for another thread to
         * perform a particular action.
         *
         * For example, a thread that has called <tt>Object.wait()</tt>
         * on an object is waiting for another thread to call
         * <tt>Object.notify()</tt> or <tt>Object.notifyAll()</tt> on
         * that object. A thread that has called <tt>Thread.join()</tt>
         * is waiting for a specified thread to terminate.
         */
        WAITING,

        /**
         * Thread state for a waiting thread with a specified waiting time.
         * A thread is in the timed waiting state due to calling one of
         * the following methods with a specified positive waiting time:
         * <ul>
         *   <li>{@link #sleep Thread.sleep}</li>
         *   <li>{@link Object#wait(long) Object.wait} with timeout</li>
         *   <li>{@link #join(long) Thread.join} with timeout</li>
         *   <li>{@link LockSupport#parkNanos LockSupport.parkNanos}</li>
         *   <li>{@link LockSupport#parkUntil LockSupport.parkUntil}</li>
         * </ul>
         */
        TIMED_WAITING,

        /**
         * Thread state for a terminated thread.
         * The thread has completed execution.
         */
        TERMINATED;
    }
```

NEW

NEW 表示线程创建了，但是还没有开始执行。我们看一个NEW的例子：

```
public class NewThread implements Runnable{
    public static void main(String[] args) {
        Runnable runnable = new NewThread();
        Thread t = new Thread(runnable);
        log.info(t.getState().toString());
    }

    @Override
    public void run() {

    }
}
```

上面的代码将会输出：

```
NEW
```

Runnable

Runnable表示线程正在可执行状态。包括正在运行和准备运行两种。

为什么这两种都叫做Runnable呢？我们知道在多任务环境中，CPU的个数是有限的，所以任务都是轮循占有CPU来处理的，JVM中的线程调度器会为每个线程分配特定的执行时间，当执行时间结束后，线程调度器将会释放CPU，以供其他的Runnable线程执行。

我们看一个Runnable的例子：

```
public class RunnableThread implements Runnable {
    @Override
    public void run() {

    }

    public static void main(String[] args) {
        Runnable runnable = new RunnableThread();
        Thread t = new Thread(runnable);
        t.start();
        log.info(t.getState().toString());
    }
}
```

上面的代码将会输出：

```
RUNNABLE
```

BLOCKED

BLOCKED表示线程正在等待资源锁，而目前该资源正在被其他线程占有。

我们举个例子：

```
public class BlockThread implements Runnable {
    @Override
    public void run() {
        loopResource();
    }

    public static synchronized void loopResource() {
        while(true) {
            //无限循环
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new BlockThread());
        Thread t2 = new Thread(new BlockThread());

        t1.start();
        t2.start();

        Thread.sleep(1000);
        log.info(t1.getState().toString());
        log.info(t2.getState().toString());
        System.exit(0);
    }
}
```

上面的例子中，由于t1是无限循环，将会一直占有资源锁，导致t2无法获取资源锁，从而位于BLOCKED状态。

我们会得到如下结果：

```
12:40:11.710 [main] INFO com.flydean.BlockThread - RUNNABLE
12:40:11.713 [main] INFO com.flydean.BlockThread - BLOCKED
```

WAITING

WAITING 状态表示线程正在等待其他的线程执行特定的操作。有三种方法可以导致线程处于WAITTING状态：

1. object.wait()
2. thread.join()
3. LockSupport.park()

其中1，2方法不需要传入时间参数。

我们看下使用的例子：

```
public class WaitThread implements  Runnable{
```

```
    public static Thread t1;
    @Override
    public void run() {
        Thread t2 = new Thread(()->{
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                log.error("Thread interrupted", e);
            }
            log.info("t1"+t1.getState().toString());
        });
        t2.start();

        try {
            t2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            log.error("Thread interrupted", e);
        }
        log.info("t2"+t2.getState().toString());
    }

    public static void main(String[] args) {
        t1 = new Thread(new WaitThread());
        t1.start();

    }
}
```

在这个例子中，我们调用的t2.join()，这会使调用它的t1线程处于WAITTING状态。

我们看下输出结果：

```
12:44:12.958 [Thread-1] INFO com.flydean.WaitThread - t1 WAITING
12:44:12.964 [Thread-0] INFO com.flydean.WaitThread - t2 TERMINATED
```

TIMED_WAITING

TIMED_WAITING状态表示在一个有限的时间内等待其他线程执行特定的某些操作。

java中有5中方式来达到这种状态：

1. thread.sleep(long millis)
2. wait(int timeout) 或者 wait(int timeout, int nanos)
3. thread.join(long millis)
4. LockSupport.parkNanos
5. LockSupport.parkUntil

我们举个例子：

```
public class TimedWaitThread implements  Runnable{
    @Override
    public void run() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            log.error("Thread interrupted", e);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        TimedWaitThread obj1 = new TimedWaitThread();
        Thread t1 = new Thread(obj1);
        t1.start();

        // The following sleep will give enough time for ThreadScheduler
        // to start processing of thread t1
        Thread.sleep(1000);
        log.info(t1.getState().toString());
    }
}
```

上面的例子中我们调用了Thread.sleep(5000)来让线程处于TIMED_WAITING状态。

看下输出：

```
12:58:02.706 [main] INFO com.flydean.TimedWaitThread - TIMED_WAITING
```

那么问题来了，TIMED_WAITING和WAITTING有什么区别呢?

TIMED_WAITING如果在给定的时间内没有等到其他线程的特定操作，则会被唤醒，从而进入争夺资源锁的队列，如果能够获取到锁，则会变成Runnable状态，如果获取不到锁，则会变成BLOCKED状态。

TERMINATED

TERMINATED表示线程已经执行完毕。我们看下例子：

```
public class TerminatedThread implements Runnable{
    @Override
    public void run() {

    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new TerminatedThread());
        t1.start();
```

```
        // The following sleep method will give enough time for
        // thread t1 to complete
        Thread.sleep(1000);
        log.info(t1.getState().toString());
    }
}
```

输出结果：

```
13:02:38.868 [main] INFO com.flydean.TerminatedThread - TERMINATED
```

第十章 java中join的使用

join()应该是我们在java中经常会用到的一个方法，它主要是将当前线程置为WAITTING状态，然后等待调用的线程执行完毕或被interrupted。

join()是Thread中定义的方法，我们看下他的定义：

```
   /**
     * Waits for this thread to die.
     *
     * <p> An invocation of this method behaves in exactly the same
     * way as the invocation
     *
     * <blockquote>
     * {@linkplain #join(long) join}{@code (0)}
     * </blockquote>
     *
     * @throws  InterruptedException
     *          if any thread has interrupted the current thread. The
     *          <i>interrupted status</i> of the current thread is
     *          cleared when this exception is thrown.
     */
    public final void join() throws InterruptedException {
        join(0);
    }
```

我们看下join是怎么使用的，通常我们需要在线程A中调用线程B.join():

```
public class JoinThread implements Runnable{
    public int processingCount = 0;

    JoinThread(int processingCount) {
        this.processingCount = processingCount;
        log.info("Thread Created");
    }
```

```
    @Override
    public void run() {
        log.info("Thread " + Thread.currentThread().getName() + " started");
        while (processingCount > 0) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                log.info("Thread " + Thread.currentThread().getName() + "
interrupted");
            }
            processingCount--;
        }
        log.info("Thread " + Thread.currentThread().getName() + " exiting");
    }

    @Test
    public void joinTest()
            throws InterruptedException {
        Thread t2 = new Thread(new JoinThread(1));
        t2.start();
        log.info("Invoking join");
        t2.join();
        log.info("Returned from join");
        log.info("t2 status {}",t2.isAlive());
    }
}
```

我们在主线程中调用了t2.join(),则主线程将会等待t2执行完毕，我们看下输出结果：

```
06:17:14.775 [main] INFO com.flydean.JoinThread - Thread Created
06:17:14.779 [main] INFO com.flydean.JoinThread - Invoking join
06:17:14.779 [Thread-0] INFO com.flydean.JoinThread - Thread Thread-0 started
06:17:15.783 [Thread-0] INFO com.flydean.JoinThread - Thread Thread-0 exiting
06:17:15.783 [main] INFO com.flydean.JoinThread - Returned from join
06:17:15.783 [main] INFO com.flydean.JoinThread - t2 status false
```

当线程已经执行完毕或者还没开始执行的时候，join（）将会立即返回：

```
Thread t1 = new SampleThread(0);
t1.join();  //returns immediately
```

join还有两个带时间参数的方法：

```
public final void join(long millis) throws InterruptedException
```

```
public final void join(long millis,int nanos) throws InterruptedException
```

如果在给定的时间内调用的线程没有返回，则主线程将会继续执行：

```
    @Test
    public void testJoinTimeout()
            throws InterruptedException {
        Thread t3 =  new Thread(new JoinThread(10));
        t3.start();
        t3.join(1000);
        log.info("t3 status {}", t3.isAlive());
    }
```

上面的例子将会输出：

```
06:30:58.159 [main] INFO com.flydean.JoinThread - Thread Created
06:30:58.163 [Thread-0] INFO com.flydean.JoinThread - Thread Thread-0 started
06:30:59.172 [main] INFO com.flydean.JoinThread - t3 status true
```

Join()还有个happen-before的特性，这就是如果thread t1调用 t2.join(), 那么当t2返回时，所有t2的变动都会t1可见。

之前我们讲volatile关键词的时候也提到了这个happen-before规则。 我们看下例子：

```
    @Test
    public void testHappenBefore() throws InterruptedException {
        JoinThread t4 =  new JoinThread(10);
        t4.start();
        // not guaranteed to stop even if t4 finishes.
        do {
            log.info("inside the loop");
            Thread.sleep(1000);
        } while ( t4.processingCount > 0);
    }
```

我们运行下，可以看到while循环一直在进行中，即使t4中的变量已经变成了0。

所以如果我们需要在这种情况下使用的话，我们需要用到join（），或者其他的同步机制。

第十一章 怎么在java中关闭一个thread

我们经常需要在java中用到thread，我们知道thread有一个start()方法可以开启一个线程。那么怎么关闭这个线程呢?

有人会说可以用Thread.stop（）方法。但是这个方法已经被废弃了。

根据Oracle的官方文档，Thread.stop是不安全的。因为调用stop方法的时候，将会释放它获取的所有监视器锁（通过传递ThreadDeath异常实现）。如果有资源该监视器锁所保护的话，就可能会出现数据不一致的异常。并且这种异常很难被发现。 所以现在已经不推荐是用Thread.stop方法了。

那我们还有两种方式来关闭一个Thread。

1. Flag变量

如果我们有一个无法自动停止的Thread，我们可以创建一个条件变量，通过不断判断该变量的值，来决定是否结束该线程的运行。

```
public class KillThread implements Runnable {
    private Thread worker;
    private final AtomicBoolean running = new AtomicBoolean(false);
    private int interval;

    public KillThread(int sleepInterval) {
        interval = sleepInterval;
    }

    public void start() {
        worker = new Thread(this);
        worker.start();
    }

    public void stop() {
        running.set(false);
    }

    public void run() {
        running.set(true);
        while (running.get()) {
            try {
                Thread.sleep(interval);
            } catch (InterruptedException e){
                Thread.currentThread().interrupt();
                log.info("Thread was interrupted, Failed to complete operation");
            }
            // do something here
        }
        log.info("finished");
    }

    public static void main(String[] args) {
        KillThread killThread= new KillThread(1000);
        killThread.start();
        killThread.stop();
    }


}
```

上面的例子中，我们通过定义一个AtomicBoolean 的原子变量来存储Flag标志。

我们将会在后面的文章中详细的讲解原子变量。

2. 调用interrupt()方法

通过调用interrupt()方法，将会中断正在等待的线程，并抛出InterruptedException异常。

根据Oracle的说明，如果你想自己处理这个异常的话，需要reasserts出去，注意，这里是reasserts而不是rethrows，因为有些情况下，无法rethrow这个异常，我们需要这样做：

```
Thread.currentThread().interrupt();
```

这将会reasserts InterruptedException异常。

看下我们第二种方法怎么调用：

```
public class KillThread implements Runnable {
    private Thread worker;
    private final AtomicBoolean running = new AtomicBoolean(false);
    private int interval;

    public KillThread(int sleepInterval) {
        interval = sleepInterval;
    }

    public void start() {
        worker = new Thread(this);
        worker.start();
    }

    public void interrupt() {
        running.set(false);
        worker.interrupt();
    }

    public void stop() {
        running.set(false);
    }

    public void run() {
        running.set(true);
        while (running.get()) {
            try {
                Thread.sleep(interval);
            } catch (InterruptedException e){
                Thread.currentThread().interrupt();
                log.info("Thread was interrupted, Failed to complete operation");
            }
            // do something here
        }
        log.info("finished");
    }
```

```
    public static void main(String[] args) {
        KillThread killThread= new KillThread(1000);
        killThread.start();
        killThread.interrupt();
    }
}
```

上面的例子中，当线程在Sleep中时，调用了interrupt方法，sleep会退出，并且抛出InterruptedException异常。

第十二章 java中的Atomic类

问题背景

在多线程环境中，我们最常遇到的问题就是变量的值进行同步。因为变量需要在多线程中进行共享，所以我们必须需要采用一定的同步机制来进行控制。

通过之前的文章，我们知道可以采用Lock的机制，当然也包括今天我们讲的Atomic类。

下面我们从两种方式来分别介绍。

Lock

在之前的文章中，我们也讲了同步的问题，我们再回顾一下。 如果定义了一个计数器如下：

```
public class Counter {

    int counter;

    public void increment() {
        counter++;
    }

}
```

如果是在单线程环境中，上面的代码没有任何问题。但是如果在多线程环境中，counter++将会得到不同的结果。

因为虽然counter++看起来是一个原子操作，但是它实际上包含了三个操作：读数据，加一，写回数据。

我们之前的文章也讲了，如何解决这个问题：

```
public class LockCounter {

    private volatile int counter;

    public synchronized void increment() {
        counter++;
    }
}
```

通过加synchronized，保证同一时间只会有一个线程去读写counter变量。

通过volatile，保证所有的数据直接操作的主缓存，而不使用线程缓存。

这样虽然解决了问题，但是性能可能会受影响，因为synchronized会锁住整个LockCounter实例。

使用Atomic

通过引入低级别的原子化语义命令（比如compare-and-swap (CAS)），从而能在保证效率的同时保证原子性。

一个标准的CAS包含三个操作：

1. 将要操作的内存地址M。
2. 现有的变量A。
3. 新的需要存储的变量B。

CAS将会先比较A和M中存储的值是否一致，一致则表示其他线程未对该变量进行修改，则将其替换为B。 否则不做任何操作。

使用CAS可以不用阻塞其他的线程，但是我们需要自己处理好当更新失败的情况下的业务逻辑处理情况。

Java提供了很多Atomic类，最常用的包括AtomicInteger, AtomicLong, AtomicBoolean, 和 AtomicReference.

其中的主要方法：

1. get() – 直接中主内存中读取变量的值，类似于volatile变量。
2. set() – 将变量写回主内存。类似于volatile变量。
3. lazySet() – 延迟写回主内存。一种常用的情景是将引用重置为null的情况。
4. compareAndSet() – 执行CAS操作，成功返回true，失败返回false。
5. weakCompareAndSet() – 比较弱的CAS操作，不同的是它不执行happens-before操作，从而不保证能够读取到其他变量最新的值。

我们看下怎么用：

```
public class AtomicCounter {

    private final AtomicInteger counter = new AtomicInteger(0);

    public int getValue() {
        return counter.get();
    }
    public void increment() {
        while(true) {
            int existingValue = getValue();
            int newValue = existingValue + 1;
            if(counter.compareAndSet(existingValue, newValue)) {
                return;
            }
        }
    }
}
```

第十三章 java中interrupt，interrupted和isInterrupted的区别

前面的文章我们讲到了调用interrupt()来停止一个Thread，本文将会详细讲解java中三个非常相似的方法interrupt，interrupted和isInterrupted。

isInterrupted

首先看下最简单的isInterrupted方法。isInterrupted是Thread类中的一个实例方法：

```
    public boolean isInterrupted() {
        return isInterrupted(false);
    }
```

通过调用isInterrupted（）可以判断实例线程是否被中断。

它的内部调用了isInterrupted(false)方法：

```
/**
     * Tests if some Thread has been interrupted.  The interrupted state
     * is reset or not based on the value of ClearInterrupted that is
     * passed.
     */
    private native boolean isInterrupted(boolean ClearInterrupted);
```

这个方法是个native方法，接收一个是否清除Interrupted标志位的参数。

我们可以看到isInterrupted()传入的参数是false，这就表示isInterrupted()只会判断是否被中断，而不会清除中断状态。

interrupted

interrupted是Thread中的一个类方法：

```
public static boolean interrupted() {
        return currentThread().isInterrupted(true);
    }
```

我们可以看到，interrupted（）也调用了isInterrupted(true)方法，不过它传递的参数是true，表示将会清除中断标志位。

> 注意，因为interrupted()是一个类方法，调用isInterrupted(true)判断的是当前线程是否被中断。注意这里currentThread（）的使用。

interrupt

前面两个是判断是否中断的方法，而interrupt（）就是真正触发中断的方法。

我们先看下interrupt的定义：

```
    public void interrupt() {
        if (this != Thread.currentThread())
            checkAccess();

        synchronized (blockerLock) {
            Interruptible b = blocker;
            if (b != null) {
                interrupt0();           // Just to set the interrupt flag
                b.interrupt(this);
                return;
            }
        }
        interrupt0();
    }
```

从定义我们可以看到interrupt（）是一个实例方法。

它的工作要点有下面4点：

1. 如果当前线程实例在调用Object类的wait（），wait（long）或wait（long，int）方法或join（），join（long），join（long，int）方法，或者在该实例中调用了Thread.sleep（long）或Thread.sleep（long，int）方法，并且正在阻塞状态中时，则其中断状态将被清除，并将收到InterruptedException。
2. 如果此线程在InterruptibleChannel上的I / O操作中处于被阻塞状态，则该channel将被关闭，该线程的中断状态将被设置为true，并且该线程将收到java.nio.channels.ClosedByInterruptException异常。
3. 如果此线程在java.nio.channels.Selector中处于被被阻塞状态，则将设置该线程的中断状态为true，并且它将立即从select操作中返回。
4. 如果上面的情况都不成立，则设置中断状态为true。

我们来举个例子：

```
@Slf4j
public class InterruptThread extends Thread {
    @Override
    public  void run() {
        for (int i = 0; i < 1000; i++) {
            log.info("i= {}", (i+1));
            log.info("call inside thread.interrupted():  {}", Thread.interrupted());
        }
    }

    @Test
    public void testInterrupt(){
        InterruptThread thread=new InterruptThread();
        thread.start();
        thread.interrupt();
        //test isInterrupted
        log.info("first call isInterrupted(): {}", thread.isInterrupted());
```

```
        log.info("second call isInterrupted(): {}", thread.isInterrupted());

        //test interrupted ()
        log.info("first call outside thread.interrupted():  {}", Thread.interrupted());
        log.info("second call outside thread.interrupted() {}: ", Thread.interrupted());
        log.info("thread is alive : {}",thread.isAlive() );
    }
}
```

输出结果如下：

```
13:07:17.804 [main] INFO com.flydean.InterruptThread - first call isInterrupted(): true
13:07:17.808 [main] INFO com.flydean.InterruptThread - second call isInterrupted():
true

13:07:17.808 [Thread-1] INFO com.flydean.InterruptThread - call inside
thread.interrupted():  true
13:07:17.808 [Thread-1] INFO com.flydean.InterruptThread - call inside
thread.interrupted():  false

13:07:17.808 [main] INFO com.flydean.InterruptThread - first call outside
thread.interrupted():  false
13:07:17.809 [main] INFO com.flydean.InterruptThread - second call outside
thread.interrupted() false
```

上面的例子中，两次调用thread.isInterrupted()的值都是true。

在线程内部调用Thread.interrupted()， 只有第一次返回的是ture，后面返回的都是false，这表明第一次被重置了。

在线程外部，因为并没有中断外部线程，所以返回的值一直都是false。

第十四章 java中的daemon thread

java中有两种类型的thread，user threads 和 daemon threads。

User threads是高优先级的thread，JVM将会等待所有的User Threads运行完毕之后才会结束运行。

daemon threads是低优先级的thread，它的作用是为User Thread提供服务。 因为daemon threads的低优先级，并且仅为user thread提供服务，所以当所有的user thread都结束之后，JVM会自动退出，不管是否还有daemon threads在运行中。

因为这个特性，所以我们通常在daemon threads中处理无限循环的操作，因为这样不会影响user threads的运行。

daemon threads并不推荐使用在I/O操作中。

但是有些不当的操作也可能导致daemon threads阻塞JVM关闭，比如在daemon thread中调用join（）方法。

我们看下怎么创建daemon thread：

```
public class DaemonThread extends Thread{

    public void  run(){
        while(true){
            log.info("Thread A run");
            try {
                log.info("Thread A is daemon {}" ,Thread.currentThread().isDaemon());
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }


    public static void main(String[] args) throws InterruptedException {
        DaemonThread daemonThread = new DaemonThread();
        daemonThread.setDaemon(true);
        daemonThread.start();
    }
}
```

创建 daemon thread很简单，只需要在创建之后，设置其daemon属性为true即可。

> 注意setDaemon(true)必须在thread start（）之前执行，否则会抛出IllegalThreadStateException

上面的例子将会立刻退出。

如果我们将daemonThread.setDaemon(true);去掉，则因为user thread一直执行，JVM将会一直运行下去，不会退出。

这是在main中运行的情况，如果我们在一个@Test中运行，会发生什么现象呢?

```
public class DaemonThread extends Thread{

    public void  run(){
        while(true){
            log.info("Thread A run");
            try {
                log.info("Thread A is daemon {}" ,Thread.currentThread().isDaemon());
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    @Test
    public void testDaemon() throws InterruptedException {
        DaemonThread daemonThread = new DaemonThread();
```

```
        daemonThread.start();
    }
}
```

我们将main方法改成了@Test执行。执行之后我们会发现，不管是不是daemon thread， Test都会立刻结束。

再看一个daemon线程中启动一个user thread的情况：

```
public class DaemonBThread extends Thread{

    Thread worker = new Thread(()->{
        while(true){
            log.info("Thread B run");
            log.info("Thread B is daemon {}",Thread.currentThread().isDaemon());
        }
    });
    public void  run(){
            log.info("Thread A run");
            worker.start();
    }

    public static void main(String[] args) {
        DaemonBThread daemonThread = new DaemonBThread();
        daemonThread.setDaemon(true);
        daemonThread.start();
    }
}
```

这个例子中，daemonThread启动了一个user thread，运行之后我们会发现，即使有user thread正在运行，JVM也会立刻结束执行。

第十五章 java中ThreadPool的介绍和使用

Thread Pool简介

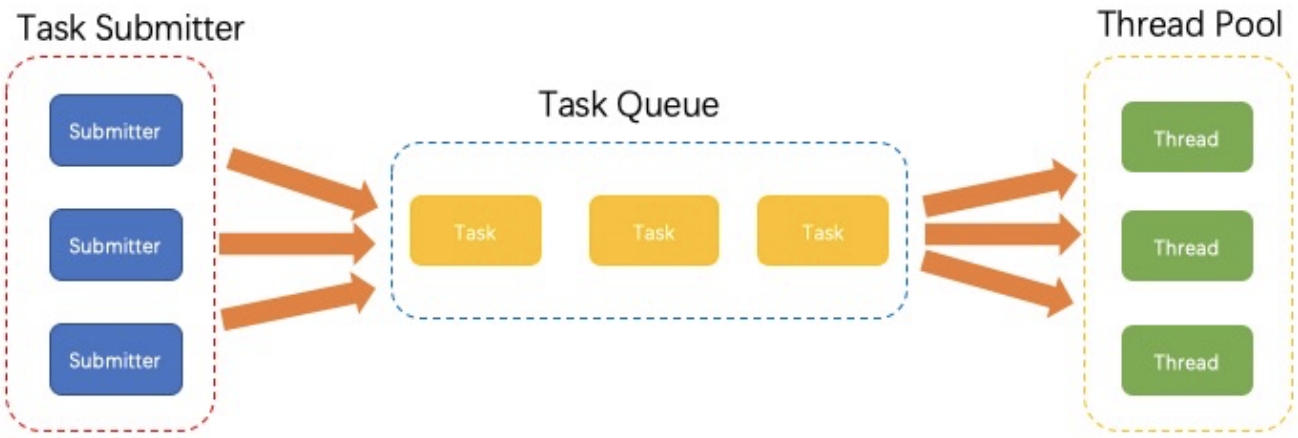
在Java中，threads是和系统的threads相对应的，用来处理一系列的系统资源。不管在windows和linux下面，能开启的线程个数都是有限的，如果你在java程序中无限制的创建thread，那么将会遇到无线程可创建的情况。

CPU的核数是有限的，如果同时有多个线程正在运行中，那么CPU将会根据线程的优先级进行轮循，给每个线程分配特定的CPU时间。所以线程也不是越多越好。

在java中，代表管理ThreadPool的接口有两个：ExecutorService和Executor。

我们运行线程的步骤一般是这样的：1. 创建一个ExecutorService。 2.将任务提交给ExecutorService。 3.ExecutorService调度线程来运行任务。

画个图来表示：



下面我讲一下，怎么在java中使用ThreadPool。

Executors, Executor 和 ExecutorService

Executors 提供了一系列简便的方法，来帮助我们创建ThreadPool。

Executor接口定义了一个方法：

```
public interface Executor {

    /**
     * Executes the given command at some time in the future.  The command
     * may execute in a new thread, in a pooled thread, or in the calling
     * thread, at the discretion of the {@code Executor} implementation.
     *
     * @param command the runnable task
     * @throws RejectedExecutionException if this task cannot be
     * accepted for execution
     * @throws NullPointerException if command is null
     */
    void execute(Runnable command);
}
```

ExecutorService继承了Executor，提供了更多的线程池的操作。是对Executor的补充。

根据接口实现分离的原则，我们通常在java代码中使用ExecutorService或者Executor，而不是具体的实现类。

我们看下怎么通过Executors来创建一个Executor和ExecutorService：

```
        Executor executor = Executors.newSingleThreadExecutor();
        executor.execute(() -> log.info("in Executor"));


        ExecutorService executorService= Executors.newCachedThreadPool();
        executorService.submit(()->log.info("in ExecutorService"));
        executorService.shutdown();
```

关于ExecutorService的细节，我们这里就多讲了，感兴趣的朋友可以参考之前我写的ExecutorService的详细文章。

ThreadPoolExecutor

ThreadPoolExecutor是ExecutorService接口的一个实现，它可以为线程池添加更加精细的配置，具体而言它可以控制这三个参数：corePoolSize, maximumPoolSize, 和 keepAliveTime。

PoolSize就是线程池里面的线程个数，corePoolSize表示的是线程池里面初始化和保持的最小的线程个数。

如果当前等待线程太多，可以设置maximumPoolSize来提供最大的线程池个数，从而线程池会创建更多的线程以供任务执行。

keepAliveTime是多余的线程未分配任务将会等待的时间。超出该时间，线程将会被线程池回收。

我们看下怎么创建一个ThreadPoolExecutor：

```
        ThreadPoolExecutor threadPoolExecutor =
                new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,
                        new LinkedBlockingQueue<Runnable>());
        threadPoolExecutor.submit(()->log.info("submit through threadPoolExecutor"));
        threadPoolExecutor.shutdown();
```

上面的例子中我们通过ThreadPoolExecutor的构造函数来创建ThreadPoolExecutor。

通常来说Executors已经内置了ThreadPoolExecutor的很多实现，我们来看下面的例子：

```
ThreadPoolExecutor executor1 =
                (ThreadPoolExecutor) Executors.newFixedThreadPool(2);
        executor1.submit(() -> {
            Thread.sleep(1000);
            return null;
        });
        executor1.submit(() -> {
            Thread.sleep(1000);
            return null;
        });
        executor1.submit(() -> {
            Thread.sleep(1000);
            return null;
        });
        log.info("executor1 poolsize {}",executor1.getPoolSize());
```

```
        log.info("executor1 queuesize {}", executor1.getQueue().size());
        executor1.shutdown();
```

上的例子中我们Executors.newFixedThreadPool(2)来创建一个ThreadPoolExecutor。

上面的例子中我们提交了3个task。但是我们pool size只有2。所以还有一个1个不能立刻被执行，需要在queue中等待。

我们再看一个例子：

```
ThreadPoolExecutor executor2 =
                (ThreadPoolExecutor) Executors.newCachedThreadPool();
        executor2.submit(() -> {
            Thread.sleep(1000);
            return null;
        });
        executor2.submit(() -> {
            Thread.sleep(1000);
            return null;
        });
        executor2.submit(() -> {
            Thread.sleep(1000);
            return null;
        });

        log.info("executor2 poolsize {}", executor2.getPoolSize());
        log.info("executor2 queue size {}", executor2.getQueue().size());
        executor2.shutdown();
```

上面的例子中我们使用Executors.newCachedThreadPool()来创建一个ThreadPoolExecutor。 运行之后我们可以看到poolsize是3，而queue size是0。这表明newCachedThreadPool会自动增加pool size。

如果thread在60秒钟之类没有被激活，则会被收回。

这里的Queue是一个SynchronousQueue，因为插入和取出基本上是同时进行的，所以这里的queue size基本都是0.

ScheduledThreadPoolExecutor

还有个很常用的ScheduledThreadPoolExecutor，它继承自ThreadPoolExecutor, 并且实现了ScheduledExecutorService接口。

```
public class ScheduledThreadPoolExecutor
        extends ThreadPoolExecutor
        implements ScheduledExecutorService
```

我们看下怎么使用：

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(5);
        executor.schedule(() -> {
            log.info("Hello World");
        }, 500, TimeUnit.MILLISECONDS);
```

上面的例子中，我们定义了一个定时任务将会在500毫秒之后执行。

之前我们也讲到了ScheduledExecutorService还有两个非常常用的方法：

- scheduleAtFixedRate - 以开始时间为间隔。
- scheduleWithFixedDelay - 以结束时间为间隔。

```
CountDownLatch lock = new CountDownLatch(3);

        ScheduledExecutorService executor2 = Executors.newScheduledThreadPool(5);
        ScheduledFuture<?> future = executor2.scheduleAtFixedRate(() -> {
            log.info("in ScheduledFuture");
            lock.countDown();
        }, 500, 100, TimeUnit.MILLISECONDS);

        lock.await(1000, TimeUnit.MILLISECONDS);
        future.cancel(true);
```

ForkJoinPool

ForkJoinPool是在java 7 中引入的新框架，我们将会在后面的文章中详细讲解。 这里做个简单的介绍。

ForkJoinPool主要用来生成大量的任务来做算法运算。如果用线程来做的话，会消耗大量的线程。但是在fork/join框架中就不会出现这个问题。

在fork/join中，任何task都可以生成大量的子task，然后通过使用join（）等待子task结束。

这里我们举一个例子：

```
static class TreeNode {

    int value;

    Set<TreeNode> children;

    TreeNode(int value, TreeNode... children) {
        this.value = value;
        this.children = Sets.newHashSet(children);
    }
}
```

定义一个TreeNode，然后遍历所有的value，将其加起来：

```
public  class CountingTask extends RecursiveTask<Integer> {

    private final TreeNode node;

    public CountingTask(TreeNode node) {
        this.node = node;
    }

    @Override
    protected Integer compute() {
        return node.value + node.children.stream()
                .map(childNode -> new
CountingTask(childNode).fork()).mapToInt(ForkJoinTask::join).sum();
    }
}
```

下面是调用的代码：

```
    public static void main(String[] args) {
        TreeNode tree = new TreeNode(5,
                new TreeNode(3), new TreeNode(2,
                new TreeNode(2), new TreeNode(8)));

        ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
        int sum = forkJoinPool.invoke(new CountingTask(tree));
    }
```

第十六章 java 中的fork join框架

fork join框架是java 7中引入框架，这个框架的引入主要是为了提升并行计算的能力。

fork join主要有两个步骤，第一就是fork，将一个大任务分成很多个小任务，第二就是join，将第一个任务的结果join起来，生成最后的结果。如果第一步中并没有任何返回值，join将会等到所有的小任务都结束。

还记得之前的文章我们讲到了thread pool的基本结构吗?

1. ExecutorService - ForkJoinPool 用来调用任务执行。
2. workerThread - ForkJoinWorkerThread 工作线程，用来执行具体的任务。
3. task - ForkJoinTask 用来定义要执行的任务。

下面我们从这三个方面来详细讲解fork join框架。

ForkJoinPool

ForkJoinPool是一个ExecutorService的一个实现，它提供了对工作线程和线程池的一些便利管理方法。

```
public class ForkJoinPool extends AbstractExecutorService
```

一个work thread一次只能处理一个任务，但是ForkJoinPool并不会为每个任务都创建一个单独的线程，它会使用一个特殊的数据结构double-ended queue来存储任务。这样的结构可以方便的进行工作窃取（work-stealing）。

什么是work-stealing呢?

默认情况下，work thread从分配给自己的那个队列头中取出任务。如果这个队列是空的，那么这个work thread会从其他的任务队列尾部取出任务来执行，或者从全局队列中取出。这样的设计可以充分利用work thread的性能，提升并发能力。

下面看下怎么创建一个ForkJoinPool。

最常见的方法就是使用ForkJoinPool.commonPool()来创建，commonPool()为所有的ForkJoinTask提供了一个公共默认的线程池。

```
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
```

另外一种方式是使用构造函数：

```
ForkJoinPool forkJoinPool = new ForkJoinPool(2);
```

这里的参数是并行级别，2指的是线程池将会使用2个处理器核心。

ForkJoinWorkerThread

ForkJoinWorkerThread是使用在ForkJoinPool的工作线程。

```
public class ForkJoinWorkerThread extends Thread
}
```

和一般的线程不一样的是它定义了两个变量：

```
    final ForkJoinPool pool;                // the pool this thread works in
    final ForkJoinPool.WorkQueue workQueue; // work-stealing mechanics
```

一个是该worker thread所属的ForkJoinPool。 另外一个是支持 work-stealing机制的Queue。

再看一下它的run方法：

```
   public void run() {
        if (workQueue.array == null) { // only run once
            Throwable exception = null;
            try {
                onStart();
                pool.runWorker(workQueue);
            } catch (Throwable ex) {
                exception = ex;
            } finally {
                try {
                    onTermination(exception);
```

```
                } catch (Throwable ex) {
                    if (exception == null)
                        exception = ex;
                } finally {
                    pool.deregisterWorker(this, exception);
                }
            }
        }
    }
```

简单点讲就是从Queue中取出任务执行。

ForkJoinTask

ForkJoinTask是ForkJoinPool中运行的任务类型。通常我们会用到它的两个子类：RecursiveAction和RecursiveTask。

他们都定义了一个需要实现的compute（）方法用来实现具体的业务逻辑。不同的是RecursiveAction只是用来执行任务，而RecursiveTask可以有返回值。

既然两个类都带了Recursive，那么具体的实现逻辑也会跟递归有关，我们举个使用RecursiveAction来打印字符串的例子：

```
public class CustomRecursiveAction extends RecursiveAction {

    private String workload = "";
    private static final int THRESHOLD = 4;

    private static Logger logger =
            Logger.getAnonymousLogger();

    public CustomRecursiveAction(String workload) {
        this.workload = workload;
    }

    @Override
    protected void compute() {
        if (workload.length() > THRESHOLD) {
            ForkJoinTask.invokeAll(createSubtasks());
        } else {
            processing(workload);
        }
    }

    private List<CustomRecursiveAction> createSubtasks() {
        List<CustomRecursiveAction> subtasks = new ArrayList<>();

        String partOne = workload.substring(0, workload.length() / 2);
        String partTwo = workload.substring(workload.length() / 2, workload.length());
```

```
        subtasks.add(new CustomRecursiveAction(partOne));
        subtasks.add(new CustomRecursiveAction(partTwo));

        return subtasks;
    }

    private void processing(String work) {
        String result = work.toUpperCase();
        logger.info("This result - (" + result + ") - was processed by "
                + Thread.currentThread().getName());
    }
}
```

上面的例子使用了二分法来打印字符串。

我们再看一个RecursiveTask的例子：

```
public class CustomRecursiveTask extends RecursiveTask<Integer> {
    private int[] arr;

    private static final int THRESHOLD = 20;

    public CustomRecursiveTask(int[] arr) {
        this.arr = arr;
    }

    @Override
    protected Integer compute() {
        if (arr.length > THRESHOLD) {
            return ForkJoinTask.invokeAll(createSubtasks())
                    .stream()
                    .mapToInt(ForkJoinTask::join)
                    .sum();
        } else {
            return processing(arr);
        }
    }

    private Collection<CustomRecursiveTask> createSubtasks() {
        List<CustomRecursiveTask> dividedTasks = new ArrayList<>();
        dividedTasks.add(new CustomRecursiveTask(
                Arrays.copyOfRange(arr, 0, arr.length / 2)));
        dividedTasks.add(new CustomRecursiveTask(
                Arrays.copyOfRange(arr, arr.length / 2, arr.length)));
        return dividedTasks;
    }

    private Integer processing(int[] arr) {
        return Arrays.stream(arr)
```

```
                .filter(a -> a > 10 && a < 27)
                .map(a -> a * 10)
                .sum();
    }
}
```

和上面的例子很像，不过这里我们需要有返回值。

在ForkJoinPool中提交Task

有了上面的两个任务，我们就可以在ForkJoinPool中提交了：

```
int[] intArray= {12,12,13,14,15};
        CustomRecursiveTask customRecursiveTask= new CustomRecursiveTask(intArray);

        int result = forkJoinPool.invoke(customRecursiveTask);
        System.out.println(result);
```

上面的例子中，我们使用invoke来提交，invoke将会等待任务的执行结果。

如果不使用invoke，我们也可以将其替换成fork（）和join（）：

```
customRecursiveTask.fork();
        int result2= customRecursiveTask.join();
        System.out.println(result2);
```

fork() 是将任务提交给pool，但是并不触发执行， join()将会真正的执行并且得到返回结果。

第十七章 java并发中CountDownLatch的使用

在java并发中，控制共享变量的访问非常重要，有时候我们也想控制并发线程的执行顺序，比如：等待所有线程都执行完毕之后再执行另外的线程，或者等所有线程都准备好了才开始所有线程的执行等。

这个时候我们就可以使用到CountDownLatch。

简单点讲，CountDownLatch存有一个放在QueuedSynchronizer中的计数器。当调用countdown() 方法时，该计数器将会减一。然后再调用await()来等待计数器归零。

```

private static final class Sync extends AbstractQueuedSynchronizer {
    ...
}

private final Sync sync;

    public void countDown() {
        sync.releaseShared(1);
    }
```

```
    public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }

    public boolean await(long timeout, TimeUnit unit)
        throws InterruptedException {
        return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }
```

下面我们举两个使用的例子:

主线程等待子线程全都结束之后再开始运行

这里我们定义子线程类，在子线程类里面，我们传入一个CountDownLatch用来计数，然后在子线程结束之前，调用该CountDownLatch的countDown方法。最后在主线程中调用await（）方法来等待子线程结束执行。

```
@Slf4j
public class MainThreadWaitUsage implements Runnable {

    private List<String> outputScraper;
    private CountDownLatch countDownLatch;

    public MainThreadWaitUsage(List<String> outputScraper, CountDownLatch
countDownLatch) {
        this.outputScraper = outputScraper;
        this.countDownLatch = countDownLatch;
    }

    @Override
    public void run() {
        outputScraper.add("Counted down");
        countDownLatch.countDown();
    }
}
```

看下怎么调用：

```
    @Test
    public void testCountDownLatch()
            throws InterruptedException {

        List<String> outputScraper = Collections.synchronizedList(new ArrayList<>());
        CountDownLatch countDownLatch = new CountDownLatch(5);
        List<Thread> workers = Stream
                .generate(() -> new Thread(new MainThreadWaitUsage(outputScraper,
countDownLatch)))
                .limit(5)
```

```
                .collect(toList());

        workers.forEach(Thread::start);
        countDownLatch.await();
        outputScraper.add("Latch released");
        log.info(outputScraper.toString());

    }
```

执行结果如下：

```
07:37:27.388 [main] INFO MainThreadWaitUsageTest - [Counted down, Counted down, Counted down, Counted down, Counted down, Latch released]
```

等待所有线程都准备好再一起执行

上面的例子中，我们是主线程等待子线程，那么在这个例子中，我们将会看看怎么子线程一起等待到准备好的状态，再一起执行。

思路也很简单，在子线程开始之后，将等待的子线程计数器减一，在主线程中await该计数器，等计数器归零之后，主线程再通知子线程运行。

```
public class ThreadWaitThreadUsage implements Runnable {

    private List<String> outputScraper;
    private CountDownLatch readyThreadCounter;
    private CountDownLatch callingThreadBlocker;
    private CountDownLatch completedThreadCounter;

    public ThreadWaitThreadUsage(
            List<String> outputScraper,
            CountDownLatch readyThreadCounter,
            CountDownLatch callingThreadBlocker,
            CountDownLatch completedThreadCounter) {

        this.outputScraper = outputScraper;
        this.readyThreadCounter = readyThreadCounter;
        this.callingThreadBlocker = callingThreadBlocker;
        this.completedThreadCounter = completedThreadCounter;
    }

    @Override
    public void run() {
        readyThreadCounter.countDown();
        try {
            callingThreadBlocker.await();
            outputScraper.add("Counted down");
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        } finally {
            completedThreadCounter.countDown();
        }
    }
}
```

看下怎么调用：

```
    @Test
    public void testCountDownLatch()
            throws InterruptedException {

        List<String> outputScraper = Collections.synchronizedList(new ArrayList<>());
        CountDownLatch readyThreadCounter = new CountDownLatch(5);
        CountDownLatch callingThreadBlocker = new CountDownLatch(1);
        CountDownLatch completedThreadCounter = new CountDownLatch(5);
        List<Thread> workers = Stream
                .generate(() -> new Thread(new ThreadWaitThreadUsage(
                        outputScraper, readyThreadCounter, callingThreadBlocker,
completedThreadCounter)))
                .limit(5)
                .collect(toList());

        workers.forEach(Thread::start);
        readyThreadCounter.await();
        outputScraper.add("Workers ready");
        callingThreadBlocker.countDown();
        completedThreadCounter.await();
        outputScraper.add("Workers complete");

        log.info(outputScraper.toString());

    }
```

输出结果如下：

```
07:41:47.861 [main] INFO ThreadWaitThreadUsageTest - [Workers ready, Counted down,
Counted down, Counted down, Counted down, Counted down, Workers complete]
```

停止CountdownLatch的await

如果我们调用await（）方法，该方法将会等待一直到count=0才结束。但是如果在线程执行过程中出现了异常，可能导致countdown方法执行不了。那么await（）方法可能会出现无限等待的情况。

这个时候我们可以使用：

```
    public boolean await(long timeout, TimeUnit unit)
        throws InterruptedException {
        return sync.tryAcquireSharedNanos(1, unit.toNanos(timeout));
    }
```

第十八章 java中CyclicBarrier的使用

CyclicBarrier是java 5中引入的线程安全的组件。它有一个barrier的概念，主要用来等待所有的线程都执行完毕，然后再去执行特定的操作。

假如我们有很多个线程，每个线程都计算出了一些数据，然后我们需要等待所有的线程都执行完毕，再把各个线程计算出来的数据加起来，的到最终的结果，那么我们就可以使用CyclicBarrier。

CyclicBarrier的方法

我们先看下CyclicBarrier的构造函数：

```
    public CyclicBarrier(int parties, Runnable barrierAction) {
        if (parties <= 0) throw new IllegalArgumentException();
        this.parties = parties;
        this.count = parties;
        this.barrierCommand = barrierAction;
    }

        public CyclicBarrier(int parties) {
        this(parties, null);
    }
```

CyclicBarrier有两个构造函数，第一个只接受一个参数，表示需要统一行动的线程个数。第二个参数叫做barrierAction，表示出发barrier是需要执行的方法。

其中barrierAction是一个Runnable，我们可以在其中定义最后需要执行的工作。

再看下重要await方法：

```
    public int await() throws InterruptedException, BrokenBarrierException {
        try {
            return dowait(false, 0L);
        } catch (TimeoutException toe) {
            throw new Error(toe); // cannot happen
        }
    }

        public int await(long timeout, TimeUnit unit)
        throws InterruptedException,
               BrokenBarrierException,
               TimeoutException {
        return dowait(true, unit.toNanos(timeout));
```

```
    }
```

await也有两个方法，一个是带时间参数的，一个是不带时间参数的。

await本质上调用了lock.newCondition().await()方法。

因为有多个parties，下面我们考虑两种情况。

1. 该线程不是最后一个调用await的线程

在这种情况下，该线程将会进入等待状态，直到下面的情况发送：

- 最后一个线程调用await（）
- 其他线程中断了当前线程
- 其他线程中断了其他正在等待的线程
- 其他线程在等待barrier的时候超时
- 其他线程在该barrier上调用的reset（）方法

如果该线程在调用await（）的时候已经设置了interrupted的状态，或者在等待的时候被interrupted，那么将会抛出InterruptedException异常，并清除中断状态。（这里和Thread的interrupt()方法保持一致）

如果任何线程正在等待状态中，这时候barrier被重置。或者在线程调用await方法或者正在等待中，barrier被broken，那么将会抛出BrokenBarrierException。

如果任何线程在等待的时候被中断，那么所有其他等待的线程将会抛出BrokenBarrierException，barrier将会被置为broken状态。

2. 如果该线程是最后一个调用await方法的

在这种情况，如果barrierAction不为空，那么该线程将会在其他线程继续执行前调用这个barrierAction。

如果该操作抛出异常，那么barrier的状态将会被置为broken状态。

再看看这个reset() 方法：

```
    public void reset() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            breakBarrier();   // break the current generation
            nextGeneration(); // start a new generation
        } finally {
            lock.unlock();
        }
    }
```

该方法将会将barrier置为broken状态，并且开启一个新的generation，来进行下一轮的操作。

CyclicBarrier的使用

我们在子线程中生成一个随机的整数队列，当所有的线程都生成完毕之后，我们再将生成的整数全都加起来。看下怎么实现。

定义生成整数队列的子线程：

```
public class CyclicBarrierUsage implements Runnable {

    private CyclicBarrier cyclicBarrier;
    private List<List<Integer>> partialResults;
    private Random random = new Random();

    public CyclicBarrierUsage(CyclicBarrier cyclicBarrier,List<List<Integer>>
partialResults){
        this.cyclicBarrier=cyclicBarrier;
        this.partialResults=partialResults;
    }

    @Override
    public void run() {
        String thisThreadName = Thread.currentThread().getName();
        List<Integer> partialResult = new ArrayList<>();

        // Crunch some numbers and store the partial result
        for (int i = 0; i < 10; i++) {
            Integer num = random.nextInt(10);
            System.out.println(thisThreadName
                    + ": Crunching some numbers! Final result - " + num);
            partialResult.add(num);
        }

        partialResults.add(partialResult);
        try {
            System.out.println(thisThreadName
                    + " waiting for others to reach barrier.");
            cyclicBarrier.await();
        } catch (InterruptedException e) {
            // ...
        } catch (BrokenBarrierException e) {
            // ...
        }
    }

}
```

上面的子线程接收外部传入的cyclicBarrier和保存数据的partialResults，并在运行完毕调用cyclicBarrier.await()来等待其他线程执行完毕。

看下CyclicBarrier的构建：

```
CyclicBarrier cyclicBarrier=new CyclicBarrier(5,()->{
            String thisThreadName = Thread.currentThread().getName();
```

```
            System.out.println(
                    thisThreadName + ": Computing sum of 5 workers, having 10 results
each.");
            int sum = 0;

            for (List<Integer> threadResult : partialResults) {
                System.out.print("Adding ");
                for (Integer partialResult : threadResult) {
                    System.out.print(partialResult+" ");
                    sum += partialResult;
                }
                System.out.println();
            }
            System.out.println(thisThreadName + ": Final result = " + sum);
        });
```

在CyclicBarrier中，我们定义了一个BarrierAction来做最后数据的汇总处理。

运行：

```
        for (int i = 0; i < 5; i++) {
            Thread worker = new Thread(new
CyclicBarrierUsage(cyclicBarrier,partialResults));
            worker.setName("Thread " + i);
            worker.start();
        }
```

输出结果如下：

```
Spawning 5 worker threads to compute 10 partial results each
Thread 0: Crunching some numbers! Final result - 5
Thread 0: Crunching some numbers! Final result - 3
Thread 1: Crunching some numbers! Final result - 1
Thread 0: Crunching some numbers! Final result - 7
Thread 1: Crunching some numbers! Final result - 8
Thread 0: Crunching some numbers! Final result - 4
Thread 0: Crunching some numbers! Final result - 6
Thread 0: Crunching some numbers! Final result - 9
Thread 1: Crunching some numbers! Final result - 3
Thread 2: Crunching some numbers! Final result - 1
Thread 0: Crunching some numbers! Final result - 0
Thread 2: Crunching some numbers! Final result - 9
Thread 1: Crunching some numbers! Final result - 3
Thread 2: Crunching some numbers! Final result - 7
Thread 0: Crunching some numbers! Final result - 2
Thread 2: Crunching some numbers! Final result - 6
Thread 1: Crunching some numbers! Final result - 6
Thread 2: Crunching some numbers! Final result - 5
Thread 0: Crunching some numbers! Final result - 0
```

```
Thread 2: Crunching some numbers! Final result - 1
Thread 1: Crunching some numbers! Final result - 5
Thread 2: Crunching some numbers! Final result - 1
Thread 0: Crunching some numbers! Final result - 7
Thread 2: Crunching some numbers! Final result - 8
Thread 1: Crunching some numbers! Final result - 2
Thread 2: Crunching some numbers! Final result - 4
Thread 0 waiting for others to reach barrier.
Thread 2: Crunching some numbers! Final result - 0
Thread 2 waiting for others to reach barrier.
Thread 1: Crunching some numbers! Final result - 7
Thread 1: Crunching some numbers! Final result - 6
Thread 1: Crunching some numbers! Final result - 9
Thread 1 waiting for others to reach barrier.
Thread 3: Crunching some numbers! Final result - 9
Thread 3: Crunching some numbers! Final result - 3
Thread 3: Crunching some numbers! Final result - 8
Thread 3: Crunching some numbers! Final result - 8
Thread 3: Crunching some numbers! Final result - 1
Thread 3: Crunching some numbers! Final result - 8
Thread 3: Crunching some numbers! Final result - 0
Thread 3: Crunching some numbers! Final result - 5
Thread 3: Crunching some numbers! Final result - 9
Thread 3: Crunching some numbers! Final result - 1
Thread 3 waiting for others to reach barrier.
Thread 4: Crunching some numbers! Final result - 2
Thread 4: Crunching some numbers! Final result - 2
Thread 4: Crunching some numbers! Final result - 5
Thread 4: Crunching some numbers! Final result - 5
Thread 4: Crunching some numbers! Final result - 3
Thread 4: Crunching some numbers! Final result - 7
Thread 4: Crunching some numbers! Final result - 4
Thread 4: Crunching some numbers! Final result - 8
Thread 4: Crunching some numbers! Final result - 4
Thread 4: Crunching some numbers! Final result - 3
Thread 4 waiting for others to reach barrier.
Thread 4: Computing sum of 5 workers, having 10 results each.
Adding 5 3 7 4 6 9 0 2 0 7
Adding 1 9 7 6 5 1 1 8 4 0
Adding 1 8 3 3 6 5 2 7 6 9
Adding 9 3 8 8 1 8 0 5 9 1
Adding 2 2 5 5 3 7 4 8 4 3
Thread 4: Final result = 230

Process finished with exit code 0
```

第十九章 在java中使用JMH（Java Microbenchmark Harness）做性能测试

JMH的全称是Java Microbenchmark Harness，是一个open JDK中用来做性能测试的套件。该套件已经被包含在了JDK 12中。

本文将会讲解如何使用JMH来在java中做性能测试。

如果你使用的不是JDK 12，那么需要添加如下依赖：

```
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-core</artifactId>
    <version>1.19</version>
</dependency>
<dependency>
    <groupId>org.openjdk.jmh</groupId>
    <artifactId>jmh-generator-annprocess</artifactId>
    <version>1.19</version>
</dependency>
```

使用JMH做性能测试

如果我们想测试某个方法的性能，一般来说就是重复执行某个方法n次，求出总的执行时间，然后求平均值。

但是这样通常会有一些问题，比如程序的头几次执行通常会比较慢，因为JVM会对多次执行的代码进行优化。另外得出的统计结果也不够直观，需要我们自行解析。

如果使用JMH可以轻松解决这些问题。

在JMH中，将要测试的方法添加@Benchmark注解即可：

```
    @Benchmark
    public void measureThroughput() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(100);
    }
```

看下怎么调用：

```
    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(BenchMarkUsage.class.getSimpleName())
//                .include(BenchMarkUsage.class.getSimpleName()+".*measureThroughput*")
                // 预热3轮
                .warmupIterations(3)
                // 度量5轮
                .measurementIterations(5)
                .forks(1)
                .build();

        new Runner(opt).run();
    }
```

上面的例子，我们通过OptionsBuilder的include方法添加了需要进行测试的类。

默认情况下，该类的所有@Benchmark方法都将会被测试，如果我们只想测试其中的某个方法，我们可以在类后面加上方法的名字：

```
.include(BenchMarkUsage.class.getSimpleName()+".*measureAll*")
```

上面的代码支持通配符。

warmupIterations(3)意思是在真正的执行前，先热身三次。

measurementIterations(5)表示我们将方法运行5次来测试性能。

forks(1)表示启动一个进程来执行这个任务。

上面是最基本的运行，我们看下运行结果：

```
# JMH version: 1.19
# VM version: JDK 1.8.0_171, VM 25.171-b11
# VM invoker:
/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/jre/bin/java
# VM options: -javaagent:/Applications/IntelliJ IDEA
2.app/Contents/lib/idea_rt.jar=55941:/Applications/IntelliJ IDEA 2.app/Contents/bin -
Dfile.encoding=UTF-8
# Warmup: 3 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: com.flydean.BenchMarkUsage.measureThroughput

# Run progress: 26.66% complete, ETA 00:01:42
# Fork: 1 of 1
# Warmup Iteration   1: 9.727 ops/s
# Warmup Iteration   2: 9.684 ops/s
# Warmup Iteration   3: 9.678 ops/s
```

```
Iteration   1: 9.652 ops/s
Iteration   2: 9.678 ops/s
Iteration   3: 9.733 ops/s
Iteration   4: 9.651 ops/s
Iteration   5: 9.678 ops/s


Result "com.flydean.BenchMarkUsage.measureThroughput":
  9.678 ±(99.9%) 0.129 ops/s [Average]
  (min, avg, max) = (9.651, 9.678, 9.733), stdev = 0.034
  CI (99.9%): [9.549, 9.808] (assumes normal distribution)
```

ops/s 是每秒的OPS次数。程序会给出运行的最小值，平均值和最大值。同时给出标准差stdev和置信区间CI。

BenchmarkMode

上面的例子中， 我们只用了最简单的@Benchmark。如果想实现更加复杂和自定义的BenchMark，我们可以使用@BenchmarkMode。

先举个例子：

```
    @Benchmark
    @BenchmarkMode(Mode.Throughput)
    @OutputTimeUnit(TimeUnit.SECONDS)
    public void measureThroughput() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(100);
    }
```

上面的例子中，我们指定了@BenchmarkMode(Mode.Throughput)，Throughput的意思是整体吞吐量，表示给定的时间内执行的次数。

这里我们通过 @OutputTimeUnit(TimeUnit.SECONDS)来指定时间单位。

Mode除了Throughput还有如下几种模式：

- AverageTime - 调用的平均时间
- SampleTime - 随机取样，最后输出取样结果的分布
- SingleShotTime - 只会执行一次，通常用来测试冷启动时候的性能。
- All - 所有的benchmark modes。

Fork和Warmup

上面的例子中我们通过代码来显式的制定Fork和Warmup，我们也可以使用注解来实现：

```
    @Fork(value = 1, warmups = 2)
    @Warmup(iterations = 5)
```

上面的例子中value表示该benchMark执行多少次，warmups表示fork多少个进程来执行。iterations表示warmup的iterations个数。

如果你同时在代码中和注解中都配置了相关的信息，那么注解将会覆盖掉代码中的显示配置。

State和Scope

如果我们在多线程环境中使用beachMark,那么多线程中用到的类变量是共享还是每个线程一个呢?

这个时候我们就要用到@State注解。

```
@State(Scope.Benchmark)
public class StateUsage {
}
```

Scope有三种：

- Scope.Thread：默认的State，每个测试线程分配一个实例；
- Scope.Benchmark：所有测试线程共享一个实例，用于测试有状态实例在多线程共享下的性能；
- Scope.Group：每个线程组共享一个实例；

第二十章 java中ThreadLocalRandom的使用

在java中我们通常会需要使用到java.util.Random来便利的生产随机数。但是Random是线程安全的，如果要在线程环境中的话就有可能产生性能瓶颈。

我们以Random中常用的nextInt方法为例来具体看一下：

```
    public int nextInt() {
        return next(32);
    }
```

nextInt方法实际上调用了下面的方法：

```
    protected int next(int bits) {
        long oldseed, nextseed;
        AtomicLong seed = this.seed;
        do {
            oldseed = seed.get();
            nextseed = (oldseed * multiplier + addend) & mask;
        } while (!seed.compareAndSet(oldseed, nextseed));
        return (int)(nextseed >>> (48 - bits));
    }
```

从代码中我们可以看到，方法内部使用了AtomicLong，并调用了它的compareAndSet方法来保证线程安全性。所以这个是一个线程安全的方法。

其实在多个线程环境中，Random根本就需要共享实例，那么该怎么处理呢?

在JDK 7 中引入了一个ThreadLocalRandom的类。ThreadLocal大家都知道就是线程的本地变量，而ThreadLocalRandom就是线程本地的Random。

我们看下怎么调用：

```
ThreadLocalRandom.current().nextInt();
```

我们来为这两个类分别写一个benchMark测试：

```
public class RandomUsage {

    public void testRandom() throws InterruptedException {
        ExecutorService executorService=Executors.newFixedThreadPool(2);
        Random random = new Random();
        List<Callable<Integer>> callables = new ArrayList<>();
        for (int i = 0; i < 1000; i++) {
            callables.add(() -> {
                return random.nextInt();
            });
            }
        executorService.invokeAll(callables);
    }

    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(RandomUsage.class.getSimpleName())
                // 预热5轮
                .warmupIterations(5)
                // 度量10轮
                .measurementIterations(10)
                .forks(1)
                .build();

        new Runner(opt).run();
    }
}
```

```
public class ThreadLocalRandomUsage {

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.MICROSECONDS)
    public void testThreadLocalRandom() throws InterruptedException {
        ExecutorService executorService=Executors.newFixedThreadPool(2);
        List<Callable<Integer>> callables = new ArrayList<>();
        for (int i = 0; i < 1000; i++) {
            callables.add(() -> {
                return ThreadLocalRandom.current().nextInt();
            });
            }
        executorService.invokeAll(callables);
    }
```

```
    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(ThreadLocalRandomUsage.class.getSimpleName())
                // 预热5轮
                .warmupIterations(5)
                // 度量10轮
                .measurementIterations(10)
                .forks(1)
                .build();

        new Runner(opt).run();
    }
}
```

分析运行结果，我们可以看出ThreadLocalRandom在多线程环境中会比Random要快。

第二十一章 java中FutureTask的使用

FutureTask简介

FutureTask是java 5引入的一个类，从名字可以看出来FutureTask既是一个Future，又是一个Task。

我们看下FutureTask的定义：

```
public class FutureTask<V> implements RunnableFuture<V> {
    ...
}
```

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     */
    void run();
}
```

FutureTask实现了RunnableFuture接口，RunnableFuture接口是Runnable和Future的综合体。

作为一个Future，FutureTask可以执行异步计算，可以查看异步程序是否执行完毕，并且可以开始和取消程序，并取得程序最终的执行结果。

除此之外，FutureTask还提供了一个runAndReset()的方法， 该方法可以运行task并且重置Future的状态。

Callable和Runnable的转换

我们知道Callable是有返回值的，而Runnable是没有返回值的。
Executors提供了很多有用的方法，将Runnable转换为Callable：

```
    public static <T> Callable<T> callable(Runnable task, T result) {
        if (task == null)
            throw new NullPointerException();
        return new RunnableAdapter<T>(task, result);
    }
```

FutureTask内部包含一个Callable，并且可以接受Callable和Runnable作为构造函数：

```
    public FutureTask(Callable<V> callable) {
        if (callable == null)
            throw new NullPointerException();
        this.callable = callable;
        this.state = NEW;       // ensure visibility of callable
    }
```

```
    public FutureTask(Runnable runnable, V result) {
        this.callable = Executors.callable(runnable, result);
        this.state = NEW;       // ensure visibility of callable
    }
```

它的内部就是调用了Executors.callable(runnable, result);方法进行转换的。

以Runnable运行

既然是一个Runnable，那么FutureTask就可以以线程的方式执行，我们来看一个例子：

```
@Test
    public void convertRunnableToCallable() throws ExecutionException,
InterruptedException {
        FutureTask<Integer> futureTask = new FutureTask<>(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
               log.info("inside callable future task ...");
                return 0;
            }
        });

        Thread thread= new Thread(futureTask);
        thread.start();
        log.info(futureTask.get().toString());
    }
```

上面例子是以单个线程来执行的，同样我们也可以将FutureTask提交给线程池来执行：

```
    @Test
    public void workWithExecutorService() throws ExecutionException,
InterruptedException {
```

```
        FutureTask<Integer> futureTask = new FutureTask<>(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                log.info("inside futureTask");
                return 1;
            }
        });
        ExecutorService executor = Executors.newCachedThreadPool();
        executor.submit(futureTask);
        executor.shutdown();
        log.info(futureTask.get().toString());
    }
```

第二十二章 java中CompletableFuture的使用

之前的文章中，我们讲解了Future， 本文我们将会继续讲解java 8中引入的CompletableFuture的用法。

CompletableFuture首先是一个Future，它拥有Future所有的功能，包括获取异步执行结果，取消正在执行的任务等。

除此之外，CompletableFuture还是一个CompletionStage。

我们看下CompletableFuture的定义：

```
public class CompletableFuture<T> implements Future<T>, CompletionStage<T>
```

什么是CompletionStage呢?

在异步程序中，如果将每次的异步执行都看成是一个stage的话，我们通常很难控制异步程序的执行顺序，在javascript中，我们需要在回调中执行回调。这就会形成传说中的回调地狱。

好在在ES6中引入了promise的概念，可以将回调中的回调转写为链式调用，从而大大的提升了程序的可读性和可写性。

同样的在java中，我们使用CompletionStage来实现异步调用的链式操作。

CompletionStage定义了一系列的then*** 操作来实现这一功能。

CompletableFuture作为Future使用

调用CompletableFuture.complete方法可以立马返回结果，我们看下怎么使用这个方法来构建一个基本的Future：

```
    public Future<String> calculateAsync() throws InterruptedException {
        CompletableFuture<String> completableFuture
                = new CompletableFuture<>();

        Executors.newCachedThreadPool().submit(() -> {
            Thread.sleep(500);
            completableFuture.complete("Hello");
            return null;
        });

        return completableFuture;
    }
```

上面我们通过调动ExecutorService来提交一个任务从而得到一个Future。如果你知道执行的结果，那么可以使用CompletableFuture的completedFuture方法来直接返回一个Future。

```
    public Future<String> useCompletableFuture(){
        Future<String> completableFuture =
                CompletableFuture.completedFuture("Hello");
        return completableFuture;
    }
```

CompletableFuture还提供了一个cancel方法来立马取消任务的执行：

```
    public Future<String> calculateAsyncWithCancellation() throws InterruptedException
{
    CompletableFuture<String> completableFuture = new CompletableFuture<>();

    Executors.newCachedThreadPool().submit(() -> {
        Thread.sleep(500);
        completableFuture.cancel(false);
        return null;
    });
    return completableFuture;
    }
```

如果这个时候调用Future的get方法，将会报CancellationException异常。

```
Future<String> future = calculateAsyncWithCancellation();
future.get(); // CancellationException
```

异步执行code

CompletableFuture提供了runAsync和supplyAsync的方法，可以以异步的方式执行代码。

我们看一个runAsync的基本应用，接收一个Runnable参数：

```
    public  void runAsync(){
        CompletableFuture<Void> runAsync= CompletableFuture.runAsync(()->{
            log.info("runAsync");
        });
    }
```

而supplyAsync接受一个Supplier：

```
    public void supplyAsync(){
        CompletableFuture<String> supplyAsync=CompletableFuture.supplyAsync(()->{
            return "supplyAsync";
        });
    }
```

他们两个的区别是一个没有返回值，一个有返回值。

组合Futures

上面讲到CompletableFuture的一个重大作用就是将回调改为链式调用，从而将Futures组合起来。

而链式调用的返回值还是CompletableFuture，我们看一个thenCompose的例子：

```
CompletableFuture<String> completableFuture
  = CompletableFuture.supplyAsync(() -> "Hello")
    .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " World"));
```

thenCompose将前一个Future的返回结果作为后一个操作的输入。

如果我们想合并两个CompletableFuture的结果，则可以使用thenCombine：

```
    public void thenCombine(){
        CompletableFuture<String> completableFuture
                = CompletableFuture.supplyAsync(() -> "Hello")
                .thenCombine(CompletableFuture.supplyAsync(
                        () -> " World"), (s1, s2) -> s1 + s2));
    }
```

如果你不想返回结果，则可以使用thenAcceptBoth：

```
    public void thenAcceptBoth(){
        CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> "Hello")
                .thenAcceptBoth(CompletableFuture.supplyAsync(() -> " World"),
                        (s1, s2) -> System.out.println(s1 + s2));
    }
```

thenApply() 和 thenCompose()的区别

thenApply()和thenCompose()两个方法都可以将CompletableFuture连接起来，但是两个有点不一样。

thenApply()接收的是前一个调用返回的结果，然后对该结果进行处理。

thenCompose()接收的是前一个调用的stage，返回flat之后的的CompletableFuture。

简单点比较，两者就像是map和flatMap的区别。

并行执行任务

当我们需要并行执行任务时，通常我们需要等待所有的任务都执行完毕再去处理其他的任务，那么我们可以用到CompletableFuture.allOf方法：

```
    public void allOf(){
        CompletableFuture<String> future1
                = CompletableFuture.supplyAsync(() -> "Hello");
        CompletableFuture<String> future2
                = CompletableFuture.supplyAsync(() -> "Beautiful");
        CompletableFuture<String> future3
                = CompletableFuture.supplyAsync(() -> "World");

        CompletableFuture<Void> combinedFuture
                = CompletableFuture.allOf(future1, future2, future3);
    }
```

allOf只保证task全都执行，而并没有返回值，如果希望带有返回值，我们可以使用join：

```
    public void join(){
        CompletableFuture<String> future1
                = CompletableFuture.supplyAsync(() -> "Hello");
        CompletableFuture<String> future2
                = CompletableFuture.supplyAsync(() -> "Beautiful");
        CompletableFuture<String> future3
                = CompletableFuture.supplyAsync(() -> "World");

        String combined = Stream.of(future1, future2, future3)
                .map(CompletableFuture::join)
                .collect(Collectors.joining(" "));
    }
```

上面的程序将会返回：“Hello Beautiful World”。

异常处理

如果在链式调用的时候抛出异常，则可以在最后使用handle来接收：

```
    public void handleError(){
        String name = null;

        CompletableFuture<String> completableFuture
                =  CompletableFuture.supplyAsync(() -> {
            if (name == null) {
                throw new RuntimeException("Computation error!");
            }
            return "Hello, " + name;
        }).handle((s, t) -> s != null ? s : "Hello, Stranger!");
    }
```

这和Promise中的catch方法使用类似。

第二十三章 java中使用Semaphore构建阻塞对象池

Semaphore是java 5中引入的概念，叫做计数信号量。主要用来控制同时访问某个特定资源的访问数量或者执行某个操作的数量。

Semaphore中定义了一组虚拟的permits，通过获取和释放这些permits，Semaphore可以控制资源的个数。

Semaphore的这个特性可以用来构造资源池，比如数据库连接池等。

Semaphore有两个构造函数：

```
    public Semaphore(int permits) {
        sync = new NonfairSync(permits);
    }
```

```
    public Semaphore(int permits, boolean fair) {
        sync = fair ? new FairSync(permits) : new NonfairSync(permits);
    }
```

permits定义了许可资源的个数，而fair则表示是否支持FIFO的顺序。

两个比较常用的方法就是acquire和release了。

```
    public void acquire() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }
```

```
    public void release() {
        sync.releaseShared(1);
    }
```

其中acquire用来获取资源，release用来释放资源。

有了这两个特性， 我们看一下怎么使用Semaphore来定义一个一个有界容器。

我们可以将Semaphore初始化为容器池大小，并且在容器池获取资源时调用acquire，将资源返回给容器池之后再调用release。

我们看下面的一个实现：

```
public class SemaphoreUsage<T> {

    private final Set<T> set;
    private final Semaphore sem;

    public SemaphoreUsage(int bound){
        this.set = Collections.synchronizedSet(new HashSet<T>());
        sem= new Semaphore(bound);
    }

    public boolean add (T o) throws InterruptedException{
        sem.acquire();
        boolean wasAdded = false;
        try{
            wasAdded=set.add(o);
            return wasAdded;
        }finally {
            if(!wasAdded){
                sem.release();
            }
        }
    }

    public boolean remove(Object o){
        boolean wasRemoved = set.remove(o);
        if(wasRemoved){
            sem.release();
        }
        return wasRemoved;
    }

}
```

上面的例子我们定义了一个有界的synchronizedSet。 要注意一点是在add方法中，只有add成功之后才会调用release方法。

第二十四章 在java中构建高效的结果缓存

缓存是现代应用服务器中非常常用的组件。除了第三方缓存以外，我们通常也需要在java中构建内部使用的缓存。那么怎么才能构建一个高效的缓存呢？ 本文将会一步步的进行揭秘。

使用HashMap

缓存通常的用法就是构建一个内存中使用的Map，在做一个长时间的操作比如计算之前，先在Map中查询一下计算的结果是否存在，如果不存在的话再执行计算操作。

我们定义了一个代表计算的接口：

```
public interface Calculator<A, V> {

    V calculate(A arg) throws InterruptedException;
}
```

该接口定义了一个calculate方法，接收一个参数，并且返回计算的结果。

我们要定义的缓存就是这个Calculator具体实现的一个封装。

我们看下用HashMap怎么实现：

```
public class MemoizedCalculator1<A, V> implements Calculator<A, V> {

    private final Map<A, V> cache= new HashMap<A, V>();
    private final Calculator<A, V> calculator;

    public MemoizedCalculator1(Calculator<A, V> calculator){
        this.calculator=calculator;
    }
    @Override
    public synchronized V calculate(A arg) throws InterruptedException {
        V result= cache.get(arg);
        if( result ==null ){
            result= calculator.calculate(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

MemoizedCalculator1封装了Calculator，在调用calculate方法中，实际上调用了封装的Calculator的calculate方法。

因为HashMap不是线程安全的，所以这里我们使用了synchronized关键字，从而保证一次只有一个线程能够访问calculate方法。

虽然这样的设计能够保证程序的正确执行，但是每次只允许一个线程执行calculate操作，其他调用calculate方法的线程将会被阻塞，在多线程的执行环境中这会严重影响速度。从而导致使用缓存可能比不使用缓存需要的时间更长。

使用ConcurrentHashMap

因为HashMap不是线程安全的，那么我们可以尝试使用线程安全的ConcurrentHashMap来替代HashMap。如下所示：

```
public class MemoizedCalculator2<A, V> implements Calculator<A, V> {

    private final Map<A, V> cache= new ConcurrentHashMap<>();
    private final Calculator<A, V> calculator;

    public MemoizedCalculator2(Calculator<A, V> calculator){
        this.calculator=calculator;
    }
    @Override
    public V calculate(A arg) throws InterruptedException {
        V result= cache.get(arg);
        if( result ==null ){
            result= calculator.calculate(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```

上面的例子中虽然解决了之前的线程等待的问题，但是当有两个线程同时在进行同一个计算的时候，仍然不能保证缓存重用，这时候两个线程都会分别调用计算方法，从而导致重复计算。

我们希望的是如果一个线程正在做计算，其他的线程只需要等待这个线程的执行结果即可。很自然的，我们想到了之前讲到的FutureTask。FutureTask表示一个计算过程，我们可以通过调用FutureTask的get方法来获取执行的结果，如果该执行正在进行中，则会等待。

下面我们使用FutureTask来进行改写。

FutureTask

```
@Slf4j
public class MemoizedCalculator3<A, V> implements Calculator<A, V> {

    private final Map<A, Future<V>> cache= new ConcurrentHashMap<>();
    private final Calculator<A, V> calculator;

    public MemoizedCalculator3(Calculator<A, V> calculator){
        this.calculator=calculator;
    }
    @Override
    public V calculate(A arg) throws InterruptedException {
        Future<V> future= cache.get(arg);
        V result=null;
        if( future ==null ){
            Callable<V> callable= new Callable<V>() {
                @Override
```

```
                public V call() throws Exception {
                    return calculator.calculate(arg);
                }
            };
            FutureTask<V> futureTask= new FutureTask<>(callable);
            future= futureTask;
            cache.put(arg, futureTask);
            futureTask.run();
        }
        try {
            result= future.get();
        } catch (ExecutionException e) {
           log.error(e.getMessage(),e);
        }
        return result;
    }
}
```

上面的例子，我们用FutureTask来封装计算，并且将FutureTask作为Map的value。

上面的例子已经体现了很好的并发性能。但是因为if语句是非原子性的，所以对这一种先检查后执行的操作，仍然可能存在同一时间调用的情况。

这个时候，我们可以借助于ConcurrentHashMap的原子性操作putIfAbsent来重写上面的类：

```
@Slf4j
public class MemoizedCalculator4<A, V> implements Calculator<A, V> {

    private final Map<A, Future<V>> cache= new ConcurrentHashMap<>();
    private final Calculator<A, V> calculator;

    public MemoizedCalculator4(Calculator<A, V> calculator){
        this.calculator=calculator;
    }
    @Override
    public V calculate(A arg) throws InterruptedException {
        while (true) {
            Future<V> future = cache.get(arg);
            V result = null;
            if (future == null) {
                Callable<V> callable = new Callable<V>() {
                    @Override
                    public V call() throws Exception {
                        return calculator.calculate(arg);
                    }
                };
                FutureTask<V> futureTask = new FutureTask<>(callable);
                future = cache.putIfAbsent(arg, futureTask);
                if (future == null) {
                    future = futureTask;
```

```
                    futureTask.run();
                }

                try {
                    result = future.get();
                } catch (CancellationException e) {
                    log.error(e.getMessage(), e);
                    cache.remove(arg, future);
                } catch (ExecutionException e) {
                    log.error(e.getMessage(), e);
                }
                return result;
            }
        }
    }
}
```

上面使用了一个while循环，来判断从cache中获取的值是否存在，如果不存在则调用计算方法。

上面我们还要考虑一个缓存污染的问题，因为我们修改了缓存的结果，如果在计算的时候，计算被取消或者失败，我们需要从缓存中将FutureTask移除。

第二十五章 java中CompletionService的使用

之前的文章中我们讲到了ExecutorService，通过ExecutorService我们可以提交一个个的task，并且返回Future，然后通过调用Future.get方法来返回任务的执行结果。

这种方式虽然有效，但是需要保存每个返回的Future值，还是比较麻烦的，幸好ExecutorService提供了一个invokeAll的方法，来保存所有的Future值，我们看一个具体的实现：

```
    public void useExecutorService() throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(10);

        Callable<String> callableTask = () -> {
            TimeUnit.MILLISECONDS.sleep(300);
            return "Task's execution";
        };

        List<Callable<String>> callableTasks = new ArrayList<>();
        callableTasks.add(callableTask);
        callableTasks.add(callableTask);
        callableTasks.add(callableTask);

        List<Future<String>> futures = executor.invokeAll(callableTasks);

        executor.shutdown();

    }
```

上面的例子中，我们定义了3个task，通过调用executor.invokeAll(callableTasks)返回了一个 List<Future>，这样我们就可以得到所有的返回值了。

除了上面的invokeAll方法外，我们今天要介绍一个CompletionService接口。

CompletionService实际上是ExecutorService和BlockingQueue的结合体，ExecutorService用来提交任务，而BlockingQueue用来保存封装成Future的执行结果。通过调用take和poll的方法来获取到Future值。

CompletionService是一个接口，我们看下它的一个具体实现ExecutorCompletionService：

```
    public ExecutorCompletionService(Executor executor) {
        if (executor == null)
            throw new NullPointerException();
        this.executor = executor;
        this.aes = (executor instanceof AbstractExecutorService) ?
            (AbstractExecutorService) executor : null;
        this.completionQueue = new LinkedBlockingQueue<Future<V>>();
    }
```

ExecutorCompletionService接收一个Executor作为参数。

我们看下上面的例子如果用ExecutorCompletionService重写是怎么样的：

```
    public void useCompletionService() throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(10);
        CompletionService<String> completionService=new
ExecutorCompletionService<String>(executor);
        Callable<String> callableTask = () -> {
            TimeUnit.MILLISECONDS.sleep(300);
            return "Task's execution";
        };
        for(int i=0; i< 5; i ++){
            completionService.submit(callableTask);
        }

        for(int i=0; i<5; i++){
            Future<String> result=completionService.take();
            System.out.println(result.get());
        }
    }
```

上面的例子通过completionService.submit来提交任务，通过completionService.take()来获取结果值。

其实CompletionService还有一个poll的方法，poll和take的区别在于：take如果获取不到值则会等待，而poll则会返回null。

第二十六章 使用ExecutorService来停止线程服务

之前的文章中我们提到了ExecutorService可以使用shutdown和shutdownNow来关闭。

这两种关闭的区别在于各自的安全性和响应性。shutdownNow强行关闭速度更快，但是风险也更大，因为任务可能正在执行的过程中被结束了。而shutdown正常关闭虽然速度比较慢，但是却更安全，因为它一直等到队列中的所有任务都执行完毕之后才关闭。

使用shutdown

我们先看一个使用shutdown的例子:

```
    public void useShutdown() throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(10);

        Runnable runnableTask = () -> {
            try {
                TimeUnit.MILLISECONDS.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        };

        executor.submit(runnableTask);
        executor.shutdown();
        executor.awaitTermination(800, TimeUnit.MILLISECONDS);
    }
```

awaitTermination将会阻塞直到所有正在执行的任务完成，或者达到指定的timeout时间。

使用shutdownNow

当通过shutdownNow来强行关闭ExecutorService是， 它会尝试取消正在执行的任务，并返回所有已经提交但是还没有开始的任务。从而可以将这些任务保存起来，以便以后进行处理。

但是这样我们只知道了还没有开始执行的任务，对于那些已经开始执行但是没有执行完毕却被取消的任务我们无法获取。

我们看下如何获得开始执行但是还没有执行完毕的任务：

```
public class TrackingExecutor extends AbstractExecutorService {
    private final ExecutorService executorService;
    private final Set<Runnable> taskCancelledAtShutdown=
Collections.synchronizedSet(new HashSet<Runnable>());

    public TrackingExecutor(ExecutorService executorService){
         this.executorService=executorService;
    }
    @Override
    public void shutdown() {
        executorService.shutdown();
    }
```

```
    @Override
    public List<Runnable> shutdownNow() {
        return executorService.shutdownNow();
    }

    @Override
    public boolean isShutdown() {
        return executorService.isShutdown();
    }

    @Override
    public boolean isTerminated() {
        return executorService.isTerminated();
    }

    @Override
    public boolean awaitTermination(long timeout, TimeUnit unit) throws
InterruptedException {
        return executorService.awaitTermination(timeout,unit);
    }

    @Override
    public void execute(Runnable command) {
        executorService.execute(() -> {
            try {
                command.run();
            }finally {
                if(isShutdown() && Thread.currentThread().isInterrupted()){
                    taskCancelledAtShutdown.add(command);
                }
            }
        });
    }

    public List<Runnable> getCancelledTask(){
        if(! executorService.isTerminated()){
            throw new IllegalStateException("executorService is not terminated");
        }
        return new ArrayList<>(taskCancelledAtShutdown);
    }
}
```

上面的例子中我们构建了一个新的ExecutorService，他传入一个ExecutorService，并对其进行封装。

我们重写了execute方法，在执行完毕判断该任务是否被中断，如果被中断则将其添加到CancelledTask列表中。

并提供一个getCancelledTask方法来返回未执行完毕的任务。

我们看下怎么使用：

```
    public void useShutdownNow() throws InterruptedException {
        TrackingExecutor trackingExecutor=new
TrackingExecutor(Executors.newCachedThreadPool());

        Runnable runnableTask = () -> {
            try {
                TimeUnit.MILLISECONDS.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        };

        trackingExecutor.submit(runnableTask);
        List<Runnable> notrunList=trackingExecutor.shutdownNow();
        if(trackingExecutor.awaitTermination(800, TimeUnit.SECONDS)){
            List<Runnable> runButCancelledList= trackingExecutor.getCancelledTask();
        }
    }
```

trackingExecutor.shutdownNow()返回的是未执行的任务。而trackingExecutor.getCancelledTask()返回的是被取消的任务。

上面的任务其实还有一个缺点，因为我们在存储被取消的任务列表的额时候taskCancelledAtShutdown.add(command)，因为之前的判断不是原子操作，则可能会产生误报。

第二十七章 我们的线程被饿死了

我们在构建线程池的时候可以构建单个线程的线程池和多个线程的线程池。

那么线程池使用不当可不可能产生死锁呢？我们知道死锁是循环争夺资源而产生的。线程池中的线程也是资源的一种，那么如果对线程池中的线程进行争夺的话也是可能产生死锁的。

在单个线程的线程池中，如果一个正在执行的线程中，使用该线程池再去提交第二个任务，因为线程池中的线程只有一个，那么第二个任务将会等待第一个任务的执行完成来释放线程，而第一个任务又在等待第二任务的执行来完成任务。从而产生了线程饥饿死锁（Thread Starvation Deadlock）.

线程饥饿死锁并不一定在单个线程的线程池中产生，只要有这种循环使用线程池的情况都可能产生这种问题。

我们看下例子：

```
public class ThreadPoolDeadlock {

    ExecutorService executorService= Executors.newSingleThreadExecutor();

    public class RenderPageTask implements Callable<String> {
        public String call() throws Exception{
            Future<String> header, footer;
            header= executorService.submit(()->{
                return "header";
            });
```

```
            footer= executorService.submit(()->{
                return "footer";
            });
            return header.get()+ footer.get();
        }
    }

    public void submitTask(){
        executorService.submit(new RenderPageTask());
    }
}
```

我们在executorService中提交了一个RenderPageTask，而RenderPageTask又提交了两个task。因为ExecutorService线程池只有一个线程，则会产生死锁。

我们的线程被饿死了！

第二十八章 java中有界队列的饱和策略(reject policy)

我们在使用ExecutorService的时候知道，在ExecutorService中有个一个Queue来保存提交的任务，通过不同的构造函数，我们可以创建无界的队列（ExecutorService.newCachedThreadPool）和有界的队列(ExecutorService newFixedThreadPool(int nThreads))。

无界队列很好理解，我们可以无限制的向ExecutorService提交任务。那么对于有界队列来说，如果队列满了该怎么处理呢?

今天我们要介绍一下java中ExecutorService的饱和策略(reject policy)。

以ExecutorService的具体实现ThreadPoolExecutor来说，它定义了4种饱和策略。分别是AbortPolicy，DiscardPolicy，DiscardOldestPolicy和CallerRunsPolicy。

如果要在ThreadPoolExecutor中设定饱和策略可以调用setRejectedExecutionHandler方法，如下所示：

```
        ThreadPoolExecutor threadPoolExecutor= new ThreadPoolExecutor(5, 10, 10,
TimeUnit.SECONDS, new LinkedBlockingDeque<Runnable>(20));
        threadPoolExecutor.setRejectedExecutionHandler(
                new ThreadPoolExecutor.AbortPolicy()
        );
```

上面的例子中我们定义了一个初始5个，最大10个工作线程的Thread Pool，并且定义其中的Queue的容量是20。如果提交的任务超出了容量，则会使用AbortPolicy策略。

AbortPolicy

AbortPolicy意思是如果队列满了，最新的提交任务将会被拒绝，并抛出RejectedExecutionException异常：

```
    public static class AbortPolicy implements RejectedExecutionHandler {
        /**
         * Creates an {@code AbortPolicy}.
         */
        public AbortPolicy() { }

        /**
         * Always throws RejectedExecutionException.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         * @throws RejectedExecutionException always
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            throw new RejectedExecutionException("Task " + r.toString() +
                                                 " rejected from " +
                                                 e.toString());
        }
    }
```

上面的代码中，rejectedExecution方法中我们直接抛出了RejectedExecutionException异常。

DiscardPolicy

DiscardPolicy将会悄悄的丢弃提交的任务，而不报任何异常。

```
public static class DiscardPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code DiscardPolicy}.
         */
        public DiscardPolicy() { }

        /**
         * Does nothing, which has the effect of discarding task r.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        }
    }
```

DiscardOldestPolicy

DiscardOldestPolicy将会丢弃最老的任务，保存最新插入的任务。

```
    public static class DiscardOldestPolicy implements RejectedExecutionHandler {
        /**
```

```
         * Creates a {@code DiscardOldestPolicy} for the given executor.
         */
        public DiscardOldestPolicy() { }

        /**
         * Obtains and ignores the next task that the executor
         * would otherwise execute, if one is immediately available,
         * and then retries execution of task r, unless the executor
         * is shut down, in which case task r is instead discarded.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                e.getQueue().poll();
                e.execute(r);
            }
        }
    }
```

我们看到在rejectedExecution方法中，poll了最老的一个任务，然后使用ThreadPoolExecutor提交了一个最新的任务。

CallerRunsPolicy

CallerRunsPolicy和其他的几个策略不同，它既不会抛弃任务，也不会抛出异常，而是将任务回退给调用者，使用调用者的线程来执行任务，从而降低调用者的调用速度。我们看下是怎么实现的：

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {
        /**
         * Creates a {@code CallerRunsPolicy}.
         */
        public CallerRunsPolicy() { }

        /**
         * Executes task r in the caller's thread, unless the executor
         * has been shut down, in which case the task is discarded.
         *
         * @param r the runnable task requested to be executed
         * @param e the executor attempting to execute this task
         */
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                r.run();
            }
        }
    }
```

在rejectedExecution方法中，直接调用了 r.run()方法，这会导致该方法直接在调用者的主线程中执行，而不是在线程池中执行。从而导致主线程在该任务执行结束之前不能提交任何任务。从而有效的阻止了任务的提交。

使用Semaphore

如果我们并没有定义饱和策略，那么有没有什么方法来控制任务的提交速度呢？考虑下之前我们讲到的Semaphore，我们可以指定一定的资源信号量来控制任务的提交，如下所示：

```
public class SemaphoreUsage {

    private final Executor executor;
    private final Semaphore semaphore;

    public SemaphoreUsage(Executor executor, int count) {
        this.executor = executor;
        this.semaphore = new Semaphore(count);
    }

    public void submitTask(final Runnable command) throws InterruptedException {
        semaphore.acquire();
        try {
            executor.execute(() -> {
                        try {
                            command.run();
                        } finally {
                            semaphore.release();
                        }
                    }
            );
        } catch (RejectedExecutionException e) {
            semaphore.release();
        }
    }

}
```

第二十九章 由于不当的执行顺序导致的死锁

为了保证线程的安全，我们引入了加锁机制，但是如果不加限制的使用加锁，就有可能会导致顺序死锁（Lock-Ordering Deadlock）。上篇文章我们也提到了在线程词中因为资源的不足而导致的资源死锁（Resource Deadlock）。

本文将会讨论一下顺序死锁的问题。

我们来讨论一个经常存在的账户转账的问题。账户A要转账给账户B。为了保证在转账的过程中A和B不被其他的线程意外的操作，我们需要给A和B加锁，然后再进行转账操作， 我们看下转账的代码：

```
    public void transferMoneyDeadLock(Account from,Account to, int amount) throws
InsufficientAmountException {
```

```
        synchronized (from){
            synchronized (to){
                transfer(from,to,amount);
            }
        }
    }

    private void transfer(Account from,Account to, int amount) throws
InsufficientAmountException {
        if(from.getBalance() < amount){
            throw new InsufficientAmountException();
        }else{
            from.debit(amount);
            to.credit(amount);
        }
    }
```

看起来上面的程序好像没有问题，因为我们给from和to都加了锁，程序应该可以很完美的按照我们的要求来执行。

那如果我们考虑下面的一个场景：

```
A: transferMoneyDeadLock (accountA, accountB, 20)
B: transferMoneyDeadLock (accountB, accountA, 10)
```

如果A和B同时执行，则可能会产生A获得了accountA的锁，而B获得了accountB的锁。从而后面的代码无法继续执行，从而导致了死锁。

对于这样的情况，我们有没有什么好办法来处理呢?

加入不管参数怎么传递，我们都先lock accountA再lock accountB是不是就不会出现死锁的问题了呢?

我们看下代码实现：

```
    private void transfer(Account from,Account to, int amount) throws
InsufficientAmountException {
        if(from.getBalance() < amount){
            throw new InsufficientAmountException();
        }else{
            from.debit(amount);
            to.credit(amount);
        }
    }

    public void transferMoney(Account from,Account to, int amount) throws
InsufficientAmountException {

        int fromHash= System.identityHashCode(from);
        int toHash = System.identityHashCode(to);

        if(fromHash < toHash){
```

```
            synchronized (from){
                synchronized (to){
                    transfer(from,to, amount);
                }
            }
        }else if(fromHash < toHash){
             synchronized (to){
                 synchronized (from){
                     transfer(from,to, amount);
                 }
             }
         }else{
            synchronized (lock){
            synchronized (from) {
                synchronized (to) {
                    transfer(from, to, amount);
                }
              }
            }
        }
    }
```

上面的例子中，我们使用了System.identityHashCode来获得两个账号的hash值，通过比较hash值的大小来选定lock的顺序。

如果两个账号的hash值恰好相等的情况下，我们引入了一个新的外部lock，从而保证同一时间只有一个线程能够运行内部的方法，从而保证了任务的执行而不产生死锁。

第三十章 非阻塞同步机制和CAS

我们知道在java 5之前同步是通过Synchronized关键字来实现的，在java 5之后，java.util.concurrent包里面添加了很多性能更加强大的同步类。这些强大的类中很多都实现了非阻塞的同步机制从而帮助其提升性能。

什么是非阻塞同步

非阻塞同步的意思是多个线程在竞争相同的数据时候不会发生阻塞，从而能够在更加细粒度的维度上进行协调，从而极大的减少线程调度的开销，从而提升效率。非阻塞算法不存在锁的机制也就不存在死锁的问题。

在基于锁的算法中，如果一个线程持有了锁，那么其他的线程将无法进行下去。使用锁虽然可以保证对资源的一致性访问，但是在挂起和恢复线程的执行过程中存在非常大的开销，如果锁上面存在着大量的竞争，那么有可能调度开销比实际工作开销还要高。

悲观锁和乐观锁

我们知道独占锁是一个悲观锁，悲观锁的意思就是假设最坏的情况，如果你不锁定该资源，那么就有其他的线程会修改该资源。悲观锁虽然可以保证任务的顺利执行，但是效率不高。

乐观锁就是假设其他的线程不会更改要处理的资源，但是我们在更新资源的时候需要判断该资源是否被别的线程所更改。如果被更改那么更新失败，我们可以重试，如果没有被更改，那么更新成功。

使用乐观锁的前提是假设大多数时间系统对资源的更新是不会产生冲突的。

乐观锁的原子性比较和更新操作，一般都是由底层的硬件支持的。

CAS

大多数的处理器都实现了一个CAS指令（compare and swap）,通常来说一个CAS接收三个参数，数据的现值V，进行比较的值A，准备写入的值B。只有当V和A相等的时候，才会写入B。无论是否写入成功，都会返回V。翻译过来就是“我认为V现在的值是A，如果是那么将V的值更新为B，否则不修改V的值，并告诉我现在V的值是多少。”

这就是CAS的含义，JDK中的并发类是通过使用Unsafe类来使用CAS的，我们可以自己构建一个并发类，如下所示：

```
public class CasCounter {

    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;
    private volatile int value;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                    (CasCounter.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    public CasCounter(int initialValue) {
        value = initialValue;
    }

    public CasCounter() {
    }

    public final int get() {
        return value;
    }

    public final void set(int newValue) {
        value = newValue;
    }

    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }

}
```

上面的例子中，我们定义了一个原子操作compareAndSet， 它内部调用了unsafe的compareAndSwapInt方法。

看起来上面的CAS使用比直接使用锁复杂，但实际上在JVM中实现锁定时需要遍历JVM中一条非常复杂的代码路径，并可能导致操作系统级的锁定，线程挂机和上下文切换等操作。在最好的情况下，锁定需要执行一次CAS命令。

CAS的主要缺点就是需要调用者自己来处理竞争问题（重试，回退，放弃），而在锁中可以自动处理这些问题。

前面的文章我们也讲到了原子变量，原子变量的底层就是使用CAS。

第三十一章 非阻塞算法（Lock-Free）的实现

上篇文章我们讲到了使用锁会带来的各种缺点，本文将会讲解如何使用非阻塞算法。非阻塞算法一般会使用CAS来协调线程的操作。

虽然非阻塞算法有诸多优点，但是在实现上要比基于锁的算法更加繁琐和负责。

本文将会介绍两个是用非阻塞算法实现的数据结构。

非阻塞的栈

我们先使用CAS来构建几个非阻塞的栈。栈是最简单的链式结构，其本质是一个链表，而链表的根节点就是栈顶。

我们先构建Node数据结构：

```
public class Node<E> {
    public final E item;
    public Node<E> next;

    public Node(E item){
        this.item=item;
    }
}
```

这个Node保存了内存item和它的下一个节点next。

然后我们构建非阻塞的栈，在该栈中我们需要实现pop和push方法，我们使用一个Atomic类来保存top节点的引用，在pop和push之前调用compareAndSet命令来保证命令的原子性。同时，我们需要不断的循环，以保证在线程冲突的时候能够重试更新。

```
public class ConcurrentStack<E> {

    AtomicReference<Node<E>> top= new AtomicReference<>();

    public void push(E item){
        Node<E> newNode= new Node<>(item);
        Node<E> oldNode;
        do{
            oldNode=top.get();
            newNode.next= oldNode;
        }while(!top.compareAndSet(oldNode, newNode));
    }
```

```
    public E pop(){
        Node<E> oldNode;
        Node<E> newNode;
        do {
            oldNode = top.get();
            if(oldNode == null){
                return null;
            }
            newNode=oldNode.next;
        }while(!top.compareAndSet(oldNode, newNode));
        return oldNode.item;
    }

}
```

非阻塞的链表

构建链表要比构建栈复杂。因为我们要维持头尾两个指针。以put方法来说，我们需要执行两步操作：1. 在尾部插入新的节点。2.将尾部指针指向最新的节点。

我们使用CAS最多只能保证其中的一步是原子执行。那么对于1和2的组合步骤该怎么处理呢?

我们再仔细考虑考虑，其实1和2并不一定要在同一个线程中执行，其他线程在检测到有线程插入了节点，但是没有将tail指向最后的节点时，完全帮忙完成这个操作。

我们看下具体的代码实现：

```
public class LinkedNode<E> {
    public final E item;
    public final AtomicReference<LinkedNode<E>> next;

    public LinkedNode(E item, LinkedNode<E> next){
        this.item=item;
        this.next=new AtomicReference<>(next);
    }
}
```

先构建一个LinkedNode类。

```
public class LinkedQueue<E> {
    private final LinkedNode<E> nullNode= new LinkedNode<>(null, null);
    private final AtomicReference<LinkedNode<E>> head= new AtomicReference<>(nullNode);
    private final AtomicReference<LinkedNode<E>> tail= new AtomicReference<>(nullNode);

    public boolean put(E item){
    LinkedNode<E> newNode = new LinkedNode<>(item, null);
    while (true){
        LinkedNode<E> currentTail= tail.get();
```

```
        LinkedNode<E> tailNext= currentTail.next.get();
        if(currentTail == tail.get()){
            if (tailNext != null) {
                //有其他的线程已经插入了一个节点，但是还没有将tail指向最新的节点
                tail.compareAndSet(currentTail, tailNext);
            }else{
                //没有其他的线程插入节点，那么做两件事情：1. 插入新节点，2.将tail指向最新的节点
                if(currentTail.next.compareAndSet(null, newNode)){
                    tail.compareAndSet(currentTail, newNode);
                }
            }
        }
    }
    }
}
```

第三十二章 java内存模型(JMM)和happens-before

我们知道java程序是运行在JVM中的，而JVM就是构建在内存上的虚拟机，那么内存模型JMM是做什么用的呢?

我们考虑一个简单的赋值问题：

```
int a=100;
```

JMM考虑的就是什么情况下读取变量a的线程可以看到值为100。看起来这是一个很简单的问题，赋值之后不就可以读到值了吗?

但是上面的只是我们源码的编写顺序，当把源码编译之后，在编译器中生成的指令的顺序跟源码的顺序并不是完全一致的。处理器可能采用乱序或者并行的方式来执行指令（在JVM中只要程序的最终执行结果和在严格串行环境中执行结果一致，这种重排序是允许的）。并且处理器还有本地缓存，当将结果存储在本地缓存中，其他线程是无法看到结果的。除此之外缓存提交到主内存的顺序也肯能会变化。

上面提到的种种可能都会导致在多线程环境中产生不同的结果。在多线程环境中，大部分时间多线程都是在执行各自的任务，只有在多个线程需要共享数据的时候，才需要协调线程之间的操作。

而JMM就是JVM中必须遵守的一组最小保证，它规定了对于变量的写入操作在什么时候对其他线程是可见的。

重排序

上面讲了JVM中的重排序，这里我们举个例子，以便大家对重排序有一个更深入的理解：

```
@Slf4j
public class Reorder {

    int x=0, y=0;
    int a=0, b=0;
```

```
    private  void reorderMethod() throws InterruptedException {

        Thread one = new Thread(()->{
            a=1;
            x=b;
        });

        Thread two = new Thread(()->{
            b=1;
            y=a;
        });
        one.start();
        two.start();
        one.join();
        two.join();
        log.info("{},{}", x, y);
    }

    public static void main(String[] args) throws InterruptedException {

        for (int i=0; i< 100; i++){
            new Reorder().reorderMethod();
        }
    }
}
```

上面的例子是一个很简单的并发程序。由于我们没有使用同步限制，所以线程one和two的执行顺序是不定的。有可能one在two之前执行，也有可能在two之后执行，也可能两者同时执行。不同的执行顺序可能会导致不同的输出结果。

同时虽然我们在代码中指定了先执行a=1, 再执行x=b，但是这两条语句实际上是没有关系的，在JVM中完全可能将两条语句重排序成x=b在前，a=1在后，从而导致输出更多意想不到的结果。

Happens-Before

为了保证java内存模型中的操作顺序，JMM为程序中的所有操作定义了一个顺序关系，这个顺序叫做Happens-Before。要想保证操作B看到操作A的结果，不管A和B是在同一线程还是不同线程，那么A和B必须满足Happens-Before的关系。如果两个操作不满足happens-before的关系，那么JVM可以对他们任意重排序。

我们看一下happens-before的规则：

1. 程序顺序规则： 如果在程序中操作A在操作B之前，那么在同一个线程中操作A将会在操作B之前执行。

> 注意，这里的操作A在操作B之前执行是指在单线程环境中，虽然虚拟机会对相应的指令进行重排序，但是最终的执行结果跟按照代码顺序执行是一样的。虚拟机只会对不存在依赖的代码进行重排序。

2. 监视器锁规则： 监视器上的解锁操作必须在同一个监视器上面的加锁操作之前执行。

> 锁我们大家都很清楚了，这里的顺序必须指的是同一个锁，如果是在不同的锁上面，那么其执行顺序也不能得到保证。

3. volatile变量规则： 对volatile变量的写入操作必须在对该变量的读操作之前执行。

> 原子变量和volatile变量在读写操作上面有着相同的语义。

4. 线程启动规则： 线程上对Thread.start的操作必须要在该线程中执行任何操作之前执行。
5. 线程结束规则： 线程中的任何操作都必须在其他线程检测到该线程结束之前执行。
6. 中断规则： 当一个线程再另一个线程上调用interrupt时，必须在被中断线程检测到interrupt调用之前执行。
7. 终结器规则： 对象的构造函数必须在启动该对象的终结器之前执行完毕。
8. 传递性： 如果操作A在操作B之前执行，并且操作B在操作C之前执行，那么操作A必须在操作C之前执行。

上面的规则2很好理解，在加锁的过程中，不允许其他的线程获得该锁，也意味着其他的线程必须等待锁释放之后才能加锁和执行其业务逻辑。

4，5，6，7规则也很好理解，只有开始，才能结束。这符合我们对程序的一般认识。

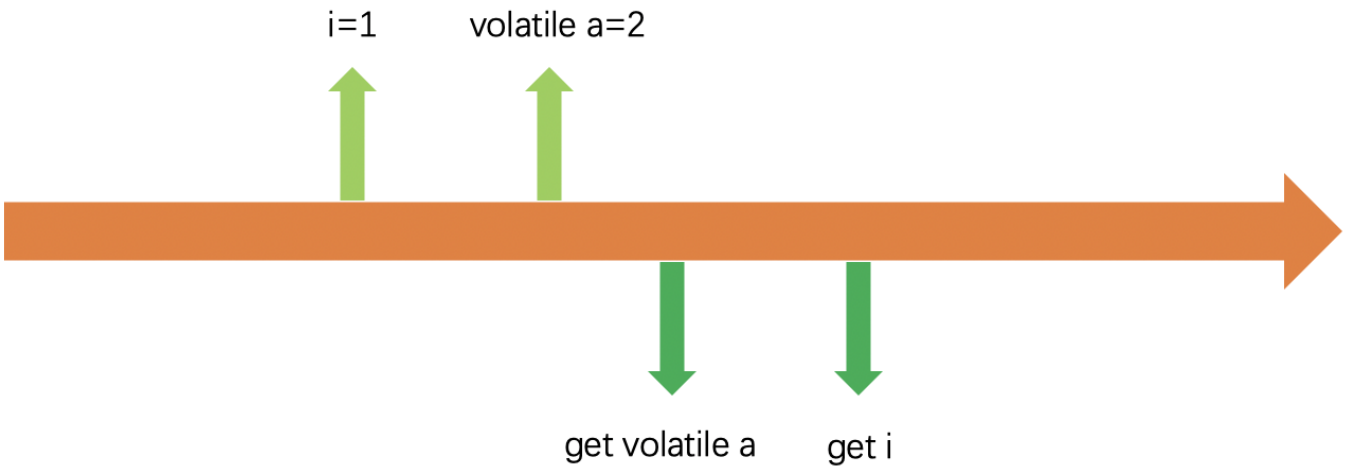
8的传递性相信学过数学的人应该也不难理解。

接下来我们重点讨论一下规则3和规则1的结合。讨论之前我们再总结一下happens-before到底是做什么的。

因为JVM会对接收到的指令进行重排序，为了保证指令的执行顺序，我们才有了happens-before规则。上面讲到的2，3，4，5，6，7规则可以看做是重排序的节点，这些节点是不允许重排序的，只有在这些节点之间的指令才允许重排序。

结合规则1程序顺序规则，我们得到其真正的含义：代码中写在重排序节点之前的指令，一定会在重排序节点执行之前执行。

重排序节点就是一个分界点，它的位置是不能够移动的。看一下下面的直观例子：



线程1中有两个指令：set i=1, set volatile a=2。
线程2中也有两个指令：get volatile a, get i。

按照上面的理论，set和get volatile是两个重排序节点，set必须排在get之前。而依据规则1，代码中set i=1 在set volatile a=2之前，因为set volatile是重排序节点，所以需要遵守程序顺序执行规则，从而set i=1要在set volatile a=2之前执行。同样的道理get volatile a在get i之前执行。最后导致i=1在get i之前执行。

这个操作叫做借助同步。

安全发布

我们经常会用到单例模式来创建一个单的对象，我们看下下面的方法有什么不妥：

```
public class Book {

    private static Book book;

    public static Book getBook(){
        if(book==null){
            book = new Book();
        }
        return book;
    }
}
```

上面的类中定义了一个getBook方法来返回一个新的book对象，返回对象之前，我们先判断了book是否为空，如果不为空的话就new一个book对象。

初看起来，好像没什么问题，但是如果仔细考虑JMM的重排规则，就会发现问题所在。
book=new Book（）其实一个复杂的命令，并不是原子性操作。它大概可以分解为1.分配内存，2.实例化对象，3.将对象和内存地址建立关联。

其中2和3有可能会被重排序，然后就有可能出现book返回了，但是还没有初始化完毕的情况。从而出现不可以预见的错误。

根据上面我们讲到的happens-before规则， 最简单的办法就是给方法前面加上synchronized关键字：

```
public class Book {

    private static Book book;

    public synchronized static Book getBook(){
        if(book==null){
            book = new Book();
        }
        return book;
    }
}
```

我们再看下面一种静态域的实现：

```
public class BookStatic {
    private static BookStatic bookStatic= new BookStatic();

    public static BookStatic getBookStatic(){
        return bookStatic;
    }
}
```

JVM在类被加载之后和被线程使用之前，会进行静态初始化，而在这个初始化阶段将会获得一个锁，从而保证在静态初始化阶段内存写入操作将对所有的线程可见。

上面的例子定义了static变量，在静态初始化阶段将会被实例化。这种方式叫做提前初始化。

下面我们再看一个延迟初始化占位类的模式：

```
public class BookStaticLazy {

    private static class BookStaticHolder{
        private static BookStaticLazy bookStatic= new BookStaticLazy();
    }

    public static BookStaticLazy getBookStatic(){
        return BookStaticHolder.bookStatic;
    }
}
```

上面的类中，只有在调用getBookStatic方法的时候才会去初始化类。

接下来我们再介绍一下双重检查加锁。

```
public class BookDLC {
    private volatile static BookDLC bookDLC;

    public static BookDLC getBookDLC(){
        if(bookDLC == null ){
            synchronized (BookDLC.class){
                if(bookDLC ==null){
                    bookDLC=new BookDLC();
                }
            }
        }
        return bookDLC;
    }
}
```

上面的类中检测了两次bookDLC的值，只有bookDLC为空的时候才进行加锁操作。看起来一切都很完美，但是我们要注意一点，这里bookDLC一定要是volatile。

因为bookDLC的赋值操作和返回操作并没有happens-before，所以可能会出现获取到一个仅部分构造的实例。这也是为什么我们要加上volatile关键词。

初始化安全性

本文的最后，我们将讨论一下在构造函数中含有final域的对象初始化。

对于正确构造的对象，初始化对象保证了所有的线程都能够正确的看到由构造函数为对象给各个final域设置的正确值，包括final域可以到达的任何变量（比如final数组中的元素，final的hashMap等）。

```
public class FinalSafe {
    private final HashMap<String,String> hashMap;

    public FinalSafe(){
        hashMap= new HashMap<>();
        hashMap.put("key1","value1");
    }
}
```

上面的例子中，我们定义了一个final对象，并且在构造函数中初始化了这个对象。那么这个final对象是将不会跟构造函数之后的其他操作重排序。

第三十三章 java多线程之Phaser

前面的文章中我们讲到了CyclicBarrier、CountDownLatch的使用，这里再回顾一下CountDownLatch主要用在一个线程等待多个线程执行完毕的情况，而CyclicBarrier用在多个线程互相等待执行完毕的情况。

Phaser是java 7 引入的新的并发API。他引入了新的Phaser的概念，我们可以将其看成一个一个的阶段，每个阶段都有需要执行的线程任务，任务执行完毕就进入下一个阶段。所以Phaser特别适合使用在重复执行或者重用的情况。

基本使用

在CyclicBarrier、CountDownLatch中，我们使用计数器来控制程序的顺序执行，同样的在Phaser中也是通过计数器来控制。在Phaser中计数器叫做parties， 我们可以通过Phaser的构造函数或者register()方法来注册。

通过调用register()方法，我们可以动态的控制phaser的个数。如果我们需要取消注册，则可以调用arriveAndDeregister()方法。

我们看下arrive：

```
    public int arrive() {
        return doArrive(ONE_ARRIVAL);
    }
```

Phaser中arrive实际上调用了doArrive方法，doArrive接收一个adjust参数，ONE_ARRIVAL表示arrive，ONE_DEREGISTER表示arriveAndDeregister。

Phaser中的arrive()、arriveAndDeregister()方法，这两个方法不会阻塞，但是会返回相应的phase数字，当此phase中最后一个party也arrive以后，phase数字将会增加，即phase进入下一个周期，同时触发（onAdvance）那些阻塞在上一phase的线程。这一点类似于CyclicBarrier的barrier到达机制；更灵活的是，我们可以通过重写onAdvance方法来实现更多的触发行为。

下面看一个基本的使用：

```
    void runTasks(List<Runnable> tasks) {
        final Phaser phaser = new Phaser(1); // "1" to register self
        // create and start threads
        for (final Runnable task : tasks) {
```

```
            phaser.register();
            new Thread() {
                public void run() {
                    phaser.arriveAndAwaitAdvance(); // await all creation
                    task.run();
                }
            }.start();
        }

        // allow threads to start and deregister self
        phaser.arriveAndDeregister();
    }
```

上面的例子中，我们在执行每个Runnable之前调用register（）来注册， 然后调用arriveAndAwaitAdvance()来等待这一个Phaser周期结束。最后我们调用 phaser.arriveAndDeregister();来取消注册主线程。

下面来详细的分析一下运行步骤：

1. final Phaser phaser = new Phaser(1);

这一步我们初始化了一个Phaser，并且指定其现在party的个数为1。

2. phaser.register();

这一步注册Runnable task到phaser，同时将party+1。

3. phaser.arriveAndAwaitAdvance()

这一步将会等待直到所有的party都arrive。这里只会将步骤2中注册的party标记为arrive，而步骤1中初始化的party一直都没有被arrive。

4. phaser.arriveAndDeregister();

在主线程中，arrive了步骤1中的party，并且将party的个数减一。

5. 步骤3中的phaser.arriveAndAwaitAdvance()将会继续执行，因为最后一个phaser在步骤4中arrive了。

多个Phaser周期

Phaser的值是从0到Integer.MAX_VALUE，每个周期过后该值就会加一，如果到达Integer.MAX_VALUE则会继续从0开始。

如果我们执行多个Phaser周期，则可以重写onAdvance方法：

```
    protected boolean onAdvance(int phase, int registeredParties) {
        return registeredParties == 0;
    }
```

onAdvance将会在最后一个arrive（）调用的时候被调用，如果这个时候registeredParties为0的话，该Phaser将会调用isTerminated方法结束该Phaser。

如果要实现多周期的情况，我们可以重写这个方法：

```
protected boolean onAdvance(int phase, int registeredParties) {
                return phase >= iterations || registeredParties == 0;
            }
```

上面的例子中，如果phase次数超过了指定的iterations次数则就会自动终止。

我们看下实际的例子：

```
    void startTasks(List<Runnable> tasks, final int iterations) {
         final Phaser phaser = new Phaser() {
             protected boolean onAdvance(int phase, int registeredParties) {
                 return phase >= iterations || registeredParties == 0;
             }
         };
         phaser.register();
         for (final Runnable task : tasks) {
             phaser.register();
             new Thread() {
                 public void run() {
                     do {
                         task.run();
                         phaser.arriveAndAwaitAdvance();
                     } while (!phaser.isTerminated());
                 }
             }.start();
         }
         phaser.arriveAndDeregister(); // deregister self, don't wait
     }
```

上面的例子将会执行iterations次。

第三十四章 java中Locks的使用

之前文章中我们讲到，java中实现同步的方式是使用synchronized block。在java 5中，Locks被引入了，来提供更加灵活的同步控制。

本文将会深入的讲解Lock的使用。

Lock和Synchronized Block的区别

我们在之前的Synchronized Block的文章中讲到了使用Synchronized来实现java的同步。既然Synchronized Block那么好用，为什么会引入新的Lock呢?

主要有下面几点区别：

1. synchronized block只能写在一个方法里面，而Lock的lock()和unlock()可以分别在不同的方法里面。
2. synchronized block 不支持公平锁，一旦锁被释放，任何线程都有机会获取被释放的锁。而使用 Lock APIs则可以支持公平锁。从而让等待时间最长的线程有限执行。
3. 使用synchronized block，如果线程拿不到锁，将会被Blocked。 Lock API 提供了一个tryLock() 的方法，可

以判断是否可以获得lock，这样可以减少线程被阻塞的时间。
4. 当线程在等待synchronized block锁的时候，是不能被中断的。如果使用Lock API，则可以使用lockInterruptibly()来中断线程。

Lock interface

我们来看下Lock interface的定义, Lock interface定义了下面几个主要使用的方法：

- void lock() - 尝试获取锁，如果获取不到锁，则会进入阻塞状态。
- void lockInterruptibly() - 和lock（）很类似，但是它可以将正在阻塞的线程中断，并抛出java.lang.InterruptedException。
- boolean tryLock() – 这是lock()的非阻塞版本，它回尝试获取锁，并立刻返回是否获取成功。
- boolean tryLock(long timeout, TimeUnit timeUnit) – 和tryLock()很像，只是多了一个尝试获取锁的时间。
- void unlock() – unlock实例。
- Condition newCondition() - 生成一个和当前Lock实例绑定的Condition。

在使用Lock的时候，一定要unlocked，以避免死锁。所以，通常我们我们要在try catch中使用：

```
Lock lock = ...;
lock.lock();
try {
    // access to the shared resource
} finally {
    lock.unlock();
}
```

除了Lock接口，还有一个ReadWriteLock接口，在其中定义了两个方法，实现了读锁和写锁分离：

- Lock readLock() – 返回读锁
- Lock writeLock() – 返回写锁

其中读锁可以同时被很多线程获得，只要不进行写操作。写锁同时只能被一个线程获取。

接下来，我们几个Lock的常用是实现类。

ReentrantLock

ReentrantLock是Lock的一个实现，什么是ReentrantLock（可重入锁）呢?

简单点说可重入锁就是当前线程已经获得了该锁，如果该线程的其他方法在调用的时候也需要获取该锁，那么该锁的lock数量+1，并且允许进入该方法。

> 不可重入锁：只判断这个锁有没有被锁上，只要被锁上申请锁的线程都会被要求等待。实现简单

> 可重入锁：不仅判断锁有没有被锁上，还会判断锁是谁锁上的，当就是自己锁上的时候，那么他依旧可以再次访问临界资源，并把加锁次数加一。

我们看下怎么使用ReentrantLock：

```
    public void perform() {

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

下面是使用tryLock（）的例子：

```
    public void performTryLock() throws InterruptedException {
        boolean isLockAcquired = lock.tryLock(1, TimeUnit.SECONDS);

        if(isLockAcquired) {
            try {
                counter++;
            } finally {
                lock.unlock();
            }
        }
    }
```

ReentrantReadWriteLock

ReentrantReadWriteLock是ReadWriteLock的一个实现。上面也讲到了ReadWriteLock主要有两个方法：

- Read Lock - 如果没有线程获得写锁，那么可以多个线程获得读锁。
- Write Lock - 如果没有其他的线程获得读锁和写锁，那么只有一个线程能够获得写锁。

我们看下怎么使用writeLock：

```
    Map<String,String> syncHashMap = new HashMap<>();
    ReadWriteLock lock = new ReentrantReadWriteLock();

    Lock writeLock = lock.writeLock();

    public void put(String key, String value) {
        try {
            writeLock.lock();
            syncHashMap.put(key, value);
        } finally {
            writeLock.unlock();
        }
    }

    public String remove(String key){
```

```
        try {
            writeLock.lock();
            return syncHashMap.remove(key);
        } finally {
            writeLock.unlock();
        }
    }
```

再看下怎么使用readLock：

```
    Lock readLock = lock.readLock();
    public String get(String key){
        try {
            readLock.lock();
            return syncHashMap.get(key);
        } finally {
            readLock.unlock();
        }
    }

    public boolean containsKey(String key) {
        try {
            readLock.lock();
            return syncHashMap.containsKey(key);
        } finally {
            readLock.unlock();
        }
    }
```

StampedLock

StampedLock也支持读写锁，获取锁的是会返回一个stamp，通过该stamp来进行释放锁操作。

上我们讲到了如果写锁存在的话，读锁是无法被获取的。但有时候我们读操作并不想进行加锁操作，这个时候我们就需要使用乐观读锁。

StampedLock中的stamped类似乐观锁中的版本的概念，当我们在
StampedLock中调用lock方法的时候，就会返回一个stamp，代表锁当时的状态，在乐观读锁的使用过程中，在读取数据之后，我们回去判断该stamp状态是否变化，如果变化了就说明该stamp被另外的write线程修改了，这说明我们之前的读是无效的，这个时候我们就需要将乐观读锁升级为读锁，来重新获取数据。

我们举个例子，先看下write排它锁的情况：

```
    private double x, y;
    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY) { // an exclusively locked method
        long stamp = sl.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }
```

再看下乐观读锁的情况：

```
    double distanceFromOrigin() { // A read-only method
        long stamp = sl.tryOptimisticRead();
        double currentX = x, currentY = y;
        if (!sl.validate(stamp)) {
            stamp = sl.readLock();
            try {
                currentX = x;
                currentY = y;
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }
```

上面使用tryOptimisticRead（）来尝试获取乐观读锁，然后通过sl.validate(stamp)来判断该stamp是否被改变，如果改变了，说明之前的read是无效的，那么需要重新来读取。

最后，StampedLock还提供了一个将read锁和乐观读锁升级为write锁的功能：

```
    void moveIfAtOrigin(double newX, double newY) { // upgrade
        // Could instead start with optimistic, not read mode
        long stamp = sl.readLock();
        try {
            while (x == 0.0 && y == 0.0) {
                long ws = sl.tryConvertToWriteLock(stamp);
                if (ws != 0L) {
                    stamp = ws;
                    x = newX;
                    y = newY;
                    break;
                }
                else {
```

```
                    sl.unlockRead(stamp);
                    stamp = sl.writeLock();
                }
            }
        } finally {
            sl.unlock(stamp);
        }
    }
```

上面的例子是通过使用tryConvertToWriteLock(stamp)来实现升级的。

Conditions

上面讲Lock接口的时候有提到其中的一个方法：

```
Condition newCondition();
```

Condition提供了await和signal方法，类似于Object中的wait和notify。

不同的是Condition提供了更加细粒度的等待集划分。我们举个例子：

```
public class ConditionUsage {
    final Lock lock = new ReentrantLock();
    final Condition notFull  = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public Object take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
```

```
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

上面的例子实现了一个ArrayBlockingQueue，我们可以看到在同一个Lock实例中，创建了两个Condition，分别代表队列未满，队列未空。通过这种细粒度的划分，我们可以更好的控制业务逻辑。

第三十五章 ABA问题的本质及其解决办法

简介

CAS的全称是compare and swap，它是java同步类的基础，java.util.concurrent中的同步类基本上都是使用CAS来实现其原子性的。

CAS的原理其实很简单，为了保证在多线程环境下我们的更新是符合预期的，或者说一个线程在更新某个对象的时候，没有其他的线程对该对象进行修改。在线程更新某个对象（或值）之前，先保存更新前的值，然后在实际更新的时候传入之前保存的值，进行比较，如果一致的话就进行更新，否则失败。

注意，CAS在java中是用native方法来实现的，利用了系统本身提供的原子性操作。

那么CAS在使用中会有什么问题呢？一般来说CAS如果设计的不够完美的话，可能会产生ABA问题，而ABA问题又可以分为两类，我们先看来看一类问题。

> 更多内容请访问www.flydean.com

第一类问题

我们考虑下面一种ABA的情况：

1. 在多线程的环境中，线程a从共享的地址X中读取到了对象A。
2. 在线程a准备对地址X进行更新之前，线程b将地址X中的值修改为了B。
3. 接着线程b将地址X中的值又修改回了A。
4. 最新线程a对地址X执行CAS，发现X中存储的还是对象A，对象匹配，CAS成功。

上面的例子中CAS成功了，但是实际上这个CAS并不是原子操作，如果我们想要依赖CAS来实现原子操作的话可能就会出现隐藏的bug。

第一类问题的关键就在2和3两步。这两步我们可以看到线程b直接替换了内存地址X中的内容。

在拥有自动GC环境的编程语言，比如说java中，2，3的情况是不可能出现的，因为在java中，只要两个对象的地址一致，就表示这两个对象是相等的。

2，3两步可能出现的情况就在像C++这种，不存在自动GC环境的编程语言中。因为可以自己控制对象的生命周期，如果我们从一个list中删除掉了一个对象，然后又重新分配了一个对象，并将其add back到list中去，那么根据MRU memory allocation算法，这个新的对象很有可能和之前删除对象的内存地址是一样的。这样就会导致ABA的问题。

第二类问题

如果我们在拥有自动GC的编程语言中，那么是否仍然存在CAS问题呢?

考虑下面的情况，有一个链表里面的数据是A->B->C,我们希望执行一个CAS操作，将A替换成D，生成链表D->B->C。考虑下面的步骤：

1. 线程a读取链表头部节点A。
2. 线程b将链表中的B节点删掉，链表变成了A->C
3. 线程a执行CAS操作，将A替换成D。

最后我们的到的链表是D->C，而不是D->B->C。

问题出在哪呢？CAS比较的节点A和最新的头部节点是不是同一个节点，它并没有关心节点A在步骤1和3之间是否内容发生变化。

我们举个例子：

```
public void useABAReference(){
        CustUser a= new CustUser();
        CustUser b= new CustUser();
        CustUser c= new CustUser();
        AtomicReference<CustUser> atomicReference= new AtomicReference<>(a);
        log.info("{}",atomicReference.compareAndSet(a,b));
        log.info("{}",atomicReference.compareAndSet(b,a));
        a.setName("change for new name");
        log.info("{}",atomicReference.compareAndSet(a,c));
    }
```

上面的例子中，我们使用了AtomicReference的CAS方法来判断对象是否发生变化。在CAS b和a之后，我们将a的name进行了修改，我们看下最后的输出结果：

```
[main] INFO com.flydean.aba.ABAUsage - true
[main] INFO com.flydean.aba.ABAUsage - true
[main] INFO com.flydean.aba.ABAUsage - true
```

三个CAS的结果都是true。说明CAS确实比较的两者是否为统一对象，对其中内容的变化并不关心。

第二类问题可能会导致某些集合类的操作并不是原子性的，因为你并不能保证在CAS的过程中，有没有其他的节点发送变化。

第一类问题的解决

第一类问题在存在自动GC的编程语言中是不存在的，我们主要看下怎么在C++之类的语言中解决这个问题。

根据官方的说法，第一类问题大概有四种解法：

1. 使用中间节点 - 使用一些不代表任何数据的中间节点来表示某些节点是标记被删除的。
2. 使用自动GC。
3. 使用hazard pointers - hazard pointers 保存了当前线程正在访问的节点的地址，在这些hazard pointers中的节点不能够被修改和删除。

4. 使用read-copy update (RCU) - 在每次更新的之前，都做一份拷贝，每次更新的是拷贝出来的新结构。

第二类问题的解决

第二类问题其实算是整体集合对象的CAS问题了。一个简单的解决办法就是每次做CAS更新的时候再添加一个版本号。如果版本号不是预期的版本，就说明有其他的线程更新了集合中的某些节点，这次CAS是失败的。

我们举个AtomicStampedReference的例子：

```
public void useABAStampReference(){
        Object a= new Object();
        Object b= new Object();
        Object c= new Object();
        AtomicStampedReference<Object> atomicStampedReference= new
AtomicStampedReference(a,0);
        log.info("{}",atomicStampedReference.compareAndSet(a,b,0,1));
        log.info("{}",atomicStampedReference.compareAndSet(b,a,1,2));
        log.info("{}",atomicStampedReference.compareAndSet(a,c,0,1));
    }
```

AtomicStampedReference的compareAndSet方法，多出了两个参数，分别是expectedStamp和newStamp，两个参数都是int型的，需要我们手动传入。

总结

ABA问题其实是由两类问题组成的，需要我们分开来对待和解决。

第三十六章 并发和Read-copy update(RCU)

简介

在上一篇文章中的并发和ABA问题的介绍中，我们提到了要解决ABA中的memory reclamation问题，有一个办法就是使用RCU。

详见ABA问题的本质及其解决办法,今天本文将会深入的探讨一下RCU是什么，RCU和COW(Copy-On-Write)之间的关系。

RCU(Read-copy update)是一种同步机制，并在2002年被加入了Linux内核中。它的优点就是可以在更新的过程中，运行多个reader进行读操作。

熟悉锁的朋友应该知道，对于排它锁，同一时间只允许一个操作进行，不管这个操作是读还是写。

对于读写锁，可以允许同时读，但是不能允许同时写，并且这个写锁是排他的，也就是说写的同时是不允许进行读操作的。

RCU可以支持一个写操作和多个读操作同时进行。

> 更多内容请访问www.flydean.com

Copy on Write和RCU

什么是Copy on Write? 它和read copy update有什么关系呢?

我们把Copy on Write简写为COW，COW是并发中经常会用到的一种算法，java里面就有java.util.concurrent.CopyOnWriteArrayList和java.util.concurrent.CopyOnWriteArraySet。

COW的本质就是，在并发的环境中，如果想要更新某个对象，首先将它拷贝一份，在这个拷贝的对象中进行修改，最后把指向原对象的指针指回更新好的对象。

CopyOnWriteArrayList和CopyOnWriteArraySet中的COW使用在遍历的时候。

我们知道使用Iterator来遍历集合的时候，是不允许在Iterator外部修改集合的数据的，只能在Iterator内部遍历的时候修改，否则会抛出ConcurrentModificationException。

而对于CopyOnWriteArrayList和CopyOnWriteArraySet来说，在创建Iterator的时候，就对原List进行了拷贝，Iterator的遍历是在拷贝过后的List中进行的，这时候如果其他的线程修改了原List对象，程序正常执行，不会抛出ConcurrentModificationException。

同时CopyOnWriteArrayList和CopyOnWriteArraySet中的Iterator是不支持remove，set，add方法的，因为这是拷贝过来的对象，在遍历过后是要被丢弃的。在它上面的修改是没有任何意义的。

在并发情况下，COW其实还有一个问题没有处理，那就是对于拷贝出来的对象什么时候回收的问题，是不是可以马上将对象回收？有没有其他的线程在访问这个对象？ 处理这个问题就需要用到对象生命周期的跟踪技术，也就是RCU中的RCU-sync。

所以RCU和COW的关系就是：RCU是由RCU-sync和COW两部分组成的。

因为java中有自动垃圾回收功能，我们并不需要考虑拷贝对象的生命周期问题，所以在java中我们一般只看到COW，看不到RCU。

RCU的流程和API

我们将RCU和排它锁和读写锁进行比较。

对于排它锁来说，需要这两个API：

```
lock()
unlock()
```

对于读写锁来说，需要这四个API：

```
read_lock()
read_unlock()
write_lock()
write_unlock()
```

而RCU需要下面三个API：

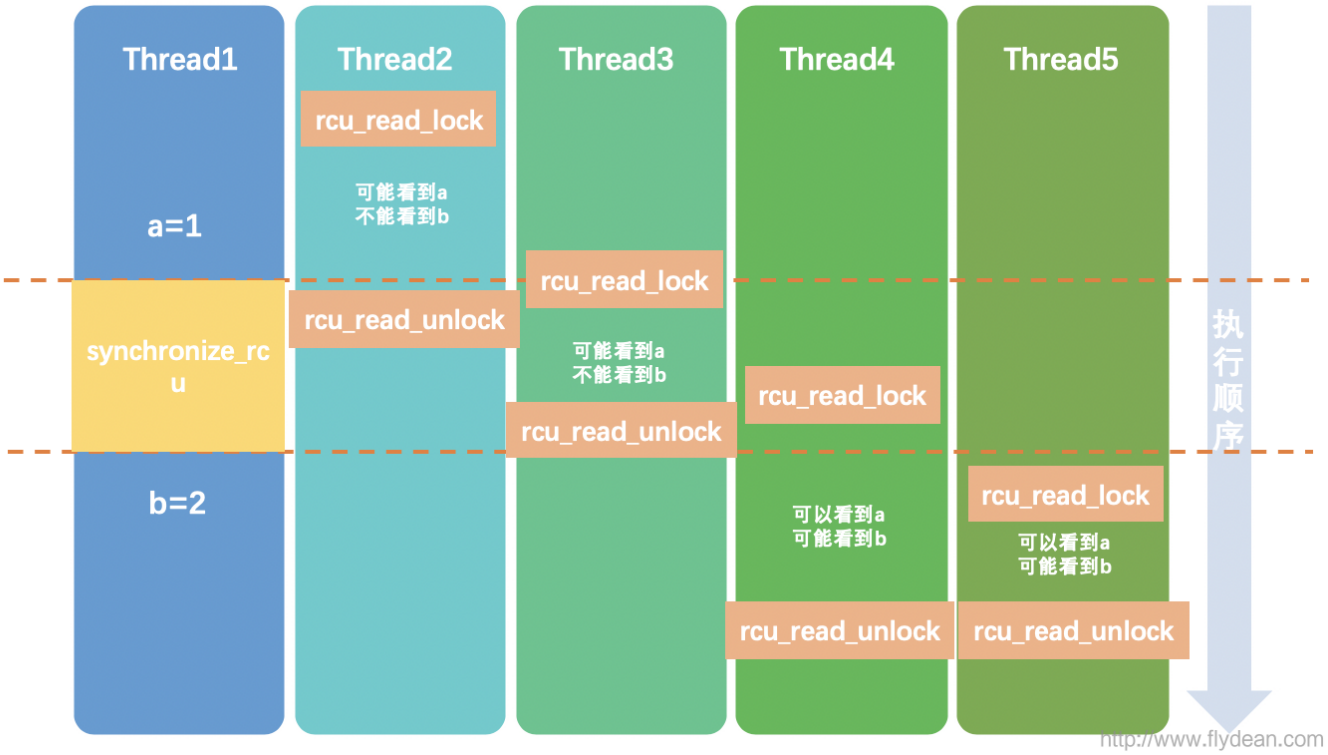
```
rcu_read_lock()
rcu_read_unlock()
synchronize_rcu()
```

rcu_read_lock和rcu_read_unlock必须是成对出现的，并且synchronize_rcu不能出现在rcu_read_lock和rcu_read_unlock之间。

虽然RCU并不提供任何排他锁，但是RCU必须要满足下面的两个条件：

1. 如果Thread1(T1)中synchronize_rcu方法在Thread2（T2)的rcu_read_lock方法之前返回，则happens before synchronize_rcu的操作一定在T2的rcu_read_lock方法之后可见。
2. 如果T2的rcu_read_lock方法调用在T1的synchronize_rcu方法调用之前，则happens after synchronize_rcu的操作一定在T2的rcu_read_unlock方法之前不可见。

听起来很拗口，没关系，我们画个图来理解一下：



记住RCU比较的是synchronize_rcu和rcu_read_lock的顺序。

Thread2和Thread3中rcu_read_lock在synchronize_rcu之前执行，则b=2在T2，T3中一定不可见。

Thread4中rcu_read_lock虽然在synchronize_rcu启动之后才开始执行的，但是rcu_read_unlock是在synchronize_rcu返回之后才执行的，所以可以等同于看做Thread5的情况。

Thread5中，rcu_read_lock在synchronize_rcu返回之后才执行的，所以a=1一定可见。

RCU要注意的事项

RCU虽然没有提供锁的机制，但允许同时多个线程进行读操作。注意，RCU同时只允许一个synchronize_rcu操作，所以需要我们自己来实现synchronize_rcu的排它锁操作。

所以对于RCU来说，它是一个写多个读的同步机制，而不是多个写多个读的同步机制。

RCU的java实现

最后放上一段大神的RCU的java实现代码：

```
public class RCU {
    final static long NOT_READING = Long.MAX_VALUE;
    final static int MAX_THREADS = 128;
    final AtomicLong reclaimerVersion = new AtomicLong(0);
    final AtomicLongArray readersVersion = new AtomicLongArray(MAX_THREADS);

    public RCU() {
        for (int i=0; i < MAX_THREADS; i++) readersVersion.set(i, NOT_READING);
    }

    public static int getTID() {
        return (int)(Thread.currentThread().getId() % MAX_THREADS);
    }

    public void read_lock(final int tid) {  // rcu_read_lock()
        final long rv = reclaimerVersion.get();
        readersVersion.set(tid, rv);
        final long nrv = reclaimerVersion.get();
        if (rv != nrv) readersVersion.lazySet(tid, nrv);
    }

    public void read_unlock(final int tid) { // rcu_read_unlock()
        readersVersion.set(tid, NOT_READING);
    }

    public void synchronize_rcu() {
        final long waitForVersion = reclaimerVersion.incrementAndGet();
        for (int i=0; i < MAX_THREADS; i++) {
            while (readersVersion.get(i) < waitForVersion) { } // spin
        }
    }
}
```

简单讲解一下这个RCU的实现：

readersVersion是一个长度为128的Long数组，里面存放着每个reader的读数。默认情况下reader存储的值是NOT_READING，表示未存储任何数据。

在RCU初始化的时候，将会初始化这些reader。

read_unlock方法会将reader的值重置为NOT_READING。

reclaimerVersion存储的是修改的数据，它的值将会在synchronize_rcu方法中进行更新。

同时synchronize_rcu将会遍历所有的reader，只有当所有的reader都读取完毕才继续执行。

最后，read_lock方法将会读取reclaimerVersion的值。这里会读取两次，如果两次的结果不同，则会调用readersVersion.lazySet方法，延迟设置reader的值。

为什么要读取两次呢？因为虽然reclaimerVersion和readersVersion都是原子性操作，但是在多线程环境中，并不能保证reclaimerVersion一定就在readersVersion之前执行，所以我们需要添加一个内存屏障：memory barrier来实现这个功能。

总结

本文介绍了RCU算法和应用。希望大家能够喜欢。

第三十七章 同步类的基础 AbstractQueuedSynchronizer(AQS)

我们之前介绍了很多同步类，比如ReentrantLock,Semaphore, CountDownLatch, ReentrantReadWriteLock,FutureTask等。

AQS封装了实现同步器时设计的大量细节问题。他提供了FIFO的wait queues并且提供了一个int型的state表示当前的状态。

根据JDK的说明，并不推荐我们直接使用AQS，我们通常需要构建一个内部类来继承AQS并按照需要重写下面几个方法：

- tryAcquire
- tryRelease
- tryAcquireShared
- tryReleaseShared
- isHeldExclusively

在这些方法中，我们需要调用getState, setState 或者 compareAndSetState这三种方法来改变state值。

上面的方法提到了两种操作，独占操作（如：ReentrantLock）和共享操作(如：Semaphore,CountdownLatch)。

两种的区别在于同一时刻能否有多个线程同时获取到同步状态。

比如我们运行同时多个线程去读，但是同时只允许一个线程去写，那么这里的读锁就是共享操作，而写锁就是独占操作。

在基于QAS构建的同步类中，最基本的操作就是获取操作和释放操作。而这个state就表示的是这些获取和释放操作所依赖的值。

State是一个int值，你可以使用它来表示任何状态，比如ReentrantLock用它来表示所有者线程重复获取该锁的次数。Semaphore用它来表示剩余的许可量，而FutureTask用它来表示任务的状态（开始，运行，完成或者取消）。当然你还可以自定义额外的状态变量来表示其他的信息。

下的伪代码表示的是AQS中获取和释放操作的形式：

```
   Acquire:
       while (!tryAcquire(arg)) {
          enqueue thread if it is not already queued;
          possibly block current thread;
       }

   Release:
       if (tryRelease(arg))
          unblock the first queued thread;
```

获取操作，首先判断当前状态是否允许获取操作，如果如果不允许，则将当前的线程入Queue，并且有可能阻塞当前线程。

释放操作，则先判断是否运行释放操作，如果允许，则解除queue中的thread，并运行。

我们看一个具体的实现：

```
public class AQSUsage {

    private final Sync sync= new Sync();

    private class Sync extends AbstractQueuedSynchronizer{
        protected int tryAcquireShared(int ignored){
            return (getState() ==1 )? 1: -1;
        }
        protected boolean tryReleaseShared(int ignored){
            setState(1);
            return true;
        }
    }

    public void release() {
        sync.releaseShared(0);
    }
    public void acquire() throws InterruptedException {
        sync.acquireSharedInterruptibly(0);
    }
}
```

上面的例子中，我们定义了一个内部类Sync，在这个类中我们实现了tryAcquireShared和tryReleaseShared两个方法，在这两个方法中我们判断并设置了state的值。

sync.releaseShared和sync.acquireSharedInterruptibly会分别调用tryAcquireShared和tryReleaseShared方法。

前面我们也提到了很多同步类都是使用AQS来实现的，我们可以再看看其他标准同步类中tryAcquire的实现。

首先看下ReentrantLock:

```
   final boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
```

```
            int c = getState();
            if (c == 0) {
                if (compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0) // overflow
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }
```

ReentrantLock只支持独占锁。所以它需要实现tryAcquire方法。除此之外它还维护了一个owner变量来保存当前所有者线程的标志符，从而来实现可重入锁。

我们再看下Semaphore和CountDownLatch的实现，因为他们是共享操作，所以需要实现tryAcqureShared方法：

```
        final int tryAcquireShared(int acquires) {
            for (;;) {
                int available = getState();
                int remaining = available - acquires;
                if (remaining < 0 ||
                    compareAndSetState(available, remaining))
                    return remaining;
            }
        }
```

第三十八章 java并发Exchanger的使用

简介

Exchanger是java 5引入的并发类，Exchanger顾名思义就是用来做交换的。这里主要是两个线程之间交换持有的对象。当Exchanger在一个线程中调用exchange方法之后，会等待另外的线程调用同样的exchange方法。

两个线程都调用exchange方法之后，传入的参数就会交换。

类定义

```
public class Exchanger<V>
```

其中V表示需要交换的对象类型。

类继承

```
java.lang.Object
↳ java.util.concurrent.Exchanger<V>
```

Exchanger直接继承自Object。

构造函数

```
Exchanger()
```

Exchanger提供一个无参构造函数。

两个主要方法

1. public V exchange(V x) throws InterruptedException

当这个方法被调用的时候，当前线程将会等待直到其他的线程调用同样的方法。当其他的线程调用exchange之后，当前线程将会继续执行。

在等待过程中，如果有其他的线程interrupt当前线程，则会抛出InterruptedException。

2. public V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException, TimeoutException

和第一个方法类似，区别是多了一个timeout时间。如果在timeout时间之内没有其他线程调用exchange方法，则会抛出TimeoutException。

具体的例子

我们先定义一个带交换的类：

```
@Data
public class CustBook {

    private String name;
}
```

然后定义两个Runnable，在run方法中调用exchange方法：

```
@Slf4j
public class ExchangerOne implements Runnable{

    Exchanger<CustBook> ex;

    ExchangerOne(Exchanger<CustBook> ex){
      this.ex=ex;
    }
```

```
    @Override
    public void run() {
    CustBook custBook= new CustBook();
        custBook.setName("book one");

        try {
            CustBook exhangeCustBook=ex.exchange(custBook);
            log.info(exhangeCustBook.getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
@Slf4j
public class ExchangerTwo implements Runnable{

    Exchanger<CustBook> ex;

    ExchangerTwo(Exchanger<CustBook> ex){
      this.ex=ex;
    }

    @Override
    public void run() {
    CustBook custBook= new CustBook();
        custBook.setName("book two");

        try {
            CustBook exhangeCustBook=ex.exchange(custBook);
            log.info(exhangeCustBook.getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

最后在主方法中调用：

```
public class ExchangerUsage {

    public static void main(String[] args) {
        Exchanger<CustBook> exchanger = new Exchanger<>();
        // Starting two threads
        new Thread(new ExchangerOne(exchanger)).start();
        new Thread(new ExchangerTwo(exchanger)).start();
    }
}
```

我们看下结果：

```
22:14:09.069 [Thread-1] INFO com.flydean.ExchangerTwo - book one
22:14:09.073 [Thread-0] INFO com.flydean.ExchangerOne - book two
```

可以看到对象已经被交换了。

结语

Exchanger在两个线程需要交换对象的时候非常好用。大家可以在实际工作生活中使用。

本文的例子https://github.com/ddean2009/learn-java-concurrency/

> 欢迎关注我的公众号:程序那些事，更多精彩等着您！
> 更多内容请访问 www.flydean.com