共享模型之并发工具

一、线程池

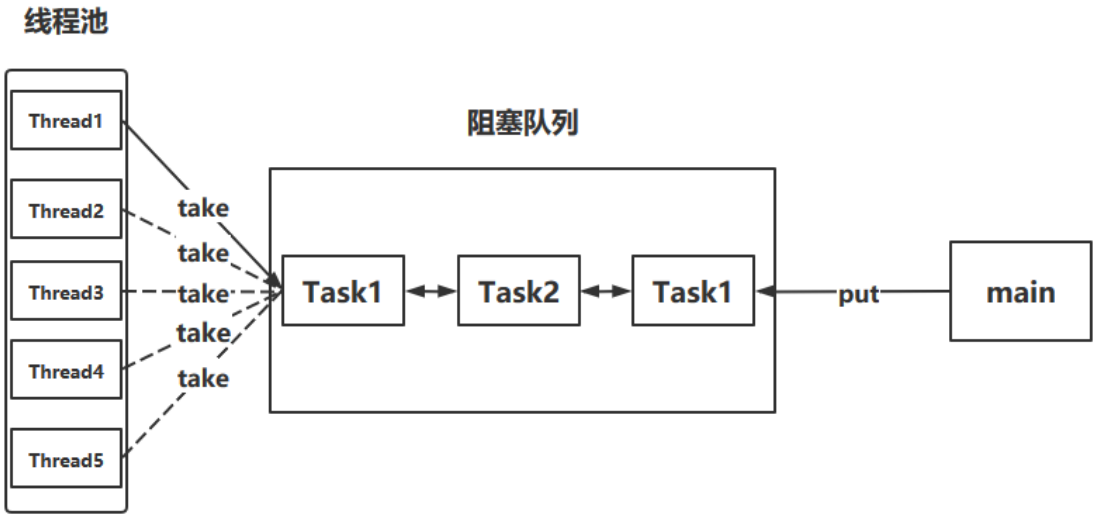
1.1 概述

线程池（Thread Pool）是一种基于池化思想管理线程的工具，经常出现在多线程服务器中，如MySQL。

线程过多会带来额外的开销，其中包括创建销毁线程的开销、调度线程的开销等等，同时也降低了计算机的整体性能。线程池维护多个线程，等待监督管理者分配可并发执行的任务。这种做法，一方面避免了处理任务时创建销毁线程开销的代价，另一方面避免了线程数量膨胀导致的过分调度问题，保证了对内核的充分利用。

线程池可以带来一系列好处：

- **降低资源消耗**：通过池化技术重复利用已创建的线程，降低线程创建和销毁造成的损耗。
- **提高响应速度**：任务到达时，无需等待线程创建即可立即执行。
- **提高线程的可管理性**：线程是稀缺资源，如果无限制创建，不仅会消耗系统资源，还会因为线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、调优和监控。
- **提供更多更强大的功能**：线程池具备可拓展性，允许开发人员向其中增加更多的功能。比如延时定时线程池ScheduledThreadPoolExecutor，就允许任务延期执行或定期执行。



阻塞队列：

- 成员变量
  - 任务列表，用 `ArrayDeque` 实现
  - 锁，用 `ReentrantLock` 实现
  - 消费者条件变量，当没有任务可以消费的时候, 进入消费者条件变量中等待
  - 生产者条件变量，当阻塞队列塞满任务的时候, 没有空间, 此时进入生产者条件变量中等待
  - 容量
- 方法
  - 将任务交给线程池中的线程，take（没有超时时间）
    - 如果队列为空，加入消费者条件变量等待，直到队列不为空进入下一步
    - 获取阻塞队列的头元素

        - 唤醒生产者条件变量挂起的进程（阻塞队列已满，生产者无法添加任务）
    - 将任务交给线程池中的线程，poll（有超时时间）
    - 向阻塞队列中添加任务，put
        - 如果阻塞队列已满，加入生产者条件变量等待，直到队列有空闲进入下一步
        - 向阻塞队列尾部添加任务
        - 唤醒消费者条件变量中挂起的进程（阻塞队列为空，消费者无法执行任务）
    - 向阻塞队列中添加任务，offer（有超时时间）
    - 尝试向阻塞队列中添加任务，tryPut
        - 如果队列已满，执行拒绝策略
        - 如果空闲，加入阻塞队列
    - 获取队列的大小

线程池：

- 成员变量
    - 阻塞队列
    - 线程集合 `workers` ，集合泛型为Worker类（内部类，继承了Thread类）
    - 核心线程数 coreSize
    - 任务超时时间 timeout
    - 时间单位 TimeUtil类对象
    - 拒绝策略
- 内部类
    - Worker类（Thread子类）
        - 成员变量
            - Runnable对象，变量名为task
        - 方法
            - run方法
                - 如果task不为空，直接调用task的run方法执行任务
                - 当task执行完毕，从阻塞队列中获取任务并且执行
- 方法
    - `execute()` ：将任务交给线程（worker对象）执行

        - 当任务数没有超过核心线程数 `coreSize` ，直接交给 `worker` 对象（Worker类的构造方法会传入任务对象，任务对象通过对应的worker对象的run方法执行）
        - 如果任务数超过核心线程数 `coreSize`
            - 死等
            - 带超时等待
            - 让调用者放弃执行任务
            - 抛出异常
            - 让调用者自己执行任务

拒绝策略（interface）

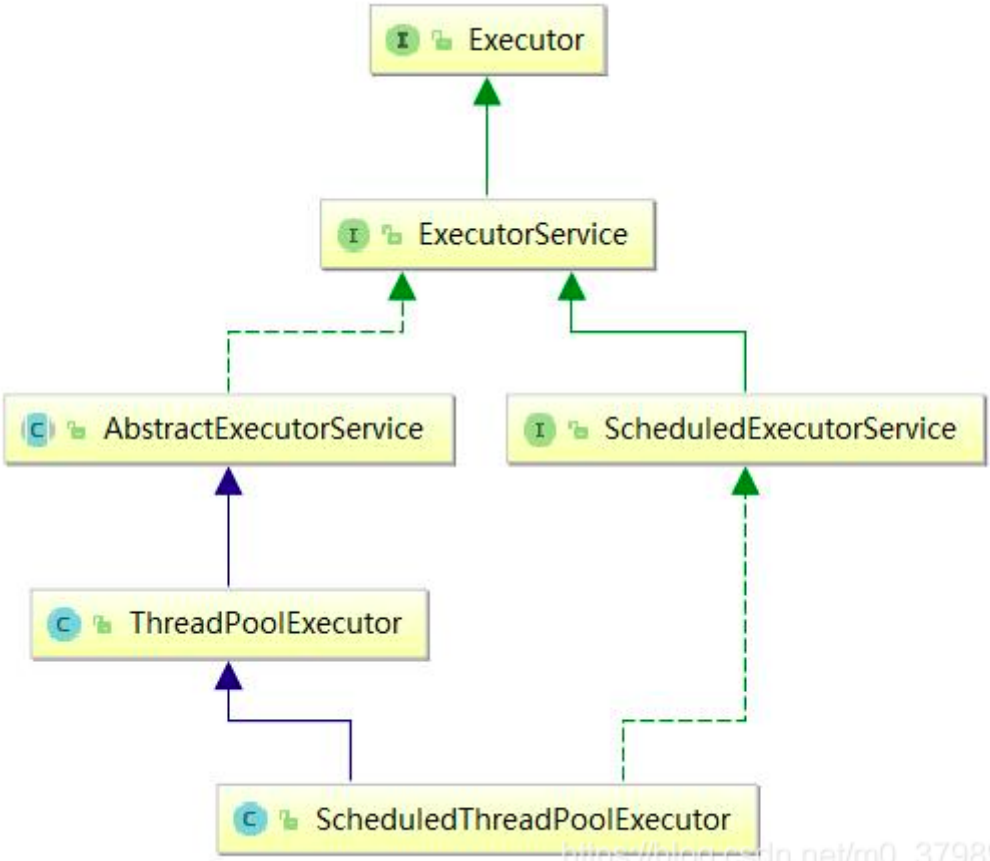
- 方法
    - `reject()` ,拒绝策略方法，传入的参数为阻塞队列和任务

main：

- 充当生产者
- put 向阻塞队列中添加任务

1.2 ThreadPoolExecutor

1.2.1 线程池的继承关系



1.2.2 线程池状态

ThreadPoolExecutor 使用 int 的高 3 位来表示线程池状态，低 29 位表示线程数量

状态名称	高3位的值	描述
RUNNING	111	接收新任务，同时处理任务队列中的任务
SHUTDOWN	000	不接受新任务，但是处理任务队列中的任务
STOP	001	中断正在执行的任务，同时抛弃阻塞队列中的任务
TIDYING	010	任务执行完毕，活动线程为0时，即将进入终结阶段
TERMINATED	011	终结状态

从数字上比较，TERMINATED > TIDYING > STOP > SHUTDOWN > RUNNING

> 注：第一位为符号位，故RUNNING的数值为负值，故最小

这些信息存储在一个原子变量中，目的是将线程池状态与线程个数合二为一，这样就可以用一次cas原子操作进行赋值

1.2.4 构造方法

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

- corePoolSize：核心线程数
- maximumPoolSize：最大线程数
  - maximumPoolSize - corePoolSize = 救急线程数 （救急线程在**没有空闲的核心线程**和**任务队列满了**的情况才使用救急线程）
- keepAliveTime：救急线程空闲时的最大生存时间 (核心线程可以一直运行)
- unit：时间单位 (针对救急线程)
- workQueue

  ：阻塞队列（存放任务）
  - 有界阻塞队列 ArrayBlockingQueue
  - 无界阻塞队列 LinkedBlockingQueue
  - 最多只有一个同步元素的 SynchronousQueue
  - 优先队列 PriorityBlockingQueue
- threadFactory：线程工厂（给线程取名字）
- handler：拒绝策略 （如果所有的救济线程和核心线程都在执行任务时，才会采用拒接策略），下面是JDK提供的4种实现
  - `AbortPolicy 中止策略`：丢弃任务并抛出RejectedExecutionException异常。**默认策略**
  - 这是线程池默认的拒绝策略，在任务不能再提交的时候，抛出异常，及时反馈程序运行状态。如果是比较关键的业务，推荐使用此拒绝策略，这样子在系统不能承载更大的并发量的时候，能够及时的通过异常发现。
    - 功能：当触发拒绝策略时，直接抛出拒绝执行的异常，中止策略的意思也就是打断当前执行流程.
    - 使用场景：这个就没有特殊的场景了，但是有一点要正确处理抛出的异常。ThreadPoolExecutor中默认的策略就是AbortPolicy，

ExecutorService接口的系列ThreadPoolExecutor因为都没有显示的设置拒绝策略，所以默认的都是这个。但是请注意，ExecutorService中的线程池实例队列都是无界的，也就是说把内存撑爆了都不会触发拒绝策略。当自己自定义线程池实例时，使用这个策略一定要处理好触发策略时抛的异常，因为他会打断当前的执行流程。

- `DiscardPolicy 丢弃策略`：丢弃任务，但是不抛出异常。如果线程队列已满，则后续提交的任务都会被丢弃，且是**静默**丢弃。
  - 使用此策略，可能会使我们无法发现系统的异常状态。建议是一些无关紧要的业务采用此策略。例如，本人的博客网站统计阅读量就是采用的这种拒绝策略。
  - 功能：直接静悄悄的丢弃这个任务，不触发任何动作。
    使用场景：如果你提交的任务无关紧要，你就可以使用它 。因为它就是个空实现，会悄无声息的吞噬你的的任务。所以这个策略基本上不用了。
- `DiscardOldestPolicy 弃老策略`：丢弃队列最前面的任务，本任务取而代之。
  - 此拒绝策略，是一种喜新厌旧的拒绝策略。是否要采用此种拒绝策略，还得根据实际业务是否允许丢弃老任务来认真衡量。
- 功能：如果线程池未关闭，就弹出队列头部的元素，然后尝试执行
  - 使用场景：这个策略还是会丢弃任务，丢弃时也是毫无声息，但是特点是丢弃的是老的未执行的任务，而且是待执行优先级较高的任务。基于这个特性，想到的场景就是，发布消息和修改消息，当消息发布出去后，还未执行，此时更新的消息又来了，这个时候未执行的消息的版本比现在提交的消息版本要低就可以被丢弃了。因为队列中还有可能存在消息版本更低的消息会排队执行，所以在真正处理消息的时候一定要做好消息的版本比较。
- `CallerRunsPolicy 调用者运行策略`：由调用线程处理该任务。
  - 功能：当触发拒绝策略时，只要线程池没有关闭，就由提交任务的当前线程处理。
  - 使用场景：一般在不允许失败的、对性能要求不高、并发量较小的场景下使用，因为线程池一般情况下不会关闭，也就是提交的任务一定会被运行，但是由于是调用者线程自己执行的，当多次提交任务时，就会阻塞后续任务执行，性能和效率自然就慢了。

> 注：除了JDK提供的4种实现之外，其他框架也提供了它们自己的拒绝策略
>
> - Dubbo：在抛出RejectExecutionException异常之前记录日志，并dump线程栈信息，方便定位问题
> - Netty：创建一个新线程来执行任务
> - ActiveMQ：带超时等待60s尝试放入等待队列

1.2.5 执行/提交任务

```
// 执行任务
void execute(Runnable command);

// 提交任务 task，用返回值 Future 获得任务执行结果，Future的原理就是利用我们之前讲到的保护性暂停模式来接受返回结果的，主线程可以执行 FutureTask.get() 方法来等待任务执行完成
<T> Future<T> submit(Callable<T> task);

// 提交 tasks 中所有任务
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException;

// 提交 tasks 中所有任务，带超时时间
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit) throws InterruptedException;

// 提交 tasks 中所有任务，哪个任务先成功执行完毕，返回此任务执行结果，其它任务取消
<T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException;

// 提交 tasks 中所有任务，哪个任务先成功执行完毕，返回此任务执行结果，其它任务取消，带超时时间
<T> T invokeAny(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException;
```

1.2.6 关闭线程池

(1) shutdown()

- **将线程池的状态改为 SHUTDOWN**
- **不再接受新任务，但是会将阻塞队列中的任务执行完**

```
/**
* 将线程池的状态改为 SHUTDOWN
* 不再接受新任务，但是会将阻塞队列中的任务执行完
*/
public void shutdown() {
```

```
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();

        // 修改线程池状态为 SHUTDOWN
        advanceRunState(SHUTDOWN);

        // 中断空闲线程（没有执行任务的线程）
        // Idle：空闲的
        interruptIdleWorkers();
        onShutdown(); // hook for ScheduledThreadPoolExecutor，扩
展点
    } finally {
        mainLock.unlock();
    }
    // 尝试终结，不一定成功
    tryTerminate();
}

final void tryTerminate() {
    for (;;) {
        int c = ctl.get();
        // 终结失败的条件
        // 线程池状态为RUNNING
        // 线程池状态为 RUNNING SHUTDOWN STOP （状态值大于TIDYING）
        // 线程池状态为SHUTDOWN，但阻塞队列中还有任务等待执行
        if (isRunning(c) ||
            runStateAtLeast(c, TIDYING) ||
            (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
            return;

        // 如果活跃线程数不为0
        if (workerCountOf(c) != 0) { // Eligible to terminate
            // 中断空闲线程
            interruptIdleWorkers(ONLY_ONE);
            return;
        }

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // 处于可以终结的状态
            // 通过CAS将线程池状态改为TIDYING
            if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
```

```
                try {
                    terminated();
                } finally {
                    // 通过CAS将线程池状态改为TERMINATED
                    ctl.set(ctlOf(TERMINATED, 0));
                    termination.signalAll();
                }
                return;
            }
        } finally {
            mainLock.unlock();
        }
        // else retry on failed CAS
    }
}
```

(2) shutdownNow()

- **将线程池的状态改为 STOP**
- **不再接受新任务，也不会在执行阻塞队列中的任务**
- 会将阻塞队列中未执行的任务返回给调用者
- 并用 interrupt 的方式中断正在执行的任务

```
/**
* 将线程池的状态改为 STOP
* 不再接受新任务，也不会在执行阻塞队列中的任务
* 会将阻塞队列中未执行的任务返回给调用者
* 并用 interrupt 的方式中断正在执行的任务
*/
public List<Runnable> shutdownNow() {
    List<Runnable> tasks;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        checkShutdownAccess();

        // 修改状态为STOP，不执行任何任务
        advanceRunState(STOP);

        // 中断所有线程
        interruptWorkers();

        // 将未执行的任务从队列中移除，然后返回给调用者
```

```
            tasks = drainQueue();
        } finally {
            mainLock.unlock();
        }
        // 尝试终结，一定会成功，因为阻塞队列为空了
        tryTerminate();
        return tasks;
    }
```

（3）其他方法

```
// 不在 RUNNING 状态的线程池，此方法就返回 true
boolean isShutdown();

// 线程池状态是否是 TERMINATED
boolean isTerminated();

// 调用 shutdown 后，由于调用使线程结束线程的方法是异步的并不会等待所有任务运行结束就返回，因此如果它想在线程池 TERMINATED 后做些其它事情，可以利用此方法等待
boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;
```

1.3 异步模式之工作模式

1.3.1 定义

让有限的工作线程（Worker Thread）来轮流异步处理无限多的任务 。也可以将其归类为分工模式，它的典型实现就是 线程池 ，也体现了经典设计模式中的 享元模式 。

例如，海底捞的服务员（线程），轮流处理每位客人的点餐（任务），如果为每位客人都配一名专属的服务员，那么成本就太高了（对比另一种多线程设计模式：Thread-Per-Message）

> 注意: **不同任务类型应该使用不同的线程池，这样能够避免饥饿，并能提升效率**
>
> 例如，如果一个餐馆的工人既要招呼客人（任务类型A），又要到后厨做菜（任务类型B）显然效率不咋地，分成服务员（线程池A）与厨师（线程池B）更为合理，当然你能想到更细致的分工

1.3.2 饥饿

固定大小线程池会有饥饿现象

例如：

1. 两个工人是同一个线程池中的两个线程
2. 他们要做的事情是：为客人点餐和到后厨做菜，这是两个阶段的工作
    1. 客人点餐：必须先点完餐，等菜做好，上菜，在此期间处理点餐的工人必须等待
    2. 后厨做菜：没啥说的，做就是了
3. 比如工人A 处理了点餐任务，接下来它要等着 工人B 把菜做好，然后上菜，他俩也配合的蛮好; 但现在同时来了两个客人，这个时候工人A 和工人B 都去处理点餐了，这时没人做饭了，饥饿

解决方法：增加线程池的大小，不过不是根本解决方案，还是前面提到的， 不同的任务类型，采用不同的线程池

1.3.3 线程池中线程设置多少为好?

过小会导致程序不能充分地利用系统资源、容易导致饥饿，过大会导致更多的线程上下文切换，占用更多内存

- **CPU 密集型运算**
    - 通常采用 `cpu 核数 + 1` 能够实现最优的 CPU 利用率，+1 是保证当线程由于页缺失故障（操作系统）或其它原因导致暂停时，额外的这个线程就能顶上去，保证 CPU 时钟周期不被浪费
- I/O 密集型运算
    - CPU 不总是处于繁忙状态，例如，当你执行业务计算时，这时候会使用 CPU 资源，但当你执行 I/O 操作时、远程RPC 调用时，包括进行数据库操作时，这时候 CPU 就闲下来了，你可以利用多线程提高它的利用率。
        - 经验公式如下：线程数 = 核数 * 期望 CPU 利用率 * 总时间(CPU计算时间+等待时间) / CPU 计算时间
            - 例如 4 核 CPU 计算时间是 50% ，其它等待时间是 50%，期望 cpu 被 100% 利用，套用公式
              4 * 100% * 100% / 50% = 8
            - 例如 4 核 CPU 计算时间是 10% ，其它等待时间是 90%，期望 cpu 被 100% 利用，套用公式

4 * 100% * 100% / 10% = 40

1.4 任务调度线程池 ScheduledExecutorService

1.4.1 Timer的缺点

- 在『任务调度线程池』功能加入之前，可以使用 `java.util.Timer` 来实现定时功能，Timer 的优点在于简单易用，但由于所有任务都是由 同一个线程 来调度，因此所有任务都是 串行 执行的，同一时间只能有一个任务在执行，前一个任务的延迟或异常都将会影响到之后的任务。

```
@Slf4j(topic = "guizy.TestTimer")
public class TestTimer {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask task1 = new TimerTask() {
            @Override
            public void run() {
                log.debug("task 1");
                Sleeper.sleep(2);
            }
        };

        TimerTask task2 = new TimerTask() {
            @Override
            public void run() {
                log.debug("task 2");
            }
        };
        // 使用timer添加两个任务，希望他们都在1s后执行
        // 由于timer内只有一个线程来执行队列中的任务，所以task2必须等待task1执行完成才能执行
        timer.schedule(task1, 1000);
        timer.schedule(task2, 1000);
    }
}

08:21:17.548 guizy.TestTimer [Timer-0] - task 1
08:21:19.553 guizy.TestTimer [Timer-0] - task 2
```

1.4.2 使用 ScheduledExecutorService的schedule方法

```
ScheduledFuture<?> schedule(Runnable command, long delay,
TimeUnit unit)
```

command：要执行的任务
delay：表示延迟执行时间
unit：延迟时间的单位，例如秒、毫秒等

```
public class TestTimer {
    public static void main(String[] args) {
        ScheduledExecutorService executor =
Executors.newScheduledThreadPool(2);

        executor.schedule(() -> System.out.println("任务1，执行时
间:" + new Date()), 1000, TimeUnit.MILLISECONDS);

        executor.schedule(() -> System.out.println("任务2，执行时
间:" + new Date()), 1000, TimeUnit.MILLISECONDS);
    }
}

任务1，执行时间:Sun Jan 03 08:53:54 CST 2021
任务2，执行时间:Sun Jan 03 08:53:54 CST 2021
```

1.4.3 ScheduledExecutorService 中 scheduleAtFixedRate方法的使用

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long
initialDelay, long period, TimeUnit unit)
```

command - the task to execute
initialDelay - the time to delay first execution
period - the period between successive executions
unit - the time unit of the initialDelay and period parameters

```
public class TestTimer {
    public static void main(String[] args) {
        ScheduledExecutorService executor =
Executors.newScheduledThreadPool(1);
        log.debug("start....");
        // 延迟1s后，按1s的速率打印running
```

```
        executor.scheduleAtFixedRate(() -> log.debug("running"),
1, 1, TimeUnit.SECONDS);
    }
}

08:51:59.930 guizy.TestTimer [main] - start....
08:52:01.050 guizy.TestTimer [pool-1-thread-1] - running
08:52:02.049 guizy.TestTimer [pool-1-thread-1] - running
08:52:03.045 guizy.TestTimer [pool-1-thread-1] - running
08:52:04.046 guizy.TestTimer [pool-1-thread-1] - running
08:52:05.045 guizy.TestTimer [pool-1-thread-1] - running
08:52:06.047 guizy.TestTimer [pool-1-thread-1] - running


public class TestTimer {
    public static void main(String[] args) {
        ScheduledExecutorService executor =
Executors.newScheduledThreadPool(1);
        log.debug("start....");
        // 延迟1s后，按1s的速率打印running
        executor.scheduleAtFixedRate(() -> {
            log.debug("running");
            Sleeper.sleep(2);
        }, 1, 1, TimeUnit.SECONDS);
    }
}

// 睡眠时间 > 速率，按睡眠时间打印
08:54:58.567 guizy.TestTimer [main] - start....
08:54:59.675 guizy.TestTimer [pool-1-thread-1] - running
08:55:01.684 guizy.TestTimer [pool-1-thread-1] - running
08:55:03.685 guizy.TestTimer [pool-1-thread-1] - running
08:55:05.690 guizy.TestTimer [pool-1-thread-1] - running
```

1.4.4 ScheduledExecutorService 中scheduleWithFixedDelay 方法的使用

```
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long
initialDelay, long delay, TimeUnit unit)
```

`command` - the task to execute

`initialDelay` - the time to delay first execution

`delay` - the delay between the **termination of one execution** and the commencement of the next

`unit` - the time unit of the initialDelay and delay parameters

```
public class TestTimer {
    public static void main(String[] args) {
        ScheduledExecutorService executor =
Executors.newScheduledThreadPool(1);
        log.debug("start....");
        // 延迟1s后，按1s的速率打印running
        executor.scheduleWithFixedDelay(() -> {
            log.debug("running");
            Sleeper.sleep(2);
        }, 1, 1, TimeUnit.SECONDS);
    }
}

08:56:22.581 guizy.TestTimer [main] - start....
08:56:23.674 guizy.TestTimer [pool-1-thread-1] - running
08:56:26.679 guizy.TestTimer [pool-1-thread-1] - running
08:56:29.680 guizy.TestTimer [pool-1-thread-1] - running
08:56:32.689 guizy.TestTimer [pool-1-thread-1] - running
```

1.4.5 正确处理执行任务异常

可以发现，如果线程池中的线程执行任务时，**如果任务抛出了异常，默认是中断执行该任务而不是抛出异常或者打印异常信息。**

方法1：**主动捉异常**

```
ExecutorService pool = Executors.newFixedThreadPool(1);
pool.submit(() -> {
     try {
     log.debug("task1");
     int i = 1 / 0;
     } catch (Exception e) {
     log.error("error:", e);
     }
});
123456789
```

方法2：**使用 Future，错误信息都被封装进submit方法的返回方法中！**

```
ExecutorService pool = Executors.newFixedThreadPool(1);
Future<Boolean> f = pool.submit(() -> {
     log.debug("task1");
     int i = 1 / 0;
     return true;
});
    log.debug("result:{}", f.get());
```

二、 Fork/Join

2.1 概述

`Fork/Join` 是 JDK 1.7 加入的 新的线程池 实现，它体现的是一种 分治思想 ，适用于能够进行任务拆分的 cpu 密集型运算

所谓的任务拆分，是将一个大任务拆分为算法上相同的小任务，直至不能拆分可以直接求解。跟递归相关的一些计算，如归并排序、斐波那契数列、都可以用分治思想进行求解

Fork/Join 在分治的基础上加入了多线程，可以把每个任务的分解和合并交给不同的线程来完成，进一步提升了运算效率

Fork/Join 默认会创建与 cpu 核心数大小相同的线程池

2.2 使用

提交给 Fork/Join 线程池的任务需要继承 RecursiveTask（有返回值）或 RecursiveAction（没有返回值）

> 当调用 `fork`，会重新执行 `compute` 方法, 进行 递归 运算

```
@Slf4j(topic = "guizy.TestForkJoin2")
public class TestForkJoin2 {

    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(4);
        System.out.println(pool.invoke(new MyTask(5)));

        // new MyTask(5)  5+ new MyTask(4)  4 + new MyTask(3)  3
+ new MyTask(2)  2 + new MyTask(1)
    }
```

```
}

// 1~n 之间整数的和
@Slf4j(topic = "guizy.MyTask")
class MyTask extends RecursiveTask<Integer> {

    private int n;

    public MyTask(int n) {
        this.n = n;
    }

    @Override
    public String toString() {
        return "{" + n + '}';
    }

    @Override
    protected Integer compute() {
        // 如果 n 已经为 1，可以求得结果了
        if (n == 1) {
            log.debug("join() {}", n);
            return n;
        }

        // 将任务进行拆分(fork)
        AddTask1 t1 = new AddTask1(n - 1);
        t1.fork();
        log.debug("fork() {} + {}", n, t1);

        // 合并(join)结果
        int result = n + t1.join();
        log.debug("join() {} + {} = {}", n, t1, result);
        return result;
    }
}
```