云享 bbs.huaweicloud.com | 书库 第8期

深入理解高并发编程

Java线程池 核心技术

冰河

分布式与微服务架构专家

多年大型互联网公司研发经验

全网超 35w 技术粉丝

- 系统拆解线程池核心源码的开源小册
- 透过源码看清线程池背后的设计理念和思路
- 详细解析AQS 并发工具类

- 华为云开发者社区出品 -

全书目录

关于作者

冰河，互联网资深技术专家、MySQL 技术专家、分布式与微服务架构专家。

多年来，一直致力于分布式系统架构、微服务、分布式数据库、分布式事务与大数据技术的研究，在高并发、高可用、高可扩展性、高可维护性和大数据等领域拥有丰富的架构经验。

畅销书《深入理解高并发编程：核心原理与案例实战》《深入理解分布式事务：原理与实战》、《海量数据处理与大数据技术实战》和《MySQL 技术大全：开发、优化与运维实战》作者；“冰河技术”微信公众号作者。

公众号

分享各种编程语言、开发技术、分布式与微服务架构、分布式数据库、分布式事务、云原生、大数据与云计算技术和渗透技术。另外，还会分享各种面试题和面试技巧。内容在 **冰河技术** 微信公众号首发，建议大家关注。

公众号：冰河技术

一、程序员究竟要不要读源码?

很多人觉得读源码比较枯燥，确实，读源码是要比看那些表面教你如何使用的文章要枯燥的多，也比不上刷抖音和微博来的轻松愉快。但是，读源码是一名程序员突破自我瓶颈，获得高薪和升职加薪的一个有效途径。通过阅读优秀的开源框架的源码，我们能够领略到框架作者设计框架的思维和思路，从中学习优秀的架构设计和代码设计。这些都是在那些只告诉你如何使用的文章中所学不到的，就更别提是刷抖音和微博了。

当你只停留在业务层面的 CRUD 开发而不思进取时，工作几年之后，你会发现你几乎除了使用啥都不会！此时，你在职场其实是毫无竞争优势的。你所反反复复做的工作对于刚入行的毕业生来说，给他们 3 个月时间，他们就能熟练上手。而你，反反复复做了几年的 CRUD，没啥改变。对于企业来说，他们更加愿意雇佣那些成本低廉的新手，而不愿雇佣你！为啥？因为你给企业产出的价值未必比新入行的新手高，而你为企业带来的成本却远远高于新手！看到这里，知道为啥你工作几年后，想跳槽时，面试一个月薪几万+的职位，却只能仰望叹气了吧！！而比你工作年限少的人，却能够轻松面试比你薪资高出好几倍的职位！！不是他们运气好，而是他们比你掌握了更加深入的技能！！

当你在几年的工作时间里做的都是 CRUD 时，其实你的工作经验只有 3 个月；当你在 3 个月里，充分为自己规划好，在掌握基础业务开发的同时，抽时间为自己充电，掌握一些更加深入的技能，则你的工作经验会高于那些混迹职场几年的 CRUD 人员。

在职场还有一个现象，就是往往那些疯狂加班撸代码的都是长期的 CRUD 者，他们干的比谁都累，拿的比谁都少。往往那些掌握了深入技能的人，看似很轻松，但是他们单位时间产出的价值远远高于 CRUD 人员疯狂撸一天代码产出的价值，因为那些 CRUD 人员一天下来产出的 Bug，需要三天时间进行修正！！！

其实在职场，对于每个人非常重要的技能就是提升自己的核心竞争力，让自己变得更加有价值。

希望能够唤起你对知识的渴望。记住：工作年限并不等于工作经验！！！

二、线程与线程池

线程与多线程

1.线程

在操作系统中，线程是比进程更小的能够独立运行的基本单位。同时，它也是 CPU 调度的基本单位。线程本身基本上不拥有系统资源，只是拥有一些在运行时需要用到的系统资源，例如程序计数器，寄存器和栈等。一个进程中的所有线程可以共享进程中的所有资源。

2.多线程

多线程可以理解为在同一个程序中能够同时运行多个不同的线程来执行不同的任务，这些线程可以同时利用 CPU 的多个核心运行。多线程编程能够最大限度的利用 CPU 的资源。如果某一个线程的处理不需要占用 CPU 资源时（例如 IO 线程），可以使当前线程让出 CPU 资源来让其他线程能够获取到 CPU 资源，进而能够执行其他线程对应的任务，达到最大化利用 CPU 资源的目的。

线程的实现方式

在 Java 中，实现线程的方式大体上分为三种，通过继承 Thread 类、实现 Runnable 接口，实现 Callable 接口。简单的示例代码分别如下所示。

1.继承 Thread 类代码

```
package io.binghe.concurrent.executor.test;
/**
 * @author binghe
 * @version 1.0.0
 * @description 继承 Thread 实现线程
 */
public class ThreadTest extends Thread {
    @Override
    public void run() {
        //TODO 在此写在线程中执行的业务逻辑
    }
}
```

2.实现 Runnable 接口的代码

```
package io.binghe.concurrent.executor.test;
/**
```

```
 * @author binghe
 * @version 1.0.0
 * @description 实现 Runnable 实现线程
 */
public class RunnableTest implements Runnable {
    @Override
    public void run() {
        //TODO 在此写在线程中执行的业务逻辑
    }
}
```

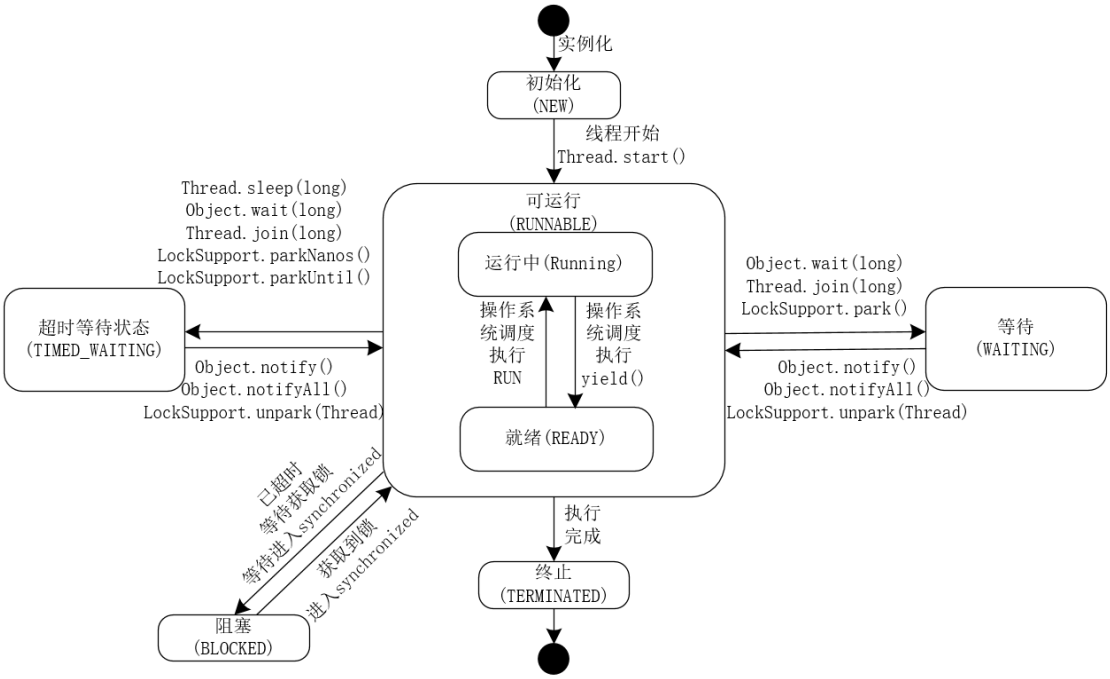
3.实现 Callable 接口的代码

```
package io.binghe.concurrent.executor.test;

import java.util.concurrent.Callable;

/**
 * @author binghe
 * @version 1.0.0
 * @description 实现 Callable 实现线程
 */
public class CallableTest implements Callable<String> {
    @Override
    public String call() throws Exception {
        //TODO 在此写在线程中执行的业务逻辑
        return null;
    }
}
```

线程的生命周期

1.生命周期

一个线程从创建，到最终的消亡，需要经历多种不同的状态，而这些不同的线程状态，由始至终也构成了线程生命周期的不同阶段。线程的生命周期可以总结为下图。



其中，几个重要的状态如下所示。

- NEW：初始状态，线程被构建，但是还没有调用 start()方法。
- RUNNABLE：可运行状态，可运行状态可以包括：运行中状态和就绪状态。
- BLOCKED：阻塞状态，处于这个状态的线程需要等待其他线程释放锁或者等待进入 synchronized。
- WAITING：表示等待状态，处于该状态的线程需要等待其他线程对其进行通知或中断等操作，进而进入下一个状态。
- TIME_WAITING：超时等待状态。可以在一定的时间自行返回。
- TERMINATED：终止状态，当前线程执行完毕。

2.代码示例

为了更好的理解线程的生命周期，以及生命周期中的各个状态，接下来使用代码示例来输出线程的每个状态信息。

- WaitingTime

创建 WaitingTime 类，在 while(true)循环中调用 TimeUnit.SECONDS.sleep(long)方法来验证线程的 TIMED_WARTING 状态，代码如下所示。

```
package io.binghe.concurrent.executor.state;
import java.util.concurrent.TimeUnit;

/**
 * @author binghe
 * @version 1.0.0
 * @description 线程不断休眠
 */
public class WaitingTime implements Runnable{
    @Override
    public void run() {
        while (true){
            waitSecond(200);
        }
    }
    //线程等待多少秒
    public static final void waitSecond(long seconds){
        try {
            TimeUnit.SECONDS.sleep(seconds);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- WaitingState

创建 WaitingState 类，此线程会在一个 while(true)循环中，获取当前类 Class 对象的 synchronized 锁，也就是说，这个类无论创建多少个实例，synchronized 锁都是同一个，并且线程会处于等待状态。接下来，在 synchronized 中使用当前类的 Class 对象的 wait()方法，来验证线程的 WAITING 状态，代码如下所示。

```
package io.binghe.concurrent.executor.state;
/**
 * @author binghe
```

```
 * @version 1.0.0
 * @description 线程在 Warting 上等待
 */
public class WaitingState implements Runnable {
    @Override
    public void run() {
        while (true){
            synchronized (WaitingState.class){
                try {
                    WaitingState.class.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

- BlockedThread

BlockedThread 主要是在 synchronized 代码块中的 while(true)循环中调用 TimeUnit.SECONDS.sleep(long)方法来验证线程的 BLOCKED 状态。当启动两个 BlockedThread 线程时，首先启动的线程会处于 TIMED_WAITING 状态，后启动的线程会处于 BLOCKED 状态。代码如下所示。

```
package io.binghe.concurrent.executor.state;
/**
 * @author binghe
 * @version 1.0.0
 * @description 加锁后不再释放锁
 */
public class BlockedThread implements Runnable {
    @Override
    public void run() {
        synchronized (BlockedThread.class){
            while (true){
                WaitingTime.waitSecond(100);
            }
        }
```

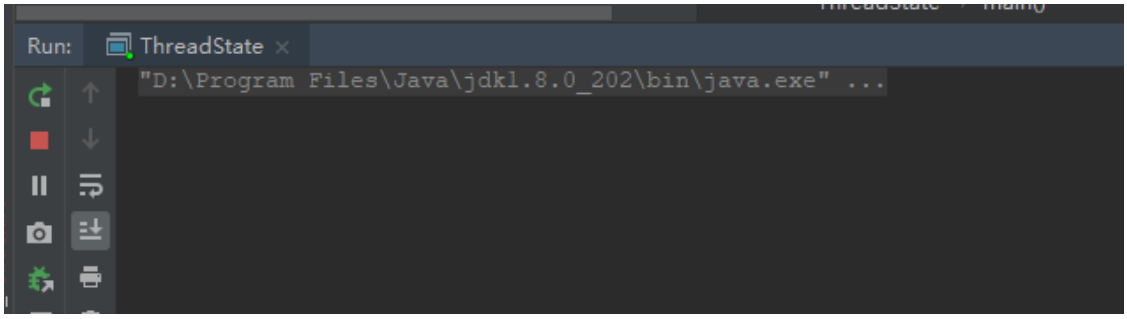
```
    }
}
```

- ThreadState

启动各个线程，验证各个线程输出的状态，代码如下所示。

```
package io.binghe.concurrent.executor.state;

/**
 * @author binghe
 * @version 1.0.0
 * @description 线程的各种状态，测试线程的生命周期
 */
public class ThreadState {

    public static void main(String[] args){
        new Thread(new WaitingTime(), "WaitingTimeThread").start();
        new Thread(new WaitingState(), "WaitingStateThread").start();

        //BlockedThread-01 线程会抢到锁，BlockedThread-02 线程会阻塞
        new Thread(new BlockedThread(), "BlockedThread-01").start();
        new Thread(new BlockedThread(), "BlockedThread-02").start();
    }
}
```

运行 ThreadState 类，如下所示。



可以看到，未输出任何结果信息。可以在命令行输入“jps”命令来查看运行的 Java 进程。

```
c:\>jps
21584 Jps
17828 KotlinCompileDaemon
12284 Launcher
24572
28492 ThreadState
```

可以看到 ThreadSate 进程的进程号为 28492，接下来，输入“jstack 28492”来查看 ThreadSate 进程栈的信息，如下所示。

```
c:\>jstack 28492
2020-02-15 00:27:08
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.202-b08 mixed mode):

"DestroyJavaVM" #16 prio=5 os_prio=0 tid=0x000000001ca05000 nid=0x1a4 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"BlockedThread-02" #15 prio=5 os_prio=0 tid=0x000000001ca04800 nid=0x6eb0 waiting for monitor entry [0x000000001da4f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at io.binghe.concurrent.executor.state.BlockedThread.run(BlockedThread.java:28)
        - waiting to lock <0x0000000780a7e4e8> (a java.lang.Class for io.binghe.concurrent.executor.state.BlockedThread)
        at java.lang.Thread.run(Thread.java:748)

"BlockedThread-01" #14 prio=5 os_prio=0 tid=0x000000001ca01800 nid=0x6e28 waiting on condition [0x000000001d94f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at java.lang.Thread.sleep(Thread.java:340)
        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
        at io.binghe.concurrent.executor.state.WaitingTime.waitSecond(WaitingTime.java:36)
        at io.binghe.concurrent.executor.state.BlockedThread.run(BlockedThread.java:28)
        - locked <0x0000000780a7e4e8> (a java.lang.Class for io.binghe.concurre
```

```
nt.executor.state.BlockedThread)
        at java.lang.Thread.run(Thread.java:748)

"WaitingStateThread" #13 prio=5 os_prio=0 tid=0x000000001ca06000 nid=0x6f
e4 in Object.wait() [0x000000001d84f000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0000000780a7b488> (a java.lang.Class for io.binghe.conc
urrent.executor.state.WaitingState)
        at java.lang.Object.wait(Object.java:502)
        at io.binghe.concurrent.executor.state.WaitingState.run(WaitingState.java:
29)
        - locked <0x0000000780a7b488> (a java.lang.Class for io.binghe.concurre
nt.executor.state.WaitingState)
        at java.lang.Thread.run(Thread.java:748)

"WaitingTimeThread" #12 prio=5 os_prio=0 tid=0x000000001c9f8800 nid=0x385
8 waiting on condition [0x000000001d74f000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(Native Method)
        at java.lang.Thread.sleep(Thread.java:340)
        at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
        at io.binghe.concurrent.executor.state.WaitingTime.waitSecond(WaitingTi
me.java:36)
        at io.binghe.concurrent.executor.state.WaitingTime.run(WaitingTime.java:2
9)
        at java.lang.Thread.run(Thread.java:748)

"Service Thread" #11 daemon prio=9 os_prio=0 tid=0x000000001c935000 nid=0
x6864 runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C1 CompilerThread3" #10 daemon prio=9 os_prio=2 tid=0x000000001c88c800
nid=0x6a28 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread2" #9 daemon prio=9 os_prio=2 tid=0x000000001c880000
nid=0x6498 waiting on condition [0x0000000000000000]
```

```
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #8 daemon prio=9 os_prio=2 tid=0x000000001c87c000 n
id=0x693c waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #7 daemon prio=9 os_prio=2 tid=0x000000001c87b800
nid=0x5d00 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Monitor Ctrl-Break" #6 daemon prio=5 os_prio=0 tid=0x000000001c862000 nid
=0x6034 runnable [0x000000001d04e000]
   java.lang.Thread.State: RUNNABLE
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
        at java.net.SocketInputStream.read(SocketInputStream.java:171)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
        at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
        at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
        - locked <0x0000000780b2fd88> (a java.io.InputStreamReader)
        at java.io.InputStreamReader.read(InputStreamReader.java:184)
        at java.io.BufferedReader.fill(BufferedReader.java:161)
        at java.io.BufferedReader.readLine(BufferedReader.java:324)
        - locked <0x0000000780b2fd88> (a java.io.InputStreamReader)
        at java.io.BufferedReader.readLine(BufferedReader.java:389)
        at com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:6
4)

"Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x000000001c788800 nid=0x
6794 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x000000001c7e3800 nid
=0x3354 runnable [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x000000001c771000 nid=0x6968 in
```

```
 Object.wait() [0x000000001cd4f000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0000000780908ed0> (a java.lang.ref.ReferenceQueue$Lo
ck)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:144)
        - locked <0x0000000780908ed0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:165)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:216)

"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000000001c770800 ni
d=0x6590 in Object.wait() [0x000000001cc4f000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x0000000780906bf8> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:502)
        at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
        - locked <0x0000000780906bf8> (a java.lang.ref.Reference$Lock)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"VM Thread" os_prio=2 tid=0x000000001a979800 nid=0x5c2c runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x00000000033b9000 nid=0x4d
c0 runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x00000000033ba800 nid=0x66
90 runnable

"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x00000000033bc000 nid=0x30
b0 runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x00000000033be800 nid=0x6f
68 runnable

"GC task thread#4 (ParallelGC)" os_prio=0 tid=0x00000000033c1000 nid=0x64
78 runnable

"GC task thread#5 (ParallelGC)" os_prio=0 tid=0x00000000033c2000 nid=0x4f
```

e4 runnable

"GC task thread#6 (ParallelGC)" os_prio=0 tid=0x00000000033c5000 nid=0x584 runnable

"GC task thread#7 (ParallelGC)" os_prio=0 tid=0x00000000033c6800 nid=0x6988 runnable

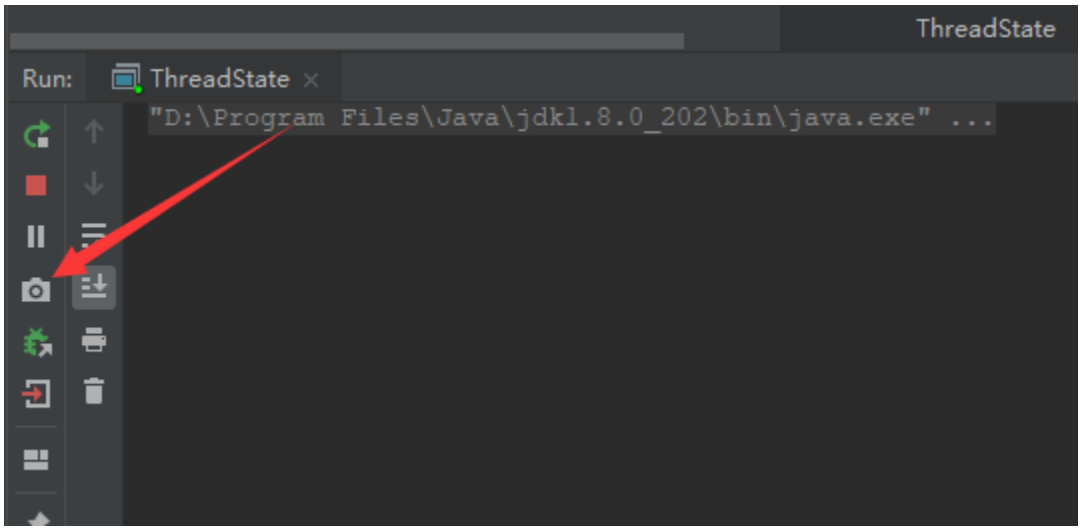
"VM Periodic Task Thread" os_prio=2 tid=0x000000001c959800 nid=0x645c waiting on condition

JNI global references: 12

由以上输出的信息可以看出：名称为 WaitingTimeThread 的线程处于 TIMED*WAITING 状态；名称为 WaitingStateThread 的线程处于 WAITING 状态；名称为 BlockedThread-01 的线程处于 TIMED*WAITING 状态；名称为 BlockedThread-02 的线程处于 BLOCKED 状态。

注意：使用 jps 结合 jstack 命令可以分析线上生产环境的 Java 进程的异常信息。

也可以直接点击 IDEA 下图所示的图表直接打印出线程的堆栈信息。



输出的结果信息与使用“jstack 进程号”命令输出的信息基本一致。

三、线程的执行顺序

线程的执行顺序是不确定的

调用 Thread 的 start()方法启动线程时，线程的执行顺序是不确定的。也就是说，在同一个方法中，连续创建多个线程后，调用线程的 start()方法的顺序并不能决定线程的执行顺序。

例如，这里，看一个简单的示例程序，如下所示。

```
package io.binghe.concurrent.lab03;

/**
 * @author binghe
 * @version 1.0.0
 * @description 线程的顺序，直接调用 Thread.start()方法执行不能确保线程的执行顺序
 */
public class ThreadSort01 {
    public static void main(String[] args){
        Thread thread1 = new Thread(() -> {
            System.out.println("thread1");
        });
        Thread thread2 = new Thread(() -> {
            System.out.println("thread2");
        });
        Thread thread3 = new Thread(() -> {
            System.out.println("thread3");
        });

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

在 ThreadSort01 类中分别创建了三个不同的线程，thread1、thread2 和 thread3，接下来，在程序中按照顺序分别调用 thread1.start()、thread2.start()和 thread3.start()方法来分别启动三个不同的线程。

那么，问题来了，线程的执行顺序是否按照 thread1、thread2 和 thread3 的顺序执行呢？运行 ThreadSort01 的 main 方法，结果如下所示。

```
thread1
thread2
thread3
```

再次运行时，结果如下所示。

```
thread1
thread3
thread2
```

第三次运行时，结果如下所示。

```
thread2
thread3
thread1
```

注意：每个人运行的情况可能都不一样。

可以看到，每次运行程序时，线程的执行顺序可能不同。线程的启动顺序并不能决定线程的执行顺序。

如何确保线程的执行顺序

1.确保线程执行顺序的简单示例

在实际业务场景中，有时，后启动的线程可能需要依赖先启动的线程执行完成才能正确的执行线程中的业务逻辑。此时，就需要确保线程的执行顺序。那么如何确保线程的执行顺序呢？

可以使用 Thread 类中的 join()方法来确保线程的执行顺序。例如，下面的测试代码。

```
package io.binghe.concurrent.lab03;
/**
 * @author binghe
 * @version 1.0.0
 * @description 线程的顺序，Thread.join()方法能够确保线程的执行顺序
 */
public class ThreadSort02 {
    public static void main(String[] args) throws InterruptedException {
```

```
        Thread thread1 = new Thread(() -> {
            System.out.println("thread1");
        });
        Thread thread2 = new Thread(() -> {
            System.out.println("thread2");
        });
        Thread thread3 = new Thread(() -> {
            System.out.println("thread3");
        });

        thread1.start();

        //实际上让主线程等待子线程执行完成
        thread1.join();

        thread2.start();
        thread2.join();

        thread3.start();
        thread3.join();
    }
}
```

可以看到，ThreadSort02 类比 ThreadSort01 类，在每个线程的启动方法下面添加了调用线程的 join()方法。此时，运行 ThreadSort02 类，结果如下所示。

```
thread1
thread2
thread3
```

再次运行时，结果如下所示。

```
thread1
thread2
thread3
```

第三次运行时，结果如下所示。

thread1
thread2
thread3

可以看到，每次运行的结果都是相同的，所以，使用 Thread 的 join()方法能够保证线程的先后执行顺序。

2.join 方法如何确保线程的执行顺序

既然 Thread 类的 join()方法能够确保线程的执行顺序，我们就一起来看看 Thread 类的 join()方法到底是个什么鬼。

进入 Thread 的 join()方法，如下所示。

```
public final void join() throws InterruptedException {
        join(0);
}
```

可以看到 join()方法调用同类中的一个有参 join()方法，并传递参数 0。继续跟进代码，如下所示。

```
public final synchronized void join(long millis)
throws InterruptedException {
        long base = System.currentTimeMillis();
        long now = 0;

        if (millis < 0) {
                throw new IllegalArgumentException("timeout value is negative");
        }

        if (millis == 0) {
                while (isAlive()) {
                        wait(0);
                }
        } else {
                while (isAlive()) {
                        long delay = millis - now;
                        if (delay <= 0) {
                                break;
                        }
                        wait(delay);
```

```
                    now = System.currentTimeMillis() - base;
                }
            }
        }
```

可以看到，有一个 long 类型参数的 join()方法使用了 synchroinzed 修饰，说明这个方法同一时刻只能被一个实例或者方法调用。由于，传递的参数为 0，所以，程序会进入如下代码逻辑。

```
if (millis == 0) {
        while (isAlive()) {
                wait(0);
        }
}
```

首先，在代码中以 while 循环的方式来判断当前线程是否已经启动处于活跃状态，如果已经启动处于活跃状态，则调用同类中的 wait()方法，并传递参数 0。继续跟进 wait()方法，如下所示。

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

可以看到，wait()方法是一个本地方法，通过 JNI 的方式调用 JDK 底层的方法来使线程等待执行完成。

需要注意的是，调用线程的 wait()方法时，会使主线程处于等待状态，等待子线程执行完成后再次向下执行。也就是说，在 ThreadSort02 类的 main()方法中，调用子线程的 join()方法，会阻塞 main()方法的执行，当子线程执行完成后，main()方法会继续向下执行，启动第二个子线程，并执行子线程的业务逻辑，以此类推。

四、Java 中的 Callable 和 Future

在 Java 的多线程编程中，除了 Thread 类和 Runnable 接口外，不得不说的就是 Callable 接口 Future 接口了。使用继承 Thread 类或者实现 Runnable 接口的线程，无法返回最终的执行结果数据，只能等待线程执行完成。此时，如果想要获取线程执行后的返回结果，那么，Callable 和 Future 就派上用场了。

Callable 接口

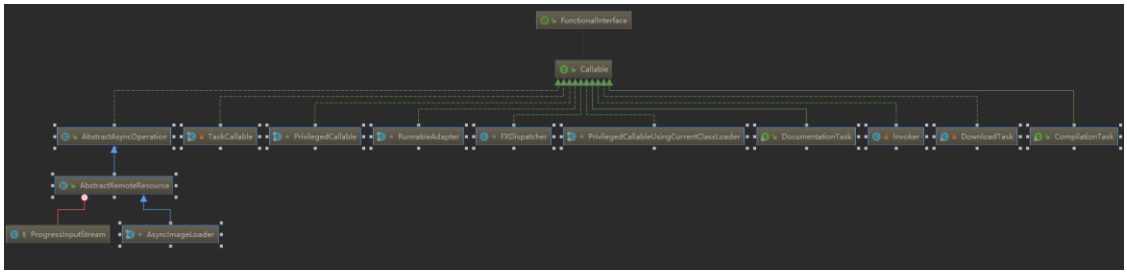
1.Callable 接口介绍

Callable 接口是 JDK1.5 新增的泛型接口，在 JDK1.8 中，被声明为函数式接口，如下所示。

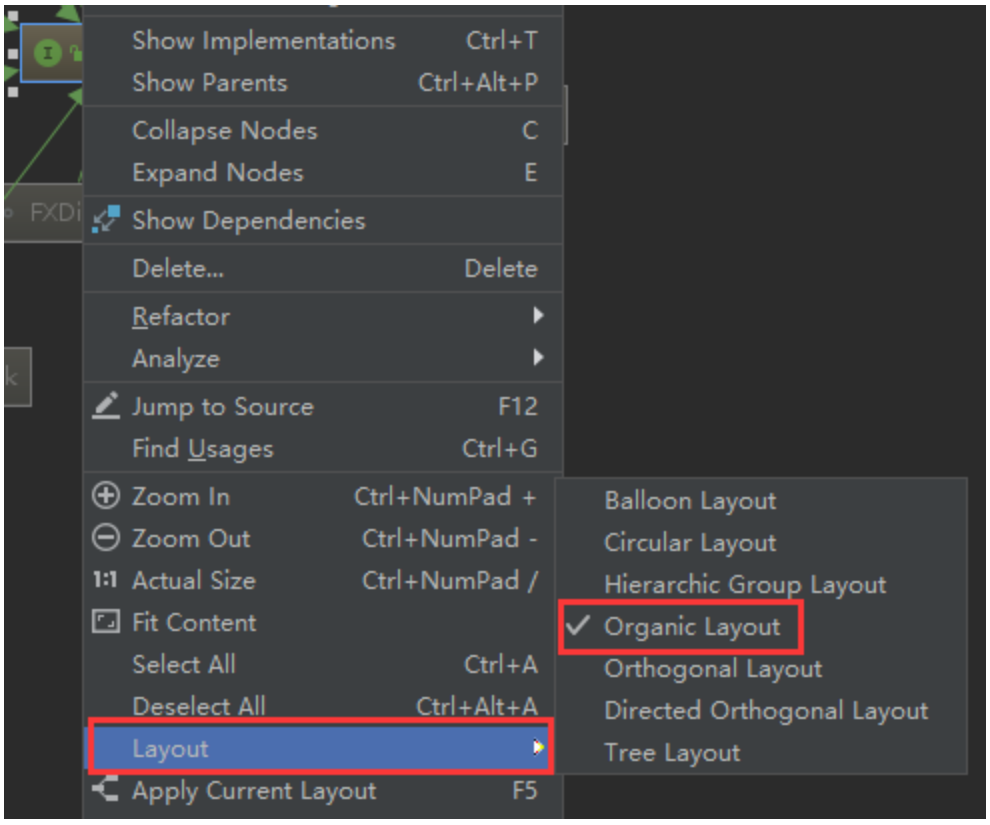
```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

在 JDK 1.8 中只声明有一个方法的接口为函数式接口，函数式接口可以使用@FunctionalInterface 注解修饰，也可以不使用@FunctionalInterface 注解修饰。只要一个接口中只包含有一个方法，那么，这个接口就是函数式接口。

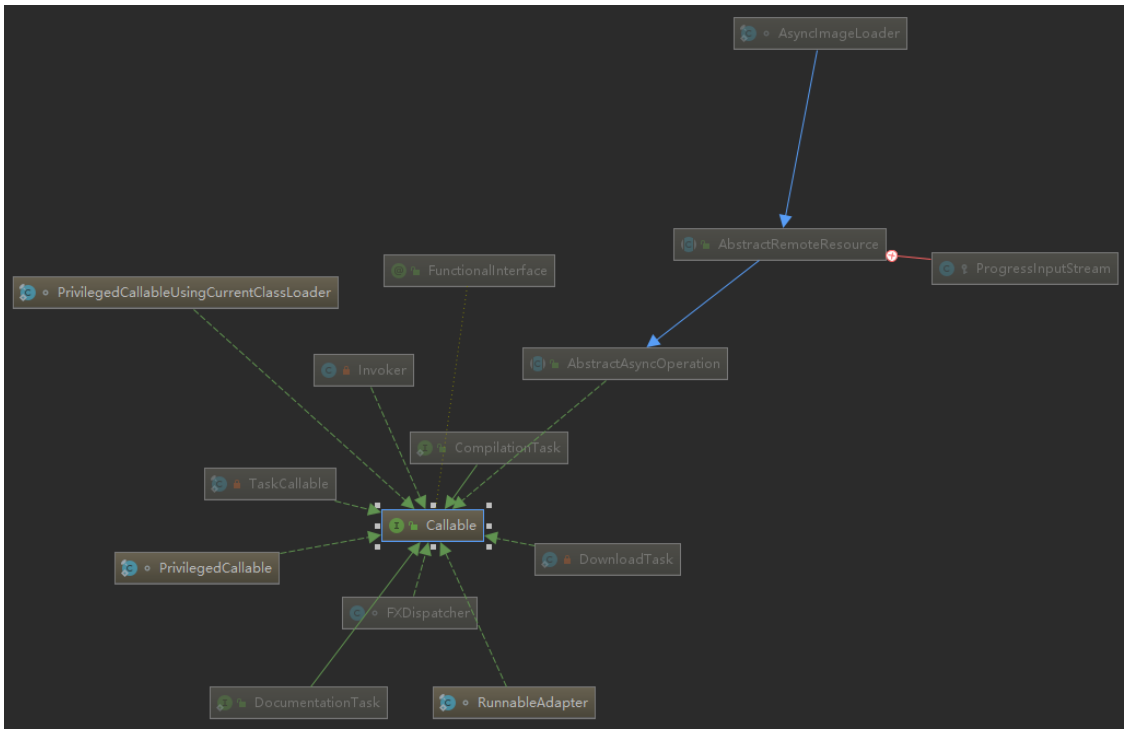
在 JDK 中，实现 Callable 接口的子类如下图所示。



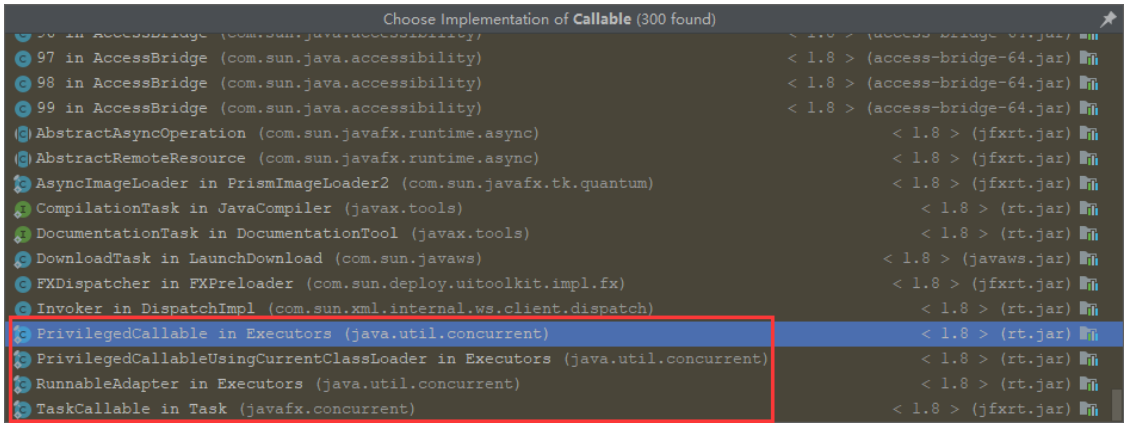
默认的子类层级关系图看不清，这里，可以通过 IDEA 右键 Callable 接口，选择“Layout”来指定 Callable 接口的实现类图的不同结构，如下所示。



这里，可以选择“Organic Layout”选项，选择后的 Callable 接口的子类的结构如下图所示。



在实现 Callable 接口的子类中，有几个比较重要的类，如下图所示。



分别是：Executors 类中的静态内部类：PrivilegedCallable、PrivilegedCallableUsingCurrentClassLoader、RunnableAdapter 和 Task 类下的 TaskCallable。

2.实现 Callable 接口的重要类分析

接下来，分析的类主要有：PrivilegedCallable、PrivilegedCallableUsingCurrentClassLoader、RunnableAdapter 和 Task 类下的 TaskCallable。虽然这些类在实际工作中很少被直接用到，但是作为一名合格的开发工程师，设置是秃顶的资深专家来说，了解并掌握这些类的实现有助你进一步理解 Callable 接口，并提高专业技能（头发再掉一批，哇哈哈哈。。。）。

- PrivilegedCallable

PrivilegedCallable 类是 Callable 接口的一个特殊实现类，它表明 Callable 对象有某种特权来访问系统的某种资源，PrivilegedCallable 类的源代码如下所示。

```
/**
 *A callable that runs under established access control settings
 */
static final class PrivilegedCallable<T> implements Callable<T> {
        private final Callable<T> task;
        private final AccessControlContext acc;

        PrivilegedCallable(Callable<T> task) {
                this.task = task;
                this.acc = AccessController.getContext();
        }

        public T call() throws Exception {
                try {
                        return AccessController.doPrivileged(
                                new PrivilegedExceptionAction<T>() {
                                        public T run() throws Exception {
                                                return task.call();
                                        }
                                }, acc);
                } catch (PrivilegedActionException e) {
                        throw e.getException();
                }
        }
}
```

从 PrivilegedCallable 类的源代码来看，可以将 PrivilegedCallable 看成是对 Callable 接口的封装，并且这个类也继承了 Callable 接口。

在 PrivilegedCallable 类中有两个成员变量，分别是 Callable 接口的实例对象和 AccessControlContext 类的实例对象，如下所示。

```
private final Callable<T> task;
private final AccessControlContext acc;
```

其中，AccessControlContext 类可以理解为一个具有系统资源访问决策的上下文类，通过这个类可以访问系统的特定资源。通过类的构造方法可以看出，在实例化 AccessControlContext 类的对象时，只需要传递 Callable 接口子类的对象即可，如下所示。

```
PrivilegedCallable(Callable<T> task) {
        this.task = task;
        this.acc = AccessController.getContext();
}
```

AccessControlContext 类的对象是通过 AccessController 类的 getContext()方法获取的，这里，查看 AccessController 类的 getContext()方法，如下所示。

```
public static AccessControlContext getContext(){
        AccessControlContext acc = getStackAccessControlContext();
        if (acc == null) {
                return new AccessControlContext(null, true);
        } else {
                return acc.optimize();
        }
}
```

通过 AccessController 的 getContext()方法可以看出，首先通过 getStackAccessControlContext()方法来获取 AccessControlContext 对象实例。如果获取的 AccessControlContext 对象实例为空，则通过调用 AccessControlContext 类的构造方法实例化，否则，调用 AccessControlContext 对象实例的 optimize()方法返回 AccessControlContext 对象实例。

这里，我们先看下 getStackAccessControlContext()方法是个什么鬼。

```
private static native AccessControlContext getStackAccessControlContext();
```

原来是个本地方法，方法的字面意思就是获取能够访问系统栈的决策上下文对象。

接下来，我们回到 PrivilegedCallable 类的 call()方法，如下所示。

```
public T call() throws Exception {
        try {
                return AccessController.doPrivileged(
                        new PrivilegedExceptionAction<T>() {
                                public T run() throws Exception {
                                        return task.call();
                                }
                        }, acc);
        } catch (PrivilegedActionException e) {
                throw e.getException();
        }
}
```

通过调用 AccessController.doPrivileged()方法，传递 PrivilegedExceptionAction。接口对象和 AccessControlContext 对象，并最终返回泛型的实例对象。

首先，看下 AccessController.doPrivileged()方法，如下所示。

```
@CallerSensitive
public static native <T> T
   doPrivileged(PrivilegedExceptionAction<T> action,
                AccessControlContext context)
   throws PrivilegedActionException;
```

可以看到，又是一个本地方法。也就是说，最终的执行情况是将 PrivilegedExceptionAction 接口对象和 AccessControlContext 对象实例传递给这个本地方法执行。并且在 PrivilegedExceptionAction 接口对象的 run()方法中调用 Callable 接口的 call()方法来执行最终的业务逻辑，并且返回泛型对象。

- PrivilegedCallableUsingCurrentClassLoader

此类表示为在已经建立的特定访问控制和当前的类加载器下运行的 Callable 类，源代码如下所示。

```
/**
 *A callable that runs under established access control settings and
 *current ClassLoader
```

```
 */
static final class PrivilegedCallableUsingCurrentClassLoader<T> implements C
allable<T> {
        private final Callable<T> task;
        private final AccessControlContext acc;
        private final ClassLoader ccl;

        PrivilegedCallableUsingCurrentClassLoader(Callable<T> task) {
                SecurityManager sm = System.getSecurityManager();
                if (sm != null) {
                        sm.checkPermission(SecurityConstants.GET_CLASSLOA
DER_PERMISSION);
                        sm.checkPermission(new RuntimePermission("setContext
ClassLoader"));
                }
                this.task = task;
                this.acc = AccessController.getContext();
                this.ccl = Thread.currentThread().getContextClassLoader();
        }

        public T call() throws Exception {
                try {
                        return AccessController.doPrivileged(
                                new PrivilegedExceptionAction<T>() {
                                        public T run() throws Exception {
                                                Thread t = Thread.currentThread();
                                                ClassLoader cl = t.getContextClassLoa
der();

                                                if (ccl == cl) {
                                                        return task.call();
                                                } else {
                                                        t.setContextClassLoader(ccl);
                                                        try {
                                                                return task.call();
                                                        } finally {
                                                                t.setContextClassLoader
(cl);

                                                        }
                                                }
```

```
                    }
                }, acc);
            } catch (PrivilegedActionException e) {
                throw e.getException();
            }
        }
}
```

这个类理解起来比较简单，首先，在类中定义了三个成员变量，如下所示。

```
private final Callable<T> task;
private final AccessControlContext acc;
private final ClassLoader ccl;
```

接下来，通过构造方法注入 Callable 对象，在构造方法中，首先获取系统安全管理器对象实例，通过系统安全管理器对象实例检查是否具有获取 ClassLoader 和设置 ContextClassLoader 的权限。并在构造方法中为三个成员变量赋值，如下所示。

```
PrivilegedCallableUsingCurrentClassLoader(Callable<T> task) {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPermission(SecurityConstants.GET_CLASSLOADER_PERMISSION);
            sm.checkPermission(new RuntimePermission("setContextClassLoader"));
        }
        this.task = task;
        this.acc = AccessController.getContext();
        this.ccl = Thread.currentThread().getContextClassLoader();
}
```

接下来，通过调用 call()方法来执行具体的业务逻辑，如下所示。

```
public T call() throws Exception {
        try {
            return AccessController.doPrivileged(
                new PrivilegedExceptionAction<T>() {
                    public T run() throws Exception {
                        Thread t = Thread.currentThread();
                        ClassLoader cl = t.getContextClassLoader();
```

```
                                    if (ccl == cl) {
                                        return task.call();
                                    } else {
                                        t.setContextClassLoader(ccl);
                                        try {
                                            return task.call();
                                        } finally {
                                            t.setContextClassLoader(cl);
                                        }
                                    }
                                }
                            }, acc);
        } catch (PrivilegedActionException e) {
            throw e.getException();
        }
}
```

在 call()方法中同样是通过调用 AccessController 类的本地方法 doPrivileged，传递 PrivilegedExceptionAction 接口的实例对象和 AccessControlContext 类的对象实例。

具体执行逻辑为：在 PrivilegedExceptionAction 对象的 run()方法中获取当前线程的 ContextClassLoader 对象，如果在构造方法中获取的 ClassLoader 对象与此处的 ContextClassLoader 对象是同一个对象（不止对象实例相同，而且内存地址也相同），则直接调用 Callable 对象的 call()方法返回结果。否则，将 PrivilegedExceptionAction 对象的 run()方法中的当前线程的 ContextClassLoader 设置为在构造方法中获取的类加载器对象，接下来，再调用 Callable 对象的 call()方法返回结果。最终将当前线程的 ContextClassLoader 重置为之前的 ContextClassLoader。

- RunnableAdapter

RunnableAdapter 类比较简单，给定运行的任务和结果，运行给定的任务并返回给定的结果，源代码如下所示。

```
/**
 *A callable that runs given task and returns given result
 */
static final class RunnableAdapter<T> implements Callable<T> {
        final Runnable task;
```

```
        final T result;
        RunnableAdapter(Runnable task, T result) {
                this.task = task;
                this.result = result;
        }
        public T call() {
                task.run();
                return result;
        }
}
```

- TaskCallable

TaskCallable 类是 javafx.concurrent.Task 类的静态内部类，TaskCallable 类主要是实现了 Callable 接口并且被定义为 FutureTask 的类，并且在这个类中允许我们拦截 call()方法来更新 task 任务的状态。源代码如下所示。

```
private static final class TaskCallable<V> implements Callable<V> {

        private Task<V> task;
        private TaskCallable() { }

        @Override
        public V call() throws Exception {
                task.started = true;
                task.runLater(() -> {
                        task.setState(State.SCHEDULED);
                        task.setState(State.RUNNING);
                });
                try {
                        final V result = task.call();
                        if (!task.isCancelled()) {
                                task.runLater(() -> {
                                        task.updateValue(result);
                                        task.setState(State.SUCCEEDED);
                                });
                                return result;
                        } else {
                                return null;
                        }
```

```
                } catch (final Throwable th) {
                        task.runLater(() -> {
                                task._setException(th);
                                task.setState(State.FAILED);
                        });
                        if (th instanceof Exception) {
                                throw (Exception) th;
                        } else {
                                throw new Exception(th);
                        }
                }
        }
}
```

从 TaskCallable 类的源代码可以看出，只定义了一个 Task 类型的成员变量。下面主要分析 TaskCallable 类的 call()方法。

当程序的执行进入到 call()方法时，首先将 task 对象的 started 属性设置为 true，表示任务已经开始，并且将任务的状态依次设置为 State.SCHEDULED 和 State.RUNNING，依次触发任务的调度事件和运行事件。如下所示。

```
task.started = true;
task.runLater(() -> {
        task.setState(State.SCHEDULED);
        task.setState(State.RUNNING);
});
```

接下来，在 try 代码块中执行 Task 对象的 call()方法，返回泛型对象。如果任务没有被取消，则更新任务的缓存，将调用 call()方法返回的泛型对象绑定到 Task 对象中的 ObjectProperty 对象中，其中，ObjectProperty 在 Task 类中的定义如下。

```
private final ObjectProperty<V> value = new SimpleObjectProperty<>(this, "valu
e");
```

接下来，将任务的状态设置为成功状态。如下所示。

```
try {
        final V result = task.call();
        if (!task.isCancelled()) {
                task.runLater(() -> {
```

```
                task.updateValue(result);
                task.setState(State.SUCCEEDED);
            });
            return result;
        } else {
            return null;
        }
}
```

如果程序抛出了异常或者错误，会进入 catch()代码块，设置 Task 对象的 Exception 信息并将状态设置为 State.FAILED，也就是将任务标记为失败。接下来，判断异常或错误的类型，如果是 Exception 类型的异常，则直接强转为 Exception 类型的异常并抛出。否则，将异常或者错误封装为 Exception 对象并抛出，如下所示。

```
catch (final Throwable th) {
        task.runLater(() -> {
            task._setException(th);
            task.setState(State.FAILED);
        });
        if (th instanceof Exception) {
            throw (Exception) th;
        } else {
            throw new Exception(th);
        }
}
```

两种异步模型与深度解析 Future 接口

两种异步模型

在 Java 的并发编程中，大体上会分为两种异步编程模型，一类是直接以异步的形式来并行运行其他的任务，不需要返回任务的结果数据。一类是以异步的形式运行其他任务，需要返回结果。

1.无返回结果的异步模型

无返回结果的异步任务，可以直接将任务丢进线程或线程池中运行，此时，无法直接获得任务的执行结果数据，一种方式是可以使用回调方法来获取任务的运行结果。

具体的方案是：定义一个回调接口，并在接口中定义接收任务结果数据的方法，具体逻辑在回调接口的实现类中完成。将回调接口与任务参数一同放进线程或线程池中运行，任务运行后调用接口方法，执行回调接口实现类中的逻辑来处理结果数据。这里，给出一个简单的示例供参考。

- 定义回调接口

```
package io.binghe.concurrent.lab04;

/**
 * @author binghe
 * @version 1.0.0
 * @description 定义回调接口
 */
public interface TaskCallable<T> {
    T callable(T t);
}
```

便于接口的通用型，这里为回调接口定义了泛型。

- 定义任务结果数据的封装类

```
package io.binghe.concurrent.lab04;

import java.io.Serializable;

/**
 * @author binghe
 * @version 1.0.0
 * @description 任务执行结果
 */
public class TaskResult implements Serializable {
    private static final long serialVersionUID = 8678277072402730062L;
    /**
     *任务状态
     */
    private Integer taskStatus;

    /**
     *任务消息
```

```
     */
    private String taskMessage;

    /**
     *任务结果数据
     */
    private String taskResult;

        //省略 getter 和 setter 方法
        @Override
    public String toString() {
        return "TaskResult{" +
                "taskStatus=" + taskStatus +
                ", taskMessage='" + taskMessage + '\'' +
                ", taskResult='" + taskResult + '\'' +
                '}';
    }
}
```

- 创建回调接口的实现类

回调接口的实现类主要用来对任务的返回结果进行相应的业务处理，这里，为了方便演示，只是将结果数据返回。大家需要根据具体的业务场景来做相应的分析和处理。

```
package io.binghe.concurrent.lab04;

/**
 * @author binghe
 * @version 1.0.0
 * @description 回调函数的实现类
 */
public class TaskHandler implements TaskCallable<TaskResult> {
    @Override
public TaskResult callable(TaskResult taskResult) {
//TODO 拿到结果数据后进一步处理
    System.out.println(taskResult.toString());
        return taskResult;
```

```
    }
}
```

- 创建任务的执行类

任务的执行类是具体执行任务的类，实现 Runnable 接口，在此类中定义一个回调接口类型的成员变量和一个 String 类型的任务参数（模拟任务的参数），并在构造方法中注入回调接口和任务参数。在 run 方法中执行任务，任务完成后将任务的结果数据封装成 TaskResult 对象，调用回调接口的方法将 TaskResult 对象传递到回调方法中。

```
package io.binghe.concurrent.lab04;
/**
 * @author binghe
 * @version 1.0.0
 * @description 任务执行类
 */
public class TaskExecutor implements Runnable{
    private TaskCallable<TaskResult> taskCallable;
    private String taskParameter;

    public TaskExecutor(TaskCallable<TaskResult> taskCallable, String taskPar
ameter){
        this.taskCallable = taskCallable;
        this.taskParameter = taskParameter;
    }

    @Override
    public void run() {
        //TODO 一系列业务逻辑,将结果数据封装成 TaskResult 对象并返回
        TaskResult result = new TaskResult();
        result.setTaskStatus(1);
        result.setTaskMessage(this.taskParameter);
        result.setTaskResult("异步回调成功");
        taskCallable.callable(result);
    }
}
```

到这里，整个大的框架算是完成了，接下来，就是测试看能否获取到异步任务的结果了。

- 异步任务测试类

```
package io.binghe.concurrent.lab04;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试回调
 */
public class TaskCallableTest {
    public static void main(String[] args){
        TaskCallable<TaskResult> taskCallable = new TaskHandler();
        TaskExecutor taskExecutor = new TaskExecutor(taskCallable, "测试回调任务");
        new Thread(taskExecutor).start();
    }
}
```

在测试类中，使用 Thread 类创建一个新的线程，并启动线程运行任务。运行程序最终的接口数据如下所示。

```
TaskResult{taskStatus=1, taskMessage='测试回调任务', taskResult='异步回调成功'}
```

大家可以细细品味下这种获取异步结果的方式。这里，只是简单的使用了 Thread 类来创建并启动线程，也可以使用线程池的方式实现。大家可自行实现以线程池的方式通过回调接口获取异步结果。

2.有返回结果的异步模型

尽管使用回调接口能够获取异步任务的结果，但是这种方式使用起来略显复杂。在 JDK 中提供了可以直接返回异步结果的处理方案。最常用的就是使用 Future 接口或者其实现类 FutureTask 来接收任务的返回结果。

- 使用 Future 接口获取异步结果

使用 Future 接口往往配合线程池来获取异步执行结果，如下所示。

```
package io.binghe.concurrent.lab04;

import java.util.concurrent.*;
```

```
/**
 * @author binghe
 * @version 1.0.0
 * @description 测试 Future 获取异步结果
 */
public class FutureTest {

    public static void main(String[] args) throws ExecutionException, InterruptedE
xception {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future<String> future = executorService.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                return "测试 Future 获取异步结果";
            }
        });
        System.out.println(future.get());
        executorService.shutdown();
    }
}
```

运行结果如下所示。

```
测试 Future 获取异步结果
```

- 使用 FutureTask 类获取异步结果

FutureTask 类既可以结合 Thread 类使用也可以结合线程池使用，接下来，就看下这两种使用方式。

结合 Thread 类的使用示例如下所示。

```
package io.binghe.concurrent.lab04;

import java.util.concurrent.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试 FutureTask 获取异步结果
```

```
 */
public class FutureTaskTest {

    public static void main(String[] args)throws ExecutionException, InterruptedE
xception{
        FutureTask<String> futureTask = new FutureTask<>(new Callable<String>
() {
            @Override
            public String call() throws Exception {
                return "测试 FutureTask 获取异步结果";
            }
        });
        new Thread(futureTask).start();
        System.out.println(futureTask.get());
    }
}
```

运行结果如下所示。

```
测试 FutureTask 获取异步结果
```

结合线程池的使用示例如下。

```
package io.binghe.concurrent.lab04;

import java.util.concurrent.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试 FutureTask 获取异步结果
 */
public class FutureTaskTest {

    public static void main(String[] args) throws ExecutionException, InterruptedE
xception {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        FutureTask<String> futureTask = new FutureTask<>(new Callable<String>
() {
            @Override
```

```
            public String call() throws Exception {
                return "测试 FutureTask 获取异步结果";
            }
        });
        executorService.execute(futureTask);
        System.out.println(futureTask.get());
        executorService.shutdown();
    }
}
```

运行结果如下所示。

```
测试 FutureTask 获取异步结果
```

可以看到使用 Future 接口或者 FutureTask 类来获取异步结果比使用回调接口获取异步结果简单多了。注意：实现异步的方式很多，这里只是用多线程举例。

接下来，就深入分析下 Future 接口。

深度解析 Future 接口

1.Future 接口

Future 是 JDK1.5 新增的异步编程接口，其源代码如下所示。

```
package java.util.concurrent;

public interface Future<V> {

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

可以看到，在 Future 接口中，总共定义了 5 个抽象方法。接下来，就分别介绍下这 5 个方法的含义。

- cancel(boolean)

取消任务的执行，接收一个 boolean 类型的参数，成功取消任务，则返回 true，否则返回 false。当任务已经完成，已经结束或者因其他原因不能取消时，方法会返回 false，表示任务取消失败。当任务未启动调用了此方法，并且结果返回 true（取消成功），则当前任务不再运行。如果任务已经启动，会根据当前传递的 boolean 类型的参数来决定是否中断当前运行的线程来取消当前运行的任务。

- isCancelled()

判断任务在完成之前是否被取消，如果在任务完成之前被取消，则返回 true；否则，返回 false。

这里需要注意一个细节：只有任务未启动，或者在完成之前被取消，才会返回 true，表示任务已经被成功取消。其他情况都会返回 false。

- isDone()

判断任务是否已经完成，如果任务正常结束、抛出异常退出、被取消，都会返回 true，表示任务已经完成。

- get()

当任务完成时，直接返回任务的结果数据；当任务未完成时，等待任务完成并返回任务的结果数据。

- get(long, TimeUnit)

当任务完成时，直接返回任务的结果数据；当任务未完成时，等待任务完成，并设置了超时等待时间。在超时时间内任务完成，则返回结果；否则，抛出 TimeoutException 异常。

2.RunnableFuture 接口

Future 接口有一个重要的子接口，那就是 RunnableFuture 接口，RunnableFuture 接口不但继承了 Future 接口，而且继承了 java.lang.Runnable 接口，其源代码如下所示。

```
package java.util.concurrent;

public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
```

这里，问一下，RunnableFuture 接口中有几个抽象方法？想好了再说！哈哈哈。。。

这个接口比较简单 run()方法就是运行任务时调用的方法。

3.FutureTask 类

FutureTask 类是 RunnableFuture 接口的一个非常重要的实现类，它实现了 RunnableFuture 接口、Future 接口和 Runnable 接口的所有方法。FutureTask 类的源代码比较多，这个就不粘贴了，大家自行到 java.util.concurrent 下查看。

（1）FutureTask 类中的变量与常量

在 FutureTask 类中首先定义了一个状态变量 state，这个变量使用了 volatile 关键字修饰，这里，大家只需要知道 volatile 关键字通过内存屏障和禁止重排序优化来实现线程安全，后续会单独深度分析 volatile 关键字是如何保证线程安全的。紧接着，定义了几个任务运行时的状态常量，如下所示。

```
private volatile int state;
private static final int NEW          = 0;
private static final int COMPLETING   = 1;
private static final int NORMAL       = 2;
private static final int EXCEPTIONAL  = 3;
private static final int CANCELLED    = 4;
private static final int INTERRUPTING = 5;
private static final int INTERRUPTED  = 6;
```

其中，代码注释中给出了几个可能的状态变更流程，如下所示。

```
NEW -> COMPLETING -> NORMAL
NEW -> COMPLETING -> EXCEPTIONAL
NEW -> CANCELLED
NEW -> INTERRUPTING -> INTERRUPTED
```

接下来，定义了其他几个成员变量，如下所示。

```
private Callable<V> callable;
private Object outcome;
private volatile Thread runner;
private volatile WaitNode waiters;
```

又看到我们所熟悉的 Callable 接口了，Callable 接口那肯定就是用来调用 call() 方法执行具体任务了。

- outcome：Object 类型，表示通过 get()方法获取到的结果数据或者异常信息。
- runner：运行 Callable 的线程，运行期间会使用 CAS 保证线程安全，这里大家只需要知道 CAS 是 Java 保证线程安全的一种方式，后续文章中会深度分析 CAS 如何保证线程安全。
- waiters：WaitNode 类型的变量，表示等待线程的堆栈，在 FutureTask 的实现中，会通过 CAS 结合此堆栈交换任务的运行状态。

看一下 WaitNode 类的定义，如下所示。

```
static final class WaitNode {
        volatile Thread thread;
        volatile WaitNode next;
        WaitNode() { thread = Thread.currentThread(); }
}
```

可以看到，WaitNode 类是 FutureTask 类的静态内部类，类中定义了一个 Thread 成员变量和指向下一个 WaitNode 节点的引用。其中通过构造方法将 thread 变量设置为当前线程。

（2）构造方法

接下来，是 FutureTask 的两个构造方法，比较简单，如下所示。

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

public FutureTask(Runnable runnable, V result) {
```

```
        this.callable = Executors.callable(runnable, result);
        this.state = NEW;
}
```

（3）是否取消与完成方法

继续向下看源码，看到一个任务是否取消的方法，和一个任务是否完成的方法，如下所示。

```
public boolean isCancelled() {
        return state >= CANCELLED;
}

public boolean isDone() {
        return state != NEW;
}
```

这两方法中，都是通过判断任务的状态来判定任务是否已取消和已完成的。为啥会这样判断呢？再次查看 FutureTask 类中定义的状态常量发现，其常量的定义是有规律的，并不是随意定义的。其中，大于或者等于 CANCELLED 的常量为 CANCELLED、INTERRUPTING 和 INTERRUPTED，这三个状态均可以表示线程已经被取消。当状态不等于 NEW 时，可以表示任务已经完成。

通过这里，大家可以学到一点：以后在编码过程中，要按照规律来定义自己使用的状态，尤其是涉及到业务中有频繁的状态变更的操作，有规律的状态可使业务处理变得事半功倍，这也是通过看别人的源码设计能够学到的，这里，建议大家还是多看别人写的优秀的开源框架的源码。

（4）取消方法

我们继续向下看源码，接下来，看到的是 cancel(boolean)方法，如下所示。

```
public boolean cancel(boolean mayInterruptIfRunning) {
        if (!(state == NEW &&
                UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
                        mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
                return false;
        try {    // in case call to interrupt throws exception
                if (mayInterruptIfRunning) {
                        try {
                                Thread t = runner;
```

```
                        if (t != null)
                            t.interrupt();
                    } finally { // final state
                        UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
                    }
                }
            } finally {
                finishCompletion();
            }
            return true;
        }
```

接下来，拆解 cancel(boolean)方法。在 cancel(boolean)方法中，首先判断任务的状态和 CAS 的操作结果，如果任务的状态不等于 NEW 或者 CAS 的操作返回 false，则直接返回 false，表示任务取消失败。如下所示。

```
if (!(state == NEW &&
          UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
              mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
        return false;
```

接下来，在 try 代码块中，首先判断是否可以中断当前任务所在的线程来取消任务的运行。如果可以中断当前任务所在的线程，则以一个 Thread 临时变量来指向运行任务的线程，当指向的变量不为空时，调用线程对象的 interrupt()方法来中断线程的运行，最后将线程标记为被中断的状态。如下所示。

```
try {
        if (mayInterruptIfRunning) {
            try {
                Thread t = runner;
                if (t != null)
                    t.interrupt();
            } finally { // final state
                UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
            }
        }
}
```

这里，发现变更任务状态使用的是 UNSAFE.putOrderedInt()方法，这个方法是个什么鬼呢？点进去看一下，如下所示。

```
public native void putOrderedInt(Object var1, long var2, int var4);
```

可以看到，又是一个本地方法，嘿嘿，这里先不管它，后续文章会详解这些方法的作用。

接下来，cancel(boolean)方法会进入 finally 代码块，如下所示。

```
finally {
        finishCompletion();
}
```

可以看到在 finallly 代码块中调用了 finishCompletion()方法，顾名思义，finishCompletion()方法表示结束任务的运行，接下来看看它是如何实现的。点到 finishCompletion()方法中看一下，如下所示。

```
private void finishCompletion() {
        // assert state > COMPLETING;
        for (WaitNode q; (q = waiters) != null;) {
                if (UNSAFE.compareAndSwapObject(this, waitersOffset, q, null)) {
                        for (;;) {
                                Thread t = q.thread;
                                if (t != null) {
                                        q.thread = null;
                                        LockSupport.unpark(t);
                                }
                                WaitNode next = q.next;
                                if (next == null)
                                        break;
                                q.next = null; // unlink to help gc
                                q = next;
                        }
                        break;
                }
        }
        done();
        callable = null;        // to reduce footprint
}
```

在 finishCompletion()方法中，首先定义一个 for 循环，循环终止因子为 waiters 为 null，在循环中，判断 CAS 操作是否成功，如果成功进行 if 条件中的逻辑。首先，定义一个 for 自旋循环，在自旋循环体中，唤醒 WaitNode 堆栈中的线程，使其运行完成。当 WaitNode 堆栈中的线程运行完成后，通过 break 退出外层 for 循环。接下来调用 done()方法。done()方法又是个什么鬼呢？点进去看一下，如下所示。

```
protected void done() { }
```

可以看到，done()方法是一个空的方法体，交由子类来实现具体的业务逻辑。

当我们的具体业务中，需要在取消任务时，执行一些额外的业务逻辑，可以在子类中覆写 done()方法的实现。

（5）get()方法

继续向下看 FutureTask 类的代码，FutureTask 类中实现了两个 get()方法，如下所示。

```
public V get() throws InterruptedException, ExecutionException {
        int s = state;
        if (s <= COMPLETING)
                s = awaitDone(false, 0L);
        return report(s);
}

public V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException {
        if (unit == null)
                throw new NullPointerException();
        int s = state;
        if (s <= COMPLETING &&
                (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING)
                throw new TimeoutException();
        return report(s);
}
```

没参数的 get()方法为当任务未运行完成时，会阻塞，直到返回任务结果。有参数的 get()方法为当任务未运行完成，并且等待时间超出了超时时间，会 TimeoutException 异常。

两个 get()方法的主要逻辑差不多，一个没有超时设置，一个有超时设置，这里说一下主要逻辑。判断任务的当前状态是否小于或者等于 COMPLETING，也就是说，任务是 NEW 状态或者 COMPLETING，调用 awaitDone()方法，看下 awaitDone()方法的实现，如下所示。

```
private int awaitDone(boolean timed, long nanos)
        throws InterruptedException {
        final long deadline = timed ? System.nanoTime() + nanos : 0L;
        WaitNode q = null;
        boolean queued = false;
        for (;;) {
                if (Thread.interrupted()) {
                        removeWaiter(q);
                        throw new InterruptedException();
                }

                int s = state;
                if (s > COMPLETING) {
                        if (q != null)
                                q.thread = null;
                        return s;
                }
                else if (s == COMPLETING) // cannot time out yet
                        Thread.yield();
                else if (q == null)
                        q = new WaitNode();
                else if (!queued)
                        queued = UNSAFE.compareAndSwapObject(this, waitersO
ffset,

         q.next = waiters, q);
                else if (timed) {
                        nanos = deadline - System.nanoTime();
                        if (nanos <= 0L) {
                                removeWaiter(q);
                                return state;
                        }
                        LockSupport.parkNanos(this, nanos);
                }
```

```
                else
                    LockSupport.park(this);
            }
        }
```

接下来，拆解 awaitDone()方法。在 awaitDone()方法中，最重要的就是 for 自旋循环，在循环中首先判断当前线程是否被中断，如果已经被中断，则调用 removeWaiter()将当前线程从堆栈中移除，并且抛出 InterruptedException 异常，如下所示。

```
if (Thread.interrupted()) {
        removeWaiter(q);
        throw new InterruptedException();
}
```

接下来，判断任务的当前状态是否完成，如果完成，并且堆栈句柄不为空，则将堆栈中的当前线程设置为空，返回当前任务的状态，如下所示。

```
int s = state;
if (s > COMPLETING) {
        if (q != null)
            q.thread = null;
        return s;
}
```

当任务的状态为 COMPLETING 时，使当前线程让出 CPU 资源，如下所示。

```
else if (s == COMPLETING)
        Thread.yield();
```

如果堆栈为空，则创建堆栈对象，如下所示。

```
else if (q == null)
        q = new WaitNode();
```

如果 queued 变量为 false，通过 CAS 操作为 queued 赋值，如果 awaitDone()方法传递的 timed 参数为 true，则计算超时时间，当时间已超时，则在堆栈中移除当前线程并返回任务状态，如下所示。如果未超时，则重置超时时间，如下所示。

```
else if (!queued)
        queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                             q.next = wait
```

```
ers, q);
else if (timed) {
        nanos = deadline - System.nanoTime();
        if (nanos <= 0L) {
                removeWaiter(q);
                return state;
        }
        LockSupport.parkNanos(this, nanos);
}
```

如果不满足上述的所有条件，则将当前线程设置为等待状态，如下所示。

```
else
        LockSupport.park(this);
```

接下来，回到 get()方法中，当 awaitDone()方法返回结果，或者任务的状态不满足条件时，都会调用 report()方法，并将当前任务的状态传递到 report()方法中，并返回结果，如下所示。

```
return report(s);
```

看来，这里还要看下 report()方法啊，点进去看下 report()方法的实现，如下所示。

```
private V report(int s) throws ExecutionException {
        Object x = outcome;
        if (s == NORMAL)
                return (V)x;
        if (s >= CANCELLED)
                throw new CancellationException();
        throw new ExecutionException((Throwable)x);
}
```

可以看到，report()方法的实现比较简单，首先，将 outcome 数据赋值给 x 变量，接下来，主要是判断接收到的任务状态，如果状态为 NORMAL，则将 x 强转为泛型类型返回；当任务的状态大于或者等于 CANCELLED，也就是任务已经取消，则抛出 CancellationException 异常，其他情况则抛出 ExecutionException 异常。

至此，get()方法分析完成。注意：一定要理解 get()方法的实现，因为 get()方法是我们使用 Future 接口和 FutureTask 类时，使用的比较频繁的一个方法。

（6）set()方法与 setException()方法

继续看 FutureTask 类的代码，接下来看到的是 set()方法与 setException()方法，如下所示。

```
protected void set(V v) {
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING))
 {
                outcome = v;
                UNSAFE.putOrderedInt(this, stateOffset, NORMAL); // final state
                finishCompletion();
        }
}

protected void setException(Throwable t) {
        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING))
 {
                outcome = t;
                UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final
state
                finishCompletion();
        }
}
```

通过源码可以看出，set()方法与 setException()方法整体逻辑几乎一样，只是在设置任务状态时一个将状态设置为 NORMAL，一个将状态设置为 EXCEPTIONAL。

至于 finishCompletion()方法，前面已经分析过。

（7）run()方法与 runAndReset()方法

接下来，就是 run()方法了，run()方法的源代码如下所示。

```
public void run() {
        if (state != NEW ||
                !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                                null, Thread.curren
tThread()))
                return;
        try {
                Callable<V> c = callable;
                if (c != null && state == NEW) {
```

```
                    V result;
                    boolean ran;
                    try {
                        result = c.call();
                        ran = true;
                    } catch (Throwable ex) {
                        result = null;
                        ran = false;
                        setException(ex);
                    }
                    if (ran)
                        set(result);
                }
            } finally {
                // runner must be non-null until state is settled to
                // prevent concurrent calls to run()
                runner = null;
                // state must be re-read after nulling runner to prevent
                // leaked interrupts
                int s = state;
                if (s >= INTERRUPTING)
                    handlePossibleCancellationInterrupt(s);
            }
        }
```

可以这么说，只要使用了 Future 和 FutureTask，就必然会调用 run()方法来运行任务，掌握 run()方法的流程是非常有必要的。在 run()方法中，如果当前状态不是 NEW，或者 CAS 操作返回的结果为 false，则直接返回，不再执行后续逻辑，如下所示。

```
if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                                    null, Thread.currentThrea
d()))
        return;
```

接下来，在 try 代码块中，将成员变量 callable 赋值给一个临时变量 c，判断临时变量不等于 null，并且任务状态为 NEW，则调用 Callable 接口的 call()方法，并接收结果数据。并将 ran 变量设置为 true。当程序抛出异常时，将接收结果的变

量设置为 null，ran 变量设置为 false，并且调用 setException()方法将任务的状态设置为 EXCEPTIONA。接下来，如果 ran 变量为 true，则调用 set()方法，如下所示。

```
try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
                V result;
                boolean ran;
                try {
                        result = c.call();
                        ran = true;
                } catch (Throwable ex) {
                        result = null;
                        ran = false;
                        setException(ex);
                }
                if (ran)
                        set(result);
        }
}
```

接下来，程序会进入 finally 代码块中，如下所示。

```
finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
        runner = null;
        // state must be re-read after nulling runner to prevent
        // leaked interrupts
        int s = state;
        if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s);
}
```

这里，将 runner 设置为 null，如果任务的当前状态大于或者等于 INTERRUPTING，也就是线程被中断了。则调用 handlePossibleCancellationInterrupt()方法，接下来，看下 handlePossibleCancellationInterrupt()方法的实现。

```
private void handlePossibleCancellationInterrupt(int s) {
        if (s == INTERRUPTING)
                while (state == INTERRUPTING)
                        Thread.yield();
}
```

可以看到，handlePossibleCancellationInterrupt()方法的实现比较简单，当任务的状态为 INTERRUPTING 时，使用 while()循环，条件为当前任务状态为 INTERRUPTING，将当前线程占用的 CPU 资源释放，也就是说，当任务运行完成后，释放线程所占用的资源。

runAndReset()方法的逻辑与 run()差不多，只是 runAndReset()方法会在 finally 代码块中将任务状态重置为 NEW。runAndReset()方法的源代码如下所示，就不重复说了。

```
protected boolean runAndReset() {
        if (state != NEW ||
                !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                                                        null, Thread.curren
tThread()))
                return false;
        boolean ran = false;
        int s = state;
        try {
                Callable<V> c = callable;
                if (c != null && s == NEW) {
                        try {
                                c.call(); // don't set result
                                ran = true;
                        } catch (Throwable ex) {
                                setException(ex);
                        }
                }
        } finally {
                // runner must be non-null until state is settled to
                // prevent concurrent calls to run()
                runner = null;
                // state must be re-read after nulling runner to prevent
                // leaked interrupts
                s = state;
```

```
                if (s >= INTERRUPTING)
                        handlePossibleCancellationInterrupt(s);
        }
        return ran && s == NEW;
}
```

（8）removeWaiter()方法

removeWaiter()方法中主要是使用自旋循环的方式来移除 WaitNode 中的线程，比较简单，如下所示。

```
private void removeWaiter(WaitNode node) {
        if (node != null) {
                node.thread = null;
                retry:
                for (;;) {          // restart on removeWaiter race
                        for (WaitNode pred = null, q = waiters, s; q != null; q = s) {
                                s = q.next;
                                if (q.thread != null)
                                        pred = q;
                                else if (pred != null) {
                                        pred.next = s;
                                        if (pred.thread == null) // check for race
                                                continue retry;
                                }
                                else if (!UNSAFE.compareAndSwapObject(this, wait
ersOffset,

          q, s))
                                        continue retry;
                        }
                        break;
                }
        }
}
```

最后，在 FutureTask 类的最后，有如下代码。

```
// Unsafe mechanics
private static final sun.misc.Unsafe UNSAFE;
private static final long stateOffset;
```

```
private static final long runnerOffset;
private static final long waitersOffset;
static {
        try {
                UNSAFE = sun.misc.Unsafe.getUnsafe();
                Class<?> k = FutureTask.class;
                stateOffset = UNSAFE.objectFieldOffset
                        (k.getDeclaredField("state"));
                runnerOffset = UNSAFE.objectFieldOffset
                        (k.getDeclaredField("runner"));
                waitersOffset = UNSAFE.objectFieldOffset
                        (k.getDeclaredField("waiters"));
        } catch (Exception e) {
                throw new Error(e);
        }
}
```

关于这些代码的作用，会在后续深度解析 CAS 文章中详细说明，这里就不再探讨。

至此，关于 Future 接口和 FutureTask 类的源码就分析完了。

五、SimpleDateFormat 类的线程安全问题

提起 SimpleDateFormat 类，想必做过 Java 开发的童鞋都不会感到陌生。没错，它就是 Java 中提供的日期时间的转化类。这里，为什么说 SimpleDateFormat 类有线程安全问题呢？有些小伙伴可能会提出疑问：我们生产环境上一直在使用 SimpleDateFormat 类来解析和格式化日期和时间类型的数据，一直都没有问题啊！我的回答是：没错，那是因为你们的系统达不到 SimpleDateFormat 类出现问题的并发量，也就是说你们的系统没啥负载！

接下来，我们就一起看下在高并发下 SimpleDateFormat 类为何会出现安全问题，以及如何解决 SimpleDateFormat 类的安全问题。

重现 SimpleDateFormat 类的线程安全问题

为了重现 SimpleDateFormat 类的线程安全问题，一种比较简单的方式就是使用线程池结合 Java 并发包中的 CountDownLatch 类和 Semaphore 类来重现线程安全问题。

有关 CountDownLatch 类和 Semaphore 类的具体用法和底层原理与源码解析在【高并发专题】后文会深度分析。这里，大家只需要知道 CountDownLatch 类可以使一个线程等待其他线程各自执行完毕后再执行。而 Semaphore 类可以理解为一个计数信号量，必须由获取它的线程释放，经常用来限制访问某些资源的线程数量，例如限流等。

好了，先来看下重现 SimpleDateFormat 类的线程安全问题的代码，如下所示。

```
package io.binghe.concurrent.lab06;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试 SimpleDateFormat 的线程不安全问题
 */
```

```
public class SimpleDateFormatTest01 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;
    //SimpleDateFormat 对象
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat
("yyyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUT
E_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        simpleDateFormat.parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程： " + Thread.currentThread().getName() +
" 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程： " + Thread.currentThread().getName() +
" 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
```

```
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

可以看到，在 SimpleDateFormatTest01 类中，首先定义了两个常量，一个是程序执行的总次数，一个是同时运行的线程数量。程序中结合线程池和 CountDownLatch 类与 Semaphore 类来模拟高并发的业务场景。其中，有关日期转化的代码只有如下一行。

```
simpleDateFormat.parse("2020-01-01");
```

当程序捕获到异常时，打印相关的信息，并退出整个程序的运行。当程序正确运行后，会打印“所有线程格式化日期成功”。

运行程序输出的结果信息如下所示。

```
Exception in thread "pool-1-thread-4" Exception in thread "pool-1-thread-1" E
xception in thread "pool-1-thread-2" 线程：pool-1-thread-7 格式化日期失败
线程：pool-1-thread-9 格式化日期失败
线程：pool-1-thread-10 格式化日期失败
Exception in thread "pool-1-thread-3" Exception in thread "pool-1-thread-5" E
xception in thread "pool-1-thread-6" 线程：pool-1-thread-15 格式化日期失败
线程：pool-1-thread-21 格式化日期失败
Exception in thread "pool-1-thread-23" 线程：pool-1-thread-16 格式化日期失败
线程：pool-1-thread-11 格式化日期失败
java.lang.ArrayIndexOutOfBoundsException
线程：pool-1-thread-27 格式化日期失败
        at java.lang.System.arraycopy(Native Method)
       at  java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:597)
        at java.lang.StringBuffer.append(StringBuffer.java:367)
        at java.text.DigitList.getLong(DigitList.java:191)线程：pool-1-thread-25
格式化日期失败

        at java.text.DecimalFormat.parse(DecimalFormat.java:2084)
        at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:1869)
        at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)
线程：pool-1-thread-14 格式化日期失败
```

```
        at java.text.DateFormat.parse(DateFormat.java:364)
        at io.binghe.concurrent.lab06.SimpleDateFormatTest01.lambda$main$0
(SimpleDateFormatTest01.java:47)
线程：pool-1-thread-13 格式化日期失败  at java.util.concurrent.ThreadPoolExec
utor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExec
utor.java:624)

        at java.lang.Thread.run(Thread.java:748)
java.lang.NumberFormatException: For input string: ""
        at java.lang.NumberFormatException.forInputString(NumberFormatExce
ption.java:65)
线程：pool-1-thread-20 格式化日期失败 at java.lang.Long.parseLong(Long.java:
601)
        at java.lang.Long.parseLong(Long.java:631)

        at java.text.DigitList.getLong(DigitList.java:195)
        at java.text.DecimalFormat.parse(DecimalFormat.java:2084)
        at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:2162)
        at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)
        at java.text.DateFormat.parse(DateFormat.java:364)
        at io.binghe.concurrent.lab06.SimpleDateFormatTest01.lambda$main$0
(SimpleDateFormatTest01.java:47)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecu
tor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExec
utor.java:624)
        at java.lang.Thread.run(Thread.java:748)
java.lang.NumberFormatException: For input string: ""
        at java.lang.NumberFormatException.forInputString(NumberFormatExce
ption.java:65)
        at java.lang.Long.parseLong(Long.java:601)
        at java.lang.Long.parseLong(Long.java:631)
        at java.text.DigitList.getLong(DigitList.java:195)
        at java.text.DecimalFormat.parse(DecimalFormat.java:2084)
        at java.text.SimpleDateFormat.subParse(SimpleDateFormat.java:1869)
        at java.text.SimpleDateFormat.parse(SimpleDateFormat.java:1514)
        at java.text.DateFormat.parse(DateFormat.java:364)
```

```
Process finished with exit code 1
```

说明，在高并发下使用 SimpleDateFormat 类格式化日期时抛出了异常，SimpleDateFormat 类不是线程安全的！！！

接下来，我们就看下，SimpleDateFormat 类为何不是线程安全的。

SimpleDateFormat 类为何不是线程安全的?

那么，接下来，我们就一起来看看真正引起 SimpleDateFormat 类线程不安全的根本原因。

通过查看 SimpleDateFormat 类的源码，我们得知：SimpleDateFormat 是继承自 DateFormat 类，DateFormat 类中维护了一个全局的 Calendar 变量，如下所示。

```
/**
 * The {@link Calendar} instance used for calculating the date-time fields
 * and the instant of time. This field is used for both formatting and
 * parsing.
 *
 * <p>Subclasses should initialize this field to a {@link Calendar}
 * appropriate for the {@link Locale} associated with this
 * <code>DateFormat</code>.
 * @serial
 */
protected Calendar calendar;
```

从注释可以看出，这个 Calendar 对象既用于格式化也用于解析日期时间。接下来，我们再查看 parse()方法接近最后的部分。

```
@Override
```

```
public Date parse(String text, ParsePosition pos){
    ################此处省略 N 行代码##################
    Date parsedDate;
    try {
        parsedDate = calb.establish(calendar).getTime();
        // If the year value is ambiguous,
        // then the two-digit year == the default start year
        if (ambiguousYear[0]) {
            if (parsedDate.before(defaultCenturyStart)) {
                parsedDate = calb.addYear(100).establish(calendar).getTime();
            }
        }
    }
    // An IllegalArgumentException will be thrown by Calendar.getTime()
    // if any fields are out of range, e.g., MONTH == 17.
    catch (IllegalArgumentException e) {
        pos.errorIndex = start;
        pos.index = oldStart;
        return null;
    }
    return parsedDate;
}
```

可见，最后的返回值是通过调用 CalendarBuilder.establish()方法获得的，而这个方法的参数正好就是前面的 Calendar 对象。

接下来，我们再来看看 CalendarBuilder.establish()方法，如下所示。

```
Calendar establish(Calendar cal) {
    boolean weekDate = isSet(WEEK_YEAR)
        && field[WEEK_YEAR] > field[YEAR];
    if (weekDate && !cal.isWeekDateSupported()) {
        // Use YEAR instead
        if (!isSet(YEAR)) {
            set(YEAR, field[MAX_FIELD + WEEK_YEAR]);
        }
        weekDate = false;
    }

    cal.clear();
    // Set the fields from the min stamp to the max stamp so that
    // the field resolution works in the Calendar.
    for (int stamp = MINIMUM_USER_STAMP; stamp < nextStamp; stamp++) {
        for (int index = 0; index <= maxFieldIndex; index++) {
            if (field[index] == stamp) {
                cal.set(index, field[MAX_FIELD + index]);
                break;
            }
        }
```

```
    }


    if (weekDate) {
        int weekOfYear = isSet(WEEK_OF_YEAR) ? field[MAX_FIELD + WEEK_O
F_YEAR] : 1;
        int dayOfWeek = isSet(DAY_OF_WEEK) ?
            field[MAX_FIELD + DAY_OF_WEEK] : cal.getFirstDayOfWeek();
        if (!isValidDayOfWeek(dayOfWeek) && cal.isLenient()) {
            if (dayOfWeek >= 8) {
                dayOfWeek--;
                weekOfYear += dayOfWeek / 7;
                dayOfWeek = (dayOfWeek % 7) + 1;
            } else {
                while (dayOfWeek <= 0) {
                    dayOfWeek += 7;
                    weekOfYear--;
                }
            }
            dayOfWeek = toCalendarDayOfWeek(dayOfWeek);
        }
        cal.setWeekDate(field[MAX_FIELD + WEEK_YEAR], weekOfYear, dayOf
Week);
    }
    return cal;
}
```

在 CalendarBuilder.establish()方法中先后调用了 cal.clear()与 cal.set()，也就是先清除 cal 对象中设置的值，再重新设置新的值。由于 Calendar 内部并没有线程安全机制，并且这两个操作也都不是原子性的，所以当多个线程同时操作一个 SimpleDateFormat 时就会引起 cal 的值混乱。类似地， format()方法也存在同样的问题。

因此， SimpleDateFormat 类不是线程安全的根本原因是：DateFormat 类中的 Calendar 对象被多线程共享，而 Calendar 对象本身不支持线程安全。

那么，得知了 SimpleDateFormat 类不是线程安全的，以及造成 SimpleDateFormat 类不是线程安全的原因，那么如何解决这个问题呢？接下来，我们就一起探讨下如何解决 SimpleDateFormat 类在高并发场景下的线程安全问题。

解决 SimpleDateFormat 类的线程安全问题

解决 SimpleDateFormat 类在高并发场景下的线程安全问题可以有多种方式，这里，就列举几个常用的方式供参考，大家也可以在评论区给出更多的解决方案。

1.局部变量法

最简单的一种方式就是将 SimpleDateFormat 类对象定义成局部变量，如下所示的代码，将 SimpleDateFormat 类对象定义在 parse(String)方法的上面，即可解决问题。

```
package io.binghe.concurrent.lab06;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
```

```
 * @author binghe
 * @version 1.0.0
 * @description 局部变量法解决 SimpleDateFormat 类的线程安全问题
 */
public class SimpleDateFormatTest02 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUTE_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
                        simpleDateFormat.parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程：" + Thread.currentThread().getName() + " 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
```

```
            }
            countDownLatch.countDown();
        });
    }
    countDownLatch.await();
    executorService.shutdown();
    System.out.println("所有线程格式化日期成功");
  }
}
```

此时运行修改后的程序，输出结果如下所示。

```
所有线程格式化日期成功
```

至于在高并发场景下使用局部变量为何能解决线程的安全问题，会在【 JVM 专题 】的 JVM 内存模式相关内容中深入剖析，这里不做过多的介绍了。

当然，这种方式在高并发下会创建大量的 SimpleDateFormat 类对象，影响程序的性能，所以，这种方式在实际生产环境不太被推荐。

2.synchronized 锁方式

将 SimpleDateFormat 类对象定义成全局静态变量，此时所有线程共享 SimpleDateFormat 类对象，此时在调用格式化时间的方法时，对 SimpleDateFormat 对象进行同步即可，代码如下所示。

```
package io.binghe.concurrent.lab06;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过 Synchronized 锁解决 SimpleDateFormat 类的线程安全问题
 */
public class SimpleDateFormatTest03 {
```

```
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;
    //SimpleDateFormat 对象
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat
("yyyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUT
E_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        synchronized (simpleDateFormat){
                            simpleDateFormat.parse("2020-01-01");
                        }
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() +
" 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程：" + Thread.currentThread().getName() +
" 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
```

```
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

此时，解决问题的关键代码如下所示。

```
synchronized (simpleDateFormat){
            simpleDateFormat.parse("2020-01-01");
}
```

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

需要注意的是，虽然这种方式能够解决 SimpleDateFormat 类的线程安全问题，但是由于在程序的执行过程中，为 SimpleDateFormat 类对象加上了 synchronized 锁，导致同一时刻只能有一个线程执行 parse(String)方法。此时，会影响程序的执行性能，在要求高并发的生产环境下，此种方式也是不太推荐使用的。

3.Lock 锁方式

Lock 锁方式与 synchronized 锁方式实现原理相同，都是在高并发下通过 JVM 的锁机制来保证程序的线程安全。通过 Lock 锁方式解决问题的代码如下所示。

```
package io.binghe.concurrent.lab06;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
```

```
 * @author binghe
 * @version 1.0.0
 * @description 通过 Lock 锁解决 SimpleDateFormat 类的线程安全问题
 */
public class SimpleDateFormatTest04 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;
    //SimpleDateFormat 对象
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat
("yyyy-MM-dd");
    //Lock 对象
    private static Lock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUT
E_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        lock.lock();
                        simpleDateFormat.parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程： " + Thread.currentThread().getName() +
" 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }catch (NumberFormatException e){
                        System.out.println("线程： " + Thread.currentThread().getName() +
" 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }finally {
                        lock.unlock();
```

```
                }
                semaphore.release();
            } catch (InterruptedException e) {
                System.out.println("信号量发生错误");
                e.printStackTrace();
                System.exit(1);
            }
            countDownLatch.countDown();
        });
    }
    countDownLatch.await();
    executorService.shutdown();
    System.out.println("所有线程格式化日期成功");
  }
}
```

通过代码可以得知，首先，定义了一个 Lock 类型的全局静态变量作为加锁和释放锁的句柄。然后在 simpleDateFormat.parse(String)代码之前通过 lock.lock()加锁。这里需要注意的一点是：为防止程序抛出异常而导致锁不能被释放，一定要将释放锁的操作放到 finally 代码块中，如下所示。

```
finally {
        lock.unlock();
}
```

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

此种方式同样会影响高并发场景下的性能，不太建议在高并发的生产环境使用。

4.ThreadLocal 方式

使用 ThreadLocal 存储每个线程拥有的 SimpleDateFormat 对象的副本，能够有效的避免多线程造成的线程安全问题，使用 ThreadLocal 解决线程安全问题的代码如下所示。

```
package io.binghe.concurrent.lab06;

import java.text.DateFormat;
import java.text.ParseException;
```

```
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过 ThreadLocal 解决 SimpleDateFormat 类的线程安全问题
 */
public class SimpleDateFormatTest05 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    private static ThreadLocal<DateFormat> threadLocal = new ThreadLocal<Da
teFormat>(){
        @Override
        protected DateFormat initialValue() {
            return new SimpleDateFormat("yyyy-MM-dd");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUT
E_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        threadLocal.get().parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程： " + Thread.currentThread().getName() +
 " 格式化日期失败");
```

```
                    e.printStackTrace();
                    System.exit(1);
                }catch (NumberFormatException e){
                    System.out.println("线程：" + Thread.currentThread().getName() +
" 格式化日期失败");
                    e.printStackTrace();
                    System.exit(1);
                }
                semaphore.release();
            } catch (InterruptedException e) {
                System.out.println("信号量发生错误");
                e.printStackTrace();
                System.exit(1);
            }
            countDownLatch.countDown();
        });
    }
    countDownLatch.await();
    executorService.shutdown();
    System.out.println("所有线程格式化日期成功");
  }
}
```

通过代码可以得知，将每个线程使用的 SimpleDateFormat 副本保存在 ThreadLocal 中，各个线程在使用时互不干扰，从而解决了线程安全问题。

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

此种方式运行效率比较高，推荐在高并发业务场景的生产环境使用。

另外，使用 ThreadLocal 也可以写成如下形式的代码，效果是一样的。

```
package io.binghe.concurrent.lab06;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
 * @author binghe
 * @version 1.0.0
 * @description 通过 ThreadLocal 解决 SimpleDateFormat 类的线程安全问题
 */
public class SimpleDateFormatTest06 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    private static ThreadLocal<DateFormat> threadLocal = new ThreadLocal<Da
teFormat>();

    private static DateFormat getDateFormat(){
        DateFormat dateFormat = threadLocal.get();
        if(dateFormat == null){
            dateFormat = new SimpleDateFormat("yyyy-MM-dd");
            threadLocal.set(dateFormat);
        }
        return dateFormat;
    }

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUT
E_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        getDateFormat().parse("2020-01-01");
                    } catch (ParseException e) {
                        System.out.println("线程：" + Thread.currentThread().getName() +
```

```
" 格式化日期失败");
                    e.printStackTrace();
                    System.exit(1);
                }catch (NumberFormatException e){
                    System.out.println("线程：" + Thread.currentThread().getName() +
" 格式化日期失败");
                    e.printStackTrace();
                    System.exit(1);
                }
                semaphore.release();
            } catch (InterruptedException e) {
                System.out.println("信号量发生错误");
                e.printStackTrace();
                System.exit(1);
            }
            countDownLatch.countDown();
        });
    }
    countDownLatch.await();
    executorService.shutdown();
    System.out.println("所有线程格式化日期成功");
  }
}
```

5.DateTimeFormatter 方式

DateTimeFormatter 是 Java8 提供的新的日期时间 API 中的类，DateTimeFormatter 类是线程安全的，可以在高并发场景下直接使用 DateTimeFormatter 类来处理日期的格式化操作。代码如下所示。

```
package io.binghe.concurrent.lab06;

import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
```

```
 * @author binghe
 * @version 1.0.0
 * @description 通过 DateTimeFormatter 类解决线程安全问题
 */
public class SimpleDateFormatTest07 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

   private static DateTimeFormatter formatter = DateTimeFormatter.ofPattern("y
yyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUT
E_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        LocalDate.parse("2020-01-01", formatter);
                    }catch (Exception e){
                        System.out.println("线程： " + Thread.currentThread().getName() +
 " 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
```

```
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

可以看到，DateTimeFormatter 类是线程安全的，可以在高并发场景下直接使用 DateTimeFormatter 类来处理日期的格式化操作。

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

使用 DateTimeFormatter 类来处理日期的格式化操作运行效率比较高，推荐在高并发业务场景的生产环境使用。

6.joda-time 方式

joda-time 是第三方处理日期时间格式化的类库，是线程安全的。如果使用 joda-time 来处理日期和时间的格式化，则需要引入第三方类库。这里，我以 Maven 为例，如下所示引入 joda-time 库。

```
<dependency>
        <groupId>joda-time</groupId>
        <artifactId>joda-time</artifactId>
        <version>2.9.9</version>
</dependency>
```

引入 joda-time 库后，实现的程序代码如下所示。

```
package io.binghe.concurrent.lab06;

import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

/**
```

```
 * @author binghe
 * @version 1.0.0
 * @description 通过 DateTimeFormatter 类解决线程安全问题
 */
public class SimpleDateFormatTest08 {
    //执行总次数
    private static final int EXECUTE_COUNT = 1000;
    //同时运行的线程数量
    private static final int THREAD_COUNT = 20;

    private static DateTimeFormatter dateTimeFormatter = DateTimeFormat.forP
attern("yyyy-MM-dd");

    public static void main(String[] args) throws InterruptedException {
        final Semaphore semaphore = new Semaphore(THREAD_COUNT);
        final CountDownLatch countDownLatch = new CountDownLatch(EXECUT
E_COUNT);
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < EXECUTE_COUNT; i++){
            executorService.execute(() -> {
                try {
                    semaphore.acquire();
                    try {
                        DateTime.parse("2020-01-01", dateTimeFormatter).toDate();
                    }catch (Exception e){
                        System.out.println("线程：" + Thread.currentThread().getName() +
 " 格式化日期失败");
                        e.printStackTrace();
                        System.exit(1);
                    }
                    semaphore.release();
                } catch (InterruptedException e) {
                    System.out.println("信号量发生错误");
                    e.printStackTrace();
                    System.exit(1);
                }
                countDownLatch.countDown();
            });
        }
```

```
        countDownLatch.await();
        executorService.shutdown();
        System.out.println("所有线程格式化日期成功");
    }
}
```

这里，需要注意的是：DateTime 类是 org.joda.time 包下的类，DateTimeFormat 类和 DateTimeFormatter 类都是 org.joda.time.format 包下的类，如下所示。

```
import org.joda.time.DateTime;
import org.joda.time.format.DateTimeFormat;
import org.joda.time.format.DateTimeFormatter;
```

运行程序，输出结果如下所示。

```
所有线程格式化日期成功
```

使用 joda-time 库来处理日期的格式化操作运行效率比较高，推荐在高并发业务场景的生产环境使用。

综上所示：在解决解决 SimpleDateFormat 类的线程安全问题的几种方案中，局部变量法由于线程每次执行格式化时间时，都会创建 SimpleDateFormat 类的对象，这会导致创建大量的 SimpleDateFormat 对象，浪费运行空间和消耗服务器的性能，因为 JVM 创建和销毁对象是要耗费性能的。所以，不推荐在高并发要求的生产环境使用。

synchronized 锁方式和 Lock 锁方式在处理问题的本质上是一致的，通过加锁的方式，使同一时刻只能有一个线程执行格式化日期和时间的操作。这种方式虽然减少了 SimpleDateFormat 对象的创建，但是由于同步锁的存在，导致性能下降，所以，不推荐在高并发要求的生产环境使用。

ThreadLocal 通过保存各个线程的 SimpleDateFormat 类对象的副本，使每个线程在运行时，各自使用自身绑定的 SimpleDateFormat 对象，互不干扰，执行性能比较高，推荐在高并发的生产环境使用。

DateTimeFormatter 是 Java 8 中提供的处理日期和时间的类，DateTimeFormatter 类本身就是线程安全的，经压测，DateTimeFormatter 类处理日期和时间的性能效果还不错（后文单独写一篇关于高并发下性能压测的文章）。所以，推荐在高并发场景下的生产环境使用。

joda-time 是第三方处理日期和时间的类库，线程安全，性能经过高并发的考验，推荐在高并发场景下的生产环境使用。

六、不得不说的线程池与 ThreadPoolExecutor 类浅析

抛砖引玉

既然 Java 中支持以多线程的方式来执行相应的任务，但为什么在 JDK1.5 中又提供了线程池技术呢？这个问题大家自行脑补，多动脑，肯定没坏处，哈哈哈。。。

说起 Java 中的线程池技术，在很多框架和异步处理中间件中都有涉及，而且性能经受起了长久的考验。可以这样说，Java 的线程池技术是 Java 最核心的技术之一，在 Java 的高并发领域中，Java 的线程池技术是一个永远绕不开的话题。既然 Java 的线程池技术这么重要（怎么能说是这么重要呢？那是相当的重要，那家伙老重要了，哈哈哈），那么，本文我们就来简单的说下线程池与 ThreadPoolExecutor 类。至于线程池中的各个技术细节和 ThreadPoolExecutor 的底层原理和源码解析，我们会在【高并发专题】专栏中进行深度解析。

引言：本文是高并发中线程池的开篇之作，就暂时先不深入讲解，只是让大家从整体上认识下线程池中最核心的类之一——ThreadPoolExecutor，关于 ThreadPoolExecutor 的底层原理和源码实现，以及线程池中的其他技术细节的底层原理和源码实现，我们会在【高并发专题】接下来的文章中，进行死磕。

Thread 直接创建线程的弊端

（1）每次 new Thread 新建对象，性能差。
（2）线程缺乏统一管理，可能无限制的新建线程，相互竞争，有可能占用过多系统资源导致死机或 OOM。
（3）缺少更多的功能，如更多执行、定期执行、线程中断。
（4）其他弊端，大家自行脑补，多动脑，没坏处，哈哈。

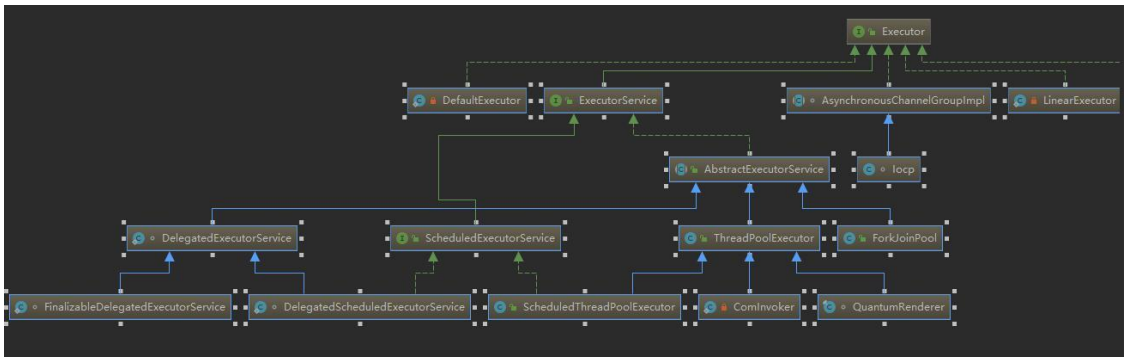
线程池的好处

（1）重用存在的线程，减少对象创建、消亡的开销，性能佳。
（2）可以有效控制最大并发线程数，提高系统资源利用率，同时可以避免过多资源竞争，避免阻塞。
（3）提供定时执行、定期执行、单线程、并发数控制等功能。
（4）提供支持线程池监控的方法，可对线程池的资源进行实时监控。
（5）其他好处，大家自行脑补，多动脑，没坏处，哈哈。

线程池

1.线程池类结构关系

线程池中的一些接口和类的结构关系如下图所示。



后文会死磕这些接口和类的底层原理和源码。

2.创建线程池常用的类——Executors

- Executors.newCachedThreadPool：创建一个可缓存的线程池，如果线程池的大小超过了需要，可以灵活回收空闲线程，如果没有可回收线程，则新建线程
- Executors.newFixedThreadPool：创建一个定长的线程池，可以控制线程的最大并发数，超出的线程会在队列中等待
- Executors.newScheduledThreadPool：创建一个定长的线程池，支持定时、周期性的任务执行
- Executors.newSingleThreadExecutor: 创建一个单线程化的线程池，使用一个唯一的工作线程执行任务，保证所有任务按照指定顺序（先入先出或者优先级）执行
- Executors.newSingleThreadScheduledExecutor:创建一个单线程化的线程池，支持定时、周期性的任务执行
- Executors.newWorkStealingPool：创建一个具有并行级别的 work-stealing 线程池

3.线程池实例的几种状态

- Running:运行状态，能接收新提交的任务，并且也能处理阻塞队列中的任务
- Shutdown: 关闭状态，不能再接收新提交的任务，但是可以处理阻塞队列中已经保存的任务，当线程池处于 Running 状态时，调用 shutdown()方法会使线程池进入该状态
- Stop: 不能接收新任务，也不能处理阻塞队列中已经保存的任务，会中断正在处理任务的线程，如果线程池处于 Running 或 Shutdown 状态，调用 shutdownNow()方法，会使线程池进入该状态
- Tidying: 如果所有的任务都已经终止，有效线程数为 0（阻塞队列为空，线程池中的工作线程数量为 0），线程池就会进入该状态。
- Terminated: 处于 Tidying 状态的线程池调用 terminated()方法，会使用线程池进入该状态

注意：不需要对线程池的状态做特殊的处理，线程池的状态是线程池内部根据方法自行定义和处理的。

4.合理配置线程的一些建议

（1）CPU 密集型任务，就需要尽量压榨 CPU，参考值可以设置为 NCPU+1(CPU 的数量加 1)。
（2）IO 密集型任务，参考值可以设置为 2*NCPU（CPU 数量乘以 2）

线程池最核心的类之一——ThreadPoolExecutor

1.构造方法

ThreadPoolExecutor 参数最多的构造方法如下：

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

其他的构造方法都是调用的这个构造方法来实例化对象，可以说，我们直接分析这个方法之后，其他的构造方法我们也明白是怎么回事了！接下来，就对此构造方法进行详细的分析。

注意：为了更加深入的分析 ThreadPoolExecutor 类的构造方法，会适当调整参数的顺序进行解析，以便于大家更能深入的理解 ThreadPoolExecutor 构造方法中每个参数的作用。

上述构造方法接收如下参数进行初始化：

（1）corePoolSize：核心线程数量。

（2）maximumPoolSize：最大线程数。

（3）workQueue：阻塞队列，存储等待执行的任务，很重要，会对线程池运行过程产生重大影响。

其中，上述三个参数的关系如下所示：

- 如果运行的线程数小于 corePoolSize，直接创建新线程处理任务，即使线程池中的其他线程是空闲的。
- 如果运行的线程数大于等于 corePoolSize，并且小于 maximumPoolSize，此时，只有当 workQueue 满时，才会创建新的线程处理任务。
- 如果设置的 corePoolSize 与 maximumPoolSize 相同，那么创建的线程池大小是固定的，此时，如果有新任务提交，并且 workQueue 没有满时，就把请求放入到 workQueue 中，等待空闲的线程，从 workQueue 中取出任务进行处理。
- 如果运行的线程数量大于 maximumPoolSize，同时，workQueue 已经满了，会通过拒绝策略参数 rejectHandler 来指定处理策略。

根据上述三个参数的配置，线程池会对任务进行如下处理方式：

当提交一个新的任务到线程池时，线程池会根据当前线程池中正在运行的线程数量来决定该任务的处理方式。处理方式总共有三种：直接切换、使用无限队列、使用有界队列。

- 直接切换常用的队列就是 SynchronousQueue。

- 使用无限队列就是使用基于链表的队列，比如：LinkedBlockingQueue，如果使用这种方式，线程池中创建的最大线程数就是 corePoolSize，此时 maximumPoolSize 不会起作用。当线程池中所有的核心线程都是运行状态时，提交新任务，就会放入等待队列中。
- 使用有界队列使用的是 ArrayBlockingQueue，使用这种方式可以将线程池的最大线程数量限制为 maximumPoolSize，可以降低资源的消耗。但是，这种方式使得线程池对线程的调度更困难，因为线程池和队列的容量都是有限的了。

根据上面三个参数，我们可以简单得出如何降低系统资源消耗的一些措施：

- 如果想降低系统资源的消耗，包括 CPU 使用率，操作系统资源的消耗，上下文环境切换的开销等，可以设置一个较大的队列容量和较小的线程池容量。这样，会降低线程处理任务的吞吐量。
- 如果提交的任务经常发生阻塞，可以考虑调用设置最大线程数的方法，重新设置线程池最大线程数。如果队列的容量设置的较小，通常需要将线程池的容量设置的大一些，这样，CPU 的使用率会高些。如果线程池的容量设置的过大，并发量就会增加，则需要考虑线程调度的问题，反而可能会降低处理任务的吞吐量。

接下来，我们继续看 ThreadPoolExecutor 的构造方法的参数。

（4）keepAliveTime：线程没有任务执行时最多保持多久时间终止
当线程池中的线程数量大于 corePoolSize 时，如果此时没有新的任务提交，核心线程外的线程不会立即销毁，需要等待，直到等待的时间超过了 keepAliveTime 就会终止。

（5）unit：keepAliveTime 的时间单位

（6）threadFactory：线程工厂，用来创建线程
默认会提供一个默认的工厂来创建线程，当使用默认的工厂来创建线程时，会使新创建的线程具有相同的优先级，并且是非守护的线程，同时也设置了线程的名称

（7）rejectHandler：拒绝处理任务时的策略

如果 workQueue 阻塞队列满了，并且没有空闲的线程池，此时，继续提交任务，需要采取一种策略来处理这个任务。

线程池总共提供了四种策略：

- 直接抛出异常，这也是默认的策略。实现类为 AbortPolicy。
- 用调用者所在的线程来执行任务。实现类为 CallerRunsPolicy。
- 丢弃队列中最靠前的任务并执行当前任务。实现类为 DiscardOldestPolicy。
- 直接丢弃当前任务。实现类为 DiscardPolicy。

2.ThreadPoolExecutor 提供的启动和停止任务的方法

（1）execute():提交任务，交给线程池执行
（2）submit():提交任务，能够返回执行结果 execute+Future
（3）shutdown():关闭线程池，等待任务都执行完
（4）shutdownNow():立即关闭线程池，不等待任务执行完

3.ThreadPoolExecutor 提供的适用于监控的方法

（1）getTaskCount()：线程池已执行和未执行的任务总数
（2）getCompletedTaskCount()：已完成的任务数量
（3）getPoolSize()：线程池当前的线程数量
（4）getCorePoolSize()：线程池核心线程数
（5）getActiveCount():当前线程池中正在执行任务的线程数量

七、深度解析线程池中那些重要的顶层接口和抽象类

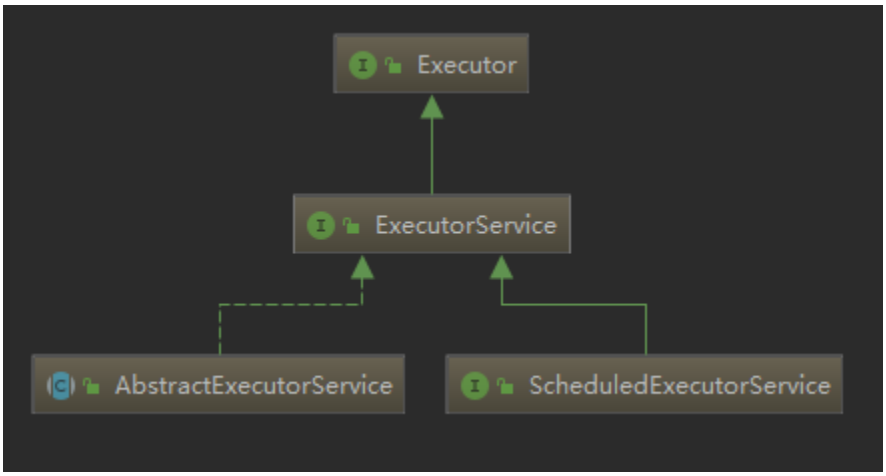
前面我们从整体上介绍了 Java 的线程池。如果细细品味线程池的底层源码实现，你会发现整个线程池体系的设计是非常优雅的！这些代码的设计值得我们去细细品味和研究，从中学习优雅代码的设计规范，形成自己的设计思想，为我所用！哈哈，说多了，接下来，我们就来看看线程池中那些非常重要的接口和抽象类，深度分析下线程池中是如何将抽象这一思想运用的淋漓尽致的！

通过对线程池中接口和抽象类的分析，你会发现，整个线程池设计的是如此的优雅和强大，从线程池的代码设计中，我们学到的不只是代码而已！！

题外话：膜拜 Java 大神 Doug Lea，Java 中的并发包正是这位老爷子写的，他是这个世界上对 Java 影响力最大的一个人。

接口和抽象类总览

说起线程池中提供的重要的接口和抽象类，基本上就是如下图所示的接口和类。



接口与类的简单说明：

- Executor 接口：这个接口也是整个线程池中最顶层的接口，提供了一个无返回值的提交任务的方法。
- ExecutorService 接口：派生自 Executor 接口，扩展了很过功能，例如关闭线程池，提交任务并返回结果数据、唤醒线程池中的任务等。
- AbstractExecutorService 抽象类：派生自 ExecutorService 接口，实现了几个非常实现的方法，供子类进行调用。

- ScheduledExecutorService 定时任务接口，派生自 ExecutorService 接口，拥有 ExecutorService 接口定义的全部方法，并扩展了定时任务相关的方法。

接下来，我们就分别从源码角度来看下这些接口和抽象类从顶层设计上提供了哪些功能。

Executor 接口

Executor 接口的源码如下所示。

```
public interface Executor {
        //提交运行任务，参数为 Runnable 接口对象，无返回值
    void execute(Runnable command);
}
```

从源码可以看出，Executor 接口非常简单，只提供了一个无返回值的提交任务的 execute(Runnable)方法。

由于这个接口过于简单，我们无法得知线程池的执行结果数据，如果我们不再使用线程池，也无法通过 Executor 接口来关闭线程池。此时，我们就需要 ExecutorService 接口的支持了。

ExecutorService 接口

ExecutorService 接口是非定时任务类线程池的核心接口，通过 ExecutorService 接口能够向线程池中提交任务（支持有返回结果和无返回结果两种方式）、关闭线程池、唤醒线程池中的任务等。ExecutorService 接口的源码如下所示。

```
package java.util.concurrent;
import java.util.List;
import java.util.Collection;
public interface ExecutorService extends Executor {

        //关闭线程池，线程池中不再接受新提交的任务，但是之前提交的任务继续运行，直到完成
    void shutdown();

        //关闭线程池，线程池中不再接受新提交的任务，会尝试停止线程池中正在执行的任务。
```

```
    List<Runnable> shutdownNow();

        //判断线程池是否已经关闭
    boolean isShutdown();

        //判断线程池中的所有任务是否结束，只有在调用 shutdown 或者 shutdownNo
w 方法之后调用此方法才会返回 true。
    boolean isTerminated();

        //等待线程池中的所有任务执行结束，并设置超时时间
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;

        //提交一个 Callable 接口类型的任务，返回一个 Future 类型的结果
    <T> Future<T> submit(Callable<T> task);

        //提交一个 Callable 接口类型的任务，并且给定一个泛型类型的接收结果数据参
数，返回一个 Future 类型的结果
    <T> Future<T> submit(Runnable task, T result);

        //提交一个 Runnable 接口类型的任务，返回一个 Future 类型的结果
    Future<?> submit(Runnable task);

        //批量提交任务并获得他们的 future，Task 列表与 Future 列表一一对应
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;

        //批量提交任务并获得他们的 future，并限定处理所有任务的时间
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                  long timeout, TimeUnit unit) throws InterruptedException;

        //批量提交任务并获得一个已经成功执行的任务的结果
    <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

        //批量提交任务并获得一个已经成功执行的任务的结果，并限定处理任务的时间
    <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                    long timeout, TimeUnit unit)
```

```
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

关于 ExecutorService 接口中每个方法的含义，直接上述接口源码中的注释即可，这些接口方法都比较简单，我就不一一重复列举描述了。这个接口也是我们在使用非定时任务类的线程池中最常使用的接口。

AbstractExecutorService 抽象类

AbstractExecutorService 类是一个抽象类，派生自 ExecutorService 接口，在其基础上实现了几个比较实用的方法，提供给子类进行调用。我们还是来看下 AbstractExecutorService 类的源码。

注意：大家可以到 java.util.concurrent 包下查看完整的 AbstractExecutorService 类的源码，这里，我将 AbstractExecutorService 源码进行拆解，详解每个方法的作用。

- newTaskFor 方法

```
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {
        return new FutureTask<T>(runnable, value);
}

protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
        return new FutureTask<T>(callable);
}
```

RunnableFuture 类用于获取执行结果，在实际使用时，我们经常使用的是它的子类 FutureTask，newTaskFor 方法的作用就是将任务封装成 FutureTask 对象，后续将 FutureTask 对象提交到线程池。

- doInvokeAny 方法

```
private <T> T doInvokeAny(Collection<? extends Callable<T>> tasks,
                          boolean timed, long nanos)
    throws InterruptedException, ExecutionException, TimeoutException {
    //提交的任务为空，抛出空指针异常
    if (tasks == null)
        throw new NullPointerException();
    //记录待执行的任务的剩余数量
    int ntasks = tasks.size();
```

```
//任务集合中的数据为空，抛出非法参数异常
if (ntasks == 0)
    throw new IllegalArgumentException();
ArrayList<Future<T>> futures = new ArrayList<Future<T>>(ntasks);
//以当前实例对象作为参数构建 ExecutorCompletionService 对象
// ExecutorCompletionService 负责执行任务，后面调用用 poll 返回第一个执行结果
ExecutorCompletionService<T> ecs =
    new ExecutorCompletionService<T>(this);

try {
    // 记录可能抛出的执行异常
    ExecutionException ee = null;
    // 初始化超时时间
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    Iterator<? extends Callable<T>> it = tasks.iterator();

    //提交任务，并将返回的结果数据添加到 futures 集合中
    //提交一个任务主要是确保在进入循环之前开始一个任务
    futures.add(ecs.submit(it.next()));
    --ntasks;
    //记录正在执行的任务数量
    int active = 1;

    for (;;) {
        //从完成任务的 BlockingQueue 队列中获取并移除下一个将要完成的任务的结果。
        //如果 BlockingQueue 队列中中的数据为空，则返回 null
        //这里的 poll()方法是非阻塞方法
        Future<T> f = ecs.poll();
        //获取的结果为空
        if (f == null) {
            //集合中仍有未执行的任务数量
            if (ntasks > 0) {
                //未执行的任务数量减 1
                --ntasks;
                //提交完成并将结果添加到 futures 集合中
                futures.add(ecs.submit(it.next()));
                //正在执行的任务数量加•1
                ++active;
            }
```

```
            //所有任务执行完成，并且返回了结果数据，则退出循环
            //之所以处理 active 为 0 的情况，是因为 poll()方法是非阻塞方法，可能导致未返回结果时 active 为 0
            else if (active == 0)
                break;
            //如果 timed 为 true，则执行获取结果数据时设置超时时间，也就是超时获取结果表示
            else if (timed) {
                f = ecs.poll(nanos, TimeUnit.NANOSECONDS);
                if (f == null)
                    throw new TimeoutException();
                nanos = deadline - System.nanoTime();
            }
            //没有设置超时，并且所有任务都被提交了，则一直阻塞，直到返回一个执行结果
            else
                f = ecs.take();
        }
        //获取到执行结果，则将正在执行的任务减 1，从 Future 中获取结果并返回
        if (f != null) {
            --active;
            try {
                return f.get();
            } catch (ExecutionException eex) {
                ee = eex;
            } catch (RuntimeException rex) {
                ee = new ExecutionException(rex);
            }
        }
    }

    if (ee == null)
        ee = new ExecutionException();
    throw ee;

  } finally {
    //如果从所有执行的任务中获取到一个结果数据，则取消所有执行的任务，不再向下执行
    for (int i = 0, size = futures.size(); i < size; i++)
```

```
            futures.get(i).cancel(true);
        }
    }
```

这个方法是批量执行线程池的任务，最终返回一个结果数据的核心方法，通过源代码的分析，我们可以发现，这个方法只要获取到一个结果数据，就会取消线程池中所有运行的任务，并将结果数据返回。这就好比是很多要进入一个居民小区一样，只要有一个人有门禁卡，门卫就不再检查其他人是否有门禁卡，直接放行。

在上述代码中，我们看到提交任务使用的 ExecutorCompletionService 对象的 submit 方法，我们再来看下 ExecutorCompletionService 类中的 submit 方法，如下所示。

```
public Future<V> submit(Callable<V> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<V> f = newTaskFor(task);
        executor.execute(new QueueingFuture(f));
        return f;
}

public Future<V> submit(Runnable task, V result) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<V> f = newTaskFor(task, result);
        executor.execute(new QueueingFuture(f));
        return f;
}
```

可以看到，ExecutorCompletionService 类中的 submit 方法本质上调用的还是 Executor 接口的 execute 方法。

- invokeAny 方法

```
public <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException {
        try {
                return doInvokeAny(tasks, false, 0);
        } catch (TimeoutException cannotHappen) {
                assert false;
                return null;
        }
```

```
}

public <T> T invokeAny(Collection<? extends Callable<T>> tasks,
                                    long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException {
        return doInvokeAny(tasks, true, unit.toNanos(timeout));
}
```

这两个 invokeAny 方法本质上都是在调用 doInvokeAny 方法，在线程池中提交多个任务，只要返回一个结果数据即可。

直接看上面的代码，大家可能有点晕。这里，我举一个例子，我们在使用线程池的时候，可能会启动多个线程去执行各自的任务，比如线程 A 负责 task*a*，*线程 B 负责 task*b，这样可以大规模提升系统处理任务的速度。如果我们希望其中一个线程执行完成返回结果数据时立即返回，而不需要再让其他线程继续执行任务。此时，就可以使用 invokeAny 方法。

- invokeAll 方法

```
public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException {
        if (tasks == null)
                throw new NullPointerException();
        ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size());
        //标识所有任务是否完成
        boolean done = false;
        try {
                //遍历所有任务
                for (Callable<T> t : tasks) {
                        将每个任务封装成 RunnableFuture 对象提交任务
                        RunnableFuture<T> f = newTaskFor(t);
                        //将结果数据添加到 futures 集合中
                        futures.add(f);
                        //执行任务
                        execute(f);
                }
                //遍历结果数据集合
                for (int i = 0, size = futures.size(); i < size; i++) {
                        Future<T> f = futures.get(i);
                        //任务没有完成
```

```
                    if (!f.isDone()) {
                        try {
                            //阻塞等待任务完成并返回结果
                            f.get();
                        } catch (CancellationException ignore) {
                        } catch (ExecutionException ignore) {
                        }
                    }
                }
                //任务完成（不管是正常结束还是异常完成）
                done = true;
                //返回结果数据集合
                return futures;
            } finally {
                //如果发生中断异常 InterruptedException 则取消已经提交的任务
                if (!done)
                    for (int i = 0, size = futures.size(); i < size; i++)
                        futures.get(i).cancel(true);
            }
        }

        public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                                             long timeout, Time
        Unit unit)
                throws InterruptedException {
                if (tasks == null)
                    throw new NullPointerException();
                long nanos = unit.toNanos(timeout);
                ArrayList<Future<T>> futures = new ArrayList<Future<T>>(tasks.size());
                boolean done = false;
                try {
                    for (Callable<T> t : tasks)
                        futures.add(newTaskFor(t));

                    final long deadline = System.nanoTime() + nanos;
                    final int size = futures.size();

                    for (int i = 0; i < size; i++) {
                        execute((Runnable)futures.get(i));
```

```
                    // 在添加执行任务时超时判断，如果超时则立刻返回 futures 集合
                    nanos = deadline - System.nanoTime();
                    if (nanos <= 0L)
                        return futures;
                }
                // 遍历所有任务
                for (int i = 0; i < size; i++) {
                    Future<T> f = futures.get(i);
                    if (!f.isDone()) {
                        //对结果进行判断时进行超时判断
                        if (nanos <= 0L)
                            return futures;
                        try {
                            f.get(nanos, TimeUnit.NANOSECONDS);
                        } catch (CancellationException ignore) {
                        } catch (ExecutionException ignore) {
                        } catch (TimeoutException toe) {
                            return futures;
                        }
                        //重置任务的超时时间
                        nanos = deadline - System.nanoTime();
                    }
                }
                done = true;
                return futures;
            } finally {
                if (!done)
                    for (int i = 0, size = futures.size(); i < size; i++)
                        futures.get(i).cancel(true);
            }
        }
```

invokeAll 方法同样实现了无超时时间设置和有超时时间设置的逻辑。

无超时时间设置的 invokeAll 方法总体逻辑为：将所有任务封装成 RunnableFuture 对象，调用 execute 方法执行任务，将返回的结果数据添加到 futures 集合，之后对 futures 集合进行遍历判断，检测任务是否完成，如果没有完成，则调用 get 方法阻塞任务，直到返回结果数据，此时会忽略异常。最终在

finally 代码块中对所有任务是否完成的标识进行判断，如果存在未完成的任务，则取消已经提交的任务。

有超时设置的 invokeAll 方法总体逻辑与无超时时间设置的 invokeAll 方法总体逻辑基本相同，只是在两个地方添加了超时的逻辑判断。一个是在添加执行任务时进行超时判断，如果超时，则立刻返回 futures 集合；另一个是每次对结果数据进行判断时添加了超时处理逻辑。

invokeAll 方法中本质上还是调用 Executor 接口的 execute 方法来提交任务。

- submit 方法

submit 方法的逻辑比较简单，就是将任务封装成 RunnableFuture 对象并提交，执行任务后返回 Future 结果数据。如下所示。

```
public Future<?> submit(Runnable task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<Void> ftask = newTaskFor(task, null);
        execute(ftask);
        return ftask;
}

public <T> Future<T> submit(Runnable task, T result) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task, result);
        execute(ftask);
        return ftask;
}

public <T> Future<T> submit(Callable<T> task) {
        if (task == null) throw new NullPointerException();
        RunnableFuture<T> ftask = newTaskFor(task);
        execute(ftask);
        return ftask;
}
```

从源码中可以看出 submit 方法提交任务时，本质上还是调用的 Executor 接口的 execute 方法。

综上所述，在非定时任务类的线程池中提交任务时，本质上都是调用的 Executor 接口的 execute 方法。至于调用的是哪个具体实现类的 execute 方法，我们在后面的文章中深入分析。

ScheduledExecutorService 接口

ScheduledExecutorService 接口派生自 ExecutorService 接口，继承了 ExecutorService 接口的所有功能，并提供了定时处理任务的能力，ScheduledExecutorService 接口的源代码比较简单，如下所示。

```
package java.util.concurrent;

public interface ScheduledExecutorService extends ExecutorService {

    //延时 delay 时间来执行 command 任务，只执行一次
    public ScheduledFuture<?> schedule(Runnable command,
                                       long delay, TimeUnit unit);

        //延时 delay 时间来执行 callable 任务，只执行一次
    public <V> ScheduledFuture<V> schedule(Callable<V> callable,
                                           long delay, TimeUnit unit);

        //延时 initialDelay 时间首次执行 command 任务，之后每隔 period 时间执行一次
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                  long initialDelay,
                                                  long period,
                                                  TimeUnit unit);


        //延时 initialDelay 时间首次执行 command 任务，之后每延时 delay 时间执行一次
    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
                                                     long initialDelay,
                                                     long delay,
                                                     TimeUnit unit);

}
```

至此，我们分析了线程池体系中重要的顶层接口和抽象类。

通过对这些顶层接口和抽象类的分析，我们需要从中感悟并体会软件开发中的抽象思维，深入理解抽象思维在具体编码中的实现，最终，形成自己的编程思维，运用到实际的项目中，这也是我们能够从源码中所能学到的众多细节之一。这也是高级或资深工程师和架构师必须了解源码细节的原因之一。

八、从源码角度分析创建线程池究竟有哪些方式

前言

在 Java 的高并发领域，线程池一直是一个绕不开的话题。有些童鞋一直在使用线程池，但是，对于如何创建线程池仅仅停留在使用 Executors 工具类的方式，那么，创建线程池究竟存在哪几种方式呢？就让我们一起从创建线程池的源码来深入分析究竟有哪些方式可以创建线程池。

使用 Executors 工具类创建线程池

在创建线程池时，初学者用的最多的就是 Executors 这个工具类，而使用这个工具类创建线程池时非常简单的，不需要关注太多的线程池细节，只需要传入必要的参数即可。Executors 工具类提供了几种创建线程池的方法，如下所示。

- Executors.newCachedThreadPool：创建一个可缓存的线程池，如果线程池的大小超过了需要，可以灵活回收空闲线程，如果没有可回收线程，则新建线程
- Executors.newFixedThreadPool：创建一个定长的线程池，可以控制线程的最大并发数，超出的线程会在队列中等待
- Executors.newScheduledThreadPool：创建一个定长的线程池，支持定时、周期性的任务执行
- Executors.newSingleThreadExecutor: 创建一个单线程化的线程池，使用一个唯一的工作线程执行任务，保证所有任务按照指定顺序（先入先出或者优先级）执行
- Executors.newSingleThreadScheduledExecutor:创建一个单线程化的线程池，支持定时、周期性的任务执行
- Executors.newWorkStealingPool：创建一个具有并行级别的 work-stealing 线程池

其中，Executors.newWorkStealingPool 方法是 Java 8 中新增的创建线程池的方法，它能够为线程池设置并行级别，具有更高的并发度和性能。除了此方法外，其他创建线程池的方法本质上调用的是 ThreadPoolExecutor 类的构造方法。

例如，我们可以使用如下代码创建线程池。

```
Executors.newWorkStealingPool();
Executors.newCachedThreadPool();
Executors.newScheduledThreadPool(3);
```

使用 ThreadPoolExecutor 类创建线程池

从代码结构上看 ThreadPoolExecutor 类继承自 AbstractExecutorService，也就是说，ThreadPoolExecutor 类具有 AbstractExecutorService 类的全部功能。

既然 Executors 工具类中创建线程池大部分调用的都是 ThreadPoolExecutor 类的构造方法，所以，我们也可以直接调用 ThreadPoolExecutor 类的构造方法来创建线程池，而不再使用 Executors 工具类。接下来，我们一起看下 ThreadPoolExecutor 类的构造方法。

ThreadPoolExecutor 类中的所有构造方法如下所示。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue) {
        this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
                Executors.defaultThreadFactory(), defaultHandler);
}

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory) {
this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        threadFactory, defaultHandler);
}

public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
```

```
                                BlockingQueue<Runnable> workQueue,
                                RejectedExecutionHandler handler) {
this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        Executors.defaultThreadFactory(), handler);
}

public ThreadPoolExecutor(int corePoolSize,
                                int maximumPoolSize,
                                long keepAliveTime,
                                TimeUnit unit,
                              BlockingQueue<Runnable> workQueue,
                                ThreadFactory threadFactory,
                                RejectedExecutionHandler handler) {
        if (corePoolSize < 0 ||
                maximumPoolSize <= 0 ||
                maximumPoolSize < corePoolSize ||
                keepAliveTime < 0)
                throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
                throw new NullPointerException();
        this.acc = System.getSecurityManager() == null ?
                        null :
                        AccessController.getContext();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
}
```

由 ThreadPoolExecutor 类的构造方法的源代码可知，创建线程池最终调用的构造方法如下。

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

```
        if (corePoolSize < 0 ||
                maximumPoolSize <= 0 ||
                maximumPoolSize < corePoolSize ||
                keepAliveTime < 0)
                throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
                throw new NullPointerException();
        this.acc = System.getSecurityManager() == null ?
                        null :
                        AccessController.getContext();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
}
```

大家可以自行调用 ThreadPoolExecutor 类的构造方法来创建线程池。例如，我们可以使用如下形式创建线程池。

```
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                60L, TimeUnit.SECONDS,
                new SynchronousQueue<Runnable>());
```

使用 ForkJoinPool 类创建线程池

在 Java8 的 Executors 工具类中，新增了如下创建线程池的方式。

```
public static ExecutorService newWorkStealingPool(int parallelism) {
        return new ForkJoinPool
                (parallelism,
                 ForkJoinPool.defaultForkJoinWorkerThreadFactory,
                 null, true);
}

public static ExecutorService newWorkStealingPool() {
        return new ForkJoinPool
                (Runtime.getRuntime().availableProcessors(),
                 ForkJoinPool.defaultForkJoinWorkerThreadFactory,
```

```
                null, true);
}
```

从源代码可以可以，本质上调用的是 ForkJoinPool 类的构造方法类创建线程池，而从代码结构上来看 ForkJoinPool 类继承自 AbstractExecutorService 抽象类。接下来，我们看下 ForkJoinPool 类的构造方法。

```
public ForkJoinPool() {
        this(Math.min(MAX_CAP, Runtime.getRuntime().availableProcessors()),
                defaultForkJoinWorkerThreadFactory, null, false);
}
 public ForkJoinPool(int parallelism) {
        this(parallelism, defaultForkJoinWorkerThreadFactory, null, false);
}

public ForkJoinPool(int parallelism,
                                ForkJoinWorkerThreadFactory factory,
                                UncaughtExceptionHandler handler,
                                boolean asyncMode) {
        this(checkParallelism(parallelism),
                checkFactory(factory),
                handler,
                asyncMode ? FIFO_QUEUE : LIFO_QUEUE,
                "ForkJoinPool-" + nextPoolId() + "-worker-");
        checkPermission();
}

private ForkJoinPool(int parallelism,
                                ForkJoinWorkerThreadFactory factory,
                                UncaughtExceptionHandler handler,
                                int mode,
                                String workerNamePrefix) {
        this.workerNamePrefix = workerNamePrefix;
        this.factory = factory;
        this.ueh = handler;
        this.config = (parallelism & SMASK) | mode;
        long np = (long)(-parallelism); // offset ctl counts
        this.ctl = ((np << AC_SHIFT) & AC_MASK) | ((np << TC_SHIFT) & TC_M
```

```
ASK);
}
```

通过查看源代码得知，ForkJoinPool 的构造方法，最终调用的是如下私有构造方法。

```
private ForkJoinPool(int parallelism,
                     ForkJoinWorkerThreadFactory factory,
                     UncaughtExceptionHandler handler,
                     int mode,
                     String workerNamePrefix) {
        this.workerNamePrefix = workerNamePrefix;
        this.factory = factory;
        this.ueh = handler;
        this.config = (parallelism & SMASK) | mode;
        long np = (long)(-parallelism); // offset ctl counts
        this.ctl = ((np << AC_SHIFT) & AC_MASK) | ((np << TC_SHIFT) & TC_M
ASK);
}
```

其中，各参数的含义如下所示。

- parallelism：并发级别。
- factory：创建线程的工厂类对象。
- handler：当线程池中的线程抛出未捕获的异常时，统一使用 UncaughtExceptionHandler 对象处理。
- mode：取值为 FIFO*QUEUE 或者 LIFO*QUEUE。
- workerNamePrefix：执行任务的线程名称的前缀。

当然，私有构造方法虽然是参数最多的一个方法，但是其不会直接对外方法，我们可以使用如下方式创建线程池。

```
new ForkJoinPool();
new ForkJoinPool(Runtime.getRuntime().availableProcessors());
new ForkJoinPool(Runtime.getRuntime().availableProcessors(),
        ForkJoinPool.defaultForkJoinWorkerThreadFactory,
        null, true);
```

使用 ScheduledThreadPoolExecutor 类创建线程池

在 Executors 工具类中存在如下方法类创建线程池。

```
public static ScheduledExecutorService newSingleThreadScheduledExecutor()
{
        return new DelegatedScheduledExecutorService
                (new ScheduledThreadPoolExecutor(1));
}

public static ScheduledExecutorService newSingleThreadScheduledExecutor(T
hreadFactory threadFactory) {
        return new DelegatedScheduledExecutorService
                (new ScheduledThreadPoolExecutor(1, threadFactory));
}

public static ScheduledExecutorService newScheduledThreadPool(int corePool
Size) {
        return new ScheduledThreadPoolExecutor(corePoolSize);
}

public static ScheduledExecutorService newScheduledThreadPool(
                int corePoolSize, ThreadFactory threadFactory) {
       return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
}
```

从源码来看，这几个方法本质上调用的都是 ScheduledThreadPoolExecutor 类的构造方法，ScheduledThreadPoolExecutor 中存在的构造方法如下所示。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue());
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadF
actory) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), threadFactory);
}
```

```
public ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHan
dler handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), handler);
}

public ScheduledThreadPoolExecutor(int corePoolSize,ThreadFactory threadF
actory, RejectedExecutionHandler handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), threadFactory, handler);
}
```

ThreadPoolExecutor 类继承自 ThreadPoolExecutor 类，本质上还是调用 ThreadPoolExecutor 类的构造方法，只不过此时传递的队列为 DelayedWorkQueue。我们可以直接调用 ScheduledThreadPoolExecutor 类的构造方法来创建线程池，例如以如下形式创建线程池。

```
new ScheduledThreadPoolExecutor(3)
```

九、通过源码深度解析 ThreadPoolExecutor 类

问题：

对于线程池的核心类 ThreadPoolExecutor 来说，有哪些重要的属性和内部类为线程池的正确运行提供重要的保障呢?

ThreadPoolExecutor 类中的重要属性

在 ThreadPoolExecutor 类中，存在几个非常重要的属性和方法，接下来，我们就介绍下这些重要的属性和方法。

ctl 相关的属性

AtomicInteger 类型的常量 ctl 是贯穿线程池整个生命周期的重要属性，它是一个原子类对象，主要用来保存线程的数量和线程池的状态，我们看下与这个属性相关的代码如下所示。

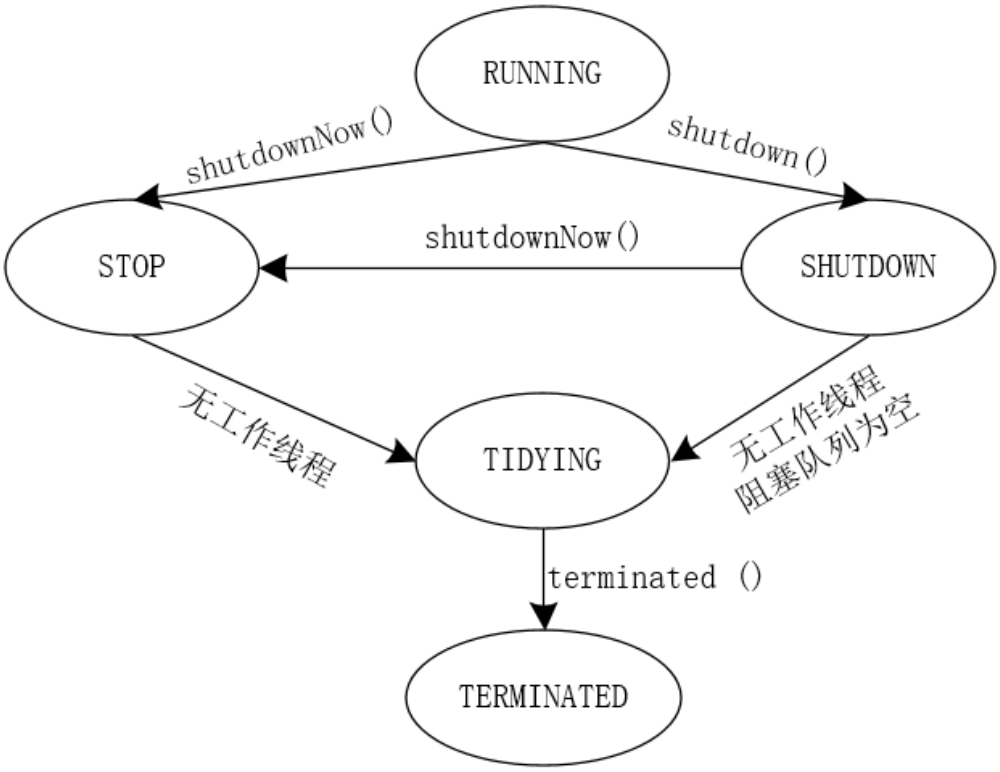
```
//主要用来保存线程数量和线程池的状态，高 3 位保存线程状态，低 29 位保存线程数量
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
//线程池中线程的数量的位数（32-3）
private static final int COUNT_BITS = Integer.SIZE - 3;
//表示线程池中的最大线程数量
//将数字 1 的二进制值向右移 29 位，再减去 1
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
//线程池的运行状态
private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;
//获取线程状态
private static int runStateOf(int c)     { return c & ~CAPACITY; }
//获取线程数量
private static int workerCountOf(int c)  { return c & CAPACITY; }
private static int ctlOf(int rs, int wc) { return rs | wc; }
private static boolean runStateLessThan(int c, int s) {
        return c < s;
}
private static boolean runStateAtLeast(int c, int s) {
        return c >= s;
```

```
}
private static boolean isRunning(int c) {
        return c < SHUTDOWN;
}
private boolean compareAndIncrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect + 1);
}
private boolean compareAndDecrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect - 1);
}
private void decrementWorkerCount() {
        do {} while (! compareAndDecrementWorkerCount(ctl.get()));
}
```

对于线程池的各状态说明如下所示。

- RUNNING:运行状态，能接收新提交的任务，并且也能处理阻塞队列中的任务
- SHUTDOWN: 关闭状态，不能再接收新提交的任务，但是可以处理阻塞队列中已经保存的任务，当线程池处于 RUNNING 状态时，调用 shutdown() 方法会使线程池进入该状态
- STOP: 不能接收新任务，也不能处理阻塞队列中已经保存的任务，会中断正在处理任务的线程，如果线程池处于 RUNNING 或 SHUTDOWN 状态，调用 shutdownNow()方法，会使线程池进入该状态
- TIDYING: 如果所有的任务都已经终止，有效线程数为 0（阻塞队列为空，线程池中的工作线程数量为 0），线程池就会进入该状态。
- TERMINATED: 处于 TIDYING 状态的线程池调用 terminated ()方法，会使用线程池进入该状态

也可以按照 ThreadPoolExecutor 类的注释，将线程池的各状态之间的转化总结成如下图所示。



- RUNNING -> SHUTDOWN：显式调用 shutdown()方法, 或者隐式调用了 finalize()方法
- (RUNNING or SHUTDOWN) -> STOP：显式调用 shutdownNow()方法
- SHUTDOWN -> TIDYING：当线程池和任务队列都为空的时候
- STOP -> TIDYING：当线程池为空的时候
- TIDYING -> TERMINATED：当 terminated() hook 方法执行完成时候

其他重要属性

除了 ctl 相关的属性外，ThreadPoolExecutor 类中其他一些重要的属性如下所示。

```
//用于存放任务的阻塞队列
private final BlockingQueue<Runnable> workQueue;
//可重入锁
private final ReentrantLock mainLock = new ReentrantLock();
//存放线程池中线程的集合，访问这个集合时，必须获得 mainLock 锁
```

```
private final HashSet<Worker> workers = new HashSet<Worker>();
//在锁内部阻塞等待条件完成
private final Condition termination = mainLock.newCondition();
//线程工厂，以此来创建新线程
private volatile ThreadFactory threadFactory;
//拒绝策略
private volatile RejectedExecutionHandler handler;
//默认的拒绝策略
private static final RejectedExecutionHandler defaultHandler = new AbortPolicy
();
```

ThreadPoolExecutor 类中的重要内部类

在 ThreadPoolExecutor 类中存在对于线程池的执行至关重要的内部类，Worker 内部类和拒绝策略内部类。接下来，我们分别看这些内部类。

Worker 内部类

Worker 类从源代码上来看，实现了 Runnable 接口，说明其本质上是一个用来执行任务的线程，接下来，我们看下 Worker 类的源代码，如下所示。

```
private final class Worker extends AbstractQueuedSynchronizer implements R
unnable{
        private static final long serialVersionUID = 6138294804551838833L;
        //真正执行任务的线程
        final Thread thread;
        //第一个 Runnable 任务，如果在创建线程时指定了需要执行的第一个任务
        //则第一个任务会存放在此变量中，此变量也可以为 null
        //如果为 null，则线程启动后，通过 getTask 方法到 BlockingQueue 队列中获
取任务
        Runnable firstTask;
        //用于存放此线程完全的任务数，注意：使用了 volatile 关键字
        volatile long completedTasks;

        //Worker 类唯一的构造放大，传递的 firstTask 可以为 null
        Worker(Runnable firstTask) {
                //防止在调用 runWorker 之前被中断
                setState(-1);
                this.firstTask = firstTask;
                //使用 ThreadFactory 来创建一个新的执行任务的线程
```

```
        this.thread = getThreadFactory().newThread(this);
    }
    //调用外部 ThreadPoolExecutor 类的 runWorker 方法执行任务
    public void run() {
        runWorker(this);
    }

    //是否获取到锁
    //state=0 表示锁未被获取
    //state=1 表示锁被获取
    protected boolean isHeldExclusively() {
        return getState() != 0;
    }

    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }

    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }

    public void lock()        { acquire(1); }
    public boolean tryLock()  { return tryAcquire(1); }
    public void unlock()      { release(1); }
    public boolean isLocked() { return isHeldExclusively(); }

    void interruptIfStarted() {
        Thread t;
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
```

```
                    } catch (SecurityException ignore) {
                    }
                }
            }
        }
```

在 Worker 类的构造方法中，可以看出，首先将同步状态 state 设置为-1，设置为-1 是为了防止 runWorker 方法运行之前被中断。这是因为如果其他线程调用线程池的 shutdownNow()方法时，如果 Worker 类中的 state 状态的值大于 0，则会中断线程，如果 state 状态的值为-1，则不会中断线程。

Worker 类实现了 Runnable 接口，需要重写 run 方法，而 Worker 的 run 方法本质上调用的是 ThreadPoolExecutor 类的 runWorker 方法，在 runWorker 方法中，会首先调用 unlock 方法，该方法会将 state 置为 0，所以这个时候调用 shutDownNow 方法就会中断当前线程，而这个时候已经进入了 runWork 方法，就不会在还没有执行 runWorker 方法的时候就中断线程。

注意：大家需要重点理解 Worker 类的实现

拒绝策略内部类

在线程池中，如果 workQueue 阻塞队列满了，并且没有空闲的线程池，此时，继续提交任务，需要采取一种策略来处理这个任务。而线程池总共提供了四种策略，如下所示。

- 直接抛出异常，这也是默认的策略。实现类为 AbortPolicy。
- 用调用者所在的线程来执行任务。实现类为 CallerRunsPolicy。
- 丢弃队列中最靠前的任务并执行当前任务。实现类为 DiscardOldestPolicy。
- 直接丢弃当前任务。实现类为 DiscardPolicy。

在 ThreadPoolExecutor 类中提供了 4 个内部类来默认实现对应的策略，如下所示。

```
public static class CallerRunsPolicy implements RejectedExecutionHandler {

    public CallerRunsPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
```

```
                if (!e.isShutdown()) {
                        r.run();
                }
        }
}

public static class AbortPolicy implements RejectedExecutionHandler {

        public AbortPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
                throw new RejectedExecutionException("Task " + r.toString() + " r
ejected from " + e.toString());
        }
}

public static class DiscardPolicy implements RejectedExecutionHandler {

        public DiscardPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        }
}

public static class DiscardOldestPolicy implements RejectedExecutionHandler {

        public DiscardOldestPolicy() { }


        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
                if (!e.isShutdown()) {
                        e.getQueue().poll();
                        e.execute(r);
                }
        }
}
```

我们也可以通过实现 RejectedExecutionHandler 接口，并重写 RejectedExecutionHandler 接口的 rejectedExecution 方法来自定义拒绝策略，在创建线程池时，调用 ThreadPoolExecutor 的构造方法，传入我们自己写的拒绝策略。

例如，自定义的拒绝策略如下所示。

```
public class CustomPolicy implements RejectedExecutionHandler {

        public CustomPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
                if (!e.isShutdown()) {
                        System.out.println("使用调用者所在的线程来执行任务")
                        r.run();
                }
        }
}
```

使用自定义拒绝策略创建线程池。

```
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                60L, TimeUnit.SECONDS,
                new SynchronousQueue<Runnable>(),
                Executors.defaultThreadFactory(),
                     new CustomPolicy());
```

十、通过 ThreadPoolExecutor 类的源码深度解析线程池执行任务

核心逻辑概述

ThreadPoolExecutor 是 Java 线程池中最核心的类之一，它能够保证线程池按照正常的业务逻辑执行任务，并通过原子方式更新线程池每个阶段的状态。

ThreadPoolExecutor 类中存在一个 workers 工作线程集合，用户可以向线程池中添加需要执行的任务，workers 集合中的工作线程可以直接执行任务，或者从任务队列中获取任务后执行。ThreadPoolExecutor 类中提供了整个线程池从创建到执行任务，再到消亡的整个流程方法。本文，就结合 ThreadPoolExecutor 类的源码深度分析线程池执行任务的整体流程。

在 ThreadPoolExecutor 类中，线程池的逻辑主要体现在 execute(Runnable)方法，addWorker(Runnable, boolean)方法，addWorkerFailed(Worker)方法和拒绝策略上，接下来，我们就深入分析这几个核心方法。

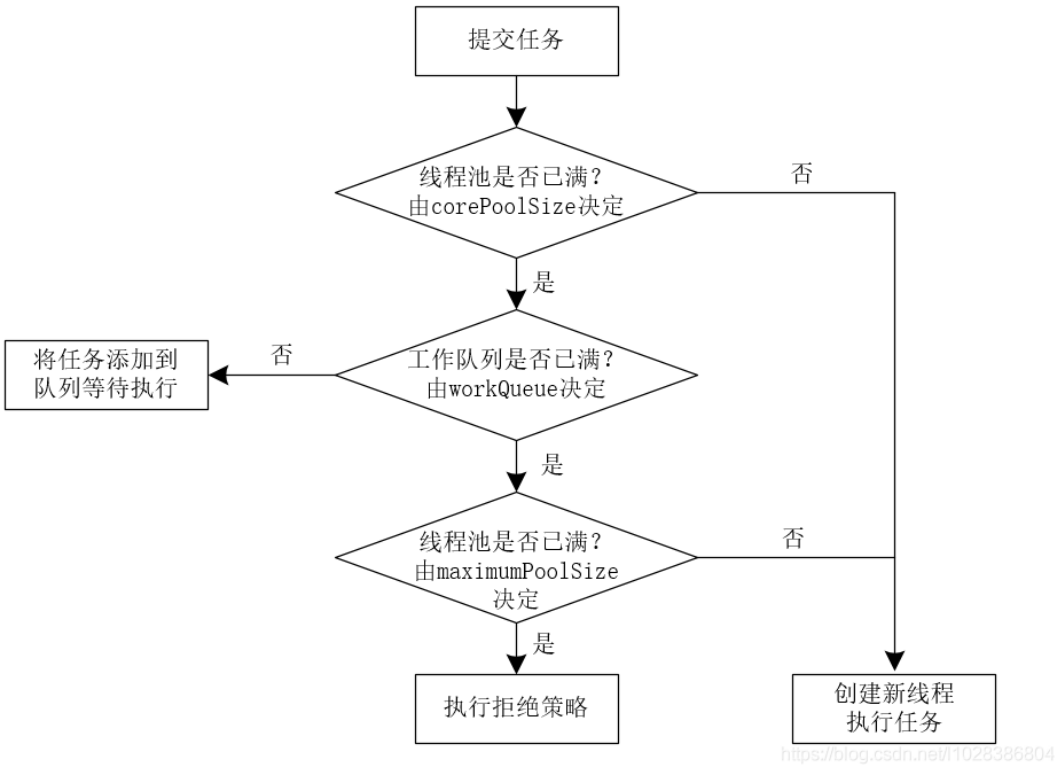
execute(Runnable)方法

execute(Runnable)方法的作用是提交 Runnable 类型的任务到线程池中。我们先看下 execute(Runnable)方法的源码，如下所示。

```
public void execute(Runnable command) {
        //如果提交的任务为空，则抛出空指针异常
        if (command == null)
                throw new NullPointerException();
        //获取线程池的状态和线程池中线程的数量
        int c = ctl.get();
        //线程池中的线程数量小于 corePoolSize 的值
        if (workerCountOf(c) < corePoolSize) {
                //重新开启线程执行任务
                if (addWorker(command, true))
                        return;
                c = ctl.get();
        }
        //如果线程池处于 RUNNING 状态，则将任务添加到阻塞队列中
        if (isRunning(c) && workQueue.offer(command)) {
                //再次获取线程池的状态和线程池中线程的数量，用于二次检查
                int recheck = ctl.get();
                //如果线程池没有未处于 RUNNING 状态，从队列中删除任务
```

```
                if (! isRunning(recheck) && remove(command))
                        //执行拒绝策略
                        reject(command);
                //如果线程池为空，则向线程池中添加一个线程
                else if (workerCountOf(recheck) == 0)
                        addWorker(null, false);
        }
        //任务队列已满，则新增 worker 线程，如果新增线程失败，则执行拒绝策略
        else if (!addWorker(command, false))
                reject(command);
}
```

整个任务的执行流程，我们可以简化成下图所示。



接下来，我们拆解 execute(Runnable)方法，具体分析 execute(Runnable)方法的执行逻辑。

（1）线程池中的线程数是否小于 corePoolSize 核心线程数，如果小于 corePoolSize 核心线程数，则向 workers 工作线程集合中添加一个核心线程执行任务。代码如下所示。

```
//线程池中的线程数量小于 corePoolSize 的值
if (workerCountOf(c) < corePoolSize) {
        //重新开启线程执行任务
        if (addWorker(command, true))
                return;
        c = ctl.get();
}
```

（2）如果线程池中的线程数量大于 corePoolSize 核心线程数，则判断当前线程池是否处于 RUNNING 状态，如果处于 RUNNING 状态，则添加任务到待执行的任务队列中。注意：这里向任务队列添加任务时，需要判断线程池是否处于 RUNNING 状态，只有线程池处于 RUNNING 状态时，才能向任务队列添加新任务。否则，会执行拒绝策略。代码如下所示。

```
if (isRunning(c) && workQueue.offer(command))
```

（3）向任务队列中添加任务成功，由于其他线程可能会修改线程池的状态，所以这里需要对线程池进行二次检查，如果当前线程池的状态不再是 RUNNING 状态，则需要将添加的任务从任务队列中移除，执行后续的拒绝策略。如果当前线程池仍然处于 RUNNING 状态，则判断线程池是否为空，如果线程池中不存在任何线程，则新建一个线程添加到线程池中，如下所示。

```
//再次获取线程池的状态和线程池中线程的数量，用于二次检查
int recheck = ctl.get();
//如果线程池没有未处于 RUNNING 状态，从队列中删除任务
if (! isRunning(recheck) && remove(command))
        //执行拒绝策略
        reject(command);
//如果线程池为空，则向线程池中添加一个线程
else if (workerCountOf(recheck) == 0)
        addWorker(null, false);
```

（4）如果在步骤（3）中向任务队列中添加任务失败，则尝试开启新的线程执行任务。此时，如果线程池中的线程数量已经大于线程池中的最大线程数 maximumPoolSize，则不能再启动新线程。此时，表示线程池中的任务队列已满，并且线程池中的线程已满，需要执行拒绝策略，代码如下所示。

```
//任务队列已满，则新增 worker 线程，如果新增线程失败，则执行拒绝策略
else if (!addWorker(command, false))
        reject(command);
```

这里，我们将 execute(Runnable)方法拆解，结合流程图来理解线程池中任务的执行流程就比较简单了。可以这么说，execute(Runnable)方法的逻辑基本上就是一般线程池的执行逻辑，理解了 execute(Runnable)方法，就基本理解了线程池的执行逻辑。

注意：有关 ScheduledThreadPoolExecutor 类和 ForkJoinPool 类执行线程池的逻辑，在【高并发专题】系列文章中的后文中会详细说明，理解了这些类的执行逻辑，就基本全面掌握了线程池的执行流程。**

在分析 execute(Runnable)方法的源码时，我们发现 execute(Runnable)方法中多处调用了 addWorker(Runnable, boolean)方法，接下来，我们就一起分析下 addWorker(Runnable, boolean)方法的逻辑。

addWorker(Runnable, boolean)方法

总体上，addWorker(Runnable, boolean)方法可以分为三部分，第一部分是使用 CAS 安全的向线程池中添加工作线程；第二部分是创建新的工作线程；第三部分则是将任务通过安全的并发方式添加到 workers 中，并启动工作线程执行任务。

接下来，我们看下 addWorker(Runnable, boolean)方法的源码，如下所示。

```
private boolean addWorker(Runnable firstTask, boolean core) {
        //标记重试的标识
        retry:
        for (;;) {
                int c = ctl.get();
                int rs = runStateOf(c);

                // 检查队列是否在某些特定的条件下为空
                if (rs >= SHUTDOWN &&
                        ! (rs == SHUTDOWN &&
                          firstTask == null &&
                          ! workQueue.isEmpty()))
                        return false;
                //下面循环的主要作用为通过 CAS 方式增加线程的个数
                for (;;) {
```

```
                    //获取线程池中的线程数量
                    int wc = workerCountOf(c);
                    //如果线程池中的线程数量超出限制，直接返回 false
                    if (wc >= CAPACITY ||
                            wc >= (core ? corePoolSize : maximumPoolSize))
                            return false;
                    //通过 CAS 方式向线程池新增线程数量
                    if (compareAndIncrementWorkerCount(c))
                            //通过 CAS 方式保证只有一个线程执行成功，跳出最外层
循环
                            break retry;
                    //重新获取 ctl 的值
                    c = ctl.get();
                    //如果 CAS 操作失败了，则需要在内循环中重新尝试通过 CAS 新
增线程数量
                    if (runStateOf(c) != rs)
                            continue retry;
            }
        }

        //跳出最外层 for 循环，说明通过 CAS 新增线程数量成功
        //此时创建新的工作线程
        boolean workerStarted = false;
        boolean workerAdded = false;
        Worker w = null;
        try {
            //将执行的任务封装成 worker
            w = new Worker(firstTask);
            final Thread t = w.thread;
            if (t != null) {
                //独占锁，保证操作 workers 时的同步
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                    //此处需要重新检查线程池状态
                    //原因是在获得锁之前可能其他的线程改变了线程池的状态
                    int rs = runStateOf(ctl.get());

                    if (rs < SHUTDOWN ||
```

```
                                (rs == SHUTDOWN && firstTask == null)) {
                                if (t.isAlive())
                                        throw new IllegalThreadStateException
();
                                //向 worker 中添加新任务
                                workers.add(w);
                                int s = workers.size();
                                if (s > largestPoolSize)
                                        largestPoolSize = s;
                                //将是否添加了新任务的标识设置为 true
                                workerAdded = true;
                        }
                } finally {
                        //释放独占锁
                        mainLock.unlock();
                }
                //添加新任成功，则启动线程执行任务
                if (workerAdded) {
                        t.start();
                        //将任务是否已经启动的标识设置为 true
                        workerStarted = true;
                }
        }
    } finally {
        //如果任务未启动或启动失败，则调用 addWorkerFailed(Worker)方法
        if (! workerStarted)
                addWorkerFailed(w);
    }
    //返回是否启动任务的标识
    return workerStarted;
}
```

乍一看，addWorker(Runnable, boolean)方法还蛮长的，这里，我们还是将addWorker(Runnable, boolean)方法进行拆解。

（1）检查任务队列是否在某些特定的条件下为空，代码如下所示。

```
// 检查队列是否在某些特定的条件下为空
if (rs >= SHUTDOWN &&
    ! (rs == SHUTDOWN &&
```

```
            firstTask == null &&
            ! workQueue.isEmpty()))
        return false;
```

（2）在通过步骤（1）的校验后，则进入内层 for 循环，在内层 for 循环中通过 CAS 来增加线程池中的线程数量，如果 CAS 操作成功，则直接退出双重 for 循环。如果 CAS 操作失败，则查看当前线程池的状态是否发生了变化，如果线程池的状态发生了变化，则通过 continue 关键字重新通过外层 for 循环校验任务队列，检验通过再次执行内层 for 循环的 CAS 操作。如果线程池的状态没有发生变化，此时上一次 CAS 操作失败了，则继续尝试 CAS 操作。代码如下所示。

```
for (;;) {
        //获取线程池中的线程数量
        int wc = workerCountOf(c);
        //如果线程池中的线程数量超出限制，直接返回 false
        if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
        //通过 CAS 方式向线程池新增线程数量
        if (compareAndIncrementWorkerCount(c))
                //通过 CAS 方式保证只有一个线程执行成功，跳出最外层循环
                break retry;
        //重新获取 ctl 的值
        c = ctl.get();
        //如果 CAS 操作失败了，则需要在内循环中重新尝试通过 CAS 新增线程数量
        if (runStateOf(c) != rs)
                continue retry;
}
```

（3）CAS 操作成功后，表示向线程池中成功添加了工作线程，此时，还没有线程去执行任务。使用全局的独占锁 mainLock 来将新增的工作线程 Worker 对象安全的添加到 workers 中。

总体逻辑就是：创建新的 Worker 对象，并获取 Worker 对象中的执行线程，如果线程不为空，则获取独占锁，获取锁成功后，再次检查线线程的状态，这是避免在获取独占锁之前其他线程修改了线程池的状态，或者关闭了线程池。如果线程池关闭，则需要释放锁。否则将新增加的线程添加到工作集合中，释放锁并启动线程执行任务。将是否启动线程的标识设置为 true。最后，判断线程是否启动，如果

没有启动，则调用 addWorkerFailed(Worker)方法。最终返回线程是否起送的标识。

```
//跳出最外层 for 循环，说明通过 CAS 新增线程数量成功
//此时创建新的工作线程
boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
        //将执行的任务封装成 worker
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
                //独占锁，保证操作 workers 时的同步
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                        //此处需要重新检查线程池状态
                        //原因是在获得锁之前可能其他的线程改变了线程池的状态
                        int rs = runStateOf(ctl.get());

                        if (rs < SHUTDOWN ||
                                (rs == SHUTDOWN && firstTask == null)) {
                                if (t.isAlive())
                                        throw new IllegalThreadStateException();
                                //向 worker 中添加新任务
                                workers.add(w);
                                int s = workers.size();
                                if (s > largestPoolSize)
                                        largestPoolSize = s;
                                //将是否添加了新任务的标识设置为 true
                                workerAdded = true;
                        }
                } finally {
                        //释放独占锁
                        mainLock.unlock();
                }
                //添加新任成功，则启动线程执行任务
                if (workerAdded) {
```

```
                    t.start();
                    //将任务是否已经启动的标识设置为 true
                    workerStarted = true;
                }
            }
        } finally {
            //如果任务未启动或启动失败，则调用 addWorkerFailed(Worker)方法
            if (! workerStarted)
                addWorkerFailed(w);
        }
        //返回是否启动任务的标识
        return workerStarted;
```

addWorkerFailed(Worker)方法

在 addWorker(Runnable, boolean)方法中，如果添加工作线程失败或者工作线程启动失败时，则会调用 addWorkerFailed(Worker)方法，下面我们就来看看 addWorkerFailed(Worker)方法的实现，如下所示。

```
private void addWorkerFailed(Worker w) {
        //获取独占锁
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //如果 Worker 任务不为空
            if (w != null)
                //将任务从 workers 集合中移除
                workers.remove(w);
            //通过 CAS 将任务数量减 1
            decrementWorkerCount();
            tryTerminate();
        } finally {
            //释放锁
            mainLock.unlock();
        }
    }
```

addWorkerFailed(Worker)方法的逻辑就比较简单了，获取独占锁，将任务从 workers 中移除，并且通过 CAS 将任务的数量减 1，最后释放锁。

拒绝策略

我们在分析 execute(Runnable)方法时，线程池会在适当的时候调用 reject(Runnable)方法来执行相应的拒绝策略，我们看下 reject(Runnable)方法的实现，如下所示。

```
final void reject(Runnable command) {
        handler.rejectedExecution(command, this);
}
```

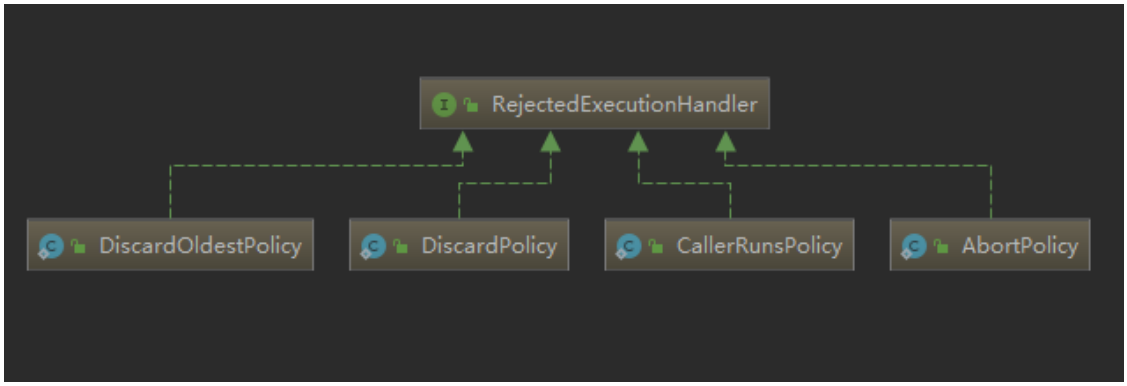
通过代码，我们发现调用的是 handler 的 rejectedExecution 方法，handler 又是个什么鬼，我们继续跟进代码，如下所示。

```
private volatile RejectedExecutionHandler handler;
```

再看看 RejectedExecutionHandler 是个啥类型，如下所示。

```
package java.util.concurrent;

public interface RejectedExecutionHandler {

   void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

可以发现 RejectedExecutionHandler 是个接口，定义了一个 rejectedExecution(Runnable, ThreadPoolExecutor)方法。既然 RejectedExecutionHandler 是个接口，那我们就看看有哪些类实现了 RejectedExecutionHandler 接口。



看到这里，我们发现 RejectedExecutionHandler 接口的实现类正是线程池默认提供的四种拒绝策略的实现类。

至于 reject(Runnable)方法中具体会执行哪个类的拒绝策略，是根据创建线程池时传递的参数决定的。如果没有传递拒绝策略，则默认会执行 AbortPolicy 类的拒绝策略。否则会执行传递的类的拒绝策略。

在创建线程池时，除了能够传递 JDK 默认提供的拒绝策略外，还可以传递自定义的拒绝策略。如果想使用自定义的拒绝策略，则只需要实现 RejectedExecutionHandler 接口，并重写 rejectedExecution(Runnable, ThreadPoolExecutor)方法即可。例如，下面的代码。

```
public class CustomPolicy implements RejectedExecutionHandler {

        public CustomPolicy() { }

        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
                if (!e.isShutdown()) {
                        System.out.println("使用调用者所在的线程来执行任务")
                        r.run();
                }
        }
}
```

使用如下方式创建线程池。

```
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                60L, TimeUnit.SECONDS,
                new SynchronousQueue<Runnable>(),
                Executors.defaultThreadFactory(),
                     new CustomPolicy());
```

至此，线程池执行任务的整体核心逻辑分析结束。

十一、通过源码深度分析线程池中 Worker 线程的执行流程

在《高并发之——通过 ThreadPoolExecutor 类的源码深度解析线程池执行任务的核心流程》一节中我们深度分析了线程池执行任务的核心流程，在 ThreadPoolExecutor 类的 addWorker(Runnable, boolean)方法中，使用 CAS 安全的更新线程的数量之后，接下来就是创建新的 Worker 线程执行任务，所以，我们先来分析下 Worker 类的源码。

Worker 类分析

Worker 类从类的结构上来看，继承了 AQS（AbstractQueuedSynchronizer 类）并实现了 Runnable 接口。本质上，Worker 类既是一个同步组件，也是一个执行任务的线程。接下来，我们看下 Worker 类的源码，如下所示。

```
private final class Worker extends AbstractQueuedSynchronizer implements R
unnable {
        private static final long serialVersionUID = 6138294804551838833L;
        //执行任务的线程类
        final Thread thread;
        //初始化执行的任务，第一次执行的任务
        Runnable firstTask;
        //完成任务的计数
        volatile long completedTasks;
        //Worker 类的构造方法，初始化任务并调用线程工厂创建执行任务的线程
        Worker(Runnable firstTask) {
                setState(-1);
                this.firstTask = firstTask;
                this.thread = getThreadFactory().newThread(this);
        }
        //重写 Runnable 接口的 run()方法
        public void run() {
                //调用 ThreadPoolExecutor 类的 runWorker(Worker)方法
                runWorker(this);
        }

        //检测是否是否获取到锁
        //state=0 表示未获取到锁
        //state=1 表示已获取到锁
        protected boolean isHeldExclusively() {
```

```
            return getState() != 0;
        }

        //使用 AQS 设置线程状态
        protected boolean tryAcquire(int unused) {
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }

        //尝试释放锁
        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }

        public void lock()       { acquire(1); }
        public boolean tryLock()  { return tryAcquire(1); }
        public void unlock()     { release(1); }
        public boolean isLocked() { return isHeldExclusively(); }

        void interruptIfStarted() {
            Thread t;
            if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                }
            }
        }
    }
```

在 Worker 类的构造方法中，可以看出，首先将同步状态 state 设置为-1，设置为-1 是为了防止 runWorker 方法运行之前被中断。这是因为如果其他线程调用线程池的 shutdownNow()方法时，如果 Worker 类中的 state 状态的值大于 0，则会中断线程，如果 state 状态的值为-1，则不会中断线程。

Worker 类实现了 Runnable 接口，需要重写 run 方法，而 Worker 的 run 方法本质上调用的是 ThreadPoolExecutor 类的 runWorker 方法，在 runWorker 方法中，会首先调用 unlock 方法，该方法会将 state 置为 0，所以这个时候调用 shutDownNow 方法就会中断当前线程，而这个时候已经进入了 runWork 方法，就不会在还没有执行 runWorker 方法的时候就中断线程。

注意：大家需要重点理解 Worker 类的实现。

Worker 类中调用了 ThreadPoolExecutor 类的 runWorker(Worker)方法。接下来，我们一起看下 ThreadPoolExecutor 类的 runWorker(Worker)方法的实现。

runWorker(Worker)方法

首先，我们看下 RunWorker(Worker)方法的源码，如下所示。

```
final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        //释放锁，将 state 设置为 0,允许中断任务的执行
        w.unlock();
        boolean completedAbruptly = true;
        try {
                //如果任务不为空，或者从任务队列中获取的任务不为空，则执行 while 循
环
                while (task != null || (task = getTask()) != null) {
                        //如果任务不为空，则获取 Worker 工作线程的独占锁
                        w.lock();
                        //如果线程已经停止，或者中断线程后线程终止并且没有成功中断
线程
                        //大家好好理解下这个逻辑
                        if ((runStateAtLeast(ctl.get(), STOP) ||
                                (Thread.interrupted() &&
                                 runStateAtLeast(ctl.get(), STOP))) &&
                                !wt.isInterrupted())
                                //中断线程
                                wt.interrupt();
                        try {
                                //执行任务前执行的逻辑
```

```
                    beforeExecute(wt, task);
                    Throwable thrown = null;
                    try {
                        //调用 Runable 接口的 run 方法执行任务
                        task.run();
                    } catch (RuntimeException x) {
                        thrown = x; throw x;
                    } catch (Error x) {
                        thrown = x; throw x;
                    } catch (Throwable x) {
                        thrown = x; throw new Error(x);
                    } finally {
                        //执行任务后执行的逻辑
                        afterExecute(task, thrown);
                    }
                } finally {
                    //任务执行完成后，将其设置为空
                    task = null;
                    //完成的任务数量加 1
                    w.completedTasks++;
                    //释放工作线程获得的锁
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            //执行退出 Worker 线程的逻辑
            processWorkerExit(w, completedAbruptly);
        }
}
```

这里，我们拆解 runWorker(Worker)方法。

（1）获取当前线程的句柄和工作线程中的任务，并将工作线程中的任务设置为空，执行 unlock 方法释放锁，将 state 状态设置为 0，此时可以中断工作线程，代码如下所示。

```
Thread wt = Thread.currentThread();
Runnable task = w.firstTask;
w.firstTask = null;
```

```
//释放锁，将 state 设置为 0,允许中断任务的执行
w.unlock();
```

（2）在 while 循环中进行判断，如果任务不为空，或者从任务队列中获取的任务不为空，则执行 while 循环，否则，调用 processWorkerExit(Worker, boolean)方法退出 Worker 工作线程。

```
while (task != null || (task = getTask()) != null)
```

（3）如果满足 while 的循环条件，首先获取工作线程内部的独占锁，并执行一系列的逻辑判断来检测是否需要中断当前线程的执行，代码如下所示。

```
//如果任务不为空，则获取 Worker 工作线程的独占锁
w.lock();
//如果线程已经停止，或者中断线程后线程终止并且没有成功中断线程
//大家好好理解下这个逻辑
if ((runStateAtLeast(ctl.get(), STOP) ||
         (Thread.interrupted() &&
          runStateAtLeast(ctl.get(), STOP))) &&
        !wt.isInterrupted())
        //中断线程
        wt.interrupt();
```

（4）调用执行任务前执行的逻辑，如下所示

```
//执行任务前执行的逻辑
beforeExecute(wt, task);
```

（5）调用 Runable 接口的 run 方法执行任务

```
//调用 Runable 接口的 run 方法执行任务
task.run();
```

（6）调用执行任务后执行的逻辑

```
//执行任务后执行的逻辑
afterExecute(task, thrown);
```

（7）将完成的任务设置为空，完成的任务数量加 1 并释放工作线程的锁。

```
//任务执行完成后，将其设置为空
task = null;
//完成的任务数量加 1
```

```
w.completedTasks++;
//释放工作线程获得的锁
w.unlock();
```

（8）退出 Worker 线程的执行，如下所示

```
//执行退出 Worker 线程的逻辑
processWorkerExit(w, completedAbruptly);
```

从代码分析上可以看到，当从 Worker 线程中获取的任务为空时，会调用 getTask()方法从任务队列中获取任务，接下来，我们看下 getTask()方法的实现。

getTask()方法

我们先来看下 getTask()方法的源代码，如下所示。

```
private Runnable getTask() {
        //轮询是否超时的标识
        boolean timedOut = false;
        //自旋 for 循环
        for (;;) {
                //获取 ctl
                int c = ctl.get();
                //获取线程池的状态
                int rs = runStateOf(c);

                //检测任务队列是否在线程池停止或关闭的时候为空
                //也就是说任务队列是否在线程池未正常运行时为空
                if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                        //减少 Worker 线程的数量
                        decrementWorkerCount();
                        return null;
                }
                //获取线程池中线程的数量
                int wc = workerCountOf(c);

                //检测当前线程池中的线程数量是否大于 corePoolSize 的值或者是否正在
等待执行任务
                boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

                //如果线程池中的线程数量大于 corePoolSize
```

```
                //获取大于 corePoolSize 或者是否正在等待执行任务并且轮询超时
                //并且当前线程池中的线程数量大于 1 或者任务队列为空
                if ((wc > maximumPoolSize || (timed && timedOut))
                        && (wc > 1 || workQueue.isEmpty())) {
                        //成功减少线程池中的工作线程数量
                        if (compareAndDecrementWorkerCount(c))
                                return null;
                        continue;
                }

                try {
                        //从任务队列中获取任务
                        Runnable r = timed ?
                                workQueue.poll(keepAliveTime, TimeUnit.NANOSEC
ONDS) :
                                workQueue.take();
                        //任务不为空直接返回任务
                        if (r != null)
                                return r;
                        timedOut = true;
                } catch (InterruptedException retry) {
                        timedOut = false;
                }
        }
}
```

getTask()方法的逻辑比较简单，大家看源码就可以了，我这里就不重复描述了。

接下来，我们看下在正式调用 Runnable 的 run()方法前后，执行的 beforeExecute 方法和 afterExecute 方法。

beforeExecute(Thread, Runnable)方法

beforeExecute(Thread, Runnable)方法的源代码如下所示。

```
protected void beforeExecute(Thread t, Runnable r) { }
```

可以看到，beforeExecute(Thread, Runnable)方法的方法体为空，我们可以创建 ThreadPoolExecutor 的子类来重写 beforeExecute(Thread, Runnable)方法，使得线程池正式执行任务之前，执行我们自己定义的业务逻辑。

afterExecute(Runnable, Throwable)方法

afterExecute(Runnable, Throwable)方法的源代码如下所示。

```
protected void afterExecute(Runnable r, Throwable t) { }
```

可以看到，afterExecute(Runnable, Throwable)方法的方法体同样为空，我们可以创建 ThreadPoolExecutor 的子类来重写 afterExecute(Runnable, Throwable)方法，使得线程池在执行任务之后执行我们自己定义的业务逻辑。

接下来，就是退出工作线程的 processWorkerExit(Worker, boolean)方法。

processWorkerExit(Worker, boolean)方法

processWorkerExit(Worker, boolean)方法的逻辑主要是执行退出 Worker 线程，并且对一些资源进行清理，源代码如下所示。

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
        //执行过程中出现了异常，突然中断
        if (completedAbruptly)
                //将工作线程的数量减 1
                decrementWorkerCount();
        //获取全局锁
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                //累加完成的任务数量
                completedTaskCount += w.completedTasks;
                //将完成的任务从 workers 集合中移除
                workers.remove(w);
        } finally {
                //释放锁
                mainLock.unlock();
        }
        //尝试终止工作线程的执行
        tryTerminate();
        //获取 ctl
        int c = ctl.get();
        //判断当前线程池的状态是否小于 STOP（RUNNING 或者 SHUTDOWN）
        if (runStateLessThan(c, STOP)) {
                //如果没有突然中断完成
```

```
            if (!completedAbruptly) {
                //如果 allowCoreThreadTimeOut 为 true，为 min 赋值为 0，否
则赋值为 corePoolSize
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                //如果 min 为 0 并且工作队列不为空
                if (min == 0 && ! workQueue.isEmpty())
                    //min 的值设置为 1
                    min = 1;
                //如果线程池中的线程数量大于 min 的值
                if (workerCountOf(c) >= min)
                    //返回，不再执行程序
                    return;
            }
            //调用 addWorker 方法
            addWorker(null, false);
        }
}
```

接下来，我们拆解 processWorkerExit(Worker, boolean)方法。

（1）执行过程中出现了异常，突然中断执行，则将工作线程数量减 1，如下所示。

```
//执行过程中出现了异常，突然中断
if (completedAbruptly)
    //将工作线程的数量减 1
    decrementWorkerCount();
```

（2）获取锁累加完成的任务数量，并将完成的任务从 workers 集合中移除，并释放，如下所示。

```
//获取全局锁
final ReentrantLock mainLock = this.mainLock;
mainLock.lock();
try {
    //累加完成的任务数量
    completedTaskCount += w.completedTasks;
    //将完成的任务从 workers 集合中移除
    workers.remove(w);
} finally {
    //释放锁
```

```
        mainLock.unlock();
}
```

（3）尝试终止工作线程的执行

```
//尝试终止工作线程的执行
tryTerminate();
```

（4）处判断当前线程池中的线程个数是否小于核心线程数，如果是，需要新增一个线程保证有足够的线程可以执行任务队列中的任务或者提交的任务。

```
//获取 ctl
int c = ctl.get();
//判断当前线程池的状态是否小于 STOP（RUNNING 或者 SHUTDOWN）
if (runStateLessThan(c, STOP)) {
        //如果没有突然中断完成
        if (!completedAbruptly) {
                //如果 allowCoreThreadTimeOut 为 true，为 min 赋值为 0，否则赋值为 corePoolSize
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                //如果 min 为 0 并且工作队列不为空
                if (min == 0 && ! workQueue.isEmpty())
                        //min 的值设置为 1
                        min = 1;
                //如果线程池中的线程数量大于 min 的值
                if (workerCountOf(c) >= min)
                        //返回，不再执行程序
                        return;
        }
        //调用 addWorker 方法
        addWorker(null, false);
}
```

接下来，我们看下 tryTerminate()方法。

tryTerminate()方法

tryTerminate()方法的源代码如下所示。

```
final void tryTerminate() {
        //自旋 for 循环
        for (;;) {
```

```
        //获取 ctl
        int c = ctl.get();
        //如果线程池的状态为 RUNNING
        //或者状态大于 TIDYING
        //或者状态为 SHUTDOWN 并且任务队列为空
        //直接返回程序，不再执行后续逻辑
        if (isRunning(c) ||
                runStateAtLeast(c, TIDYING) ||
                (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
                return;
        //如果当前线程池中的线程数量不等于 0
        if (workerCountOf(c) != 0) {
                //中断线程的执行
                interruptIdleWorkers(ONLY_ONE);
                return;
        }
        //获取线程池的全局锁
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                //通过 CAS 将线程池的状态设置为 TIDYING
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                        try {
                                //调用 terminated()方法
                                terminated();
                        } finally {
                                //将线程池状态设置为 TERMINATED
                                ctl.set(ctlOf(TERMINATED, 0));
                                //唤醒所有因为调用线程池的 awaitTermination 方
法而被阻塞的线程
                                termination.signalAll();
                        }
                        return;
                }
        } finally {
                //释放锁
                mainLock.unlock();
        }
```

```
        }
}
```

（1）获取 ctl，根据情况设置线程池状态或者中断线程的执行，并返回。

```
//获取 ctl
int c = ctl.get();
//如果线程池的状态为 RUNNING
//或者状态大于 TIDYING
//或者状态为 SHUTDOWN 并且任务队列为空
//直接返回程序，不再执行后续逻辑
if (isRunning(c) ||
        runStateAtLeast(c, TIDYING) ||
        (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
        return;
//如果当前线程池中的线程数量不等于 0
if (workerCountOf(c) != 0) {
        //中断线程的执行
        interruptIdleWorkers(ONLY_ONE);
        return;
}
```

（2）获取全局锁，通过 CAS 设置线程池的状态，调用 terminated()方法执行逻辑，最终将线程池的状态设置为 TERMINATED，唤醒所有因为调用线程池的 awaitTermination 方法而被阻塞的线程，最终释放锁，如下所示。

```
//获取线程池的全局
final ReentrantLock mainLock = this.mainLock;
mainLock.lock();
try {
        //通过 CAS 将线程池的状态设置为 TIDYING
        if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                try {
                        //调用 terminated()方法
                        terminated();
                } finally {
                        //将线程池状态设置为 TERMINATED
                        ctl.set(ctlOf(TERMINATED, 0));
                        //唤醒所有因为调用线程池的 awaitTermination 方法而被阻塞的
线程
```

```
                    termination.signalAll();
                }
                return;
            }
        } finally {
            //释放锁
            mainLock.unlock();
        }
```

接下来，看下 terminated()方法。

terminated()方法

terminated()方法的源代码如下所示。

```
protected void terminated() { }
```

可以看到，terminated()方法的方法体为空，我们可以创建 ThreadPoolExecutor 的子类来重写 terminated()方法，值得 Worker 线程调用 tryTerminate()方法时执行我们自己定义的 terminated()方法的业务逻辑。

十二、从源码角度深度解析线程池是如何实现优雅退出的

本文，我们就来从源码角度深度解析线程池是如何优雅的退出程序的。首先，我们来看下 ThreadPoolExecutor 类中的 shutdown()方法。

shutdown()方法

当使用线程池的时候，调用了 shutdown()方法后，线程池就不会再接受新的执行任务了。但是在调用 shutdown()方法之前放入任务队列中的任务还是要执行的。此方法是非阻塞方法，调用后会立即返回，并不会等待任务队列中的任务全部执行完毕后再返回。我们看下 shutdown()方法的源代码，如下所示。

```
public void shutdown() {
        //获取线程池的全局锁
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                //检查是否有关闭线程池的权限
                checkShutdownAccess();
                //将当前线程池的状态设置为 SHUTDOWN
                advanceRunState(SHUTDOWN);
                //中断 Worker 线程
                interruptIdleWorkers();
                //为 ScheduledThreadPoolExecutor 调用钩子函数
                onShutdown(); // hook for
        } finally {
                //释放线程池的全局锁
                mainLock.unlock();
        }
        //尝试将状态变为 TERMINATED
        tryTerminate();
}
```

总体来说，shutdown()方法的代码比较简单，首先检查了是否有权限来关闭线程池，如果有权限，则再次检测是否有中断工作线程的权限，如果没有权限，则会抛出 SecurityException 异常，代码如下所示。

```
//检查是否有关闭线程池的权限
checkShutdownAccess();
//将当前线程池的状态设置为 SHUTDOWN
```

```
advanceRunState(SHUTDOWN);
//中断 Worker 线程
interruptIdleWorkers();
```

其中，checkShutdownAccess()方法的实现代码如下所示。

```
private void checkShutdownAccess() {
        SecurityManager security = System.getSecurityManager();
        if (security != null) {
                security.checkPermission(shutdownPerm);
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {
                        for (Worker w : workers)
                                security.checkAccess(w.thread);
                } finally {
                        mainLock.unlock();
                }
        }
}
```

对于 checkShutdownAccess()方法的代码理解起来比较简单，就是检测是否具有关闭线程池的权限，期间使用了线程池的全局锁。

接下来，我们看 advanceRunState(int)方法的源代码，如下所示。

```
private void advanceRunState(int targetState) {
        for (;;) {
                int c = ctl.get();
                if (runStateAtLeast(c, targetState) ||
                        ctl.compareAndSet(c, ctlOf(targetState, workerCountOf(c))))
                        break;
        }
}
```

advanceRunState(int)方法的整体逻辑就是：判断当前线程池的状态是否为指定的状态，在 shutdown()方法中传递的状态是 SHUTDOWN，如果是 SHUTDOWN，则直接返回；如果不是 SHUTDOWN，则将当前线程池的状态设置为 SHUTDOWN。

接下来，我们看看 showdown()方法调用的 interruptIdleWorkers()方法，如下所示。

```
private void interruptIdleWorkers() {
        interruptIdleWorkers(false);
}
```

可以看到，interruptIdleWorkers()方法调用的是 interruptIdleWorkers(boolean)方法，继续看 interruptIdleWorkers(boolean)方法的源代码，如下所示。

```
private void interruptIdleWorkers(boolean onlyOne) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                for (Worker w : workers) {
                        Thread t = w.thread;
                        if (!t.isInterrupted() && w.tryLock()) {
                                try {
                                        t.interrupt();
                                } catch (SecurityException ignore) {
                                } finally {
                                        w.unlock();
                                }
                        }
                        if (onlyOne)
                                break;
                }
        } finally {
                mainLock.unlock();
        }
}
```

上述代码的总体逻辑为：获取线程池的全局锁，循环所有的工作线程，检测线程是否被中断，如果没有被中断，并且 Worker 线程获得了锁，则执行线程的中断方法，并释放线程获取到的锁。此时如果 onlyOne 参数为 true，则退出循环。否则，循环所有的工作线程，执行相同的操作。最终，释放线程池的全局锁。

接下来，我们看下 shutdownNow()方法。

shutdownNow()方法

如果调用了线程池的 shutdownNow()方法，则线程池不会再接受新的执行任务，也会将任务队列中存在的任务丢弃，正在执行的 Worker 线程也会被立即中断，同时，方法会立刻返回，此方法存在一个返回值，也就是当前任务队列中被丢弃的任务列表。

shutdownNow()方法的源代码如下所示。

```
public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                //检查是否有关闭权限
                checkShutdownAccess();
                //设置线程池的状态为 STOP
                advanceRunState(STOP);
                //中断所有的 Worker 线程
                interruptWorkers();
                //将任务队列中的任务移动到 tasks 集合中
                tasks = drainQueue();
        } finally {
                mainLock.unlock();
        }
        /尝试将状态变为 TERMINATED
        tryTerminate();
        //返回 tasks 集合
        return tasks;
}
```

shutdownNow()方法的源代码的总体逻辑与 shutdown()方法基本相同，只是 shutdownNow()方法将线程池的状态设置为 STOP，中断所有的 Worker 线程，并且将任务队列中的所有任务移动到 tasks 集合中并返回。

可以看到，shutdownNow()方法中断所有的线程时，调用了 interruptWorkers()方法，接下来，我们就看下 interruptWorkers()方法的源代码，如下所示。

```
private void interruptWorkers() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
```

```
        try {
            for (Worker w : workers)
                w.interruptIfStarted();
        } finally {
            mainLock.unlock();
        }
}
```

interruptWorkers()方法的逻辑比较简单，就是获得线程池的全局锁，循环所有的工作线程，依次中断线程，最后释放线程池的全局锁。

在 interruptWorkers()方法的内部，实际上调用的是 Worker 类的 interruptIfStarted()方法来中断线程，我们看下 Worker 类的 interruptIfStarted()方法的源代码，如下所示。

```
void interruptIfStarted() {
        Thread t;
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
            try {
                t.interrupt();
            } catch (SecurityException ignore) {
            }
        }
}
```

发现其本质上调用的还是 Thread 类的 interrupt()方法来中断线程。

awaitTermination(long, TimeUnit)方法

当线程池调用了 awaitTermination(long, TimeUnit)方法后，会阻塞调用者所在的线程，直到线程池的状态修改为 TERMINATED 才返回，或者达到了超时时间返回。接下来，我们看下 awaitTermination(long, TimeUnit)方法的源代码，如下所示。

```
public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException {
        //获取距离超时时间剩余的时长
        long nanos = unit.toNanos(timeout);
        //获取 Worker 线程的的全局锁
        final ReentrantLock mainLock = this.mainLock;
        //加锁
```

```
        mainLock.lock();
        try {
            for (;;) {
                //当前线程池状态为 TERMINATED 状态，会返回 true
                if (runStateAtLeast(ctl.get(), TERMINATED))
                    return true;
                //达到超时时间，已超时，则返回 false
                if (nanos <= 0)
                    return false;
                //重置距离超时时间的剩余时长
                nanos = termination.awaitNanos(nanos);
            }
        } finally {
            //释放锁
            mainLock.unlock();
        }
}
```

上述代码的总体逻辑为：首先获取 Worker 线程的独占锁，后在循环判断当前线程池是否已经是 TERMINATED 状态，如果是则直接返回 true，否则检测是否已经超时，如果已经超时，则返回 false。如果未超时，则重置距离超时时间的剩余时长。接下来，进入下一轮循环，再次检测当前线程池是否已经是 TERMINATED 状态，如果是则直接返回 true，否则检测是否已经超时，如果已经超时，则返回 false。如果未超时，则重置距离超时时间的剩余时长。以此循环，直到线程池的状态变为 TERMINATED 或者已经超时。

十三、ScheduledThreadPoolExecutor 与 Timer 的区别

JDK 1.5 开始提供 ScheduledThreadPoolExecutor 类，ScheduledThreadPoolExecutor 类继承 ThreadPoolExecutor 类重用线程池实现了任务的周期性调度功能。在 JDK 1.5 之前，实现任务的周期性调度主要使用的是 Timer 类和 TimerTask 类。本文，就简单介绍下 ScheduledThreadPoolExecutor 类与 Timer 类的区别，ScheduledThreadPoolExecutor 类相比于 Timer 类来说，究竟有哪些优势，以及二者分别实现任务调度的简单示例。

二者的区别

线程角度

- Timer 是单线程模式，如果某个 TimerTask 任务的执行时间比较久，会影响到其他任务的调度执行。
- ScheduledThreadPoolExecutor 是多线程模式，并且重用线程池，某个 ScheduledFutureTask 任务执行的时间比较久，不会影响到其他任务的调度执行。

系统时间敏感度

- Timer 调度是基于操作系统的绝对时间的，对操作系统的时间敏感，一旦操作系统的时间改变，则 Timer 的调度不再精确。
- ScheduledThreadPoolExecutor 调度是基于相对时间的，不受操作系统时间改变的影响。

是否捕获异常

- Timer 不会捕获 TimerTask 抛出的异常，加上 Timer 又是单线程的。一旦某个调度任务出现异常，则整个线程就会终止，其他需要调度的任务也不再执行。
- ScheduledThreadPoolExecutor 基于线程池来实现调度功能，某个任务抛出异常后，其他任务仍能正常执行。

任务是否具备优先级

- Timer 中执行的 TimerTask 任务整体上没有优先级的概念，只是按照系统的绝对时间来执行任务。

- ScheduledThreadPoolExecutor 中执行的 ScheduledFutureTask 类实现了 java.lang.Comparable 接口和 java.util.concurrent.Delayed 接口，这也就说明了 ScheduledFutureTask 类中实现了两个非常重要的方法，一个是 java.lang.Comparable 接口的 compareTo 方法，一个是 java.util.concurrent.Delayed 接口的 getDelay 方法。在 ScheduledFutureTask 类中 compareTo 方法方法实现了任务的比较，距离下次执行的时间间隔短的任务会排在前面，也就是说，距离下次执行的时间间隔短的任务的优先级比较高。而 getDelay 方法则能够返回距离下次任务执行的时间间隔。

是否支持对任务排序

- Timer 不支持对任务的排序。
- ScheduledThreadPoolExecutor 类中定义了一个静态内部类 DelayedWorkQueue，DelayedWorkQueue 类本质上是一个有序队列，为需要调度的每个任务按照距离下次执行时间间隔的大小来排序

能否获取返回的结果

- Timer 中执行的 TimerTask 类只是实现了 java.lang.Runnable 接口，无法从 TimerTask 中获取返回的结果。
- ScheduledThreadPoolExecutor 中执行的 ScheduledFutureTask 类继承了 FutureTask 类，能够通过 Future 来获取返回的结果。

通过以上对 ScheduledThreadPoolExecutor 类和 Timer 类的分析对比，相信在 JDK 1.5 之后，就没有使用 Timer 来实现定时任务调度的必要了。

二者简单的示例

这里，给出使用 Timer 和 ScheduledThreadPoolExecutor 实现定时调度的简单示例，为了简便，我这里就直接使用匿名内部类的形式来提交任务。

Timer 类简单示例

源代码示例如下所示。

```
package io.binghe.concurrent.lab09;

import java.util.Timer;
import java.util.TimerTask;
```

```java
/**
 * @author binghe
 * @version 1.0.0
 * @description 测试 Timer
 */
public class TimerTest {

    public static void main(String[] args) throws InterruptedException {
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
            @Override
            public void run() {
                System.out.println("测试 Timer 类");
            }
        }, 1000, 1000);
        Thread.sleep(10000);
        timer.cancel();
    }
}
```

运行结果如下所示。

```
测试 Timer 类
测试 Timer 类
测试 Timer 类
测试 Timer 类
测试 Timer 类
测试 Timer 类
测试 Timer 类
测试 Timer 类
测试 Timer 类
测试 Timer 类
```

ScheduledThreadPoolExecutor 类简单示例

源代码示例如下所示。

```java
package io.binghe.concurrent.lab09;
```

```
import java.util.concurrent.*;

/**
 * @author binghe
 * @version 1.0.0
 * @description 测试 ScheduledThreadPoolExecutor
 */
public class ScheduledThreadPoolExecutorTest {
    public static void main(String[] args) throws  InterruptedException {
        ScheduledExecutorService scheduledExecutorService = Executors.newSc
heduledThreadPool(3);
        scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                System.out.println("测试测试 ScheduledThreadPoolExecutor");
            }
        }, 1, 1, TimeUnit.SECONDS);

        //主线程休眠 10 秒
        Thread.sleep(10000);

        System.out.println("正在关闭线程池...");
        // 关闭线程池
        scheduledExecutorService.shutdown();
        boolean isClosed;
        // 等待线程池终止
        do {
            isClosed = scheduledExecutorService.awaitTermination(1, TimeUnit.DAY
S);
            System.out.println("正在等待线程池中的任务执行完成");
        } while(!isClosed);

        System.out.println("所有线程执行结束，线程池关闭");
    }
}
```

运行结果如下所示。

测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
测试测试 ScheduledThreadPoolExecutor
正在关闭线程池...
测试测试 ScheduledThreadPoolExecutor
正在等待线程池中的任务执行完成
所有线程执行结束，线程池关闭

注意：关于 Timer 和 ScheduledThreadPoolExecutor 还有其他的使用方法，这里，我就简单列出以上两个使用示例，更多的使用方法大家可以自行实现。

十四、深度解析 ScheduledThreadPoolExecutor 类的源代码

在【高并发专题】的专栏中，我们深度分析了 ThreadPoolExecutor 类的源代码，而 ScheduledThreadPoolExecutor 类是 ThreadPoolExecutor 类的子类。今天我们就来一起手撕 ScheduledThreadPoolExecutor 类的源代码。

构造方法

我们先来看下 ScheduledThreadPoolExecutor 的构造方法，源代码如下所示。

```
public ScheduledThreadPoolExecutor(int corePoolSize) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS, new Del
ayedWorkQueue());
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadF
actory) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), threadFactory);
}

public ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHan
dler handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), handler);
}

public ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadF
actory, RejectedExecutionHandler handler) {
        super(corePoolSize, Integer.MAX_VALUE, 0, NANOSECONDS,
                new DelayedWorkQueue(), threadFactory, handler);
}
```

从代码结构上来看，ScheduledThreadPoolExecutor 类是 ThreadPoolExecutor 类的子类，ScheduledThreadPoolExecutor 类的构造方法实际上调用的是 ThreadPoolExecutor 类的构造方法。

schedule 方法

接下来，我们看一下 ScheduledThreadPoolExecutor 类的 schedule 方法，源代码如下所示。

```
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUni
t unit) {
        //如果传递的 Runnable 对象和 TimeUnit 时间单位为空
        //抛出空指针异常
        if (command == null || unit == null)
                throw new NullPointerException();
        //封装任务对象，在 decorateTask 方法中直接返回 ScheduledFutureTask 对
象
        RunnableScheduledFuture<?> t = decorateTask(command, new Schedu
ledFutureTask<Void>(command, null, triggerTime(delay, unit)));
        //执行延时任务
        delayedExecute(t);
        //返回任务
        return t;
}

public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, Tim
eUnit unit)
        //如果传递的 Callable 对象和 TimeUnit 时间单位为空
        //抛出空指针异常
        if (callable == null || unit == null)
                throw new NullPointerException();
        //封装任务对象，在 decorateTask 方法中直接返回 ScheduledFutureTask 对
象
        RunnableScheduledFuture<V> t = decorateTask(callable,
                new ScheduledFutureTask<V>(callable, triggerTime(delay, unit)));
        //执行延时任务
        delayedExecute(t);
        //返回任务
        return t;
}
```

从源代码可以看出，ScheduledThreadPoolExecutor 类提供了两个重载的 schedule 方法，两个 schedule 方法的第一个参数不同。可以传递 Runnable 接口对象，也可以传递 Callable 接口对象。在方法内部，会将 Runnable 接口对象

和 Callable 接口对象封装成 RunnableScheduledFuture 对象，本质上就是封装成 ScheduledFutureTask 对象。并通过 delayedExecute 方法来执行延时任务。

在源代码中，我们看到两个 schedule 都调用了 decorateTask 方法，接下来，我们就看看 decorateTask 方法。

decorateTask 方法

decorateTask 方法源代码如下所示。

```
protected <V> RunnableScheduledFuture<V> decorateTask(Runnable runnable, RunnableScheduledFuture<V> task) {
        return task;
}

protected <V> RunnableScheduledFuture<V> decorateTask(Callable<V> callable, RunnableScheduledFuture<V> task) {
        return task;
}
```

通过源码可以看出 decorateTask 方法的实现比较简单，接收一个 Runnable 接口对象或者 Callable 接口对象和封装的 RunnableScheduledFuture 任务，两个方法都是将 RunnableScheduledFuture 任务直接返回。在 ScheduledThreadPoolExecutor 类的子类中可以重写这两个方法。

接下来，我们继续看下 scheduleAtFixedRate 方法。

scheduleAtFixedRate 方法

scheduleAtFixedRate 方法源代码如下所示。

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) {
        //传入的 Runnable 对象和 TimeUnit 为空，则抛出空指针异常
        if (command == null || unit == null)
                throw new NullPointerException();
        //如果执行周期 period 传入的数值小于或者等于 0
        //抛出非法参数异常
        if (period <= 0)
                throw new IllegalArgumentException();
```

```
        //将 Runnable 对象封装成 ScheduledFutureTask 任务，
        //并设置执行周期
        ScheduledFutureTask<Void> sft =
                new ScheduledFutureTask<Void>(command, null, triggerTime(initi
alDelay, unit), unit.toNanos(period));
        //调用 decorateTask 方法，本质上还是直接返回 ScheduledFutureTask 对象
        RunnableScheduledFuture<Void> t = decorateTask(command, sft);
        //设置执行的任务
        sft.outerTask = t;
        //执行延时任务
        delayedExecute(t);
        //返回执行的任务
        return t;
}
```

通过源码可以看出，scheduleAtFixedRate 方法将传递的 Runnable 对象封装成 ScheduledFutureTask 任务对象，并设置了执行周期，下一次的执行时间相对于上一次的执行时间来说，加上了 period 时长，时长的具体单位由 TimeUnit 决定。采用固定的频率来执行定时任务。

ScheduledThreadPoolExecutor 类中另一个定时调度任务的方法是 scheduleWithFixedDelay 方法，接下来，我们就一起看看 scheduleWithFixedDelay 方法。

scheduleWithFixedDelay 方法

scheduleWithFixedDelay 方法的源代码如下所示。

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long
 initialDelay, long delay, TimeUnit unit) {
        //传入的 Runnable 对象和 TimeUnit 为空，则抛出空指针异常
        if (command == null || unit == null)
                throw new NullPointerException();
        //任务延时时长小于或者等于 0，则抛出非法参数异常
        if (delay <= 0)
                throw new IllegalArgumentException();
        //将 Runnable 对象封装成 ScheduledFutureTask 任务
        //并设置固定的执行周期来执行任务
        ScheduledFutureTask<Void> sft =
                new ScheduledFutureTask<Void>(command, null,triggerTime(initi
```

```
alDelay, unit), unit.toNanos(-delay));
        //调用 decorateTask 方法，本质上直接返回 ScheduledFutureTask 任务
        RunnableScheduledFuture<Void> t = decorateTask(command, sft);
        //设置执行的任务
        sft.outerTask = t;
        //执行延时任务
        delayedExecute(t);
        //返回任务
        return t;
}
```

从 scheduleWithFixedDelay 方法的源代码，我们可以看出在将 Runnable 对象封装成 ScheduledFutureTask 时，设置了执行周期，但是此时设置的执行周期与 scheduleAtFixedRate 方法设置的执行周期不同。此时设置的执行周期规则为：下一次任务执行的时间是上一次任务完成的时间加上 delay 时长，时长单位由 TimeUnit 决定。也就是说，具体的执行时间不是固定的，但是执行的周期是固定的，整体采用的是相对固定的延迟来执行定时任务。

如果大家细心的话，会发现在 scheduleWithFixedDelay 方法中设置执行周期时，传递的 delay 值为负数，如下所示。

```
ScheduledFutureTask<Void> sft =
                new ScheduledFutureTask<Void>(command, null, triggerTime(initi
alDelay, unit), unit.toNanos(-delay));
```

这里的负数表示的是相对固定的延迟。

在 ScheduledFutureTask 类中，存在一个 setNextRunTime 方法，这个方法会在 run 方法执行完任务后调用，这个方法更能体现 scheduleAtFixedRate 方法和 scheduleWithFixedDelay 方法的不同，setNextRunTime 方法的源码如下所示。

```
private void setNextRunTime() {
        //距离下次执行任务的时长
        long p = period;
        //固定频率执行，
        //上次执行任务的时间
        //加上任务的执行周期
        if (p > 0)
                time += p;
```

```
        //相对固定的延迟
        //使用的是系统当前时间
        //加上任务的执行周期
        else
                time = triggerTime(-p);
}
```

在 setNextRunTime 方法中通过对下次执行任务的时长进行判断来确定是固定频率执行还是相对固定的延迟。

triggerTime 方法

在 ScheduledThreadPoolExecutor 类中提供了两个 triggerTime 方法，用于获取下一次执行任务的具体时间。triggerTime 方法的源码如下所示。

```
private long triggerTime(long delay, TimeUnit unit) {
        return triggerTime(unit.toNanos((delay < 0) ? 0 : delay));
}

long triggerTime(long delay) {
        return now() +
                ((delay < (Long.MAX_VALUE >> 1)) ? delay : overflowFree(delay));
}
```

这两个 triggerTime 方法的代码比较简单，就是获取下一次执行任务的具体时间。有一点需要注意的是：delay < (Long.MAX*VALUE >> 1 判断 delay 的值是否小于 Long.MAX*VALUE 的一半，如果小于 Long.MAX_VALUE 值的一半，则直接返回 delay，否则需要处理溢出的情况。

我们看到在 triggerTime 方法中处理防止溢出的逻辑使用了 overflowFree 方法，接下来，我们就看看 overflowFree 方法的实现。

overflowFree 方法

overflowFree 方法的源代码如下所示。

```
private long overflowFree(long delay) {
        //获取队列中的节点
        Delayed head = (Delayed) super.getQueue().peek();
        //获取的节点不为空，则进行后续处理
        if (head != null) {
```

```
                //从队列节点中获取延迟时间
                long headDelay = head.getDelay(NANOSECONDS);
                //如果从队列中获取的延迟时间小于 0，并且传递的 delay
                //值减去从队列节点中获取延迟时间小于 0
                if (headDelay < 0 && (delay - headDelay < 0))
                        //将 delay 的值设置为 Long.MAX_VALUE + headDelay
                        delay = Long.MAX_VALUE + headDelay;
        }
        //返回延迟时间
        return delay;
}
```

通过对 overflowFree 方法的源码分析，可以看出 overflowFree 方法本质上就是为了限制队列中的所有节点的延迟时间在 Long.MAX_VALUE 值之内，防止在 ScheduledFutureTask 类中的 compareTo 方法中溢出。

ScheduledFutureTask 类中的 compareTo 方法的源码如下所示。

```
public int compareTo(Delayed other) {
        if (other == this) // compare zero if same object
                return 0;
        if (other instanceof ScheduledFutureTask) {
                ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;
                long diff = time - x.time;
                if (diff < 0)
                        return -1;
                else if (diff > 0)
                        return 1;
                else if (sequenceNumber < x.sequenceNumber)
                        return -1;
                else
                        return 1;
        }
        long diff = getDelay(NANOSECONDS) - other.getDelay(NANOSECOND
S);
        return (diff < 0) ? -1 : (diff > 0) ? 1 : 0;
}
```

compareTo 方法的主要作用就是对各延迟任务进行排序，距离下次执行时间靠前的任务就排在前面。

delayedExecute 方法

delayedExecute 方法是 ScheduledThreadPoolExecutor 类中延迟执行任务的方法，源代码如下所示。

```
private void delayedExecute(RunnableScheduledFuture<?> task) {
        //如果当前线程池已经关闭
        //则执行线程池的拒绝策略
        if (isShutdown())
                reject(task);
        //线程池没有关闭
        else {
                //将任务添加到阻塞队列中
                super.getQueue().add(task);
                //如果当前线程池是 SHUTDOWN 状态
                //并且当前线程池状态下不能执行任务
                //并且成功从阻塞队列中移除任务
                if (isShutdown() &&
                        !canRunInCurrentRunState(task.isPeriodic()) &&
                        remove(task))
                        //取消任务的执行，但不会中断执行中的任务
                        task.cancel(false);
                else
                        //调用 ThreadPoolExecutor 类中的 ensurePrestart()方法
                        ensurePrestart();
        }
}
```

可以看到在 delayedExecute 方法内部调用了 canRunInCurrentRunState 方法，canRunInCurrentRunState 方法的源码实现如下所示。

```
boolean canRunInCurrentRunState(boolean periodic) {
        return isRunningOrShutdown(periodic ? continueExistingPeriodicTasksA
fterShutdown : executeExistingDelayedTasksAfterShutdown);
}
```

可以看到 canRunInCurrentRunState 方法的逻辑比较简单，就是判断线程池当前状态下能够执行任务。

另外，在 delayedExecute 方法内部还调用了 ThreadPoolExecutor 类中的 ensurePrestart()方法，接下来，我们看下 ThreadPoolExecutor 类中的 ensurePrestart()方法的实现，如下所示。

```
void ensurePrestart() {
        int wc = workerCountOf(ctl.get());
        if (wc < corePoolSize)
                addWorker(null, true);
        else if (wc == 0)
                addWorker(null, false);
}
```

在 ThreadPoolExecutor 类中的 ensurePrestart()方法中，首先获取当前线程池中线程的数量，如果线程数量小于 corePoolSize 则调用 addWorker 方法传递 null 和 true，如果线程数量为 0，则调用 addWorker 方法传递 null 和 false。

reExecutePeriodic 方法

reExecutePeriodic 方法的源代码如下所示。

```
void reExecutePeriodic(RunnableScheduledFuture<?> task) {
        //线程池当前状态下能够执行任务
        if (canRunInCurrentRunState(true)) {
                //将任务放入队列
                super.getQueue().add(task);
                //线程池当前状态下不能执行任务，并且成功移除任务
                if (!canRunInCurrentRunState(true) && remove(task))
                        //取消任务
                        task.cancel(false);
                else
                        //调用 ThreadPoolExecutor 类的 ensurePrestart()方法
                        ensurePrestart();
        }
}
```

总体来说 reExecutePeriodic 方法的逻辑比较简单，但是，这里需要注意和 delayedExecute 方法的不同点：调用 reExecutePeriodic 方法的时候已经执行过一次任务，所以，并不会触发线程池的拒绝策略；传入 reExecutePeriodic 方法的任务一定是周期性的任务。

onShutdown 方法

onShutdown 方法是 ThreadPoolExecutor 类中的钩子函数，它是在 ThreadPoolExecutor 类中的 shutdown 方法中调用的，而在 ThreadPoolExecutor 类中的 onShutdown 方法是一个空方法，如下所示。

```
void onShutdown() {
}
```

ThreadPoolExecutor 类中的 onShutdown 方法交由子类实现，所以 ScheduledThreadPoolExecutor 类覆写了 onShutdown 方法，实现了具体的逻辑，ScheduledThreadPoolExecutor 类中的 onShutdown 方法的源码实现如下所示。

```
@Override
void onShutdown() {
        //获取队列
        BlockingQueue<Runnable> q = super.getQueue();
        //在线程池已经调用 shutdown 方法后，是否继续执行现有延迟任务
        boolean keepDelayed = getExecuteExistingDelayedTasksAfterShutdown
Policy();
        //在线程池已经调用 shutdown 方法后，是否继续执行现有定时任务
        boolean keepPeriodic = getContinueExistingPeriodicTasksAfterShutdow
nPolicy();
        //在线程池已经调用 shutdown 方法后，不继续执行现有延迟任务和定时任务
        if (!keepDelayed && !keepPeriodic) {
                //遍历队列中的所有任务
                for (Object e : q.toArray())
                        //取消任务的执行
                        if (e instanceof RunnableScheduledFuture<?>)
                                ((RunnableScheduledFuture<?>) e).cancel(false);
                //清空队列
                q.clear();
        }
        //在线程池已经调用 shutdown 方法后，继续执行现有延迟任务和定时任务
        else {
                //遍历队列中的所有任务
                for (Object e : q.toArray()) {
                        //当前任务是 RunnableScheduledFuture 类型
                        if (e instanceof RunnableScheduledFuture) {
```

```
                        //将任务强转为 RunnableScheduledFuture 类型
                        RunnableScheduledFuture<?> t = (RunnableScheduledFuture<?>)e;
                        //在线程池调用 shutdown 方法后不继续的延迟任务或周期任务
                        //则从队列中删除并取消任务
                        if ((t.isPeriodic() ? !keepPeriodic : !keepDelayed) ||
                                t.isCancelled()) {
                                if (q.remove(t))
                                        t.cancel(false);
                        }
                }
        }
    }
    //最终调用 tryTerminate()方法
    tryTerminate();
}
```

ScheduledThreadPoolExecutor 类中的 onShutdown 方法的主要逻辑就是先判断线程池调用 shutdown 方法后，是否继续执行现有的延迟任务和定时任务，如果不再执行，则取消任务并清空队列；如果继续执行，将队列中的任务强转为 RunnableScheduledFuture 对象之后，从队列中删除并取消任务。大家需要好好理解这两种处理方式。最后调用 ThreadPoolExecutor 类的 tryTerminate 方法。

至此，ScheduledThreadPoolExecutor 类中的核心方法的源代码，我们就分析完了。

十五、朋友去面试竟然栽在了 Thread 类的源码上

前言

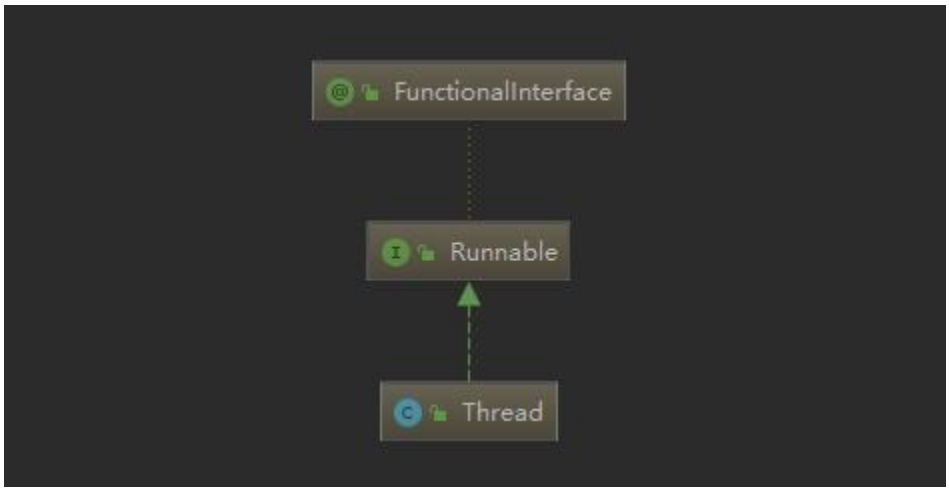
最近和一个朋友聊天，他跟我说起了他去 XXX 公司面试的情况，面试官的一个问题把他打懵了！竟然问他：你经常使用 Thread 创建线程，那你看过 Thread 类的源码吗？我这个朋友自然是没看过 Thread 类的源码，然后，就没有然后了！！！

所以，我们学习技术不仅需要知其然，更需要知其所以然，今天，我们就一起来简单看看 Thread 类的源码。

注意：本文是基于 JDK 1.8 来进行分析的。

Thread 类的继承关系

我们可以使用下图来表示 Thread 类的继承关系。



由上图我们可以看出，Thread 类实现了 Runnable 接口，而 Runnable 在 JDK 1.8 中被@FunctionalInterface 注解标记为函数式接口，Runnable 接口在 JDK 1.8 中的源代码如下所示。

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

Runnable 接口的源码比较简单，只是提供了一个 run()方法，这里就不再赘述了。

接下来，我们再来看看@FunctionalInterface 注解的源码，如下所示。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

可以看到，@FunctionalInterface 注解声明标记在 Java 类上，并在程序运行时生效。

Thread 类的源码剖析

Thread 类定义

Thread 在 java.lang 包下，Thread 类的定义如下所示。

```
public class Thread implements Runnable {
```

加载本地资源

打开 Thread 类后，首先，我们会看到在 Thread 类的最开始部分，定义了一个静态本地方法 registerNatives()，这个方法主要用来注册一些本地系统的资源。并在静态代码块中调用这个本地方法，如下所示。

```
//定义 registerNatives()本地方法注册系统资源
private static native void registerNatives();
static {
    //在静态代码块中调用注册本地系统资源的方法
    registerNatives();
}
```

Thread 中的成员变量

Thread 类中的成员变量如下所示。

```
//当前线程的名称
private volatile String name;
//线程的优先级
private int          priority;
private Thread       threadQ;
private long         eetop;
//当前线程是否是单步线程
private boolean    single_step;
```

```
//当前线程是否在后台运行
private boolean     daemon = false;
//Java 虚拟机的状态
private boolean     stillborn = false;
//真正在线程中执行的任务
private Runnable target;
//当前线程所在的线程组
private ThreadGroup group;
//当前线程的类加载器
private ClassLoader contextClassLoader;
//访问控制上下文
private AccessControlContext inheritedAccessControlContext;
//为匿名线程生成名称的编号
private static int threadInitNumber;
//与此线程相关的 ThreadLocal,这个 Map 维护的是 ThreadLocal 类
ThreadLocal.ThreadLocalMap threadLocals = null;
//与此线程相关的 ThreadLocal
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
//当前线程请求的堆栈大小，如果未指定堆栈大小，则会交给 JVM 来处理
private long stackSize;
//线程终止后存在的 JVM 私有状态
private long nativeParkEventPointer;
//线程的 id
private long tid;
//用于生成线程 id
private static long threadSeqNumber;
//当前线程的状态，初始化为 0，代表当前线程还未启动
private volatile int threadStatus = 0;
//由（私有）java.util.concurrent.locks.LockSupport.setBlocker 设置
//使用 java.util.concurrent.locks.LockSupport.getBlocker 访问
volatile Object parkBlocker;
//Interruptible 接口中定义了 interrupt 方法，用来中断指定的线程
private volatile Interruptible blocker;
//当前线程的内部锁
private final Object blockerLock = new Object();
//线程拥有的最小优先级
public final static int MIN_PRIORITY = 1;
//线程拥有的默认优先级
public final static int NORM_PRIORITY = 5;
```

```
//线程拥有的最大优先级
public final static int MAX_PRIORITY = 10;
```

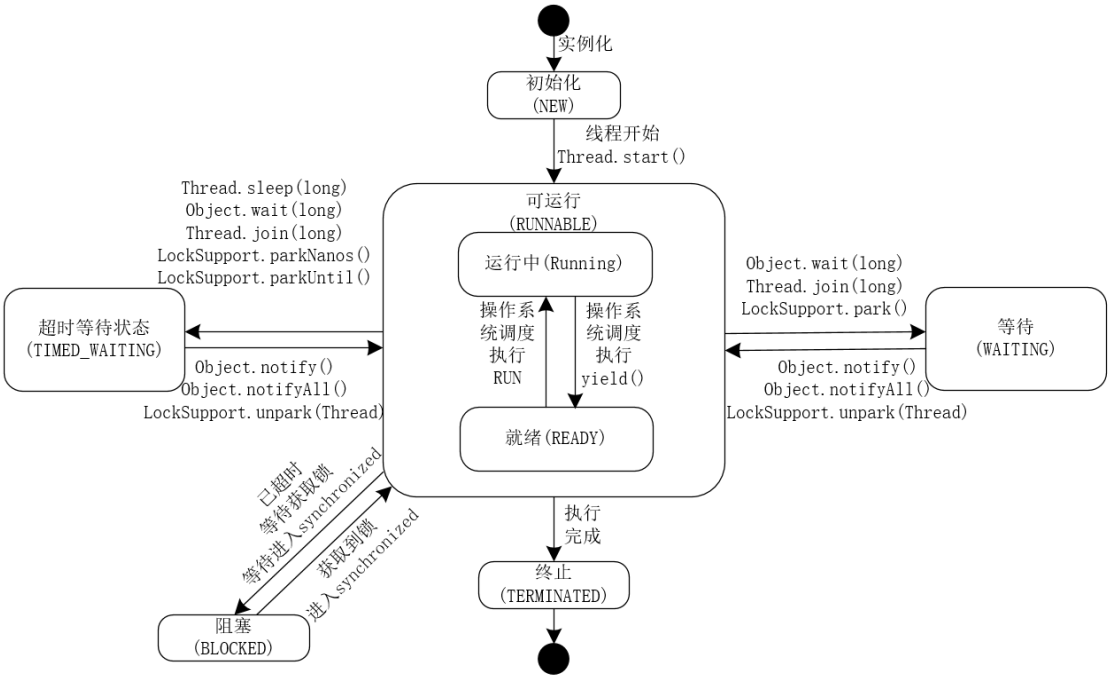
从 Thread 类的成员变量，我们可以看出，Thread 类本质上不是一个任务，它是一个实实在在的线程对象，在 Thread 类中拥有一个 Runnable 类型的成员变量 target，而这个 target 成员变量就是需要在 Thread 线程对象中执行的任务。

线程的状态定义

在 Thread 类的内部，定义了一个枚举 State，如下所示。

```
public enum State {
    //初始化状态
    NEW,
    //可运行状态，此时的可运行包括运行中的状态和就绪状态
    RUNNABLE,
    //线程阻塞状态
    BLOCKED,
    //等待状态
    WAITING,
    //超时等待状态
    TIMED_WAITING,
    //线程终止状态
    TERMINATED;
}
```

这个枚举类中的状态就代表了线程生命周期的各状态。我们可以使用下图来表示线程各个状态之间的转化关系。



NEW：初始状态，线程被构建，但是还没有调用 start()方法。

RUNNABLE：可运行状态，可运行状态可以包括：运行中状态和就绪状态。

BLOCKED：阻塞状态，处于这个状态的线程需要等待其他线程释放锁或者等待进入 synchronized。

WAITING：表示等待状态，处于该状态的线程需要等待其他线程对其进行通知或中断等操作，进而进入下一个状态。

TIME_WAITING：超时等待状态。可以在一定的时间自行返回。

TERMINATED：终止状态，当前线程执行完毕。

Thread 类的构造方法

Thread 类中的所有构造方法如下所示。

```
public Thread() {
    init(null, null, "Thread-" + nextThreadNum(), 0);
}
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
```

```
Thread(Runnable target, AccessControlContext acc) {
    init(null, target, "Thread-" + nextThreadNum(), 0, acc, false);
}
public Thread(ThreadGroup group, Runnable target) {
    init(group, target, "Thread-" + nextThreadNum(), 0);
}
public Thread(String name) {
    init(null, null, name, 0);
}
public Thread(ThreadGroup group, String name) {
    init(group, null, name, 0);
}
public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}
public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name, 0);
}
public Thread(ThreadGroup group, Runnable target, String name,
              long stackSize) {
    init(group, target, name, stackSize);
}
```

其中，我们最经常使用的就是如下几个构造方法了。

```
public Thread() {
    init(null, null, "Thread-" + nextThreadNum(), 0);
}
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
public Thread(String name) {
    init(null, null, name, 0);
}
public Thread(ThreadGroup group, String name) {
    init(group, null, name, 0);
}
public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}
```

```
public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name, 0);
}
```

通过 Thread 类的源码，我们可以看出，Thread 类在进行初始化的时候，都是调用的 init()方法，接下来，我们看看 init()方法是个啥。

init()方法

```
private void init(ThreadGroup g, Runnable target, String name, long stackSize) {
    init(g, target, name, stackSize, null, true);
}
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc,
                  boolean inheritThreadLocals) {
    //线程的名称为空，抛出空指针异常
    if (name == null) {
        throw new NullPointerException("name cannot be null");
    }

    this.name = name;
    Thread parent = currentThread();
    //获取系统安全管理器
    SecurityManager security = System.getSecurityManager();
    //线程组为空
    if (g == null) {
        //获取的系统安全管理器不为空
        if (security != null) {
            //从系统安全管理器中获取一个线程分组
            g = security.getThreadGroup();
        }
        //线程分组为空，则从父线程获取
        if (g == null) {
            g = parent.getThreadGroup();
        }
    }
    //检查线程组的访问权限
    g.checkAccess();
    //检查权限
    if (security != null) {
```

```
        if (isCCLOverridden(getClass())) {
            security.checkPermission(SUBCLASS_IMPLEMENTATION_PERMISSI
ON);
        }
    }
    g.addUnstarted();
    //当前线程继承父线程的相关属性
    this.group = g;
    this.daemon = parent.isDaemon();
    this.priority = parent.getPriority();
    if (security == null || isCCLOverridden(parent.getClass()))
        this.contextClassLoader = parent.getContextClassLoader();
    else
        this.contextClassLoader = parent.contextClassLoader;
    this.inheritedAccessControlContext =
        acc != null ? acc : AccessController.getContext();
    this.target = target;
    setPriority(priority);
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
        ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    /* Stash the specified stack size in case the VM cares */
    this.stackSize = stackSize;

    //设置线程 id
    tid = nextThreadID();
}
```

Thread 类中的构造方法是被创建 Thread 线程的线程调用的，此时，调用 Thread 的构造方法创建线程的线程就是父线程，在 init()方法中，新创建的 Thread 线程会继承父线程的部分属性。

run()方法

既然 Thread 类实现了 Runnable 接口，则 Thread 类就需要实现 Runnable 接口的 run()方法，如下所示。

```
@Override
public void run() {
    if (target != null) {
```

```
            target.run();
        }
    }
```

可以看到，Thread 类中的 run()方法实现非常简单，只是调用了 Runnable 对象的 run()方法。所以，真正的任务是运行在 run()方法中的。另外，**需要注意的是：直接调用 Runnable 接口的 run()方法不会创建新线程来执行任务，如果需要创建新线程执行任务，则需要调用 Thread 类的 start()方法。**

start()方法

```
public synchronized void start() {
    //线程不是初始化状态，则直接抛出异常
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    //添加当前启动的线程到线程组
    group.add(this);
            //标记线程是否已经启动
    boolean started = false;
    try {
        //调用本地方法启动线程
        start0();
        //将线程是否启动标记为 true
        started = true;
    } finally {
        try {
            //线程未启动成功
            if (!started) {
                //将线程在线程组里标记为启动失败
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
            /* do nothing. If start0 threw a Throwable then
                it will be passed up the call stack */
        }
    }
}

private native void start0();
```

从 start()方法的源代码，我们可以看出：**start()方法使用 synchronized 关键字修饰，说明 start()方法是同步的，它会在启动线程前检查线程的状态，如果不是初始化状态，则直接抛出异常。所以，一个线程只能启动一次，多次启动是会抛出异常的。**

这里，**也是面试的一个坑：面试官：【问题一】能不能多次调用 Thread 类的 start()方法来启动线程吗？【问题二】多次调用 Thread 线程的 start()方法会发生什么？【问题三】为什么会抛出异常？**

调用 start()方法后，新创建的线程就会处于就绪状态（如果没有分配到 CPU 执行），当有空闲的 CPU 时，这个线程就会被分配 CPU 来执行，此时线程的状态为运行状态，JVM 会调用线程的 run()方法执行任务。

sleep()方法

sleep()方法可以使当前线程休眠，其代码如下所示。

```
//本地方法，真正让线程休眠的方法
public static native void sleep(long millis) throws InterruptedException;

public static void sleep(long millis, int nanos)
    throws InterruptedException {
    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (nanos < 0 || nanos > 999999) {
        throw new IllegalArgumentException(
            "nanosecond timeout value out of range");
    }

    if (nanos >= 500000 || (nanos != 0 && millis == 0)) {
        millis++;
    }
        //调用本地方法
    sleep(millis);
}
```

sleep()方法会让当前线程休眠一定的时间，这个时间通常是毫秒值，这里需要注意的是：**调用 sleep()方法使线程休眠后，线程不会释放相应的锁。**

join()方法

join()方法会一直等待线程超时或者终止，代码如下所示。

```
public final synchronized void join(long millis)
    throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}

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

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (nanos < 0 || nanos > 999999) {
        throw new IllegalArgumentException(
            "nanosecond timeout value out of range");
```

```
    }

    if (nanos >= 500000 || (nanos != 0 && millis == 0)) {
        millis++;
    }

    join(millis);
}

public final void join() throws InterruptedException {
    join(0);
}
```

join()方法的使用场景往往是启动线程执行任务的线程，调用执行线程的 join()方法，等待执行线程执行任务，直到超时或者执行线程终止。

interrupt()方法

interrupt()方法是中断当前线程的方法，它通过设置线程的中断标志位来中断当前线程。此时，如果为线程设置了中断标志位，可能会抛出 InteruptedExeption 异常，同时，会清除当前线程的中断状态。这种方式中断线程比较安全，它能使正在执行的任务执行能够继续执行完毕，而不像 stop()方法那样强制线程关闭。代码如下所示。

```
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;
        }
    }
    //调用本地方法中断线程
    interrupt0();
```

```
}
private native void interrupt0();
```

总结

作为技术人员，要知其然，更要知其所以然，我那个朋友技术本身不错，各种框架拿来就用，基本没看过常用的框架源码和 JDK 中常用的 API，属于那种 CRUD 型程序员，这次面试就栽在了一个简单的 Thread 类上，所以，大家在学会使用的时候，一定要了解下底层的实现才好啊！

十六、AQS 中的 CountDownLatch、Semaphore 与 CyclicBarrier

CountDownLatch

概述

同步辅助类，通过它可以阻塞当前线程。也就是说，能够实现一个线程或者多个线程一直等待，直到其他线程执行的操作完成。使用一个给定的计数器进行初始化，该计数器的操作是原子操作，即同时只能有一个线程操作该计数器。

调用该类 await()方法的线程会一直阻塞，直到其他线程调用该类的 countDown()方法，使当前计数器的值变为 0 为止。每次调用该类的 countDown()方法，当前计数器的值就会减 1。当计数器的值减为 0 的时候，所有因调用 await()方法而处于等待状态的线程就会继续往下执行。这种操作只能出现一次，因为该类中的计数器不能被重置。如果需要一个可以重置计数次数的版本，可以考虑使用 CyclicBarrier 类。

CountDownLatch 支持给定时间的等待，超过一定的时间不再等待，使用时只需要在 countDown()方法中传入需要等待的时间即可。此时，countDown()方法的方法签名如下：

```
public boolean await(long timeout, TimeUnit unit)
```

使用场景

在某些业务场景中，程序执行需要等待某个条件完成后才能继续执行后续的操作。典型的应用为并行计算：当某个处理的运算量很大时，可以将该运算任务拆分成多个子任务，等待所有的子任务都完成之后，父任务再拿到所有子任务的运算结果进行汇总。

代码示例

调用 ExecutorService 类的 shutdown()方法，并不会第一时间内把所有线程全部都销毁掉，而是让当前已有的线程全部执行完，之后，再把线程池销毁掉。

示例代码如下：

```
package io.binghe.concurrency.example.aqs;
```

```
import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
@Slf4j
public class CountDownLatchExample {
    private static final int threadCount = 200;

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

        ExecutorService exec = Executors.newCachedThreadPool();
        final CountDownLatch countDownLatch = new CountDownLatch(threadCo
unt);
        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    countDownLatch.countDown();
                }
            });
        }
        countDownLatch.await();
        log.info("finish");
        exec.shutdown();
    }

    private static void test(int threadNum) throws InterruptedException {
        Thread.sleep(100);
        log.info("{}", threadNum);
        Thread.sleep(100);
    }
}
```

支持给定时间等待的示例代码如下：

```java
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
@Slf4j
public class CountDownLatchExample {
    private static final int threadCount = 200;

    public static void main(String[] args) throws InterruptedException {
        ExecutorService exec = Executors.newCachedThreadPool();
        final CountDownLatch countDownLatch = new CountDownLatch(threadCo
unt);
        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    test(threadNum);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }finally {
                    countDownLatch.countDown();
                }
            });
        }
        countDownLatch.await(10, TimeUnit.MICROSECONDS);
        log.info("finish");
        exec.shutdown();
    }

    private static void test(int threadNum) throws InterruptedException {
        Thread.sleep(100);
        log.info("{}", threadNum);
    }
}
```

Semaphore

概述

控制同一时间并发线程的数目。能够完成对于信号量的控制，可以控制某个资源可被同时访问的个数。

提供了两个核心方法——acquire()方法和 release()方法。acquire()方法表示获取一个许可，如果没有则等待，release()方法则是在操作完成后释放对应的许可。Semaphore 维护了当前访问的个数，通过提供同步机制来控制同时访问的个数。Semaphore 可以实现有限大小的链表。

使用场景

Semaphore 常用于仅能提供有限访问的资源，比如：数据库连接数。

代码示例

每次获取并释放一个许可，示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
@Slf4j
public class SemaphoreExample {
    private static final int threadCount = 200;

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

        ExecutorService exec = Executors.newCachedThreadPool();
        final Semaphore semaphore  = new Semaphore(3);

        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    semaphore.acquire(); //获取一个许可
                    test(threadNum);
```

```
                semaphore.release();  //释放一个许可
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
    exec.shutdown();
  }

  private static void test(int threadNum) throws InterruptedException {
    log.info("{}", threadNum);
    Thread.sleep(1000);
  }
}
```

每次获取并释放多个许可，示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
@Slf4j
public class SemaphoreExample {
  private static final int threadCount = 200;

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

    ExecutorService exec = Executors.newCachedThreadPool();
    final Semaphore semaphore  = new Semaphore(3);

    for (int i = 0; i < threadCount; i++){
      final int threadNum = i;
      exec.execute(() -> {
        try {
          semaphore.acquire(3);  //获取多个许可
          test(threadNum);
          semaphore.release(3);  //释放多个许可
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }
        log.info("finish");
        exec.shutdown();
    }

    private static void test(int threadNum) throws InterruptedException {
        log.info("{}", threadNum);
        Thread.sleep(1000);
    }
}
```

假设有这样一个场景，并发太高了，即使使用 Semaphore 进行控制，处理起来也比较棘手。假设系统当前允许的最高并发数是 3，超过 3 后就需要丢弃，使用 Semaphore 也能实现这样的场景，示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
@Slf4j
public class SemaphoreExample {
    private static final int threadCount = 200;

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

        ExecutorService exec = Executors.newCachedThreadPool();
        final Semaphore semaphore  = new Semaphore(3);

        for (int i = 0; i < threadCount; i++){
            final int threadNum = i;
            exec.execute(() -> {
                try {
                    //尝试获取一个许可，也可以尝试获取多个许可，
```

```
                //支持尝试获取许可超时设置，超时后不再等待后续线程的执行
                //具体可以参见 Semaphore 的源码
                if (semaphore.tryAcquire()) {
                    test(threadNum);
                    semaphore.release(); //释放一个许可
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
    }
    log.info("finish");
    exec.shutdown();
}
private static void test(int threadNum) throws InterruptedException {
    log.info("{}", threadNum);
    Thread.sleep(1000);
}
}
```

CyclicBarrier

概述

是一个同步辅助类，允许一组线程相互等待，直到到达某个公共的屏障点，通过它可以完成多个线程之间相互等待，只有当每个线程都准备就绪后，才能各自继续往下执行后面的操作。

与 CountDownLatch 有相似的地方，都是使用计数器实现，当某个线程调用了 CyclicBarrier 的 await()方法后，该线程就进入了等待状态，而且计数器执行加 1 操作，当计数器的值达到了设置的初始值，调用 await()方法进入等待状态的线程会被唤醒，继续执行各自后续的操作。CyclicBarrier 在释放等待线程后可以重用，所以，CyclicBarrier 又被称为循环屏障。

使用场景

可以用于多线程计算数据，最后合并计算结果的场景

CyclicBarrier 与 CountDownLatch 的区别

- CountDownLatch 的计数器只能使用一次，而 CyclicBarrier 的计数器可以使用 reset()方法进行重置，并且可以循环使用
- CountDownLatch 主要实现 1 个或 n 个线程需要等待其他线程完成某项操作之后，才能继续往下执行，描述的是 1 个或 n 个线程等待其他线程的关系。而 CyclicBarrier 主要实现了多个线程之间相互等待，直到所有的线程都满足了条件之后，才能继续执行后续的操作，描述的是各个线程内部相互等待的关系。
- CyclicBarrier 能够处理更复杂的场景，如果计算发生错误，可以重置计数器让线程重新执行一次。
- CyclicBarrier 中提供了很多有用的方法，比如：可以通过 getNumberWaiting()方法获取阻塞的线程数量，通过 isBroken()方法判断阻塞的线程是否被中断。

代码示例

示例代码如下。

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
@Slf4j
public class CyclicBarrierExample {

    private static CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++){
            final int threadNum = i;
            Thread.sleep(1000);
            executorService.execute(() -> {
                try {
                    race(threadNum);
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
                executorService.shutdown();
  }
  private static void race(int threadNum) throws Exception{
    Thread.sleep(1000);
    log.info("{} is ready", threadNum);
    cyclicBarrier.await();
    log.info("{} continue", threadNum);
  }
}
```

设置等待超时示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.*;
@Slf4j
public class CyclicBarrierExample {

  private static CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

  public static void main(String[] args) throws Exception {
    ExecutorService executorService = Executors.newCachedThreadPool();
    for (int i = 0; i < 10; i++){
      final int threadNum = i;
      Thread.sleep(1000);
      executorService.execute(() -> {
        try {
          race(threadNum);
        } catch (Exception e) {
          e.printStackTrace();
        }
      });
    }
```

```
        executorService.shutdown();
    }
    private static void race(int threadNum) throws Exception{
        Thread.sleep(1000);
        log.info("{} is ready", threadNum);
        try{
            cyclicBarrier.await(2000, TimeUnit.MILLISECONDS);
        }catch (BrokenBarrierException | TimeoutException e){
            log.warn("BarrierException", e);
        }
        log.info("{} continue", threadNum);
    }
}
```

在声明 CyclicBarrier 的时候，还可以指定一个 Runnable，当线程达到屏障的时候，可以优先执行 Runnable 中的方法。
示例代码如下：

```
package io.binghe.concurrency.example.aqs;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
@Slf4j
public class CyclicBarrierExample {

    private static CyclicBarrier cyclicBarrier = new CyclicBarrier(5, () -> {
        log.info("callback is running");
    });

    public static void main(String[] args) throws Exception {
        ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++){
            final int threadNum = i;
            Thread.sleep(1000);
            executorService.execute(() -> {
                try {
                    race(threadNum);
```

```
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
        executorService.shutdown();
    }
    private static void race(int threadNum) throws Exception{
        Thread.sleep(1000);
        log.info("{} is ready", threadNum);
        cyclicBarrier.await();
        log.info("{} continue", threadNum);
    }
}
```

十七、AQS 中的 ReentrantLock、ReentrantReadWriteLock、StampedLock 与 Condition

ReentrantLock

概述

Java 中主要分为两类锁，一类是 synchronized 修饰的锁，另外一类就是 J.U.C 中提供的锁。J.U.C 中提供的核心锁就是 ReentrantLock。

ReentrantLock（可重入锁）与 synchronized 区别：

（1）可重入性
二者都是同一个线程进入 1 次，锁的计数器就自增 1，需要等到锁的计数器下降为 0 时，才能释放锁。

（2）锁的实现
synchronized 是基于 JVM 实现的，而 ReentrantLock 是 JDK 实现的。

（3）性能的区别
synchronized 优化之前性能比 ReentrantLock 差很多，但是自从 synchronized 引入了偏向锁，轻量级锁也就是自旋锁后，性能就差不多了。

（4）功能区别

- 便利性

synchronized 使用起来比较方便，并且由编译器保证加锁和释放锁；ReentrantLock 需要手工声明加锁和释放锁，最好是在 finally 代码块中声明释放锁。

- 锁的灵活度和细粒度

在这点上 ReentrantLock 会优于 synchronized。

ReentrantLock 独有的功能

- ReentrantLock 可指定是公平锁还是非公平锁。而 synchronized 只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。
- 提供了一个 Condition 类，可以分组唤醒需要唤醒的线程。而 synchronized 只能随机唤醒一个线程，或者唤醒全部的线程

- 提供能够中断等待锁的线程的机制，lock.lockInterruptibly()。ReentrantLock 实现是一种自旋锁，通过循环调用 CAS 操作来实现加锁，性能上比较好是因为避免了使线程进入内核态的阻塞状态。

synchronized 能做的事情 ReentrantLock 都能做，而 ReentrantLock 有些能做的事情，synchronized 不能做。

在性能上，ReentrantLock 不会比 synchronized 差。

synchronized 的优势

- 不用手动释放锁，JVM 自动处理，如果出现异常，JVM 也会自动释放锁。
- JVM 用 synchronized 进行管理锁定请求和释放时，JVM 在生成线程转储时能够锁定信息，这些对调试非常有价值，因为它们能标识死锁或者其他异常行为的来源。而 ReentrantLock 只是普通的类，JVM 不知道具体哪个线程拥有 lock 对象。
- synchronized 可以在所有 JVM 版本中工作，ReentrantLock 在某些 1.5 之前版本的 JVM 中可能不支持。

ReentrantLock 中的部分方法说明

- boolean tryLock():仅在调用时锁定未被另一个线程保持的情况下才获取锁定。
- boolean tryLock(long, TimeUnit): 如果锁定在给定的等待时间内没有被另一个线程保持，且当前线程没有被中断，则获取这个锁定。
- void lockInterruptibly():如果当前线程没有被中断，就获取锁定；如果被中断，就抛出异常。
- boolean isLocked():查询此锁定是否由任意线程保持。
- boolean isHeldByCurrentThread(): 查询当前线程是否保持锁定状态。
- boolean isFair():判断是否是公平锁。
- boolean hasQueuedThread(Thread)：查询指定线程是否在等待获取此锁定。
- boolean hasQueuedThreads():查询是否有线程正在等待获取此锁定。
- boolean getHoldCount():查询当前线程保持锁定的个数。

代码示例

示例代码如下：

```
package io.binghe.concurrency.example.lock;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
@Slf4j
public class LockExample {
    //请求总数
    public static int clientTotal = 5000;
    //同时并发执行的线程数
    public static int threadTotal = 200;
    public static int count = 0;
    private static final Lock lock = new ReentrantLock();
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newCachedThreadPool();
        final Semaphore semaphore = new Semaphore(threadTotal);
        final CountDownLatch countDownLatch = new CountDownLatch(clientTota
l);
        for(int i = 0; i < clientTotal; i++){
            executorService.execute(() -> {
                try{
                    semaphore.acquire();
                    add();
                    semaphore.release();
                }catch (Exception e){
                    log.error("exception", e);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
```

```
        log.info("count:{}", count);
    }
    private static void add(){
        lock.lock();
        try{
            count ++;
        }finally {
            lock.unlock();
        }
    }
}
```

ReentrantReadWriteLock

概述

在没有任何读写锁的时候，才可以取得写锁。如果一直有读锁存在，则无法执行写锁，这就会导致写锁饥饿。

代码示例

示例代码如下：

```
package io.binghe.concurrency.example.lock;

import lombok.extern.slf4j.Slf4j;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
@Slf4j
public class LockExample {

    private final Map<String, Data> map = new TreeMap<>();
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock readLock = lock.readLock();
    private final Lock writeLock = lock.writeLock();

    public Data get(String key){
```

```
        readLock.lock();
        try{
            return map.get(key);
        }finally {
            readLock.unlock();
        }
    }

    public Set<String> getAllKeys(){
        readLock.lock();
        try{
            return map.keySet();
        }finally {
            readLock.unlock();
        }
    }

    public Data put(String key, Data value){
        writeLock.lock();
        try{
            return map.put(key, value);
        }finally {
            writeLock.unlock();
        }
    }

    class Data{

    }
}
```

StampedLock

概述

控制锁三种模式：写、读、乐观读。

StampedLock 的状态由版本和模式两个部分组成，锁获取方法返回的是一个数字作为票据，用相应的锁状态来表示并控制相关的访问，数字 0 表示没有写锁被授权访问。

在读锁上分为悲观锁和乐观锁，乐观读就是在读操作很多，写操作很少的情况下，可以乐观的认为写入和读取同时发生的几率很小。因此，不悲观的使用完全的读取锁定。程序可以查看读取资料之后，是否遭到写入进行了变更，再采取后续的措施，这样的改进可以大幅度提升程序的吞吐量。

总之，在读线程越来越多的场景下，StampedLock 大幅度提升了程序的吞吐量。

StampedLock 源码中的案例如下，这里加上了注释。

```
class Point {
    private double x, y;
    private final StampedLock sl = new StampedLock();

    void move(double deltaX, double deltaY) { // an exclusively locked metho
d
        long stamp = sl.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    //下面看看乐观读锁案例
    double distanceFromOrigin() { // A read-only method
        long stamp = sl.tryOptimisticRead(); //获得一个乐观读锁
        double currentX = x, currentY = y;  //将两个字段读入本地局部变量
      if (!sl.validate(stamp)) {  //检查发出乐观读锁后同时是否有其他写锁发生?
            stamp = sl.readLock();  //如果没有，我们再次获得一个读悲观锁
            try {
                currentX = x; // 将两个字段读入本地局部变量
                currentY = y; // 将两个字段读入本地局部变量
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }
```

```
        //下面是悲观读锁案例
        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) { //循环，检查当前状态是否符合
                                long ws = sl.tryConvertToWriteLock(stamp); //将读锁
转为写锁
                                if (ws != 0L) { //这是确认转为写锁是否成功
                                        stamp = ws; //如果成功 替换票据
                                        x = newX; //进行状态改变
                                        y = newY;  //进行状态改变
                                        break;
                                } else { //如果不能成功转换为写锁
                                        sl.unlockRead(stamp);  //我们显式释放读锁
                                        stamp = sl.writeLock();  //显式直接进行写锁 然后
再通过循环再试
                                }
                        }
                } finally {
                        sl.unlock(stamp); //释放读锁或写锁
                }
        }
}
```

代码示例

示例代码如下：

```
package io.binghe.concurrency.example.lock;
import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.StampedLock;
@Slf4j
public class LockExample {
   //请求总数
```

```
    public static int clientTotal = 5000;
    //同时并发执行的线程数
    public static int threadTotal = 200;

    public static int count = 0;

    private static final StampedLock lock = new StampedLock();

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newCachedThreadPool();
        final Semaphore semaphore = new Semaphore(threadTotal);
        final CountDownLatch countDownLatch = new CountDownLatch(clientTota
l);
        for(int i = 0; i < clientTotal; i++){
            executorService.execute(() -> {
                try{
                    semaphore.acquire();
                    add();
                    semaphore.release();
                }catch (Exception e){
                    log.error("exception", e);
                }
                countDownLatch.countDown();
            });
        }
        countDownLatch.await();
        executorService.shutdown();
        log.info("count:{}", count);
    }

    private static void add(){
        //加锁时返回一个 long 类型的票据
        long stamp = lock.writeLock();
        try{
            count ++;
        }finally {
            //释放锁的时候带上加锁时返回的票据
            lock.unlock(stamp);
        }
```

```
    }
}
```

我们可以这样选择使用 synchronozed 锁还是 ReentrantLock 锁：

- 当只有少量竞争者时，synchronized 是一个很好的通用锁实现
- 竞争者不少，但是线程的增长趋势是可预估的，此时，ReentrantLock 是一个很好的通用锁实现
- synchronized 不会引发死锁，其他的锁使用不当可能会引发死锁。

Condition

概述

Condition 是一个多线程间协调通信的工具类，Condition 除了实现 wait 和 notify 的功能以外，它的好处在于一个 lock 可以创建多个 Condition，可以选择性的通知 wait 的线程

特点：

- Condition 的前提是 Lock，由 AQS 中 newCondition()方法 创建 Condition 的对象
- Condition await 方法表示线程从 AQS 中移除，并释放线程获取的锁，并进入 Condition 等待队列中等待，等待被 signal
- Condition signal 方法表示唤醒对应 Condition 等待队列中的线程节点，并加入 AQS 中，准备去获取锁。

代码示例

示例代码如下

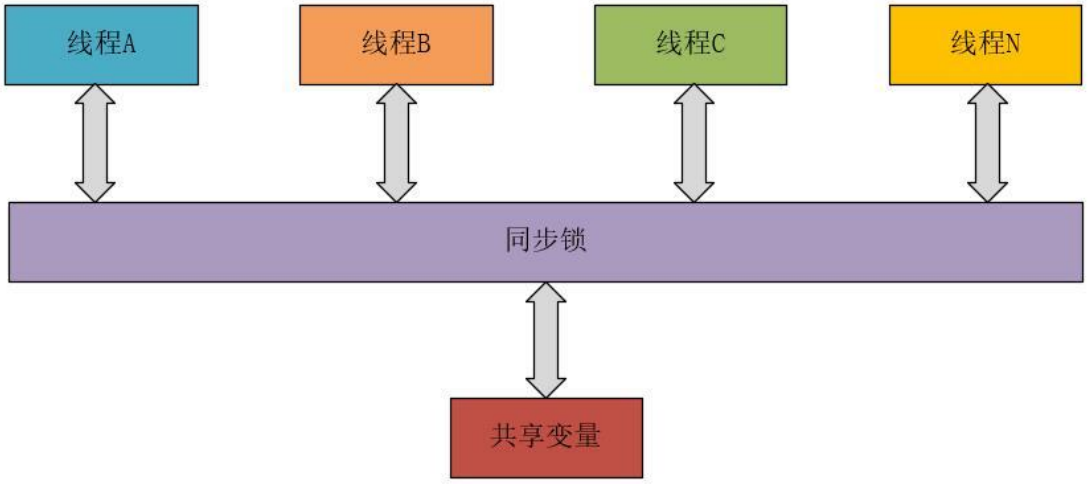
```
package io.binghe.concurrency.example.lock;

import lombok.extern.slf4j.Slf4j;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
@Slf4j
public class LockExample {
    public static void main(String[] args) {
```

```
        ReentrantLock reentrantLock = new ReentrantLock();
        Condition condition = reentrantLock.newCondition();

        new Thread(() -> {
            try {
                reentrantLock.lock();
                log.info("wait signal"); // 1
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.info("get signal"); // 4
            reentrantLock.unlock();
        }).start();

        new Thread(() -> {
            reentrantLock.lock();
            log.info("get lock"); // 2
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            condition.signalAll();
            log.info("send signal ~ "); // 3
            reentrantLock.unlock();
        }).start();
    }
}
```

十八、ThreadLocal 学会了这些，你也能和面试官扯皮了！

前言

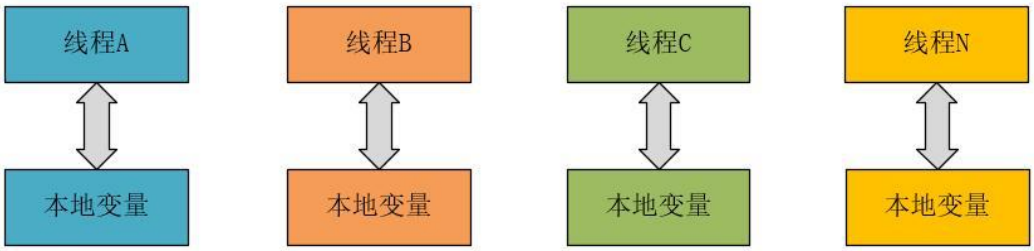
我们都知道，在多线程环境下访问同一个共享变量，可能会出现线程安全的问题，为了保证线程安全，我们往往会在访问这个共享变量的时候加锁，以达到同步的效果，如下图所示。



对共享变量加锁虽然能够保证线程的安全，但是却增加了开发人员对锁的使用技能，如果锁使用不当，则会导致死锁的问题。而 ThreadLocal **能够做到在创建变量后，每个线程对变量访问时访问的是线程自己的本地变量**。

什么是 ThreadLocal?

> ThreadLocal 是 JDK 提供的，支持线程本地变量。也就是说，如果我们创建了一个 ThreadLocal 变量，则访问这个变量的每个线程都会有这个变量的一个本地副本。如果多个线程同时对这个变量进行读写操作时，实际上操作的是线程自己本地内存中的变量，从而避免了线程安全的问题。



ThreadLocal 使用示例

例如，我们使用 ThreadLocal 保存并打印相关的变量信息，程序如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //创建第一个线程
        Thread threadA = new Thread(()->{
            threadLocal.set("ThreadA：" + Thread.currentThread().getName());
            System.out.println("线程 A 本地变量中的值为：" + threadLocal.get());
        });
        //创建第二个线程
        Thread threadB = new Thread(()->{
            threadLocal.set("ThreadB：" + Thread.currentThread().getName());
            System.out.println("线程 B 本地变量中的值为：" + threadLocal.get());
        });
        //启动线程 A 和线程 B
        threadA.start();
        threadB.start();
    }
}
```

运行程序，打印的结果信息如下所示。

```
线程 A 本地变量中的值为：ThreadA：Thread-0
线程 B 本地变量中的值为：ThreadB：Thread-1
```

此时，我们为线程 A 增加删除 ThreadLocal 中的变量的操作，如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //创建第一个线程
        Thread threadA = new Thread(()->{
            threadLocal.set("ThreadA：" + Thread.currentThread().getName());
            System.out.println("线程 A 本地变量中的值为：" + threadLocal.get());
            threadLocal.remove();
            System.out.println("线程 A 删除本地变量后 ThreadLocal 中的值为：" + threa
dLocal.get());
        });
        //创建第二个线程
        Thread threadB = new Thread(()->{
            threadLocal.set("ThreadB：" + Thread.currentThread().getName());
            System.out.println("线程 B 本地变量中的值为：" + threadLocal.get());
            System.out.println("线程 B 没有删除本地变量：" + threadLocal.get());
        });
        //启动线程 A 和线程 B
        threadA.start();
        threadB.start();
    }
}
```

此时的运行结果如下所示。

```
线程 A 本地变量中的值为：ThreadA：Thread-0
线程 B 本地变量中的值为：ThreadB：Thread-1
线程 B 没有删除本地变量：ThreadB：Thread-1
线程 A 删除本地变量后 ThreadLocal 中的值为：null
```

通过上述程序我们可以看出，**线程 A 和线程 B 存储在 ThreadLocal 中的变量互不干扰，线程 A 存储的变量只能由线程 A 访问，线程 B 存储的变量只能由线程 B 访问。**

ThreadLocal 原理

首先，我们看下 Thread 类的源码，如下所示。

```
public class Thread implements Runnable {
    /**********省略 N 行代码 **********/
    ThreadLocal.ThreadLocalMap threadLocals = null;
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
    /**********省略 N 行代码 **********/
}
```

由 Thread 类的源码可以看出，在 ThreadLocal 类中存在成员变量 threadLocals 和 inheritableThreadLocals，这两个成员变量都是

ThreadLocalMap 类型的变量，而且二者的初始值都为 null。只有当前线程第一次调用 ThreadLocal 的 set()方法或者 get()方法时才会实例化变量。

这里需要注意的是：**每个线程的本地变量不是存放在 ThreadLocal 实例里面的，而是存放在调用线程的 threadLocals 变量里面的。**也就是说，调用 ThreadLocal 的 set()方法存储的本地变量是存放在具体线程的内存空间中的，而 ThreadLocal 类只是提供了 set()和 get()方法来存储和读取本地变量的值，当调用 ThreadLocal 类的 set()方法时，把要存储的值放入**调用线程**的 threadLocals 中存储起来，当调用 ThreadLocal 类的 get()方法时，从当前线程的 threadLocals 变量中将存储的值取出来。

接下来，我们分析下 ThreadLocal 类的 set()、get()和 remove()方法的实现逻辑。

set()方法

set()方法的源代码如下所示。

```
public void set(T value) {
   //获取当前线程
   Thread t = Thread.currentThread();
   //以当前线程为 Key，获取 ThreadLocalMap 对象
   ThreadLocalMap map = getMap(t);
   //获取的 ThreadLocalMap 对象不为空
   if (map != null)
      //设置 value 的值
      map.set(this, value);
   else
      //获取的 ThreadLocalMap 对象为空，创建 Thread 类中的 threadLocals 变量
      createMap(t, value);
}
```

在 set()方法中，首先获取调用 set()方法的线程，接下来，使用当前线程作为 Key 调用 getMap(t)方法来获取 ThreadLocalMap 对象，getMap(Thread t)的方法源码如下所示。

```
ThreadLocalMap getMap(Thread t) {
   return t.threadLocals;
}
```

可以看到，getMap(Thread t)方法获取的是线程变量自身的 threadLocals 成员变量。

在 set()方法中，如果调用 getMap(t)方法返回的对象不为空，则把 value 值设置到 Thread 类的 threadLocals 成员变量中，而传递的 key 为当前 ThreadLocal 的 this 对象，value 就是通过 set()方法传递的值。

如果调用 getMap(t)方法返回的对象为空，则程序调用 createMap(t, value)方法来实例化 Thread 类的 threadLocals 成员变量。

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

也就是创建当前线程的 threadLocals 变量。

get()方法

get()方法的源代码如下所示。

```
public T get() {
    //获取当前线程
    Thread t = Thread.currentThread();
    //获取当前线程的 threadLocals 成员变量
    ThreadLocalMap map = getMap(t);
    //获取的 threadLocals 变量不为空
    if (map != null) {
        //返回本地变量对应的值
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    //初始化 threadLocals 成员变量的值
    return setInitialValue();
}
```

通过当前线程来获取 threadLocals 成员变量，如果 threadLocals 成员变量不为空，则直接返回当前线程绑定的本地变量，否则调用 setInitialValue()方法初始化 threadLocals 成员变量的值。

```
private T setInitialValue() {
    //调用初始化 Value 的方法
    T value = initialValue();
    Thread t = Thread.currentThread();
    //根据当前线程获取 threadLocals 成员变量
    ThreadLocalMap map = getMap(t);
    if (map != null)
        //threadLocals 不为空，则设置 value 值
        map.set(this, value);
    else
        //threadLocals 为空,创建 threadLocals 变量
        createMap(t, value);
    return value;
}
```

其中，initialValue()方法的源码如下所示。

```
protected T initialValue() {
    return null;
}
```

通过 initialValue()方法的源码可以看出，这个方法可以由子类覆写，在 ThreadLocal 类中，这个方法直接返回 null。

remove()方法

remove()方法的源代码如下所示。

```
public void remove() {
    //根据当前线程获取 threadLocals 成员变量
    ThreadLocalMap m = getMap(Thread.currentThread());
    if (m != null)
        //threadLocals 成员变量不为空，则移除 value 值
        m.remove(this);
}
```

remove()方法的实现比较简单，首先根据当前线程获取 threadLocals 成员变量，不为空，则直接移除 value 的值。

注意：如果调用线程一致不终止，则本地变量会一直存放在调用线程的 threadLocals 成员变量中，所以，如果不需要使用本地变量时，可以通过调用 ThreadLocal 的 remove()方法，将本地变量从当前线程的 threadLocals 成员变量中删除，以免出现内存溢出的问题。

ThreadLocal 变量不具有传递性

使用 ThreadLocal 存储本地变量不具有传递性，也就是说，同一个 ThreadLocal 在父线程中设置值后，在子线程中是无法获取到这个值的，这个现象说明 ThreadLocal 中存储的本地变量不具有传递性。

接下来，我们来看一段代码，如下所示。

```
public class ThreadLocalTest {

    private static ThreadLocal<String> threadLocal = new ThreadLocal<String>();

    public static void main(String[] args){
        //在主线程中设置值
        threadLocal.set("ThreadLocalTest");
        //在子线程中获取值
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
```

```
            System.out.println("子线程获取值：" + threadLocal.get());
        }
    });
    //启动子线程
    thread.start();
    //在主线程中获取值
    System.out.println("主线程获取值：" + threadLocal.get());
  }
}
```

运行这段代码输出的结果信息如下所示。

```
主线程获取值：ThreadLocalTest
子线程获取值：null
```

通过上述程序，我们可以看出在主线程中向 ThreadLocal 设置值后，在子线程中是无法获取到这个值的。**那有没有办法在子线程中获取到主线程设置的值呢？此时，我们可以使用 InheritableThreadLocal 来解决这个问题。**

InheritableThreadLocal 使用示例

InheritableThreadLocal 类继承自 ThreadLocal 类，它能够让子线程访问到在父线程中设置的本地变量的值，例如，我们将 ThreadLocalTest 类中的 threadLocal 静态变量改写成 InheritableThreadLocal 类的实例，如下所示。

```
public class ThreadLocalTest {

  private static ThreadLocal<String> threadLocal = new InheritableThreadLocal
<String>();

  public static void main(String[] args){
    //在主线程中设置值
    threadLocal.set("ThreadLocalTest");
    //在子线程中获取值
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        System.out.println("子线程获取值：" + threadLocal.get());
      }
    });
    //启动子线程
```

```
        thread.start();
        //在主线程中获取值
        System.out.println("主线程获取值：" + threadLocal.get());
    }
}
```

此时，运行程序输出的结果信息如下所示。

```
主线程获取值：ThreadLocalTest
子线程获取值：ThreadLocalTest
```

可以看到，**使用 InheritableThreadLocal 类存储本地变量时，子线程能够获取到父线程中设置的本地变量。**

InheritableThreadLocal 原理

首先，我们来看下 InheritableThreadLocal 类的源码，如下所示。

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {
    protected T childValue(T parentValue) {
        return parentValue;
    }

    ThreadLocalMap getMap(Thread t) {
       return t.inheritableThreadLocals;
    }
```

```
    void createMap(Thread t, T firstValue) {
        t.inheritableThreadLocals = new ThreadLocalMap(this, firstValue);
    }
}
```

由 InheritableThreadLocal 类的源代码可知，InheritableThreadLocal 类继承自 ThreadLocal 类，并且重写了 ThreadLocal 类的 childValue()方法、getMap()方法和 createMap()方法。也就是说，当调用 ThreadLocal 的 set()方法时，创建的是当前 Thread 线程的 inheritableThreadLocals 成员变量而不再是 threadLocals 成员变量。

这里，**我们需要思考一个问题：InheritableThreadLocal 类的 childValue()方法是何时被调用的呢？**这就需要我们来看下 Thread 类的构造方法了，如下所示。

```
public Thread() {
    init(null, null, "Thread-" + nextThreadNum(), 0);
}

public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}

Thread(Runnable target, AccessControlContext acc) {
    init(null, target, "Thread-" + nextThreadNum(), 0, acc, false);
}

public Thread(ThreadGroup group, Runnable target) {
    init(group, target, "Thread-" + nextThreadNum(), 0);
}

public Thread(String name) {
    init(null, null, name, 0);
}

public Thread(ThreadGroup group, String name) {
    init(group, null, name, 0);
}
```

```
public Thread(Runnable target, String name) {
    init(null, target, name, 0);
}

public Thread(ThreadGroup group, Runnable target, String name) {
    init(group, target, name, 0);
}

public Thread(ThreadGroup group, Runnable target, String name,
              long stackSize) {
    init(group, target, name, stackSize);
}
```

可以看到，Thread 类的构造方法最终调用的是 init()方法，那我们就来看下 init()方法，如下所示。

```
private void init(ThreadGroup g, Runnable target, String name,
                  long stackSize, AccessControlContext acc,
                  boolean inheritThreadLocals) {
    /************省略部分源码 ************/
    if (inheritThreadLocals && parent.inheritableThreadLocals != null)
        this.inheritableThreadLocals =
            ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);
    /* Stash the specified stack size in case the VM cares */
    this.stackSize = stackSize;

    /* Set thread ID */
    tid = nextThreadID();
}
```

可以看到，在 init()方法中会判断传递的 inheritThreadLocals 变量是否为 true，同时父线程中的 inheritableThreadLocals 是否为 null，如果传递的 inheritThreadLocals 变量为 true，同时，父线程中的 inheritableThreadLocals 不为 null，则调用 ThreadLocal 类的 createInheritedMap()方法。

```
static ThreadLocalMap createInheritedMap(ThreadLocalMap parentMap) {
    return new ThreadLocalMap(parentMap);
}
```

在 createInheritedMap()中，使用父线程的 inheritableThreadLocals 变量作为参数创建新的 ThreadLocalMap 对象。然后在 Thread 类的 init()方法中会将这个 ThreadLocalMap 对象赋值给子线程的 inheritableThreadLocals 成员变量。

接下来，我们来看看 ThreadLocalMap 的构造函数都干了啥，如下所示。

```
private ThreadLocalMap(ThreadLocalMap parentMap) {
    Entry[] parentTable = parentMap.table;
    int len = parentTable.length;
    setThreshold(len);
    table = new Entry[len];

    for (int j = 0; j < len; j++) {
        Entry e = parentTable[j];
        if (e != null) {
            @SuppressWarnings("unchecked")
            ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();
            if (key != null) {
                //调用重写的 childValue 方法
                Object value = key.childValue(e.value);
                Entry c = new Entry(key, value);
                int h = key.threadLocalHashCode & (len - 1);
                while (table[h] != null)
                    h = nextIndex(h, len);
                table[h] = c;
                size++;
            }
        }
    }
}
```

在 ThreadLocalMap 的构造函数中，调用了 InheritableThreadLocal 类重写的 childValue()方法。而 InheritableThreadLocal 类通过重写 getMap()方法和 createMap()方法，让本地变量保存到了 Thread 线程的 inheritableThreadLocals 变量中，线程通过 InheritableThreadLocal 类的 set()方法和 get()方法设置变量时，就会创建当前线程的 inheritableThreadLocals 变量。此时，如果父线程创建子线程，在 Thread 类

的构造函数中会把父线程中的 inheritableThreadLocals 变量里面的本地变量复制一份保存到子线程的 inheritableThreadLocals 变量中。

十九、又一个朋友面试栽在了 Thread 类上！

一个工作了几年的朋友今天打电话和我聊天，说前段时间出去面试，面试官问他做过的项目，他讲起业务来那是头头是道，犹如滔滔江水连绵不绝，可面试官最后问了一个问题：Thread 类的 stop()方法和 interrupt 方法有啥区别。这一问不要紧，当场把那个朋友打懵了！结果可想而知。。。

事后，我也是感慨颇多，现在的程序员只知道做些简单的 CRUD 吗？哎，不多说了，今天就简单的说说 Thread 类的 stop()方法和 interrupt()方法到底有啥区别吧！

stop()方法

stop()方法会真的杀死线程。如果线程持有 ReentrantLock 锁，被 stop()的线程并不会自动调用 ReentrantLock 的 unlock()去释放锁，那其他线程就再也没机会获得 ReentrantLock 锁， 这样其他线程就再也不能执行 ReentrantLock 锁锁住的代码逻辑。 所以该方法就不建议使用了， 类似的方法还有 suspend()和 resume()方法， 这两个方法同样也都不建议使用了， 所以这里也就不多介绍了。

interrupt()方法

interrupt()方法仅仅是通知线程，线程有机会执行一些后续操作，同时也可以无视这个通知。被 interrupt 的线程，有两种方式接收通知：**一种是异常， 另一种是主动检测。**

通过异常接收通知

当线程 A 处于 WAITING、 TIMED*WAITING 状态时， 如果其他线程调用线程 A 的 interrupt()方法，则会使线程 A 返回到 RUNNABLE 状态，同时线程 A 的代码会触发 InterruptedException 异常。线程转换到 WAITING*、*TIMED*WAITING 状态的触发条件，都是调用了类似 wait()、join()、sleep()这样的方法， 我们看这些方法的签名时，发现都会 throws InterruptedException 这个异常。这个异常的触发条件就是：其他线程调用了该线程的 interrupt()方法。

当线程 A 处于 RUNNABLE 状态时，并且阻塞在 java.nio.channels.InterruptibleChannel 上时， 如果其他线程调用线程 A 的 interrupt()方法，线程 A 会触发 java.nio.channels.ClosedByInterruptException 这个异常；当阻塞在 java.nio.channels.Selector 上

时，如果其他线程调用线程 A 的 interrupt()方法，线程 A 的 java.nio.channels.Selector 会立即返回。

主动检测通知

如果线程处于 RUNNABLE 状态，并且没有阻塞在某个 I/O 操作上，例如中断计算基因组序列的线程 A，此时就得依赖线程 A 主动检测中断状态了。如果其他线程调用线程 A 的 interrupt()方法， 那么线程 A 可以通过 isInterrupted()方法，来检测自己是不是被中断了。

本书完

感谢阅读

HUAWEI

华为云开发者社区

更多精彩内容，扫码关注交流讨论

华为云开发者社区

华为云开发者联盟公众号