

- 前言
- 业务需求
- 项目示例
- 线程池
- 单个任务
- 任务入口
- 结果分析
- 结语
- 学习交流



主要基于小米最近的多线程项目，剥离出里面的多线程实例。

昨天听到袁老去世的消息，非常震惊，一代巨星的陨落，希望袁老在天堂能安好，贴了一张袁老的照片，这张照片也让我想起已在天堂的爷爷。

## 前言

Java多线程的学习，也有大半个月了，从开始学习Java多线程时，就给自己定了一个小目标，希望能写一个多线程的Demo，今天主要是兑现这个小目标。

这个多线程的示例，其实是结合最近小米的一个多线程异步任务的项目，我把里面涉及到多线程的代码剥离出来，然后进行一定的改造，之所以不自己原创一个，一方面是自己能力还不够，另一方面是想学习现在项目中多线程的实现姿势，至少这个示例是实际项目中应用的，先学习别人怎么造轮子，后面就知道自己怎么去造轮子了。

## 业务需求

做这个多线程异步任务，主要是因为我们有很多永动的异步任务，什么是永动呢？就是任务跑起来后，需要一直跑下去。比如消息Push任务，因为一直有消息过来，所以需要一直去消费DB中的未推送消息，就需要整一个Push的永动异步任务。

我们的需求其实不难，简单总结一下：

- 能同时执行多个永动的异步任务；
- 每个异步任务，支持开多个线程去消费这个任务的数据；
- 支持永动异步任务的优雅关闭，即关闭后，需要把所有的数据消费完毕后，再关闭。

注意上面的需求，需要注意几个点：

- 每个永动任务，可以开一个线程去执行；
- 每个子任务，因为需要支持并发，需要用线程池控制；
- 永动任务的关闭，需要通知子任务的并发线程，并支持永动任务和并发子任务的优雅关闭。

## 项目示例

### 线程池

对于子任务，需要支持并发，如果每个并发都开一个线程，用完就关闭，对资源消耗太大，所以引入线程池：

```
public class TaskProcessUtil {
    // 每个任务，都有自己单独的线程池
    private static Map<String, ExecutorService> executors = new ConcurrentHashMap<>();

    // 初始化一个线程池
    private static ExecutorService init(String poolName, int poolSize) {
        return new ThreadPoolExecutor(poolSize, poolSize,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>(),
            new ThreadFactoryBuilder().setNameFormat("Pool-" + poolName).setDaemon(false).build(),
            new ThreadPoolExecutor.CallerRunsPolicy());
    }

    // 获取线程池
    public static ExecutorService getOrInitExecutors(String poolName, int poolSize) {
        ExecutorService executorService = executors.get(poolName);
        if (null == executorService) {
            synchronized (TaskProcessUtil.class) {
                executorService = executors.get(poolName);
                if (null == executorService) {
                    executorService = init(poolName, poolSize);
                    executors.put(poolName, executorService);
                }
            }
        }
        return executorService;
    }

    // 回收线程资源
    public static void releaseExecutors(String poolName) {
        ExecutorService executorService = executors.remove(poolName);
        if (executorService != null) {
            executorService.shutdown();
        }
    }
}
```

这是一个线程池的工具类，这里初始化线程池和回收线程资源很简单，我们主要讨论获取线程池。获取线程池可能存在并发情况，所以需要加一个synchronized锁，然后锁住后，需要对executorService进行二次判空校验，这个和Java单例的实现很像，具体可参考《【设计模式系列5】单例模式》这篇文章。

### 单个任务

为了更好地讲解单个任务的实现方式，我们的任务主要就是把Cat的数据打印出来，Cat定义如下：

```
@Data
@Service
public class Cat {
    private String catName;
    public Cat setCatName(String name) {
        this.catName = name;
        return this;
    }
}
```

单个任务主要包括以下功能：

- 获取永动任务数据：这里一般都是扫描DB，我直接就简单用queryData()代替。
- 多线程执行任务：需要把数据拆分成4份，然后分别由多线程并发执行，这里可以通过线程池支持；
- 永动任务优雅停机：当外面通知任务需要停机，需要执行完剩余任务数据，并回收线程资源，退出任务；
- 永动执行：如果未收到停机命令，任务需要一直执行下去。

直接看代码：

```
public class ChildTask {

    private final int POOL_SIZE = 3; // 线程池大小
    private final int SPLIT_SIZE = 4; // 数据拆分大小
    private String taskName;

    // 接收jvm关闭信号，实现优雅停机
    protected volatile boolean terminal = false;

    public ChildTask(String taskName) {
        this.taskName = taskName;
    }

    // 程序执行入口
    public void doExecute() {
        int i = 0;
        while(true) {
            System.out.println(taskName + ":Cycle-" + i + "-Begin");
            // 获取数据
            List<Cat> datas = queryData();
            // 处理数据
            taskExecute(datas);
            System.out.println(taskName + ":Cycle-" + i + "-End");
            if (terminal) {
                // 只有应用关闭，才会走到这里，用于实现优雅的下线
                break;
            }
            i++;
        }
        // 回收线程池资源
        TaskProcessUtil.releaseExecutors(taskName);
    }

    // 优雅停机
    public void terminal() {
        // 关闭
        terminal = true;
        System.out.println(taskName + " shut down");
    }

    // 处理数据
    public void doProcessData(List<Cat> datas, CountDownLatch latch) {
        try {
            for (Cat cat : datas) {
                System.out.println(":" + cat.toString() + ", ThreadName:" + Thread.currentThread().getName());
                Thread.sleep(1000L);
            }
        } catch (Exception e) {
            System.out.println(e.getStackTrace());
        } finally {
            if (latch != null) {
                latch.countDown();
            }
        }
    }

    // 处理单个任务数据
    private void taskExecute(List<Cat> sourceDatas) {
        if (CollectionUtils.isEmpty(sourceDatas)) {
            return;
        }
        // 将数据拆成4份
        List<List<Cat>> splitDatas = Lists.partition(sourceDatas, SPLIT_SIZE);
        final CountDownLatch latch = new CountDownLatch(splitDatas.size());

        // 并发处理拆分的数据，共用一个线程池
        for (final List<Cat> datas : splitDatas) {
            ExecutorService executorService = TaskProcessUtil.getOrInitExecutors(taskName, POOL_SIZE);
            executorService.submit(new Runnable() {
                @Override
                public void run() {
                    doProcessData(datas, latch);
                }
            });
        }
        try {
            latch.await();
        } catch (Exception e) {
            System.out.println(e.getStackTrace());
        }
    }

    // 获取永动任务数据
    private List<Cat> queryData() {
        List<Cat> datas = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            datas.add(new Cat().setCatName("罗小黑" + i));
        }
        return datas;
    }
}
```

简单解释一下：

- queryData：用于获取数据，实际应用中其实是需要把queryData定为抽象方法，然后由各个任务实现自己的方法。
- doProcessData：数据处理逻辑，实际应用中其实是需要把doProcessData定为抽象方法，然后由各个任务实现自己的方法。
- taskExecute：有数据拆分成4份，获取该任务的线程池，并交给线程池并发执行，然后通过latch.await()阻塞，当这4份数据都执行成功后，阻塞结束，该方法才返回。
- terminal：仅用于接收停机命令，这里我定义为volatile，所以多线程内存可见，详见《[Java并发编程系列2] volatile》；
- doExecute：程序执行入口，封装了每个任务执行的流程，当terminal=true时，先执行完任务数据，然后回收线程池，最后退出。

## 任务入口

直接上代码：

```
public class LoopTask {

    private List<ChildTask> childTasks;

    public void initLoopTask() {
        childTasks = new ArrayList<>();
        childTasks.add(new ChildTask("childTask1"));
        childTasks.add(new ChildTask("childTask2"));
        for (final ChildTask childTask : childTasks) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    childTask.doExecute();
                }
            }).start();
        }

        public void shutdownLoopTask() {
            if (!CollectionUtils.isEmpty(childTasks)) {
                for (ChildTask childTask : childTasks) {
                    childTask.terminal();
                }
            }
        }

        public static void main(String args[]) throws Exception{
            LoopTask loopTask = new LoopTask();
            loopTask.initLoopTask();
            Thread.sleep(5000L);
            loopTask.shutdownLoopTask();
        }
}
```

每个任务都开一个单独的Thread，这里我初始化了2个永动任务，分别为childTask1和childTask2，然后分别执行，后面Sleep了5秒后，再关闭任务，我们可以看看是否可以按照我们的预期优雅退出。

## 结果分析

执行结果如下：

```
childTask1:Cycle-0-Begin
childTask2:Cycle-0-Begin
childTask1:Cat(catName=罗小黑0),ThreadName:Pool-childTask1
childTask1:Cat(catName=罗小黑0),ThreadName:Pool-childTask1
childTask2:Cat(catName=罗小黑0),ThreadName:Pool-childTask2
childTask2:Cat(catName=罗小黑0),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑1),ThreadName:Pool-childTask1
childTask2:Cat(catName=罗小黑1),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑2),ThreadName:Pool-childTask1
childTask2:Cat(catName=罗小黑2),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑3),ThreadName:Pool-childTask1
childTask2:Cat(catName=罗小黑3),ThreadName:Pool-childTask2
childTask1:Cycle-0-End
childTask2:Cycle-0-End
childTask1:Cycle-0-End
childTask1:Cycle-0-End
childTask1:Cycle-1-Begin
childTask2:Cycle-1-Begin
childTask2:Cat(catName=罗小黑0),ThreadName:Pool-childTask2
childTask2:Cat(catName=罗小黑0),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑0),ThreadName:Pool-childTask1
childTask1:Cat(catName=罗小黑0),ThreadName:Pool-childTask1
childTask1 shut down
childTask2 shut down
childTask2:Cat(catName=罗小黑1),ThreadName:Pool-childTask2
childTask2:Cat(catName=罗小黑1),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑2),ThreadName:Pool-childTask1
childTask2:Cat(catName=罗小黑2),ThreadName:Pool-childTask2
childTask1:Cat(catName=罗小黑3),ThreadName:Pool-childTask1
childTask2:Cat(catName=罗小黑3),ThreadName:Pool-childTask2
childTask1:Cycle-1-End
childTask2:Cycle-1-End
```

输出数据中，“Pool-childTask”是线程池名称，“childTask”是任务名称，“Cat(catName=罗小黑)”是执行的结果，“childTask shut down”是关闭标记，“childTask:Cycle-X-Begin”和“childTask:Cycle-X-End”是每一轮循环的开始和结束标记。

我们分析一下下执行结果：childTask1和childTask2分别执行，在第一轮循环中都正常输出了5条罗小黑数据，第二轮执行过程中，我启动了关闭指令，这次第二轮执行没有直接停止，而是先执行完任务中的数据，再执行退出，所以完全符合我们的优雅退出结论。

## 结语

这是一个比较经典的线程池使用示例，是我们公司的一位同事写的，感觉整个流程没有毛病，实现的也非常优雅，非常值得我学习的。

然后学习Java多线程的过程中，我感觉我目前的掌握速度还算是比较快的，从Java内存模型、到Java多线程的基本知识和常用工具，到后面的多线程实践，一共8篇文章，真的是可以让你从Java小白到能写出比较健壮的多线程程序。

其实之前学习语言或者技术，更多是偏向看一些八股文，其实八股文要看，更重要的是自己实践，需要多写，所以之前的文章很多是纯理论，现在更多是理论和实践相结合，那怕是看到网上的一些示例，我都会Copy一下，后面再跑一遍才安心。

Java多线程部分，后面打算再写1-2篇文章，这个系列就先暂停，因为我的目标是把Java生态的相关技术都学完，所以先尽快吃一遍，等全部学习完后，再重点学习更深入的知识。

## 学习交流

可以扫下面二维码，关注「楼仔」公众号。



### 一枚小小的Go/Java代码搬运工

获取更多干货，包括Java、Go、消息中间件、ETCD、MySQL、Redis、RPC、DDD等后端常用技术，并对管理、职业规划、业务也有深度思考。



**扫一扫 长按 关注我** 让你懂技术、懂管理、懂业务，也懂生活

长按二维码，回复「加群」，欢迎一起学习交流咯~~ 🍻🍻🍻



**楼仔**  
湖北 武汉

扫一扫 长按  
加技术群的备注：加群



尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。