jdk拓展线程池增加监控

> 摘自: http://lixiaohui.iteye.com/blog/2330086
>
> MonitorableThreadPoolExecutor类代码如下:

```
package com.sitech.crm_cmi.util.concurrent;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * 摘自: http://lixiaohui.iteye.com/blog/2330086
 * 可监控的线程池, 可有多个监控处理器,如果监控的逻辑是比较耗时的话,最好另起个线程或者线程池专门用来跑MonitorHandler的方法.
 */
public class MonitorableThreadPoolExecutor extends ThreadPoolExecutor {
    /**
     * 可有多个监控处理器
     */
    private final ConcurrentMap<String, MonitorHandler> handlerMap = new ConcurrentHashMap<String, MonitorHandler>();

    public MonitorableThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
            long keepAliveTime, TimeUnit unit,
            BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }

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

    public MonitorableThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
            long keepAliveTime, TimeUnit unit,
            BlockingQueue<Runnable> workQueue,
            RejectedExecutionHandler handler) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
                handler);
    }

    public MonitorableThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
            long keepAliveTime, TimeUnit unit,
            BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
            RejectedExecutionHandler handler) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
                threadFactory, handler);
    }

    @Override
    protected void beforeExecute(Thread t, Runnable r) {
        super.beforeExecute(t, r);
        // 依次调用处理器
        for (MonitorHandler handler : handlerMap.values())
            if (handler.usable())
                handler.before(t, r);
    }

    @Override
    protected void afterExecute(Runnable r, Throwable t) {
        super.afterExecute(r, t);
        // 依次调用处理器
        for (MonitorHandler handler : handlerMap.values())
```

```
            if (handler.usable())
                handler.after(r, t);
    }

    @Override
    protected void terminated() {
        super.terminated();
        for (MonitorHandler handler : handlerMap.values())
            if (handler.usable())
                handler.terminated(getLargestPoolSize(),
                        getCompletedTaskCount());
    }

    public MonitorHandler addMonitorTask(String key, MonitorHandler task,
            boolean overrideIfExist) {
        if (overrideIfExist)
            return handlerMap.put(key, task);
        else
            return handlerMap.putIfAbsent(key, task);
    }

    public MonitorHandler addMonitorTask(String key, MonitorHandler task) {
        return addMonitorTask(key, task, true);
    }

    public MonitorHandler removeMonitorTask(String key) {
        return handlerMap.remove(key);
    }

}
```

MonitorHandler类代码如下:

```
package com.sitech.crm_cmi.util.concurrent;

/**
 * 监控处理器, 目的是把before和after抽象出来, 以便在{@link MonitorableThreadPoolExecutor}
 * 中形成一条监控处理器链,观察者模式
 */
public interface MonitorHandler {

    /**
     * 改监控任务是否可用
     *
     * @return
     */
    boolean usable();

    /**
     * 任务执行前回调
     *
     * @param thread
     *            即将执行该任务的线程
     * @param runnable
     *            即将执行的任务
     */
    void before(Thread thread, Runnable runnable);

    /**
     * <pre>
     * 任务执行后回调 注意:
     *          1.当你往线程池提交的是{@link Runnable} 对象时, 参数runnable就是一个
     * {@link Runnable}对象
     *         2.当你往线程池提交的是{@link java.util.concurrent.Callable<?>}
     * 对象时, 参数runnable实际上就是一个{@link java.util.concurrent.FutureTask<?>}对象
     * 这时你可以通过把参数runnable downcast为FutureTask<?>或者Future来获取任务执行结果
     * </pre>
     *
     * @param runnable
     *            执行完后的任务
     * @param throwable
     *            异常信息
```

```
     */
    void after(Runnable runnable, Throwable throwable);

    /**
     * 线程池关闭后回调
     *
     * @param largestPoolSize
     * @param completedTaskCount
     */
    void terminated(int largestPoolSize, long completedTaskCount);

}
```

TimeMonitorHandler类代码如下:

```
package com.sitech.crm_cmi.util.concurrent;

import java.util.Map;
import java.util.concurrent.CancellationException;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.concurrent.FutureTask;

public class TimeMonitorHandler implements MonitorHandler {
    // 任务开始时间记录map, 多线程增删, 需用ConcurrentHashMap
    private final Map<Runnable, Long> timeRecords = new ConcurrentHashMap<Runnable, Long>();
    private final boolean usable;

    public TimeMonitorHandler() {
        this(true);
    }

    public TimeMonitorHandler(boolean usable) {
        this.usable = usable;
    }

    @Override
    public boolean usable() {
        return usable;
    }

    @Override
    public void terminated(int largestPoolSize, long completedTaskCount) {
        System.out.println(
                String.format("%s:largestPoolSize=%d, completedTaskCount=%s",
                        time(), largestPoolSize, completedTaskCount));
    }

    @Override
    public void before(Thread thread, Runnable runnable) {
        System.out.println(String.format("%s: before[%s -> %s]", time(),
                thread.getName(), runnable));
        timeRecords.put(runnable, System.currentTimeMillis());
    }

    @Override
    public void after(Runnable runnable, Throwable throwable) {
        long costTime = System.currentTimeMillis()
                - timeRecords.remove(runnable);

        Object result = null;
        if (throwable == null && runnable instanceof FutureTask<?>) {
            // 有返回值的异步任务，不一定是Callable<?>，也有可能是Runnable
            try {
                result = ((Future<?>) runnable).get();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // reset
            } catch (ExecutionException e) {
                throwable = e;
            } catch (CancellationException e) {
                throwable = e;
            }
```

```
        }
        if (throwable == null) {
            // 任务正常结束
            if (result != null)
                // 有返回值的异步任务
                System.out.println(String.format(
                        "%s: after[%s -> %s], costs %d millisecond, result: %s",
                        time(), Thread.currentThread().getName(), runnable,
                        costTime, result));
            else
                System.out.println(String.format(
                        "%s: after[%s -> %s], costs %d millisecond", time(),
                        Thread.currentThread().getName(), runnable, costTime));
        } else
            System.err.println(String.format(
                    "%s: after[%s -> %s], costs %d millisecond, exception: %s",
                    time(), Thread.currentThread().getName(), runnable,
                    costTime, throwable.getCause()));
    }

    private static String time() {
        return Long.toString(System.currentTimeMillis());
    }

}
```

Tester类代码如下:

```
package com.sitech.crm_cmi.util.concurrent;

import java.io.IOException;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class Tester {
    private static volatile boolean stop = false;
    private static final Random random = new Random(47);

    public static void main(String[] args)
            throws InterruptedException, IOException {
        // fixed size 10
        final MonitorableThreadPoolExecutor exec = new MonitorableThreadPoolExecutor(
                10, 10, 30, TimeUnit.SECONDS,
                new LinkedBlockingQueue<Runnable>());

        exec.addMonitorTask("TimeMonitorTask", newTimeMonitorHandler());
        // 起一个线程不断地往线程池丢任务
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                startAddTask(exec);
            }
        });
        t.start();

        // 丢任务丢50ms
        TimeUnit.MILLISECONDS.sleep(50);
        stop = true;
        t.join();
        exec.shutdown();
        // 等线程池任务跑完
        exec.awaitTermination(100, TimeUnit.SECONDS);
    }

    // 随机runnable或者callable<?>, 任务随机抛异常
    private static void startAddTask(MonitorableThreadPoolExecutor pool) {
        int count = 0;
        while (!stop) {
            if (random.nextBoolean())
                // 丢Callable<?>任务
                pool.submit(new Callable<Boolean>() {
```

```
                    @Override
                    public Boolean call() throws Exception {
                        // 随机抛异常
                        boolean bool = random.nextBoolean();
                        // 随机耗时 0~100 ms
                        TimeUnit.MILLISECONDS.sleep(random.nextInt(100));
                        if (bool)
                            throw new RuntimeException("thrown randomly");
                        return bool;
                    }
                });
            else
                // 丢Runnable
                pool.submit(new Runnable() {
                    @Override
                    public void run() {
                        // 随机耗时 0~100 ms
                        try {
                            TimeUnit.MILLISECONDS.sleep(random.nextInt(100));
                        } catch (InterruptedException e) {
                            Thread.currentThread().interrupt();
                        }
                        // 随机抛异常
                        if (random.nextBoolean())
                            throw new RuntimeException("thrown randomly");
                    };
                });
            System.out.println(
                    String.format("%s:submitted %d task", time(), ++count));
        }
    }

    private static MonitorHandler newTimeMonitorHandler() {
        return new TimeMonitorHandler();
    }

    private static String time() {
        return String.valueOf(System.currentTimeMillis());
    }
}
```