

GarfieldEr007

[HOME](#)[CONTACT](#)[GALLERY](#)[SUBSCRIBE](#)

彻底理解Java的feature模式

2018-12-23 22:47 GarfieldEr007 阅读(3095) 评论(0) 编辑 收藏

先上一个场景：假如你突然想做饭，但是没有厨具，也没有食材。网上购买厨具比较方便，食材去超市买更放心。

实现分析：在快递员送厨具的期间，我们肯定不会闲着，可以去超市买食材。所以，在主线程里面另起一个子线程去网购厨具。

但是，子线程执行的结果是要返回厨具的，而run方法是没有返回值的。所以，这才是难点，需要好好考虑一下。

模拟代码1：



```
package test;

public class CommonCook {

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        // 第一步 网购厨具
        OnlineShopping thread = new OnlineShopping();
        thread.start();
        thread.join(); // 保证厨具送到
        // 第二步 去超市购买食材
        Thread.sleep(2000); // 模拟购买食材时间
        Shicai shicai = new Shicai();
        System.out.println("第二步：食材到位");
```

```
// 第三步 用厨具烹饪食材
System.out.println("第三步：开始展现厨艺");
cook(thread.chuju, shicai);

System.out.println("总共用时" + (System.currentTimeMillis() - startTime) + "ms");
}
```

// 网购厨具线程

```
static class OnlineShopping extends Thread {

    private Chuju chuju;

    @Override
    public void run() {
        System.out.println("第一步：下单");
        System.out.println("第一步：等待送货");
        try {
            Thread.sleep(5000); // 模拟送货时间
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("第一步：快递送到");
        chuju = new Chuju();
    }

}
```

// 用厨具烹饪食材

```
static void cook(Chuju chuju, Shicai shicai) {}
```

// 厨具类

```
static class Chuju {}
```

// 食材类

```
static class Shicai {}  
}
```



运行结果：



第一步：下单

第一步：等待送货

第一步：快递送到

第二步：食材到位

第三步：开始展现厨艺

总共用时7013ms



可以看到，多线程已经失去了意义。在厨具送到期间，我们不能干任何事。对应代码，就是调用join方法阻塞主线程。

有人问了，不阻塞主线程行不行？？？

不行！！！！

从代码来看的话，run方法不执行完，属性chuju就没有被赋值，还是null。换句话说，没有厨具，怎么做饭。

Java现在的多线程机制，核心方法run是没有返回值的；如果要保存run方法里面的计算结果，必须等待run方法计算完，无论计算过程多么耗时。

面对这种尴尬的处境，程序员就会想：在子线程run方法计算的期间，能不能在主线程里面继续异步执行？？？

Where there is a will, there is a way！！！！

这种想法的核心就是Future模式，下面先应用一下Java自己实现的Future模式。

模拟代码2：



```
package test;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class FutureCook {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        long startTime = System.currentTimeMillis();
        // 第一步 网购厨具
        Callable<Chuju> onlineShopping = new Callable<Chuju>() {

            @Override
            public Chuju call() throws Exception {
                System.out.println("第一步: 下单");
                System.out.println("第一步: 等待送货");
                Thread.sleep(5000); // 模拟送货时间
                System.out.println("第一步: 快递送到");
                return new Chuju();
            }

        };

        FutureTask<Chuju> task = new FutureTask<Chuju>(onlineShopping);
        new Thread(task).start();
        // 第二步 去超市购买食材
        Thread.sleep(2000); // 模拟购买食材时间
        Shicai shicai = new Shicai();
        System.out.println("第二步: 食材到位");
        // 第三步 用厨具烹饪食材
```

```
        if (!task.isDone()) {    // 联系快递员，询问是否到货
            System.out.println("第三步：厨具还没到，心情好就等着（心情不好就调用cancel方法取消订单）");
        }
        Chuju chuju = task.get();
        System.out.println("第三步：厨具到位，开始展现厨艺");
        cook(chuju, shicai);

        System.out.println("总共用时" + (System.currentTimeMillis() - startTime) + "ms");
    }

    // 用厨具烹饪食材
    static void cook(Chuju chuju, Shicai shicai) {}

    // 厨具类
    static class Chuju {}

    // 食材类
    static class Shicai {}
}
```



运行结果：



第一步：下单

第一步：等待送货

第二步：食材到位

第三步：厨具还没到，心情好就等着（心情不好就调用cancel方法取消订单）

第一步：快递送到

第三步：厨具到位，开始展现厨艺

总共用时5005ms



可以看见，在快递员送厨具的期间，我们没有闲着，可以去买食材；而且我们知道厨具到没到，甚至可以在厨具没到的时候，取消订单不要了。

好神奇，有没有。

下面具体分析一下第二段代码：

1) 把耗时的网购厨具逻辑，封装到了一个Callable的call方法里面。



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



Callable接口可以看作是Runnable接口的补充，call方法带有返回值，并且可以抛出异常。

2) 把Callable实例当作参数，生成一个FutureTask的对象，然后把这个对象当作一个Runnable，作为参数另起线程。

```
public class FutureTask<V> implements RunnableFuture<V>
```

```
public interface RunnableFuture<V> extends Runnable, Future<V>
```



```
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;  
}
```



这个继承体系中的核心接口是Future。Future的核心思想是：一个方法f，计算过程可能非常耗时，等待f返回，显然不明智。可以在调用f的时候，立马返回一个Future，可以通过Future这个数据结构去*控制*方法f的计算过程。

这里的*控制*包括：

get方法：获取计算结果（如果还没计算完，也是必须等待的）

cancel方法：还没计算完，可以取消计算过程

isDone方法：判断是否计算完

isCancelled方法：判断计算是否被取消

这些接口的设计很完美，FutureTask的实现注定不会简单，后面再说。

3) 在第三步里面，调用了isDone方法查看状态，然后直接调用task.get方法获取厨具，不过这时还没送到，所以还是会等待3秒。对比第一段代码的执行结果，这里我们节省了2秒。这是因为在快递员送货期间，我们去超市购买食材，这两件事在同一时间段内异步执行。

通过以上3步，我们就完成了对Java原生Future模式最基本的应用。下面具体分析下FutureTask的实现，先看JDK8的，再比较一下JDK6的实现。

既然FutureTask也是一个Runnable，那就看看它的run方法



```
public void run() {
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                       null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable; // 这里的callable是从构造方法里面传人的
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex); // 保存call方法抛出的异常
            }
        }
    }
}
```



```

        }
        if (ran)
            set(result); // 保存call方法的执行结果
    }
} 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);
}
}

```



先看try语句块里面的逻辑，发现run方法的主要逻辑就是运行Callable的call方法，然后将保存结果或者异常（用的一个属性result）。这里比较难想到的是，将call方法抛出的异常也保存起来了。

这里表示状态的属性state是个什么鬼



```

* Possible state transitions:
* NEW -> COMPLETING -> NORMAL
* NEW -> COMPLETING -> EXCEPTIONAL
* NEW -> CANCELLED
* NEW -> INTERRUPTING -> INTERRUPTED
*/
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;
```



把FutureTask看作一个Future，那么它的作用就是控制Callable的call方法的执行过程，在执行的过程中自然会有状态的转换：

- 1) 一个FutureTask新建出来，state就是NEW状态；COMPLETING和INTERRUPTING用的进行时，表示瞬时状态，存在时间极短(为什么要设立这种状态??? 不解)；NORMAL代表顺利完成；EXCEPTIONAL代表执行过程出现异常；CANCELED代表执行过程被取消；INTERRUPTED被中断
- 2) 执行过程顺利完成：NEW -> COMPLETING -> NORMAL
- 3) 执行过程出现异常：NEW -> COMPLETING -> EXCEPTIONAL
- 4) 执行过程被取消：NEW -> CANCELLED
- 5) 执行过程中，线程中断：NEW -> INTERRUPTING -> INTERRUPTED

代码中状态判断、CAS操作等细节，请读者自己阅读。

再看看get方法的实现：



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



```
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, waitersOffset,
                                                  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);
}
}

```



get方法的逻辑很简单，如果call方法的执行过程已完成，就把结果给出去；如果未完成，就将当前线程挂起等待。awaitDone方法里面死循环的逻辑，推演几遍就能看懂；它里面挂起线程的主要创新是定义了WaitNode类，来将多个等待线程组织成队列，这是与JDK6的实现最大的不同。

挂起的线程何时被唤醒：



```

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); // 唤醒线程
                }
                WaitNode next = q.next;
                if (next == null)
                    break;
                q.next = null; // unlink to help gc
            }
        }
    }
}

```

```

        q = next;
    }
    break;
}

done();

callable = null;        // to reduce footprint
}

```



以上就是JDK8的大体实现逻辑，像cancel、set等方法，也请读者自己阅读。

再看看JDK6的实现。

JDK6的FutureTask的基本操作都是通过自己的内部类Sync来实现的，而Sync继承自AbstractQueuedSynchronizer这个出镜率极高的并发工具类



```

/** State value representing that task is running */
private static final int RUNNING    = 1;
/** State value representing that task ran */
private static final int RAN        = 2;
/** State value representing that task was cancelled */
private static final int CANCELLED = 4;

/** The underlying callable */
private final Callable<V> callable;
/** The result to return from get() */
private V result;

```

```
/** The exception to throw from get() */  
private Throwable exception;
```



里面的状态只有基本的几个，而且计算结果和异常是分开保存的。



```
V innerGet() throws InterruptedException, ExecutionException {  
    acquireSharedInterruptibly(0);  
    if (getState() == CANCELLED)  
        throw new CancellationException();  
    if (exception != null)  
        throw new ExecutionException(exception);  
    return result;  
}
```



这个get方法里面处理等待线程队列的方式是调用了acquireSharedInterruptibly方法，看过我之前几篇博客文章的读者应该非常熟悉了。其中的等待线程队列、线程挂起和唤醒等逻辑，这里不再赘述，如果不明白，请出门左转。

最后来看看，Future模式衍生出来的更高级的应用。

再上一个场景：我们自己写一个简单的数据库连接池，能够复用数据库连接，并且能在高并发情况下正常工作。

实现代码1:



```
package test;

import java.util.concurrent.ConcurrentHashMap;

public class ConnectionPool {

    private ConcurrentHashMap<String, Connection> pool = new ConcurrentHashMap<String, Connection>();

    public Connection getConnection(String key) {
        Connection conn = null;
        if (pool.containsKey(key)) {
            conn = pool.get(key);
        } else {
            conn = createConnection();
            pool.putIfAbsent(key, conn);
        }
        return conn;
    }

    public Connection createConnection() {
        return new Connection();
    }

    class Connection {}
}
```



我们用了ConcurrentHashMap，这样就不必把getConnection方法置为synchronized(当然也可以用Lock)，当多个线程同时调用getConnection方法时，性能大幅提升。

貌似很完美了，但是有可能导致多余连接的创建，推演一遍：

某一时刻，同时有3个线程进入getConnection方法，调用pool.containsKey(key)都返回false，然后3个线程各自都创建了连接。虽然ConcurrentHashMap的put方法只会加入其中一个，但还是生成了2个多余的连接。如果是真正的数据库连接，那会造成极大的资源浪费。

所以，我们现在的难点是：如何在多线程访问getConnection方法时，只执行一次createConnection。

结合之前Future模式的实现分析：当3个线程都要创建连接的时候，如果只有一个线程执行createConnection方法创建一个连接，其它2个线程只需要用这个连接就行了。再延伸，把createConnection方法放到一个Callable的call方法里面，然后生成FutureTask。我们只需要让一个线程执行FutureTask的run方法，其它的线程只执行get方法就好了。

上代码：



```
package test;

import java.util.concurrent.Callable;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

public class ConnectionPool {

    private ConcurrentHashMap<String, FutureTask<Connection>> pool = new ConcurrentHashMap<String, FutureTask<Connection>>();

    public Connection getConnection(String key) throws InterruptedException, ExecutionException {
        FutureTask<Connection> connectionTask = pool.get(key);
        if (connectionTask != null) {
            return connectionTask.get();
        }
    }
}
```



```

    } else {
        Callable<Connection> callable = new Callable<Connection>() {
            @Override
            public Connection call() throws Exception {
                return createConnection();
            }
        };
        FutureTask<Connection> newTask = new FutureTask<Connection>(callable);
        connectionTask = pool.putIfAbsent(key, newTask);
        if (connectionTask == null) {
            connectionTask = newTask;
            connectionTask.run();
        }
        return connectionTask.get();
    }
}

public Connection createConnection() {
    return new Connection();
}

class Connection {
}
}

```



推演一遍：当3个线程同时进入else语句块时，各自都创建了一个FutureTask，但是ConcurrentHashMap只会加入其中一个。第一个线程执行pool.putIfAbsent方法后返回null，然后connectionTask被赋值，接着就执行run方法去创建连接，最后get。后面的线程执行pool.putIfAbsent方法不会返回null，就只会执行get方法。

在并发的环境下，通过FutureTask作为中间转换，成功实现了让某个方法只被一个线程执行。

from: <https://www.cnblogs.com/cz123/p/7693064.html>