目录

第1章 基础

01 开篇词：为什么学习本专栏

02 String、Long 源码解析和面试题

03 Java 常用关键字理解

04 Arrays、Collections、Objects 常用方法源码解析

第2章 集合

05 ArrayList 源码解析和设计思路

06 LinkedList 源码解析

07 List 源码会问哪些面试题

08 HashMap 源码解析

09 TreeMap 和 LinkedHashMap 核心源码解析

10 Map源码会问哪些面试题

11 HashSet、TreeSet 源码解析

12 彰显细节：看集合源码对我们实际工作的帮助和应用

13 差异对比：集合在 Java 7 和 8 有何不同和改进

14 简化工作：Guava Lists Maps 实际工作运用和源码

第3章 并发集合类

15 CopyOnWriteArrayList 源码解析和设计思路

16 ConcurrentHashMap 源码解析和设计思路

17 并发 List、Map源码面试题

18 场景集合：并发 List、Map的应用

28 Future、ExecutorService 源码解析

更新时间：2019-11-05 10:29:43

今天应做的事没有做，明天再早也是耽误了。

——裴斯泰洛齐

引导语

本章和大家一起看下有返回值的线程如何创建，两种线程 API 之间如何关联，介绍一下和线程相关的其余 API。

1 整体架构

画了一个关于线程 API 之间关系的依赖图，如下：

目录

在上一章节，我们说了 Thread 和 Runnable，本小节我们按照这个图把剩下的几个 API 也说完，然后把 API 之间的关系理清楚。

为了方便大家更好的理解，我们首先看一个 demo，这个场景说的是我们往线程池里面提交一个有返回值的线程，代码如下：

```
// 首先我们创建了一个线程池
ThreadPoolExecutor executor = new ThreadPoolExecutor(3, 3, 0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<>());
// futureTask 我们叫做线程任务，构造器的入参是 Callable
FutureTask futureTask = new FutureTask(new Callable<String> () {
  @Override
  public String call() throws Exception {
    Thread.sleep(3000);
    // 返回一句话
    return "我是子线程"+Thread.currentThread().getName();
  }
});
```

目录

```
String result = (String) futureTask.get();
log.info("result is {}",result);
```

从上面这个 demo 中，我们大概可以看出各个 API 的作用：

1. Callable 定义我们需要做的事情，是可以有返回值的；
2. FutureTask 我们叫做任务，入参是 Callable，是对 Callable 的包装，方便线程池的使用；
3. 最后通过 FutureTask.get() 得到子线程的计算结果。

接着我们分别来看看各种 API 的底层实现。

2 Callable

Callable 是一个接口，约定了线程要做的事情，和 Runnable 一样，不过这个线程任务是有返回值的，我们来看下接口定义：

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

返回值是一个泛型，可以定义成任何类型，但我们使用的时候，都不会直接使用 Callable，而是会结合 FutureTask 一起使用。

3 FutureTask

FutureTask 我们可以当做是线程运行的具体任务，从上图中，我们可以看到 FutureTask 实现了 RunnableFuture 接口，源码如下：

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

而 RunnableFuture 又实现了 Runnable, Future 两个接口，接下来我们先看 Future，再看 RunnableFuture，最后看 FutureTask。

3.1 Future

我们刚才说 Callable 是可以返回子线程执行结果的，在获取结果的时候，就需要用到 Future 接口了。

Future 接口注释上写了这些：

1. 定义了异步计算的接口，提供了计算是否完成的 check、等待完成和取回等多种方法；
2. 如果想得到结果可以使用 get 方法，此方法(无参方法)会一直阻塞到异步任务计算完成；
3. 取消可以使用 cancel 方法，但一旦任务计算完成，就无法被取消了。

Future 接口定义了这些方法：

```
// 如果任务已经成功了，或已经取消了，是无法再取消的，会直接返回取消成功(true)
// 如果任务还没有开始进行时，发起取消，是可以取消成功的。
// 如果取消时，任务已经在运行了，mayInterruptIfRunning 为 true 的话，就可以打断运行中的线程
```

目录

```
// 返回线程是否已经被取消了，true 表示已经被取消了
// 如果线程已经运行结束了，isCancelled 和 isDone 返回的都是 true
boolean isCancelled();

// 线程是否已经运行结束了
boolean isDone();

// 等待结果返回
// 如果任务被取消了，抛 CancellationException 异常
// 如果等待过程中被打断了，抛 InterruptedException 异常
V get() throws InterruptedException, ExecutionException;

// 等待，但是带有超时时间的，如果超时时间外仍然没有响应，抛 TimeoutException 异常
V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
```

从接口上可以看出，Future 定义了各种方法对任务进行了管理，比如说取消任务，得到任务的计算结果等等。

3.2 RunnableFuture

RunnableFuture 也是一个接口，定义如下：

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


RunnableFuture 接口的最大目的，是让 Future 可以对 Runnable 进行管理，可以取消 Runnable，查看 Runnable 是否完成等等。

3.3 统一 Callable 和 Runnable

我们现在清楚了，新建任务有两种方式，一种是无返回值的 Runnable，一种是有返回值的 Callable，但对 Java 其他 API 来说使用起来并不是很方便，没有一个统一的接口，比如说线程池在提交任务时，是不是应该针对 Runnable 和 Callable 两种情况提供不同的实现思路呢？所以 FutureTask 出现了，FutureTask 实现了 RunnableFuture 接口，又集合了 Callable（Callable 是 FutureTask 的属性），还提供了两者一系列的转化方法，这样 FutureTask 就统一了 Callable 和 Runnable，我们一起来细看下。

3.3.1 FutureTask 的类定义

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

从类定义上可以看出来 FutureTask 实现了 RunnableFuture 接口，也就是说间接实现了 Runnnable 接口（RunnableFuture 实现了 Runnnable 接口），就是说 FutureTask 本身就是个 Runnnable，同时 FutureTask 也实现了 Future，也就是说 FutureTask 具备对任务进行管理的功能（Future 具备对任务进行管理的功能）。

3.3.2 FutureTask 的属性

我们一起来看下 FutureTask 有哪些重要属性：

目录

```
private static final int NEW          = 0;//线程任务创建
private static final int COMPLETING   = 1;//任务执行中
private static final int NORMAL       = 2;//任务执行结束
private static final int EXCEPTIONAL  = 3;//任务异常
private static final int CANCELLED    = 4;//任务取消成功
private static final int INTERRUPTING = 5;//任务正在被打断中
private static final int INTERRUPTED  = 6;//任务被打断成功

// 组合了 Callable
private Callable<V> callable;
// 异步线程返回的结果
private Object outcome;
// 当前任务所运行的线程
private volatile Thread runner;
// 记录调用 get 方法时被等待的线程
private volatile WaitNode waiters;
```

从属性上我们明显看到 Callable 是作为 FutureTask 的属性之一，这也就让 FutureTask 具备了转化 Callable 和 Runnable 的功能，接着我们看下 FutureTask 的构造器，看看两者是如何转化的。

3.3.3 FutureTask 的构造器

FutureTask 有两个构造器，分别接受 Callable 和 Runnable，如下：

```
// 使用 Callable 进行初始化
public FutureTask(Callable<V> callable){
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    // 任务状态初始化
    this.state = NEW;       // ensure visibility of callable
}

// 使用 Runnable 初始化，并传入 result 作为返回结果。
// Runnable 是没有返回值的，所以 result 一般没有用，置为 null 就好了
public FutureTask(Runnable runnable, V result) {
    // Executors.callable 方法把 runnable 适配成 RunnableAdapter，RunnableAdapter 实现了 ca
    this.callable = Executors.callable(runnable, result);
    this.state = NEW;       // ensure visibility of callable
}
```


Runnable 的两个构造器，只有一个目的，就是把入参都转化成 Callable，那么为什么不都转化成 Runnnable 呢？主要是因为 Callable 的功能比 Runnnable 丰富，Callable 有返回值，而 Runnnable 没有。

我们注意到入参是 Runnable 的构造器，会使用 Executors.callable 方法来把 Runnnable 转化成 Callable，Runnnable 和 Callable 两者都是接口，两者之间是无法进行转化的，所以 Java 新建了一个转化类：RunnableAdapter 来进行转化，我们来看下转化的逻辑：

```
// 转化 Runnable 成 Callable 的工具类
static final class RunnableAdapter<T> implements Callable<T> {
    final Runnable task;
    final T result;
    RunnableAdapter(Runnable task, T result) {
        this.task = task;
```

目录

```
        task.run();
        return result;
    }
}
```

我们可以看到：

1. 首先 RunnableAdapter 实现了 Callable，所以 RunnableAdapter 就是 Callable；
2. 其次 Runnable 是 RunnableAdapter 的一个属性，在构造 RunnableAdapter 的时候会传进来，并且在 call 方法里面调用 Runnable 的 run 方法。

这是一个典型的适配模型，我们要把 Runnable 适配成 Callable，首先要实现 Callable 的接口，接着在 Callable 的 call 方法里面调用被适配对象（Runnable）的方法。

FutureTask 构造器设计很巧妙，将 Runnable 和 Callable 灵活的打通，向内和向外只提供功能更加丰富的 Callable 接口，值得我们学习。

3.3.4 FutureTask 对 Future 接口方法的实现

我们主要看几个关键的方法实现源码。

3.3.4.1 get

get 有无限阻塞和带超时时间两种方法，我们通常建议使用带超时时间的方法，源码如下：

```
public V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException {
    if (unit == null)
        throw new NullPointerException();
    int s = state;
    // 如果任务已经在执行中了，并且等待一定的时间后，仍然在执行中，直接抛出异常
    if (s <= COMPLETING &&
        (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLETING)
        throw new TimeoutException();
    // 任务执行成功，返回执行的结果
    return report(s);
}
// 等待任务执行完成
private int awaitDone(boolean timed, long nanos)
    throws InterruptedException {
    // 计算等待终止时间，如果一直等待的话，终止时间为 0
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    // 不排队
    boolean queued = false;
    // 无限循环
    for (;;) {
        // 如果线程已经被打断了，删除，抛异常
        if (Thread.interrupted()) {
            removeWaiter(q);
            throw new InterruptedException();
        }
        // 当前任务状态
        int s = state;
        // 当前任务已经执行完了，返回
        if (s > COMPLETING) {
```

目录

```
            return s;
        }
        // 如果正在执行，当前线程让出 cpu，重新竞争，防止 cpu 飙高
        else if (s == COMPLETING) // cannot time out yet
            Thread.yield();
            // 如果第一次运行，新建 waitNode，当前线程就是 waitNode 的属性
        else if (q == null)
            q = new WaitNode();
            // 默认第一次都会执行这里，执行成功之后，queued 就为 true，就不会再执行了
            // 把当前 waitNode 当做 waiters 链表的第一个
        else if (!queued)
            queued = UNSAFE.compareAndSwapObject(this, waitersOffset,
                                                 q.next = waiters, q);
            // 如果设置了超时时间，并过了超时时间的话，从 waiters 链表中删除当前 wait
        else if (timed) {
            nanos = deadline - System.nanoTime();
            if (nanos <= 0L) {
                removeWaiter(q);
                return state;
            }
            // 没有过超时时间，线程进入 TIMED_WAITING 状态
            LockSupport.parkNanos(this, nanos);
        }
        // 没有设置超时时间，进入 WAITING 状态
        else
            LockSupport.park(this);
    }
}
```


get 方法虽然名字叫做 get，但却做了很多 wait 的事情，当发现任务还在进行中，没有完成时，就会阻塞当前进程，等待任务完成后再返回结果值。阻塞底层使用的是 LockSupport.park 方法，使当前线程进入 WAITING 或 TIMED_WAITING 状态。

3.3.4.2 run

```
/**
 * run 方法可以直接被调用
 * 也可以开启新的线程进行调用
 */
public void run() {
    // 状态不是任务创建，或者当前任务已经有线程在执行了，直接返回
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject(this, runnerOffset,
                                     null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        // Callable 不为空，并且已经初始化完成
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                // 调用执行
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex);
```

目录

```
                set(result);
            }
        } finally {
            runner = null;
            int s = state;
            if (s >= INTERRUPTING)
                handlePossibleCancellationInterrupt(s);
        }
    }
```

run 方法我们再说明几点：

1. run 方法是没有返回值的，通过给 outcome 属性赋值（set(result)），get 时就能从 outcome 属性中拿到返回值；
2. FutureTask 两种构造器，最终都转化成了 Callable，所以在 run 方法执行的时候，只需要执行 Callable 的 call 方法即可，在执行 c.call() 代码时，如果入参是 Runnable 的话， 调用路径为 c.call() -> RunnableAdapter.call() -> Runnable.run()，如果入参是 Callable 的话，直接调用。

3.3.4.3 cancel

```
// 取消任务，如果正在运行，尝试去打断
public boolean cancel(boolean mayInterruptIfRunning) {
    if (!(state == NEW &&//任务状态不是创建 并且不能把 new 状态置为取消，直接返回 false
          UNSAFE.compareAndSwapInt(this, stateOffset, NEW,
              mayInterruptIfRunning ? INTERRUPTING : CANCELLED)))
        return false;
    // 进行取消操作，打断可能会抛出异常，选择 try finally 的结构
    try {    // in case call to interrupt throws exception
        if (mayInterruptIfRunning) {
            try {
                Thread t = runner;
                if (t != null)
                    t.interrupt();
            } finally { // final state
                //状态设置成已打断
                UNSAFE.putOrderedInt(this, stateOffset, INTERRUPTED);
            }
        }
    } finally {
        // 清理线程
        finishCompletion();
    }
    return true;
}
```


4 总结

大家现在可以回头看看一开始我们贴出来的图，看看自己照着图能否想起来各个 API 的作用，比如 Callable 是干啥的，FutureTask 又有什么作用，Runnable 和 Calllable 之间又是如何关联起来，几个 API 之间的关系的确很复杂，FutureTask 是关键，通过 FutureTask 把 Runnnable、Callable、Future 都串起来了，使 FutureTask 具有三者的功能，统一了 Runnnable 和 Callable，更方便使用。

目录

精选留言 2

欢迎在这里发表留言，作者筛选后可公开显示

大LOVE辉

1. // 如果任务已经成功了，或已经取消了，是无法再取消的，会直接返回取消成功(true) 这句话没理解 2. 下面get()获取内容抛出Exceptionxception异常吗

0 回复 2019-12-05

文贺 回复 **大LOVE辉**

同学你好 1 可以看下 cancel 的源码，当任务结束时，或已经被取消时，源码直接返回了 true。2 可以看下 get 的源码，get 方法明确抛出了 ExecutionException 异常了哈。

回复 7天前

敲木鱼的小和尚

跟随者老师的步伐，每天过着源码，累并快乐着，学到好多，给力给力

1 回复 2019-11-27

文贺 回复 **敲木鱼的小和尚**

maybe this is life.

回复 2019-11-30 13:14:07

千学不如一看，千看不如一练