

(/apps/redirect?  
utm\_source=side-  
banner-click)

# FutureTask 源码分析 (基于Java 8)



爱吃鱼的KK (/u/f9577ef03ae0) [+ 关注](#)

2016.12.15 21:35\* 字数 800 阅读 704 评论 1 喜欢 6 赞赏 1

(/u/f9577ef03ae0)

## FutureTask 是java实现异步编程的基础

### 1. Future API

```
1) V get() throws InterruptedException, ExecutionException;  
    获取计算的结果, 若计算没完成, 直接 await, 直到 计算结束或线程中断  
2) V get (long timeout, TimeUnit unit) throws InterruptedException, ExecutionException  
    获取计算的结果, 若计算没完成, 直接 await, 直到 计算结束或线程中断或time时间超时  
3) boolean isDone();  
    返回计算是否完成, 若任务完成则返回true ( 任务完成 state = normal, exception, interrup  
4) int awaitDone(boolean timed, long nanos) throws InterruptedException  
    等待任务完成, 或时间超时, 返回值是 future 的state的状态(**await是实现future的重要方法**)
```

### 2. 以下以一个FutureTask实现cache的例子来进行介绍



```

// 常用connection接口
public interface Connection {
    String getName();
}

// abstract cache 类
public abstract class AbstractLocalCache<K, V> {

    protected Logger logger = Logger.getLogger(getClass());

    /** 本地缓存存储地址 */
    private ConcurrentHashMap<K, Future<V>> pool = new ConcurrentHashMap<>();

    private ExecutorService executorService = Executors.newFixedThreadPool(Runtime.g

// 模版方法
public abstract V computeV(K k);

public Future<V> getResult(K k){
    Future<V> result = null;
    if(pool.containsKey(k)){
        return pool.get(k);
    }
    FutureTask<V> future = new FutureTask<V>(new Callable<V>() {
        @Override
        public V call() throws Exception {
            return computeV(k);
        }
    });

    // 说明map中以前没有对应的 futureTask
    // 仔细体会 putIfAbsent 的作用
    if(pool.putIfAbsent(k, future) == null){
        executorService.submit(future);
    }
    return future;
}

// connection cache
public class LocalCacheConnection extends AbstractLocalCache<String , Connection> {

```

(/apps/redirect?  
utm\_source=side-  
banner-click)



```
@Override
public Connection computeV(String s) {

    logger.info("创建connection开始");
    logger.info("睡觉开始");

    try {
        Thread.sleep(3*1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    logger.info("睡觉结束");
    logger.info("创建connection结束");

    return new Connection() {
        @Override
        public String getName() {
            return s;
        }

        @Override
        public String toString() {
            return "A connection(" + s + ")";
        }
    };
}

// future main 方法
public class FutureMain {

    private static final Logger logger = Logger.getLogger(FutureMain.class);

    public static void main(String[] args) throws Exception{

        LocalCacheConnection localCacheConnection = new LocalCacheConnection();
        Future<?> future = localCacheConnection.getResult("connection");

        new Thread(){
            @Override
            public void run() {
```

(/apps/redirect?  
utm\_source=side-  
banner-click)



```
        try {
            logger.info("future.get() : " + future.get(2 , TimeUnit.SECONDS))
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            logger.info("future.get() over");
        }
    }
}.start();

new Thread(){
    @Override
    public void run() {
        try {
            logger.info("future.get() : " + future.get(4, TimeUnit.SECONDS))
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            logger.info("future.get() over");
        }
    }
}.start();
    }
}
```

(/apps/redirect?  
utm\_source=side-  
banner-click)

计算结果：



```
[2016-12-17 21:25:22,945] INFO pool-1-thread-1 (LocalCacheConnection.java:11) - 创建
[2016-12-17 21:25:22,949] INFO pool-1-thread-1 (LocalCacheConnection.java:12) - 睡觉
[2016-12-17 21:25:24,948] INFO Thread-0 (FutureMain.java:31) - future.get() over
java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
    at com.lami.tuomatuo.search.base.concurrent.future.example.FutureMain$1.run(Futu
[2016-12-17 21:25:25,955] INFO pool-1-thread-1 (LocalCacheConnection.java:20) - 睡觉
[2016-12-17 21:25:25,955] INFO pool-1-thread-1 (LocalCacheConnection.java:21) - 创建
[2016-12-17 21:25:25,956] INFO Thread-1 (FutureMain.java:40) - future.get() : A con
[2016-12-17 21:25:25,957] INFO Thread-1 (FutureMain.java:44) - future.get() over
```

(/apps/redirect?  
utm\_source=side-  
banner-click)

### 3.FutureTask 的运行方式是这样的

1. 将一个 Callable 置为 FutureTask 的内置成员
2. 执行 Callable 中的 call 方法
3. 调用futureTask.get(timeout, TimeUnit) 方法, 获取call的执行结果, 超时的话就报  
TimeoutException

从上面可以看出: 只要将耗时的任务丢给FutureTask, 不必等待程序运行结束,继续往下执行, 从而实现程序异步执行的功能

看到上面的例子, 你可能会有疑问: 两个都是调用future.get(timeout, TimeUnit) 方法, 一个报异常, 一个确得到了结果, futureTask 的内部执行机制到底是什么??

(PS: 对了, 有这样的疑惑才能往代码的深处走)

我们都知道ExecutorService是个线程的工具类, 将FutureTask丢给它后会执行对应的run方法, 那我们就先看 FutureTask的run方法

#### FutureTask.run 方法



```
public void run() {
    // 判断 state 是否是new, 防止并发重复执行
    if(state != NEW ||
        !unsafe.compareAndSwapObject(this, runnerOffset, null, Thread.current))
        return;
    }

    try {
        Callable<V> c = callable;
        if(c != null && state == NEW){
            V result ;
            boolean ran;
            try{ // 调用call方法执行计算
                result = c.call();
                ran = true;
            }catch (Throwable ex){
                result = null;
                ran = false;
                // 执行中抛异常, 更新state状态, 释放等待的线程(调用finishCompletion)
                setException(ex);
            }
            if(ran){ // 执行成功, 进行赋值操作
                set(result);
            }
        }
    }finally {
        // runner must be non-null until state is settled to prevent concurrent
        runner = null;
        // state must be re-read after nulling runner to prevent leaked interrup
        int s = state;
        if(s >= INTERRUPTING){
            handlePossibleCancellationInterrupt(s);
        }
    }
}
```

(/apps/redirect?  
utm\_source=side-  
banner-click)

这里看到state这个变量, 它是futureTask执行任务的状态(一个有7种)



```

/**
 * 这几种状态比较重要, 下面是 FutureTask 中 state 的状态转变的几种情况
 * Possible state's transitions
 * NEW -> COMPLETING -> NORMAL
 * NEW -> COMPLETING -> EXCEPTIONAL
 * NEW -> CANCELLED
 * NEW -> INTERRUPTING -> INETRRUPTED
 */

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;

```

(/apps/redirect?  
utm\_source=side-  
banner-click)

而run其实没做什么, 就是执行 callable.call方法, 成功的话将执行结果调用set进行赋值, 并更新state的值(通过cas)

**下面看例子中的 future.get(timeout, TimeUnit) 的源码**

```

public V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionExce
    // get(timeout, unit) 也很简单, 主要还是在 awaitDone里面
    if(unit == null){
        throw new NullPointerException();
    }
    int s = state;
    // 判断state状态是否 <= Completing, 调用awaitDone进行旋转
    if(s <= COMPLETING && (s = awaitDone(true, unit.toNanos(timeout))) <= COMPLE
        throw new TimeoutException();
    }
    // 根据state的值进行返回结果或抛出异常
    return report(s);
}

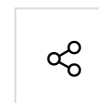
```



get() 方法中涉及到 awaitDone 方法, 将awaitDone的运行结果赋值给state, 最后report方法根据state值进行返回相应的值, 而awaitDone是整个 FutureTask 运行的核心

**那下面来看 awaitDone的方法**

(/apps/redirect?  
utm\_source=side-  
banner-click)





```

/**
 * Awaits completion or aborts on interrupt or timeout
 * 调用 awaitDone 进行线程的自旋
 * 自旋一般调用步骤
 * 1) 若支持线程中断, 判断当前的线程是否中断
 *     a. 中断, 退出自旋, 在线程队列中移除对应的节点
 *     b. 进行下面的步骤
 * 2) 将当前的线程构造成一个 WaiterNode 节点, 加入到当前对象的队列里面 (进行 cas 操作)
 * 3) 判断当前的调用是否设置阻塞超时时间
 *     a. 有 超时时间, 调用 LockSupport.parkNanos; 阻塞结束后, 再次进行 自旋 , 还是到
 *     b. 没 超时时间, 调用 LockSupport.park
 *
 * @param timed true if use timed waits
 * @param nanos time to waits, if timed
 * @return state upon completion
 */
private int awaitDone(boolean timed, long nanos) throws InterruptedException{
    // default timed = false, nanos = 0, so deadline = 0
    final long deadline = timed ? System.nanoTime() + nanos : 0L;
    WaitNode q = null;
    boolean queued = false;
    for(;;){
        // Thread.interrupted 判断当前的线程是否中断(调用两次会清楚对应的状态位)
        // Thread.interrupt 将当前的线程设置成中断状态
        if(Thread.interrupted()){
            removeWaiter(q, Thread.currentThread().getId());
            throw new InterruptedException();
        }

        int s = state;
        /** 1. s = NORMAL, 说明程序执行成功, 直接获取对应的 V
         */
        if(s > COMPLETING){
            if(q != null){
                q.thread = null;
            }
            return s;
        }
        // s = COMPLETING ; 看了全部的代码说明整个任务在处理的中间状态, s紧接着会进行改变
        // s 变成 NORMAL 或 EXCEPTION
        // 所以调用 yield 让线程状态变更, 重新进行CPU时间片竞争, 并且进行下次循环
        else if(s == COMPLETING){ // cannot time out yet

```

(/apps/redirect?  
utm\_source=side-  
banner-click)



```
        Thread.yield();
    }
    // 当程序调用 get 方法时, 一定会调用一次下面的方法, 对 q 进行赋值
    else if(q == null){
        q = new WaitNode();
    }
    // 判断有没有将当前的线程构造造成一个节点, 赋值到对象对应的属性里面
    // 第一次 waiters 一定是 null 的, 进行赋值的是一个以 q 为首节点的栈(JUC里面还有一
    else if(!queued){
        queued = unsafe.compareAndSwapObject(this, waitersOffset, q.next = w
    }
    // 调用默认的 get()时, timed = false, 所以不执行这一步
    else if(timed){
        // 进行阻塞时间的判断, 第二次循环时, nanos = 0L, 直接 removeWaiter 返回现在
        nanos = deadline - System.nanoTime();
        if(nanos <= 0L){
            removeWaiter(q, Thread.currentThread().getId());
            return state;
        }
        LockSupport.parkNanos(this, nanos);
    }
    // 进行线程的阻塞
    else{
        LockSupport.park(this);
    }
}
}
```

(/apps/redirect?  
utm\_source=side-  
banner-click)

结合我们刚才例子(FutureMain)中的两个调用futureTask.get()方法

第一个futureTask.get(2. TimeUnit.SECOND), 因为执行的任务需要花费3秒, 所以它先会 LockSupport.parkNanos(210001000\*1000) 阻塞2秒, 之后再次进行同样的地方, 但nanos 已是0, 所以调用removeWaiter方法, 最后抛出异常



第二个futureTask.get(4. TimeUnit.SECOND), 因为执行的任务需要花费3秒, 所以它先会LockSupport.parkNanos(410001000\*1000) 阻塞4秒, 但是任务只花费3秒, 所以执行完成后会调用set方法进行赋值, 在set方法中有一个finishCompletion方法, 这个方法会唤醒所有阻塞的节点, 所以第二个futureTask.get只花费3秒就得到了结果

**分析一下 removeWaiter 方法(这是实现并发链表中移除队列节点的一个操作)**

(/apps/redirect?  
utm\_source=side-  
banner-click)



```

/**
 * 这个 removeWaiter 个人认为是最搞人的, 尤其在多线程环境中, 同时进行节点的删除
 * 时隔一个月回头再看, 下面的代码就是一个并发安全的栈中进行一个节点的删除操作
 * Tries to unlink a time-out
 * @param node
 */
private void removeWaiter(WaitNode node, long i){
    logger.info("removeWaiter node" + node + ", i: " + i + " begin");
    if(node != null){
        node.thread = null; // 将移除的节点的thread=null, 为移除做标示

        retry:
        for(;;){ // restart on removeWaiter race
            for(WaitNode pred = null, q = waiters, s; q != null; q = s){
                logger.info("q : " + q + ", i:" + i);
                s = q.next;
                // 通过 thread 判断当前 q 是否是需要移除的 q 节点
                if(q.thread != null){
                    pred = q;
                    logger.info("q : " + q + ", i:" + i);
                }
                // 何时执行到这个if条件 ?
                // hehe 只有第一步不满足时, 也就是q.thread=null (p就是应该移除的节点)
                else if(pred != null){
                    logger.info("q : " + q + ", i:" + i);
                    pred.next = s; // 将前一个节点的 next 指向当前节点的 next 节点
                    // pred.thread == null 这种情况是在多线程进行并发 removeWaiter 时
                    // 而此时正好移除节点 node 和 pred, 所以loop跳到retry, 在进行一次
                    if(pred.thread == null){ // check for race
                        continue retry;
                    }
                }
            }
            // 这一步何时操作呢?
            // 想想 若p是头节点
            else if(!unsafe.compareAndSwapObject(this, waitersOffset, q, s))
                logger.info("q : " + q + ", i:" + i);
                continue retry; // 这一步还是 cheak for race
        }
    }
    break ;
}
logger.info("removeWaiter node" + node + ", i: " + i + " end");

```

(/apps/redirect?  
utm\_source=side-  
banner-click)



```
    }  
}
```

removeWaiter 这个方法我认为是最复杂的, 你需要考虑多种情况(1. 移除的节点是队列的头节点, 2. 移除的节点是队列中的中间节点, 3. 在并发情况下, 两个线程同时removeWaiter操作), 重要的地方我都加了注解

(/apps/redirect?utm\_source=side-banner-click)

至此FutureTask的源码我们都差不多看到了, 总结一下实现Future的要点

- 1. 需要实现一个链表(每个节点包含当前线程的引用)
  - 2. 通过 Thread.wait 或LockSupport.park 对线程进行阻塞
  - 3. 有个公共方法进行节点的唤醒(task完成, 线程Interrupt, 或await超时), 并且次方法要线程安全
- 有了这些在大脑中, 相信自己实现一个简易Future也不是什么难事

小礼物走一走，来简书关注我


赞赏支持



(/u/bbb6b32d09d6)

📖 日记本 (/nb/541675)

📄 举报文章    © 著作权归作者所有



爱吃鱼的KK (/u/f9577ef03ae0)

写了 38903 字, 被 173 人关注, 获得了 100 个喜欢

(/u/f9577ef03ae0)


+ 关注



喜欢 | 6

   更多分享

(/apps/redirect?  
utm\_source=side-  
banner-click)



下载简书 App ▶  
随时随地发现和创作内容



(/apps/redirect?utm\_source=note-bottom-click)



登录后发表评论 (/sign-in?utm\_source=desktop&utm\_medium=not-signed-in-comment-fo

1条评论 只看作者

按时间倒序 按时间正序



rockjh (/u/9c59d564e5ad)

2楼 · 2017.11.24 22:34

(/u/9c59d564e5ad)

不错不错，正在看多线程源码这块

赞 回复



被以下专题收入，发现更多相似内容



IT面试 (/c/7f66d90f31dd?utm\_source=desktop&utm\_medium=notes-included-collection)



JVM虚拟机&... (/c/4b6fb8ba9eac?utm\_source=desktop&utm\_medium=notes-included-collection)

(/apps/redirect?utm\_source=side-banner-click)

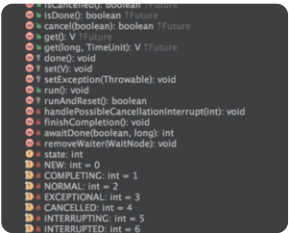
Java并发 ( 2 ) (/p/106a86ef02c7?utm\_campaign=maleskine&utm\_cont...

1.如何暂停或恢复线程 在JDK中提供了以下两个方法 ( 类Thread ) 用来暂停线程和恢复线程。 Øsuspend方法: 暂停线程 Øresume方法: 恢复线程 Østop方法: 终止线程 这两个方法和stop方法一样是被废弃的方法...



pjl2011 (/u/270d1aaf2df8?utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/23f6ef4cf441?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

Android Handler机制12之Callable、 Future和FutureTask (/p/23f6ef4cf44...

Android Handler机制系列文章整体内容如下: Android Handler机制1之ThreadAndroid Handler机制2之ThreadLocalAndroid Handler机制3之SystemClock类Android Handler机制4之Loo...




隔壁老李头 (/u/8b9c629f69dd?utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

多线程知识梳理(1) - 并发编程的艺术笔记 (/p/0d77a717c52a?utm\_campai...



第三章 Java内存模型 3.1 Java内存模型的基础 通信在共享内存的模型里, 通过写-读内存中的公共状态进行隐式通信; 在消息传递的并发模型里, 线程之间必须通过发送消息来进行显示的通信。 同步在共享内存并...



 泽毛 (/u/37baa8a86582?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/4b0721633009?



(/apps/redirect?  
utm\_source=side-  
banner-click)

utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## Java核心技术点之多线程 (/p/4b0721633009?utm\_campaign=maleskine...)

\*\*本文主要从整体上介绍Java中的多线程技术，对于一些重要的基础概念会进行相对详细的介绍，若有叙述不清晰或是不正确的地方，希望大家指出，谢谢大家：）\*\* 为什么使用多线程 并发与并行 我们知道，在...




**A** absfree (/u/640ce09fd6ec?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

Java 并发入门 (/p/9415c51ea38a?utm\_campaign=maleskine&utm\_cont...

一、并发 进程：每个进程都拥有自己的一套变量 线程：线程之间共享数据 1.线程 Java中为多线程任务提供了很多的类。包括最基础的Thread类、Runnable等接口，用于线程同步的锁、阻塞队列、同步器，使用线...



 SeanMa (/u/e9e33496e846?)

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

(/p/1a851da3b7b9?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendauiuiui)



## 杰出人物老王 (/p/1a851da3b7b9?utm\_campaign=maleskine&utm\_cont...

老王连续四年被评为村里的杰出青年，因为老实本分，生活寂静无声顽固原则性又特别强，老王对于杰出人物做到极致，雷打不动的锦旗一面面送到他手里。这天，老王走到一家金店门口，想想口袋的空空如也...



甜暖我歆 (/u/7f76f2e47f66?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation) (/apps/redirect?utm\_source=side-banner-click)

---

## UIBezierPath贝塞尔弧线 (/p/84cf6d2d4d3f?utm\_campaign=maleskine&...

下面两个网页是我学习时发现的相关的知识网页，总结了常用的知识点。以免忘记，特记录在此。

UIBezierPath贝塞尔弧线常用方法iOS UIBezierPath类介绍 几点需要注意：绘制的图形要写在drawRect函...



落夏简叶 (/u/d01bec4245b2?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

---

(/p/23864770fe00?



utm\_campaign=maleskine&utm\_content=note&utm\_medium=seo\_notes&utm\_source=recommendation)

## 长投学堂——坐拥财富自由，笑傲理财江湖 (/p/23864770fe00?utm\_campa...

前言：水湄物语是谁？她是谁？所有见过她本人的，听过她声音的都很喜欢她，为什么很多理财小白喜欢她？她创办了什么公司？她是畅销书《三十岁前每一天》的作者，而每一个长投理财小白课的院生都可以...



慢10拍 (/u/fcaae510e51d?

utm\_campaign=maleskine&utm\_content=user&utm\_medium=seo\_notes&utm\_source=recommendation)

