疯狂创客圈

Future 异步回调 大起底之 Java Future 与 Guava Future

文章很长，而且持续更新，建议收藏起来，慢慢读！**疯狂创客圈总目录 博客园版** 为您奉上珍贵的学习资源：

目录

公告

昵称： 疯狂创客圈
园龄： 5年4个月
粉丝： 1480
关注： 0
关注成功

< 2024年1月 >

日	一	二	三	四	五	六
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

写在前面
1.1. 从泡茶的案例说起
1.2. 何为异步回调
» 1.2.1. 同步、异步、阻塞、非阻塞
» 1.2.2. 阻塞模式的泡茶案例图解
» 1.2.3. 回调模式的泡茶方法
1.3. 异步阻塞闷葫芦——join
» 1.3.1. 线程的join 合并
» 1.3.2. join 异步阻塞实例代码
» 1.3.3. join方法的详细介绍
1.4. 异步阻塞重武器——FutureTask系列类
» 1.4.1. Callable接口
» 1.4.2. FutureTask类初探
» 1.4.3. Future接口
» 1.4.4. FutureTask再次深入
» 1.4.5. 喝茶实例演进之——获取异步结果
» 1.4.6. FutureTask使用流程
1.5. Guava 的异步回调
» 1.5.1. 能力导入 —— FutureCallback
» 1.5.2. 能力扩展 —— ListenableFuture
» 1.5.3. ListenableFuture 实例从何而来
» 1.5.4. Guava异步回调的流程
» 1.5.5. 喝茶实例 —— 异步回调演进
写在最后

回到：疯狂创客圈总目录
免费领取：入大厂、拿高薪、做架构、必备
的经典图书

源码IDEA工程获取链接：[Java 聊天室 实战 源码]

写在前面

大家好，我是作者尼恩。 目前和几个小伙伴一起，组织了一个高并发的实战社群【疯狂创客圈】。正在开始 高并发、亿级流程的 IM 聊天程序 学习和实战，此文是：

疯狂创客圈 Java 分布式聊天室【 亿级流量】实战系列之 -17

前面，已经完成一个高性能的 Java 聊天程序的四件大事：

1. 完成了协议选型，选择了性能更佳的 **Protobuf协议**。具体的文章为：Netty+Protobuf 整合一：实战案例，带源码
2. 介绍了 **通讯消息数据包的几条设计准则**。具体的文章为： Netty +Protobuf 整合二：protobuf 消息通讯协议设计的几个准则
3. 解决了一个非常基础的问题，这就是通讯的 **粘包和半包问题。**具体的文章为：Netty 粘包/半包 全解 | 史上最全解读
4. 前一篇文件，已经完成了 系统三大组成模块的组成介绍。 具体的文章为：Netty聊天程序（实战一）：从0开始实战100w级流量应用

在设计客户端之前，发现一个非常重要的基础知识点，没有讲到。这个知识点就是异步回调。

由于异步回调使用频率是如此之高，所以不得不停下来，详细介绍一下。

> 说明：本文会以pdf格式持续更新，更多最新尼恩3高pdf笔记，请从下面的链接获取：语雀 或者 码云

1. Future模式异步回调大起底

随着移动互联网的蓬勃发展，业务架构也随之变得错综复杂，业务系统越来越多。打个简单的比方：之前一个业务只需要调取一次第三方接口，如今，该业务需调取多个甚至N个不同的第三方接口，获取N种上游数据。通常，我们处理方法是异步去调取这些接口。

问题就来了，如何获取处理异步调用的结果呢？

或者说，异步线程执行完成后，如何与发起线程交互呢?

这就涉及到线程的异步回调问题，这也是大流量高并发不可回避的问题。

首先，了解下同步、异步、阻塞、非阻塞、回调等相关概念；

其次，简单介绍java future和guava future相关技术，并通过示例代码进一步对其进行理解；

最后，对java future和guava future进行比较。

1.1. 从泡茶的案例说起

写到这里，尼恩就想到了在中学8年级的语文课。在课本中，有一篇华罗庚的课文——《统筹方法》，课文介绍的是统筹方法，该方法的主要目的是合理安排工作流程中的各道工序。

里边举了一个泡茶的例子。列出了三种泡茶的工序模型。在文中的三种工序流程中，有多重排列组合的模式。

工序模型一：顺序模式

洗好水壶，灌上凉水，放在火上；

等水开，洗茶壶、洗茶杯；

洗完茶杯后，泡茶喝。

工序模型二：并发模式

洗好水壶，灌上凉水，放在火上；

在等待水开的时间里，洗茶壶、洗茶杯；

等水开了，泡茶喝。

《统筹方法》这篇文章中，忽略了一个很很重要的问题： 就是等水开是一段数量级最大的时间，这个时间，远远超过了准备水、准备茶杯的时间。

从实际出发，为了不浪费等水开时间，尼恩在这里增加一个动作 —— 读书。并且，当水烧好后，通知作者停止读书，去泡茶喝。这就相当于回调模式。

工序模式三：回调模式

洗好水壶，灌上凉水，放在火上；

在等待水开的时间里，洗茶壶、洗茶杯；

在等水开的时间里，读书；

水开了，通知作者泡茶喝。

对比起来：顺序模式效率最低，回调模式效率最高。

以上三种模式泡茶喝的方式，使用Java，如何实现呢？

先来看一些基本的概念吧！

> 说明：本文会以pdf格式持续更新，更多最新尼恩3高pdf笔记，请从下面的链接获取：语雀 或者 码云

1.2. 何为异步回调

前面只是一个例子，对并发的主要模式进行形象的说明。

下面正式来说下常用的几个和并发相关的概念。

1.2.1. 同步、异步、阻塞、非阻塞

一：同步

所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做，等前一件做完了才能做下一件事。

单线程模式，就是绝对同步的。

二： 异步

异步首先必须是多线程模式。是指当前线程，向其他的异步线程发出调用指令。当前线程和异步线程，逻辑上同时执行。

三：阻塞

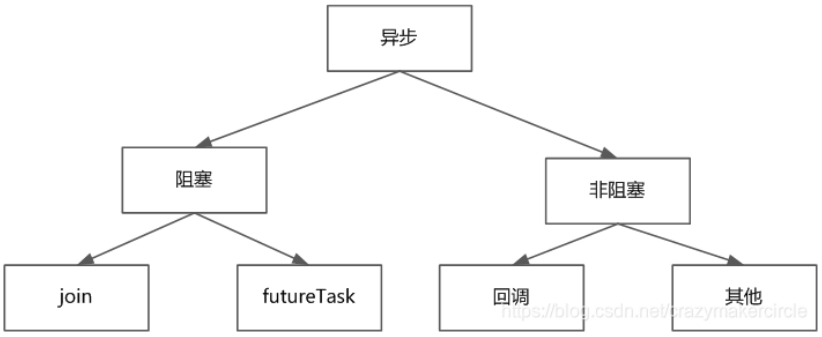
在异步的场景下，当前线程阻塞住，等待异步线程的执行结果。阻塞是指线程进入非可执行状态，在这个状态下，cpu不会给线程分配时间片，即线程暂停运行。

阻塞模式是效率比较低的，如果阻塞严重的话，相当于又回到了同步的时代。

四：非阻塞

非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，当前线程不会阻塞住，而会继续向下执行。

回调就是一种非阻塞的异步模式。并发线程通过回调，可以将结果返回给发起线程。除了回调，还有其他的非阻塞异步模式，比如消息通讯、信号量等等。



1.2.2. 阻塞模式的泡茶案例图解

阻塞模式的泡茶模型，对应到前面的第二种泡茶喝的工序模型。

在阻塞模式泡茶喝的模型中，有三条线程，他们分别是：

线程一：烧水线程

洗好水壶，灌上凉水，放在火上；

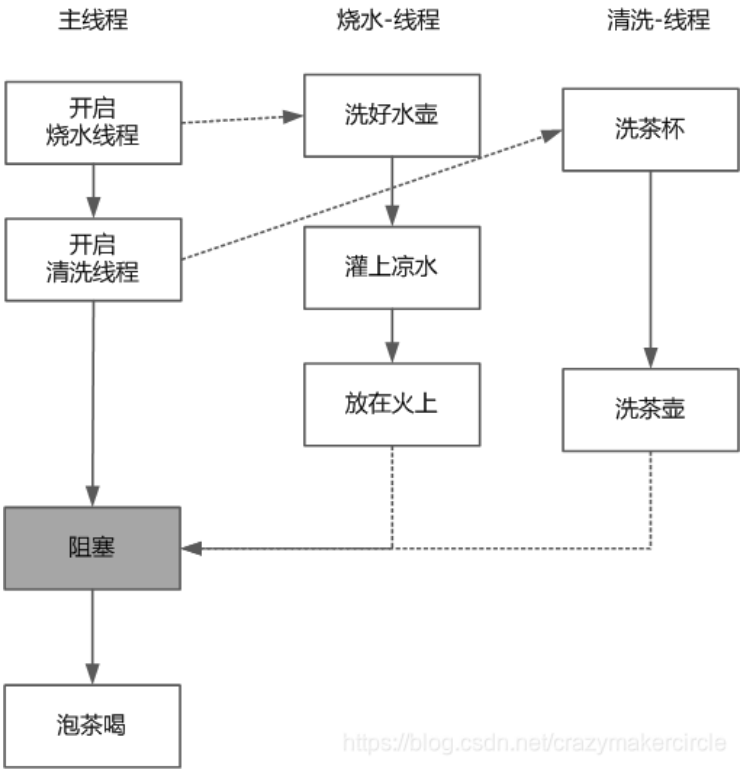
线程二：清洗线程

洗茶壶、洗茶杯；

线程三：主线程

分别启动烧水线程、清洗线程。等水开了，等水杯洗好了，然后泡茶喝。

具体如下图：



1.2.3. 回调模式的泡茶方法

前面提到，阻塞模式的效率不是最高的。

更高效率的是回调模式。主线程在等待的时间了，不是死等，而是去干读书的活儿。等其他两条线程完成后，通过回调方式，去完成泡茶的动作。

在回调模式泡茶喝的模型中，还是三条线程，他们的工作稍微有些变动：

线程一：烧水线程

洗好水壶，灌上凉水，放在火上；烧好水后，去执行泡茶回调。

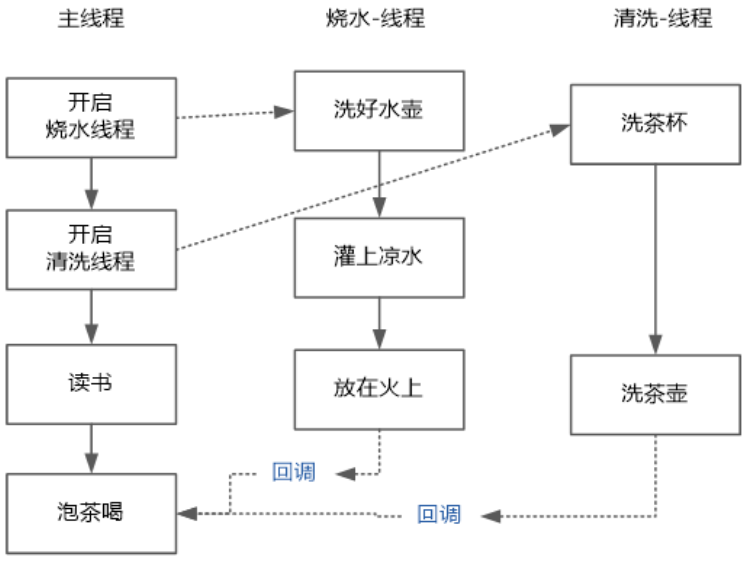
线程二：清洗线程

洗茶壶、洗茶杯；清洗完成后，也去执行一下泡茶的动作。

线程三：主线程

分别启动烧水线程、清洗线程。然后去读书。

具体如下图：



严格来说，上图是经不起推敲的。

为啥呢？ 那个泡茶喝回调方法，在执行的流程上，不属于主线程在执行。只是在业务逻辑上，泡茶喝这个动作与主线程上的其他动作，关联性更强。

上图，更好的理解方式是，尽量站在业务流程的角度去理解。

回调不是唯一的非阻塞方式。

还有线程间通信、信号量等等，很多的非阻塞方式。但是回调却是一种最好用的、也是开发中用的最多的线程间非阻塞的交互方式。

下面，从最原始的阻塞模式讲起，起底整个异步回调模式。

> 说明：本文会以pdf格式持续更新，更多最新尼恩3高pdf笔记，请从下面的链接获取：语雀 或者 码云

1.3. 异步阻塞闷葫芦——join

Java中，线程有一个join操作，也叫线程的合并。

join操作的作用，就是完成异步阻塞的工作——阻塞当前的线程，直到异步的并发线程的执行完成。

1.3.1. 线程的join 合并

如果线程A的执行过程中，通过B.join操作，合并B线程，叫做线程的合并。合并的重要特点之一是，线程A进入阻塞模式，直到B线程执行完成。

为了方便表达，模拟一下包工头的甲方和乙方。

将发起合并的线程A叫做甲方线程，被发起的线程B为乙方线程。

简单的说，线程合并就是——甲方等待乙方执行完成。换句话说，甲方将乙方线程合并到甲方线程。

在泡茶喝的例子中，主线程通过join操作，等待烧水线程和清洗线程。这就是一种异步阻塞。

具体如下图：

![img](file:///C:\Users\qinglin\AppData\Local\Temp\ksohtml\wps2659.tmp.png)

1.3.2. join 异步阻塞实例代码

先看实例，再看方法的详细介绍。

泡茶喝的异步阻塞版本，实现如下：

```
package com.crazymakercircle.coccurent;

import com.crazymakercircle.util.Print;

/**
 * Created by 尼恩 at 疯狂创客圈
 */

public class JoinDemo {

    public static final int SLEEP_GAP = 500;


    public static String getCurThreadName() {
        return Thread.currentThread().getName();
    }

    static class HotWarterThread extends Thread {


        public HotWarterThread() {
            super("** 烧水-Thread");
        }

        public void run() {

            try {
                Print.tcfo("洗好水壶");
                Print.tcfo("灌上凉水");
                Print.tcfo("放在火上");

                //线程睡眠一段时间，代表烧水中
                Thread.sleep(SLEEP_GAP);
                Print.tcfo("水开了");

            } catch (InterruptedException e) {
```

```
                Print.tcfo(" 发生异常被中断.");
            }
            Print.tcfo(" 运行结束.");
        }

    }

  static class WashThread extends Thread {


        public WashThread() {
            super("$$ 清洗-Thread");
        }

        public void run() {

            try {
                Print.tcfo("洗茶壶");
                Print.tcfo("洗茶杯");
                Print.tcfo("拿茶叶");
                //线程睡眠一段时间，代表清洗中
                Thread.sleep(SLEEP_GAP);
                Print.tcfo("洗完了");

            } catch (InterruptedException e) {
                Print.tcfo(" 发生异常被中断.");
            }
            Print.tcfo(" 运行结束.");
        }

    }


    public static void main(String args[]) {

        Thread hThread = new HotWarterThread();
        Thread wThread = new WashThread();

        hThread.start();
        wThread.start();
        try {
            // 合并烧水-线程
            hThread.join();
            // 合并清洗-线程
            wThread.join();

            Thread.currentThread().setName("主线程");
            Print.tcfo("泡茶喝");

        } catch (InterruptedException e) {
            Print.tcfo(getCurThreadName() + "发生异常被中断.");
        }
        Print.tcfo(getCurThreadName() + " 运行结束.");
    }
}
```

演示程序中有三条线程：

一条是主线程main；

一条是烧水线程“hThread”；

一条是清洗线程“wThread”；

main线程，调用了hThread.join()实例方法，合并烧水线程，也调用了 wThread.join()实例方法，合并清洗线程。

另外说明一下：hThread是这里的烧水线程实例的句柄，"** 烧水-Thread"是烧水线程实例的线程名称，两者不能混淆。

1.3.3. join方法的详细介绍

join的方法应用场景：异步阻塞场景。

具体来说：甲方（发起线程）的调用乙方（被发起线程）的join方法，等待乙方执行完成；如果乙方没有完成，甲方阻塞。

join是Thread类的一个实例方法，使用的方式大致如下：

```
// 合并烧水-线程
hThread.join();
// 合并清洗-线程
wThread.join();
```

实际上，join方法是有三个重载版本：

（1）void join(): 等待乙方线程执行结束，甲方线程重启执行。

（2）void join(long millis): 等待乙方线程执行一段时间，最长等待时间为 millis 毫秒。超过millis 毫秒后，不论乙方是否结束，甲方线程重启执行。

（3）void join(long millis， int nanos): 等待乙方线程执行一段时间，最长等待时间为millis 毫秒，加nanos 纳秒。超过时间后，不论乙方是否结束，甲方线程重启执行。

强调一下容易混淆的几点：

（1）join方法是实例方法，需要使用线程句柄去调用，如thread.join()；

（2）执行到join代码的时候，不是thread所指向的线程阻塞，而是当前线程阻塞；

（3）thread线程代表的是被合并线程（乙方），当前线程阻塞线程（甲方）。当前线程让出CPU，进入等待状态。

（4）只有等到thread线程执行完成，或者超时，当前线程才能启动执行。

join合并有一个很大的问题，就是没有返回值。

如果烧水线程的水有问题，或者烧水壶坏了，mian线程是没有办法知道的。

如果清洗线程的茶杯有问题，清洗不来了，mian线程是没有办法知道的。

形象的说，join线程就是一个闷葫芦。

还是异步阻塞，但是需要获得结果，怎么办呢？

可以使用java 的FutureTask 系列类。

> 说明：本文会以pdf格式持续更新，更多最新尼恩3高pdf笔记，请从下面的链接获取：语雀 或者 码云

1.4. 异步阻塞重武器——FutureTask系列类

FutureTask相关的类型，处于java.util.concurrent包中，不止一个类，是一个系列。同时，这也是Java语言在1.5 版本之后提供了一种的新的多线程使用方法。

1.4.1. Callable接口

我们知道，异步线程的一个重要接口是Runnable，这里执行异步线程的业务代码。但是，Runnable的run方法有一个问题，它是没有返回的。

因此，Runnable不能用在需要有异步返回值的异步场景。

Java语言在1.5 版本之后重新定义了一个新的、类似Runnable的接口，Callable接口，将run方法改为了call方法，并且带上了返回值。

Callable的代码如下：

```
package java.util.concurrent;

@FunctionalInterface

public interface Callable<V> {

    V call() throws Exception;

}
```

Callable接口位于java.util.concurrent包中，Callable接口是一个泛型接口。也是一个“函数式接口”。唯一的抽象方法call有返回值，返回值类型为泛型形参类型。call抽象方法还有一个Exception的异常声明，容许方法的实现版本内部的异常不经过捕获。

Callable接口类似于Runnable。不同的是，Runnable的唯一抽象方法run没有返回值，也没有强制审查异常的异常声明。比较而言，Callable接口的功能更强大一些。

有一个异想天开的问题：

作为新版的Callable接口实例，能否作为Thread线程实例的target来使用呢？

答案是不能。

Callable接口与Runnable接口之间没有任何的继承关系，而且二者唯一方法在的名字上也不同。Callable接口实例没有办法作为Thread线程实例的target来使用。

我们知道，java里边的线程类型，就是Thread。Callable需要异步执行，就需要和Thread建立联系。java提供了一个搭桥的角色——FutureTask类。

1.4.2. FutureTask类初探

顾名思义，这个是一个未来执行的任务，就相当于新线程所执行的操作。

FutureTask 类也位于 java.util.concurrent包。

FutureTask类 构造函数的参数为 Callable，并且间接的继承了Runnable接口。其构造器代码如下：

```
public FutureTask(Callable<V> callable) {
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;       // ensure visibility of callable
}
```

到了这里，FutureTask类的作用就大致明白了。

如果还不明白，看一段实例代码：

```
Callable<Boolean> hJob = new HotWarterJob();
FutureTask<Boolean> hTask =
        new FutureTask<Boolean>(hJob);
Thread hThread = new Thread(hTask, "** 烧水-Thread");
```

FutureTask就像一座位于Callable与Thread之间的桥。FutureTask 封装一个Callable，然后自身又作为Thread线程的target。

FutureTask还有一个十分重要的贡献。

Thread线程执行过程中，异步线程的代码逻辑在Callable的call方法中，而call方法返回的结果，则需要通过 FutureTask 去获取。

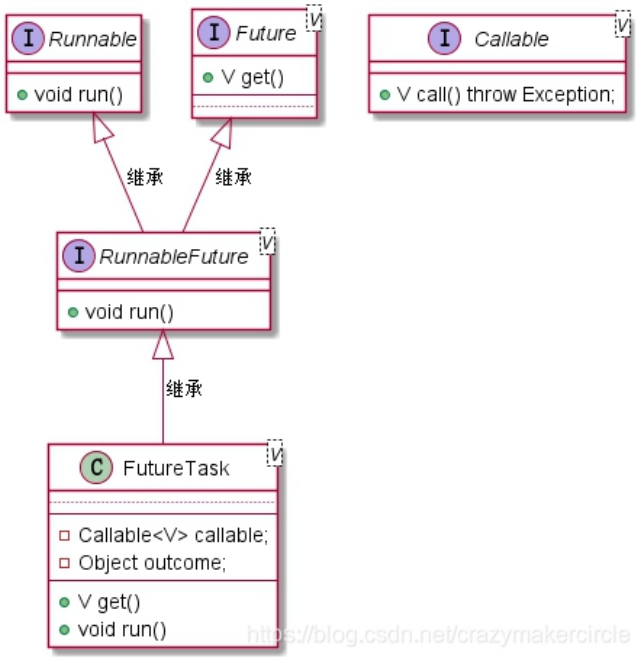
好了，这下就应该基本清楚了。

总结一下FutureTask这个媒婆的作用：

（1）负责牵线

（2）通过媒婆取得结果

为了完成这个两个伟大的使命，FutureTask有个相对比较复杂的继承关系，具体如下图：



首先，FutureTask实现了一个接口——RunnableFuture接口，而该RunnableFuture接口继承了Runnable接口和Future接口。

Runnable接口我们很熟悉，就是那个java 线程Runnable，代表异步线程的代码逻辑。

Future接口又是啥呢？

提前剧透下，这个接口，就是用来获取异步线程结果的。

Future接口和Runnable接口一样，都是牛气冲天的接口。 而FutureTask 间接的实现这个两大接口。

正因为FutureTask能够有两个很牛逼的爹，所以自己家才很牛逼。

FutureTask 既能当做一个Runnable 作为 target ，直接被Thread执行；也能作为Future用来去取得Callable的计算结果。

1.4.3. Future接口

Future接口这个不是一个复杂的接口，梳理一下，主要提供了3大功能：

（1）获取并发的任务完成后的执行结果。

（2）能够取消并发执行中的任务；

（3）判断并发任务是否执行完成；

当然，第一点是最为常用的。也是这个接口的最初使命。

Future接口的代码如下：

```
package java.util.concurrent;

public interface Future<V> {

    boolean cancel(boolean mayInterruptRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException,  ExecutionException;

    V get(long timeout, TimeUnit unit) throws InterruptedException,
ExecutionException,  TimeoutException;
}
```

对Future接口的方法，详细说明如下：

V get()：获取并发任务执行的结果。注意，这个方法是阻塞性的。如果并发任务没有执行完成，调用此方法的线程会一直阻塞，直到并发任务执行完成。

V get(Long timeout， TimeUnit unit)：获取并发任务执行的结果。也是阻塞性的，但是会有阻塞的时间限制，如果阻塞时间超过设定的timeout时间，该方法将抛出异常。

boolean isDone()：获取并发任务的执行状态。如果任务执行结束，返回true。

boolean isCancelled()：获取并发任务的取消状态。如果任务完成前被取消，则返回true。

boolean cancel(boolean mayInterruptRunning)：取消并发任务的执行。

1.4.4. FutureTask再次深入

说完了FutureTask的两个爹，再来到FutureTask自身。

在FutureTask内部，又有哪些成员和方法，具体的执行并发任务、异步获取任务结果的呢?

首先，FutureTask内部有一个 Callable类型的成员：

private Callable callable;

这个callable实例属性，是构造器传进来的。用来保存并发执行的 Callable类型的任务。callable实例属性，是构造器强制性的，必须要在FutureTask实例构造的时候进行初始化。

其次，FutureTask内部有一个run方法。

这个run方法，是Runnable接口在FutureTask内部的实现。在这个run方法其中，会执行到callable成员的call方法。执行完成后，结果如何提供出去呢？这就是到了最后一

点。

最后，FutureTask内部有另一个 Object 类型的重要成员——outcome实例属性：

```
private Object outcome;
```

掐指一算，就知道这个outcome属性，是用来保存callable成员call方法的执行结果。FutureTask类run方法执行完成callable成员的call方法后，会将结果保存在outcome实例属性，供FutureTask类的get实例方法获取。

好了，重要将这个媒婆介绍完了。

如果还没有清楚，不要紧，看一个实例就一目了然了。

1.4.5. 喝茶实例演进之——获取异步结果

回顾一下，前面的join闷葫芦合并阻塞有一个很大的问题，就是没有返回值。

如果烧水线程的水有问题，或者烧水壶坏了，mian线程是没有办法知道的。

如果清洗线程的茶杯有问题，清洗不来了，mian线程是没有办法知道的。

为了演示结果，给主类增加两个成员：

```
static boolean  warterOk = false;
static boolean  cupOk =false;
```

代表烧水成功和清洗成功。初始值都为false。

烧水线程、清洗线程执行完后，都需要返回结果。 主线程获取后，保存在上面的两个主类成员中。

废话不多说，看代码：

```
package com.crazymakercircle.coccurent;

import com.crazymakercircle.util.Print;

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

/**
 * Created by 尼恩 at 疯狂创客圈
 */

public class JavaFutureDemo
{

    public static final int SLEEP_GAP = 500;


    public static String getCurThreadName()
    {
        return Thread.currentThread().getName();
    }

    static class HotWarterJob implements Callable<Boolean> //①
    {

        @Override
        public Boolean call() throws Exception //②
        {
```

```
            try
            {
                Print.tcfo("洗好水壺");
                Print.tcfo("灌上凉水");
                Print.tcfo("放在火上");

                //线程睡眠一段时间，代表烧水中
                Thread.sleep(SLEEP_GAP);
                Print.tcfo("水开了");

            } catch (InterruptedException e)
            {
                Print.tcfo(" 发生异常被中断.");
                return false;
            }
            Print.tcfo(" 运行结束.");

            return true;
        }
    }

    static class WashJob implements Callable<Boolean>
    {

        @Override
        public Boolean call() throws Exception
        {


            try
            {
                Print.tcfo("洗茶壺");
                Print.tcfo("洗茶杯");
                Print.tcfo("拿茶叶");
                //线程睡眠一段时间，代表清洗中
                Thread.sleep(SLEEP_GAP);
                Print.tcfo("洗完了");

            } catch (InterruptedException e)
            {
                Print.tcfo(" 清洗工作 发生异常被中断.");
                return false;
            }
            Print.tcfo(" 清洗工作  运行结束.");
            return true;
        }

    }

    static boolean  warterOk = false;
    static boolean cupOk =false;


    public static void drinkTea()
    {
        if (warterOk && cupOk)
        {
            Print.tcfo("泡茶喝");
        }
        else if (!warterOk)
        {
            Print.tcfo("烧水失败，没有茶喝了");
        }
        else if (!cupOk)
        {
```

的经典图书

»领取：《Java高并发核心编程（卷1）》

回到：疯狂创客圈总目录

免费领取：入大厂、拿高薪、做架构、必备的经典图书

```
            Print.tcfo("杯子洗不了，没有茶喝了");
        }

    }
    public static void main(String args[])
    {

        Callable<Boolean> hJob = new HotWarterJob();//③
        FutureTask<Boolean> hTask =
                new FutureTask<Boolean>(hJob);//④
        Thread hThread = new Thread(hTask, "** 烧水-Thread");//⑤

        Callable<Boolean> wJob = new WashJob();//③
        FutureTask<Boolean> wTask =
                new FutureTask<Boolean>(wJob);//④
        Thread wThread = new Thread(wTask, "$$ 清洗-Thread");//⑤


        hThread.start();
        wThread.start();
        Thread.currentThread().setName("主线程");

        try
        {

             warterOk = hTask.get();
             cupOk = wTask.get();

//            hThread.join();
//            wThread.join();
             drinkTea();


        } catch (InterruptedException e)
        {
            Print.tcfo(getCurThreadName() + "发生异常被中断.");
        } catch (ExecutionException e)
        {
            e.printStackTrace();
        }
        Print.tcfo(getCurThreadName() + " 运行结束.");
    }
}
```

1.4.6. FutureTask使用流程

借助上面的喝茶实例代码，说明一下通过FutureTask获取异步结果的流程步骤：

（1）异步代码逻辑需要继承Callable，通过call方法返回具体的值

```
static class WashJob implements Callable<Boolean>
{
    @Override
    public Boolean call() throws Exception
    {

//..业务代码，并且有返回值

}
```

（3）从异步逻辑到异步线程，需要媒婆类FutureTask搭桥

```
Callable<Boolean> hJob = new HotWarterJob();//异步逻辑
FutureTask<Boolean> hTask =
```

```
        new FutureTask<Boolean>(hJob);//媒婆实例
Thread hThread = new Thread(hTask, "** 烧水-Thread");//异步线程
```

FutureTask和Callable都是泛型类，泛型参数表示返回结果的类型。所以，在使用的时候，俩个类型的泛型参数一定需要一致的。

（3）取得异步线程的执行结果，也需要FutureTask 媒婆实例做下二传

```
warterOk = hTask.get();
```

通过FutureTask 实例的get方法，可以获取线程的执行结果。

三步至此，结果到手。

总结一下，FutureTask 比 join 线程合并高明，能取得异步线程的结果。

但是，也就未必高明到哪里去了。为啥呢？

因为，通过FutureTask的get方法，获取异步结果时，主线程也会被阻塞的。这一点，FutureTask和join也是一致的，他们俩都是异步阻塞模式。

异步阻塞的效率是比较低的，被阻塞的主线程，不能干任何事情，唯一能干的，就是在傻傻等待。

如果想提高效率，需要用到非阻塞模式。这里只讲回调模式的非阻塞，其他模式的非阻塞，请关注疯狂创客圈的后续文章。

原生Java，除了阻塞模式的获取结果，并没有实现非阻塞模式的异步回调。如果需要用到异步回调，得引入一些额外的框架。

> 说明：本文会以pdf格式持续更新，更多最新尼恩3高pdf笔记，请从下面的链接获取：语雀 或者 码云

1.5. Guava 的异步回调

在非常著名的google 提供的扩展包 Guava中，提供了一种异步回调的解决方案。

为了实现异步回调，Guava 对Java的Future 异步模式进行能力导入：

（1）导入了一个新的接口 FutureCallback，代表回调执行的业务逻辑

（2）对Java并发包中的 Future 接口进行了扩展，将回调逻辑作为监听器绑定到异步线程

1.5.1. 能力导入 —— FutureCallback

FutureCallback 是一个新增的接口，用来填写回调逻辑。这个接口，是在实际开发中编程使用到的。回调的代码，编写在它的实现类中。

FutureCallback拥有两个回调方法：

（1）onSuccess ，在异步线程执行成功回调

（2）onFailure，在异步线程抛出异常时回调

FutureCallback的源码如下：

```
public interface FutureCallback<V> {
    void onSuccess(@Nullable V var1);
    void onFailure(Throwable var1);
}
```

1.5.2. 能力扩展 —— ListenableFuture

如果将回调方法，绑定到异步线程去呢？

Guava中，有一个非常关键的角色，ListenableFuture。看名称，就能对应出它与Java 中的原生接口的亲戚关系。

如果没有猜错，这个接口是 Guava 对java 的Future接口的扩展。

来看看 ListenableFuture接口的源码，如下：

```
package com.google.common.util.concurrent;
import java.util.concurrent.Executor;
import java.util.concurrent.Future;
public interface ListenableFuture<V> extends Future<V> {
    void addListener(Runnable var1, Executor var2);
}
```

前面讲到，通过Java的Future接口，可以阻塞取得异步的结果。在这个基础上，ListenableFuture增加了一个方法 —— addListener 。

这个方法的作用，就是将前一小节的FutureCallback 回调逻辑，绑定到异步线程上。可以是，addListener 不直接在实际编程中使用。这个方法只在Guava内部使用，如果对它感兴趣，可以查看Guava源码。

既然addListener 方法不能直接使用，那么，在实际编程中，如何将 FutureCallback 回调逻辑绑定到异步线程呢？

不慌，办法总是有的。

需要用到Guava的Futures 工具类。这个类有一个addCallback 静态方法，将ListenableFuture 的实例和FutureCallback 的回调实例，进行绑定。

绑定的示意代码如下：

```
Futures.addCallback( hFuture ,  new FutureCallback<Boolean>()
{
    public void onSuccess(Boolean r)
    {

//成功时候的回调逻辑
     }
    public void onFailure(Throwable t)
    {
    //异常时候的回调逻辑

    }
});
```

1.5.3. ListenableFuture 实例从何而来

从上文已知，原生java的Future接口的实例，一种方法是——直接构建媒婆类FutureTask的实例，就是Future接口的实例。

当然，还有第二种方法，就是通过线程池获取Future接口的实例。具体的做法是向Java线程池提交异步任务，包括Runnable或者Callable实例。

方法如下：

```
Future<Boolean> hTask = pool.submit(hJob);
Future<Boolean> wTask = pool.submit(wJob);
```

注意，pool 是一个Java 线程池。

如果要获取Guava的ListenableFuture 实例，主要是通过类似上面的第二种方式——向线程池提交任务的异步任务的方式获取。不过，用到的线程池，是Guava的线程池，不是Java的线程池。

Guava线程池，而是对Java线程池的一种装饰。

两种线程池的创建代码，具体如下：

```
//java 线程池
ExecutorService jPool =
        Executors.*newFixedThreadPool*(10);
 //guava 线程池
ListeningExecutorService gPool =
        MoreExecutors.*listeningDecorator*(jPool);
```

有了Guava的线程池之后，就可以通过提交任务，来获取ListenableFuture 实例了。代码如下 ：

```
ListenableFuture<Boolean> hFuture = gPool.submit(hJob);
```

关于Gava的线程池，请关注【疯狂创客圈】的线程池的博客文章。

1.5.4. Guava异步回调的流程

总结一下，Guava异步回调的流程如下：

第一步：创建Java的 Callable的异步任务实例。实例如下：

```
Callable<Boolean> hJob = new HotWarterJob();//异步任务
Callable<Boolean> wJob = new WashJob();//异步任务
```

异步任务也可以是Runnable类型。

第二步： 获取Guava线程池

```
//java 线程池
ExecutorService jPool =
        Executors.*newFixedThreadPool*(10);
//guava 线程池
ListeningExecutorService gPool =
        MoreExecutors.*listeningDecorator*(jPool);
```

第三步： 提交异步任务到Guava线程池，获取ListenableFuture 实例

```
ListenableFuture<Boolean> hFuture = gPool.submit(hJob);
```

第四步：创建回调的 FutureCallback 实例，通过Futures.addCallback，将回调逻辑绑定到ListenableFuture 实例。

```
Futures.*addCallback*( hFuture ,  new FutureCallback<Boolean>()
{
    public void onSuccess(Boolean r)
    {

//成功时候的回调逻辑
     }
    public void onFailure(Throwable t)
    {
```

```
    //异常时候的回调逻辑

    }
});
```

完成以上四步，当异步逻辑执行完成后，就会回调FutureCallback 实例中的回调代码。

1.5.5. 喝茶实例 —— 异步回调演进

已经对喝茶实例的代码非常熟悉下，下面是Guava的异步回调的演进版本，代码如下：

```
package com.crazymakercircle.coccurent;

import com.crazymakercircle.util.Print;
import com.google.common.util.concurrent.*;

import java.util.concurrent.*;

/**
 * Created by 尼恩 at 疯狂创客圈
 */

public class GuavaFutureDemo
{

    public static final int SLEEP_GAP = 500;


    public static String getCurThreadName()
    {
        return Thread.currentThread().getName();
    }

    static class HotWarterJob implements Callable<Boolean> //①
    {

        @Override
        public Boolean call() throws Exception //②
        {

            try
            {
                Print.tcfo("洗好水壶");
                Print.tcfo("灌上凉水");
                Print.tcfo("放在火上");

                //线程睡眠一段时间，代表烧水中
                Thread.sleep(SLEEP_GAP);
                Print.tcfo("水开了");

            } catch (InterruptedException e)
            {
                Print.tcfo(" 发生异常被中断.");
                return false;
            }
            Print.tcfo(" 运行结束.");

            return true;
        }
    }

    static class WashJob implements Callable<Boolean>
```

```
    {

        @Override
        public Boolean call() throws Exception
        {


            try
            {
                Print.tcfo("洗茶壶");
                Print.tcfo("洗茶杯");
                Print.tcfo("拿茶叶");
                //线程睡眠一段时间，代表清洗中
                Thread.sleep(SLEEP_GAP);
                Print.tcfo("洗完了");

            } catch (InterruptedException e)
            {
                Print.tcfo(" 清洗工作 发生异常被中断.");
                return false;
            }
            Print.tcfo(" 清洗工作  运行结束.");
            return true;
        }

    }

    static boolean warterOk = false;
    static boolean cupOk = false;


    public synchronized static void drinkTea()
    {
        if (warterOk && cupOk)
        {
            Print.tcfo("泡茶喝");
        }
        else if (!warterOk)
        {
            Print.tcfo("烧水失败，没有茶喝了");
        }
        else if (!cupOk)
        {
            Print.tcfo("杯子洗不了，没有茶喝了");
        }

    }

    public static void main(String args[])
    {
        Thread.currentThread().setName("主线程");

        Callable<Boolean> hJob = new HotWarterJob();//③

        Callable<Boolean> wJob = new WashJob();//③

        //java 线程池
        ExecutorService jPool =
                Executors.newFixedThreadPool(10);

        //guava 线程池
        ListeningExecutorService gPool =
                MoreExecutors.listeningDecorator(jPool);
```

的经典图书

»领取：《Java高并发核心编程（卷1）》

回到：疯狂创客圈总目录

免费领取：入大厂、拿高薪、做架构、必备的经典图书

```
        ListenableFuture<Boolean> hFuture = gPool.submit(hJob);

        Futures.addCallback(hFuture, new FutureCallback<Boolean>()
        {
            public void onSuccess(Boolean r)
            {
                if (r)
                {
                    warterOk = true;
                    drinkTea();
                }
                else
                {

                    Print.tcfo("烧水失败，没有茶喝了");

                }

            }

            public void onFailure(Throwable t)
            {
                Print.tcfo("烧水失败，没有茶喝了");
            }
        });


        ListenableFuture<Boolean> wFuture = gPool.submit(wJob);

        Futures.addCallback(wFuture, new FutureCallback<Boolean>()
        {
            public void onSuccess(Boolean r)
            {
                if (r)
                {
                    cupOk = true;
                    drinkTea();
                }
                else
                {

                    Print.tcfo("清洗失败，没有茶喝了");

                }
            }

            public void onFailure(Throwable t)
            {
                Print.tcfo("杯子洗不了，没有茶喝了");
            }
        });

        try
        {

            Print.tcfo("读书中......");
            Thread.sleep(100000);

        } catch (InterruptedException e)
        {
            Print.tcfo(getCurThreadName() + "发生异常被中断.");
        }
        Print.tcfo(getCurThreadName() + " 运行结束.");

        gPool.shutdown();
```

```
    }


}
```

本文已经太长，还有很多内容

未完待续

写在最后

为什么说异步回调是如此的重要呢？ 因为高并发编程，到处都用到Future模式和Callback模式。

下一篇：Netty 中的Future 回调实现与线程池详解。这个也是一个非常重要的基础篇。

> 说明：本文会以pdf格式持续更新，更多最新尼恩3高pdf笔记，请从下面的链接获取：语雀 或者 码云

标签: netty , java

« 上一篇： Netty聊天室-源码
» 下一篇： 第101次提醒：++ 不是线程安全的

posted @ 2018-12-07 23:58 疯狂创客圈 阅读(3846) 评论(0) 编辑 收藏 举报

编辑推荐：
· 一个例子形象地理解同步与异步
· C# 线程本地存储 为什么线程间值不一样
· ［动画进阶］神奇的 3D 卡片反光闪烁动效
· 记一次缓存失效引发的惨案！
· 记一次 .NET某MES自动化桌面程序 卡死分析

阅读排行：
· 通义灵码，降临博客园
· 他凌晨1:30给我开源的游戏加了UI｜模拟龙生，挂机冒险
· C# 线程本地存储 为什么线程间值不一样
· .NET集成IdGenerator生成分布式全局唯一ID
· 什么是 doris，为什么几乎国内大厂都会使用它