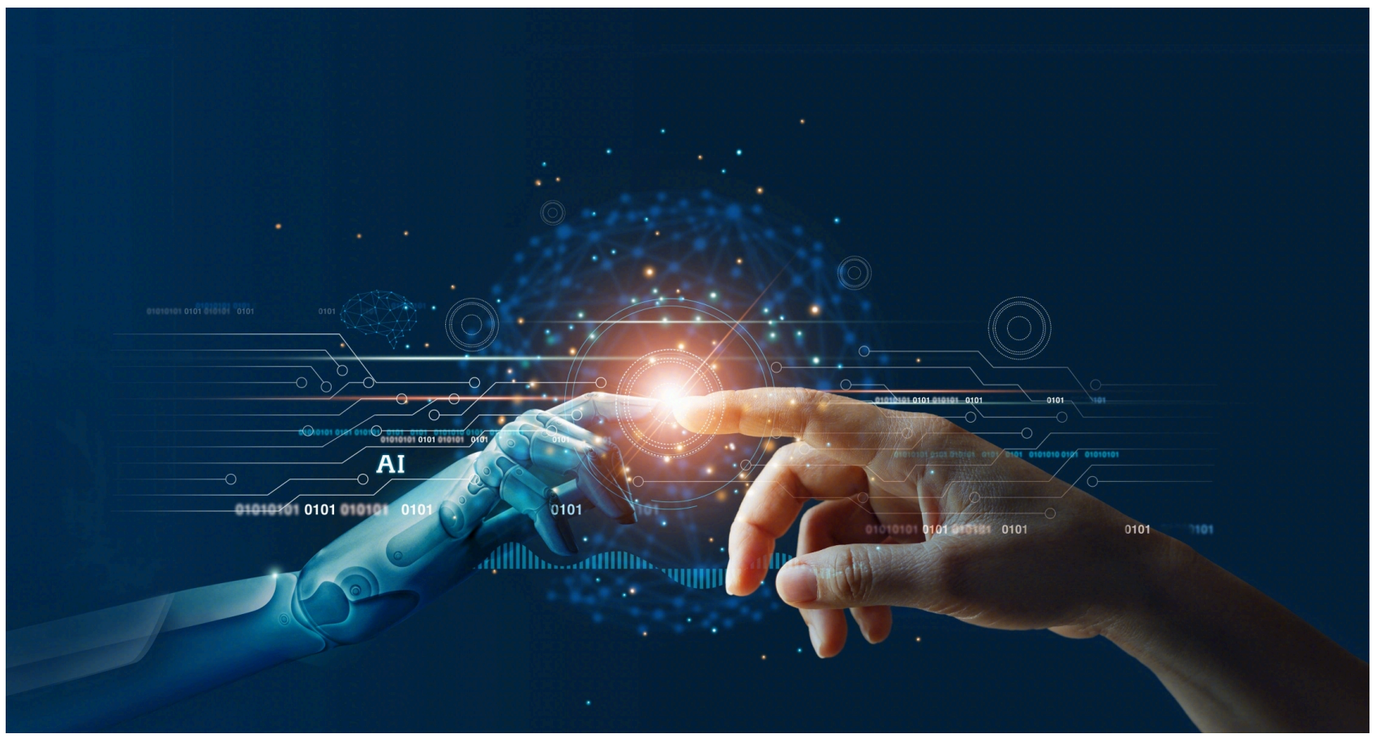
Java CompletableFuture 异步超时实现探索

作者：京东科技开发者

2023-02-08 北京 本文字数：5508 字 阅读完需：约 18 分钟



作者：京东科技 张天赐

前言

JDK 8 是一次重大的版本升级，新增了非常多的特性，其中之一便是 `CompletableFuture`。自此从 JDK 层面真正意义上的支持了基于事件的异步编程范式，弥补了 `Future` 的缺陷。

在我们的日常优化中，最常用手段便是多线程并行执行。这时候就会涉及到 `CompletableFuture` 的使用。

常见使用方式

下面举例一个常见场景。

假如我们有两个 RPC 远程调用服务，我们需要获取两个 RPC 的结果后，再进行后续逻辑处理。

复制代码

```
public static void main(String[] args) {
    // 任务 A，耗时 2 秒
    int resultA = compute(1);
    // 任务 B，耗时 2 秒
    int resultB = compute(2);

    // 后续业务逻辑处理
    System.out.println(resultA + resultB);
}

```

可以预估到，串行执行最少耗时 4 秒，并且 B 任务并不依赖 A 任务结果。

对于这种场景，我们通常会选择并行的方式优化，Demo 代码如下：

复制代码

```
public static void main(String[] args) {
    // 仅简单举例，在生产代码中可别这么写！

    // 统计耗时的函数
    time(() -> {
        CompletableFuture<Integer> result = Stream.of(1, 2)
                                                  // 创建异步任务
                                                  .map(x -> CompletableFuture.supplyAsync(() -> com
                                                  // 聚合
                                                  .reduce(CompletableFuture.completedFuture(0), (x,

        // 等待结果
        try {
            System.out.println("结果：" + result.get());
        } catch (ExecutionException | InterruptedException e) {
            System.err.println("任务执行异常");
        }
    });
}

输出：
[async-1]：任务执行开始：1
[async-2]：任务执行开始：2
[async-1]：任务执行完成：1
[async-2]：任务执行完成：2
结果：3
耗时：2 秒

```

存在的问题

分析

看上去 `CompletableFuture` 现有功能可以满足我们诉求。但当我们引入一些现实常见情况时，一些潜在的不足便暴露出来了。

`compute(x)` 如果是一个根据入参查询用户某类型优惠券列表的任务，我们需要查询两种优惠券并组合在一起返回给上游。假如上游要求我们 2 秒内处理完毕并返回结果，但 `compute(x)` 耗时却在 0.5 秒 ~ 无穷大波动。这时候我们就需要把耗时过长的 `compute(x)` 任务结果放弃，仅处理在指定时间内完成的任务，尽可能保证服务可用。

那么以上代码的耗时由耗时最长的服务决定，无法满足现有诉求。通常我们会使用 `get(long timeout, TimeUnit unit)` 来指定获取结果的超时时间，并且我们会给 `compute(x)` 设置一个超时时间，达到后自动抛异常来中断任务。

复制代码

```
public static void main(String[] args) {
    // 仅简单举例，在生产代码中可别这么写！

    // 统计耗时的函数
    time(() -> {
        List<CompletableFuture<Integer>> result = Stream.of(1, 2)
                                                        // 创建异步任务，compute(x) 超时抛出异常
                                                        .map(x -> CompletableFuture.supplyAsync(()
                                                        .toList();

        // 等待结果
        int res = 0;
        for (CompletableFuture<Integer> future : result) {
            try {
                res += future.get(2, SECONDS);
            } catch (ExecutionException | InterruptedException | TimeoutException e) {
                System.err.println("任务执行异常或超时");
            }
        }

        System.out.println("结果：" + res);
    });
}

输出：
[async-2]：任务执行开始：2
[async-1]：任务执行开始：1
[async-1]：任务执行完成：1
任务执行异常或超时
结果：1
耗时：2 秒

```

可以看到，只要我们能够给 `compute(x)` 设置一个超时时间将任务中断，结合 `get`、`getNow` 等获取结果的方式，就可以很好地管理整体耗时。

那么问题也就转变成了，**如何给任务设置异步超时时间呢**?

现有做法

当异步任务是一个 RPC 请求时，我们可以设置一个 JSF 超时，以达到异步超时效果。

当请求是一个 R2M 请求时，我们也可以控制 R2M 连接的最大超时时间来达到效果。

这么看好像我们都是在依赖三方中间件的能力来管理任务超时时间？那么就存在一个问题，中间件超时控制能力有限，如果异步任务是中间件 IO 操作 + 本地计算操作怎么办?

用 JSF 超时举一个具体的例子，反编译 JSF 的获取结果代码如下：

复制代码

```
public V get(long timeout, TimeUnit unit) throws InterruptedException {
    // 配置的超时时间
    timeout = unit.toMillis(timeout);
    // 剩余等待时间
    long remaintime = timeout - (this.sentTime - this.genTime);
    if (remaintime <= 0L) {
        if (this.isDone()) {
            // 反序列化获取结果
            return this.getNow();
        }
    } else if (this.await(remaintime, TimeUnit.MILLISECONDS)) {
        // 等待时间内任务完成，反序列化获取结果
        return this.getNow();
    }

    this.setDoneTime();
    // 超时抛出异常
    throw this.clientTimeoutException(false);
}

```

当这个任务刚好卡在超时边缘完成时，这个任务的耗时时间就变成了超时时间 + 获取结果时间。而获取结果（反序列化）作为纯本地计算操作，耗时长短受 CPU 影响较大。

某些 CPU 使用率高的情况下，就会出现异步任务没能触发抛出异常中断，导致我们无法准确控制超时时间。对上游来说，本次请求全部失败。

解决方式

JDK 9

这类问题非常常见，如大促场景，服务器 CPU 瞬间升高就会出现以上问题。

那么如何解决呢？其实 JDK 的开发大佬们早有研究。在 JDK 9，`CompletableFuture` 正式提供了 `orTimeout`、`completeTimeout` 方法，来准确实现异步超时控制。

复制代码

```
public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit) {
    if (unit == null)
        throw new NullPointerException();
    if (result == null)
        whenComplete(new Canceller(Delayer.delay(new Timeout(this), timeout, unit)));
    return this;
}

```

JDK 9 `orTimeout` 其实现原理是通过一个定时任务，在给定时间之后抛出异常。如果任务在指定时间内完成，则取消抛异常的操作。

以上代码我们按执行顺序来看下：

首先执行 `new Timeout(this)`。

复制代码

```
static final class Timeout implements Runnable {
    final CompletableFuture<?> f;
    Timeout(CompletableFuture<?> f) { this.f = f; }
    public void run() {
        if (f != null && !f.isDone())
            // 抛出超时异常
            f.completeExceptionally(new TimeoutException());
    }
}

```

通过源码可以看到，`Timeout` 是一个实现 Runnable 的类，`run()` 方法负责给传入的异步任务通过 `completeExceptionally` CAS 赋值异常，将任务标记为异常完成。

那么谁来触发这个 `run()` 方法呢？我们看下 `Delayer` 的实现。

复制代码

```
static final class Delayer {
    static ScheduledFuture<?> delay(Runnable command, long delay,
                                    TimeUnit unit) {
        // 到时间触发 command 任务
        return delayer.schedule(command, delay, unit);
    }

    static final class DaemonThreadFactory implements ThreadFactory {
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r);
            t.setDaemon(true);
            t.setName("CompletableFutureDelayScheduler");
            return t;
        }
    }

    static final ScheduledThreadPoolExecutor delayer;
    static {
        (delayer = new ScheduledThreadPoolExecutor(
            1, new DaemonThreadFactory())).
            setRemoveOnCancelPolicy(true);
    }
}

```

`Delayer` 其实就是一个单例定时调度器，`Delayer.delay(new Timeout(this), timeout, unit)` 通过 `ScheduledThreadPoolExecutor` 实现指定时间后触发 `Timeout` 的 `run()` 方法。

```
static final class Canceller implements BiConsumer<Object, Throwable> {
    final Future<?> f;
    Canceller(Future<?> f) { this.f = f; }
    public void accept(Object ignore, Throwable ex) {
        if (ex == null && f != null && !f.isDone())
        // 3 未触发抛异常任务则取消
            f.cancel(false);
    }
}
```

当任务执行完成，或者任务执行异常时，我们也就没必要抛出超时异常了。因此我们可以把 `delayer.schedule(command, delay, unit)` 返回的定时超时任务取消，不再触发 `Timeout`。 当我们的异步任务完成，并且定时超时任务未完成的时候，就是我们取消的时机。因此我们可以通过 `whenComplete(BiConsumer<? super T, ? super Throwable> action)` 来完成。

`Canceller` 就是一个 `BiConsumer` 的实现。其持有了 `delayer.schedule(command, delay, unit)` 返回的定时超时任务，`accept(Object ignore, Throwable ex)` 实现了定时超时任务未完成后，执行 `cancel(boolean mayInterruptIfRunning)` 取消任务的操作。

JDK 8

如果我们使用的是 JDK 9 或以上，我们可以直接用 JDK 的实现来完成异步超时操作。那么 JDK 8 怎么办呢?

其实我们也可以根据上述逻辑简单实现一个工具类来辅助。

以下是我们营销自己的工具类以及用法，贴出来给大家作为参考，大家也可以自己写的更优雅一些~

调用方式：

```
CompletableFutureExpandUtils.orTimeout(异步任务, 超时时间, 时间单位);
```

工具类源码：

```
package com.jd.jr.market.reduction.util;

import com.jdpay.market.common.exception.UncheckedException;

import java.util.concurrent.*;
import java.util.function.BiConsumer;

/**
 * CompletableFuture 扩展工具
 *
 * @author zhangtianci7
 */
public class CompletableFutureExpandUtils {

    /**
     * 如果在给定超时之前未完成，则异常完成此 CompletableFuture 并抛出 {@link TimeoutException} 。
     *
     * @param timeout 在出现 TimeoutException 异常完成之前等待多长时间，以 {@code unit} 为单位
     * @param unit    一个 {@link TimeUnit}，结合 {@code timeout} 参数，表示给定粒度单位的持续时间
     * @return 入参的 CompletableFuture
     */
    public static <T> CompletableFuture<T> orTimeout(CompletableFuture<T> future, long timeout, Tim
        if (null == unit) {
            throw new UncheckedException("时间的给定粒度不能为空");
        }
        if (null == future) {
            throw new UncheckedException("异步任务不能为空");
        }
        if (future.isDone()) {
            return future;
        }

        return future.whenComplete(new Canceller(Delayer.delay(new Timeout(future), timeout, unit))
    }

    /**
     * 超时时异常完成的操作
     */
    static final class Timeout implements Runnable {
        final CompletableFuture<?> future;

        Timeout(CompletableFuture<?> future) {
            this.future = future;
        }

        public void run() {
            if (null != future && !future.isDone()) {
                future.completeExceptionally(new TimeoutException());
```

```
/**
 * 取消不需要的超时的操作
 */
static final class Canceller implements BiConsumer<Object, Throwable> {
    final Future<?> future;

    Canceller(Future<?> future) {
        this.future = future;
    }

    public void accept(Object ignore, Throwable ex) {
        if (null == ex && null != future && !future.isDone()) {
            future.cancel(false);
        }
    }
}

/**
 * 单例延迟调度器，仅用于启动和取消任务，一个线程就足够
 */
static final class Delayer {
    static ScheduledFuture<?> delay(Runnable command, long delay, TimeUnit unit) {
        return delayer.schedule(command, delay, unit);
    }

    static final class DaemonThreadFactory implements ThreadFactory {
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r);
            t.setDaemon(true);
            t.setName("CompletableFutureExpandUtilsDelayScheduler");
            return t;
        }
    }

    static final ScheduledThreadPoolExecutor delayer;

    static {
        delayer = new ScheduledThreadPoolExecutor(1, new DaemonThreadFactory());
        delayer.setRemoveOnCancelPolicy(true);
    }
  }
}
```

参考资料

1. JEP 266: JDK 9 并发包更新提案

发布于: 2023-02-08 | 阅读数: 792

Java jdk RPC 多线程并发 企业号 2 月 PK 榜

京东科技开发者 + 关注
拥抱技术，与开发者携手创造未来！ 2018-11-20 加入
我们将持续为人工智能、大数据、云计算、物联网等相关领域的开发者，提供技术干货、行业技术内容、技术落地实践等文章内容。京东云开发者社区官方网站【https://developer.jdcloud.com/】，欢迎大家来玩

评论

快抢沙发！虚位以待 发布

暂无评论