CompletableFuture实现的工具类及使用示例

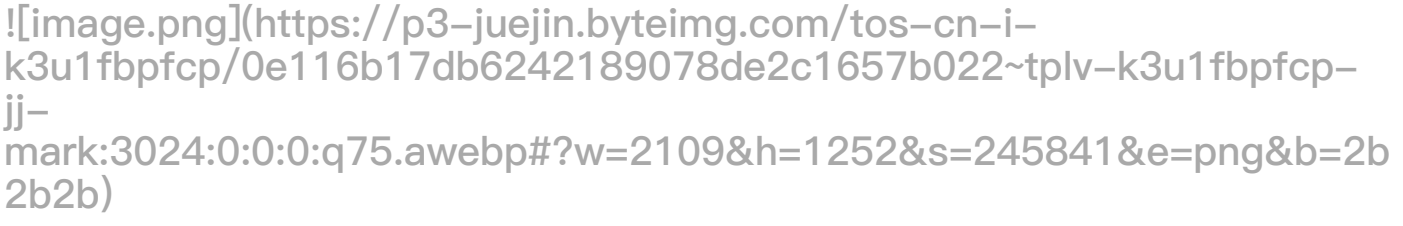
工具类：

以下方法均已在本地实验测试
创建一个多线程工具类，可以包含执行任务、处理返回值、异常操作以及多线程合并操作等功能。下面是一个示例，这个工具类使用 `ExecutorService` 来执行任务，并使用 `CompletableFuture` 来处理异步操作和合并结果。

```
package com.geekplus.leto.wes.stocktake.flowscript.listener.box;

import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collectors;

/**
 * @Author derek_smart
 * @Date 202/4/24 11:05
 * @Description 多线程工具类
 */
public class MultiThreadedUtility {

    private final ExecutorService executorService;

    public MultiThreadedUtility(int numberOfThreads) {
        executorService =
Executors.newFixedThreadPool(numberOfThreads);
    }

    // 执行一个任务，返回一个包含结果的 CompletableFuture
    public <T> CompletableFuture<T> executeTask(Callable<T> task) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                return task.call();
```

```
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }, executorService);
    }

    // 执行多个任务，返回一个包含所有结果的 CompletableFuture 列表
    public <T> List<CompletableFuture<T>>
executeTasks(List<Callable<T>> tasks) {
        return tasks.stream()
                .map(this::executeTask)
                .collect(Collectors.toList());
    }

    // 合并多个 CompletableFuture 的结果
    public <T> CompletableFuture<List<T>>
combineFutures(List<CompletableFuture<T>> futures) {
        CompletableFuture<Void> allDoneFuture =
                CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0]));
        return allDoneFuture.thenApply(v ->
                futures.stream()
                        .map(CompletableFuture::join) // join 会等待每个 future
完成，并返回结果
                        .collect(Collectors.toList())
        );
    }


    // 异步执行任务并返回 CompletableFuture
    public <T> CompletableFuture<T> supplyAsync(Supplier<T> supplier) {
        return CompletableFuture.supplyAsync(supplier, executorService);
    }

    // 异步执行任务并在完成后应用函数
    public <T, U> CompletableFuture<U> supplyAsync(Supplier<T>
supplier, Function<T, U> function) {
        return CompletableFuture.supplyAsync(supplier, executorService)
                .thenApplyAsync(function, executorService);
    }

    // 关闭线程池
    public void shutdown() {
        executorService.shutdown();
    }

    // 处理 CompletableFuture 的异常
    public <T> CompletableFuture<T>
```

```
exceptionally(CompletableFuture<T> future, Function<Throwable, T>
exceptionHandler) {
        return future.exceptionally(exceptionHandler);
    }


    // 创建一个异常处理后的 CompletableFuture
    public static <T> CompletableFuture<T> handleExceptions(
            Supplier<CompletableFuture<T>> taskSupplier,
            Function<Throwable, T> exceptionHandler) {
        try {
            return taskSupplier.get().exceptionally(exceptionHandler);
        } catch (Exception ex) {
            CompletableFuture<T> failedFuture = new
CompletableFuture<>();
            failedFuture.completeExceptionally(ex);
            return failedFuture;
        }
    }

    // 重试逻辑
    public static <T> CompletableFuture<T>
retry(Supplier<CompletableFuture<T>> taskSupplier, int maxRetries) {
        return taskSupplier.get().handle((result, ex) -> {
            if (ex == null) {
                return CompletableFuture.completedFuture(result);
            } else if (maxRetries > 0) {
                // 递归调用 retry 方法以进行重试
                return retry(taskSupplier, maxRetries - 1);
            } else {
                // 如果重试次数用完，重新抛出异常
                CompletableFuture<T> failedFuture = new
CompletableFuture<>();
                failedFuture.completeExceptionally(ex);
                return failedFuture;
            }
        }).thenCompose(Function.identity());
    }
}

```



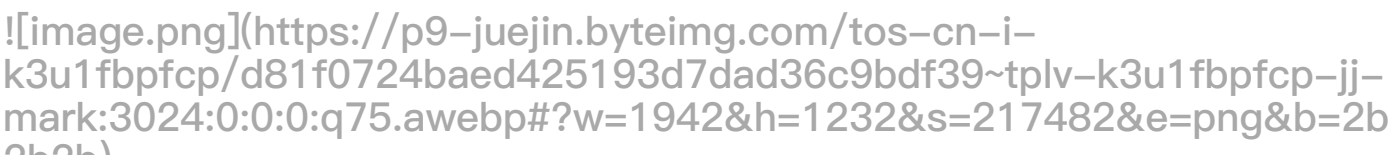
测试类：

使用这个工具类的示例：

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

/**
 * @Author derek_smart
 * @Date 202/4/24 11:05
 * @Description 多线程测试类
 */
public class ExampleUsage {

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        MultiThreadedUtility utility = new MultiThreadedUtility(4);

        // 创建任务
        List<Callable<String>> tasks = Arrays.asList(
                () -> "Task 1 result",
                () -> "Task 2 result",
                () -> "Task 3 result"
        );

        // 执行任务并获取结果的 CompletableFuture 列表
        List<CompletableFuture<String>> futures =
utility.executeTasks(tasks);

        // 合并所有 future 的结果
        CompletableFuture<List<String>> allResultsFuture =
utility.combineFutures(futures);

        // 当所有任务完成时，处理合并后的结果
        allResultsFuture.thenAccept(results -> {
            results.forEach(System.out::println);
            utility.shutdown(); // 所有任务完成后关闭线程池
        }).exceptionally(ex -> {
            // 处理异常
```

```
            System.out.println("Error occurred: " + ex.getMessage());
            utility.shutdown();
            return null;
        });

        // 异步执行任务并获取结果
        CompletableFuture<String> future = utility.supplyAsync(() -> {
            // 模拟长时间运行的任务
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Result of the asynchronous computation";
        });

        // 注册完成事件
        future.thenAccept(result -> System.out.println("Async task
completed: " + result));

        // 等待结果完成并处理
        String result = future.get(); // 阻塞直到任务完成
        System.out.println("Result: " + result);

        // 异步执行任务并在完成后应用函数
        CompletableFuture<Integer> futureWithFunction =
utility.supplyAsync(() -> "123", Integer::parseInt);

        // 注册完成事件
        futureWithFunction.thenAccept(parsedResult ->
System.out.println("Parsed result: " + parsedResult));

        // 等待结果完成并处理
        Integer parsedResult = futureWithFunction.get(); // 阻塞直到任务完
成
        System.out.println("Parsed Result: " + parsedResult);

        CompletableFuture<String> recoveredFuture3 =
utility.exceptionally(futures.get(0), ex -> "Recovered from error");

        // 等待 future3 完成，并打印结果或恢复的值
        System.out.println("Task 1 result: " + recoveredFuture3.get());

        // 关闭线程池
        utility.shutdown();


        // 异步任务可能会抛出异常
```

```
        CompletableFuture<String> futureWithException =
CompletableFuture.supplyAsync(() -> {
            throw new RuntimeException("Exception occurred!");
        });

        // 使用工具类处理异常
        CompletableFuture<String> handledFuture =
utility.handleExceptions(
                () -> futureWithException,
                ex -> "Recovered from: " + ex.getMessage()
        );

        // 获取处理后的结果
        System.out.println(handledFuture.get()); // 输出: Recovered from:
Exception occurred!

        // 重试逻辑的使用
        CompletableFuture<String> futureWithRetry = utility.retry(
                () -> CompletableFuture.supplyAsync(() -> {
                    double randomValue = Math.random();
                    System.out.println("Random value: " + randomValue);
                    if (randomValue < 0.5) {
                        throw new RuntimeException("Bad luck, try again!");
                    }
                    return "Success!";
                }),
                3 // 最大重试次数
        );
        // 获取重试后的结果
        System.out.println(futureWithRetry.get()); // 输出可能是 "Success!"
或者异常
    }
}

```



在这个示例中，我们创建了一个 `MultiThreadedUtility` 实例，定义了三个任务，并执行了它们。然后，我们使用 `combineFutures` 方法来合并所有的 `CompletableFuture` 对象，并在所有任务完成时打印结果。如果在执行过程中发生异常，我们使用 `exceptionally` 方法来处理它。

这个工具类提供了一种灵活的方式来处理多线程任务，包括执行任务、合并结果以及异常处理。通过使用 `CompletableFuture`，你可以轻松地实现复杂的异步逻辑，并且可以链式调用多个操作。记得在不再需要线程池时调用 `shutdown` 方法来释放资源。

原文链接: https://juejin.cn/post/7362743871980519439