# CompletableFuture

Java 8 有大量的新特性和增强如 [Lambda 表达式](https://www.callicoder.com/java-lambda-expression-tutorial/" \t "_blank)，[Streams](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \t "_blank)，[CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html" \t "_blank)等。

# 1、什么是CompletableFuture

在Java中**CompletableFuture用于异步编程，异步编程是编写非阻塞的代码，运行的任务在一个单独的线程**，与主线程隔离，并且会通知主线程它的进度，成功或者失败。

在这种方式中，主线程不会被阻塞，不需要一直等到子线程完成。主线程可以并行的执行其他任务。

使用这种并行方式，可以极大的提高程序的性能。

# 2、Future vs CompletableFuture

CompletableFuture 是 [Future API](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html" \t "_blank)的扩展。

Future 被用于作为一个异步计算结果的引用。提供一个 isDone() 方法来检查计算任务是否完成。当任务完成时，get()方法用来接收计算任务的结果。

Future API 是非常好的 Java 异步编程进阶，但是它缺乏一些非常重要和有用的特性。

# Future 的局限性

* 1. **不能手动完成** 当你写了一个函数，用于通过一个远程API获取一个电子商务产品最新价格。因为这个 API 太耗时，你把它允许在一个独立的线程中，并且从你的函数中返回一个 Future。 现在假设这个API服务宕机了，这时你想通过该产品的最新缓存价格手工完成这个Future 。你会发现无法这样做。
  2. Future 的结果在非阻塞的情况下，**不能执行更进一步的操作** Future 不会通知你它已经完成了，它提供了一个阻塞的 get() 方法通知你结果。 你无法给 Future 植入一个回调函数，当 Future 结果可用的时候，用该回调函数自动的调用 Future 的结果。
  3. **多个 Future 不能串联在一起组成链式调用** 有时候你需要执行一个长时间运行的计算任务，并且当计算任务完成的时候，你需要把它的计算结果发送给另外一个长时间运行的计算任务等等。 你会发现你无法使用 Future 创建这样的一个工作流。
  4. **不能组合多个 Future 的结果** 假设你有10个不同的Future，你想并行的运行，然后在它们运行未完成后运行一些函数。你会发现你也无法使用 Future 这样做。
  5. **没有异常处理** Future API 没有任务的异常处理结构

居然有如此多的限制，幸好我们有CompletableFuture，你可以使用 CompletableFuture 达到以上所有目的。

CompletableFuture 实现了 Future 和 CompletionStage接口，并且提供了许多关于创建，链式调用和组合多个 Future 的便利方法集，而且有广泛的异常处理支持。

# 4、基本属性介绍

## 4.1、runAsync 和 supplyAsync方法

CompletableFuture 提供了四个静态方法来创建一个异步操作。

public static CompletableFuture<Void>runAsync(Runnable runnable);  
public static CompletableFuture<Void>runAsync(Runnable runnable,Executor executor);  
public static<U> CompletableFuture<U> supplyAsync(Supplier<U> supplier);  
public static<U>CompletableFuture<U>supplyAsync(Supplier<U> supplier, Executor executor);

没有指定Executor的方法会**使用ForkJoinPool.commonPool()** 作为它的线程池执行异步代码。如果指定线程池，则使用指定的线程池运行。以下所有的方法都类同。

**runAsync方法不支持返回值。**

**supplyAsync可以支持返回值。**

示例

//无返回值  
public static voidrunAsync*()*throwsException *{* CompletableFuture*<*Void*>* future = CompletableFuture.*runAsync(()* -> *{* try *{* TimeUnit.*SECONDS*.sleep*(*1*)*;  
 *}* catch *(*InterruptedException e*) {  
  
 }* System.*out*.println*(*"run end ..."*)*;  
 *})*;  
 future.get*()*;  
*}*//有返回值  
public static void supplyAsync*()*throwsException *{* CompletableFuture*<*Long*>* future = CompletableFuture.*supplyAsync(()* -> *{* try *{* TimeUnit.*SECONDS*.sleep*(*1*)*;  
 *}* catch *(*InterruptedException e*) {  
 }* System.*out*.println*(*"run end ..."*)*;  
 return System.currentTimeMillis*()*;  
 }*)*;  
 long time = future.get*()*;  
 System.*out*.println*(*"time = " + time*)*;  
*}*

## 4.2、计算结果完成时的回调方法

当CompletableFuture的计算结果完成，或者抛出异常的时候，可以执行特定的Action。主要是下面的方法：

public CompletableFuture<T> whenComplete(BiConsumer<? superT, ? superThrowable>action);  
public CompletableFuture<T> whenCompleteAsync(BiConsumer<? superT, ? superThrowable>action);  
public CompletableFuture<T> whenCompleteAsync(BiConsumer<? superT, ? superThrowable>action, Executor executor);  
public CompletableFuture<T> exceptionally(Function<Throwable, ? extendsT>fn);

可以看到Action的类型是BiConsumer它可以处理正常的计算结果，或者异常情况。

**whenComplete 和 whenCompleteAsync 的区别：**

whenComplete：是执行当前任务的线程继续执行 whenComplete 的任务。

whenCompleteAsync：是执行把 whenCompleteAsync 这个任务继续提交给线程池来进行执行。

**示例**

public static void whenComplete*()* throws Exception *{* CompletableFuture*<*Void*>* future = CompletableFuture.*runAsync(()* -> *{* try *{* TimeUnit.*SECONDS*.sleep*(*1*)*;  
 *}* catch *(*InterruptedException e*) {  
 }* if*(*new Random*()*.nextInt*()*%2>=0*) {* int i = 12/0;  
 *}* System.*out*.println*(*"run end ..."*)*;  
 *})*;  
  
 future.whenComplete*(*new BiConsumer*<*Void, Throwable*>() {* @Override  
 public void accept*(*Void t, Throwable action*) {* System.*out*.println*(*"执行完成！"*)*;  
 *}  
 })*;  
 future.exceptionally*(*new Function*<*Throwable, Void*>() {* @Override  
 public Void apply*(*Throwable t*) {* System.*out*.println*(*"执行失败！"+t.getMessage*())*;  
 return null;  
 *}  
 })*;  
 TimeUnit.*SECONDS*.sleep*(*2*)*;  
*}*

## 4.3、thenApply 方法

当**一个线程依赖另一个线程时**，可以使用 thenApply 方法来把这**两个线程串行化**。

public<U>CompletableFuture<U>thenApply(Function<?super T,?extends U> fn);  
public<U>CompletableFuture<U>thenApplyAsync(Function<?super T,?extends U> fn);  
public<U>CompletableFuture<U>thenApplyAsync(Function<?super T,?extends U> fn,Executor executor);

Function

T：上一个任务返回结果的类型

U：当前任务的返回值类型

**示例**

private static void thenApply*()* throws Exception *{* CompletableFuture*<*Long*>* future = CompletableFuture.*supplyAsync(*new Supplier*<*Long*>() {* @Override  
 public Long get*() {* long result = new Random*()*.nextInt*(*100*)*;  
 System.*out*.println*(*"result1=" + result*)*;  
 return result;  
 *}  
 })*.thenApply*(*new Function*<*Long, Long*>() {* @Override  
 public Long apply*(*Long t*) {* long result = t \* 5;  
 System.*out*.println*(*"result2=" + result*)*;  
 return result;  
 *}  
 })*;  
  
 long result = future.get*()*;  
 System.*out*.println*(*result*)*;  
*}*

第二个任务依赖第一个任务的结果。

## 4.4、handle 方法

handle 是执行任务完成时**对结果的处理**。

handle 方法和 thenApply 方法处理方式基本一样。不同的是 **handle 是在任务完成后再执行，还可以处理异常的任务**。**thenApply 只可以执行正常的任务，任务出现异常则不执行 thenApply 方法**。

public <U> CompletionStage<U> handle(BiFunction<? super T, Throwable, ? extends U>fn);  
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U> fn);  
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ? extends U>fn, Executor executor);

**示例**

public static void handle*()*throwsException *{* CompletableFuture*<*Integer*>* future = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int i = 10 / 0;  
 return new Random*()*.nextInt*(*10*)*;  
 *}  
 })*.handle*(*new BiFunction*<*Integer, Throwable, Integer*>() {* @Override  
 public Integer apply*(*Integer param, Throwable throwable*) {* int result = -1;  
 if *(*throwable == null*) {* result = param \* 2;  
 *}* else *{* System.*out*.println*(*throwable.getMessage*())*;  
 *}* return result;  
 *}  
 })*;  
 System.*out*.println*(*future.get*())*;  
*}*

从示例中可以看出，在 handle 中可以根据任务是否有异常来进行做相应的后续处理操作。而 thenApply 方法，如果上个任务出现错误，则不会执行 thenApply 方法。

## 4.5、thenAccept 消费处理结果

**接收任务的处理结果，并消费处理，无返回结果**。

public CompletionStage<Void> thenAccept(Consumer<? super T>action);  
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T>action);  
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T>action, Executor executor);

**示例**

public static void thenAccept*()*throwsException *{* CompletableFuture*<*Void*>* future = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* return new Random*()*.nextInt*(*10*)*;  
 *}  
 })*.thenAccept*(*integer -> *{* System.*out*.println*(*integer*)*;  
 *})*;  
 future.get*()*;  
*}*

从示例代码中可以看出，该方法只是消费执行完成的任务，并可以根据上面的任务返回的结果进行处理。并没有后续的输错操作。

## 4.6、thenRun 方法

跟 thenAccept 方法不一样的是，**不关心任务的处理结果**。只要上面的任务执行完成，就开始执行 thenAccept 。

public CompletionStage<Void> thenRun(Runnable action);  
public CompletionStage<Void> thenRunAsync(Runnable action);  
public CompletionStage<Void> thenRunAsync(Runnable action,Executor executor);

**示例**

public static void thenRun*()* throws Exception*{* CompletableFuture*<*Void*>* future = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* return new Random*()*.nextInt*(*10*)*;  
 *}  
 })*.thenRun*(()* -> *{* System.*out*.println*(*"thenRun ..."*)*;  
 *})*;  
 future.get*()*;  
*}*

该方法同 thenAccept 方法类似。不同的是上个任务处理完成后，并不会把计算的结果传给 thenRun 方法。只是处理玩任务后，执行 thenAccept 的后续操作。

## 4.7、thenCombine 合并任务

thenCombine 会把 两个 CompletionStage 的任务都执行完成后，把两个任务的结果一块交给 thenCombine 来处理。

public <U,V> CompletionStage<V> thenCombine(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn);  
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn);  
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends U> other,BiFunction<? super T,? super U,? extends V> fn,Executor executor);

**示例**

private static void thenCombine*()* throws Exception *{* CompletableFuture*<*String*>* future1 = CompletableFuture.*supplyAsync(*new Supplier*<*String*>() {* @Override  
 public String get*() {* return "hello";  
 *}  
 })*;  
 CompletableFuture*<*String*>* future2 = CompletableFuture.*supplyAsync(*new Supplier*<*String*>() {* @Override  
 public String get*() {* return "hello";  
 *}  
 })*;  
 CompletableFuture*<*String*>* result = future1.thenCombine*(*future2, new BiFunction*<*String, String, String*>() {* @Override  
 public String apply*(*String t, String u*) {* return t+" "+u;  
 *}  
 })*;  
 System.*out*.println*(*result.get*())*;  
*}*

## 4.8、thenAcceptBoth

当两个CompletionStage都执行完成后，把结果一块交给thenAcceptBoth来进行消耗

public <U> CompletionStage<Void> thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super T, ? super U> action);  
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T, ? super U> action);  
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T, ? super U> action, Executor executor);

**示例**

private static void thenAcceptBoth*()* throws Exception *{* CompletableFuture*<*Integer*>* f1 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f1=" + t*)*;  
 return t;  
 *}  
 })*;  
  
 CompletableFuture*<*Integer*>* f2 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f2=" + t*)*;  
 return t;  
 *}  
 })*;  
 f1.thenAcceptBoth*(*f2, new BiConsumer*<*Integer, Integer*>() {* @Override  
 public void accept*(*Integer t, Integer u*) {* System.*out*.println*(*"f1=" + t + ";f2=" + u + ";"*)*;  
 *}  
 })*;  
*}*

## 4.9、applyToEither 方法

两个CompletionStage，**谁执行返回的结果快，就用那个**CompletionStage的结果进行下一步的转化操作。

public <U> CompletionStage<U> applyToEither(CompletionStage<? extends T> other,Function<? super T, U> fn);  
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends T> other,Function<? super T, U> fn);  
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends T> other,Function<? super T, U> fn,Executor executor);

**示例**

private static void applyToEither*()* throws Exception *{* CompletableFuture*<*Integer*>* f1 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f1=" + t*)*;  
 return t;  
 *}  
 })*;  
 CompletableFuture*<*Integer*>* f2 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f2=" + t*)*;  
 return t;  
 *}  
 })*;  
  
 CompletableFuture*<*Integer*>* result = f1.applyToEither*(*f2, new Function*<*Integer, Integer*>() {* @Override  
 public Integer apply*(*Integer t*) {* System.*out*.println*(*t*)*;  
 return t \* 2;  
 *}  
 })*;  
 System.*out*.println*(*result.get*())*;  
*}*

## 4.10、acceptEither 方法

两个CompletionStage，**谁执行返回的结果快，就用那个**CompletionStage的结果进行下一步的消耗操作。

public CompletionStage<Void> acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action);  
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action);  
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action, Executor executor);

**示例**

private static void acceptEither*()* throws Exception *{* CompletableFuture*<*Integer*>* f1 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f1=" + t*)*;  
 return t;  
 *}  
 })*;  
  
 CompletableFuture*<*Integer*>* f2 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f2=" + t*)*;  
 return t;  
 *}  
 })*;  
 f1.acceptEither*(*f2, new Consumer*<*Integer*>() {* @Override  
 public void accept*(*Integer t*) {* System.*out*.println*(*t*)*;  
 *}  
 })*;  
*}*

## 4.11、runAfterEither 方法

两个CompletionStage，**任何一个完成了都会执行下一步的操作**（Runnable）

public CompletionStage<Void> runAfterEither(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other,Runnable action,Executor executor);

**示例**

private static void runAfterEither*()* throws Exception *{* CompletableFuture*<*Integer*>* f1 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f1="+t*)*;  
 return t;  
 *}  
 })*;  
  
 CompletableFuture*<*Integer*>* f2 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f2="+t*)*;  
 return t;  
 *}  
 })*;  
 f1.runAfterEither*(*f2, new Runnable*() {* @Override  
 public void run*() {* System.*out*.println*(*"上面有一个已经完成了。"*)*;  
 *}  
 })*;  
*}*

## 4.12、runAfterBoth

两个CompletionStage，**都完成**了计算才会执行下一步的操作（Runnable）

public CompletionStage<Void> runAfterBoth(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other,Runnable action);  
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other,Runnable action,Executor executor);

**示例**

private static void runAfterBoth*()* throws Exception *{* CompletableFuture*<*Integer*>* f1 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f1=" + t*)*;  
 return t;  
 *}  
 })*;  
  
 CompletableFuture*<*Integer*>* f2 = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 try *{* TimeUnit.*SECONDS*.sleep*(*t*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* System.*out*.println*(*"f2=" + t*)*;  
 return t;  
 *}  
 })*;  
 f1.runAfterBoth*(*f2, new Runnable*() {* @Override  
 public void run*() {* System.*out*.println*(*"上面两个任务都执行完成了。"*)*;  
 *}  
 })*;  
*}*

## 4.13、thenCompose 方法

thenCompose 方法允许你对两个 CompletionStage 进行**流水线操作**，第一个操作完成时，将其结果作为参数传递给第二个操作。

public <U> CompletableFuture<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn);  
public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn) ;  
public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn, Executor executor) ;

**示例**

private static void thenCompose*()* throws Exception *{* CompletableFuture*<*Integer*>* f = CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = new Random*()*.nextInt*(*3*)*;  
 System.*out*.println*(*"t1=" + t*)*;  
 return t;  
 *}  
 })*.thenCompose*(*new Function*<*Integer, CompletionStage*<*Integer*>>() {* @Override  
 public CompletionStage*<*Integer*>* apply*(*Integer param*) {* return CompletableFuture.*supplyAsync(*new Supplier*<*Integer*>() {* @Override  
 public Integer get*() {* int t = param \* 2;  
 System.*out*.println*(*"t2=" + t*)*;  
 return t;  
 *}  
 })*;  
 *}  
  
 })*;  
 System.*out*.println*(*"thenCompose result : " + f.get*())*;  
*}*

# 使用callback基于特定任务的结果进行异步处理

程序中经常需要主线程创建新的线程来处理某一任务，然后基于任务的完成结果(返回值或者exception)来执行特定逻辑, 对于这种场景我们可以很方面的使用whenComplete或者whenCompleteAsync来注册callback方法

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

// long running task

return "task result";

});

future.whenComplete((result, exception) -> {

if (null == exception) {

System.out.println("result from previous task: " + result);

}

});

对于任务执行中抛错的情况:

CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {

// long running task

throw new RuntimeException("error!");

});

future.whenComplete((result, exception) -> {

if (null == exception) {

System.out.println("result from previous task: " + result);

} else {

System.err.println("Exception thrown from previous task: " + exception.getMessage());

}

});

也可以用exceptionally来显示的处理错误:

CompletableFuture.supplyAsync(() -> {

throw new IllegalArgumentException("error");

}).exceptionally(ex -> {

System.out.println("Exception caught: " + ex.getMessage());

return ex.getMessage();

}).thenAccept(result -> {

System.out.println("result: " + result);

});

如果不需关心任务执行中是否有exception,则可以使用thenAccept方法, 需要注意的是如果执行中抛了exception, 则thenAccept里面的回调方法不会被执行

CompletableFuture.supplyAsync(() -> {

// long running task

return "task result";

}).thenAccept((result) -> {

System.out.println("result from previous task: " + result);

});

# 6、多节点并行计算

常见的场景是将一个大的任务切分为数个子任务，并行处理所有子任务，当所有子任务都成功结束时再继续处理后面的逻辑。以前的做法是利用CountDownLatch, 主线程构造countDownLatch对象，latch的大小为子任务的总数，每一个任务持有countDownLatch的引用，任务完成时对latch减1，主线程阻塞在countDownLatch.await方法上，当所有子任务都成功执行完后，latch=0, 主线程继续执行。

方式一、countDownLatch 方法

int size = 5;

CountDownLatch latch = new CountDownLatch(size);

for (int i = 0; i < size; i++) {

Executors.newFixedThreadPool(size).submit(() -> {

try {

// long running task

System.out.println(Thread.currentThread().getName() + " " + latch.getCount());

} finally {

latch.countDown();

}

});

}

try {

latch.await();

} catch (InterruptedException e) {

e.printStackTrace();

}

// continue...

System.out.println(Thread.currentThread().getName());

这样的代码繁琐且很容易出错，我们可以用CompletableFuture.allOf来方便的处理上述场景。直接贴例子, 根据一组账户ID并行查找对应账户：

方式二、CompletableFuture

@Override  
public CompletableFuture*<*String*>* findAccount*(*String accountId*) {* return CompletableFuture.*supplyAsync(()* -> *{* try *{* TimeUnit.*SECONDS*.sleep*(*5*)*;  
 *}* catch *(*InterruptedException e*) {* e.printStackTrace*()*;  
 *}* return accountId;  
 *})*.exceptionally*(*ex -> *{* //发生异常时执行  
 System.*out*.println*(*"发生异常"*)*;  
 return "exception";  
 *})*;  
*}*

@Override  
public Object parallelComputing*(*List*<*NodeBean*>* list*) {* if *(*list == null || list.size*()* == 0*) {* throw new RuntimeException*(*"list is empty"*)*;  
 *}* long start = System.*currentTimeMillis()*;  
 //获取并行处理的数据集  
 List*<*CompletableFuture*<*String*>>* futureList = list.stream*()*.map*(*nodeBean -> findAccount*(*nodeBean.getIp*()))*.collect*(*Collectors.*toList())*;  
 // 使用allOf方法来表示所有的并行任务  
 CompletableFuture*<*Void*>* allFutures = CompletableFuture.*allOf(*futureList.toArray*(*new CompletableFuture*[*futureList.size*()]))*;  
 // 下面的方法可以帮助我们获得所有子任务的处理结果  
 CompletableFuture*<*List*<*String*>>* finalResults = allFutures.thenApply*(*v -> *{* System.*out*.println*(*"执行完成"*)*;  
 long end = System.*currentTimeMillis()*;  
 System.*out*.println*(*"耗时={}" + *(*end - start*))*;  
 return futureList.stream*()*.map*(*accountFindingFuture -> accountFindingFuture.join*())*.collect*(*Collectors.*toList())*;  
 *})*;  
 return finalResults;  
*}*

如果后续逻辑没有必要等待所有子任务全部结束，而是只要任一一个任务成功结束就可以继续执行，我们可以使用CompletableFuture.anyOf方法：

CompletableFuture<Object> anyOfFutures = CompletableFuture.anyOf(taskFutureA, taskFutureB, taskFutureC);

假设三个任务中taskFutureA最先执行完毕并成功返回，则anyOfFutures里得到的是taskFutureA的执行结果.

案例：

@Slf4j

@Service

public class OrderService {

public CompletableFuture<String> todayOrderCount() {

return CompletableFuture.supplyAsync(() -> this.getTodayOrderCount());

}

public CompletableFuture<String> todayTurnover() {

return CompletableFuture.supplyAsync(() -> this.getTodayTurnover());

}

public CompletableFuture<String> totalTurnover() {

return CompletableFuture.supplyAsync(() -> this.getTotalTurnover());

}

private String getTodayOrderCount() {

System.out.println(">>>>>>> 查询今日订单数:" + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

return "50";

}

private String getTodayTurnover() {

System.out.println(">>>>>>> 查询今日交易额:" + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

return "200";

}

private String getTotalTurnover() {

System.out.println(">>>>>>> 查询总交易额:" + Thread.currentThread().getName());

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

return "800";

}

}

@Slf4j

@RestController

@RequestMapping("/order")

public class OrderController {

@Autowired

private OrderService orderService;

@GetMapping("/report")

public JSONObject report() {

long start = System.currentTimeMillis();

JSONObject json = orderReport();

System.out.println("耗时:" + (System.currentTimeMillis() - start));

return json;

}

private JSONObject orderReport() {

CompletableFuture<String> todayOrderCountFuture = orderService.todayOrderCount();

CompletableFuture<String> todayTurnoverFuture = orderService.todayTurnover();

CompletableFuture<String> totalTurnoverFuture = orderService.totalTurnover();

JSONObject json = new JSONObject();

todayOrderCountFuture.whenComplete((v, t) -> {

json.put("todayOrderCountFuture", v);

});

todayTurnoverFuture.whenComplete((v, t) -> {

json.put("todayTurnoverFuture", v);

});

totalTurnoverFuture.whenComplete((v, t) -> {

json.put("totalTurnoverFuture", v);

});

CompletableFuture.allOf(todayOrderCountFuture, todayTurnoverFuture, totalTurnoverFuture)

.thenRun(() -> System.out.println("完成！！！！"))

.join();

return json;

}

}

# 7、任务的链式处理

在应用中经常会遇到任务的pipeline处理，任务A执行完后触发任务B，任务B执行完后触发任务C，上一个任务的结果是下一个任务的输入，对于这种场景，我们可以使用thenApply方法。

CompletableFuture.supplyAsync(() -> {

// long running task

return "task1";

}).thenApply(previousResult -> {

return previousResult + " task2";

}).thenApply(previousResult -> {

return previousResult + " task3";

}).thenAccept(previousResult -> {

System.out.println(previousResult);

});

output: task1 task2 task3

某一应用需要先根据accountId从数据库找到对应的账号信息，然后对该账号执行特定的处理逻辑:

CompletableFuture<Account> getAccount(String accountId) {

return CompletableFuture.supplyAsync(() -> {

return accountService.findAccount(accountId);

});

}

CompletableFuture<String> processAccount(Account account) {

return CompletableFuture.supplyAsync(() -> {

return accountService.updateAccount(account);

});

}

如果使用thenApply方法，其返回的结果是一个嵌套的CompletableFuture对象:

CompletableFuture<CompletableFuture<String>> res = getAccount("123").thenApply(account -> {

return processAccount(account);

});

如果不希望结果是嵌套的CompletableFuture，我们可以使用thenCompose方法来替代thenApply

CompletableFuture<String> res = getAccount("123").thenCompose(account -> {

return processAccount(account);

});