**CompletableFuture**

**Его нельзя загрузить используя CompletionService!!!**

Единственный его плюс это то, что его можно закрыть по timeOut-у когда ждешь инфу с сервера. А так лучше вместо него написать доп.метод (в котором и сделать связку всех нужных методов и действий с результатами).

**Что такое CompletableFuture?**  
CompletableFuture используется для асинхронного программирования в Java. Асинхронное программирование — это средство написания *неблокирующего* кода путём выполнения задачи в отдельном, отличном от главного, потоке, а также уведомление главного потока о ходе выполнения, завершении или сбое.

Причем на каждом шаге может использоваться пул из необходимого числа потоков созданных на основе ExecutorService.  
  
Таким образом, основной поток не блокируется и не ждёт завершения задачи, а значит может параллельно выполнять и другие задания.  
  
Наличие такого рода параллелизма значительно повышает производительность программ.  
  
  
**Future vs CompletableFuture**  
CompletableFuture это расширение [Future API](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html), представленного в Java 5.  
  
Future используется как ссылка на результат асинхронной задачи. В нём есть метод isDone() для проверки, завершилась ли задача или нет, а также метод get() для получения результата после его завершения.  
  
Future API был хорошим шагом на пути к асинхронному программированию, но ему не хватало некоторых важных и полезных функций.  
  
  
**Ограничения Future**  
**1. Его нельзя завершить вручную.**  
  
Допустим, вы написали функцию получения актуальной цены продукта из удалённого API. Поскольку этот вызов API занимает много времени, вы запускаете его в отдельном потоке и возвращаете Future из функции.  
  
Теперь предположим, что удалённый сервис перестал работать и вы хотите завершить Future вручную, передав актуальную цену продукта из кэша.  
  
Сможете ли вы сделать это с Future? Нет!  
  
**2. Нельзя выполнять дальнейшие действия над результатом Future без блокирования.**  
  
Future не уведомляет о своём завершении. В нём есть метод get(), который **блокирует** поток до тех пор, пока результат не станет доступным.  
  
Также в Future нельзя повесить функцию-колбэк, чтобы она срабатывала автоматически, как только станет доступен результат.  
  
**3. Невозможно выполнить множество Future один за другим.**  
  
Бывают случаи, когда требуется выполнить длительную операцию и после её завершения передать результат другой длительной операции и так далее.  
  
Такой алгоритм асинхронной работы невозможен при использовании Future.  
  
**4. Невозможно объединить несколько Future.**  
  
Предположим, что у вас есть 10 различных задач во Future, которые вы хотите запустить параллельно, и как только все они завершатся, вызвать некоторую функцию. С Future вы не можете сделать и это.  
  
**5. Нет обработки исключений.**  
  
Future API не имеет механизма обработки исключений.

Ого! Так много ограничений? Именно! Поэтому у нас и появился CompletableFuture. С его помощью можно достичь всего вышеперечисленного.  
  
CompletableFuture реализует интерфейсы Future и CompletionStage и предоставляет огромный набор удобных методов для создания и объединения нескольких Future. Он также имеет полноценную поддержку обработки исключений.  
  
  
**Создание CompletableFuture**  
**1. Простейший пример**  
Можно создать CompletableFuture, используя конструктор по умолчанию:

1. **CompletableFuture<String> completableFuture = new CompletableFuture<String>();**

Это самый простой CompletableFuture, который можно создать. Чтобы получить результат этого CompletableFuture, можно вызвать get():

1. **String result = completableFuture.get();**

Метод get() блокирует поток до тех пор, пока Future не завершится. Таким образом, этот вызов заблокирует поток навсегда, потому что Future никогда не завершается.  
  
Чтобы завершить CompletableFuture вручную, можно использовать метод complete():

1. **completableFuture.complete("Результат Future");**

Все клиенты, ожидающие этот Future, получат указанный результат, а последующие вызовы completableFuture.complete() будут игнорироваться.  
  
**2. Выполнение асинхронных задач с использованием runAsync()**  
Если вы хотите асинхронно выполнить некоторую фоновую задачу, которая не возвращает, результат, можно использовать метод CompletableFuture.runAsync(). Он принимает объект Runnable и возвращает CompletableFuture<Void>.

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Асинхронно запускаем задачу, заданную объектом Runnable***
2. **CompletableFuture<Void> future = CompletableFuture.runAsync(new Runnable() {**
3. **@Override**
4. **public void run()**
5. **{**
6. ***// Имитация длительной работы***
7. **try  {  TimeUnit.SECONDS.sleep(1); }**
8. **catch (InterruptedException e) { throw new IllegalStateException(e); }**
9. **System.out.println("Я буду работать в отдельном потоке, а не в**

**главном.");**

1. **}**

**});**

2. ***// Блокировка и ожидание завершения Future***
3. **future.get();**

Вы также можете передать объект Runnable в виде лямбда-выражения:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Использование лямбда-выражения***
2. **CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {**
3. ***// Имитация длительной работы***
4. **try {**
5. **TimeUnit.SECONDS.sleep(1);**
6. **} catch (InterruptedException e) {**
7. **throw new IllegalStateException(e);**
8. **}**
9. **System.out.println("Я буду работать в отдельном потоке, а не в главном.");**
10. **});**

В этой статье я буду часто использовать лямбда-выражения. Если вы всё ещё не используете их в своём коде, самое время начать это делать.

**3. Выполнение асинхронной задачи и возврат результата с использованием supplyAsync()**  
CompletableFuture.runAsync() полезен для задач, которые ничего не возвращают. Но что, если всё же нужно вернуть какой-нибудь результат из фоновой задачи?  
  
В таком случае вам придёт на помощь метод CompletableFuture.supplyAsync(). Он принимает Supplier<T> и возвращает CompletableFuture<T>, где T это тип возвращаемого функцией-поставщиком значения:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Запуск асинхронной задачи, заданной объектом Supplier***
2. **CompletableFuture<String> future = CompletableFuture.supplyAsync(new Supplier<String>() {**
3. **@Override**
4. **public String get() {**
5. **try {**
6. **TimeUnit.SECONDS.sleep(1);**
7. **} catch (InterruptedException e) {**
8. **throw new IllegalStateException(e);**
9. **}**
10. **return "Результат асинхронной задачи";**
11. **}**
12. **});**
14. ***// Блокировка и получение результата Future***
15. **String result = future.get();**
16. **System.out.println(result);**

[Supplier<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) это функциональный интерфейс, представляющий поставщика результатов. У него есть всего один метод get(), в котором можно указать фоновое задание и вернуть результат.  
  
Напомню, можно использовать лямбда-выражения, чтобы сократить код:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Использование лямбда-выражения***
2. **CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {**
3. **try {**
4. **TimeUnit.SECONDS.sleep(1);**
5. **} catch (InterruptedException e) {**
6. **throw new IllegalStateException(e);**
7. **}**
8. **return "Результат асинхронной задачи";**
9. **});**

**Заметка о пуле потоков и Executor**  
Вы можете поинтересоваться: хорошо, runAsync() и supplyAsync() выполняются в отдельном потоке, но мы ведь нигде не создавали новый поток, верно?  
  
Верно! CompletableFuture выполняет эти задачи в потоке, полученном из глобального [ForkJoinPool.commonPool()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html" \l "commonPool--).  
  
Также вы можете создать пул потоков и передать его методам runAsync() и supplyAsync(), чтобы они выполняли свои задачи в потоке, полученном уже из вашего пула потоков.  
  
Все методы CompletableFuture API представлены в двух вариантах: один принимает Executor в качестве аргумента, а второй нет.

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Вариации методов runAsync() и supplyAsync()***
2. **static CompletableFuture<Void>  runAsync(Runnable runnable)**
3. **static CompletableFuture<Void>  runAsync(Runnable runnable, Executor executor)**
4. **static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)**
5. **static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)**

Вот как можно создать пул потоков и передать его в один из этих методов:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **Executor executor = Executors.newFixedThreadPool(10);**
2. **CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {**
3. **try {**
4. **TimeUnit.SECONDS.sleep(1);**
5. **} catch (InterruptedException e) {**
6. **throw new IllegalStateException(e);**
7. **}**
8. **return "Результат асинхронной задачи";**
9. **}, executor);**

**Преобразование действий с CompletableFuture**  
Метод CompletableFuture.get() блокирующий. Он ждет, пока Future завершится и вернёт результат.  
  
Но это же не то, что нам нужно, верно? Для построения асинхронных систем мы должны иметь возможность повесить на CompletableFuture колбэк, который автоматически вызовется при завершении Future.  
  
Так что нам не потребуется ждать результат и внутри функции-колбэка мы сможем написать логику, которая отработает после завершения Future.  
  
Вы можете повесить колбэк на CompletableFuture, используя методы thenApply(), thenAccept() и thenRun().  
  
**1. thenApply()**  
Вы можете использовать метод thenApply() для обработки и преобразования результата CompletableFuture при его поступлении. В качестве аргумента он принимает Function<T, R>.  
[Function<T, R>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) это тоже функциональный интерфейс, представляющий функцию, которая принимает аргумент типа T и возвращает результат типа R:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Создаём CompletableFuture***
2. **CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {**
3. **try {**
4. **TimeUnit.SECONDS.sleep(1);**
5. **} catch (InterruptedException e) {**
6. **throw new IllegalStateException(e);**
7. **}**
8. **return "Rajeev";**
9. **});**
11. ***// Добавляем колбэк к Future, используя thenApply()***
12. **CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(name -> {**
13. **return "Привет," + name;**
14. **});**
16. ***// Блокировка и получение результата Future***
17. **System.out.println(greetingFuture.get()); *// Привет, Rajeev***

Вы также можете сделать **несколько последовательных преобразований**, используя серию вызов thenApply(). Результат одного thenApply() передаётся следующему:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture<String> welcomeText = CompletableFuture**
2. **.supplyAsync(() -> {**
3. **try {**
4. **TimeUnit.SECONDS.sleep(1);**
5. **} catch (InterruptedException e) {**
6. **throw new IllegalStateException(e);**
7. **}**
8. **return "Rajeev";**
9. **})**

**.thenApply(name -> {**

1. **return "Привет," + name;**
2. **})**

**.thenApply(greeting -> {**

1. **return greeting + ". Добро пожаловать в блог CalliCoder";**
2. **});**

**System.out.println(welcomeText.get());**

1. ***// Выводит: Привет, Rajeev. Добро пожаловать в блог CalliCoder***

**2. thenAccept() и thenRun()**  
Если вы не хотите возвращать результат, а хотите просто выполнить часть кода после завершения Future, можете воспользоваться методами thenAccept() и thenRun(). Эти методы являются потребителями и часто используются в качестве завершающего метода в цепочке.  
  
CompletableFuture.thenAccept() принимает [Consumer<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) и возвращает CompletableFuture<Void>. Он имеет доступ к результату CompletableFuture, к которому он прикреплён.

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Пример thenAccept()***
2. **CompletableFuture.supplyAsync(() -> {**
3. **return ProductService.getProductDetail(productId);**
4. **}).thenAccept(product -> {**
5. **System.out.println("Получена информация о продукте из удалённого сервиса " + product.getName())**
6. **});**

В отличие от thenAccept(), thenRun() не имеет доступа к результату Future. Он принимает Runnable и возвращает CompletableFuture<Void>:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Пример thenRun()***
2. **CompletableFuture.supplyAsync(() -> {**
3. ***// Выполняем некоторые расчёты***
4. **}).thenRun(() -> {**
5. ***// Расчёты завершены***
6. **});**

**Заметка об асинхронных колбэках**  
Все методы-колбэки в CompletableFuture имеют два асинхронных вида:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Виды thenApply()***
2. **<U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)**
3. **<U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn)**
4. **<U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)**

Эти асинхронные виды колбэков помогут распараллелить задачи, выполнив их в отдельном потоке.  
Например:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture.supplyAsync(() -> {**
2. **try {**
3. **TimeUnit.SECONDS.sleep(1);**
4. **} catch (InterruptedException e) {**
5. **throw new IllegalStateException(e);**
6. **}**
7. **return "Некоторый результат";**
8. **}).thenApply(result -> {**
9. ***/\****
10. ***Выполняется в том же потоке, где и задача supplyAsync()***
11. ***или в главном потоке, если задача supplyAsync() завершается сразу(чтобы проверить это удалите sleep())***
12. ***\*/***
13. **return "Обработанный результат";**
14. **});**

В приведенном выше примере задача thenApply() выполняется в том же потоке, где и задача supplyAsync(), либо в главном потоке, если задача supplyAsync() завершается достаточно быстро (попробуйте удалить вызов sleep() для проверки).  
  
Чтобы иметь больше контроля над потоком, выполняющим задачу, вы можете использовать асинхронные колбэки. Если вы используете thenApplyAsync(), он будет выполнен в другом потоке, полученном из ForkJoinPool.commonPool():

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture.supplyAsync(() -> {**
2. **return "Некоторый результат";**
3. **}).thenApplyAsync(result -> {**
4. ***// Выполняется в другом потоке, взятом из ForkJoinPool.commonPool()***
5. **return "Обработанный результат";**
6. **});**

Более того, если вы передадите Executor в thenApplyAsync(), задача будет выполнена в потоке, полученном из пула потоков Executor.

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **Executor executor = Executors.newFixedThreadPool(2);**
2. **CompletableFuture.supplyAsync(() -> {**
3. **return "Некоторый результат";**
4. **}).thenApplyAsync(result -> {**
5. ***// Выполняется в потоке, полученном от Executor***
6. **return "Обработанный результат"**
7. **}, executor);**

**Объединение двух CompletableFuture**  
**1. Комбинирование двух зависимых задач, с использованием thenCompose()**  
Предположим, что вы хотите получить информацию о пользователе из удалённого сервиса, и, как только информация будет доступна, получить кредитный рейтинг пользователя уже из другого сервиса.  
Вот реализации методов getUserDetail() и getCreditRating():

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture<User> getUsersDetail(String userId) {**
2. **return CompletableFuture.supplyAsync(() -> {**
3. **UserService.getUserDetails(userId);**
4. **});**
5. **}**
7. **CompletableFuture<Double> getCreditRating(User user) {**
8. **return CompletableFuture.supplyAsync(() -> {**
9. **CreditRatingService.getCreditRating(user);**
10. **});**
11. **}**

Теперь давайте посмотрим, что произойдет, если мы воспользуемся методом thenApply() для достижения желаемого результата:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture<CompletableFuture<Double>> result = getUserDetail(userId)**
2. **.thenApply(user -> getCreditRating(user));**

В предыдущих примерах Supplier, переданный в thenApply(), возвращал простое значение, но в этом случае он возвращает CompletableFuture. Следовательно, конечным результатом в приведенном выше примере является вложенный CompletableFuture.  
  
Чтобы избавиться от вложенного Future, используйте метод thenCompose():

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture<Double> result = getUserDetail(userId)**
2. **.thenCompose(user -> getCreditRating(user));**

***Правило таково: если функция-колбэк возвращает CompletableFuture, а вы хотите простой результат, (а в большинстве случаев именно он вам и нужен), тогда используйте thenCompose().***

**2. Комбинирование двух независимых задач, с использованием thenCombine()**  
Если thenCompose() используется для объединения двух задач, когда одна зависит от другой, то thenCombine() используется, когда вы хотите, чтобы две задачи работали независимо друг от друга и по завершению обоих выполнялось какое-нибудь действие.

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **System.out.println("Получение веса.");**
2. **CompletableFuture<Double> weightInKgFuture = CompletableFuture.supplyAsync(() -> {**
3. **try {**
4. **TimeUnit.SECONDS.sleep(1);**
5. **} catch (InterruptedException e) {**
6. **throw new IllegalStateException(e);**
7. **}**
8. **return 65.0;**
9. **});**
11. **System.out.println("Получение роста.");**
12. **CompletableFuture<Double> heightInCmFuture = CompletableFuture.supplyAsync(() -> {**
13. **try {**
14. **TimeUnit.SECONDS.sleep(1);**
15. **} catch (InterruptedException e) {**
16. **throw new IllegalStateException(e);**
17. **}**
18. **return 177.8;**
19. **});**
21. **System.out.println("Расчёт индекса массы тела.");**
22. **CompletableFuture<Double> combinedFuture = weightInKgFuture**
23. **.thenCombine(heightInCmFuture, (w, h) -> {**
24. **Double heightInMeter = h / 100;**
25. **return w/(heightInMeter \* heightInMeter);**
26. **});**
28. **System.out.println("Ваш индекс массы тела - " + combinedFuture.get());**

Колбэк, переданный методу thenCombine(), вызовется, когда обе задачи завершатся.  
  
  
**Объединение нескольких CompletableFuture *(используй вместо этой мути ListenableFuture из Guava)***  
Мы использовали thenCompose() и thenCombine(), чтобы объединить два CompletableFuture вместе. Но что, если вы хотите объединить произвольное количество CompletableFuture? Можно воспользоваться следующими методами:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)**
2. **static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)**

**1. CompletableFuture.allOf()**  
Когда нам нужно выполнить несколько *Futures*параллельно, мы обычно хотим дождаться их выполнения и затем обработать их объединенные результаты.

Статический метод *CompletableFuture.allOf*позволяет ожидать завершения всех *Futures*, предоставляемых как var-arg:

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> "Beautiful");

CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> "World");

CompletableFuture<Void> combinedFuture = CompletableFuture.allOf(future1, future2, future3);

//...

combinedFuture.get();

assertTrue(future1.isDone());

assertTrue(future2.isDone());

assertTrue(future3.isDone());

Обратите внимание, что тип возвращаемого значения *CompletableFuture.allOf()* является *CompletableFuture <Void>*. Ограничением этого метода является то, что он не возвращает объединенные результаты всех *Futures*. Вместо этого вы должны вручную получить результаты от *Futures*. К счастью, метод *CompletableFuture.join ()*и API Java 8 Streams делают это простым:

String combined = Stream.of(future1, future2, future3)

.map(CompletableFuture::join)

.collect(Collectors.joining(" "));

System.out.println(combined);

Метод *CompletableFuture.join ()*аналогичен методу *get*, но выдает непроверенное исключение в случае, если *Future*не завершается нормально. Это позволяет использовать его как ссылку на метод в методе *Stream.map ()*.

Более сложная задача:

Предположим, вы хотите загрузить содержимое 100 различных веб-страниц. Вы можете выполнить эту операцию последовательно, но это займет много времени. Поэтому вы написали функцию, которая получает ссылку на веб-страницу и возвращает CompletableFuture, то есть загружает контент страницы асинхронно:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture<String> downloadWebPage(String pageLink) {**
2. **return CompletableFuture.supplyAsync(() -> {**
3. ***// Код загрузки и возврата содержимого веб-страницы***
4. **});**
5. **}**

Теперь, когда все веб-страницы загрузились, вы хотите подсчитать количество страниц, содержащих ключевое слово 'CompletableFuture'. Воспользуемся для этого методом CompletableFuture.allOf():

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **List<String> webPageLinks = Arrays.asList(...) *// список из 100 ссылок***
3. ***// Асинхронно загружаем содержимое всех веб-страниц***
4. **List<CompletableFuture<String>> pageContentFutures = webPageLinks.stream()**
5. **.map(webPageLink -> downloadWebPage(webPageLink))**
6. **.collect(Collectors.toList());**
8. ***// Создаём комбинированный Future, используя allOf()***
9. **CompletableFuture<Void> allFutures = CompletableFuture.allOf(**
10. **pageContentFutures.toArray(new CompletableFuture[0])**
11. **);**

Проблема с CompletableFuture.allOf() заключается в том, что он возвращает CompletableFuture<Void>. Но мы можем получить результаты всех завершённых CompletableFuture, дописав несколько строк кода:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Когда все задачи завершены, вызываем future.join(), чтобы получить результаты и собрать их в список***
2. **CompletableFuture<List<String>> allPageContentsFuture = allFutures.thenApply(v -> {**
3. **return pageContentFutures.stream()**
4. **.map(pageContentFuture -> pageContentFuture.join())**
5. **.collect(Collectors.toList());**
6. **});**

Поскольку мы вызываем future.join(), когда все задачи уже завершены, блокировка нигде не происходит :-)  
  
Метод join() похож на get(). Единственное отличие заключается в том, что он бросает unchecked-исключение, если CompletableFuture завершается с ошибкой.  
  
Давайте теперь подсчитаем количество веб-страниц, содержащих наше ключевое слово:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. ***// Подсчитываем количество веб-страниц, содержащих ключевое слово "CompletableFuture"***
2. **CompletableFuture<Long> countFuture = allPageContentsFuture.thenApply(pageContents -> {**
3. **return pageContents.stream()**
4. **.filter(pageContent -> pageContent.contains("CompletableFuture"))**
5. **.count();**
6. **});**
8. **System.out.println("Количество веб-страниц с ключевым словом CompletableFuture - " +**
9. **countFuture.get());**

**2. CompletableFuture.anyOf()**  
CompletableFuture.anyOf(), как следует из названия, завершается сразу же, как только завершается **любой** из заданных CompletableFuture. Конечным результатом будет результат этого первого завершившегося CompletableFuture.  
Вот пример:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {**
2. **try {**
3. **TimeUnit.SECONDS.sleep(2);**
4. **} catch (InterruptedException e) {**
5. **throw new IllegalStateException(e);**
6. **}**
7. **return "Результат Future 1";**
8. **});**
10. **CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {**
11. **try {**
12. **TimeUnit.SECONDS.sleep(1);**
13. **} catch (InterruptedException e) {**
14. **throw new IllegalStateException(e);**
15. **}**
16. **return "Результат Future 2";**
17. **});**
19. **CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> {**
20. **try {**
21. **TimeUnit.SECONDS.sleep(3);**
22. **} catch (InterruptedException e) {**
23. **throw new IllegalStateException(e);**
24. **}**
25. **return "Результат Future 3";**
26. **});**
28. **CompletableFuture<Object> anyOfFuture = CompletableFuture.anyOf(future1, future2, future3);**
30. **System.out.println(anyOfFuture.get()); *// Результат Future 2***

В приведенном выше примере anyOfFuture завершается, когда завершается любой из трёх CompletableFuture. Поскольку в future2 задержка меньше, он завершится первым, значит, конечным результатом будет:  
*Результат Future 2*.  
  
CompletableFuture.anyOf() принимает переменное число аргументов Future и возвращает CompletableFuture<Object>. Проблема CompletableFuture.anyOf() в том, что если у вас есть задачи, которые возвращают результаты разных типов, то вы не будете знать тип вашего конечного CompletableFuture.  
  
  
**Обработка исключений CompletableFuture**  
Мы рассмотрели, как создать, преобразовать и объединить CompletableFuture. Теперь давайте разберёмся, что делать, если что-то пошло не так.  
  
Сперва рассмотрим, как ошибки распространяются в цепочке задач. Например:

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **CompletableFuture.supplyAsync(() -> {**
2. ***// Код, который может выбросить исключение***
3. **return "Некоторый результат";**
4. **}).thenApply(result -> {**
5. **return "Обработанный результат";**
6. **}).thenApply(result -> {**
7. **return "Результат дальнейшей обработки";**
8. **}).thenAccept(result -> {**
9. ***// Какие-то действия с окончательным результатом***
10. **});**

Если в исходной задаче supplyAsync() возникнет ошибка, тогда ни одна из последующих задач thenApply() не будет вызвана и Future завершится с исключением. Если ошибка возникнет в первом thenApply(), то все последующие задачи в цепочке не будут запущены и Future всё так же завершится с исключением.  
  
**1. Обработка исключений с использованием метода exceptionally()**Метод exceptionally() даёт возможность обойти возможные ошибки, если они есть. Можно залогировать исключение и вернуть значение по умолчанию.

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **Integer age = -1;**
3. **CompletableFuture<String> maturityFuture = CompletableFuture.supplyAsync(() -> {**
4. **if (age < 0) {**
5. **throw new IllegalArgumentException("Возраст не может быть отрицательным");**
6. **}**
7. **if (age > 18) {**
8. **return "Взрослый";**
9. **} else {**
10. **return "Ребёнок";**
11. **}**
12. **}).exceptionally(ex -> {**
13. **System.out.println("Ой! У нас тут исключение - " + ex.getMessage());**
14. **return "Неизвестно!";**
15. **});**
17. **System.out.println("Зрелость: " + maturityFuture.get());**

Обратите внимание, что ошибка не будет распространяться далее по цепочке, если вы её обработаете.  
  
**2. Обработка исключений с использованием метода handle()**  
Для восстановления после исключений API также предоставляет более общий метод handle(). Он вызывается независимо от того, возникло исключение или нет.

[[копировать]](javascript:void(0);) [[скачать]](javascript:void(0);)

1. **Integer age = -1;**
3. **CompletableFuture<String> maturityFuture = CompletableFuture**
4. **.supplyAsync(() -> {**
5. **if (age < 0) {**
6. **throw new IllegalArgumentException("Возраст не может быть отрицательным");**
7. **}**
8. **if (age > 18) {**
9. **return "Взрослый";**
10. **} else {**
11. **return "Ребёнок";**
12. **}**

**})**

**.handle((res, ex) -> { //где rez – результат, ex - исключение**

1. **if (ex != null) {**
2. **System.out.println("Ой! У нас тут исключение - " + ex.getMessage());**
3. **return "Неизвестно!";**
4. **}**
5. **return res;**

**});**



**System.out.println("Зрелость: " + maturityFuture.get());**

Если возникает исключение, аргумент res будет null, если не возникает, то ex будет null.

Настоящий пример с логгированием:

CompletableFuture<Integer> safa = future.handle((result, ex)->

{ if(result!=null) {return result}

else {log.warn(“Problem”, ex);

return -1}});  
  
**Примеры**

**1.**

Вопрос:

У меня есть 1000 больших файлов, которые нужно обработать в порядке, указанном ниже:

1. Сначала эти файлы необходимо параллельно скопировать в другой каталог. Для этого я планирую использовать ExecutorService с 10 потоками.

2. Как только какой-либо файл будет скопирован в другое место (№1), я отправлю этот файл для дальнейшей обработки в ExecutorService с 10 потоками.

3. И, наконец, над этими файлами необходимо параллельно выполнить еще одно действие, например, №2 получает входные данные от №1, №3 получает входные данные от №2.

Теперь я могу использовать здесь CompletionService, чтобы обрабатывать результаты потоков с №1 по №2 и с №2 по №3 в том порядке, в котором они завершаются. CompletableFuture говорит, что мы можем объединить асинхронные задачи, и это похоже на то, что я могу использовать в этом случае.

Я не уверен, следует ли мне реализовать свое решение с помощью CompletableFuture (поскольку оно относительно новое и должно быть лучше) или достаточно CompletionService? И почему в этом случае я должен выбирать одно, а не другое?

Ответ:

Вероятно, было бы лучше, если бы вы попробовали оба подхода, а затем выбрали тот, который вам более удобен. Хотя похоже, что CompletableFutures лучше подходят для этой задачи, потому что они действительно упрощают цепочку шагов/стадий обработки. Например, в вашем случае код может выглядеть так:

ExecutorService copyingExecutor = ...

// Непонятно из требований, но предположим, что у вас для этого есть отдельный исполнитель

ExecutorService processingExecutor = ...

public CompletableFuture<MyResult> process(Path file) {

return CompletableFuture

.supplyAsync(

() -> {

// Получить путь назначения, куда следует скопировать файл

Path destination = ...

try {

Files.copy(file, destination);

} catch (IOException e) {

throw new UncheckedIOException(e);

}

return destination;

},

copyingExecutor

)

.thenApplyAsync(

copiedFile -> {

// Process the copied file

...

},

processingExecutor

)

// Этот отдельный этап не имеет особого смысла, поэтому, если у вас нет еще одного исполнителя для этого или этого этапа, который применяется в другом месте вашего кода, его, вероятно, следует объединить с предыдущим этапом. .thenApply(

previousResult -> {

// Обработка предыдущего результата

...

}

);

}

**2.**

**join внутри Future нужно использовать очень аккуратно, вот пример дэдлока:**

import java.util.concurrent.\*;

public class Program

{

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

var pr = Executors.newSingleThreadExecutor();

var someCompletionStage = CompletableFuture.**supplyAsync**(()->{

try {

Thread.sleep(1000);

} catch (Exception ex) {}

return "hi!";

},

pr

)

.**thenApplyAsync**(s -> {

try { Thread.sleep(1000); } catch (Exception ex) {}

return s;

}, pr)**;**

CompletableFuture.**runAsync**(() -> {

String string = someCompletionStage.join();

System.out.println(string);

}, pr)

.join();

System.out.println("bye!");

pr.shutdown();

}

}

Происходит это потому, что "третий" CompletableFuture.runAsync успевает вклиниться между "первым" supplyAsync и "вторым" thenApplyAsync, блокирует поток джойном, поэтому вторая часть уже никогда не выполнится.

С цепочкой вызовов, естественно, дэдлок не случится.