## Future缺点：

Future虽然可以实现获取异步执行结果的需求，但是它没有提供通知的机制，我们无法得知Future什么时候完成。

要么使用阻塞，在future.get()的地方等待future返回的结果，这时又变成同步操作。要么使用isDone()轮询地判断Future是否完成，这样会耗费CPU的资源。

## CompletableFuture优势：

CompletableFuture能够将回调放到与任务不同的线程中执行，也能将回调作为继续执行的同步函数，在与任务相同的线程中执行。

它避免了传统回调最大的问题，那就是能够将控制流分离到不同的事件处理器中。

CompletableFuture弥补了Future模式的缺点。在异步的任务完成后，需要用其结果继续操作时，无需等待。

可以直接通过thenAccept、thenApply、thenCompose等方式将前面异步处理的结果交给另外一个异步事件处理线程来处理。

Runnable类型的参数会忽略计算的结果，Consumer是纯消费计算结果，BiConsumer会组合另外一个CompletionStage纯消费，

Function会对计算结果做转换，BiFunction会组合另外一个CompletionStage的计算结果做转换

## CompletableFuture常用方法：

### 1、CompletableFuture的静态工厂方法

runAsync(Runnable runnable)

使用ForkJoinPool.commonPool()作为它的线程池执行异步代码。

runAsync(Runnable runnable, Executor executor)

使用指定的thread pool执行异步代码。

supplyAsync(Supplier<U> supplier)

使用ForkJoinPool.commonPool()作为它的线程池执行异步代码，异步操作有返回值

supplyAsync(Supplier<U> supplier, Executor executor)

使用指定的thread pool执行异步代码，异步操作有返回值

### 2 主动完成计算

complete(T t) 完成异步执行，并返回future的结果

completeExceptionally(Throwable ex) 异步执行不正常的结束

### 3 转换

thenApply(Function<? super T,? extends U> fn)

接受一个Function<? super T,? extends U>参数用来转换CompletableFuture

thenApplyAsync(Function<? super T,? extends U> fn)

接受一个Function<? super T,? extends U>参数用来转换CompletableFuture，使用ForkJoinPool

thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)

接受一个Function<? super T,? extends U>参数用来转换CompletableFuture，使用指定的线程池

thenCompose(Function<? super T, ? extends CompletionStage<U>> fn)

在异步操作完成的时候对异步操作的结果进行一些操作，并且仍然返回CompletableFuture类型。

thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn)

在异步操作完成的时候对异步操作的结果进行一些操作，并且仍然返回CompletableFuture类型。使用ForkJoinPool。

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

在异步操作完成的时候对异步操作的结果进行一些操作，并且仍然返回CompletableFuture类型。使用指定的线程池。

### 4 组合

有返回值：

thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)

当两个CompletableFuture都正常完成后，执行提供的fn，用它来组合另外一个CompletableFuture的结果，即你需要将两个完全不相干的CompletableFuture对象的结果整合起来，而且你也不希望等到第一个任务完全结束才开始第二项任务。

thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)

当两个CompletableFuture都正常完成后，执行提供的fn，用它来组合另外一个CompletableFuture的结果。使用ForkJoinPool。

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

当两个CompletableFuture都正常完成后，执行提供的fn，用它来组合另外一个CompletableFuture的结果。使用指定的线程池。

纯消费：

thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)

当两个CompletableFuture都正常完成后，执行提供的action，用它来组合另外一个CompletableFuture的结果。

thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)

当两个CompletableFuture都正常完成后，执行提供的action，用它来组合另外一个CompletableFuture的结果。使用ForkJoinPool。

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

当两个CompletableFuture都正常完成后，执行提供的action，用它来组合另外一个CompletableFuture的结果。使用指定的线程池。

### 5 计算完成时

whenComplete(BiConsumer<? super T,? super Throwable> action)

当CompletableFuture完成计算结果时对结果进行处理，或者当CompletableFuture产生异常的时候对异常进行处理。

whenCompleteAsync(BiConsumer<? super T,? super Throwable> action)

当CompletableFuture完成计算结果时对结果进行处理，或者当CompletableFuture产生异常的时候对异常进行处理。使用ForkJoinPool。

whenCompleteAsync(BiConsumer<? super T,? super Throwable> action, Executor executor)

当CompletableFuture完成计算结果时对结果进行处理，或者当CompletableFuture产生异常的时候对异常进行处理。使用指定的线程池。

对象的值和原来的CompletableFuture计算的值不同。方法兼有whenComplete和转换的两个功能

handle(BiFunction<? super T, Throwable, ? extends U> fn)

当CompletableFuture完成计算结果或者抛出异常的时候，执行提供的fn

handleAsync(BiFunction<? super T, Throwable, ? extends U> fn)

当CompletableFuture完成计算结果或者抛出异常的时候，执行提供的fn，使用ForkJoinPool。

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

当CompletableFuture完成计算结果或者抛出异常的时候，执行提供的fn，使用指定的线程池。

### 6 纯消费

thenAccept(Consumer<? super T> action)

当CompletableFuture完成计算结果，只对结果执行Action，而不返回新的计算值

thenAcceptAsync(Consumer<? super T> action)

当CompletableFuture完成计算结果，只对结果执行Action，而不返回新的计算值，使用ForkJoinPool。

thenAcceptAsync(Consumer<? super T> action, Executor executor)

当CompletableFuture完成计算结果，只对结果执行Action，而不返回新的计算值

### 7 Either

纯消费：

acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action)

当任意一个CompletableFuture完成的时候，action这个消费者就会被执行。

acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action)

当任意一个CompletableFuture完成的时候，action这个消费者就会被执行。使用ForkJoinPool

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

当任意一个CompletableFuture完成的时候，action这个消费者就会被执行。使用指定的线程池

有返回值：

applyToEither(CompletionStage<? extends T> other, Function<? super T,U> fn)

当任意一个CompletableFuture完成的时候，fn会被执行，它的返回值会当作新的CompletableFuture<U>的计算结果。

applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn)

当任意一个CompletableFuture完成的时候，fn会被执行，它的返回值会当作新的CompletableFuture<U>的计算结果。使用ForkJoinPool

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

当任意一个CompletableFuture完成的时候，fn会被执行，它的返回值会当作新的CompletableFuture<U>的计算结果。使用指定的线程池

### 8 allOf anyOf

allOf(CompletableFuture<?>... cfs) 在所有Future对象完成后结束，并返回一个future。

anyOf(CompletableFuture<?>... cfs) 在任何一个Future对象结束后结束，并返回一个future。

### 9 get()和join()区别：

CompletableFuture类中的join方法和Future接口中的get有相同的含义，并且也声明在Future接口中，它们唯一的不同是join不会抛出任何检测到的异常。

## 参考博客链接：

https://juejin.im/post/59eae61b51882549fc512b34#comment

http://colobu.com/2016/02/29/Java-CompletableFuture/#%E4%B8%BB%E5%8A%A8%E5%AE%8C%E6%88%90%E8%AE%A1%E7%AE%97