并发

一、基本概念

1. `Java` 提供了围绕着共享内存和锁构建的并发支持。虽然这种支持是完备的，但是这种并发方案在实际过程中很难正确使用。

   `Scala` 标准库提供了另一种能够规避这些难点的选择，将程序员的精力集中在不可变状态的异步变换上，即 `Future` 。

   虽然 `Java` 也提供 `Future`，但是它和 `Scala` 的 `Future` 不同：

   - 虽然两个 `Future` 都代表某个异步计算的结果，但是 `Java` 的 `Future` 要求通过阻塞的 `get` 方法来访问这个结果。

     虽然在调用 `get` 方法之前可以先通过调用 `idDone` 来判断某个 `Java` 的 `Future` 是否已经完成，从而避免阻塞，但是你却必须等到 `Java` 的 `Future` 完成之后才能继续用这个结果来进行下一步的计算。

   - `Scala` 的 `Future` 不同：无论 `Future` 的计算是否完成，都可以指定对它的变换逻辑。每个变换都会产生新的 `Future` 来表示对原始的 `Future` 经过给定的函数变换之后产生的异步结果。执行计算的线程由隐式给出的执行上下文 `execution context` 来决定。这使得你可以将异步的计算描述成一系列的、对不可变值的变换，完成不用考虑共享内存的锁。

2. 在 `Java` 平台上，每个对象都关联了一个逻辑监视器`moniter` ，可以用于控制对数据的多线程访问。此时，需要由你来决定哪些数据将被多个线程共享，并将访问共享数据或者控制对这些共享数据访问的代码段标记为 `synchronized`。

   `Java` 在运行时将运用一种锁机制来确保同一时间只有一个线程进入由同一个锁控制的同步代码段，从而让你可以协调共享数据的多线程访问。

   为了兼容，`Scala` 提供了对 `Java` 并发原语的支持。我们可以用 `Scala` 调用 `wait, notify, notifyAll` 等方法，它们的含义跟 `Java` 中的方法一样。

   从技术上讲，`Scala` 并没有 `synchronized` 关键字，不过它有一个 `synchronized` 方法，可以像这样来调用：

   ```
   var counter = 0
   synchronized{
     counter += 1  // 这里每次只有一个线程在执行
   }
   ```

3. 事实上，程序员发现通过使用共享数据和锁模型来构建健壮的多线程的应用程序十分困难。难点在于：在程序中的每一点，你都必须推断出哪些你正在修改或访问的数据有可能被其它线程修改或访问，以及在这一点你持有哪些锁。

   每次方法调用，你都必须推断它将会尝试持有那些锁，并判断它有没有可能死锁。而这种推断过程并不是在编译器决定的，而是在运行过程中动态决定的，因为程序在运行过程中可以任意创建新的锁。这进一步加剧了问题的复杂性。

   另外，对于多线程的代码而言，测试是不可靠的。由于线程是非确定性的，可能当你测试 `1000` 次都是成功的，但是程序第一次在线上运行时就出问题了。

   对于共享数据和锁，你必须通过推断来验证程序的正确性，别无他途。

另外，你也无法通过过度的同步来解决问题。同步一切并不比什么都不同步要更好。原因是：尽管使用尽可能多的锁来解决竞争问题，但是也增加了死锁的可能性。正确的程序既不应该存在竞争情况，也不应该存在死锁情况。因此无论偏向哪个方向都是不安全的。

`java.util.concurrent` 类库提供了并发编程的更高级别的抽象。使用这个工具包来进行多线程编程要比你自己使用更低级别的同步语法更方便，并且引入问题的可能性要小得多。尽管如此，这个工具包还是基于共享数据和锁的，因此并没有从根本上解决这类模型的问题。

4. `Scala` 的 `Future` 提供了一种减少（甚至免去）对共享数据和锁进行推断的方式。
    - 当你调用 `Scala` 方法时，它“在你等待的过程中”执行某项计算并返回结果。
    - 如果该结果是一个 `Future`，则表示另一个将要被异步执行的计算，并且这个计算通常是由另一个完全不同的线程来执行。

    因此，对 `Future` 的很多操作都需要一个隐式的执行上下文 `execution context` 来提供异步执行函数的策略。可以通过使用 `Scala` 提供的一个全局的执行上下文。对 `JVM` 而言，这个全局的执行上下文使用的是一个线程池。一旦将隐式的执行上下文纳入到作用域中，在可以创建 `Future`：

    ```
    import scala.concurrent.ExecutionContext.Implicits.global
    import scala.concurrent.Future

    val fut = Future{ Thread.sleep(10000); 21 + 21} // 对应的线程先睡眠10秒，然后计算出 42
    ```

5. `Future` 有两个方法让你轮询：
    - `isCompleted`：判断 `Future` 的异步计算是否完成。如果已完成则返回 `true`；如果未完成则返回 `false`。
    - `value`：返回 `Future` 异步计算的结果。如果已完成则返回 `Some`；如果未完成则返回 `None`。

    ```
    val fut = Future{ Thread.sleep(10000); 21 + 21}
    //****************** 10 秒钟以内 **************//
    fut.isCompleted  // 返回 false
    fut.value        // 返回 None
    //****************** 10 秒钟以后 **************//
    fut.isCompleted  // 返回 true
    fut.value        // 返回 Some(Success(42))
    ```

    `value` 的返回值类型其实是 `Option[scala.util.Try[T]]`。当 `Future` 计算已完成时，`Some` 包含一个 `Try` 类型的对象。`Try` 对象要么是包含类型为 `T` 的值的 `Success`、要么是包含一个异常（`java.lang.Throwable` 实例）的 `Failure`。

    `Try` 的目的是为异步计算提供一种与同步计算中 `try` 表达式类似的东西：允许你处理那些有可能异常终止而不是返回正常结果的情况。

    `Try` 的继承关系如下：

```
                scala.util
                  Try[+T]
            << sealed abstract >>
            ^                   ^
           /                     \
          /                       \
   scala.util                    scala.util
   Success[T]                    Failure[T]
<< final case >>              << final case >>
```

对同步计算而言，你可以通过使用 `try / catch` 来确保调用某个方法的线程可以捕获并处理由该方法抛出的异常。不过对于异步计算而言，发起该计算的线程通常都转到别的任务了。在这之后，如果异步计算因为某个异常失败了，原始的线程就无法再用 `catch` 来捕获这个异常了。因此，当处理表示异步计算的 `Future` 时，你需要用 `Try` 来处理这种情况：该活动未能交出某个结果，而是异常终止了。

二、使用 Future

2.1 基础变换

1. 在 `Scala` 中，你可以对 `Future` 的结果指定变换，然后得到一个新的 `Future`，从而表示这两个异步计算的组合：原始的异步计算和异步变换。
2. 最基础的变换是 `map`，可以直接将下一个计算 `map` 到当前的 `Future`，而不是阻塞等待结果。`map` 之后得到一个新的 `Future`，表示原始的异步计算经过传给 `map` 的函数异步变换之后的结果。

```
val fut = Future{ Thread.sleep(10000); 21 + 21} // 异步计算
val fut2 = fut.map( x => x +1)                  // 异步变换
```

当异步计算和异步变换完成之后，`fut2.value` 返回 `Some(Success(43))`。

注意：这里创建 `fut, fut2` 的线程、`fut` 的异步计算线程、`fut2` 的异步变换线程可能分别在三个不同的线程中执行。

3. 还可以使用 `for` 表达式来对 `Future` 进行变换。考虑两个 `Future`：

```
val fut1 = Future{ Thread.sleep(10000); 21 + 21}
val fut2 = Future{ Thread.sleep(10000); 23 + 23}
```

可以通过 `for` 表达式来构造一个新的 `Future`：

```
for {
  x <- fut1
  y <- fut2
} yield x + y
```

一旦这三个 `Future` 完成，你将会得到最终结果为 `Some(Success(88))`。

注意：因为 `for` 表达式会串行化它们的变换，因此如果你没有在 `for` 之前创建好 `Future`，则它们并不会并行执行。如：

```
//************************** fut1, fut2 并行执行 ******************//
val fut1 = Future{ Thread.sleep(10000); 21 + 21}
val fut2 = Future{ Thread.sleep(10000); 23 + 23}
val result = for {
  x <- fut1
  y <- fut2
} yield x + y
//************************** fut1, fut2 串行执行 ******************//
val result2 = for {
  x <- Future{ Thread.sleep(10000); 21 + 21}
  y <- Future{ Thread.sleep(10000); 23 + 23}
} yield x + y
```

事实上这里的 `for` 表达式会被重写为 `fut1.flatMap(x => fut2.map(y => x + y))` 。

4. 可以通过 `Future` 伴生对象的几个方法来创建 `Future` ：
   - `Future` 伴生对象的 `apply` 方法，如 `Future{ Thread.sleep(10000); 21 + 21}` 。
   - `Future` 伴生对象的 `Future.failed, Future.successful, Future.fromTry` 等工厂方法。这些工厂方法并不需要 `ExecutionContext` 。
     - `successful` 工厂方法将创建一个已经成功完成的 `Future` ：

       ```
       Future.successful{ 21 + 21}
       ```

     - `failed` 工厂方法将创建一个已经失败的 `Future` ：

       ```
       Future.failed( new Exception("xxx Exception"))
       ```

     - `fromTry` 工厂方法将从给定的 `Try` 创建一个已经完成的 `Future` ：

       ```
       import scala.util.{Success, Failure}
       Future.fromTry(Success {21 + 21})
       Future.fromTry(Failure(new Exception("xxx Exception")))
       ```

   - 创建 `Future` 最一般化的方法是使用 `Promise` 。给定一个 `Promise` ，可以得到由这个 `Promise` 控制的 `Future` 。当你完成 `Promise` 时，对应的 `Future` 也会完成。

     ```
     val pro = Promise[Int]
     val fut = pro.future
     ```

     可以用名为 `success, failure, complete` 的方法来完成 `Promise` 。这些方法和 `Future` 中的方法很相似：

     ```
     val pro = Promise[Int]
     pro.success(42)
     pro.failure(new Exception("xxx Exception"))
     pro.complete(Failure(new Exception("xxx Exception")))
     ```

     还有一个 `completeWith` ，这个方法使得该 `Promise` 的 `future` 的完成状态和你传入的 `future` 保持同步：

```
val fut1 = Future{ Thread.sleep(10000); 21 + 21}
val pro = Promise[Int]
pro.completeWith(fut1)
```

5. `Future` 的 `filter` 方法对 `Future` 的结果进行校验，如果合法就原样保留；如果非法，则 `filter` 返回的这个 `Future` 就以 `NoSuchElementException` 失败：

```
val fut1 = Future{ Thread.sleep(10000); 21 + 21}
val valid = fut1.filter( x => x > 0) // 合法的结果
valid.value  // 返回 Some(Success(42))

val invalid = fut1.filter( x => x <0) // 非法的结果
invalid.value // 返回 Some(Failure(java.util.NoSuchElementException))
```

另外，`Future` 还提供了 `withFilter` 方法，因此可以用 `for` 表达式的过滤器来执行相同的操作：

```
val fut1 = Future{ Thread.sleep(10000); 21 + 21}
val valid = for (x <- fut1 if x >0) yield x // 合法的结果
valid.value  // 返回 Some(Success(42))

val invalid = for (x <- fut1 if x <0) yield x  // 非法的结果
invalid.value // 返回 Some(Failure(java.util.NoSuchElementException))
```

6. `Future` 的 `collect` 方法允许你在一次操作中同时完成校验和变换。
   - 如果传给 `collect` 方法的偏函数对 `Future` 结果有定义，则 `collect` 返回的 `Future` 就会以经过该函数变换后的值成功完成。

     ```
     val fut1 = Future{ Thread.sleep(10000); 21 + 21}
     val valid = fut1 collect {case x if x >0 => x + 46}
     valid.value // 返回 Some(Success(88))
     ```

   - 如果传给 `collect` 方法的偏函数对 `Future` 结果没有定义，则 `collect` 返回的 `Future` 就以 `NoSuchElementException` 失败。

     ```
     val fut1 = Future{ Thread.sleep(10000); 21 + 21}
     val invalid = fut1 collect {case x if x < 0 => x + 46}
     valid.value // 返回 Some(Failure(java.util.NoSuchElementException))
     ```

2.2 处理失败

1. `Scala` 的 `Future` 提供了处理失败的 `Future` 的方式，包括：`failed, fallbackTo, recover, recoverWith`。
2. `Future` 的 `failed` 方法将任何类型的失败的 `future` 变换成一个成功的 `Future[Throwable]`，并带上引发失败的异常。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}  // 类型为 Future[Int]
fut1.value // 返回 Some(Failure(java.lang.ArithmeticException))

val fut2 = fut1.failed // 类型为 Future[Throwable]
fut2.value // 返回 Some(Success(java.lang.ArithmeticException))
```

如果调用 `failed` 的 `Future` 最终成功了，则 `failed` 返回的这个 `Future` 将以 `NoSuchElementException` 失败。因此 `failed` 方法只有在你预期某个 `future` 一定会失败的情况下才适用。

```
val fut1 = Future{ Thread.sleep(10000); 21/1}  // 类型为 Future[Int]
fut1.value // 返回 Some(Success(21))

val fut2 = fut1.failed
fut2.value // 返回 Some(Failure(java.util.NoSuchElementException))
```

3. `Future` 的 `fallbackTo` 方法允许你提供一个额外可选的 `Future`，这个新的 `Future` 将用于在你调用 `fallbackTo` 的那个 `Future` 失败的情况。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); 21/1}
val fut3 = fut1.fallbackTo(fut2)
fut3.value // 返回 Some(Success(21))
```

   - 如果 `fut1` 成功，则 `fut1.fallbackTo(fut2)` 返回 `fut1`。
   - 如果 `fut1` 失败，而 `fut2` 成功，则 `fut1.fallbackTo(fut2)` 返回 `fut2`。
   - 如果 `fut1` 失败，且 `fut2` 失败，则 `fut1.fallbackTo(fut2)` 返回 `fut1`。此时完全不考虑 `fut2` 的失败。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); assert(21 <0)}
val fut3 = fut1.fallbackTo(fut2)
fut3.value // 返回 Some(Failure(java.lang.ArithmeticException))
```

4. `recover` 方法让你把失败的 `future` 变换成成功的 `future`，同时将成功的 `future` 结果原样透传。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = fut1 recover {
  case ex => -1
}
fut2.value // 返回 Some(Success(-1))
```

如果原始的 `Future` 没有失败，则 `recover` 返回的这个 `Future` 就会以相同的值完成：

```
val fut1 = Future{ Thread.sleep(10000); 21/1}
val fut2 = fut1 recover {
  case ex => -1
}
fut2.value // 返回 Some(Success(21))
```

如果传给 `recover` 的偏函数没有对引发原始 `Future` 最终失败的那个异常有定义，则原始的失败会被透传：

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = fut1 recover {
  case ex : IllegalArgumentException => -1
}
fut2.value // 返回 Some(Failure(java.lang.ArithmeticException))
```

5. `recoverWith` 方法和 `recover` 很像，不过它并不是像 `recover` 一样恢复成某个值，而是恢复成一个 `Future`：

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = fut1 recoverWith {
  case ex => Future{ Thread.sleep(10000); 21+21 } // 这里恢复成一个 Future，而不是一个值
}
fut2.value // 返回 Some(Success(42))
```

和 `recover` 一样，如果原始的`Future` 没有失败，则 `recoverWith` 返回的这个 `Future` 就会以相同的值完成：

```
val fut1 = Future{ Thread.sleep(10000); 21/1}
val fut2 = fut1 recoverWith {
  case ex => Future{ Thread.sleep(10000); 21+21 }
}
fut2.value // 返回 Some(Success(21))
```

和 `recover` 一样，如果传给 `recoverWih` 的偏函数没有对引发原始 `Future` 最终失败的那个异常有定义，则原始的失败会被透传：

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = fut1 recoverWith {
  case ex : IllegalArgumentException => Future{ Thread.sleep(10000); 21+21
}
}
fut2.value // 返回 Some(Failure(java.lang.ArithmeticException))
```

2.3 transform

1. `Future` 的 `transform` 方法接收两个函数来对 `Future` 进行变换：一个用于处理成功、另一个用于处理失败。

```
val factor = 1
val fut1 = Future{ Thread.sleep(10000); 21/factor}
val fut2 = fut1.transform(
  x => x * -1,        // 如果 fut1 成功，则调用这一行。此时 fut2 返回
Some(Success(-21))
  ex => new Exception("exception because:",ex) // 如果 fut1 失败（例如
factor = 0时），则调用这一行，此时 fut2 返回 Some(Failure(java.lang.Exception))
)
```

如果 `fut1` 成功了，则调用第一个函数；如果 `fut1` 失败了，则调用第二个函数。

注意：`transform` 方法并不能将成功的 `Future` 改变成失败的 `Future`，也不能将失败的 `Future` 改变成成功的 `Future`。

2. 为了将成功的 `Future` 改变成失败的 `Future`，或者将失败的 `Future` 改变成成功的 `Future`，`Scala 2.12` 引入了一个重载的 `transform` 形式，接收一个从 `Try` 到 `Try` 的函数：

```
val factor = 1
val fut1 = Future{ Thread.sleep(10000); 21/factor}
val fut2 = fut1.transform{  // 在 Scala 2.12 之后支持
  case Success(x) => Success(x.abs + 1)
  case Failure(_) => Success(0)  // 将 Failure 变换为 Success
}
```

2.4 组合

1. `Future` 和它的伴生对象提供了用于组合多个 `Future` 的方法。
2. `zip` 方法将两个成功的 `Future` 变换成这两个值的元组的 `Future` 。

```
val fut1 = Future{ Thread.sleep(10000); 21/1}
val fut2 = Future{ Thread.sleep(10000); 21+21}
val fut3 = fit1.zip(fut2) // 一个 Future[(Int,Int)] 对象
fut3.value // 返回 Some(Success(21,42))
```

如果任何一个 `Future` 失败了，`zip` 返回的这个 `Future` 也会以相同的异常失败。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); 21+21}
val fut3 = fit1.zip(fut2) // 一个 Future[(Int,Int)] 对象
fut3.value // 返回 Some(Failure(java.lang.ArithmeticException))
```

如果两个 `Future` 都失败了，则 `zip` 返回的是第一个 `Future` （也就是调用方）的那个异常。

3. `Future` 伴生对象提供了一个 `fold` 方法，用于累积一个 `TranserableOnce` 集合中所有 `Future` 的结果，并交出一个 `Future` 的结果。

如果集合中所有 `Future` 都成功了，则累积的 `Future` 成功完成；如果集合中有任何一个 `Future` 失败了，则累积的 `Future` 失败，并且返回集合中首个失败的 `Future` 的失败。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); 21+21}
val fut3 = Future.fold(List(fut1,fut2))(0){
  (acc, num) => acc + num
}
```

4. `Future` 伴生对象提供的 `reduce` 方法和 `fold` 方法都是一样的折叠操作，但是 `reduce` 不需要提供初始值，而是用第一个 `Future` 结果作为初始值。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); 21+21}
val fut3 = Future.reduce(List(fut1,fut2)){
  (acc, num) => acc + num
}
```

   如果传入的集合为空，则没有第一个 `Future`，此时 `reduce` 将以 `NoSuchElementException` 作为失败。

5. `Future.sequence` 方法将一个 `TransversableOnece[Future]` 集合变换成 `Future[TranversableOnece]` 的 `Future`：

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); 21+21}
val list = List(fut1, fut2) // List[Future[Int]]
val fut3 = Future.sequence(list) // Future[List[Int]]
```

6. `Future.traverse` 方法会将任何元素类型的 `TranversableOnce` 集合变成一个由 `Future` 组成的 `TranversableOnce`，并将它 `sequence` 成一个由值组成的 `TraversableOnce` 的 `Future`：

```
val fut1 = Future.traverse(List(1,2,3)){i => Future(i)}
fut1.value // Some(Success(1,2,3))
```

2.5 执行副作用

1. 有可能你希望在执行完某个 `Future` 完成之后执行一个副作用（而不是返回一个新的 `Future`），为此 `Future` 提供了好几种方法。
2. 最基本的方法是 `foreach`，如果 `Future` 成功则它会执行一个副作用。

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); 21/1}
fut1.foreach(x => println(x)) // fut1 执行失败，所以不会执行 println
fut2.foreach(x => println(x)) // 当 fut2 执行成功的时候执行 println
```

   由于不带 `yield` 的 `for` 会被编译器重写为对 `foreach` 的调用，因此也可以通过 `for` 表达式来达成同样的效果：

```
val fut1 = Future{ Thread.sleep(10000); 21/0}
val fut2 = Future{ Thread.sleep(10000); 21/1}
for (x <- fut1) println(x)
for (y <- fut2) println(y)
```

3. `Future` 还提供了方法来“注册”函数。`onComplete` 方法在 `Future` 最终成功或失败时都会被执行。被注册的函数会被传入一个 `Try` 对象（如果 `Future` 成功了，那么就算一个包含结果的 `Success`；否则是一个包含失败原因的 `Failure`）。

```
import scala.util.{Success, Failure}
val fut1 = Future{ Thread.sleep(10000); 21/1}
fut1 onComplete {
  case Success(x) => println(x)
  case Failure(ex) => println(ex)
}
```

4. 可以多次调用 `Future.onComplete` 来注册多个函数，但是 `onComplete` 不能保证这些函数之间的执行顺序。如果你希望强制回调函数的顺序，则可以考虑使用 `Future.andThen` 方法。

   `Future.andThen` 方法会返回一个新的 `Future`，这个新的 `Future` 是对原始的 `Future` 调用回调函数：

```
import scala.util.{Success, Failure}
val fut1 = Future{ Thread.sleep(10000); 21/1}
val fut2 = fut1 andThen {
  case Success(x) => println(x)
  case Failure(ex) => println(ex)
}
```

   如果传入 `adThen` 的回调函数在执行时抛出异常，则这个异常是不会被传导到后续的 `adThen`，也不会体现在结果的 `Future` 中。

2.6 Scala 2.12 中的新方法

1. `scala 2.12` 中添加的 `Future.flatten` 方法将一个 `Future[Future[Int]]` 变换成一个 `Future[Int]`。如：

```
val fut = Future{ Future{Thread.sleep(10000); 21/1} } //
Future[Future[Int]]
val fut1 = fut.flatten // Future[Int]
```

2. `scala 2.12` 中添加的 `zipWith` 方法将两个 `Future zip` 到一起，然后在对结果的元组执行 `map`。

```
val fut1 = Future{ Future{Thread.sleep(10000); 21/1} }
val fut2 = Future{ Future{Thread.sleep(10000); "hello world"} }
val fut3 = fut1 zip fut2 // Future[(Int,String)]
val fut4 = fut3 map{     // Future[String]
  case (num, str) => s"$num and $str"
}
fut4.value // 返回 Some(Success(21 and hello world))
```

`zipWith` 允许你一步完成相同的操作：

```
val fut1 = Future{ Future{Thread.sleep(10000); 21/1} }
val fut2 = Future{ Future{Thread.sleep(10000); "hello world"} }
val fut4 = fut1.zipWith(fut2){
   case (num, str) => s"$num and $str"
}
fut4.value // 返回 Some(Success(21 and hello world))
```

3. `scala 2.12` 中还添加了 `transformWith` 方法，可以用一个从 `Try` 到 `Future` 的函数对 `Future` 进行变换。

```
val fut1 = Future{ Future{Thread.sleep(10000); 21/1} }
val fut2 = fut1.transformWith{
  case Success(res) => Future{ throw new Exception(res.toString)}
  case Failure(ex) => Future(0)
}
```

`transformWith` 方法和 `Future.transform` 类似，但是 `transformWith` 方法允许你交出 `Future`（相比较之下，`transform` 方法需要你交出 `Try`）。

三、测试

1. `scala` 的 `Future` 的一个优势是它能帮你避免阻塞。

   在大多数 `JVM` 实现中，创建几千个线程之后，在线程之间的上下文切换会使得性能无法接受。通过避免阻塞，可以持有一组数量有限的线程，让它们不停地工作。

   尽管如此，`Scala` 也允许你在需要的时候阻塞线程（从而等待它的结果）。`Scala` 的 `Await` 对象提供了等待 `Future` 结果的手段。

```
import scala.concurrent.Await
import scala.concurrent.duration._
val fut = Future{ Future{Thread.sleep(10000); 21/1} }
val x = Await.result(fut, 15.seconds) // 阻塞调用，返回 21
```

   `Await.result` 接收一个 `Future` 和一个 `Duration`（表示 `Await.result` 最长等待多长时间等待 `Future` 完成）。如果时间超过 `Duration` 但是 `Future` 尚未完成，则触发超时。

2. 通常用线程阻塞来对异步代码进行测试。既然 `Await.result` 已经返回结果，那么就可以对这个结果进行测试：

```
import scala.concurrent.Await
import scala.concurrent.duration._
import org.scalatest.Matchers._
val fut = Future{ Future{Thread.sleep(10000); 21/1} }
val x = Await.result(fut, 15.seconds) // 阻塞调用，返回 21
x should be (21) // 测试代码
```

3. 也可以使用 `ScalaTest` 的 `ScalaFutures` 特质提供的阻塞结构。如 `ScalaFuture` 为 `Future` 隐式添加的 `futureValue` 方法，该方法会阻塞直到 `future` 完成。
   - 如果 `future` 失败了，则 `futureValue` 方法会抛出 `TestFailedException` 来描述这个失败。
   - 如果 `future` 成功了，则 `futureValue` 方法会返回成功的结果。

```
import org.scalatest.concurrent.ScalaFutures._
val fut =  Future{ Future{Thread.sleep(10000); 21/1} }
fut.futureValue should be (42)   // futureValue 是阻塞的
```

4. 虽然阻塞线程并进行测试没什么问题，但是如果能够不阻塞线程并异步执行测试，则和生产环境保持一致。为此，`ScalaTest 3.0` 添加了异步测试的风格。

   拿到 `Future` 之后，你并不是先阻塞然后对结果执行断言，而是将断言直接 `map` 到 `future` 上，然后返回 `Future[Assertion]` 给 `ScalaTest` 。

   当这个 `Future` 的断言完成时，`ScalaTest` 会异步地将事件（测试成功、测试失败等）发送给测试报告程序。

```
import org.scalatest.AsyncFunSpec
import scala.concurrent.Future

class AddSpec extends AsyncFunSpec{
  def addSoon(addends: Int*) : Future[Int] = Future{addends.sum}

  describe("addSoon"){
    it("will compute a sum of ints"){
      val futureSum:Future[Int] = addSoon(1,2,3)
      futureSum map { sum => assert(sum == 6)} // 将断言映射成 Future，然后返回 Future[Assertion] 给 ScalaTest
    }
  }
}
```

5. 异步测试的用例展示了处理 `Future` 的一般原则：一旦进入 `Future` 空间，就尽量呆在 `Future` 空间。不要对一个 `Future` 阻塞拿到结果后再继续进行计算，而是通过执行一系列的变换，并在每个变换返回新的 `Future` 以供下游进一步变换从而保持异步性。

   当需要从 `Future` 空间拿到结果时，则注册副作用，并在 `Future` 完成时异步执行。

   这种方式可以让你以最大限度的利用线程。