Chapter12 - 코루틴 단위 테스트

12.1. 단위 테스트 기초

12.1.1. 단위 테스트란 무엇인가?

단위(Unit) : 명확히 정의된 역할의 범위를 갖는 코드의 집합으로, 소프트웨어의 기능을 담는 코드 블록. 즉, 정의된 동작을 실행하는 개별 함수나 클래스 또는 모듈이 모두 단위가 될 수 있다.
단위 테스트 : '단위'에 대한 자동화된 테스트를 작성하고 실행하는 프로세스
단위 테스트를 작성한다 : 소프트웨어의 특정 기능이 제대로 동작하는지 확인하는 테스트를 작성하는 것을 의미.

객체 지향 프로그래밍에서의 단위 : 객체 지향 프로그래밍이란 책임을 객체에 할당하고, 객체 간의 유연한 협력관계를 구축하는 것을 의미. 다시 말해 소프트웨어의 기능을 담는 역할은 객체가 한다. 따라서 객체 지향 프로그래밍에서 테스트 대상이 되는 단위는 주로 **객체**가 된다.

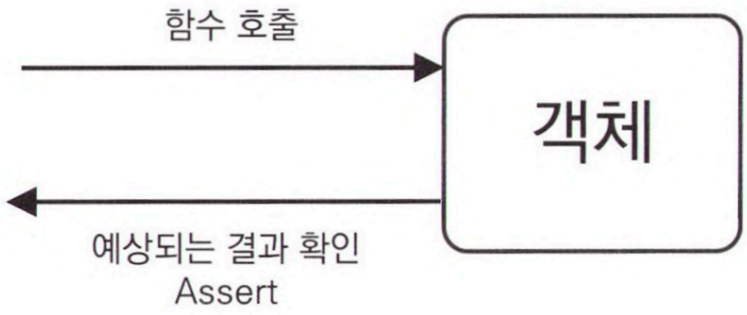

그림 12-1 코틀린의 단위 테스트

테스트 대상이 객체인 경우 객체의 함수를 호출하고 함수가 호출되면 객체가 예상한 대로 동작하는지 확인하는 과정을 통해 테스트를 진행할 수 있다.
객체가 예상한 대로 동작하는지 확인하는 방법 : 결과값이 제대로 반환되는지, 해당 객체가 가진 상태(변수 값)가 잘 변화하는지, 함수 호출 시 해당 객체가 의존성을 가진 다른 객체와 제대로 상호 작용하는지 등을 확인할 수 있다.

12.1.3. 간단한 테스트 만들고 실행하기

```
class AddUseCase {
  fun add(vararg args: Int): Int {
    return args.sum()
  }
}
```

```
class AddUseCaseTest {
  @Test
  fun `1 더하기 2는 3이다`() {
    val addUseCase: AddUseCase = AddUseCase()
    val result = addUseCase.add(1, 2)
    assertEquals(3, result)
  }
}
```

개별 테스트는 @Test 어노테이션이 붙은 함수로 작성된다.
단언(Assert)이란 테스트를 검증하는 데 사용하는 개념으로 특정한 조건이 참(true)임을 검증한다. 여기서는 결과값이 3과 동일한지 확인하기 위해 assertEquals 단언을 사용했다.

12.1.4. @BeforeEach 어노테이션을 사용한 테스트 환경 설정

```
class AddUseCaseTestMultipleCase {
  @Test
  fun `1 더하기 2는 3이다`() {
    val addUseCase: AddUseCase = AddUseCase()
    val result = addUseCase.add(1, 2)
    assertEquals(3, result)
  }

  @Test
  fun `-1 더하기 2는 1이다`() {
    val addUseCase: AddUseCase = AddUseCase()
    val result = addUseCase.add(-1, 2)
    assertEquals(1, result)
  }
}
```

위와 같이 AddUseCase 객체에 대한 테스트가 두 개 이상 존재할 때, AddUseCase 클래스를 인스턴스화 하는 코드가 똑같이 반복된다.

```
class AddUseCaseTestBeforeEach {
  lateinit var addUseCase: AddUseCase

  @BeforeEach
  fun setUp() {
    addUseCase = AddUseCase()
  }

  @Test
```

```
  fun `1 더하기 2는 3이다`() {
    val result = addUseCase.add(1, 2)
    println(result)
    assertEquals(3, result)
  }

  @Test
  fun `-1 더하기 2는 1이다`() {
    val result = addUseCase.add(-1, 2)
    println(result)
    assertEquals(1, result)
  }
}
```

이런 문제 해결을 위해 @BeforeEach 어노테이션을 사용할 수 있다. @BeforeEach 함수를 만들면 해당 함수는 모든 테스트 실행 전에 공통으로 실행된다. 이제 테스트가 아무리 늘어나도 AddUseCase 클래스를 인스턴스화하는 코드를 중복적으로 추가할 필요가 없다.

12.1.5. 테스트 더블을 사용해 의존성 있는 객체 테스트하기

```
data class UserProfile(val id: String, val name: String, val phoneNumber:
String)
```

```
interface UserNameRepository {
    fun saveUserName(id: String, name: String)
    fun getNameByUserId(id: String): String
}
```

```
interface UserPhoneNumberRepository {
    fun saveUserPhoneNumber(id: String, phoneNumber: String)
    fun getPhoneNumberByUserId(id: String): String
}
```

```
class UserProfileFetcher(
    private val userNameRepository: UserNameRepository,
    private val userPhoneNumberRepository: UserPhoneNumberRepository
) {
    fun getUserProfileById(id: String): UserProfile {
        // 유저의 이름을 UserNameRepository로부터 가져옴
        val userName = userNameRepository.getNameByUserId(id)
        // 유저의 전화번호를 UserPhoneNumberRepository로부터 가져옴
        val userPhoneNumber =
```

```
        userPhoneNumberRepository.getPhoneNumberByUserId(id)
        return UserProfile(
            id = id,
            name = userName,
            phoneNumber = userPhoneNumber
        )
    }
}
```

UserNameRepository, UserPhoneNumberRepository 인터페이스에 대한 구현체가 없어 UserProfileFetcher 객체에 대한 테스트 코드를 작성하는 것이 쉽지 않은 상황이다. 그리고 실제 구현체가 있다 하더라도 UserProfileFetcher 객체의 테스트가 UserNameRepository나 UserPhoneNumberRepository의 구현체에 영향을 받기 때문에 제대로 된 테스트를 할 수 없다.

다른 객체와의 의존성을 가진 객체를 테스트하기 위해서는 테스트 더블(Test Double)이 필요하다. 테스트 더블은 객체에 대한 대체물을 뜻하며, 객체의 행동을 모방하는 객체를 만드는 데 사용한다.

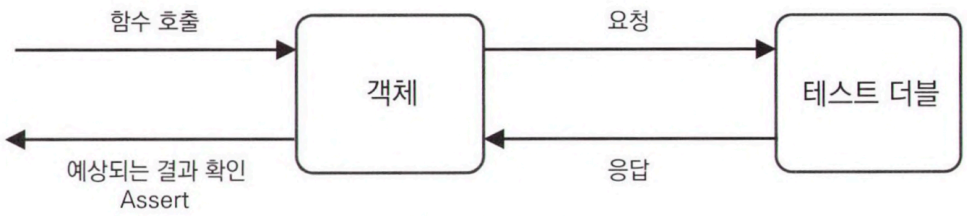

그림 12-8 테스트 더블

테스트 더블의 종류 : 스텁(Stub), 페이크(Fake), 목(Mock), 더미(Dummy), 스파이(Spy) 등

스텁

스텁 객체 : 미리 정의된 데이터를 반환하는 모방 객체. 반환값이 없는 동작은 구현 X, 반환값이 있는 동작만 미리 정의된 데이터를 반환하도록 구현.

```
// 유연한 StubUserNameRepository
class StubUserNameRepository(
    private val userNameMap: Map<String, String> // 데이터 주입
) : UserNameRepository {
    override fun saveUserName(id: String, name: String) {
        // 구현하지 않는다.
    }

    override fun getNameByUserId(id: String): String {
        return userNameMap[id] ?: ""
    }
}
```

StubUserNameRepository 클래스의 인스턴스 변수로 userNameMap을 선언할 수도 있지만, 그렇게 하면 userNameMap이 특정한 값으로 고정돼 있기 때문에 유연하지 못하다. 스텁을 좀 더 유연하게 만들기 위해서 userNameMap을 주입받도록 만든 코드이다.

페이크

페이크 객체 : 실제 객체와 비슷하게 동작하도록 구현된 모방 객체

```
class FakeUserPhoneNumberRepository : UserPhoneNumberRepository {
  private val userPhoneNumberMap = mutableMapOf<String, String>()

  override fun saveUserPhoneNumber(id: String, phoneNumber: String) {
    userPhoneNumberMap[id] = phoneNumber
  }

  override fun getPhoneNumberByUserId(id: String): String {
    return userPhoneNumberMap[id] ?: ""
  }
}
```

UserPhoneNumberRepository 인터페이스의 실제 구현체가 로컬 DB를 사용해 유저의 전화번호를 저장한다고 가정했을 때, FakeUserPhoneNumberRepository 객체는 로컬 DB 대신 인메모리에 저장해 실제 객체처럼 동작할 수 있도록 만든다.

```
class UserProfileFetcherTest {
  @Test
  fun `UserPhoneNumberRepository에 휴대폰 번호가 저장되어 있으면, UserProfile를 가져왔을 때 해당 휴대폰 번호가 반환되어야 한다`() {
    // Given
    val userProfileFetcher = UserProfileFetcher(
      userNameRepository = StubUserNameRepository(
        userNameMap = mapOf<String, String>(
          "0x1111" to "홍길동",
          "0x2222" to "조세영"
        )
      ),
      userPhoneNumberRepository = FakeUserPhoneNumberRepository().apply {
        this.saveUserPhoneNumber("0x1111", "010-xxxx-xxxx")
      }
    )

    // When
    val userProfile = userProfileFetcher.getUserProfileById("0x1111")
```

```
    // Then
    assertEquals("010-xxxx-xxxx", userProfile.phoneNumber)
  }
}
```

Given-When-Then 주석은 테스트의 시나리오를 설명함으로써 가독성을 높이는 데 사용된다.
Given : 테스트 환경을 설정
When : 동작이나 이벤트 발생시키고 결과 얻는다
Then : 테스트 결과를 검증

12.2. 코루틴 단위 테스트 시작하기

12.2.1. 첫 코루틴 테스트 작성하기

```
class RepeatAddUseCase {
  suspend fun add(repeatTime: Int): Int = withContext(Dispatchers.Default) {
    var result = 0
    repeat(repeatTime) {
      result += 1
    }
    return@withContext result
  }
}
```

```
class RepeatAddUseCaseTest {
  @Test
  fun `100번 더하면 100이 반환된다`() = runBlocking {
    // Given
    val repeatAddUseCase = RepeatAddUseCase()

    // When
    val result = repeatAddUseCase.add(100)

    // Then
    assertEquals(100, result)
  }
}
```

일시 중단 함수는 코루틴 블록 내부에서 실행되어야 하므로 테스트 함수를 runBlocking으로 감싸주었다. 일반적인 일시 중단 함수는 runBlocking을 사용하는 것만으로도 충분하다.

12.2.2. runBlocking을 사용한 테스트의 한계

runBlocking 함수를 사용한 테스트에서는 실행에 오랜 시간이 걸리는 일시 중단 함수를 실행하면 문제가 나타난다.

```
class RepeatAddWithDelayUseCase {
    suspend fun add(repeatTime: Int): Int = withContext(Dispatchers.Default)
{
        var result = 0
        repeat(repeatTime) {
            delay(100L)
            result += 1
        }
        return@withContext result
    }
}
```

위 코드는 반복 때마다 100밀리초간 일시 중단 후 더하기를 반복한다.

```
class RepeatAddWithDelayUseCaseTest {
    @Test
    fun `runBlocking_100번 더하면 100이 반환된다`() = runBlocking {
        // Given
        val repeatAddUseCase = RepeatAddWithDelayUseCase()

        // When
        var result = 0
        result = repeatAddUseCase.add(100)

        // Then
        assertEquals(100, result)
    }
}
```

테스트는 정상적으로 통과하지만 테스트 하나를 실행하는 데 약 10초가 소요된다.
좋은 테스트를 작성하는 원칙 중 하나는 테스트에 걸리는 시간을 짧게 해서 테스트를 부담 없이 실행할 수 있도록 만드는 것이다.

12.3. 코루틴 테스트 라이브러리

앞서 보았듯이, 시간이 걸리는 작업이 포함된 일시 중담 함수를 테스트할 때 runBlocking 함수를 사용하면 시간이 오래 걸릴 수 있다. 이런 문제를 해결하기 위해 코루틴 테스트 라이브러리는 가상 시간을 사용하는 코루틴 스케줄러를 제공한다.

의존성 추가 : `testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.9.0")`

12.3.2. TestCoroutineScheduler 사용해 가상 시간에서 테스트 진행하기

가상 시간을 사용해 코루틴의 동작이 자신이 원하는 시간까지 단번에 진행될 수 있도록 만들어 빠르게 테스트를 완료할 수 있다. 이를 위해 테스트 라이브러리에서는 코루틴 스케줄러 TestCoroutineScheduler 객체를 제공.

12.3.2.1. advanceTimeBy 사용해 가상 시간 흐르게 만들기

advanceTimeBy() : 인자로 입력된 Long 타입의 값만큼 가상 시간이 밀리초 단위로 흐르게 하는 함수
currentTime : 흐른 가상 시간을 밀리초 단위로 반환하는 TestCoroutineScheduler 객체의 프로퍼티

```
@Test
fun `가상 시간 조절 테스트`() {
    // 테스트 환경 설정
    val testCoroutineScheduler = TestCoroutineScheduler()

    testCoroutineScheduler.advanceTimeBy(5000L) // 가상 시간에서 5초를 흐르게 만듦
: 현재 시간 5초
    assertEquals(5000L, testCoroutineScheduler.currentTime) // 현재 시간이 5초임
을 단언
    testCoroutineScheduler.advanceTimeBy(6000L) // 가상 시간에서 6초를 흐르게 만듦
: 현재 시간 11초
    assertEquals(11000L, testCoroutineScheduler.currentTime) // 현재 시간이 11초
임을 단언
    testCoroutineScheduler.advanceTimeBy(10000L) // 가상 시간에서 10초를 흐르게 만
듦 : 현재 시간 21초
    assertEquals(21000L, testCoroutineScheduler.currentTime) // 현재 시간이 21초
임을 단언
}
```

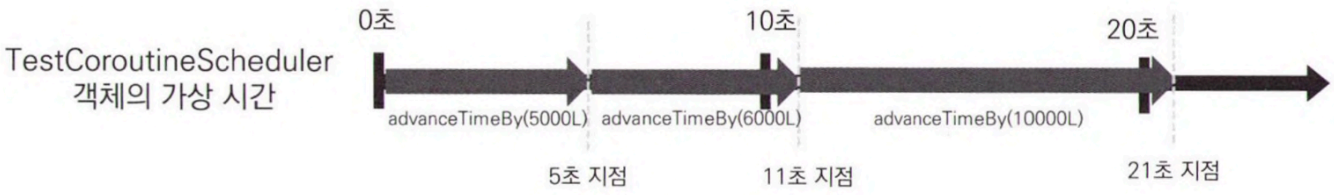

그림 12-13 TestCoroutineScheduler 객체의 가상 시간 흐르게 만들기

12.3.2.2. TestCoroutineScheduler와 StandardTestDispatcher 사용해 가상 시간 위에서 테스트 진행하기

가상 시간을 실질적으로 어떻게 사용할 수 있을까?

```
@Test
fun `가상 시간 위에서 테스트 진행`() {
    // 테스트 환경 설정
    val testCoroutineScheduler: TestCoroutineScheduler =
TestCoroutineScheduler()
    val testDispatcher: TestDispatcher = StandardTestDispatcher(scheduler =
testCoroutineScheduler)
    val testCoroutineScope = CoroutineScope(context = testDispatcher)

    // Given
    var result = 0

    // When
    testCoroutineScope.launch {
        delay(10000L) // 10초간 대기
        result = 1
        delay(10000L) // 10초간 대기
        result = 2
        println(Thread.currentThread().name)
    }

    // Then
    assertEquals(0, result)
    testCoroutineScheduler.advanceTimeBy(5000L) // 가상 시간에서 5초를 흐르게 만듦
: 현재 시간 5초
    assertEquals(0, result)
    testCoroutineScheduler.advanceTimeBy(6000L) // 가상 시간에서 6초를 흐르게 만듦
: 현재 시간 11초
    assertEquals(1, result)
    testCoroutineScheduler.advanceTimeBy(10000L) // 가상 시간에서 10초를 흐르게 만
듦 : 현재 시간 21초
    assertEquals(2, result)
}
```

위 코드에서 `testCoroutineScope`는 `testCoroutineScheduler`에 의해 시간이 관리되기 때문에 이 범위에서 실행되는 코루틴들은 가상 시간이 흐르지 않으면 실행되지 않는다. 가상 시간이 흐르게 하려면 아까와 같이 advanceTimeBy 함수를 호출해야 한다.

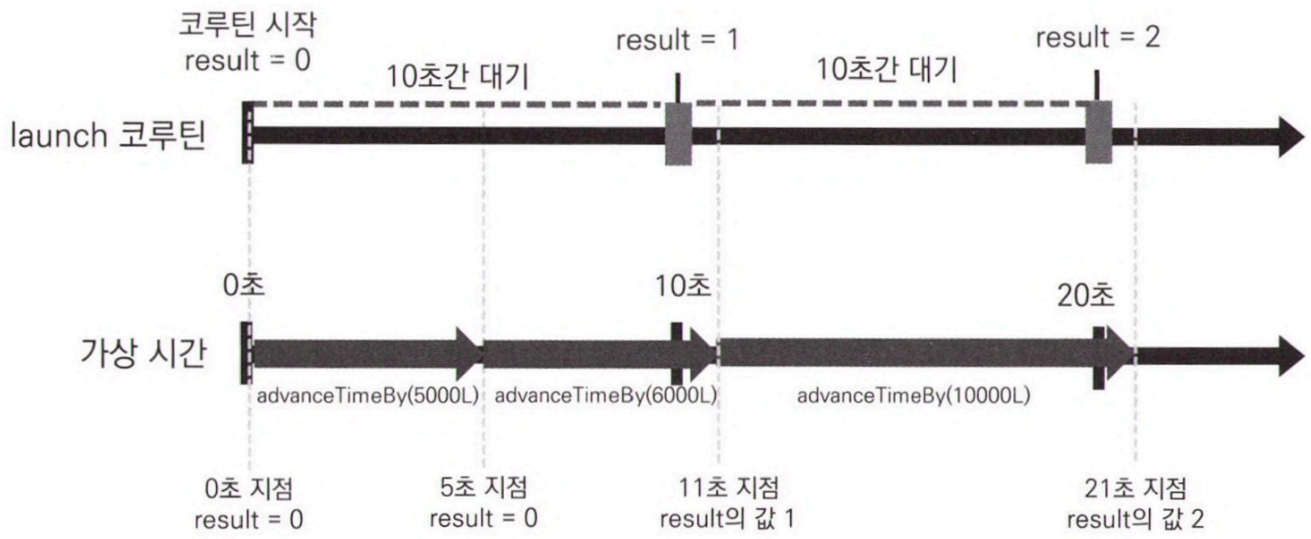

그림 12-14 가상 시간 흐르게 만들기

12.3.2.3. advanceUntilIdle 사용해 모든 코루틴 실행시키기

테스트가 제대로 실행되기 위해서는 테스트 대상 코드가 모두 실행되고 나서 단언(assert)이 실행돼야 한다. 이를 위해 TestCoroutineScheduler 객체는 이 객체를 사용하는 모든 디스패처와 연결된 작업이 모두 완료될 때까지 가상 시간을 흐르게 만드는 advanceUntilIdle 함수를 제공한다.

```
@Test
fun `advanceUntilIdle의 동작 살펴보기`() {
    // 테스트 환경 설정
    val testCoroutineScheduler: TestCoroutineScheduler =
TestCoroutineScheduler()
    val testDispatcher: TestDispatcher = StandardTestDispatcher(scheduler =
testCoroutineScheduler)
    val testCoroutineScope = CoroutineScope(context = testDispatcher)

    // Given
    var result = 0

    // When
    testCoroutineScope.launch {
        delay(10_000L) // 10초간 대기
        result = 1
        delay(10_000L) // 10초간 대기
        result = 2
    }
    testCoroutineScheduler.advanceUntilIdle() // testCoroutineScope 내부의 코루
틴이 모두 실행되게 만듦

    // Then
```

```
    assertEquals(2, result)
}
```

위 코드에서 advanceUntilIdle()이 호출되면 `testCoroutineScheduler`와 연결된 코루틴이 모두 실행 완료될 때까지 가상 시간이 흐른다. (여기서는 launch 코루틴 하나뿐)

12.3.3. TestCoroutineScheduler를 포함하는 StandardTestDispatcher

```
public fun StandardTestDispatcher(
    scheduler: TestCoroutineScheduler? = null,
    name: String? = null
): TestDispatcher = StandardTestDispatcherImpl(
    scheduler ?: TestMainDispatcher.currentTestScheduler ?:
TestCoroutineScheduler(), name)
```

StandardTestDispatcher 함수에는 기본적으로 TestCoroutineScheduler 객체를 생성하는 부분이 포함돼 있어 직접 생성할 필요가 없다.

```
@Test
fun `StandardTestDispatcher 사용하기`() {
    // 테스트 환경 설정
    val testDispatcher: TestDispatcher = StandardTestDispatcher()
    val testCoroutineScope = CoroutineScope(context = testDispatcher)

    // ...

    testDispatcher.scheduler.advanceUntilIdle() // testCoroutineScope 내부의
코루틴이 모두 실행되게 만듦
    assertEquals(2, result)
}
```

TestDispatcher 객체의 scheduler 프로퍼티를 통해 TestCoroutineSheduler 객체에 접근 가능하다.

12.3.4. TestScope 사용해 가상 시간에서 테스트 진행하기

```
@Test
fun `TestScope 사용하기`() {
    // 테스트 환경 설정
    val testCoroutineScope: TestScope = TestScope()

    // Given
    var result = 0
```

```
    // When
    testCoroutineScope.launch {
        delay(10000L) // 10초간 대기
        result = 1
        delay(10000L) // 10초간 대기
        result = 2
    }

    testCoroutineScope.advanceUntilIdle() // testCoroutineScope 내부의 코루틴이
모두 실행되게 만듦
    assertEquals(2, result)
}
```

코루틴 테스트 라이브러리는 TestScope에 대한 확장 함수를 통해 TestCoroutineScheduler 객체의 함수들(advanceTimeBy, advanceUntilIdle 등)과 프로퍼티(currentTime 등)를 TestScope 객체가 직접 호출할 수 있도록 만든다.

TestScope 함수는 기본적으로 StandardTestDispatcher 함수로 생성되는 TestDispatcher 객체를 포함한다.

```
public fun TestScope(context: CoroutineContext = EmptyCoroutineContext):
TestScope {
    val ctxWithDispatcher = context.withDelaySkipping()
    var scope: TestScopeImpl? = null
    val exceptionHandler = when
(ctxWithDispatcher[CoroutineExceptionHandler]) {
        null -> CoroutineExceptionHandler { _, exception ->
            scope!!.reportException(exception)
        }
        else -> throw IllegalArgumentException(
            "A CoroutineExceptionHandler was passed to TestScope. " +
                "Please pass it as an argument to a `launch` or `async`
block on an already-created scope " +
                "if uncaught exceptions require special treatment."
        )
    }
    return TestScopeImpl(ctxWithDispatcher + exceptionHandler).also { scope
= it }
}
```

```
internal fun CoroutineContext.withDelaySkipping(): CoroutineContext {
    val dispatcher: TestDispatcher = when (val dispatcher =
get(ContinuationInterceptor)) {
```

```
        is TestDispatcher -> {
            val ctxScheduler = get(TestCoroutineScheduler)
            if (ctxScheduler != null) {
                require(dispatcher.scheduler === ctxScheduler) {
                    "Both a TestCoroutineScheduler $ctxScheduler and
TestDispatcher $dispatcher linked to " +
                            "another scheduler were passed."
                }
            }
            dispatcher
        }
        null -> StandardTestDispatcher(get(TestCoroutineScheduler))
        else -> throw IllegalArgumentException("Dispatcher must implement
TestDispatcher: $dispatcher")
    }
    return this + dispatcher + dispatcher.scheduler
}
```

12.3.5. runTest 사용해 테스트 만들기

12.3.5.1. runTest 사용해 TestScope 대체하기

runTest 함수는 TestScope 객체를 사용해 코루틴을 실행시키고, 그 코루틴 내부에서 일시 중단 함수가 실행되더라도 작업이 곧바로 실행 완료될 수 있도록 가상시간을 흐르게 만드는 기능을 가진 코루틴 빌더이다.

```
@Test
fun `runTest 사용하기`() {
    // Given
    var result = 0

    // When
    runTest { // this: TestScope
        delay(10000L) // 10초간 대기
        result = 1
        delay(10000L) // 10초간 대기
        result = 2
    }

    // Then
    assertEquals(2, result)
}
```

테스트를 원하는 코드 블록을 runTest 함수의 람다식에서 실행하면, runBlocking 함수로 만들어지는 코루틴과 유사하게 동작하지만 지연되는 부분을 건너뛰는 코루틴이 만들어진다.

12.3.5.2. runTest로 테스트 전체 감싸기

```
@Test
fun `runTest로 테스트 감싸기`() = runTest {  // this: TestScope
    // Given
    var result = 0

    // When
    delay(10000L) // 10초간 대기
    result = 1
    delay(10000L) // 10초간 대기
    result = 2

    // Then
    assertEquals(2, result)
}
```

일반적으로는 위와 같이 runTest 함수를 사용해 테스트 전체를 감싼다.
모든 테스트를 runTest 함수로 감싸는 이유는 테스트 환경을 설정하는 부분, 결과를 비교하는 부분에서도 일시 중단 함수가 호출될 수 있기 때문이다.

12.3.5.3. runTest 함수의 람다식에서 TestScope 사용하기

runTest 함수는 람다식에서 TestScope 객체를 수신 객체로 갖기 때문에 this를 통해 TestScope 객체에 접근 가능하다. 따라서 TestScope 객체가 사용할 수 있는 확장 함수들과 확장 프로퍼티 모두 사용 가능하다.

```
@Test
fun `runTest에서 가상 시간 확인`() = runTest {  // this: TestScope
    delay(10000L) // 10초간 대기
    println("가상 시간: ${this.currentTime}ms") // 가상 시간: 10000ms
    delay(10000L) // 10초간 대기
    println("가상 시간: ${this.currentTime}ms") // 가상 시간: 20000ms
}
```

이렇게 일시 중단 시간 동안 가상 시간을 자동으로 흐르게 만드는 것이 바로 runTest 함수를 통해 생성되는 코루틴의 기능이다.

```
@Test
fun `runTest 내부에서 advanceUntilIdle 사용하기`() = runTest {  // this:
TestScope
    var result = 0
    launch {
        delay(1000L)
```

```
        result = 1
    }

    println("가상 시간: ${this.currentTime}ms, result = ${result}") // 가상 시간:
0ms, result = 0
    advanceUntilIdle()
    println("가상 시간: ${this.currentTime}ms, result = ${result}") // 가상 시간:
1000ms, result = 1
}
```

하지만 TestScope상에서 새로 실행된 launch 코루틴에 대해서는 자동으로 시간을 흐르게 하지 않는다. 이 경우에는 advanceUntilIdle()이 호출돼야 runTest 블록 내부에서 새로 생성된 코루틴들이 실행 완료될 때까지 시간이 흐른다.

```
@Test
fun `runTest 내부에서 join 사용하기`() = runTest {  // this: TestScope
    var result = 0
    launch {
        delay(1000L)
        result = 1
    }.join()

    println("가상 시간: ${this.currentTime}ms, result = ${result}") // 가상 시간:
1000ms, result = 1
}
```

만약 runTest 코루틴의 자식 코루틴에 대해 join을 호출하면 advanceUntilIdle()을 호출하지 않더라도 runTest 코루틴의 가상 시간이 흐른다. join 함수의 호출이 launch 코루틴이 실행완료될 때까지 runTest 코루틴을 일시 중단시키기 때문이다. (join 함수는 호출부의 코루틴을 일시 중단시킨다)
위에서도 설명했듯이, runTest 코루틴은 일시 중단 시간 동안 가상 시간을 자동으로 흐르게 만들기 때문에 runTest 코루틴의 가상 시간이 흐르는 것이다.

12.4. 코루틴 단위 테스트 만들어 보기

12.5. 코루틴 테스트 심화

12.5.1. 함수 내부에서 새로운 코루틴을 실행하는 객체에 대한 테스트

일시 중단 함수가 아닌 일반 함수 내부에서 새로운 코루틴을 실행하는 경우가 있다.

```
class StringStateHolder {
    private val coroutineScope = CoroutineScope(Dispatchers.IO)

    var stringState = ""
        private set

    fun updateStringWithDelay(string: String) {
        coroutineScope.launch {
            delay(1000L)
            stringState = string
        }
    }
}
```

```
class StringStateHolderTestFail {
    @Test
    fun `updateStringWithDelay(ABC)가 호출되면 문자열이 ABC로 변경된다`() = runTest
{
        // Given
        val stringStateHolder = StringStateHolder()

        // When
        stringStateHolder.updateStringWithDelay("ABC")

        // Then
        advanceUntilIdle()
        Assertions.assertEquals("ABC", stringStateHolder.stringState)
    }
}
```

위 테스트가 실패하는 이유는 StringStateHolder 객체 내부에 있는 CoroutineScope 객체에 있다.
이 CoroutineScope 객체는 별도의 루트 Job 객체를 갖기 때문에 runTest로 생성되는 코루틴과 구조화되지 않으며, 코루틴을 실행할 때 Dispatchers.IO를 사용하기 때문에 실제 시간 위에서 실행된다. (가상 시간 위에서 실행되려면 TestDispatcher를 사용해야 한다)
그래서 테스트용 스케줄러의 영향을 받지 않는다.

즉 assert가 실행될 때 stringState 값은 업데이트 되기 전의 빈 문자열이어서 테스트가 실패한다.

이를 해결하기 위해서는 StringStateHolder 객체의 CoroutineScope 객체가 TestCoroutineScheduler 객체를 사용할 수 있게 해야 한다.

```
class StringStateHolder(
    private val dispatcher: CoroutineDispatcher = Dispatchers.IO
```

```
) {
    private val coroutineScope = CoroutineScope(dispatcher)

    var stringState = ""
        private set

    fun updateStringWithDelay(string: String) {
        coroutineScope.launch {
            delay(1000L)
            stringState = string
        }
    }
}
```

```
class StringStateHolderTestSuccess {
  @Test
  fun `updateStringWithDelay(ABC)가 호출되면 문자열이 ABC로 변경된다`() {
    // Given
    val testDispatcher = StandardTestDispatcher()
    val stringStateHolder = StringStateHolder(
      dispatcher = testDispatcher
    )

    // When
    stringStateHolder.updateStringWithDelay("ABC")

    // Then
    testDispatcher.scheduler.advanceUntilIdle()
    Assertions.assertEquals("ABC", stringStateHolder.stringState)
  }
}
```

12.5.2. backgroundScope를 사용해 테스트 만들기

runTest 함수를 호출해 생성되는 코루틴은 내부의 모든 코루틴이 실행될 때까지 종료되지 않는다.
따라서 runTest 코루틴 내부에서 launch 함수가 호출돼 코루틴이 생성되고, 이 코루틴 내부에서 while문 같은 무한히 실행되는 작업이 실행된다면 테스트가 계속해서 실행된다.

```
class BackgroundScopeTest {
    @OptIn(ExperimentalCoroutinesApi::class)
    @Test
    fun `끝나지 않아 실패하는 테스트`() = runTest {
```

```
        var result = 0

        launch {
            while (true) {
                delay(1000L)
                result += 1
            }
        }

        advanceTimeBy(1500L)
        Assertions.assertEquals(1, result)
        advanceTimeBy(1000L)
        Assertions.assertEquals(2, result)
    }
}
```

이 테스트 코드가 실패하는 이유는 runTest 코루틴이 마지막 코드를 실행하고 나서 '실행 완료 중' 상태로 변경됐지만 while문에 의해 launch 코루틴이 계속해서 실행되어 테스트가 종료되지 않기 때문이다.
(실행 완료 중 : 부모 코루틴의 코드는 모두 실행되었지만 자식 코루틴이 아직 실행완료되지 않은 상태)

그래서 일정 시간 뒤에도 테스트가 종료되지 않으면 UncompletedCoroutinesError 예외를 던져 테스트를 실패하게 만든다.

이렇게 무한히 실행되는 작업을 테스트하기 위해서는 backgroundScope를 사용해야 한다.
backgroundScope는 runTest 코루틴의 모든 코드가 실행되면 자동으로 취소되며 테스트가 무한히 실행되는 것을 방지할 수 있다.

```
class BackgroundScopeTest {
    @Test
    fun `backgroundScope를 사용하는 테스트`() = runTest {
        var result = 0

        backgroundScope.launch {
            while (true) {
                delay(1000L)
                result += 1
            }
        }

        advanceTimeBy(1500L)
        Assertions.assertEquals(1, result)
        advanceTimeBy(1000L)
        Assertions.assertEquals(2, result)
```

```
    }
}
```

이 코드에서는 무한히 실행되는 launch 코루틴이 backgroundScope를 사용해 실행되며, 이 backgroundScope는 runTest 코루틴의 마지막 코드인 `Assertions.assertEquals(2, result)` 가 실행되면 취소된다.