JUnit 5 用户指南

本文档由 ThoughtWorks 咨询师 袁慎建、王亚鑫 与 何疆乐 倾情力作！正式版即将发布，敬请期待！

Original online document: JUnit 5 User Guide

1. 概述

本文档的目标是为那些编写测试、扩展作者和引擎作者以及构建工具和IDE供应商的程序员提供综合全面的参考。

本文档英文PDF格式 下载链接。

本文档中文PDF格式 预览版下载链接。

1.1. JUnit 5 是什么?

与以前版本的JUnit不同，JUnit5由几个不同的模块组成，它们分别来自于三个不同的子项目。

JUnit 5* = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage

JUnit Platform是在JVM上 启动测试框架 的基础平台。它还定义了TestEngine API，该API可用于开发运行在平台上的测试框架。此外，平台还提供了一个从命令行或者 Gradle 和 Maven 插件来启动平台的 控制台启动器 ，它就好比一个 基于JUnit 4的Runner 在平台上运行任何 `TestEngine` 。

JUnit Jupiter 是一个组合体，由在JUnit5中编写测试和扩展的新 编程模型 和 扩展模型 组成。另外，Jupiter子项目还提供了一个 `TestEngine` ，用于在平台上运行基于Jupiter的测试。

JUnit Vintage 提供了一个 `TestEngine` ，用于在平台上运行基于JUnit 3和JUnit 4 的测试。

1.2. 支持的Java版本

JUnit 5需要Java 8（或更高）的运行时环境。不过，你仍然可以测试那些经由老版本JDK编译的代码。

1.3. 获取帮助

与JUnit 5相关问题，可以在 Stack Overflow 进行提问，或者在 Gitter 上跟我们进行交流。

2. 安装

最终版本和里程碑的包已经被部署到Maven仓库中心了。

快照版本被部署到 Sonatype 快照库 中的 /org/junit目录下。

2.1. 依赖元数据

2.1.1. JUnit Platform

- **Group ID**: `org.junit.platform`
- **Version**: `1.0.2`
- **Artifact IDs:**

`junit-platform-commons`

These utilities are intended solely for usage within the JUnit framework itself. Any usage by external parties is not supported. Use at your own risk!

JUnit 内部通用类库/实用工具。这些实用工具仅用于JUnit框架本身。不支持任何外部使用。外部使用风险自负。

`junit-platform-console`

支持从控制台中查找和执行JUnit Platform上的测试。详情参考 控制台启动器。

`junit-platform-console-standalone`

一个包含了Maven仓库中的 junit-platform-console-standalone 目录下所有依赖项的可执行JAR包。详情参考 控制台启动器。

`junit-platform-engine`

测试引擎的公共API。详情参考 插入你自己的测试引擎

`junit-platform-gradle-plugin`

支持使用 Gralde 来查找和执行JUnit Platform上的测试。

`junit-platform-launcher`

配置和加载测试计划的公共API -- 典型的使用场景是IDE和构建工具。详情参考 JUnit Platform启动器API。

`junit-platform-runner`

在JUnit Platform上以JUnit 4的环境执行测试和测试套件的运行器。详情参考 使用JUnit 4 运行JUnit Platform。

`junit-platform-suite-api`

在JUnit Platform上配置测试套件的注解。被 JUnit Platform 运行器 所支持，也有可能被第三方的 `TestEngine` 实现所支持。

`junit-platform-surefire-provider`

支持使用 Maven Surefire 来查找和执行JUnit Platform上的测试。

2.1.2. JUnit Jupiter

- **Group ID**: `org.junit.jupiter`
- **Version**: `5.0.2`
- **Artifact IDs**:

`junit-jupiter-api`

编写测试 和 扩展 的JUnit Jupiter API。

`junit-jupiter-engine`

JUnit Jupiter测试引擎的实现，仅仅在运行时需要。

`junit-jupiter-params`

支持JUnit Jupiter中的 参数化的测试。

`junit-jupiter-migration-support`

支持从JUnit 4的迁移到JUnit Jupiter，仅仅在运行选择了JUnit 4规则的测试时需要。

2.1.3. JUnit Vintage

- **Group ID**: `org.junit.vintage`
- **Version**: `4.12.2`
- **Artifact ID**:

`junit-vintage-engine`

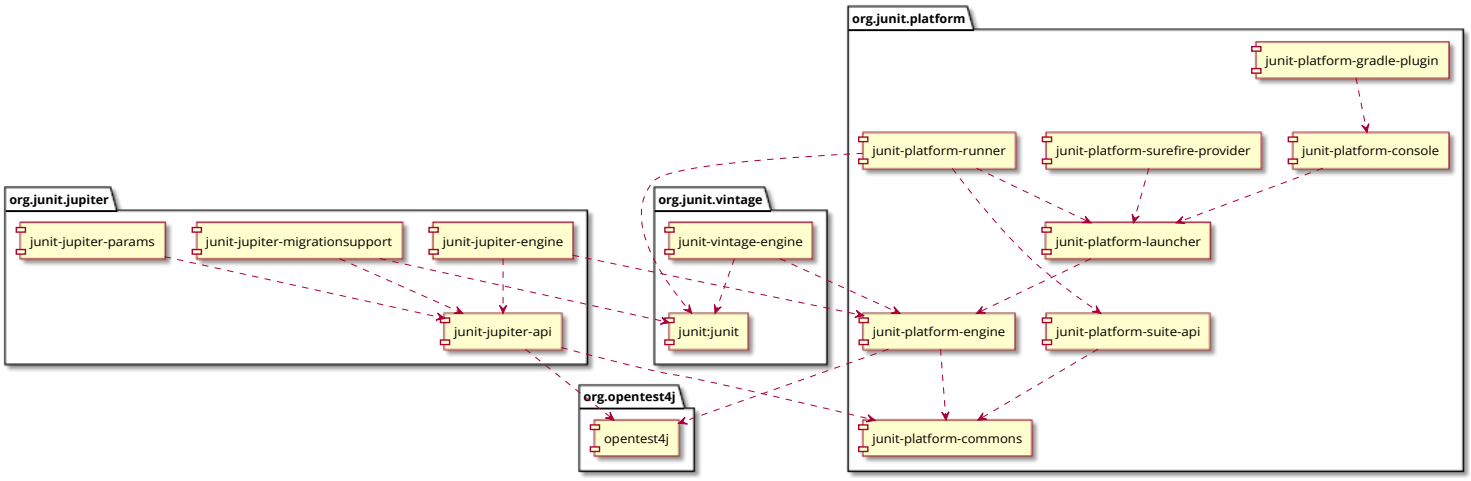
JUnit Vintage测试引擎实现，允许在新的JUnit Platform上运行低版本的JUnit测试，即那些以JUnit 3或JUnit 4风格编写的测试。

2.1.4. 可选的依赖

以上所有的包在它们已发布的Maven POM中都有一个可选的依赖，位于紧随其后的@API Guardian JAR包中。

- **Group ID**: org.apiguardian
- **Artifact ID**: apiguardian-api
- **Version**: 1.0.0

2.2. 依赖关系图



2.3 JUnit Jupiter示例工程

junit5-samples 代码库中包含了一系列基于JUnit Jupiter和JUnit Vintage的示例工程。你可以在下面的项目中找到相应的 `build.gradle` 和 `pom.xml` 文件：

- Gradle工程：junit5-gradle-consumer.
- Maven工程：junit5-maven-consumer.

3. 编写测试

第一个测试用例

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class FirstJUnit5Tests {

    @Test
    void myFirstTest() {
        assertEquals(2, 1 + 1);
    }

}
```

3.1. 注解

JUnit Jupiter 支持使用下面表格中的注解来配置测试和扩展框架。

所有的核心注解都位于 `junit-jupiter-api` 模块的 org.junit.jupiter.api 包中。

注解	描述
@Test	表示该方法是一个测试方法。与JUnit 4的 `@Test` 注解不同的是，它没有声明任何属性，因为JUnit Jupiter中的测试扩展是基于他们自己的专用注解来完成的。这样的方法会被*继承*，除非它们被*覆盖*了。
@ParameterizedTest	表示该方法是一个 参数化测试。这样的方法会被*继承*，除非它们被*覆盖*了。
@RepeatedTest	表示该方法是一个 重复测试 的测试模板。这样的方法会被*继承*，除非它们被*覆盖*了。
@TestFactory	表示该方法是一个 动态测试 的测试工厂。这样的方法会被*继承*，除非它们被*覆盖*了。
@TestInstance	用于配置所标注的测试类的 测试实例生命周期。这些注解会被*继承*。
@TestTemplate	表示该方法是一个 测试用例的模板，它会依据注册的 提供者所返回的的调用上下文的数量被多次调用。 这样的方法会被*继承*，除非它们被*覆盖*了。
@DisplayName	为测试类或测试方法声明一个定制化的展示名字。该注解不能被*继承*。
@BeforeEach	表示使用了该注解的方法应该在当前类中**每一个**使用了 `@Test` 、 `@RepeatedTest` 、 `@ParameterizedTest` 或者 `@TestFactory` 注解的方法*之前*执行；类似于JUnit 4的 `@Before` 。这样的方法会被*继承*，除非它们被*覆盖*了。
@AfterEach	表示使用了该注解的方法应该在当前类中**每一个**使用了 `@Test` 、 `@RepeatedTest` 、 `@ParameterizedTest` 或者 `@TestFactory` 注解的方法*之后*执行；类似于JUnit 4的 `@After` 。这样的方法会被*继承*，除非它们被*覆盖*了。
@BeforeAll	表示使用了该注解的方法应该在当前类中**所有**使用了 `@Test` 、 `@RepeatedTest` 、 `@ParameterizedTest` 或者 `@TestFactory` 注解的方法*之前*执行；类似于JUnit 4的 `@BeforeClass` 。这样的方法会被*继承*（除非它们被*隐藏* 或*覆盖*），并且它必须是 `static` 方法（除非"per-class" 测试实例声明周期 被使用）。
@AfterAll	表示使用了该注解的方法应该在当前类中所有使用了 `@Test` 、 `@RepeatedTest` 、 `@ParameterizedTest` 或者 `@TestFactory` 注解的方法之后执行；类似于JUnit 4的 `@AfterClass` 。这样的方法会被*继承*（除非它们被*隐藏* 或*覆盖*），并且它必须是 `static` 方法（除非"per-class" 测试实例声明周期 被使用）。
@Nested	表示使用了该注解的类是一个内嵌、非静态的测试类。 `@BeforeAll` 和 `@AfterAll` 方法不能直接在 `@Nested` 测试类中使用，（除非"per-class" 测试实例声明周期 被使用）。该注解不能被*继承*。
@Tag	用于声明过滤测试的*tags*，该注解可以用在方法或类上；类似于TesgNG的测试组或JUnit 4的分类。该注解能被*继承*，但仅限于类级别，而非方法级别。
@Disable	用于*禁用*一个测试类或测试方法；类似于JUnit 4的 `@Ignore` 。该注解不能被继承。
@ExtendWith	用于注册自定义扩展。该注解不能被*继承*。

被 `@Test` 、 `@TestTemplate` 、 `@RepeatedTest` 、 `@BeforeAll` 、 `@AfterAll` 、 `@BeforeEach` 或 `@AfterEach` 注解标注的方法不可以有返回值。

> ⚠️ 某些注解目前可能还处于实验阶段。详细信息请参阅 试验性APIs 中的表格。

3.1.1. 元注解和组合注解

JUnit Jupiter注解可以被用作*元注解*。这意味着你可以定义你自己的*组合注解*，而自定义的组合注解会自动*继承*其元注解的语义。

例如，为了避免在代码库中到处复制粘贴 `@Tag("fast")` （见 标记和过滤），你可以自定义一个名为 `@Fast` 的*组合注解*。然后你就可以用 `@Fast` 来替换 `@Tag("fast")` ，如下面代码所示。

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.junit.jupiter.api.Tag;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Tag("fast")
public @interface Fast {
}
```

3.2. 标准测试类

一个标准的测试用例

```
import static org.junit.jupiter.api.Assertions.fail;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class StandardTests {

    @BeforeAll
    static void initAll() {
    }

    @BeforeEach
    void init() {
    }

    @Test
    void succeedingTest() {
    }

    @Test
    void failingTest() {
        fail("a failing test");
    }

    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }

    @AfterEach
    void tearDown() {
    }

    @AfterAll
    static void tearDownAll() {
    }

}
```

> 不必将测试类和测试方法声明为 `public`

3.3. 显示名称

测试类和测试方法可以声明自定义的显示名称 -- 空格、特殊字符甚至是emojis表情 -- 都可以显示在测试运行器和测试报告中。

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case")
class DisplayNameDemo {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("╯°□°）╯")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

    @Test
    @DisplayName("😱")
    void testWithDisplayNameContainingEmoji() {
    }

}
```

3.4. 断言

JUnit Jupiter沿用了很多JUnit 4就已经存在的断言方法，并且增加了一些适合与Java8 Lambda一起使用的断言。所有的JUnit Jupiter断言都是 org.junit.jupiter.Assertions 类中 `static` 方法。

```
import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.assertAll;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTimeout;
import static org.junit.jupiter.api.Assertions.assertTimeoutPreemptively;
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.Test;

class AssertionsDemo {

    @Test
    void standardAssertions() {
        assertEquals(2, 2);
        assertEquals(4, 4, "The optional assertion message is now the last parameter.");
        assertTrue(2 == 2, () -> "Assertion messages can be lazily evaluated -- "
                + "to avoid constructing complex messages unnecessarily.");
    }

    @Test
    void groupedAssertions() {
        // In a grouped assertion all assertions are executed, and any
        // failures will be reported together.
        assertAll("person",
            () -> assertEquals("John", person.getFirstName()),
            () -> assertEquals("Doe", person.getLastName())
        );
    }

    @Test
    void dependentAssertions() {
        // Within a code block, if an assertion fails the
        // subsequent code in the same block will be skipped.
        assertAll("properties",
            () -> {
                String firstName = person.getFirstName();
                assertNotNull(firstName);

                // Executed only if the previous assertion is valid.
                assertAll("first name",
                    () -> assertTrue(firstName.startsWith("J")),
                    () -> assertTrue(firstName.endsWith("n"))
                );
            },
            () -> {
                // Grouped assertion, so processed independently
                // of results of first name assertions.
                String lastName = person.getLastName();
                assertNotNull(lastName);

                // Executed only if the previous assertion is valid.
                assertAll("last name",
                    () -> assertTrue(lastName.startsWith("D")),
                    () -> assertTrue(lastName.endsWith("e"))
                );
            }
        );
    }

    @Test
    void exceptionTesting() {
        Throwable exception = assertThrows(IllegalArgumentException.class, () -> {
            throw new IllegalArgumentException("a message");
        });
        assertEquals("a message", exception.getMessage());
    }
```

```
    @Test
    void timeoutNotExceeded() {
        // The following assertion succeeds.
        assertTimeout(ofMinutes(2), () -> {
            // Perform task that takes less than 2 minutes.
        });
    }

    @Test
    void timeoutNotExceededWithResult() {
        // The following assertion succeeds, and returns the supplied object.
        String actualResult = assertTimeout(ofMinutes(2), () -> {
            return "a result";
        });
        assertEquals("a result", actualResult);
    }

    @Test
    void timeoutNotExceededWithMethod() {
        // The following assertion invokes a method reference and returns an object.
        String actualGreeting = assertTimeout(ofMinutes(2), AssertionsDemo::greeting);
        assertEquals("hello world!", actualGreeting);
    }

    @Test
    void timeoutExceeded() {
        // The following assertion fails with an error message similar to:
        // execution exceeded timeout of 10 ms by 91 ms
        assertTimeout(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        });
    }

    @Test
    void timeoutExceededWithPreemptiveTermination() {
        // The following assertion fails with an error message similar to:
        // execution timed out after 10 ms
        assertTimeoutPreemptively(ofMillis(10), () -> {
            // Simulate task that takes more than 10 ms.
            Thread.sleep(100);
        });
    }

    private static String greeting() {
        return "hello world!";
    }

}
```

3.4.1. 第三方断言类库

虽然JUnit Jupiter提供的断言工具包已经满足了很多测试场景，但有时候我们会遇到需要更加强大且具备例如*匹配器*功能的场景。在这些场景中，JUnit团队推荐使用第三方断言类库，例如：AssertJ、Hamcrest、Truth 等等。所以说，使用哪个断言类库完全取决于开发人员自己的喜好。

举个例子，*匹配器*和一个流式调用的API的组合可以使得断言更加具有描述性和可读性。然而，JUnit Jupiter的 org.junit.jupiter.Assertions 类没有提供一个类似于JUnit 4的 `org.junit.Assert` 类中 assertThat() 方法，该方法接受一个Hamcrest Matcher。所以，我们鼓励开发人员使用由第三方断言库提供的匹配器的内置支持。

下面的例子演示如何在JUnit Jupiter中使用Hamcrest提供的 `assertThat()` 。只要Hamcrest库已经被添加到classpath中，你就可以静态导入诸如 `assertThat()` 、 `is()` 以及 `equalTo()` 方法，然后在测试方法中使用它们，如下面代码所示的 `assertWithHamcrestMatcher()` 方法。

```
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;

class HamcrestAssertionDemo {

    @Test
    void assertWithHamcrestMatcher() {
        assertThat(2 + 1, is(equalTo(3)));
    }

}
```

当然，那些基于JUnit 4编程模型的遗留测试可以继续使用 `org.junit.Assert#assertThat` 。

3.5. 假设

JUnit Jupiter附带了JUnit 4所提供的假设方法的一个子集，并增加了一些适合与Java 8 lambda一起使用的假设方法。所有的JUnit Jupiter假设都是 org.junit.jupiter.Assumptions 类中的静态方法。

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;

import org.junit.jupiter.api.Test;

class AssumptionsDemo {

    @Test
    void testOnlyOnCiServer() {
        assumeTrue("CI".equals(System.getenv("ENV")));
        // remainder of test
    }

    @Test
    void testOnlyOnDeveloperWorkstation() {
        assumeTrue("DEV".equals(System.getenv("ENV")),
            () -> "Aborting test: not on developer workstation");
        // remainder of test
    }

    @Test
    void testInAllEnvironments() {
        assumingThat("CI".equals(System.getenv("ENV")),
            () -> {
                // perform these assertions only on the CI server
                assertEquals(2, 2);
            });

        // perform these assertions in all environments
        assertEquals("a string", "a string");
    }

}
```

3.6. 禁用测试

下面是一个被禁用的测试用例。

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

@Disabled
class DisabledClassDemo {
    @Test
    void testWillBeSkipped() {
    }
}
```

下面是一个包含被禁用测试方法的测试用例。

```
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class DisabledTestsDemo {

    @Disabled
    @Test
    void testWillBeSkipped() {
    }

    @Test
    void testWillBeExecuted() {
    }
}
```

3.7. 标记和过滤

测试类和测试方法可以被标记。那些标签可以在后面被用来过滤 测试发现和执行。

3.7.1. 标记的语法规则

- 标记不能为 `null` 或空。
- *trimmed* 的标记不能包含空格。
- *trimmed* 的标记不能包含IOS字符。
- *trimmed* 的标记不能包含一下*保留*字符。
    - `,` , `(` , `)` , `&` , `|` , `!`

> 上述的"trimmed"指的是两端的空格字符被去除掉。

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("fast")
@Tag("model")
class TaggingDemo {

    @Test
    @Tag("taxes")
    void testingTaxCalculation() {
    }

}
```

3.8. 测试实例生命周期

为了隔离地执行单个测试方法，以及避免由于不稳定的测试实例状态引发的非预期的副作用，JUnit会在执行每个测试方法执行之前创建一个新的实例（参考下面的注释说明如何定义一个*测试*方法）。这个"per-method"测试实例生命周期是JUnit Jupiter的默认行为，这点类似于JUnit以前的所有版本。

如果你希望JUnit Jupiter在同一个实例上执行所有的测试方法，在你的测试类上加上注解 `@TestInstance(Lifecycle.PER_CLASS)` 即可。启用了该模式后，每一个测试类只会创建一次实例。因此，如果你的测试方法依赖实例变量存储的状态，你可能需要在 `@BeforeEach` 或 `@AfterEach` 方法中重置状态。

"per-class"模式相比于默认的"per-method"模式有一些额外的好处。具体来说，使用了"per-class"模式之后，你就可以在非静态方法和接口 `default` 方法上声明 `@BeforeAll` 和 `@AfterAll` ，就像接口的默认方法一样。因此"per-class"模式使得在 `@Nested` 测试类中使用 `@BeforeAll` 和 `@AfterAll` 注解成为了可能。

如果你使用Kotlin编程语言来编写测试，你会发现通过将测试实例的生命周期模式切换到"per-class"更容易实现 `@BeforeAll` 和 `@AfterAll` 方法。

> 在测试实例生命周期的上下文中，任何使用了 `@Test` , `@RepeatedTest` , `@ParameterizedTest` , `@TestFactory` , 或者 `@TestTemplate` 注解的方法都是一个*测试*方法。

3.8.1. 更改默认的测试实例生命周期

如果测试类或测试接口没有用 `@TestInstance` 标注，JUnit Jupiter 将使用*默认*的生命周期模式。标准的*默认*模式是 `PER_METHOD` 。然而，整个测试计划执行的*默认值*是可以被更改的。要更改默认测试实例生命周期模式，只需将 `junit.jupiter.testinstance.lifecycle.default` *配置参数*设置为在 `TestInstance.Lifecycle` 中定义的枚举常量的名称即可，名称忽略大小写。它也通过JVM系统属性提供，作为一个传递给 `Launcher` 的 `LauncherDiscoveryRequest` 中的*配置参数*，或通过JUnit Platform配置文件提供（详细信息请参阅配置参数）。

例如，要将默认测试实例生命周期模式设置为 `Lifecycle.PER_CLASS` ，你可以使用以下系统属性启动JVM。

`-Djunit.jupiter.testinstance.lifecycle.default=per_class`

但是请注意，通过JUnit Platform配置文件设置缺省测试实例生命周期模式是一个更强大的解决方案，因为配置文件可以与项目一起被提交到版本控制系统中，因此可用于IDE和构建软件。

要通过JUnit Platform配置文件将默认测试实例生命周期模式设置为 `Lifecycle.PER_CLASS` ，请在类路径的根目录（例如， `src/test/resources` ）中创建一个名为 `junit-platform.properties` 的文件，并写入以下内容。

`junit.jupiter.testinstance.lifecycle.default = per_class`

> ⚠️ 更改*默认*的测试实例生命周期模式后，如果没有做到一致地应用，将会导致不可预测的结果和脆弱的构建。例如，如果构建将"per-class"语义配置为默认值，但是IDE中的测试使用"per-method"的语义来执行，则可能使调试构建服务器上发生的错误变得困难。因此，建议更改JUnit Platform配置文件中的默认值，而不是通过JVM系统属性。

3.9. 嵌套测试

嵌套测试使得测试编写者能够表示出几组测试用例之间的关系。下面来看一个精心设计的例子。

一个用于测试栈的嵌套测试套件

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.EmptyStackException;
import java.util.Stack;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

@DisplayName("A stack")
class TestingAStackDemo {

    Stack<Object> stack;

    @Test
    @DisplayName("is instantiated with new Stack()")
    void isInstantiatedWithNew() {
        new Stack<>();
    }

    @Nested
    @DisplayName("when new")
    class WhenNew {

        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        @DisplayName("is empty")
        void isEmpty() {
            assertTrue(stack.isEmpty());
        }

        @Test
        @DisplayName("throws EmptyStackException when popped")
        void throwsExceptionWhenPopped() {
            assertThrows(EmptyStackException.class, () -> stack.pop());
        }

        @Test
        @DisplayName("throws EmptyStackException when peeked")
        void throwsExceptionWhenPeeked() {
            assertThrows(EmptyStackException.class, () -> stack.peek());
        }

        @Nested
        @DisplayName("after pushing an element")
        class AfterPushing {

            String anElement = "an element";
```

```
            @BeforeEach
            void pushAnElement() {
                stack.push(anElement);
            }

            @Test
            @DisplayName("it is no longer empty")
            void isNotEmpty() {
                assertFalse(stack.isEmpty());
            }

            @Test
            @DisplayName("returns the element when popped and is empty")
            void returnElementWhenPopped() {
                assertEquals(anElement, stack.pop());
                assertTrue(stack.isEmpty());
            }

            @Test
            @DisplayName("returns the element when peeked but remains not empty")
            void returnElementWhenPeeked() {
                assertEquals(anElement, stack.peek());
                assertFalse(stack.isEmpty());
            }
        }
    }
}
```

> 只有*非静态嵌套类*（即内部类）可以作为@Nested测试类。嵌套可以是任意深的，这些内部类被认为是测试类家族的正式成员，但有一个例外：`@BeforeAll` 和 `@AfterAll` 方法*默认*不会工作。原因是Java不允许内部类中存在 `static` 成员。不过这种限制可以使用 `@TestInstance(Lifecycle.PER_CLASS)` 标注 `@Nested` 测试类来绕开（请参阅 测试实例生命周期）

3.10. 构造函数和方法的依赖注入

在之前的所有JUnit版本中，测试构造函数和方法是不允许传入参数的（至少不能使用标准的 `Runner` 实现）。JUnit Jupiter一个主要的改变是：允许给测试类的构造函数和方法传入参数。这带来了更大的灵活性，并且可以在构造函数和方法上使用*依赖注入*。

ParameterResolver 为测试扩展定义了API，它可以在运行时*动态*解析参数。如果一个测试的构造函数或者 `@Test` 、 `@TestFactory` 、 `@BeforeEach` 、 `@AfterEach` 、 `@BeforeAll` 或者 `@AfterAll` 方法接收一个参数，这个参数就必须在运行时被一个已注册的 `ParameterResolver` 解析。

目前有三种被自动注册的内建的解析器。

- TestInfoParameterResolver：如果一个方法参数的类型是 TestInfo， `TestInfoParameterResolver` 将根据当前的测试提供一个 `TestInfo` 的实例用于填充参数的值。然后， `TestInfo` 就可以被用来检索关于当前测试的信息，例如：显示名称、测试类、测试方法或相关的标签。显示名称要么是一个类似于测试类或测试方法的技术名称，要么是一个通过 `@DisplayName` 配置的自定义名称。

  TestInfo就像JUnit 4规则中 `TestName` 规则的代替者。以下演示如何将 `TestInfo` 注入到测试构造函数， `@BeforeEach` 方法和 `@Test` 方法中。

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInfo;

@DisplayName("TestInfo Demo")
class TestInfoDemo {

    TestInfoDemo(TestInfo testInfo) {
        assertEquals("TestInfo Demo", testInfo.getDisplayName());
    }

    @BeforeEach
    void init(TestInfo testInfo) {
        String displayName = testInfo.getDisplayName();
        assertTrue(displayName.equals("TEST 1") || displayName.equals("test2()"));
    }

    @Test
    @DisplayName("TEST 1")
    @Tag("my-tag")
    void test1(TestInfo testInfo) {
        assertEquals("TEST 1", testInfo.getDisplayName());
        assertTrue(testInfo.getTags().contains("my-tag"));
    }

    @Test
    void test2() {
    }

}
```

- `RepetitionInfoParameterResolver`：如果一个位于 `@RepeatedTest`、`@BeforeEach` 或者 `@AfterEach` 方法的参数的类型是 RepetitionInfo，`RepetitionInfoParameterResolver` 会提供一个 `RepetitionInfo` 实例。然后，`RepetitionInfo` 就可以被用来检索对应 `@RepeatedTest` 方法的当前重复以及总重复次数等相关信息。但是请注意，`RepetitionInfoParameterResolver` 不是在 `@RepeatedTest` 的上下文之外被注册的。请参阅重复测试示例
- TestReporterParameterResolver：如果一个方法参数的类型是 TestReporter，`TestReporterParameterResolver` 会提供一个 `TestReporter` 实例。然后，`TestReporter` 就可以被用来发布有关当前测试运行的其他数据。这些数据可以通过 TestExecutionListener 的 `reportingEntryPublished()` 方法来消费，因此可以被IDE查看或包含在报告中。

  在JUnit Jupiter中，你应该使用 `TestReporter` 来代替你在JUnit 4中打印信息到 `stdout` 或 `stderr` 的习惯。使用 `@RunWith(JUnitPlatform.class)` 会将报告的所有条目都输出到 `stdout` 中。

```
import java.util.HashMap;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestReporter;

class TestReporterDemo {

    @Test
    void reportSingleValue(TestReporter testReporter) {
        testReporter.publishEntry("a key", "a value");
    }

    @Test
    void reportSeveralValues(TestReporter testReporter) {
        HashMap<String, String> values = new HashMap<>();
        values.put("user name", "dk38");
        values.put("award year", "1974");

        testReporter.publishEntry(values);
    }

}
```

> 其他的参数解析器必须通过 `@ExtendWith` 注册合适的 扩展 来明确地开启。

可以查看 MockitoExtension 获取自定义 ParameterResolver 的示例。虽然并不打算大量使用它，但它演示了扩展模型和参数解决过程中的简单性和表现力。`MyMockitoTest` 演示了如何将Mockito mocks注入到 `@BeforeEach` 和 `@Test` 方法中。

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import com.example.Person;
import com.example.mockito.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class MyMockitoTest {

    @BeforeEach
    void init(@Mock Person person) {
        when(person.getName()).thenReturn("Dilbert");
    }

    @Test
    void simpleTestWithInjectedMock(@Mock Person person) {
        assertEquals("Dilbert", person.getName());
    }

}
```

3.11. 测试接口和默认方法

JUnit Jupiter允许将 `@Test` 、 `@RepeatedTest` 、 `@ParameterizedTest` 、 `@TestFactory` 、 `TestTemplate` 、 `@BeforeEach` 和 `@AfterEach` 注解声明在接口的 `default` 方法上。*如果* 测试接口或测试类使用了 `@TestInstance(Lifecycle.PER_CLASS)` 注解（请参阅 测试实例生命周期），则可以在测试接口中的 `static` 方法或接口的 `default` 方法上声明 `@BeforeAll` 和 `@AfterAll` 。下面来看一些例子。

```
@TestInstance(Lifecycle.PER_CLASS)
interface TestLifecycleLogger {

    static final Logger LOG = Logger.getLogger(TestLifecycleLogger.class.getName());

    @BeforeAll
    default void beforeAllTests() {
        LOG.info("Before all tests");
    }

    @AfterAll
    default void afterAllTests() {
        LOG.info("After all tests");
    }

    @BeforeEach
    default void beforeEachTest(TestInfo testInfo) {
        LOG.info(() -> String.format("About to execute [%s]",
            testInfo.getDisplayName()));
    }

    @AfterEach
    default void afterEachTest(TestInfo testInfo) {
        LOG.info(() -> String.format("Finished executing [%s]",
            testInfo.getDisplayName()));
    }

}
```

```
interface TestInterfaceDynamicTestsDemo {

    @TestFactory
    default Collection<DynamicTest> dynamicTestsFromCollection() {
        return Arrays.asList(
            dynamicTest("1st dynamic test in test interface", () -> assertTrue(true)),
            dynamicTest("2nd dynamic test in test interface", () -> assertEquals(4, 2 * 2))
        );
    }

}
```

可以在测试接口上声明 `@ExtendWith` 和 `@Tag`，以便实现该接口的类自动继承它的标记和扩展。请参阅 测试执行回调之前和之后 章节的 TimingExtension 源代码。

```
@Tag("timed")
@ExtendWith(TimingExtension.class)
interface TimeExecutionLogger {
}
```

在你的测试类中，你可以通过实现这些测试接口来获取那些配置信息。

```
class TestInterfaceDemo implements TestLifecycleLogger,
        TimeExecutionLogger, TestInterfaceDynamicTestsDemo {

    @Test
    void isEqualValue() {
        assertEquals(1, 1, "is always equal");
    }

}
```

运行 `TestInterfaceDemo`，你会看到类似于如下的输出：

```
:junitPlatformTest
INFO  example.TestLifecycleLogger - Before all tests
INFO  example.TestLifecycleLogger - About to execute [dynamicTestsFromCollection()]
INFO  example.TimingExtension - Method [dynamicTestsFromCollection] took 13 ms.
INFO  example.TestLifecycleLogger - Finished executing [dynamicTestsFromCollection()]
INFO  example.TestLifecycleLogger - About to execute [isEqualValue()]
INFO  example.TimingExtension - Method [isEqualValue] took 1 ms.
INFO  example.TestLifecycleLogger - Finished executing [isEqualValue()]
INFO  example.TestLifecycleLogger - After all tests

Test run finished after 190 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         3 tests found           ]
[         0 tests skipped         ]
[         3 tests started         ]
[         0 tests aborted         ]
[         3 tests successful      ]
[         0 tests failed          ]

BUILD SUCCESSFUL
```

此功能的另一个可能的应用是编写接口合同的测试。例如，你可以编写测试，以了解 `Object.equals` 或 `Comparable.compareTo` 的实现应该如何执行。

```
public interface Testable<T> {

    T createValue();

}
```

```java
public interface EqualsContract<T> extends Testable<T> {

    T createNotEqualValue();

    @Test
    default void valueEqualsItself() {
        T value = createValue();
        assertEquals(value, value);
    }

    @Test
    default void valueDoesNotEqualNull() {
        T value = createValue();
        assertFalse(value.equals(null));
    }

    @Test
    default void valueDoesNotEqualDifferentValue() {
        T value = createValue();
        T differentValue = createNotEqualValue();
        assertNotEquals(value, differentValue);
        assertNotEquals(differentValue, value);
    }

}
```

```java
public interface ComparableContract<T extends Comparable<T>> extends Testable<T> {

    T createSmallerValue();

    @Test
    default void returnsZeroWhenComparedToItself() {
        T value = createValue();
        assertEquals(0, value.compareTo(value));
    }

    @Test
    default void returnsPositiveNumberComparedToSmallerValue() {
        T value = createValue();
        T smallerValue = createSmallerValue();
        assertTrue(value.compareTo(smallerValue) > 0);
    }

    @Test
    default void returnsNegativeNumberComparedToSmallerValue() {
        T value = createValue();
        T smallerValue = createSmallerValue();
        assertTrue(smallerValue.compareTo(value) < 0);
    }

}
```

在你的测试类中，你可以实现两个契约接口，从而继承相应的测试。当然，你还得实现那些抽象方法。

```java
class StringTests implements ComparableContract<String>, EqualsContract<String> {

    @Override
    public String createValue() {
        return "foo";
    }

    @Override
    public String createSmallerValue() {
        return "bar"; // 'b' < 'f' in "foo"
    }

    @Override
    public String createNotEqualValue() {
        return "baz";
    }

}
```

> 📒 上述测试仅仅作为例子，因此它们是不完整的。

3.12. 重复测试

在JUnit Jupiter中，我们可以通过 `@RepeatedTest` 注解并指定所需的重复次数来重复运行一个测试方法。每个重复测试的调用都像执行常规的 `@Test` 方法一样，完全支持相同的生命周期回调和扩展。

下面示例演示了如何声明名一个为 `repeatedTest()` 的测试，该测试将自动重复10次。

```
@RepeatedTest(10)
void repeatedTest() {
    // ...
}
```

除了指定重复次数之外，我们还可以通过 `@RepeatedTest` 注解的 `name` 属性为每次重复配置自定义的显示名称。此外，显示名称可以是一个由静态文本和动态占位符的组合组成的模式。目前支持以下占位符。

- `{displayName}` : `@RepeatedTest` 方法的显示名称。
- `{currentRepetition}` : 当前的重复次数。
- `{totalRepetitions}` : 总的重复次数。

给定重复的默认显示名称基于以下模式生成： `"repetition {currentRepetition} of {totalRepetitions}"` 。因此，之前的 `repeatTest()` 例子的单个重复的显示名称将是： `repetition 1 of 10, repetition 2 of 10` ，等等。如果你希望每个重复的名称中包含 `@RepeatedTest` 方法的显示名称，你可以自定义自己的模式或使用预定义的 `RepeatedTest.LONG_DISPLAY_NAME` 。后者等同于 `"{displayName} :: repetition {currentRepetition} of {totalRepetitions}"` ，在这种模式下， `repeatedTest()` 方法单次重复的显示名称长成这样： `repeatedTest() :: repetition 1 of 10, repeatedTest() :: repetition 2 of 10` ，等等。

为了以编程方式获取有关当前重复和总重复次数的信息，开发人员可以选择将一个 `RepetitionInfo` 的实例注入到 `@RepeatedTest` ， `@BeforeEach` 或 `@AfterEach` 方法中。

3.12.1. 重复测试的例子

本节末尾的 `RepeatedTestsDemo` 类将演示重复测试的几个示例

`repeatedTest()` 方法与上一节中的示例相同;而 `repeatedTestWithRepetitionInfo()` 演示了如何将 `RepetitionInfo` 实例注入到测试中，从而获取当前重复测试的总重复次数。

接下来的两个方法演示了如何在每个重复的显示名称中包含 `@RepeatedTest` 方法的自定义 `@DisplayName` 。 `customDisplayName()` 将自定义显示名称与自定义模式组合在一起，然后使用 `TestInfo` 来验证生成的显示名称的格式。 `Repeat! 是来自 @DisplayName` 中声明的 `{displayName}` ， `1/1` 来自 `{currentRepetition}/{totalRepetitions}` 。而 `customDisplayNameWithLongPattern()` 使用了上述预定义的 `RepeatedTest.LONG_DISPLAY_NAME` 模式。

`repeatedTestInGerman()` 演示了将重复测试的显示名称翻译成外语的能力 - 比如例子中的德语，所以结果看起来像： `Wiederholung 1 von 5, Wiederholung 2 von 5` ，等等。

由于 `beforeEach()` 方法使用了 `@BeforeEach` 注解，所以在每次重复测试之前都会执行它。通过往方法中注入 `TestInfo` 和 `RepetitionInfo` ，我们可以看到有可能获得有关当前正在执行的重复测试的信息。启用 `INFO` 的日志级别，执行 `RepeatedTestsDemo` 可以看到如下的输出：

```
INFO: About to execute repetition 1 of 10 for repeatedTest
INFO: About to execute repetition 2 of 10 for repeatedTest
INFO: About to execute repetition 3 of 10 for repeatedTest
INFO: About to execute repetition 4 of 10 for repeatedTest
INFO: About to execute repetition 5 of 10 for repeatedTest
INFO: About to execute repetition 6 of 10 for repeatedTest
INFO: About to execute repetition 7 of 10 for repeatedTest
INFO: About to execute repetition 8 of 10 for repeatedTest
INFO: About to execute repetition 9 of 10 for repeatedTest
INFO: About to execute repetition 10 of 10 for repeatedTest
INFO: About to execute repetition 1 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 2 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 3 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 4 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 5 of 5 for repeatedTestWithRepetitionInfo
INFO: About to execute repetition 1 of 1 for customDisplayName
INFO: About to execute repetition 1 of 1 for customDisplayNameWithLongPattern
INFO: About to execute repetition 1 of 5 for repeatedTestInGerman
INFO: About to execute repetition 2 of 5 for repeatedTestInGerman
INFO: About to execute repetition 3 of 5 for repeatedTestInGerman
INFO: About to execute repetition 4 of 5 for repeatedTestInGerman
INFO: About to execute repetition 5 of 5 for repeatedTestInGerman
```

```java
import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.logging.Logger;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.RepetitionInfo;
import org.junit.jupiter.api.TestInfo;

class RepeatedTestsDemo {

    private Logger logger = // ...

    @BeforeEach
    void beforeEach(TestInfo testInfo, RepetitionInfo repetitionInfo) {
        int currentRepetition = repetitionInfo.getCurrentRepetition();
        int totalRepetitions = repetitionInfo.getTotalRepetitions();
        String methodName = testInfo.getTestMethod().get().getName();
        logger.info(String.format("About to execute repetition %d of %d for %s", //
            currentRepetition, totalRepetitions, methodName));
    }

    @RepeatedTest(10)
    void repeatedTest() {
        // ...
    }

    @RepeatedTest(5)
    void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
        assertEquals(5, repetitionInfo.getTotalRepetitions());
    }

    @RepeatedTest(value = 1, name = "{displayName} {currentRepetition}/{totalRepetitions}")
    @DisplayName("Repeat!")
    void customDisplayName(TestInfo testInfo) {
        assertEquals(testInfo.getDisplayName(), "Repeat! 1/1");
    }

    @RepeatedTest(value = 1, name = RepeatedTest.LONG_DISPLAY_NAME)
    @DisplayName("Details...")
    void customDisplayNameWithLongPattern(TestInfo testInfo) {
        assertEquals(testInfo.getDisplayName(), "Details... :: repetition 1 of 1");
    }

    @RepeatedTest(value = 5, name = "Wiederholung {currentRepetition} von {totalRepetitions}")
    void repeatedTestInGerman() {
        // ...
    }

}
```

在启用了unicode主题的情况下使用 `ConsoleLauncher` 或 `junitPlatformTest` Gradle插件时，执行 `RepeatedTestsDemo` ，在控制台你会看到如下输出：

```
├─ RepeatedTestsDemo ✔
|  ├─ repeatedTest() ✔
|  |  ├─ repetition 1 of 10 ✔
|  |  ├─ repetition 2 of 10 ✔
|  |  ├─ repetition 3 of 10 ✔
|  |  ├─ repetition 4 of 10 ✔
|  |  ├─ repetition 5 of 10 ✔
|  |  ├─ repetition 6 of 10 ✔
|  |  ├─ repetition 7 of 10 ✔
|  |  ├─ repetition 8 of 10 ✔
|  |  ├─ repetition 9 of 10 ✔
|  |  └─ repetition 10 of 10 ✔
|  ├─ repeatedTestWithRepetitionInfo(RepetitionInfo) ✔
|  |  ├─ repetition 1 of 5 ✔
|  |  ├─ repetition 2 of 5 ✔
|  |  ├─ repetition 3 of 5 ✔
|  |  ├─ repetition 4 of 5 ✔
|  |  └─ repetition 5 of 5 ✔
|  ├─ Repeat! ✔
|  |  └─ Repeat! 1/1 ✔
|  ├─ Details... ✔
|  |  └─ Details... :: repetition 1 of 1 ✔
|  └─ repeatedTestInGerman() ✔
|     ├─ Wiederholung 1 von 5 ✔
|     ├─ Wiederholung 2 von 5 ✔
|     ├─ Wiederholung 3 von 5 ✔
|     ├─ Wiederholung 4 von 5 ✔
|     └─ Wiederholung 5 von 5 ✔
```

3.13. 参数化测试

参数化测试可以用不同的参数多次运行测试。除了使用 `@ParameterizedTest` 注解，它们的声明跟 `@Test` 的方法没有区别。此外，你必须声明至少一个给每次调用提供参数的来源。

> ⚠️ 参数化测试目前是一个试验性功能。详细信息请参阅 试验性API 中的表格。

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(isPalindrome(candidate));
}
```

上面这个参数化测试使用 `@ValueSource` 注解来指定一个 `String` 数组作为参数源。执行上述方法时，每次调用会被分别报告。例如， `ConsoleLauncher` 会打印类似下面的信息：

```
palindromes(String) ✔
├─ [1] racecar ✔
├─ [2] radar ✔
└─ [3] able was I ere I saw elba ✔
```

3.13.1. 必需的设置

为了使用参数化测试，你必须添加 `junit-jupiter-params` 依赖。详细信息请参考 依赖元数据。

3.13.2. 参数源

Junit Jupiter提供一些开箱即用的*源*注解。接下来每个子章节将提供一个简要的概述和一个示例。更多信息请参阅 `org.junit.jupiter.params.provider` 包中的JavaDoc。

@ValueSource

`@ValueSource` 是最简单来源之一。它允许你指定一个基本类型的数组（String、int、long或double），并且它只能为每次调用提供一个参数。

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testWithValueSource(int argument) {
    assertNotNull(argument);
}
```

@EnumSource

`@EnumSource` 能够很方便地提供 `Enum` 常量。该注解提供了一个可选的 `names` 参数，你可以用它来指定使用哪些常量。如果省略了，就意味着所有的常量将被使用，就像下面的例子所示。

```
@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithEnumSource(TimeUnit timeUnit) {
    assertNotNull(timeUnit);
}
```

```
@ParameterizedTest
@EnumSource(value = TimeUnit.class, names = { "DAYS", "HOURS" })
void testWithEnumSourceInclude(TimeUnit timeUnit) {
    assertTrue(EnumSet.of(TimeUnit.DAYS, TimeUnit.HOURS).contains(timeUnit));
}
```

`@EnumSource` 注解还提供了一个可选的 `mode` 参数，它能够细粒度地控制哪些常量将会被传递到测试方法中。例如，你可以从枚举常量池中排除一些名称或者指定正则表达式，如下面代码所示。

```
@ParameterizedTest
@EnumSource(value = TimeUnit.class, mode = EXCLUDE, names = { "DAYS", "HOURS" })
void testWithEnumSourceExclude(TimeUnit timeUnit) {
    assertFalse(EnumSet.of(TimeUnit.DAYS, TimeUnit.HOURS).contains(timeUnit));
    assertTrue(timeUnit.name().length() > 5);
}
```

```
@ParameterizedTest
@EnumSource(value = TimeUnit.class, mode = MATCH_ALL, names = "^(M|N).+SECONDS$")
void testWithEnumSourceRegex(TimeUnit timeUnit) {
    String name = timeUnit.name();
    assertTrue(name.startsWith("M") || name.startsWith("N"));
    assertTrue(name.endsWith("SECONDS"));
}
```

@MethodSource

`@MethodSource` 允许你引用测试类中的一个或多个工厂方法。这些工厂方法必须返回一个 `Stream` 、 `Iterable` 、 `Iterator` 或者参数数组。另外，它们不能接收任何参数。默认情况下，它们必须是 `static` 方法，除非测试类使用了 `@TestInstance(Lifecycle.PER_CLASS)` 注解。

如果你只需要一个参数，你可以返回一个参数类型的实例的 `Stream` ，如下面示例所示。

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithSimpleMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("foo", "bar");
}
```

同样支持基本类型的Stream(`DoubleStream` 、 `IntStream` 、 `LongStream`)，如下面示例所示。

```
@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int argument) {
    assertNotEquals(9, argument);
}

static IntStream range() {
    return IntStream.range(0, 20).skip(10);
}
```

如果测试方法声明了多个参数，则需要返回一个 `Arguments` 实例的集合或Stream，如下面代码所示。请注意， `Arguments.of(Object ...)` 是 `Arguments` 接口中定义的静态工厂方法。

```
@ParameterizedTest
@MethodSource("stringIntAndListProvider")
void testWithMultiArgMethodSource(String str, int num, List<String> list) {
    assertEquals(3, str.length());
    assertTrue(num >=1 && num <=2);
    assertEquals(2, list.size());
}

static Stream<Arguments> stringIntAndListProvider() {
    return Stream.of(
        Arguments.of("foo", 1, Arrays.asList("a", "b")),
        Arguments.of("bar", 2, Arrays.asList("x", "y"))
    );
}
```

@CsvSource

`@CsvSource` 允许你将参数列表定义为以逗号分隔的值（即 `String` 类型的值）。

```
@ParameterizedTest
@CsvSource({ "foo, 1", "bar, 2", "'baz, qux', 3" })
void testWithCsvSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}
```

@CsvSource使用单引号 `'` 作为引用字符。请参考上述示例和下表中的 `'baz, qux'` 值。一个空的引用值 `''` 表示一个空的 `String` ；而一个完全空的值被当成一个 `null` 引用。如果 `null` 引用的目标类型是基本类型，则会抛出一个 `ArgumentConversionException` 。

示例输入	生成的参数列表
`@CsvSource({ "foo, bar" })`	`"foo"` , `"bar"`
`@CsvSource({ "foo, 'baz, qux'" })`	`"foo"` , `"baz, qux"`
`@CsvSource({ "foo, ''" })`	`"foo"` , `""`
`@CsvSource({ "foo, " })`	`"foo"` , `null`

@CsvFileSource

`@CsvFileSource` 允许你使用类路径中的CSV文件。CSV文件中的每一行都会触发参数化测试的一次调用。

```
@ParameterizedTest
@CsvFileSource(resources = "/two-column.csv")
void testWithCsvFileSource(String first, int second) {
    assertNotNull(first);
    assertNotEquals(0, second);
}
```

two-column.csv

```
foo, 1
bar, 2
"baz, qux", 3
```

> 与 `@CsvSource` 中使用的语法相反， `@CsvFileSource` 使用双引号 `"` 作为引号字符，请参考上面例子中的 `"baz, qux"` 值，一个空的带引号的值 `""` 表示一个空 `String` ，一个完全为 `空` 的值被当成 `null` 引用，如果 `null` 引用的目标类型是基本类型，则会抛出一个 `ArgumentConversionException` 。

@ArgumentsSource

`@ArgumentsSource` 可以用来指定一个自定义且能够复用的 `ArgumentsProvider` 。

```
@ParameterizedTest
@ArgumentsSource(MyArgumentsProvider.class)
void testWithArgumentsSource(String argument) {
    assertNotNull(argument);
}

static class MyArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
        return Stream.of("foo", "bar").map(Arguments::of);
    }
}
```

3.13.3. 参数转换

隐式转换

为了支持像 `@CsvSource` 这样的使用场景，JUnit Jupiter提供了一些内置的隐式类型转换器。转换过程取决于每个方法参数的声明类型。

例如，如果一个 `@ParameterizedTest` 方法声明了 `TimeUnit` 类型的参数，而实际上提供了一个 `String` ，此时字符串会被自动转换成对应的 `TimeUnit` 枚举常量。

```
@ParameterizedTest
@ValueSource(strings = "SECONDS")
void testWithImplicitArgumentConversion(TimeUnit argument) {
    assertNotNull(argument.name());
}
```

`String` 实例目前会被隐式地转换成以下目标类型：

目标类型	类型示例
`boolean/Boolean`	`"true" → true`
`byte/Byte`	`"1" → (byte) 1`
`char/Character`	`"o" → 'o'`
`short/Short`	`"1" → (short) 1`
`int/Integer`	`"1" → 1`
`long/Long`	`"1" → 1L`
`float/Float`	`"1.0" → 1.0f`
`double/Double`	`"1.0" → 1.0d`
`Enum subclass`	`"SECONDS" → TimeUnit.SECONDS`
`java.time.Instant`	`"1970-01-01T00:00:00Z" → Instant.ofEpochMilli(0)`
`java.time.LocalDate`	`"2017-03-14" → LocalDate.of(2017, 3, 14)`
`java.time.LocalDateTime`	`"2017-03-14T12:34:56.789" → LocalDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000)`
`java.time.LocalTime`	`"12:34:56.789" → LocalTime.of(12, 34, 56, 789_000_000)`
`java.time.OffsetDateTime`	`"2017-03-14T12:34:56.789Z" → OffsetDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000, ZoneOffset.UTC)`
`java.time.OffsetTime`	`"12:34:56.789Z" → OffsetTime.of(12, 34, 56, 789_000_000, ZoneOffset.UTC)`
`java.time.Year`	`"2017" → Year.of(2017)`
`java.time.YearMonth`	`"2017-03" → YearMonth.of(2017, 3)`
`java.time.ZonedDateTime`	`"2017-03-14T12:34:56.789Z" → ZonedDateTime.of(2017, 3, 14, 12, 34, 56, 789_000_000, ZoneOffset.UTC)`

显式转换

除了使用隐式转换参数，你还可以使用 `@ConvertWith` 注解来显式指定一个 `ArgumentConverter` 用于某个参数，例如下面代码所示。

```
@ParameterizedTest
@EnumSource(TimeUnit.class)
void testWithExplicitArgumentConversion(@ConvertWith(ToStringArgumentConverter.class) String argument) {
    assertNotNull(TimeUnit.valueOf(argument));
}

static class ToStringArgumentConverter extends SimpleArgumentConverter {

    @Override
    protected Object convert(Object source, Class<?> targetType) {
        assertEquals(String.class, targetType, "Can only convert to String");
        return String.valueOf(source);
    }
}
```

显式参数转换器意味着开发人员要自己去实现它。正因为这样， `junit-jupiter-params` 仅仅提供了一个可以作为参考实现的显式参数转换器： `JavaTimeArgumentConverter` 。你可以通过组合注解 `JavaTimeArgumentConverter` 来使用它。

```
@ParameterizedTest
@ValueSource(strings = { "01.01.2017", "31.12.2017" })
void testWithExplicitJavaTimeConverter(@JavaTimeConversionPattern("dd.MM.yyyy") LocalDate argument) {
    assertEquals(2017, argument.getYear());
}
```

3.13.4. 自定义显示名称

默认情况下，参数化测试调用的显示名称包含了该特定调用的索引和所有参数的 `String` 表示形式。不过，你可以通过 `@ParameterizedTest` 注解的 `name` 属性来自定义调用的显示名称，如下面代码所示。

```
@DisplayName("Display name of container")
@ParameterizedTest(name = "{index} ==> first=''{0}'', second={1}")
@CsvSource({ "foo, 1", "bar, 2", "'baz, qux', 3" })
void testWithCustomDisplayNames(String first, int second) {
}
```

使用 `ConsoleLauncher` 执行上面方法，你会看到类似于下面的输出。

```
Display name of container ✔
├─ 1 ==> first='foo', second=1 ✔
├─ 2 ==> first='bar', second=2 ✔
└─ 3 ==> first='baz, qux', second=3 ✔
```

自定义显示名称支持下面表格中的占位符。

占位符	描述
`{index}`	当前调用的索引 (1-based)
`{arguments}`	完整的参数列表，以逗号分隔
`{0}, {1}, …`	单个参数

3.13.5. 生命周期和互操作性

参数化测试的每次调用拥有跟普通 `@Test` 方法相同的生命周期。例如， `@BeforeEach` 方法将在每次调用之前执行。类似于动态测试，调用将逐个出现在IDE的测试树中。你可能会在一个测试类中混合常规 `@Test` 方法和 `@ParameterizedTest` 方法。

你可以在 `@ParameterizedTest` 方法上使用 `ParameterResolver` 扩展。但是，被参数源解析的方法参数必须出现在参数列表的首位。由于测试类可能包含常规测试和具有不同参数列表的参数化测试，因此，参数源的值不会针对生命周期方法（例如 `@BeforeEach` ）和测试类构造函数进行解析。

```
@BeforeEach
void beforeEach(TestInfo testInfo) {
    // ...
}

@ParameterizedTest
@ValueSource(strings = "foo")
void testWithRegularParameterResolver(String argument, TestReporter testReporter) {
    testReporter.publishEntry("argument", argument);
}

@AfterEach
void afterEach(TestInfo testInfo) {
    // ...
}
```

3.14. 测试模板

`@TestTemplate` 方法不是一个常规的测试用例，它是测试用例的模板。因此，它的设计初衷是用来被多次调用，而调用次数取决于注册提供者返回的调用上下文数量。所以，它必须结合 TestTemplateInvocationContextProvider 扩展一起使用。测试模板方法每一次调用跟执行常规 `@Test` 方法一样，它也完全支持相同的生命周期回调和扩展。关于它的用例请参阅 为测试模板提供调用上下文。

3.15. 动态测试

JUnit Juppiter的 Annotations 章节描述的标准 `@Test` 注解跟JUnit 4中的 `@Test` 注解非常类似。两者都描述了实现测试用例的方法。这些测试用例都是静态的，因为它们是在编译时完全指定的，而且它们的行为不能在运行时被改变。*假设提供了一种基本的动态行为形式，但其表达性却被故意地加以限制。*

除了这些标准的测试以外，JUnit Jupiter还引入了一种全新的测试编程模型。这中新的测试是一个*动态测试*，它们由一个使用了 `@TestRactory` 注解的工厂方法在运行时生成。

相比于 `@Test` 方法， `@TestFactory` 方法本身不是测试用例，它是测试用例的工厂。因此，动态测试是工厂的产品。从技术上讲， `@TestFactory` 方法必须返回一个 `DynamicNode` 实例的 `Stream` 、 `Collection` 、 `Iterable` 、 `Iterator` 。 `DynamicNode` 的可实例化子类是 `DynamicContainer` 和 `DynamicTest` 。 `DynamicContainer` 实例由一个*显示名称*和一个动态子节点列表组成，它允许创建任意嵌套的动态节点层次结构。而 `DynamicTest` 实例会被延迟执行，从而生成动态甚至非确定性的测试用例。

任何由 `@TestFactory` 方法返回的 `Stream` 在调用 `stream.close()` 的时候会被正确地关闭，这样我们就可以安全地使用一个资源，例如： `Files.lines()` 。

跟 `@Test` 方法一样， `@TestFactory` 方法不能是 `private` 或 `static` 的。但它可以声明被 `ParameterResolvers` 解析的参数。

`DynamicTest` 是运行时生成的测试用例。它由一个*显示名称*和 `Executable` 组成。 `Executable` 是一个 `@FunctionalInterface` ，这意味着动态测试的实现可以是一个*lambda表达式* 或*方法引用*。

> ⚠️ *动态测试生命周期*
>
> 动态测试执行生命周期跟标准的 `@Test` 测试截然不同。具体而言，动态测试不存在任何生命周期回调。这意味着 `@BeforeEach` 和 `@AfterEach` 方法以及它们相应的扩展回调函数对 `@TestFactory` 方法执行，而不是对每个动态测试执行。换言之，如果你从一个lambda表达式的测试实例中访问动态测试的字段，那么由同一个 `@TestFactory` 方法生成的各个动态测试执行之间的回调方法或扩展不会重置那些字段。
>
> 译者注：同一个 `@TestFactory` 所生成的n个动态测试， `@BeforeEach` 和 `@AfterEach` 只会在这n个动态测试开始前和结束后各执行一次，不会为每一个单独的动态测试都执行。

从JUnit Jupiter 5.0.2开始，动态测试必须始终由工厂方法创建；不过，在后续的发行版中，这可以通过注册工具来提供。

> ⚠️ 动态测试目前是一个试验性功能。详细信息请参阅 试验性API 中的表格。

3.15.1. 动态测试示例

下面的 `DynamicTestsDemo` 类演示了测试工厂和动态测试的几个示例。

第一个方法返回一个无效的返回类型。由于在编译时无法检测到无效的返回类型，因此在运行时会抛出 `JUnitException` 。

接下来五个方法是非常简单的例子，它们演示了生成一个 `DynamicTest` 实例的 `Collection` 、 `Iterable` 、 `Iterator` 、 `Stream` 。这些例子中大多数并不真正表现出动态行为，而只是为了证明原则上所支持的返回类型。然而， `dynamicTestsFromStream()` 和 `dynamicTestsFromIntStream()` 演示了为给定的一组字符串或一组输入数字生成动态测试是多么的容易。

下一个方法是真正意义上动态的。 `generateRandomNumberOfTests()` 实现了一个生成随机数的 `Iterator` ，一个显示名称生成器和一个测试执行器，然后将这三者提供给 `DynamicTest.stream()` 。因为 `generateRandomNumberOfTests()` 的非确定性行为会与测试的可重复性发生冲突，因此应该谨慎使用，这里只是用它来演示动态测试的表现力和强大。

最后一个方法使用 `DynamicContainer` 来生成动态测试的嵌套层次结构。

```
import static org.junit.jupiter.api.Assertions.assertEquals;
```

```java
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.DynamicContainer.dynamicContainer;
import static org.junit.jupiter.api.DynamicTest.dynamicTest;

import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.Random;
import java.util.function.Function;
import java.util.stream.IntStream;
import java.util.stream.Stream;

import org.junit.jupiter.api.DynamicNode;
import org.junit.jupiter.api.DynamicTest;
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.TestFactory;
import org.junit.jupiter.api.function.ThrowingConsumer;

class DynamicTestsDemo {

    // This will result in a JUnitException!
    @TestFactory
    List<String> dynamicTestsWithInvalidReturnType() {
        return Arrays.asList("Hello");
    }

    @TestFactory
    Collection<DynamicTest> dynamicTestsFromCollection() {
        return Arrays.asList(
            dynamicTest("1st dynamic test", () -> assertTrue(true)),
            dynamicTest("2nd dynamic test", () -> assertEquals(4, 2 * 2))
        );
    }

    @TestFactory
    Iterable<DynamicTest> dynamicTestsFromIterable() {
        return Arrays.asList(
            dynamicTest("3rd dynamic test", () -> assertTrue(true)),
            dynamicTest("4th dynamic test", () -> assertEquals(4, 2 * 2))
        );
    }

    @TestFactory
    Iterator<DynamicTest> dynamicTestsFromIterator() {
        return Arrays.asList(
            dynamicTest("5th dynamic test", () -> assertTrue(true)),
            dynamicTest("6th dynamic test", () -> assertEquals(4, 2 * 2))
        ).iterator();
    }

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromStream() {
        return Stream.of("A", "B", "C")
            .map(str -> dynamicTest("test" + str, () -> { /* ... */ }));
    }

    @TestFactory
    Stream<DynamicTest> dynamicTestsFromIntStream() {
        // Generates tests for the first 10 even integers.
        return IntStream.iterate(0, n -> n + 2).limit(10)
            .mapToObj(n -> dynamicTest("test" + n, () -> assertTrue(n % 2 == 0)));
    }

    @TestFactory
    Stream<DynamicTest> generateRandomNumberOfTests() {

        // Generates random positive integers between 0 and 100 until
        // a number evenly divisible by 7 is encountered.
        Iterator<Integer> inputGenerator = new Iterator<Integer>() {

            Random random = new Random();
            int current;
```

```
            @Override
            public boolean hasNext() {
                current = random.nextInt(100);
                return current % 7 != 0;
            }

            @Override
            public Integer next() {
                return current;
            }
        };

        // Generates display names like: input:5, input:37, input:85, etc.
        Function<Integer, String> displayNameGenerator = (input) -> "input:" + input;

        // Executes tests based on the current input value.
        ThrowingConsumer<Integer> testExecutor = (input) -> assertTrue(input % 7 != 0);

        // Returns a stream of dynamic tests.
        return DynamicTest.stream(inputGenerator, displayNameGenerator, testExecutor);
    }

    @TestFactory
    Stream<DynamicNode> dynamicTestsWithContainers() {
        return Stream.of("A", "B", "C")
            .map(input -> dynamicContainer("Container " + input, Stream.of(
                dynamicTest("not null", () -> assertNotNull(input)),
                dynamicContainer("properties", Stream.of(
                    dynamicTest("length > 0", () -> assertTrue(input.length() > 0)),
                    dynamicTest("not empty", () -> assertFalse(input.isEmpty()))
                ))
            )));
    }

}
```

4. 运行测试

即将上线

5. 扩展模型

即将上线

6. 从JUnit 4迁移

即将上线

7. 高级主题

7.1 JUnit Platform启动器API

JUnit 5的主要目标之一是让JUnit和其编程客户端（构建工具和IDE）之间的接口更加强大和稳定。目的是将发现和执行测试的内部构件和外部必需的所有过滤和配置分离开来。

JUnit 5 引入了 `Launcher` 的概念，它可以被用来发现、过滤和执行测试。此外，诸如 Spock、Cucumber 和 FitNesse 等第三方测试库都可以通过提供自定义的 `TestEngine` 来集成到JUnit 5平台的启动基础设施中。

启动API在 `junit-platform-launcher` 模块中。

`junit-platform-console` 项目中的 `ConsoleLauncher` 就是一个具体的使用例示。

7.1.1 发现测试

将*测试发现* 作为平台本身的一个专用功能而引入，会（希望能够）将IDE和构建工具从过去难以识别测试类和测试方法的大部分困难中释放出来。

使用示例：

```
import static org.junit.platform.engine.discovery.ClassNameFilter.includeClassNamePatterns;
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectClass;
import static org.junit.platform.engine.discovery.DiscoverySelectors.selectPackage;

import org.junit.platform.launcher.Launcher;
import org.junit.platform.launcher.LauncherDiscoveryRequest;
import org.junit.platform.launcher.TestExecutionListener;
import org.junit.platform.launcher.TestPlan;
import org.junit.platform.launcher.core.LauncherDiscoveryRequestBuilder;
import org.junit.platform.launcher.core.LauncherFactory;
import org.junit.platform.launcher.listeners.SummaryGeneratingListener;
```

```
LauncherDiscoveryRequest request = LauncherDiscoveryRequestBuilder.request()
    .selectors(
        selectPackage("com.example.mytests"),
        selectClass(MyTestClass.class)
    )
    .filters(
        includeClassNamePatterns(".*Tests")
    )
    .build();

Launcher launcher = LauncherFactory.create();

TestPlan testPlan = launcher.discover(request);
```

目前，搜索范围涵盖了类、方法、包中的所有类，甚至所有类路径中的测试。测试发现发生在所有参与的测试引擎。

生成的 `TestPlan` 是符合 `LauncherDiscoveryRequest` 对象的所有引擎、类、和测试方法的结构化（只读）描述。客户端可以遍历树，检索节点的详细信息，并获取到原始源的链接（如类，方法或文件位置）。测试计划中的每个节点都有一个*唯一的ID*，可以用来调用特定的测试或一组测试。

7.1.2 执行测试

要执行测试，客户端可以使用与发现阶段相同的 `LauncherDiscoveryRequest` ，或者创建一个新的请求。测试进度和报告可以通过使用 `Launcher` 注册一个或多个 `TestExecutionListener` 实现来获取，如下面例子所示。

```
LauncherDiscoveryRequest request = LauncherDiscoveryRequestBuilder.request()
    .selectors(
        selectPackage("com.example.mytests"),
        selectClass(MyTestClass.class)
    )
    .filters(
        includeClassNamePatterns(".*Tests")
    )
    .build();

Launcher launcher = LauncherFactory.create();

// 注册一个你选择的监听器
TestExecutionListener listener = new SummaryGeneratingListener();
launcher.registerTestExecutionListeners(listener);

launcher.execute(request);
```

`execute()` 方法没有返回值，但你可以轻松地使用监听器将最终结果聚合到你自己的对象中。相关示例请参阅 `SummaryGeneratingListener` 。

7.1.3 插入你自己的测试引擎

Junit 目前提供了两种开箱即用的 `TestEngine` ：

- `junit-jupiter-engine` : JUnit Jupiter的核心。
- `junit-vintage-engine` : JUnit 4之上的一个薄层，它允许使用启动器基础设施来运行 `老版本` 的测试。

第三方也可以通过在 `junit-platform-engine` 模块中实现接口并*注册*引擎来提供他们自己的 `TestEngine` 。 目前Java的 `java.util.ServiceLoader` 机制支持引擎注册。 例如， `junit-jupiter-engine` 模块将其 `org.junit.jupiter.engine.JupiterTestEngine` 注册到一个名为 `org.junit.platform.engine.TestEngine` 的文件中，该文件位于 `junit-jupiter-engine` JAR包中的 `/META-INF/services` 目录。

7.1.4 插入你自己的测试执行监听器

In addition to the public Launcher API method for registering test execution listeners programmatically, custom TestExecutionListener implementations discovered at runtime via

Java's java.util.ServiceLoader facility are automatically registered with the DefaultLauncher

除了以编程方式来注册测试执行监听器的公共 `Launcher` API方法之外，在运行时由Java的 `java.util.ServiceLoader` 工具发现的自定义 `TestExecutionListener` 实现会被自动注册到 `DefaultLauncher` 。 例如，一个实现了 `TestExecutionListener` 并声明在 `/META-INF/services/org.junit.platform.launcher.TestExecutionListener` 文件中的 `example.TestInfoPrinter` 类会被自动加载和注册。

8. API演变

One of the major goals of JUnit 5 is to improve maintainers' capabilities to evolve JUnit despite its being used in many projects. With JUnit 4 a lot of stuff that was originally added as an internal construct only got used by external extension writers and tool builders. That made changing JUnit 4 especially difficult and sometimes impossible.

JUnit 5的主要目标之一是提高维护者演进改善JUnit的能力，尽管它正在很多项目中被使用。使用JUnit 4中，很多最初作为内部构造而被添加的内容只能被外部扩展编写器和工具构建器使用。这就使得改变JUnit 4异常困难，甚至有时是不可能的。

That's why JUnit 5 introduces a defined lifecycle for all publicly available interfaces, classes, and methods.

这就是为什么JUnit 5为所有公开的接口、类和方法引入了一个明确的生命周期。

8.1 API 版本和状态

每个发布的包都有一个版本号 `<major>.<minor>.<patch>` ，所有公开的接口、类和方法都使用 @API Guardian 项目中的 @API 进行标注。 `@API` 注解的 `status` 属性可以被赋予下面表格中的值。

状态	描述
INTERNAL	只能被JUnit自身使用，可能会被删除，但不事先另行通知。
DEPRECATED	不应再使用；可能会在下一个小版本中消失。
EXPERIMENTAL	用于我们正在收集反馈的新的试验性功能。谨慎使用这个元素；它可能会在未来被提升为 `MAINTAINED` 或 `STABLE` ，但也可能在没有事先通知的情况下被移除，即使在一个补丁中。
MAINTAINED	用于*至少*在当前主要版本的下一个次要版本中不会以反向不兼容的方式更改的功能。如果计划删除，则会首先将其降为 `DEPRECATED` 。
STABLE	用于在当前主版本（5.*）中不会以反向不兼容的方式更改的功能。

如果 `@API` 注解出现在某个类型上，则认为它也适用于该类型的所有公共成员。一个成员可以声明一个稳定性更低的 `status` 值。

8.2 试验性API

下表列出了哪些API当前被指定为*试验性的*（通过 `@API(status = EXPERIMENTAL)` ）。使用这样的API时应该谨慎。

包名	类名	类型
org.junit.jupiter.api	DynamicContainer	类
org.junit.jupiter.api	DynamicNode	类
org.junit.jupiter.api	DynamicTest	类
org.junit.jupiter.api	TestFactory	注解
org.junit.jupiter.migrationsupport.rules	EnableRuleMigrationSupport	注解
org.junit.jupiter.migrationsupport.rules	ExpectedExceptionSupport	类
org.junit.jupiter.migrationsupport.rules	ExternalResourceSupport	类
org.junit.jupiter.migrationsupport.rules	VerifierSupport	类
org.junit.jupiter.params	ParameterizedTest	注解
org.junit.jupiter.params.converter	ArgumentConversionException	类
org.junit.jupiter.params.converter	ArgumentConverter	接口
org.junit.jupiter.params.converter	ConvertWith	注解
org.junit.jupiter.params.converter	JavaTimeConversionPattern	注解
org.junit.jupiter.params.converter	SimpleArgumentConverter	类
org.junit.jupiter.params.provider	Arguments	接口
org.junit.jupiter.params.provider	ArgumentsProvider	接口
org.junit.jupiter.params.provider	ArgumentsSource	注解
org.junit.jupiter.params.provider	ArgumentsSources	注解
org.junit.jupiter.params.provider	CsvFileSource	注解
org.junit.jupiter.params.provider	CsvSource	注解
org.junit.jupiter.params.provider	EnumSource	注解
org.junit.jupiter.params.provider	MethodSource	注解
org.junit.jupiter.params.provider	ValueSource	注解
org.junit.jupiter.params.support	AnnotationConsumer	接口
org.junit.platform.gradle.plugin	EnginesExtension	类
org.junit.platform.gradle.plugin	FiltersExtension	类
org.junit.platform.gradle.plugin	JUnitPlatformExtension	类
org.junit.platform.gradle.plugin	JUnitPlatformPlugin	类
org.junit.platform.gradle.plugin	PackagesExtension	类
org.junit.platform.gradle.plugin	SelectorsExtension	类
org.junit.platform.gradle.plugin	TagsExtension	类
org.junit.platform.surefire.provider	JUnitPlatformProvider	类

8.3. @API工具支持

@API Guardian 项目计划为使用 @API 注解的API的发布者和消费者提供工具支持。例如，工具支持可能会提供一种方法来检查是否按照 `@API` 注解声明来使用JUnit API。

9. 贡献者

可以在GitHub上直接浏览 当前贡献者列表

10. 发布记录

5.0.2

发布时间： 2017.11.12

范围：自5.0.1版本以来的错误修复和小的改进。

关于此版本所有已关闭的问题和请求的完整列表，请参阅GitHub上的JUnit仓库中的 5.0.2 里程碑页面。

JUnit Platform

Bug修复

- 修复后，Maven Surefire对于不使用 `MethodSource` 的测试引擎（例如Spek）能正确地报告失败的测试。
- 修复后，当一个非零的 `forkCount` 与Maven Surefire一起执行时，可以正确地报告写入 `System.out` 或 `System.err` 的测试，特别是通过一个日志框架的时候。

新功能和改进

- JUnit Platform Maven Surefire提供者程序现在支持 `redirectTestOutputToFile` Surefire功能。
- JUnit Platform Maven Surefire提供者程序现在会忽略通过 `<includeTags/>`， `<groups/>`， `<excludeTags/>` 和 `<excludedGroups/>` 提供的空字符串。

JUnit Jupiter

Bug修复

- `@CsvSource` 或 `@CsvFileSource` 输入行中的尾随空格不再生成空值。
- 以前， `@EnableRuleMigrationSupport` 无法识别 `@Rule` 方法，该方法返回一个已支持的 `TestRule` 类型的子类型。而且，它错误地实例化了某些多次使用方法声明的规则。现在，一旦启用，它将实例化所有声明的规则（字段*和*方法），并按照JUnit 4使用的顺序来调用它们。

Previously, disabled test classes were eagerly instantiated when Lifecycle.PER_CLASS was used. Now, ExecutionCondition evaluation always takes place before test class instantiation.

- 以前，当使用 `Lifecycle.PER_CLASS` 时，被禁用的测试类会被迫切地实例化。现在， `ExecutionCondition` 总是在测试类实例化之前就被解析。
- `unit-jupiter-migrationsupport` 模块不再会错误地尝试通过 `ServiceLoader` 机制来注册 `JupiterTestEngine`，从而允许将其用作Java 9模块路径上的模块。

新功能和改进

- 现在， `Assertions` 类中的 `assertTrue()` 和 `assertFalse()` 的失败消息包含了关于预期和实际布尔值的详细信息。
  - 例如，调用 `assertTrue(false)` 生成的失败消息现在变成了 `"expected:<true>but was: <false>"`，而不是空字符串。
- 如果参数化测试没有消费通过参数源提供给它的所有参数，那么未使用的参数将不再被包含在显示名称中。

JUnit Vintage

没有变化。

5.0.1

发布时间： 2017.10.03

范围：修复了5.0.0版的错误

关于此版本所有已关闭的问题和请求的完整列表，请参阅GitHub上的JUnit仓库中的 5.0.1 里程碑页面。

整体改进

- 所有的包现在都有一个 `optional` 的依赖，而不需要在其发布的Maven POM中强制依赖*@API Guardian* JAR包。

JUnit Platform

没有变化。

JUnit Jupiter

Bug修复

- 如果测试类中未声明JUnit 4 `ExpectedException` 规则， `junit-jupiter-migrationsupport` 模块中的 `ExpectedExceptionSupport` 不会再吃掉异常。
  - 因此，现在可以使用 `@EnableRuleMigrationSupport` 和 `ExpectedExceptionSupport`，而不用声明 `ExpectedException` 规则。

JUnit Vintage

Bug 修复

- `PackageNameFilters` 现在应用于通过 `ClassSelector` ， `MethodSelector` 或 `UniqueIdSelector` 选择的测试。

5.0.0

发布时间： 2017.09.10

范围：首个通用版本

关于此版本所有已关闭的问题和请求的完整列表，请参阅GitHub上的JUnit仓库中的 5.0 GA 里程碑页面。

JUnit Platform

Bug修复

- `AbstractTestDescriptor` 中的 `removeFromHierarchy()` 实现现在也清除了所有子级的父级关系。

启用和彻底改变

- `@API` 注释已经从 `junit-platform-commons` 项目中删除，并重新定位到GitHub上一个名为 @API Guardian 的独立新项目。
- 标签不再允许包含以下任何保留字符。
  - `,` , `(` , `)` , `&` , `|` , `!`
- `FilePosition` 的构造函数已被替换为一个名为 `from(int, int)` 的静态工厂方法。
- 一个 `FilePosition` 现在全完可以通过新的 `from(int)` 静态工厂方法从一个行号进行构建。

新功能和改进

JUnit Jupiter

Bug修复

JUnit Vintage

没有变化。

5.0.0-RC3

发布时间： 2017.08.23

范围:

5.0.0-RC2

发布时间： 2017.07.30

范围:

5.0.0-RC1

发布时间： 2017.07.30

范围:

5.0.0-M6

发布时间： 2017.07.18

范围:

5.0.0-M5

发布时间： 2017.07.04

范围:

5.0.0-M4

发布时间：2017.04.01

范围：

5.0.0-M3

发布时间：2016.11.30

范围：

5.0.0-M2

发布时间：2016.07.23

范围：

5.0.0-M1

发布时间：2016.07.07

范围：

5.0.0-ALPHA

发布时间：2016.02.01

范围：