Junit 5官方文档中文版

书栈(BookStack.CN)

目　录

- 致谢
- 介绍
- 用户指南
- 概况
    - Junit5是什么？
    - 支持的Java版本
    - 获取帮助
- 安装
    - 依赖元数据
    - 依赖关系图
    - JUnit Jupiter示例项目
- 编写测试
    - 注解
    - 标准测试类
    - 显示名称
    - 断言
    - 假设
    - 禁用
    - 标签和过滤
    - 测试实例生命周期
    - 嵌套测试
    - 构造函数和方法的依赖注入
    - 测试接口和默认方法
    - 重复测试
    - 参数化测试
    - 测试模板
    - 动态测试
- 运行测试
    - IDE支持
    - 构建支持
    - 控制台启动器
    - 使用Junit4运行JUnit Platfrom
    - 配置参数
- 扩展模型
    - 概述
    - 注册扩展

有条件的测试执行
测试实例后处理
测试生命周期回调
异常处理
为测试模板提供调用上下文
在扩展中维持状态
在扩展中支持的实用程序
用户代码和扩展的相对执行顺序
从Junit4迁移
在Junit Platform上运行JUnit4测试
迁移Tips
受限的JUnit4规则支持
高级主题
JUnit Platform Launcher API
API演进
API版本和状态
实验性API
@API工具支持

致谢

当前文档 《Junit 5官方文档中文版》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于2018-06-05。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/junit5

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

- Junit 5官方文档中文版
  - 介绍
  - doczh.cn
  - 来源(书栈小编注)

Junit 5官方文档中文版

介绍

JUnit 5是JUnit的下一代。目标是为JVM上的开发人员端测试创建一个最新的基础。这包括专注于Java 8及更高版本，以及启用许多不同风格的测试。

JUnit 5团队于2017年9月10日发布了第一个[GA版本 5.0.0][ga]。

这里是Junit5官方文档的中文翻译版，由doczh.cn组织翻译并更新维护。

文档内容发布于gitbook，请点击下面的链接阅读或者下载电子版本：

- 在线阅读
  - 国外服务器：gitbook提供的托管，服务器在国外，速度比较慢，偶尔被墙，HTTPS
  - 国内服务器：~~腾讯云加速，国内网速极快，非HTTPS~~ 即将推出
- 下载pdf格式
- 下载mobi格式
- 下载epub格式

本文内容可以任意转载，但是需要注明来源并提供链接。

请勿用于商业出版。

doczh.cn

doczh.cn是一个纯技术的，面向技术社区，非商业性质的组织，专注于IT产品的中文文档翻译。

http://doczh.cn/

如果对文档翻译工作感兴趣，欢迎加入，也欢迎为现有文档的改善提供帮助。

来源(书栈小编注)

> *https://github.com/doczhcn/junit5*
> 注意：当前文档，译者并未翻译完全部内容，请实时关注上面的地址库，以跟进译者的最新翻译。

用户指南

- JUnit5用户指南

JUnit5用户指南

JUnit5官方user guide文档翻译，原英文文档地址为：

http://junit.org/junit5/docs/current/user-guide/

英文文档的github托管地址为：

https://github.com/junit-team/junit5/tree/master/documentation

当前版本是：Junit Version **5.0.2**

概况

- 概况

概况

本文档的目的是提供全面的参考文档，用于编写测试的编程人员，扩展作者和引擎作者，以及构建工具和IDE供应商。

本文档也可作为PDF下载。

Junit5是什么？

- Junit 5是什么？

Junit 5是什么？

与以前的JUnit版本不同，JUnit 5由三个不同子项目的多个不同模块组成。

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

JUnit Platform为在JVM上启动测试框架提供基础。它还定义了TestEngine API，用来开发在平台上运行的测试框架。此外，平台提供了一个控制台启动器，用于从命令行启动平台，并为Gradle和Maven提供构建插件以及基于JUnit 4的Runner，用于在平台上运行任意 `TestEngine` 。

JUnit Jupiter是在JUnit 5中编写测试和扩展的新型编程模型和扩展模型的组合.Jupiter子项目提供了 `TestEngine` ，用于在平台上运行基于Jupiter的测试。

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

支持的Java版本

- 支持的Java版本

支持的Java版本

JUnit 5在运行时需要Java 8（或更高版本）。当然，您仍然可以测试使用以前版本的JDK编译的代码。

获取帮助

- 获取帮助

获取帮助

在Stack Overflow上询问JUnit 5相关问题或在Gitter上与我们聊天。

安装

- 安装

安装

最终版本和里程碑的工件（artifact）已经部署到Maven Central。

Snapshot工件部署到Sonatype的快照仓库，在/org/junit下。

依赖元数据

- 依赖元数据
  - JUnit Platform
  - JUnit Jupiter
  - JUnit Vintage
  - 可选依赖

依赖元数据

JUnit Platform

- **Group ID**: `org.junit.platform`
- **Version**: `{{book.platformVersion}}`
- **Artifact IDs**:

Artifact	说明
junit-platform-commons	JUnit的内部公共库/工具。 这些工具预期仅用于在JUnit框架本身内部使用。 不支持任何外部使用。使用它需要自己承担风险！
junit-platform-console	支持从控制台发现和执行JUnit Platform上的测试。 有关详情，请参阅控制台启动器
`junit-platform-console-standalone`	Maven Central中，在 junit-platform-console-standalone 目录下，提供了一个包含所有依赖的可执行JAR 。 有关详情，请参阅控制台启动器
`junit-platform-engine`	用于test engine的公共API. 有关详细信息，请参阅插入自己的测试引擎。
`junit-platform-gradle-plugin`	支持用Gradle在JUnit Platform上发现和执行测试 。
`junit-platform-launcher`	用于配置和启动test plan的公共API。 通常被IDE和构建工具使用。 详情请参阅JUnit Platform Launcher API。
`junit-platform-runner`	用于在JUnit 4环境中，在JUnit Platform上 执行test和test suite的运行程序。 有关详细信息，请参阅使用JUnit4运行JUnit平台。
`junit-platform-suite-api`	在JUnit Platform上配置test suite的注解。 由JUnitPlatform runner转换器支持， 也可能由第三方 TestEngine实现。
`junit-platform-surefire-provider`	支持使用Maven Surefire在JUnit Platform上 发现和执行测试 。。

JUnit Jupiter

- **Group ID**: `org.junit.jupiter`
- **Version**: `{{book.jupiterVersion}}`
- **Artifact IDs**:

Artifact ID	说明
	JUnit Jupiter API，用来编写测试 和扩展

	JUnit Jupiter API，用来编写测试 和扩展
`junit-jupiter-engine`	JUnit Jupiter测试引擎实现，仅在运行时需要
`junit-jupiter-params`	支持在JUnit Jupiter中参数化测试
`junit-jupiter-migrationsupport`	从JUnit4到JUnit Jupiter的迁移支持， 仅在运行选定的JUnit规则时需要

JUnit Vintage

- **Group ID**: `org.junit.vintage`
- **Version**: `{{book.vintageVersion}}`
- **Artifact IDs**:

Artifact ID	说明
`junit-vintage-engine`	JUnit Vintage测试引擎实现，容许在新的JUnit Platform上 运行vintage JUnit测试，例如用JUnit3或者JUnit4风格写的测试

可选依赖

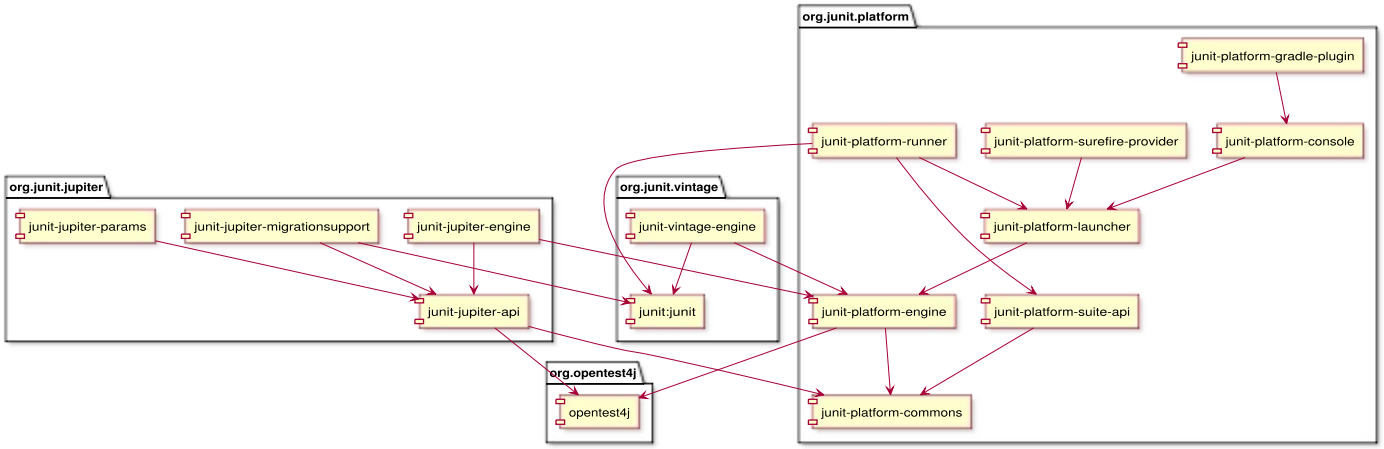
上面的所有构建，在他们发布的maven pom文件中，都有一个 `可选` 依赖，就是下面的@API Guardian JAR.

- *Group ID*: `org.apiguardian`
- *Artifact ID*: `apiguardian-api`
- *Version*: `{{book.apiguardianVersion}}`

依赖关系图

- 依赖关系图

依赖关系图



JUnit Jupiter示例项目

- JUnit Jupiter示例项目

JUnit Jupiter示例项目

`junit5-samples` 仓库托管了很多基于JUnit Jupiter和JUnit Vintage的示例项目。可以在下面的项目中找到各自的 `build.gradle` 和 `pom.xml` .

- 对于Gradle, check out `junit5-gradle-consumer` 项目.
- 对于Maven, check out `junit5-maven-consumer` 项目.

编写测试

- 编写测试

编写测试

第一个测试案例：

```
1.  import static org.junit.jupiter.api.Assertions.assertEquals;
2.
3.  import org.junit.jupiter.api.Test;
4.
5.  class FirstJUnit5Tests {
6.
7.      @Test
8.      void myFirstTest() {
9.          assertEquals(2, 1 + 1);
10.     }
11. }
```

注解

- 注解
  - 元注解和组合注解

注解

JUnit Jupiter支持下列注解，用于配置测试和扩展框架。

所有核心注解位于 `junit-jupiter-api` 模块中的 `org.junit.jupiter.api` 包中。

注解	描述
@Test	表示方法是测试方法。与JUnit4的@Test注解不同的是，这个注解没有声明任何属性，因为JUnit Jupiter中的测试扩展是基于他们自己的专用注解来操作的。除非被覆盖，否则这些方法可以继承。
@ParameterizedTest	表示方法是参数化测试。 除非被覆盖，否则这些方法可以继承。
@RepeatedTest	表示方法是用于重复测试的测试模板。除非被覆盖，否则这些方法可以继承。
@TestFactory	表示方法是用于动态测试的测试工厂。除非被覆盖，否则这些方法可以继承。
@TestInstance	用于为被注解的测试类配置测试实例生命周期。 这个注解可以继承。
@TestTemplate	表示方法是测试用例的模板，设计为被调用多次，调用次数取决于自注册的提供者返回的调用上下文。除非被覆盖，否则这些方法可以继承。
@DisplayName	声明测试类或测试方法的自定义显示名称。这个注解不被继承。
@BeforeEach	表示被注解的方法应在当前类的每个@Test，@RepeatedTest，@ParameterizedTest或@TestFactory方法之前执行； 类似于JUnit 4的@Before。 除非被覆盖，否则这些方法可以继承。
@AfterEach	表示被注解的方法应在当前类的每个@Test，@RepeatedTest，@ParameterizedTest或@TestFactory方法之后执行； 类似于JUnit 4的@After。 除非被覆盖，否则这些方法可以继承。
@BeforeAll	表示被注解的方法应该在当前类的所有@Test，@RepeatedTest，@ParameterizedTest和@TestFactory方法之前执行； 类似于JUnit 4的@BeforeClass。 这样的方法可以继承（除非被隐藏或覆盖），并且必须是静态的（除非使用“per-class”测试实例生命周期）。
@AfterAll	表示被注解的方法应该在当前类的所有@Test，@RepeatedTest，@ParameterizedTest和@TestFactory方法之后执行； 类似于JUnit 4的@AfterClass。 这样的方法可以继承（除非被隐藏或覆盖），并且必须是静态的（除非使用“per-class”测试实例生命周期）。
@Nested	表示被注解的类是一个嵌套的非静态测试类。除非使用“per-class”测试实例生命周期，否则@BeforeAll和@AfterAll方法不能直接在@Nested测试类中使用。 这个注解不能继承。
@Tag	在类或方法级别声明标签，用于过滤测试； 类似于TestNG中的test group或JUnit 4中的Categories。这个注释可以在类级别上继承，但不能在方法级别上继承。
@Disabled	用于禁用测试类或测试方法； 类似于JUnit4的@Ignore。这个注解不能继承。
@ExtendWith	用于注册自定义扩展。 这个注解可以继承。

使用@Test，@TestTemplate，@RepeatedTest，@BeforeAll，@AfterAll，@BeforeEach或@AfterEach注解的方法不能有返回值。

元注解和组合注解

JUnit Jupiter注解可以用作元注解。这意味着您可以定义自己的组合注释，它将自动继承其元注释的语义。

例如，您可以像下面那样创建一个名为 `@Fast` 的自定义组合注释，而不必在整个代码库（请参阅标签和过滤）中复

制和粘贴 `@Tag("fast")` 。然后`@Fast`可以用作 `@Tag("fast")` 的一个替代品。

标准测试类

- 标准测试类

标准测试类

标准测试案例：

```
1.  import static org.junit.jupiter.api.Assertions.fail;
2.
3.  import org.junit.jupiter.api.AfterAll;
4.  import org.junit.jupiter.api.AfterEach;
5.  import org.junit.jupiter.api.BeforeAll;
6.  import org.junit.jupiter.api.BeforeEach;
7.  import org.junit.jupiter.api.Disabled;
8.  import org.junit.jupiter.api.Test;
9.
10. class StandardTests {
11.
12.     @BeforeAll
13.     static void initAll() {
14.     }
15.
16.     @BeforeEach
17.     void init() {
18.     }
19.
20.     @Test
21.     void succeedingTest() {
22.     }
23.
24.     @Test
25.     void failingTest() {
26.         fail("a failing test");
27.     }
28.
29.     @Test
30.     @Disabled("for demonstration purposes")
31.     void skippedTest() {
32.         // not executed
33.     }
34.
35.     @AfterEach
36.     void tearDown() {
37.     }
38.
39.     @AfterAll
```

```
40.     static void tearDownAll() {
41.     }
42.
43. }
```

> 测试类和测试方法都不必是 `public` 。

显示名称

- 显示名称

显示名称

测试类和测试方法可以声明自定义显示名称 - 使用空格，特殊字符，甚至emojis表情符号 - 将由测试runner和测试报告显示。

```
1.  import org.junit.jupiter.api.DisplayName;
2.  import org.junit.jupiter.api.Test;
3.
4.  @DisplayName("A special test case")
5.  class DisplayNameDemo {
6.
7.      @Test
8.      @DisplayName("Custom test name containing spaces")
9.      void testWithDisplayNameContainingSpaces() {
10.     }
11.
12.     @Test
13.     @DisplayName("╯°□°）╯")
14.     void testWithDisplayNameContainingSpecialCharacters() {
15.     }
16.
17.     @Test
18.     @DisplayName("?")
19.     void testWithDisplayNameContainingEmoji() {
20.     }
21.
22. }
```

断言

- 断言
  - 第三方断言类库

断言

JUnit Jupiter提供了许多JUnit4已有的断言方法，并增加了一些适合与Java 8 lambda一起使用的断言方法。所有JUnit Jupiter断言都是org.junit.jupiter.Assertions类中的静态方法。

```
1.  import static java.time.Duration.ofMillis;
2.  import static java.time.Duration.ofMinutes;
3.  import static org.junit.jupiter.api.Assertions.assertAll;
4.  import static org.junit.jupiter.api.Assertions.assertEquals;
5.  import static org.junit.jupiter.api.Assertions.assertNotNull;
6.  import static org.junit.jupiter.api.Assertions.assertThrows;
7.  import static org.junit.jupiter.api.Assertions.assertTimeout;
8.  import static org.junit.jupiter.api.Assertions.assertTimeoutPreemptively;
9.  import static org.junit.jupiter.api.Assertions.assertTrue;
10.
11. import org.junit.jupiter.api.Test;
12.
13. class AssertionsDemo {
14.
15.     @Test
16.     void standardAssertions() {
17.         assertEquals(2, 2);
18.         assertEquals(4, 4, "The optional assertion message is now the last parameter.");
19.         assertTrue(2 == 2, () -> "Assertion messages can be lazily evaluated -- "
20.                 + "to avoid constructing complex messages unnecessarily.");
21.     }
22.
23.     @Test
24.     void groupedAssertions() {
25.         // In a grouped assertion all assertions are executed, and any
26.         // failures will be reported together.
27.         assertAll("person",
28.             () -> assertEquals("John", person.getFirstName()),
29.             () -> assertEquals("Doe", person.getLastName())
30.         );
31.     }
32.
33.     @Test
34.     void dependentAssertions() {
35.         // Within a code block, if an assertion fails the
36.         // subsequent code in the same block will be skipped.
37.         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!";
    }

}
```

第三方断言类库

虽然JUnit Jupiter提供的断言功能足以满足许多测试场景的需要，但是有时需要更强大和附加功能，例如匹配器。在这种情况下，JUnit小组推荐使用AssertJ，Hamcrest，Truth等第三方断言库。开发人员可以自由使用他们选择的断言库。

例如，匹配器和fluent API的组合可以用来使断言更具描述性和可读性。但是，JUnit Jupiter的org.junit.jupiter.Assertions类没有提供类似于JUnit 4的 `org.junit.Assert` 类的 `assertThat()` 方法，以接受Hamcrest Matcher。相反，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。

以下示例演示如何在JUnit Jupiter测试中使用来自Hamcrest的 `assertThat()` 支持。 只要Hamcrest库已经添加到classpath中，就可以静态地导入诸如 `assertThat()` ， `is()` 和 `equalTo()` 之类的方法，然后像下面

的 `assertWithHamcrestMatcher()` 方法那样在测试中使用它们。

```
1.  import static org.hamcrest.CoreMatchers.equalTo;
2.  import static org.hamcrest.CoreMatchers.is;
3.  import static org.hamcrest.MatcherAssert.assertThat;
4.
5.  import org.junit.jupiter.api.Test;
6.
7.  class HamcrestAssertionDemo {
8.
9.      @Test
10.     void assertWithHamcrestMatcher() {
11.         assertThat(2 + 1, is(equalTo(3)));
12.     }
13.
14. }
```

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

假设

- 假设

假设

JUnit Jupiter附带了JUnit4提供的一些assumption方法的子集，并增加了一些适合与Java 8 lambda一起使用的方法。所有的JUnit Jupiter assumption都是 `org.junit.jupiter.Asumptions` 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");
    }

}
```

禁用

- 禁用

禁用

这是一个禁用的测试案例：

```
1.  import org.junit.jupiter.api.Disabled;
2.  import org.junit.jupiter.api.Test;
3.
4.  @Disabled
5.  class DisabledClassDemo {
6.      @Test
7.      void testWillBeSkipped() {
8.      }
9.  }
```

这是一个带有禁用测试方法的测试案例：

```
1.  import org.junit.jupiter.api.Disabled;
2.  import org.junit.jupiter.api.Test;
3.
4.  class DisabledTestsDemo {
5.
6.      @Disabled
7.      @Test
8.      void testWillBeSkipped() {
9.      }
10.
11.     @Test
12.     void testWillBeExecuted() {
13.     }
14. }
```

标签和过滤

- 标签和过滤
  - 标签的语法规则

标签和过滤

测试类和方法可以打标签。这些标签以后可以用来过滤测试发现和执行。

标签的语法规则

- 标签不能为 `null` 或空白。
- 裁剪标签不能包含空格(whitespace)。
- 裁剪标签不得包含ISO控制字符。
- 裁剪标签不得包含以下任何保留字符。
  - `,` , `(` , `)` , `&` , `|` , `!`

> 在上面的上下文中，“裁剪”意思是前后的空白字符已被删除。

```
1.  import org.junit.jupiter.api.Tag;
2.  import org.junit.jupiter.api.Test;
3.
4.  @Tag("fast")
5.  @Tag("model")
6.  class TaggingDemo {
7.
8.      @Test
9.      @Tag("taxes")
10.     void testingTaxCalculation() {
11.     }
12.
13. }
```

测试实例生命周期

- 测试实例生命周期
  - 修改默认测试实例生命周期

测试实例生命周期

为了允许隔离执行单个的测试方法，并避免由于可变测试实例状态而产生的意外副作用，JUnit在执行每个测试方法之前创建每个测试类的新实例（请参阅下面的讲解，何为测试方法）。这个”per-method”测试实例生命周期是JUnit Jupiter中的默认行为，类似于JUnit以前的所有版本。

如果您希望JUnit Jupiter在同一个测试实例上执行所有测试方法，只需使用 `@TestInstance(Lifecycle.PER_CLASS)` 对您的测试类进行注解即可。当使用这种模式时，每个测试类将创建一个新的测试实例。因此，如果您的测试方法依赖于存储在实例变量中的状态，则可能需要在 `@BeforeEach` 或 `@AfterEach` 方法中重置该状态。

“per-class”模式比默认的”per-method”模式有一些额外的好处。具体来说，使用”per-class”模式，可以在非静态方法和接口默认方法上声明 `@BeforeAll` 和 `@AfterAll` 。因此，”per-class”模式也可以在 `@Nested` 测试类中使用 `@BeforeAll` 和 `@AfterAll` 方法。

如果使用Kotlin编程语言编写测试，则可能会发现，通过切换到”per-class”测试实例生命周期模式，可以更轻松地实现 `@BeforeAll` 和 `@AfterAll` 方法。

> 在测试实例生命周期的上下文中，测试方法是用 *`@Test`* ， *`@RepeatedTest`* ， *`@ParameterizedTest`* ， *`@TestFactory`* 或 *`@TestTemplate`* 注解的任何方法。

修改默认测试实例生命周期

如果测试类或测试接口没有用 `@TestInstance` 注解，JUnit Jupiter将使用默认的生命周期模式。标准默认模式是 `PER_METHOD` ；但是，可以更改整个测试计划执行的默认值。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，忽略大小写。这可以作为JVM系统属性提供，作为传递给 `Launcher` 的 `LauncherDiscoveryRequest` 中的配置参数，或通过JUnit Platform配置文件（请参阅配置参数了解详细信息）。

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

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

但是请注意，通过JUnit Platform配置文件设置默认测试实例生命周期模式是一个更健壮的解决方案，因为可以将配置文件与项目一起检入到版本控制系统中，因此可以在IDE和构建软件中使用。

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class path 的根目录（例如 `src/test/resources` ）中使用以下内容创建名为 `junit-platform.properties` 的文件。

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

> 更改默认的测试实例生命周期模式可能会导致不可预测的结果和脆弱的构建，如果应用的不一致。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。

嵌套测试

- 嵌套测试

嵌套测试

嵌套测试给测试编写者更多的能力，来表达几组测试之间的关系。这里有一个详细的例子。

用于测试stack的嵌套测试套件：

```
1.  import static org.junit.jupiter.api.Assertions.assertEquals;
2.  import static org.junit.jupiter.api.Assertions.assertFalse;
3.  import static org.junit.jupiter.api.Assertions.assertThrows;
4.  import static org.junit.jupiter.api.Assertions.assertTrue;
5.
6.  import java.util.EmptyStackException;
7.  import java.util.Stack;
8.
9.  import org.junit.jupiter.api.BeforeEach;
10. import org.junit.jupiter.api.DisplayName;
11. import org.junit.jupiter.api.Nested;
12. import org.junit.jupiter.api.Test;
13.
14. @DisplayName("A stack")
15. class TestingAStackDemo {
16.
17.     Stack<Object> stack;
18.
19.     @Test
20.     @DisplayName("is instantiated with new Stack()")
21.     void isInstantiatedWithNew() {
22.         new Stack<>();
23.     }
24.
25.     @Nested
26.     @DisplayName("when new")
27.     class WhenNew {
28.
29.         @BeforeEach
30.         void createNewStack() {
31.             stack = new Stack<>();
32.         }
33.
34.         @Test
35.         @DisplayName("is empty")
36.         void isEmpty() {
37.             assertTrue(stack.isEmpty());
```

```
38.         }
39.
40.         @Test
41.         @DisplayName("throws EmptyStackException when popped")
42.         void throwsExceptionWhenPopped() {
43.             assertThrows(EmptyStackException.class, () -> stack.pop());
44.         }
45.
46.         @Test
47.         @DisplayName("throws EmptyStackException when peeked")
48.         void throwsExceptionWhenPeeked() {
49.             assertThrows(EmptyStackException.class, () -> stack.peek());
50.         }
51.
52.         @Nested
53.         @DisplayName("after pushing an element")
54.         class AfterPushing {
55.
56.             String anElement = "an element";
57.
58.             @BeforeEach
59.             void pushAnElement() {
60.                 stack.push(anElement);
61.             }
62.
63.             @Test
64.             @DisplayName("it is no longer empty")
65.             void isNotEmpty() {
66.                 assertFalse(stack.isEmpty());
67.             }
68.
69.             @Test
70.             @DisplayName("returns the element when popped and is empty")
71.             void returnElementWhenPopped() {
72.                 assertEquals(anElement, stack.pop());
73.                 assertTrue(stack.isEmpty());
74.             }
75.
76.             @Test
77.             @DisplayName("returns the element when peeked but remains not empty")
78.             void returnElementWhenPeeked() {
79.                 assertEquals(anElement, stack.peek());
80.                 assertFalse(stack.isEmpty());
81.             }
82.         }
83.     }
84. }
```

构造函数和方法的依赖注入

- 构造函数和方法的依赖注入

构造函数和方法的依赖注入

在之前的所有JUnit版本中，测试构造函数或方法都不允许有参数（至少不能使用标准的Runner实现）。作为JUnit Jupiter的主要变化之一，测试构造函数和方法现在都允许有参数。这带来了更大的灵活性，并为构造函数和方法启用依赖注入。

`ParameterResolver` 定义了测试扩展的API，希望在运行时动态解析参数。如果测试构造函数或 `@Test` ， `@TestFactory` ， `@BeforeEach` ， `@AfterEach` ， `@BeforeAll` 或 `@AfterAll` 方法接受参数，则该参数必须在运行时由已注册的 `ParameterResolver` 解析。

目前有三个自动注册的内置解析器。

- `TestInfoParameterResolver`

  如果方法参数是TestInfo类型，则 `TestInfoParameterResolver` 将提供一个TestInfo的实例，对应当前测试，作为参数的值。然后TestInfo可以用来获取有关当前测试的信息，例如测试的显示名称，测试类，测试方法或关联的标签。显示名称可以是技术名称，例如测试类或测试方法的名称，也可以是通过 `@DisplayName` 配置的自定义名称。

  TestInfo充当JUnit4的 `TestName` 规则的替换品。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。

```
1.  import static org.junit.jupiter.api.Assertions.assertEquals;
2.  import static org.junit.jupiter.api.Assertions.assertTrue;
3.
4.  import org.junit.jupiter.api.BeforeEach;
5.  import org.junit.jupiter.api.DisplayName;
6.  import org.junit.jupiter.api.Tag;
7.  import org.junit.jupiter.api.Test;
8.  import org.junit.jupiter.api.TestInfo;
9.
10. @DisplayName("TestInfo Demo")
11. class TestInfoDemo {
12.
13.     TestInfoDemo(TestInfo testInfo) {
14.         assertEquals("TestInfo Demo", testInfo.getDisplayName());
15.     }
16.
17.     @BeforeEach
18.     void init(TestInfo testInfo) {
19.         String displayName = testInfo.getDisplayName();
20.         assertTrue(displayName.equals("TEST 1") || displayName.equals("test2()"));
21.     }
22.
```

```
23.         @Test
24.         @DisplayName("TEST 1")
25.         @Tag("my-tag")
26.         void test1(TestInfo testInfo) {
27.             assertEquals("TEST 1", testInfo.getDisplayName());
28.             assertTrue(testInfo.getTags().contains("my-tag"));
29.         }
30.
31.         @Test
32.         void test2() {
33.         }
34.
35.     }
```

- `RepetitionInfoParameterResolver`

  如果 `@RepeatedTest` ， `@BeforeEach` 或 `@AfterEach` 方法中的方法参数类型为RepetitionInfo，则 `RepetitionInfoParameterResolver` 将提供 `RepetitionInfo` 的实例。然后可以使用 `RepetitionInfo` 获取当前重复信息以及相应的 `@RepeatedTest` 的重复总数。但是请注意， `RepetitionInfoParameterResolver` 不在 `@RepeatedTest` 的上下文之外注册。请参阅重复测试示例。

- `TestReporterParameterResolver`

  如果方法参数的类型是TestReporter， `TestReporterParameterResolver` 将提供一个 `TestReporter` 的实例。 `TestReporter` 可用于发布有关当前测试运行的额外数据。数据可以通过 `TestExecutionListener.reportingEntryPublished()` 来使用，因此可以被IDE查看或包含在报告中。

  在JUnit Jupiter中，当你需要打印信息时，就像在JUnit4使用 `stdout` 或 `stderr` ，你应该使用 `TestReporter` 。使用 `@RunWith(JUnitPlatform.class)` 甚至会将所有报告的条目输出到 `stdout` 。

```
1.  import java.util.HashMap;
2.
3.  import org.junit.jupiter.api.Test;
4.  import org.junit.jupiter.api.TestReporter;
5.
6.  class TestReporterDemo {
7.
8.      @Test
9.      void reportSingleValue(TestReporter testReporter) {
10.         testReporter.publishEntry("a key", "a value");
11.     }
12.
13.     @Test
14.     void reportSeveralValues(TestReporter testReporter) {
15.         HashMap<String, String> values = new HashMap<>();
16.         values.put("user name", "dk38");
```

```
            values.put("award year", "1974");

            testReporter.publishEntry(values);
        }

    }
```

> 其他参数解析器必须通过 `@ExtendWith` 注册适当的扩展来显式启用。

查看MockitoExtension获取自定义 `ParameterResolver` 的示例。虽然不打算生产就绪，它展示了扩展模型和参数解析过程的简单性和表达性。 `MyMockitoTest` 演示了如何将Mockito mock注入 `@BeforeEach` 和 `@Test` 方法。

```
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());
    }

}
```

测试接口和默认方法

- 测试接口和默认方法

测试接口和默认方法

JUnit Jupiter允许在接口 `default` 方法中声明 `@Test` , `@RepeatedTest` , `@ParameterizedTest` , `@TestFactory` , `@TestTemplate` , `@BeforeEach` 和 `@AfterEach` 。如果测试接口或测试类用 `@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 `default` 方法上声明 `@BeforeAll` 和 `@AfterAll` 。这里有些例子。

```
@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其注解和扩展。请参阅测试执行前后的回调来获取TimingExtension的源代码。

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

在你的测试类中，你可以实现这些测试接口来应用它们。

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

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

}
```

运行 `TestInterfaceDemo` ，输出类似以下内容：

```
: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
```

这个特性的另一个可以应用的地方是为接口契约编写测试。例如，您可以编写测试，验证 `Object.equals` 或 `Comparable.compareTo` 的实现应该如何如下表现。

```
public interface Testable<T> {

    T createValue();

}
```

```
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);
    }

}
```

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

    T createSmallerValue();
```

```
4.
5.      @Test
6.      default void returnsZeroWhenComparedToItself() {
7.          T value = createValue();
8.          assertEquals(0, value.compareTo(value));
9.      }
10.
11.     @Test
12.     default void returnsPositiveNumberComparedToSmallerValue() {
13.         T value = createValue();
14.         T smallerValue = createSmallerValue();
15.         assertTrue(value.compareTo(smallerValue) > 0);
16.     }
17.
18.     @Test
19.     default void returnsNegativeNumberComparedToSmallerValue() {
20.         T value = createValue();
21.         T smallerValue = createSmallerValue();
22.         assertTrue(smallerValue.compareTo(value) < 0);
23.     }
24.
25. }
```

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。当然你必须实现抽象方法。

```
1.  class StringTests implements ComparableContract<String>, EqualsContract<String> {
2.
3.      @Override
4.      public String createValue() {
5.          return "foo";
6.      }
7.
8.      @Override
9.      public String createSmallerValue() {
10.         return "bar"; // 'b' < 'f' in "foo"
11.     }
12.
13.     @Override
14.     public String createNotEqualValue() {
15.         return "baz";
16.     }
17.
18. }
```

> 上述测试仅仅是作为例子，因此不完整。

重复测试

- 重复测试
  - 重复测试示例

重复测试

JUnit Jupiter通过使用 `@RepeatedTest` 注解方法并指定所需的重复次数，提供了重复测试指定次数的功能。每次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。

以下示例演示了如何声明名为 `repeatedTest()` 的测试，该测试将自动重复10次。

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

除了指定重复次数外，还可以通过 `@RepeatedTest` 注解的 `name` 属性为每次重复配置自定义显示名称。此外，显示名称可以是模式，由静态文本和动态占位符的组合而成。目前支持以下占位符：

- {displayName}: `@RepeatedTest`方法的显示名称
- {currentRepetition}: 当前重复次数
- {totalRepetitions}: 重复的总次数

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，前面的 `repeatedTest()` 示例的单个重复的显示名称将是： `repetition 1 of 1` 0， `repetition 2 of 10` 等等。如果您希望每个重复的名称中包含 `@RepeatedTest` 方法的显示名称，您可以定义自己的自定义模式或使用预定义的 `RepeatedTest.LONG_DISPLAY_NAME` 模式。后者相当于 `"{displayName} :: repetition {currentRepetition} of {totalRepetitions}"` ，这会导致重复测试的显示名称变成这样： `repeatedTest() :: repetition 1 of 10` ， `repeatedTest() :: repetition 2 of 10` 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 `RepetitionInfo` 的实例注入 `@RepeatedTest` ， `@BeforeEach` 或 `@AfterEach` 方法。

重复测试示例

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` 的实例注入到测试中，以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()` 用自定义模式组合自定义显示名称，然后使用 `TestInfo` 来验证生成的显示名称的格式。 `Repeat!` 是来自 `@DisplayName` ，来自 `{displayName}` 声明， `1/1` 来自 `{currentRepetition}/{totalRepetitions}` 。相反， `customDisplayNameWithLongPattern()` 使用前面提到

的预定义的 `RepeatedTest.LONG_DISPLAY_NAME` 模式。

`repeatedTestInGerman()` 展示了将重复测试的显示名称翻译成外语的能力 - 在这种情况下是德语，从而得到单个重复的名称，例如： `Wiederholung 1 von 5` ， `Wiederholung 2 von 5` 等。

由于 `beforeEach()` 方法用 `@BeforeEach` 标注，所以在每次重复测试之前都会执行它。通过将 `TestInfo` 和 `RepetitionInfo` 注入到方法中，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。在启用了 `INFO` 日志级别的情况下，执行 `RepeatedTestsDemo` 会得到以下输出。

```
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
```

```
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
```

```
16.     void beforeEach(TestInfo testInfo, RepetitionInfo repetitionInfo) {
17.         int currentRepetition = repetitionInfo.getCurrentRepetition();
18.         int totalRepetitions = repetitionInfo.getTotalRepetitions();
19.         String methodName = testInfo.getTestMethod().get().getName();
20.         logger.info(String.format("About to execute repetition %d of %d for %s", //
21.             currentRepetition, totalRepetitions, methodName));
22.     }
23.
24.     @RepeatedTest(10)
25.     void repeatedTest() {
26.         // ...
27.     }
28.
29.     @RepeatedTest(5)
30.     void repeatedTestWithRepetitionInfo(RepetitionInfo repetitionInfo) {
31.         assertEquals(5, repetitionInfo.getTotalRepetitions());
32.     }
33.
34.     @RepeatedTest(value = 1, name = "{displayName} {currentRepetition}/{totalRepetitions}")
35.     @DisplayName("Repeat!")
36.     void customDisplayName(TestInfo testInfo) {
37.         assertEquals(testInfo.getDisplayName(), "Repeat! 1/1");
38.     }
39.
40.     @RepeatedTest(value = 1, name = RepeatedTest.LONG_DISPLAY_NAME)
41.     @DisplayName("Details...")
42.     void customDisplayNameWithLongPattern(TestInfo testInfo) {
43.         assertEquals(testInfo.getDisplayName(), "Details... :: repetition 1 of 1");
44.     }
45.
46.     @RepeatedTest(value = 5, name = "Wiederholung {currentRepetition} von
    {totalRepetitions}")
47.     void repeatedTestInGerman() {
48.         // ...
49.     }
50.
51. }
```

在启用了unicode主题的情况下，使用 `ConsoleLauncher` 或 `junitPlatformTest` Gradle插件时，执行 `RepeatedTestsDemo` 会将以下输出给控制台。

```
1. ├─ RepeatedTestsDemo ✔
2. │  ├─ repeatedTest() ✔
3. │  │  ├─ repetition 1 of 10 ✔
4. │  │  ├─ repetition 2 of 10 ✔
5. │  │  ├─ repetition 3 of 10 ✔
6. │  │  ├─ repetition 4 of 10 ✔
7. │  │  ├─ 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 ✔
```

参数化测试

- 参数化测试
  - 必须的设置
  - 参数来源
    - @ValueSource
    - @EnumSource
  - @MethodSource
    - @CsvSource
    - @CsvFileSource
    - @ArgumentsSource
  - 参数转换
    - 隐式转换
    - 显式转换
  - 自定义显示名称
  - 生命周期和互操作性

参数化测试

参数化测试可以用不同的参数多次运行测试。它们和普通的 `@Test` 方法一样声明，但是使用 `@ParameterizedTest` 注解。另外，您必须声明至少一个将为每次调用提供参数的来源*(source)*。

> 参数化测试目前是实验性功能。有关详细信息，请参阅实验性*API*中的表格。

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

这个参数化的测试使用 `@ValueSource` 注解来指定一个 `String` 数组作为参数的来源。执行上述方法时，每次调用将分别报告。例如， `ConsoleLauncher` 将打印输出类似于以下内容。

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

必须的设置

为了使用参数化测试，您需要添加对 `junit-jupiter-params` 构建的依赖。有关详细信息，请参阅依赖元数据。

参数来源

JUnit Jupiter开箱即用，提供了不少*source*注解。下面的每个小节都为他们提供了简要的概述和示例。请参

阅 `org.junit.jupiter.params.provider` 包中的JavaDoc以获取更多信息。

@ValueSource

`@ValueSource` 是最简单的source之一。它可以让你指定一个原生类型（`String`，`int`，`long`或`double`）的数组，并且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` 提供了一个使用 `Enum` 常量的简便方法。该注释提供了一个可选的 `name` 参数，可以指定使用哪些常量。如果省略，所有的常量将被用在下面的例子中。

```
@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，可以对将哪些常量传递给测试方法进行细化控制。例如，您可以从枚举常量池中排除名称或指定正则表达式，如下例所示。

```
@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` 允许你引用一个或多个测试类的工厂方法。这样的方法必须返回一个 `Stream` ， `Iterable` ， `Iterator` 或者参数数组。另外，这种方法不能接受任何参数。默认情况下，除非测试类用 `@TestInstance(Lifecycle.PER_CLASS)` 注解，否则这些方法必须是静态的。

如果只需要一个参数，则可以返回参数类型的实例 `Stream` ，如以下示例所示。

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

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

支持原始类型（ `DoubleStream` ， `IntStream` 和 `LongStream` ）的流，示例如下：

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

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

如果测试方法声明多个参数，则需要返回一个集合或 `Arguments` 实例流，如下所示。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` 允许您将参数列表表示为以逗号分隔的值（例如，字符串文字）。

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

`@CsvSource` 使用 `'` 作为转义字符。 请参阅上述示例和下表中的’baz, qux’值。 一个空的引用值 `''` 会导致一个空的 `String` ；而一个完全空的值被解释为一个null引用。如果null引用的目标类型是基本类型，则引发 `ArgumentConversionException` 。

示例输入	结果字符列表
@CsvSource({ “foo, bar” })	`"foo"`, `"bar"`
@CsvSource({ “foo, ‘baz, qux’” })	`"foo"`, `"baz, qux"`
@CsvSource({ “foo, ‘’” })	`"foo"`, `""`
@CsvSource({ “foo, “ })	`"foo"`, null

@CsvFileSource

`@CsvFileSource` 让你使用classpath中的CSV文件。CSV文件中的每一行都会导致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 `"` 作为转义字符，请参阅上面例子中的 `"baz, qux"` 值，一个空的转义值 `""` 会产生一个空字符串， 一个完全为空的值被解释为null引用，如果

null引用的目标类型是基本类型，则引发 `ArgumentConversionException` 。

@ArgumentsSource

可以使用 `@ArgumentsSource` 指定一个自定义的，可重用的 `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);
    }
}
```

参数转换

隐式转换

为了支持像 `@CsvSource` 这样的情况，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 `@ParameterizedTest` 声明 `TimeUnit` 类型的参数，并且声明的源提供的实际类型是 `String` ，则该字符串将自动转换为相应的 `TimeUnit` 枚举常量。

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

`String` 实例目前隐式转换为以下目标类型。

目标类型	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

您可以使用 `@ConvertWith` 注解来显式指定 `ArgumentConverter` 来用于某个参数，而不是像下面的示例那样使用隐式参数转换。

```
@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由测试作者实现。因此， `junit-jupiter-params` 只提供一个显式的参数转换器，可以作为参考实现： `JavaTimeArgumentConverter` 。通过组合的注解 `JavaTimeConversionPattern` 使用。

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

```
}
```

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该特定调用的所有参数的调用索引和字符串表示。 但是，您可以通过 `@ParameterizedTest` 注解的 `name` 属性自定义调用显示名称，如下所示：

```
@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) {
}
```

当使用 `ConsoleLauncher` 执行上述方法时，您将看到类似于以下内容的输出。

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

自定义显示名称中支持以下占位符。

占位符	描述
`{index}`	当前调用下标(从1开始)
`{arguments}`	完成的，逗号分隔的参数列表
`{0}`, `{1}`, …	单个参数

生命周期和互操作性

参数化测试的每个调用与普通的 `@Test` 方法具有相同的生命周期。例如， `@BeforeEach` 方法将在每次调用之前执行。与动态测试类似，调用将逐个出现在IDE的测试树中。可能会在同一个测试类中混合常规的 `@Test` 方法和 `@ParameterizedTest` 方法。

可以在 `@ParameterizedTest` 方法中使用 `ParameterResolver` 扩展。但是，由参数来源解析的方法参数需要先在参数列表中找到。由于测试类可能包含常规测试，以及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) {
    // ...
}
```

测试模板

- 测试模板

测试模板

`@TestTemplate` 方法不是常规的测试用例，而是测试用例的模板。为此，它被设计成根据已注册的提供者返回的调用上下文的数量被调用多次。因此，它必须与注册的 `TestTemplateInvocationContextProvider` 扩展一起使用。测试模板方法的每次调用都像执行常规 `@Test` 方法一样，完全支持相同的生命周期回调和扩展。有关使用示例，请参阅为测试模板提供调用上下文。

动态测试

- 动态测试
  - 动态测试示例

动态测试

注解中描述的JUnit Jupiter中的标准 `@Test` 注解与JUnit 4中的 `@Test` 注解非常相似。两者都描述了实现测试用例的方法。这些测试用例是静态的，因为它们是在编译时完全指定的，而且它们的行为不能由运行时发生的任何事情来改变。 *`Assumption`* 提供了一种基本的动态行为形式，但是刻意在表达方面受到限制。

除了这些标准测试外，JUnit Jupiter还引入了一种全新的测试编程模型。这种新的测试是 `动态测试` ，它是由 `@TestFactory` 注解的工厂方法在运行时生成的。

与 `@Test` 方法相比， `@TestFactory` 方法本身不是测试用例，而是测试用例的工厂。因此，动态测试是工厂的产物。从技术上讲， `@TestFactory` 方法必须返回 `DynamicNode` 实例的 `Stream` ， `Collection` ， `Iterable` 或 `Iterator` 。 `DynamicNode` 的可实例化的子类是 `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，可以创建任意嵌套的动态节点层次结构。然后， `DynamicTest` 实例将被延迟执行，从而实现测试用例的动态甚至非确定性生成。

任何由 `@TestFactory` 返回的 `Stream` 都要通过调用 `stream.close()` 来正确关闭，使得使用诸如 `Files.lines()` 之类的资源变得安全。

与 `@Test` 方法一样， `@TestFactory` 方法不能是 `private` 或 `static` ，并且可以选择声明参数，以便通过 `ParameterResolvers` 解析。

`DynamicTest` 是运行时生成的测试用例。它由显示名称和 `Executable` 组成。 `Executable` 是 `@FunctionalInterface` ，这意味着动态测试的实现可以作为*lambda*表达式或方法引用来提供。

> 动态生命周期
>
> 动态测试的执行生命周期与标准的 *`@Test`* 情况完全不同。具体而言，个别动态测试没有生命周期回调。这意味着 *`@BeforeEach`* 和 *`@AfterEach`* 方法及其相应的扩展回调函数是为 *`@TestFactory`* 方法执行，而不是对每个动态测试执行。换句话说，如果您从一个*lambda*表达式的测试实例中访问动态测试的字段，这些字段将不会由同一个 *`@TestFactory`* 方法生成的各个动态测试之间的回调方法或扩展重置。

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，在稍后的发行版中，这可以通过注册设施来补充。

> 动态测试目前是实验性功能。有关详细信息，请参阅实验性*API*中的表格。

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。

第一种方法返回无效的返回类型。由于在编译时无法检测到无效的返回类型，因此在运行时检测并抛出 `JUnitException` 异常。

接下来的五个方法是非常简单的例子，演示了 `Collection` ， `Iterable` ， `Iterator` 或者 `DynamicTest` 实例的生成。这些例子中的大多数并不真正表现出动态行为，而只是在原则上展示了支持的返回类型。
而 `dynamicTestsFromStream()` 和 `dynamicTestsFromIntStream()` 演示了如何为给定的一组字符串或一组输入数字生成动态测试是何等的简单。

下一个方法本质上是真正动态的。 `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，然后将这三者全部提供给 `DynamicTest.stream()` 。尽管 `generateRandomNumberOfTests()` 的非确定性行为理所当然的会与测试的可重复性相冲突，应谨慎使用，它可以演示动态测试的表现力和力量。

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

```
1.  import static org.junit.jupiter.api.Assertions.assertEquals;
2.  import static org.junit.jupiter.api.Assertions.assertFalse;
3.  import static org.junit.jupiter.api.Assertions.assertNotNull;
4.  import static org.junit.jupiter.api.Assertions.assertTrue;
5.  import static org.junit.jupiter.api.DynamicContainer.dynamicContainer;
6.  import static org.junit.jupiter.api.DynamicTest.dynamicTest;
7.
8.  import java.util.Arrays;
9.  import java.util.Collection;
10. import java.util.Iterator;
11. import java.util.List;
12. import java.util.Random;
13. import java.util.function.Function;
14. import java.util.stream.IntStream;
15. import java.util.stream.Stream;
16.
17. import org.junit.jupiter.api.DynamicNode;
18. import org.junit.jupiter.api.DynamicTest;
19. import org.junit.jupiter.api.Tag;
20. import org.junit.jupiter.api.TestFactory;
21. import org.junit.jupiter.api.function.ThrowingConsumer;
22.
23. class DynamicTestsDemo {
24.
25.     // This will result in a JUnitException!
26.     @TestFactory
27.     List<String> dynamicTestsWithInvalidReturnType() {
28.         return Arrays.asList("Hello");
29.     }
30.
31.     @TestFactory
32.     Collection<DynamicTest> dynamicTestsFromCollection() {
33.         return Arrays.asList(
34.             dynamicTest("1st dynamic test", () -> assertTrue(true)),
35.             dynamicTest("2nd dynamic test", () -> assertEquals(4, 2 * 2))
36.         );
```

```
37.     }
38.
39.     @TestFactory
40.     Iterable<DynamicTest> dynamicTestsFromIterable() {
41.         return Arrays.asList(
42.             dynamicTest("3rd dynamic test", () -> assertTrue(true)),
43.             dynamicTest("4th dynamic test", () -> assertEquals(4, 2 * 2))
44.         );
45.     }
46.
47.     @TestFactory
48.     Iterator<DynamicTest> dynamicTestsFromIterator() {
49.         return Arrays.asList(
50.             dynamicTest("5th dynamic test", () -> assertTrue(true)),
51.             dynamicTest("6th dynamic test", () -> assertEquals(4, 2 * 2))
52.         ).iterator();
53.     }
54.
55.     @TestFactory
56.     Stream<DynamicTest> dynamicTestsFromStream() {
57.         return Stream.of("A", "B", "C")
58.             .map(str -> dynamicTest("test" + str, () -> { /* ... */ }));
59.     }
60.
61.     @TestFactory
62.     Stream<DynamicTest> dynamicTestsFromIntStream() {
63.         // Generates tests for the first 10 even integers.
64.         return IntStream.iterate(0, n -> n + 2).limit(10)
65.             .mapToObj(n -> dynamicTest("test" + n, () -> assertTrue(n % 2 == 0)));
66.     }
67.
68.     @TestFactory
69.     Stream<DynamicTest> generateRandomNumberOfTests() {
70.
71.         // Generates random positive integers between 0 and 100 until
72.         // a number evenly divisible by 7 is encountered.
73.         Iterator<Integer> inputGenerator = new Iterator<Integer>() {
74.
75.             Random random = new Random();
76.             int current;
77.
78.             @Override
79.             public boolean hasNext() {
80.                 current = random.nextInt(100);
81.                 return current % 7 != 0;
82.             }
83.
84.             @Override
```

```
85.             public Integer next() {
86.                 return current;
87.             }
88.         };
89.
90.         // Generates display names like: input:5, input:37, input:85, etc.
91.         Function<Integer, String> displayNameGenerator = (input) -> "input:" + input;
92.
93.         // Executes tests based on the current input value.
94.         ThrowingConsumer<Integer> testExecutor = (input) -> assertTrue(input % 7 != 0);
95.
96.         // Returns a stream of dynamic tests.
97.         return DynamicTest.stream(inputGenerator, displayNameGenerator, testExecutor);
98.     }
99.
100.     @TestFactory
101.     Stream<DynamicNode> dynamicTestsWithContainers() {
102.         return Stream.of("A", "B", "C")
103.             .map(input -> dynamicContainer("Container " + input, Stream.of(
104.                 dynamicTest("not null", () -> assertNotNull(input)),
105.                 dynamicContainer("properties", Stream.of(
106.                     dynamicTest("length > 0", () -> assertTrue(input.length() > 0)),
107.                     dynamicTest("not empty", () -> assertFalse(input.isEmpty()))
108.                 ))
109.             )));
110.     }
111.
112. }
```

运行测试

- 运行测试

运行测试

IDE支持

- IDE支持

IDE支持

构建支持

- 构建支持
    - Gradle
    - Maven

构建支持

Gradle

Maven

控制台启动器

- 控制台启动器

控制台启动器

http://junit.org/junit5/docs/current/user-guide/#running-tests-console-launcher

使用Junit4运行JUnit Platfrom

- 使用Junit4运行JUnit Platfrom

使用Junit4运行JUnit Platfrom

配置参数

- 配置参数

配置参数

扩展模型

- 扩展模型

扩展模型

概述

- 概述

概述

注册扩展

- 注册扩展

注册扩展

有条件的测试执行

- 有条件的测试执行

有条件的测试执行

测试实例后处理

- 测试实例后处理

测试实例后处理

测试生命周期回调

- 测试生命周期回调

测试生命周期回调

异常处理

- 异常处理

异常处理

为测试模板提供调用上下文

- 为测试模板提供调用上下文

为测试模板提供调用上下文

在扩展中维持状态

- 在扩展中维持状态

在扩展中维持状态

在扩展中支持的实用程序

- 在扩展中支持的实用程序

在扩展中支持的实用程序

用户代码和扩展的相对执行顺序

- 用户代码和扩展的相对执行顺序

用户代码和扩展的相对执行顺序

从Junit4迁移

- 从Junit4迁移

从Junit4迁移

在Junit Platform上运行JUnit4测试

- 在Junit Platform上运行JUnit4测试

在Junit Platform上运行JUnit4测试

迁移Tips

- 迁移Tips

迁移Tips

受限的**JUnit4**规则支持

- 受限的JUnit4规则支持

受限的JUnit4规则支持

高级主题

- 高级主题

高级主题

JUnit Platform Launcher API

- JUnit Platform Launcher API
  - 插入自己的测试引擎

JUnit Platform Launcher API

插入自己的测试引擎

API演进

- API演进

API演进

API版本和状态

- API版本和状态

API版本和状态

实验性API

- 实现性的API

实现性的API

@API工具支持

- @API工具支持

@API工具支持